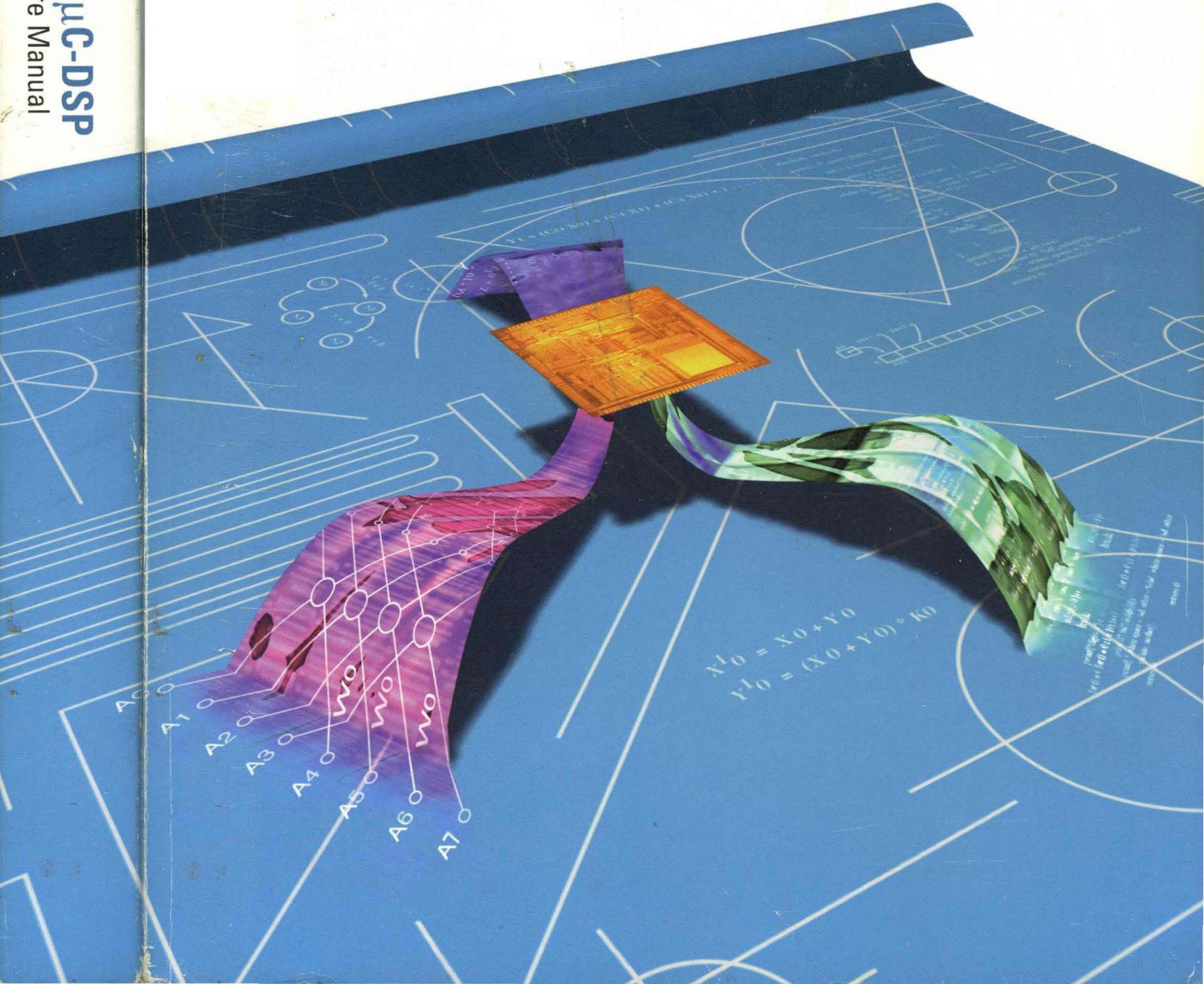
SIEMENS

SIEMENS

TriCore µC-DSP
Architecture Manual

TriCore µC-DSP

Architecture Manual



SIEMENS

TriCore Architecture Manual

Erin Farquhar
Elaine Hadad

Version 1.1
09/17/97

♦ PRELIMINARY EDITION ♦

1. A critical component is a component used in a life-support device whose failure can reasonably be expected to cause the failure of that life-support device or system, and/or to affect the safety or effectiveness of that device or system.
2. Life-support devices or systems are intended: (a) to be implemented in the human body, or (b) to support and/or maintain human life. If they fail, it is reasonable to assume that the health of the user may be endangered.

Front Matter

Front Matter

1.1 Revision History

Release Version	Release Date	Contents of Revision
1.0	06/01/97	Beta Release.
1.1	09/17/97	Preliminary Release.

List of Figures

List of Tables

Preface

Preface

This document contains the following parts:

- Chapter 1, "Architecture Overview," provides a general description of the TriCore architecture and its features.
- Chapter 2, "Programming Model," describes the data formats, data types, addressing modes, and memory model of the TriCore architecture.
- Chapter 3, "Core Registers," describes the core registers, which are categorized according to function.
- Chapter 4, "Managing Tasks and Functions," describes the TriCore's task management operation.
- Chapter 5, "Interrupt System," describes the elements of the TriCore interrupt system including arbitration, the priority level scheme, and interrupt handling.
- Chapter 6, "Traps," lists the eight classes of traps and describes how the TriCore architecture handles traps.
- Chapter 7, "Protection System," describes the components of the TriCore protection system including access permissions and the connection to the debug system.
- Chapter 8, "Instruction Set Overview," describes the instructions by type.
- Chapter 9, "TriCore Instruction Set," describes the individual TriCore instructions.

Where to Look for More Information

Additional information about the TriCore product line can be found in the following publications. Please call your regional sales office to request these publications.

- TriCore Instruction Set Simulator User's Guide
- TriCore Architectural Overview Handbook
- Introducing TriCore (Brochure)
- TriCore Development Tools (Brochure)

Acknowledgments

We would like to acknowledge the extraordinary effort of all of the individuals who participated in the development and production of this manual, particularly the TriCore Architecture group for providing expertise, input, and technical reviews; and, the Marcom group for the coordination of collateral materials and graphic production of this manual.

Architecture Overview

SIEMENS

Architecture Overview

TriCore is the first single-core 32-bit microcontroller-DSP architecture optimized for real-time embedded systems. TriCore unifies the best of three worlds—real-time capabilities of microcontrollers, the computational prowess of DSPs, and the highest performance/price implementations of RISC load-store architectures.

Figure 1 shows a high-level view of the TriCore architecture.

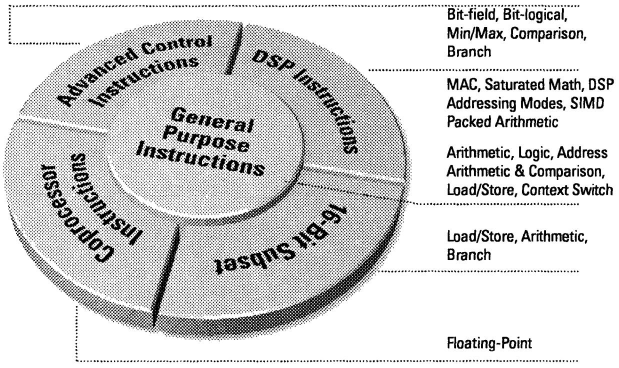

Advanced Control Instructions
DSP Instructions
General Purpose Instructions
Coprocessor Instructions
16-Bit Subset
Bit-field, Bit-logical, Min/Max, Comparison, Branch
MAC, Saturated Math, DSP Addressing Modes, SIMD Packed Arithmetic
Arithmetic, Logic, Address Arithmetic & Comparison, Load/Store, Context Switch
Load/Store, Arithmetic, Branch
Floating-Point

Figure 1: TriCore: A Modular Instruction Set Architecture

The architecture supports a uniform, 32-bit address space, with memory-mapped I/O. It allows for a wide range of implementations, ranging from simple scalar to superscalar. Furthermore, the ISA is capable of interacting with different system architectures, including those with multiprocessing. This flexibility at the implementation and system levels allows for different trade-offs between performance and cost at any point in time.

To support TriCore implementations with 32-bit instructions and simplified instruction fetching, the entire TriCore architecture is represented in 32-bit instruction formats. In addition, the architecture includes 16-bit instruction formats for the most frequently occurring instructions. These instructions significantly reduce code space, lowering memory requirements, system cost, and power consumption.

Real-time responsiveness is largely determined by interrupt latency and context-switch time. The high-performance architecture minimizes interrupt latency by avoiding long multicycle instructions and by providing a flexible hardware-supported interrupt scheme. Furthermore, the architecture supports fast context switching.

1.1 TriCore Architecture Feature Overview

The following list summarizes the basic features of the TriCore architecture.

- 32-bit architecture
- 4-GByte unified data, program, and input/output address space
- 16-/32-bit instructions for reduced code size
- Low interrupt latency
- Fast automatic context switching
- Multiply-accumulate unit
- Saturating integer arithmetic
- Bit handling
- Packed data operations
- Zero-overhead loop
- Flexible power management
- Byte and bit addressing
- Little-endian byte ordering
- Support for big- and little-endian byte ordering at bus interface
- Precise exceptions
- Flexible interrupt prioritization scheme

1.2 Program State Registers

The TriCore program state registers consist of 32 general-purpose registers (GPRs), two 32-bit registers with program status information (PCXI and PSW), and a program counter (PC). PCXI, PSW, and PC are core special function registers (CSFRs).

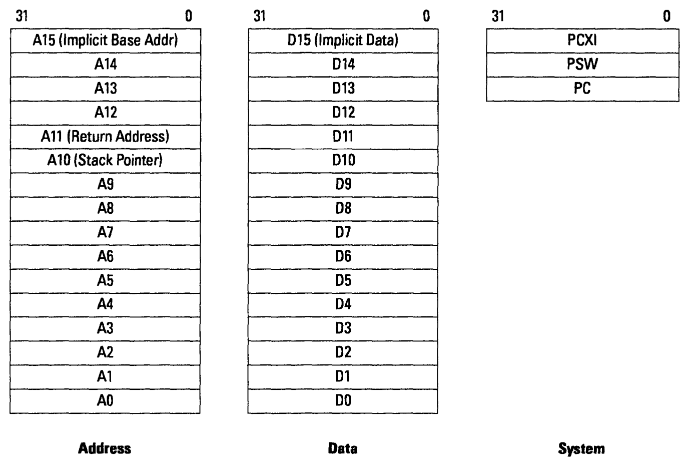


Figure 2: Program State Registers

The 32 general-purpose registers are divided into 16, 32-bit data registers (D0 through D15) and 16, 32-bit address registers (A0 through A15). Four GPRs have special functions: D15 is used as an implicit data register, A10 is the stack pointer (SP), A11 is the return address register, and A15 is the implicit base address register.

Registers A0 and A1 in the lower address registers and A8 and A9 in the upper address registers are defined as **SYSTEM GLOBAL REGISTERS**. These registers are not included in either context partition, and are not saved and restored across calls or interrupts. The operating system normally uses them to reduce system overhead.

The PCXI and PSW registers contain status flags, previous execution information, and protection information.

Refer to Chapter 3, "Core Registers," for complete information on each register.

1.3 Data Types

The TriCore instruction set supports operations on booleans, bit strings, characters, signed fractions, addresses, signed and unsigned integers, and single-precision floating-point numbers. Most instructions work on a specific data type, while others are useful for manipulating several data types.

Refer to Section 2.1, "Data Types," and Section 2.2, "Data Formats," for more specifics on the data types and formats, respectively.

1.4 Addressing Modes

Addressing modes allow load and store instructions to efficiently access simple data elements within data structures like records, randomly and sequentially accessed arrays, stacks, and circular buffers. Simple data elements are 1, 8, 16, 32, or 64 bits wide.

The addressing modes provide efficient compilation of C, easy access to peripheral registers, and efficient implementation of typical DSP data structures (circular buffers for filters and bit-reversed indexing for FFTs). The following seven addressing modes are supported in the Trillium architecture.

- Absolute
- Base + Short Offset
- Base + Long Offset
- Pre-increment or decrement
- Post-increment or decrement
- Circular
- Bit Reverse

Refer to Section 2.4, "Addressing Model," for more details on each addressing mode.

1.5 Instruction Formats

The TriCore architecture supports both 16- and 32-bit instruction formats. All instructions have a 32-bit format; the 16-bit instructions are a subset of the 32-bit instructions, chosen because of their frequency of use and are included to reduce code space.

Refer to Chapter 8, "Instruction Set Overview," and Chapter 9, "TriCore Instruction Set," for more detailed information on the 16-bit and 32-bit instruction formats.

1.6 Tasks and Contexts

Throughout this book, the term **TASK** refers to an independent thread of control. There are two types of tasks: **SOFTWARE-MANAGED TASKS (SMTs)** and **INTERRUPT SERVICE ROUTINES (ISRs)**. Software-managed tasks are created through the services of a real-time kernel or OS, and dispatched under the control of scheduling software. ISRs are dispatched by hardware in response to an interrupt. In this architecture, ISR refers only to the code that is invoked by the hardware directly. Software-managed tasks are sometimes referred to as **USER TASKS**, assuming that they will execute in user mode.

Each task is allocated its own permission level. The individual permissions are enabled/disabled primarily by IO mode bits in the Program Status Word (PSW).

Associated with any task is a set of state elements known collectively as the task's **CONTEXT**. The context is everything the processor needs in order to define the state of the associated task and enable its continued execution. It includes the CPU general-purpose registers that the task uses, the task's program counter (PC), and its Program Status Information (PCXI and PSW). The TriCore architecture efficiently manages and maintains the tasks' contexts through hardware.

Chapter 4, "Managing Tasks and Functions," provides more details on task management. The registers associated with task management are described in Section 3.4, "Context Management Registers."

1.6.1 Upper and Lower Contexts

The context is subdivided into the **UPPER CONTEXT** and the **LOWER CONTEXT**, as illustrated in Figure 3. The upper context consists of the upper address registers, A10 - A15, and the upper data registers, D8 - D15. These registers are designated as non-volatile, for purposes of function calling. The upper context also includes the PCXI and PSW registers.

The lower context consists of the lower address registers, A2 through A7, the lower data registers, D0 through D7, and the PC.

Both upper and lower contexts include a **LINK WORD**. Contexts are saved in fixed- size areas (see next section); they are linked together via the link word.

The upper context is saved automatically on interrupts and is restored on returns. The lower context is saved and restored explicitly by the interrupt service routine (ISR) if the ISR needs to use more registers than provided by the upper context.

Refer to Chapter 4, "Managing Tasks and Functions," for more information.

Lower Context	Upper Context
D7	D15
D6	D14
D5	D13
D4	D12
D3	D11
D2	D10
D1	D9
D0	D8
A7	A15
A6	A14
A5	A13
A4	A12
A3	A11 (RA)
A2	A10 (SP)
Saved PC	PSW
PCXI (Link Word)	PCXI (Link Word)

Figure 3: Upper and Lower Contexts

1.6.2 Context Save Areas

The Trillium architecture uses linked lists of fixed-size **CONTEXT SAVE AREAS** (CSAs), which accommodate systems with multiple interacting threads of control. A CSA is 16 words of on-chip memory storage, aligned on a 16-word boundary. A single CSA can hold exactly one upper or one lower context. Unused CSAs are linked together on a free list. They are allocated from the free list as needed, and returned to it when no longer needed. The processor hardware handles the allocation and freeing. They are transparent to the applications code. Only the system start-up code and certain OS exception handling routines need to access the CSA lists and memory storage explicitly.

1.6.3 Fast Context Switching

To increase performance, the TriCore architecture implements a uniform context-switch mechanism for function calls, interrupts, and traps. In all cases, the task's upper context is automatically saved and restored by hardware; saving (and restoring) the lower context is left as an option for the new task.

Fast context switching is further enhanced by the TriCore's unique memory subsystem design, which allows transfers of up to 16 data words between processor registers and memory, thus permitting the entire context to be saved in one operation.

1.7 Interrupt System

In this manual, a **SERVICE REQUEST** is defined as an interrupt request from a peripheral, a DMA request, or an external interrupt. For simplicity, a service request may also be referred to as an interrupt.

The entry code for the ISR is a block within a vector of code blocks. Each code block provides an entry for one interrupt source. Each source is assigned a priority number. All priority numbers are programmable. The service routine uses the priority number to determine the location of the entry code block.

The prioritization of service routines enables nested interrupts. A service request can interrupt the servicing of a lower priority interrupt. Interrupt sources with the same priority cannot interrupt each other.

Refer to Chapter 5, "Interrupt System," for more information on service requests and the interrupt system.

1.8 Trap System

A trap occurs as a result of an exception within one of the following eight classes:

- Reset
- Internal Protection
- Instruction Errors

- Context Management
- Internal Bus and Peripheral Errors
- Assertion
- System Call
- Non-Maskable Interrupt

The entry code for the trap handler is comprised of a vector of code blocks. Each code block provides an entry for one trap. When a trap is taken, the trap's Trap Identification Number (TIN) is placed in data register D15. The trap handler uses the TIN to identify precisely the cause of the trap. The trap with the lowest TIN wins during arbitration.

Refer to Chapter 6, "Traps," for more information.

1.9 Protection System

The protection system allows the programmer to assign access permissions to memory regions for both data and code. This capability is useful for protecting core system functionality from bugs that may have slipped through testing and from transient hardware errors.

The TriCore's protection system also provides the essential features needed to isolate errors, and thus facilitates debugging.

The registers associated with the protection system are defined in Section 3.8, "Memory Protection Registers." Chapter 7, "Protection System," describes the Memory Protection System in more detail. A list of Debug registers is located in Section 3.9, "Debug Registers."

1.9.1 Permission Levels

The TriCore's embedded architecture allows each task to be allocated the specific permission level it needs to perform its function. Individual permissions are enabled through the IO mode bits in the Program Status Word (PSW). The three permission levels are User-0, User-1, and Supervisor:

- **USER-0 MODE** is used for tasks that do not access peripheral devices. Tasks at this level do not have permission to enable or disable interrupts.
- **USER-1 MODE** is used for tasks that access common, unprotected peripherals. Accesses typically include read/write accesses to SIO ports and read accesses to timers and most I/O status registers. Tasks at this level may disable interrupts.
- **SUPERVISOR MODE** permits read/write access to system registers and protected peripheral devices.

1.9.2 Protection Model

The memory protection model for the TriCore architecture is based on address ranges, where each address range has an associated permission setting. Address ranges and their associated permissions are specified in two to four identical sets of tables residing in core SFR (CSFR) space. Each set is referred to as a **PROTECTION REGISTER SET (PRS)**.

When the protection system is enabled, the TriCore checks every load/store or instruction fetch address for legality before performing the access. To be legal, the address must fall within one of the ranges specified in the currently selected PRS, and permission for that type of access must be present in the matching range.

1.10 Reset System

Most of the reset functions and options are located external to the core and are not described in this architecture manual. Several events can force a reset of the TriCore device:

- Power-On Reset: activated through an external pin when the power to the device is turned on (cold reset).
- Hard Reset: activated through an external pin during run time (warm reset).
- Soft Reset: activated through a software write to a reset request register. This register has a special protection mechanism to prevent accidental accesses. Implementation-specific controls in this register facilitate either a partial or a full reset of the device.
- Watchdog Timer Reset: activated through an error condition detected by a watchdog timer.
- Wake-up Reset: activated through an external pin to wake the device from a power saving mode.

A reset status register allows the core to check which one of the different triggers caused the reset.

1.11 Debug System

The TriCore contains mechanisms and resources to support on-chip debugging. These are used by the Debug Control Unit, which is an off-core module. Most functions and details of the Debug Control Unit are implementation specific. Thus, this document does not provide further descriptions of the debug control unit and its associated registers. Please contact your local Siemens sales office for literature information.

Programming Model

2

Programming Model

This chapter discusses the following aspects of the TriCore architecture that are visible to software: the supported data types, the formats of the data types in registers and memory, the various addressing modes that the architecture provides, and the memory model.

2.1 Data Types

The TriCore instruction set supports operations on booleans, bit strings, characters, signed fractions, addresses, signed and unsigned integers, and single-precision floating-point numbers. Most instructions operate on a specific data type, while others are useful for manipulating several data types.

Boolean	A boolean is either TRUE or FALSE. TRUE is the value one (1) when generated and non-zero when tested; FALSE is the value zero (0). Booleans are produced as the result in comparison and logic instructions, and are used as source operands in logical and conditional jump instructions.
Bit String	A bit string is a packed field of bits. Bit strings are produced and used by logical, shift, and bit field instructions.
Character	A character is an eight-bit value that is a very short unsigned integer. No specific coding is assumed.
Signed Fraction	The TriCore architecture supports 16-bit signed fractional data for DSP arithmetic. Data values in this format have a single, high-order sign bit, with a value of 0 or -1, followed by an implied binary point and fraction. Thus their values are in the range [-1,1). When stored in registers, fractional data occupies the register's most-significant 16 bits, with the least-significant 16 bits set to zeros.
Address	An address is a 32-bit unsigned value.

Signed/Unsigned Integers

Signed and unsigned integers are normally 32 bits. Shorter signed or unsigned integers are sign-extended or zero-extended to 32 bits when loaded from memory into a register. Multi-precision integers are supported with addition and subtract using carry. Integers are considered to be bit strings for shifting and masking operations. Multi-precision shifts can be done using a combination of single-precision shifts and bit field extracts.

IEEE-754 single-precision floating-point number

Depending on the particular implementation of the core architecture, IEEE-754 floating-point numbers are supported by direct hardware instructions or by software emulation.

2.2 Data Formats

All the general-purpose registers are 32 bits wide, and most instructions operate on word (32-bit) values. Thus when data with fewer bits than a word is loaded from memory, it must be sign or zero-extended before operations can be applied to the full word.

Alignment requirements differ for addresses and data. Addresses (32 bits) must be aligned on a word boundary to permit transfers between address registers and memory. For transfers between data registers and memory, data may be aligned on any halfword boundary, regardless of size; bytes may be accessed on any valid byte address.

Figure 4 on page 15 illustrates the supported data formats.

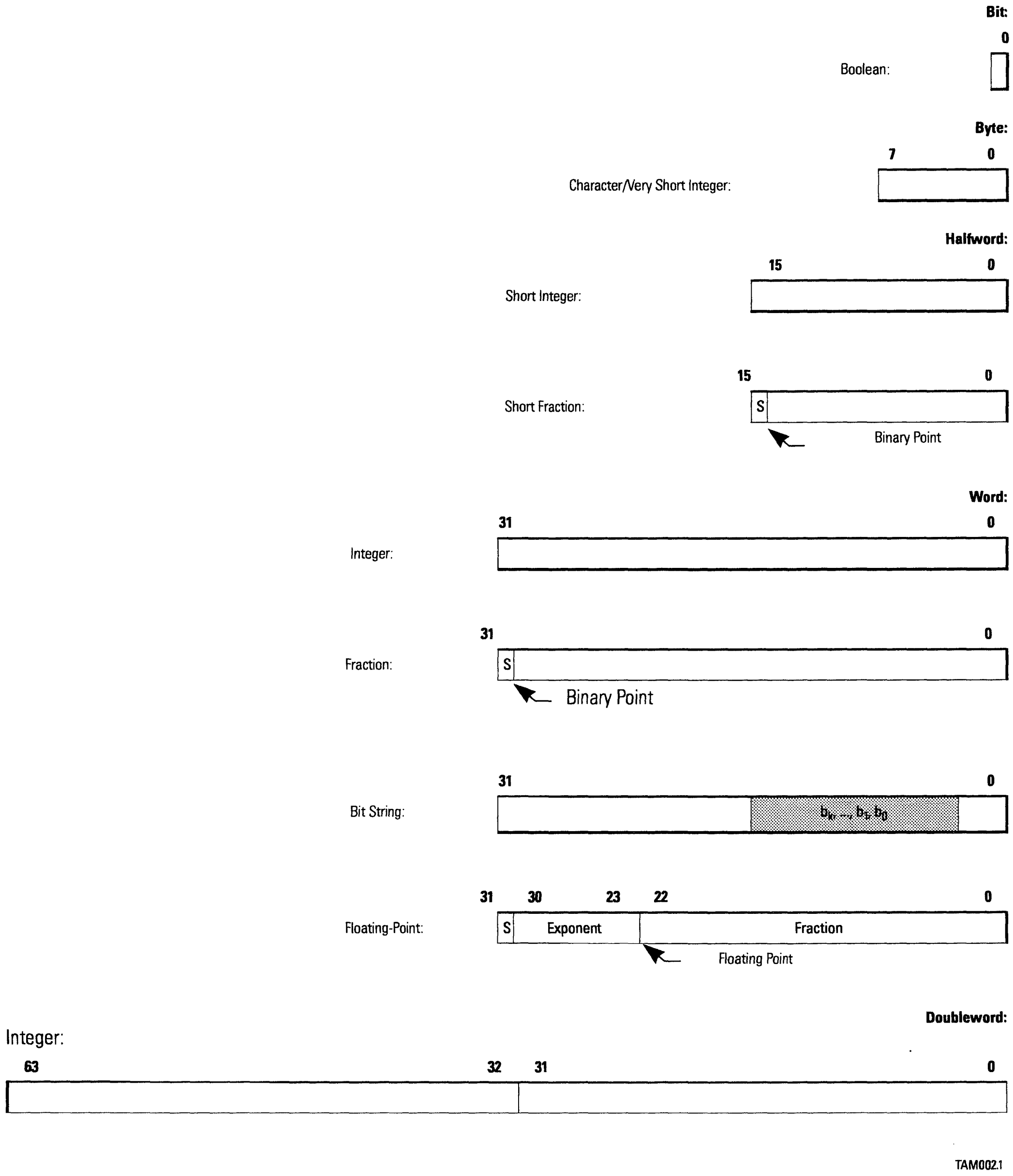


Figure 4: TriCore Data Formats

The data memory and CPU registers store data in little-endian byte order (the least-significant bytes are at lower addresses). Figure 5 illustrates the byte ordering. Little-endian memory referencing is used consistently for data and instructions.

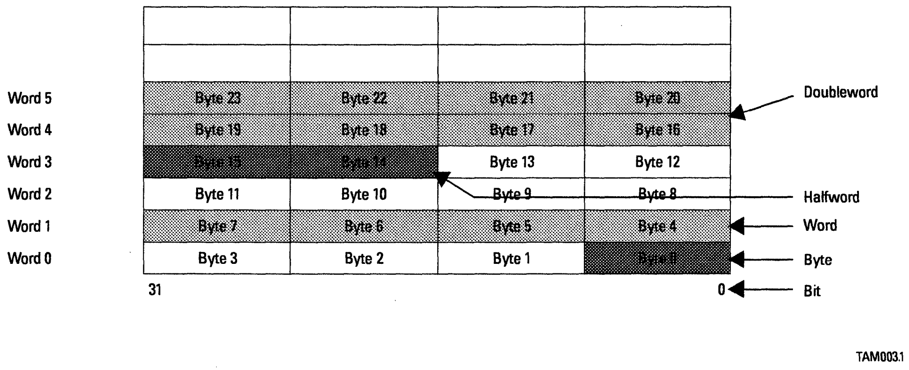


Figure 5: Byte Ordering

When the TriCore system is connected to an external big-endian device, translation between big- and little-endian format is performed by the bus interface.

As stated previously, bytes must be stored on byte boundaries; halfwords, words, and doublewords must be stored on halfword boundaries.

2.3 Memory Model

The TriCore architecture can access up to 4 Gbytes of memory. The address width is 32 bits. The address space is divided into 16 regions or segments (0 through 15). Each segment is 256 Mbytes. The upper four bits of an address select the specific segment. The first 16-Kbytes of each segment can be accessed using either absolute addressing or absolute bit addressing.

Segment 0 is the local static data memory space for the core. Segment 1 is the local dynamic data memory space for the core. Segment 2 is the local code memory space for the core. The upper 16-Kbytes of the local code space in Segment 2 are reserved for the core special function registers (CSFRs).

Segments 14 and 15 are excluded from speculative read accesses. Accesses to this space are initiated only when the core knows that the access will be completed successfully. Segment 14 can be used for external peripherals. FIFOs, peripherals with status registers, and other devices should be located in this address segment so that they will receive no speculative reads that could destroy information. Segment 15 is reserved for the peripheral SFRs (PSFRs) of the internal, on-chip peripherals.

Addresses in Segments 3 through 15 are routed to the System bus. Addresses within Segments 3 through 14 may be either on-chip or off-chip. Devices in the Segment 14 are usually off-chip.

Many data accesses use addresses computed by adding a displacement to the value of a base address register. Using a displacement to cross one of the segment boundaries is not allowed, and, if done, will cause a trap. This restriction allows direct determination of the accessed segment.

Figure 6 shows the TriCore architecture's address space mapping. The figure also shows how the Load/Store Unit, the Instruction Fetch Unit, and other devices on the System bus view the address space.

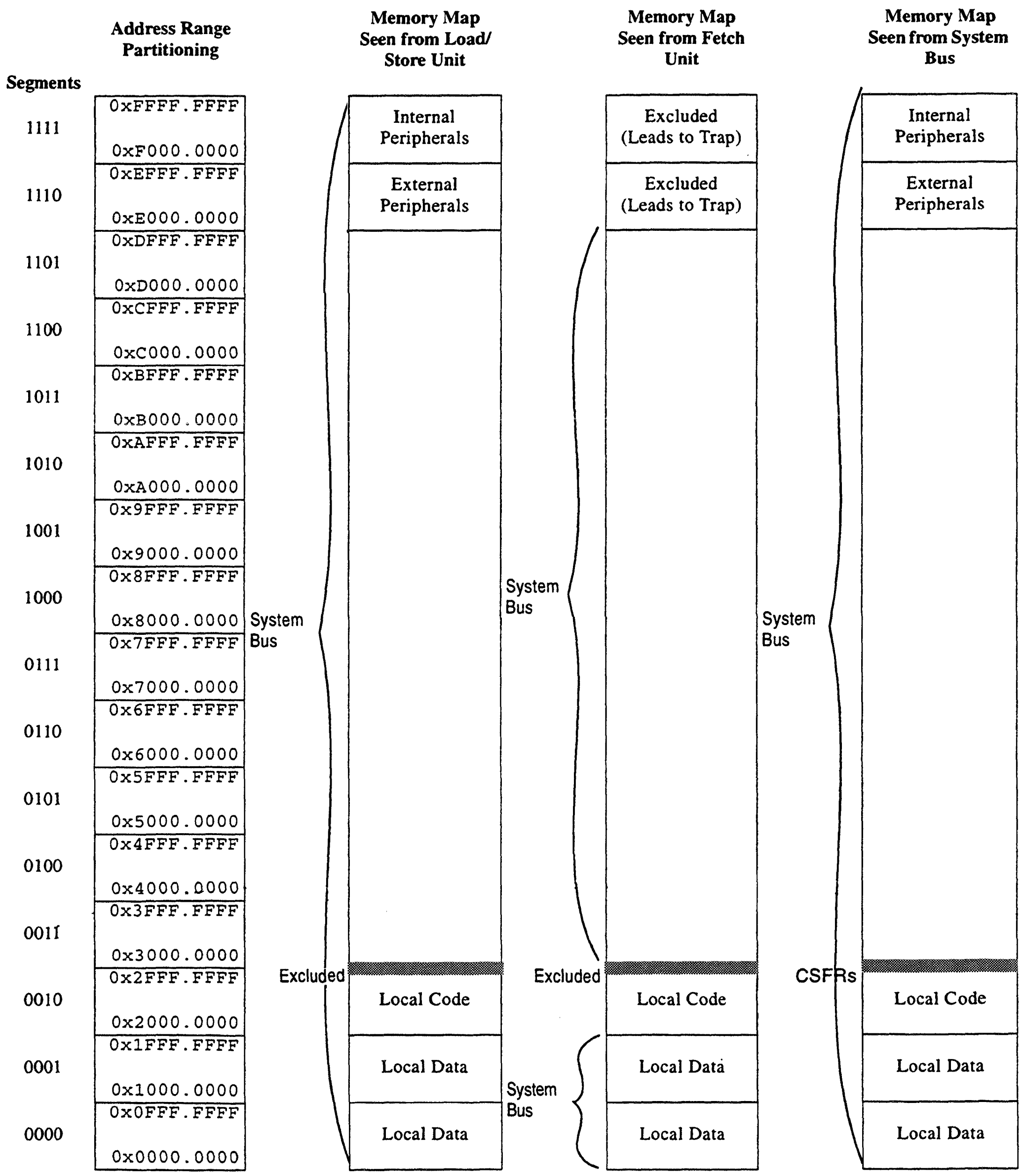


Figure 6. Address Map and Memory Model

The Load/Store Unit regards Segments 0 and 1 as the local data memory and the local code Segment 2 as a "data memory" on the System Bus. This means a data access to the local code memory (for example, access to data constants in code memory) by the Load/Store Unit is routed to the code memory via the System Bus. The Load/Store Unit views Segments 2 through 15 to be on the System Bus. No System Bus access is initiated when the unit accesses its local data space in Segments 0 or 1. Accesses to the core SFR space (CSFR) are not allowed and will cause a trap.

The Instruction Fetch Unit regards Segment 2 as the local code memory and the data Segments 0 and 1 as a "code memory" on the System Bus. This means a code access to the local data memory (for example, execute code out of data memory) by the Instruction Fetch Unit is routed to the data memory via the System Bus. Instruction fetches from Segments 14 and 15 are not allowed and will cause a trap. Instruction fetches from the core SFR space (CSFR) are not allowed and will cause a trap.

The System bus views the entire address space. Devices on the System bus can access all resources, including the local code and data memories and the core SFRs.

2.4 Addressing Model

The first subsection in this section describes the addressing modes that the TriCore architecture supports. The second subsection describes how extended addressing modes can be synthesized through short instruction sequences.

2.4.1 TriCore Addressing Modes

Addressing modes allow load and store instructions to efficiently access simple data elements within data structures such as records, randomly and sequentially accessed arrays, stacks, and circular buffers. Simple data elements are 1, 8, 16, 32, or 64 bits wide.

The TriCore architecture supports seven addressing modes, as listed in Table 1. These addressing modes support efficient compilation of C, easy access to peripheral registers, and efficient implementation of typical DSP data structures (circular buffers for filters and bit-reversed indexing for FFTs). Each addressing mode is described in detail in the following subsections.

Table 1: Addressing Modes of the TriCore Architecture

Addressing Mode	Address Register Use	Offset Size (bits)
Absolute	None	18
Base + Short Offset	Address Register	10
Base + Long Offset	Address Register	16
Pre-increment	Address Register	10
Post-increment	Address Register	10
Circular	Address Register Pair	10
Bit-reverse	Address Register Pair	—

2.4.1.1 Absolute Addressing

Absolute addressing is useful for referencing I/O peripheral registers and global data. The instruction specifies an 18-bit constant as the memory address. As shown in Figure 7, the full 32-bit address results from moving the four most-significant bits of the 18-bit constant to the four most-significant bits of the 32-bit address. The other bits are zero filled.

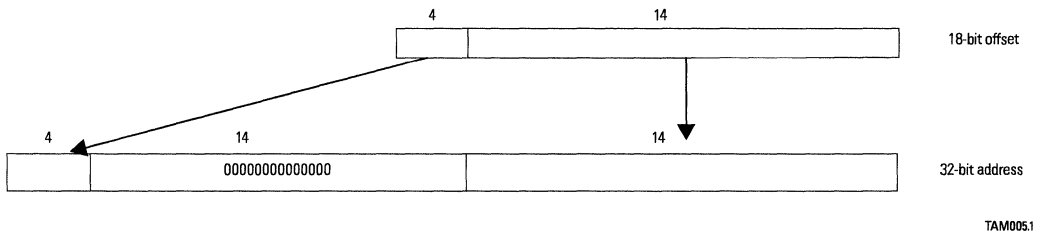


Figure 7: Translation of Absolute Address to Full Effective Address

The special treatment of the four high-order address bits allows absolute addressing to be used in the first 16 KBytes of each address segment.

2.4.1.2 Base+Offset Addressing

Base+offset addressing is used for referencing record elements, local variables (using the stack pointer SP as the base), and static data (using an address register pointing to the static data area).

The effective address is the sum of an address register and the sign-extended offset. The size of the offset depends on the specific instruction. A few of the most common load/store instructions that would be generated by a compiler are allocated 16-bit offsets. Less common instructions are allocated 10 bit offsets.

2.4.1.3 Pre-Increment Addressing

Pre-incrementing and pre-decrementing are used to push data onto an upward or downward growing stack, respectively. The pre-increment addressing mode uses the sum of the address register and the sign-extended 10-bit offset both as the effective address and as the value written back into the address register.

2.4.1.4 Post-Increment Addressing

Post-incrementing and post-decrementing allow forward and backward sequential access of arrays, respectively. Post-decrementing uses a negative offset. This mode also can be used to pop down (post-increment) or up (post-decrement) a growing stack.

The post-increment addressing mode uses the value of the address register as the effective address, and then updates this register by adding the sign-extended 10-bit offset to its previous value.

2.4.1.5 Circular Addressing

Circular addressing is used primarily for accessing data values in circular buffers while performing filter calculations.

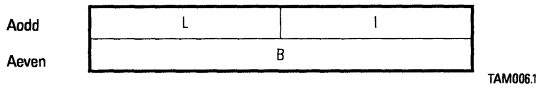


Figure 8: Circular Addressing Mode

The circular addressing mode uses an address register pair to hold the state it requires. The even register is always a base address (B). The most-significant half of the odd register is the buffer size (L). The least significant half holds the index into the buffer (I). The effective address is (B+I). The buffer occupies memory from addresses B to B + L – 1.

The index is post-incremented using the following algorithm:

```
tmp = I + sign_ext(offset10);
if (tmp < 0)
  I = tmp + L;
else if (tmp >= L)
  I = tmp - L;
else
  I = tmp;
```

The 10-bit offset is specified in the instruction word and is a byte-offset that can be either positive or negative. Note that correct "wraparound" behavior is guaranteed as long as the magnitude of the offset is smaller than size of the buffer.

For example, consider a circular buffer consisting of 25, 16-bit values (50 bytes). If the current index is 48, then the next item is obtained using an offset of 2 (two bytes per value). The new value of the index wraps around to 0. If instead the index is 48 and the offset is 4 (two entries per step), the new value of the index would be 2 ((48 + 4) – 50). If the current index is 4 and the offset is -8, then the new index would be 46 ((4 – 8) + 50).

Note that in the end case where a memory access runs off the end of the circular buffer, the data access also wraps around to the start of the buffer. For example, consider a circular buffer containing n elements, where each element is a 16-bit value. If a load word is performed using the circular addressing mode and the effective address of the operation points to element n-1, the 32-bit result will contain element n-1 in the bottom 16 bits and element 0 in the top 16 bits.

The size and length of a circular buffer have the following restrictions placed on them:

1. The start of the buffer start must be aligned to a multiple of the data size, where the data size is determined from the instruction being used to access the buffer. For example, a buffer accessed using a load word instruction must be aligned to a word boundary and a buffer being accessed using a load doubleword must be aligned to a doubleword boundary.
2. The length of the buffer must be a multiple of the data size, where the data size is determined from the instruction being used to access the buffer. For example, a buffer accessed using a load word instruction must be a multiple of four in length and a buffer accessed using a load doubleword instruction must be a multiple of eight in length.

If the two restrictions are not met, then an alignment trap is taken.

2.4.1.6 Bit-Reverse Addressing

Figure 9 shows bit-reverse addressing, which is used to access arrays used in FFT algorithms. The most common implementation of the FFT ends with results stored in bit-reversed order.

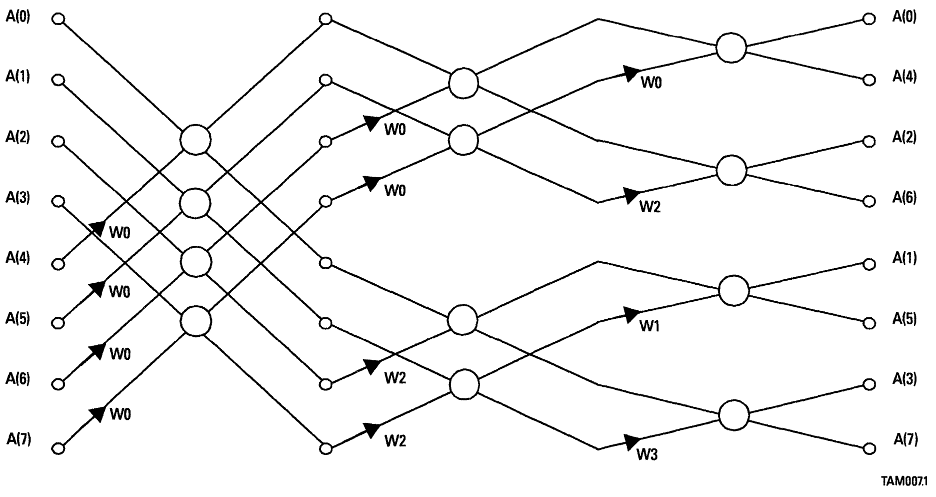


Figure 9: Bit-Reverse Addressing

Bit-reverse addressing uses an address register pair to hold the required state (see Figure 10).

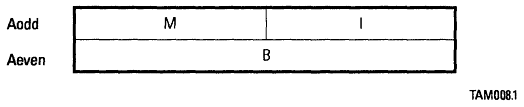


Figure 10: Register Pair for Bit-Reverse Addressing

The even register is the base address of the array (B). The least-significant half of the odd register is the index into the array (I); the most-significant half is the modifier (M), which is added to I after every access.

The effective address is B+I. The index I is post-incremented; its new value is reverse (reverse (I) + reverse (M)), where M is the most-significant half of the odd register. The reverse() function exchanges bit n with bit (15 – n) for n = 0, ..., 7.

2.4.2 Synthesized Addressing Modes

This section describes how addressing not supported directly in the hardware addressing modes can be synthesized through short instruction sequences.

2.4.2.1 Indexed Addressing

Indexed addressing can be synthesized using the ADDSC.A instruction, which adds a scaled data register to an address register. The scale factor can be one, two, four, or eight for addressing indexed arrays of bytes, halfwords, words, or doublewords.

For support of addressing of indexed bit arrays, the ADDSC.AT instruction scales the index value by one eighth (shifts right three bits) and adds it to the address register. The two low-order bits of the resulting byte address are cleared to give the address of the word containing the indexed bit. To extract the bit, the word containing it is loaded, and the bit index is used in an EXTRACT instruction. A bit field, beginning at the indexed bit position, can be extracted also. To store a bit or bit field at an indexed bit position, ADDSC.AT is used in conjunction with the LDMST (Load/Modify/Store) instruction.

2.4.2.2 PC-Relative Addressing

PC-relative addressing is the normal mode for branches and calls. However, the TriCore architecture does not support direct PC-relative addressing of data. The main reason is that the separate on-chip instruction and data memories make data access to the program memory expensive. It typically adds two cycles of added access time.

When PC-relative addressing of data is required, the address of a nearby code label is placed into an address register and used as a base register in base + 16-bit offset mode to access the data. Once the base register is loaded, it can be used to address other PC-relative data items nearby.

A code address can be loaded into an address register in various ways. If the code is statically linked—as it almost always is for embedded systems—then the absolute address of the code label is known, and can be loaded using the LEA instruction (load effective address), or with a sequence to load an extended absolute address (see next subsection below). The absolute address of the PC relative data is also known, and there is no need to synthesize PC-relative addressing.

For code that is dynamically loaded, or assembled into a binary image from position-independent pieces without the benefit of a relocating linker, the appropriate way to load a code address for use in PC-relative data addressing is to use the JL (jump and link) instruction. A jump and link to the next instruction is executed, placing the address of that instruction into the return address register (A11). Before doing so, it is necessary to copy the actual return address of the current function to another register.

2.4.2.3 Extended Absolute Addressing

Extended absolute addressing is synthesized using two instructions: the MOVH.A (Move Highword) instruction and the LEA (load effective address). The LEA instruction loads a 32-bit address into an address register. After execution of the MOVH.A instruction, a base + 16-bit offset is used to address data in order to establish a base register.

Core Registers

SIEMENS

3 Core Registers

The TriCore architecture defines a set of Core Special Function Registers (CSFRs). These CSFRs control the operation of the core and provide status information about the core's operation. The CSFRs are split into the following groups:

- Program State Information
- Stack Management
- Context Management
- Interrupt and Trap Control
- System Control
- Memory Protection
- Debug Control

The following sections describe these registers in detail. The CSFRs are complemented by a set of general purpose registers (GPRs). Table 2 shows all CSFRs and GPRs.

Note that most of the memory protection system and debug control unit is implementation specific, therefore, this architecture manual only summarizes these topics. Note also that the reset functions and options are located in a block outside of the core; their functionality is briefly described in this manual. Please contact your local Siemens Sales office for more information on literature availability.

Table 2: Core Register Map

Register Name	Description	Page #
D0 – D15	Data Registers	29
A0 – A15	Address Registers	29
PSW	Program Status Word	30
PCXI	Previous Context Information	32
PC	Program Counter (read only)	30
FCX	Free Context List Head Pointer	34
LCX	Free Context List Limit Pointer	35
ISP	Interrupt Stack Pointer	35
ICR	Interrupt Control Register	36
BIV	Base Address of Interrupt Vector Table	37
BTV	Base Address of Trap Vector Table	37
SYSCON	System Configuration Register	38
PMUCON	Program Memory Control Register	38
DMUCON	Data Memory Control Register	38
DPRx_0 – DPRx_3	Data Segment Protection Register Sets (x = 0 – 3)	39
CPRx_0 – CPRx_3	Code Segment Protection Register Sets (x = 0 – 3)	39
DPMx_0 – DPMx_3	Data Protection Mode Register Sets (x = 0 – 3)	39
CPMx_0 – CPMx_3	Code Protection Mode Register Sets (x = 0 – 3)	39
DBGSR	Debug Status Register	44
GPRWB	GPR Write Back Trigger	44
EXEVT	External Break Input Event Specifier	44
SWEVT	Software Break Event Specifier	44
CREVT	Core SFR Access Event Specifier	44
TRnEVT	Trigger Event n Specifier (n = 0, 1)	44

3.1 Access to the Core Registers

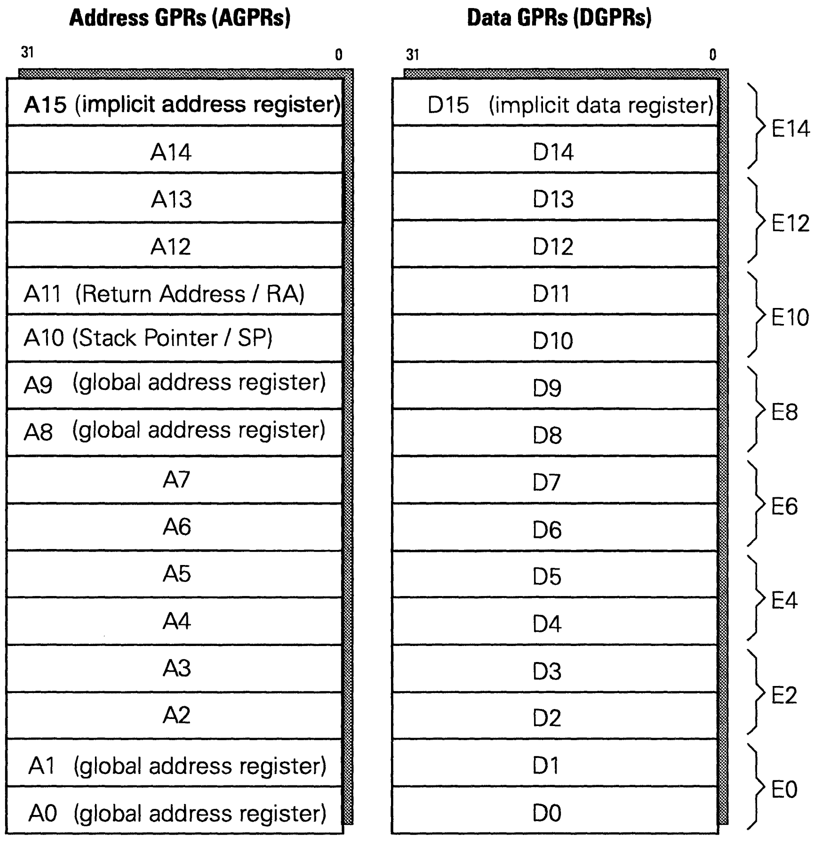
The core accesses the CSFRs through two instructions: MFCR and MTCR. The MFCR instruction (Move From Core Register) moves the contents of the addressed CSFR into a data register. MFCR can be executed on any privilege level. The MTCR instruction (Move To Core Register) moves the contents of a data register to the addressed CSFR. To prevent unauthorized writes to the CSFRs, the MTCR instruction can only be executed on the supervisor privilege level.

The CSFRs are also mapped into the top of the local code segment in the memory address space. This mapping makes the complete architectural state of the core visible in the address map. This feature allows efficient debug and emulator support. Note it is not permitted for the core to access the CSFRs through this mechanism— it must use MFCR and MTCR.

There are no instructions allowing bit, bit field or load-modify store accesses to the CSFRs. The RSTV instruction (Reset Overflow Flags) resets only the overflow flags in the PSW, without modifying any of the other PSW bits. This instruction can be executed at any privilege level.

3.2 General-Purpose Registers (GPRs)

Figure 11 shows the general-purpose registers. The 32-bit wide general-purpose registers are split evenly into 16 data registers, or DGPRs, (D0 to D15) and 16 address registers, or AGPRs, (A0 to A15). Separation of data and address registers facilitates efficient implementations in which arithmetic and memory operations are performed in parallel. Several instructions allow the interchange of information between data and address registers in order to create or derive table indexes, etc. Two consecutive even-odd data registers can be concatenated to form eight extended-size registers (E0, E2, E4, E6, E8, E10, E12, and E14), in order to support 64-bit values.



Address GPRs (AGPRs)
Data GPRs (DGPRs)
31
0
A15 (implicit address register)
A14
A13
A12
A11 (Return Address / RA)
A10 (Stack Pointer / SP)
A9 (global address register)
A8 (global address register)
A7
A6
A5
A4
A3
A2
A1 (global address register)
A0 (global address register)
D15 (implicit data register)
D14
D13
D12
D11
D10
D9
D8
D7
D6
D5
D4
D3
D2
D1
D0
E14
E12
E10
E8
E6
E4
E2
E0

Figure 11: General-Purpose Registers (GPRs)

Registers A0, A1, A8, and A9 are defined as **SYSTEM GLOBAL REGISTERS**. Their contents are not saved and restored across calls, traps, or interrupts. Register A10 is used as the stack pointer (SP); register A11 is used to store the return address (RA) for calls and linked jumps and to store the return program counter (PC) value for interrupts and traps. Refer to Chapter 4, "Managing Tasks and Functions," for more information.

While the 32-bit instructions have unlimited used of the GPRs, many 16-bit instructions implicitly use A15 as their address register and D15 as their data register. This implicit use eases the encoding of these instructions into 16 bits.

In order to support 64-bit data values, an even/odd register pair holds these values. In the assembler syntax, these register pairs are either referred to as a pair of 32-bit registers (for example, D9/D8) or as an extended 64-bit register (for example, E8 is the concatenation of D9 and D8, where D8 is the least significant word of E8).

Note that there are no separate floating-point registers—the data registers are used to perform floating-point operations. The floating-point data is saved/restored automatically using the fast context switch support.

The GPRs are an essential part of a task's context. When saving or restoring a task's context to and from memory, the context is split into the upper and lower contexts. Registers A2 through A7 and D0 through D7 are part of the lower context. Registers A10 through A15 and D8 through D15 are part of the upper context. Refer to Section 1.6.1, "Upper and Lower Contexts," on page 7 and Chapter 4, "Managing Tasks and Functions," for more information.

3.3 Program State Information (PC, PSW, and PCXI)

The PC, PSW, and PCXI registers hold and reflect program state information. When storing and restoring a task's context, the contents of these registers are an important part of this procedure and are stored/restored or modified during this process.

3.3.1 Program Counter

Figure 12 shows the 32-bit program counter (PC). The PC contains the address of the instruction that is currently executing. The PC is part of a task's state information.

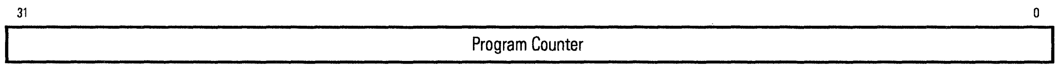


Figure 12: Program Counter (PC)

3.3.2 Program Status Word (PSW)

Figure 13 shows the Program Status Word (PSW). The five most-significant bits of PSW contain ALU status flags that are set and cleared by arithmetic instructions. The remaining bits of PSW control the permission levels, protection register sets, and the call depth counter. The PSW is part of a task's state information.

A single instruction, RSTV, resets all overflow status bits (V, SV, AV, SAV). RSTV can be executed at any privilege level.

31	30	29	28	27	26 … 14	13 12	11 10	9	8	7	6 … 0
C	V	SV	AV	SAV	Res	PRS	IO	IS	GW	CDE	CDC

Figure 13: Program Status Word (PSW)

C — Carry (Bit 31) This flag is set when a carry occurs.

V — Overflow (Bit 30) This flag is set when an overflow occurs.

SV — Sticky Overflow (Bit 29)
This flag is set when an overflow occurs. This flag remains set until it is explicitly reset by an RSTV (Reset Overflow bits) instruction.

AV — Advanced Overflow (Bit 28)
This flag is set when an arithmetic instruction "almost" caused an overflow. This flag is updated after every arithmetic instruction.

SAV — Sticky Advanced Overflow (Bit 27)
This flag is set when an arithmetic instruction "almost" caused an overflow. This flag remains set until it is explicitly reset by an RSTV (Reset Overflow bits) instruction.

PRS — Protection Register Set (Bits 13:12)
This two-bit field selects one of up to four sets of memory protection registers.

00	Protection Register Set 0
01	Protection Register Set 1
10	Protection Register Set 2
11	Protection Register Set 3

IO — I/O Privilege (Bits 11:10) This field selects the I/O privilege mode.

00	User-0
01	User-1
10	Supervisor
11	Reserved

IS — Interrupt Stack (Bit 9)

This bit reflects the status of the current task.

0	Current task uses a user stack
1	Current task uses the global interrupt stack

GW — Global Register Write Permission (Bit 8)

This bit enables write permission to the global registers.

0	Write permission to global registers A0, A1, A8, A9 is disabled
1	Write permission to global registers A0, A1, A8, A9 is enabled

CDE — Call Depth Count Enable (Bit 7)

This bit is the enable for call depth counting.

0	Call depth counting is temporarily disabled. It is automatically re-enabled following execution of the next Call instruction.
1	Call depth counting is enabled. If CDC = 111.1111_2, call depth counting is disabled regardless of the setting on this bit.

CDC — Call Depth Counter (Bits 6:0)

The CDC field consists of two variable-width subfields. The first subfield is a mask field, consisting of a string of zero or more initial "1" bits, terminated by the first "0" bit. The remaining bits comprise the subfield, which constitutes the Call Depth Counter. Refer to Section 7.1.1.6, "CDC," on page 79 for more information on the call depth counter.

Refer to Section 8.1, "Arithmetic Instructions," for more information on the ALU status flags C, V, SV, AV, and SAV. Refer to Chapter 7, "Protection System," for more information on the PRS, IO, GW, CDE, and CDC fields. Refer to Section 4.4, "Context Switching with Interrupts," for more information on the IS bit.

3.3.3 Previous Context Information Register (PCXI)

PCXI contains linkage information to the previous execution context, supporting fast interrupts and automatic context switching. The PCXI is part of a task's state information.

31 24	23	22	21 20	19 16	15 0
PCPN	PIE	UL	Res	PCXS	PCXO

Figure 14: Previous Context Information Register (PCXI)

PCPN — Previous CPU Priority Number (Bits 31:24)

This field contains the priority level number of the interrupted task.

PIE — Previous Interrupt Enable (Bit 23)

This bit indicates the state of the interrupt enable bit (ICR.IE) for the interrupted task.

UL — Upper/Lower Context Tag (Bit 22)

The U/L context tag bit identifies the type of context saved. A one indicates upper context; a zero indicates lower context. If the type does not match the type expected when a context restore operation is performed, a trap is generated.

PCXS — PCX Segment Address (Bits 19:16)

This field contains the segment address portion of the PCX.

PCXO — Previous Context Pointer Offset Field (Bits 15:0)

The PCXO and PCXS fields form the pointer PCX, which points to the CSA of the previous context. Note that the PCX pointer contained in register PCXI is used for context management.

Note that the PCX pointer contained in PCXI is used for context management. Section 3.4, "Context Management Registers," and Chapter 4, "Managing Tasks and Functions," provide more information on the PCX pointer.

3.4 Context Management Registers

This section describes the context management registers, which are comprised of three pointers. These pointers handle context management and are used during context save/restore operations. Refer to Chapter 4, "Managing Tasks and Functions," for more information on the usage of these registers. Table 3 summarizes these registers.

Table 3: Context Management Registers

Register	Category
FCX	Free CSA List Head Pointer
PCX	Previous Context Pointer (contained in register PCXI)
LCX	Free CSA List Limit Pointer

Each pointer consists of two fields: a 16-bit offset and a 4-bit segment specifier. Figure 15 shows how the effective address of a CSA is generated using the two fields. A context save area (CSA) is an address range containing 16 word locations (64 bytes), which is the space required to save one upper or one lower context. Incrementing the pointer offset value by one always increments the effective address to the address that is 16 word locations above the previous one. The total usable range in each address segment for CSAs is 4 MBytes, resulting in storage space for 64 K context save areas.

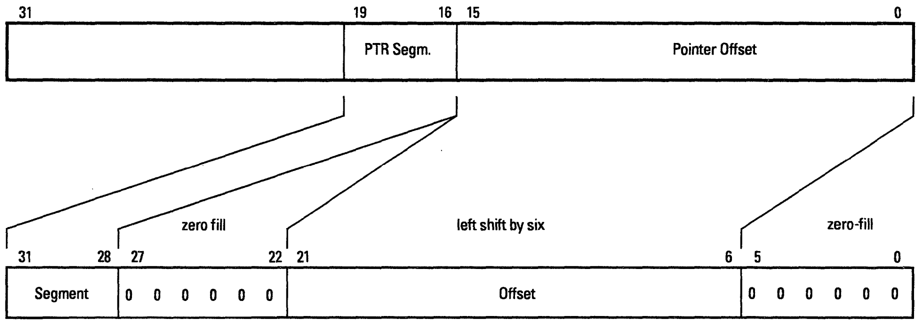


Figure 15: Generation of the Effective Address for the Context Save Areas (CSAs)

Note that the effective address should result in a physical memory address. Address ranges not covered by physical memories could lead to unexpected results. Segments 14 and 15, which are reserved for external and internal peripherals, should also not be used for context save areas.

3.4.1 Free CSA List Head Pointer (FCX)

The FCX pointer register holds the free CSA list head pointer, which always points to an available CSA.

31 – 20	19 – 16	15 – 0
Res	FCXS	FCXO

FCXS — FCX Segment Address Field (Bits 19:16)

This field is used in conjunction with the FCXO field.

FCXO — FCX Offset Address Field (Bits 15:0)

The FCXO and FCXS fields together form the FCX pointer, which points to the next available CSA.

3.4.2 Previous Context Pointer (PCX)

The previous context pointer (PCX) holds the address of the CSA of the previous task. PCX is part of the previous context information register PCXI. Refer to Section 3.3.3, "Previous Context Information Register (PCXI)," for a description of the PCXI register. It is shown below for easy reference. The bits not relevant to the pointer function are shaded.

31 – 24	23	22	21 – 20	19 – 16	15 – 0
PCPN	PIE	U/L	Res	PCXS	PCXO

PCXS — PCX Segment Address Field (Bits 19:16)
This field is used in conjunction with the PCXO field.

PCXO — Previous Context Pointer Offset Field (Bits 15:0)
The PCXO and PCXS fields form the pointer PCX, which points to the CSA of the previous context.

3.4.3 Free CSA List Limit Pointer (LCX)

The LCX pointer register is used to recognize impending CSA list underflows. If the value of FCX used on an interrupt or CALL matches the limit value, the context save operation completes, but the target address is forced to the trap vector address for CSA list depletion.

31 … 20	19 … 16	15 … 0
Res	LCXS	LCXO

LCXS — PCX Segment Address (Bits 19:16)
This field is used in conjunction with the LCXO field.

LCXO — Previous Context Pointer Offset Field (Bits 15:0)
The LCXO and LCXS fields form the pointer LCX, which points to the last available CSA.

3.5 Stack Management

The stack management in the TriCore architecture supports a user stack and an interrupt stack. Address register A10, the Interrupt Stack Pointer (ISP), and a PSW bit are involved in the management of the stack.

31 … 0
A10/SP

Figure 16: A10/SP

31 … 0
ISP

Figure 17: Interrupt Stack Pointer (ISP)

General-purpose address register A10 is used as the stack pointer. The initial contents of this register are usually set by an RTOS when a task is created, which allows a private stack area to be assigned to individual tasks.

The Interrupt Stack pointer (ISP) helps to prevent interrupt service routines (ISRs) from accessing the private stack areas and possibly interfering with the software managed task's context. An automatic switch to the use of the interrupt stack pointer instead of the private stack pointer is implemented in the TriCore architecture. The PSW.IS bit indicates which stack pointer is in effect. When an interrupt is taken and the interrupted task was using its private stack (IS = 0), then after saving its contents with the upper context of the interrupted task (see Chapter 4, "Managing Tasks and Functions," for information on context management), SP/A10 is loaded with the current contents of the interrupt stack pointer ISP.

When an interrupt is taken and the interrupted task was already using the interrupt stack (IS = 1), then no preloading of SP/A10 is performed. The interrupt service routine continues to use the interrupt stack at the point where the interrupted routine had left it.

Usually it is only necessary to initialize ISP once during the initialization routine. However, depending on application needs, ISP can be modified during execution.

