



# 32-Bit Single-Chip Microcontroller



Never stop thinking.

**Edition 2000-01**

**Published by Infineon Technologies AG,  
St.-Martin-Strasse 53,  
D-81541 München, Germany**

**© Infineon Technologies AG 2001-04-30 @ 15:16.  
All Rights Reserved.**

**Attention please!**

The information herein is given to describe certain components and shall not be considered as warranted characteristics.

Terms of delivery and rights to technical change reserved.

We hereby disclaim any and all warranties, including but not limited to warranties of non-infringement, regarding circuits, descriptions and charts stated herein.

Infineon Technologies is an approved CECC manufacturer.

**Information**

For further information on technology, delivery terms and conditions and prices please contact your nearest Infineon Technologies Office in Germany or our Infineon Technologies Representatives worldwide (see address list).

**Warnings**

Due to technical requirements components may contain dangerous substances. For information on the types in question please contact your nearest Infineon Technologies Office.

Infineon Technologies Components may only be used in life-support devices or systems with the express written approval of Infineon Technologies, if a failure of such components can reasonably be expected to cause the failure of that life-support device or system, or to affect the safety or effectiveness of that device or system. Life support devices or systems are intended to be implanted in the human body, or to support and/or maintain and sustain and/or protect human life. If they fail, it is reasonable to assume that the health of the user or other persons may be endangered.

# 32-Bit Single-Chip Microcontroller

Microcontrollers


N e v e r   s t o p   t h i n k i n g .

|                   |  |
|-------------------|--|
| <Device>          |  |
| <hr/>             |  |
| Revision History: | 2000-01 V 1.3.1                              |
| <hr/>             |  |
| Previous Version: | -  |
| <hr/>             |  |
| Page              | Subjects (major changes since last revision) |
| <hr/>             | <hr/>  |
| <hr/>             | <hr/>  |
| <hr/>             | <hr/>  |
| <hr/>             | <hr/>  |
| <hr/>             | <hr/>  |
| <hr/>             |  |

Controller Area Network (CAN): License of Robert Bosch GmbH

**We Listen to Your Comments**

Any information within this document that you feel is wrong, unclear or missing at all?  
Your feedback will help us to continuously improve the quality of this document.  
Please send your proposal (including a reference to this document) to:  
**[mcdocu.comments@infineon.com](mailto:mcdocu.comments@infineon.com)**



---

# TriCore Architecture v1.3 Manual

---

## About this document

This document was created with Adobe® FrameMaker® 5.5.6 at Infineon Technologies, 1730 North First Street, San Jose, California 95112, USA. This document is not controlled. No distribution list is maintained and the reader is responsible for ensuring that he/she is not using an obsolete version.

Your comments are invited via e-mail to: [editor@infineon.com](mailto:editor@infineon.com)

## Revision History

| Release Version | Release Date | Notes  |
|-----------------|--------------|--|
| 1.3.0           | Jan 13, 2000 | First release, v1.3 architecture   |
| 1.3.1           | Jun 7, 2000  | Section numbering corrected, significant content changes to PSW and Instruction Set chapter, reformatted document        |
| 1.3.2           | Aug 14, 2000 | Significant improvements to Chapter 6, 7, 9 and some minor changes in LCXO description and some instruction descriptions |

2001-04-30 @ 15:16

For questions on technology, delivery and prices please contact the Infineon Technologies Offices in Germany or the Infineon Technologies Companies and Representatives worldwide:  
see our webpage at <http://www.infineon.com>

**Attention please!**

The information herein is given to describe certain components and shall not be considered as warranted characteristics.

Terms of delivery and rights to technical change reserved.

We hereby disclaim any and all warranties, including but not limited to warranties of non-infringement, regarding circuits, descriptions and charts stated herein.

Infineon Technologies is an approved CECC manufacturer.

**Information**

For further information on technology, delivery terms and conditions and prices please contact your nearest Infineon Technologies Office in Germany or our Infineon Technologies Representatives worldwide (see address list).

**Warnings**

Due to technical requirements components may contain dangerous substances. For information on the types in question please contact your nearest Infineon Technologies Office.

Infineon Technologies Components may only be used in life-support devices or systems with the express written approval of Infineon Technologies, if a failure of such components can reasonably be expected to cause the failure of that life-support device or system, or to affect the safety or effectiveness of that device or system. Life support devices or systems are intended to be implanted in the human body, or to support and/or maintain and sustain and/or protect human life. If they fail, it is reasonable to assume that the health of the user or other persons may be endangered.

2001-04-30 @ 15:16



### Table of Contents

Page

## TriCore Architecture v1.3 Manual i

### Architecture Overview 1

|          |                              |          |
|----------|------------------------------|----------|
| <b>1</b> | <b>Architecture Overview</b> | <b>2</b> |
| 1.1      | Feature Overview             | 3        |
| 1.2      | Program State Registers      | 3        |
| 1.3      | Data Types                   | 4        |
| 1.4      | Addressing Modes             | 5        |
| 1.5      | Instruction Formats          | 5        |
| 1.6      | Tasks and Contexts           | 5        |
| 1.7      | Interrupt System             | 7        |
| 1.8      | Trap System                  | 7        |
| 1.9      | Protection System            | 7        |
| 1.10     | Memory Management System     | 8        |
| 1.11     | Reset System                 | 9        |
| 1.12     | Debug System                 | 9        |

### Programming Model 11

|          |                          |           |
|----------|--------------------------|-----------|
| <b>2</b> | <b>Programming Model</b> | <b>12</b> |
| 2.1      | Data Types               | 12        |
| 2.2      | Data Formats             | 13        |
| 2.3      | Memory Model             | 15        |
| 2.4      | Addressing Model         | 16        |

### Core Registers 23

|          |                              |           |
|----------|------------------------------|-----------|
| <b>3</b> | <b>Core Registers</b>        | <b>24</b> |
| 3.1      | Access to the Core Registers | 25        |
| 3.2      | General-Purpose Registers    | 25        |
| 3.3      | Program State Information    | 27        |
| 3.4      | Context Management Registers | 30        |
| 3.5      | Stack Management             | 33        |
| 3.6      | Interrupt and Trap Control   | 34        |
| 3.7      | System Control Registers     | 37        |
| 3.8      | MMU Registers                | 37        |
| 3.9      | Memory Protection Registers  | 38        |
| 3.10     | Debug Registers              | 43        |

### Managing Tasks & Functions 45

|          |                                       |           |
|----------|---------------------------------------|-----------|
| <b>4</b> | <b>Managing Tasks &amp; Functions</b> | <b>46</b> |
| 4.1      | Upper and Lower Contexts              | 46        |
| 4.2      | Task Switching Operation              | 47        |
| 4.3      | CSAs and Context Lists                | 48        |
| 4.4      | Context Switching with Interrupts     | 50        |
| 4.5      | Context Switching with Function Calls | 51        |
| 4.6      | Context Save/Restore Examples         | 51        |

| Table of Contents            |  | Page |
|------------------------------|--|------|
| Interrupt System 57          |  |      |
| 5                            | Interrupt System .....                       | 58   |
| 5.1                          | Service Request Node .....                   | 59   |
| 5.2                          | Interrupt Control Unit .....                 | 62   |
| 5.3                          | Entering an Interrupt Service Routine .....  | 65   |
| 5.4                          | The Interrupt Vector Table .....             | 66   |
| 5.5                          | Usage of the TriCore Interrupt System .....  | 67   |
| 5.6                          | CPU Service Request Nodes .....              | 71   |
| Traps 73                     |  |      |
| 6                            | Traps .....                                  | 74   |
| 6.1                          | Trap Types .....                             | 74   |
| 6.2                          | Trap Handling .....                          | 77   |
| 6.3                          | Trap Descriptions .....                      | 78   |
| Memory Management 85         |  |      |
| 7                            | Memory Management .....                      | 86   |
| 7.1                          | Address Spaces .....                         | 87   |
| 7.2                          | Address translation .....                    | 88   |
| 7.3                          | Translation Lookaside Buffers .....          | 89   |
| 7.4                          | Cacheability .....                           | 90   |
| 7.5                          | Protection .....                             | 91   |
| 7.6                          | Multiple Address Spaces .....                | 92   |
| 7.7                          | MMU traps .....                              | 92   |
| 7.8                          | MMU instructions .....                       | 94   |
| 7.9                          | MMU Special Function Registers .....         | 96   |
| Protection System 103        |  |      |
| 8                            | Protection System .....                      | 104  |
| 8.1                          | Direct and Page Table Entry Addressing ..... | 104  |
| 8.2                          | Protection System Registers .....            | 104  |
| 8.3                          | Sample Protection Register Set .....         | 111  |
| 8.4                          | Memory Access Checking .....                 | 112  |
| Instruction Set Overview 115 |  |      |
| 9                            | Instruction Set Overview .....               | 116  |
| 9.1                          | Arithmetic Instructions .....                | 116  |
| 9.2                          | Compare Instructions .....                   | 126  |
| 9.3                          | Bit Operations .....                         | 130  |
| 9.4                          | Address Arithmetic .....                     | 132  |
| 9.5                          | Address Comparison .....                     | 132  |
| 9.6                          | Branch Instructions .....                    | 133  |
| 9.7                          | Load and Store Instructions .....            | 136  |
| 9.8                          | Context Related Instructions .....           | 138  |
| 9.9                          | System Instructions .....                    | 139  |

|      |                           |     |
|------|---------------------------|-----|
| 9.10 | 16-bit Instructions ..... | 141 |
|------|---------------------------|-----|

## TriCore Instruction Set 143

|           |                                      |            |
|-----------|--------------------------------------|------------|
| <b>10</b> | <b>TriCore Instruction Set .....</b> | <b>144</b> |
| 10.1      | Instruction Syntax .....             | 144        |
| 10.2      | Instruction Operation .....          | 147        |
| 10.3      | Status .....                         | 148        |
| 10.4      | Instruction Descriptions .....       | 149        |

## Global PartnerChip for Systems on Silicon 301

## Total Quality Management 305

## Index 307



---

# Architecture Overview

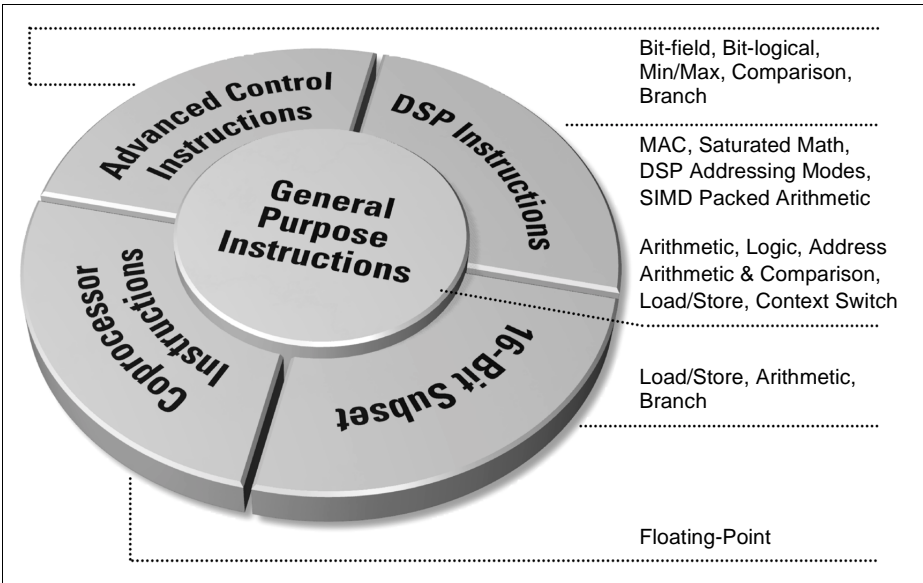
---

2001-04-30 @ 15:16

## 1 Architecture Overview

TriCore is the first single-core 32-bit microcontroller-DSP architecture optimized for real-time embedded systems. TriCore unifies the best of 3 worlds - real-time capabilities of microcontrollers, computational prowess of DSPs, and highest performance/price implementations of RISC load-store architectures.

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



**Figure 1-1**  
**TriCore: A Modular Instruction Set Architecture**

The Instruction Set Architecture (ISA) supports a uniform, 32-bit address space, with optional virtual addressing and 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 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

2001-04-30 @ 15:16

and by providing a flexible hardware-supported interrupt scheme. Furthermore, the architecture supports fast context switching.

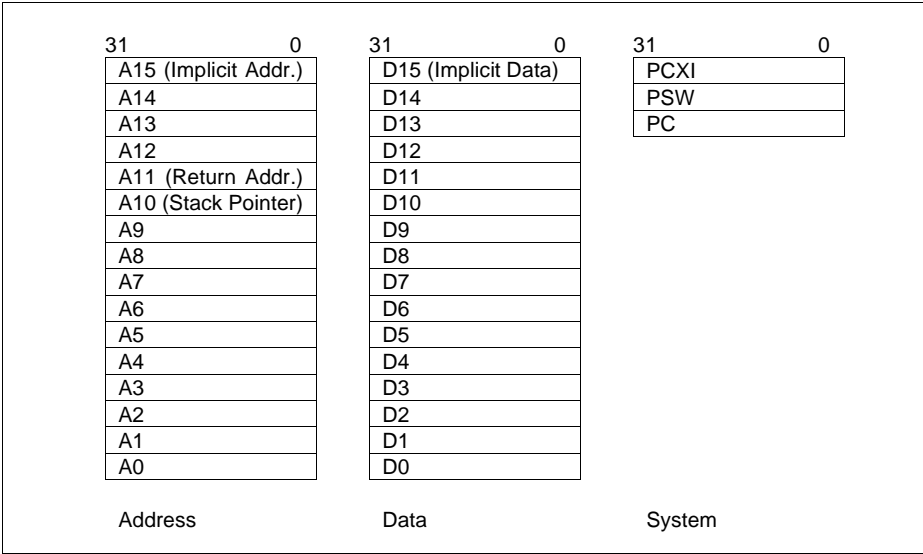
## **1.1 Feature Overview**

- 32-bit architecture
- 4-GByte virtual or physical data, program, and input/output address spaces
- Full-featured memory management system
- 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
- Byte and bit addressing
- Little-endian byte ordering
- Flexible interrupt prioritization scheme
- Memory protection
- Debug support
- Flexible power management

## **1.2 Program State Registers**

The 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).

2001-04-30 @ 15:16



**Figure 1-2**  
**Program State Registers**

The 32 General-purpose Registers (GPRs) are divided into sixteen 32-bit data registers (D0 through D15); and sixteen 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 address register.

Registers 0-7 are called the lower registers and 8-15 are called the upper registers.

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.

### 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.



## 1.4 Addressing Modes

Addressing modes allow load and store instructions to efficiently access simple variables and data elements within data structures such as records, randomly and sequentially accessed arrays, stacks, and circular buffers. Simple variables and 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 7 addressing modes are supported in the architecture.

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

## 1.5 Instruction Formats

The 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; they are included to reduce code space.

## 1.6 Tasks and Contexts

Throughout this book, the term task refers to an independent thread of control. There are 2 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 I/O 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.

### 1.6.1 Upper and Lower Contexts

The context is subdivided into the upper context and the lower context ([Figure 1-3](#)). 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.

2001-04-30 @ 15:16

Both upper and lower contexts include the PCXI 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 ISR if the ISR needs to use more registers than are provided by the upper context.

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

**Figure 1-3**  
**Upper and Lower Contexts**

**1.6.2 Context Save Areas**

The 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 memory storage, aligned on a 16-word boundary. A single CSA can hold exactly 1 upper or 1 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 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 a complete upper or lower context to be saved in as little as 2 clock cycles.

## 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.

## 1.8 Trap System

A trap occurs as a result of an exception within one of the following 8 classes:

- Internal protection
- Instruction errors
- Context management
- Memory 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.

## 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.

2001-04-30 @ 15:16

#### 1.10 Memory Management System

The principal features of the TriCore memory management include:

- 4-GBYTE virtual address space divided into sixteen 256 MB segments
- 4-GBYTE physical address space divided into sixteen 256 MB segments
- Addressing by direct translation or via Page Table Entries (PTE)
- Two addressing modes: physical and virtual (physical page attributes override virtual page attributes)

The virtual address space is divided into 16 segments of 256 MB each. The physical address space is also divided into 16 segments of 256 MB each. Virtual addresses are always translated into physical addresses before accessing memory.

The virtual address is translated into a physical address using either direct translation or Page Table Entry (PTE) translation.

- Direct translation: If the virtual address belongs to the upper half of the virtual address space then the virtual address is directly used as the physical address. If the virtual address belongs to the lower half of the address space, then the virtual address is used directly as the physical address if the processor is operating in Physical mode.
- PTE translation: If the virtual address belongs to the lower half of the address space, then the virtual address is translated using a Page Table Entry if the processor is operating in Virtual mode.

PTE translation is performed by replacing the Virtual Page Number (VPN) of the virtual address by a Physical Page Number (PPN) to obtain a physical address. Six memory-mapped MMU Core Special Function Registers (CSFRs) control the memory management system.

##### 1.10.1 Permission Levels

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 I/O mode bits in the PSW. The 3 permission levels are User-0, User-1, and Supervisor:

- User-0 mode
  - 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
  - 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 all peripheral devices.
  - Tasks at this level may disable interrupts.

##### 1.10.2 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

2001-04-30 @ 15:16

permissions are specified in 2-4 identical sets of tables residing in the Core Registration Function Register (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 and instruction fetch address for legality before performing the access. To be legal, the address must fall within 1 of the ranges specified in the currently selected PRS, and permission for that type of access must be present in the matching range.

### **1.11 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 which has a special protection mechanism to prevent accidental accesses.
  - Implementation-specific controls in this register facilitate either partial or 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 of the triggers caused the reset.

### **1.12 Debug System**

The TriCore contains mechanisms and resources to support on-chip debugging. These are used by the Debug Control Unit. Most functions and details of the Debug Control Unit are implementation-specific. This document does not provide further descriptions of the debug control unit and its associated registers. Please contact your local Infineon sales office for literature or further information.

2001-04-30 @ 15:16

---

# Programming Model

---

2001-04-30 @ 15:16

## 2 Programming Model

This chapter discusses the following aspects of the TriCore architecture that are visible to software:

- Supported data types
- Formats of data types in registers and memory
- Various addressing modes that the architecture provides
- 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.

#### 2.1.1 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.

#### 2.1.2 Bit String

A bit string is a packed field of bits. Bit strings are produced and used by logical, shift, and bit field instructions.

#### 2.1.3 Character

A character is an 8-bit value that is a very short unsigned integer. No specific coding is assumed.

#### 2.1.4 Signed Fraction

The TriCore architecture supports 16-/32-bit signed fractional data for DSP arithmetic. Data values in this format have a single high-order sign bit, where 0 represents positive (+) and 1 represents negative (–), followed by an implied binary point and fraction. Thus their values are in the range [–1,1).

#### 2.1.5 Address

An address is a 32-bit unsigned value.

#### 2.1.6 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.



### 2.1.7 IEEE-754 Single-precision Floating-point Number

Depending on the particular implementation of the core architecture, IEEE-754 floating-point numbers are supported by co-processor hardware instructions or by software calls to a library.

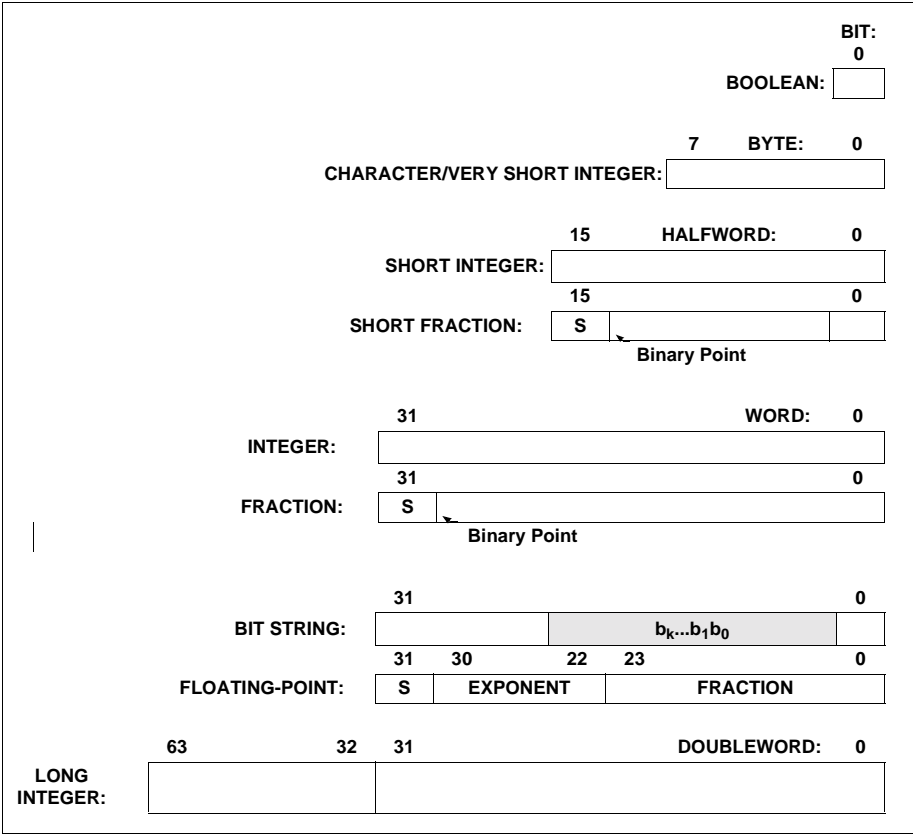
## 2.2 Data Formats

All the General-purpose registers (GPRs) are 32 bits wide, and most instructions operate on word (32-bit) values. When byte or halfword data elements are loaded from memory, they are automatically sign-extended or zero-extended to fill the register. The type of filling is implicit in the load instruction: e.g., `LD.B` to load a byte with sign extension, vs. `LD.BU` to load a byte with zero extension.

Alignment requirements differ for addresses and data. Address variables, loaded into or stored from address registers, must always be word-aligned. For transfers between data registers and memory, there is some relaxation of the natural alignment restrictions. In most cases, data may be aligned on any halfword boundary, regardless of size. This facilitates the use of packed arithmetic operations in DSP applications, by allowing two or four packed 16-bit data elements to be loaded or stored together on any halfword boundary. However, there are some restrictions of which programmers must be aware. Specifically:

- The `LDMST` and `SWAP` instructions require their operands to be word-aligned;
- Halfword alignment for `LD.D` and `ST.D` is only allowed when the source or destination address is targeted at local data or cached memory. For all other addresses doubleword accesses must be word-aligned.

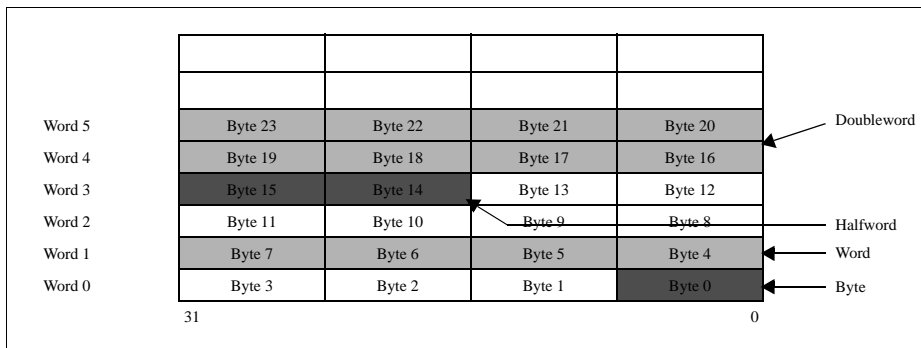
Figure 2-1 illustrates the data types supported.



**Figure 2-1**  
**Supported Data Formats**

The data memory and CPU registers store data in little-endian byte order (the least-significant bytes are at lower addresses). **Figure 2-2** illustrates the byte ordering. Little-endian memory referencing is used consistently for data and instructions.

2001-04-30 @ 15:16



**Figure 2-2**  
**Byte Ordering**

## 2.3 Memory Model

The TriCore architecture has an address width of 32 bits and can access up to 4 GB of memory. The address space (Table 2-1) is divided into 16 regions or segments (0 through 15). Each segment is 256 MB. The upper 4 bits of an address select the specific segment. The first 16 KB of each segment can be accessed using either absolute addressing or absolute bit addressing.

**Table 2-1**  
**Physical Address Space**

| Address                         | Segments | Description   |
|---------------------------------|----------|---|
| 0xFFFF FFFF<br>:<br>0xE000 0000 | 14 - 15  | non-speculative access<br>no User 0 access  |
| 0xDFFF FFFF<br>:<br>0x8000 0000 | 8-13     | normal access<br>Detailed limitations are implementation-specific.<br>See device-specific System User Guide |
| 0x7FFF FFFF<br>:<br>0x0000 0000 | 0-7      | reserved  |

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 attempted, will cause a trap. This restriction allows direct determination of the accessed segment from the base address.

Segments 0-7 are reserved for future use. Accesses to these segments will cause a trap. Accesses to segment 14 and 15 are guaranteed to be non-speculative but they are not accessible in User 0 mode. These segments can thus be used for mapping peripheral registers. The Core Special Function Registers (CSFRs) are mapped to a 64-KB space in the memory map. The base location of this 64-KB space is implementation-dependent. Segments 8-13 will have further limitations placed

2001-04-30 @ 15:16

upon them in some implementations. For example, specific segments for program and data may be defined by device-specific implementations. Other details of the memory mapping are implementation-specific. Please refer to the device-specific User's Manual.

## 2.4 Addressing Model

### 2.4.1 Addressing Modes

Addressing modes (Table 2-2) allow load and store instructions to access simple data elements such as records, randomly and sequentially accessed arrays, stacks, and circular buffers effectively within data structures. Simple data elements are 8, 16, 32, or 64 bits wide.

The addressing modes were selected to 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).

**Table 2-2**  
**Supported Addressing Modes**

| Addressing Mode       | Address Register Use  | Offset Size (bits) |
|-----------------------|-----------------------|--------------------|
| <b>absolute</b>       | none                  | 18                 |
| <b>base+offset</b>    | address register      | 10 <sup>1)</sup>   |
| <b>pre-increment</b>  | address register      | 10                 |
| <b>post-increment</b> | address register      | 10                 |
| <b>circular</b>       | address register pair | 10                 |
| <b>bit-reverse</b>    | address register pair | —                  |

1) A subset of memory operations support a base + long offset addressing mode which provides a 16-bit offset.

The instruction formats were chosen to provide as many bits of address as possible for absolute addressing and as large of range of offsets as possible for base+offset addressing.

Note it is possible for an address register to be both the target of a load and an update associated with a particular addressing mode. For example, in the following case, the contents of the address register are not architecturally defined:

```
ld.a a0, [a0+]4
```

In a similar manner, consider the following case:

```
st.a [+a0]4, a0
```

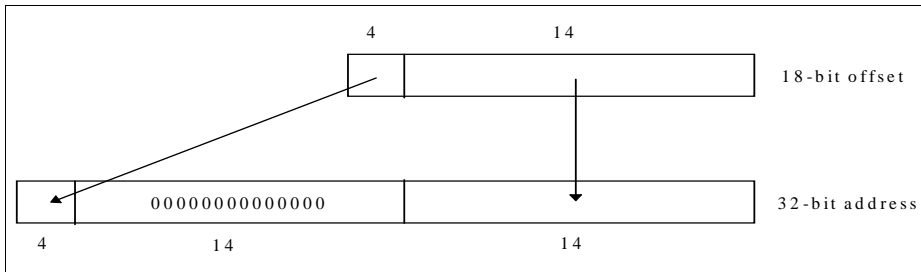
It is not architecturally defined whether the original or updated value of a0 is stored into memory. This is true for all addressing modes in which there is an update of the address register.

#### 2.4.1.1 Absolute Addressing

Absolute addressing is useful for referencing I/O peripheral registers and global data. It uses an 18-bit constant specified by the instruction as the memory address. The full 32-bit address results from

2001-04-30 @ 15:16

moving the most significant 4 bits of the 18-bit constant to the most significant bits of the 32-bit address (Figure 2-3). The other bits are zero-filled.



**Figure 2-3**  
**Translation of Absolute Address to Full Effective Address**

#### 2.4.1.2 Base+Offset Addressing

Base+offset is useful for referencing record elements, local variables (using 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 10-bit offset.

A subset of the memory operations are provided with a base + long offset addressing mode. In this mode the offset is a 16-bit sign-extended value. This allows any location in memory to be addressed using a two instruction sequence.

#### 2.4.1.3 Post-Increment Addressing

Post-increment and post-decrement addressing, where the latter is obtained by the use of a negative offset, may be used for forward or backward sequential access of arrays, respectively. Further, the two versions of the mode may be used to pop from a downward- or upward-growing stack, respectively.

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.

#### 2.4.1.4 Pre-Increment Addressing

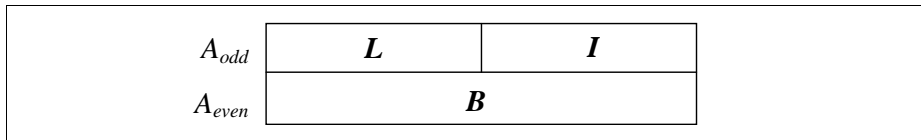
Pre-increment and pre-decrement addressing, where the latter is obtained by the use of a negative offset, may be used to push onto an upward- or downward-growing stack, respectively.

The pre-increment addressing mode uses the sum of the address register and the offset both as the effective address and as the value written back into the address register.

#### 2.4.1.5 Circular Addressing

The primary use of circular addressing (Figure 2-4) is for accessing data values in circular buffers while performing filter calculations.