Nothing prevents an ISR or system service routine from executing on a private stack. Usage of the SP/A10 in an ISR is at the discretion of the application programmer.

3.6 Interrupt and Trap Control

Three CSFRs support interrupt and trap handling: the Interrupt Control Register (ICR), the interrupt vector table pointer (BIV), and the trap vector table pointer (BTV). Refer to Chapter 5, "Interrupt System," and Chapter 6, "Traps," for more information on interrupts and traps, respectively.

3.6.1 Interrupt Control Register (ICR)

The Interrupt Control Register (ICR) holds the current CPU priority number (CCPN), the enable/disable bit for the interrupt system (IE), the pending interrupt priority number (PIPN) and an implementation specific control for the interrupt arbitration scheme. The other two registers hold the base addresses for the interrupt and trap vector tables. The Interrupt Control Register (ICR) register is shown in Figure 18.

31 … 26	25 … 24	23 … 16	15 … 9	8	7 … 0
Res	ARBCYC	PIPN	Res	IE	CCPN

Figure 18: Interrupt Control Register (ICR)

ARBCYC — Arbitration Cycle Control (Bits 25:24)
The function of this field is implementation-specific.

PIPN — Pending Interrupt Priority Number (Bits 23:16)
This read-only field contains the priority number of the pending interrupt.

IE — Interrupt System Enable (Bit 8)
This bit determines whether the interrupt system is enabled (IE = 1) or not (IE = 0).

CCPN — Current CPU Priority Number (Bits 7:0)
This field contains the current CPU priority number.

Special instructions control the enabling and disabling of the interrupt system. Refer to Section 5.7, "Enabling/Disabling the Interrupt System," on page 64 for more details.

3.6.2 Interrupt Vector Table Pointer (BIV)

The BIV contains the base address of the interrupt vector table. When an interrupt is accepted, the entry address into the interrupt vector table is generated from the priority number (taken from the PIPN) of that interrupt, left shifted by five bits, and then ORed with the contents of the BIV register. The left-shift of the interrupt priority number results in a spacing of eight words (32 bytes) between the individual entries in the vector table.

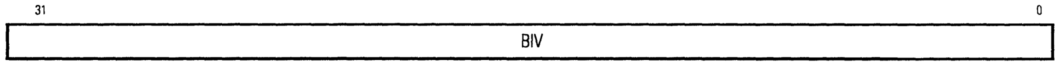


Figure 19: Interrupt Vector Table Pointer (BIV)

Care must be taken regarding the alignment of the address contained in the BIV register. First, the address in the BIV register must be aligned to an even byte address (halfword address). Second, due to the simple ORing of the left-shifted priority number and the contents of the BIV register, the alignment of the base address of the vector table must be to a power of two boundary. It depends on the number of interrupt entries used. For the full range of 256 interrupt entries, an alignment to an 8-KByte boundary is required. If fewer sources are used, the alignment requirements are correspondingly relaxed.

3.6.3 Trap Vector Table Pointer (BTV)

The BTV contains the base address of the trap vector table. When a trap occurs, the entry address into the trap vector table is generated from the trap identification number (TIN) of that trap, left-shifted by five bits and then ORed with the contents of the BTV register. The left-shift of the trap identification number results in a spacing of eight words (32 bytes) between the individual entries in the vector table.

Figure 20: Trap Vector Table Pointer (BTV)

Care must be taken regarding the alignment of the address contained in the BTV register. First, the address in the BTV register must be aligned to an even byte address (halfword address). Second, due to the simple ORing of the left-shifted trap identification number and the contents of the BTV register, the alignment of the base address of the vector table must be to a power of two boundary. There are eight different trap classes, resulting in TINs from 0 to 7. Thus, the contents of BTV should be set at least to a 256-byte boundary (8 TINs * 8 word spacing).

Refer to Section 6.2, "Trap Handling," for more information on the trap vector table.

3.7 System Control Registers

Three registers provide system control: the System Configuration Control Register (SYSCON), the local Program Memory Unit Control Register (PMUCON), and the local Data Memory Unit Control Register (DMUCON).

3.7.1 SYSCON Register

The SYSCON Register is shown in Figure 21.

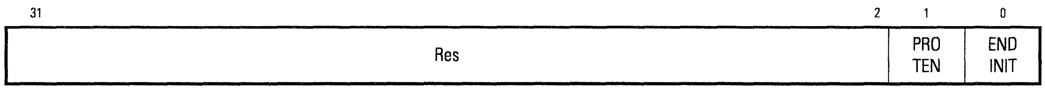


Figure 21: SYSCON Register

PROTEN — Memory Protection Enable (Bit 1) This bit enables the memory protection system. Memory protection is controlled through the memory protection register sets. Note that it is required to initialize the protection register sets prior to setting PROTEN to one.

0	Memory Protection is disabled
1	Memory Protection is enabled.

ENDINIT — End of Initialization (Bit 0) This bit controls access to critical configuration and control registers. Software can set ENDINIT only to one. A one indicates that the basic initialization and configuration of the device is finished. Once set, ENDINIT can be cleared only through a reset. Any registers or control bits protected with ENDINIT are locked against modifications as long as ENDINIT is set. Note that the exact definition of which registers/control bits are protected with ENDINIT is implementation-specific.

3.7.2 PMUCON Register

Figure 22 shows the PMUCON Register. Control for the local program memory is implementation-specific. Please contact your local Siemens Sales Office for additional information.

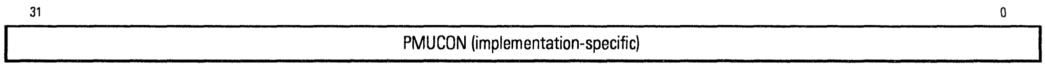


Figure 22: PMUCON Register

3.7.3 DMUCON Register

Figure 23 shows the DMUCON Register. Control for the local data memory is implementation-specific. Please contact your local Siemens Sales Office for additional information.

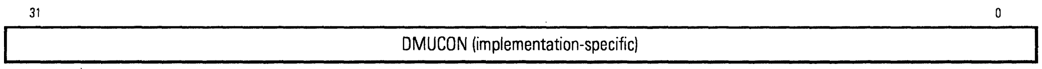


Figure 23: DMUCON Register

3.8 Memory Protection Registers

The TriCore architecture incorporates hardware mechanisms that protect user-specified memory ranges from unauthorized read, write, or instruction fetch accesses. In addition, the protection hardware can be used to generate signals to the debug unit. The TriCore contains register sets that specify the address range and the access permissions for a number of memory ranges. There are separate register sets for code and data memory. Figure 24 shows the Data and Code Memory Protection Register Sets.

Data Memory Protection Register Set 0	PSW.PRS = 00_2	Code Memory Protection Register Set 0
Data Memory Protection Register Set 1	PSW.PRS = 01_2	Code Memory Protection Register Set 1
Data Memory Protection Register Set 2	PSW.PRS = 10_2	Code Memory Protection Register Set 2
Data Memory Protection Register Set 3	PSW.PRS = 11_2	Code Memory Protection Register Set 3

Figure 24: Memory Protection Register Sets

The two-bit PRS field in the PSW selects which register set is active at a given time. As shown in Figure 24, two register sets are selected at one time: one data memory protection and one code memory protection.

The PSW.PRS field allows selection of up to four such register sets (four for data and four for code). The number of register sets provided for memory protection is specific to each implementation of the TriCore architecture. Thus this document only describes the generic format of these register sets. For detailed information on the number of register sets and their organization, please refer to the appropriate product specifications. Contact your local Siemens Sales Office for additional information.

Each register set contains a minimum of four range table entries (see Figure 25). The number of range table entries is specific to each implementation of the TriCore architecture. Each range table entry consists of a Segment Protection register pair and a Mode register. The register pair specifies the lower and the upper boundary addresses of the memory range, while the Mode register contains the access permission and debug control bits. The control options are different for the data and the code memory protection.

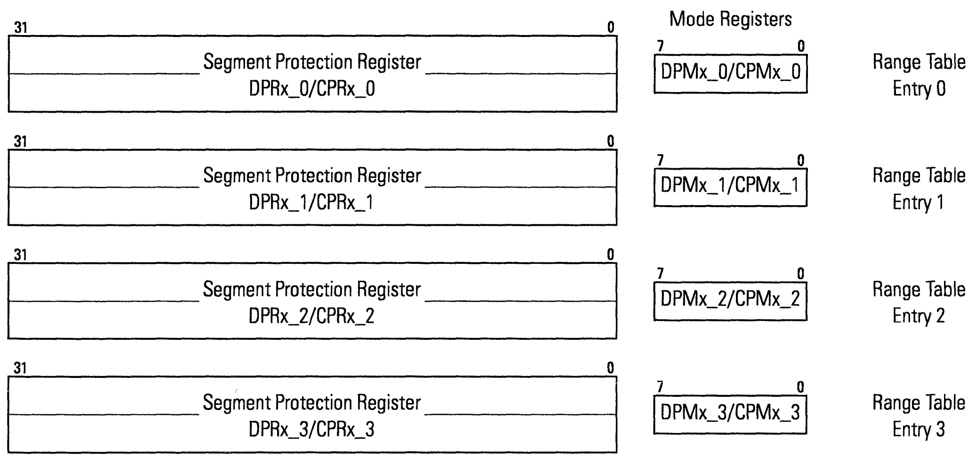


Figure 25: Range Table Entries in a Protection Register Set

Table 4 lists the Memory Protection Registers. Index x indicates the protection register set number, while index n indicates the range table entry number.

Table 4: Memory Protection Registers

Register	Description
DPRx_n	Data Segment Protection Registers (x, n = 0, 1, 2, 3)
DPMx_n	Data Protection Mode Registers (x, n = 0, 1, 2, 3)
CPRx_n	Code Segment Protection Registers (x, n = 0, 1, 2, 3)
CPMx_n	Code Protection Mode Registers (x, n = 0, 1, 2, 3)

3.8.1 Data and Code Segment Protection Registers

Figure 26 and Figure 27 show the segment protection registers of a range table entry. The register pair DPRx_n/CPRx_n contains the two word registers specifying the lower and the upper boundary address of the associated memory range. The range defined by a range table entry is:

lower bound $\leq$ address < upper bound

Range checking is not performed if the lower bound is greater than the upper bound. If the lower bound is equal to the upper bound, the range is regarded as empty.

For the generation of debug signals, instead of defining a range, the values in DPRx_n/CPRx_n are regarded as individual addresses. Signals to the debug unit are generated if the address of a memory access equals one or more of the DPRx_n/CPRx_n contents (note that for this purpose, an equality compare with the contents of the upper bound register is performed).

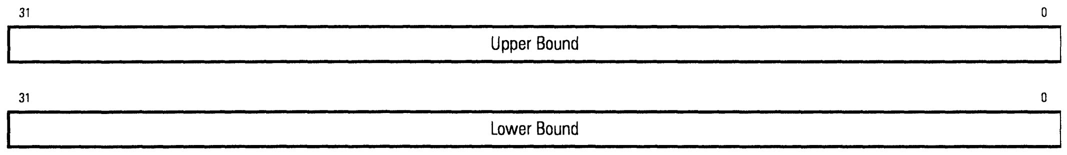


Figure 26: Data Segment Protection Registers (DPRx_n)

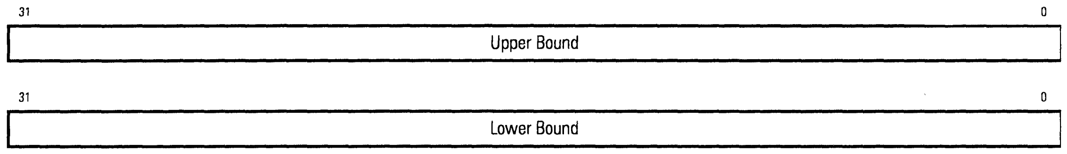


Figure 27: Code Segment Protection Registers (CPRx_n)

3.8.2 Data Protection Mode Registers

The eight-bit Data Protection Mode Registers determine the access permissions and debug signal conditions for the ranges specified in their corresponding Data Segment Protection Registers. Figure 28 shows the assignment and definition of bits within a mode table entry for the data range. The WE and RE bits relate directly to memory protection. The remaining bits generate signals to the Debug Control Unit.

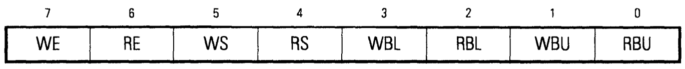


Figure 28: Data Protection Mode Register (DPMx_n)

WE — Address Range Data Write Enable (Bit 7)

This bit controls writes to the addresses in the associated range.

0	Data write accesses to the associated address range are not permitted
1	Data write accesses to the associated address range are permitted

RE — Address Range Data Read Enable (Bit 6)

This bit permits reads to the addresses in the associated range.

0	Data read accesses to the associated address range are not permitted
1	Data read accesses to the associated address range are permitted

WS — Address Range Data Write Signal (Bit 5)

0	Data write signal is disabled
1	A signal is asserted to the debug unit on data read accesses to the associated address range.

RS — Address Range Data Read Signal (Bit 4)

0	Data read signal is disabled
1	A signal is asserted to the debug unit on data read accesses to the associated address range

WBL — Data Write Signal on Lower Bound Access (Bit 3)

0	Data write signal is disabled
1	A signal is asserted to the debug unit on a data write access to an address that matches the lower bound address of the associated address range

RBL — Data Read Signal on Lower Bound Access (Bit 2)

0	Data read signal is disabled
1	A signal is asserted to the debug unit on a data read access to an address that matches the low-er bound address of the associated address range

WBU — Write Signal on Upper Bound Access (Bit 1)

0	Write signal is disabled
1	A signal is asserted to the debug unit on a write access to an address that matches the upper bound address of the associated address range

RBU — Data Read Signal on Upper Bound Access (Bit 0)

0	Data read signal is disabled
1	A signal is asserted to the debug unit on a data read access to an address that matches the up-per bound address of the associated address range

3.8.3 Code Protection Mode Registers

The eight-bit Code Protection Mode Registers determine the access permissions and debug signal conditions for their corresponding range as specified in the associated Code Segment Protection Registers. Figure 29 shows the assignment and definition of bits within a mode table entry for the code range. The XE bit is related directly to memory protection. All remaining bits generate signals to the Debug Control Unit.

7	6	5	4	3	2	1	0
XE	Res	XS	Res	BL	Res	Res	BU

Figure 29: Code Protection Mode Register (CPMx_n)

XE — Address Range Execute Enable (Bit 7)

0	Instruction fetch accesses to the associated address range are not permitted
1	Instruction fetch accesses to the associated address range are permitted

XS — Address Range Execute Signal (Bit 5)

0	Execute signal is disabled
1	A signal is asserted to the debug unit on instruction fetch accesses to the associated address range

BL — Execute Signal on Lower Bound Access (Bit 3)

0	Lower bound execute signal is disabled
1	A signal is asserted to the debug unit on an instruction fetch access to an address that matches the lower bound address of the associated address range

BU — Execute Signal on Upper Bound Access (Bit 0)

0	Upper bound execute signal is disabled
1	A signal is asserted to the debug unit on an instruction fetch access to an address that matches the upper bound address of the associated address range

Refer to Chapter 7, "Protection System," for a description of the Memory Protection Registers within the Protection System.

3.9 Debug Registers

Seven registers are implemented in the core to support debugging. These registers define the conditions under which a debug event is generated, the actions taken on the assertion of a debug event, and provide status information on the debug control unit. Table 5 summarizes the debug registers.

Table 5: Debug Registers

Register	Description
DSR	Debug Status Register
GPRWB	GPR Write Back Trigger Register
EXEVT	External Break Input Event Specifier
SWEVT	Debug Instruction Break Event Specifier
CREVT	Core SFR Access Break Event Specifier
TRnEVT	Trigger Event n Specifier

The functions and details of the Debug Control Unit are implementation specific. Thus this document does not provide further descriptions of the Debug Control Unit and its associated registers. Contact your local Siemens Sales Office for the appropriate literature.

Managing Tasks and Functions

4

Managing Tasks and Functions

Most embedded and real-time control systems are designed according to a model in which interrupt handlers and software-managed tasks are each considered to be executing on their own "virtual" microcontroller. That model is generally supported by the services of a real time executive or operating system (RTOS), layered on top of the features and capabilities of the underlying machine architecture.

In the Trillium architecture, however, the RTOS layer can be very "thin." The hardware can efficiently handle much of the switching between one task and another. At the same time, the architecture allows for considerable flexibility in the tasking model used. System designers can choose the real-time executive and software design approach that best suits the needs of their application, with relatively few constraints imposed by the architecture.

The mechanisms for low overhead task switching and for function calling within the TriCore architecture are closely related. They are discussed together in this chapter.

4.1 Upper and Lower Contexts

As stated in Section 1.6, "Tasks and Contexts," on page 6, a task is an independent thread of control. The task's context defines the state of the task. Should the task be interrupted, the processor uses the context to re-enable the continued execution of the task.

The context is subdivided into the **UPPER CONTEXT** and the **LOWER CONTEXT**, as illustrated in Figure 3 on page 7. The upper context consists of the upper address registers, A10 - A15, and the upper data registers, D8 - D15. These registers are designated as non-volatile, for purposes of function calling. The upper context also includes PCXI and PSW.

The lower context consists of the lower address registers, A2 through A7, and the lower data registers, D0 through D7, plus the Program Counter (PC).

Both upper and lower contexts, when saved to memory, occupy 16-word blocks of storage referred to as Context Save Areas (CSAs). CSAs were introduced in Section 1.6.2, "Context Save Areas," on page 8. The first word in a CSA is the **LINK WORD**; the link word includes two fields that link the given

CSA to the next one in a chain. The fields are a four-bit **LINK SEGMENT** and a 16-bit **LINK INDEX**. The link segment and link index are used to generate the effective address of the linked CSA, as shown in Figure 30.

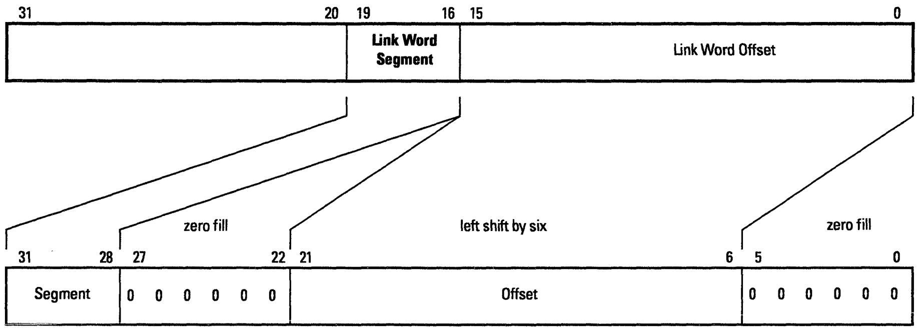


Figure 30: Generation of the Effective Address of a Context Save Area (CSA)

If the CSA is in use (for example, it holds an upper or lower context image for a suspended task), then the link word also contains other information about the linked context. The entire link word, in fact, is simply a copy of the PCXI register for the associated task. Refer to Section 4.3, "CSAs and Context Lists," for further information on how linked CSAs support context switching.

4.2 Task Switching Operation

The TriCore architecture switches tasks when one of the events or instructions listed in Table 6 occurs. Upon occurrence of one of these events or instructions, the upper or lower context of the task is saved or restored. Note that the upper context is saved automatically as a result of an external interrupt, trap, or function call. The lower context is saved explicitly through instructions. In the table, *Save* is a store through the FCX after the next value for the FCX is read from the link word. *Store* is a store through the effective address of the instruction with no change to the CSA list or the FCX register. *Restore* is the converse of Save. *Load* is the converse of Store.

Table 6: Context-Related Events and Instructions

Event/ Instruction	Description	Context Operation	Complement Instruction	Description	Context Operation
Interrupt	Interrupt	Save Upper	RFE	Return From Exception	Restore Upper
Trap	Trap	Save Upper	RFE	Return From Exception	Restore Upper
CALL	Function Call	Save Upper	RET	Return from Call	Restore Upper
BISR	Begin ISR	Save Lower	RSLCX	Restore Lower Context	Restore Lower
SVLCX	Save Lower Context	Save Lower	RSLCX	Restore Lower Context	Restore Lower
STLCX	Store Lower Context	Store Lower	LDLCX	Load Lower Context	Load Lower
STUCX	Store Upper Context	Store Upper	LDUCX	Load Upper Context	Load Upper

4.3 CSAs and Context Lists

As previously mentioned, the upper and lower contexts are saved in CSAs. Unused CSAs are linked together in the **FREE CONTEXT LIST**. CSAs that contain saved upper or lower contexts are linked together in the **PREVIOUS CONTEXT LIST**. Figure 31 shows a simple configuration of CSAs within both context lists.

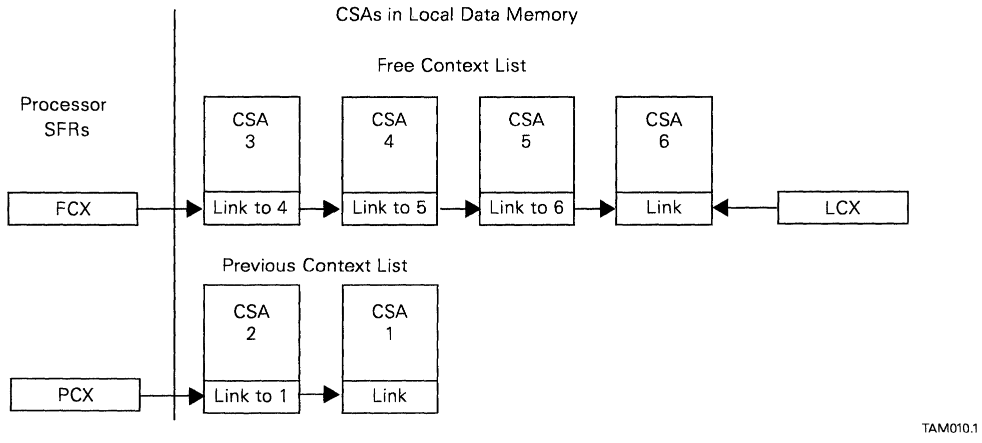


Figure 31: CSAs in Context Lists

The contents of the FCX register always points to an available CSA in the free context list. That CSA's link word points to the next available CSA in the free context list. Before an upper or lower context is saved in the first available CSA, its link word is read, supplying a new value for the FCX. To the memory subsystem, context saving is therefore a read/modify/write operation. The new value of FCX, which points to the next available CSA, is available immediately for subsequent upper or lower context saves.

The LCX register points to the last CSA in the free list and is used to recognize impending CSA list underflow. If the value of FCX used on a context save matches the limit value, the context save operation completes but the target address is forced to the CSA list depletion trap entry (FCD trap). The action taken by the trap handler depends on the implementation; it might issue a system reset, if it is determined that the CSA list depletion resulted from an unrecoverable software error. Normally, however, it will extend the free list, either by allocating additional memory, or by terminating one or more tasks and reclaiming their CSA call chains. In those cases, the trap handler will exit with a return from exception instruction (RFE).

The PCXI.PCX field points to CSA where the previous context was saved. The PCXI.UL bit identifies whether the saved context is upper or lower (1 = upper; 0 = lower). If the type does not match the type expected when a context restore operation is performed, an exception occurs and a context management trap is taken.

After being saved with the upper context, the return address register (RA) is loaded with the interrupting PC (if an exception or interrupt occurred) or the function return address (if a CALL instruction was executed). RA also supplies the saved PC value when the lower context is saved; it is loaded from the saved PC value when the lower context is restored.

The Call Depth Control field (PSW.CDC) consists of two subfields: a call depth counter, and a mask that determines the width of the counter and when it overflows. The call depth counter is incremented on calls, and is restored to its previous value on returns. An exception occurs when the counter overflows. Its purpose is to prevent software errors from causing "runaway recursion" and depleting the CSA free list. Refer to Section 7.1.1, "PSW Protection Fields," on page 77 for a more detailed description of the use of the call depth counter.

4.4 Context Switching with Interrupts

When an interrupt occurs, the processor saves the context of the current task in memory and suspends execution of the current task. The processor then starts execution of the interrupt handler. An interrupt is asynchronous. All registers must be saved in order to ensure that the register(s) that the interrupted task is using are saved.

When an interrupt is taken and the processor was not previously using the interrupt stack (PSW.IS bit = 0), then after being saved with the upper context of the interrupted task, the stack pointer (SP) is loaded with the current contents of the interrupt stack pointer (ISP). The PSW.IS bit is then set to one to indicate execution from the interrupt stack.

The Interrupt Control Register (ICR) holds the current CPU priority number (ICR.CCPN) and the interrupt enable bit (ICR.IE). These fields, along with the previous CPU priority number (PCXI.PCPN), and pending interrupt priority number (ICR.PIPN) are all part of the interrupt management system. PIPN is output from the Interrupt Control Unit, and is the priority number of the highest priority pending interrupt. A non-zero value in this register indicates the presence of a pending interrupt. For the interrupt to be serviced, PIPN must be greater than ICR.CCPN, and the interrupt enable bit (ICR.IE) must be set.

PCXI.PCPN is used just before loading the previous context on a call or interrupt return. The states of PCXI.PCPN and the previous interrupt enable bit (PCXI.PIE) allow predetermination of whether a pending interrupt that was previously blocked should be serviced now. If PIPN is greater than PCPN and PCXI.PIE is set, then instead of restoring the previous context, the control logic takes the interrupt by forcing a branch to the interrupt handler for the pending interrupt. This "interrupt folding" avoids an unnecessary context load that restores the previous context, followed by an immediate context save that services the pending interrupt. It is particularly helpful in the case where software posted interrupts are used frequently, as a means for ISRs to safely lower their execution priority, or as a means to access the RTOS task dispatcher after an interrupt has been serviced, and there are no other pending interrupts. See Chapter 5, "Interrupt System," for details on the interrupt system and software-posted interrupts.

PCXI.PCPN and ICR.CCPN are logically part of the current processor state. However, they are not part of the state that an RTOS needs to deal with for software-managed tasks, because they are zero for all software-managed tasks (SMTs). ICR.CCPN is non-zero only within ISRs, where it is used to order interrupt servicing. Accordingly, it is held in a register that is separate from the PSW, and is not part of the context that the RTOS handles for switching among SMTs. On an interrupt, the CCPN value becomes the PCPN value, after saving the old PCPN value along with the old PCXI value in the CSA for the upper context.

Once the interrupt is handled, the saved context is reloaded and execution of the interrupted task is resumed.

On an interrupt, half of the current task context is saved by hardware as an implicit part of the interrupt sequence. For small interrupt handlers that can execute entirely within the set of registers saved on the interrupt, no further context saving is needed. The interrupt handler can execute immediately and return, leaving the unsaved portions of the interrupted task's context untouched. For interrupt handlers that make calls, only one additional instruction is needed to save the registers that were not saved as part of the interrupt sequence. That instruction must be issued before any of the associated registers are modified, but it need not be the first instruction in the handler. Interrupt handlers with critical response time requirements can perform their initial, time-critical processing immediately, using registers that were already saved when the interrupt was taken. After that, they can save the remaining registers of the interrupted task's context, and continue with less time-critical processing.

Refer to Chapter 5, "Interrupt System," for more information.

4.5 Context Switching with Function Calls

When a function call is made (the CALL instruction is executed), the context of the calling routine must be saved and then restored, in order to resume the caller's execution after return from the function.

On a function call, the entire set of non-volatile registers (those registers whose contents are preserved across context switches) is saved by hardware. Furthermore, the saving of the non-volatile registers is integrated with the CALL instruction, so it happens in parallel with the call jump. Likewise, the restoring of the registers is integrated with the RET instruction, and happens in parallel with the return jump. The called function need not concern itself with saving and restoring the caller's context, and it is freed of any need to minimize the number of non-volatile registers that it uses.

The calling function and called functions can cooperate to minimize the amount of context that must be saved and restored. The general-purpose registers (GPRs) are partitioned into two subsets: those whose contents are preserved across the call (non-volatile registers), and those whose contents are not preserved (scratch registers). The caller is responsible for preserving any of its context that resides in scratch registers before the call, while the called function is responsible for preserving the caller's values in any non-volatile registers that the called function uses. To preserve its scratch register context, when necessary, the calling function either saves the registers in memory or copies them to non-volatile registers. The compiler's register allocator tries to minimize the need for either action, by tracking what data items are live across a call—defined before the call and used after it—and allocating those items to non-volatile registers. Likewise, the compiler tries to minimize the amount of context saving and restoring in the called function by minimizing the number of non-volatile registers that it uses.

4.6 Context Save/Restore Examples

This section provides an example of a context save operation and another example of a context restore operation.

4.6.1 Context Save

Figure 32 on page 52 shows the free and previous context lists for this example. The free context list contains three free CSAs (3, 4, and 5), and the previous context list contains two CSAs (2 and 1). The FCX points to CSA3, the first available CSA. The link word of CSA3 points to CSA4; the link word of CSA4 points to CSA5. The PCX points to the top CSA in the previous context list. The link word of CSA2 points to CSA1. CSA1 contains the saved context prior to CSA2.

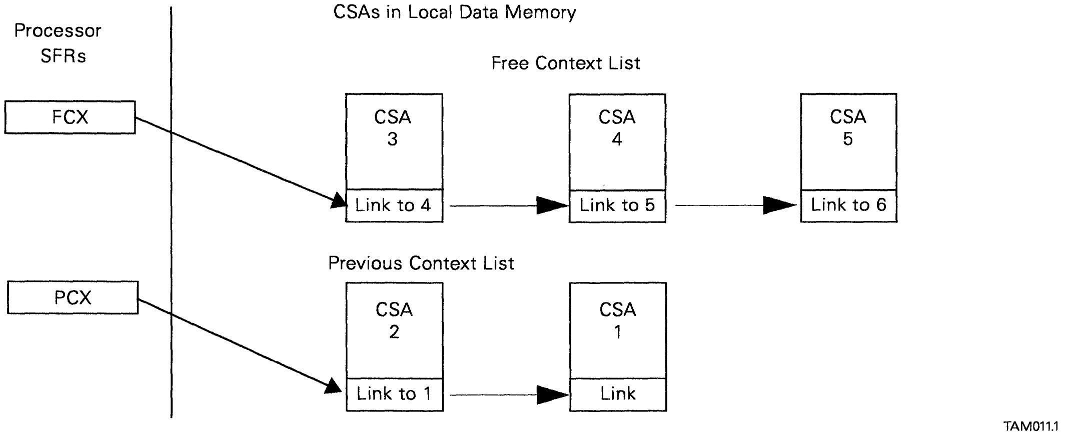
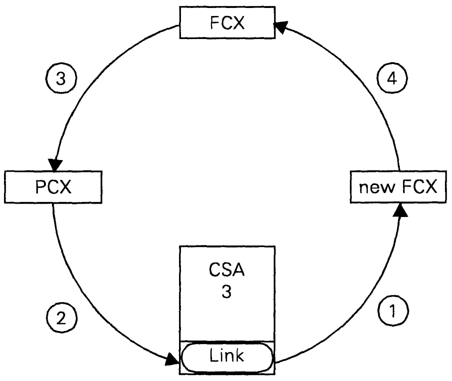


Figure 32: CSAs and Processor State Prior to Context Save

When the context save operation is performed, the first CSA in the free context list (CSA3) is pulled off and is placed on the front of the previous context list. Figure 33 shows the steps taken during the context save operation. The numbers in the figure correspond to the steps below:

1. The contents of the link word in CSA3 are loaded into the new FCX. The new FCX will now point to CSA4. Note that the new FCX is an internal buffer and is not accessible by the user.
2. The contents of the PCX are written into the link word of CSA3. The link word of CSA3 now points to CSA2.
3. The contents of the old FCX are written into the PCX. The PCX now points to CSA3, which is at the front of the Previous Context List.
4. The new FCX is loaded into the FCX.



TAM012.1

Figure 33: CSA and Processor SFR Updates on a Context Save Process

The processor SFRs and CSAs now look as shown in Figure 34. The processor context to be saved is now written into the rest of CSA3.

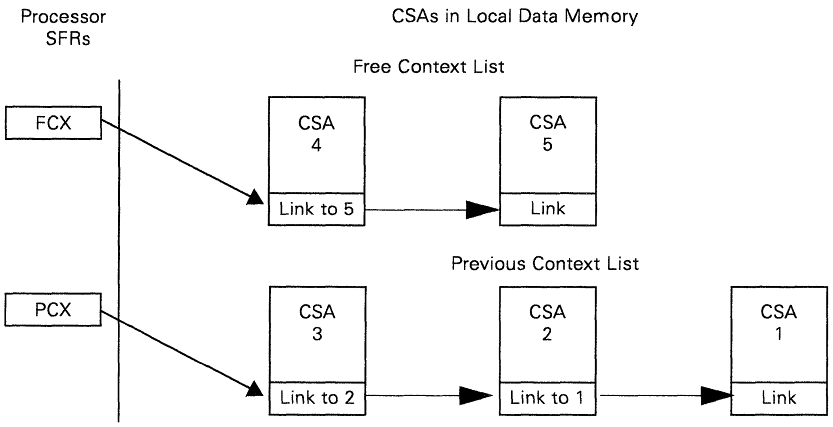


Figure 34: CSAs and Processor State After Context Save

4.6.2 Context Restore

Figure 35 shows an example where the previous context list contains three CSAs (3, 2, and 1) and the free context list contains two CSAs (4 and 5). The FCX points to CSA4, the first available CSA in the free context list. PCX points to CSA3, the most recently saved CSA in the previous context list. The link word of CSA3 points to CSA2; the link word of CSA2 points to CSA1; the link word of CSA4 points to CSA5.

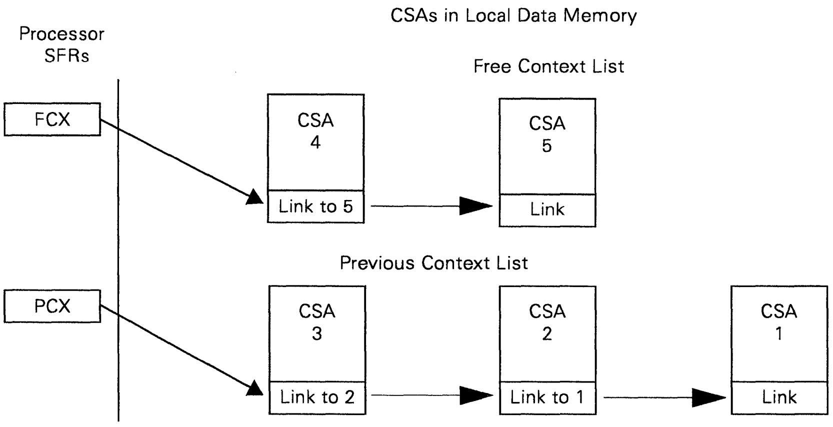


Figure 35: CSAs and Processor State Prior to Context Restore

When the context restore operation is performed, the first CSA in the previous context list (CSA3) is pulled off and is placed on the front of the free context list. Figure 36 shows the steps taken during the context restore operation. The numbers in the figure correspond to the steps below:

1. The contents of the link word in CSA3 are loaded into the new PCX. The new PCX will now point to CSA2. Note that the new PCX is an internal buffer and is not accessible by the user.
2. The contents of the FCX are written into the link word of CSA3. The link word of CSA3 now points to CSA4.
3. The contents of the old PCX are written into the FCX. The FCX now points to CSA3, which is at the front of the free context list.
4. The new PCX is loaded into the PCX.

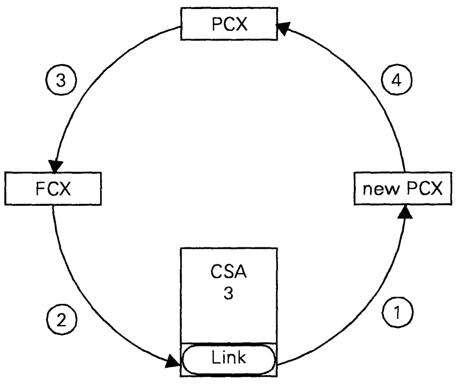


Figure 36: CSA and Processor SFR Updates on a Context Restore Process

The processor SFRs and CSAs now look as shown in Figure 37. The restored context now is written into the upper or lower context registers.

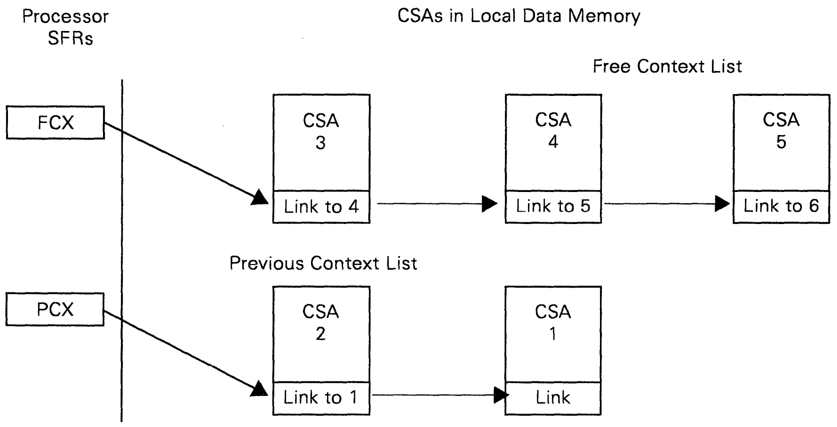


Figure 37: CSAs and Processor State After Context Restore

Interrupt System

SIEMENS

5 Interrupt System

This chapter describes the interrupt system, including arbitration, the priority level scheme, and the access of the vector table.

5.1 System Overview

Multiple sources can interrupt the TriCore device including internal peripherals, external inputs, and software (see Figure 38).

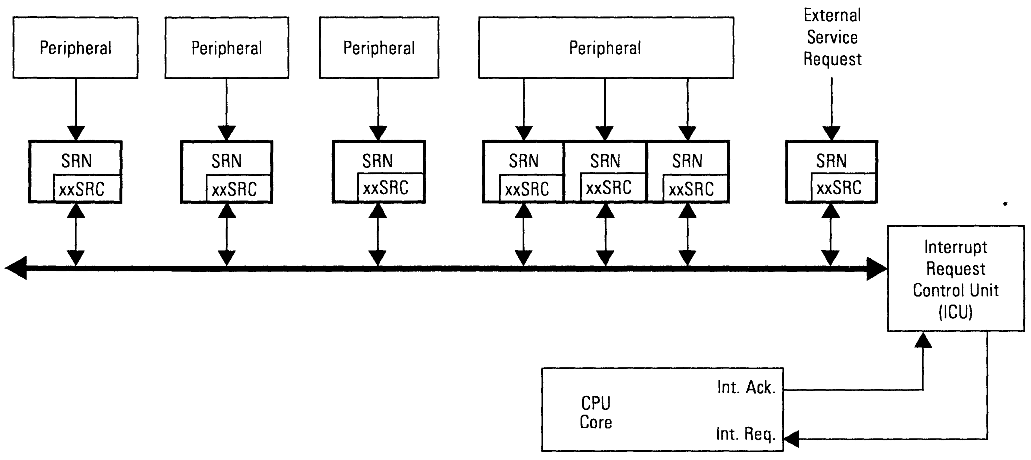


Figure 38: Block Diagram of Interrupt System

Every service request line from a peripheral or an external input is connected to a service request node (SRN). Each SRN contains a Service Request Control Register (xxSRC), where xx refers to the requesting source. The xxSRC register contains control bits and a priority number (SRPN) that the Interrupt Request Control Unit (ICU) uses when handling the interrupt. Up to 255 priority numbers can be assigned. The CPU core also is assigned a priority number (ICR.CCPN). When an interrupt occurs, the ICU determines which source will win arbitration, including the CPU.

The TriCore architecture requires that the xxSRC Register looks as shown in Figure 39.

15 11	10	9	8	7 0
Res	xxSR	xxSRE	xxTS	SRPN

Figure 39: Service Request Control Register (xxSRC)

xxSR — Service Request (Bit 10)

This bit indicates whether a service request has occurred.

0	A service request is not pending
1	A service request is pending

xxSRE — Service Request Enable (Bit 9)

This bit enables service requests.

0	Service requests are disabled
1	Service requests are enabled

xxTS — Type of Service (Bit 8)

This bit specifies the type of interrupt service.

0	Interrupt service is requested
1	The type of service is implementation-specific (for example, it can be used to request DMA service).

SRPN — Service Request Priority Number (Bits 7:0)

This eight-bit field determines the priority of the request and the entry point into the interrupt vector table. This number must be unique among all SRNs requesting the same type of service.

5.2 The Service Request Priority Number (SRPN)

The SRPN of a service request indicates its priority with respect to other sources requesting CPU service and to the priority of the CPU itself. Each SRPN used in a system must be unique; no sources are allowed to use the same SRPN (except for the default SRPN of 0x00, which excludes an SRN from taking part in the arbitration). The range for the SRPN depends on the number of interrupt sources used in a system. The interrupt arbitration scheme allows up to 255 sources to be active at one time. This value does not limit the number of sources that can be implemented in a Trillium derivative. More than 255 service request nodes can be implemented in future derivatives, however, only a subset of 255 can be used at a time to request an interrupt service; all others must be disabled.

The SRPN also identifies the entry into the interrupt vector table. Unlike other interrupt systems, the Trillium vector table provides an entry for each priority number, not for a specific interrupt source. In this way, the vector table is decoupled from the specific peripherals implemented in the various future derivatives, and a single peripheral can have multiple entry points for different purposes.

5.3 The Interrupt Control Unit (ICU)

The ICU manages the interrupt system and performs all the actions necessary to arbitrate incoming interrupt requests, to find the one with the highest priority, and to determine whether to interrupt the CPU or not.

The ICU contains an Interrupt Control Register (ICR), which holds the current CPU priority number (CCPN), the global interrupt enable/disable bit (IE), the pending interrupt priority number PIPN, as well as two bits to control the required number of interrupt arbitration cycles. Figure 40 shows the ICR. Refer to Section 3.6.1, "Interrupt Control Register (ICR)," on page 36 for detailed descriptions of the ICR bits.

31 … 26	25 24	23 … 16	15 … 9	8	7 … 0
Res	ARBCYC	PIPN	Res	IE	CCPN

Figure 40: ICR Register

5.4 Interrupt Arbitration

When an interrupt service is requested by one or more sources, these requests are serviced depending on their priority ranking. Thus the TriCore architecture must determine which request has the highest priority each time. The TriCore architecture implements a scheme that performs the arbitration in parallel with normal CPU operation. The Interrupt Control Unit controls this scheme, which takes place in several arbitration cycles over the arbitration bus. The arbitration bus connects the ICU with all service request nodes. The number of arbitration cycles is implementation-specific.

The ICU automatically starts an arbitration round when a new interrupt request is detected. At the end of the arbitration, the ICU has detected the service request with the highest priority number. It stores this number in the Pending Interrupt Priority Number field (PIPN) of register ICR.

The ICU checks the CPU's current priority number CCPN in register ICR against the PIPN. The CPU can be interrupted only if PIPN is greater than CCPN. If this is the case, the ICU generates an interrupt request to the CPU. If the CPU can enter the service routine, it acknowledges the ICU, which in turn activates an acknowledge cycle over the arbitration bus to inform the 'winner' node that it will be serviced. This node then resets its service request flag.

Several conditions could block the CPU from immediately responding to the interrupt request, even if the priority of the request is higher than the CCPN:

- The interrupt system is globally disabled (ICR.IE = 0)
- The CPU operates on the highest possible priority level (CCPN = 0xFF)

- The CPU is in the process of entering an interrupt service or trap routine
- The CPU is in the process of returning from an interrupt service or trap routine
- The CPU is operating on non-interruptible trap services
- The CPU is in the process of changing state in the power management
- The CPU executes a multi-cycle instruction
- the CPU is executing an instruction which modifies the conditions of the interrupt system, such as modifying the ICR

The CPU will respond to the interrupt request when these conditions are not true anymore.

If the priority of the CPU is greater than or equal to the detected PIPN, no immediate further actions are performed. The ICU goes into an idle state until one of the following conditions is true:

- A new arbitration round will be started only when a new service request is detected. In this case, the PIPN is first set to 0 to indicate it is invalid. (The new request might have a higher priority, then this will be the new PIPN. If the new request has a lower priority then the previous PIPN, the previous priority number will be detected again).
- If the current CPU priority number is changed due to explicit software modification or through the return from an interrupt, the pending interrupt will be serviced if the new CCPN is lower than the PIPN. Otherwise, no actions are performed and the service request is left pending.

Note that an arbitration is performed when a new service request is detected, regardless of whether the interrupt system is globally enabled or not, or whether there are other conditions preventing the CPU from servicing interrupts. In this way, the PIPN field always reflects the pending service request with the highest priority. This scheme also has the advantage of reducing the power consumption because arbitrations are not performed continuously but only when required.

Having the PIPN as an indication on a pending interrupt request also allows an immediate reaction on the return from an interrupt or trap routine if the priority of the pending request is greater than the one of the task which is returned to. The Trillium architecture immediately checks whether PIPN is greater than the CCPN of the interrupted task and directly performs a branch to the new interrupt service routine if this is the case. This "interrupt folding" saves time and reduces power consumption through avoiding the unnecessary context restore and save operations.

5.5 Entry into an Interrupt Service Routine (ISR)

When all conditions are clear for the CPU to service an interrupt request, the following actions are performed:

1. The upper context of the current task is saved.
2. The interrupt system is disabled (ICR.IE = 0).
3. The current CPU priority number (CCPN) is set to PIPN.
4. The PSW is set to a default value.

5. The interrupt vector table is accessed to fetch the first instruction of the interrupt service routine (ISR).

5.5.1 Default State of the PSW upon an Interrupt

The default state of the PSW upon occurrence of an interrupt is defined as follows:

1. All permissions are enabled.
2. Memory protection using the interrupt memory protection map (PSW.PRS) is enabled.
3. The stack pointer bit is set for using the interrupt stack.
4. The call depth counter is cleared, and the call depth limit selector is set for 64.

5.5.2 The Interrupt Vector Table

The interrupt vector table is organized according to the priority number of the interrupts. The priority of the arbitration winner, determined automatically at the end of an arbitration round, identifies the entry into the vector code. Interrupt latency is reduced because the extra cycle for the transfer of an identifier can be omitted.

The interrupt handler vectors are stored in code memory. The BIV register specifies the base address of the interrupt vector code. The vectors are made up of a number of short code segments, evenly spaced by eight words.

If an interrupt handler is very short, it may fit entirely within the eight words available in the vector code segment. Otherwise, it should contain some initial instructions, followed by a jump to the rest of the handler.

The size of the vector code depends only on the number of interrupts actually used in a system. Up to 256 vector entries, for 256 distinct interrupt handlers, are supported, but systems requiring fewer interrupt sources need not dedicate the full 256 entry's worth of memory required by the largest configurations.

When the CPU takes an interrupt, the interrupt priority number associated with the interrupt is used to index into the interrupt vector code. This number, detected by the ICU as PIPN and then taken as the new CCPN, is left-shifted by five bits and OR-ed with the address in the BIV register to generate the entry address of the interrupt handler.

The BIV address must be aligned on a power of two boundary, sufficient to generate correct interrupt vector addresses without using addition. Alignment to an 8-KByte boundary is sufficient for the full range of 256 interrupt sources. If fewer sources are used, the alignment requirements can be relaxed.

The BIV register accommodates partitioning of internal memory between RAM and one or more types of ROM. Its default on power-up is a fixed value, which is normally the base address for internal code ROM. However, the BIV register can be written to using the MTCR instruction during the power-on/reset phase of execution, before interrupts are enabled.

5.6 Interrupt Priority Levels

The interrupt system of the TriCore architecture is a flexible, programmable priority-leveling scheme. All service requests are assigned priority numbers (SRPNs), including the CPU.

Different service requests must be assigned different priority numbers. The maximum number of interrupting sources is 255. Programmable options range from one priority level with 255 sources up to 255 priority levels with one source each.

Interrupt numbers are assumed to be assigned in linear order of interrupt priority. This is feasible, because interrupt numbers are not hardwired to individual sources. They are assigned by software executed during the power-on boot sequence.

Disabling the interrupt system and setting the new CCPN to PIPN (the priority of the interrupt request which is now serviced) on entry into an ISR will block interrupts of equal or lower priority than the currently serviced interrupt when the interrupt system is enabled again. However, the interrupt service routine can set the CCPN to any value (usually a higher value) before enabling interrupts, thereby blocking an entire group of interrupts (including a reoccurrence of the current interrupt). This capability results in a set of effective priority levels on top of the individual priority numbers in the SRNs.

To group multiple interrupt sources into the same priority level, set the CCPN in each ISR to the priority number of the service request with the highest SRPN in that priority group. Each time the CPU services an interrupt that is part of a priority group, its CCPN is set to the highest priority number of that group. This service cannot be interrupted by another source within that same group because none has a higher priority.

Interrupt service routines are easily divided into parts with different priorities. For example, an interrupt is placed on a very high priority because response time and reaction to an event is very critical. The necessary actions are carried out immediately on that high-priority level. Then the priority level of this interrupt is lowered, and the interrupt request bit is set again (indicating a pending interrupt) while still in the service routine. Returning to the interrupted program terminates the service routine. The pending interrupt is serviced when the CPU priority is lower than its own. After entering the service routine, which now can be at a different address in the program memory, the outstanding but low-priority actions of the interrupt can be performed.

The priority of a service request might be low because the response time to an event is not critical. But, once it has been granted service, this service should not be interrupted. To prevent any interruption, the TriCore architecture allows the priority level of this service request to be raised within the ISR, and also allows interrupts to be disabled.

5.7 Enabling/Disabling the Interrupt System

There are several ways to enable or disable the interrupt system:

1. The ENABLE and DISABLE instructions set or clear ICR.IE.
2. The BISR instruction automatically enables the interrupt system (ICR.IE = 1).

3. The MTCR instruction can be used to set or clear the ICR.IE bit.

The first two options are recommended because their actions are synchronized with the pipeline operation. Using the MTCR instruction to directly modify register ICR is only recommended together with an ISYNC instruction (synchronize instruction stream) in order to avoid unexpected pipeline side effects.

5.8 Special Handling of Interrupt Requests

Interrupts are normally generated in response to interrupt requests coming from an external hardware source. However, it is also possible for software posted interrupts to be generated in response to software actions.

5.8.1 Software-Posted Interrupts

A software-posted interrupt is a true hardware interrupt, carrying an interrupt priority that is processed through the regular interrupt subsystem to determine when the interrupt is taken. The only difference is that the interrupt request is generated by setting the service request bit in a service request node explicitly, through a software update of the node's control register.

Once the interrupt request bit in a service request node has been set, there is no way to distinguish between a software-posted interrupt request and a true hardware interrupt request. For that reason, it is generally advisable to use service request nodes and interrupt priority numbers for software posted interrupts that are not used for hardware interrupts.

5.8.2 Interrupt One

Interrupt 1 is the first and lowest priority entry in the interrupt vector. It is best used for ISRs performing task management. ISRs whose actions affect the launching of software-managed tasks will post a software interrupt request at priority level one to signal the change. (Normally, the posting is not done from the ISR directly, but from RTOS code in a service function called from the ISR.) The ISR then can execute a normal return from interrupt, rather than jumping to an ISR exit function in the kernel. There is no need for an exit function to check whether the ISR is returning to the background task level or to a lower priority ISR that it interrupted, in order to determine when to invoke the task dispatch function.

When there is a pending interrupt at a priority higher than the return context for the current interrupt, the return from interrupt effectively becomes a jump to the new ISR.

Traps

6

Traps

A trap occurs as a result of an event such as a non-maskable interrupt, an instruction exception, or illegal access. Traps are always active; they cannot be disabled by software action.

This chapter describes the different traps that can occur and the TriCore architecture's trap handling mechanism.

6.1 Trap Types

The Trillium architecture contains eight trap classes. These traps are further classified as synchronous or asynchronous, and hardware or software. Each trap is assigned a Trap Identification Number (TIN), that identifies the cause of the trap within its class. The TIN is loaded into register D15 before the first instruction of the trap handler is executed.

Table 7 summarizes and classifies all TriCore-supported traps.

Table 7: Supported Traps

Trap ID # (TIN)	Trap Name	Sync/Async	Hardware/ Software	Description
Class 0 — Reset				
0	RESET	Synchronous	Hardware	System reset; raised at end of hardware reset sequence, with hardware in known state
Class 1 — Internal Protection Traps				
1	PRIV	Synchronous	Hardware	Privileged Instruction
2	MPR	Synchronous	Hardware	Memory Protection: Read Access
3	MPW	Synchronous	Hardware	Memory Protection: Write Access
4	MPX	Synchronous	Hardware	Memory Protection: Execution Access
5	MPP	Synchronous	Hardware	Memory Protection: Peripheral Access
6	MPN	Synchronous	Hardware	Memory Protection: Null Address
7	GRWP	Synchronous	Hardware	Global Register Write Protection
Class 2 — Instruction Errors				
1	IOPC	Synchronous	Hardware	Illegal Opcode
2	UOPC	Synchronous	Hardware	Unimplemented Opcode
3	OPD	Synchronous	Hardware	Invalid operand specification
4	ALN	Synchronous	Hardware	Data address alignment error
5	MEM	Synchronous	Hardware	Invalid local memory address
Class 3 — Context Management				
1	FCD	Synchronous	Hardware	Free context list depleted (FCX == LCX)
2	CDO	Synchronous	Hardware	Call depth overflow
3	CDU	Synchronous	Hardware	Call depth underflow
4	FCU	Synchronous	Hardware	Free context list underflow (FCX == 0)
5	CSU	Synchronous	Hardware	Context list underflow (PCX == 0)
6	CTYP	Synchronous	Hardware	Context type error (PCXI.UL wrong)
7	NEST	Synchronous	Hardware	Nesting error: RFE with non-zero call depth

Table 7: Supported Traps(*Continued*)

Trap ID # (TIN)	Trap Name	Sync/Async	Hardware/ Software	Description
Class 4 —.System Bus and Peripheral Errors				
1	PRVP	Asynchronous	Hardware	Privilege violation on peripheral access
2	BUS	Asynchronous	Hardware	Bus error
3	PARI	Asynchronous	Hardware	Parity / CRC error
4	BLTO	Asynchronous	Hardware	Bus Lock Time-out
5	PKEY	Asynchronous	Hardware	Key violation for protected peripheral (bad source value)
Class 5— Assertion Traps				
1	OVF	Synchronous	Software	Arithmetic overflow
2	SOVF	Synchronous	Software	Sticky arithmetic overflow
Class 6 — System Call				
See footnote[1]	SYS	Synchronous	Software	System call
Class 7 — Non-Maskable Interrupt				
0	NMI	Asynchronous	Hardware	Non-maskable interrupt

1. For the system call trap, the TIN is taken from the immediate constant specified in the SYSCALL instruction. The range of values that may be specified is 0 to 255, inclusive.

6.1.1 Synchronous Traps

Synchronous traps are associated with the execution or attempted execution of specific instructions. The instruction causing the trap is known precisely. The trap is taken immediately and serviced before execution can proceed beyond that instruction.

6.1.2 Asynchronous Traps

Asynchronous traps are similar to interrupts, in that they are associated with hardware conditions detected externally and signaled back to the core. Some result indirectly from instructions that have been previously executed, but the direct association with those instructions has been lost. Others, such as the non-maskable interrupt, are external events. The difference between an asynchronous trap and an interrupt is that asynchronous traps are routed via the trap vector instead of the interrupt vector code and cannot be masked.

6.1.3 Hardware Traps

Hardware traps are generated as a result of certain TriCore instructions. Examples are the illegal instruction trap, memory protection traps, and data memory address misalignment traps. When a hardware trap condition is detected, the control logic supplies a two-part number that identifies the cause of the trap to the hardware's trap entry logic. The first part is a three-bit trap class number; the second part is an eight-bit Trap Identification Number (TIN). The trap class number is left-shifted by five and ORed with the BTV register value to generate the address of the handler for that trap class. The TIN is loaded into the trap handler's D15 register, to further identify the cause of the trap.

6.1.4 Software Traps

Software traps include system calls and the assertion traps. Through the SYSCALL instruction, an application code can call a system function whose execution requires a permission that has not been allocated to the calling code. There is a single trap vector entry for all system calls. The specific system function desired is identified by an immediate constant specified in the SYSCALL instruction, which becomes the TIN for the SYSCALL trap.

6.2 Trap Handling

This section describes the trap handling mechanisms supported by the TriCore architecture. The actions taken on traps are slightly different than those taken on external or software interrupts. Trap handlers reside in a different vector from interrupt handlers. The return PC saved in the return address register is the PC of the instruction that caused the trap. For an interrupt, the return PC is that of the instruction that would have been executed next, if the interrupt had not been taken. A trap does not change the CPU's interrupt priority, so the ICR.CCPN field is not updated.

6.2.1 Trap Vector Format

The trap handler vectors are stored in code memory in the trap vector table. The BTV register specifies the base address of the trap vector table. The vectors are made up of a number of short code segments, evenly spaced by eight words.

If a trap handler is very short, it may fit entirely within the eight words available in the vector code segment. Otherwise, it should contain some initial instructions, followed by a jump to the rest of the handler.

6.2.2 Accessing the Trap Vector Table

When a trap occurs, a trap identifier is generated by hardware. The trap identifier has two components: the trap class number, used to index into the trap vector table, and the trap identification number (TIN), which is loaded into D15. The trap class number is left shifted by five bits and ORed with the address in the BTV register to generate the entry address of the trap handler.

6.2.3 Default State upon a Trap

The default state when a trap occurs is defined as follows:

1. All permissions are enabled.
2. Memory protection using the interrupt memory protection map (PSW.PRS = 00_2) is enabled.
3. The stack pointer bit is set for using the interrupt stack.
4. The call depth counter is cleared, and the call depth limit selector is set for 64.

5. Interrupts are disabled; they remain disabled until explicitly enabled.
6. The ICR.CCPN remains unchanged.

Although traps leave the ICR.CCPN unchanged, their handlers still begin execution with interrupts disabled. They can therefore perform critical initial operations without interruptions, until they specifically re-enable interrupts.

Protection System

7

Protection System

Protection is increasingly important as embedded applications increase in size and complexity. The focus for embedded systems is different than it is for workstations and PCs, because embedded systems normally are not faced with the problem of maintaining their integrity against unknown and perhaps hostile user code. However, protection capabilities are useful for protecting core system functionality from bugs that may have slipped through testing. They are also important aids to testing and debugging.

The TriCore's protection system provides the essential features needed to isolate errors and facilitate debugging. It protects critical system functions against both software and transient hardware errors. The TriCore protection system is unobtrusive, imposing little overhead and avoiding non-deterministic run-time behavior.

This chapter describes the hardware operation of the protection system. In addition, later sections introduce the use of the protection features by software in real-time systems.

7.1 Protection System Registers

There are two major components to the protection system:

1. The control bit fields in the PSW.
2. The memory protection registers which control program execution and memory access.

Chapter 3, "Core Registers," describes these registers in detail.

7.1.1 PSW Protection Fields

The control fields in the PSW that deal with the protection system are shaded in the figure below. Their functions are described after the figure. (The other PSW fields are described in Section 3.3, "Program State Information (PC, PSW, and PCXI)," on page 30.)

31	30	29	28	27	26 … 14	13 12	11 10	9	8	7	6 … 0
C	V	SV	AV	SAV	Res	PRS	IO	IS	GW	CDE	CDC

7.1.1.1 PRS

The PRS field selects one of up to four sets of memory protection register values controlling load and store operations and instruction fetches within the current process. This field indicates the current protection register set. See Section 7.1.2, "Memory Protection Registers," on page 79, for a description of memory protection registers.

7.1.1.2 IO

The IO field determines the access level to special function registers (SFRs) and peripheral devices. There are three I/O privilege levels:

- 00 — User-0; no peripheral access. Used for tasks that have no requirement to directly access peripheral devices. Tasks at this level do not have permission to enable or disable interrupts.
- 01 — User-1; regular peripheral access. Enables access to common peripheral devices that are not specially protected. Typically includes read/write access to SIO ports and read access to timers and most I/O status registers. Tasks at this level may disable interrupts.
- 10 — Supervisor. Enables read/write access to core registers and protected peripheral devices.
- 11 — Reserved. This encoding is reserved and not defined.

7.1.1.3 IS

The IS bit determines whether the current execution thread is using the shared global (interrupt) stack or a user stack. A "1" in this bit indicates use of the interrupt stack; a "0" indicates use of the user stack. If an interrupt is taken when the IS bit is 0, then the stack pointer register is loaded from the ISP register before execution starts at the first instruction of the interrupt service routine.

7.1.1.4 GW

The GW bit controls whether the current execution thread has permission to modify the global address registers. Most tasks and ISRs will use the global address registers as "read only" registers, pointing to the global literal pool and key data structures. However, a task or ISR can be designated as the "owner" of a particular global address register, and is allowed to modify it.

The system designer must determine which global address variables are used with sufficient frequency and/or in sufficiently time-critical code to justify allocation to a global address register. By compiler convention, global address register A0 is reserved as the base register for short form loads and stores. Register A1 is also reserved for compiler use. Registers A8 and A9 are not used by the compiler, and are available for holding critical system address variables.

7.1.1.5 CDE

The CDE bit enables call depth counting, provided that the CDC mask field is not all 1's. It is one by default, but is cleared by the call trace trap handler to enable a trapped call to execute without re-trapping after return from the trap handler. It is then set again on execution of the CALL instruction.

7.1.1.6 CDC

The CDC field consists of two variable-width fields. The first is a mask field, consisting of a string of zero or more initial '1' bits, terminated by the first '0' bit. The remaining bits of the field are the call depth counter. The following table illustrates the division:

PSW.CDC Bits	Definition
0cccccc	6-bit counter; trap on overflow
10ccccc	5-bit counter; trap on overflow
110cccc	4-bit counter; trap on overflow
1110ccc	3-bit counter; trap on overflow
11110cc	2-bit counter; trap on overflow
111110c	1-bit counter; trap on overflow
1111110	trap every call (call trace mode)
1111111	disable call depth counting

When the call depth counter overflows, a trap is generated. Depending on the width of the mask field, the call depth counter can be set to overflow at any power of two boundary from 1 (2^0) to 64 (2^6). Setting the mask field to 1111110_2 allows no bits for the counter, and causes every call to be trapped. This is used for call tracing. Setting the field to mask field to 1111111_2 disables call depth counting altogether.

7.1.2 Memory Protection Registers

The memory protection model for the TriCore architecture is based on address ranges, with specific access permissions associated with each range. Ranges and their associated permissions are specified in two to four identical sets of tables residing in core SFR (CSFR) space. Each set is referred to as a **PROTECTION REGISTER SET**. A protection register set consists of Data Segment Protection Registers, Data Protection Mode Registers, Code Segment Protection Registers, and Code Protection Mode Registers (see Figures 41 through 44). Refer to Section 3.8, "Memory Protection Registers," for more details on these registers.

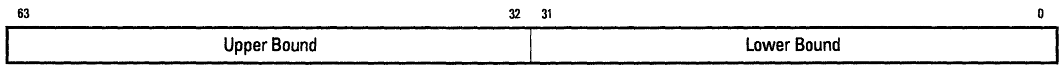


Figure 41: Data Segment Protection Register

63 … 32	31 … 0
Upper Bound	Lower Bound

Figure 42: Code Segment Protection Register Pair

7	6	5	4	3	2	1	0
WE	RE	WS	RS	WBL	RBL	WBU	RBU

Figure 43: Data Protection Mode Register

7	6	5	4	3	2	1	0
XE	Res	XS	Res	BL	Res	Res	BU

Figure 44: Code Protection Mode Register

At any given time, one of the sets is the **CURRENT PROTECTION REGISTER SET**, which determines the legality of memory accesses by the current task or ISR. The PRS field in the PSW indicates the current protection register set number.

Each protection register set contains separate address range tables for checking data accesses and code accesses. This reflects the fact that there are separate buses for data and program memory. The **RANGE TABLE ENTRY** is a pair of words specifying a lower and an upper bound for the associated range. The range defined by one range table entry is the address interval:

```
lower bound ≤ address < upper bound
```

Each range table entry has an associated mode table entry where access permissions and debug signal conditions for that range are specified. On load and store operations, data address values are checked against the entries in the data range table. On new instruction fetches, the PC value for the fetch is checked against the entries in the code range table. When an address is found to fall within a range defined in the appropriate range table, the associated mode table entry is checked for access permissions and debug signal generation.

The number of protection register sets in a TriCore derivative is implementation dependent. The minimum number in a conforming implementation is two, and the maximum number is four.

In a two-set implementation, one of the sets corresponds to the current background task, and the other is common to any interrupt service routine. (In this case "background" task means the control thread executes at hardware priority level 0 when the interrupt stack is empty.) This configuration allows taking an interrupt and then returning from the interrupt to the interrupted task without changing any protection register or address range table values. Only the selection of the active set of protection registers changes.

7.1.2.1 Modes of Use for Range Table Entries

Individual range table entries can be used just for memory protection or for debugging. One entry rarely is used for both purposes. If the upper and lower bound values have been set for debug breakpoints, they probably are not meaningful for defining protection ranges, and vice versa. However, it is both possible and reasonable to have some entries used for memory protection and other used for debugging.

To disable an entry for use in memory protection, clear both the RE and WE bits in a data range table entry or clear the XE bit in a code range table entry. The entry can be disabled for use in debugging by clearing any debug signal bits.

When a range entry is being used for debugging, the debug signal bits that are set determine whether it is used as a single range comparator (giving an in-range/not in-range signal) or as a pair of equal comparators. The two uses are not mutually exclusive.

7.1.2.2 Using Protection Register Sets

If there were only one protection register set, then either the mappings used would have to be general enough to apply to all tasks and ISRs—and hence not terribly useful for isolating software errors in individual tasks—or there would have to be a substantial overhead paid on interrupts and task context switches for updating the tables to match the currently executing task or ISR. By providing for multiple sets of tables, with two bits in the PSW to select the currently active set, those drawbacks are avoided.

Note that supervisor mode does not automatically disable memory protection. The protection register set that is selected for supervisor tasks will normally be set up to allow write access to regions of memory that are protected from user mode access. In addition, of course, supervisor tasks can execute instructions to change the protection maps, or to disable the protection system entirely. But supervisor mode does not implicitly override memory protection, and it is possible for a supervisor task to take a memory protection trap.

7.2 Sample Protection Register Set

Figure 45 illustrates Data Protection Register Set n, where n is one of the four sets as selected by the PSW.PRS field. Each register set in this example consists of four range table entries. The ranges defined can potentially overlap, or be nested. Nesting of ranges can be used, for example, to allow write access to a subrange of a larger range in which the current task is allowed read access.

The four Data Segment Protection Registers and four Data Protection Mode Registers are set up as follows:

- Data Segment Protection Register 3 (DPRn_3) defines the upper and lower bound for Data Range 4. Data Protection Mode Register 3 (DPMn_3) defines the permissions and debug conditions for Data Range 4.
- Data Segment Protection Register 2 (DPRn_2) defines the upper and lower bound for Data Range 3. Data Protection Mode Register 2 (DPMn_2) defines the permissions and debug conditions for Data Range 3. Note that Data Range 3 is nested within Data Range 4.
- Data Segment Protection Register 1 (DPRn_1) defines the upper and lower bound for Data Range 2. Data Protection Mode Register 1 (DPMn_1) defines the permissions and debug conditions for Data Range 2.

- Data Segment Protection Register 0 (DPRn_0) defines the upper and lower bound for Data Range 1. Data Protection Mode Register 0 (DPMn_0) defines the permissions and debug conditions for Data Range 1.

This same configuration can be used to illustrate Code Protection Register Set n.

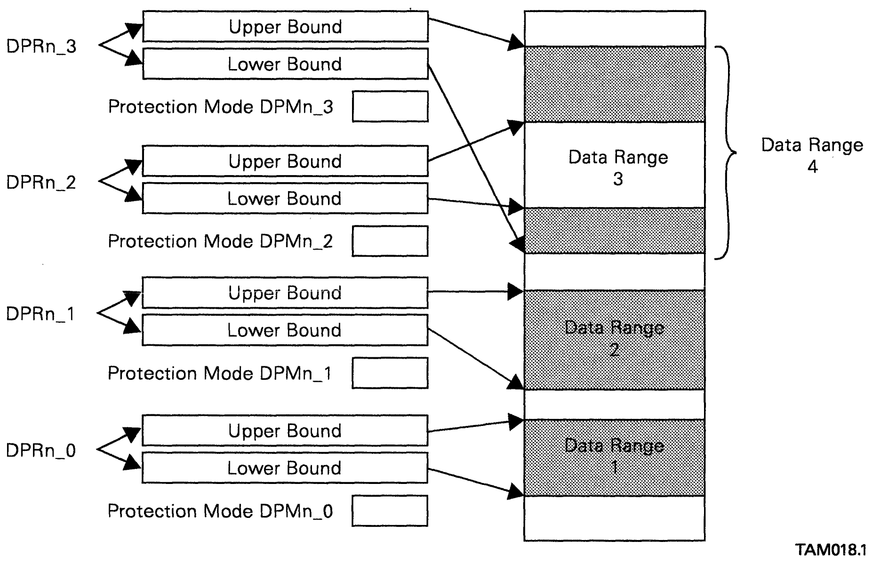


Figure 45: Example Configuration of a Data Protection Register Set

7.3 Memory Access Checking

When the protection system is enabled, every memory access (read, write, or execute) is checked for legality before the access is performed. The legality is determined by all of the following:

- the protection enable bits in the Syscon Register,
- the current I/O privilege level (0 = User-0; 1 = User-1; 2 = Supervisor), and
- the ranges defined in the currently selected protection register set.

Data addresses (read and write accesses) are checked against the currently selected data address range table, while instruction fetch addresses are checked against the code address range tables. The mode entries for the data range table entries enable only read and write accesses, while the mode entries for the code range table entries enable only execute access. In order for data to be read from program space, there must be an entry in the data address range table that covers the address being read. Conversely there must be an entry in the code address range table that covers the instruction being read.

Access to the internal and external peripherals is through the two upper segments of the TriCore address space (high-order address bits equal to 1110_2 and 1111_2). Access checking for addresses in the peripheral segments is independent of access checking in the remainder of the address space. Access to the peripheral segments is not allowed for tasks at I/O privilege level 0 (User-0 tasks). Tasks at I/O privilege 1 and higher have access rights to the peripheral segment space, however, the validity of any given access attempt depends on the presence of a peripheral at the accessed address, and any restrictions it may impose on its own access. Protected peripherals, for example, require that the I/O privilege be 2, as reflected by the supervisor line value on the system bus. Refer to Section 2.3, "Memory Model," on page 16 for the memory map showing the peripheral segments.

If the memory protection system is disabled, then any access to any memory address outside of the peripheral segments is permitted, regardless of the I/O privilege level. There are no memory regions reserved for supervisor access only, when the memory protection system is disabled.

When the memory protection system is enabled, for an access to be permitted, the address for the access must fall within one or more of the ranges specified in the currently selected protection register set. Furthermore, the mode entry for at least one of the matching ranges must enable the requested type of access.

7.3.1 Permitted vs. Valid Accesses

A memory access can be permitted within the ranges specified in the data and code range tables without necessarily being valid. A range specified in a range table entry could cover one or more address regions where no physical memory was implemented. Although that would normally reflect an error in the system code that set up the address range, the memory protection system only uses the range table entries when determining whether an access is permitted. In addition, if the memory protection system is disabled, all accesses must be taken as permitted, though individual accesses may or may not be valid.

An access that is not permitted under the memory protection system results in a memory protection trap. When permitted, an access to an unimplemented memory address results in a bus error trap, provided that the memory address is in one of the segments reserved for local memory. If the address is an external memory address, the result depends on the memory implementation, and is not architecturally defined.

An access can also be permitted but invalid due to a misaligned address. Misaligned accesses result in an alignment trap, rather than a protection trap.

7.3.2 Crossing Protection Boundaries

An access can straddle two regions. For example, Figure 46 illustrates the condition where Instruction A lies in an execute region of memory, Instruction C lies in a no-execute region of memory, and Instruction B straddles the execute/no execute boundary.

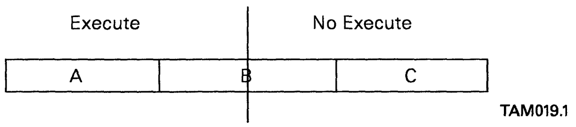


Figure 46: Protection Boundaries

Because the PC is used in the comparison with the comparator registers, the program error exception is not signaled until Instruction C is fetched. The same is true for all comparisons—the address of the first accessed byte is compared against the memory protection comparator registers. Hence, an access assumes the memory protection properties of the first byte in the access regardless of the number of bytes involved in the access.

For normal accesses, this assumption is not a problem, because the regions are set up according to the natural access boundaries for the code or data that the region contains. For wild accesses due to software or hardware errors, stores are the main concern. In the worst case, a doubleword store that is aligned on a halfword boundary can extend three halfwords beyond the end of the region in which its address lies.

One way to prevent boundary crossings is to leave at least three halfwords of buffer space between regions. This configuration prevents wild stores from destroying data in adjacent read-only regions, for example.

Instruction Set Overview

Instruction Set Overview

This chapter provides an overview of the TriCore instruction set architecture. The basic properties and usage of each instruction type are described, as well as the selection and usage of the 16-bit (short) instructions. The instructions are described individually in Chapter 9.

8.1 Arithmetic Instructions

Arithmetic instructions operate on data and addresses in registers. Status information about the result of the arithmetic operations is recorded in the five status flags in the Program Status Word (PSW). The status flags are described in Table 8.

Table 8: PSW Status Flags

Status Flag	Description
C	Carry. This flag is set as the result of a carry out from an addition or subtraction instruction. Carry out can result from either signed or unsigned operations. It is also set by automatic shift.
V	Overflow. This flag is updated by most arithmetic instructions. It is set when the result cannot be represented in the data size of the result; for example, when the result of a signed 32-bit operation is greater than $2^{31}-1$.
SV	Sticky Overflow. This flag is set when the overflow flag is set. It remains set until it is explicitly cleared by an RSTV (Reset Overflow bits) instruction.
AV	Advanced Overflow. This flag is updated by all instructions that update the overflow flag and no others. This flag is determined as the boolean exclusive of the two most-significant bits of the result.
SAV	Sticky Advanced Overflow. This flag is set whenever the advanced overflow flag is set. It remains set until it is explicitly cleared by an RSTV (Reset Overflow bits) instruction.

The two overflow conditions (overflow and advanced overflow) are calculated for all arithmetic instructions. In the case of packed instructions, the conditions are calculated for each byte or halfword (parallel) operation. In the case of the multiply-accumulate instructions, the conditions are calculated after the accumulate operation.

(parallel) operation. In the case of the multiply-accumulate instructions, the conditions are calculated after the accumulate operation.

Numerically, for signed 32-bit values, overflow occurs when a positive result is greater than 0x7FFF.FFFF or a negative result is smaller than 0x8000.0000. For unsigned 32-bit values, overflow occurs when the result of a 32-bit addition is greater than 0xFFFF.FFFF.

The status flags can be read by software using the Move From Core Register (MFCR) instruction and can be written using the Move to Core Register (MTCR) instruction. The Trap on Overflow (TRAPV) and Trap on Sticky Overflow (TRAPSV) instructions can be used to cause a trap if the V and SV bits, respectively, are set. The overflow bits can be cleared using the Reset Overflow Bits instruction (RSTV).

Individual arithmetic operations can be checked for overflow by reading and testing V. If one is only interested in knowing if an overflow occurred somewhere in an entire block of computation, then the SV bit is reset before the block (using the RSTV instruction) and tested after completion of the block (using MFCR). Jumping based on the overflow result can be done using a MFCR followed by a JEQZ.T (conditional jump on the value of a bit).

The AV and SAV bits are set as a result of the exclusive OR of the two most-significant bits of the particular data type (byte, halfword, word, or doubleword) of the result, which indicates that an overflow almost occurred.

Because most signal processing applications can handle overflow by simply saturating the result, most of the arithmetic instructions have a saturating version for signed and unsigned overflow. Note that saturating versions of all instructions can be synthesized using short code sequences.

When saturation is used for 32-bit signed arithmetic overflow, if the true result of the computation is greater than $(2^{31}-1)$ or less than -2^{31}, the result is set to $(2^{31}-1)$ or -2^{31}, respectively. The bounds for 16-bit signed arithmetic are $(2^{15}-1)$ and -2^{15}, and the bounds for 8-bit signed arithmetic are $(2^{7}-1)$ and -2^{7}. When saturation is used for unsigned arithmetic, the lower bound is always zero and the upper bounds are $(2^{32}-1)$, $(2^{16}-1)$, and $(2^{8}-1)$. Saturation is indicated in the instruction mnemonic by an "S" preceding the period (.), and unsigned is indicated by a "U" following the period (.). For example, the instruction mnemonic for a signed saturating addition is ADDS, and the mnemonic for an unsigned saturating addition is ADDS.U.

Saturation is also used for signed fractions in DSP operations, as described in Section 8.1.2, "DSP Arithmetic," on page 94 .

8.1.1 Integer Arithmetic

8.1.1.1 Move

The move instructions move a value in a data register or a constant value in the instruction to a destination data register, and can be used to quickly load a large constant into a data register. The least-significant 16-bits of a register are moved using MOV (which sign-extends the value to 32 bits) or MOV.U (which zero-extends to 32 bits). The MOVH (Move Highword) instruction loads a 16-bit constant into the most-significant 16 bits of the register and zero fills the least-significant 16 bits, which is useful for loading a left-justified constant fraction. Loading a 32-bit constant can be done using a

8.1.1.2 Addition and Subtraction

The addition instructions have three versions: no saturation (ADD), signed saturation (ADDS), and unsigned saturation (ADDS.U). For extended precision addition, the ADDX (Add Extended) instruction sets the PSW carry bit to the value of the ALU carry out. The ADDC (Add with Carry) instruction uses the PSW carry bit as the carry in, and updates the PSW carry bit with the ALU carry out. For extended precision addition, the least-significant word of the operands is added using the ADDX instruction, and the remaining words are added using the ADDC instruction. The ADDC and ADDX instructions do not support saturation.

Often it is necessary to add 16- or 32-bit constants to integers. The ADDI (Add Immediate) and ADDIH (Add Immediate High) instructions add a 16-bit, sign-extended constant or a 16-bit constant, left-shifted by 16. Addition of any 32-bit constant can be done using ADDI followed by an ADDIH.

All add instructions except those with 16-bit immediates have similar corresponding subtract instructions. Because the large immediate of ADDI is sign-extended, it may be used for both addition and subtraction.

The RSUB (Reverse Subtract) instruction subtracts a register from a constant. Using zero as the constant yields negation as a special case.

8.1.1.3 Multiply and Multiply-Add

Multiplication of two, 32-bit integers that produce a 32-bit result can be handled using MUL (Multiply Signed), MULS (Multiply Signed with Saturation), and MULS.U (Multiply Unsigned with Saturation). The MULM (Multiply with Multiword Result) and MULM.U (Multiply with Multiword Result Unsigned) instructions produce the full 64-bit result, which is stored to a register pair; MULM is for signed integers, and MULM.U is for unsigned integers. There are also special multiply instructions that are used for DSP operations, which are described in Section 8.1.2, "DSP Arithmetic."

The multiply-add instruction (MADD) multiplies two signed operands, adds the result to a third operand, and stores the result in a fourth operand. Because the operands do not use the same registers, the intermediate sums of a multi-term multiply-add instruction can be saved without requiring any additional register moves. The MADD, MADDS (Multiply-Add with Saturation), and MADDS.U (Multiply-Add with Saturation Unsigned) instructions operate on and produce 32-bit integers; MADDS and MADDS.U will saturate on signed and unsigned overflow, respectively. To add the 64-bit product to a 64-bit source and produce a 64-bit result, the instructions MADDM (Multiply-Add with Multiword Result), MADDM.U (Multiply-Add with Multiword Result Unsigned), MADDMS (Multiply-Add Multiword with Saturation), and MADDMS.U (Multiply-Add Multiword with Saturation Unsigned) can be used.

The set of Multiply-Subtract (MSUB) instructions, which supports the accumulation of products using subtraction instead of addition, provides the same set of variations as the MADD instructions.

8.1.1.4 Division

Division of 32-bit by 32-bit integers is supported for both signed and unsigned integers. Because an atomic divide instruction would require an excessive number of cycles to execute, a divide-step sequence is used, which keeps down interrupt latency. The divide step sequence allows the divide time to be proportional to the number of significant quotient bits expected.

The sequence begins with a Divide-Initialize instruction (DVINIT(.U), DVINIT.H(U), or DVINIT.B(U), depending on the size of the quotient and on whether the operands are to be treated as signed or unsigned). The divide initialization instruction extends the 32-bit dividend to 64 bits, then shifts it left by 0, 16, or 24 bits. Simultaneously it shifts in that many copies of the quotient sign bit to the low-order bit positions. Then follows four, two, or one Divide-Step instructions (DVSTEP or DVSTEP.U). Each divide step instruction develops eight bits of quotient.

At the end of the divide step sequence, the 32-bit quotient occupies the low-order word of the 64-bit dividend register pair, and the remainder is held in the high-order word. If the divide operation was signed, the Divide-Adjust instruction (DVADJ) is required to perform a final adjustment of negative values. If the dividend and the divisor are both known to be positive, the DVADJ instruction can be omitted.

8.1.1.5 Absolute Value, Absolute Difference

A common operation on data is the computation of the absolute value of a signed number or the absolute value of the difference between two signed numbers. These operations are provided directly by the ABS and ABSDIF instructions, and there is a version of each instruction which saturates when the result is too large to be represented as a signed number.

8.1.1.6 Min, Max, Saturate

Instructions are provided that directly calculate the minimum or maximum of two operands. The MIN and MAX instructions are used for signed integers, and MIN.U and MAX.U are used for unsigned integers.

The SAT instructions can be used to saturate the result of a 32-bit calculation before storing it in a byte or halfword in memory or a register.

8.1.1.7 Conditional Arithmetic Instructions

The conditional instructions—Conditional Add (CADD), Conditional Subtract (CSUB), and Select (SEL)—provide efficient alternatives to conditional jumps around very short sequences of code. All of the conditional instructions use a condition operand that controls the execution of the instruction. The condition operand is a data register, with any non-zero value interpreted as TRUE, and a zero value interpreted as FALSE. For the CADD and CSUB instructions, the addition/subtraction is performed if the condition value matches the value specified in the instruction mnemonic: CADD and CSUB if the condition is TRUE, and CADDN and CSUBN if the condition is FALSE. The instructions CADD.A, CSUB.A, CADDN.A, and CSUBN.A are the corresponding instructions that apply to address registers (refer to Section 8.5, "Address Comparison," on page 102).

The SEL instruction copies one of its two source operands to its destination operand, with the selection of source operands determined by the value of the condition operand. (This operation is the same as the C language "?" operation.) A typical use might be to record the index value yielding the larger of two array elements:

```
index_max = (a[i] > a[j]) ? i : j;
```

If one of the two source operands in a Select instruction is the same as the destination operand, then the Select instruction implements a simple conditional move. This occurs fairly often, in source statements of the general form:

```
if (<condition>) then <variable> = <expression>;
```

Provided that *<expression>* is simple, it is more efficient to evaluate it unconditionally into a source register, using a SEL instruction to perform the conditional assignment, rather than conditionally jumping around the assignment statement.

8.1.1.8 Logical

The TriCore architecture provides a complete set of two-operand, bit-wise logic operations. In addition to the AND, OR, and XOR functions, there are the negations of the output — NAND, NOR, and XNOR — and negations of one of the inputs — ANDN and ORN (the negation of an input for XOR is the same as XNOR).

8.1.1.9 Count Leading Zeroes, Ones, and Signs

To provide efficient support for normalization of numerical results, prioritization, and certain graphics operations, three Count Leading instructions are provided: CLZ (Count Leading Zeros), CLO (Count Leading Ones), and CLS (Count Leading Signs). These instructions are used to determine the amount of left shifting necessary to remove redundant zeros, ones, or signs. Note that the CLS instruction returns the number of redundant signs, which is the number of leading signs minus one. Further, the following special cases are defined: CLZ(0) = 32, CLO(-1) = 32, and CLS(0) = CLS(-1) = 31.

For example, CLZ returns the number of consecutive zeros starting from the most-significant bit of the value in the source data register. In the example shown in Figure 47, there are 7 zeros in the most-significant portion of the input register. If the most-significant bit of the input is a one, CLZ returns 0.

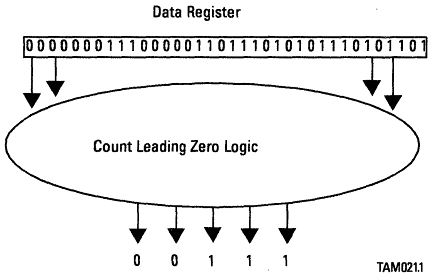


Figure 47: Operation of CLZ Instruction

The Count Leading instructions are useful for parsing certain Huffman codes and bit strings consisting of boolean flags, since the code or bit string can be quickly classified by determining the position of the first one (scanning from left to right).

8.1.1.10 Shift

The shift instructions support multi-bit shifts. The shift amount is specified by a signed integer (n), which may be the contents of a register or a sign-extended constant in the instruction. If n >= 0, the data is shifted left by n[4:0]; otherwise, the data is shifted right by (-n)[4:0]. The (logical) shift instruction, SH, shifts in zeroes for both right and left shifts; the arithmetic shift instruction, SHA, shifts in sign bits for right shifts and zeroes for left shifts. The arithmetic shift with saturation instruction, SHAS, will saturate (on a left shift) if the sign bits that are shifted out are not identical to the sign bit of the result.

8.1.1.11 Bit-Field Extract and Insert

The TriCore architecture supports three bit-field extract instructions. The EXTR.U and EXTR instructions extract w (width) consecutive bits from the source, beginning with the bit number specified by the pos (position) operand. The width and position can be specified by two immediate values, by a data register and an immediate value, or by a data register pair. The EXTR.U instruction, shown in Figure 48, zero-fills the most-significant (32-w) bits of the result.

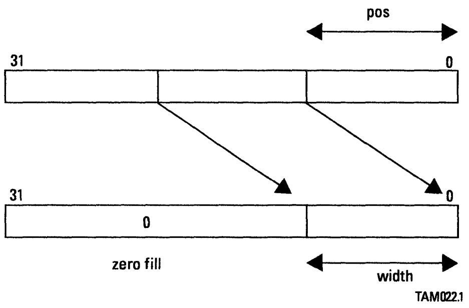


Figure 48: Operation of EXTR.U Instruction

The EXTR instruction (refer to Figure 49), fills the most-significant bits of the result by sign-extending the bit field extracted (i.e. duplicating the most-significant bit of the bit field).

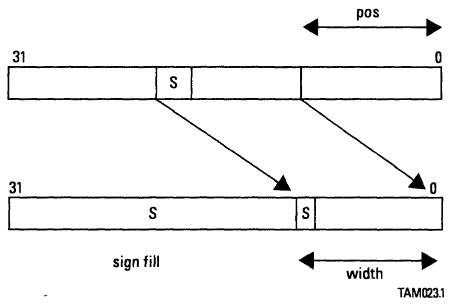


Figure 49: Operation of EXTR Instruction

The DEXTR instruction (refer to Figure 50), concatenates two data register sources to form a 64-bit value from which 32 consecutive bits are extracted. The operation can be thought of as a left shift by pos bits, followed by the truncation of the least-significant 32 bits of the result. The value of pos is contained in a data register or is an immediate value in the instruction.

The DEXTR instruction can be used to normalize the result of a DSP filter accumulation in which a 64-bit accumulator is used with several guard bits. The value of pos can be determined by using the CLS (Count Leading Signs) instruction. The DEXTR instruction can also be used to perform a multi-bit rotation by using the same source register for both of the sources (that are concatenated).

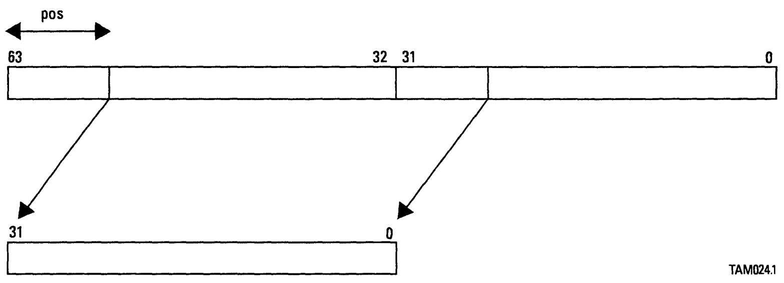


Figure 50: Operation of DEXTR Instruction

The INSERT instruction (shown in Figure 51) takes the w least-significant bits of a source data register and substitutes them into the value of another source register, shifted left by pos bits. All other (32-w) bits of the destination register are unchanged. The values of width and pos are specified in the same way as for EXTR(.U). There is also an alternative form of INSERT that allows a zero-extended 4-bit constant to be the value which is inserted.

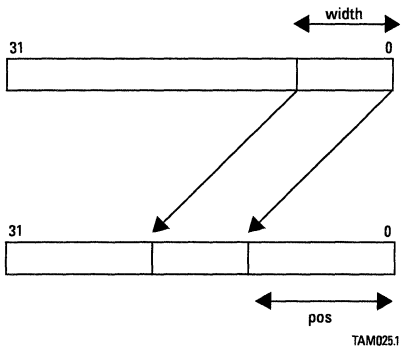


Figure 51: Operation of INSERT Instruction

8.1.2 DSP Arithmetic

DSP arithmetic instructions operate on 16-bit, signed fractional data in the 1.15 format (also known as Q15) and 32-bit signed fractional data in 1.31 format (also known as Q31). Data values in this format have a single, high-order sign bit, with a value of 0 or -1, followed by an implied binary point and fraction. Their values are in the range [-1, 1).

16-bit DSP data is loaded into the most-significant half of a data register, with the 16 least-significant bits set to zero. The left alignment of 16-bit data allows it to be directly added to 32-bit data in 1.31 format. All other fractional formats can be synthesized by explicitly shifting data as required.

Operations created for this format are multiplication, multiply-add, and multiply-subtract. The signed fractional formats 1.15 and 1.31 are supported with the MUL.Q and MULR.Q instructions. These instructions operate on two, left-justified, signed fractions and return a 32-bit signed fraction.

8.1.2.1 Scaling

The multiplier result can be shifted in two ways:

- left shifted by 1: one sign bit is suppressed and the result is left-aligned, thus conserving the input format.
- not shifted: the result retains its 2 sign bits (2.30 format). This format can be used with IIR filters, in which some of the coefficients are between 1 and 2, and to have 1 guard bit for accumulation.

8.1.2.2 Special case = -1 * -1 => +1

When multiplying the two, maximum negative values (-1), the result should be the maximum positive number (+1). For example,

```
0x8000 * 0x8000 = 0x4000 0000
```

is correctly interpreted in Q format as:

```
-1(1.15 format) * -1(1.15 format) = +1 (2.30 format)
```

However, when the result is shifted left by 1, the result is 0x8000 0000, which is incorrectly interpreted as:

```
-1(1.15 format) * -1(1.15 format) = -1 (1.31 format)
```

To avoid this problem, the result of a Q format operation (-1 * -1) that has been left-shifted by 1 (left-justified), is saturated to the maximum positive value. Thus,

```
0x8000 * 0x8000 = 0x7FFF FFFF
```

is correctly interpreted in Q format as:

```
-1(1.15 format) * -1(1.15 format) = (nearest representation of)+1 (1.31
format)
```

This operation is completely transparent to the user and does not set the overflow flags.

8.1.2.3 Guard bits

When accumulating sums (for example, in filter calculations) guard bits are often required to prevent overflow. The instruction set directly supports the use of 1 guard bit when using a 32-bit accumulator; when more guard bits are required, a register pair (64 bits) can be used.

8.1.2.4 Rounding

Rounding is used to retain the 16-bit most-significant bits of a 32-bit result. Rounding is combined with the MUL, MADD, MSUB instructions, and is implemented by adding 1 to bit 15 of a 32-bit register.

8.1.2.5 Overflow and Saturation

Saturation on signed and unsigned overflow is implemented as part of the MUL, MADD, MSUB instructions.

8.1.2.6 Sticky Advanced Overflow and Block Scaling in FFT

The Sticky Advanced Overflow (SAV) bit, which is set whenever an overflow "almost" occurred, can be used in block scaling of intermediate results during an FFT calculation. Before each pass of applying a butterfly operation, the SAV bit is cleared, and after the pass the SAV bit is tested. If it is set, then all of the data is scaled (using an arithmetic right shift) before starting the next pass. This procedure gives the greatest dynamic range for intermediate results without the risk of overflow.

8.1.3 Packed Arithmetic

The packed arithmetic instructions partition a 32-bit word into several identical objects, which can then be fetched, stored, and operated on in parallel. These instructions, in particular, allow the full exploitation of the 32-bit word of the TriCore architecture in signal and data processing applications.

The TriCore architecture supports two packed formats. The first format divides the 32-bit word into two, 16-bit (halfword) values. Instructions which operate on data in this way are denoted in the instruction mnemonic by the ".H" and ".HU" data type modifiers. Refer to Figure 52.

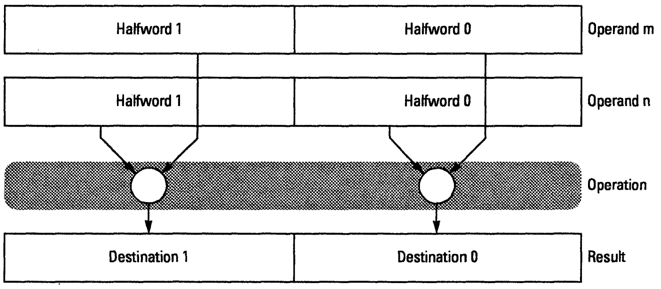


Figure 52: Packed Halfword Data Format

The second packed format divides the 32-bit word into four, 8-bit values. Instructions which operate on the data in this way are denoted by the ".B" and ".BU" data type modifiers. Refer to Figure 53.

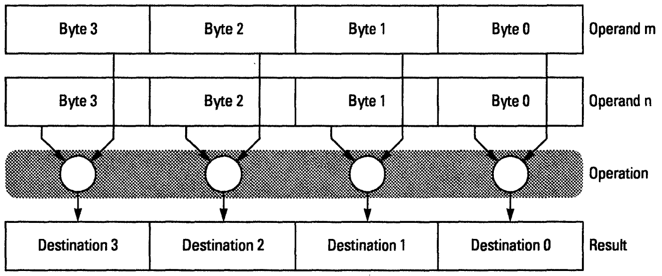


Figure 53: Packed Byte Data Format

The loading and storing of packed values into data registers is supported by the normal Load Word and Store Word instructions (LD.W and ST.W). The packed objects can then be manipulated in parallel by a set of special packed arithmetic instructions that perform such arithmetic operations as addition, subtraction, multiplication, etc.

Addition is performed on individual packed bytes or halfwords using the ADD.B and ADD.H instructions and their saturating variations ADDS.B and ADDS.BU. ADD.B ignores overflow/underflow within individual bytes, while the ADDS.B will saturate individual bytes to the most positive, 8-bit signed integer (127) on individual overflow, or to the most negative, 8-bit signed integer (-128) on individual underflow. Similarly, the ADD.H instruction ignores overflow/underflow within individual halfwords, while the ADDS.H will saturate individual halfwords to the most positive 16-bit signed integer (2^{15}-1) on individual overflow, or to the most negative 16-bit signed integer (-2^{15}) on individual underflow. Saturation for unsigned integers is also supported by the ADDS.BU and ADDS.HU instructions.

Besides addition, arithmetic on packed data includes subtraction, multiplication, absolute value, subtract absolute, and shift operations.

8.2 Compare Instructions

The compare (and conditional jump) instructions use a compare operation on the contents of two registers. The boolean result (1 = true and 0 = false) is stored in the least-significant bit of a data register, and the remaining bits in the register are cleared to zero. Figure 54 illustrates the operation of the LT (Less Than) compare instruction.

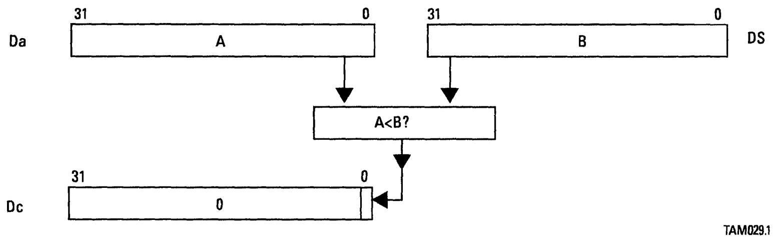


Figure 54: LT Comparison

The comparison instructions are: equal (EQ), not equal (NE), less than (LT), and greater than or equal to (GE), with versions for both signed and unsigned integers.

Comparison conditions not explicitly provided in the instruction set can be obtained by either swapping the operands when comparing two registers, or by incrementing the constant by one when comparing a register and a constant. Refer to the table below.

"Missing" Comparison Operation	TriCore Equivalent Comparison Operation
LE Dc, Da, Db	GE Dc, Db, Da
LE Dc, Da, const	LT Dc, Da, (const+1)
GT Dc, Da, Db	LT Dc, Db, Da
GT Dc, Da, const	GE Dc, Da, (const+1)

To accelerate the computation of complex conditional expressions, accumulating versions of the comparison instructions are supported. These instructions, indicated in the instruction mnemonic by "op" preceding the " . " (for example, op.LT), combine the result of the comparison with a previous comparison result. The combination is a logic AND, OR, or XOR; for example, AND.LT, OR.LT, and XOR.LT. Figure 55 illustrates combining the LT instruction with a boolean operation.

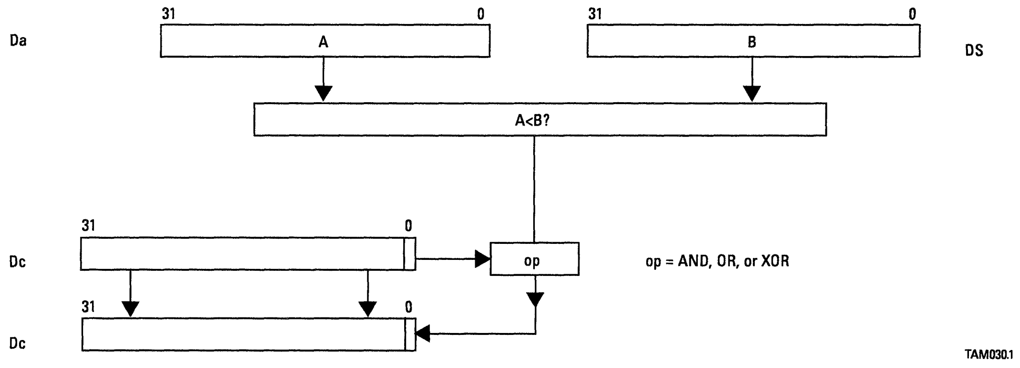


Figure 55: Combining LT Comparison with Boolean Operation

The evaluation of the following C expression can be optimized using the combined compare-boolean operation:

```
d5 = (d1 < d2) || (d3 == d4);
```

Assuming all variables are in registers, the following two instructions will compute the value in d5:

```
lt      d5,d1,d2    ; compute (d1 < d2)
or.eq   d5,d3,d4  ; or with (d3 == d4)
```

Certain control applications require that several booleans be packed into a single register. These packed bits can be used as an index into a table of constants or a jump table, which permits complex boolean functions and/or state machines to be evaluated efficiently. To facilitate the packing of boolean results into a register, compound Compare with Shift instructions (for example, SH.EQ) are supported. The result of the comparison is placed in the least-significant bit of the result after the contents of the destination register have been shifted left by one position. Figure 56 illustrates the operation of the SH.LT (Shift Less Than) instruction.

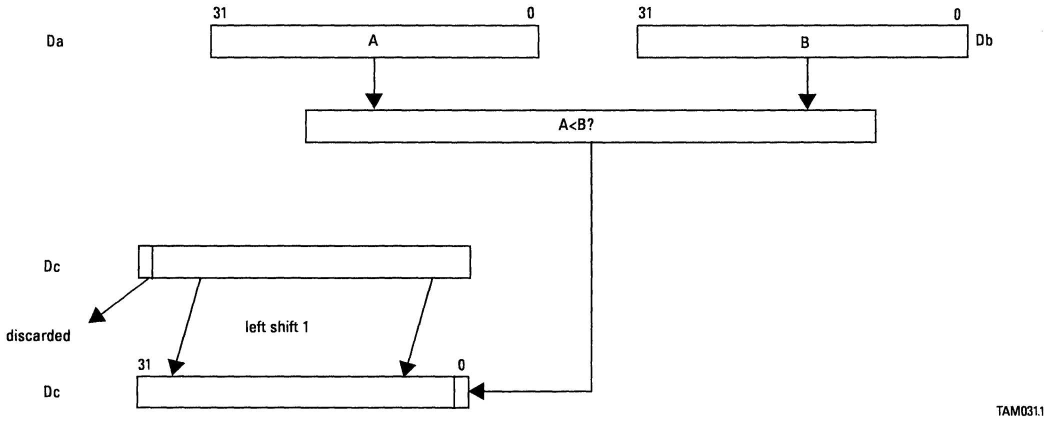


Figure 56: SH.LT Instruction

For packed bytes, there are special compare instructions that perform four individual byte comparisons and produce a 32-bit mask consisting of four "extended" booleans. For example, EQ.B yields a result where individual bytes are 0xFF for a match or 0x00 for no match. Similarly, for packed halfwords there are special compare instructions that perform two individual halfword comparisons and produce two extended booleans. The EQ.H instruction results in two extended booleans: 0xFFFF for a match and 0x0000 for no match. There are even abnormal packed-word compare instructions that compare two words in the normal way but produce a single extended boolean. The EQ.W instruction results in the extended boolean 0xFFFF.FFFF for match and 0x0000.0000 for no match.

Extended booleans are useful as masks, which can be used by subsequent bit-wise logic operations. Also, CLZ (count leading zeros) or CLO (count leading ones) can be used on the result to quickly find the position of the left-most match. Figure 57 shows an example of the EQ.B instruction:

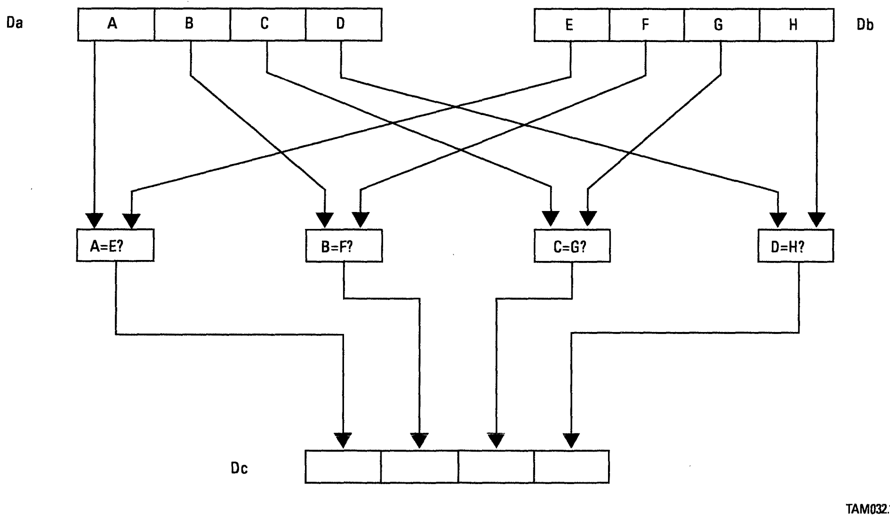


Figure 57: EQ.B Instruction Operation

8.3 Bit Operations

Instructions are provided that operate on single bits, denoted in the instruction mnemonic by the "T" data type modifier (for example, AND.T).

There are eight instructions for combinatorial logic functions with two inputs, and 12 instructions with three inputs.

The one-bit result of a two-input function (for example, AND.T) is stored in the least-significant bit of the destination data register, and the most-significant 31 bits are set to zero. The source bits can be any bit of any data register. This is illustrated in Figure 58. The available Boolean operations are: AND, NAND, OR, NOR, XOR, XNOR, ANDN, and ORN.

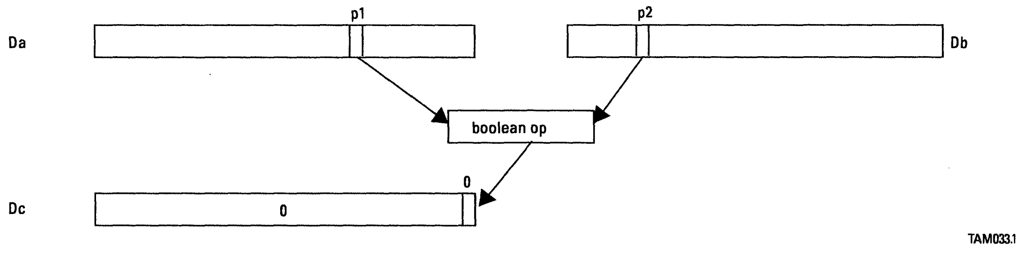


Figure 58: Boolean Operations

Evaluation of complex boolean equations can use the 3-input Boolean operations, in which the output of a two-input instruction, together with the least-significant bit of a third data register, forms the input to a further operation. The result is written to bit 0 of the third data register, with the remaining bits unchanged. Refer to Figure 59.

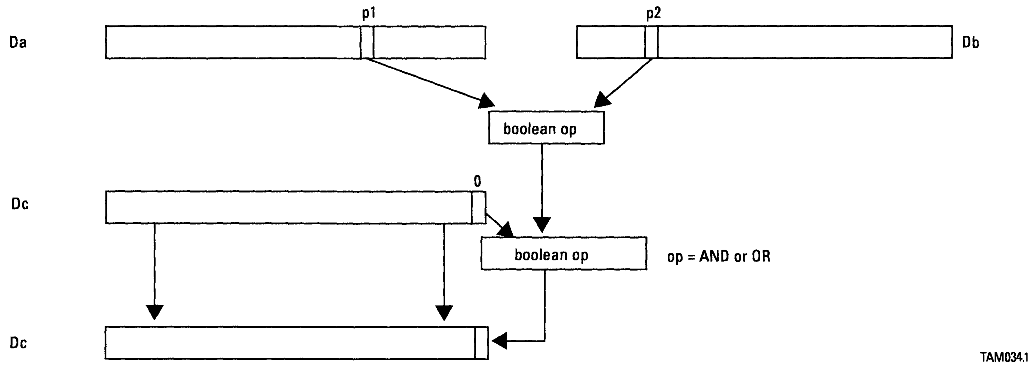


Figure 59: 3-Input Boolean Operation

Of the many possible 3-input operations eight have been singled out for the efficient evaluation of logical expressions.

The instructions provided are: AND.AND.T, AND.ANDN.T, AND.NOR.T, AND.OR.T, OR.AND.T, OR.ANDN.T, OR.NOR.T, and OR.OR.T.

Just as for the comparison instructions, the results of bit operations often need to be packed into a single register for controller applications. For this reason, the basic two-input instructions can be combined with a shift prefix (for example, SH.AND.T). These operations first perform a single-bit left shift on the destination register and then store the result of the two-input logic function into its least-significant bit, as illustrated in Figure 60.

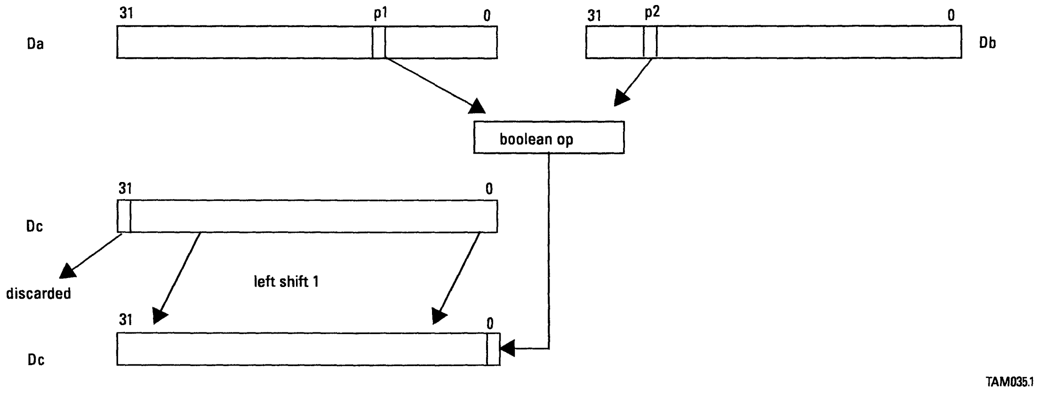


Figure 60: Shift Plus Boolean Operation

8.4 Address Arithmetic

The TriCore architecture provides selected arithmetic operations on the address registers. These operations supplement the address calculations inherent in the addressing modes used by the load and store instructions.

Initialization of base pointers requires loading a constant into an address register. When the base pointer is in the first 16 Kbytes of each segment, this can be done using the Load Effective Address (LEA) instruction, using the absolute addressing mode. Loading a 32-bit constant into an address register can be accomplished using MOVH.A followed by an LEA that uses the base plus 16-bit offset addressing mode. For example,

```
movh.a      a5, ((ADDRESS+0x8000)>>16) & 0xffff
lea         a5, [a5](ADDRESS & 0xffff)
```

The MOVH.A instruction loads a 16-bit immediate into the most-significant 16-bits of an address register and zero-fills the least-significant 16-bits.

Adding a 16-bit constant to an address register can be done using the LEA instruction with the base plus offset addressing mode. Adding a 32-bit constant to an address register can be done in two instructions: an Add Immediate High Word (ADDIH.A), which adds a 16-bit immediate to the most-significant 16 bits of an address register, followed by an LEA using the base plus offset addressing mode. For example,

```
addih.a     a8, ((OFFSET+0x8000)>>16) & 0xffff
lea         a8, [a8](OFFSET & 0xffff)
```

The Add Scaled (ADDSC.A) instruction directly supports the use of a data variable as an index into an array of bytes, halfwords, words, or doublewords.

A common C language operation is to subtract one address pointer from another. The result is the number of data elements between the two pointers (the two pointers must reference the same data type). The Difference Scaled Address (DIFSC.A) instruction supports this operation directly for byte, halfword, word, and doubleword data types.

The basic operations on address registers are completed by instructions that provide addition and subtraction of two address registers (ADD.A and SUB.A) and data movement to and from a data register (MOV.D and MOV.A) and between address registers (MOV.AA).

The instructions SEL.A, SELN.A, CADD.A, CSUB.A, CADDN.A, and CSUBN.A are the conditional instructions that apply to the address registers. Refer to "Conditional Arithmetic Instructions" on page 90.

8.5 Address Comparison

As with the comparison instructions that use the data registers (refer to Section 8.2, "Compare Instructions," on page 97), the comparison instructions using the address registers put the result of the comparison in the least-significant bit of the destination data register and clear the remaining register bits to zeros. An example using the Less Than (LT.A) instruction is shown in Figure 61.

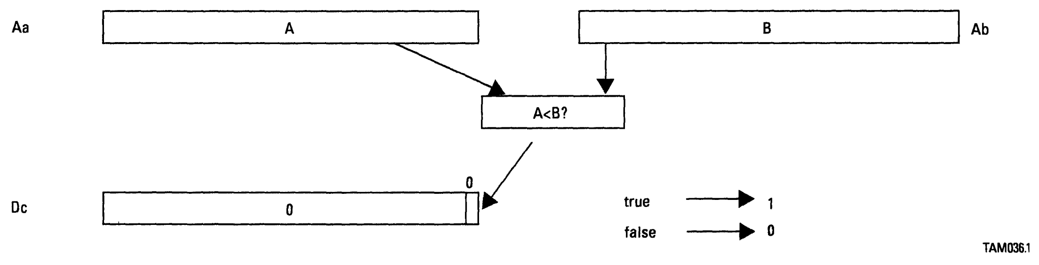


Figure 61: LT.A Comparison Operation

There are comparison instructions for equal (EQ.A), not equal (NE.A), less than (LT.A), and greater than or equal to (GE.A). As with the comparison instructions using the data registers, comparison conditions not explicitly provided in the instruction set can be obtained by swapping the two operand registers:

"Missing" Comparison Operation	TriCore Equivalent Comparison Operation
LE.A Dc, Aa, Ab	GE.A Dc, Ab, Aa
GT.A Dc, Aa, Ab	LT.A Dc, Ab, Aa

In addition to these instructions, instructions that test whether an address register is equal to zero (EQZ.A), or not equal to zero (NEZ.A) are supported. These instructions are useful to test for null pointers, which is a frequent operation when dealing with linked lists and complex data structures.

8.6 Branch Instructions

Branch instructions change the flow of program control by modifying the value in the PC register. There are two types of branch instructions: conditional and unconditional. Whether or not a conditional branch is taken depends on the result of a Boolean compare operation, as described in Section 8.2, "Compare Instructions," on page 97 , rather than on the state of condition codes.

8.6.1 Unconditional Branch

There are three groups of unconditional branch instructions: Jump instructions, Jump and Link instructions, and Call and Return instructions.

A Jump instruction simply loads the Program Counter with the address specified in the instruction. A Jump and Link instruction does the same, and also stores the address of the next instruction in the "return address register" A11/RA.A Jump and Link instruction can be used to implement a subroutine call when the called routine does not modify any of the caller's non-volatile registers. The Call instructions differ from a Jump and Link in that they save the caller's non-volatile registers in a dynamically-allocated save area (refer to Section 4.3, "CSAs and Context Lists," on page 49). The Return instruction, in addition to performing the return jump, restores the non-volatile registers.

Each group of unconditional jump instructions contains separate instructions that differ in how the target address is specified. There are instructions using a relative 24-bit signed offset (J, JL, and CALL), instructions using 24 bits of displacement as an absolute address (JA, JLA, and CALLA), and instructions using the address contained in an address register (JI, JLI, CALLI, RET, and RFE).

There are additional 16-bit instructions for a relative jump using an 8-bit offset (J), an instruction for an indirect jump (JI), and an instruction for a return (RET).

Both the 24-bit and 8-bit relative offsets and displacements are scaled by two before they are used, because all instructions must be aligned on an even address. The use of a 24-bit displacement is shown in Figure 62.

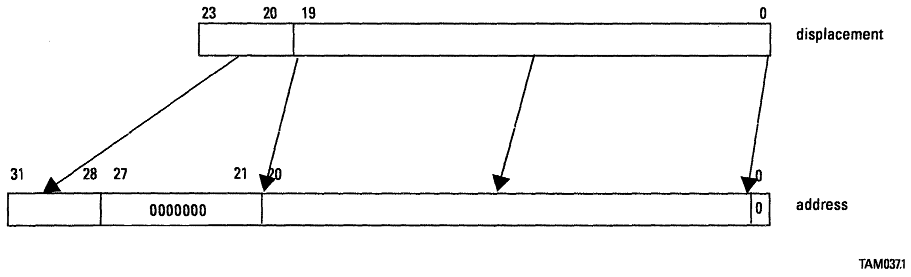


Figure 62: Jump Target Address with Displacement

8.6.2 Conditional Branch

The conditional branch instructions use the absolute addressing mode, with an offset value encoded in 4, 8, or 15 bits. The offset is scaled by 2 before it is used, because all instructions must be aligned on an even (halfword) address. The scaled offset is sign-extended to 32 bits before it is added to the program counter, unless otherwise noted.

The Boolean test uses the contents of data registers, address registers, or individual bits in data registers.

8.6.2.1 Conditional Jumps on Data Registers

Six of the conditional jump instructions use a 15-bit signed offset field: comparison for equality (JEQ), non-equality (JNE), less than (JLT), less than unsigned (JLT.U), greater than or equal (JGE), and greater than or equal unsigned (JGE.U). Testing for less than zero and greater than or equal to zero can be done with a conditional jump on the sign bit of the value in a data register, as described in Section 8.6.2.3, "Conditional Jumps on Bits," on page 105.

There are two 16-bit instructions that test whether the implicit D15 register is equal to zero (JZ) or not equal to zero (JNZ). The offset is 8-bit in this case. These instructions are typically used in combination with the 16-bit compare instructions that use D15 as the implicit destination register.

Another two 16-bit instructions compare the implicit D15 register with a 4-bit, sign-extended constant (JEQ, JNE). The jump displacement field is limited to 4 bits in this case.

There is a full set of 16-bit instructions that compare a data register to zero: JZ, JNZ, JLTZ, JLEZ, JGTZ, and JGEZ. Because any data register may be specified, the jump displacement is limited to 4 bits.

8.6.2.2 Conditional Jumps on Address Registers

The conditional jump instructions that use address registers are a subset of the data register conditional jump instructions. Four conditional jump instructions use a 15-bit signed offset field: comparison for equality (JEQ.A), non equality (JNE.A), equal to zero (JZ.A), and non-equal to zero (JNZ.A).

Because testing pointers for equality to zero is so frequent, two 16-bit instructions, JZ.A and JNZ.A, are provided, with a displacement field limited to 4 bits.

8.6.2.3 Conditional Jumps on Bits

Conditional jumps can be performed based on the value of any bit in any data register. The JZ.T instruction jumps when the bit is clear, and the JNZ.T instruction jumps when the bit is set. For these instructions, the jump displacement field is 15 bits.

8.6.2.4 Loop Instructions

Four special versions of conditional jump instructions are intended for efficient implementation of loops. The JNEI and JNED instructions are like a normal JNE instruction, but with an additional increment or decrement operation of the first register operand. The increment or decrement operation is performed after the comparison. The jump offset field is 15 bits. For example, a loop that should be executed for D3 = 3, ..., 10 can be implemented as follows:

```
    mov     d3,3
loop1:
    ...
    jnei    d3,10,loop1
```

The LOOP instruction is a special kind of jump which utilizes the special TriCore hardware that implements "zero overhead" loops. The LOOP instruction only requires execution time in the pipeline the first and last time it is executed (for a given loop); for all other iterations of the loop, the LOOP instruction has zero execution time. For example, a loop that should be executed 100 times may be implemented as:

```
    mova    a2,99
loop2:
    ...
    loop    a2,loop2
```

The LOOP instruction above requires execution cycles the first and 100th time it is executed, but the other 98 executions require no cycles.

Note that the LOOP instruction differs from the other conditional jump instructions in that it uses an address register, rather than a data register, for the iteration count. This allows it to be used in filter calculations in which a large number of data register reads and writes occur each cycle. Using an address register for the LOOP instruction reduces the need for an extra data register read port.

The LOOP instruction has a 32-bit version using a 15-bit displacement field (left-shifted by one bit and sign-extended), and a 16-bit version that uses a 4-bit displacement field. Unlike all other relative jumps, the 4-bit value is one-extended rather than sign-extended, because this instruction is specifically intended for loops.

8.7 Load and Store Instructions

The load and store instructions move data between registers and memory, using the seven addressing modes shown in Table 9. (Addressing modes are described in detail in Section 2.4, "Addressing Model," on page 19 .) The addressing mode determines the effective byte address for the load or store instruction and any update of the base pointer address register.

Table 9: Addressing Modes

Addressing Mode	Syntax	Effective Address	Instruction Format
Absolute	constant	zero_ext(offset18)	ABS
Base + Short Offset	[An]offset	A[a]+sign_ext(offset10)	BOL
Base + Long Offset	[An]offset	A[a]+sign_ext(offset16)	BOL
Pre-increment	[+An]offset	A[a]+sign_ext(offset10)	BO
Post-increment	[An+]offset	A[a]	BO
Circular	[An+c]offset	Refer to page 21	BO
Bit-reverse	[An+r]	Refer to page 22	BO

8.7.1 Load/Store Basic Data Types

The TriCore architecture defines loads and stores for the basic data types — bytes, halfwords, words and doublewords — as well as for signed fractions and packed data. The movement of data between registers and memory for the basic data types is illustrated in Figure 63. Note that when the data loaded from memory is smaller than the destination register (i.e. 8- and 16-bit quantities), the data is loaded into the least-significant bits of the register, and the remaining register bits are sign- or zero-extended to 32 bits, depending on the particular instruction.

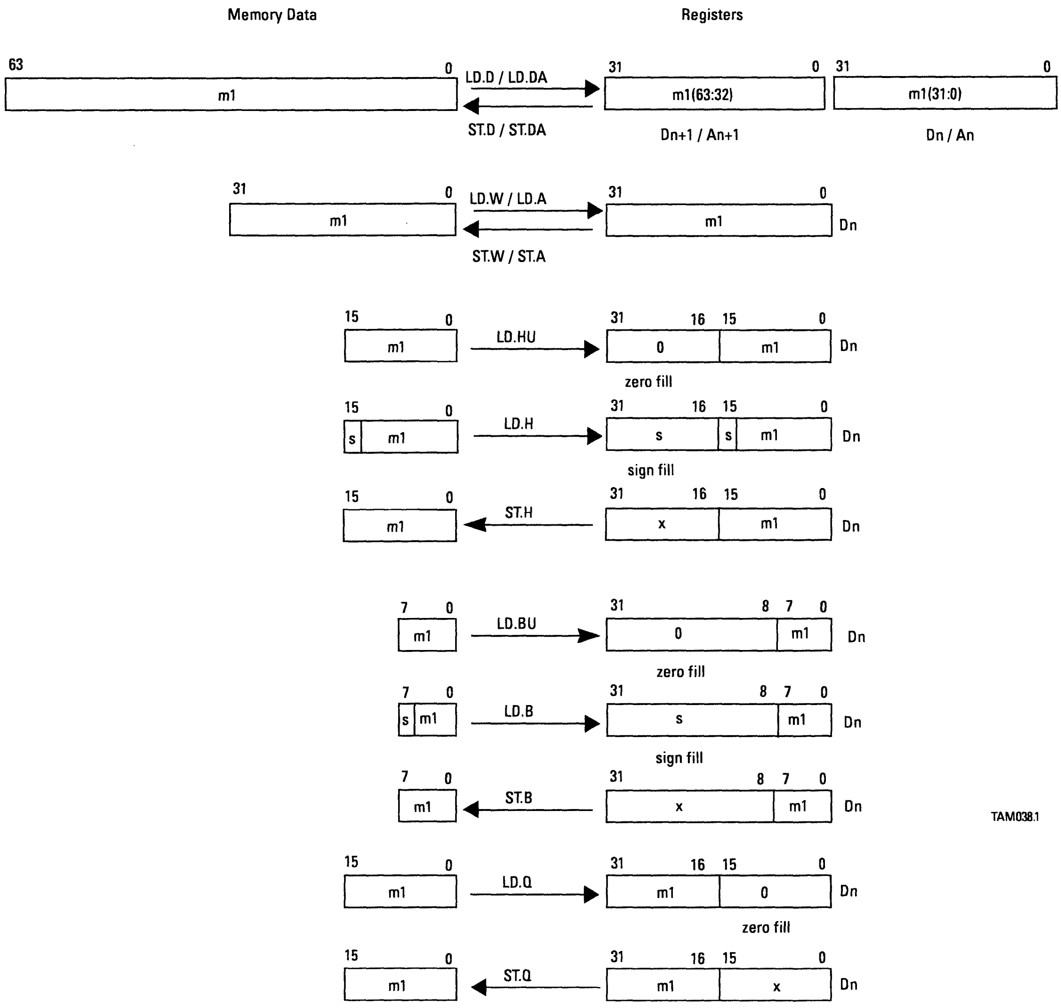


Figure 63: Load/Store Basic Data Types

8.7.2 Load Bit

The approaches for loading individual bits depend on whether the bit within the word (or byte) is given statically or dynamically.

Loading a single bit with a fixed bit offset from a byte pointer is accomplished with an ordinary load instruction. One then can extract, logically operate on, or jump on any bit in a register.