2001-04-30 @ 15:16



**Figure 2-4**  
**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.

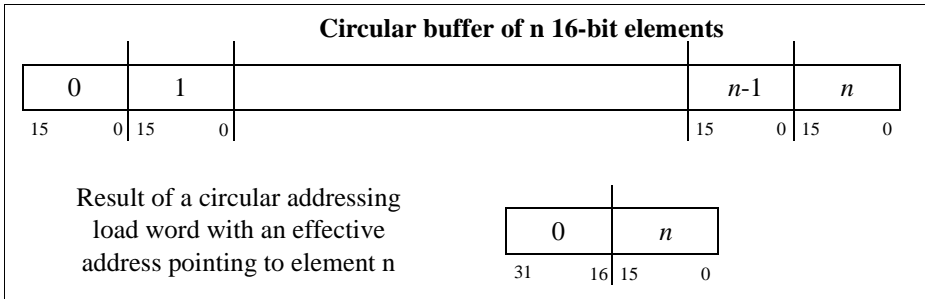
The index is post-incremented using the following algorithm:

```
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 “wrap around” behavior is guaranteed as long as the magnitude of the offset is smaller than size of the buffer.

To illustrate the use of circular addressing, consider a circular buffer consisting of 25 16-bit values. If the current index is 48, then the next item is obtained using an offset of 2 (2 bytes per value). The new value of the index “wraps around” to 0. Instead if we are at an index of 48 and use an offset of 4, the new value of the index is 2. If the current index is 4 and we use an offset of -8, then the new index is 46 (4-8+50).

In the end case, where a memory access runs off the end of the circular buffer (**Figure 2-5**), the data access also wraps round to the start of the buffer. For example, consider a circular buffer containing  $n+1$  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  $n$ , the 32-bit result will contain element  $n$  in the bottom 16 bits and element 0 in the top 16 bits.



**Figure 2-5**  
**Circular Buffer End Case**

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

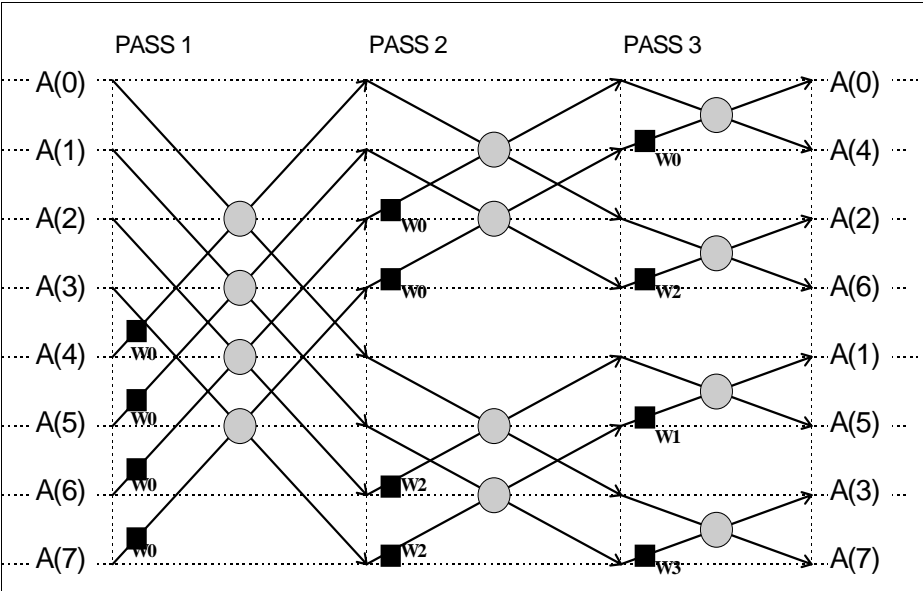
- The start of the buffer must be aligned to a 64-bit boundary.
  - An implementation is free to advise the user of optimal alignment of circular buffers etc., but must support the above.
- The length of the buffer must be a multiple of the data size, where the data size is determined from the instruction being used to access the buffer.
  - For example, a buffer accessed using a load word instruction must be a multiple of 4 bytes in length, and a buffer accessed using a load double word instruction must be a multiple of 8 bytes in length.

If the above restrictions are not met, the implementation will take an alignment trap. An alignment trap is also taken if the index (I)  $\geq$  length (L).

#### 2.4.1.6 Bit-reverse Addressing

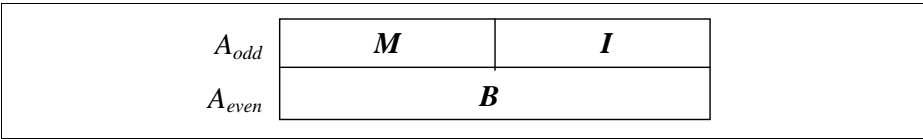
Bit-reverse addressing is used to access arrays used in FFT algorithms. The most common implementation of the FFT ends with results stored in bit-reversed order (**Figure 2-6**).

2001-04-30 @ 15:16



**Figure 2-6**  
**Bit-Reverse Addressing**

Bit-reverse addressing (Figure 2-7) uses an address register pair to hold the required state.



**Figure 2-7**  
**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 ( $I$ ), and the most-significant half is the modifier ( $M$ ) which is used to update  $I$  after every access.

The effective address is  $B+I$ . The index,  $I$ , is post-incremented and its new value is  $reverse(I) + reverse(M)$ . The  $reverse(I)$  function exchanges bit  $n$  with bit  $(15-n)$  for  $n = 0, \dots, 7$ .



To illustrate, for a 1024-point real FFT using 16-bit values, the buffer size is 2048 bytes. Stepping through this array using a bit-reverse index would give the sequence of byte indices: 0, 1024, 512, 1536, .... This sequence can be obtained by initializing I to 0 and M to 0x0400 (see [Table 2-3](#)).

**Table 2-3**  
**1024-point FFT Using 16-bit Values**

| I (decimal) | I (binary)       | Reverse(I)       | Rev(I) + Rev(M)  |
|-------------|------------------|------------------|------------------|
| 0           | 0000000000000000 | 0000000000000000 | 0000010000000000 |
| 1024        | 0000010000000000 | 0000000000100000 | 0000001000000000 |
| 512         | 0000001000000000 | 0000000001000000 | 0000011000000000 |
| 1536        | 0000011000000000 | 0000000001100000 | 0000010001100000 |

The value of M required is given by buffer size/2 where the buffer size is given in bytes.

## 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 1, 2, 4, or 8 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 1/8 (shifts right 3 bits) and adds it to the address register. The 2 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 EXTR.U 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 2 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 + 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. The absolute address of the PC relative data is also known, and there is no need to synthesize PC-relative addressing.

2001-04-30 @ 15:16

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.

---

# Core Registers

---

2001-04-30 @ 15:16

### 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
- Context Management
- Stack Management
- Interrupt and Trap Control
- System Control
- Memory Protection
- Memory Management
- Debug Control

The following sections describe these registers in detail. The CSFRs are complemented by a set of General-purpose Registers (GPRs). **Table 3-1** shows all CSFRs and GPRs.

Most of the memory protection system and debug control unit is implementation-specific; therefore, this architecture manual only summarizes these topics. 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 Infineon Sales office for more information on literature availability.

**Table 3-1**  
**Core Register Map**

| Register Name/Acronym  | Description                                       |
|------------------------|---|
| <b>D0 – D15</b>        | Data Registers                                    |
| <b>A0 – A15</b>        | Address Registers                                 |
| <b>PSW</b>             | Program Status Word                               |
| <b>PCXI</b>            | Previous Context Information                      |
| <b>PC</b>              | Program Counter (read only)                       |
| <b>FCX</b>             | Free Context List Head Pointer                    |
| <b>LCX</b>             | Free Context List Limit Pointer                   |
| <b>ISP</b>             | Interrupt Stack Pointer                           |
| <b>ICR</b>             | Interrupt Control Register                        |
| <b>BIV</b>             | Base Address of Interrupt Vector Table            |
| <b>BTB</b>             | Base Address of Trap Vector Table                 |
| <b>SYSCON</b>          | System Configuration Register                     |
| <b>DPRx_0 – DPRx_3</b> | Data Segment Protection Register Sets (x = 0 – 3) |
| <b>CPRx_0 – CPRx_3</b> | Code Segment Protection Register Sets (x = 0 – 3) |
| <b>DPMx_0 – DPMx_3</b> | Data Protection Mode Register Sets (x = 0 – 3)    |
| <b>CPMx_0 – CPMx_3</b> | Code Protection Mode Register Sets (x = 0 – 3)    |

**Table 3-1  
Core Register Map (cont'd)**

| Register Name/Acronym | Description                          |
|-----------------------|--------------------------------------|
| <b>DBGSR</b>          | Debug Status Register                |
| <b>EXEVT</b>          | External Break Input Event Specifier |
| <b>SWEVT</b>          | Software Break Event Specifier       |
| <b>CREVT</b>          | Core SFR Access Event Specifier      |
| <b>TRnEVT</b>         | Trigger Event n Specifier (n = 0, 1) |
| <b>MMUCON</b>         | MMU Configuration register           |
| <b>ASI</b>            | MMU Address Space Identifier         |
| <b>TVA</b>            | MMU Translation Virtual Address      |
| <b>TPA</b>            | MMU Translation Physical Address     |
| <b>TPX</b>            | MMU Translation Page Index           |
| <b>TFA</b>            | MMU Translation Fault Address        |

### 3.1 Access to the Core Registers

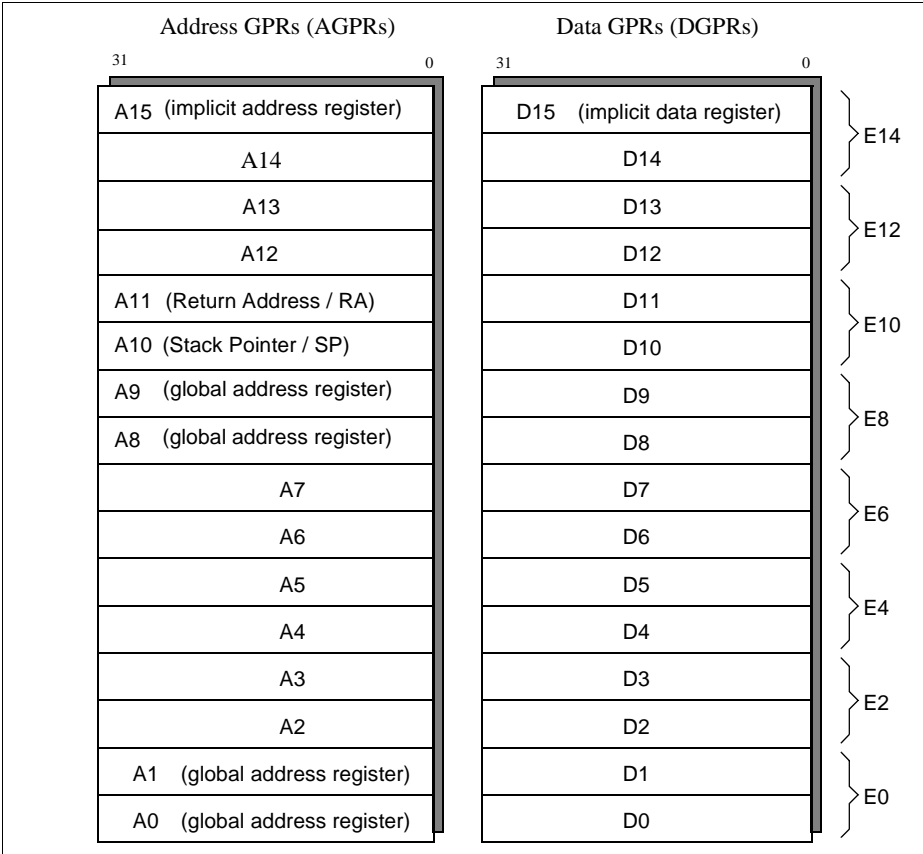
The core accesses the CSFRs through two instructions: MFCR and MTCR. The MFCR instruction (Move From Core Register) moves the contents of the addressed CSFR into a data register. MFCR can be executed on any privilege level. The MTCR instruction (Move To Core Register) moves the contents of a data register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

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

Figure 3-1 shows the GPRs. The 32-bit wide GPRs 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 8 extended-size registers (E0, E2, E4, E6, E8, E10, E12, and E14), in order to support 64-bit values.

2001-04-30 @ 15:16



**Figure 3-1**  
**General-Purpose Registers**

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.

While the 32-bit instructions have unlimited use 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.

### 3.3 Program State Information

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

The 32-bit Program Counter (PC) is shown below. The PC contains the address of the instruction that is currently executing. The PC is part of a task's state information.

|                        |   |          |
|------------------------|---|----------|
| 31                     | 1 | 0        |
| <b>Program Counter</b> |   | <b>0</b> |

#### 3.3.2 Program Status Word

The **Program Status Word** (PSW) is a 32-bit register that contains task-specific architectural state not captured in the general purpose register values. The lower half holds control values and parameters related to the protection system. Included are the Protection Register Set (PRS), the I/O privilege level (IO), the Interrupt Stack flag (IS), the Global register Write permission flag (GW) and the Call Depth Counter and Call Depth count Enable field (CDC and CDE). These are described in more detail below.

|                  |  |     |  |    |  |    |    |     |     |    |    |   |   |   |   |   |  |  |  |
|------------------|--|-----|--|----|--|----|----|-----|-----|----|----|---|---|---|---|---|--|--|--|
| 31               |  |     |  |    |  |    | 24 |     |     |    | 16 |   |   |   |   |   |  |  |  |
| User Status Bits |  |     |  |    |  |    |    | Res |     |    |    |   |   |   |   |   |  |  |  |
| 15               |  |     |  |    |  |    | 14 | 13  | 12  | 11 | 10 | 9 | 8 | 7 | 6 | 0 |  |  |  |
| Res              |  | PRS |  | IO |  | IS | GW | CDE | CDC |    |    |   |   |   |   |   |  |  |  |

| Field      | Bits  | Type | Value  | Description               |
|------------|-------|------|--|---------------------------|
| <b>USB</b> | 31:24 | rw   |  | User Status Bits          |
| <b>PRS</b> | 13:12 | rw   | Protection Register Set. 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 |

2001-04-30 @ 15:16

| Field      | Bits  | Type | Value   | Description   |
|------------|-------|------|---|---|
| <b>IO</b>  | 11:10 | rw   | I/O Privilege. This field selects the I/O privilege mode.   |   |
|            |       |      | 00  | User-0  |
|            |       |      | 01  | User-1  |
|            |       |      | 10  | Supervisor  |
|            |       |      | 11  | Reserved  |
| <b>IS</b>  | 9     | rw   | Interrupt Stack. This bit reflects the status of the current task.  |   |
|            |       |      | 0   | Current task uses a user stack  |
|            |       |      | 1   | Current task uses the global interrupt stack  |
| <b>GW</b>  | 8     | rw   | Global Register Write Permission. 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  |
| <b>CDE</b> | 7     | rw   | Call Depth Count Enable. 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 = 111111 <sub>2</sub> , call depth counting is disabled regardless of the setting on this bit. |
| <b>CDC</b> | 6:0   | rw   | The Call Depth Overflow 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. |   |
|            |       |      | 0cccccc   | 6-bit counter; trap on overflow   |
|            |       |      | 10cccc  | 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   |

The eight most significant bits of the PSW are designated as **User Status Bits**. These bits may be set or cleared as execution side effects of user instructions, typically recording result status. Individual bits can also be used to condition the operation of particular instructions. For example, the ADDX and ADDC instructions use bit 31 to record the carry out from the ADD operation, and the pre-execution value of the bit is reflected in the result of the ADDC instruction.

There are two classes of instructions that employ the user status bits. The first class are the arithmetic instructions that may produce carry and overflow results. The second class are the implementation-specific coprocessor instructions. Instructions in the first class use five of the bits,



2001-04-30 @ 15:16

with the labels and meanings described in the figure and table below. Instructions in the second class may use any or all of the eight bits, in a manner that is entirely implementation-specific. The RSTV instruction resets the four bits, 30:27, used by the arithmetic instructions to record overflow status. It does not modify any other bits.

| 31       | 30       | 29        | 28        | 27         | 26         | 25 | 24 |
|----------|----------|-----------|-----------|------------|------------|----|----|
| <b>C</b> | <b>V</b> | <b>SV</b> | <b>AV</b> | <b>SAV</b> | <b>Res</b> |    |    |

| Field      | Bits  | Type | Value | Description             |
|------------|-------|------|-------|-------------------------|
| <b>C</b>   | 31    | rw   |       | Carry                   |
| <b>V</b>   | 30    | rw   |       | Overflow                |
| <b>SV</b>  | 29    | rw   |       | Sticky Overflow         |
| <b>AV</b>  | 28    | rw   |       | Advance Overflow        |
| <b>SAV</b> | 27    | rw   |       | Sticky Advance Overflow |
| <b>Res</b> | 26:24 |      |       | Reserved                |

Bits 23:16 of the PSW are reserved bits, with no defined use in current revisions of the architecture. They read as zero, when the PSW is read via the MFCR instruction after a system reset. Their value, after writing to the PSW via the MTCR instruction, is architecturally undefined.

### 3.3.3 Previous Context Information Register

Previous Context Information Register (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 |
| PCPN |    | PIE | UL | -  |    | PCXS |    |
| 15   |    |     |    |    |    |      | 0  |
| PCXO |    |     |    |    |    |      |    |

2001-04-30 @ 15:16

| Field       | Bits  | Type | Value | Description  |
|-------------|-------|------|-------|--|
| <b>PCPN</b> | 31:24 | rw   |       | Previous CPU Priority Number. This field contains the priority level number of the interrupted task.   |
| <b>PIE</b>  | 23    | rw   |       | Previous Interrupt Enable. This bit indicates the state of the interrupt enable bit (ICR.IE) for the interrupted task.   |
| <b>UL</b>   | 22    | rw   |       | Upper/Lower Context Tag. 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. |
| <b>PCXS</b> | 19:16 | rw   |       | PCX Segment Address. This field contains the segment address portion of the PCX.   |
| <b>PCXO</b> | 15:0  | rw   |       | Previous Context Pointer Offset Field. The PCXO and PCXS fields form the pointer PCX, which points to the CSA of the previous context.   |

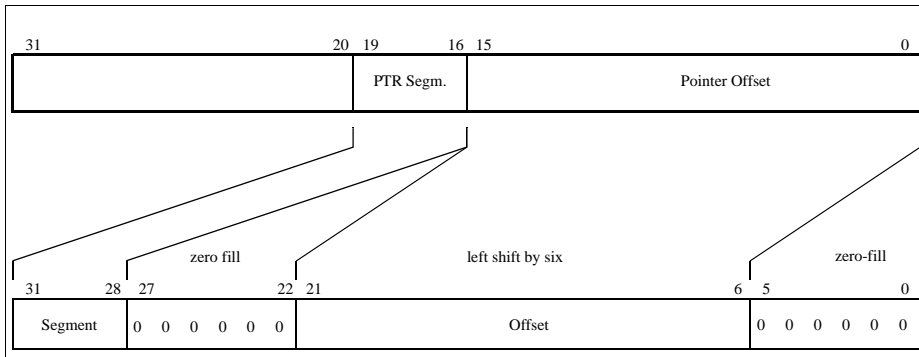
### 3.4 Context Management Registers

This section describes the context management registers, which are comprised of 3 pointers that handle context management and are used during context save/restore operations ([Table 3-2](#)).

**Table 3-2**  
**Context Management Registers**

| Register   | Category  |
|------------|---|
| <b>FCX</b> | Free CSA List Head Pointer                            |
| <b>PCX</b> | Previous Context Pointer (contained in register PCXI) |
| <b>LCX</b> | Free CSA List Limit Pointer                           |

Each pointer consists of 2 fields: a 16-bit offset and a 4-bit segment specifier. [Figure 3-2](#) shows how the effective address of a CSA is generated using the 2 fields. A Context Save Area (CSA) is an address range containing 16 word locations (64 bytes), which is the space required to save 1 upper or 1 lower context. Incrementing the pointer offset value by 1 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 CSAs.


**Figure 3-2**
**Generation of the Effective Address for the Context Save Areas**

Note that the effective address is a physical memory address which must map to local data or cached memory. 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 CSAs.

**3.4.1 Free CSA List Head Pointer**

The Free CSA List Head Pointer (FCX) register holds the free CSA list head pointer, which always points to an available CSA.

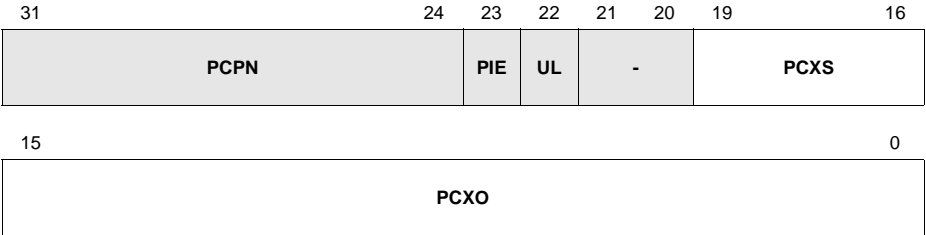


| Field       | Bits  | Type | Value | Description   |
|-------------|-------|------|-------|---|
| <b>FCXS</b> | 19:16 | rw   |       | FCX Segment Address Field. This field is used in conjunction with the FCXO field.   |
| <b>FCXO</b> | 15:0  | rw   |       | FCX Offset Address Field. The FCXO and FCXS fields together form the FCX pointer, which points to the next available CSA. |

2001-04-30 @ 15:16

### 3.4.2 Previous Context Pointer

The Previous Context Pointer (PCX) holds the address of the CSA of the previous task. PCX is part of PCXI. It is shown below for easy reference. The bits not relevant to the pointer function are shaded.



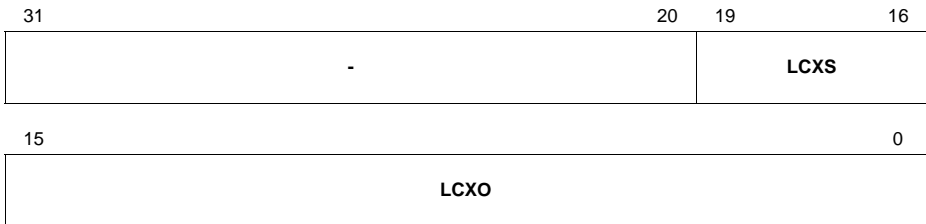
| Field       | Bits  | Type | Value | Description  |
|-------------|-------|------|-------|--|
| <b>PCXS</b> | 19:16 | rw   |       | PCX Segment Address Field. This field is used in conjunction with the PCXO field.  |
| <b>PCXO</b> | 15:0  | rw   |       | Previous Context Pointer Offset Field. The 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

The Free CSA List Limit Pointer (LCX) register is used to recognize impending CSA list underflows. If the new value of FCX resulting after a CALL or interrupt matches the limit value, the "free context depletion" condition is recognized, which triggers an FCD trap immediately after completion of the call or interrupt entry sequence. It is important to see the FCD trap description for details on the use and setting of LCX.

#### LCX

##### Free CSA List Limit Pointer



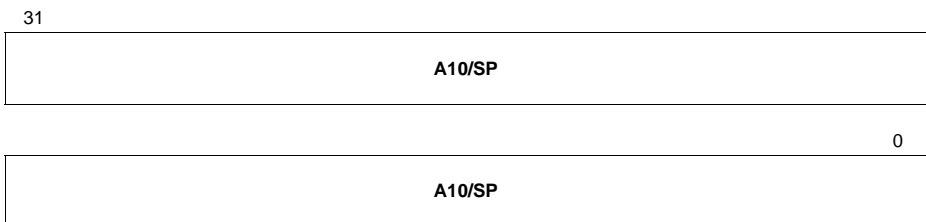
| Field       | Bits  | Type | Value | Description  |
|-------------|-------|------|-------|--|
| <b>LCXS</b> | 19:16 | rw   |       | LCX Segment Address. This field is used in conjunction with the LCXO field.                              |
| <b>LCXO</b> | 15:0  | rw   |       | LCX Offset Field. The LCXO and LCXS fields form the pointer LCX, which points to the last available CSA. |

### 3.5 Stack Management

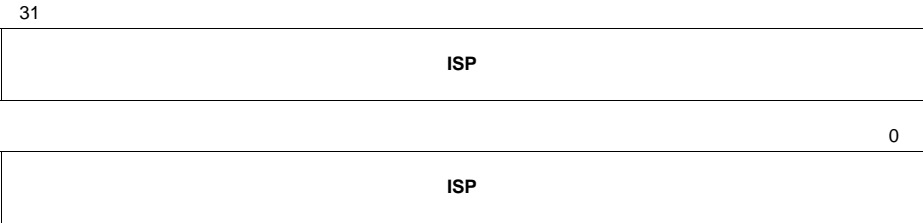
The stack management in the TriCore architecture supports a user stack and an interrupt stack. Address register A10, the Interrupt Stack Pointer (ISP), and a PSW bit are involved in the management of the stack.

#### A10/SP

##### Address Register A10/Stack Pointer



ISP  
Interrupt Stack Pointer



GPR 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 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 ISP 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, SP/A10 is loaded with the current contents of the 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).

3.6.1 Interrupt Control Register

The 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

2001-04-30 @ 15:16

and trap vector tables. The ICR register is below. Special instructions control the enabling and disabling of the interrupt system.

## ICR

### Interrupt Control Register

Reset Value : 0x0000 0000

|    |    |    |    |    |             |                                 |      |      |    |    |    |    |    |    |    |
|----|----|----|----|----|-------------|---------------------------------|------|------|----|----|----|----|----|----|----|
| 31 | 30 | 29 | 28 | 27 | 26          | 25                              | 24   | 23   | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
| -  |    |    |    |    | impl. spec. | implemen-<br>tation<br>specific | PIPN |      |    |    |    |    |    |    |    |
| 15 | 14 | 13 | 12 | 11 | 10          | 9                               | 8    | 7    | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
| -  |    |    |    |    |             |                                 | IE   | CCPN |    |    |    |    |    |    |    |

| Field                   | Bits       | Type | Value        | Function  |
|-------------------------|------------|------|--------------|---|
| implementation specific | ICR[26]    |      |              | Implementation specific control of the arbitration. See documentation for specific TriCore product.             |
| implementation specific | ICR[25:24] |      |              | Implementation specific control of the arbitration. See documentation for specific TriCore product.             |
| PIPN                    | ICR[23:16] | rh   | 0x00<br>0xYY | Pending Interrupt Priority Number:<br>No valid pending request.<br>A request with priority YY is pending.       |
| IE                      | ICR[8]     | rwh  | 0<br>1       | Global Interrupt Enable Bit:<br>Interrupt system is globally disabled.<br>Interrupt system is globally enabled. |
| CCPN                    | ICR[7:0]   | rwh  |              | Current CPU Priority Number:  |
| -                       |            | r    |              | Reserved.   |

Note: Type: "h" means this bit(s) may be updated by hardware.

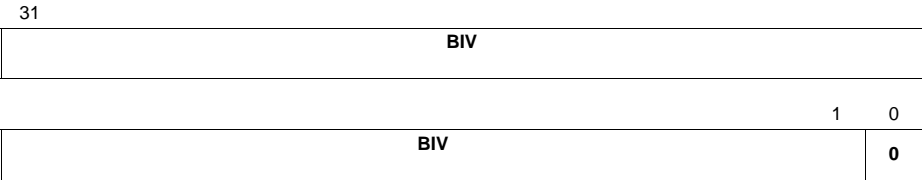
### 3.6.2 Interrupt Vector Table Pointer

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.

2001-04-30 @ 15:16

The left-shift of the interrupt priority number results in a spacing of 8 words (32 bytes) between the individual entries in the vector table.

**BIV**  
**Interrupt Vector Table Pointer**

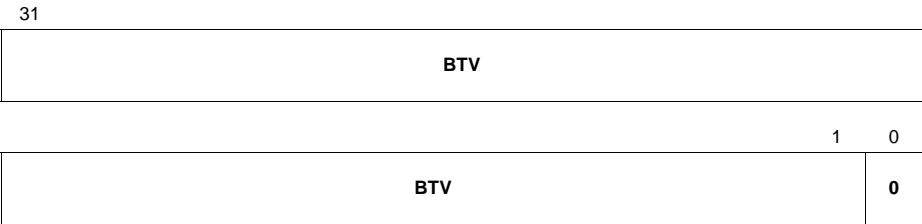


Care must be taken regarding the alignment of the address contained in the BIV register. First, the address in the BIV register must be aligned to an even byte address (halfword address). Second, due to the simple ORing of the left-shifted priority number and the contents of the BIV register, the alignment 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**

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 Class of that trap, left-shifted by 5 bits and then ORed with the contents of the BTV register. The left-shift of the Trap Class results in a spacing of 8 words (32 bytes) between the individual entries in the vector table.

**BTV**  
**Trap Vector Table Pointer**



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 8 different trap classes, resulting in Trap Classes from 0 to 7. Thus, the contents of BTV should be set at least to a 256-byte boundary (8 Trap Classes \* 8 word spacing).





2001-04-30 @ 15:16

Table 3-3  
MMU Registers

| Register Name/Acronym | Description                      |
|-----------------------|----------------------------------|
| MMUCON                | MMU Configuration register       |
| ASI                   | MMU Address Space Identifier     |
| TVA                   | MMU Translation Virtual Address  |
| TPA                   | MMU Translation Physical Address |
| TPX                   | MMU Translation Page Index       |
| TFA                   | MMU Translation Fault Address    |

### 3.9 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 3-3).

|                                       |                           |                                       |
|---------------------------------------|---------------------------|---------------------------------------|
| Data Memory Protection Register Set 0 | PSW.PRS = 00 <sub>2</sub> | Code Memory Protection Register Set 0 |
| Data Memory Protection Register Set 1 | PSW.PRS = 01 <sub>2</sub> | Code Memory Protection Register Set 1 |
| Data Memory Protection Register Set 2 | PSW.PRS = 10 <sub>2</sub> | Code Memory Protection Register Set 2 |
| Data Memory Protection Register Set 3 | PSW.PRS = 11 <sub>2</sub> | Code Memory Protection Register Set 3 |

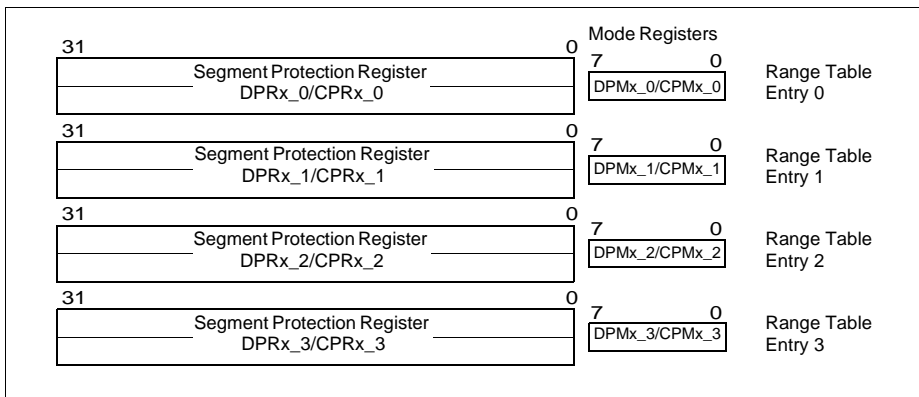
Figure 3-3  
Memory Protection Register Sets

The 2-bit PRS field in the PSW selects which register set is active at a given time. Two register sets are selected simultaneously: One data memory protection and one code memory protection.

The PSW.PRS field allows selection of up to 4 such register sets (4 for data and 4 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 Infineon Sales Office for additional information.

2001-04-30 @ 15:16

The number of range table entries is specific to each implementation of the TriCore architecture. Each range table entry (Figure 3-4) 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 3-4**  
**Range Table Entries in a Protection Register Set**

Table 3-4 lists the Memory Protection Registers. Index x indicates the protection register set number, while index n indicates the range table entry number.

**Table 3-4**  
**Memory Protection Registers**

| Register | Description                       |
|----------|-----------------------------------|
| DPRx_n   | Data Segment Protection Registers |
| DPMx_n   | Data Protection Mode Registers    |
| CPRx_n   | Code Segment Protection Registers |
| CPMx_n   | Code Protection Mode Registers    |

### 3.9.1 Data and Code Segment Protection Registers

Below are the segment protection registers of a range table entry. The registers DPRx\_n and CPRx\_n are 2-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

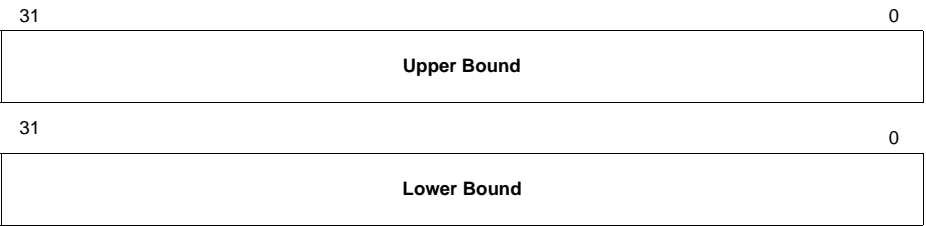
Range checking is not performed if the lower bound is greater than the upper bound. If the lower bound is equal to the upper bound, the range is regarded as empty.

2001-04-30 @ 15:16

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 1 or more of the DPRx\_n/CPRx\_n contents. For this purpose, an equality compare with the contents of the upper bound register is performed.

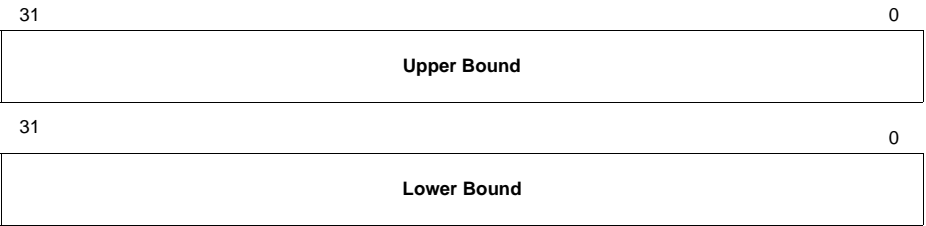
### DPRx\_n

#### Data Segment Protection Registers



### CPRx\_n

#### Code Segment Protection Registers



### 3.9.2 Data Protection Mode Registers

The 8-bit Data Protection Mode Registers determine the access permissions and debug signal conditions for the ranges specified in their corresponding Data Segment Protection Registers. The assignment and definition of bits within a mode table entry for the data range is shown below. The WE and RE bits relate directly to memory protection. The remaining bits generate signals to the Debug Control Unit.

### DPMx\_n

#### Data Protection Mode Register

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

2001-04-30 @ 15:16

| Field      | Bits | Type | Value | Description   |
|------------|------|------|-------|---|
| <b>WE</b>  | 7    | rw   |       | Address Range Write Enable  |
|            |      |      | 0     | Data write accesses to associated address range not permitted   |
|            |      |      | 1     | Data write accesses to associated address range permitted   |
| <b>RE</b>  | 6    | rw   |       | Address Range Read Enable   |
|            |      |      | 0     | Data read accesses to associated address range not permitted  |
|            |      |      | 1     | Data read accesses to associated address range permitted  |
| <b>WS</b>  | 5    | rw   |       | Address Range Data Write Signal   |
|            |      |      | 0     | Data write signal disabled  |
|            |      |      | 1     | Signal asserted to debug unit on data write accesses to associated address range  |
| <b>RS</b>  | 4    | rw   |       | Address Range Data Read Signal  |
|            |      |      | 0     | Data read signal disabled   |
|            |      |      | 1     | Signal asserted to debug unit on data read accesses to associated address range   |
| <b>WBL</b> | 3    | rw   |       | Data Write Signal on Lower Bound Access   |
|            |      |      | 0     | Data write signal disabled  |
|            |      |      | 1     | Signal asserted to debug unit on data write access to an address that matches lower bound address of associated address range |
| <b>RBL</b> | 2    | rw   |       | Data Read Signal on Lower Bound Access  |
|            |      |      | 0     | Data read signal disabled   |
|            |      |      | 1     | Signal asserted to debug unit on data read access to an address that matches lower bound address of associated address range  |
| <b>WBU</b> | 1    | rw   |       | Data Write Signal on Upper Bound Access   |
|            |      |      | 0     | Write signal disabled   |
|            |      |      | 1     | Signal asserted to debug unit on data write access to an address that matches upper bound address of associated address range |
| <b>RBU</b> | 0    | rw   |       | Data Read Signal on Upper Bound Access  |
|            |      |      | 0     | Data read signal disabled   |
|            |      |      | 1     | Signal asserted to debug unit on data read access to an address that matches upper bound address of associated address range  |

2001-04-30 @ 15:16

### 3.9.3 Code Protection Mode Registers

The 8-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. Below are 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.

CPMx\_n

#### Code Protection Mode Register

|           |   |           |   |           |   |   |           |
|-----------|---|-----------|---|-----------|---|---|-----------|
| 7         | 6 | 5         | 4 | 3         | 2 | 1 | 0         |
| <b>XE</b> | - | <b>XS</b> | - | <b>BL</b> | - | - | <b>BU</b> |

| Field     | Bits | Type | Value | Description  |
|-----------|------|------|-------|--|
| <b>XE</b> | 7    | rw   |       | Address Range Execute Enable   |
|           |      |      | 0     | Instruction fetch accesses to associated address range not permitted   |
|           |      |      | 1     | Instruction fetch accesses to associated address range permitted   |
| -         | 6    | r    | -     | -  |
| <b>XS</b> | 5    | rw   |       | Address Range Execute Signal   |
|           |      |      | 0     | Execute signal disabled  |
|           |      |      | 1     | Signal asserted to debug unit on instruction fetch accesses to associated address range  |
| -         | 4    | r    | -     |  |
| <b>BL</b> | 3    | rw   |       | Execute Signal on Lower Bound Access   |
|           |      |      | 0     | Lower bound execute signal disabled  |
|           |      |      | 1     | Signal asserted to debug unit on instruction fetch access to an address that matches lower bound address of associated address range |
| -         | 2:1  | r    | -     |  |
| <b>BU</b> | 0    | rw   |       | Execute Signal on Upper Bound Access   |
|           |      |      | 0     | Upper bound execute signal disabled  |
|           |      |      | 1     | Signal asserted to debug unit on instruction fetch access to an address that matches upper bound address of associated address range |

### 3.10 Debug Registers

Seven registers are implemented in the core to support debugging (Table 3-5). These registers define the conditions under which a debug event is generated and the actions taken on the assertion of a debug event, and provide status information on the debug control unit.

**Table 3-5**  
**Debug Registers**

| Register                   | Description                             |
|----------------------------|---|
| <b>DBGSR</b>               | Debug Status Register                   |
| <b>EXEVT</b>               | External Break Input Event Specifier    |
| <b>SWEVT</b>               | Debug Instruction Break Event Specifier |
| <b>CREVT</b>               | Core SFR Access Break Event Specifier   |
| <b>TR<math>n</math>EVT</b> | Trigger Event $n$ Specifier             |

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





---

# Managing Tasks & Functions

---

2001-04-30 @ 15:16

### 4 Managing Tasks & 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 Real-time Operating System (RTOS), layered on top of the features and capabilities of the underlying machine architecture.

In the TriCore 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

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. 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). 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 offset. The link segment and link offset are used to generate the effective address of the linked CSA (**Figure 4-1**).

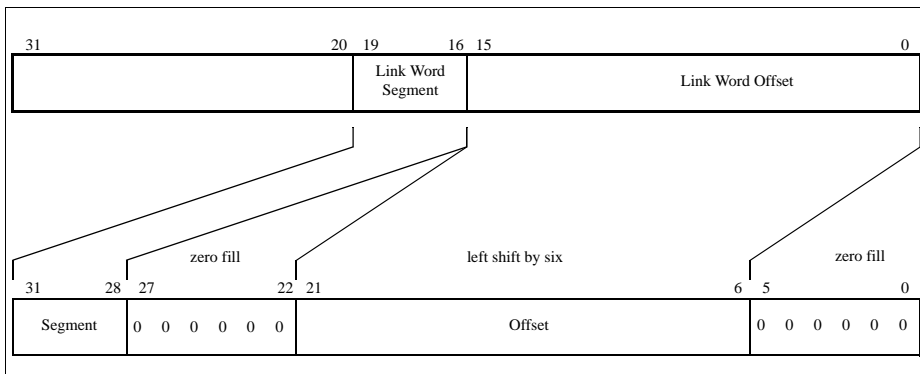


Figure 4-1