Loading a single bit with a variable bit offset from a word-aligned byte pointer is done with a special scaled offset instruction. This offset instruction shifts the bit offset to the right by three positions (producing a byte offset), adds this result to the byte pointer above, and finally zeroes out the two lower bits, thus aligning the access on a word boundary. A word load can then access the word that contains the bit, which can be extracted with an extract instruction that only uses the lower five bits of the bit pointer, that is, the bits that were either shifted out or masked out above. An example is:

```
ADDSC.AT    A8,A9,D8        ; A9 = byte pointer. D8 = bit offset.
LD.W        D9,[A8]
EXTRACT.U   D10,D9,D8,1     ; D10[0] = loaded bit.
```

8.7.3 Store Bit and Bit Field

The ST.T instruction can clear or set single memory or peripheral bits, resulting in reduced code size. ST.T statically specifies a byte address and a bit number within that byte, and indicates whether the bit should be set or cleared. The addressable range for this instruction is the first 16 KBytes of each of the 16 memory segments.

The Insert Mask (IMASK) instruction can be used in conjunction with the Load-Modify-Store (LDMDST instruction) to store a single bit or a bit field to a location in memory, using any of the addressing modes. This operation is especially useful for reading and writing memory-mapped peripherals. The IMASK instruction is very similar to the INSERT instruction, but IMASK generates a data register pair that contains a mask and a value. The LDMDST instruction uses the mask to indicate which portion of the word to modify. An example of a typical instruction sequence is:

```
imask       E8,3,4,2        ; insert value = 3, position = 4, width = 2
ldmdst      _IOREG,E8       ; at absolute address "_IOREG"
```

To clarify the operation of the IMASK instruction, consider the following example. The binary value 1011_2 is to be inserted starting at bit position 7 (the width is four). The IMASK instruction would result in the following two values:

```
0000 0000 0000 0000 0000 0111 1000 0000        MASK
0000 0000 0000 0000 0000 0101 1000 0000        VALUE
```

To store a single bit with a variable bit offset from a word-aligned byte pointer, first the word address is determined in the same way as for the load above. Again the special scaled offset instruction shifts the bit offset to the right by three positions, which produces a byte offset, then adds this offset to the byte pointer above, and finally zeroes out the two lower bits, thus aligning the access on a word boundary. An IMASK and LDMDST instruction can store the bit into the proper position in the word. An example is:

```
ADDSC.AT    A8,A9,D8        ; A9 = byte pointer. D8 = bit offset.
IMASK       E10,D9,D8,1     ; D9[0] = data bit.
LDMDST      [A8],E10
```

8.8 Context Related Instructions

Besides the instructions that implicitly save and restore contexts (such as Calls and Returns), the TriCore instruction set includes instructions that allow a task's contexts to be explicitly saved, restored, loaded, and stored. These instructions are detailed in the following sections.

Refer also to Section 4.2, "Task Switching Operation," on page 48 .

8.8.1 Context Saving and Restoring

The upper context of a task is always automatically saved on a call, interrupt, or trap, and is automatically restored on a return. However, the lower context of a task must be saved/restored explicitly.

The SVLCX instruction (Save Lower Context) saves registers A2 through A7 and D0 through D15 together with the return address in register A11/RA and the PCXI. This operation is performed when using the FCX and PCX pointers to manage the CSA lists.

The RSLCX instruction (Restore Lower Context) restores the lower context. It loads registers A2 through A7 and D0 through D7 from the CSA. It also loads A11/RA from the saved PC field. This operation is performed when using the FCX and PCX pointers to manage the CSA lists.

The BISR instruction (Begin Interrupt Service Routine) enables the interrupt system (ICR.IE is set to one), allows the modification of the CPU priority number (CCPN), and saves the lower context in the same manner as the SVLCX instruction.

8.8.2 Context Loading and Storing

The effective address of the memory area where the context is stored to or loaded from is part of the Load or Store instruction. The effective address must resolve to a memory location aligned on a 16-word boundary, otherwise a data address alignment trap (ALN) is generated.

The STUCX instruction (Store Upper Context) stores the same context information that is saved with an implicit upper context save operation: Registers A10 - A15 and D8 - D15, and the current PSW and PCXI.

The LDUCX instruction (Load Upper Context) loads registers A10 - A15 and D8 - D15. The PSW and link word fields in the saved context in memory are ignored. The PSW, FCX, and PCXI are unaffected.

The STLCX instruction (Store Lower Context) stores the same context information that is saved with an explicit lower context save operation: Registers A10 - A15 and D8 - D15, and the current PSW and PCXI.

The LDLCX instruction (Load Lower Context) loads registers A2 through A7 and D0 through D7. The saved return address and the link word fields in the context stored in memory are ignored. Registers A11/RA, FCX, and PCXI are not affected.

8.9 System Instructions

The system instructions allow user-mode and supervisor-mode programs to access and control various system services, including interrupts, and the TriCore's debugging facilities. There are also instructions that read and write the core registers, for both user and supervisor-only mode programs.

8.9.1 System Call

The SYSCALL instruction generates a system call trap, providing a secure mechanism for user-mode application code to request supervisor services. The system call trap, like other traps, vectors to the trap handler table, using the three-bit hardware-furnished trap class ID as an index. The trap class ID for system call traps is six. The trap identification number (TIN) is specified by an immediate constant in the SYSCALL instruction, and serves to identify the specific supervisor service that is being requested. Refer to Chapter 6, "Traps," for more information.

8.9.2 Synchronization Primitives

The TriCore architecture provides two synchronization primitives. These primitives provide a mechanism to software through which it can guarantee the ordering of various events within the machine.

8.9.2.1 DSYNC

The first primitive, DSYNC, provides a mechanism through which a data memory barrier can be implemented. The DSYNC instruction guarantees that all data accesses associated with instructions semantically prior to the DSYNC instruction are completed before any data memory accesses associated with an instruction semantically after DSYNC are initiated. This includes all accesses to the system bus and local data memory.

8.9.2.2 ISYNC

The second primitive, ISYNC, provides a mechanism through which the following can be guaranteed:

- If an instruction semantically prior to ISYNC changes a piece of architectural state, then the effects of this change are seen by all instructions semantically after ISYNC. For example, if an instruction changes a code range in the protection table, the use of an ISYNC will guarantee that all instructions after the ISYNC are fetched and matched against the new protection table entry.
- All cached states in the pipeline, such as branch target buffers, are invalidated.

The operation of the ISYNC instruction, therefore, is described as follows:

1. Wait until all instructions semantically prior to the ISYNC have completed.
2. Flush the CPU pipeline and cancel all instructions semantically after the ISYNC.
3. Invalidate all cached states in the pipeline.
4. Refetch the next instruction after the ISYNC.

8.9.3 Access to the Core Special Function Registers

The TriCore accesses the CSFRs through two instructions: MFCR and MTCR. The MFCR instruction (Move From Core Register) moves the contents of the addressed CSFR into a data register. MFCR can be executed at any privilege level. The MTCR instruction (Move To Core Register) moves the contents of a data register to the addressed CSFR. To prevent unauthorized writes to the CSFRs, the MTCR instruction can only be executed at the supervisor privilege level.

The CSFRs are also mapped into the top of the local code segment in the memory address space. This mapping makes the complete architectural state of the core visible in the address map, which allows efficient debug and emulator support. Note it is not permitted for the core to access the CSFRs through this mechanism; it must use MFCR and MTCR.

There are no instructions allowing bit, bit field or load-modify store accesses to the CSFRs. The RSTV instruction (Reset Overflow Flags) resets the overflow flags in the PSW, without modifying any of the other bits in the PSW. This instruction can be executed at any privilege level.

8.9.4 Enabling/Disabling the Interrupt System

For non-interruptible operations, the ENABLE and DISABLE instructions allow the explicit enabling and disabling of interrupts in user and supervisor modes. While disabled, an interrupt will not be taken by the CPU regardless of the relative priorities of the CPU and the highest interrupt pending. The only interrupt that will be serviced while interrupts are disabled is the NMI (non-maskable interrupt).

If a user process accidentally disables interrupts for longer than a specified time, watchdog timers can be used to recover.

Programs executing in supervisor mode can use the 16-bit Begin ISR (BISR) instruction to save the lower context of the current task, set the current CPU priority number, and re-enable interrupts (which are disabled by the processor when an interrupt is taken).

8.9.5 RET and RFE

The function return instruction, RET, is used to return from a function that was invoked via a CALL instruction. The return from exception instruction, RFE, is used to return from an interrupt or trap handler. The two instructions perform very similar operations; they restore the upper context of the calling function or interrupted task, and branch to the return address contained in register A11 (prior to the context restore operation). The instructions differ in the error checking they perform for call depth management. Issuing an RFE instruction when the current call depth (as tracked in the PSW) is nonzero generates a context nesting error trap. Conversely, a context call depth underflow trap is generated when an RET instruction is issued when the current call depth is zero.

8.9.6 Trap Instructions

The Trap on Overflow (TRAPV) and Trap on Sticky Overflow (TRAPSV) instructions can be used to cause a trap if the PSW's V and SV bits, respectively, are set. Refer to Section 8.1, "Arithmetic Instructions," on page 123.

8.9.7 No-operation

Although there are many ways to represent a no-operation (for example, adding zero to a register), an explicit NOP instruction is included so that it can be easily recognized, and the CPU can then minimize power consumption during its execution. For example, a sequence of NOP instructions in a loop could be used as a low-power state that has a very fast interrupt response time.

8.10 16-bit Instructions

The 16-bit instructions are a subset of the 32-bit instruction set, chosen because of their frequency of use. They significantly reduce static code size and thus provide a reduction in the cost of code memory and a higher effective instruction bandwidth. Because the 16-bit and 32-bit instructions all differ in the primary opcode, the two instruction sizes can be freely intermixed.

The 16-bit instructions are formed by imposing one or more of the following format constraints: smaller constants, smaller displacements, smaller offsets, implicit source, destination, or base address registers, and combined source and destination registers (the two-operand format). In addition, the 16-bit load and store instructions support only a limited set of addressing modes.

The registers D15 and A15 are used as implicit registers in many 16-bit instructions. For example, there is a 16-bit compare instruction (EQ) that puts a Boolean result in D15, and a 16-bit conditional move instruction (CMOV) which is controlled by the Boolean in D15.

The 16-bit load and store instructions are limited to the register indirect and stack-pointer relative (SP+offset) addressing modes.

TriCore Instruction Set

SIEMENS

9 TriCore Instruction Set

This chapter contains descriptions of all the TriCore instructions arranged alphabetically by instruction mnemonic. Each instruction page is organized into the following sections:

Syntax	Assembler syntax (Table 10 on page 116) followed by the instruction format in parentheses.
Description	A brief verbal description of the instruction's operation
Operation	A description of the instruction's operation in Register Transfer Language (RTL) (Table 13 on page 119)
Status	Any status flags that are affected by the instruction's execution (Table 14 on page 120)
Example(s)	One or more instruction examples
See Also	Related instructions

Throughout this chapter, information relating to 16-bit instructions is highlighted in screened areas.

9.1 Instruction Syntax

The syntax definition for an instruction specifies the operation to be performed and the operands used in the operation. Instruction operands are separated by commas.

Table 10 describes the terms used in the syntax definitions.

Table 10: Instruction Syntax Definitions

Symbol	Description
D*n*	Data register n
A*n*	Address register n
E*n*	Extended data register n containing a 64-bit value made from an even/odd pair of registers (Dn, Dn+1)
dispn	Displacement value of n bits used to form the effective address in branch instructions.
constn	Constant value of n bits used as instruction operand
offsetn	Offset value of n bits used to form the effective address in load and store instructions
p1, p2	Specifies the position of a single bit in bit and bit field instructions
w	Specifies the width of the bit field in bit and bit field instructions
<mode>	An addressing mode. Refer to Section 2.4.1, "TriCore Addressing Modes," on page 19.
CR	Core Registers (see Chapter 3, "Core Registers," for more information)

An instruction mnemonic is composed of up to three basic parts: a base operation, an operation modifier, and an operand (data type) modifier. For example, in the instruction:

```
ADDS.U
```

'ADD' is the base operation, 'S' is an operation modifier specifying that the result is saturated, and 'U' is a data type modifier specifying that the operands are unsigned.

The base operation specifies the basic operation that the instruction performs, for example, ADD for addition, J for jump, and LD for memory load. The operation modifier specifies more exactly the operation performed, for example, ADDI for addition using an immediate value, JL for a jump that includes a link, and LEA for a memory load of an effective address. More than one operation modifier may be used for some instructions (for example, ADDIH). The data type modifier indicates the data type of the source operands, for example, ADD.B for byte addition, JZ.A for a jump using an address register, and LD.H for a halfword load. The data type modifier is separated by a period ("."). Some instructions, for example, OR.EQ, have more than one base operation, and these base operations are also separated by a period.

Note that some 16-bit instructions use a general-purpose register as an implicit source or destination:

D15	Implicit Data Register for many 16-bit instructions
A10	Stack Pointer (SP)
A11	Return Address Register (RA) for CALL, JL, JLA, and JLI instructions, and Return PC value on interrupts
A15	Implicit base address register for many 16-bit load/store instructions

In the syntax section of the instruction descriptions, the implicit registers are included as explicit operand fields. However, they are not explicitly present in the encoded 16-bit instructions.

The operation modifiers are shown in Table 11. The order of the modifiers in the table is the same as the order in which they appear as modifiers in an instruction mnemonic.

Table 11: Operation Modifiers

Operation Modifier	Name	Description	Example
L	Link	Record link (jump subroutine)	JL
I	Indirect	Register indirect (jump)	JLI
A	Absolute	Absolute (jump)	JLA
EQ	Equal	Comparison equal	JEQ
NE	Not equal	Comparison not equal	JNE
LT	Less than	Comparison less than	JLT
GE	Greater than	Comparison greater than or equal	JGE
N	Not	Logical NOT	SELN
I	Immediate	Large immediate	ADDI
H	High word	Immediate value put in most-significant bits	ADDIH
Z	Zero	Use zero immediate	JNZ
R	Round	Round result (Q format data)	MULR
M	Multi-word	Multi-word result	MULM
S	Saturation	Saturate result	ADDS
X	Carry out	Update PSW carry bit	ADDX
C	Carry	Use and update PSW carry bit	ADDC
I	Increment	Increment counter	JNEI
D	Decrement	Decrement counter	JNED

The data type modifiers used in the instruction mnemonics are listed in Table 12. When multiple suffixes occur in an instruction, their order of occurrence in the mnemonic is the same as their order in the table.

Table 12: Data Type Modifiers

Data Type Modifier	Name	Description	Example
D	Doubleword	64-bit data/address	MOV.D
W	Word	32-bit (word) data	EQ.W
A	Address	32-bit address	ADD.A
Q	Q Format	16-bit signed fraction (Q format)	MADD.Q
H	Halfword	16-bit (halfword) data	ADD.H
B	Byte	packed byte data	ADD.B
T	Bit	1-bit data	AND.T
U	Unsigned	Unsigned data type	ADDS.U

9.2 Instruction Operation

The operation of each instruction is described using a C-like Register Transfer Level (RTL) notation, summarized in Table 13.

Note that the numbering of bits begins with bit zero, which is the least-significant bit of the word. Concatenation of bits and bit fields is specified using the notation "{x, y}" where "x" and "y" are expressions representing a bit or bit field. Any number of expressions can be concatenated, for example, "{x, y, z}."

Table 13: RTL Syntax Description

Symbol	Description
D[n]	Data register n
A[n]	Address register n
E[n]	Data register containing a 64-bit value, with the least-significant bit in D[n] and the most-significant bit in D[n+1], where n is even. The two parts are also referred to as E[n] (upper) and E[n] (lower)
p	Single bit p
(expression)[p]	Single bit p in multi-bit value
n'h pppp	Constant bit string, where n is the number of bits in the constant and "pppp" is the constant in hexadecimal; for example, "16'h FFFF"
n'b pppp	Constant bit string, where n is the number of bits in the constant and "pppp" is the constant in binary; for example, "2'b 11"
{x, y}	A bit string. x and y are expressions representing a bit or bit field. Any number of expressions can be concatenated, for example, "{x,y,z}".
dispn	Displacement value of n bits used to form the effective address in branch instructions
constn	Constant value of n bits used as instruction operand
offsetn	Offset value of n bits used to form the effective address in load and store instructions
sign_ext	Sign extension; high-order bit is left extended
one_ext	One extension; high-order bits are set to 1
zero_ext	High order bits are set to 0[1]
round16	The operation of adding 8000_{16} to a 32-bit value and then zeroing the least-significant 16 bits of the result
M	Memory address
EA	Effective address
target address	Address from which next instruction will be fetched. Used in call instructions.
M(EA, data_size)	Memory locations beginning at the specified byte location, EA, and extending to EA+data_size–1
and	Bit-wise logical AND
or	Bit-wise logical OR
xor	Bit-wise logical exclusive OR
!	Logical NOT

1. Zero_ext (const9) is actually zero_ext (const8), because const9 is a 9-bit constant for signed values but an 8-bit constant for unsigned values. For unsigned values, bit 8 is cleared.

9.3 Status

The Status section of the instruction page lists any of the five status flags in the Program Status Word (PSW) that may be affected by the operation. The status flags are described in Table 14 below.

Table 14: PSW Status Flags

Status Flag	Description
C	Carry. This flag is set as the result of a carry out from an addition or subtraction instruction. Carry out can result from either signed or unsigned operations. It is also set by automatic shift.
V	Overflow. This flag is updated by most arithmetic instructions. It is set when the result cannot be represented in the data size of the result; for example, when the result of a signed 32-bit operation is greater than $2^{31}-1$.
SV	Sticky Overflow. This flag is set when the overflow flag is set. It remains set until it is explicitly cleared by an RSTV (Reset Overflow bits) instruction.
AV	Advanced Overflow. This flag is updated by all instructions that update the overflow flag and no others. This flag is determined as the boolean exclusive of the two most-significant bits of the result.
SAV	Sticky Advanced Overflow. This flag is set whenever the advanced overflow flag is set. It remains set until it is explicitly cleared by an RSTV (Reset Overflow bits) instruction.

Refer also to the description of overflow conditions in Section 8.1.1, "Integer Arithmetic," on page 88 and to the description of the divide-step algorithm in Section 8.1.1.4, "Division," on page 90.

9.4 Instruction Formats

This section of the instruction page includes the format(s) used by the instruction. The 32-bit instruction formats are shown in Figure 64, and the 16-bit instruction formats are shown in Figure 65. The two opcode fields in the instruction format, op1 and op2, are specified for each instruction in Appendix A, Opcodes.

	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
ABS	off18[9..6]				op2		off18[13..10]				off18[5..0]						off18[17..14]				s1/d				op1							
ABSB	off18[9..6]				op2		off18[13..10]				off18[5..0]						off18[17..14]				b	bpos3			op1							
B	disp24[15..0]																disp24[23..16]								op1							
BIT	d				p2					op2		p1					s2				s1				op1							
BO	off10[9..6]				op2						off10[5..0]						s2				s1/d				op1							
BOL	off16[9..6]				off16[15..10]						off16[5..0]						s2				s1/d				op1							
BRC	op2	disp15															const4				s1				op1							
BRN	op2	disp15															n[3..0]				s1				n4	op1						
BRR	op2	disp15															s2				s1				op1							
RC	d				op2							const9									s1				op1							
RCPW	d				p					op2		w					const4				s1				op1							
RCR	d				s3				op2			const9									s1				op1							
RCRR	d				s3				op2								const4				s1				op1							
RCRW	d				s3				op2			w					const4				s1				op1							
RLC	d				const16																s1				op1							
RR	d				op2										n		s2				s1				op1							
RRPW	d				p					op2		w					s2				s1				op1							
RRR	d				s3				op2						n		s2				s1				op1							
RRRR	d				s3				op2								s2				s1				op1							
RRRW	d				s3				op2			w					s2				s1				op1							
SYS						op2																			op1							

Figure 64: 32-Bit Instruction Formats

TriCore Instruction Set 9

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SB	disp8								op1							
SBC	const4				disp4				op1							
SBR	s2				disp4				op1							
SBRN	n[3..0]				disp4				n4	op1						
SC	const8								op1							
SLR	s2				d				op1							
SLRO	off4				d				op1							
SR	op2				s1/d				op1							
SRC	const4				s1/d				op1							
SRO	s2				off4				op1							
SRR	s2				s1/d				op1							
SRRS	s2				s1/d				n		op1					
SSR	s2				s1				op1							
SSRO	off4				s1				op1							

Figure 65: 16-Bit Instruction Formats

9.5 Instruction Descriptions

The following pages describe the TriCore instruction set in detail.

ABS — Absolute Value — ABS

Syntax:

abs Dc, Da (RR)

Description:

Put the absolute value of data register *Da* in data register *Dc*; that is, if the contents of *Da* are greater than or equal to zero, copy it to *Dc*; otherwise, change the sign of *Da* and copy it to *Dc*. The operands are treated as signed, 32-bit integers. If Da = 0x8000.0000 (the maximum negative value), then Dc = 0x8000.0000, and an overflow is generated.

Operation:

```
if (D[a] >= 0) then D[c] = D[a]
else D[c] = –D[a]; signed
```

Status:

V, SV, AV, SAV

Example:

```
abs   d3, d1
```

See Also:

ABSDIF (pg 125), **ABSDIFS** (pg 127), **ABSS** (pg 129)

ABS.B — Absolute Value Packed Byte — ABS.B
ABS.H — Absolute Value Packed Halfword — ABS.H

Syntax:

```
abs.b    Dc, Da (RR)
abs.h    Dc, Da (RR)
```

Description:

Put the absolute value of each byte/halfword in data register *Da* into the corresponding byte/halfword of data register *Dc*. The operands are treated as signed, 8-bit/16-bit integers. The overflow condition is calculated for each byte/halfword of the packed quantity. Overflow occurs only if D[a] [(n +7):n] / D[a] [(n+15):n] has the maximum negative value of 0x80/0x8000. On overflow, D[a] [(n+ i):n] is unchanged, and the V flag is set.

Refer also to Section 8.1.3, "Packed Arithmetic," on page 95.

Operation:

```
if (D[a][(n+7):n] >= 0)
then D[c][(n+7):n] = D[a][(n+7):n]
else D[c][(n+7):n] = –D[a][(n+7):n]; n = 0, 8, 16, 24; signed

if (D[a][(n+15):n] >= 0)
then D[c][(n+15):n] = D[a][(n+15):n]
else D[c][(n+15):n] = –D[a][(n+15):n]; n = 0, 16; signed
```

Status:

V, SV, AV, SAV

Examples:

```
abs.b    d3, d1
abs.h    d3, d1
```

See Also:

ABSS.B (pg 130), **ABSS.H** (pg 130), **ABSDIF.B** (pg 126), **ABSDIF.H** (pg 126), **ABSDIFS.B** (pg 128), **ABSDIFS.H** (pg 128)

ABSDIF Absolute Value of Difference ABSDIF

Syntax:

absdif Dc, Da, Db (RR)
absdif Dc, Da, const9 (RC)

Description:

Put the absolute value of the difference between *Da* and *Db/const9* in *Dc*; namely, if the contents of data register *Da* are greater than *Db/const9*, then subtract *Db/const9* from *Da* and put the result in data register *Dc*; otherwise, subtract *Da* from *Db/const9* and put the result in *Dc*. The operands are treated as signed, 32-bit integers, and the *const9* value is sign-extended to 32 bits.

Operation:

```
if (D[a] > D[b]) then D[c] = D[a] – D[b]
else D[c] = D[b] – D[a]; signed

if (D[a] > sign_ext(const9)) then D[c] = D[a] – sign_ext(const9)
else D[c] = sign_ext(const9) – D[a]; signed
```

Status:

V, SV, AV, SAV

Examples:

```
absdif   d3, d1, d2
absdif   d3, d1, 126
```

See Also:

ABSS (pg 129), **ABSDIFS** (pg 127)

ABSDIF.B Absolute Value of Difference Packed Byte ABSDIF.B
ABSDIF.H Absolute Value of Difference Packed Halfword ABSDIF.H

Syntax:

```
absdif.b   Dc, Da, Db (RR)
absdif.h   Dc, Da, Db (RR)
```

Description:

Compute the absolute value of the difference between the corresponding bytes/halfwords of *Da* and *Db* and put each result in the corresponding byte/halfword of *Dc*. The operands are treated as signed, 8-bit/16-bit integers. The overflow condition is calculated for each byte/halfword of the packed quantity.

Refer also to Section 8.1.3, "Packed Arithmetic," on page 95.

Operation:

```
if (D[a][(n+7):n]>D[b][(n+7):n])
then D[c][(n+7):n] = D[a][(n+7):n] – D[b][(n+7):n]
else D[c][(n+7):n] = D[b][(n+7):n] – D[a][(n+7):n]; n = 0, 8, 16, 24; signed
```

```
if (D[a][(n+15):n] > D[b][(n+15):n])
then D[c][(n+15):n] = D[a][(n+15):n] – D[b][(n+15):n]
else D[c][(n+15):n] = D[b][(n+15):n] – D[a][(n+15):n]; n = 0, 16; signed
```

Status:

V, SV, AV, SAV

Examples:

```
absdif.b d3, d1, d2
absdif.h d3, d1, d2
```

See Also:

ABS.B (pg 124), **ABS.H** (pg 124), **ABSS.B** (pg 130), **ABSS.H** (pg 130), **ABSDIFS.B** (pg 128), **ABSDIFS.H** (pg 128)

ABSDIFS Absolute Value of Difference with Saturation ABSDIFS

Syntax:

absdifs Dc, Da, Db (RR)
absdifs Dc, Da, const9 (RC)

Description:

Put the absolute value of the difference between *Da* and *Db/const9* in *Dc*: namely, if the contents of data register *Da* are greater than *Db/const9*, then subtract *Db/const9* from *Da* and put the result in data register *Dc*; otherwise, subtract *Da* from *Db/const9* and put the result in *Dc*. The operands are treated as signed, 32-bit integers, with saturation on signed overflow. The *const9* value is sign-extended to 32 bits.

Operation:

```
if (D[a] > D[b]) then D[c] = D[a] – D[b]
else D[c] = D[b] – D[a]; signed; ssov

if (D[a] > sign_ext(const9)) then D[c] = D[a] – sign_ext(const9)
else D[c] = sign_ext(const9) – D[a]; signed; ssov
```

Status:

V, SV, AV, SAV

Examples:

```
absdifs  d3, d1, d2
absdifs  d3, d1, 126
```

See Also:

ABS (pg 123), **ABSDIF** (pg 125), **ABSS** (pg 129)

ABSDIFS.B Absolute Value of Difference Packed Byte ABSDIFS.B
ABSDIFS.H Absolute Value of Difference Packed Halfword ABSDIFS.H

Syntax:

absdif.b Dc, Da, Db (RR)
absdif.h Dc, Da, Db (RR)

Description:

Compute the absolute value of the difference of the corresponding bytes/halfwords of *Da* and *Db* and put each result in the corresponding byte/halfword of *Dc*. The operands are treated as signed, 8-bit/16-bit integers, with saturation on signed overflow. The overflow condition is calculated for each byte/halfword of the packed quantity.

Refer also to Section 8.1.3, "Packed Arithmetic," on page 95.

Operation:

```
if (D[a][(n+7):n] > D[b][(n+7):n])
then D[c][(n+7):n] = D[a][(n+7):n] - D[b][(n+7):n]
else D[c][(n+7):n] = D[b][(n+7):n] - D[a][(n+7):n]; n = 0, 8, 16, 24; signed; ssov

if (D[a][(n+15):n] > D[b][(n+15):n])
then D[c][(n+15):n] = D[a][(n+15):n] - D[b][(n+15):n]
else D[c][(n+15):n] = D[b][(n+15):n] - D[a][(n+15):n]; n = 0, 16; signed; ssov
```

Status:

V, SV, AV, SAV

Examples:

```
absdif.b    d3, d1, d2
absdif.h    d3, d1, d2
```

See Also:

ABS.B (pg 124), **ABS.H** (pg 124), **ABSS.B** (pg 130), **ABSS.H** (pg 130), **ABSDIF.B** (pg 126), **ABSDIF.H** (pg 126)

ABSS Absolute Value with Saturation ABSS

Syntax:

abss Dc, Da (RR)

Description:

Put the absolute value of data register *Da* in data register *Dc*; that is, if the contents of *Da* are greater than or equal to zero, copy it to *Dc*; otherwise, change the sign of *Da* and copy it to *Dc*. The operands are treated as signed, 32-bit integers, with saturation on signed overflow. If Da = 0x8000.0000 (the maximum negative value), then Dc = 0x8000.0000, and an overflow is generated.

Operation:

```
if (D[a] >= 0) then D[c] = D[a]
else D[c] = –D[a]; signed; ssov
```

Status:

V, SV, AV, SAV

Example:

```
abss  d3, d1
```

See Also:

ABS (pg 123), **ABSDIF** (pg 125), **ABSDIFS** (pg 127)

ABSS.B Absolute Value Packed Byte with Saturation ABSS.B

ABSS.H Absolute Value Packed Halfword with Saturation ABSS.H

Syntax:

```
abss.b    Dc, Da (RR)
abss.h    Dc, Da (RR)
```

Description:

Put the absolute value of each byte/halfword in data register *Da* in the corresponding byte/halfword of data register *Dc*. The operands are treated as signed, 8-bit/16-bit integers, with saturation on signed overflow. The overflow condition is calculated for each byte/halfword of the packed quantity. Overflow occurs only if D[a] [(n +7):n] / D[a] [(n+15):n] has the maximum negative value of 0x80/ 0x8000. On overflow, D[a] [(n+ i):n] is unchanged, and the V flag is set.

Refer also to Section 8.1.3, "Packed Arithmetic," on page 95.

Operation:

```
if (D[a][(n+7):n] >= 0)
then D[c][(n+7):n] = D[a][(n+7):n]
else D[c][(n+7):n] = –D[a][(n+7):n]; n = 0, 8, 16, 24; signed; ssov

if (D[a][(n+15):n] >= 0)
then D[c][(n+15):n] = D[a][(n+15):n]
else D[c][(n+15):n] = –D[a][(n+15):n];n = 0, 16; signed; ssov
```

Status:

V, SV, AV, SAV

Examples:

```
abss.b   d3, d1
abss.h   d3, d1
```

See Also:

ABS.B (pg 124), **ABS.H** (pg 124), **ABSDIF.B** (pg 126), **ABSDIF.H** (pg 126), **ABSDIFS.B** (pg 128), **ABSDIFS.H** (pg 128)

ADD Add ADD

Syntax:

```
add    Dc, Da, Db, (RR)
add    Dc, Da, const9 (RC)
add    Da, Db (SRR)
add    Da, const4 (SRC)
add    D15, Da, Db (SRR)
add    D15, Da, const4 (SRC)
```

Description:

Add the contents of data register *Da* to the contents of data register *Db*/*const9* and put the result in data register *Dc*. The operands are treated as 32-bit integers, and the *const9* value is sign-extended to 32 bits before the addition is performed.

Add the contents of data register *Da* to the contents of data register *Db*/*const4* and put the result in data register *Da*/D15. The operands are treated as unsigned, 32-bit integers, and the *const4* value is sign-extended to 32 bits before the addition is performed.

Operation:

```
D[c] = D[a] + D[b]
D[c] = D[a] + sign_ext(const9)
D[a] = D[a] + D[b]
D[a] = D[a] + sign_ext(const4)
D[15] = D[a] + D[b]
D[15] = D[a] + sign_ext(const4)
```

Status:

V, SV, AV, SAV

Examples:

```
add   d3, d1, d2
add   d3, d1, 126
add   d1, d2
add   d1, 6
add   d15, d1, d2
add   d15, d1, 6
```

See Also:

ADDC (pg 135), **ADDI** (pg 136), **ADDIH** (pg 137), **ADDS** (pg 139), **ADDS.U** (pg 142), **ADDX** (pg 144)

ADD.A Add Address ADD.A

Syntax:

add.a Ac, Aa, Ab (RR)

Description:

Add the contents of address register *Aa* to the contents of address register *Ab* and put the result in address register *Ac*.

Operation:

A[c] = A[a] + A[b]

Example:

```
add.a a3, a4, a2
```

See Also:

ADDIH.A (pg 138), **ADDSC.A** (pg 143), **ADDSC.AT** (pg 143), **DIFSC.A** (pg 176), **SUB.A** (pg 380), **SUBSC.A** (pg 386)

ADD.B Add Packed Byte ADD.B
ADD.H Add Packed Halfword ADD.H

Syntax:

add.b	Dc, Da, Db (RR)
add.h	Dc, Da, Db (RR)

Description:

Add the contents of each byte/halfword of *Da* and *Db* and put the result in each corresponding byte/halfword of *Dc*. The overflow condition is calculated for each byte/halfword of the packed quantity, and the status flags are set if any of the bytes/halfwords generate or almost generate an overflow.

Refer also to Section 8.1.3, "Packed Arithmetic," on page 95.

Operation:

```
D[c][(n+7):n] = D[a][(n+7):n] + D[b][(n+7):n], n = 0, 8, 16, 24
D[c][(n+15):n] = D[a][(n+15):n] + D[b][(n+15):n]; n = 0, 16
```

Status:

V, SV, AV, SAV

Examples:

```
add.b d3, d1, d2
add.h d3, d1, d2
```

See Also:

ADD.H (pg 134), **ADDS.B** (pg 140), **ADDS.BU** (pg 140), **ADDS.H** (pg 141), **ADDS.HU** (pg 141)

ADDC Add with Carry ADDC

Syntax:

```
addc    Dc, Da, Db (RR)
addc    Dc, Da, const9 (RC)
```

Description:

Add the contents of data register *Da* to the contents of data register *Db/const9* and put the result in data register *Dc*. The operands are treated as 32-bit integers, and the value *const9* is sign-extended to 32 bits before the addition is performed. The PSW carry bit is used as the carry in and updates the PSW carry bit with the ALU carry out.

Operation:

```
D[c] = D[a] + D[b] + PSW.C; PSW.C = carry_out
D[c] = D[a] + sign_ext(const9) + PSW.C; PSW.C = carry_out
```

Status:

C, V, SV, AV, SAV

Examples:

```
addc  d3, d1, d2
addc  d3, d1, 126
addc  d3, d1, 253
```

See Also:

ADD (pg 131), **ADDI** (pg 136), **ADDIH** (pg 137), **ADDS** (pg 139), **ADDS.U** (pg 142), **ADDX** (pg 144)

ADDI Add Immediate ADDI

Syntax:

addi Dc, Da, const16 (RLC)

Description:

Add the contents of data register *Da* to the value *const16*, and put the result in data register *Dc*. The operands are treated as 32-bit integers. The value *const16* is sign-extended to 32 bits before the addition is performed.

Operation:

D[c] = D[a] + sign_ext(const16)

Status:

V, SV, AV, SAV

Example:

```
addi  d3, d1, -14526
```

See Also:

ADD (pg 131), **ADDC** (pg 135), **ADDIH** (pg 137), **ADDS** (pg 139), **ADDS.U** (pg 142), **ADDX** (pg 144)

ADDIH Add Immediate High ADDIH

Syntax:

addih Dc, Da, const16 (RLC)

Description:

Const16 is left-shifted 16 bits, zero-filled, and added to D[a]. The results are put in D[c].

Operation:

D[c] = D[a] + {const16, 16'h 0000}

Status:

V, SV, AV, SAV

Example:

```
addih d3, d1, -14526
```

See Also:

ADD (pg 131), **ADDC** (pg 135), **ADDI** (pg 136), **ADDS** (pg 139), **ADDS.U** (pg 142), **ADDX** (pg 144)

ADDIH.A Add Immediate High to Address ADDIH.A

Syntax:

addih.a Ac, Aa, const16 (RLC)

Description:

Left-shift *const16* by 16 bits, add the contents of address register *Aa*, and put the result in address register *Ac*.

Operation:

A[c] = A[a] + {const16, 16'h 0000}

Example:

```
addih.a  a3, a4, -14526
```

See Also:

ADD.A (pg 133), **ADDSC.A** (pg 143), **ADDSC.AT** (pg 143), **DIFSC.A** (pg 176), **SUBSC.A** (pg 386)

ADDS Add Signed with Saturation ADDS

Syntax:

```
adds    Dc, Da, Db (RR)
adds    Dc, Da, const9 (RC)
```

Description:

Add the contents of data register *Da* to the value in data register *Db/const9* and put the result in data register *Dc*. The operands are treated as signed, 32-bit integers, with saturation on signed overflow. The value *const9* is sign-extended to 32 bits before the addition is performed.

Operation:

```
D[c] = D[a] + D[b]; signed; ssov
D[c] = D[a] + sign_ext(const9); signed; ssov
```

Status:

V, SV, AV, SAV

Examples:

```
adds  d3, d1, d2
adds  d3, d1, 126
adds  d3, d1, 253
```

See Also:

ADD (pg 131), **ADDC** (pg 135), **ADDI** (pg 136), **ADDIH** (pg 137), **ADDS.U** (pg 142), **ADDX** (pg 144)

ADDS.B Add Signed Packed Byte with Saturation ADDS.B
ADDS.BU Add Unsigned Packed Byte with Saturation ADDS.BU

Syntax:

adds.b	Dc, Da, Db (RR)
adds.bu	Dc, Da, Db (RR)

Description:

Add the contents of each byte of *Da* and *Db* and put each result in the corresponding byte of *Dc*, with saturation on signed/unsigned overflow. The overflow and advanced overflow conditions are calculated for each byte of the packed quantity.

Refer also to Section 8.1.3, "Packed Arithmetic," on page 95.

Operation:

```
D[c][(n+7):n] = D[a][(n+7):n] + D[b][(n+7):n]; n = 0, 8, 16, 24; signed; ssov
D[c][(n+7):n] = D[a][(n+7):n] + D[b][(n+7):n]; n = 0, 8, 16,24; unsigned; suov
```

Status:

V, SV, AV, SAV

Examples:

```
adds.b   d3, d1, d2
adds.bu  d3, d1, d2
```

See Also:

ADD.B (pg 134), **ADD.H** (pg 134), **ADDS.H** (pg 141), **ADDS.HU** (pg 141)

ADDS.H Add Signed Packed Halfword with Saturation ADDS.H
ADDS.HU Add Unsigned Packed Halfword with Saturation ADDS.HU

Syntax:

```
adds.h    Dc, Da, Db (RR)
adds.hu   Dc, Da, Db (RR)
```

Description:

Add the contents of each halfword of *Da* and *Db* and put the result in each corresponding halfword of *Dc*, with saturation on signed/unsigned overflow. The overflow and advanced overflow conditions are calculated for each halfword of the packed quantity.

Operation:

```
D[c][(n+15):n] = D[a][(n+15):n] + D[b][(n+15):n]; n = 0, 16; signed; ssov
D[c][(n+15):n] = D[a][(n+15):n] + D[b][(n+15):n]; n = 0, 16; unsigned; suov
```

Status:

V, SV, AV, SAV

Examples:

```
adds.h    d3, d1, d2
adds.h    d3, d1, 126
adds.h    d3, d1, 253
adds.hu   d3, d1, d2
adds.hu   d3, d1, 126
adds.hu   d3, d1, 253
```

See Also:

ADD.B (pg 134), **ADD.H** (pg 134), **ADDS.B** (pg 140), **ADDS.BU** (pg 140)

ADDS.U Add Unsigned with Saturation ADDS.U

Syntax:

```
adds.u    Dc, Da, Db (RR)
adds.u    Dc, Da, const9 (RC)
```

Description:

Add the contents of data register *Da* to the contents of data register *Db/const9* and put the result in data register *Dc*. The operands are treated as unsigned, 32-bit integers, with saturation on unsigned overflow. The *const9* value is zero-extended to 32 bits.

Operation:

D[c] = D[a] + D[b]; unsigned; suov
D[c] = D[a] + zero_ext(const9); unsigned; suov

Status:

V, SV, AV, SAV

Examples:

```
adds.u   d3, d1, d2
adds.u   d3, d1, 126
adds.u   d3, d1, 253
```

See Also:

ADD (pg 131), **ADDC** (pg 135), **ADDI** (pg 136), **ADDIH** (pg 137), **ADDS** (pg 139), **ADDX** (pg 144)

ADDSC.A — Add Scaled Index to Address — ADDSC.A
ADDSC.AT — Add Bit-Scaled Index to Address — ADDSC.AT

Syntax:

addsc.a	Ac, Aa, Db, n (RRS)
addsc.a	Aa, Db, n (SRRS)
addsc.at	Ac, Aa, Db (RRS)

Description:

Left-shift the contents of data register *Db* by the amount specified by *n*, where *n* can be 0, 1, 2, or 3. Add that value to the contents of address register *Aa* and put the result in address register *Ac*.

Left-shift the contents of data register *Db* by the amount specified by *n*, where *n* can be 0, 1, 2, or 3. Add that value to the contents of address register *Aa* and put the result in address register *Aa*.

Right-shift the contents of *Db* by 3 (with sign fill). Add that value to the contents of address register *Aa* and clear the bottom two bits to zero. Put the result in *Ac*.

The instruction ADDSC.AT generates the address of the word containing the bit indexed by *Db*, starting from the base address in *Aa*.

Operation:

```
A[c] = A[a] + (D[b] << n), n = 0, 1, 2, or 3
A[a] = A[a] + (D[b] << n), n = 0, 1, 2, or 3
A[c] = A[a] + (D[b] >> 3) and ! 2'b 11
```

Example:

```
addsc.at a3, a4, d2
```

See Also:

ADD.A (pg 133), **ADDIH.A** (pg 138), **DIFSC.A** (pg 176), **SUB.A** (pg 380), **SUBSC.A** (pg 386)

ADDX Add Extended ADDX

Syntax:

```
addx    Dc, Da, Db (RR)
addx    Dc, Da, const9 (RC)
```

Description:

Add the contents of data register *Da* to the contents of data register *Db/const9* and put the result in data register *Dc*. The operands are treated as 32-bit integers, and the *const9* value is sign-extended to 32 bits before the addition is performed. The PSW carry bit is set to the value of the ALU carry out.

Operation:

D[c] = D[a] + D[b]; PSW.C = carry_out
D[c] = D[a] + sign_ext(const9); PSW.C = carry_out

Status:

C, V, SV, AV, SAV

Examples:

```
addx  d3, d1, d2,
addx  d3, d1, 126
addx  d3, d1, 253
```

See Also:

ADD (pg 131), **ADDC** (pg 135), **ADDI** (pg 136), **ADDIH** (pg 137), **ADDS** (pg 139), **ADDS.U** (pg 142)

AND Logical AND AND

Syntax

and	Dc, Da, Db (RR)
and	Dc, Da, const9 (RC)
and	Da, Db (SRR)
and	D15, const8 (SC)

Description:

Compute the bitwise logical AND of the contents of data register *Da* and the contents of data register *Db/const9* and put the result in data register *Dc*. The operands are treated as unsigned, 32-bit integers, and the *const9* value is zero-extended to 32 bits.

Compute the bitwise logical AND of the contents of data register *Da*/D15 and the contents of data register *Db/const8* and put the result in data register *Da*/D15. The operands are treated as unsigned, 32-bit integers, and the *const8* value is zero-extended to 32 bits.

Operation:

```
D[c] = D[a] and D[b]
D[c] = D[a] and zero_ext(const9)
D[a] = D[a] and D[b]
D[15] = D[15] and zero_ext(const8)
```

Examples:

```
and   d3, d1, d2
and   d3, d1, 126
and   d1, d2
and   d15, 126
```

See Also:

ANDN (pg 152), **NAND** (pg 319), **NOR** (pg 325), **NOT** (pg 327), **OR** (pg 328), **ORN** (pg 335), **XNOR** (pg 394), **XOR** (pg 396)

AND.AND.T Accumulating Logical AND-AND AND.AND.T
AND.ANDN.T Accumulating Logical AND-AND-Not AND.ANDN.T
AND.NOR.T Accumulating Logical AND-NOR AND.NOR.T
AND.OR.T Accumulating Logical AND-OR AND.OR.T

Syntax:

```
and.and.t  Dc, Da, p1, Db, p2 (BIT)
and.andn.tDc, Da, p1, Db, p2 (BIT)
and.nor.t  Dc, Da, p1, Db, p2 (BIT)
and.or.t   Dc, Da, p1, Db, p2 (BIT)
```

Description:

Compute the logical AND/ANDN/NOR/OR of the value of bit *p1* of data register *Da* and bit *p2* of *Db*. Then compute the logical AND of that result and bit 0 of *Dc*, and put the result back in bit 0 of *Dc*. All other bits in *Dc* are unchanged.

Refer also to Section 8.3, "Bit Operations," on page 100.

Operation:

```
and.and.t :  D[c] = {D[c][31:1], D[c][0] and (D[a][p1] and D[b][p2])}
and.andn.t : D[c] = {D[c][31:1], D[c][0] and (D[a][p1] and !D[b][p2])}
and.or.t :   D[c] = {D[c][31:1], D[c][0] and !(D[a][p1] or D[b][p2])
and.or.t :   D[c] = {D[c][31:1], D[c][0] and (D[a][p1] or D[b][p2])
```

Examples:

```
and.and.t    d3, d1, 4, d2, 9
and.andn.t   d3, d1, 6, d2, 15
and.nor.t    d3, d1, 5, d2, 9
and.or.t     d3, d1, 4, d2, 6
```

See Also:

OR.AND.T (pg 329), **OR.ANDN.T** (pg 329), **OR.NOR.T** (pg 329), **OR.OR.T** (pg 329), **SH.AND.T** (pg 358), **SH.ANDN.T** (pg 358), **SH.NAND.T** (pg 358), **SH.NOR.T** (pg 358), **SH.OR.T** (pg 358), **SH.ORN.T** (pg 358), **SH.XNOR.T** (pg 358), **SH.XOR.T** (pg 358)

AND.EQ Equal Accumulating AND.EQ

Syntax:

and.eq	Dc, Da, Db (RR)
and.eq	Dc, Da, const9 (RC)

Description:

Compute the logical AND of *Dc*[0] and the Boolean result of the EQ operation on the contents of data register *Da* and data register *Db*/*const9*. Put the result in *Dc*[0]. All other bits in *Dc* are unchanged. The *const9* value is sign-extended to 32 bits.

Refer also to Section 8.2, "Compare Instructions," on page 97.

Operation:

```
D[c][0] = D[c][0] AND (D[a] == D[b])
D[c][0] = D[c][0] AND (D[a] == sign_ext(const9))
```

Examples:

```
and.eq d3, d1, d2
and.eq d3, d1, 126
```

See Also:

OR.EQ (pg 330), **XOR.EQ** (pg 397)

AND.GE — Greater Than or Equal Accumulating — AND.GE
AND.GE.U — Greater Than or Equal Accumulating Unsigned — AND.GE.U

Syntax:

```
and.ge    Dc, Da, Db (RR)
and.ge    Dc, Da,const9 (RC)
and.ge.u  Dc, Da, Db (RR)
and.ge.u  Dc, Da,const9 (RC)
```

Description:

Calculate the logical AND of *Dc*[0] and the Boolean result of the GE operation on the contents of data register *Da* and data register *Db*/*const9*. Put the result in *Dc*[0]. All other bits in *Dc* are unchanged. *Da* and *Db* are treated as 32-bit signed integers. The *const9* value is sign-extended to 32 bits.

Calculate the logical AND of *Dc*[0] and the Boolean result of the GE.U operation on the contents of data register *Da* and data register *Db*/*const9*. Put the result in *Dc*[0]. All other bits in *Dc* are unchanged. *Da* and *Db* are treated as 32-bit unsigned integers. The *const9* value is zero-extended to 32 bits.

Refer also to Section 8.2, "Compare Instructions," on page 97.

Operation:

```
D[c] = D[c][0] op (D[a] >= D[b]); signed
D[c] = D[c][0] op (D[a] >= sign_ext(const9)); signed

D[c] = D[c][0] op (D[a] >= D[b]); unsigned
D[c] = D[c][0] op (D[a] >= zero_ext(const9)); unsigned
```

Examples:

```
and.ge   d3, d1, d2
and.ge   d3, d1, 126
and.ge.u d3, d1, d2
and.ge.u d3, d1, 126
```

See Also:

OR.GE (pg 331), **OR.GE.U** (pg 331), **XOR.GE** (pg 398), **XOR.GE.U** (pg 398)

AND.LT — Less Than Accumulating — AND.LT
AND.LT.U — Less Than Accumulating Unsigned — AND.LT.U

Syntax:

and.lt	Dc, Da, Db (RR)
and.lt	Dc, Da,const9 (RC)
and.lt.u	Dc, Da, Db (RR)
and.lt.u	Dc, Da,const9 (RC)

Description:

Calculate the logical AND of *Dc*[0] and the Boolean result of the LT operation on the contents of data register *Da* and data register *Db*/*const9*. Put the result in *Dc*[0]. All other bits in *Dc* are unchanged. *Da* and *Db* are treated as 32-bit signed integers. The *const9* value is sign-extended to 32 bits.

Calculate the logical AND of *Dc*[0] and the Boolean result of the LT.U operation on the contents of data register *Da* and data register *Db*/*const9*. Put the result in *Dc*[0]. All other bits in *Dc* are unchanged. *Da* and *Db* are treated as 32-bit unsigned integers. The *const9* value is zero-extended to 32 bits.

Refer also to Section 8.2, "Compare Instructions," on page 97.

Operation:

```
D[c] = D[c][0] AND (D[a] < D[b]); signed
D[c] = D[c][0] AND (D[a] < sign_ext(const9)); signed

D[c] = D[c][0] AND (D[a] < D[b]); unsigned
D[c] = D[c][0] AND (D[a] < zero_ext(const9)); unsigned
```

Examples:

```
and.lt   d3, d1, d2
and.lt   d3, d1, 126
and.lt.u d3, d1, d2
and.lt.u d3, d1, 126
```

See Also:

OR.LT (pg 332), **OR.LT.U** (pg 332), **XOR.LT** (pg 399), **XOR.LT.U** (pg 399)

AND.NE Not Equal Accumulating AND.NE

Syntax:

```
and.ne    Dc, Da, Db (RR)
and.ne    Dc, Da,const9 (RC)
```

Description:

Calculate the logical AND of *Dc*[0] and the Boolean result of the NE operation on the contents of data register *Da* and data register *Db*/*const9*. Put the result in *Dc*[0]. All other bits in *Dc* are unchanged. The *const9* value is sign-extended to 32 bits.

Refer also to Section 8.2, "Compare Instructions," on page 97.

Operation:

```
D[c] = D[c][0] AND (D[a] != D[b])
D[c] = D[c][0] AND (D[a] != sign_ext(const9))
```

Examples:

```
and.ne    d3, d1, d2
and.ne    d3, d2, 126
```

See Also:

OR.NE (pg 333), **XOR.NE** (pg 400)

AND.T Bit Logical AND AND.T

Syntax:

and.t Dc, Da, p1, Db, p2 (BIT)

Description:

Compute the logical AND of bit *p1* of data register *Da* and bit *p2* of data register *Db*. Put the result in the least-significant bit of data register *Dc* and clear the remaining bits of Dc to zero.

Refer also to Section 8.3, "Bit Operations," on page 100.

Operation:

D[c] = D[a][p1] and D[b][p2]

Example:

```
and.t    d3, d1, 7, d2, 2
```

See Also:

ANDN.T (pg 153), **NAND.T** (pg 320), **NOR.T** (pg 326), **OR.T** (pg 334), **ORN.T** (pg 336), **XNOR.T** (pg 395), **XOR.T** (pg 401)

ANDN AND-Not ANDN

Syntax:

```
andn    Dc, Da, Db (RR)
andn    Dc, Da,const9 (RC)
```

Description:

Compute the bitwise logical AND of the contents of data register *Da* and the ones-complement of the contents of data register *Db*/*const9* and put the result in data register *Dc*. The operands are treated as unsigned, 32-bit integers. The *const9* value is zero-extended to 32 bits.

Operation:

```
D[c] = D[a] and !D[b]
D[c] = D[a] and !zero_ext(const9)
```

Examples:

```
andn  d3, d1, d2
andn  d3, d1, 126
```

See Also:

AND (pg 145), **NAND** (pg 319), **NOR** (pg 325), **NOT** (pg 327), **OR** (pg 328), **ORN** (pg 335), **XNOR** (pg 394), **XOR** (pg 396)

ANDN.T — Bit Logical AND-Not — ANDN.T

Syntax:

andn.t Dc, Da, p1, Db, p2 (BIT)

Description:

Compute the logical AND of bit *p1* of data register *Da* and the inverse of bit *p2* of data register *Db*. Put the result in the least-significant bit of data register *Dc* and clear the remaining bits of Dc to zero.

Refer also to Section 8.3, "Bit Operations," on page 100.

Operation:

D[c] = D[a][p1] and !D[b][p2]

Example:

```
andn.t   d3, d1, 2, d2, 5
```

See Also:

AND.T (pg 151), **NAND.T** (pg 320), **NOR.T** (pg 326), **OR.T** (pg 334), **ORN.T** (pg 336), **XNOR.T** (pg 395), **XOR.T** (pg 401)

BISR Begin ISR BISR

Syntax:

bisr	const9 (RC)
bisr	const8 (SC)

Description:

Save the lower context by storing the contents of A2 – A7, D0 – D7, and the current PC to the current memory location pointed to by the FCX. Set the current CPU priority number (ICR.CCPN) to the value of *const9[7:0]*/*const8*, and enable interrupts (set ICR.IE to one). Note that BISR can be executed only in supervisor privilege mode.

This instruction is intended to be one of the first executed instructions in an interrupt routine. If the interrupt routine has not altered the lower context, the saved lower context is from the interrupted task.

If a BISR instruction is issued at the beginning of an interrupt, then an RSLCX instruction should be performed before returning with the RFE instruction.

Refer also to Section 8.9.4, "Enabling/Disabling the Interrupt System," on page 111.

Example:

```
bisr  126
```

See Also:

DISABLE (pg 177), **ENABLE** (pg 185), **LDLCX** (pg 233), **LDUCX** (pg 235), **RET** (pg 337), **RFE** (pg 338), **RSLCX** (pg 339), **STLCX** (pg 377), **STUCX** (pg 378), **SVLCX** (pg 388)

CADD Conditional Add CADD

Syntax:

```
cadd    Dc, Dd, Da, Db (RRR)
cadd    Dc, Dd, Da, const9 (RCR)

cadd    Da, D15, Db (SRR)
cadd    Da, D15, const4 (SCR)
```

Description:

If the contents of data register *Dd* are non-zero, add the contents of data register *Da* and the contents of register *Db*/*const9* and put the result in data register *Dc*; otherwise, put the contents of *Da* in *Dc*. The *const9* value is sign-extended to 32 bits.

If the contents of data register D15 are non-zero, add the contents of data register *Da* and the contents of register *Db*/*const4* and put the result in data register *Da*; otherwise, the contents of *Da* is unchanged. The *const4* value is sign-extended to 32 bits.

Operation:

```
D[c] = ((D[d]  != 0) ? D[a] + D[b] : D[a])
D[c] = ((D[d]  != 0) ? D[a] + sign_ext(const9) : D[a])

D[a] = ((D[15] != 0) ? D[a] + D[b] : D[a])
D[a] = ((D[15] != 0) ? D[a] + sign_ext(const4) : D[a])
```

Status:

V, SV, AV, SAV

Examples:

```
cadd  d3, d4, d1, d2
cadd  d3, d4, d1, 126
cadd  d1, d15, d2
cadd  d1, d15, 6
```

See Also:

CADDN (pg 157), **CMOV** (pg 168), **CMOVN** (pg 169), **CSUB** (pg 170), **CSUBN** (pg 172), **SEL** (pg 347), **SELN** (pg 349)

CADD.A Conditional Add to Address CADD.A

Syntax:

cadd.a	Ac, Dd, Aa, Ab (RRR)
cadd.a	Ac, Dd, Aa, const9 (RCR)

Description:

If the contents of data register *Dd* are non-zero, add the contents of address register *Aa* and the contents of register *Ab/const9* and put the result in address register *Ac*; otherwise, put the contents of A*a* in *Ac*. The *const9* value is sign-extended to 32 bits.

Operation:

A[c] = ((D[d] != 0) ? A[a] + A[b] : A[a])
A[c] = ((D[d] != 0) ? A[a] + sign_ext(const9) : A[a])

Examples:

```
cadd.a   a3, d4, a4, a2
cadd.a   a3, d4, a4, 126
```

See Also:

CADDN.A (pg 158), **CSUB.A** (pg 171), **CSUBN.A** (pg 173), **SEL.A** (pg 348), **SELN.A** (pg 350)

CADDN Conditional Add-Not CADDN

Syntax:

```
caddn   Dc, Dd, Da, Db (RRR)
caddn   Dc, Dd, Da, const9 (RCR)

caddn   Da, D15, Db (SRR)
caddn   Da, D15, const4 (SRC)
```

Description:

If the contents of data register *Dd* are zero, add the contents of data register *Da* and the contents of register *Db*/*const9* and put the result in data register *Dc*; otherwise, put the contents of *Da* in *Dc*.The *const9* value is sign-extended to 32 bits.

If the contents of data register D15 are zero, add the contents of data register *Da* and the contents of register *Db*/*const4* and put the result in data register *Da*; otherwise, the contents of *Da* is unchanged The *const4* value is sign-extended to 32 bits.

Operation:

```
D[c] = ((D[d] == 0) ? D[a] + D[b] : D[a])
D[c] = ((D[d] == 0) ? D[a] + sign_ext(const9) : D[a])

D[a] = ((D[15] == 0) ? D[a] + D[b] : D[a])
D[a] = ((D[15] == 0) ? D[a] + sign_ext(const4) : D[a])
```

Status:

V, SV, AV, SAV

Examples:

```
caddn d3, d4, d1, d2
caddn d3, d4, d1, 126
caddn d1, d15, d2
caddn d1, d15, 6
```

See Also:

CADD (pg 155), **CMOV** (pg 168), **CMOVN** (pg 169), **CSUB** (pg 170), **CSUBN** (pg 172), **SEL** (pg 347), **SELN** (pg 349)

CADDN.A Conditional Add-Not to Address CADDN.A

Syntax:

```
caddn.a   Ac, Dd, Aa, Ab (RRR)
caddn.a   Ac, Dd, Aa, const9 (RCR)
```

Description:

If the contents of data register *Dd* are zero, add the contents of address register *Aa* and the contents of register *Ab*/*const9* and put the result in address register *Ac*; otherwise, put the contents of A*a* in *Ac*. The *const9* value is sign-extended to 32 bits.

Operation:

```
A[c] = ((D[d] == 0) ? A[a] + A[b] : A[a])
A[c] = ((D[d] == 0) ? A[a] + sign_ext(const9) : A[a])
```

Examples:

```
caddn.a  a3, d4, a4, a2
caddn.a  a3, d4, a4, 126
```

See Also:

CADD.A (pg 156), **CSUB.A** (pg 171), **CSUBN.A** (pg 173), **SEL.A** (pg 348), **SELN.A** (pg 350)

CALL Call CALL

Syntax:

call disp24 (B)

Description:

Add the value specified by disp24, multiplied by two and sign-extended to 32 bits, to the address of the CALL instruction, and jump to the resulting address. The target address range is ± 16 MBytes relative to the current PC. In parallel with the jump, save the caller's upper context to an available context save area (CSA). Then set register A11 to the address of the next instruction beyond the call.

Refer to Section 8.6.1, "Unconditional Branch," on page 103 for an overview of all unconditional control transfer instructions. Refer also to Section 4.2, "Task Switching Operation," on page 48 for details of CSA management.

Operation:

```
ret_addr = PC + 4;
PC = PC + sign_ext(2 * disp24);
Save upper context;
A11 = ret_addr;
```

Example:

```
call   foobar
```

See Also:

CALLA (pg 160), **CALLI** (pg 161), **RET** (pg 337)

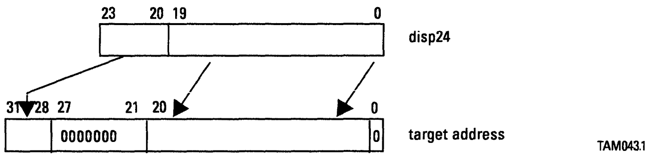
CALLA Call Absolute CALLA

Syntax:

calla disp24 (B)

Description:

Jump to the address specified by *disp24,* as shown below. In parallel with the jump, save the caller's upper context to an available context save area (CSA). Then set register A11 to the address of the next instruction beyond the call.



Refer to Section 8.6.1, "Unconditional Branch," on page 103 for an overview of all unconditional control transfer instructions. Refer also to Section 4.2, "Task Switching Operation," on page 48 for details of CSA management.

Operation:

ret_addr = PC + 4;
PC = PC + sign_ext(2 * disp24);
Save upper context;
A11 = ret_addr;

Example:

```
calla foobar
```

See Also:

CALL (pg 159), **CALLI** (pg 161), **JL** (pg 206), **JLA** (pg 207), **RET** (pg 337)

CALLI Call Indirect CALLI

Syntax:

calli Ab (RR)

Description:

Jump to the address specified by the contents of address register *Ab*. In parallel with the jump, save the caller's upper context to an available context save area (CSA). Then set register A11 to the address of the next instruction beyond the call.

Refer to Section 8.6.1, "Unconditional Branch," on page 103 for an overview of all unconditional control transfer instructions. Refer also to Section 4.2, "Task Switching Operation," on page 48 for details of CSA management.

Operation:

```
ret_addr = PC + 4;
PC = PC + sign_ext(2 * Ab);
Save upper context;
A11 = ret_addr;
```

Example:

```
calli a2
```

See Also:

CALL (pg 159), **CALLA** (pg 160), **RET** (pg 337)

CLO Count Leading Ones CLO

Syntax:

clo Dc, Da (RR)

Description:

Count the number of consecutive ones in *Da*, starting with bit 31, and put the result in *Dc*.
Refer also to Section 8.1.1.9, "Count Leading Zeroes, Ones, and Signs," on page 91.

Operation:

D[c] = #leading_ones (D[a])

Example:

```
clo d3, d1
```

See Also:

CLO.B (pg 163), **CLO.H** (pg 163), **CLS** (pg 164), **CLS.B** (pg 165), **CLS.H** (pg 165), **CLZ** (pg 166), **CLZ.B** (pg 167), **CLZ.H** (pg 167)

CLO.B Count Leading Ones in Packed Bytes CLO.B

CLO.H Count Leading Ones in Packed Halfwords CLO.H

Syntax:

clo.b	Dc, Da (RR)
clo.h	Dc, Da (RR)

Description:

Count the number of consecutive ones in each byte/halfword of *Da*, starting with the most-significant bit, and put each result in the corresponding byte/halfword of *Dc*.

Refer also to Section 8.1.1.9, "Count Leading Zeroes, Ones, and Signs," on page 91.

Operation:

```
clo.b : D[c][(n+7):n] = #leading_ones(D[a][(n+7):n]); n = 0, 8, 16, 24
clo.h : D[c][(n+15):n] = #leading_ones(D[a][(n+15):n]); n = 0, 16
```

Examples:

```
clo.b    d3, d1
clo.h    d3, d1
```

See Also:

CLO (pg 162), **CLS** (pg 164), **CLS.B** (pg 165), **CLS.H** (pg 165), **CLZ** (pg 166), **CLZ.B** (pg 167), **CLZ.H** (pg 167)

CLS Count Leading Signs CLS

Syntax:

cls Dc, Da (RR)

Description:

Count the number of consecutive bits which have the same value as bit 31 in Da, starting with bit 30, and put the result in *Dc*. The result is the number of leading sign bits minus one, giving the number of redundant sign bits in Da.

Refer also to Section 8.1.1.9, "Count Leading Zeroes, Ones, and Signs," on page 91.

Operation:

D[c] = #leading_signs(D[a]) – 1

Example:

```
cls   d3, d1
```

See Also:

CLO (pg 162), **CLO.B** (pg 163), **CLO.H** (pg 163), **CLS.B** (pg 165), **CLS.H** (pg 165), **CLZ** (pg 166), **CLZ.B** (pg 167), **CLZ.H** (pg 167)

CLS.B Count Leading Signs in Packed Bytes CLS.B

CLS.H Count Leading Signs in Packed Halfwords CLS.H

Syntax:

cls.b	Dc, Da (RR)
cls.h	Dc, Da (RR)

Description:

Count the number of consecutive bits in each byte/halfword in data register Da, which have the same state as the most-significant bit (msb) in that byte/halfword, starting with the next bit right of the msb. Put each result in the corresponding byte/halfword of *Dc*. The results are the number of leading sign bits minus one in each byte/halfword, giving the number of redundant sign bits in the bytes/halfwords of Da.

Refer also to Section 8.1.1.9, "Count Leading Zeroes, Ones, and Signs," on page 91.

Operation:

cls.b : D[c][(n+7):n] = #leading_signs(D[a][(n+7):n]) – 1; n = 0, 8, 16, 24
cls.h : D[c][(n+15):n] = #leading_signs(D[a][(n+15):n]) – 1; n = 0, 16

Examples:

```
cls.b    d3, d1
cls.h    d3, d1
```

See Also:

CLO (pg 162), **CLO.B** (pg 163), **CLO.H** (pg 163), **CLS** (pg 164), **CLZ** (pg 166), **CLZ.B** (pg 167), **CLZ.H** (pg 167)

CLZ Count Leading Zeroes CLZ

Syntax:

clz Dc, Da (RR)

Description:

Count the number of consecutive zeroes in *Da*, starting with bit 31, and put the result in *Dc*.

Refer also to Section 8.1.1.9, "Count Leading Zeroes, Ones, and Signs," on page 91.

Operation:

```
D[c] = #leading_zeroes(D[a])
```

Example:

```
clz   d3, d1
```

See Also:

CLO (pg 162), **CLO.B** (pg 163), **CLO.H** (pg 163), **CLS** (pg 164), **CLS.B** (pg 165), **CLS.H** (pg 165), **CLZ.B** (pg 167), **CLZ.H** (pg 167)

CLZ.B Count Leading Zeroes in Packed Bytes CLZ.B
CLZ.H Count Leading Zeroes in Packed Halfwords CLZ.H

Syntax:

clz.b	Dc, Da (RR)
clz.h	Dc, Da (RR)

Description:

Count the number of consecutive zeroes in each byte/halfword of *Da*, starting with the most-significant bit of each byte/halfword, and put each result in the corresponding byte/halfword of *Dc*.

Refer also to Section 8.1.1.9, "Count Leading Zeroes, Ones, and Signs," on page 91.

Operation:

```
clz.b : D[c][(n+7):n] = #leading_zeroes(D[a][(n+7):n]); n = 0, 8, 16, 24
clz.h : D[c][(n+15):n] = #leading_zeroes(D[a][(n+15):n]); n = 0, 16
```

Examples:

```
clz.b    d3, d1
clz.h    d3, d1
```

See Also:

CLO (pg 162), **CLO.B** (pg 163), **CLO.H** (pg 163), **CLS** (pg 164), **CLS.B** (pg 165), **CLS.H** (pg 165), **CLZ** (pg 166)

CMOV Conditional Move CMOV

Syntax:

```
cmov    Da, Dl5, Db (SRR)
cmov    Da, Dl5, const4 (SRC)
```

Description:

If the contents of data register D15 are non-zero, copy the contents of data register *Db/ const4* to data register *Da*; otherwise, the contents of *Da* is unchanged. The *const4* value is sign-extended to 32 bits.

Operation:

```
D[a] = ((D[15]  != 0) ? D[b] : D[a])
D[a] = ((D[15]  != 0) ? sign_ext(const4) : D[a])
```

Examples:

```
cmov  d1, d15, d2
cmov  d1, d15, 6
```

See Also:

CADD (pg 155), **CADDN** (pg 157), **CMOVN** (pg 169), **CSUB** (pg 170), **CSUBN** (pg 172), **SEL** (pg 347), **SELN** (pg 349)

CMOVN Conditional Move-Not CMOVN

Syntax:

```
cmovn    Da, Dl5, Db (SRR)
cmovn    Da, Dl5, const4 (SRC)
```

Description:

If the contents of data register D15 are zero, copy the contents of data register *Db/const4* to data register *Da*; otherwise, the contents of *Da* is unchanged.The *const4* value is sign-extended to 32 bits.

Operation:

```
D[a] = ((D[15] ==0) ? D[b] : D[a])
D[a] = ((D[15] = =0) ? sign_ext(const4) : D[a])
```

Examples:

```
cmovn d1, d15, d2
cmovn d1, d15, 6
```

See Also:

CADD (pg 155), **CADDN** (pg 157), **CMOV** (pg 168), **CSUB** (pg 170), **CSUBN** (pg 172), **SEL** (pg 347), **SELN** (pg 349)

CSUB Conditional Subtract CSUB

Syntax:

csub Dc, Dd, Da, Db (RRR)

Description:

If the contents of data register *Dd* are non-zero, subtract the contents of data register *Db* from the contents of data register *Da* and put the result in data register *Dc*; otherwise, put the contents of *Da* in *Dc*.

Operation:

D[c] = ((D[d] != 0) ? D[a] – D[b] : D[a])

Status:

V, SV, AV, SAV

Example:

```
csub  d3, d4, d1, d2
```

See Also:

CADD (pg 155), **CADDN** (pg 157), **CMOV** (pg 168), **CMOVN** (pg 169), **CSUBN** (pg 172), **SEL** (pg 347), **SELN** (pg 349)

CSUB.A Conditional Subtract from Address CSUB.A

Syntax:

csub.a Ac, Dd, Aa, Ab (RRR)

Description:

If the contents of data register *Dd* are non-zero, subtract the contents of address register *Ab* from the contents of address register *Aa* and put the result in address register *Ac*; otherwise, put the contents of *Aa* in *Ac*.

Operation:

A[c] = ((D[d] != 0) ? A[a] – A[b] : A[a])

Example:

```
csub.a   a3, d4, a4, a2
```

See Also:

CADD.A (pg 156), **CADDN.A** (pg 158), **CSUBN.A** (pg 173), **SEL.A** (pg 348), **SELN.A** (pg 350)

CSUBN Conditional Subtract-Not CSUBN

Syntax:

csubn Dc, Dd, Da, Db (RRR)

Description:

If the contents of data register *Dd* are zero, subtract the contents of data register *Db/const9* from the contents of data register *Da* and put the result in data register *Dc*; otherwise, put the contents of *Da* in *Dc*.

Operation:

D[c] = ((D[d] == 0) ? D[a] – D[b] : D[a])

Status:

V, SV, AV, SAV

Example:

```
csubn d3, d4, d1, d2
```

See Also:

CADD (pg 155), **CADDN** (pg 157), **CMOV** (pg 168), **CMOVN** (pg 169), **CSUB** (pg 170), **SEL** (pg 347), **SELN** (pg 349)

CSUBN.A Conditional Subtract-Not from Address CSUBN.A

Syntax:

csubn.a Ac, Dd, Aa, Ab (RRR)

Description:

If the contents of data register *Dd* are zero, subtract the contents of address register *Ab* from the contents of address register *Aa* and put the result in address register *Ac*; otherwise, put the contents of *Aa* in *Ac*.

Operation:

A[c] = ((D[d] == 0) ? A[a] – A[b] : A[a])

Example:

```
csubn.a  a3, d4, a4, a2
```

See Also:

CADD.A (pg 156), **CADDN.A** (pg 158), **CSUB.A** (pg 171), **SEL.A** (pg 348), **SELN.A** (pg 350)

DEBUG Debug DEBUG

Syntax:

debug (SYS)

debug (SR)

Description:

If the debug mode is enabled, cause a debug event; otherwise, execute a NOP.

If the debug mode is enabled, cause a debug event; otherwise, execute a NOP.

Example:

```
debug
```

DEXTR Extract from Double Register DEXTR

Syntax:

```
dextr    Dc, Da, Db, Dd (RRRR)
dextr    Dc, Da, Db, p (RRPW)
```

Description:

Extract 32 bits from the register pair *Da*/*Db* (where *Da* contains the most-significant 32 bits of the value) starting at the bit number specified by 63 – *Dd*[4:0]/*p*. Put the result in *Dc*.

Refer also to Section 8.1.1.11, "Bit-Field Extract and Insert," on page 92.

Operation:

```
D[c] = ({D[a], D[b]} << pos)[63:32];
pos = D[d][4:0] / p;
```

Examples:

```
dextr    d1, d3, d5, d7
dextr    d1, d3, d5, 11
```

See Also:

EXTR (pg 191), **EXTR.U** (pg 191), **INSERT** (pg 196), **INS.T** (pg 195), **INSN.T** (pg 195)

DIFSC.A Difference Scaled Address DIFSC.A

Syntax:

difsc.a Dc, Aa, Ab, n (RR)

Description:

Subtract the contents of address register *Ab* from the contents of address register *Aa* and arithmetically right-shift the result by *n*, where *n* is 0, 1, 2, or 3. Put the shifted value in data register *Dc*.

Operation:

D[c] = (A[a] – A[b]) >> n, n = 0, 1, 2, or 3

See Also:

ADD.A (pg 133), **ADDIH** (pg 137), **ADDSC.A** (pg 143), **ADDSC.AT** (pg 143), **SUB.A** (pg 380), **SUBSC.A** (pg 386)

DISABLE Disable Interrupts DISABLE

Syntax:

disable (SYS)

Description:

Disable interrupts by clearing the Interrupt Enable bit (ICR.IE) in the Interrupt Control Register.

Operation:

Refer to Section 8.9.4, "Enabling/Disabling the Interrupt System," on page 111 and Chapter 5, "Interrupt System," on page 59.

Example:

```
disable
```

See Also:

ENABLE (pg 185)

DSYNC — Synchronize Data — DSYNC

Syntax:

dsync (SYS)

Description:

Forces all data accesses to complete before any data accesses associated with an instruction semantically after the DSYNC are initiated. Refer to Section 8.9.2.1, "DSYNC," on page 110 for more information on this synchronization primitive.

Example:

```
dsync
```

See Also:

ISYNC (pg 197)

DVADJ Divide-Adjust DVADJ

Syntax:

```
dvadj    Ec, Ed, Db (RRR)
dvadj    Ea, Db (SRR)
```

Description:

Divide-adjust the contents of extended register Ed, using the value in data register Db, and put the result in extended register Ec. Ed contains the unadjusted quotient and remainder resulting from a sequence of divide-step (DVSTEP) operations, with the quotient in the least-significant word of Ed (data register Dd) and the remainder in the most-significant word of Ed (data register Dd+1). Db contains the divisor that was used to generate the values in Ed. All three values are inspected, and an adjusted quotient and remainder are written to Ec.

Divide-adjust the contents of extended register Ea, using the value in data register Db, and put the result in extended register Ea. Ea contains the unadjusted quotient and remainder resulting from a sequence of divide-step (DVSTEP) operations, with the quotient in the least-significant word of Ea (data register Da) and the remainder in the most-significant word of Ec (data register Da+1). Db contains the divisor that was used to generate the values in Ea. All three values are inspected, and an adjusted quotient and remainder are written to Ea.

Two types of adjustment are performed, as needed. Following a divide-step sequence, the sign of the remainder is always the same as the sign of the original dividend. If the original dividend was negative, and was exactly divisible by the divisor, then the unadjusted remainder will be equal in magnitude to the divisor, and the magnitude of the quotient will be one too small. In that case, the remainder will be set to zero, and the magnitude of the quotient will be increased by one.

Negative quotient and remainder values produced by the divide-step algorithm are developed in 1's complement form. The DVADJ operation converts negative quotient and remainder values to 2's complement representation.

If the quotient and remainder are statically known to be non-negative (the original dividend was non-negative, and the divisor was positive), then the DVADJ operation is not required. This operation is never required following an unsigned divide sequence.

Operation:

```
E[c] = divide_adjust(E[d], D[b])
E[c] = divide_adjust(E[a], D[b])
```

Status:

V, SV

See Also:

DVINIT (pg 181), **DVINIT.B** (pg 181), **DVINIT.BU** (pg 181), **DVINIT.H** (pg 181), **DVINIT.HU** (pg 181), **DVINIT.U** (pg 181), **DVSTEP** (pg 183), **DVSTEP.U** (pg 183)

DVINIT	**Divide-Initialization Word**	**DVINIT**
DVINIT.U	**Divide-Initialization Word Unsigned**	**DVINIT.U**
DVINIT.B	**Divide-Initialization Byte**	**DVINIT.B**
DVINIT.BU	**Divide-Initialization Byte Unsigned**	**DVINIT.BU**
DVINIT.H	**Divide-Initialization Halfword**	**DVINIT.H**
DVINIT.HU	**Divide-Initialization Halfword Unsigned**	**DVINIT.HU**

Syntax:

```
dvinit     Ec, Da, Db (RR)
dvinit.u   Ec, Da, Db (RR)
dvinit.b   Ec, Da, Db (RR)
dvinit.bu  Ec, Da, Db (RR)
dvinit.h   Ec, Da, Db (RR)
dvinit.hu  Ec, Da, Db (RR)
```

Description:

Sign-extend (DVINT, DVINIT.B, DVINIT.H) or zero-extend (DVINIT.U, DVINIT.BU, DVINIT.HU) to 64 bits and left-shift the contents of data register Da, and put the result in extended register Ec. The shift amount depends on the expected size of the quotient: for DVINIT and DVINIT.U, the shift amount is zero, for DVINIT.H and DVINIT.HU it is 16, and for DVINIT.B and DVINIT.BU it is 24. The vacated bits are filled with the sign bit of the quotient. Overflow occurs if the magnitude of the partial remainder in the most-significant word of Ec is greater than or equal to the magnitude of the divisor, in register Db.

When the shift amount is nonzero, this instruction performs the same operation as a divide initialization with no shift amount (DVINIT or DVINIT.U) followed by two or three divide-step instructions (DVSTEP). The shifting is effectively substituting for an initial group of divide-step instructions, which would be expected to develop quotient bits that were exclusively copies of the quotient sign bit.

Operation:

```
E[c] = divide_init(D[a], D[b])
E[c] = divide_init_u(D[a], D[b])

E[c] = divide_init_b(D[a], D[b])
E[c] = divide_init_b_u(D[a], D[b])

E[c] = divide_init_h(D[a], D[b])
E[c] = divide_init_h_u(D[a], D[b])
```

Status:

V, SV

See Also:

DVINIT (pg 181), **DVINIT.B** (pg 181), **DVINIT.BU** (pg 181), **DVINIT.H** (pg 181), **DVINIT.HU** (pg 181), **DVINIT.U** (pg 181), **DVSTEP** (pg 183), **DVSTEP.U** (pg 183)

DVSTEP Divide-Step DVSTEP
DVSTEP.U Divide-Step Unsigned DVSTEP.U

Syntax:

dvstep	Ec, Ed, Db (RRR)
dvstep.u	Ec, Ed, Db (RRR)
dvstep	Ea, Db (SRR)
dvstep.u	Ea, Db (SRR)

Description:

Divide the contents of extended register *Ed*, 8 bits at a time, by data register *Db*, and put the result in extended register *Ec*. *Ed* contains the result of a previous divide-initialization (DVINIT) or divide-step (DVSTEP) instruction. *Db* contains the divisor for the current divide operation.

Divide the contents of extended register *Ea*, 8 bits at a time, by data register *Db*, and put the result in extended register *Ea*. Before the operation, *Ea* contains the result of a previous divide-initialization (DVINIT) or divide-step (DVSTEP) instruction. *Db* contains the divisor for the current divide operation.

The most-significant word of *Ed* or *Ea* (data register *D[d+1]* or *D[a+1]*) contains the 32-bit partial remainder for the divide operation, up to the current point in the divide-step sequence. The least-significant word of *Ed* or *Ea* (data register *Dd* or *Da*) contains a mix of unprocessed bits from the dividend and quotient bits developed up to this point. The unprocessed dividend bits occupy the most-significant bit positions of *Dd* or *Da*, while the quotient bits occupy the least-significant bits. The total of the two bit sets is always 32 bits, but the boundary between them depends on the current instruction's position within the divide sequence.

Each divide-step instruction processes eight additional dividend bits, and develops eight additional bits of quotient. A divide operation yielding a 32-bit quotient value requires four divide-step instructions, optionally terminated by a divide-adjust (refer to the description of the DVADJ instruction). A divide operation yielding a halfword quotient requires two divide-step instructions, while a divide operation yielding an 8-bit quotient requires only one divide-step instruction. All cases also require the appropriate divide-initialization instruction, and may require a terminating divide adjust.

The unsigned divide-step instructions treat the dividend and partial remainders as unsigned, 32-bit values and develops positive quotients. An unsigned divide operation does not require a terminating DVADJ instruction. The signed divide-step instructions treat the dividend and partial remainders as signed values, and normally require terminating DVADJ instructions. The terminating DVADJ may be omitted, however, if the original dividend and the divisor are known to be non-negative.

Operation:

```
E[c] = divide_step(E[d], D[b])
E[c] = divide_step_u(E[d], D[b])
E[c] = divide_step(E[a], D[b])
E[c] = divide_step_u(E[a], D[b])
```

See Also:

DVADJ (pg 179), **DVINIT** (pg 181), **DVINIT.B** (pg 181), **DVINIT.BU** (pg 181), **DVINIT.H** (pg 181), **DVINIT.HU** (pg 181), **DVINIT.U** (pg 181)

ENABLE Enable Interrupts ENABLE

Syntax:

enable (SYS)

Description:

Enable interrupts by setting the Interrupt Enable bit (ICR.IE) in the Interrupt Control Register to 1.

Operation:

Refer to Section 8.9.4, "Enabling/Disabling the Interrupt System," on page 111 and Chapter 5, "Interrupt System," on page 59.

Example:

```
enable
```

See Also:

DISABLE (pg 177)

EQ Equal EQ

Syntax:

eq	Dc, Da, Db (RR)
eq	Dc, Da, const9 (RC)
eq	D15, Da, Db (SRR)
eq	D15, Da, const4 (SRC)

Description:

If the contents of data register *Da* are equal to the contents of data register *Db/const9*, set the least-significant bit of *Dc* to 1 and clear the remaining bits to zero; otherwise, clear all bits in *Dc*. The *const9* value is sign-extended to 32 bits.

If the contents of data register *Da* are equal to the contents of data register *Db/const4*, set the least-significant bit of D15 to 1 and clear the remaining bits to zero; otherwise, clear all bits in D15. The *const4* value is sign-extended to 32 bits.

Refer also to Section 8.2, "Compare Instructions," on page 97.

Operation:

```
D[c] = (D[a] == D[b])
D[c] = (D[a] == sign_ext(const9))
D[15] = (D[a] == D[b])
D[15] = (D[a] == sign_ext(const4))
```

Examples:

```
eq    d3, d1, d2
eq    d3, d1, 126
eq    d3, d1 253
eq    d15, d1, d2
eq    d15, d1, 6
eq    d15, d1 253
```

See Also:

GE (pg 192), **GE.U** (pg 192), **LT** (pg 238), **LT.U** (pg 238), **NE** (pg 321)

EQ.A — Equal to Address — EQ.A

Syntax:

eq.a Dc, Aa, Ab (RR)

Description:

If the contents of address registers *Aa* and *Ab* are equal, set the least-significant bit of *Dc* to 1 and clear the remaining bits to zero; otherwise, clear all bits in *Dc*.

Operation:

D[c] = (A[a] == A[b])

Example:

```
eq.a d3, a4, a2
```

See Also:

EQZ.A (pg 190), **GE.A** (pg 193), **LT.A** (pg 240), **NE.A** (pg 322), **NEZ.A** (pg 323)

EQ.B — Equal Packed Byte — EQ.B
EQ.H — Equal Packed Halfword — EQ.H
EQ.W — Equal Packed Word — EQ.W

Syntax:

```
eq.b    Dc, Da, Db (RR)
eq.h    Dc, Da, Db (RR)
eq.w    Dc, Da, Db (RR)
```

Description:

Compare each byte/halfword/word of *Da* with the corresponding byte/halfword/word of *Db*. In each case, if the two are equal, set the corresponding byte/halfword/word of *Dc* to all 1's; otherwise, set the corresponding byte/halfword/word of *Dc* to all 0's. Refer also to Section 8.1.3, "Packed Arithmetic," on page 95.

Operation:

```
if (D[a][(n+7):n] == D[b][(n+7):n])
then D[c][(n+7):n] = 8'h FF
else D[c][(n+7):n] = 8'h 00; n = 0, 8, 16, 24

if (D[a][(n+15):n] == D[b][(n+15):n])
then D[c][(n+15):n] = 16'h FFFF
else D[c][(n+15):n] = 16'h 0000; n = 0, 16

if (D[a] == D[b])
then D[c] = 32'h FFFFFFFF
else D[c] = 32'h 00000000
```

Examples:

```
eq.b  d3, d1, d2
eq.h  d3, d1, d2
eq.w  d3, d1, d2
```

See Also:

LT.B (pg 241), **LT.BU** (pg 241), **LT.H** (pg 242), **LT.HU** (pg 242), **LT.W** (pg 243), **LT.WU** (pg 243)

EQANY.B — Equal Any Byte — EQANY.B
EQANY.H — Equal Any Halfword — EQANY.H

Syntax:

```
eqany.b   Dc, Da, Db (RR)
eqany.b   Dc, Da, const9 (RC)
eqany.h   Dc, Da, Db (RR)
eqany.h   Dc, Da, const9 (RC)
```

Description:

Compare each byte/halfword of *Da* with the corresponding byte/halfword of *Db/const9*. If the logical OR of the boolean results from each comparison is TRUE, set the least-significant bit of *Dc* to 1 and clear the remaining bits to zero; otherwise, clear all bits in *Dc*.

Refer also to Section 8.2, "Compare Instructions," on page 97.

Operation:

```
D[c] = (D[a][31:24] == D[b][31:24])
or (D[a][23:16] == D[b][23:16])
or (D[a][15:8]  == D[b][15:8])
or (D[a][7:0] == D[b][7:0])
```

```
D[c] = (D[a][31:24] == sign_ext(const9)[31:24])
or (D[a][23:16] == sign_ext(const9)[23:16])
or (D[a][15:8] == sign_ext(const9)[15:8])
or (D[a][7:0] == sign_ext(const9)[7:0])
```

Examples:

```
eqany.b d3, d1, d2
eqany.b d3, d1, 126
eqany.h d3, d1, d2
eqany.h d3, d1, 126
```

See Also:

EQ (pg 186), **GE** (pg 192), **GE.U** (pg 192), **LT** (pg 238), **LT.U** (pg 238), **NE** (pg 321)

EQZ.A Equal Zero Address EQZ.A

Syntax:

eqz.a Dc, Aa (RR)

Description:

If the contents of address register *Aa* are equal to zero, set the least significant bit of *Dc* to 1 and clear the remaining bits to zero; otherwise, clear all bits in *Dc*.

Operation:

D[c] = (A[a] == 0)

Example:

```
eqz.a d3, a4
```

See Also:

EQ.A (pg 187), **GE.A** (pg 193), **LT.A** (pg 240), **NE.A** (pg 322), **NEZ.A** (pg 323)

EXTR — Extract Bit Field — EXTR
EXTR.U — Extract Bit Field Unsigned — EXTR.U

Syntax:

extr	Dc, Da, Ed (RRRR)
extr	Dc, Da, Dd, w (RRRW)
extr	Dc, Da, p, w (RRPW)
extr.u	Dc, Da, Ed (RRRR)
extr.u	Dc, Da, Dd, w (RRRW)
extr.u	Dc, Da, p, w (RRPW)

Description:

Extract from *Da* the number of consecutive bits specified by *Ed(upper)*/*w*, starting at the bit number specified by *Ed(lower)/Dd*/p, and put the result, sign-extended (extr) or zero-extended (extr.u) to 32 bits, in *Dc*.

Refer also to Section 8.1.1.11, "Bit-Field Extract and Insert," on page 92.

Operation:

extr : D[c] = sign_ext((D[a] >> pos) and (2^{width}–1));
pos = E[d](lower)[4:0] / D[d][4:0]/p;
width = E[d](upper)[4:0] / w

extr.u : D[c] = zero_ext((D[a] >> pos) and (2^{width}–1))
pos = E[d](lower)[4:0] / D[d][4:0]/p;
width = E[d](upper)[4:0] / w

Examples:

```
extr      d3, d1, e4
extr.u    d3, d1, d4, 12
```

See Also:

DEXTR (pg 175), **INSERT** (pg 196), **INS.T** (pg 195), **INSNT.T** (pg 195)

GE — Greater Than or Equal — GE
GE.U — Greater Than or Equal Unsigned — GE.U

Syntax:

ge	Dc, Da, Db (RR)
ge	Dc, Da, const9 (RC)
ge.u	Dc, Da, Db (RR)
ge.u	Dc, Da, const9 (RC)

Description:

If the contents of data register *Da* are greater than or equal to the contents of data register *Db*/*const9*, set the least-significant bit of *Dc* to 1 and clear the remaining bits to zero; otherwise, clear all bits in *Dc*. *Da* and *Db* are treated as 32-bit signed integers, and the *const9* value is sign-extended to 32 bits.

If the contents of data register *Da* are greater than or equal to the contents of data register *Db*/*const9*, set the least-significant bit of *Dc* to 1 and clear the remaining bits to zero; otherwise, clear all bits in *Dc*. *Da* and *Db* are treated as 32-bit unsigned integers, and the *const9* value is zero-extended to 32 bits.

Refer also to Section 8.2, "Compare Instructions," on page 97.

Operation:

```
D[c] = (D[a] >= D[b]); signed
D[c] = (D[a] >= sign_ext(const9)); signed

D[c] = (D[a] >= D[b]); unsigned
D[c] = (D[a] >= zero_ext(const9)); unsigned
```

Examples:

```
ge d3, d1, d2
ge d3, d1, 126
ge.u d3, d1, d2
ge.u d3, d1, 126
```

See Also:

EQ (pg 186), **LT** (pg 238), **LT.U** (pg 238), **NE** (pg 321)

GE.A Greater Than or Equal Address GE.A

Syntax:

ge.a Dc, Aa, Ab (RR)

Description:

If the contents of address register *Aa* are greater than or equal to the contents of address register *Ab*, set the least-significant bit of *Dc* to 1 and clear the remaining bits to zero; otherwise, clear all bits in *Dc*. The operands are treated as unsigned 32-bit integers.

Operation:

D[c] = (A[a] >= A[b]); unsigned

Example:

```
ge.a  d3, a4, a2
```

See Also:

EQ.A (pg 187), **EQZ.A** (pg 190), **LT.A** (pg 240), **NE.A** (pg 322), **NEZ.A** (pg 323)

IMASK Insert Mask IMASK

Syntax:

imask	Ec, Db, Dd, w (RRRW)
imask	Ec, Db, p, w (RRPW)
imask	Ec, const4, Dd, w (RCRW)
imask	Ec, const4, p, w (RCPW)

Description:

Create a mask containing the number of bits specified by *w*, starting at the bit number specified by *Dd*[4:0]/*p*, and put the mask in data register *Ec(upper)*. Left-shift the value in *Db*/*const4* by the amount specified by *Dd*[4:0]/*p* and put the result value in *Ec(lower)*. The value *const4* is zero-extended to 32 bits. This mask and value can be used by the Load-Modify-Store (LDMST) instruction to write a specified bit field to a location in memory.

Refer also to Section 8.7.3, "Store Bit and Bit Field," on page 108.

Operation:

```
Ec(upper) = ((2^w–1) << pos);
Ec(lower) = (D[b] << pos);
pos = D[d][4:0] / p;
zero_ext(const4) may replace D[b]
```

Examples:

```
imask    e2, d1, d2, 11
imask    e2, d1, 5, 11
imask    e2, 6, d2, 11
imask    e2, 6, 5, 11
```

See Also:

LDMDST (pg 234), **ST.T** (pg 374)

INS.T — Insert Bit — INS.T
INSN.T — Insert Bit-Not — INSN.T

Syntax:

ins.t	Dc, Da, p1, Db, p2 (BIT)
insn.t	Dc, Da, p1, Db, p2 (BIT)

Description:

Move the value of Da, with bit *p1* of this value replaced with bit *p2* of register *Db,* to *Dc.*

Move the value of Da, with bit *p1* of this value replaced with the inverse of bit *p2* of register *Db,* to *Dc.*

Operation:

```
D[c] = {D[a][31:(p1+1)], D[b][p2], D[a][(p1−1):0]}
D[c] = {D[a][31:(p1+1)], !D[b][p2], D[a][(p1−1):0]}
```

Examples:

```
ins.t    d3, d1, 5, d2, 7
insn.t   d3, d1, 5, d2, 7
```

See Also:

DEXTR (pg 175), **EXTR** (pg 191), **EXTR.U** (pg 191), **INSERT** (pg 196)

INSERT Insert Bit Field INSERT

Syntax:

insert	Dc, Da, Db, Ed (RRRR)
insert	Dc, Da, Db, Dd, w (RRRW)
insert	Dc, Da, Db, p, w (RRPW)
insert	Dc, Da, const4, Ed (RCRR)
insert	Dc, Da, const4, Dd, w (RCRW)
insert	Dc, Da, const4, p, w (RCPW)

Description:

Extract from *Db/const4* the number of consecutive bits specified by *Ed(upper)/w*, starting at the bit number specified by *Ed(lower)/Dd/p*; extract from *Da* all bits not included in the bits specified by *Ed(upper)/w and Ed(lower)/Dd/p*. Put the logical OR of the two extracted words in *Dc*.

Refer also to Section 8.1.1.11, "Bit-Field Extract and Insert," on page 92.

Operation:

```
D[c] = (D[a] and !m) or (D[b] and m);
m = (2^width–1) << pos;
pos = E[d](lower)[4:0] / D[d][4:0]/p;
width = E[d](upper)[4:0] / w
zero_ext(const4) may replace D[b]
```

Example:

```
insert   d3, d1, d2, e4
```

See Also:

DEXTR (pg 175), **EXTR** (pg 191), **EXTR.U** (pg 191), **INS.T** (pg 195), **INSN.T** (pg 195)

ISYNC Synchronize Instructions ISYNC

Syntax:

isync (SYS)

Description:

Forces completion of all previous instructions, then flushes the CPU pipelines, and invalidates any cached state before proceeding to the next instruction. Refer to Section 8.9.2.2, "ISYNC," on page 110 for more information on this synchronization primitive.

Example:

```
isync
```

See Also:

DSYNC (pg 178)

J Jump Unconditional J

Syntax:

j	disp24 (B)
j	disp8 (SB)

Description:

Add the value specified by *disp24*, multiplied by two and sign-extended to 32 bits, to the contents of the PC, and jump to that address.

Refer also to the description in Section 8.6.1, "Unconditional Branch," on page 103.

Operation:

```
PC = PC + sign_ext(2 * disp24)
PC = PC + sign_ext(2 * disp8)
```

Example:

```
j  foobar
```

See Also:

JA (pg 199), **JI** (pg 205), **JL** (pg 206), **JLA** (pg 207), **JLI** (pg 209), **LOOP** (pg 237)

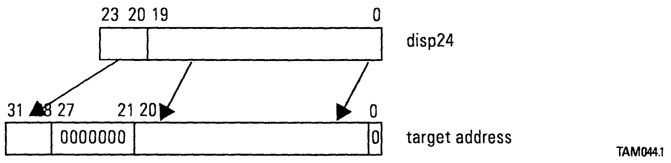
JA Jump Unconditional Absolute JA

Syntax:

ja disp24 (B)

Description:

Load the value specified by *disp24* into the PC and jump to that address. The value *disp24* is used to form the effective address as shown below:



Refer also to the description in Section 8.6.1, "Unconditional Branch," on page 103.

Operation:

PC = {disp24[23:20], 0000000, disp24[19:0], 1'h 0}

Example:

```
ja foobar
```

See Also:

JI (pg 205), **JL** (pg 206), **JLA** (pg 207), **JLI** (pg 209)

JEQ Jump if Equal JEQ

Syntax:

```
jeq     Da, Db, disp15 (BRR)
jeq     Da, const4, disp15 (BRC)
jeq     D15, Db, disp4 (SBR)
jeq     D15, const4, disp4 (SBC)
```

Description:

If the contents of *Da* are equal to the contents *Db/const4*, then add the value specified by *disp15*, multiplied by two and sign-extended to 32 bits, to the contents of the PC, and jump to that address. The *const4* value is sign-extended to 32 bits.

If the contents of D15 are equal to the contents *Db/const4*, then add the value specified by *disp4*, multiplied by two and zero-extended to 32 bits, to the contents of the PC, and jump to that address. The *const4* value is sign-extended to 32 bits.

Operation:

```
if (D[a]==D[b]) then (PC = PC+sign_ext(2 * disp15))
if (D[a]==sign_ext(const4)) then (PC = PC+sign_ext(2 * disp15))
if (D[15] ==D[b]) then (PC = PC+zero_ext(2 * disp4))
if (D[15] ==sign_ext(const4)) then (PC = PC+zero_ext(2 * disp4))
```

Examples:

```
jeq   d1, d2, foobar
jeq   d1, 6, foobar
jeq   d15, d2, foobar
```

See Also:

JEQ.A (pg 201), **JGTZ** (pg 204), **JLT** (pg 210), **JLT.U** (pg 210), **JNE** (pg 212), **JNE.A** (pg 213)

JEQ.A Jump if Equal Address JEQ.A

Syntax:

jeq.a Aa, Ab, disp15 (BRR)

Description:

If the contents of *Aa* are equal to the contents *Ab*, then add the value specified by *disp15*, multiplied by two and sign-extended to 32 bits, to the contents of the PC, and jump to that address.

Operation:

```
if (A[a]==A[b]) then (PC = PC+sign_ext(2 * disp15))
```

Example:

```
jeq.a a4, a2, foobar
```

See Also:

JEQ (pg 200), **JGTZ** (pg 204), **JLT** (pg 210), **JLT.U** (pg 210), **JNE** (pg 212), **JNE.A** (pg 213)

JGE — Jump if Greater Than or Equal — JGE
JGE.U — Jump if Greater Than or Equal Unsigned — JGE.U

Syntax:

jge	Da, Db, disp15 (BRR)
jge	Da, const4, disp15 (BRC)
jge.u	Da, Db, disp15 (BRR)
jge.u	Da, const4, disp15 (BRC)

Description:

If the contents of *Da* are greater than or equal to the contents *Db/const4*, then add the value specified by *disp15*, multiplied by two and sign-extended to 32 bits, to the contents of the PC, and jump to that address. The *const4* value is sign-extended/zero-extended to 32 bits.

Operation:

```
if (D[a]>=D[b]) then (PC = PC+sign_ext(2 * disp15)); signed
if (D[a]>=sign_ext(const4)) then (PC = PC+sign_ext(2 * disp15)); signed

if (D[a]>=D[b]) then (PC = PC+sign_ext(2 * disp15)); unsigned
if (D[a]>=zero_ext(const4)) then (PC = PC+sign_ext(2 * disp15)); unsigned
```

Examples:

```
jge   d1, d2, foobar
jge   d1, 6, foobar
jge.u d1, d2, foobar
jge.u d1, 6, foobar
```

See Also:

JEQ (pg 200), **JEQ.A** (pg 201), **JGTZ** (pg 204), **JLT** (pg 210), **JLT.U** (pg 210)

JGEZ Jump If Greater Than or Equal to Zero JGEZ

Syntax:

```
jgez    Db, disp4 (SBR)
```

Description:

If the contents of *Db* are greater than or equal to zero, then add the value specified by *disp4*, multiplied by two and zero-extended to 32 bits, to the contents of the PC, and jump to that address.

Operation:

```
if (D[b] >= 0) then (PC = PC + zero_ext(2 * disp4))
```

Example:

```
jgez  d2, foobar
```

See Also:

JEQ (pg 200), **JGTZ** (pg 204), **JLEZ** (pg 208), **JLTZ** (pg 211), **JNZ** (pg 216), **JZ** (pg 219)

JGTZ Jump if Greater Than Zero JGTZ

Syntax:

jgtz Db, disp4 (SBR)

Description:

If the contents of *Db* are greater than zero, then add the value specified by *disp4*, multiplied by two and zero-extended to 32 bits, to the contents of the PC, and jump to that address.

Operation:

```
if (D[b] > 0) then (PC = PC + zero_ext(2 * disp4))
```

Example:

```
jgtz  d2, foobar
```

See Also:

JEQ (pg 200), **JGEZ** (pg 203), **JLEZ** (pg 208), **JLTZ** (pg 211), **JNZ** (pg 216), **JZ** (pg 219)

JI Jump Indirect JI

Syntax:

```
ji    Ab (RR)
ji    Ab (SBR)
```

Description:

Load the contents of address register *Ab* into the PC and jump to that address. The least-significant bit is always set to 0.

Load the contents of address register *Ab* into the PC and jump to that address. The least-significant bit is always set to 0.

Refer also to the description in Section 8.6.1, "Unconditional Branch," on page 103.

Operation:

PC = {A[b][31:1], 1'h 0}

PC = {A[b][31:1], 1'h 0}

Example:

```
ji    a2
```

See Also:

J (pg 198), **JA** (pg 199), **JL** (pg 206), **JLA** (pg 207), **JLI** (pg 209)

JL — Jump and Link — JL

Syntax:

jl disp24 (B)

Description:

Store the address of the next instruction in A15. Then add the value specified by *disp24*, scaled by two and sign-extended to 32 bits, to the contents of the PC, and jump to that address.

Operation:

A[15] = PC + 4; PC = PC + sign_ext(2 * disp24);

Example:

```
jl    foobar
```

See Also:

J (pg 198), **JI** (pg 205), **JA** (pg 199), **JLA** (pg 207), **JLI** (pg 209)

JLA Jump and Link Absolute JLA

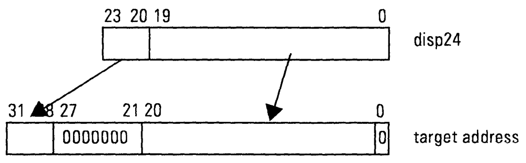
Syntax:

jla disp24 (B)

Description:

Store the address of the next instruction in A15. Then load the value specified by *disp24* into the PC and jump to that address. The value *disp24* is used to form the effective address as shown below.

Refer also to the description in Section 8.6.1, "Unconditional Branch," on page 103.



TAM045.1

Operation:

A[15] = PC + 4; PC = {disp24[23:20], 0000000, disp24[19:0], 1'h 0};

Example:

```
jla   foobar
```

See Also:

JI (pg 205), **JA** (pg 199), **JL** (pg 206), **JLI** (pg 209)

JLEZ Jump If Less Than or Equal to Zero JLEZ

Syntax:

jlez Db, disp4 (SBR)

Description:

If the contents of *Db* are less than or equal to zero, then add the value specified by *disp4*, multiplied by two and zero-extended to 32 bits, to the contents of the PC, and jump to that address.

Operation:

```
if (D[b] <= 0) then (PC = PC + zero_ext(2 * disp4))
```

Example:

```
jlez  d2, foobar
```

See Also:

JGE (pg 202), **JGEZ** (pg 203), **JGTZ** (pg 204), **JLTZ** (pg 211), **JNZ** (pg 216), **JZ** (pg 219)

JLI — Jump and Link Indirect — JLI

Syntax:

jli Ab (RR)

Description:

Store the address of the next instruction in A15. Then load the contents of address register *Ab* into the PC and jump to that address. The least-significant bit is set to 0.

Refer also to the description in Section 8.6.1, "Unconditional Branch," on page 103.

Operation:

A[15] = PC + 4; PC = {A[b][31:1], 1'h 0}

Example:

```
jli   a2
```

See Also:

J (pg 198), **JI** (pg 205), **JA** (pg 199), **JL** (pg 206), **JLA** (pg 207)

JLT — Jump if Less Than — JLT
JLT.U — Jump if Less Than Unsigned — JLT.U

Syntax:

jlt	Da, Db, disp15 (BRR)
jlt	Da, const4, disp15 (BRC)
jlt.u	Da, Db, disp15 (BRR)
jlt.u	Da, const4, disp15 (BRC)

Description:

If the contents of *Da* are less than the contents *Db/const4*, then add the value specified by *disp15*, multiplied by two and sign-extended to 32 bits, to the contents of the PC, and jump to that address. The *const4* value is sign-extended/zero-extended to 32 bits.

Operation:

```
if (D[a]<D[b]) then (PC = PC+sign_ext(2 * disp15)); signed
if (D[a]<sign_ext(const4)) then (PC = PC+sign_ext(2 * disp15)); signed

if (D[a]<D[b]) then (PC = PC+sign_ext(2 * disp15)); unsigned
if (D[a]<zero_ext(const4)) then (PC = PC+sign_ext(2 * disp15)); unsigned
```

Examples:

```
jlt   d1, d2, foobar
jlt   d1, 6, foobar
jlt.u d1, d2, foobar
jlt.u d1, 6, foobar
```

See Also:

JGE (pg 202), **JGEZ** (pg 203), **JGTZ** (pg 204), **JLEZ** (pg 208), **JLTZ** (pg 211), **JNZ** (pg 216), **JZ** (pg 219)

JLTZ Jump If Less Than Zero JLTZ

Syntax:

jltz Db, disp4 (SBR)

Description:

If the contents of *Db* are less than zero, then add the value specified by *disp4*, multiplied by two and zero-extended to 32 bits, to the contents of the PC, and jump to that address.

Operation:

```
if (D[b] < 0) then (PC = PC + zero_ext(2 * disp4))
```

Example:

```
jltz  d2, foobar
```

See Also:

JGEZ (pg 203), **JGTZ** (pg 204), **JLEZ** (pg 208), **JLT** (pg 210), **JLT.U** (pg 210), **JNZ** (pg 216), **JZ** (pg 219)

JNE Jump If Not Equal JNE

Syntax:

jne	Da, Db, disp15 (BRR)
jne	Da, const4, disp15 (BRC)
jne	D15, Db, disp4 (SBR)
jne	D15, const4, disp4 (SBC)

Description:

If the contents of *Da* are not equal to the contents *Db/const4*, then add the value specified by *disp15*, multiplied by two and sign-extended to 32 bits, to the contents of the PC, and jump to that address. The *const4* value is sign-extended to 32 bits.

If the contents of D15 are not equal to the contents *Db/const4*, then add the value specified by *disp4*, multiplied by two and sign-extended to 32 bits, to the contents of the PC, and jump to that address. The *const4* value is sign-extended to 32 bits.

Operation:

```
if (D[a] !=D[b]) then (PC = PC+sign_ext(2 * disp15))
if (D[a] !=sign_ext(const4)) then (PC = PC+sign_ext(2 * disp15))
if (D[15] != D[b]) then (PC = PC+zero_ext(2 * disp4))
if (D[15] != sign_ext(const4)) then (PC = PC+zero_ext(2 * disp4))
```

Examples:

```
jne   d1, d2, foobar
jne   d1, 6, foobar
jne   d15, d2, foobar
jne   d15, 6, foobar
```

See Also:

JEQ (pg 200), **JEQ.A** (pg 201), **JNED** (pg 214), **JNEI** (pg 215)

JNE.A Jump if Not Equal Address JNE.A

Syntax:

jne.a Aa, Ab, disp15 (BRR)

Description:

If the contents of *Aa* are not equal to the contents *Ab*, then add the value specified by *disp15*, multiplied by two and sign-extended to 32 bits, to the contents of the PC, and jump to that address.

Operation:

```
if (A[a] != A[b]) then (PC = PC+sign_ext(2 * disp15))
```

Example:

```
jne.a a4, a2, foobar
```

See Also:

JEQ (pg 200), **JEQ.A** (pg 201), **JNE** (pg 212), **JNED** (pg 214), **JNEI** (pg 215)

JNED Jump If Not Equal And Decrement JNED

Syntax:

jned	Da, Db, disp15 (BRR)
jned	Da, const4, disp15 (BRC)

Description:

If the contents of *Da* are not equal to the contents *Db/const4*, then add the value specified by *disp15*, multiplied by two and sign-extended to 32 bits, to the contents of the PC, and jump to that address. Decrement the value in *Da* by 1. The *const4* value is sign-extended to 32 bits.

Operation:

```
if (D[a] != D[b]) then PC = PC+sign_ext(2 * disp15); D[a] = D[a]–1
if (D[a] != sign_ext(const4)) then PC = PC+sign_ext(2 * disp15); D[a] = D[a]–1
```

Examples:

```
jned d1, d2, foobar
jned d1, 6, foobar
```

See Also:

JEQ (pg 200), **JEQ.A** (pg 201), **JNEI** (pg 215), **JNE.A** (pg 213), **LOOP** (pg 237)

JNEI Jump If Not Equal And Increment JNEI

Syntax:

jnei	Da, Db, disp15 (BRR)
jnei	Da, const4, disp15 (BRC)

Description:

If the contents of *Da* are not equal to the contents *Db/const4*, then add the value specified by *disp15*, multiplied by two and sign-extended to 32 bits, to the contents of the PC, and jump to that address. Increment the value in *Da* by 1. The *const4* value is sign-extended to 32 bits.

Operation:

```
if (D[a] != D[b]) then PC = PC+ sign_ext(2 * disp15); D[a] = D[a]+1
if (D[a] != sign_ext(const4)) then PC = PC + sign_ext(2 * disp15); D[a] = D[a]+1
```

Examples:

```
jnei  d1, d2, foobar
jnei  d1, 6, foobar
```

See Also:

JEQ (pg 200), **JEQ.A** (pg 201), **JNE** (pg 212), **JNE.A** (pg 213), **JNED** (pg 214), **LOOP** (pg 237)

JNZ Jump if Not Equal to Zero JNZ

Syntax:

```
jnz     Db, disp4 (SBR)
jnz     D15, disp8 (SB)
```

Description:

If the contents of *Db* are not equal to zero, then add the value specified by *disp4*, multiplied by two and zero-extended to 32 bits, to the contents of the PC, and jump to that address.

Operation:

```
if (Db != 0) then (PC = PC + zero_ext(2 * disp4))
```

Examples:

```
jnz   d2, foobar
jnz   d15, foobar
```

See Also:

JNZ.A (pg 217), **JNZ.T** (pg 218), **JZ** (pg 219), **JZ.T** (pg 221), **JZ.A** (pg 220)

JNZ.A Jump if Not Equal to Zero Address JNZ.A

Syntax:

jnz.a Aa, disp15 (BRR)
jnz.a Aa, disp4 (SBR)

Description:

If the contents of *Aa* are not equal to zero, then add the value specified by *disp15*, multiplied by two and sign-extended to 32 bits, to the contents of the PC, and jump to that address. PC.

If the contents of *Aa* are not equal to zero, then add the value specified by *disp4*, multiplied by two and zero-extended to 32 bits, to the contents of the PC, and jump to that address.

Operation:

```
if (A[b] != 0) then (PC = PC + sign_ext(2 * disp15))
if (A[b] != 0) then (PC = PC + zero_ext(2 * disp4))
```

Example:

```
jnz.a a4, foobar
```

See Also:

JNZ (pg 216), **JNZ.T** (pg 218), **JZ** (pg 219), **JZ.T** (pg 221), **JZ.A** (pg 220)

JNZ.T Jump if Not Equal to Zero Bit JNZ.T

Syntax:

jnz.t	Da, n, disp15 (BRN)
jnz.t	d15, n, disp4 (SBRN)

Description:

If bit *n* of register *Da* is not equal to zero, then add the value specified by *disp15*, multiplied by two and sign-extended/zero-extended to 32 bits, to the contents of the PC and jump to that address.

Refer also to Section 8.3, "Bit Operations," on page 100.

Operation:

```
if (D[a][n]) then (PC = PC + sign_ext(2 * disp15)); n = 0 – 31
if (D[15] [n]) then PC = PC + zero_ext(2* disp4); n = 0 – 31
```

Example:

```
jnz.t d1, n, foobar
```

See Also:

JNZ.A (pg 217), **JZ.T** (pg 221), **JZ.A** (pg 220)

JZ Jump if Zero JZ

Syntax:

```
jz      D15, disp8 (SB)
jz      Db, disp4 (SBR)
```

Description:

If the contents of D15/*Db* are equal to zero, then add the value specified by *disp8/disp4*, multiplied by two and sign-extended/zero-extended to 32 bits, to the contents of the PC, and jump to that address.

Operation:

```
if (D[15] == 0) then (PC = PC + sign_ext(2 * disp8))
if (D[b] == 0) then (PC = PC + zero_ext(2 * disp4))
```

Examples:

```
jz    d15, foobar
jz    d2, foobar
```

See Also:

JNZ.A (pg 217), **JNZ.T** (pg 218), **JZ.A** (pg 220)

JZ.A Jump if Zero Address JZ.A

Syntax:

jz.a	Aa, disp15 (BRR)
jz.a	Ab, disp4 (SBR)

Description:

If the contents of *Aa* are equal to zero, then add the value specified by *disp15*, multiplied by two and sign-extended to 32 bits, to the contents of the PC and jump to that address.

If the contents of *Ab* are equal to zero, then add the value specified by *disp4*, multiplied by two and zero-extended to 32 bits, to the contents of the PC and jump to that address.

Operation:

```
if (A[a] == 0) then (PC = PC + sign_ext(2 * disp15))
if (A[b] == 0) then (PC = PC + zero_ext(2 * disp4)
```

Examples:

```
jz.a  a4, foobar
jz.a  a2, foobar
```

See Also:

JNZ (pg 216), **JNZ.A** (pg 217), **JNZ.T** (pg 218), **JZ.T** (pg 221)

JZ.T Jump if Zero Bit JZ.T

Syntax:

jz.t	Da, n, disp15 (BRN)
jz.t	d15, n, disp4 (SBRN)

Description:

If bit *n* of register *Da* is equal to zero, then add the value specified by *disp15*, multiplied by two and sign-extended/zero-extended to 32 bits, to the contents of the PC, and jump to that address.

Refer also to Section 8.3, "Bit Operations," on page 100.

Operation:

```
if (!D[a][n]) then (PC = PC + sign_ext(2 * disp15); n = 0 – 31)
if (!D[15] [n]) then PC = PC + zero_ext(2* disp4); n = 0 – 31
```

Example:

```
jz.t  d1, n, foobar
```

See Also:

JNZ.A (pg 217), **JNZ.T** (pg 218), **JZ.A** (pg 220)

LD.A Load Word to Address Register LD.A

Syntax:

ld.a Aa, <mode>

Description:

Load the word contents of the memory location specified by the addressing mode into address register *Aa*.

Operation:

A[a] = M(EA, word)

<mode>	Syntax	Effective Address	Instruction Format
Absolute	constant	Refer to Section 2.4.1.1, "Absolute Addressing," on page 20	ABS
Base + Short Offset	[An]offset	A[b]+sign_ext(offset10)	BO
Base + Long Offset	[An]offset	A[b]+sign_ext(offset16)	BOL
Pre-increment	[+An]offset	A[b]+sign_ext(offset10)	BO
Post-increment	[An+]offset	A[b]	BO
Circular	[An+c]offset	Refer to Section 2.4.1.5, "Circular Addressing," on page 21	BO
Bit-reverse	[An+r]	Refer to Section 2.4.1.6, "Bit-Reverse Addressing," on page 22	BO

See Also:

LD.B (pg 224), **LD.BU** (pg 224), **LD.D** (pg 226), **LD.DA** (pg 227), **LD.H** (pg 228), **LD.HU** (pg 228), **LD.Q** (pg 230), **LD.W** (pg 231), **ST.A** (pg 365)

LD.A Load Word to Address Register (16-bit) LD.A

Syntax:

ld.a Aa, <mode>

Description:

Load the word contents of the memory location specified by the addressing mode into address register *Ab*.

Operation:

A[a] = M(EA, word)

<mode>	Syntax	Effective Address	Instruction Format
Register indirect	[An]	A[b]	SLR
(Implicit) Base + Offset	[A15]offset	A[15]+zero_ext(offset4)	SLRO
Implicit destination register	[An]offset4	A[b]+zero_ext(offset4), byte)	SRO
Post-increment	[An+]offset	A[b], byte; A[b] = A[b] + 4	SLR

See Also:

LD.B (pg 224), **LD.BU** (pg 224), **LD.H** (pg 228), **LD.W** (pg 231)

LD.B Load Byte LD.B
LD.BU Load Byte Unsigned LD.BU

Syntax:

```
ld.b     Da, <mode>
ld.bu    Da, <mode>
```

Description:

Load the byte contents of the memory location specified by the addressing mode, sign-extended/zero-extended to 32 bits, into data register *Da*.

Operation:

D[a] = sign_ext(M(EA, byte))
D[a] = zero_ext(M(EA, byte))

<mode>	Syntax	Effective Address	Instruction Format
Absolute	constant	Refer to Section 2.4.1.1, "Absolute Addressing," on page 20	ABS
Base + Short Offset	[An]offset	A[b]+sign_ext(offset10)	BO
Pre-increment	[+An]offset	A[b]+sign_ext(offset10)	BO
Post-increment	[An+]offset	A[b]	BO
Circular	[An+c]offset	Refer to Section 2.4.1.5, "Circular Addressing," on page 21	BO
Bit-reverse	[An+r]	Refer to Section 2.4.1.6, "Bit-Reverse Addressing," on page 22	BO

See Also:

LD.A (pg 222), **LD.D** (pg 226), **LD.DA** (pg 227), **LD.H** (pg 228), **LD.HU** (pg 228), **LD.Q** (pg 230), **LD.W** (pg 231), **ST.B** (pg 367)

LD.B — Load Byte (16-bit) — LD.B
LD.BU — Load Byte Unsigned (16-bit) — LD.BU

Syntax: I

```
ld.b     Da, <mode>
ld.bu    Da, <mode>
```

Description:

Load the byte contents of the memory location specified by the addressing mode, sign-extended/zero-extended to 32 bits, into data register *Da*.

Operation:

```
D[a] = sign_ext(M(EA, byte))
D[a] = zero_ext(M(EA, byte))
```

<mode>	Syntax	Effective Address	Instruction Format
Register indirect	[An]	A[b]	SLR
(Implicit) Base + Offset	[A15]offset	A[15]+zero_ext(offset4)	SLRO
Implicit destination register	[An]offset4	A[b]+zero_ext(offset4), byte)	SRO
Post-increment	[An+]offset	A[b], byte; A[b] = A[b] + 1	SLR

See Also:

LD.A (pg 222), **LD.H** (pg 228), **LD.W** (pg 231)

LD.D Load Doubleword to Data Register LD.D

Syntax:

ld.d Ea, <mode>

Description:

Load the doubleword contents of the memory location specified by the addressing mode into extended data register E*a*. The least-significant word of the doubleword value is loaded into the even register (D*n*) and the most-significant word is loaded into the odd register (D*n*+1).

Operation:

E[a] = M(EA, doubleword)

<mode>	Syntax	Effective Address	Instruction Format
Absolute	constant	Refer to Section 2.4.1.1, "Absolute Addressing," on page 20	ABS
Base + Short Offset	[An]offset	A[b]+sign_ext(offset10)	BO
Pre-increment	[+An]offset	A[b]+sign_ext(offset10)	BO
Post-increment	[An+]offset	A[b]	BO
Circular	[An+c]offset	Refer to Section 2.4.1.5, "Circular Addressing," on page 21	BO
Bit-reverse	[An+r]	Refer to Section 2.4.1.6, "Bit-Reverse Addressing," on page 22	BO

See Also:

LD.A (pg 222), **LD.DA** (pg 227), **LD.H** (pg 228), **LD.HU** (pg 228), **LD.Q** (pg 230), **LD.W** (pg 231)

LD.DA Load Doubleword to Address Register LD.DA

Syntax:

ld.da Aa, <mode>

Description:

Load the doubleword contents of the memory location specified by the addressing mode into data register pair *Aa*. The least-significant word of the doubleword value is loaded into the even register (A*n*) and the most-significant word is loaded into the odd register (A*n*+1).

Operation:

A[a](pair) = M(EA, doubleword)

<mode>	Syntax	Effective Address	Instruction Format
Absolute	constant	Refer to Section 2.4.1.1, "Absolute Addressing," on page 20	ABS
Base + Short Offset	[An]offset	A[b]+sign_ext(offset10)	BO
Pre-increment	[+An]offset	A[b]+sign_ext(offset10)	BO
Post-increment	[An+]offset	A[b]	BO
Circular	[An+c]offset	Refer to Section 2.4.1.5, "Circular Addressing," on page 21	BO
Bit-reverse	[An+r]	Refer to Section 2.4.1.6, "Bit-Reverse Addressing," on page 22	BO

See Also:

LD.A (pg 222), **LD.DA** (pg 227), **LD.H** (pg 228), **LD.HU** (pg 228), **LD.Q** (pg 230), **LD.W** (pg 231)

LD.H Load Halfword LD.H

LD.HU Load Halfword Unsigned LD.HU

Syntax:

```
ld.h    Da, <mode>
ld.hu   Da, <mode>
```

Description:

Load the halfword contents of the memory location specified by the addressing mode, sign-extended/zero-extended to 32 bits, into *Da*.

Operation:

D[a] = sign_ext(M(EA, halfword))
D[a] = zero_ext(M(EA, halfword))

<mode>	Syntax	Effective Address	Instruction Format
Absolute	constant	Refer to Section 2.4.1.1, "Absolute Addressing," on page 20	ABS
Base + Short Offset	[An]offset	A[b]+sign_ext(offset10)	BO
Pre-increment	[+An]offset	A[b]+sign_ext(offset10)	BO
Post-increment	[An+]offset	A[b]	BO
Circular	[An+c]offset	Refer to Section 2.4.1.5, "Circular Addressing," on page 21	BO
Bit-reverse	[An+r]	Refer to Section 2.4.1.6, "Bit-Reverse Addressing," on page 22	BO

See Also:

LD.A (pg 222), **LD.D** (pg 226), **LD.DA** (pg 227), **LD.Q** (pg 230), **LD.W** (pg 231)

LD.H Load Halfword (16-bit) LD.H

Syntax:

ld.h Da, <mode>

Description:

Load the halfword contents of the memory location specified by the addressing mode, sign-extended/zero-extended to 32 bits, into address register *Da*.

Operation:

D[a] = sign_ext(M(EA, halfword))

<mode>	Syntax	Effective Address	Instruction Format
Register indirect	[An]	A[b]	SLR
(Implicit) Base + Offset	[A15]offset	A[15]+zero_ext(offset4)	SLRO
Implicit destination register	[An]offset4	A[b]+zero_ext(offset4), byte)	SRO
Post-increment	[An+]offset	A[b], byte; A[a] = A[a] + 2	SLR

See Also:

LD.A (pg 222), **LD.B** (pg 224), **LD.BU** (pg 224), **LD.W** (pg 231)

LD.Q Load Halfword Signed Fraction LD.Q

Syntax:

ld.q Da, <mode>

Description:

Load the halfword contents of the memory location specified by the addressing mode into the most-significant halfword of data register *Da*, setting the 16 least-significant bits of *Da* to zero.

Operation:

D[a] = {M(EA, halfword), 16'h 0000}

<mode>	Syntax	Effective Address	Instruction Format
Absolute	constant	Refer to Section 2.4.1.1, "Absolute Addressing," on page 20	ABS
Base + Short Offset	[An]offset	A[b]+sign_ext(offset10)	BO
Pre-increment	[+An]offset	A[b]+sign_ext(offset10)	BO
Post-increment	[An+]offset	A[b]	BO
Circular	[An+c]offset	Refer to Section 2.4.1.5, "Circular Addressing," on page 21	BO
Bit-reverse	[An+r]	Refer to Section 2.4.1.6, "Bit-Reverse Addressing," on page 22	BO

See Also:

LD.A (pg 222), **LD.DA** (pg 227), **LD.B** (pg 224), **LD.BU** (pg 224), **LD.H** (pg 228), **LD.HU** (pg 228), **LD.W** (pg 231)

LD.W Load Word LD.W

Syntax:

ld.w Da, <mode>

Description:

Load the word contents of the memory location specified by the addressing mode into data register *Db.*

Operation:

D[a] = M(EA, word)

<mode>	Syntax	Effective Address	Instruction Format
Absolute	constant	Refer to Section 2.4.1.1, "Absolute Addressing," on page 20	ABS
Base + Short Offset	[An]offset	A[b]+sign_ext(offset10)	BO
Base + Long Offset	[An]offset	A[b]+sign_ext(offset16)	BOL
Pre-increment	[+An]offset	A[b]+sign_ext(offset10)	BO
Post-increment	[An+]offset	A[b]	BO
Circular	[An+c]offset	Refer to Section 2.4.1.5, "Circular Addressing," on page 21	BO
Bit-reverse	[An+r]	Refer to Section 2.4.1.6, "Bit-Reverse Addressing," on page 22	BO

See Also:

LD.A (pg 222), **LD.DA** (pg 227), **LD.B** (pg 224), **LD.BU** (pg 224), **LD.H** (pg 228), **LD.HU** (pg 228), **LD.Q** (pg 230), **LD.W** (pg 231)

LD.W Load Word (16-bit) LD.W

Syntax:

```
ld.w     Da, <mode>
```

Description:

Load the word contents of the memory location specified by the addressing mode into data register *Da*.