#### 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 [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 4-1](#) 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*.

There is an essential difference in the treatment of registers in the upper and lower contexts, in terms of how their contents are maintained. The lower context registers are similar to global registers, in the sense that their contents are unchanged across a call or interrupt. The called function or interrupt handler sees the values that were present in the registers before the call or interrupt. That means the lower context registers can be used to pass arguments to a called function. Likewise, since they are not automatically restored as part of the RET or RFE semantics, they can be used to pass return values from called functions.

The upper context registers, however, are different. They are not guaranteed to be static hardware registers. Conceptually, a called function or interrupt handler always begins execution with its own private set of upper context registers. The upper context registers of the interrupted or calling function are not inherited. Only A10 and A11, the SP and Return Address (RA) registers, start out with architecturally defined values in the called function or interrupt handler. A function or interrupt handler that reads any of the other upper context registers before writing a value into it is performing a suspect operation and the result is architecturally undefined.

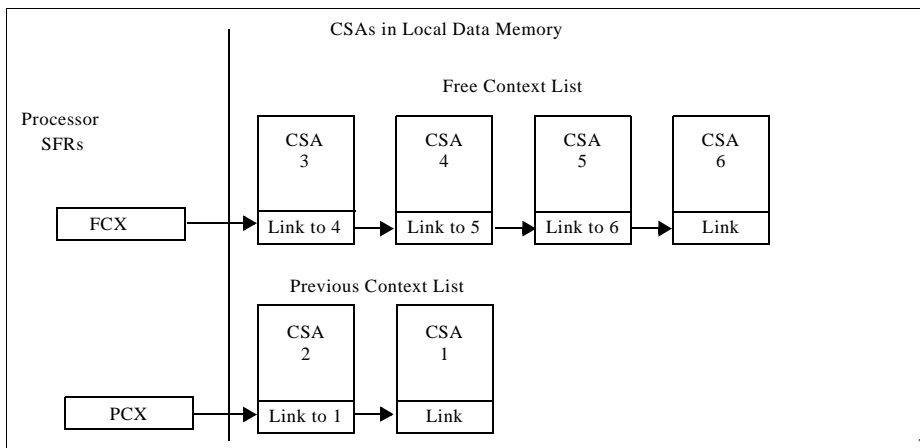
2001-04-30 @ 15:16

**Table 4-1**  
**Context-Related Events and Instructions**

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

## 4.3 CSAs and Context Lists

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 4-2** shows a simple configuration of CSAs within both context lists.



**Figure 4-2**  
**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 one of the last CSAs in the free list and is used to recognize impending CSA list underflow. If the new value of FCX after 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 RFE instruction.

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 the context save operation has been performed the RA is updated. For a call the RA is updated with the function return address. For a synchronous trap the RA is updated with the PC of the instruction which raised the trap. For an asynchronous trap or an interrupt the RA is updated with the PC of the next instruction to be executed.

When a lower context save operation is performed the value of the RA is included in the saved context and is placed in the second word of the CSA. This RA is correspondingly restored by a lower context restore.

2001-04-30 @ 15:16

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.

#### 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 SP is loaded with the current contents of the 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. ICR.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, ICR.PIPN must be greater than ICR.CCPN, and the interrupt enable bit (ICR.IE) must be set.

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 PCXI.PCPN value becomes the ICR.CCPN value, after saving the old PCPN value along with the old PCXI.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.

### 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.

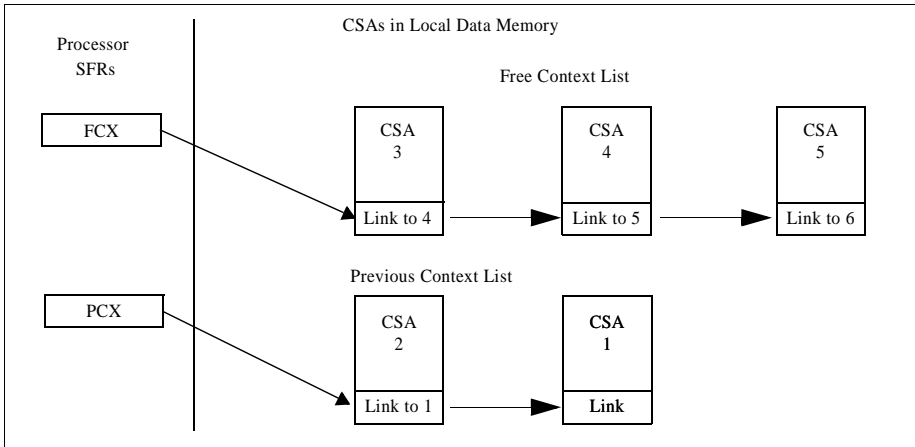
### 4.6 Context Save/Restore Examples

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

#### 4.6.1 Context Save

**Figure 4-3** 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 most recently saved CSA in the previous context list. The link word of CSA2 points to CSA1. CSA1 contains the saved context prior to CSA2.

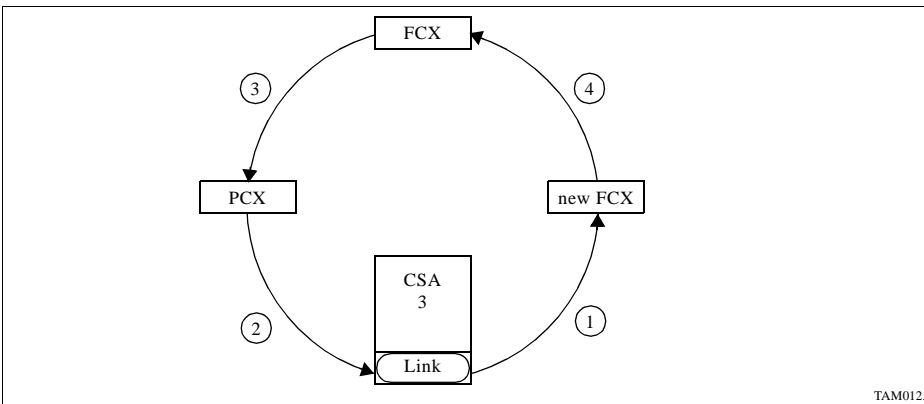
2001-04-30 @ 15:16



**Figure 4-3**  
**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 4-4** 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.



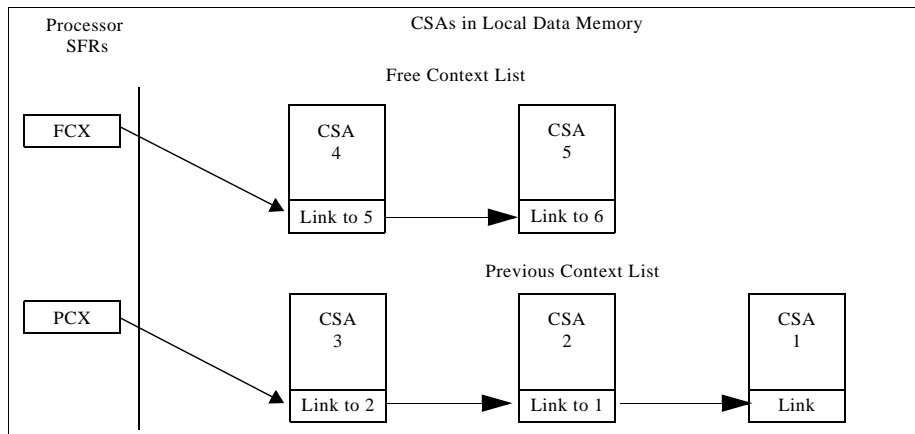
TAM012

**Figure 4-4**  
**CSA and Processor SFR Updates on a Context Save Process**



2001-04-30 @ 15:16

The processor SFRs and CSAs now look as shown in **Figure 4-5**. The processor context to be saved is now written into the rest of CSA3.

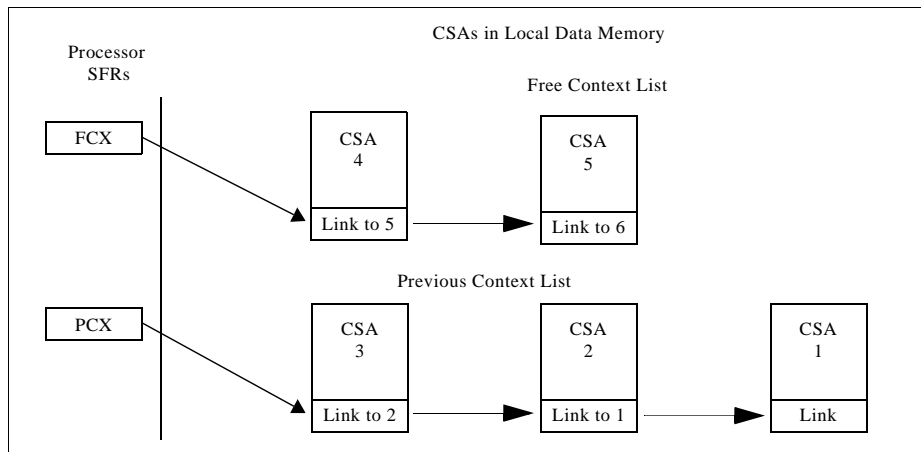


**Figure 4-5**  
**CSAs and Processor State After Context Save**

#### 4.6.2 Context Restore

**Figure 4-6** 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.

2001-04-30 @ 15:16

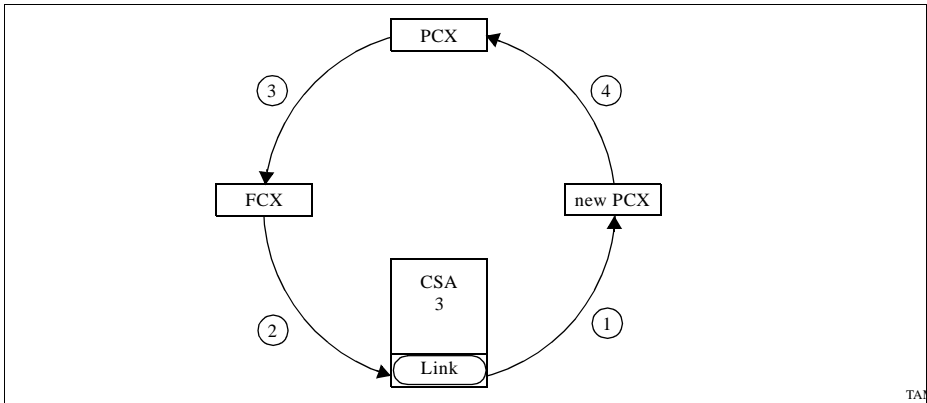


**Figure 4-6**  
**CSAs and Processor State Prior to Context Restore**

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 4-7** shows the steps taken during the context restore operation. The numbers in the figure correspond to the steps below:

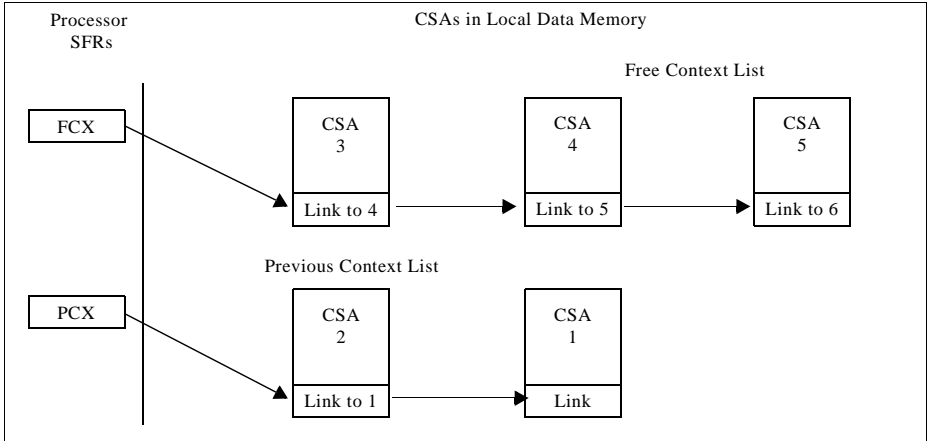
1. The contents of the link word in CSA3 are loaded into the new PCX. The new PCX will now point to CSA2. Note that the new PCX is an internal buffer and is not accessible by the user.
2. The contents of the FCX are written into the link word of CSA3. The link word of CSA3 now points to CSA4.
3. The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

2001-04-30 @ 15:16



**Figure 4-7**  
**CSA and Processor SFR Updates on a Context Restore Process**

The processor SFRs and CSAs now look as shown in **Figure 4-8**. The restored context now is written into the upper or lower context registers.



**Figure 4-8**  
**CSAs and Processor State After Context Restore**

2001-04-30 @ 15:16

---

# Interrupt System

---

2001-04-30 @ 15:16

## 5 Interrupt System

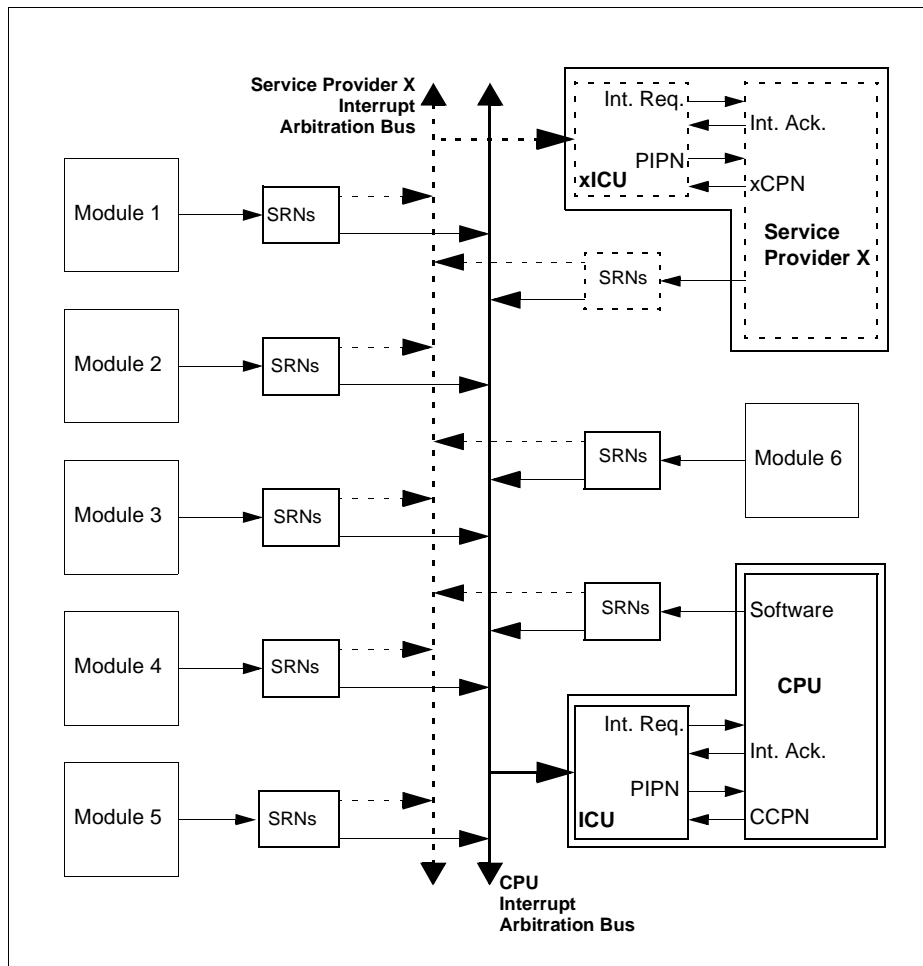
This chapter describes the interrupt system, including arbitration, the priority level scheme, and access to the vector table.

In a TriCore system, multiple sources such as peripherals or external inputs can generate an interrupt signal to the CPU to request for service. In addition, the TriCore interrupt system supports the implementation of additional units which are capable of handling interrupt requests, such as a second CPU, a standard DMA unit or a peripheral control processor. In the context of this chapter, therefore, such units are also called Service Providers, and interrupt requests are often referred to as Service Requests.

Besides the main CPU, up to three additional service providers can be handled with a TriCore interrupt SRN. The actual number of additional service providers implemented in a given device is implementation dependent.

Each interrupt or service request from a module connects to a service request node (SRN), containing a Service Request Control Register, xxSRC (xx refers to the requesting source). Interrupt arbitration busses connect the SRNs with the interrupt control units of the service providers. These control units handle the interrupt arbitration and the proper communication with the service provider.

**Figure 5-1** shows an overview of the TriCore interrupt system.



**Figure 5-1**  
**Block Diagram of the TC10 Interrupt System**

### 5.1 Service Request Node

Each service request node (SRN) contains a Service Request Control Register (xxSRC) and the necessary logic for the communication with the requesting source and the interrupt arbitration busses. A peripheral or other module can have several service request lines; each one of them connects to its own, individual SRN.

2001-04-30 @ 15:16

### Service Request Control Register

A typical service request control register in the TriCore architecture holds the individual control bits to enable/disable the request, to assign a priority number and to direct the request to one of the service providers. A request status bit shows whether or not the request is active. Besides being activated by the associated module through hardware, each request can also be set or reset through software.

The generic format and description of a service request control register, xxSRC ('xx' refers to the requesting source), is given below with a detailed description of each bit and bit field after the table (the 'xx' qualifier is omitted in these descriptions).

#### xxSRC

#### Source xx Service Request Control Register

Reset Value: 0000 0000<sub>H</sub>

|            |            |           |           |       |    |    |        |    |    |    |    |    |    |    |    |
|------------|------------|-----------|-----------|-------|----|----|--------|----|----|----|----|----|----|----|----|
| 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  |
| xx<br>SETR | xx<br>CLRR | xx<br>SRR | xx<br>SRE | xxTOS | -  | -  | xxSRPN |    |    |    |    |    |    |    |    |

| Field  | Bit          | Type | Value                | Function   |
|--------|--------------|------|----------------------|--|
| xxSETR | xxSRC[15]    | w    | 0<br>1               | Source xx Request Set Bit.<br>No action.<br>Set xxSRR (no action if xxCLRR = 1).<br>Written value is not stored. Read returns 0.   |
| xxCLRR | xxSRC[14]    | w    | 0<br>1               | Source xx Request Clear Bit:<br>No action.<br>Clear xxSRR (no action if xxSETR = 1).<br>Written value is not stored. Read returns 0.   |
| xxSRR  | xxSRC[13]    | rh   | 0<br>1               | Source xx Service Request Flag:<br>No service request pending.<br>A service request is pending.  |
| xxSRE  | xxSRC[12]    | rw   | 0<br>1               | Source xx Service Request Enable Control.<br>Service request is disabled.<br>Service request is enabled.   |
| xxTOS  | xxSRC[11:10] | rw   | 00<br>01<br>10<br>11 | Source xx Type-of-Service Control.<br>Service Provider 0 (typically CPU).<br>Request Service Provider 1 (impl. spec.).<br>Request Service Provider 2 (impl. spec.).<br>Request Service Provider 3 (impl. spec.). |



| Field         | Bit        | Type | Value | Function  |
|---------------|------------|------|-------|---|
| <b>xxSRPN</b> | xxSRC[7:0] | rw   |       | Source xx Service Request Priority Number.            |
|               |            |      | 0x00  | A service request on this priority is never serviced. |
|               |            |      | 0x01  | Lowest priority number.                               |
|               |            |      | 0xFF  | Highest priority number.                              |
| -             |            | r    |       | Reserved.   |

### Request Flag SRR

This bit can be directly set or reset by the associated hardware (e.g. an associated trigger event in a peripheral sets this bit to 1; the acknowledgment of the service request by the Service Provider causes this bit to be cleared). Bit SRR can be set or reset by software via bits SETR or CLRR, respectively. Writing directly to SRR via software has no effect.

SRR can be set or cleared (either by hardware or by software) regardless of the state of the enable bit SRE. If SRE = 1, a pending service request takes part in the interrupt arbitration of the service provider selected via the bit field TOS. Bit SRR is automatically reset by hardware when the service request is acknowledged and serviced. If SRE = 0, a pending service request is excluded from interrupt arbitrations. Software can poll SRR to check for a pending service request. SRR must be reset by software in this case (write a '1' to CLRR).

### Request Set and Clear Bits (SETR, CLRR)

These bits are intended for a software set or clear of the actual service request bit SRR. Writing a '1' to SETR causes bit SRR to be set to '1'. Writing a '1' to CLRR causes bit SRR to be cleared to '0'. Possible hardware modifications of SRR that occurred during read-modify-write instructions (for example, bit set, bit clear instructions) are lost; the software modification has priority. The value written to SETR or CLRR is not stored. Writing a 0 to these bits has no effect. These bits always return 0 when read. If both, SETR and CLRR are written to '1' at the same time, SRR is not affected.

### Enable Bit (SRE)

The SRE bit controls whether an active interrupt request takes part (SRE = 1) in the hardware arbitration or not (SRE = 0). It does not affect the setting of bit SRR by hardware or software. This has the advantage that the interrupt request flag can be polled by software, even if it is not taking part in the hardware arbitration.

### Type-of-Service Control (TOS)

The interrupt system of the TriCore architecture is designed to manage up to four Service Providers for service requests from peripherals or other sources. The TOS bit field is used to select the service provider for a request, meaning whether the service request takes part in the interrupt arbitration of the selected service provider. The number of service providers for a given TriCore device is implementation-specific.

2001-04-30 @ 15:16

### Service Request Priority Number (SRPN)

The 8-bit Service Request Priority Number of a service request indicates its priority with respect to other sources requesting an interrupt to the same service provider, and to the priority of the service provider itself. Each SRPN used by active sources requesting the same service provider must be unique at a given time; no active sources can use the same SRPN at the same time (except for the default SRPN of 0x00, which excludes an SRN from taking part in the arbitration). That means, no two or more active sources, e.g. requesting CPU service, are allowed to use the same SRPN; they can, however, use the same SRPNs as sources which are requesting another service provider. The term 'active source' in this context means a source which has its request enable bit, SRE, set to one to allow the request to participate in interrupt arbitrations. If a source is not active, meaning its service request enable bit is cleared, no restrictions are applied to the service request priority number.

The SRPN also identifies the entry into the interrupt vector table (or similar structures depending on the nature of the service provider). Unlike other interrupt systems, the TriCore 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 peripherals, and a single peripheral can have multiple entry points for different purposes depending on its priority at a given time.

The range for the service request numbers used in a system depends on the number of active service requests and the user-defineable organization of the vector table. With the 8-bit SRPN, the TriCore interrupt arbitration scheme permits up to 255 sources to be active at one time. More information on the range of SRPNs can be found in the section on [Interrupt Priority Groups](#).

## 5.2 Interrupt Control Unit

The Interrupt Control Unit 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 service provider or not. The number of interrupt control units depends on the number of service providers implemented in a TriCore device. Each one of them controls its associated interrupt arbitration bus and manages the communication with its service provider.

In this document, only the interrupt control unit of the CPU is detailed. Please refer to the respective documentation of a specific TriCore device for information on possible further service providers and associated control units.

### 5.2.1 CPU Interrupt Control Unit, ICU

The ICU is closely coupled with the CPU and its Interrupt Control Register (ICR). This register, as well as the operation of the ICU, is described below.

#### 5.2.1.1 ICU Interrupt Control Register (ICR)

The ICR holds the current CPU priority number (CCPN), the global interrupt enable/disable bit (IE), the pending interrupt priority number PIPN, as well as implementation-specific bits to control the

interrupt arbitration cycles. A detailed description of the bits and bit fields in register ICR is given after the table and in **Operation of the Interrupt Control Unit**.

### ICR

#### Interrupt Control Register

**Reset Value : 0x0000 0000**

|    |    |    |    |    |             |                                 |      |      |    |    |    |    |    |    |    |
|----|----|----|----|----|-------------|---------------------------------|------|------|----|----|----|----|----|----|----|
| 31 | 30 | 29 | 28 | 27 | 26          | 25                              | 24   | 23   | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
| -  | -  | -  | -  | -  | impl. spec. | implemen-<br>tation<br>specific | PIPN |      |    |    |    |    |    |    |    |
| 15 | 14 | 13 | 12 | 11 | 10          | 9                               | 8    | 7    | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
| -  | -  | -  | -  | -  | -           | -                               | IE   | CCPN |    |    |    |    |    |    |    |

| Field                          | Bits       | Type | Value        | Function  |
|--------------------------------|------------|------|--------------|---|
| <b>implementation specific</b> | ICR[26]    |      |              | Implementation specific control of the arbitration. See documentation for specific TriCore product.             |
| <b>implementation specific</b> | ICR[25:24] |      |              | Implementation specific control of the arbitration. See documentation for specific TriCore product.             |
| <b>PIPN</b>                    | ICR[23:16] | rh   | 0x00<br>0xYY | Pending Interrupt Priority Number:<br>No valid pending request.<br>A request with priority YY is pending.       |
| <b>IE</b>                      | ICR[8]     | rwh  | 0<br>1       | Global Interrupt Enable Bit:<br>Interrupt system is globally disabled.<br>Interrupt system is globally enabled. |
| <b>CCPN</b>                    | ICR[7:0]   | rwh  |              | Current CPU Priority Number:  |
| -                              |            | r    |              | Reserved.   |

#### Current CPU Priority Number (CCPN)

This field indicates the current priority level of the CPU and is automatically updated by hardware on entry and exit of an interrupt service routine, and through the execution of a BISR instruction. It can also be updated through an MTCR instruction.

#### Global Interrupt Enable Bit (IE)

Setting bit IE globally enables the interrupt system. The acceptance of interrupts, however, depends on the individual Service Request Enable Bits in the SRNs and the current situation in the CPU operation.

2001-04-30 @ 15:16

This bit is automatically updated by hardware on entry and exit of an interrupt service routine, and through the execution of the instructions ENABLE, DISABLE and BISR. It can also be updated through an MTCR instruction.

### Pending Interrupt Priority Number (PIPN)

This read-only field is updated by the ICU at the end of an arbitration and indicates the priority number of a pending request. PIPN is set to 0x00 when no request is pending, and at the beginning of a new arbitration.

### Arbitration Control

Three bits in register ICR are intended for implementation specific control of the arbitration. Please see the respective documentation of a specific TriCore product for information on these bits.

#### 5.2.1.2 Operation of the Interrupt Control Unit

When an interrupt service is requested by one or more sources, these requests are serviced depending on their priority ranking. Thus, the interrupt system must determine which request has the highest priority each time. The TriCore interrupt system uses a scheme that performs the arbitration in parallel with the normal CPU operation. The interrupt control unit, ICU, controls this scheme, which takes place in one or more cycles over the interrupt arbitration bus. The detailed arbitration scheme is implementation specific.

The ICU automatically starts an arbitration when a new interrupt request is detected. At the end of the arbitration, the ICU has determined the service request with the highest priority number. It stores this number in the PIPN field of register ICR and generates an interrupt request to the CPU.

The CPU checks the state of the global interrupt enable bit, ICR.IE, and compares the current CPU priority number, CCPN, in register ICR against the PIPN. The CPU can be interrupted only if ICR.IE = 1 and PIPN is greater than CCPN. If this is true and the CPU can enter the service routine, it reads the PIPN to determine the vector entry and acknowledges the ICU, which in turn sends the acknowledge back to the pending interrupt request, the 'winner' of this arbitration round, to inform it that it will be serviced. This node then resets its service request flag, SRR.

After sending the acknowledge, the ICU sets PIPN to 0x00 and automatically starts a new arbitration to check whether there is another pending interrupt request. If this is the case, the priority number of this request is written to PIPN at the end of this arbitration. Otherwise, PIPN remains at 0x00 and the ICU enters an idle state, waiting for the next interrupt request.

The further interrupt service actions in the CPU are described in [Entering an Interrupt Service Routine](#).

Several conditions could block the CPU from immediately responding to the interrupt request generated by the ICU:

- The current CPU priority, CCPN, is equal to or higher than the pending interrupt priority, PIPN
- The interrupt system is globally disabled (ICR.IE = 0)
- The CPU is in the process of entering an interrupt or trap service routine
- The CPU is operating on non-interruptible trap services
- The CPU executes a multi-cycle instruction

- The CPU is executing an instruction which modifies the conditions of the global interrupt system, such as modifying the ICR

The CPU will respond to the interrupt request when these conditions are no longer true.

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 can, for example, be used for software polling techniques to determine high priority requests while having the interrupt system globally disabled.

If a new service request is generated by an SRN while an arbitration is in progress, this request has to wait at least until the end of this arbitration.

### 5.2.1.3 Arbitration Scheme

The arbitration scheme is implementation specific and is detailed in the respective documentation of a specific TriCore product.

## 5.3 Entering an Interrupt Service Routine

When all conditions are clear for the CPU to service an interrupt request, the following actions are performed to enter an Interrupt Service Routine (ISR):

1. The upper context of the current task is saved.
2. The interrupt system is globally disabled (ICR.IE = 0).
3. The current CPU priority number (CCPN) is set to PIPN.
4. The PSW is set to a default value:
  - All permissions are enabled: PSW.IO = 0b10
  - Memory protection is switched to PRS 0: PSW.PRS = 0b00.
  - The stack pointer bit is set for using the interrupt stack: PSW.IS = 1.
  - The call depth counter is cleared, the call depth limit is set for 64: PSW.CDC = 0b1000000.
5. The interrupt vector table is accessed to fetch the first instruction of the interrupt service routine (ISR). The effective address is the contents of the BIV register, ORed with the PIPN number left-shifted by 5

As listed above, an interrupt service routine is entered with the interrupt system globally disabled and the current CPU priority CCPN set to the priority PIPN of the interrupt being serviced. It is up to the user to enable the interrupt system again and optionally modify the priority number CCPN to implement interrupt priority levels or handle special cases (see next sections).

To simply enable the interrupt system again, the ENABLE instruction can be used, which sets ICR.IE to one. The BISR instruction offers a convenient way to enable the interrupt system, to set the CCPN to a new value, and to save the lower context of the interrupted task. It is also possible to use an MTCR instruction to modify ICR.IE and ICR.CCPN.

The instructions ENABLE, BISR, MTCR and DISABLE (disable interrupts) are all executed such that the CPU is blocked from taking interrupt requests until the instruction is completely finished. This avoids pipeline side effects and eliminates the need for an ISYNC instruction (synchronize instruction stream) following these instructions.

2001-04-30 @ 15:16

### 5.4 The Interrupt Vector Table

The interrupt vector table is organized according to the priority number of the interrupts. The priority number PIPN of the interrupt now being serviced by the CPU identifies the entry into the vector table. This has some important advantages over traditional interrupt systems, where the vector entry is usually determined directly through the interrupt source.

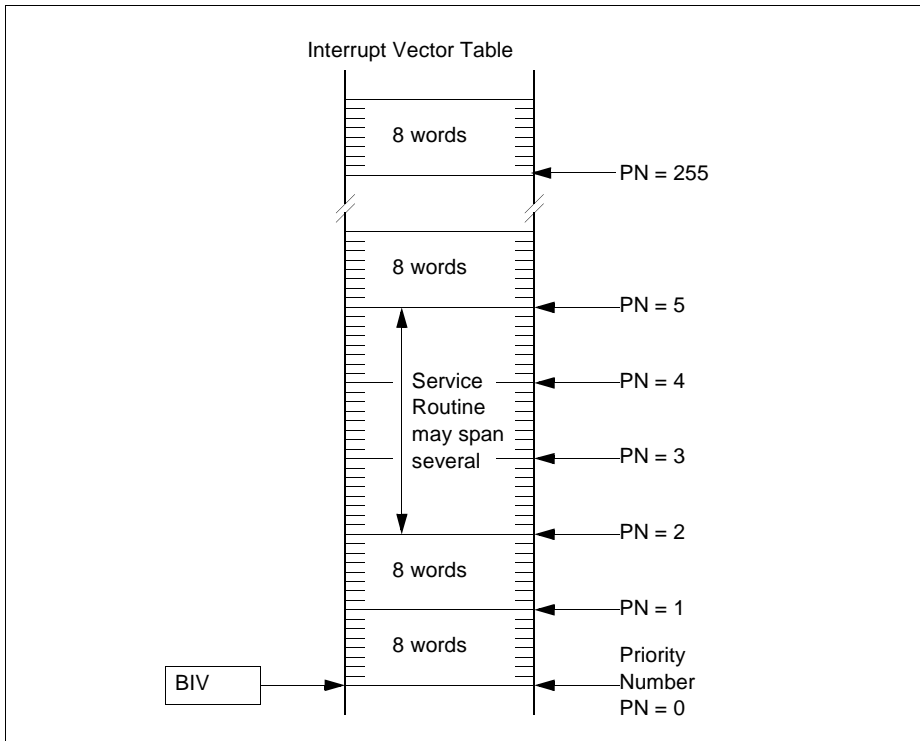
The interrupt handler vectors are stored in code memory. The BIV register specifies the base address of the interrupt vector table. When the CPU takes an interrupt, the interrupt priority number SRPN associated with the interrupt is used to index into the interrupt vector code. This number, detected by the ICU as PIPN and then taken as the new CCPN, is left-shifted by five bits and ORed 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 (due to the simple ORing). Alignment to an 8-KByte boundary is sufficient for the full range of priority numbers. If fewer numbers are used, the alignment requirements can be relaxed accordingly.

Left-shifting the CCPN by 5 bits creates entries into the vector table which are evenly spaced by 8 words. If an interrupt handler is very short, it may fit entirely within the eight words available in the vector code segment. Otherwise, the code stored at the entry location can either span several vector entries (see next section) or should contain some initial instructions, followed by a jump to the rest of the handler.

**Figure 5-2** gives an overview of the interrupt vector table.

The size of the vector table depends only on the range of priority numbers actually used in a system. Up to 256 vector entries for 255 (vector entry 0 is never used) distinct interrupt handlers are supported, but systems requiring fewer entries need not dedicate the full memory area required by the largest configurations.



**Figure 5-2**  
**Interrupt Vector Table**

The BIV register allows the interrupt vector table to be located anywhere in the available code memory. Its default on power-up is fixed to 0x0000.0000, however, the BIV register can be written to using the MTCR instruction during the initialization phase of the system, before interrupts are enabled. It is also possible to have multiple interrupt vector tables and switch between them simply by modifying the contents of the BIV register.

## 5.5 Usage of the TriCore Interrupt System

The following sections describe some examples how the flexible interrupt system of the TriCore architecture can be used to solve typical as well as special application requirements.

### Spanning ISRs across Vector Entries

Since the vector entries are not tied to the interrupt source, it is easy to span interrupt service routines across vector entry locations, as shown in [Figure 5-2](#). This eliminates the need of a jump to the rest of the interrupt handler if it would not fit into the available 8 words between entry locations.

2001-04-30 @ 15:16

Note that priority numbers, which relate to entries which are occupied by such a spanned service routine must not be used for any of the active service request nodes which request service from the same service provider. In the example shown in [Figure 5-2](#), vector locations 3 and 4 are covered through the service routine for entry 2. Thus, these numbers must not be assigned to SRNs requesting CPU service. They can, however, be used to request PCP service.

The next available vector entry is now entry 5. It must be noted that this technique increases the range of priority numbers required in a given system. The size of the vector table has to be adjusted accordingly.

### Interrupt Priority Groups

Interrupt priority groups, meaning a set of interrupts which can not interrupt each other's service routine, can easily be achieved with the TriCore interrupt system architecture.

When the CPU starts the service of an interrupt, the interrupt system is globally disabled and the CPU priority CCPN is set to the priority of the interrupt now serviced. This blocks all further interrupts from being serviced until the interrupt system is either enabled again through software or this service routine is terminated with the RFE instruction (which automatically reinstalls the previous state of the ICR.IE bit, which was a one; otherwise this interrupt would not have been serviced).

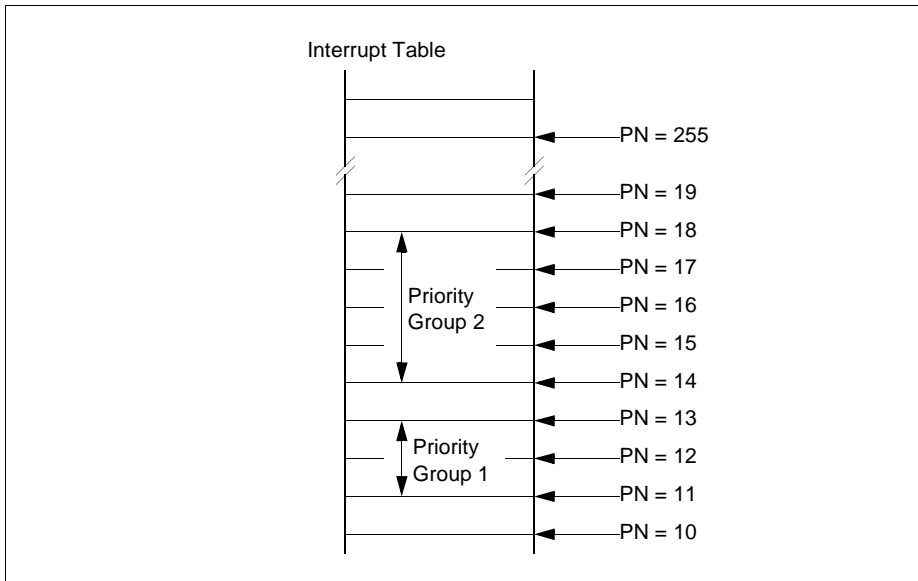
When the software of the interrupt service routine (ISR) enables the interrupt system again by setting ICR.IE without changing the CCPN, the effect is that all interrupt requests with the same or lower priority than the CCPN are still blocked from being serviced. This includes a reoccurrence of the current interrupt; it can not interrupt this service.

This ISR will, however, be interrupted by each request which has a higher priority number than the CCPN. This is not wanted in many cases; application requirements are often such that interrupt requests of similar significance being grouped together in a way that no request in that group can interrupt the ISR of another member of the same group.

This can easily be accomplished with the TriCore interrupt system. For a defined group of interrupt requests, the software of their respective service routines sets the CCPN to the number of the highest SRPN used in that group before enabling the interrupt system again. [Figure 5-3](#) shows an example for this. The interrupt requests with the priority numbers 11 and 12 form one group, while the requests with priority numbers 14 through 17 form another group. Every time one of the interrupts from group 1 is serviced, the service routine sets the CCPN to 12, the highest number in that group, before reenabling the interrupt system. Every time one of the interrupts from group 2 is serviced, the service routine sets the CCPN to 17 before reenabling the interrupt system. If interrupt 14 is serviced, for example, it can only be interrupted by requests with a priority number higher than 17, but not through a request from its own priority group or requests with lower priority.

One can easily see the flexibility of this system, which is also superior to systems with fixed priority levels. In the example shown, the interrupt request with priority number 13 forms its own single member 'group'. Setting the CCPN to the maximum number, 255, in each service routine has the same effect as not enabling the interrupt system again; all interrupt requests can be considered to be in one group. The flexibility for interrupt priority levels spans from all interrupts in one group to each interrupt request building its own group, and all possible combinations in between.





**Figure 5-3**  
**Interrupt Priority Groups**

### Splitting ISRs onto different priorities

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, but further operations in that service routine can run on a lower priority. For this, the service routine is divided into two parts, one containing the critical actions, the other part the less critical ones. First, the priority of the interrupt node is set to the high priority. When the interrupt occurs, 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 via software (indicating a pending interrupt) while still in the service routine. Returning to the interrupted program terminates the high priority service routine. The pending interrupt will now be serviced when the CPU priority is lower than its own one. After entering the service routine, which now is at a different address in the program memory, the outstanding but low-priority actions of the interrupt can be performed.

In other cases, 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 lets the priority level of this service request to be raised within the ISR, and also lets interrupts be completely disabled.

2001-04-30 @ 15:16