Operation:

D[a] = M(EA, word)

<mode>	Syntax	Effective Address	Instruction Format
Register indirect	[An]	A[b]	SLR
(Implicit) Base + Offset	[A15]offset	A[15]+zero_ext(offset4)	SLRO
Implicit destination register	[An]offset4	A[b]+zero_ext(offset4), byte)	SRO
Post-increment	[An+]offset	A[b], byte; A[b] = A[b] + 4	SLR

See Also:

LD.B (pg 224), **LD.BU** (pg 224), **LD.H** (pg 228)

LDLCX Load Lower Context LDLCX

Syntax:

ldlcx <mode>

Description:

Load the contents of the memory block specified by the addressing mode into registers A2 – A7 and D0 – D7. This operation is used normally to restore GPR values that were saved previously by an STLCX instruction.

Note that the effective address specified by the addressing mode must resolve to an on-chip memory location aligned on a 16-word boundary. For this instruction, the addressing mode is restricted to absolute (ABS) or base plus short offset (BO).

Operation:

Refer to Section 8.8.2, "Context Loading and Storing," on page 109.

<mode>	Syntax	Effective Address	Instruction Format
Absolute	constant	Refer to Section 2.4.1.1, "Absolute Addressing," on page 20	ABS
Base + Short Offset	[An]offset	A[a]+sign_ext(offset10)	BO

See Also:

LDUCX (pg 235), **RSLCX** (pg 339), **STLCX** (pg 377), **STLCX** (pg 377), **SVLCX** (pg 388)

LDMDST Load-Modify-Store LDMDST

Syntax:

ldmdst <mode>, Da

Description:

Compute the logical AND of the word contents of the memory location specified by the addressing mode and the inverse of the contents of register *D*[*a*+1]. OR the result with the AND of the contents of *Da* and *Da*+1 and store the result in the memory location specified by the addressing mode. The mask and value pair, *Da* and *Da*+1, can be generated using the IMASK instruction.

Refer also to Section 8.7.3, "Store Bit and Bit Field," on page 108.

Operation:

M(EA, word) = (M(EA, word) AND !D[b+1]) OR (D[b] AND D[b+1])

<mode>	Syntax	Effective Address	Instruction Format
Absolute	constant	Refer to Section 2.4.1.1, "Absolute Addressing," on page 20	ABS
Base + Short Offset	[An]offset	A[b]+sign_ext(offset10)	BO
Pre-increment	[+An]offset	A[b]+sign_ext(offset10)	BO
Post-increment	[An+]offset	A[b]	BO
Circular	[An+c]offset	Refer to Section 2.4.1.5, "Circular Addressing," on page 21	BO
Bit-reverse	[An+r]	Refer to Section 2.4.1.6, "Bit-Reverse Addressing," on page 22	BO

See Also:

IMASK (pg 194), **ST.T** (pg 374)

LDUCX Load Upper Context LDUCX

Syntax:

lducx <mode>

Description:

Load the contents of the memory block specified by the addressing mode into registers A10 – A15 and D8 – D15. This operation is used normally to restore GPR values that were saved previously by an STUCX instruction.

Note that the effective address specified by the addressing mode must resolve to an on-chip memory location aligned on a 16-word boundary. For this instruction, the addressing mode is restricted to absolute (ABS) or base plus short offset (BO).

Operation:

Refer to Section 8.8.2, "Context Loading and Storing," on page 109.

<mode>	Syntax	Effective Address	Instruction Format
Absolute	constant	Refer to Section 2.4.1.1, "Absolute Addressing," on page 20	ABS
Base + Short Offset	[An]offset	A[a]+sign_ext(offset10)	BO

See Also:

LDLCX (pg 233), **RSLCX** (pg 339), **STLCX** (pg 377), **STUCX** (pg 378), **SVLCX** (pg 388)

LEA Load Effective Address LEA

Syntax:

```
lea        Aa, <mode>
```

Description:

Compute the absolute (effective) address defined by the addressing mode and put the result in address register Ab. Refer to Section 2.4.1, "TriCore Addressing Modes," on page 19.

Operation:

A[a] = EA

<mode>	Syntax	Effective Address	Instruction Format
Absolute	constant	Refer to Section 2.4.1.1, "Absolute Addressing," on page 20	ABS
Base + Short Offset	[Ab]offset	A[b]+sign_ext(offset10)	BO
Base + Long Offset	[Ab]offset	A[b]+sign_ext(offset16)	BOL
Pre-increment	[+Ab]offset	A[b]+sign_ext(offset10)	BO
Post-increment	[Ab+]offset	A[b]	BO
Circular	[Ab+c]offset	Refer to Section 2.4.1.5, "Circular Addressing," on page 21	BO
Bit-reverse	[An+r]	Refer to Section 2.4.1.6, "Bit-Reverse Addressing," on page 22	BO

See Also:

MOV.A (pg 277), **MOV.D** (pg 279), **MOVH.A** (pg 282), **MOVZ.A** (pg 283)

LOOP Loop LOOP

Syntax:

loop	Aa, disp15 (BRR)
loop	Ab, disp4 (SBR)

Description:

If address register *Aa* is not equal to zero, then add the value specified by *disp15*, multiplied by two and sign-extended to 32 bits, to the contents of the PC, and jump to that address. The address register is decremented after the comparison is performed.

If address register *Aa* is not equal to zero, then add the value specified by *disp4*, multiplied by two and one-extended to a 32-bit negative number, to the contents of the PC, and jump to that address. The address register is decremented after the comparison is performed.

Refer also to Section 8.6.2.4, "Loop Instructions," on page 105.

Operation:

```
if (A[a] != 0) then (PC = PC + sign_ext(2 * disp15); A[a] = A[a]–1); ("zero overhead" loop)
if (A[a] != 0) then (PC = PC + one_ext(2 * disp4); A[a] = A[a]–1); ("zero overhead" loop)
```

Example:

```
loop  a4, iloop
```

See Also:

J (pg 198), **JA** (pg 199), **JI** (pg 205), **JL** (pg 206), **JLA** (pg 207), **JLI** (pg 209)

LT Less Than LT

LT.U Less Than Unsigned LT.U

Syntax:

lt	Dc, Da, Db (RR)
lt	Dc, Da, const9 (RC)
lt	D15, Da, Db (SRR)
lt	D15, Da, const4 (SRC)
lt.u	Dc, Da, Db (RR)
lt.u	Dc, Da, const9 (RC)
lt.u	D15, Da, Db (SRR)
lt.u	D15, Da, const4 (SRC)

Description:

If the contents of data register *Da* are less than the contents of data register *Db/const9*, set the least-significant bit of *Dc* to 1 and clear the remaining bits to zero; otherwise, clear all bits in *Dc*. *Da* and *Db* are treated as signed integers, and the *const9* value is sign-extended to 32 bits.

If the contents of data register *Da* are less than the contents of data register *Db/const4*, set the least-significant bit of D15 to 1 and clear the remaining bits to zero; otherwise, clear all bits in D15. The operands are treated as signed 32-bit integers, and the *const9* value is sign-extended to 32 bits.

If the contents of data register *Da* are less than the contents of data register *Db/const9*, set the least-significant bit of *Dc* to 1 and clear the remaining bits to zero; otherwise, clear all bits in *Dc*. *Da* and *Db* are treated as unsigned integers, and the *const9* value is zero-extended to 32 bits.

If the contents of data register *Da* are less than the contents of data register *Db/const4*, set the least-significant bit of D15 to 1 and clear the remaining bits to zero; otherwise, clear all bits in D15. The operands are treated as unsigned 32-bit integers, and the *const9* value is zero-extended to 32 bits.

Refer also to Section 8.2, "Compare Instructions," on page 97.

Operation:

```
D[c] = (D[a] < D[b]); signed
D[c] = (D[a] < sign_ext(const9)); signed
D[15] = (D[a] < D[b]); signed
D[15] = (D[a] < sign_ext(const4)); signed
```

D[c] = (D[a] < D[b]); unsigned
D[c] = (D[a] < zero_ext(const9)); unsigned
D[15] = (D[a] < D[b]); unsigned
D[15] = (D[a] < zero_ext(const4)); unsigned

Examples:

```
lt     d3, d1, d2
lt     d3, d1, 126
lt     d15, d1, d2
lt     d15, d1, 6
lt.u   d3, d1, 253
lt.u   d3, d1, d2
lt.u   d15, d1, d2
lt.u   d15, d1, 6
```

See Also:

EQ.B (pg 188), **EQ.H** (pg 188), **EQ.W** (pg 188), **LT.B** (pg 241), **LT.BU** (pg 241), **LT.H** (pg 242), **LT.HU** (pg 242), **LT.W** (pg 243), **LT.WU** (pg 243)

LT.A Less Than Address LT.A

Syntax:

lt.a Dc, Aa, Ab (RR)

Description:

If the contents of address register *Aa* are less than the contents of address register *Ab*, set the least-significant bit of *Dc* to 1 and clear the remaining bits to zero; otherwise, clear all bits in *Dc*. The operands are treated as unsigned 32-bit integers.

Operation:

D[c] = (A[a] < A[b]); unsigned

Example:

```
lt.a  d3, a4, a2
```

See Also:

EQ.A (pg 187), **EQZ.A** (pg 190), **GE.A** (pg 193), **NE.A** (pg 322), **NEZ.A** (pg 323)

LT.B — Less Than Packed Byte — LT.B
LT.BU — Less Than Packed Byte Unsigned — LT.BU

Syntax:

lt.b	Dc, Da, Db (RR)
lt.bu	Dc, Da, Db (RR)

Description:

Compare each byte of data register *Da* with the corresponding byte of *Db*. In each case, if the value of the byte in *Da* is less than the value of the byte in *Db*, set all bits in the corresponding byte of *Dc* to 1; otherwise, clear all the bits. The operands are treated as signed 8-bit integers.

Compare each byte of data register *Da* with the corresponding byte of *Db*. In each case, if the value of the byte in *Da* is less than the value of the byte in *Db*, set all bits in the corresponding byte of *Dc* to 1; otherwise, clear all the bits. The operands are treated as unsigned 8-bit integers.

Refer also to Section 8.1.3, "Packed Arithmetic," on page 95.

Operation:

```
if (D[a][(n+7):n] < D[b][(n+7):n])
then D[c][(n+7):n] = 8'h FF
else D[c][(n+7):n] = 8'h 00; n = 0, 8, 16, 24, signed

if (D[a][(n+7):n] < D[b][(n+7):n])
then D[c][(n+7):n] = 8'h FF
else D[c][(n+7):n] = 8'h 00; n = 0, 8, 16, 24,unsigned
```

Examples:

```
lt.b  d3, d1, d2
lt.bu d3, d1, d2
```

See Also:

EQ.B (pg 188), **EQ.H** (pg 188), **EQ.W** (pg 188), **LT.H** (pg 242), **LT.HU** (pg 242), **LT.W** (pg 243), **LT.WU** (pg 243)

LT.H Less Than Packed Halfword LT.H
LT.HU Less Than Packed Halfword Unsigned LT.HU

Syntax:

lt.h	Dc, Da, Db (RR)
lt.hu	Dc, Da, Db (RR)

Description:

Compare each halfword of data register *Da* with the corresponding halfword of *Db*. In each case, if the value of the halfword in *Da* is less than the value of the corresponding halfword in *Db*, set all bits of the corresponding halfword of *Dc* to 1; otherwise, clear all the bits. The operands are treated as signed 16-bit integers.

Compare each halfword of data register *Da* with the corresponding halfword of *Db*. In each case, if the value of the halfword in *Da* is less than the value of the corresponding halfword in *Db*, set all bits of the corresponding halfword of *Dc* to 1; otherwise, clear all the bits. The operands are treated as unsigned 16-bit integers.

Refer also to Section 8.1.3, "Packed Arithmetic," on page 95.

Operation:

```
if (D[a][(n+15):n] < D[b][(n+15):n])
then D[c][(n+15):n] = 16'h FFFF
else D[c][(n+15):n] = 16'h 0000; n = 0, 16, signed
```

```
if (D[a][(n+15):n] < D[b][(n+15):n])
then D[c][(n+15):n] = 16'h FFFF
else D[c][(n+15):n] = 16'h 0000; n = 0, 16, unsigned
```

Refer also to Section 8.2, "Compare Instructions."

Examples:

```
lt.h  d3, d1, d2
lt.hu d3, d1, d2
```

See Also:

EQ.B (pg 188), **EQ.H** (pg 188), **EQ.W** (pg 188), **LT.B** (pg 241), **LT.BU** (pg 241), **LT.W** (pg 243), **LT.WU** (pg 243)

LT.W — Less Than Packed Word — LT.W
LT.WU — Less Than Packed Word Unsigned — LT.WU

Syntax:

lt.w	Dc, Da, Db (RR)
lt.wu	Dc, Da, Db (RR)

Description:

If the contents of data register *Da* are less than the contents of data register *Db*, set all bits in *Dc* to 1 and clear the remaining bits to zero; otherwise, clear all bits in *Dc*. *Da* and *Db* are treated as signed 32-bit integers.

If the contents of data register *Da* are less than the contents of data register *Db*, set all bits in *Dc* to 1 and clear the remaining bits to zero; otherwise, clear all bits in *Dc*. *Da* and *Db* are treated as unsigned 32-bit integers.

Refer also to Section 8.1.3, "Packed Arithmetic," on page 95.

Operation:

```
if (D[a] < D[b])
then D[c] = 32'h FFFFFFFF
else D[c] = 32'h 00000000, signed

if (D[a] < D[b])
then D[c] = 32'h FFFFFFFF
else D[c] = 32'h 00000000, unsigned
```

Examples:

```
lt.w  d3, d1, d2
lt.wu d3, d1, d2
```

See Also:

EQ.B (pg 188), **EQ.H** (pg 188), **EQ.W** (pg 188), **LT.B** (pg 241), **LT.BU** (pg 241), **LT.H** (pg 242), **LT.HU** (pg 242)

MADD — Multiply-Add (32 x 32) + 32 => 32 — MADD

Syntax:

madd	Dc, Dd, Da, Db (RRR)
madd	Dc, Dd, Da, const9 (RCR)

Description:

Multiply the contents of data register *Da* by the contents of data register *Db/const9*, add the result to the contents of data register *Dd*, and put the result in data register *Dc*. The operands are treated as signed, 32-bit integers. The value *const9* is sign-extended to 32 bits before the multiplication is performed. Overflow and advanced overflow are calculated on the final result.

Operation:

```
D[c][31:0] = D[d][31:0] + (D[a][31:0] * D[b][31:0])
D[c][31:0] = D[d][31:0] + (D[a][31:0] * sign_ext(const9))
```

Status:

V, SV, AV, SAV

Examples:

```
madd  d3, d4, d1, d2
madd  d3, d4, d1, 126
```

See Also:

MADDM (pg 249), **MADDM.H** (pg 250), **MADDM.U** (pg 249), **MADDMS** (pg 254), **MADDMS.U** (pg 254), **MADDS** (pg 263), **MADDS.H** (pg 264), **MADDS.U** (pg 263)

MADD.H Packed Multiply-Add Q Format (16 x 16) + 32 => 32 || (16 x 16) + 32 => 32 MADD.H

Syntax:

madd.h Dc, Dd, Da, Db, n (RRR)

Description:

Multiply the most-significant halfword of data register *Da* by the most-significant half-word of data register *Db* and add the product, left-justified if *n* = 1, to the upper 32 bits of extended data register *Ed*. Put the result in the upper 32 bits of extended data register *Ec*.

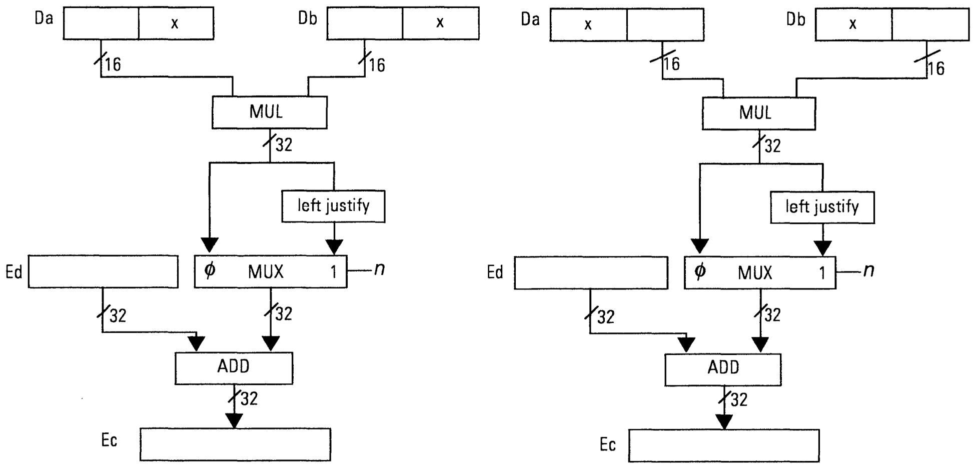
Multiply the least-significant halfword of data register *Da* by the least-significant halfword of data register *Db* and add the product, left-justified if *n* = 1, to the lower 32 bits of extended data register *Ed*. Put the result in the lower 32 bits of extended data register *Ec*. The operands are treated as signed values.

Overflow and advanced overflow are calculated on each independent final result.

If n=1, 0x8000 x 0x8000 = 0x7FFF.FFFF.

Operation:

```
E[c][63:32] = E[d][63:32] + (((D[a][31:16] * D[b][31:16]) << n)[31:0])
E[c][31:0] = E[d][31:0] + (((D[a][15:0] * D[b][15:0]) << n)[31:0]);
signed, n = 0, 1
```



Status:

V, SV, AV, SAV

Example:

```
madd.h   e0, e8, d2, d3, 1
```

See Also:

MADDM.Q (pg 252), **MADDR.Q** (pg 257), **MADDRS.H** (pg 259), **MADDRS.Q** (pg 261)

MADD.Q Multiply-Add Q Format (16 x 16) + 32 => 32 MADD.Q

Syntax:

madd.q Dc, Dd, Da, Db, n (RRR)

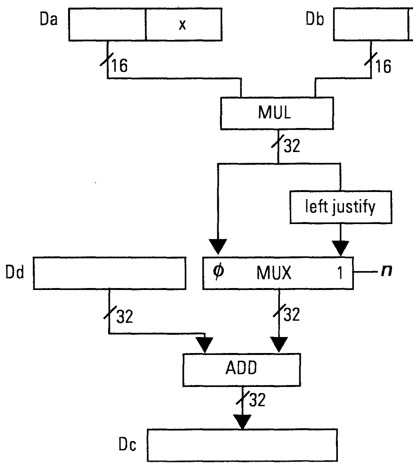
Description:

Multiply the most-significant halfword of data register *Da* by the most-significant half-word of data register *Db* and add the product, left-justified if *n* = 1, to the contents of data register *Dd*. Put the result in *Dc*. The operands are treated as signed values. Overflow and advanced overflow are calculated on the final result.

If *n* = 1, 0x8000 x 0x8000 = 0x7FFF.FFFF.

Operation:

D[c][31:0] = D[d][31:0] + (((D[a][31:16] * D[b][31:16]) << n)[31:0]); signed, n = 0, 1



TAM046.1

Status:

V, SV, AV, SAV

Example:

```
madd.q   d13, d4, d1, d12, 0
```

See Also:

MADDM.Q (pg 252), **MADDR.Q** (pg 257), **MADDRS.H** (pg 259), **MADDRS.Q** (pg 261), **MADDS.H** (pg 264), **MADDS.Q** (pg 265)

MADDM — Multiply-Add with Multiword Result (32 x 32) + 64 => 64 — MADDM

MADDM.U — Multiply-Add with Multiword Result Unsigned (32 x 32) + 32 => 64 — MADDM.U

Syntax:

```
maddm   Ec, Ed, Da, Db (RRR)
maddm   Ec, Ed, Da, const9 (RCR)
maddm.u Ec, Ed, Da, Db (RRR)
maddm.u Ec, Ed, Da, const9 (RCR)
```

Description:

Multiply the contents of data register *Da* by the contents of data register *Db/const9*, add the 64-bit result to the contents of extended data register *Ed*, and put the result in extended data register *Ec*. The operands are treated as signed integers. The value *const9* is sign-extended/zero-extended to 32 bits before the multiplication is performed.

Overflow and advanced overflow are calculated on the final result.

Operation:

```
E[c][63:0] = E[d][63:0] + (D[a][31:0] * D[b][31:0]); signed
E[c][63:0] = E[d][63:0] + (D[a][31:0] * sign_ext(const9)); signed

E[c][63:0] = D[d][63:0] + (D[a][31:0] * D[b][31:0]); unsigned
E[c][63:0] = E[d][63:0] + (D[a][31:0] * zero_ext(const9)); unsigned
```

Status:

V, SV, AV, SAV

Examples:

```
maddm e2, e4, d10, d12
maddm e0, e14, d4, 126
maddm.u  e0, e0, d8, d2
maddm.u  e12, e0, d9, 0x28
```

See Also:

MADD (pg 244), **MADDMS** (pg 254), **MADDMS.U** (pg 254), **MADDS** (pg 263), **MADDS.H** (pg 264), **MADDS.U** (pg 263)

MADDM.H **Packed Multiply-Add with Multiword Result (16 x 16) + (16 x 16) + 64 => 64** MADDM.H

Syntax:

maddm.h Ec, Ed, Da, Db (RRR)

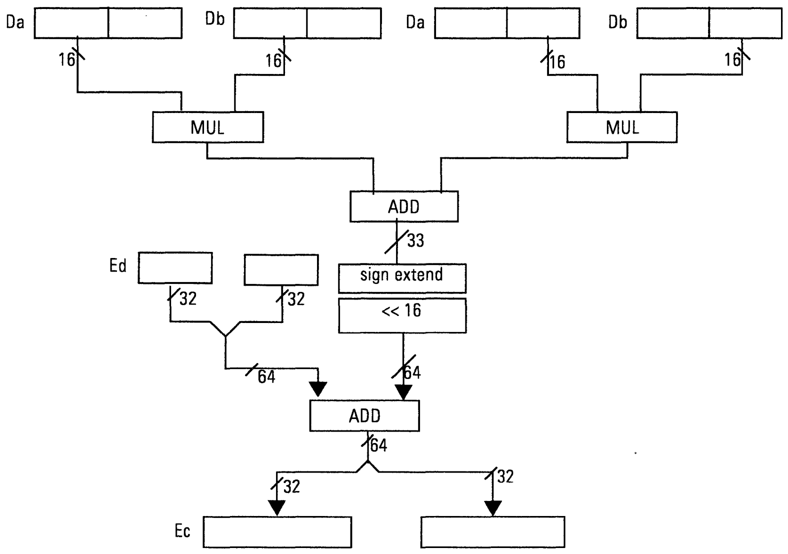
Description:

Multiply the most-significant halfword of data register *Da* by the most-significant halfword of data register *Db*. Multiply the least-significant halfword of data register *Da* by the least-significant halfword of data register *Db*. Add the products of the two multiplications, sign-extend the result to 64 bits, and left-shift by 16. Add that value to the extended data register *Ed* and put the 64-bit result in extended data register *Ec*. The operands are treated as signed values.

Overflow and advanced overflow are calculated on the final result.

Operation:

E[c][63:0] = E[d][63:0] + ((D[a][31:16] * D[b][31:16]) + (D[a] [15:0] * D[b] [15:0])<<16); signed



Status:

V, SV, AV, SAV

Example:

```
maddm.h  e0, e4, d2, d7
```

See Also:

MADD.Q (pg 247), **MADDR.H** (pg 255), **MADDR.Q** (pg 257), **MADDRS.H** (pg 259), **MADDRS.Q** (pg 261), **MADDS.H** (pg 264), **MADDS.Q** (pg 265)

MADDM.Q Multiply-Add Multiword Result Q Format (16 x 16) + 64 => 64 MADDM.Q

Syntax:

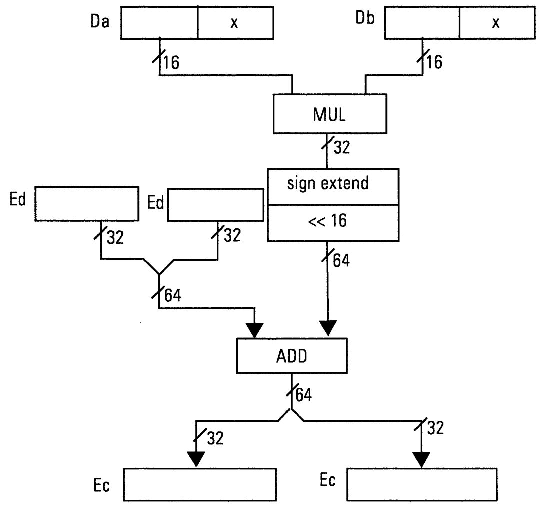
maddm.q Ec, Ed, Da, Db (RRR)

Description:

Multiply the most-significant halfword of data register *Da* by the most-significant halfword of data register *Db*, sign-extend the result to 64 bits, and left-shift by 16. Add that value to extended data register *Ed* and put the 64-bit result in extended data register *Ec*. The operands are treated as signed values. Overflow and advanced overflow are calculated on the final result.

Operation:

E[c] [63:0] = E[d][63:0] + (sign_ext((D[a][31:16] * D[b][31:16]) [31:0]))<<16 ; signed



TAM048.1

Status:

V, SV, AV, SAV

Example:

```
maddm.q  e2, e4, d1, d2
```

See Also:

MADD.Q (pg 247), **MADDR.Q** (pg 257), **MADDRS.Q** (pg 261), **MADDS.Q** (pg 265)

MADDMS — Multiply-Add Multiword with Saturation (32 x 32) + 64 => 64 — MADDMS

MADDMS.U — Multiply-Add Multiword with Saturation Unsigned (32 x 32) + 64 => 64 — MADDMS.U

Syntax:

maddms Ec, Ed, Da, Db (RRR)
maddms Ec, Ed, Da, const9 (RCR)
maddms.u Ec, Ed, Da, Db (RRR)
maddms.u Ec, Ed, Da, const9 (RCR)

Description:

Multiply the contents of data register *Da* by the contents of data register *Db/const9*, add the 64-bit result to the contents of extended data register *Ed*, and put the result in extended data register *Ec*. The operands are treated as signed/unsigned integers. The value *const9* is sign-extended/zero-extended to 32 bits before the multiplication is performed.

Overflow and advanced overflow are calculated on the final result. On overflow, the result is saturated.

Operation:

E[c][63:0] = E[d][63:0] + ((D[a][31:0] * D[b][31:0])[63:0]); signed; ssov
E[c][63:0] = E[d][63:0] + ((D[a][31:0] * sign_ext(const9))[63:0]); signed; ssov

E[c][63:0] = E[d][63:0] + ((D[a][31:0] * D[b][31:0])[63:0]); unsigned; suov
E[c][63:0] = E[d][63:0] + ((D[a][31:0] * zero_ext(const9))[63:0]); unsigned; suov

Status:

V, SV, AV, SAV

Examples:

```
maddms e12, e4, d1, d2
maddms e12, e4, d1, 0x12
maddms.u e0, e0, d4, d2
maddms.u e0, e0, d4,0x25
```

See Also:

MADD (pg 244), **MADDM** (pg 249), **MADDM.U** (pg 249), **MADDS** (pg 263), **MADDS.U** (pg 263)

MADDR.H Packed Multiply-Add with Rounding (16 x 16) + 32 => 16 (16 x 16) + 32 => 16 MADDR.H

Syntax:

maddr.h Dc, Ed, Da, Db, n (RRR)

Description:

Multiply the most-significant halfword of data register *Da* by the most-significant halfword of data register *Db* and add the product, left-justified if *n* = 1, to the upper 32 bits of extended data register *Ed.* Round the contents and put the most-significant halfword of the rounded value into the most-significant halfword of *Dc.*

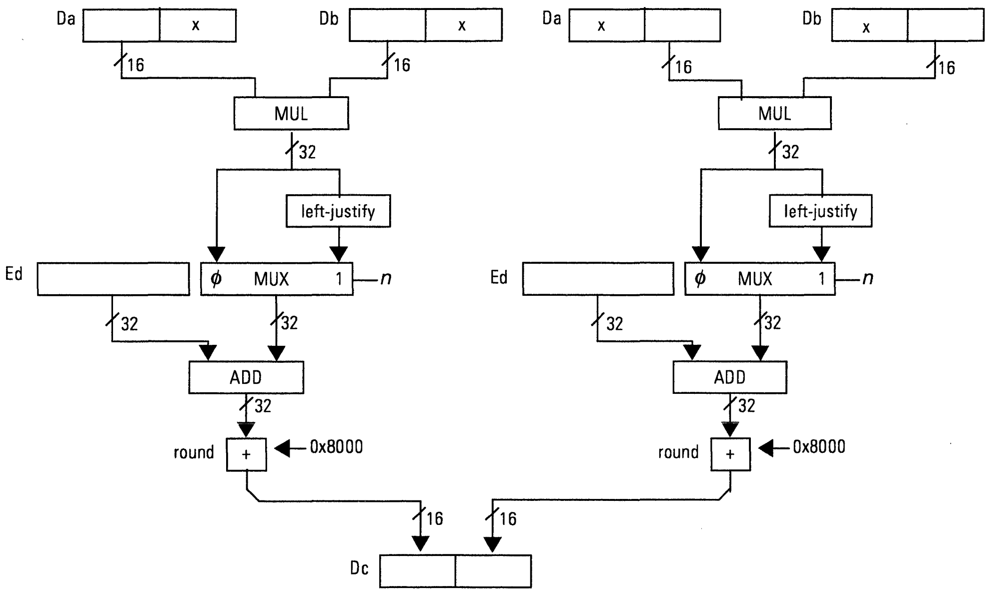
Multiply the least-significant halfword of data register *Da* by the least-significant halfword of data register *Db* and add the product, left-justified if *n* = 1, to the lower 32 bits of extended data register *Ed.* Round the contents and put the least-significant halfword of the rounded value into the least-significant halfword of *Dc.*

The operands are treated as signed values. Overflow and advanced overflow are calculated on each independent final result.

If *n* = 1, 0x8000 x 0x8000 = 0x7FFF.FFFF.

Operation:

```
D[c][31:16] = round16(E[d][63:32] + ((D[a] [31:16]  * D[b] [31:16]) << n)[31:16])
D[c][15:0] = round16(E[d][31:0] + ((D[a] [15:0]  * D[b] [15:0]) << n)[15:0]); signed; n = 0, 1
```



Status:

V, SV, AV, SAV

Example:

```
maddr.h  d3, e0, d4, d2, 0
```

See Also:

MADD.Q (pg 247), **MADDM.Q** (pg 252), **MADDRS.Q** (pg 261), **MADDS.Q** (pg 265)

MADDR.Q — Multiply-Add With Rounding (16 x 16) + 32 => 16 — MADDR.Q

Syntax:

maddr.q Dc, Dd, Da, Db, n (RRR)

Description:

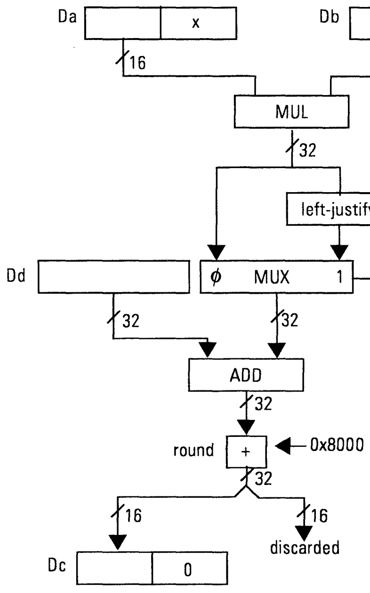
Multiply the most-significant halfword of data register *Da* by the most-significant halfword of data register *Db* and add the product, left-justified if *n* = 1, to data register *Dd*. Round the result, put the most-significant halfword of the rounded value in *Dc,* and set the least-significant halfword of *Dc* to 0.

The operands are treated as signed values. Overflow and advanced overflow are calculated on the final result.

If *n* = 1, 0x8000 * 0x8000 = 0x7FFF.FFFF.

Operation:

D[c][31:0] = round16(D[d] + ((D[a] [31:16] * D[b] [31:g16]) << n)[31:0])[31:16], 16'h 0000;
n = 0, 1; signed



TAM050.1

Status:

V, SV, AV, SAV

Example:

```
maddr.q  d3, d4, d1, d2, 0
```

See Also:

MADD.Q (pg 247), **MADDM.Q** (pg 252), **MADDRS.Q** (pg 261), **MADDS.Q** (pg 265)

MADDRS.H Packed Multiply-Add with Rounding and Saturation (16 x 16) + 32 => 16 MSB (16 x 16) + 32 => 16 LSB MADDRS.H

Syntax:

maddrs.h Dc, Ed, Da, Db, n (RRR)

Description:

Multiply the most-significant halfword of data register *Da* by the most-significant halfword of data register *Db* and add the product, left-justified if *n* = 1, to the upper 32 bits of extended data register *Ed*. Round the contents and put the most-significant halfword of the rounded value into the most-significant halfword of *Dc*.

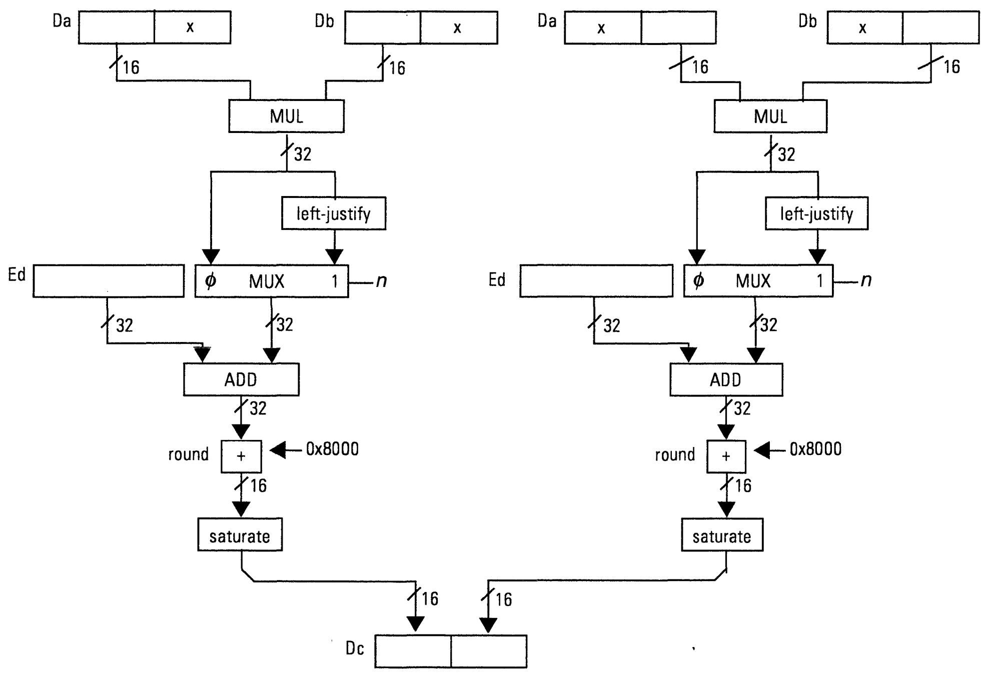
Multiply the least-significant halfword of data register *Da* by the least-significant halfword of data register *Db* and add the product, left-justified if *n* = 1, to the lower 32 bits of extended data register *Ed*. Round the contents and put the least-significant halfword of the rounded value into the least-significant halfword of *Dc*.

The operands are treated as signed/unsigned values. Overflow and advanced overflow are calculated on each independent final result. On overflow, each result is independently saturated.

If *n* = 1, 0x8000 * 0x8000 = 0x7FFF.FFFF.

Operation:

```
D[c][31:16] = round16(E[d][63:32] + ((D[a] [31:16]  * D[b] [31:16]) << n)[31:16]);
D[c][15:0] = round16(E[d][31:0] + ((D[a] [15:0]  * D[b] [15:0]) << n)[15:0]);
signed; n = 0, 1; ssov
```



Status:

V, SV, AV, SAV

Example:

```
maddr.h  d3, e4, d1, d2, 0
```

See Also:

MADD.Q (pg 247), **MADDM.H** (pg 250), **MADDM.Q** (pg 252), **MADDRS.H** (pg 259), **MADDRS.Q** (pg 261), **MADDS.H** (pg 264), **MADDS.Q** (pg 265)

MADDRS.Q Multiply-Add with Rounding and Saturation (16 x 16) + 32 => 16 MADDRS.Q

Syntax:

maddrs.q Dc, Dd, Da, Db, n (RRR)

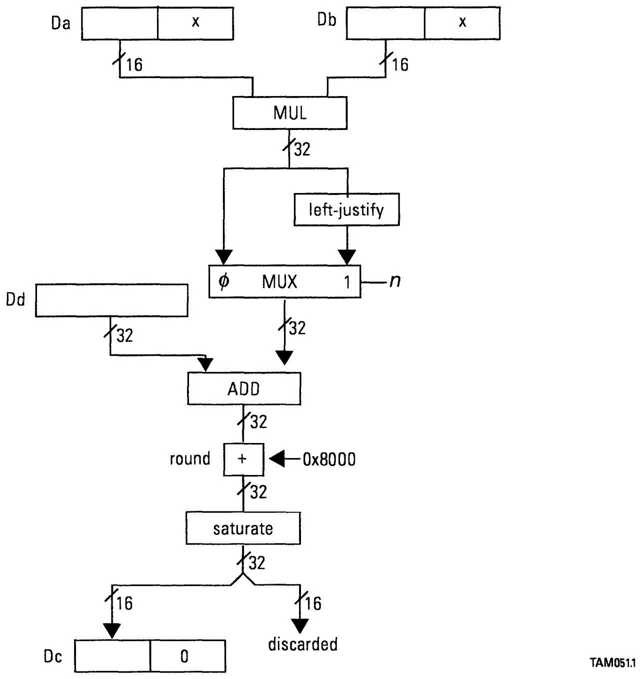
Description:

Multiply the most-significant halfword of data register *Da* by the most-significant halfword of data register *Db* and add the product, left-justified if *n* = 1, to data register *Dd.* Round the contents of *Dd;* put the most-significant halfword of the rounded value in *Dc,* and set the least-significant halfword of *Dc* to 0. Overflow and advanced overflow are calculated on the final result. On overflow, the result is saturated.

If *n* = 1, 0x8000 * 0x8000 = 0x7FFF.FFFF.

Operation:

D[c][31:0] = round16(D[d][31:0] + ((D[a] [31:16] * D[b] [31:16]) << n)[31:0])[31:16], 16'h 0000); n = 0, 1; signed; ssov



Status:

V, SV, AV, SAV

Example:

```
maddrs.q d0, d0, d1, d2, 0
```

See Also:

MADD.Q (pg 247), **MADDM.Q** (pg 252), **MADDR.Q** (pg 257), **MADDS.Q** (pg 265)

MADDS — Multiply-Add with Saturation (32 x 32) + 32 => 32 — MADDS

MADDS.U — Multiply-Add with Saturation Unsigned (32 x 32) + 32 => 32 — MADDS.U

Syntax:

```
madds    Dc, Dd, Da, Db (RRR)
madds    Dc, Dd, Da, const9 (RCR)
madds.u  Dc, Dd, Da, Db (RRR)
madds.u  Dc, Dd, Da, const9 (RCR)
```

Description

Multiply the contents of data register *Da* by the contents of data register *Db/const9*, add the result to the contents of data register *Dd*, and put the result in data register *Dc*. The operands are treated as signed/unsigned, 32-bit integers. The value *const9* is sign-extended/zero-extended to 32 bits before the multiplication is performed.

Overflow and advanced overflow are calculated on the final result. On overflow, the result is saturated.

Operation:

```
D[c][31:0] = D[d][31:0] + (D[a][31:0] * D[b][31:0]); signed; ssov
D[c][31:0] = D[d][31:0] + (D[a][31:0] * sign_ext(const9)); signed; ssov

D[c][31:0] = D[d][31:0] + (D[a][31:0] * D[b][31:0]); unsigned; suov
D[c][31:0] = D[d][31:0] + (D[a][31:0] * zero_ext(const9)); unsigned; suov
```

Status:

V, SV, AV, SAV

Examples:

```
madds d3, d4, d1, d2
madds d3, d1, d1,253
madds.u  d3, d4, d1, d2
madds.u  d3, d4, d1, 126
```

See Also:

MADD (pg 244), **MADDM** (pg 249), **MADDM.U** (pg 249), **MADDMS** (pg 254), **MADDMS.U** (pg 254)

MADDS.H **Packed Multiply-Add with Saturation (16 x 16) + 32 => 32 || (16 x 16) + 32 => 32** MADDS.H

Syntax:

madds.h Ec, Ed, Da, Db, n (RRR)

Description:

Multiply the most-significant halfword of data register *Da* by the most-significant halfword of data register *Db* and add the product, left-justified if $n = 1$, to the upper 32 bits of extended data register *Ed*. Put the result in the upper 32 bits of extended data register *Ec*.

Multiply the least-significant halfword of data register *Da* by the least-significant halfword of data register *Db* and add the product, left-justified if $n = 1$, to the lower 32 bits of extended data register *Ed*. Put the result in the lower 32 bits of extended data register *Ec*.

The operands are treated as signed values. Overflow and advanced overflow are calculated on each independent final result. On overflow, each result is independently saturated.

Operation:

```
E[c][63:32]  = E[d][63:32] + (((D[a][31:16]  * D[b][31:16]) << n)[31:0]);
E[c][31:0]   = E[d][31:0] + (((D[a][15:0]  * D[b][15:0]) << n)[31:0]);
signed, n = 0, 1; ssov
```

Status:

V, SV, AV, SAV

Example:

```
madds.h  e0, e4, d9, d2, 0
```

See Also:

MADDM.H (pg 250), **MADDM.Q** (pg 252), **MADDR.Q** (pg 257), **MADDRS.H** (pg 259), **MADDRS.Q** (pg 261), **MADDS.Q** (pg 265)

MADDS.Q Multiply-Add with Saturation (16 x 16) + 32 => 32 MADDS.Q

Syntax:

madds.q Dc, Dd, Da, Db, n (RRR)

Description:

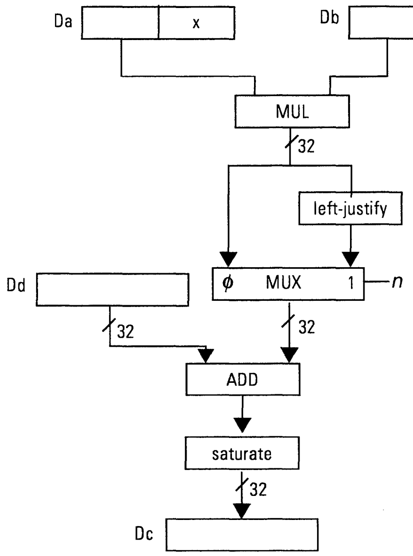
Multiply the most-significant halfword of data register *Da* by the most-significant half-word of data register *Db* and add the product, left-justified if *n* = 1, to the contents of data register *Dd*. Put the result in *Dc*. The operands are treated as signed values.

Overflow and advanced overflow are calculated on the final result. On overflow, the result is saturated.

If *n* = 1, 0x8000 * 0x8000 = 0x7FFF.FFFF.

Operation:

D[c][31:0] = D[d][31:0] + ((D[a][31:16] * D[b][31:16]) << n)[31:0]); signed, n = 0, 1;



TAM052.1

Status:

V, SV, AV, SAV

Example:

```
madds.q  d0,d7,d2,d3,0
```

See Also:

MADD.Q (pg 247), **MADDM.Q** (pg 252), **MADDR.Q** (pg 257), **MADDRS.Q** (pg 261)

MAX Maximum Value MAX

Syntax:

max Dc, Da, Db (RR)
max Dc, Da, const9 (RC)

Description:

If the contents of data register Da are greater than the contents of data register Db/const9, put the contents of Da in data register Dc; otherwise, put the contents of Db/const9 in Dc. The operands are treated as signed, 32-bit integers.

Operation:

```
if (D[a] > D[b]) then D[c] = D[a]
else D[c] = D[b]; signed

if (D[a] > sign_ext(const9)) then D[c] = D[a]
else D[c] = sign_ext(const9); signed
```

Examples:

```
max d3, d1, d2
max d3, d1, 126
max d3, d1, 253
```

See Also:

MAX.U (pg 270), **MIN** (pg 272), **MIN.U** (pg 275)

MAX.B Maximum Value Packed Byte Signed MAX.B
MAX.BU Maximum Value Packed Byte Unsigned MAX.BU

Syntax:

max.b Dc, Da, Db (RR)
max.bu Dc, Da, Db (RR)

Description:

Compute the maximum value of the corresponding bytes in *Da* and *Db* and put each result in the corresponding byte of *Dc*. The operands are treated as signed/unsigned, 8-bit integers.

Refer also to Section 8.1.3, "Packed Arithmetic," on page 95

Operation:

```
if (D[a][(n+7):n] > D[b][(n+7):n])
then D[c][(n+7):n] = D[a][(n+7):n]
else D[c][(n+7):n] = D[b][(n+7):n]; n = 0, 8, 16, 24; signed

if (D[a][(n+7):n] > D[b][(n+7):n])
then D[c][(n+7):n] = D[a][(n+7):n]
else D[c][(n+7):n] = D[b][(n+7):n]; n = 0, 8, 16, 24; unsigned
```

Examples:

```
max.b    d3, d1, d2
max.bu   d3, d1, d2
```

See Also:

MAX.H (pg 269), **MAX.HU** (pg 269), **MIN.B** (pg 273), **MIN.BU** (pg 273), **MIN.H** (pg 274), **MIN.HU** (pg 274)

MAX.H Maximum Value Packed Halfword Signed MAX.H
MAX.HU Maximum Value Packed Halfword Unsigned MAX.HU

Syntax:

```
max.h    Dc, Da, Db (RR)
max.hu   Dc, Da, Db (RR)
```

Description:

Compute the maximum value of the corresponding halfwords in *Da* and *Db* and put each result in the corresponding byte of *Dc*. The operands are treated as signed/unsigned, 16-bit integers.

Refer also to Section 8.1.3, "Packed Arithmetic," on page 95.

Operation:

```
if (D[a][(n+15):n] > D[b][(n+15):n])
then D[c][(n+15):n] = D[a][(n+15):n]
else D[c][(n+15):n] = D[b][(n+15):n]; n = 0, 16; signed

if (D[a][(n+15):n] > D[b][(n+15):n])
then D[c][(n+15):n] = D[a][(n+15):n]
else D[c][(n+15):n] = D[b][(n+15):n]; n = 0, 16; unsigned
```

Examples:

```
max.h    d3, d1, d2
max.hu   d3, d1, d2
```

See Also:

MAX.B (pg 268), **MAX.BU** (pg 268), **MIN.B** (pg 273), **MIN.BU** (pg 273), **MIN.H** (pg 274), **MIN.HU** (pg 274)

MAX.U Maximum Value Unsigned MAX.U

Syntax:

max.u	Dc, Da, Db (RR)
max.u	Dc, Da, const9 (RC)

Description:

If the contents of data register Da are greater than the contents of data register Db/const9, put the contents of Da in data register Dc; otherwise, put the contents of Db/const9 in Dc. The operands are treated as unsigned, 32-bit integers.

Operation:

```
if (D[a] > D[b]) then D[c] = D[a]
else D[c] = D[b]; unsigned

if (D[a] > zero_ext(const9)) then D[c] = D[a]
else D[c] = zero_ext(const9); unsigned
```

Examples:

```
max.u d3, d1, d2
max.u d3, d1, 126
max.u d3, d1, 253
```

See Also:

MAX (pg 267), **MIN** (pg 272), **MIN.U** (pg 275)

MFCR Move From Core Register MFCR

Syntax:

mfcr Dc, const16 (RLC)

Description:

Move the contents of the core SFR register, selected by the value *const16*, to data register *Dc*. The core SFR address is a *const16* byte offset from the core SFR base address. It must be word-aligned (the least-significant two bits equal zero). Non-aligned addresses have an undefined effect. This instruction can be executed on any privilege level.

Refer also to Chapter 3, "Core Registers," on page 27.

Operation:

D[c] = CR[const16]

Example:

```
mfcr  d3, 12
```

See Also:

MTCR (pg 307)

MIN Minimum Value MIN

Syntax:

min	Dc, Da, Db (RR)
min	Dc, Da, const9 (RC)

Description:

If the contents of data register Da are less than the contents of data register Db/const9, put the contents of Da in data register Dc; otherwise, put the contents of Db/const9 in Dc. The operands are treated as signed, 32-bit integers.

Operation:

```
if (D[a] < D[b]) then D[c] = D[a]
else D[c] = D[b]; signed

if (D[a] < sign_ext(const9)) then D[c] = D[a]
else D[c] = sign_ext(const9); signed
```

Examples:

```
min   d3, d1, d2
min   d3, d1, 126
min   d3, d1, 253
```

See Also:

MAX (pg 267), **MAX.U** (pg 270), **MIN** (pg 272)

MIN.B Minimum Value Packed Byte Signed MIN.B
MIN.BU Minimum Value Packed Byte Unsigned MIN.BU

Syntax:

```
min.b    Dc, Da, Db (RR)
min.bu   Dc, Da, Db (RR)
```

Description:

Compute the minimum value of the corresponding bytes in *Da* and *Db* and put each result in the corresponding byte of *Dc*. The operands are treated as signed/unsigned, 8-bit integers.

Refer also to Section 8.1.3, "Packed Arithmetic," on page 95.

Operation:

```
if (D[a][(n+7):n] < D[b][(n+7):n])
then D[c][(n+7):n] = D[a][(n+7):n]
else D[c][(n+7):n] = D[b][(n+7):n];n = 0, 8, 16, 24; signed

if (D[a][(n+7):n] < D[b][(n+7):n])
then D[c][(n+7):n] = D[a][(n+7):n]
else D[c][(n+7):n] = D[b][(n+7):n]; n = 0, 8, 16, 24; unsigned
```

Examples:

```
min.b  d3, d1, d2
min.bu  d3, d1, d2
```

See Also:

MAX.B (pg 268), **MAX.BU** (pg 268), **MIN.H** (pg 274), **MIN.HU** (pg 274)

MIN.H — Minimum Value Packed Halfword Signed — MIN.H
MIN.HU — Minimum Value Packed Halfword Unsigned — MIN.HU

Syntax:

```
min.h     Dc, Da, Db (RR)
min.hu    Dc, Da, Db (RR)
```

Description:

If the contents of each byte of data register Da are less than the contents of data register Db, copy the contents to data register Dc.; otherwise, copy Db to Dc. The operands are treated as signed/unsigned, 16-bit integers.

Refer also to Section 8.1.3, "Packed Arithmetic," on page 95.

Operation:

```
if (D[a][(n+15):n] < D[b][(n+15):n])
then D[c][(n+15):n] = D[a][(n+15):n]
else D[c][(n+15):n] = D[b][(n+15):n]; n = 0, 16; signed

if (D[a][(n+15):n] < D[b][(n+15):n])
then D[c][(n+15):n] = D[a][(n+15):n]
else D[c][(n+15):n] = D[b][(n+15):n]; n = 0, 16; unsigned
```

Examples:

```
min.h     d3, d1, d2
min.hu    d3, d1, d2
```

See Also:

MAX.H (pg 269), **MAX.HU** (pg 269), **MIN.B** (pg 273), **MIN.BU** (pg 273)

MIN.U Minimum Value Unsigned MIN.U

Syntax:

min.u Dc, Da, Db (RR)
min.u Dc, Da, const9 (RC)

Description:

If the contents of data register Da are less than the contents of data register Db/const9, put the contents of Da in data register Dc; otherwise, put the contents of Db/const9 in Dc. The operands are treated as unsigned, 32-bit integers.

Operation:

```
if (D[a] < D[b]) then D[c] = D[a]
else D[c] = D[b]; unsigned

if (D[a] < zero_ext(const9)) then D[c] = D[a]
else D[c] = zero_ext(const9); unsigned
```

Examples:

```
min.u d3, d1, d2
min.u d3, d1, 126
min.u d3, d1, 253
```

See Also:

MAX (pg 267), **MAX.U** (pg 270), **MIN** (pg 272)

MOV Move MOV

Syntax:

mov	Dc, Db (RR)
mov	Dc, const16 (RLC)
mov	Da, Db (SRR)
mov	Da, const4 (SRC)
mov	D15, const8 (SC)

Description:

Move the contents of data register *Db/const16* to data register *Dc*. The operands are treated as 32-bit integers. The value *const16* is sign-extended to 32 bits before it is moved.

Move the contents of data register *Db/const4/const8* to data register *Da*/D15. The operands are treated as 32-bit integers. The value *const4* is sign-extended to 32 bits before it is moved. The value *const8* is zero-extended to 32 bits before it is moved.

Operation:

```
D[c] = D[b]
D[c] = sign_ext(const16)
D[a] = D[b]
D[a] = sign_ext(const4)
D[15] = zero__ext(const8)
```

Examples:

```
mov d3, d1
mov d3, -14, 526
mov d1, d2
mov d1, 6
mov d15, 126
```

See Also:

MOV.U (pg 280), **MOVH** (pg 281)

MOV.A Move Address from Data Register MOV.A

Syntax:

mov.a	Ac, Db (RR)
mov.a	Aa, Db (SRR)

Description:

Move the contents of data register *Db* to address register *Ac*. The operands are treated as 32-bit integers.

Move the contents of data register *Db* to address register *Aa*. The operands are treated as 32-bit integers.

Operation:

A[c] = D[b]
A[a] = D[b]

Examples:

```
mov.a     a3, d1
mov.a     a4, d2
```

See Also:

LEA (pg 236), **MOV.AA** (pg 278), **MOV.D** (pg 279), **MOVH.A** (pg 282), **MOVZ.A** (pg 283)

MOV.AA Move Address from Address Register MOV.AA

Syntax:

mov.aa Ac, Ab (RR)
mov.aa Aa, Ab (SRR)

Description:

Move the contents of address register *Ab* to address register *Ac*. The operands are treated as 32-bit integers.

Move the contents of address register *Ab* to address register *Aa*. The operands are treated as 32-bit integers.

Operation:

A[c] = A[b]
A[a] = A[b]

Examples:

```
mov.aa   a3, a4
mov.aa   a4, a2
```

See Also:

LEA (pg 236), **MOV.A** (pg 277), **MOV.D** (pg 279), **MOVH.A** (pg 282), **MOVZ.A** (pg 283)

MOV.D Move Address to Data Register MOV.D

Syntax:

mov.d Dc, Ab (RR)
mov.d Da, Ab (SRR)

Description:

Move the contents of address register *Ab* to data register *Dc*. The operands are treated as 32-bit integers.

Move the contents of address register *Ab* to data register *Da*. The operands are treated as 32-bit integers.

Operation:

D[c] = A[b]
D[a] = A[b]

Examples:

```
mov.d d3, a4
mov.d d1, a2
```

See Also:

LEA (pg 236), **MOV.A** (pg 277), **MOV.AA** (pg 278), **MOVH.A** (pg 282), **MOVZ.A** (pg 283)

MOV.U Move Unsigned MOV.U

Syntax:

mov.u Dc, const16 (RLC)

Description:

Move the zero-extended value *const16* to data register *Dc*.

Operation:

D[c] = zero_ext(const16)

Example:

```
mov.u d3, 526
```

See Also:

MOV (pg 276), **MOVH** (pg 281)

MOVH Move Halfword MOVH

Syntax:

movh Dc, const16 (RLC)

Description:

Move the value *const16* to the most-significant halfword of data register *Dc* and set the least-significant 16 bits to zero.

Operation:

D[c] = {const16, 16'h 0000}

Example:

```
movh  d3, 526
```

See Also:

MOV (pg 276), **MOV.U** (pg 280)

MOVH.A Move High to Address MOVH.A

Syntax:

movh.a Ac, const16 (RLC)

Description:

Move the value *const16* to the most-significant halfword of address register *Ac* and set the least-significant 16 bits to zero.

Operation:

A[c] = {const16, 16'h 0000}

Example:

```
movh.a   a3, 526
```

See Also:

LEA (pg 236), **MOV.A** (pg 277), **MOV.AA** (pg 278), **MOV.D** (pg 279), **MOVZ.A** (pg 283)

MOVZ.A Move Zero To Address MOVZ.A

Syntax:

movz.a Aa (SR)

Description:

Move the value 0 to address register *Aa*

Operation:

A[a] = 0

Example:

```
movz.a a4
```

See Also:

LEA (pg 236), **MOV.A** (pg 277), **MOV.AA** (pg 278), **MOV.D** (pg 279), **MOVH.A** (pg 282)

MSUB Multiply-Subtract 32 – (32 x 32) => 32 MSUB

Syntax:

msub	Dc, Dd, Da, Db (RRR)
msub	Dc, Dd, Da, const9 (RCR)

Description:

Multiply the contents of data register *Da* by the contents of data register *Db/const9*, subtract the result from the contents of data register *Dd*, and put the result in data register *Dc*. The operands are treated as signed, 32-bit integers. The value *const9* is sign-extended to 32 bits before the multiplication is performed.

Overflow and advanced overflow are calculated on the final result.

Operation:

```
D[c][31:0] = D[d][31:0] – (D[a][31:0] * D[b][31:0]);
D[c][31:0] = D[d][31:0] – (D[a][31:0] * sign_ext(const9));
```

Status:

V, SV, AV, SAV

Examples:

```
msub  d3, d4, d1, d2
msub  d3, d4, d1, 126
```

See Also:

MSUBM (pg 289), **MSUBM.U** (pg 289), **MSUBMS** (pg 294), **MSUBMS.U** (pg 294), **MSUBS** (pg 303), **MSUBS.U** (pg 303)

MSUB.H Packed Multiply-Subtract MSUB.H

32 – (16 x 16) => 32 || 32 – (16 x 16) => 32

Syntax:

msub.h Ec, Ed, Da, Db, n (RRR)

Description:

Multiply the most-significant halfword of data register *Da* by the most-significant halfword of data register *Db* and subtract the product, left-justified if *n* = 1, from the upper 32 bits of extended data register *Ed*. Put the result in the upper 32 bits of extended data register *Ec*.

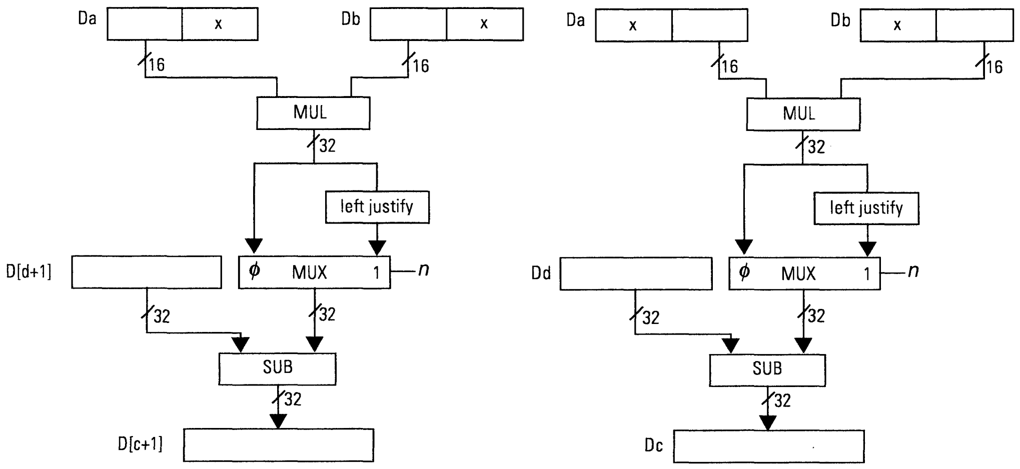
Multiply the least-significant halfword of data register *Da* by the least-significant halfword of data register *Db* and subtract the product, left-justified if *n* = 1, from the lower 32 bits of extended data register *Ed*. Put the result in the lower 32 bits of extended data register *Ec*.

The operands are treated as signed values. Overflow and advanced overflow are calculated on each independent final result.

Operation:

E[c][63:32] = E[d][63:32] – (((D[a][31:16] * D[b][31:16]) << n)[31:0])

E[c][31:0] = E[d][31:0] – (((D[a][15:0] * D[b][15:0]) << n)[31:0])
signed, n = 0, 1



Status:

V, SV, AV, SAV

Example:

```
msub.h e4, e0, d11, d2, 0
```

See Also:

MADDM.Q (pg 252), **MADDR.Q** (pg 257), **MADDRS.Q** (pg 261), **MADDS.Q** (pg 265)

MSUB.Q Multiply-Subtract in Q Format 32 – (16 x 16) => 32 MSUB.Q

Syntax:

msub.q Dc, Dd, Da, Db, n (RRR)

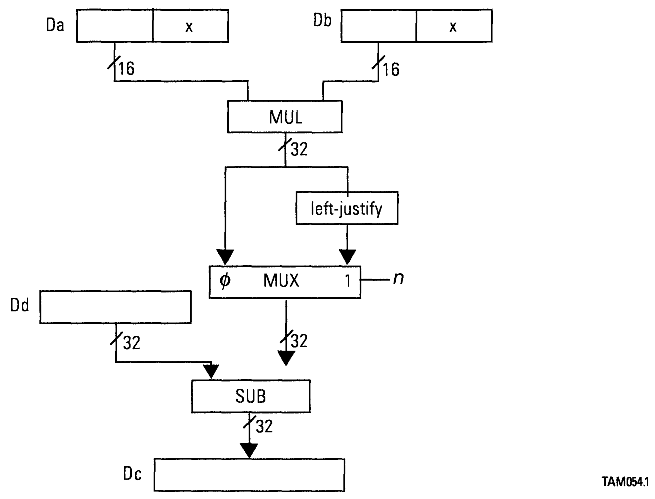
Description:

Multiply the most-significant halfword of data register *Da* by the most-significant halfword of data register *Db* and subtract the product, left-justified if *n* = 1, from the contents of data register *Dd*. Put the result in *Dc*. The operands are treated as signed values. Overflow and advanced overflow are calculated on the final result.