### Using different Priorities for the same Interrupt Source

For some applications, the priority of an interrupt request in relation to other requests is not fixed, but depends on the current situation in the system. This can be done simply by assigning different priority numbers, SRPNs, at different times to an interrupt source depending on the application needs. Usually, the ISR for that interrupt will execute different code depending on its priority. In traditional interrupt systems, the ISR has to check the current priority of that interrupt request and perform a branch to the appropriate code section, causing a delay in the response to the request. In the TriCore system, however, the interrupt will automatically have different vector entries for the different priorities; an extra check and branch in the ISR is not necessary, therefore the interrupt latency is reduced.

In case the ISR is independent of the interrupt's priority, one has to place branches to the common ISR code on each of the vector entries for that interrupt.

Note that these different priority numbers for one interrupt have to be regarded when creating the vector table.

### Software-Posted Interrupts

A software-posted interrupt is a true hardware interrupt, carrying an interrupt priority that is processed through the regular interrupt subsystem 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 control register is set, there is no way to distinguish between a software-posted interrupt request and a 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, that is, for example, interrupts which are triggered by a peripheral module.

However, how many of the hardware SRNs are available in a given system for such purposes is depending on the application needs. Thus, an RTOS can not rely on a certain number of 'free' SRNs for software-posting of interrupts.

To support the usage of software-posted interrupts, mainly for RTOS code, the TriCore architecture provides a number of service request nodes which are intended solely for the purpose of software-posting. They are not connected to any peripheral or other module on the chip; the service request flag can only be set by software. In this way, it is guaranteed for the RTOS and user code, that there are SRNs available which are not used by hardware modules.

An implementation may contain four CPU Service Request Nodes. See [CPU Service Request Nodes](#).

### Interrupt 1

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 done from the ISR directly, but from RTOS code in a service function called from the ISR). The ISR can then 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, this interrupt will then be serviced. When a return to the background task level is performed, the software-posted interrupt at priority level one will automatically be recognized and serviced.

## 5.6 CPU Service Request Nodes

To support software-posting of interrupts for RTOS code, the TriCore architecture defines four service request nodes which are not attached to a peripheral or other module on the chip. The interrupt request bit can only be set by software. These SRNs are called the CPU service request nodes, however, setting of the interrupt request can also be done through, for example, an external bus master.

The service request control registers of the four CPU SRNs are described below.

### CPUSRC0

#### CPU Service Request Control Register 0

Reset Value: 0000 0000<sub>H</sub>

| 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  |
| CPU SETR 0 | CPU CLRR 0 | CPU SRR0 | CPU SRE0 | CPUTOS0 | -  | -  | CPUSRPN0 |    |    |    |    |    |    |    |    |

### CPUSRC1

#### CPU Service Request Control Register 1

Reset Value: 0000 0000<sub>H</sub>

| 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  |
| CPU SETR 1 | CPU CLRR 1 | CPU SRR1 | CPU SRE1 | CPUTOS1 | -  | -  | CPUSRPN1 |    |    |    |    |    |    |    |    |

2001-04-30 @ 15:16

### CPUSRC2

#### CPU Service Request Control Register 2

Reset Value: 0000 0000<sub>H</sub>

| 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  |
| CPU<br>SETR<br>2 | CPU<br>CLRR<br>2 | CPU<br>SRR2 | CPU<br>SRE2 | CPUTOS2 |    | -  | -  | CPUSRPN2 |    |    |    |    |    |    |    |

### CPUSRC3

#### CPU Service Request Control Register 3

Reset Value: 0000 0000<sub>H</sub>

| 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  |
| CPU<br>SETR<br>3 | CPU<br>CLRR<br>3 | CPU<br>SRR3 | CPU<br>SRE3 | CPUTOS3 |    | -  | -  | CPUSRPN3 |    |    |    |    |    |    |    |

---

# Traps

---

2001-04-30 @ 15:16

## 6 Traps

A trap occurs as a result of an event such as a non-maskable interrupt, an instruction exception, memory-management 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 TriCore architecture specifies eight general classes for traps. Each class has its own trap handler, accessed through a trap vector of 32 bytes per entry, indexed by the hardware-defined trap class number.

Within each class, specific traps are distinguished by a Trap Identification Number (TIN) that is loaded by hardware into register D15 before the first instruction of the trap handler is executed. The trap handler must test and branch on the value in D15 to reach the subhandler for a specific TIN.

Traps can be further classified as synchronous or asynchronous, and as hardware or software generated. These are explained further below.

Table 6-1 lists the trap classes specified by the TriCore architecture, and summarizes and classifies the pre-defined set of specific traps within each class.

Table 6-1 summarizes and classifies all TriCore-supported traps.

**Table 6-1**  
**Supported Traps**

| Trap ID #<br>(TIN)                         | Trap<br>Name | Sync/<br>Async | Hardware/<br>Software | Description   |
|--|--------------|----------------|-----------------------|---|
| <b>Class 0 — MMU</b>                       |              |                |                       |   |
| 0  | VAF          | Synchronous    | Hardware              | Virtual Address Fill<br>See <a href="#">Section 7.7, "MMU traps."</a>       |
| 1  | VAP          | Synchronous    | Hardware              | Virtual Address Protection<br>See <a href="#">Section 7.7, "MMU traps."</a> |
| <b>Class 1 — Internal Protection Traps</b> |              |                |                       |   |
| 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  |
| <b>Class 2 — Instruction Errors</b>        |              |                |                       |   |
| 1  | IOPC         | Synchronous    | Hardware              | Illegal Opcode  |

**Table 6-1**  
**Supported Traps (cont'd)**

| Trap ID #<br>(TIN) | Trap<br>Name | Sync/<br>Async | Hardware/<br>Software | Description                   |
|--------------------|--------------|----------------|-----------------------|-------------------------------|
| 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 | Call stack 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 |

**Class 4 — System Bus and Peripheral Errors**

|   |     |              |          |                         |
|---|-----|--------------|----------|-------------------------|
| 1 | PSE | Synchronous  | Hardware | Program fetch bus error |
| 2 | DSE | Synchronous  | Hardware | Data access bus error   |
| 3 | DAE | Asynchronous | Hardware | Data access bus error   |

**Class 5— Assertion Traps**

|   |      |             |          |                            |
|---|------|-------------|----------|----------------------------|
| 1 | OVF  | Synchronous | Software | Arithmetic overflow        |
| 2 | SOVF | Synchronous | Software | Sticky arithmetic overflow |

**Class 6 — System Call<sup>1)</sup>**

|  |     |             |          |             |
|--|-----|-------------|----------|-------------|
|  | SYS | Synchronous | Software | System call |
|--|-----|-------------|----------|-------------|

**Class 7 — Non-Maskable Interrupt**

|   |     |              |          |                        |
|---|-----|--------------|----------|------------------------|
| 0 | NMI | Asynchronous | Hardware | Non-maskable interrupt |
|---|-----|--------------|----------|------------------------|

2001-04-30 @ 15:16

- 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.

There is a some degree of implementation dependency in the actual traps that an implementation may generate. For example, trap class 0 is reserved for MMU traps; in implementations that do not include an MMU, no traps in this class will be generated. Such an implementation might also generate UOPC traps for the MMU instructions. In addition, an implementation could conceivably add new TINs to one of the trap classes, if it made sense to do so for the particular hardware and system configuration it supported.

Consult the User's Guide for a specific implementation for any changes in the traps supported, relative to those shown in **Table 6-1, "Supported Traps,"**.

### 6.1.1 Synchronous Traps

Synchronous traps are associated with the execution or attempted execution of specific instructions, or with an attempt to access a virtual address that requires the intervention of the memory-management system. 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, cannot be masked, and do not change the current CPU interrupt priority number. The return PC for an asynchronous trap, like the return PC for an interrupt handler, is the address of the next instruction that would have been executed when the trap was taken.

### 6.1.3 Hardware Traps

Hardware traps are generated in response to exception conditions detected by the hardware. In most, but not all cases, the exception conditions are associated with the attempted execution of a particular instruction. Examples are the illegal instruction trap, memory protection traps, and data memory misalignment traps.

In the case of the MMU traps (trap class 0), the exception condition is either the failure to find a TLB entry for the virtual page referenced by an instruction (VAF trap), or an access violation for that page (VAP trap). See **Section 7.7, "MMU traps,"** for more information.



#### 6.1.4 Software Traps

Software traps are generated as an intentional result of executing a system call or an assertion instruction. The assertion instructions supported by the architecture are TRAPV and TRAPSV (trap on overflow, and trap on sticky overflow). System calls are generated by the SYSCALL instruction.

System call traps are described further in section 6.3.7 below.

### 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. The return PC saved in the return address register for a synchronous trap is the PC of the instruction that caused the trap. For an asynchronous trap, the return PC is that of the instruction that would have been executed next, if the asynchronous trap 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 8 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 TIN which is loaded into D15. The trap class number is left shifted by 5 bits and ORed with the address in the BTV register to generate the entry address of the trap handler.

2001-04-30 @ 15:16

### 6.2.3 Initial State upon a Trap

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

- All permissions are enabled.
- Memory protection using the interrupt memory protection map (PSW.PRS = 00<sub>2</sub>) is enabled.
- The stack pointer bit is set for using the interrupt stack.
- The call depth counter is cleared, and the call depth limit selector is set for 64.
- Interrupts are disabled; they remain disabled until explicitly enabled.
- 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.

## 6.3 Trap Descriptions

What follows are descriptions for all of the trap classes and specific traps listed in [Table 6-1](#), "Supported Traps,".

### 6.3.1 MMU Traps

Trap class 0 (zero) is reserved for MMU traps, for those implementations that include an MMU. There are two traps within this class:

#### 6.3.1.1 VAF Virtual Address Fill trap (TIN 0)

This trap is generated when the MMU is enabled, and the virtual address referenced by an instruction does not have a page entry in the MMU's Translation Lookaside Buffer (TLB). See "MMU Traps" in the Memory Management section for more detail.

#### 6.3.1.2 VAP Virtual Address Protection trap (TIN 1)

This trap is generated when the access permissions associated with a referenced page do not permit the type of access attempted. See "MMU Traps" in the Memory Management section for more detail.

### 6.3.2 Internal Protection Traps

Trap class 1 is for traps related to TriCore's internal protection system. The memory protection traps in this class are for the range-based protection system, and are independent of the page-based VAP protection trap of trap class zero. See section the "Protection System" section for more detail.

The following internal protection traps are defined:

#### 6.3.2.1 PRIV Privilege Violation (TIN 1)

A program executing in one of the user modes (User-0 or User-1) attempted to execute an instruction not allowed by its privilege level.

There are only two instructions that, in all implementations, are allowed only in supervisor mode: MTCR and BISR. In addition, `ENABLE` and `DISABLE` are prohibited in User-0 mode. For implementations that support the MMU, the MMU instructions `TLBMAP`, `TLBDEMAP`, `TLBFLUSH`, and `TLBPROBE`, are also privileged instructions, executable only in supervisor mode.

### 6.3.2.2 MPR Memory Protection, Read (TIN 2)

The MPR trap is generated when the memory protection system is enabled, and the effective address of a load, LDMST, or SWAP instruction does not lie within any range with read permissions enabled.

### 6.3.2.3 MPW Memory Protection, Write (TIN 3)

The MPW trap is generated when the memory protection system is enabled, and the effective address of a store, LDMST, or SWAP instruction does not lie within any range with write permissions enabled.

### 6.3.2.4 MPX Memory Protection, Execute (TIN 4)

The MPX trap is generated when the memory protection system is enabled, and the PC does not lie within any range with execute permissions enabled.

### 6.3.2.5 MPP Memory Protection, Peripheral access (TIN 5)

A program executing in User-0 mode attempted a load or store access to memory address segment 14 or 15.

### 6.3.2.6 MPN Memory Protection, Null address (TIN 6)

The MPN trap is generated whenever any program attempts a load / store operation to effective address zero.

### 6.3.2.7 GRWP Global Register Write Protection (TIN 7)

A program attempted to modify one of the global address registers (A0, A1, A8, or A9) when it did not have permission to do so.

## 6.3.3 Instruction Errors

Trap class 2 is for signalling instruction errors of various types. Instruction errors include errors in the instruction opcode, in the instruction operand encodings, or, for memory accesses, in the operand address. Specifically:

### 6.3.3.1 IOPC Illegal Opcode (TIN 1)

An invalid instruction opcode was encountered. An invalid opcode is one that does not correspond to any instruction known to the architecture.

2001-04-30 @ 15:16

### 6.3.3.2 UOPC Unimplemented Opcode (TIN 2)

An unimplemented opcode was encountered. An unimplemented opcode corresponds to a known instruction that is not supported in a given hardware implementation. The instruction may be implemented via software emulation in the trap handler.

### 6.3.3.3 OPD Invalid Operand specification (TIN 3)

The OPD trap may be raised for instructions that take an even-odd register pair as an operand, if the operand specifier is odd.

Implementations are not architecturally required to raise this trap; they are allowed the option of ignoring the low order bit of the operand specification when an even-odd register pair is expected.

### 6.3.3.4 ALN Alignment error (TIN 4)

Raised when the address for a data memory operation does not conform to the expected alignment constraints.

### 6.3.3.5 MEM Invalid Local Memory Address (TIN 5)

A program accessed an address in a range that the implementation recognizes as invalid-i.e., there is no memory at the referenced address.

This trap can only be raised if the memory protection system is disabled, or if protection ranges have been created that span invalid memory locations. Otherwise, the access will generate an internal protection trap, which preempts the MEM trap.

The MEM trap is synchronous, and is only generated when the memory system is able to recognize an address as invalid in time to generate a synchronous trap. An implementation is not architecturally required to recognize all invalid memory references in time to generate a synchronous trap. References that do not generate protection violations and are not recognized as invalid local memory references may still be invalid; if so, they will eventually raise an asynchronous DAE trap.

## 6.3.4 Context Management

Trap class 3 is for exception conditions detected by the context management subsystem, in the course of performing-or attempting to perform-context save and restore operations connected to function calls, interrupts, traps, and returns.

### 6.3.4.1 FCD Free Context list Depletion (TIN 1)

The FCD trap is generated after a context save operation, when the operation causes the free context list to become "almost empty". The "almost empty" condition is signaled when the CSA used for the save operation is the one pointed to by the context list limit register, LCX. The operation responsible for the context save completes normally, and then the FCD trap is taken.

If the operation responsible for the context save was the hardware interrupt or trap entry sequence, then the FCD trap handler will be entered before the first instruction of the original interrupt or trap handler is executed. The return PC for the FCD trap will point to the first instruction of the interrupt or trap handler.

The FCD trap handler is normally expected to take some form of action to rectify the context list depletion. The nature of that action is OS dependent, but the general choices are to allocate additional memory for CSA storage, or to terminate one or more tasks, and return the CSAs on their call chains to the free list. A third, more elaborate possibility, is not to terminate any tasks outright, but to copy the call chains for one or more inactive tasks to uncached external or secondary memory that would not be directly usable for CSA storage, and release the copied CSAs to the free list. The OS task scheduler, in that case, would need to recognize that the inactive task's call chain was not resident in CSA storage, and restore it, before dispatching the task.

Taking the FCD trap, in itself, uses one additional CSA beyond the one designated by the LCX register, so LCX must not point to the actual last entry on the free context list. In addition, it is possible that an asynchronous trap condition, such as an external bus error, will be reported after the FCD trap has been taken, interrupting the FCD trap handler, and using one more CSA. Hence, to avoid the possibility of a context list underflow, the free context list must include a minimum of two CSAs, beyond the one pointed to by the LCX register. If the FCD trap handler makes any calls, then additional CSA reserves are needed.

In order to allow the trap handlers for asynchronous traps to recognize when they have interrupted the FCD trap handler, there is a flag that is set in the system configuration register whenever an FCD trap is generated. (See the "SYSCON Register" description.) That flag, the FCDSF bit, should be tested by the handler for any asynchronous trap that could be taken while an FCD trap is being handled. If the bit is found to be set, the asynchronous trap handler must avoid making any calls, but should queue itself in some manner that allows the OS to recognize that the trap occurred. It should then do an immediate return, back to the interrupted FCD trap handler.

### 6.3.4.2 CDO Call Depth Overflow (TIN 2)

A program attempted to make a call, while executing with call depth counting enabled, and the call depth counter was already at its maximum value. (See the section "Program Status Word"). Call depth counting guards against context list depletion, by enabling the OS to detect "runaway recursion" in executing tasks.

### 6.3.4.3 CDU Call Depth Underflow (TIN 3)

A program attempted to execute a RET instruction, while call depth counting was enabled, and the call depth counter was zero.

A call depth underflow does not necessarily reflect a software error in the currently executing task. An OS can achieve finer granularity in call depth counting by using a deliberately narrow call depth counter, and incrementing or decrementing a separate software counter for the current task on each call depth overflow or underflow trap. A program error would be indicated only if the software counter were already zero when the CDU trap occurred.

### 6.3.4.4 FCU Free Context list Underflow (TIN 4)

The FCU trap is taken when a context save operation is attempted, but the free context list is found to be empty (FCX register contents null). It is also taken if any error is encountered during a context save operation, which presumably indicates a corrupted free list. The context save cannot be completed. Instead, a forced jump is made to the FCU trap handler.

2001-04-30 @ 15:16

In failing to complete the context save, architectural state is lost, so the occurrence of an FCU trap is a non-recoverable system error. The FCU trap handler should ultimately initiate a system reset.

### **6.3.4.5 CSU Call Stack Underflow (TIN 5)**

A program attempted to execute a RET instruction, or an interrupt or trap handler attempted to execute a RFE, when the contents of the PCX register were null, or otherwise invalid.

This trap indicates a system software error (kernel or OS) in task setup or context switching among software managed tasks. No software error or combination of errors in a user task can generate this condition, unless the task has been allowed write permission to the context save areas-which, in itself, can be regarded as a system software error.

### **6.3.4.6 CTYP Context Type error (TIN 6)**

This trap is raised when a context restore operation is attempted, but the context type, as indicated by the PCXI\_UL bit, is incorrect for the type of restore attempted. I.e., a restore lower context is attempted when PCXI\_UL == 1, or a restore upper context is attempted when PCXI\_UL == 0. As with the CSU trap, indicates a system software error in context list management.

### **6.3.4.7 NEST Nesting error (TIN 7)**

An RFE instruction was attempted when the call depth counter was non-zero. The return from an interrupt or trap handler should normally occur within the body of the interrupt or trap handler itself, or in code to which the handler has branched, rather than code called from the handler. Otherwise, there will be one or more saved contexts on the residual call chain that must be popped and returned to the free list, before the RFE can be legitimately issued.

## **6.3.5 System Bus and Peripheral Errors**

### **6.3.5.1 PSE Program fetch, Synchronous Error (TIN 1)**

The PSE trap is raised when a code fetch request results in an error other than those causing VAP or MPX trap.

### **6.3.5.2 DSE Data access, Synchronous Error (TIN 2)**

The DSE trap is raised when a data access (load or store) results in a synchronous error other than those causing a VAP, MPR, or MPW trap.

### **6.3.5.3 DAE Data access, Asynchronous Error (TIN 3)**

The DAE trap is raised when a the memory system reports back an error of some sort, and the error cannot immediately be linked to a currently executing instruction. Generally, this means an error returned on the system bus from a peripheral or external memory.

There is normally an implementation-dependent register that can be interrogated, to determine more precisely the source of the error. Refer to the User's Manual for the implementation in question for details.

### 6.3.6 Assertion Traps

#### 6.3.6.1 OVF Arithmetic Overflow (TIN 1)

Raised by the `TRAPV` instruction, if the overflow bit in the PSW (`PSW.V`) is set.

#### 6.3.6.2 SOVF Sticky Arithmetic Overflow (TIN 2)

Raised by the `TRAPSV` instruction, if the sticky overflow bit in the PSW (`PSW.SV`) is set.

### 6.3.7 6.3.7 System Call

#### 6.3.7.1 SYS System Call

The `SYS` trap is raised immediately after the execution of the `SYSCALL` instruction, to initiate a system call. The TIN that will be loaded into D15 when the trap is taken is not fixed, but is specified as an 8-bit unsigned immediate constant in the `SYSCALL` instruction. The return PC will point to the instruction immediately following the `SYSCALL`.

### 6.3.8 6.3.8 Non-Maskable Interrupt

#### 6.3.8.1 NMI Non-Maskable Interrupt (TIN 0)

The causes for raising a non-maskable interrupt are implementation dependent. Typically, there will be an external pin that can be used to signal the NMI, but it may also be raised in response to such things as a watchdog timer interrupt, or an impending power failure. Refer to the User's Manual for the implementation in question for details.

2001-04-30 @ 15:16



---

# Memory Management

---

2001-04-30 @ 15:16

## 7 Memory Management

This chapter provides the description of the memory management architecture of TriCore.

The principal features of the TriCore memory management include:

- 4-GBYTE virtual address space divided into sixteen 256 MB segments
- 4-GBYTE physical address space divided into sixteen 256 MB segments
- Addressing by direct translation or via Page Table Entries (PTE)
- Two addressing modes: physical and virtual (physical page attributes override virtual page attributes)

The virtual address space is divided into 16 segments of 256 MB each. The physical address space is also divided into 16 segments of 256 MB each. Virtual addresses are always translated into physical addresses before accessing memory.

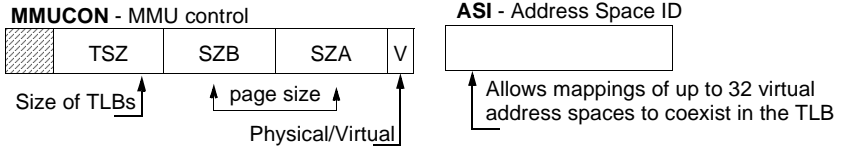
The virtual address is translated into a physical address using either direct translation or Page Table Entry (PTE) translation.

- Direct translation: If the virtual address belongs to the upper half of the virtual address space then the virtual address is directly used as the physical address. If the virtual address belongs to the lower half of the address space, then the virtual address is used directly as the physical address if the processor is operating in Physical mode.
- PTE translation: If the virtual address belongs to the lower half of the address space, then the virtual address is translated using a Page Table Entry if the processor is operating in Virtual mode.

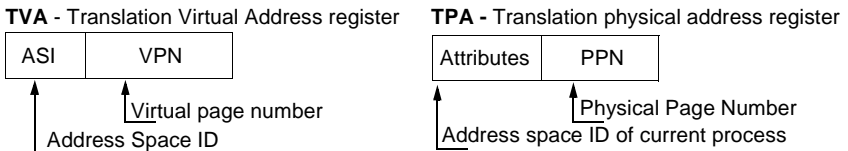
PTE translation is performed by replacing the Virtual Page Number (VPN) of the virtual address by a Physical Page Number (PPN) to obtain a physical address. Six memory-mapped MMU Core Special Function Registers (CSFRs) control the memory management system.

**Figure 7-1** shows the MMU registers and retained state (data structures). These elements are discussed at length below.

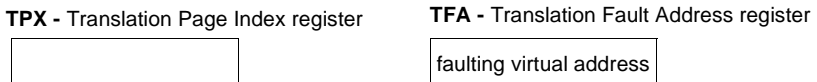
### Read/Writable Registers (via MTCR and MFCR)



### Readable Registers (filled via TLBPROBE instruction)



### Other Registers (filled via TLBPROBE instruction, or faulting virtual address)



### Translation Look-aside Buffers

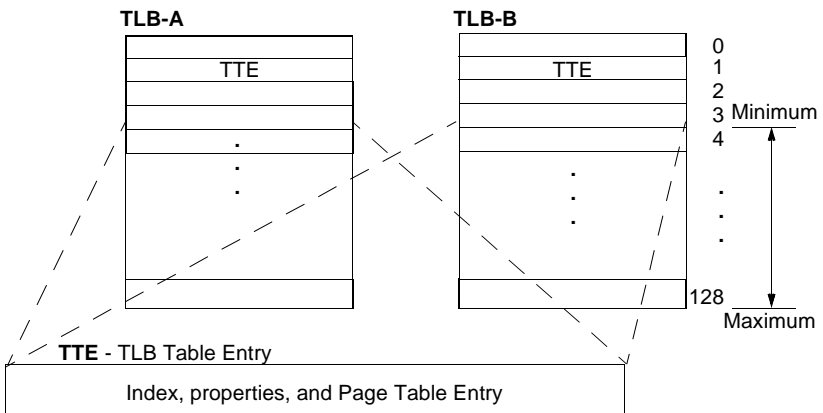


Figure 7-1 MMU Registers and Data Structures

## 7.1 Address Spaces

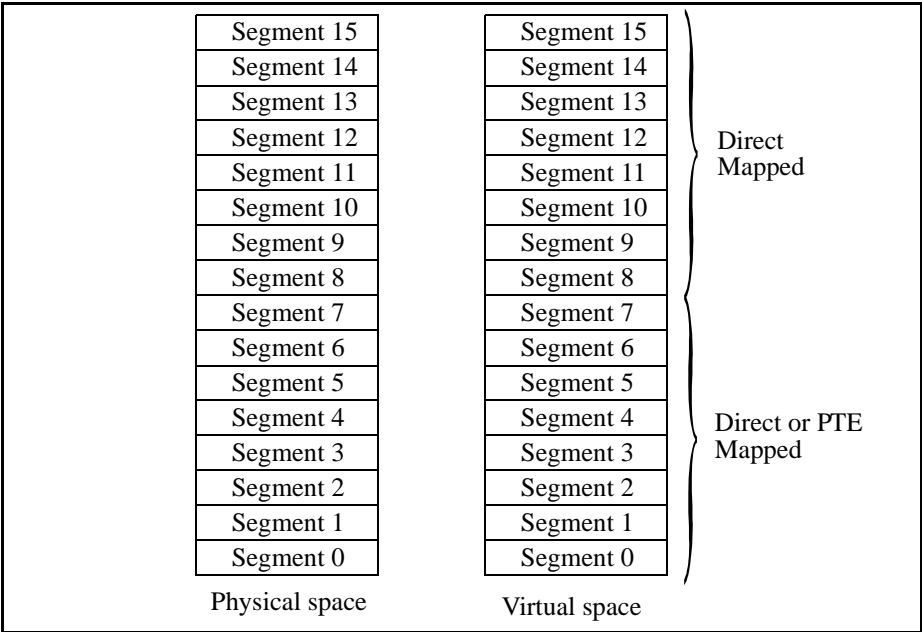
The TriCore virtual address space is 4 GB in size. The virtual address space is divided into 16 segments with each segment being 256 MB. The upper 4 bits of the 32-bit virtual address are used

2001-04-30 @ 15:16

to identify the segment. Virtual segments are numbered 0–15. A virtual address is always translated into a physical address before accessing memory.

The physical address space is 4 GB in size. The physical address space is also divided into 16 segments with each segment being 256 MB. The upper 4 bits of the 32-bit physical address are used to identify the segment. Physical segments are numbered 0–15.

The physical and virtual address space maps are shown in **Figure 7-2**.



**Figure 7-2 Physical and virtual address spaces**

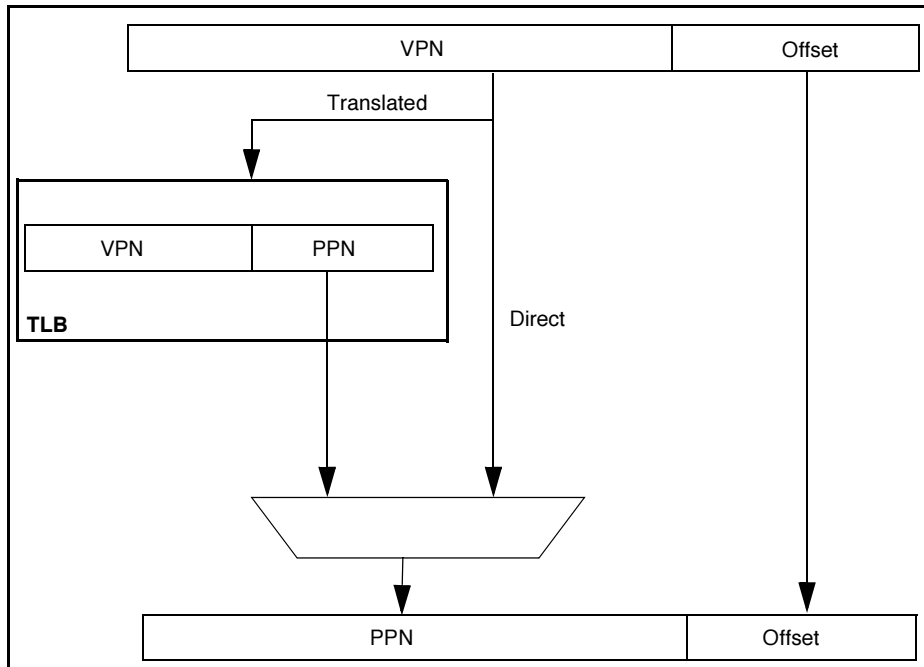
A 32-bit virtual address is comprised of a Virtual Page Number (VPN) concatenated with a Page Offset. A 32-bit physical address is comprised of a Physical Page Number (PPN) concatenated with a Page Offset.

## 7.2 Address translation

The virtual address is translated into a physical address using either direct translation or Page Table Entry (PTE) translation as shown in **Figure 7-3**. If the virtual address belongs to the upper half of the virtual address space then the virtual address is used directly as the physical address (direct translation). If the virtual address belongs to the lower half of the address space, then the virtual address is used directly as the physical address if the processor is operating in Physical mode (direct translation, or no MMU is present in the core) or translated using a Page Table Entry if the processor is operating in Virtual mode (PTE translation). The MMUCON.V bit controls the Physical/Virtual operating mode of the processor as outlined in the section on the MMUCON register.

2001-04-30 @ 15:16

Translation using the PTE is done by replacing the Virtual Page Number (VPN) of the virtual address by a Physical Page Number (PPN) to obtain a physical address.



**Figure 7-3 Virtual address translation**

### 7.2.1 Address translation for context pointers

The context pointers (PCX, FCX, LCX) are constrained to use direct translation. See [Context Management Registers](#).

## 7.3 Translation Lookaside Buffers

The MMU provides PTE-based virtual address translation through two Translation Lookaside Buffers (TLBs) which are referred to as TLB-A and TLB-B. The MMU supports up to four page sizes which include 1 KB, 4 KB, 16 KB, and 64 KB page sizes. However, at any given time, each TLB provides translations for only one particular page size. The page size setting of each TLB is determined through the MMUCON.SZA and MMUCON.SZB fields as outlined in [MMU Configuration register \(MMUCON\)](#).

Each TLB contains a number  $N$  of TLB Table Entries (TTEs) where  $N$  is a minimum of 4 and a maximum of 128. The MMUCON.TSZ field determines the size of each TLB as outlined in the section on the MMUCON register. Each TTE has an 8-bit index associated with it. Index numbers 0, ..., MMUCON.TSZ are used for the entries in TLB-A while index numbers 128, ...,

2001-04-30 @ 15:16

128+MMUCON.TSZ are used for the entries in TLB-B. Each TTE contains a Page Table Entry (PTE). The organization of each TLB is implementation-dependent.

### 7.3.1 TLB Table Entry Contents

TLB Table Entries (TTE) contain the following fields:

- Address Space Identifier (ASI) — specifies the address space corresponding to the virtual address. ASIs allow mappings of up to 32 virtual address spaces to coexist in the TLB. An ASI is similar to a Process ID.
- Virtual Page Number (VPN) — stores  $32 - \log_2 \text{PageSize}$  bits where *PageSize* is the size of the page in bytes.
- Physical Page Number (PPN) — stores  $32 - \log_2 \text{Page size}$  bits where *Page size* is the size of the page in bytes.
- Execute Enable (XE) — enables instruction fetches to the page.
- Write Enable (WE) — enables data writes to the page.
- Read Enable (RE) — enables data reads from the page.
- Cacheability bit (C) — indicates that the page is cacheable.
- Global bit (G) — indicates that the page is globally mapped thus making it visible in all address spaces.
- Valid bit (V) — indicates that the TTE contains a valid mapping.

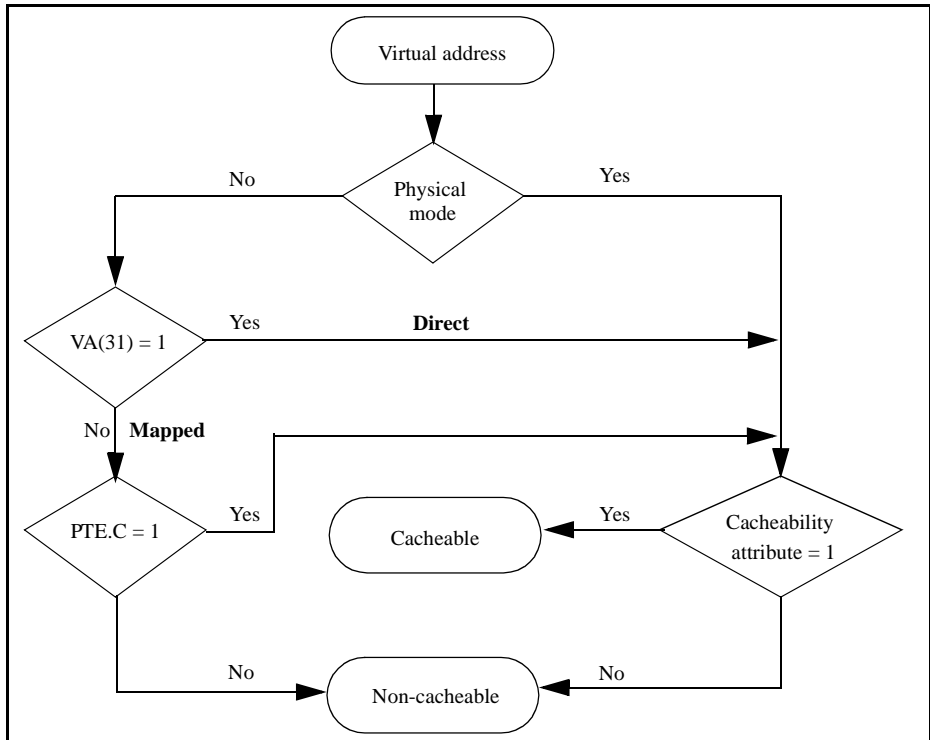
## 7.4 Cacheability

The cacheability of a virtual address is determined using separate mechanisms for the two translation paths.

### 7.4.1 Cacheability for direct translation

The cacheability status of a virtual address that undergoes direct translation is controlled by an implementation-specific cacheability attribute associated with the segment. The segment cacheability attributes are not a part of the MMU specification. These cacheability attributes are provided by the system memory map for the specific CPU core. **Figure 7-4** shows the criteria for cachability of a virtual address. A virtual address is cachable according to the following pseudocode description.

```
if (MMUCON.V == 0) { /* Physical mode */
if (Cacheability_attribute == 1)
Cacheable = True
else
Cacheable = False
} else {
if (VA(31) == 1) { /* Reference to upper half of virtual memory */
if (Cacheability_attribute == 1) /* Direct map */
Cacheable = True
else
Cacheable = False
} else if (PTE.C == 1) /* Page Table Entry cachability property set */
{ if (cacheability_attribute == 1) /* Physical Page Attribute override */
Cacheable = True
} else
Cacheable = False
}
```



**Figure 7-4 Cacheability of a virtual address**

#### 7.4.2 Cacheability for PTE based translation

The cacheability status of a virtual address that undergoes PTE-based translation is determined using the cacheability attribute of the PTE used for the address translation. Each PTE has a C bit that controls the cacheability status of the page.

### 7.5 Protection

Memory protection is enforced using separate mechanisms for the two translation paths as described in this section.

#### 7.5.1 Protection for Direct Translation

Memory protection for addresses that undergo direct translation is enforced using standard TriCore range-based protection. See the chapter titled **Protection System**. The range-based protection mechanism provides support for protecting memory ranges from unauthorized read, write, or instruction fetch accesses.

2001-04-30 @ 15:16

Furthermore, User-0 accesses to virtual addresses in the upper half of the virtual address space are disallowed when operating in Virtual mode. In Physical mode, User-0 accesses are disallowed only to segments 14 and 15. Any User-0 access to a virtual address that is restricted to User-1 or Supervisor mode will cause a Virtual Address Protection (VAP) Trap in both the Physical and Virtual modes.

### 7.5.2 Protection for PTE-Based Translation

Memory protection for addresses that undergo PTE-based translation is enforced by examining properties of the PTE used for the address translation. The PTE provides support for protecting a process from unauthorized read, write, or instruction fetches by other processes. The PTE has the following bits that are provided for this purpose:

- Execute Enable (XE) — enables instruction fetch to the page.
- Write Enable (WE) — enables data writes to the page.
- Read Enable (RE) — enables data reads from the page.

### 7.6 Multiple Address Spaces

The MMU provides efficient support for multiple virtual address spaces. Each TTE contains an Address Space Identifier (ASI) which identifies the address space corresponding to the particular virtual address. Ambiguities in virtual address mappings are avoided by the use of the Address Space Identifier. The Address Space Identifier Register (ASI) is also provided to support multiple address spaces.

Virtual address translation is performed by a TTE if:

- It is a valid non-global TTE that matches the incoming VPN of the virtual address and the Address Space Identifier contained in the ASI register, or
- It is a valid global TTE that matches the incoming VPN.

Note that global TTEs are indicated by the G bit and such mappings are visible to all virtual address spaces.

### 7.7 MMU traps

MMU traps belong to Trap Class Number (TCN) 0 in the TriCore architecture. The MMU can generate the following traps:

- VAF (Virtual Address Fill)
- VAP (Virtual Address Protection)

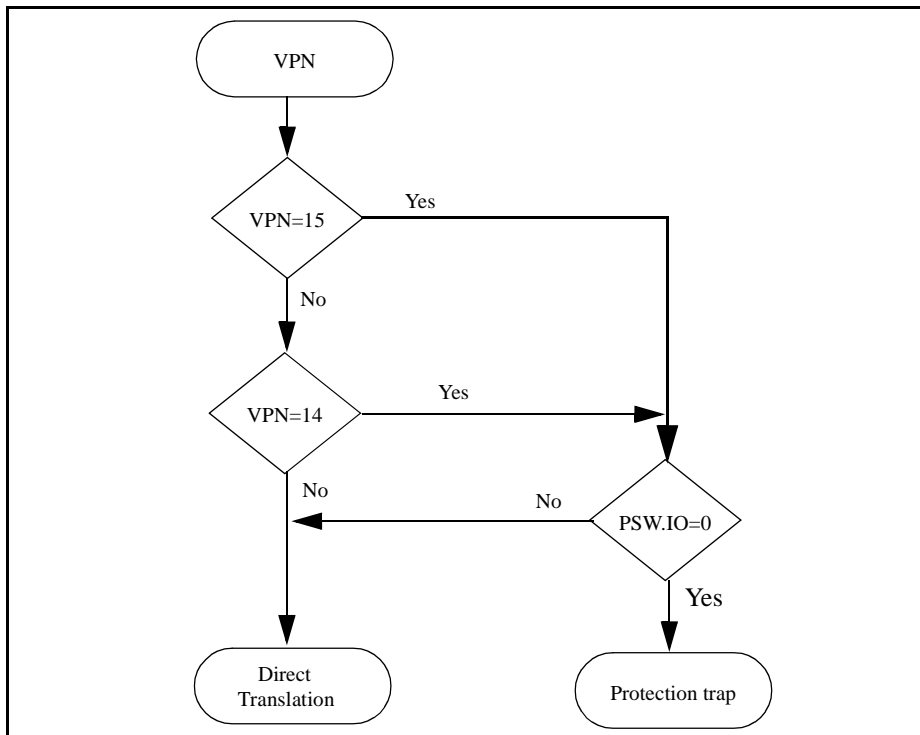
See **Trap Types** in the chapter titled **Memory Management**.

The Virtual Address Fill trap is generated if PTE-based translation is required for a virtual address and the PTE corresponding to the translation is missing in the MMU. The Virtual Address Protection trap is generated if the access is disallowed. The VAF trap is assigned a TIN (Trap Identification Number) of 0 while the VAP trap is assigned a TIN of 1. Both the VAF and VAP traps are synchronous traps.

The events that happen on an MMU trap are identical to the events that happen on any other trap. The virtual address is right shifted by  $10 + 2 \cdot \min(\text{SZA}, \text{SZB})$  and loaded into the Translation Fault Address (TFA) register.



Figure 7-5 shows how MMU traps in Physical mode are handled. In Physical mode, User-0 accesses are disallowed to segments 14 and 15. Any User-0 access to a virtual address that is restricted to User-1 or Supervisor mode will cause a Virtual Address Protection (VAP) Trap.



**Figure 7-5 MMU traps in Physical mode**

Figure 7-6 shows how MMU traps in Virtual mode are handled. User-0 accesses to virtual addresses in the upper half of the virtual address space are disallowed when operating in Virtual mode. Any User-0 access to a virtual address that is restricted to User-1 or Supervisor mode will cause a Virtual Address Protection (VAP) Trap.

2001-04-30 @ 15:16

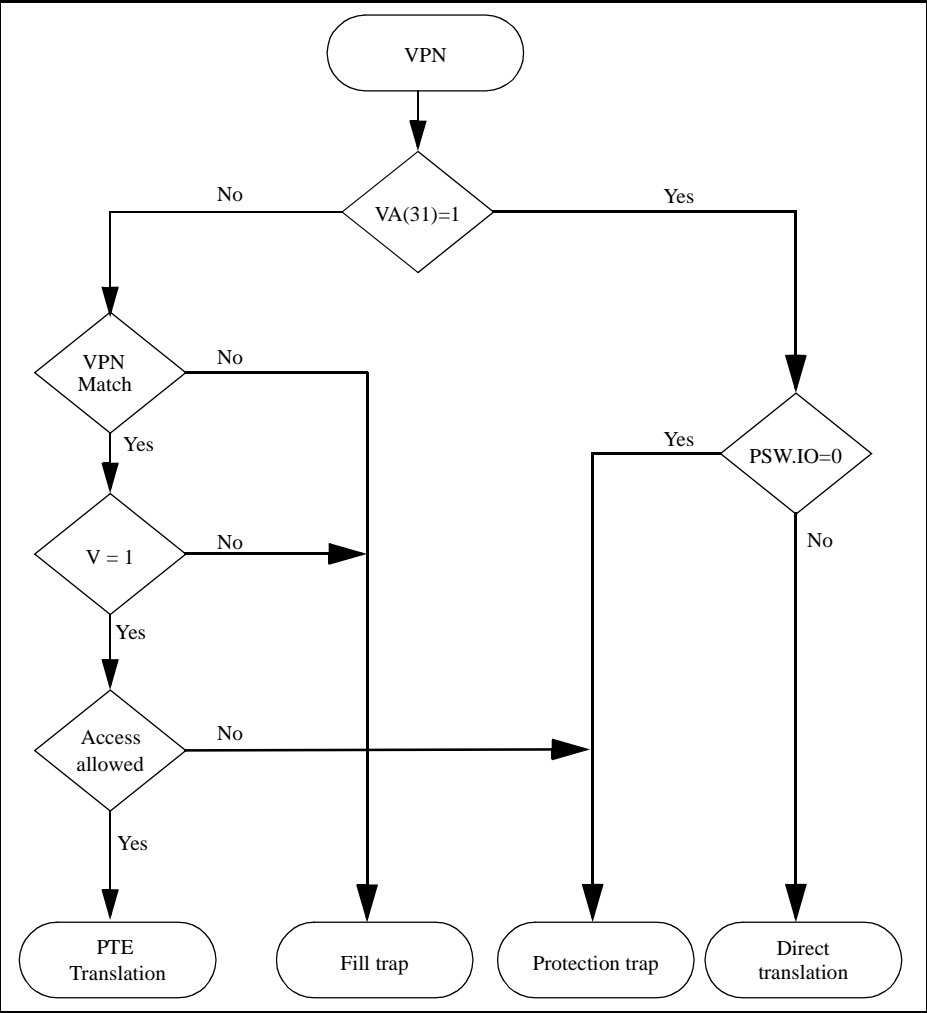


Figure 7-6 MMU traps in Virtual mode

7.8 MMU instructions

All MMU instructions are privileged instructions that require PSW.IO = 2 (Supervisor mode) for execution. If the MMU is physically present (MMUCON.MXT = 0) the instructions are non-faulting and execute normally whether the MMU is enabled or not (MMUCON.V = 0 or 1). If there is no MMU present (MMUCON.MXT = 1) then all MMU instructions casue an unimplimented instruction trap.



2001-04-30 @ 15:16

The TLBPROBE instructions return the ASI and VPN of the translation in the Translation Virtual Address register (TVA), the PPN and attributes in the Translation Physical Address register (TPA), and the TLB index of the translation in the Translation Page Index register (TPX). The TPA.V bit is set to zero if the TTE contained an invalid translation or an invalid index was used for the probe.

### 7.9 MMU Special Function Registers

All MMU Special Function Registers are memory-mapped. All registers can be read using the MFCR instruction. The MMUCON and ASI registers are the only software-writeable registers. The MMUCON and ASI registers are written using the MTCR instruction.

#### 7.9.1 MMU Configuration register (MMUCON)

##### MMUCON MMU Configuration Register

| 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  |
| MXT | -  |    |    | TSZ |    |    |    |    |    |    | SZB | SZA |    | V  |    |

| Field | Bits | Type | Value | Description   |
|-------|------|------|-------|---|
| MXT   | 15   | r    |       | MMU Exists<br>indication if there is an MMU physically instantiated. This aids SW diagnostics and OSs to determine if there is an MMU resource present, and indicates whether MMU instructions will trap            |
|       |      |      | 0     | MMU exists in the design and is present   |
|       |      |      | 1     | MMU does not exist in the design (all other bits in MMUCON undefined)   |
| TSZ   | 11:5 | r    |       | TLB Size<br>Determines the size of each of the TLBs. The entries of TLB-A are indexed 0 through TSZ while the entries of TLB-B are indexed 128, 128 through 128+TSZ. Thus, each TLB has a maximum of TSZ+1 entries. |

2001-04-30 @ 15:16

| Field      | Bits | Type | Value   | Description   |
|------------|------|------|---|---------------|
| <b>SZB</b> | 4:3  | rw   | Page Size B.<br>Page size of the mappings in TLB-B.   |               |
|            |      |      | 00  | 1KB           |
|            |      |      | 01  | 4KB           |
|            |      |      | 10  | 16KB          |
|            |      |      | 11  | 64KB          |
| <b>SZA</b> | 2:1  | rw   | Page Size A.<br>Page size of the mappings in TLB-A.   |               |
|            |      |      | 00  | 1KB           |
|            |      |      | 01  | 4KB           |
|            |      |      | 10  | 16KB          |
|            |      |      | 11  | 64KB          |
| <b>V</b>   | 0    | rw   | Virtual mode.<br>In Virtual mode, the lower half of the virtual address space undergoes PTE-based translation and the upper half of the virtual address space undergoes direct translation. Clearing this bit sets the processor in Physical mode whereby the virtual address is used directly as the physical address. |               |
|            |      |      | 0   | Physical mode |
|            |      |      | 1   | Virtual mode  |

Note: If MMUCON.MXT = 1 (MMU does not exist) then all other registers in the section do not exist and are undefined. If they are accessed no error occurs, but the read and write results are undefined.

## 7.9.2 Address Space Identifier (ASI)

### ASI

#### Address Space Identifier Register

|    |    |    |    |    |    |    |    |    |    |    |     |    |    |    |    |
|----|----|----|----|----|----|----|----|----|----|----|-----|----|----|----|----|
| 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  |
| -  |    |    |    |    |    |    |    |    |    |    | ASI |    |    |    |    |

2001-04-30 @ 15:16

| Field | Bits | Type | Value | Description  |
|-------|------|------|-------|--|
| ASI   | 4:0  | rw   |       | Address Space Identifier<br>The ASI register contains the Address Space Identifier of the current process. |

### 7.9.3 Translation Virtual Address register (TVA)

The TVA register is used to return the ASI and VPN of a translation by a TLBPROBE instruction.

#### TVA

##### Translation Virtual Address

|     |    |    |     |    |    |    |    |     |    |    |    |    |    |    |    |
|-----|----|----|-----|----|----|----|----|-----|----|----|----|----|----|----|----|
| 31  | 30 | 29 | 28  | 27 | 26 | 25 | 24 | 23  | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
| 0   | 0  | 0  | ASI |    |    |    |    | VPN |    |    |    |    |    |    |    |
| 15  | 14 | 13 | 12  | 11 | 10 | 9  | 8  | 7   | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
| VPN |    |    |     |    |    |    |    |     |    |    |    |    |    |    |    |

| Field | Bits  | Type | Value | Description  |
|-------|-------|------|-------|--|
| ASI   | 28:24 | r    |       | Address Space Identifier<br>The ASI register contains the Address Space Identifier of the current process. |
| VPN   | 23:0  | r    |       | Virtual Page Number  |

### 7.9.4 Translation Physical Address register (TPA)

The TPA register is used to return the PPN and attributes of a translation by a TLBPROBE instruction.

#### TPA

#### Translation Physical Address Register

| 31  | 30 | 29 | 28 | 27 | 26 | 25  | 24 | 23  | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
|-----|----|----|----|----|----|-----|----|-----|----|----|----|----|----|----|----|
| V   | XE | WE | RE | G  | C  | PSZ |    | PPN |    |    |    |    |    |    |    |
| 15  | 14 | 13 | 12 | 11 | 10 | 9   | 8  | 7   | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
| PPN |    |    |    |    |    |     |    |     |    |    |    |    |    |    |    |

| Field | Bits | Type | Value  | Description         |
|-------|------|------|--|---------------------|
| V     | 31   | r    | Valid bit<br>Indicates that the TTE contains a valid mapping   |                     |
|       |      |      | 0  | Invalid             |
|       |      |      | 1  | Valid               |
| XE    | 30   | r    | Execute Enable<br>Enables instruction fetches to the page.   |                     |
|       |      |      | 0  | Disabled            |
|       |      |      | 1  | Enabled             |
| WE    | 29   | r    | Write Enable<br>Enables data writes to the page.   |                     |
|       |      |      | 0  | Disabled            |
|       |      |      | 1  | Enabled             |
| RE    | 28   | r    | Read Enable<br>Enables data reads from the page.   |                     |
|       |      |      | 0  | Disabled            |
|       |      |      | 1  | Enabled             |
| G     | 27   | r    | Global bit<br>Indicates that the page is globally mapped thus making it visible in all address spaces. |                     |
|       |      |      | 0  | Not globally mapped |
|       |      |      | 1  | Globally mapped     |

2001-04-30 @ 15:16

| Field      | Bits  | Type | Value | Description  |
|------------|-------|------|-------|--|
| <b>C</b>   | 26    | r    |       | Cacheability bit<br>Indicates that the page is cacheable.  |
|            |       |      | 0     | Not cacheable  |
|            |       |      | 1     | Cacheable  |
| <b>PSZ</b> | 25:24 | r    |       | Page size<br>1 KB, 4 KB, 16 KB, and 64 KB page sizes   |
|            |       |      | 00    | 1 KB   |
|            |       |      | 01    | 4 KB   |
|            |       |      | 10    | 16 KB  |
|            |       |      | 11    | 64 KB  |
| <b>PPN</b> | 23:0  | r    |       | Physical Page Number<br>Stores $32 - \log_2 \text{Pagesize}$ bits where <i>Pagesize</i> is the size of the page in bytes based on the PSZ field. |

### 7.9.5 Translation Page Index Register (TPX)

The TPX register is used to return the TLB index of a translation by a tlbprobe instruction.

#### TPX

#### Translation Page Index Register

|    |    |    |    |    |    |    |    |       |    |    |    |    |    |    |    |
|----|----|----|----|----|----|----|----|-------|----|----|----|----|----|----|----|
| 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  |
| -  |    |    |    |    |    |    |    | Index |    |    |    |    |    |    |    |

| Field        | Bits | Type | Value | Description        |
|--------------|------|------|-------|--------------------|
| <b>Index</b> | 7:0  | r    |       | Translation index. |



### 7.9.6 Translation Fault Address register (TFA)

The TFA register contains the faulting virtual address right shifted by  $10 + 2 \cdot \min(\text{SZA}, \text{SZB})$  bits.

#### TFA

##### Translation Fault Address Register

| 31  | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23  | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
|-----|----|----|----|----|----|----|----|-----|----|----|----|----|----|----|----|
| 0   |    |    |    |    |    |    |    | FVA |    |    |    |    |    |    |    |
| 15  | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7   | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
| FVA |    |    |    |    |    |    |    |     |    |    |    |    |    |    |    |

| Field | Bits | Type | Value | Description              |
|-------|------|------|-------|--------------------------|
| FVA   | 23:0 | r    |       | Faulting virtual address |

### 7.9.7 Reset values

All registers other than the MMUCON register have undefined values at reset. The MMUCON register is set to 0b0yyyyyyy00000 at reset where yyyyyyy is implementation-dependent.

2001-04-30 @ 15:16

---

# Protection System

---

2001-04-30 @ 15:16

## 8 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 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. Other sections introduce the use of the protection features by software in real-time systems.

### 8.1 Direct and Page Table Entry Addressing

Virtual addresses are translated into physical addresses using one of two translation mechanisms: (a) direct translation, and (b) Page Table Entry (PTE) based translation. Memory protection for addresses that undergo direct address translation is enforced using the range-based memory protection system described in this chapter. The range-based protection mechanism provides support for protecting memory ranges from unauthorized read, write, or instruction fetch accesses. Page Table Entry based translation occurs if the processor is operating in Virtual mode. For a discussion of virtual addressing, see [Memory Management](#).

User-0 accesses to virtual addresses in the upper half of the virtual address space are disallowed when operating in Virtual mode. In Physical mode, User-0 accesses are disallowed only to segments 14 and 15. Any User-0 access to a virtual address that is restricted to User-1 or Supervisor mode will cause a Virtual Address Protection (VAP) Trap in both the Physical and Virtual modes.

### 8.2 Protection System Registers

There are 2 major components to the protection system:

- The control bit fields in the Program Status Word (PSW) register
- The memory protection registers which control program execution and memory access

See Program Status Word (PSW) section in the Core Registers chapter.

### 8.2.1 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 Special Function Register (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.

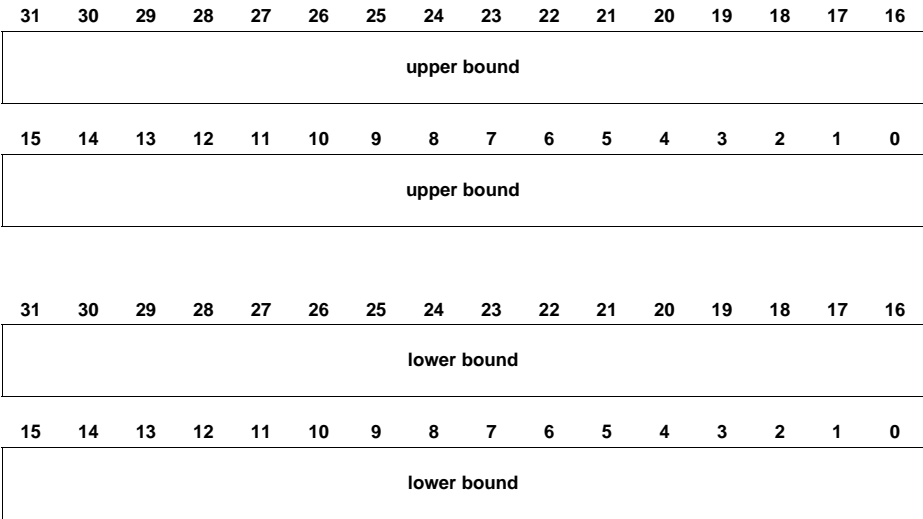
#### DPRx\_n

##### Data Segment Protection Register Pair

| 31          | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
|-------------|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| upper bound |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| 15          | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
| upper bound |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| 31          | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
| lower bound |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| 15          | 14 | 13 | 12 | 11 | 10 | 9  | 8  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |
| lower bound |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

2001-04-30 @ 15:16

CPRx\_n  
Code Segment Protection Register Pair



2001-04-30 @ 15:16

**DPMx\_n**
**Data Protection Mode Register**

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

| Field | Bits | Type | Value | Description   |
|-------|------|------|-------|---|
| WE    | 7    | rw   |       | Address Field Write Enable  |
|       |      |      | 0     | Data write accesses to associated address range not permitted   |
|       |      |      | 1     | Data write accesses to associated address range permitted   |
| RE    | 6    | rw   |       | Address Field Read Enable   |
|       |      |      | 0     | Data read accesses to associated address range not permitted  |
|       |      |      | 1     | Data read accesses to associated address range permitted  |
| WS    | 5    | rw   |       | Address Range Data Write Signal   |
|       |      |      | 0     | Data write signal disabled  |
|       |      |      | 1     | Signal asserted to debug unit on data write accesses to associated address range  |
| RS    | 4    | rw   |       | Address Range Data Read Signal  |
|       |      |      | 0     | Data read signal disabled   |
|       |      |      | 1     | Signal asserted to debug unit on data read accesses to associated address range   |
| WBL   | 3    | rw   |       | Data Write Signal on Lower Bound Access   |
|       |      |      | 0     | Data write signal disabled  |
|       |      |      | 1     | Signal asserted to debug unit on data write access to an address that matches lower bound address of associated address range |
| RBL   | 2    | rw   |       | Data Read Signal on Lower Bound Access  |
|       |      |      | 0     | Data read signal disabled   |
|       |      |      | 1     | Signal asserted to debug unit on data read access to an address that matches lower bound address of associated address range  |
| WBU   | 1    | rw   |       | Data Write Signal on Upper Bound Access   |
|       |      |      | 0     | Write signal disabled   |
|       |      |      | 1     | Signal asserted to debug unit on data write access to an address that matches upper bound address of associated address range |

2001-04-30 @ 15:16

| Field | Bits | Type | Value | Description  |
|-------|------|------|-------|--|
| RBU   | 0    | rw   |       | Data Read Signal on Upper Bound Access   |
|       |      |      | 0     | Data read signal disabled  |
|       |      |      | 1     | Signal asserted to debug unit on data read access to an address that matches upper bound address of associated address range |



**CPMx\_n**
**Code Protection Mode Register**

|           |            |           |            |           |            |            |           |
|-----------|------------|-----------|------------|-----------|------------|------------|-----------|
| 7         | 6          | 5         | 4          | 3         | 2          | 1          | 0         |
| <b>XE</b> | <b>Res</b> | <b>XS</b> | <b>Res</b> | <b>BL</b> | <b>Res</b> | <b>Res</b> | <b>BU</b> |

| Field      | Bits | Type | Value | Description  |
|------------|------|------|-------|--|
| <b>XE</b>  | 7    |      |       | Address Range Execute Enable   |
|            |      |      | 0     | Instruction fetch accesses to associated address range not permitted   |
|            |      |      | 1     | Instruction fetch accesses to associated address range permitted   |
| <b>Res</b> | 6    |      | -     |  |
| <b>XS</b>  | 5    |      |       | Address Range Execute Signal   |
|            |      |      | 0     | Execute signal disabled  |
|            |      |      | 1     | Signal asserted to debug unit on instruction fetch accesses to associated address range  |
| <b>Res</b> | 4    |      | -     |  |
| <b>BL</b>  | 3    |      |       | Execute Signal on Lower Bound Access   |
|            |      |      | 0     | Lower bound execute signal disabled  |
|            |      |      | 1     | Signal asserted to debug unit on instruction fetch access to an address that matches lower bound address of associated address range |
| <b>Res</b> | 2:1  |      | -     |  |
| <b>BU</b>  | 0    |      |       | Execute Signal on Upper Bound Access   |