If *n* = 1, 0x8000 * 0x8000 = 0x7FFF.FFFF.

Operation:

D[c][31:0] = D[d][31:0] – ((D[a][31:16] * D[b][31:16]) << n)[31:0]; signed; n = 0, 1



Status:

V, SV, AV, SAV

Example:

```
msub.q d0, d0, d2, d2, 1
```

See Also:

MSUBR.Q (pg 297), **MSUBM.Q** (pg 292), **MSUBRS.Q** (pg 301), **MSUBS.Q** (pg 305)

MSUBM — Multiply-Subtract Multiword 64 – (32 x 32) => 64 — MSUBM

MSUBM.U — Multiply-Subtract Multiword Unsigned 64 – (32 x 32) => 64 — MSUBM.U

Syntax:

```
msubm   Ec, Ed, Da, Db (RRR)
msubm   Ec, Ed, Da, const9 (RCR)
msubm.u Ec, Ed, Da, Db (RRR)
msubm.u Ec, Ed, Da, const9 (RCR)
```

Description:

Multiply the contents of data register *Da* by the contents of data register *Db/const9*, subtract the 64-bit result from the contents of extended data register *Ed*, and put the result in extended data register *Ec*. The operands are treated as signed/unsigned integers. The value *const9* is sign-extended/zero-extended to 32 bits before the multiplication is performed. Overflow and advanced overflow are calculated on the final result.

Operation:

```
E[c][63:0] = E[d][63:0] – ((D[a][31:0] * D[b][31:0])[63:0]); signed
E[c][63:0] = E[d][63:0] – ((D[a][31:0] * sign_ext(const9))[63:0]); signed

E[c][63:0] = E[d][63:0] – ((D[a][31:0] * D[b][31:0])[63:0]); unsigned
E[c][63:0] = E[d][63:0] – ((D[a][31:0] * zero_ext(const9))[63:0]); unsigned
```

Status:

V, SV, AV, SAV

Examples:

```
msubm e0, e4, d3, d2
msubm e0, e4, d14, 126
msubm.u  e10, e4, d1, d2
msubm.u  e0, e4, d10, 0x16
```

See Also:

MSUB (pg 284), **MSUBMS** (pg 294), **MSUBMS.U** (pg 294), **MSUBS** (pg 303), **MSUBS.U** (pg 303)

MSUBM.H Packed Multiply-Subtract with Multiword Result 64 – ((16 x 16) + (16 x 16)) => 64 MSUBM.H

Syntax:

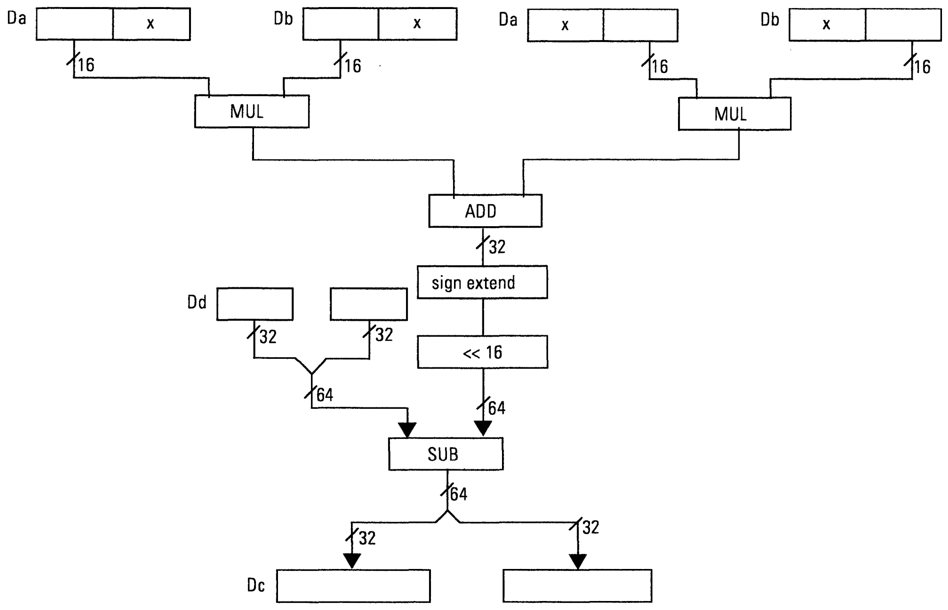
msubm.h Ec, Ed, Da, Db (RRR)

Description:

Multiply the most-significant halfword of data register *Da* by the most-significant halfword of data register *Db*. Multiply the least-significant halfword of data register *Da* by the least-significant halfword of data register *Db*. Add the products of the two multiplications, sign-extend the result to 64 bits, and left-shift by 16. Subtract that value from extended data register pair *Ed* and put the 64-bit result into extended data register *Ec*. The operands are treated as signed values. Overflow and advanced overflow are calculated on the final result.

Operation:

E[c] [63:0] = E[d][63:0] – ((D[a][31:16] * D[b][31:16]) + (D[a] [15:0] * D[b] [15:0])<<16); signed



Status:

V, SV, AV, SAV

Example:

```
msubm.h  e12, e4, d1, d2
```

See Also:

MADD.Q (pg 247), **MADDR.Q** (pg 257), **MADDRS.Q** (pg 261), **MADDS.Q** (pg 265)

MSUBM.Q Multiply-Subtract Multiword 64 – (16 x 16) => 64 MSUBM.Q

Syntax:

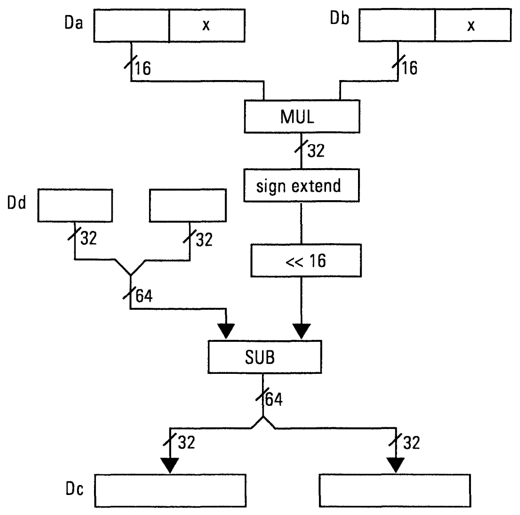
msubm.q Ec, Ed, Da, Db (RRR)

Description:

Multiply the most-significant halfword of data register *Da* by the most-significant halfword of data register *Db,* sign-extend the result to 64 bits, and left-shift by 16. Subtract that value from extended data register *Ed* and put the 64-bit result into extended data register *Ec.* The operands are treated as signed values. Overflow and advanced overflow are calculated on the final result.

Operation:

E[c] [63:0] = E[d] [63:0] – sign_ext(D[a][31:16] * D[b][31:16]) << 16 ; signed



TAM056.1

Status:

V, SV, AV, SAV

Example:

```
msubm.q  e12, e4, d1, d2
```

See Also:

MSUB.Q (pg 287), **MSUBR.Q** (pg 297), **MSUBRS.Q** (pg 301), **MSUBS.Q** (pg 305)

MSUBMS — Multiply-Subtract with Saturation 64 – (32 x 32) => 64 — MSUBMS

MSUBMS.U — Multiply-Subtract Unsigned with Saturation 64 – (32 x 32) => 64 — MSUBMS.U

Syntax:

msubms Ec, Ed, Da, Db (RRR)
msubms Ec, Ed, Da, const9 (RCR)
msubms.uEc, Ed, Da, Db (RRR)
msubms.u Ec, Ed, Da, const9 (RCR)

Description:

Multiply the contents of data register *Da* by the contents of data register *Db/const9*, subtract the 64-bit result from the contents of extended data register *Ed*, and put the result in extended data register *Ec*. The operands are treated as signed/unsigned integers. The value *const9* is sign-extended/zero-extended to 32 bits before the multiplication is performed. Overflow and advanced overflow are calculated on the final result. On overflow, the result is saturated.

Operation:

E[c][63:0] = E[d][63:0] – ((D[a][31:0] * D[b][31:0])[63:0]); signed; ssov
E[c][63:0] = E[d][63:0] – ((D[a][31:0] * sign_ext(const9))[63:0]); signed; ssov

E[c][63:0] = E[d][63:0] – ((D[a][31:0] * D[b][31:0])[63:0]); unsigned; suov
E[c][63:0] = E[d][63:0] – ((D[a][31:0] * zero_ext(const9))[63:0]); unsigned; suov

Status:

V, SV, AV, SAV

Examples:

```
msubms   e10, e4, d1, d2
msubms   e0, e4, d10, 0x100
msubms.u    e10, e4, d1, d2
msubms.u    e0, e4, d10, 0xFF
```

See Also:

MSUB (pg 284), **MSUBM.U** (pg 289), **MSUBS** (pg 303), **MSUBS.U** (pg 303)

MSUBR.H Packed Multiply-Subtract with Rounding MSUBR.H

32 – (16 x 16) => 16 MSB
32 – (16 x 16) => 16 LSB

Syntax:

msubr.h Dc, Ed, Da, Db, n (RRR)

Description:

Multiply the most-significant halfword of data register *Da* by the most-significant halfword of data register *Db* and subtract the product, left-justified if *n* = 1, from the upper 32 bits of extended data register *Ed*. Round the contents and put the most-significant halfword of the rounded value in the most-significant halfword of *Dc*.

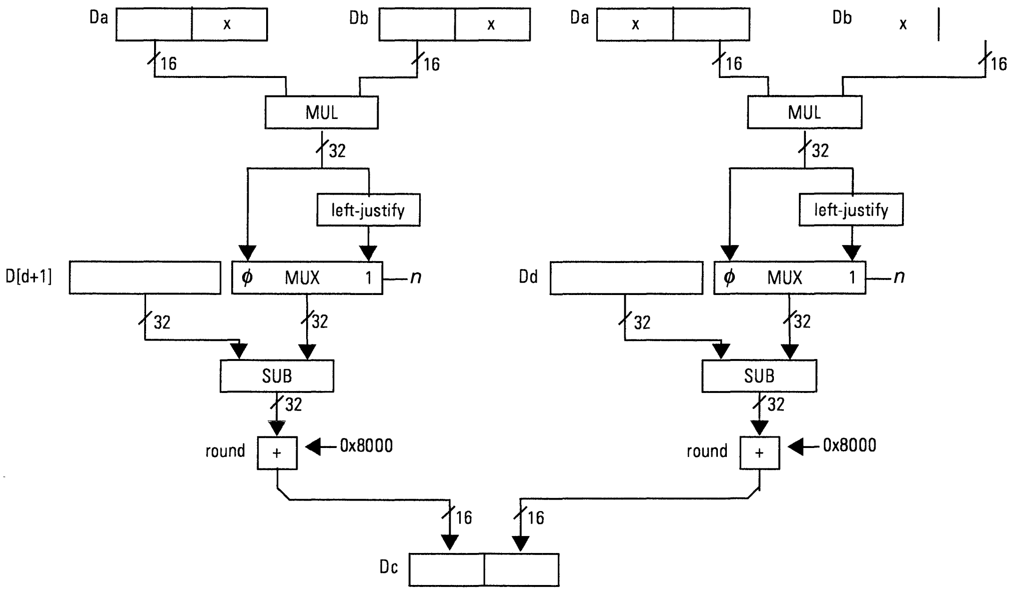
Multiply the least-significant halfword of data register *Da* by the least-significant halfword of data register *Db* and subtract the product, left-justified if *n* = 1, from the lower 32 bits of extended data register *Ed*. Round the contents and put the least-significant halfword of the rounded value in the least-significant halfword of *Dc*.

The operands are treated as signed values. Overflow and advanced overflow are calculated on each independent final result.

If *n* = 1, 0x8000 * 0x8000 = 0x7FFF.FFFF.

Operation:

```
D[c][31:16] = round16(E[d] [63:32] – ((D[a] [31:16]  * D[b] [31:16]) << n)[31:16])
D[c][15:0] = round16(E[d] [31:0] – ((D[a] [15:0]  * D[b] [15:0]) << n)[15:0]); signed; n = 0, 1
```



Status:

V, SV, AV, SAV

Example:

```
msubr.h  d10, e4, d1, d2, 1
```

See Also:

MADD.Q (pg 247), **MADDM.H** (pg 250), **MADDM.Q** (pg 252), **MADDRS.H** (pg 259), **MADDRS.Q** (pg 261), **MADDS.H** (pg 264), **MADDS.Q** (pg 265)

MSUBR.Q — Multiply-Subtract with Rounding 32 – (16 x 16) => 16 — MSUBR.Q

Syntax:

msubr.q Dc, Dd, Da, Db, n (RRR)

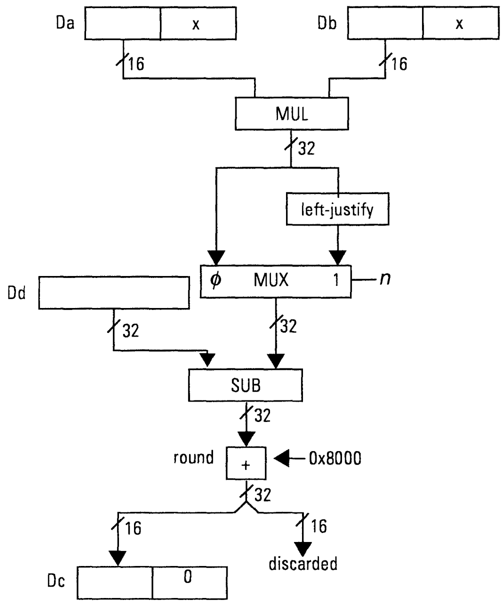
Description:

Multiply the most-significant halfword of data register *Da* by the most-significant halfword of data register *Db* and subtract the product, left-justified if *n* = 1, from data register *Dd.* Round the result, put the most-significant halfword of the rounded value in *Dc,* and set the least-significant halfword of *Dc* to 0. The operands are treated as signed values. Overflow and advanced overflow are calculated on the final result.

If *n* = 1, 0x8000 * 0x8000 = 0x7FFF.FFFF.

Operation:

D[c][31:0] = round16(D[d] – ((D[a] [31:16] * D[b] [31:g16]) << n)[31:0])[31:16], 16'h 0000;
n = 0, 1; signed



TAM058.1

Status:

V, SV, AV, SAV

Example:

```
msubr.q  d3, d4, d1, d2, 1
```

See Also:

MSUB.Q (pg 287), **MSUBM.Q** (pg 292), **MSUBRS.Q** (pg 301), **MSUBS.Q** (pg 305)

MSUBRS.H Packed Multiply-Subtract with Rounding and Saturation 32 – (16 x 16) => 16 MSB 32 – (16 x 16) => 16 LSB MSUBRS.H

Syntax:

msubrs.h Dc, Ed, Da, Db, n (RRR)

Description:

Multiply the most-significant halfword of data register *Da* by the most-significant halfword of data register *Db* and subtract the product, left-justified if *n* = 1, from the upper 32 bits of extended data register *Ed*. Round the contents and put the most-significant halfword of the rounded value into the most-significant halfword of *Dc*.

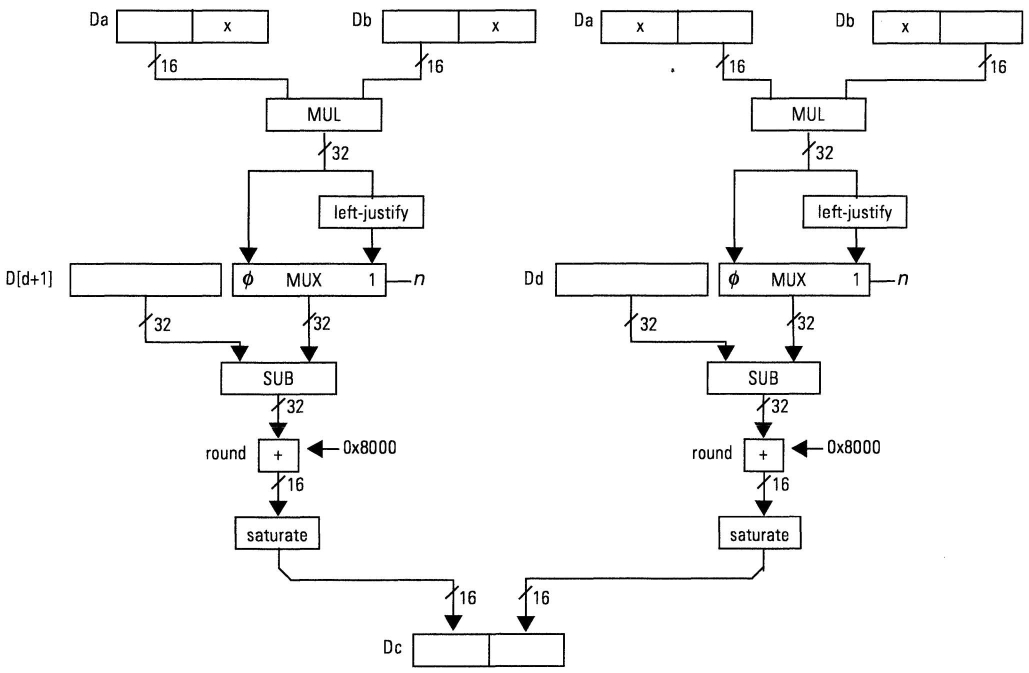
Multiply the least-significant halfword of data register *Da* by the least-significant halfword of data register *Db* and subtract the product, left-justified if *n* = 1, from the lower 32 bits of extended data register *Ed*. Round the contents and put the least-significant halfword of the rounded value into the least-significant halfword of *Dc*.

The operands are treated as signed values. Overflow and advanced overflow are calculated on each independent final result. On overflow, each result is independently saturated.

If *n* = 1, 0x8000 * 0x8000 = 0x7FFF.FFFF.

Operation:

```
D[c][31:16] = round16(E[d] [63:32] – ((D[a] [31:16]  * D[b] [31:16]) << n)[31:16]);
D[c][15:0] = round16(E[d] [3:0] – ((D[a] [15:0]  * D[b] [15:0]) << n)[15:0]);
signed; n = 0, 1; ssov
```



Status:

V, SV, AV, SAV

Example:

```
msubr.h  d10, e4, d1, d2, 1
```

See Also:

MADD.Q (pg 247), **MADDM.H** (pg 250), **MADDM.Q** (pg 252), **MADDRS.Q** (pg 261), **MADDS.Q** (pg 265)

MSUBRS.Q Multiply-Subtract with Rounding and Saturation 32 – (16 x 16) => 16 MSUBRS.Q

Syntax:

msubrs.q Dc, Dd, Da, Db, n (RRR)

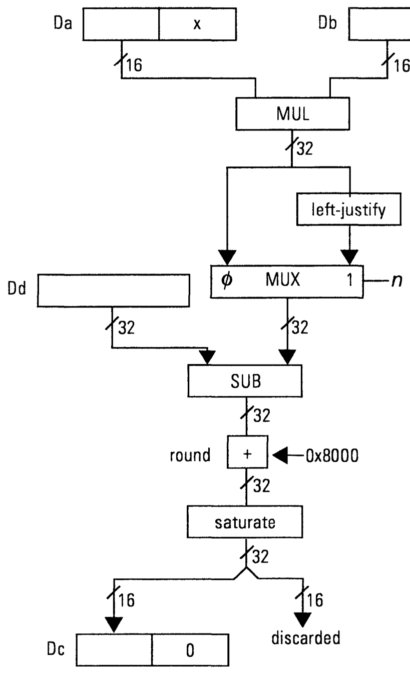
Description:

Multiply the most-significant halfword of data register *Da* by the most-significant halfword of data register *Db* and subtract the product, left-justified if *n* = 1, from data register *Dd*. Round the result, put the most-significant halfword of the rounded value in *Dc*, and set the least-significant halfword of *Dc* to 0. Overflow and advanced overflow are calculated on the final result. On overflow, the result is saturated.

If *n* = 1, 0x8000 * 0x8000 = 0x7FFF.FFFF.

Operation:

D[c][31:0] = round16(D[d][31:0] – ((D[a] [31:16] * D[b] [31:16]) << n)[31:0] [31:16], 16'h 0000; n = 0, 1; signed, ssov



TAM059.1

◆ PRELIMINARY EDITION ◆

Status:

V, SV, AV, SAV

Example:

```
msubrs.q d3, d4, d1, d2, 1
```

See Also:

MSUB.Q (pg 287), **MSUBM.Q** (pg 292), **MSUBR.Q** (pg 297), **MSUBS.Q** (pg 305)

MSUBS — Multiply-Subtract with Saturation 32 – (32 x 32) => 32 — MSUBS

MSUBS.U — Multiply-Subtract with Saturation Unsigned 32 – (32 x 32) => 32 — MSUBS.U

Syntax:

```
msubs    Dc, Dd, Da, Db (RRR)
msubs    Dc, Dd, Da, const9 (RCR)
msubs.u  Dc, Dd, Da, Db (RRR)
msubs.u  Dc, Dd, Da, const9 (RCR)
```

Description:

Multiply the contents of data register *Da* by the contents of data register *Db/const9*, subtract the result from the contents of data register *Dd*, and put the result in data register *Dc*. The operands are treated as signed/unsigned, 32-bit integers. The value *const9* is sign-extended/zero-extended to 32 bits before the multiplication is performed. Overflow and advanced overflow are calculated on the final result. On overflow, the result is saturated.

Operation:

```
D[c][31:0] = D[d][31:0] – (D[a][31:0] * D[b][31:0]); signed, ssov
D[c][31:0] = D[d][31:0] – (D[a][31:0] * sign_ext(const9)); signed, ssov

D[c][31:0] = D[d][31:0] – (D[a][31:0] * D[b][31:0]); unsigned, ssov
D[c][31:0] = D[d][31:0] – (D[a][31:0] * zero_ext(const9)); unsigned, ssov
```

Status:

V, SV, AV, SAV

Examples:

```
msubs d3, d4, d1, d2
msubs d3, d4, d1, 126
msubs.u d3, d4, d1, d2
msubs.u d3, d4, d1, 126
```

See Also:

MSUB (pg 284), **MSUBM.U** (pg 289), **MSUBMS** (pg 294), **MSUBMS.U** (pg 294)

MSUBS.H — Packed Multiply-Subtract with Saturation — MSUBS.H

32 – (16 x 16) => 32 || 32 – (16 x 16) => 32

Syntax:

msubs.h Ec, Ed, Da, Db, n (RRR)

Description:

Multiply the most-significant halfword of data register *Da* by the most-significant half-word of data register *Db* and subtract the product, left-justified if *n* = 1, from the upper 32 bits of extended data register *Ed*. Put the result in the upper 32 bits of extended data register *Ec*.

Multiply the least-significant halfword of data register *Da* by the least-significant halfword of data register *Db* and subtract the product, left-justified if *n* = 1, from the lower 32 bits of extended data register *Ed*. Put the result in the lower 32 bits of extended data register *Ec*.

The operands are treated as signed values. Overflow and advanced overflow are calculated on each independent final result. On overflow, each result is independently saturated.

Operation:

```
E[c][63:32] = E[d][63:32] – (((D[a][31:16]  * D[b][31:16]) << n)[31:0])

E[c][31:0] = E[d][31:0] – (((D[a][15:0]  * D[b][15:0]) << n)[31:0])
signed, n = 0, 1
```

Status:

V, SV, AV, SAV

Example:

```
msubs.h e4, e0, d11, d2, 0
```

See Also:

MSUB (pg 284)

MSUBS.Q **Multiply-Subtract with Saturation 32 – (16 x 16) => 32** MSUBS.Q

Syntax:

msubs.u Dc, Dd, Da, Db, n (RRR)

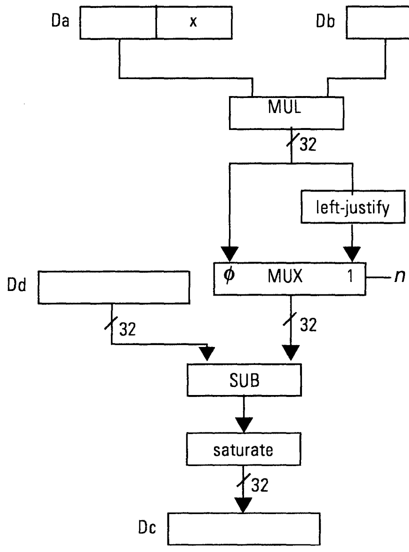
Description:

Multiply the most-significant halfword of data register *Da* by the most-significant halfword of data register *Db* and subtract the product, left-justified if *n* = 1, from the contents of data register *Dd*. Put the result in *Dc*. The operands are treated as signed values. Overflow and advanced overflow are calculated on the final result. On overflow, the result is saturated.

If *n* = 1, 0x8000 * 0x8000 = 0x7FFF.FFFF.

Operation:

D[c][31:0] = D[d][31:0] – ((D[a][31:16] * D[b][31:16]) << n)[31:0]; signed; n = 0, 1; ssov



TAM060.1

Status:

V, SV, AV, SAV

Example:

```
msubs.q  d3, d4, d1, d2, 1
```

See Also:

MSUB.Q (pg 287), **MSUBM.Q** (pg 292), **MSUBR.Q** (pg 297), **MSUBRS.Q** (pg 301)

MTCR Move To Core Register MTCR

Syntax:

mtcr const16, Da (RLC)

Description:

Move the value in data register Da to the core SFR register selected by the value *const16*. The core SFR address is a const16 byte offset from the core SFR base address. It must be word-aligned (the least-significant two bits are zero). Non-aligned address have an undefined effect.

This instruction can be executed in supervisor mode only.

Refer also to Chapter 3, "Core Registers," on page 27.

Operation:

CR[const16] = D[a]

Example:

```
mtcr  12, d1
```

See Also:

BISR (pg 154), **DISABLE** (pg 177), **ENABLE** (pg 185), **MFCR** (pg 271), **RSTV** (pg 340)

MUL — Multiply 32 x 32 => 32 — MUL

Syntax:

```
mul     Dc, Da, Db (RR)
mul     Dc, Da, const9 (RC)
mul     Da, Db (SRR)
```

Description:

Multiply the contents of data register *Da* by the contents of data register *Db*/*const9* and put the result in data register *Dc*. The operands are treated as 32-bit integers. The value *const9* is sign-extended to 32 bits before the multiplication is performed.

Multiply the contents of data register *Da* by the contents of data register *Db* and put the result in data register *Da*. The operands are treated as 32-bit integers.

Operation:

```
D[c][31:0] = D[a][31:0] * D[b][31:0]
D[c][31:0] = D[a][31:0] * const9
D[a] = D[a][31:0] * D[b][31:0]
```

Status:

V, SV, AV, SAV

Examples:

```
mul   d3, d1, d2
mul   d3, d1, 126
mul   d1, d2
```

See Also:

MULS (pg 318), **MULS.U** (pg 318), **MULM** (pg 312), **MULM.U** (pg 312)

MUL.H Packed Multiply Q Format 16 x 16 => 32 MUL.H

Syntax:

mul.h Ec, Ed, Da, Db, n (RR)

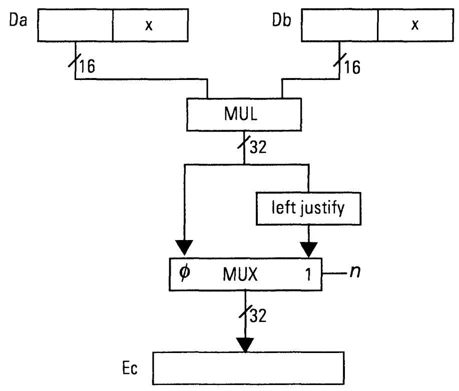
Description:

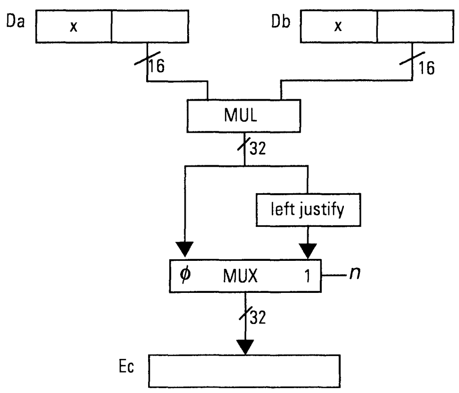
Multiply the most-significant halfword of data register *Da* by the most-significant halfword of data register *Db*. Put the result, left-justified if *n* = 1, in the most-significant 32 bits of extended data register *Ec*. Multiply the least-significant halfword of data register *Da* by the least-significant halfword of data register *Db*. Put the result, left-justified, if *n* = 1, in the least-significant 32 bits of extended data register *Ec*. The operands are treated as signed values.

If n = 1, 0x8000 x 0x8000 = 0x7FFFFFFF.

Operation:

```
E[c][63:32] = ((D[a][31:16] * D[b][31:16]) << n)[31:0])
E[c][31:0]  = ((D[a][15:0]  * D[b][15:0]) << n)[31:0]);
signed, n = 0, 1
```





Status:

V, SV, AV, SAV

Examples:

```
mul.h d3, d1, d2
```

See Also:

MUL (pg 308), **MULS** (pg 318), **MULS.U** (pg 318), **MULM** (pg 312), **MULM.U** (pg 312)

MUL.Q — Multiply (16 x 16) => 32 — MUL.Q

Syntax:

mul.q Dc, Da, Db, n (RR)

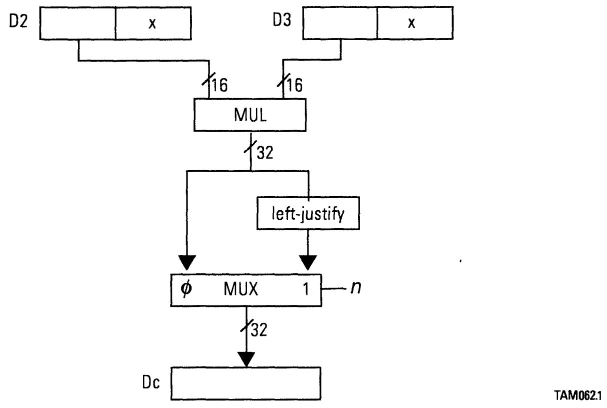
Description:

Multiply the most-significant halfword of data register *Da* by the most-significant halfword of data register *Db* and put the result, left-justified if *n* =1, in data register *Dc*. The operands are treated as signed values.

If n = 1, 0x8000 * 0x8000 = 0x7FFFFFF

Operation:

D[c][31:0] = (D[a][31:16] * D[b][31:16]) << n; signed; n = 0,1;



Status:

V, SV, AV, SAV

Example:

```
mul.q d0, d2, d3, 0 ; d0 = 0x2A00000
```

See Also:

MUL (pg 308), **MULS** (pg 318), **MULS.U** (pg 318), **MULM** (pg 312), **MULM.U** (pg 312), **MULR.Q** (pg 316)

MULM — Multiply with Multiword Result 32 x 32 => 64 — MULM

MULM.U — Multiply with Multiword Result Unsigned 32 x 32 => 64 — MULM.U

Syntax:

mulm	Dc, Da, Db (RR)
mulm	Dc, Da, const9 (RC)
mulm.u	Dc, Da, Db (RR)
mulm.u	Dc, Da, const9 (RC)

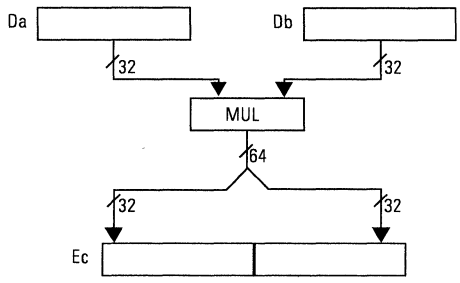
Description:

Multiply the contents of data register *Da* by the contents of data register *Db*/*const9* and put the result in data register pair *Dc*. The multiplicands are treated as signed/unsigned, 32-bit integers; the result is a signed/unsigned, 64-bit integer. The value *const9* is sign-extended/zero-extended to 32 bits before the multiplication is performed.

Operation:

```
E[c][63:0] = D[a][31:0]  * D[b][31:0]; signed
E[c][63:0] = D[a][31:0]  * sign_ext(const9); signed

E[c][63:0] = D[a][31:0] * D[b][31:0]; unsigned
E[c][63:0] = D[a][31:0] * sign_ext(const9); unsigned
```



TAM064.1

Status:

V, SV, AV, SAV

Examples:

```
mulm  d3, d1, d2
mulm  d3, d1, 126
mulm.u   d3, d1, d2
mulm.u   d3, d1, 126
```

See Also:

MUL (pg 308), **MULS** (pg 318), **MULS.U** (pg 318), **MULM.U** (pg 312), **MUL.Q** (pg 311), **MULR.Q** (pg 316)

MULR.H — Packed Multiply with Rounding 16 x 16 => 16 — MULR.H

Syntax:
mulr.h Dc, Da, Db, n (RR)

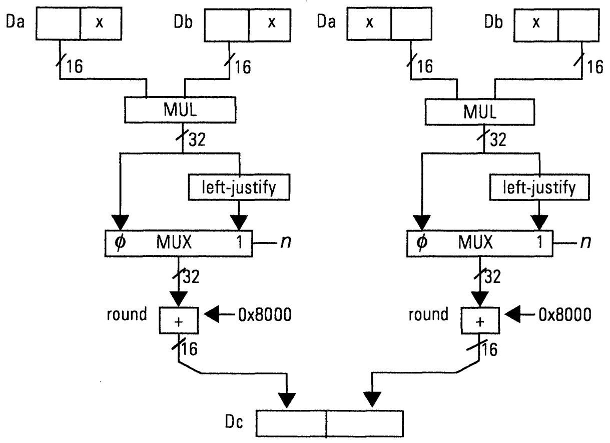
Description:

Multiply the most-significant halfword of data register *Da* by the most-significant halfword of data register *Db*. Put the rounded result, left-justified if $n = 1$, in the most-significant halfword of data register *Dc* . Multiply the least-significant halfword of data register *Da* by the least-significant halfword of data register *Db*. Put the rounded result, left-justified if $n = 1$, in the least-significant halfword of data register *Dc*. The operands are treated as signed values.

If n=1, 0x8000 x 0x8000 = 0x7FFFFFFF.

Operation:

D[c][31:16] = round16((D[a] [31:16] * D[b] [31:16]) << n)[31:0])[31:16], 16'h 0000; signed;
D[c][15:0] = round16((D[a] [15:0] * D[b] [15:0]) << n)[31:0])[31:16], 16'h 0000; signed;



TAM067.1

Status:

V, SV, AV, SAV

Example:

```
mulr.h d0, d1, d2, 1
```

See Also:

MUL.Q (pg 311),

MULR.Q Multiply in Q Format with Rounding 16 x 16 => 16 MULR.Q

Syntax:

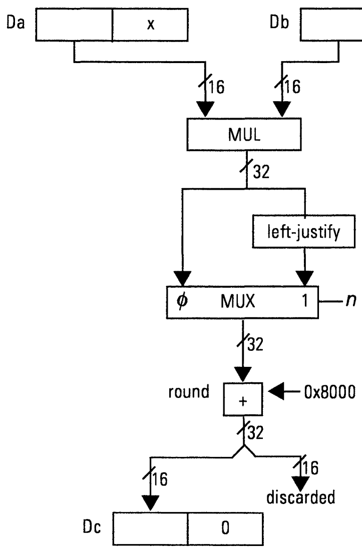
mulr.q Dc, Da, Db, n (RR)

Description:

Multiply the most-significant halfword of data register *Da* by the most-significant halfword of data register *Db*. Put the rounded result, left-justified if *n* = 1, in the most-significant halfword of data register *Dc* and set the least-significant halfword of *Dc* to 0. The operands are treated as signed values. If n = 1, 0x8000 * 0x8000 = 0x7FFFFFFF.

Operation:

D[c][31:0] = round16((D[a] [31:16] * D[b] [31:16]) << n)[31:0])[31:16], 16'h 0000; n = 0, 1; signed;



TAM067.1

Status:

V, SV, AV, SAV

Example:

```
mulr.q  d3, d1, d2, 1
```

See Also:

MUL.Q (pg 311)

MULS — Multiply with Saturation 32 x 32 => 32 — MULS

MULS.U — Multiply Unsigned with Saturation 32 x 32 => 32 — MULS.U

Syntax:

muls	Dc, Da, Db (RR)
muls	Dc, Da, const9 (RC)
muls.u	Dc, Da, Db (RR)
muls.u	Dc, Da, const9 (RC)

Description:

Multiply the contents of data register *Da* by the contents of data register *Db/const9* and put the result in data register *Dc*. The operands are treated as signed/unsigned, 32-bit integers, with saturation on signed/unsigned overflow. The value *const9* is sign-extended/zero-extended to 32 bits before the multiplication is performed.

Operation:

```
D[c][31:0] = D[a][31:0] * D[b][31:0]; signed; ssov
D[c][31:0] = D[a][31:0] * sign_ext(const9); signed; ssov

D[c][31:0] = D[a][31:0] * D[b][31:0]; unsigned; suov
D[c][31:0] = D[a][31:0] * zero_ext(const9); unsigned; suov
```

Status:

V, SV, AV, SAV

Examples:

```
muls  d3, d1, d2
muls  d3, d1, 126
muls.u d3, d1, 253
muls.u  d3, d1, d2
```

See Also:

MUL (pg 308), **MULM** (pg 312), **MULM.U** (pg 312)

NAND — Logical NAND — NAND

Syntax:

nand Dc, Da, Db (RR)
nand Dc, Da, const9 (RC)

Description:

Compute the bitwise logical NAND of the contents of data register *Da* and data register *Db/const9 and put the result* in data register *Dc*. The operands are treated as unsigned, 32-bit integers and the *const9* value is zero-extended to 32 bits.

Operation:

D[c] = !(D[a] and D[b])
D[c] = !(D[a] and zero_ext(const9)

Examples:

```
nand d3, d1, d2
nand d3, d1, 126
```

See Also:

AND (pg 145), **ANDN** (pg 152), **NOR** (pg 325), **NOT** (pg 327), **OR** (pg 328), **ORN** (pg 335), **XNOR** (pg 394), **XOR** (pg 396)

NAND.T Bit Logical NAND NAND.T

Syntax:

nand.t Dc, Da, p1, Db, p2 (BIT)

Description:

Compute the logical NAND of bit p1 of data register Da and bit p2 of data register Db. Put the result in the least-significant bit of data register *Dc* and clear the remaining bits of Dc to zero.

Refer also to Section 8.3, "Bit Operations," on page 100.

Operation:

D[c] = !(D[a][p1] and D[b][p2])

Example:

```
nand.t   d3, d1, 2, d2, 4
```

See Also:

AND.T (pg 151), **ANDN.T** (pg 153), **NOR.T** (pg 326), **OR.T** (pg 334), **ORN.T** (pg 336), **XNOR.T** (pg 395), **XOR.T** (pg 401)

NE Not Equal NE

Syntax:

ne Dc, Da, Db (RR)
ne Dc, Da, const9 (RC)

Description:

If the contents of data register *Da* are not equal to the contents of data register *Db/const9*, set the least-significant bit of *Dc* to 1 and clear the remaining bits to zero; otherwise, clear all bits in *Dc*. The *const9* value is sign-extended to 32 bits.

Refer also to Section 8.2, "Compare Instructions," on page 97.

Operation:

```
D[c] = (D[a] != D[b])
D[c] = (D[a] != sign_ext(const9))
```

Examples:

```
ne d3, d1, d2
ne d3, d1, 126
```

See Also:

EQ (pg 186), **GE** (pg 192), **GE.U** (pg 192), **LT** (pg 238), **LT.U** (pg 238)

NE.A Not Equal Address NE.A

Syntax:

ne.a Dc, Aa, Ab (RR)

Description:

If the contents of address registers *Aa* and *Ab* are not equal, set the least-significant bit of *Dc* to 1 and clear the remaining bits to zero; otherwise, clear all bits in *Dc*.

Operation:

D[c] = (A[a] != A[b])

Example:

```
ne.a  d3, a4, a2
```

See Also:

EQ.A (pg 187), **EQZ.A** (pg 190), **GE.A** (pg 193), **LT.A** (pg 240), **NEZ.A** (pg 323)

NEZ.A Not Equal Zero Address NEZ.A

Syntax:

nez.a Dc, Aa (RR)

Description:

If the contents of address register *Aa* are not equal to zero, set the least significant bit of *Dc* to 1 and clear the remaining bits to zero; otherwise, clear all bits in *Dc*.

Operation:

D[c] = (A[a] != 0)

Example:

```
nez.a d3, a4
```

See Also:

EQ.A (pg 187), **EQZ.A** (pg 190), **GE.A** (pg 193), **LT.A** (pg 240), **NE.A** (pg 322)

NOP No Operation NOP

Syntax:

nop (SYS)

nop (SR)

Description:

NOP is used to implement efficient low-power non-operational instructions.

Operation:

no operation

no operation

Example:

```
nop
```

NOR Logical NOR NOR

Syntax:

```
nor    Dc, Da, Db (RR)
nor    Dc, Da,const9 (RC)
```

Description:

Compute the bitwise logical NOR of the contents of data register *Da* and the contents of data register *Db/const9* and put the result in data register *Dc*. The operands are treated as unsigned, 32-bit integers and the *const9* value is zero-extended to 32 bits.

Operation:

```
D[c] = !(D[a] or D[b])
D[c] = !(D[a] or zero_ext(const9))
```

Examples:

```
nor d3, d1, d2
nor d3, d1, 126
```

See Also:

AND (pg 145), **ANDN** (pg 152), **NAND** (pg 319), **NOT** (pg 327), **OR** (pg 328), **ORN** (pg 335), **XNOR** (pg 394), **XOR** (pg 396)

NOR.T Bit Logical NOR NOR.T

Syntax:

nor.t Dc, Da, p1, Db, p2 (BIT)

Description:

Compute the logical NOR of bit *p1* of data register *Da* and bit p2 of data register *Db*. Put the result in the least-significant bit of data register *Dc* and clear the remaining bits of Dc to zero.

Refer also to Section 8.3, "Bit Operations," on page 100.

Operation:

D[c] = !(D[a][p1] or D[b][p2])

Example:

```
nor.t    d3, d1, 5, d2, 3
```

See Also:

AND.T (pg 151), **ANDN.T** (pg 153), **NAND.T** (pg 320), **OR.T** (pg 334), **ORN.T** (pg 336), **XNOR.T** (pg 395), **XOR.T** (pg 401)

NOT Bitwise Complement NOT

Syntax:

not Da (SR)

Description:

Compute the bitwise complement of the contents of data register *Da*.

NOTE 1: The 32-bit equivalent of the NOT instruction is a NOR with a constant of zero.

Operation:

D[a] = !D[a]

Example:

```
not   d15
```

See Also:

XNOR (pg 394)

OR — Logical OR — OR

Syntax:

or	Dc, Da, Db (RR)
or	Dc, Da, const9 (RC)
or	Da, Db (SRR)
or	D15, const8 (SC)

Description:

Compute the bitwise logical OR of the contents of data register *Da* and the contents of data register *Db/const9* and put the result in data register *Dc*. The operands are treated as unsigned, 32-bit integers and the *const9* value is zero-extended to 32 bits.

Compute the logical OR of the contents of data register *Da/D15* and the contents of data register *Db/const8* and put the result in data register *Da/D15*. The operands are treated as unsigned, 32-bit integers and the *const8* value is zero-extended to 32 bits.

Operation:

D[c] = D[a] or D[b]
D[c] = D[a] or zero_ext(const9)
D[a] = D[a] or D[b]
D[15] = D[15] or zero_ext(const8)

Examples:

```
or    d3, d1, d2
or    d3, d1, 126
or    d1, d2
or    d15, 126
```

See Also:

AND (pg 145), **ANDN** (pg 152), **NAND** (pg 319), **NOR** (pg 325), **NOT** (pg 327), **ORN** (pg 335), **XNOR** (pg 394), **XOR** (pg 396)

OR.AND.T — Accumulating Logical OR-AND — OR.AND.T
OR.ANDN.T — Accumulating Logical OR-AND-Not — OR.ANDN.T
OR.NOR.T — Accumulating Logical OR-NOR — OR.NOR.T
OR.OR.T — Accumulating Logical OR-OR — OR.OR.T

Syntax:

```
or.and.t   Dc, Da, p1, Db, p2 (BIT)
or.andn.t  Dc, Da, p1, Db, p2 (BIT)
or.nor.t   Dc, Da, p1, Db, p2 (BIT)
or.or.t    Dc, Da, p1, Db, p2 (BIT)
```

Description:

Compute the logical AND/ANDN/NOR/OR of the value of bit *p1* of data register *Da* and bit *p2* of *Db*. Then compute the logical OR of that result and bit 0 of *Dc*, and put the result back in bit 0 of *Dc*. All other bits in *Dc* are unchanged.

Refer also to Section 8.3, "Bit Operations," on page 100.

Operation:

```
or.and.t :  D[c] = {D[c][31:1], D[c][0] or (D[a][p1] and D[b][p2])}
or.andn.t : D[c] = {D[c][31:1], D[c][0] or (D[a][p1] and !D[b][p2])}
or.nor.t :  D[c] = {D[c][31:1], D[c][0] or !(D[a][p1] or D[b][p2])}
or.or.t :   D[c] = {D[c][31:1], D[c][0] or (D[a][p1] or D[b][p2])}
```

Examples:

```
or.and.t    d3, d1, 3, d2, 5
or.andn.t   d3, d1, 3, d2, 5
or.nor.t    d3, d1, 3, d2, 5
or.or.t     d3, d1, 3, d2, 5
```

See Also:

AND.AND.T (pg 146), **AND.ANDN.T** (pg 146), **AND.NOR.T** (pg 146), **AND.OR.T** (pg 146), **SH.AND.T** (pg 358), **SH.ANDN.T** (pg 358), **SH.NAND.T** (pg 358), **SH.NOR.T** (pg 358), **SH.OR.T** (pg 358), **SH.ORN.T** (pg 358), **SH.XNOR.T** (pg 358), **SH.XOR.T** (pg 358)

OR.EQ Equal Accumulating OR.EQ

Syntax:

or.eq	Dc, Da, Db (RR)
or.eq	Dc, Da,const9 (RC)

Description:

Compute the logical OR of *Dc*[0] and the Boolean result of the EQ operation on the contents of data register *Da* and data register *Db*/*const9*. Put the result in *Dc*[0]. All other bits in *Dc* are unchanged. The *const9* value is sign-extended to 32 bits.

Refer also to Section 8.2, "Compare Instructions," on page 97.

Operation:

```
D[c][0] = D[c][0] OR (D[a] == D[b])
D[c][0] = D[c][0] OR (D[a] == sign_ext(const9))
```

Examples:

```
or.eq d3, d1, d2
or.eq d3, d1, 126
```

See Also:

AND.EQ (pg 147), **XOR.EQ** (pg 397)

OR.GE — Greater Than or Equal Accumulating — OR.GE
OR.GE.U — Greater Than or Equal Accumulating Unsigned — OR.GE.U

Syntax

or.ge	Dc, Da, Db (RR)
or.ge	Dc, Da,const9 (RC)
or.ge.u	Dc, Da, Db (RR)
or.ge.u	Dc, Da,const9 (RC)

Description:

Calculate the logical OR of *Dc*[0] and the Boolean result of the GE operation on the contents of data register *Da* and data register *Db*/*const9*. Put the result in *Dc*[0]. All other bits in *Dc* are unchanged. *Da* and *Db* are treated as 32-bit signed integers. The *const9* value is sign-extended to 32 bits.

Calculate the logical OR of *Dc*[0] and the Boolean result of the GE.U operation on the contents of data register *Da* and data register *Db*/*const9*. Put the result in *Dc*[0]. All other bits in *Dc* are unchanged. *Da* and *Db* are treated as 32-bit unsigned integers. The *const9* value is zero-extended to 32 bits.

Refer also to Section 8.2, "Compare Instructions," on page 97.

Operation:

```
D[c] = D[c][0] OR (D[a] >= D[b]); signed
D[c] = D[c][0] OR (D[a] >= sign_ext(const9)); signed

D[c] = D[c][0] OR (D[a] >= D[b]); unsigned
D[c] = D[c][0] OR (D[a] >= zero_ext(const9)); unsigned
```

Examples:

```
or.ge    d3, d1, d2
or.ge    d3, d1, 126
or.ge.u  d3, d1, d2
or.ge.u  d3, d1, 126
```

See Also:

AND.GE (pg 148), **AND.GE.U** (pg 148), **XOR.GE** (pg 398), **XOR.GE.U** (pg 398)

OR.LT Less Than Accumulating OR.LT
OR.LT.U Less Than Accumulating Unsigned OR.LT.U

Syntax

or.lt	Dc, Da, Db (RR)
or.lt	Dc, Da,const9 (RC)
or.lt.u	Dc, Da, Db (RR)
or.lt.u	Dc, Da,const9 (RC)

Description:

Calculate the logical OR of *Dc*[0] and the Boolean result of the LT operation on the contents of data register *Da* and data register *Db/const9*. Put the result in *Dc*[0]. All other bits in *Dc* are unchanged. *Da* and *Db* are treated as 32-bit signed integers. The *const9* value is sign-extended to 32 bits.

Calculate the logical OR of *Dc*[0] and the Boolean result of the LT.U operation on the contents of data register *Da* and data register *Db/const9*. Put the result in *Dc*[0]. All other bits in *Dc* are unchanged. *Da* and *Db* are treated as 32-bit unsigned integers. The *const9* value is zero-extended to 32 bits.

Refer also to Section 8.2, "Compare Instructions," on page 97.

Operation:

```
D[c] = D[c][0] OR (D[a] < D[b]); signed
D[c] = D[c][0] OR (D[a] < sign_ext(const)9); signed

D[c] = D[c][0] OR (D[a] < D[b]); unsigned
D[c] = D[c][0] OR (D[a] < zero_ext(const9)); unsigned
```

Examples:

```
or.lt    d3, d1, d2
or.lt    d3, d1, 126
or.lt.u  d3, d1, d2
or.lt.u  d3, d1, 126
```

See Also:

AND.LT (pg 149), **AND.LT.U** (pg 149), **XOR.LT** (pg 399), **XOR.LT.U** (pg 399)

OR.NE Not Equal Accumulating OR.NE

Syntax:

or.ne	Dc, Da, Db (RR)
or.ne	Dc, Da, const9 (RC)

Description:

Calculate the logical OR of *Dc*[0] and the Boolean result of the NE operation on the contents of data register *Da* and data register *Db/const9*. Put the result in *Dc*[0]. All other bits in *Dc* are unchanged.

Refer also to Section 8.2, "Compare Instructions," on page 97.

Operation:

```
D[c] = D[c][0] OR (D[a] != D[b])
D[c] = D[c][0] OR (D[a] != const9)
```

Examples:

```
or.ne d3, d1, d2
or.ne d3, d1, 126
```

See Also:

AND.NE (pg 150), **XOR.NE** (pg 400)

OR.T Bit Logical OR OR.T

Syntax:

or.t Dc, Da, p1, Db, p2 (BIT)

Description:

Compute the logical OR of bit *p1* of data register *Da* and bit *p2* of data register *Db*. Put the result in the least-significant bit of data register *Dc* and clear the remaining bits of Dc to zero.

Refer also to Section 8.3, "Bit Operations," on page 100.

Operation:

D[c] = D[a][p1] or D[b][p2]

Example:

```
or.t     d3, d1, 7, d2, 9
```

See Also:

AND.T (pg 151), **ANDN.T** (pg 153), **NAND.T** (pg 320), **NOR.T** (pg 326), **ORN.T** (pg 336), **XNOR.T** (pg 395), **XOR.T** (pg 401)

ORN Logical OR-Not ORN

Syntax:

orn Dc, Da, Db (RR)
orn Dc, Da, const9 (RC)

Description:

Compute the bitwise logical OR of the contents of data register *Da* and the one's complement of the contents of data register *Db/const9* and put the result in data register *Dc*. The operands are treated as unsigned, 32-bit integers and the *const9* value is zero-extended to 32 bits.

Operation:

D[c] = D[a] or !D[b]
D[c] = D[a] or !zero_ext(const9)

Examples:

```
orn d3, d1, d2
orn d3, d1, 126
```

See Also:

AND (pg 145), **ANDN** (pg 152), **NAND** (pg 319), **NOR** (pg 325), **NOT** (pg 327), **OR** (pg 328), **XNOR** (pg 394), **XOR** (pg 396)

ORN.T Bit Logical OR-Not ORN.T

Syntax:

orn.t Dc, Da, p1, Db, p2 (BIT)

Description:

Compute the logical OR of bit *p1* of data register *Da* and the inverse of bit *p2* of data register *Db*. Put the result in the least-significant bit of data register *Dc* and clear the remaining bits of Dc to zero.

Refer also to Section 8.3, "Bit Operations," on page 100.

Operation:

D[c] = D[a][p1] or !D[b][p2]

Example:

```
orn.t    d3, d1, 2, d2, 5
```

See Also:

AND.T (pg 151), **ANDN.T** (pg 153), **NAND.T** (pg 320), **NOR.T** (pg 326), **OR.T** (pg 334), **XNOR.T** (pg 395), **XOR.T** (pg 401)

RET Return from Call RET

Syntax:

ret (SYS)

ret (SR)

Description:

Return from a function that was invoked with a CALL instruction. The return address is in register A11. The caller's upper context register values are restored as part of the return operation.

Refer also to Section 8.6.1, "Unconditional Branch," on page 103 and to Section 8.9.5, "RET and RFE," on page 111.

Operation:

Refer to Section 4.2, "Task Switching Operation," on page 48.

See Also:

CALL (pg 159), **CALLA** (pg 160), **CALLI** (pg 161), **RFE** (pg 338), **SYSCALL** (pg 391)

RFE Return From Exception RFE

Syntax:

rfe	(SYS)
rfe	(SR)

Description:

Return from an interrupt service routine or trap handler to the task whose saved upper context is specified by the contents of the Previous Context Information register (PCXI). The contents are normally the context of the task that was interrupted or that took a trap. However, in some cases, task management software may have altered the contents of the PXCI register to cause another task to be dispatched.

The return PC value is taken from register A11 in the current context. In parallel with the jump to the return PC address, the upper context registers and PSW in the saved context are restored.

Refer to Section 8.9.5, "RET and RFE," on page 111 for further details on this instruction and its use. See also Section 8.6.1, "Unconditional Branch," on page 103.

Operation:

```
PC = A11;
Restore upper context;
```

See Also:

CALL (pg 159), **CALLA** (pg 160), **CALLI** (pg 161), **RET** (pg 337), **SYSCALL** (pg 391)

RSLCX Restore Lower Context RSLCX

Syntax:

rslcx (SYS)

Description:

Load the contents of the memory block pointed to by the PCX field in PCXI into registers A2-A7, D0-D7, and A11. This operation restores the register contents of a previously saved lower context.

Refer to Section 4.1, "Upper and Lower Contexts," on page 47.

Operation:

Refer to Section 8.8.1, "Context Saving and Restoring," on page 109.

See Also:

LDLCX (pg 233), **LDUCX** (pg 235), **RSLCX** (pg 339), **STLCX** (pg 377), **STUCX** (pg 378), **SVLCX** (pg 388)

RSTV Reset Overflow Bits RSTV

Syntax:

rstv (SYS)

Description:

Reset overflow status flags in PSW. Refer to Section 8.9.3, "Access to the Core Special Function Registers," on page 111.

Operation:

PSW.{V, SV, AV, SAV} = {0, 0, 0, 0}

Example:

```
rstv
```

See Also:

BISR (pg 154), **MTCR** (pg 307), **ENABLE** (pg 185), **DISABLE** (pg 177)

RSUB Reverse-Subtract RSUB

Syntax:

rsub Dc, Da, const9 (RC)
rsub Da (SR)

Description:

Subtract the contents of data register *Da* from the value *const9* and put the result in data register *Dc*. The operands are treated as 32-bit integers. The value *const9* is sign-extended to 32 bits before the subtraction is performed.

Subtract the contents of data register *Da* from zero and put the result in data register *Da*. The operand is treated as a 32-bit integer.

Operation:

```
D[c] = sign_ext(const9) – D[a]
D[a] = 0 – D[a]
```

Status:

V, SV, AV, SAV

Examples:

```
rsub  d3, d1, 126
rsub  d1
```

See Also:

RSUBS (pg 342), **RSUBS.U** (pg 342)

RSUBS Reverse-Subtract with Saturation RSUBS
RSUBS.U Reverse-Subtract Unsigned with Saturation RSUBS.U

Syntax:

rsubs	Dc, Da, const9 (RC)
rsubs.u	Dc, Da, const9 (RC)

Description:

Subtract the contents of data register *Da* from the value *const9* and put the result in data register *Dc*. The operands are treated as signed/unsigned, 32-bit integers, with saturation on signed/unsigned overflow. The value *const9* is sign-extended/zero-extended to 32 bits before the operation is performed.

Operation:

D[c] = sign_ext(const9) – D[a]; signed; ssov
D[c] = zero_ext(const9) – D[a]; unsigned; suov

Status:

V, SV, AV, SAV

Examples:

```
rsubs d3, d1, 126
rsub  d3, d1, 253
rsubs d3, d1, 253
rsubs.u  d3, d1, 253
rsubs.u  d3, d1, 126
```

See Also:

RSUB (pg 341)

SAT.B Saturate Byte SAT.B

Syntax:

sat.b	Dc, Da (RR)
sat.b	Da (SR)

Description:

If the signed 32-bit value in *Da* is less than –128, then store the value –128 in *Dc*. If *Da* is greater than 127, then store the value 127 in *Dc*. Otherwise, copy the least-significant byte of *Da* to *Dc*.

If the signed 32-bit value in *Da* is less than –128, then store the value –128 in *Da*. If *Da* is greater than 127, then store the value 127 in *Da*. Otherwise, leave the contents of *Da* unchanged.

Operation:

```
D[c] = (D[a] < –128) ? –128 : ((D[a] > 127) ? 127 : D[a]); signed
D[a] = (D[a] < –128) ? –128 : ((D[a] > 127) ? 127 : D[a]); signed
```

Examples:

```
sat.b d3, d1
sat.b d1
```

See Also:

SAT.BU (pg 344), **SAT.H** (pg 345), **SAT.HU** (pg 346)

SAT.BU Saturate Byte Unsigned SAT.BU

Syntax:

sat.bu Dc, Da (RR)
sat.bu Da (SR)

Description:

If the unsigned 32-bit value in *Da* is greater than 255, then store the value 255 in *Dc*. Otherwise, copy the least-significant byte of *Da* to *Dc*.

If the unsigned 32-bit value in *Da* is greater than 255, then store the value 255 in *Da*. If *Da* is greater than 127, then store the value 127 in *Da*. Otherwise, leave the contents of *Da* unchanged.

Operation:

```
D[c] = (D[a] > 255) ? 255 : D[a]; unsigned
D[a] = (D[a] > 255) ? 255 : D[a]; unsigned
```

Examples:

```
sat.bu d3, d1
sat.bu d1
```

See Also:

SAT.B (pg 343), **SAT.H** (pg 345), **SAT.HU** (pg 346)

SAT.H Saturate Halfword SAT.H

Syntax:

sat.h	Dc, Da (RR)
sat.h	Da (SR)

Description:

If the signed 32-bit value in *Da* is less than –32,768, then store the value –32,768 in *Dc*. If *Da* is greater than 32,767, then store the value 32,767 in *Dc*. Otherwise, copy the least-significant halfword of *Da* to *Dc*.

If the signed 32-bit value in *Da* is less than –32,768, then store the value –32,768 in *Da*. If *Da* is greater than 32,767, then leave the contents of *Da* unchanged.

Operation:

D[a] = (D[a] < -2^{15}) ? -2^{15} : ((D[a] > $2^{15}-1$) ? $2^{15}-1$: D[a]); signed

D[a] = (D[a] < -2^{15}) ? -2^{15} : ((D[a] > $2^{15}-1$) ? $2^{15}-1$: D[a]); signed

Examples:

```
sat.h d3, d1
sat.h d1
```

See Also:

SAT.B (pg 343), **SAT.BU** (pg 344), **SAT.HU** (pg 346)

SAT.HU Saturate Halfword Unsigned SAT.HU

Syntax:

sat.hu Dc, Da (RR)
sat.hu Da (SR)

Description:

If the signed 32-bit value in *Da* is greater than 65,535, then store the value 65,535 in *Dc*; otherwise, copy the least-significant halfword of *Da* to *Dc*.

If the unsigned 32-bit value in *Da* is greater than 65,535, then store the value 65,535 in *Da*; otherwise, leave the value of *Da* unchanged.

Operation:

D[c] = (D[a] > 2^{16}–1) ? 2^{16}–1 : D[a]; unsigned
D[a] = (D[a] > 2^{16}–1) ? 2^{16}–1 : D[a]; unsigned

Examples:

```
sat.hu d3, d1
sat.hu d1
```

See Also:

SAT.BU (pg 344), **SAT.H** (pg 345)

SEL Select SEL

Syntax:

sel	Dc, Dd, Da, Db (RRR)
sel	Dc, Dd, Da, const9 (RCR)

Description:

If the contents of data register *Dd* are non-zero, copy the contents of data register *Da* to data register *Dc*; otherwise, copy the contents of *Db*/*const9* to *Dc*. The value *const9* is sign-extended to 32 bits.