|            |      |      | 0     | Upper bound execute signal disabled  |
|            |      |      | 1     | Signal asserted to debug unit on instruction fetch access to an address that matches upper bound address of associated address range |

2001-04-30 @ 15:16

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. 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 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 2, and the maximum number is 4. In a 2-set implementation, 1 of the sets corresponds to the current background task, and the other is common to any ISR. (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.

### 8.2.1.1 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 2 uses are not mutually exclusive.

### 8.2.1.2 Using Protection Register Sets

If there were only 1 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 2 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.

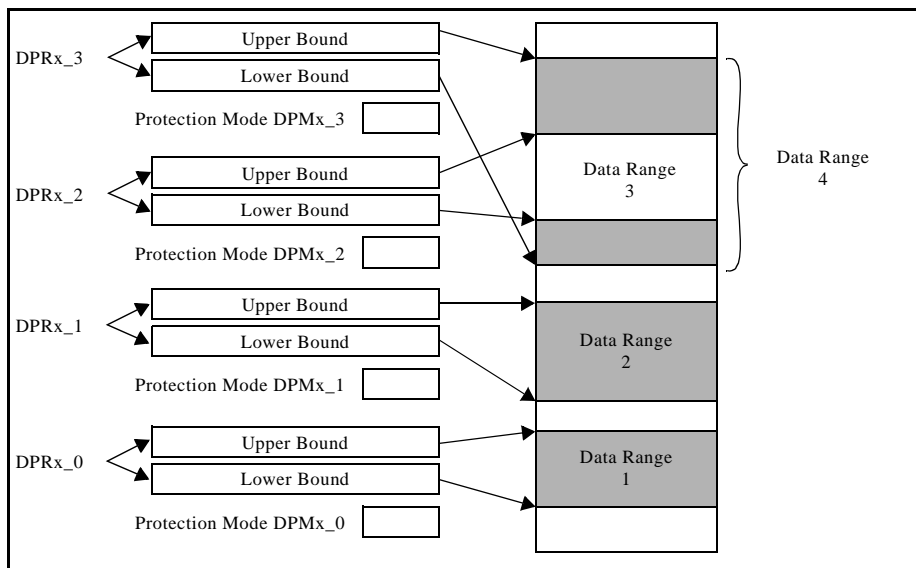
### 8.3 Sample Protection Register Set

**Figure 8-1** illustrates Data Protection Register Set  $n$ , where  $n$  is one of the 4 sets as selected by the PSW.PRS field. Each register set in this example consists of 4 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 4 Data Segment Protection Registers and 4 Data Protection Mode Registers are set up as follows:

- Data Segment Protection Register 3 (DPRx\_3) defines the upper and lower bound for Data Range 4. Data Protection Mode Register 3 (DPMx\_3) defines the permissions and debug conditions for Data Range 4.
- Data Segment Protection Register 2 (DPRx\_2) defines the upper and lower bound for Data Range 3. Data Protection Mode Register 2 (DPMx\_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 (DPRx\_1) defines the upper and lower bound for Data Range 2. Data Protection Mode Register 1 (DPMx\_1) defines the permissions and debug conditions for Data Range 2.
- Data Segment Protection Register 0 (DPRx\_0) defines the upper and lower bound for Data Range 1. Data Protection Mode Register 0 (DPMx\_0) defines the permissions and debug conditions for Data Range 1.

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

2001-04-30 @ 15:16



**Figure 8-1**

### Example Configuration of a Data Protection Register Set

#### 8.4 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 2 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 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 access attempt depends on the presence of a peripheral at the accessed address,

2001-04-30 @ 15:16

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.

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 1 or more of the ranges specified in the currently selected protection register set. Furthermore, the mode entry for at least 1 of the matching ranges must enable the requested type of access.

#### 8.4.1 Permitted vs. Valid Accesses

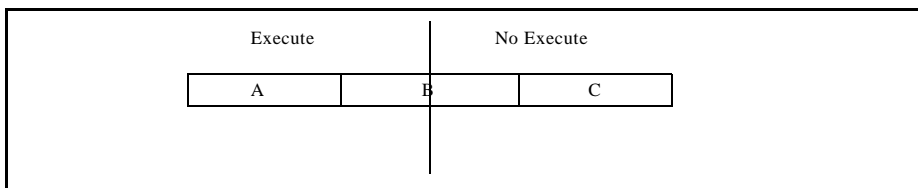
A memory access can be permitted within the ranges specified in the data and code range tables without necessarily being valid. A range specified in a range table entry could cover 1 or more address regions where no physical memory was implemented. Although that would normally reflect an error in the 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 1 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.

#### 8.4.2 Crossing Protection Boundaries

An access can straddle two regions. For example, **Figure 8-2** 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 8-2**  
**Protection Boundaries**

Because the PC is used in the comparison with the range registers, the execute permissions 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 range registers.

2001-04-30 @ 15:16

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

---

2001-04-30 @ 15:16

## 9 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.

### 9.1 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) register. The status flags are described in [Table 9-1](#).

**Table 9-1**  
**PSW Status Flags**

| Status Flag | Description   |
|-------------|---|
| <b>C</b>    | 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 arithmetic shift.       |
| <b>V</b>    | Overflow. This flag is set when the signed 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 $2^{31}-1$ .              |
| <b>SV</b>   | 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.  |
| <b>AV</b>   | Advance Overflow. This flag is updated by all instructions that update the overflow flag and no others. This flag is determined as the Boolean exclusive-or of the two most-significant bits of the result. |
| <b>SAV</b>  | Sticky Advance 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 signed overflow conditions (overflow and advance overflow) are calculated for all arithmetic instructions. In the case of packed instructions, the conditions are the OR of the conditions for each byte or halfword (parallel) operation. In the case of the multiply-accumulate instructions, the conditions are calculated after the accumulate operation.

The unsigned overflow condition is carry for addition and borrow, i.e. no carry, for subtraction.

Numerically, for signed 32-bit values, overflow occurs when a positive result is greater than 0x7FFFFFFF or a negative result is smaller than 0x80000000. For unsigned 32-bit values, overflow occurs when the result is greater than 0xFFFFFFFF or less than 0x00000000.

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



2001-04-30 @ 15:16

(using MFCR). Jumping based on the overflow result can be done using a MFCR followed by a JZ.T or JNZ.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", 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.

## **9.1.1 Integer Arithmetic**

### **9.1.1.1 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. A 16-bit constant is created 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 MOVH instruction followed by an ADDI (Add Immediate), or a MOV.U followed by ADDIH (Add Immediate High Word).

### **9.1.1.2 Addition and Subtraction**

The addition instructions have 3 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 constants have similar corresponding subtract instructions. Because the immediate of ADDI is sign-extended, it may be used for both addition and subtraction.

2001-04-30 @ 15:16

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

#### 9.1.1.3 Multiply and Multiply-Add

To efficiently support both compiled C applications code and DSP code, there are a large number of instruction forms for multiplication and for multiplication with accumulation. Many of these instructions share common assembler mnemonics, but are distinguished by the operand forms encoded in the assembly instruction.

For multiplication of 32-bit integers, the available mnemonics are MUL (Multiply Signed), MULS (Multiply Signed with Saturation), and MULS.U (Multiply Unsigned with Saturation). These translate to machine instructions producing either 32- or 64-bit results, depending on whether the destination operand encoded in the assembly instruction is a single data register (Dn, where n = 0, 1, .. 15) or an extended data register (En, where n = 0, 2, .. 14).

For multiplication of fractional data types ("Q" format), the available mnemonics are MUL.Q (Multiply Q format) and MULR.Q (Multiply Q format with Rounding). For each of these mnemonics, there are eight distinct instructions. The operand encodings for these instructions distinguish between 16-bit source operands in either the upper or lower half of a data register (DnU and DnL), 32-bit source operands (Dn), and 32- or 64-bit destination operands (Dn or En). For both mnemonics, the supported operand combinations are:

16U \* 16U  $\Rightarrow$  32

16L \* 16L  $\Rightarrow$  32

16U \* 32  $\Rightarrow$  32

16L \* 32  $\Rightarrow$  32

32 \* 32  $\Rightarrow$  32

16U \* 32  $\Rightarrow$  64

16L \* 32  $\Rightarrow$  64

32 \* 32  $\Rightarrow$  64

In those cases where the number of bits in the destination is less than the sum of the bits in the two source operands, the result is taken from the upper bits of the product.

These operations are also qualified by a shift count specifier of zero or one, which is applied to the product before it is stored into the result. A shift count of '1' removes the redundant sign bit in a fractional product, and is the normal value for multiplication of fractional numbers. A shift count of '0' leaves in the redundant sign bit, and effectively scales the result by an implicit factor of 0.5.

There are also three assembler mnemonics for various forms of multiplication on packed 16-bit fractionals. The mnemonics are MUL.H (Packed Multiply Q format), MULR.H (Packed Multiply Q format with Rounding), and MULM.H (Packed Multiply Q format, Multiprecision). All of the instructions using these mnemonics perform two 16 x 16 bit multiplications in parallel, using 16-bit source operands in the upper or lower halves of their source operand registers. MUL.H produces two 32-bit products, stored into the upper and lower registers of an extended register pair. Its results are exact, with no need for rounding. MULR.H produces two 16-bit Q-format products, stored into the upper and lower halves of a single 32-bit register. Its 32-bit intermediate products

are rounded before discarding the low order bits, to produce the 16-bit Q-format results. MULM.H sums the two intermediate products, producing a single accumulator-format result that is stored into an extended destination register pair. For all three instruction groups, there are four supported source operand combinations for the two multiplications. They are:

16U \* 16U, 16L \* 16L

16U \* 16L, 16L \* 16U

16U \* 16L, 16L \* 16L

16L \* 16U, 16U \* 16U

The instruction forms for multiplication with accumulation, MADD and MSUB and related variations (MADDS, MADDS.H, MADD.Q, MADDS.Q, etc.), parallel the instruction forms for multiplication. In all cases, a third source operand register is specified, which provides the accumulator to which the multiplier results are added. Refer to the specific MADD and MSUB instruction descriptions for details on the operand encodings supported.

#### 9.1.1.4 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 4, 2, or 1 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.

#### 9.1.1.5 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 2 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.

#### 9.1.1.6 Min, Max, Saturate

Instructions are provided that directly calculate the minimum or maximum of 2 operands. The MIN and MAX instructions are used for signed integers, and MIN.U and MAX.U are used for unsigned integers.

2001-04-30 @ 15:16

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.

#### 9.1.1.7 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 is TRUE, and for the CADDN and CSUBN instructions it is performed if the condition is FALSE.

The SEL instruction copies one of its 2 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 2 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.

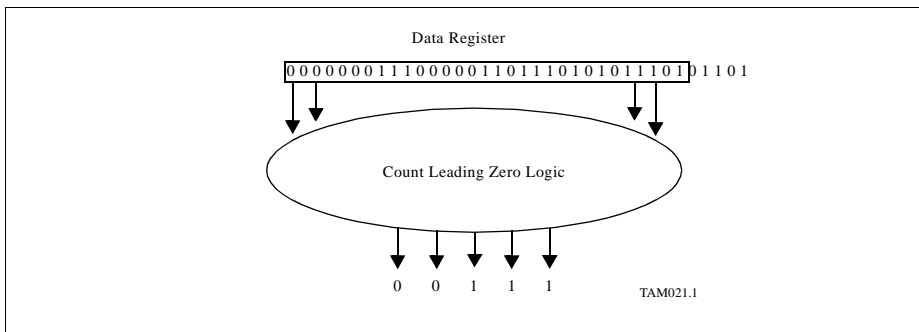
#### 9.1.1.8 Logical

The TriCore architecture provides a complete set of 2-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 1 of the inputs — ANDN and ORN (the negation of an input for XOR is the same as XNOR).

#### 9.1.1.9 Count Leading Zeroes, Ones, and Signs

To provide efficient support for normalization of numerical results, prioritization, and certain graphics operations, 3 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 leading redundant signs, which is the number of leading signs minus 1. 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 below (Figure 9-1), there are 7 zeros in the most-significant portion of the input register. If the most-significant bit of the input is a 1, CLZ returns 0.



**Figure 9-1**  
**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).

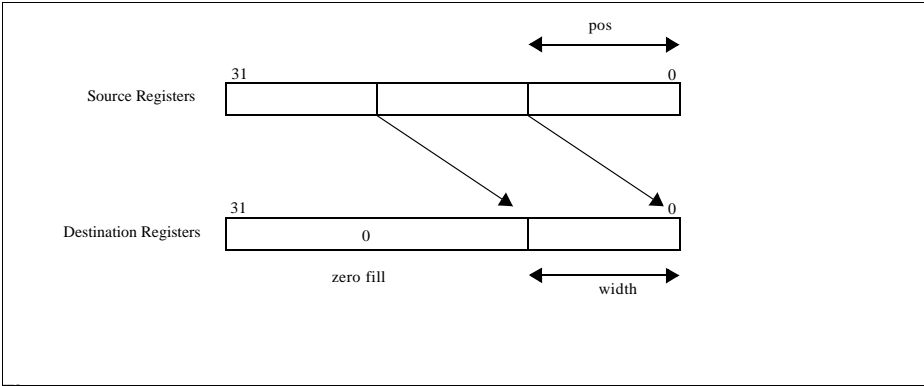
#### 9.1.1.10 Shift

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  $n \geq 0$ , 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.

#### 9.1.1.11 Bit-Field Extract and Insert

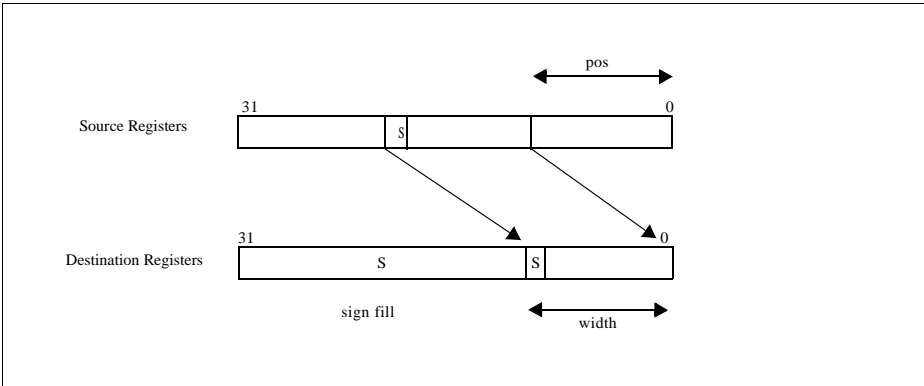
The TriCore architecture supports 3 bit-field extract instructions. The EXTR.U and EXTR instructions extract  $w$  (width) consecutive bits from the source, beginning with the bit number specified by the pos (position) operand. The width and position can be specified by 2 immediate values, by an immediate value and a data register, or by a data register pair. The EXTR.U instruction (Figure 9-2) zero-fills the most significant  $(32-w)$  bits of the result.

2001-04-30 @ 15:16



**Figure 9-2**  
**Operation of EXTR.U Instruction**

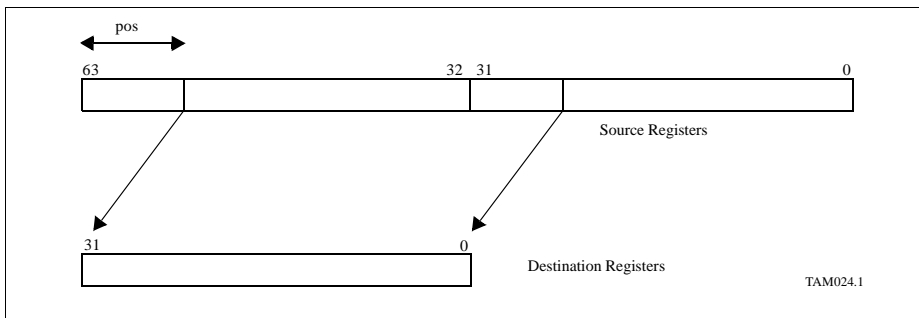
The EXTR instruction ([Figure 9-3](#)) fills the most-significant bits of the result by sign-extending the bit field extracted (i.e. duplicating the most-significant bit of the bit field).



**Figure 9-3**  
**Operation of EXTR Instruction**

The DEXTR instruction ([Figure 9-4](#)), 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.

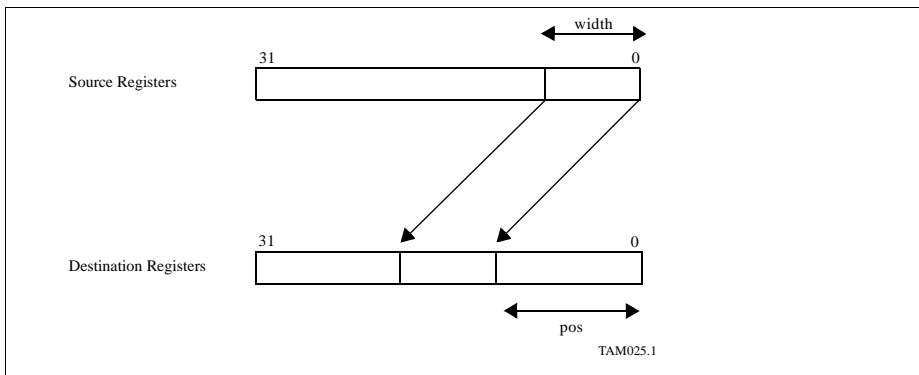
The DEXTR instruction can be used to normalize the result of a DSP filter accumulation in which a 64-bit accumulator is used with several guard bits. The value of pos can be determined by using the CLS (Count Leading Signs) instruction. The DEXTR instruction can also be used to perform a multi-bit rotation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



**Figure 9-4**

### Operation of DEXTR Instruction

The INSERT instruction (Figure 9-5) takes the  $w$  least-significant bits of a source data register, shifted left by  $pos$  bits and substitutes them into the value of another source register. All other  $(32-w)$  bits of the value of the second register are passed through. 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 9-5**

### Operation of INSERT Instruction

#### 9.1.2 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.

2001-04-30 @ 15:16

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 2 left-justified, signed fractions and return a 32-bit signed fraction.

#### 9.1.2.1 Scaling

The multiplier result can be shifted in two ways:

- Left shifted by 1
  - 1 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.

#### 9.1.2.2 Special case = $-1 * -1 \Rightarrow +1$

When multiplying two maximum-negative 16-bit 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. It applies only to 16-bit by 16-bit multiplies and does not apply to 16 by 32-bit or 32 by 32-bit multiplies.

#### 9.1.2.3 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.

#### 9.1.2.4 Rounding

Rounding is used to retain the 16 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.



2001-04-30 @ 15:16

### 9.1.2.5 Overflow and Saturation

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

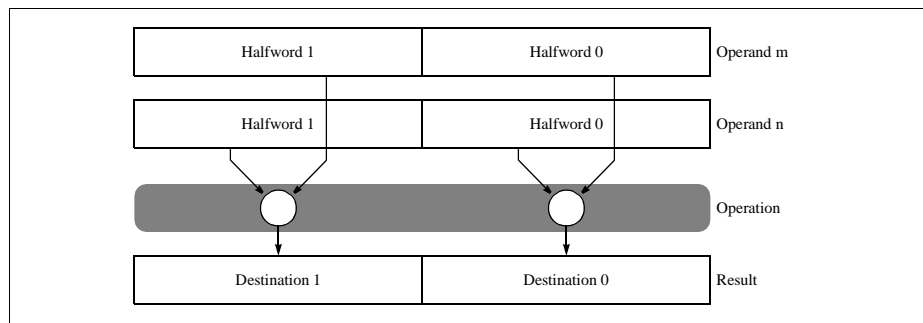
### 9.1.2.6 Sticky Advance Overflow and Block Scaling in FFT

The Sticky Advance 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.

### 9.1.3 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.

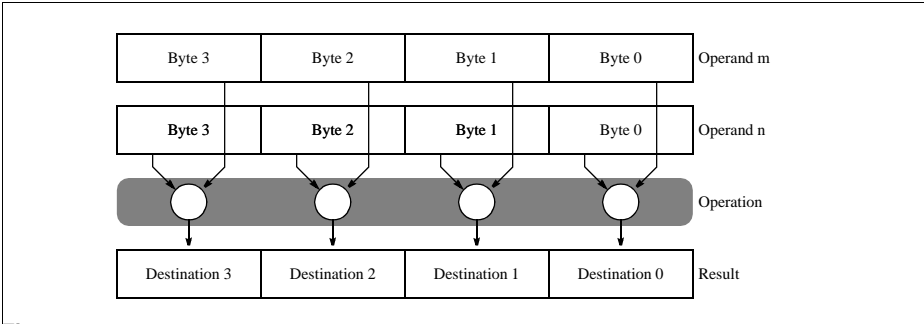
The TriCore architecture supports two packed formats. The first format (Figure 9-6) divides the 32-bit word into two, 16-bit (halfword) values. Instructions which operate on data in this way are denoted in the instruction mnemonic by the “.H” and “.HU” data type modifiers.



**Figure 9-6**

#### **Packed Halfword Data Format**

The second packed format (Figure 9-7) 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.



**Figure 9-7**  
**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.

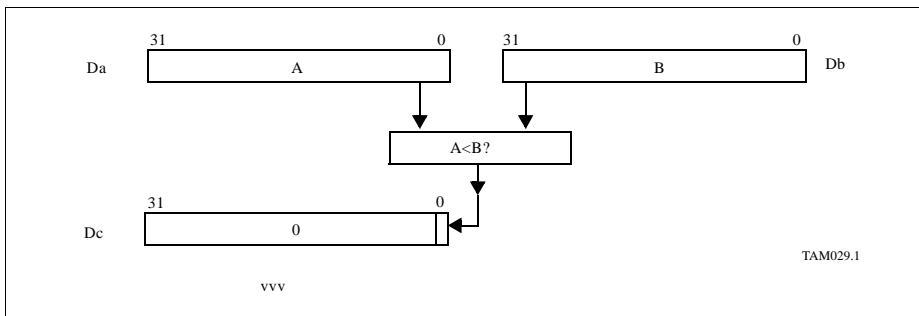
Addition is performed on individual packed bytes or halfwords using the ADD.B and ADD.H instructions and their saturating variations ADDS.B and ADDS.H. ADD.B ignores overflow/underflow within individual bytes, while ADDS.B will saturate individual bytes to the most positive, 8-bit signed integer (127) 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, and absolute difference.

## 9.2 Compare Instructions

The compare instructions perform a comparison of 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 9-8** illustrates the operation of the LT (Less Than) compare instruction.

2001-04-30 @ 15:16



**Figure 9-8**

### 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 (Table 9-2).

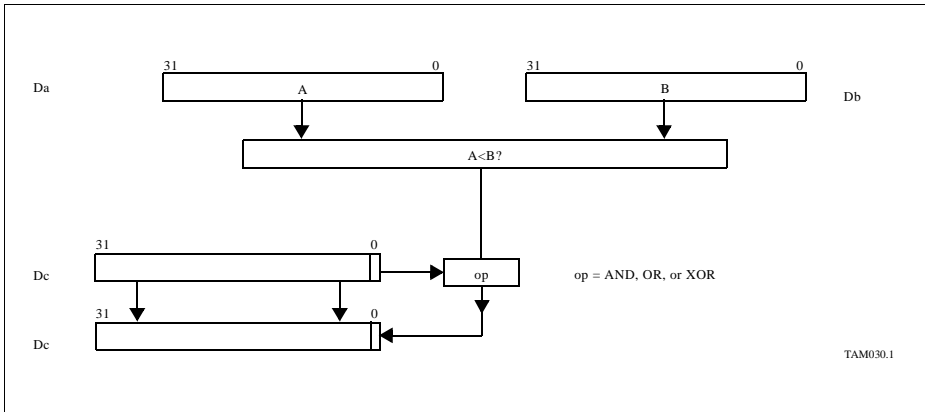
**Table 9-2**

### Equivalent Comparison Operations

| "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 9-9 illustrates combining the LT instruction with a Boolean operation.

2001-04-30 @ 15:16



**Figure 9-9**  
**Combining LT Comparison with Boolean Operation**

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

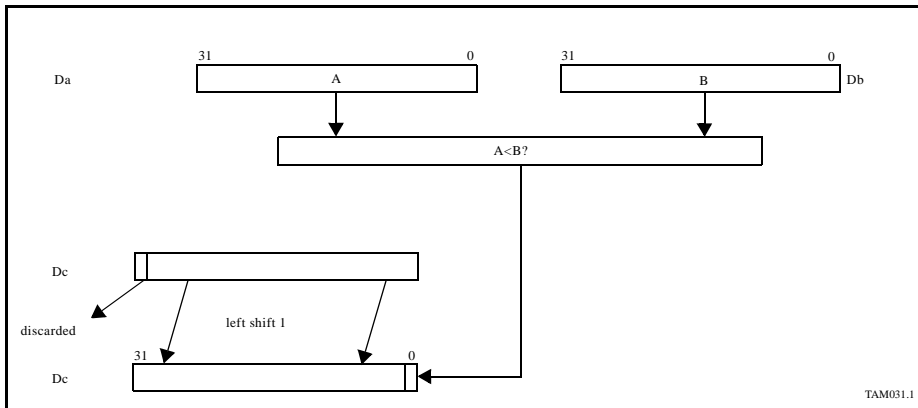
```
d5 = (d1 < d2) || (d3 == d4);
```

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 9-10** illustrates the operation of the SH.LT (Shift Less Than) instruction.

2001-04-30 @ 15:16



**Figure 9-10**  
**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 0xFFFFFFFF for match and 0x00000000 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 9-11** shows an example of the EQ.B instruction.

2001-04-30 @ 15:16

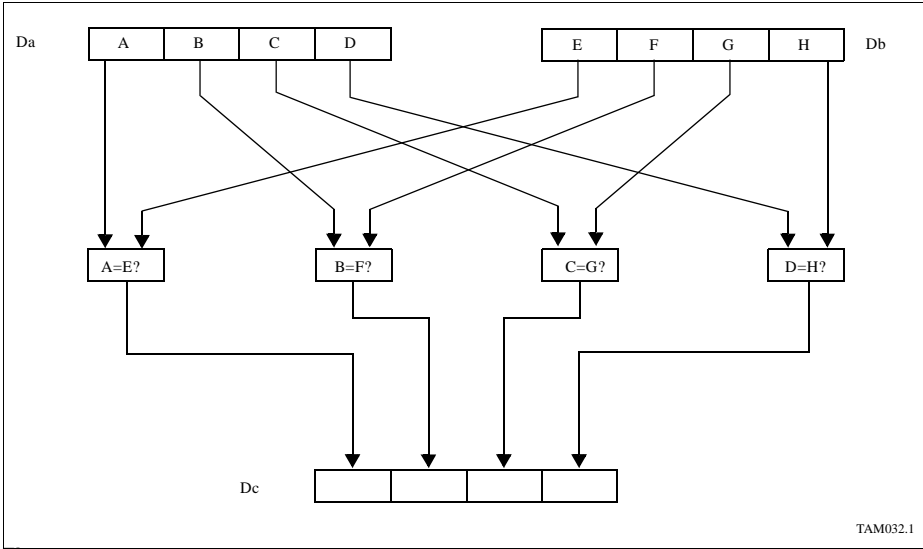


Figure 9-11  
EQ.B Instruction Operation

### 9.3 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, eight instructions with three inputs, and eight with two inputs and a shift.

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 9-12. The available Boolean operations are: AND, NAND, OR, NOR, XOR, XNOR, ANDN, and ORN.

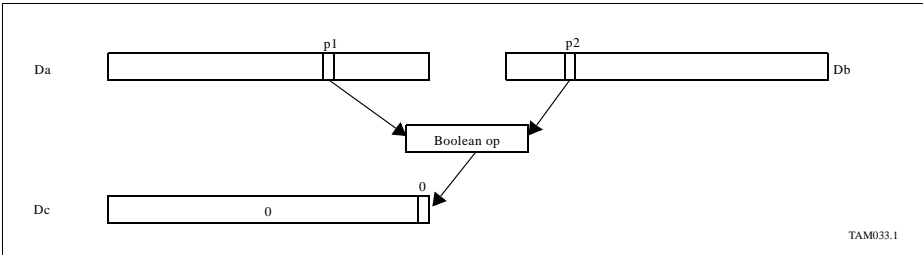
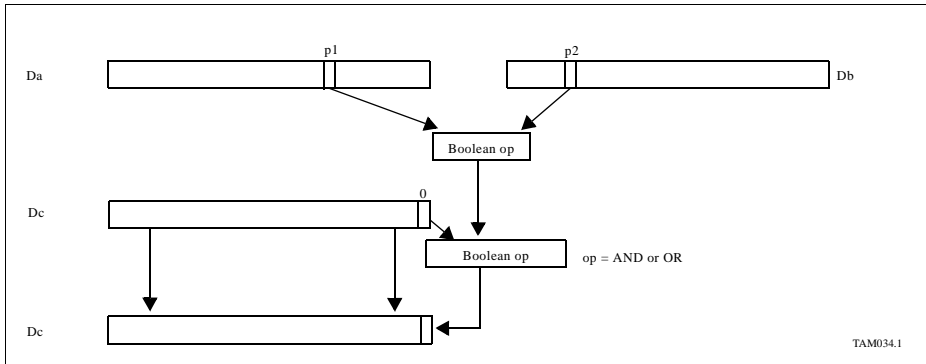


Figure 9-12  
Boolean Operations

2001-04-30 @ 15:16

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 (Figure 9-13)

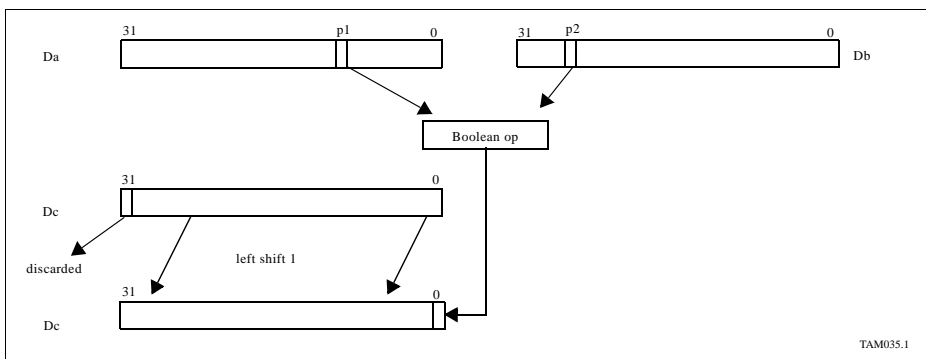


**Figure 9-13**  
**Three-input Boolean Operation**

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 (Figure 9-14).



**Figure 9-14**  
**Shift Plus Boolean Operation**

2001-04-30 @ 15:16

### 9.4 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.

### 9.5 Address Comparison

As with the comparison instructions that use the data registers (see [Compare Instructions](#)), 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 9-15](#).

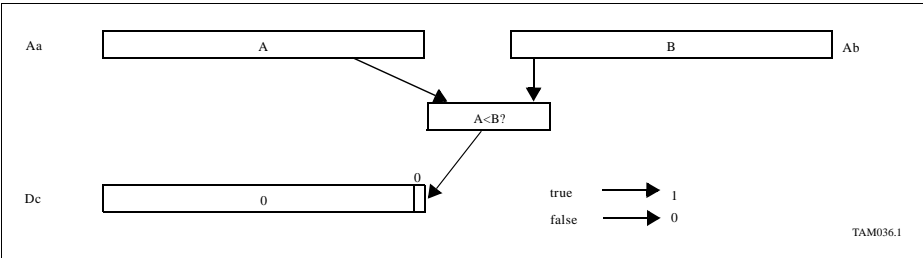


Figure 9-15

#### 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 ([Table 9-3](#)).

**Table 9-3**  
**Operation Equivalents**

| <b>“Missing” Comparison Operation</b> | <b>TriCore Equivalent Comparison Operation</b> |
|---------------------------------------|--|
| 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.

## 9.6 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 (see [Compare Instructions](#)) rather than on the state of condition codes.

### 9.6.1 Unconditional Branch

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

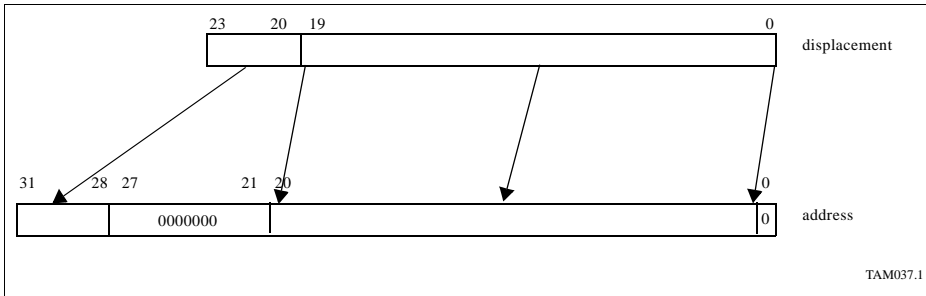
A Jump instruction simply loads the Program Counter with the address specified in the instruction. A Jump and Link instruction does the same, and also stores the address of the next instruction in the “return address register” A11/RA. A jump and Link instruction can be used to implement a subroutine 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. 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 displacement (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 displacement (J), an instruction for an indirect jump (JI), and an instruction for a return (RET).

Both the 24-bit and 8-bit relative 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 9-16](#).

2001-04-30 @ 15:16



TAM037.1

**Figure 9-16**  
**Displacement as Absolute Address**

### 9.6.2 Conditional Branch

The conditional branch instructions use the relative addressing mode, with a displacement value encoded in 4, 8, or 15 bits. The displacement is scaled by 2 before it is used, because all instructions must be aligned on an even (halfword) address. The scaled displacement 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.

#### 9.6.2.1 Conditional Jumps on Data Registers

Six of the conditional jump instructions use a 15-bit signed displacement 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). The second operand to be compared may be an 8-bit sign- or zero-extended constant. 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 displacement is 8-bit in this case. 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 zero-extended 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-bit zero-extended constant in this case.

#### 9.6.2.2 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 displacement 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 zero-extended bits.

2001-04-30 @ 15:16

### 9.6.2.3 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.

There are two 16-bit instructions that test any of the lower 16 bits in the implicit register D15 and have a displacement field of 4 zero-extended bits.

### 9.6.3 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 unconditionally after the comparison. The jump displacement field is 15 bits. For example, a loop that should be executed for D3 = 3, ..., 10 can be implemented as follows:

```
lea    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 time it is executed, but the other 99 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 other 16-bit relative jumps, the 4-bit value is one-extended rather than zero-extended, because this instruction is specifically intended for loops.

An unconditional variant of the LOOP instruction, LOOPU, is provided which utilizes the zero overhead LOOP hardware. Such an instruction is used at the end of a while LOOP body to optimize the jump back to the start of the while construct.

2001-04-30 @ 15:16

## 9.7 Load and Store Instructions

The load and store instructions move data between registers and memory, using seven addressing modes (Table 9-4). The addressing mode determines the effective byte address for the load or store instruction and any update of the base pointer address register.

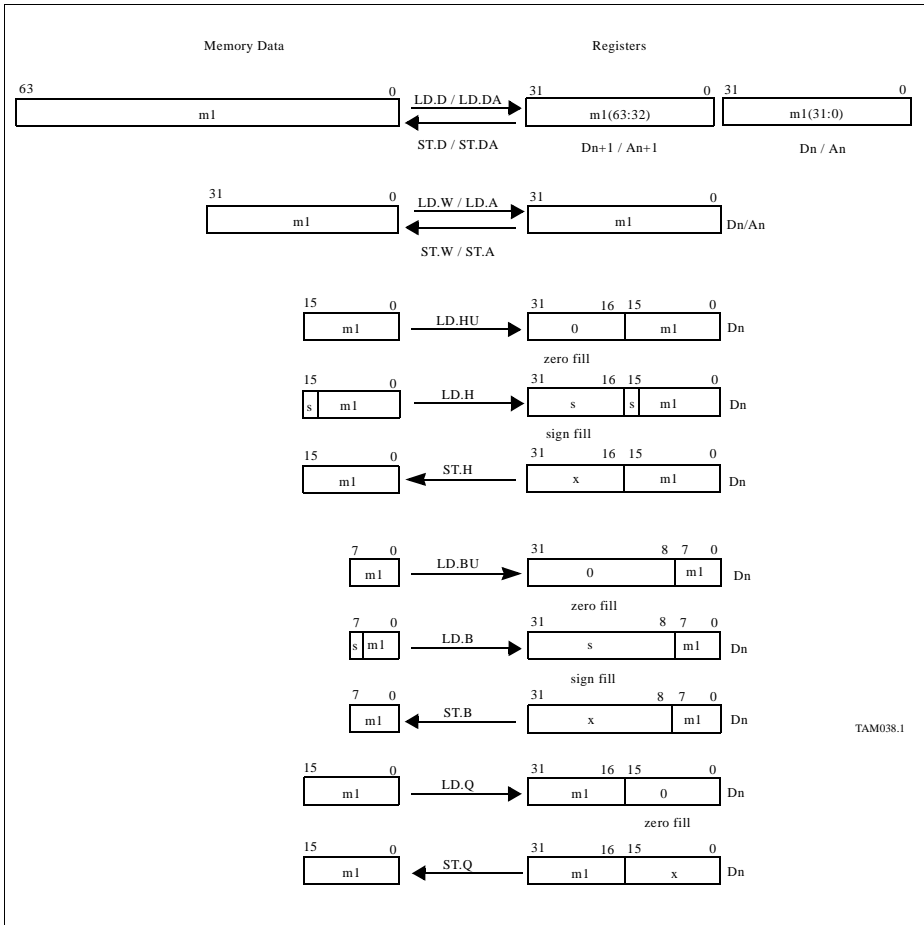
**Table 9-4**  
**Addressing Modes**

| Addressing Mode            | Syntax      | Effective Address                        | Instruction Format |
|----------------------------|-------------|--|--------------------|
| <b>Absolute</b>            | constant    | {offset18[17:14], 14'bo, offset18[13:0]} | ABS                |
| <b>Base + Short Offset</b> | [An]offset  | A[n]+sign_ext(offset10)                  | BO                 |
| <b>Base + Long Offset</b>  | [An]offset  | A[n]+sign_ext(offset16)                  | BOL                |
| <b>Pre-increment</b>       | [+An]offset | A[n]+sign_ext(offset10)                  | BO                 |
| <b>Post-increment</b>      | [An+]offset | A[n]                                     | BO                 |
| <b>Circular</b>            | [An/An+1+c] | A[n]+A[n+1][15:0] (n is even)            | BO                 |
| <b>Bit-reverse</b>         | [An/An+r]   | A[n]+A[n+1][15:0] (n is even)            | BO                 |

### 9.7.1 Load/Store Basic Data Types

The TriCore architecture defines loads and stores for the basic data types — corresponding to bytes, halfwords, words and doublewords — as well as for signed fractions and addresses. The movement of data between registers and memory for the basic data types is illustrated in Figure 9-17. 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 (except for fractions which are loaded into the most-significant bits of a register), and the remaining register bits are sign- or zero-extended to 32 bits, depending on the particular instruction.

2001-04-30 @ 15:16



**Figure 9-17**

## Load/Store Basic Data Types

### 9.7.2 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

2001-04-30 @ 15:16

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]
EXTR.U    D10,D9,D8,1    ; D10[0] = loaded bit.
```

### 9.7.3 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 (LDMST 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 LDMST 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
ldmst     _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 LDMST 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.
LDMST     [A8],E10
```

### 9.8 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.

2001-04-30 @ 15:16

### 9.8.1 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 D7 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.

### 9.8.2 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 A2-A7 and D0-D7, together with the return address (RA) in A11 and the 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.

## 9.9 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. There are special instructions for the memory-management system. See [Section 7.8, "MMU instructions,"](#) and the cache management system.

### 9.9.1 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

2001-04-30 @ 15:16

constant in the SYSCALL instruction, and serves to identify the specific supervisor service that is being requested.

### 9.9.2 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.

#### 9.9.2.1 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.

#### 9.9.2.2 ISYNC

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

- If an instruction semantically prior to ISYNC makes a software visible change to 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 loop cache 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 state in the pipeline.
4. Refetch the next instruction after the ISYNC.

### 9.9.3 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 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.



2001-04-30 @ 15:16

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.

#### **9.9.4 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) since it is a trap.

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).

#### **9.9.5 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.

#### **9.9.6 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 (see [Arithmetic Instructions](#)).

#### **9.9.7 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.

#### **9.10 16-bit Instructions**

The 16-bit instructions are a subset of the 32-bit instruction set, chosen because of their frequency of static 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

2001-04-30 @ 15:16

address registers, and combined source and destination registers (the 2-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 (base plus zero offset), base plus offset (with implicit base or source/destination register), and post-increment (with default offset) addressing modes.

---

# TriCore Instruction Set

---

2001-04-30 @ 15:16

### 10 TriCore Instruction Set

This chapter contains descriptions of all the TriCore instructions arranged alphabetically by instruction mnemonic. Each instruction page is organized as in [Table 10-1](#).

**Table 10-1**  
**MNEMONIC (Example)**

|                    |  |
|--------------------|--|
| <b>Syntax</b>      | Assembler syntax (See <a href="#">Table 10-2</a> ) followed by the instruction format in parentheses.              |
| <b>Description</b> | A brief verbal description of the instruction's operation  |
| <b>Operation</b>   | A description of the instruction's operation in Register Transfer Language (RTL) (See <a href="#">Table 10-6</a> ) |
| <b>Status</b>      | Any status flags that are affected by the instruction's execution (See <a href="#">Table 10-7</a> )                |
| <b>Examples</b>    | One or more instruction examples   |
| <b>See also</b>    | Related instructions   |

**Note :** Throughout this chapter, information relating to instructions, in particular 16-bit instructions, is shown in gray boxes such as the one below:

The shading of this box indicates that it should contain information about 16-bit instructions.

Note: See the Data Book for instruction format and encoding.

#### 10.1 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 10-2](#) describes the terms used in the syntax definitions.

**Table 10-2**  
**Instruction Syntax Definitions**

| <b>Symbol</b>                | <b>Description</b>   |
|------------------------------|--|
| <b><math>D_n</math></b>      | Data register $n$  |
| <b><math>A_n</math></b>      | Address register $n$   |
| <b><math>E_n</math></b>      | Extended data register $n$ containing a 64-bit value made from an even/odd pair of registers ( $D_n$ , $D_{n+1}$ ) |
| <b><math>disp_n</math></b>   | Displacement value of $n$ bits used to form the effective address in branch instructions.                          |
| <b><math>const_n</math></b>  | Constant value of $n$ bits used as instruction operand   |
| <b><math>offset_n</math></b> | Offset value of $n$ bits used to form the effective address in load and store instructions                         |

2001-04-30 @ 15:16

**Table 10-2  
Instruction Syntax Definitions (cont'd)**

|                     |  |
|---------------------|--|
| <b>p, p1, p2</b>    | Specifies the position of a single bit in bit field instructions       |
| <b>w</b>            | Specifies the width of the bit field in bit and bit field instructions |
| <b>&lt;mode&gt;</b> | An addressing mode   |
| <b>CR</b>           | Core Registers   |

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. 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 (Table 10-3) use a general-purpose register as an implicit source or destination:

**Table 10-3  
Registers Used for 16-bit Instructions**

| <b>Location</b> | <b>Description</b>  |
|-----------------|---|
| <b>D15</b>      | Implicit Data Register for many 16-bit instructions   |
| <b>A10</b>      | Stack Pointer (SP)  |
| <b>A11</b>      | Return Address Register (RA) for CALL, JL, JLA, and JLI instructions, and Return PC value on interrupts |
| <b>A15</b>      | Implicit 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 encoded in the 16-bit instructions.

2001-04-30 @ 15:16

The operation modifiers are shown in [Table 10-4](#). 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 10-4**  
**Operation Modifiers**

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

The data type modifiers used in the instruction mnemonics are listed in [Table 10-5](#). When multiple suffixes occur in an instruction, their order of occurrence in the mnemonic is the same as their order in the table.

**Table 10-5**  
**Data Type Modifiers**

| Data Type Modifier | Name       | Description                       | Example |
|--------------------|------------|-----------------------------------|---------|
| <b>D</b>           | Data       | 32-bit data                       | MOV.D   |
| <b>D</b>           | Doubleword | 64-bit data/address               | LD.D    |
| <b>W</b>           | Word       | 32-bit (word) data                | EQ.W    |
| <b>A</b>           | Address    | 32-bit address                    | ADD.A   |
| <b>Q</b>           | Q Format   | 16-bit signed fraction (Q format) | MADD.Q  |

2001-04-30 @ 15:16

**Table 10-5  
Data Type Modifiers (cont'd)**

|          |          |                                     |         |
|----------|----------|-------------------------------------|---------|
| <b>H</b> | Halfword | 16-bit data or two packed halfwords | ADD.H   |
| <b>B</b> | Byte     | 8-bit data or four packed bytes     | ADD.B   |
| <b>T</b> | Bit      | 1-bit data                          | AND.T   |
| <b>U</b> | Unsigned | Unsigned data type                  | ADD.S.U |

## 10.2 Instruction Operation

The operation of each instruction is described using a C-like Register Transfer Level (RTL) notation, summarized in **Table 10-6**.

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 10-6  
RTL Syntax Description**

| Symbol                   | Description  |
|--------------------------|--|
| <b>D[n]</b>              | Data register n  |
| <b>A[n]</b>              | Address register n   |
| <b>E[n]</b>              | 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) |
| <b>p</b>                 | Single bit p   |
| <b>(expression)[p]</b>   | Single bit p in multi-bit value  |
| <b>n'h pppp</b>          | 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"   |
| <b>n'b pppp</b>          | 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"   |
| <b>{x, y}</b>            | 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}".  |
| <b>[x:y]</b>             | Bits y, y+1, ... , x where x>y; for example D[a][x:y], if x=y then this is a single bit range which is also denoted by [x] as in D[a][x]. For cases where x<y this denotes an empty range.                   |
| <b>disp<sub>n</sub></b>  | Displacement value of n bits used to form the effective address in branch instructions where the constant is either sign-, 0-, or 1-extended   |
| <b>const<sub>n</sub></b> | Constant value of n bits used as instruction operand   |

2001-04-30 @ 15:16

**Table 10-6  
RTL Syntax Description (cont'd)**

| Symbol                      | Description   |
|-----------------------------|---|
| <b>offset<math>n</math></b> | Offset value of $n$ bits used to form the effective address in load and store instructions                      |
| <b>sign_ext</b>             | Sign extension; high-order bit is left extended   |
| <b>one_ext</b>              | One extension; high-order bits are set to 1   |
| <b>zero_ext</b>             | High order bits are set to 0  |
| <b>round16</b>              | The operation of adding 16'h8000 to a 32-bit value and then zeroing the least-significant 16 bits of the result |
| <b>M</b>                    | Memory  |
| <b>EA</b>                   | Effective address   |
| <b>target address</b>       | Address from which next instruction will be fetched. Used in call instructions.                                 |
| <b>M(EA, data_size)</b>     | Memory locations beginning at the specified byte location, EA, and extending to EA+data_size-1                  |
| <b>and</b>                  | Bit-wise logical AND  |
| <b>or</b>                   | Bit-wise logical OR   |
| <b>xor</b>                  | Bit-wise logical exclusive OR   |
| <b>!</b>                    | Logical NOT   |

### 10.3 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 (See [Table 10-7](#)).

**Table 10-7  
PSW Status Flags**

| Status Flag | Description   |
|-------------|---|
| <b>C</b>    | 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 arithmetic shift. |



**Table 10-7**  
**PSW Status Flags (cont'd)**

|            |   |
|------------|---|
| <b>V</b>   | Overflow. This flag is updated by most arithmetic instructions. It is set when the signed 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 $2^{31}-1$ . |
| <b>SV</b>  | 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.  |
| <b>AV</b>  | Advance Overflow. This flag is updated by all instructions that update the overflow flag and no others. This flag is determined as the Boolean exclusive-or of the two most-significant bits of the result.                                   |
| <b>SAV</b> | Sticky Advance Overflow. This flag is set whenever the advance overflow flag is set. It remains set until it is explicitly cleared by an RSTV (Reset Overflow bits) instruction.  |

## 10.4 Instruction Descriptions

**Note :** The instruction mnemonics are in alphabetical order, grouped into families of similar or related instructions (e.g., ABS.B and ABS.H are on the same page).