Operation:

```
D[c] = ((D[d]  != 0) ? D[a] : D[b])
D[c] = ((D[d]  != 0) ? D[a] : sign_ext(const9))
```

Examples:

```
sel d3, d4, d1, d2
sel d3, d4, d1, 126
```

See Also:

CADD (pg 155), **CADDN** (pg 157), **CMOV** (pg 168), **CMOVN** (pg 169), **CSUB** (pg 170), **CSUBN** (pg 172), **SELN** (pg 349)

SEL.A Select Address SEL.A

Syntax:

sel.a	Ac, Dd, Aa, Ab (RRR)
sel.a	Ac, Dd, Aa, const9 (RCR)

Description:

If the contents of data register *Dd* are non-zero, copy the contents of address register *Aa* to address register *Ac*; otherwise, copy the contents of *Ab/const9* to *Ac*. The value *const9* is sign-extended to 32 bits.

Operation:

```
A[c] = ((D[d] != 0) ? A[a] : A[b])
A[c] = ((D[d] != 0) ? A[a] : sign_ext(const9))
```

Examples:

```
sel.a a3, d4, a4, a2
sel.a a3, d4, a4, 126
```

See Also:

CADD.A (pg 156), **CADDN.A** (pg 158), **CSUB.A** (pg 171), **CSUBN.A** (pg 173), **SELN.A** (pg 350)

SELN Select-Not SELN

Syntax:

seln	Dc, Dd, Da, Db (RRR)
seln	Dc, Dd, Da, const9 (RCR)

Description:

If the contents of data register *Dd* are zero, copy the contents of data register *Da* to data register *Dc*; otherwise, copy the contents of *Db*/*const9* to *Dc*. The value *const9* is sign-extended to 32 bits.

Operation:

D[c] = ((D[d] == 0) ? D[a] : D[b])
D[c] = ((D[d] == 0) ? D[a] : sign_ext(const9))

Examples:

```
seln  d3, d4, d1, d2
seln  d3, d4, d1, 126
```

See Also:

CADD (pg 155), **CADDN** (pg 157), **CMOV** (pg 168), **CMOVN** (pg 169), **CSUB** (pg 170), **CSUBN** (pg 172), **SEL** (pg 347)

SELN.A — Select-Not Address — SELN.A

Syntax:

```
seln.a    Ac, Dd, Aa, Ab (RRR)
seln.a    Ac, Dd, Aa, const9 (RCR)
```

Description:

If the contents of data register *Dd* are zero, copy the contents of address register *Aa* to address register *Ac*; otherwise, copy the contents of *Ab/const9* to *Ac*. The value *const9* is sign-extended to 32 bits.

Operation:

```
A[c] = ((D[d] == 0) ? A[a] : A[b])
A[c] = ((D[d] == 0) ? A[a] : sign_ext(const9))
```

Examples:

```
seln.a a3, d4, a4, a2
seln.a a3, d4, a4, 126
```

See Also:

CADD.A (pg 156), **CADDN.A** (pg 158), **CSUB.A** (pg 171), **CSUBN.A** (pg 173), **SEL.A** (pg 348)

SH Shift SH

Syntax:

sh Dc, Da, Db (RR)
sh Dc, Da, const9 (RC)
sh Da. const4 (SRC)

Description:

If the shift count specified through the contents of *Db/const9* is greater than or equal to zero, then left-shift the value in *Da* by the amount specified by shift count. Otherwise, right-shift the value in *Da* by the absolute value of the shift count. Put the result in *Dc*. In both cases, the vacated bits are filled with zeroes and bits shifted out are discarded. The shift count is a 6-bit signed number, derived from Db[5:0] or const9[5:0]. The range for the shift count therefore is –32 to +31, allowing to shift left up to 31 bit positions and to shift right up to 32 bit positions (a shift right by 32 bits leaves 0s in the result).

If the shift count specified through the value *const4* is greater than or equal to zero, then left-shift the value in *Da* by the amount specified by the shift count. Otherwise, right-shift the value in *Da* by the absolute value of the shift count. Put the result in *Da*. In both cases, the vacated bits are filled with zeroes and bits shifted out are discarded. The shift count is a 6-bit signed number, derived from the sign-extension of const4[3:0]. The resulting range for the shift count therefore is –8 to +7, allowing to shift left up to 7 bit positions and to shift right up to 8 bit positions.

Operation:

```
if (shift_count >= 0) then D[c] = D[a] << shift_count; zero-fill
else D[c] = D[a] >> (–shift_count); zero-fill
shift_count = Db[5:0] or const9[5:0]
```

```
if (shift_count >= 0) then D[a] = D[a] << shift_count; zero-fill
else D[a] = D[a] >> (–shift_count); zero-fill
shift_count = sign_ext(const4[3:0])
```

Examples:

```
sh    d3, d1, d2
sh    d3, d1, 26
sh    d1, 6
```

See Also:

SH.B (pg 353), **SH.H** (pg 353), **SHA** (pg 360), **SHA.B** (pg 362), **SHA.H** (pg 362), **SHAS** (pg 364)

SH.B Shift Packed Bytes SH.B
SH.H Shift Packed Halfwords SH.H

Syntax:

sh.b	Dc, Da, Db (RR)
sh.b	Dc, Da. const9 (RC)
sh.h	Dc, Da, Db (RR)
sh.h	Dc, Da. const9 (RC)

Description:

If the shift count specified through the contents of *Db/const9* is greater than or equal to zero, then left-shift each byte/halfword in *Da* by the amount specified by shift count. Otherwise, right-shift each byte/halfword in *Da* by the absolute value of the shift count. Put the result in *Dc*. In both cases, the vacated bits are filled with zeroes and bits shifted out are discarded. Note that for these shifts, each byte/halfword is treated individually, and bits shifted out of a byte/halfword are not shifted in to the next byte/halfword.

The shift count is a signed number, derived from the sign-extension of Db[3:0] or const9[3:0] for sh.b and from the sign-extension of Db[4:0] or const9[4:0] for sh.h. The range for the shift count therefore is –8 to +7 for sh.b and –16 to +15 for sh.h. The result for a shift count of –8 for bytes and –16 for halfwords is zero.

Refer also to Section 8.1.3, "Packed Arithmetic," on page 95.

Operation:

```
sh.b : if (shift_count >= 0) then D[c] = D[a][(n+7):n] << shift_count; zero-fill
       else D[c] = D[a][(n+7):n] >> (–shift_count); zero-fill
       shift_count = sign_ext(Db[3:0]) or sign_ext(const9[3:0]); n = 0, 8, 16, 24

sh.h : if (shift_count >= 0) then D[c] = D[a][(n+15):n] << shift_count; zero-fill
       else D[c] = D[a][(n+15):n] >> (–shift_count); zero-fill
       shift_count = sign_ext(Db[4:0]) or sign_ext(const9[4:0]); n = 0, 16
```

Examples:

```
sh.b    d3, d1, d2
sh.b    d3, d1, 5
sh.h    d3, d1, d2
sh.h    d3, d1, 12
```

See Also:

SH (pg 351), **SHA** (pg 360), **SHAS** (pg 364), **SHA.B** (pg 362), **SHA.H** (pg 362)

SH.EQ Shift Equal SH.EQ

Syntax:

```
sh.eq    Dc, Da, Db (RR)
sh.eq    Dc, Da,const9 (RC)
```

Description:

If the contents of data register *Da* are equal to the contents of data register *Db/const9*, set the least-significant bit of *Dc* to 1; otherwise, set the least-significant bit of *Dc* to 0. The remaining bits in *Dc* are shifted left by 1. The value *const9* is sign-extended to 32 bits.

Refer also to Section 8.2, "Compare Instructions," on page 97.

Operation:

```
D[c] = {D[c][30:0], (D[a] == D[b])}
D[c] = {D[c][30:0], (D[a] == sign_ext(const9))}
```

Examples:

```
sh.eq d3, d1, d2
sh.eq d3, d1, 126
```

See Also:

SH.GE (pg 355), **SH.GE.U** (pg 355), **SH.LT** (pg 356), **SH.LT.U** (pg 356), **SH.NE** (pg 357)

SH.GE Shift Greater Than or Equal SH.GE
SH.GE.U Shift Greater Than or Equal SH.GE.U

Syntax:

sh.ge	Dc, Da, Db (RR)
sh.ge	Dc, Da,const9 (RC)
sh.ge.u	Dc, Da, Db (RR)
sh.ge.u	Dc, Da,const9 (RC)

Description:

If the contents of data register *Da* are greater than or equal to the contents of data register *Db/const9*, set the least-significant bit of *Dc* to 1; otherwise, set the least-significant bit of *Dc* to 0. The remaining bits in *Dc* are shifted left by 1. *Da* and *Db* are treated as signed integers. The value *const9* is sign-extended to 32 bits.

If the contents of data register *Da* are greater than or equal to the contents of data register *Db/const9*, set the least-significant bit of *Dc* to 1; otherwise, set the least-significant bit of *Dc* to 0. The remaining bits in *Dc* are shifted left by 1. *Da* and *Db* are treated as unsigned integers. The value *const9* is zero-extended to 32 bits.

Refer also to Section 8.2, "Compare Instructions," on page 97.

Operation:

```
D[c] = {D[c][30:0], (D[a] >= D[b])}; signed
D[c] = {D[c][30:0], (D[a] >= sign_ext(const9))}; signed

D[c] = (D[c][30:0], (D[a] >= D[b])}; unsigned
D[c] = (D[c][30:0], (D[a] >= zero_ext(const9))}; unsigned
```

Examples:

```
sh.ge     d3, d1, d2
sh.ge     d3, d1, 126
sh.ge     d3, d1, 253
sh.ge.u   d3, d1, d2
sh.ge.u   d3, d1, 126
sh.ge.u   d3, d1, 253
```

See Also:

SH.EQ (pg 354), **SH.LT** (pg 356), **SH.LT.U** (pg 356), **SH.NE** (pg 357)

SH.LT — Shift Less Than — SH.LT
SH.LT.U — Shift Less Than Unsigned — SH.LT.U

Syntax:

sh.lt	Dc, Da, Db (RR)
sh.lt	Dc, Da,const9 (RC)
sh.lt.u	Dc, Da, Db (RR)
sh.lt.u	Dc, Da,const9 (RC)

Description:

If the contents of data register *Da* are less the contents of data register *Db/const9*, set the least-significant bit of *Dc* to 1; otherwise, set the least-significant bit of *Dc* to 0. The remaining bits in *Dc* are shifted left by 1. *Da* and *Db/const9* are treated as signed integers. The value *const9* is sign-extended to 32 bits.

If the contents of data register *Da* are less the contents of data register *Db/const9*, set the least-significant bit of *Dc* to 1; otherwise, set the least-significant bit of *Dc* to 0. The remaining bits in *Dc* are shifted left by 1. *Da* and *Db/const9* are treated as unsigned integers. The value *const9* is zero-extended to 32 bits.

Refer also to Section 8.2, "Compare Instructions," on page 97.

Operation:

```
D[c] = {D[c][30:0], (D[a] < D[b])}; signed
D[c] = {D[c][30:0], (D[a] < sign_ext(const9))}; signed

D[c] = {D[c][30:0], (D[a] < D[b])}; unsigned
D[c] = {(D[c][30:0], (D[a] < zero_ext(const9))}; unsigned
```

Examples:

```
sh.lt    d3, d1, d2
sh.lt    d3, d1, 126
sh.lt.u  d3, d1, d2
sh.lt.u  d3, d1, 126
```

See Also:

SH.EQ (pg 354), **SH.GE** (pg 355), **SH.GE.U** (pg 355), **SH.NE** (pg 357)

SH.NE Shift Not Equal SH.NE

Syntax:

```
sh.ne    Dc, Da, Db (RR)
sh.ne    Dc, Da,const9 (RC)
```

Description:

If the contents of data register *Da* are not equal to the contents of data register *Db*/*const9*, set the least-significant bit of *Dc* to 1; otherwise, set the least-significant bit of *Dc* to 0. The remaining bits in *Dc* are shifted left by 1. The value *const9* is sign-extended to 32 bits.

Refer also to Section 8.2, "Compare Instructions," on page 97.

Operation:

```
D[c] = {D[c][30:0], (D[a] != D[b])}
D[c] = {D[c][30:0], (D[a] != sign_ext(const9))}
```

Examples:

```
sh.ne d3, d1, d2
sh.ne d3, d1, 126
sh.ne d3, d1, 253
```

See Also:

SH.EQ (pg 354), **SH.GE** (pg 355), **SH.GE.U** (pg 355), **SH.LT** (pg 356), **SH.LT.U** (pg 356)

SH.AND.T	**Accumulating Shift-AND**	**SH.AND.T**
SH.ANDN.T	**Accumulating Shift-AND-Not**	**SH.ANDN.T**
SH.NAND.T	**Accumulating Shift-NAND**	**SH.NAND.T**
SH.NOR.T	**Accumulating Shift-NOR**	**SH.NOR.T**
SH.OR.T	**Accumulating Shift-OR**	**SH.OR.T**
SH.ORN.T	**Accumulating Shift-OR-Not**	**SH.ORN.T**
SH.XNOR.T	**Accumulating Shift-XNOR**	**SH.XNOR.T**
SH.XOR.T	**Accumulating Shift-XOR**	**SH.XOR.T**

Syntax:

```
sh.and.t   Dc, Da, p1, Db, p2 (BIT)
sh.andn.t  Dc, Da, p1, Db, p2 (BIT)
sh.nand.t  Dc, Da, p1, Db, p2 (BIT)
sh.nor.t   Dc, Da, p1, Db, p2 (BIT)
sh.or.t    Dc, Da, p1, Db, p2 (BIT)
sh.orn.t   Dc, Da, p1, Db, p2 (BIT)
sh.xnor.t  Dc, Da, p1, Db, p2 (BIT)
sh.xor.t   Dc, Da, p1, Db, p2 (BIT)
```

Description:

Left shift Dc by 1. The bit shifted out is discarded. Compute the logical AND/ANDN/NAND/NOR/OR/ORN/XNOR/XOR of the value of bit *p1* of data register *Da* and bit *p2* of *Db*. Put the result in Dc[0].

Refer also to Section 8.3, "Bit Operations," on page 100.

Operation:

```
sh.and.t :  D[c] = {D[c][30:0], (D[a][p1] and D[b][p2])}
sh.andn.t : D[c] = {D[c][30:0], (D[a][p1] and !(D[b][p2]))}
sh.nand.t : D[c] = {D[c][30:0], !(D[a][p1] and D[b][p2])}
sh.nor.t :  D[c] = {D[c][30:0], !(D[a][p1] or D[b][p2])}
sh.or.t :   D[c] = {D[c][30:0], (D[a][p1] or D[b][p2])}
sh.orn.t :  D[c] = {D[c][30:0], (D[a][p1] or !(D[b][p2]))}
sh.xnor.t : D[c] = {D[c][30:0], !(D[a][p1] xor D[b][p2])}
sh.xor.t :  D[c] = {D[c][30:0], (D[a][p1] xor D[b][p2])}
```

Examples:

```
sh.and.t   d3, d1, 4, d2, 7
sh.andn.t  d3, d1, 4, d2, 7
sh.nand.t  d3, d1, 4, d2, 7
sh.nor.t   d3, d1, 4, d2, 7
sh.or.t    d3, d1, 4, d2, 7
sh.orn.t   d3, d1, 4, d2, 7
sh.xnor.t  d3, d1, 4, d2, 7
sh.xor.t   d3, d1, 4, d2, 7
```

See Also:

AND.AND.T (pg 146), **AND.ANDN.T** (pg 146), **AND.NOR.T** (pg 146), **AND.OR.T** (pg 146), **OR.AND.T** (pg 329), **OR.ANDN.T** (pg 329), **OR.NOR.T** (pg 329), **OR.OR.T** (pg 329)

SHA Arithmetic Shift SHA

Syntax:

```
sha    Dc, Da, Db (RR)
sha    Dc, Da, const9 (RC)
sha    Da, const4 (SRC)
```

Description:

If the shift count specified through the contents of *Db/const9* is greater than or equal to zero, then left-shift the value in *Da* by the amount specified by shift count. The vacated bits are filled with zeroes and bits shifted out are discarded. If the shift count is less than zero, right-shift the value in *Da* by the absolute value of the shift count. The vacated bits are filled with the sign-bit (MSB) and bits shifted out are discarded. Put the result in *Dc*. On all 1-bit or greater shifts (left or right), PSW.C is set to the bitwise logical-OR of the shifted out bits and zero. On zero-bit shifts, C is cleared.

The shift count is a 6-bit signed number, derived from Db[5:0] or const9[5:0]. The range for the shift count therefore is –32 to +31, allowing to shift left up to 31 bit positions and to shift right up to 32 bit positions (a shift right by 32 bits leaves all 0s or all 1s in the result, depending on the sign-bit).

If the shift count specified through the value *const4* is greater than or equal to zero, then left-shift the value in *Da* by the amount specified by the shift count. The vacated bits are filled with zeroes and bits shifted out are discarded. If the shift count is less than zero, right-shift the value in *Da* by the absolute value of the shift count. The vacated bits are filled with the sign-bit (MSB) and bits shifted out are discarded. Put the result in *Da*.

The shift count is a 6-bit signed number, derived from the sign-extension of const4[3:0]. The resulting range for the shift count therefore is –8 to +7, allowing to shift left up to 7 bit positions and to shift right up to 8 bit positions.

On all 1-bit or greater shifts (left or right), PSW.C is set to the bitwise logical-OR of the shifted out bits and zero. On zero-bit shifts, C is cleared.

Operation:

```
if (shift_count >= 0) then D[c] = D[a] << shift_count; zero-fill
else D[c] = D[a] >> (–shift_count); sign-fill
shift_count = Db[5:0] or const9[5:0]
```

```
if (shift_count >= 0) then D[a] = D[a] << shift_count; zero-fill
else D[a] = D[a] >> (–shift_count); sign-fill
shift_count = sign_ext(const4[3:0])
```

Status:

V, SV, AV, SAV, C

Examples:

```
sha   d3, d1, d2
sha   d3, d1, 26
sha   d1, 6
```

See Also:

SH (pg 351), **SH.B** (pg 353), **SH.H** (pg 353), **SHA.B** (pg 362), **SHA.H** (pg 362), **SHAS** (pg 364)

SHA.B Arithmetic Shift Packed Bytes SHA.B
SHA.H Arithmetic Shift Packed Halfwords SHA.H

Syntax:

sha.b	Dc, Da, Db (RR)
sha.b	Dc, Da, const9 (RC)
sha.h	Dc, Da, Db (RR)
sha.h	Dc, Da, const9 (RC)

Description:

If the shift count specified through the contents of *Db/const9* is greater than or equal to zero, then left-shift each byte/halfword in *Da* by the amount specified by shift count. The vacated bits are filled with zeros and bits shifted out are discarded. If the shift count is less than zero, right-shift each byte/halfword in *Da* by the absolute value of the shift count. The vacated bits are filled with the sign-bit (MSB) of the respective byte/halfword, and bits shifted out are discarded. Put the result in *Dc*. Note that for these shifts, each byte/halfword is treated individually, and bits shifted out of a byte/halfword are not shifted in to the next byte/halfword.

The shift count is a signed number, derived from the sign-extension of Db[3:0] or const9[3:0] for sha.b and from the sign-extension of Db[4:0] or const9[4:0] for sha.h. The range for the shift count therefore is –8 to +7 for sha.b and –16 to +15 for sha.h. The result for each byte/halfword for a shift count of –8 for sha.b and –16 for sha.h is either all zeros or all ones, depending on the sign-bit of the respective byte/halfword.

Refer also to Section 8.1.3, "Packed Arithmetic," on page 95.

Operation:

```
sha.b :  if (shift_count >= 0) then D[c] = D[a][(n+7):n] << shift_count; zero-fill
         else D[c] = D[a][(n+7):n] >> (–shift_count); sign-fill
         shift_count = sign_ext(Db[3:0]) or sign_ext(const9[3:0]); n = 0, 8, 16, 24

sha.h :  if (shift_count >= 0) then D[c] = D[a][(n+15):n] << shift_count; zero-fill
         else D[c] = D[a][(n+15):n] >> (–shift_count); sign-fill
         shift_count = sign_ext(Db[4:0]) or sign_ext(const9[4:0]); n = 0, 16
```

Status:

V, SV, AV, SAV

Examples:

```
sha.b    d3, d1, d2
sha.b    d3, d1, 6
sha.h    d3, d1, d2
sha.h    d3, d1, 12
```

See Also:

SH (pg 351), **SHA** (pg 360), **SHAS** (pg 364), **SH.B** (pg 353), **SH.H** (pg 353)

SHAS **Arithmetic Shift with Saturation** SHAS

Syntax:

shas	Dc, Da, Db (RR)
shas	Dc, Da, const9 (RC)

Description:

If the shift count specified through the contents of *Db/const9* is greater than or equal to zero, then left-shift the value in *Da* by the amount specified by shift count. The vacated bits are filled with zeroes and the result is saturated if its sign bit differs from the sign bits that are shifted out. If the shift count is less than zero, right-shift the value in *Da* by the absolute value of the shift count. The vacated bits are filled with the sign-bit (MSB) and bits shifted out are discarded. Put the result in *Dc*.

The shift count is a 6-bit signed number, derived from Db[5:0] or const9[5:0]. The range for the shift count therefore is –32 to +31, allowing to shift left up to 31 bit positions and to shift right up to 32 bit positions (a shift right by 32 bits leaves all 0s or all 1s in the result, depending on the sign-bit).

Operation:

```
if (shift_count >= 0) then D[c] = D[a] << shift_count; zero-fill; ssov
else D[c] = D[a] >> (–shift_count); sign-fill
shift_count = D[b][5:0] or const9[5:0]
```

Status:

V, SV, AV, SAV

Examples:

```
shas     d3, d1, d2
shas     d3, d1, 26
```

See Also:

SH (pg 351), **SH.B** (pg 353), **SH.H** (pg 353), **SHA** (pg 360), **SHA.B** (pg 362), **SHA.H** (pg 362)

ST.A Store Word From Address Register ST.A

Syntax:

```
st.a        <mode>, Aa
```

Description:

Store the value in address register *Aa* to the memory location specified by the addressing mode.

Operation:

M(EA, word) = A[a]

<mode>	Syntax	Effective Address	Instruction Format
Absolute	constant	Refer to Section 2.4.1.1, "Absolute Addressing," on page 20	ABS
Base + Short Offset	[An]offset	A[b]+sign_ext(offset10)	BO
Pre-increment	[+An]offset	A[b]+sign_ext(offset10)	BO
Post-increment	[An+]offset	A[b]	BO
Circular	[An+c]offset	Refer to Section 2.4.1.5, "Circular Addressing," on page 21	BO
Bit-reverse	[An+r]	Refer to Section 2.4.1.6, "Bit-Reverse Addressing," on page 22	BO

See Also:

ST.B (pg 367), **ST.D** (pg 369), **ST.DA** (pg 370), **ST.H** (pg 371), **ST.Q** (pg 373), **ST.W** (pg 376)

ST.A Store Word From Address Register (16-bit) ST.A

Syntax:

```
st.a        <mode>, Aa
```

Description:

Store the value in address register *Aa* to the memory location specified by the addressing mode.

Operation:

```
M(EA, word) = A[a]
```

<mode>	Syntax	Effective Address	Instruction Format
Register indirect	[An]	A[b]	SSR
(Implicit) Base + Offset	[A15]offset	A[15]+zero_ext(offset4)	SSRO
Implicit destination register	[An]offset4	A[b]+zero_ext(offset4), byte)	SRO
Post-increment	[An+]offset	A[b], byte; A[b] = A[b] + 4	SSR

See Also:

ST.B (pg 367), **ST.H** (pg 371), **ST.W** (pg 376)

ST.B Store Byte ST.B

Syntax:

```
st.b        <mode>, Da
```

Description:

Store the byte value in the 8 least-significant bits of data register *Da* to the byte memory location specified by the addressing mode.

Operation:

M(EA, byte) = D[a][7:0]

<mode>	Syntax	Effective Address	Instruction Format
Absolute	constant	Refer to Section 2.4.1.1, "Absolute Addressing," on page 20	ABS
Base + Short Offset	[An]offset	A[b]+sign_ext(offset10)	BO
Pre-increment	[+An]offset	A[b]+sign_ext(offset10)	BO
Post-increment	[An+]offset	A[b]	BO
Circular	[An+c]offset	Refer to Section 2.4.1.5, "Circular Addressing," on page 21	BO
Bit-reverse	[An+r]	Refer to Section 2.4.1.6, "Bit-Reverse Addressing," on page 22	BO

See Also:

ST.A (pg 365), **ST.D** (pg 369), **ST.DA** (pg 370), **ST.H** (pg 371), **ST.Q** (pg 373), **ST.W** (pg 376)

ST.B Store Byte (16-bit) ST.B

Syntax:

```
st.b        <mode>, Da
```

Description:

Store the byte value in the 8 least-significant bits of data register *Da* to the byte memory location specified by the addressing mode.

Operation:

```
M(EA, byte) = D[a][7:0]
```

<mode>	Syntax	Effective Address	Instruction Format
Register indirect	[An]	A[b]	SSR
(Implicit) Base + Offset	[A15]offset	A[15]+zero_ext(offset4)	SSRO
Implicit destination register	[An]offset4	A[b]+zero_ext(offset4), byte)	SRO
Post-increment	[An+]offset	A[b], byte; A[b] = A[b] + 1	SSR

See Also:

ST.A (pg 365), **ST.H** (pg 371), **ST.W** (pg 376)

ST.D Store Doubleword From Data Registers ST.D

Syntax:

st.d <mode>, Ea

Description:

Store the value in the extended data register pair *Ea* to the memory location specified by the addressing mode. The value in the even register (D*n*) is stored in the least-significant memory word, and the value in the odd register (D*n*+1) is stored in the most-significant memory word. This instruction must be halfword-aligned.

Operation:

M(EA, doubleword) = D[a](pair)

<mode>	Syntax	Effective Address	Instruction Format
Absolute	constant	Refer to Section 2.4.1.1, "Absolute Addressing," on page 20	ABS
Base + Short Offset	[An]offset	A[b]+sign_ext(offset10)	BO
Pre-increment	[+An]offset	A[b]+sign_ext(offset10)	BO
Post-increment	[An+]offset	A[b]	BO
Circular	[An+c]offset	Refer to Section 2.4.1.5, "Circular Addressing," on page 21	BO
Bit-reverse	[An+r]	Refer to Section 2.4.1.6, "Bit-Reverse Addressing," on page 22	BO

See Also:

ST.A (pg 365), **ST.B** (pg 367), **ST.DA** (pg 370), **ST.H** (pg 371), **ST.Q** (pg 373), **ST.W** (pg 376)

ST.DA Store Doubleword From Address Registers ST.DA

Syntax:

st.da <mode>, Aa

Description:

Store the value in the address register pair *Aa* to the memory location specified by the addressing mode. The value in the even register (A*n*) is stored in the least-significant memory word, and the value in the odd register (A*n*+1) is stored in the most-significant memory word.

Operation:

M(EA, doubleword) = A[a](pair)

<mode>	Syntax	Effective Address	Instruction Format
Absolute	constant	Refer to Section 2.4.1.1, "Absolute Addressing," on page 20	ABS
Base + Short Offset	[An]offset	A[b]+sign_ext(offset10)	BO
Pre-increment	[+An]offset	A[b]+sign_ext(offset10)	BO
Post-increment	[An+]offset	A[b]	BO
Circular	[An+c]offset	Refer to Section 2.4.1.5, "Circular Addressing," on page 21	BO
Bit-reverse	[An+r]	Refer to Section 2.4.1.6, "Bit-Reverse Addressing," on page 22	BO

See Also:

ST.A (pg 365), **ST.B** (pg 367), **ST.H** (pg 371), **ST.Q** (pg 373), **ST.W** (pg 376)

ST.H Store Halfword ST.H

Syntax:

st.h <mode>, Da

Description:

Store the halfword value in the 16 least-significant bits of data register *Da* to the halfword memory location specified by the addressing mode.

Operation:

M(EA, halfword) = D[a][15:0]

<mode>	Syntax	Effective Address	Instruction Format
Absolute	constant	Refer to Section 2.4.1.1, "Absolute Addressing," on page 20	ABS
Base + Short Offset	[An]offset	A[b]+sign_ext(offset10)	BO
Pre-increment	[+An]offset	A[b]+sign_ext(offset10)	BO
Post-increment	[An+]offset	A[b]	BO
Circular	[An+c]offset	Refer to Section 2.4.1.5, "Circular Addressing," on page 21	BO
Bit-reverse	[An+r]	Refer to Section 2.4.1.6, "Bit-Reverse Addressing," on page 22	BO

See Also:

ST.A (pg 365), **ST.B** (pg 367), **ST.Q** (pg 373), **ST.W** (pg 376)

ST.H Store Halfword (16-bit) ST.H

Syntax:

```
st.h        <mode>, Aa
```

Description:

Store the halfword value in the 16 least-significant bits of data register *Da* to the halfword memory location specified by the addressing mode.

Operation:

```
M(EA, halfword) = D[a][15:0]
```

<mode>	Syntax	Effective Address	Instruction Format
Register indirect	[An]	A[b]	SSR
(Implicit) Base + Offset	[A15]offset	A[15]+zero_ext(offset4)	SSRO
Implicit destination register	[An]offset4	A[b]+zero_ext(offset4), byte)	SRO
Post-increment	[An+]offset	A[b], byte; A[b] = A[b] + 1	SSR

See Also:

ST.A (pg 365), **ST.B** (pg 367), **ST.Q** (pg 373), **ST.W** (pg 376)

ST.Q Store Halfword Signed Fraction ST.Q

Syntax:

st.q <mode>, Da

Description:

Store the value in the most-significant halfword of data register *Da* to the memory location specified by the addressing mode.

Operation:

M(EA, halfword) = D[a][31:16]

<mode>	Syntax	Effective Address	Instruction Format
Absolute	constant	Refer to Section 2.4.1.1, "Absolute Addressing," on page 20	ABS
Base + Short Offset	[An]offset	A[b]+sign_ext(offset10)	BO
Pre-increment	[+An]offset	A[b]+sign_ext(offset10)	BO
Post-increment	[An+]offset	A[b]	BO
Circular	[An+c]offset	Refer to Section 2.4.1.5, "Circular Addressing," on page 21	BO
Bit-reverse	[An+r]	Refer to Section 2.4.1.6, "Bit-Reverse Addressing," on page 22	BO

See Also:

ST.A (pg 365), **ST.B** (pg 367), **ST.W** (pg 376)

ST.T Store Bit ST.T

Syntax:

```
st.t        offset18, bpos3, b
```

Description:

Store the bit value *b* to the byte at the memory address specified by *offset18* in the bit position specified by *bpos3*. The other bits of the byte are unchanged.

Refer also to Section 8.7.3, "Store Bit and Bit Field," on page 108.

Operation:

M(EA, byte)[bpos3] = b

<mode>	Syntax	Effective Address	Instruction Format
Absolute	constant	Refer to Section 2.4.1.1, "Absolute Addressing," on page 20	ABSB

See Also:

LDMDST (pg 234), **IMASK** (pg 194)

ST.W Store Word ST.W

Syntax:

st.w <mode>, Da

Description:

Store the word value in data register *Da* to the memory location specified by the addressing mode.

Operation:

M(EA, word) = D[a]

<mode>	Syntax	Effective Address	Instruction Format
Absolute	constant	Refer to Section 2.4.1.1, "Absolute Addressing," on page 20	ABS
Base + Short Offset	[An]offset	A[b]+sign_ext(offset10)	BO
Base + Long Offset	[An]offset	A[b]+sign_ext(offset16)	BOL
Pre-increment	[+An]offset	A[b]+sign_ext(offset10)	BO
Post-increment	[An+]offset	A[b]	BO
Circular	[An+c]offset	Refer to Section 2.4.1.5, "Circular Addressing," on page 21	BO
Bit-reverse	[An+r]	Refer to Section 2.4.1.6, "Bit-Reverse Addressing," on page 22	BO

See Also:

ST.B (pg 367), **ST.D** (pg 369), **ST.DA** (pg 370), **ST.Q** (pg 373)

ST.W Store Word (16-bit) ST.W

Syntax:

```
st.w        <mode> Da
```

Description:

Store the word value in data register *Da* to the memory location specified by the addressing mode.

Operation:

```
M(EA, word) = D[a]
```

<mode>	Syntax	Effective Address	Instruction Format
Register indirect	[An]	A[b]	SSR
(Implicit) Base + Offset	[A15]offset	A[15]+zero_ext(offset4)	SSRO
Implicit destination register	[An]offset4	A[b]+zero_ext(offset4), byte)	SRO
Post-increment	[An+]offset	A[b], byte; A[b] = A[b] + 4	SSR

See Also:

ST.A (pg 365), **ST.B** (pg 367), **ST.H** (pg 371)

STLCX Store Lower Context STLCX

Syntax:

stlcx <mode>

Description:

Store the contents of registers A2 – A7, D0 – D7, and A11 to the memory block specified by the addressing mode. Note that the effective address specified by the addressing mode must resolve to an on-chip memory location aligned on a 16-word boundary. For this instruction, the addressing mode is limited to absolute (ABS) or base plus short offset (BO).

Operation:

Refer to Section 8.8.2, "Context Loading and Storing," on page 109.

<mode>	Syntax	Effective Address	Instruction Format
Absolute	constant	Refer to Section 2.4.1.1, "Absolute Addressing," on page 20	ABS
Base + Short Offset	[An]offset	A[a]+sign_ext(offset10)	BO

See Also:

LDLCX (pg 233), **LDUCX** (pg 235), **RSLCX** (pg 339), **STUCX** (pg 378), **SVLCX** (pg 388)

STUCX Store Upper Context STUCX

Syntax:

stucx <mode>

Description:

Store the contents of registers A10 – A15, D8 – D15, and the current PSW (the registers which comprise a task's upper context) to the memory block specified by the addressing mode. Note that the effective address specified by the addressing mode must resolve to an on-chip memory location aligned on a 16-word boundary. For this instruction, the addressing mode is limited to absolute (ABS) or base plus short offset (BO).

Operation:

Refer to Section 8.8.2, "Context Loading and Storing," on page 109.

<mode>	Syntax	Effective Address	Instruction Format
Absolute	constant	Refer to Section 2.4.1.1, "Absolute Addressing," on page 20	ABS
Base + Short Offset	[An]offset	A[a]+sign_ext(offset10)	BO

See Also:

LDLCX (pg 233), **LDUCX** (pg 235), **RSLCX** (pg 339), **STUCX** (pg 378), **SVLCX** (pg 388)

SUB Subtract SUB

Syntax:

sub	Dc, Da, Db (RR)
sub	Da, Db (SRR)
sub	D15, Da, Db (SRR)

Description:

Subtract the contents of data register *Db* from the contents of data register *Da* and put the result in data register *Dc*.

Subtract the contents of data register *Db* from the contents of data register *Da* and put the result in data register *Da*/D15.

Operation:

D[c] = D[a] – D[b]
D[a] = D[a] – D[b]
D[15] = D[a] – D[b]

Status:

V, SV, AV, SAV

Examples:

```
sub   d3, d1, d2
sub   d1, d2
sub   d15, d1, d2
```

See Also:

SUBS (pg 383), **SUBS.U** (pg 383), **SUBX** (pg 387), **SUBC** (pg 382)

SUB.A Subtract Address SUB.A

Syntax:

sub.a Ac, Aa, Ab (RR)
sub.a SP, const8 (SC)

Description:

Subtract the contents of address register *Ab* from the contents of address register *Aa* and put the result in address register *Ac*. The operands are treated as unsigned, 32-bit integers.

Decrement the Stack Pointer (A10) by the zero-extended value of *const8* (a range of 0 through 255). The operands are treated as unsigned, 32-bit integers.

Operation:

A[c] = A[a] – A[b]
A[10] = A[10] – zero_ext(const8)

Examples:

```
sub.a a3, a4, a2
sub.a sp, 126
```

See Also:

ADD.A (pg 133), **ADDIH.A** (pg 138), **ADDSC.A** (pg 143), **ADDSC.AT** (pg 143), **DIFSC.A** (pg 176), **SUBSC.A** (pg 386)

SUB.B Subtract Packed Byte SUB.B
SUB.H Subtract Packed Halfword SUB.H

Syntax:

```
sub.b    Dc, Da, Db (RR)
sub.h    Dc, Da, Db (RR)
```

Description:

Subtract the contents of each byte/halfword of data register *Db* from the contents of data register *Da* and put the result in each corresponding byte/halfword of data register *Dc*.

Refer also to Section 8.1.3, "Packed Arithmetic."

Operation:

```
D[c][(n+7):n] = D[a][(n+7):n] + D[b][(n+7):n], n = 0, 8, 16, 24;
D[c][(n+15):n] = D[a][(n+15):n] + D[b][(n+15):n]; n = 0, 16
```

Status:

V, SV, AV, SAV

Examples:

```
sub.b  d3, d1, d2
sub.h  d3, d1, d2
```

See Also:

SUBS.B (pg 384), **SUBS.BU** (pg 384), **SUBS.H** (pg 385), **SUBS.HU** (pg 385)

SUBC Subtract with Carry SUBC

Syntax:

subc Dc, Da, Db (RR)

Description:

Subtract the contents of data register `Db` plus the carry bit minus 1 from the contents of data register `Da` and put the result in data register `Dc`. The operands are treated as unsigned, 32-bit integers. The PSW carry bit is updated by the ALU carry out.

Operation:

D[c] = D[a] – D[b] + psw.C–1; psw.C = carry_out

Status:

C, V, SV, AV, SAV

Example:

```
subc d3, d1, d2
```

See Also:

SUB (pg 379), **SUBS** (pg 383), **SUBS.U** (pg 383), **SUBX** (pg 387)

SUBS — Subtract Signed with Saturation — SUBS
SUBS.U — Subtract Unsigned with Saturation — SUBS.U

Syntax:

subs	Dc, Da, Db (RR)
subs.u	Dc, Da, Db (RR)
subs	Da, Db (SRR)

Description:

Subtract the contents of data register *Db* from the contents of data register *Da* and put the result in data register *Dc*. The operands are treated as signed/unsigned 32-bit integers, with saturation on signed/unsigned overflow.

Subtract the contents of data register *Db* from the contents of data register *Da* and put the result in data register *Da*. The operands are treated as signed 32-bit integers, with saturation on signed overflow.

Operation:

```
D[c] = D[a] – D[b]; signed; ssov
D[c] = D[a] – D[b]; unsigned; suov
D[a] = D[a] – D[b]; signed; ssov
```

Status:

V, SV, AV, SAV

Examples:

```
subs   d3, d1, d2
subs.u d3, d1, d2
subs   d3, d1
```

See Also:

SUB (pg 379), **SUBX** (pg 387), **SUBC** (pg 382)

SUBS.B — Subtract Packed Byte with Saturation — SUBS.B
SUBS.BU — Subtract Packed Byte Unsigned with Saturation — SUBS.BU

Syntax:

```
subs.b    Dc, Da, Db (RR)
subs.bu   Dc, Da, Db (RR)
```

Description:

Subtract the contents of each byte of data register *Db* from the contents of data register *Da* and put the result in each corresponding byte of data register *Dc*, with saturation on signed/unsigned overflow.

Refer also to Section 8.1.3, "Packed Arithmetic," on page 95.

Operation:

D[c][(n+7):n] = D[a][(n+7):n] – D[b][(n+7):n]; n = 0, 8, 16, 24; signed; ssov
D[c][(n+7):n] = D[a][(n+7):n] – D[b][(n+7):n]; n = 0, 8, 16, 24; unsigned; suov

Status:

V, SV, AV, SAV

Examples:

```
subs.b   d3, d1, d2
subs.bu  d3, d1, d2
```

See Also:

SUB.B (pg 381), **SUBS.H** (pg 385), **SUBS.HU** (pg 385)

SUBS.H Subtract Packed Halfword with Saturation SUBS.H
SUBS.HU Subtract Packed Halfword Unsigned with Saturation SUBS.HU

Syntax:

```
subs.b   Dc, Da, Db (RR)
subs.bu  Dc, Da, Db (RR)
```

Description:

Subtract the contents of each halfword of data register *Db* from the contents of data register *Da* and put the result in each corresponding halfword of data register *Dc*, with saturation on signed/unsigned overflow.

Refer also to Section 8.1.3, "Packed Arithmetic," on page 95.

Operation:

```
D[c][(n+15):n] = D[a][(n+15):n] – D[b][(n+15):n]; n = 0, 16; signed; ssov
D[c][(n+15):n] = D[a][(n+15):n] – D[b][(n+15):n]; n = 0, 16; unsigned; suov
```

Status:

V, SV, AV, SAV

Examples:

```
subs.h   d3, d1, d2
subs.hu  d3, d1, d2
```

See Also:

SUB.B (pg 381), **SUBS.B** (pg 384), **SUBS.BU** (pg 384)

SUBSC.A Subtract Scaled Address SUBSC.A

Syntax:

subsc.a Ac, Aa, Db, n (RR)

Description:

Left-shift the contents of data register *Db* by the amount specified by *n*, where *n* can be 0, 1, 2, or 3. Subtract that value from address register *Aa* and put the result in address register *Ac*.

Operation:

A[c] = A[a] – (D[b] << n), n = 0, 1, 2, or 3

Example:

```
subsc.a a3, a4, d2, 1
```

See Also:

ADD.A (pg 133), **ADDIH.A** (pg 138), **ADDSC.A** (pg 143), **ADDSC.AT** (pg 143), **DIFSC.A** (pg 176), **SUB.A** (pg 380)

SUBX Subtract Extended SUBX

Syntax:

subx Dc, Da, Db (RR)

Description:

Subtract the contents of data register `Db` from the contents of data register `Da` and put the result in data register `Dc`. The operands are treated as unsigned, 32-bit integers. The PSW carry bit is set to the value of the ALU carry out.

Operation:

D[c] = D[a] – D[b]; psw.C = carry_out

Status:

C, V, SV, AV, SAV

Example:

```
subx  d3, d1, d2
```

See Also:

SUB (pg 379), **SUBC** (pg 382), **SUBS** (pg 383), **SUBS.U** (pg 383)

SVLCX Save Lower Context SVLCX

Syntax:

svlcx (SYS)

Description:

Store the contents of registers A2 – A7, D0 – D7, and the current return address (A11) to the memory location pointed to by the FCX register. This operation saves the lower context of the currently executing task.

Refer to Section 4.1, "Upper and Lower Contexts," on page 47.

Operation:

Refer to Section 8.8.1, "Context Saving and Restoring," on page 109.

Example:

```
svlcx
```

See Also:

LDLCX (pg 233), **LDUCX** (pg 235), **RSLCX** (pg 339), **STLCX** (pg 377), **STUCX** (pg 378)

SWAP.A Swap with Address Register SWAP.A

Syntax:

swap.a Aa, <mode>

Description:

Swap the contents of address register *Aa* and the memory word specified by the addressing mode.

Operation:

```
tmp = M(EA, word);
M(EA, word) = A[a];
A[a] = tmp
```

<mode>	Syntax	Effective Address	Instruction Format
Absolute	constant	Refer to Section 2.4.1.1, "Absolute Addressing," on page 20	ABS
Base + Short Offset	[An]offset	A[b]+sign_ext(offset10)	BO
Pre-increment	[+An]offset	A[b]+sign_ext(offset10)	BO
Post-increment	[An+]offset	A[b]	BO
Circular	[An+c]offset	Refer to Section 2.4.1.5, "Circular Addressing," on page 21	BO
Bit-reverse	[An+r]	Refer to Section 2.4.1.6, "Bit-Reverse Addressing," on page 22	BO

See Also:

SWAP.W (pg 390)

SWAP.W Swap with Data Register SWAP.W

Syntax:

swap.w Da, <mode>

Description:

Swap the contents of data register *Db* and the memory word specified by the addressing mode.

Operation:

```
tmp = M(EA, word);
M(EA, word) = D[a];
D[a] = tmp
```

<mode>	Syntax	Effective Address	Instruction Format
Absolute	constant	Refer to Section 2.4.1.1, "Absolute Addressing," on page 20	ABS
Base + Short Offset	[An]offset	A[b]+sign_ext(offset10)	BO
Pre-increment	[+An]offset	A[b]+sign_ext(offset10)	BO
Post-increment	[An+]offset	A[b]	BO
Circular	[An+c]offset	Refer to Section 2.4.1.5, "Circular Addressing," on page 21	BO
Bit-reverse	[An+r]	Refer to Section 2.4.1.6, "Bit-Reverse Addressing," on page 22	BO

See Also:

SWAP.A (pg 389)

SYSCALL System Call SYSCALL

Syntax:

syscall const9 (RC)

Description:

Cause a system call trap, using the Trap Identification Number (TIN) specified by *const9*. Note that the trap return PC will be that of the instruction following the SYSCALL instruction.

Refer to Section 6.1.4, "Software Traps," on page 72.

Operation:

Refer to Section 6.2, "Trap Handling," on page 72.

Example:

```
syscall  4
```

See Also:

RET (pg 337), **RFE** (pg 338), **TRAPV** (pg 392), **TRAPSV** (pg 393)

TRAPV Trap on Overflow TRAPV

Syntax:

trapv (SYS)

Description:

If the PSW's overflow status flag (PSW.V) is set, generate a trap to the vector entry for the overflow trap handler (OVF trap).

Refer to Section 6.1.4, "Software Traps," on page 72.

Operation:

```
if PSW.V then trap (OVF)
```

Example:

```
trapv
```

See Also:

RSTV (pg 340), **SYSCALL** (pg 391), **TRAPSV** (pg 393)

TRAPSV Trap on Sticky Overflow TRAPSV

Syntax:

trapsv (SYS)

Description:

If the PSWs sticky overflow status flag (PSW.SV) is set, generate a trap to the vector entry for the sticky overflow trap handler (SOVF trap).

Refer to Section 6.1.4, "Software Traps," on page 72.

Operation:

if PSW.SV then trap (SOVF)

Example:

```
trapsv
```

See Also:

RSTV (pg 340), **SYSCALL** (pg 391), **TRAPV** (pg 392)

XNOR Logical Exclusive NOR XNOR

Syntax

xnor	Dc, Da, Db (RR)
xnor	Dc, Da, const9 (RC)

Description:

Compute the bitwise logical exclusive NOR of the contents of data register *Da* and the contents of data register *Db/const9* and put the result in data register *Dc*. The operands are treated as unsigned, 32-bit integers. The value *const9* is zero-extended to 32 bits.

Operation:

```
D[c] = !(D[a] xor D[b])
D[c] = !(D[a] xor zero_ext(const9))
```

Examples:

```
xnor  d3, d1, d2
xnor  d3, d1, 126
```

See Also:

AND (pg 145), **ANDN** (pg 152), **NAND** (pg 319), **NOR** (pg 325), **NOT** (pg 327), **OR** (pg 328), **ORN** (pg 335), **XOR** (pg 396)

XNOR.T Bit Logical XNOR XNOR.T

Syntax:

xnor.t Dc, Da, p1, Db, p2 (BIT)

Description:

Compute the logical exclusive NOR of bit *p1* of data register *Da* and bit *p2* of data register *Db*. Put the result in the least-significant bit of data register *Dc* and clear the remaining bits of Dc to zero.

Refer also to Section 8.3, "Bit Operations," on page 100.

Operation:

```
D[c] = !(D[a][p1] xor D[b][p2])
```

Example:

```
xnor.t   d3, d1, 3, d2, 5
```

See Also:

AND.T (pg 151), **ANDN.T** (pg 153), **NAND.T** (pg 320), **NOR.T** (pg 326), **OR.T** (pg 334), **ORN.T** (pg 336), **XOR.T** (pg 401)

XOR Logical Exclusive OR XOR

Syntax:

xor	Dc, Da, Db (RR)
xor	Dc, Da, const9 (RC)

Description:

Compute the bitwise logical exclusive OR of the contents of data register *Da* and the contents of data register *Db/const9* and put the result in data register *Dc*. The operands are treated as unsigned, 32-bit integers. The value *const9* is zero-extended to 32 bits.

Operation:

D[c] = D[a] xor D[b]
D[c] = D[a] xor zero_ext(const9)

Examples:

```
xor   d3, d1, d2
xor   d3, d1, 126
```

See Also:

AND (pg 145), **ANDN** (pg 152), **NAND** (pg 319), **NOR** (pg 325), **NOT** (pg 327), **OR** (pg 328), **ORN** (pg 335), **XNOR** (pg 394)

XOR.EQ Equal Accumulating XOR.EQ

Syntax:

```
xor.eq    Dc, Da, Db (RR)
xor.eq    Dc, Da,const9 (RC)
```

Description:

Compute the logical XOR of *Dc*[0] and the Boolean result of the EQ operation on the contents of data register *Da* and data register *Db*/*const9*. Put the result in *Dc*[0]. All other bits in *Dc* are unchanged. The value *const9* is sign-extended to 32 bits.

Refer also to Section 8.2, "Compare Instructions," on page 97.

Operation:

```
D[c][0] = D[c][0] XOR (D[a] == D[b])
D[c][0] = D[c][0] XOR (D[a] == sign_ext(const9))
```

Examples:

```
xor.eq    d3, d1, d2
xor.eq    d3, d1, 126
```

See Also:

AND.EQ (pg 147), **OR.EQ** (pg 330)

XOR.GE — Greater Than or Equal Accumulating — XOR.GE
XOR.GE.U — Greater Than or Equal Accumulating Unsigned — XOR.GE.U

Syntax:

xor.ge	Dc, Da, Db (RR)
xor.ge	Dc, Da,const9 (RC)
xor.ge.u	Dc, Da, Db (RR)
xor.ge.u	Dc, Da,const9 (RC)

Description:

Calculate the logical XOR of *Dc*[0] and the Boolean result of the GE operation on the contents of data register *Da* and data register *Db/const9*. Put the result in *Dc*[0]. All other bits in *Dc* are unchanged. *Da* and *Db* are treated as 32-bit signed integers. The value *const9* is sign-extended to 32 bits.

Calculate the logical XOR of *Dc*[0] and the Boolean result of the GE.U operation on the contents of data register *Da* and data register *Db/const9*. Put the result in *Dc*[0]. All other bits in *Dc* are unchanged. *Da* and *Db* are treated as 32-bit unsigned integers. The value *const9* is zero-extended to 32 bits.

Refer also to Section 8.2, "Compare Instructions," on page 97.

Operation:

```
D[c] = D[c][0] XOR (D[a] >= D[b]); signed
D[c] = D[c][0] XOR (D[a] >= sign_ext(const9)); signed

D[c] = D[c][0] XOR (D[a] >= D[b]); unsigned
D[c] = D[c][0] XOR (D[a] >= zero_ext(const9)); unsigned
```

Examples:

```
xor.ge   d3, d1, d2
xor.ge   d3, d1, 126
xor.ge.u d3, d1, d2
xor.ge.u d3, d1, 126
```

See Also:

AND.GE (pg 148), **AND.GE.U** (pg 148), **OR.GE** (pg 331), **OR.GE.U** (pg 331)

XOR.LT — Less Than Accumulating — XOR.LT
XOR.LT.U — Less Than Accumulating Unsigned — XOR.LT.U

Syntax

xor.lt	Dc, Da, Db (RR)
xor.lt	Dc, Da,const9 (RC)
xor.lt.u	Dc, Da, Db (RR)
xor.lt.u	Dc, Da,const9 (RC)

Description:

Calculate the logical XOR of *Dc*[0] and the Boolean result of the LT operation on the contents of data register *Da* and data register *Db/const9.* Put the result in *Dc*[0]. All other bits in *Dc* are unchanged. *Da* and *Db* are treated as 32-bit signed integers. The value *const9* is sign-extended to 32 bits.

Calculate the logical XOR of *Dc*[0] and the Boolean result of the LT.U operation on the contents of data register *Da* and data register *Db/const9.* Put the result in *Dc*[0]. All other bits in *Dc* are unchanged. *Da* and *Db* are treated as 32-bit unsigned integers. The value *const9* is zero-extended to 32 bits.

Refer also to Section 8.2, "Compare Instructions," on page 97.

Operation:

```
D[c] = D[c][0] XOR (D[a] < D[b]); signed
D[c] = D[c][0] XOR (D[a] < sign_ext(const9)); signed

D[c] = D[c][0] XOR (D[a] < D[b]); unsigned
D[c] = D[c][0] XOR (D[a] < zero_ext(const9)); unsigned
```

Examples:

```
xor.lt   d3, d1, d2
xor.lt   d3, d1, 126
xor.lt.u d3, d1, d2
xor.lt.u d3, d1, 126
```

See Also:

AND.LT (pg 149), **AND.LT.U** (pg 149), **OR.LT** (pg 332), **OR.LT.U** (pg 332)

XOR.NE Not Equal Accumulating XOR.NE

Syntax:

xor.ne Dc, Da, Db (RR)
xor.ne Dc, Da,const9 (RC)

Description:

Calculate the logical XOR of *Dc*[0] and the Boolean result of the NE operation on the contents of data register *Da* and data register *Db*/*const9*. Put the result in *Dc*[0]. All other bits in *Dc* are unchanged. The value *const9* is sign-extended to 32 bits.

Refer also to Section 8.2, "Compare Instructions," on page 97.

Operation:

D[c] = D[c][0] XOR (D[a] != D[b])
D[c] = D[c][0] XOR (D[a] != sign_ext(const9))

Examples:

```
xor.ne d3, d1, d2
xor.ne d3, d1, 126
```

See Also:

AND.NE (pg 150), **OR.NE** (pg 333)

XOR.T Bit Logical XOR XOR.T

Syntax:

xor.t Dc, Da, p1, Db, p2 (BIT)

Description:

Compute the logical XOR of bit *p1* of data register *Da* and bit *p2* of data register *Db*. Put the result in the least-significant bit of data register *Dc* and clear the remaining bits of Dc to zero.

Refer also to Section 8.3, "Bit Operations," on page 100.

Operation:

D[c] = D[a][p1] xor D[b][p2]

Example:

```
xor.t    d3, d1, 3, d2, 7
```

See Also:

AND.T (pg 151), **ANDN.T** (pg 153), **NAND.T** (pg 320), **NOR.T** (pg 326), **OR.T** (pg 334), **ORN.T** (pg 336), **XNOR.T** (pg 395)

Index

Global PartnerChip for Systems on Silicon

(A)

Siemens AG Österreich
Erdberger Lände 26
1030 Wien
☎ (++43)-1-1707-35611
Fax (++43)-1-1707-55973

(AUS)

Siemens Ltd., Head Office
544 Church Street
Richmond (Melbourne), Vic. 3121
☎ (03) 4207111
Tx 30425
Fax (03) 4207275

(B)

Siemens Electronic Components Benelux
Charleroisesteenweg 116/
Chaussée de Charleroi 116
B-1060 Brussel/Bruxelles
☎ (+32) 2-5362348
Fax (+32) 2-5362857

(BR)

ICOTRON S.A.
Indústria de Componentes
Eletrônicos
Avenida Mutinga, 3650-6o andar
05150 S_o Paulo-SP
☎ (011) 833-2211
Tx 11-81001
Fax (011) 831-4006

(CDN)

Siemens Electric Ltd.
Electronic Components Division
1180 Courtney Park Drive
Mississauga, Ontario L5T 1P2
☎ (416) 905-819-8000
Fax (416) 905-819-5744

(CH)

Siemens Schweiz AG
Bauelemente
Freilagerstraße 28
8047 Zürich
☎ (01) 495-3111
Fax (01) 495-5065

(D)

Siemens AG
Salzufer 6—8
10587 Berlin
☎ (030) 3863-2626
Fax (030) 3863-2490

Siemens AG
Lahnweg 10
40219 Düsseldorf
☎ (0211) 399-2930
Fax (0211) 399-1481

Siemens AG
Lindenplatz 2
20099 Hamburg
☎ (040) 2889-3819
Fax (040) 2889-3092

Siemens AG
Werner-von-Siemens-Platz 1
30880 Laatzen (Hannover)
☎ (0511) 877-2222
Fax (0511) 877-2078

Siemens AG
Halbleiter Distribution
Richard-Strauss-Straße 76
81679 München
☎ (089) 9221-3133
Fax (089) 9221-2071

Siemens AG
Von-der-Tann-Straße 30
90439 Nürnberg
☎ (0911) 654-7602
Fax (0911) 654-7624

Siemens AG
Weissacher Straße 11
70499 Stuttgart
☎ (0711) 1372864
Fax (0711) 1372448

(DK)

Siemens A/S
Borupvang 3
2750 Ballerup
☎ 44774477
Tx 1258222
Fax 44774017

(E)

Siemens S.A.
Dpto. Componentes
Ronda de Europa, 3
28760 Tres Cantos-Madrid
☎ (01) 8030085
Fax (01) 8033926

(F)

Siemens S.A.
39/47, Bd. Ornano
93527 Saint-Denis CEDEX 2
☎ (1) 49223100
Tx 234077
Fax (1) 49223970

(GB)

Siemens plc
Siemens House
Oldbury
Bracknell
Berkshire RG12 8FZ
☎ (0344) 396000
Fax (0344) 396632

(GR)

Siemens AE
Paradissou & Artemidos
P.O.B. 61011
15110 Amaroussio/Athen
☎ (01) 6864111
Tx 216292
Fax (01) 6864299

(HK)

Siemens Components Ltd
23/F., Tai Yau Building
181 Johnston Road, Wanchai
Hong Kong
☎ (852) 28320500
Fax (852) 28278421

(I)

Siemens S.p.A.
Semiconductor Sales
Via dei Valtorta, 48
20127 Milano
☎ (02) 6676-1
Fax (02) 6676-4395

IND

Siemens Ltd.
Head Office
134-A, Dr. Annie Besant Road,
Worli
P.O.B. 6597
Bombay 400018
☎ (022) 4938786
Tx 1175142
Fax (022) 4940240

IRL

Siemens Ltd.
Electronic Components Division
8 Raglan Road
Dublin 4
☎ (01) 6684727
Tx 93744
Fax (01) 684633

J

Siemens Components K.K.
Shinjuku Koyama Bldg. 2F
30-3, 4-Chome
Yoyogi, Shibuya-ku
Tokyo 151
☎ (81) 3-53888525
Fax (81) 3-33769792

N

Siemens A/S
_stre Aker vei 90
Postboks 10, Veitvet
0518 Oslo 5
☎ (02) 633000
Tx 78477
Fax (02) 633805

NL

Siemens Electronic Components Benelux
Postbus 16068
NL-2500 BB Den Haag
☎ (+31) 70-3332429
Fax (+31) 70-3332815

P

Siemens S.A.
Estrada Nacional 117, Km 2,6
Alfragide
2700 Amadora
☎ (01) 4170011
Tx 62955
Fax (01) 4172870

PL

Siemens Sp. z.o.o.
ul. Stawki 2
POB 276
00-950 Warszawa
☎ 6351619
Tx 825554
Fax 6355238

RC

Tai Engineering Co., Ltd.
6th Fl., Central Building
108, Chung Shan North Road, Sec. 2
P.O. Box 68-1882
Taipei 10449
☎ (02) 5234700
Tx 27860 taiengco
Fax (02) 5367070

ROC

Siemens Ltd.
Asia Tower Bldg, 10th floor
726 Yeoksam-dong, Kangnam-ku
CPO Box 3001, Seoul 135-080
Korea
☎ (822) 5277740
Fax (822) 5277779

Siemens AG
1. Donskoj pr., 2
Moskva 117419
☎ (095) 237-6476, -6911
Tx 414385
Fax (095) 237-6614

S

Siemens Components
Österögatan 1
Box 46
S-164 93 Kista
☎ (08) 7033500
Tx 11672
Fax (08) 7033501

FIN

Siemens Oy
P.O.B. 60
02601 Espoo
☎ (0) 51051, y 124465
Fax (0) 51052398

SGP

Siemens Components Pte. Ltd.
166 Kallang Way
Singapore 1334
☎ (65) 8400600
Fax (65) 7421080

TR

SIMKO Ticaret ve Sanayi A.S.
Meclisi Mebusan Cad. No. 125
P.K. 1001, 80007 Karaköy
80040 Findikli
☎ (01) 2510900
Tx 24233 sies tr
Fax (01) 2524134

USA

Siemens Microelectronics, Inc.
Integrated Circuit Division
10950 North Tantau Avenue
Cupertino, CA 95014
☎ (408) 777-4500
Fax (408) 777-4977

ZA

Siemens Ltd.
Siemens House,
P.O.B. 4583
Johannesburg 2000
☎ (011) 3151950
Tx 450091
Fax (011) 3151968

http://www.siemens.de/Semiconductor/index.htm
USA: http://www.tri-core.com

SIEMENS

Total Quality Management

Quality takes on an all-encompassing significance at the Siemens Semiconductor Group. For us it means living up to each and every one of your demands in the best possible way. So we are not only concerned with product quality. We direct our efforts equally at quality of supply and logistics, service and support, as well as all the other ways in which we advise and attend to you.

Part of Siemens' quality is the very special attitude of our staff. Total Quality in thought and deed, towards co-workers, suppliers and you, our customer. Our guideline is "do everything with zero defects," in an open manner that is demonstrated beyond your immediate workplace, and to constantly improve. Throughout the corporation, we also think in terms of Time Optimized Processes (TOP), greater speed on our part to give you that decisive competitive edge.

Give us the chance to prove the best of performance through the best of quality—you will be convinced.

Global PartnerChip
for Systems on Silicon

Published by Semiconductor Group

Ordering No: M32T011
Marcom No: 97/10-064D-500
Euro No: B158-H7128-X-X-7600

Siemens Microelectronics, Inc.

Printed in U.S.A.

Order Number: M32T011

Issued by:
Siemens Microelectronics, Inc.
10950 N. Tantau Avenue, Cupertino, CA 95014-0716
Tel. 1-800-596-6792 • Fax. 408-895-5020 • Web: www.tri-core.com

Marcom Number: 97/10-064D-500
Euro Number: B158-H7128-X-X-7600
Data Printed Subject to Change
Printed in the USA