### 10.4.1 Absolute Value ..... ABS

**Table 10-8**  
**ABS**

|                    |   |
|--------------------|---|
| <b>Syntax</b>      | abs Dc, Da (RR)   |
| <b>Description</b> | Put the absolute value of data register <i>Da</i> in data register <i>Dc</i> ; that is, if the contents of <i>Da</i> are greater than or equal to zero, copy it to <i>Dc</i> ; otherwise, change the sign of <i>Da</i> and copy it to <i>Dc</i> . The operands are treated as signed, 32-bit integers. If <i>Da</i> = 0x80000000 (the maximum negative value), then <i>Dc</i> = 0x80000000, and an overflow is generated. |
| <b>Operation</b>   | if ( <i>D[a]</i> >= 0) then <i>D[c]</i> = <i>D[a]</i><br>else <i>D[c]</i> = $-D[a]$ ; signed  |
| <b>Status</b>      | V, SV, AV, SAV  |
| <b>Examples</b>    | abs d3, d1  |
| <b>See also</b>    | ABSDIF, ABSDIFS, ABSS   |

2001-04-30 @ 15:16

**10.4.2 Absolute Value Packed Byte .....ABS.B**  
**Absolute Value Packed Halfword .....ABS.H**

**Table 10-9**  
**ABS.B & ABS.H**

|                    |   |
|--------------------|---|
| <b>Syntax</b>      | abs.b Dc, Da (RR)<br>abs.h Dc, Da (RR)  |
| <b>Description</b> | Put the absolute value of each byte/halfword in data register <i>Da</i> into the corresponding byte/halfword of data register <i>Dc</i> . 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, which also becomes the result. |
| <b>Operation</b>   | if $(D[a] [(n+7):n] \geq 0)$<br>then $D[c] [(n+7):n] = D[a] [(n+7):n]$<br>else $D[c] [(n+7):n] = -D[a] [(n+7):n]$ ; $n = 0, 8, 16, 24$ ; signed<br><br>if $(D[a] [(n+15):n] \geq 0)$<br>then $D[c] [(n+15):n] = D[a] [(n+15):n]$<br>else $D[c] [(n+15):n] = -D[a] [(n+15):n]$ ; $n = 0, 16$ ; signed  |
| <b>Status</b>      | V, SV, AV, SAV  |
| <b>Examples</b>    | abs.b d3, d1<br>abs.h d3, d1  |
| <b>See also</b>    | ABSS.H, ABSDIF.B, ABSDIF.H, ABSDIFS.H   |

**10.4.3 Absolute Value of Difference ..... ABSDIF**

**Table 10-10**  
**ABSDIF**

|                    |   |
|--------------------|---|
| <b>Syntax</b>      | absdif Dc, Da, Db (RR)<br>absdif Dc, Da, const9 (RC)  |
| <b>Description</b> | Put the absolute value of the difference between <i>Da</i> and <i>Db/const9</i> in <i>Dc</i> ; namely, if the contents of data register <i>Da</i> are greater than <i>Db/const9</i> , then subtract <i>Db/const9</i> from <i>Da</i> and put the result in data register <i>Dc</i> ; otherwise, subtract <i>Da</i> from <i>Db/const9</i> and put the result in <i>Dc</i> . The operands are treated as signed, 32-bit integers, and the <i>const9</i> value is sign-extended to 32 bits. |

**Table 10-10  
ABSDIF**

|                  |   |
|------------------|---|
| <b>Operation</b> | if ( $D[a] > D[b]$ ) then $D[c] = D[a] - D[b]$<br>else $D[c] = D[b] - D[a]$ ; signed<br><br>if ( $D[a] > \text{sign\_ext}(\text{const9})$ ) then $D[c] = D[a] - \text{sign\_ext}(\text{const9})$<br>else $D[c] = \text{sign\_ext}(\text{const9}) - D[a]$ ; signed |
| <b>Status</b>    | V, SV, AV, SAV  |
| <b>Examples</b>  | absdif    d3, d1, d2<br><br>absdif    d3, d1, 126   |
| <b>See also</b>  | ABS, ABSS, ABSDIFS  |

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

**Table 10-11  
ABSDIF.B & ABSDIF.H**

|                    |   |
|--------------------|---|
| <b>Syntax</b>      | absdif.b   Dc, Da, Db (RR)<br>absdif.h   Dc, Da, Db (RR)  |
| <b>Description</b> | Compute the absolute value of the difference between the corresponding bytes/halfwords of <i>Da</i> and <i>Db</i> and put each result in the corresponding byte/halfword of <i>Dc</i> . The operands are treated as signed, 8-bit/16-bit integers. The overflow condition is calculated for each byte/halfword of the packed quantity.  |
| <b>Operation</b>   | if ( $D[a][(n+7):n] > D[b][(n+7):n]$ )<br>then $D[c][(n+7):n] = D[a][(n+7):n] - D[b][(n+7):n]$<br>else $D[c][(n+7):n] = D[b][(n+7):n] - D[a][(n+7):n]$ ; $n = 0, 8, 16, 24$ ; signed<br><br>if ( $D[a][(n+15):n] > D[b][(n+15):n]$ )<br>then $D[c][(n+15):n] = D[a][(n+15):n] - D[b][(n+15):n]$<br>else $D[c][(n+15):n] = D[b][(n+15):n] - D[a][(n+15):n]$ ; $n = 0, 16$ ; signed |
| <b>Status</b>      | V, SV, AV, SAV  |
| <b>Examples</b>    | absdif.b    d3, d1, d2<br><br>absdif.h    d3, d1, d2  |
| <b>See also</b>    | ABS.B, ABS.H, ABSS.H, ABSDIFS.H   |

2001-04-30 @ 15:16

#### 10.4.5 Absolute Value of Difference with Saturation . . . . . ABSDIFS

**Table 10-12**  
**ABSDIFS**

|                    |   |
|--------------------|---|
| <b>Syntax</b>      | absdifs Dc, Da, Db (RR)<br>absdifs Dc, Da, const9 (RC)  |
| <b>Description</b> | Put the absolute value of the difference between <i>Da</i> and <i>Db/const9</i> in <i>Dc</i> : namely, if the contents of data register <i>Da</i> are greater than <i>Db/const9</i> , then subtract <i>Db/const9</i> from <i>Da</i> and put the result in data register <i>Dc</i> ; otherwise, subtract <i>Da</i> from <i>Db/const9</i> and put the result in <i>Dc</i> . The operands are treated as signed, 32-bit integers, with saturation on signed overflow. The <i>const9</i> value is sign-extended to 32 bits. |
| <b>Operation</b>   | if ( $D[a] > D[b]$ ) then $D[c] = D[a] - D[b]$<br>else $D[c] = D[b] - D[a]$ ; signed; ssov<br><br>if ( $D[a] > \text{sign\_ext}(\text{const9})$ ) then $D[c] = D[a] - \text{sign\_ext}(\text{const9})$<br>else $D[c] = \text{sign\_ext}(\text{const9}) - D[a]$ ; signed; ssov   |
| <b>Status</b>      | V, SV, AV, SAV  |
| <b>Examples</b>    | absdifs d3, d1, d2<br>absdifs d3, d1, 126   |
| <b>See also</b>    | ABS, ABSDIF, ABSS   |

#### 10.4.6 Abs Value Diff Packed Halfword w/Saturation . . . . . ABSDIFS.H

**Table 10-13**  
**ABSDIFS.H**

|                    |   |
|--------------------|---|
| <b>Syntax</b>      | absdif.h Dc, Da, Db (RR)  |
| <b>Description</b> | Compute the absolute value of the difference of the corresponding halfwords of <i>Da</i> and <i>Db</i> and put each result in the corresponding halfword of <i>Dc</i> . The operands are treated as signed, 16-bit integers, with saturation on signed overflow. The overflow condition is calculated for each halfword of the packed quantity. |
| <b>Operation</b>   | if ( $D[a][(n+15):n] > D[b][(n+15):n]$ )<br>then $D[c][(n+15):n] = D[a][(n+15):n] - D[b][(n+15):n]$<br>else $D[c][(n+15):n] = D[b][(n+15):n] - D[a][(n+15):n]$ ; $n = 0, 16$ ; signed   |
| <b>Status</b>      | V, SV, AV, SAV  |
| <b>Examples</b>    | absdifs.h d3, d1, d2  |
| <b>See also</b>    | ABS.B, ABS.H, ABSS.H, ABSDIFS.H   |

2001-04-30 @ 15:16

**10.4.7 Absolute Value with Saturation . . . . . ABSS**
**Table 10-14  
ABSS**

|                    |   |
|--------------------|---|
| <b>Syntax</b>      | <code>abss Dc, Da (RR)</code>   |
| <b>Description</b> | Put the absolute value of data register <i>Da</i> in data register <i>Dc</i> ; that is, if the contents of <i>Da</i> are greater than or equal to zero, copy it to <i>Dc</i> ; otherwise, change the sign of <i>Da</i> and copy it to <i>Dc</i> . The operands are treated as signed, 32-bit integers, with saturation on signed overflow. If <i>Da</i> = 0x80000000 (the maximum negative value), then <i>Dc</i> = 0x7FFFFFFF. |
| <b>Operation</b>   | if ( <i>D[a]</i> >= 0) then <i>D[c]</i> = <i>D[a]</i><br>else <i>D[c]</i> = - <i>D[a]</i> ; signed; ssov  |
| <b>Status</b>      | V, SV, AV, SAV  |
| <b>Examples</b>    | <code>abss d3, d1</code>  |
| <b>See also</b>    | ABS, ABSDIF, ABSDIFS  |

**10.4.8 Absolute Value Packed Halfword w/ Saturation. . . . . ABSS.H**
**Table 10-15  
ABSS.H**

|                    |   |
|--------------------|---|
| <b>Syntax</b>      | <code>abss.h Dc, Da (RR)</code>   |
| <b>Description</b> | Put the absolute value of each byte/halfword in data register <i>Da</i> in the corresponding byte/halfword of data register <i>Dc</i> . 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 <i>D[a] [(n+15):n]</i> has the maximum negative value of 0x8000, and the saturation yields 0x7FFF. |
| <b>Operation</b>   | if ( <i>D[a] [(n+15):n]</i> >= 0)<br>then <i>D[c] [(n+15):n]</i> = <i>D[a] [(n+15):n]</i><br>else <i>D[c] [(n+15):n]</i> = - <i>D[a] [(n+15):n]</i> ; n = 0, 16; signed; ssov   |
| <b>Status</b>      | V, SV, AV, SAV  |
| <b>Examples</b>    | <code>abss.h d3, d1</code>  |
| <b>See also</b>    | ABS.B, ABS.H, ABSDIF.B, ABSDIF.H, ABSDIFS.H   |

2001-04-30 @ 15:16

10.4.9 Add..... ADD

Table 10-16  
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) |
|                                       | add  | Da, D15, Db (SRR)     |
| Description                           | add  | Da, D15, const4(SRC)  |
|                                       |  |                       |
| Operation                             | Add the contents of data register <i>Da</i> to the contents of data register <i>Db/const9</i> and put the result in data register <i>Dc</i> . The operands are treated as 32-bit integers, and the <i>const9</i> value is sign-extended to 32 bits before the addition is performed.         |                       |
|                                       | Add the contents of data register <i>Da/D15</i> to the contents of data register <i>Db/const4</i> and put the result in data register <i>Da/D15</i> . The operands are treated as 32-bit integers, and the <i>const4</i> value is sign-extended to 32 bits before the addition is performed. |                       |
|                                       | $D[c] = D[a] + D[b]$   |                       |
|                                       | $D[c] = D[a] + \text{sign\_ext}(\text{const9})$  |                       |
|                                       | $D[a] = D[a] + D[b]$   |                       |
|                                       | $D[a] = D[a] + \text{sign\_ext}(\text{const4})$  |                       |
|                                       | $D[15] = D[a] + D[b]$  |                       |
| Status                                | $D[15] = D[a] + \text{sign\_ext}(\text{const4})$   |                       |
|                                       | $D[a] = D[15] + D[b]$  |                       |
| Examples                              | $D[a] = D[15] + \text{sign\_ext}(\text{const4})$   |                       |
|                                       | V, SV, AV, SAV   |                       |
|                                       | add  | d3, d1, d2            |
|                                       | add  | d3, d1, 126           |
|                                       | add  | d1, d2                |
|                                       | add  | d1, 6                 |
|                                       | add  | d15, d1, d2           |
| See also                              | add  | d15, d1, 6            |
|                                       | add  | d1, d15, d2           |
|                                       | add  | d1, d15, 6            |
|                                       | add  | d1, d15, 6            |
| ADDC, ADDI, ADDIH, ADDS, ADDS.U, ADDX |  |                       |

2001-04-30 @ 15:16

**10.4.10 Add Address ..... ADD.A**
**Table 10-17**
**ADD.A**

|                    |  |
|--------------------|--|
| <b>Syntax</b>      | add.a <i>Ac</i> , <i>Aa</i> , <i>Ab</i> (RR)   |
|                    | add.a <i>Aa</i> , <i>Ab</i> (SRR)  |
|                    | add.a <i>Aa</i> , const4 (SRC)   |
| <b>Description</b> | Add the contents of address register <i>Aa</i> to the contents of address register <i>Ab</i> and put the result in address register <i>Ac</i> .        |
|                    | Add the contents of address register <i>Aa</i> to the contents of address register <i>Ab/const4</i> and put the result in address register <i>Aa</i> . |
| <b>Operation</b>   | $A[c] = A[a] + A[b]$   |
|                    | $A[a] = A[a] + A[b]$   |
|                    | $A[a] = A[a] + \text{sign\_ext}(\text{const4})$  |
| <b>Status</b>      | -  |
| <b>Examples</b>    | add.a    a3, a4, a2  |
|                    | add.a    a1, a2  |
|                    | add.a    a3, 6   |
| <b>See also</b>    | ADDIH.A, ADDSC.A, ADDSC.AT   |

**10.4.11 Add Packed Byte ..... ADD.B**
**Add Packed Halfword ..... ADD.H**
**Table 10-18**
**ADD.B & ADD.H**

|                    |   |
|--------------------|---|
| <b>Syntax</b>      | add.b <i>Dc</i> , <i>Da</i> , <i>Db</i> (RR)  |
|                    | add.h <i>Dc</i> , <i>Da</i> , <i>Db</i> (RR)  |
| <b>Description</b> | Add the contents of each byte/halfword of <i>Da</i> and <i>Db</i> and put the result in each corresponding byte/halfword of <i>Dc</i> . 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. |
| <b>Operation</b>   | $D[c][(n+7):n] = D[a][(n+7):n] + D[b][(n+7):n]; n = 0, 8, 16, 24$<br>$D[c][(n+15):n] = D[a][(n+15):n] + D[b][(n+15):n]; n = 0, 16$  |
| <b>Status</b>      | V, SV, AV, SAV  |
| <b>Examples</b>    | add.b    d3, d1, d2   |
|                    | add.h    d3, d1, d2   |
| <b>See also</b>    | ADD.H, ADDS.H, ADDS.HU  |

2001-04-30 @ 15:16

### 10.4.12 Add with Carry. .... ADDC

**Table 10-19**  
**ADDC**

|                    |   |
|--------------------|---|
| <b>Syntax</b>      | addc Dc, Da, Db (RR)<br>addc Dc, Da, const9 (RC)  |
| <b>Description</b> | Add the contents of data register <i>Da</i> to the contents of data register <i>Db/const9</i> plus the carry bit and put the result in data register <i>Dc</i> . The operands are treated as 32-bit integers, and the value <i>const9</i> 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. |
| <b>Operation</b>   | $D[c] = D[a] + D[b] + \text{PSW.C}; \text{PSW.C} = \text{carry\_out}$<br>$D[c] = D[a] + \text{sign\_ext}(\text{const9}) + \text{PSW.C}; \text{PSW.C} = \text{carry\_out}$   |
| <b>Status</b>      | C, V, SV, AV, SAV   |
| <b>Examples</b>    | addc d3, d1, d2<br>addc d3, d1, 126   |
| <b>See also</b>    | ADD, ADDI, ADDIH, ADDS, ADDS.U, ADDX  |

### 10.4.13 Add Immediate .... ADDI

**Table 10-20**  
**ADDI**

|                    |   |
|--------------------|---|
| <b>Syntax</b>      | addi Dc, Da, const16 (RLC)  |
| <b>Description</b> | Add the contents of data register <i>Da</i> to the value <i>const16</i> , and put the result in data register <i>Dc</i> . The operands are treated as 32-bit integers. The value <i>const16</i> is sign-extended to 32 bits before the addition is performed. |
| <b>Operation</b>   | $D[c] = D[a] + \text{sign\_ext}(\text{const16})$  |
| <b>Status</b>      | V, SV, AV, SAV  |
| <b>Examples</b>    | addi d3, d1, -14526   |
| <b>See also</b>    | ADD, ADDC, ADDIH, ADDS, ADDS.U, ADDX  |

### 10.4.14 Add Immediate High .... ADDIH

**Table 10-21**  
**ADDIH**

|                    |   |
|--------------------|---|
| <b>Syntax</b>      | addih Dc, Da, const16 (RLC)   |
| <b>Description</b> | Left-shift <i>const16</i> by 16 bits, add the contents of data register <i>Da</i> , and put the result in data register <i>Dc</i> . |



**Table 10-21**
**ADDIH**

|                  |  |
|------------------|--|
| <b>Operation</b> | $D[c] = D[a] + \{\text{const16}, 16'h\ 0000\}$ |
| <b>Status</b>    | V, SV, AV, SAV                                 |
| <b>Examples</b>  | addih d3, d1, -14526                           |
| <b>See also</b>  | ADD, ADDC, ADDI, ADDS, ADDS.U, ADDX            |

**10.4.15 Add Immediate High to Address . . . . . ADDIH.A**
**Table 10-22**
**ADDIH.A**

|                    |   |
|--------------------|---|
| <b>Syntax</b>      | addih.a Ac, Aa, const16 (RLC)   |
| <b>Description</b> | Left-shift <i>const16</i> by 16 bits, add the contents of address register <i>Aa</i> , and put the result in address register <i>Ac</i> . |
| <b>Operation</b>   | $A[c] = A[a] + \{\text{const16}, 16'h\ 0000\}$  |
| <b>Status</b>      | -   |
| <b>Examples</b>    | addih.a a3, a4, -14526  |
| <b>See also</b>    | ADD.A, ADDSC.A, ADDSC.AT  |

**10.4.16 Add Signed with Saturation . . . . . ADDS**
**Table 10-23**
**ADDS**

|                    |   |
|--------------------|---|
| <b>Syntax</b>      | adds Dc, Da, Db (RR)  |
|                    | adds Dc, Da, const9 (RC)  |
|                    | adds Da, Db (SRR)   |
| <b>Description</b> | Add the contents of data register <i>Da</i> to the value in data register <i>Db/const9</i> and put the result in data register <i>Dc</i> . The operands are treated as signed, 32-bit integers, with saturation on signed overflow. The value <i>const9</i> is sign-extended to 32 bits before the addition is performed. |
|                    | Add the contents of data register <i>Db</i> to the contents of data register <i>Da</i> and put the result in data register <i>Da</i> . The operands are treated as signed 32-bit integers, with saturation on signed overflow.  |
| <b>Operation</b>   | $D[c] = D[a] + D[b]; \text{signed}; \text{ssov}$  |
|                    | $D[c] = D[a] + \text{sign\_ext}(\text{const9}); \text{signed}; \text{ssov}$   |
|                    | $D[a] = D[a] + D[b]; \text{signed}; \text{ssov}$  |

2001-04-30 @ 15:16

**Table 10-23**  
**ADDS (cont'd)**

|                 |  |
|-----------------|--|
| <b>Status</b>   | V, SV, AV, SA  |
| <b>Examples</b> | <pre>adds    d3, d1, d2 adds    d3, d1, 126 adds    d3, d1</pre> |
| <b>See also</b> | ADD, ADDC, ADDI, ADDIH, ADDS.U, ADDX                             |

**10.4.17 Add Signed Packed Halfword with Saturatio . . . . . nADDS.H**  
**Add Unsigned Packed Halfword w/ Saturation . . . . . ADDS.HU**

**Table 10-24**  
**ADDS.H & ADDS.HU**

|                    |  |
|--------------------|--|
| <b>Syntax</b>      | <pre>adds.h  Dc, Da, Db (RR) adds.hu Dc, Da, Db (RR)</pre>   |
| <b>Description</b> | Add the contents of each halfword of <i>Da</i> and <i>Db</i> and put the result in each corresponding halfword of <i>Dc</i> , with saturation on signed/unsigned overflow. The overflow and advance overflow conditions are calculated for each halfword of the packed quantity. |
| <b>Operation</b>   | $D[c][(n+15):n] = D[a][(n+15):n] + D[b][(n+15):n]; n = 0, 16; \text{signed}; \text{ssov}$ $D[c][(n+15):n] = D[a][(n+15):n] + D[b][(n+15):n]; n = 0, 16; \text{unsigned}; \text{suov}$  |
| <b>Status</b>      | V, SV, AV, SAV   |
| <b>Examples</b>    | <pre>adds.h  d3, d1, d2 adds.hu d3, d1, d2</pre>   |
| <b>See also</b>    | ADD.B, ADD.H,  |

**10.4.18 Add Unsigned with Saturation. . . . . ADDS.U**

**Table 10-25**  
**ADDS.U**

|                    |   |
|--------------------|---|
| <b>Syntax</b>      | <pre>adds.u  Dc, Da, Db (RR) adds.u  Dc, Da, const9 (RC)</pre>  |
| <b>Description</b> | Add the contents of data register <i>Da</i> to the contents of data register <i>Db/const9</i> and put the result in data register <i>Dc</i> . The operands are treated as unsigned, 32-bit integers, with saturation on unsigned overflow. The <i>const9</i> value is zero-extended to 32 bits. |

**Table 10-25**
**ADDS.U**

|                  |   |
|------------------|---|
| <b>Operation</b> | $D[c] = D[a] + D[b]$ ; unsigned; suov<br>$D[c] = D[a] + \text{zero\_ext}(\text{const9})$ ; unsigned; suov |
| <b>Status</b>    | V, SV, AV, SAV  |
| <b>Examples</b>  | <code>adds.u d3, d1, d2</code><br><code>adds.u d3, d1, 126</code>   |
| <b>See also</b>  | ADD, ADDC, ADDI, ADDIH, ADDS, ADDX  |

**10.4.19 Add Scaled Index to Address ..... ADDSC.A**  
**Add Bit-Scaled Index to Address ..... ADDSC.AT**
**Table 10-26**
**ADDSC.A & ADDSC.AT**

|                    |   |
|--------------------|---|
| <b>Syntax</b>      | <code>addsc.a Ac, Ab, Da, n (RRS)</code><br><code>addsc.at Ac, Ab, Da (RRS)</code><br><code>addsc.a Aa, Ab, D15, n (SRRS)</code>  |
| <b>Description</b> | Left-shift the contents of data register <i>Da</i> by the amount specified by <i>n</i> , where <i>n</i> can be 0, 1, 2, or 3. Add that value to the contents of address register <i>Ab</i> and put the result in address register <i>Ac</i> .<br><br>Right-shift the contents of <i>Da</i> by 3 (with sign fill). Add that value to the contents of address register <i>Ab</i> and clear the bottom two bits to zero. Put the result in <i>Ac</i> .<br>The instruction ADDSC.AT generates the address of the word containing the bit indexed by <i>Db</i> , starting from the base address in <i>Aa</i> .<br><br>Left-shift the contents of data register <i>D15</i> by the amount specified by <i>n</i> , where <i>n</i> can be 0, 1, 2, or 3. Add that value to the contents of address register <i>Ab</i> and put the result in address register <i>Aa</i> . |
| <b>Operation</b>   | $A[c] = A[b] + (D[a] \ll n)$ , $n = 0, 1, 2, \text{ or } 3$<br>$A[c] = (A[b] + (D[a] \gg 3)) \text{ and } !2'b\ 11$<br>$A[a] = (A[b] + (D[15] \ll n))$ , $n = 0, 1, 2, \text{ or } 3$   |
| <b>Status</b>      | -   |
| <b>Examples</b>    | <code>addsc.a a3, a4, d2, 2</code><br><code>addsc.at a3, a4, d2</code><br><code>addsc.a a3, a4, d15, 2</code>   |
| <b>See also</b>    | ADD.A, ADDIH.A, SUB.A   |

2001-04-30 @ 15:16

#### 10.4.20 Add Extended ..... ADDX

**Table 10-27**  
**ADDX**

|                    |  |
|--------------------|--|
| <b>Syntax</b>      | addx Dc, Da, Db (RR)<br>addx Dc, Da, const9 (RC)   |
| <b>Description</b> | Add the contents of data register <i>Da</i> to the contents of data register <i>Db/const9</i> and put the result in data register <i>Dc</i> . The operands are treated as 32-bit integers, and the <i>const9</i> 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. |
| <b>Operation</b>   | $D[c] = D[a] + D[b]$ ; PSW.C = carry_out<br>$D[c] = D[a] + \text{sign\_ext}(\text{const9})$ ; PSW.C = carry_out  |
| <b>Status</b>      | C, V, SV, AV, SAV  |
| <b>Examples</b>    | addx d3, d1, d2<br>addx d3, d1, 126  |
| <b>See also</b>    | ADD, ADDC, ADDI, ADDIH, ADDS, ADDS.U   |

#### 10.4.21 Logical ..... ANDAND

**Table 10-28**  
**AND**

|                    |  |
|--------------------|--|
| <b>Syntax</b>      | and Dc, Da, Db (RR)<br>and Dc, Da, const9 (RC)<br>and Da, Db (SRR)<br>and D15, const8 (SC)   |
| <b>Description</b> | Compute the bitwise logical AND of the contents of data register <i>Da</i> and the contents of data register <i>Db/const9</i> and put the result in data register <i>Dc</i> . The <i>const9</i> value is zero-extended to 32 bits.<br>Compute the bitwise logical AND of the contents of data register <i>Da/D15</i> and the contents of data register <i>Db/const8</i> and put the result in data register <i>Da/D15</i> . The <i>const8</i> value is zero-extended to 32 bits. |
| <b>Operation</b>   | $D[c] = D[a] \text{ AND } D[b]$<br>$D[c] = D[a] \text{ AND zero\_ext}(\text{const9})$<br>$D[a] = D[a] \text{ AND } D[b]$<br>$D[15] = D[15] \text{ AND zero\_ext}(\text{const8})$   |

**Table 10-28  
AND (cont'd)**

|                 |   |
|-----------------|---|
| <b>Status</b>   | -   |
| <b>Examples</b> | and d3, d1, d2<br>and d3, d1, 126<br>and d1, d2<br>and d15, 126 |
| <b>See also</b> | ANDN, NAND, NOR, NOT, OR, ORN, XNOR, XOR                        |

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

**Table 10-29  
AND.AND.T, AND.ANDN.T, AND.NOR.T & AND.OR.T**

|                    |  |
|--------------------|--|
| <b>Syntax</b>      | and.and.t Dc, Da, p1, Db, p2 (BIT)<br>and.andn.t Dc, Da, p1, Db, p2 (BIT)<br>and.nor.t Dc, Da, p1, Db, p2 (BIT)<br>and.or.t Dc, Da, p1, Db, p2 (BIT)   |
| <b>Description</b> | Compute the logical AND/ANDN/NOR/OR of the value of bit <i>p1</i> of data register <i>Da</i> and bit <i>p2</i> of <i>Db</i> . Then compute the logical AND of that result and bit 0 of <i>Dc</i> , and put the result back in bit 0 of <i>Dc</i> . All other bits in <i>Dc</i> are unchanged.  |
| <b>Operation</b>   | $D[c] = \{D[c][31:1], D[c][0] \text{ AND } (D[a][p1] \text{ AND } D[b][p2])\}$<br>$D[c] = \{D[c][31:1], D[c][0] \text{ AND } (D[a][p1] \text{ AND } !D[b][p2])\}$<br><br>$D[c] = \{D[c][31:1], D[c][0] \text{ AND } !(D[a][p1] \text{ OR } D[b][p2])\}$<br>$D[c] = \{D[c][31:1], D[c][0] \text{ AND } (D[a][p1] \text{ OR } D[b][p2])\}$ |
| <b>Status</b>      | -  |
| <b>Examples</b>    | and.and.t d3, d1, 4, d2, 9<br>and.andn.t d3, d1, 6, d2, 15<br>and.nor.t d3, d1, 5, d2, 9<br>and.or.t d3, d1, 4, d2, 6  |
| <b>See also</b>    | OR.AND.T, OR.ANDN.T, OR.NOR.T, OR.OR.T, SH.AND.T, SH.ANDN.T, SH.NAND.T, SH.NOR.T, SH.OR.T, SH.ORN.T, SH.XNOR.T, SH.XOR.T   |

2001-04-30 @ 15:16

**10.4.23 Equal Accumulating . . . . . AND.EQ**

**Table 10-30**  
**AND.EQ**

|                    |   |
|--------------------|---|
| <b>Syntax</b>      | and.eq Dc, Da, Db (RR)<br>and.eq Dc, Da, const9 (RC)  |
| <b>Description</b> | 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. |
| <b>Operation</b>   | $D[c] = \{D[c][31:1], D[c][0] \text{ AND } (D[a]==D[b])\}$<br><br>$D[c] = \{D[c][31:1], D[c][0] \text{ AND } (D[a]==\text{sign\_ext}(const9))\}$  |
| <b>Status</b>      | -   |
| <b>Examples</b>    | and.eq d3, d1, d2<br>and.eq d3, d1, 126   |
| <b>See also</b>    | OR.EQ, XOR.EQ   |

**10.4.24 Greater Than or Equal Accumulating . . . . . AND.GE**  
**Greater Than or Equal Accumulating Unsigned . . . . . AND.GE.U**

**Table 10-31**  
**AND.GE & AND.GE.U**

|                    |  |
|--------------------|--|
| <b>Syntax</b>      | and.ge Dc, Da, Db (RR)<br>and.ge Dc, Da, const9 (RC)<br>and.ge.uDc, Da, Db (RR)<br>and.ge.uDc, Da, const9 (RC)   |
| <b>Description</b> | 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.<br><br>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. |

**Table 10-31**  
**AND.GE & AND.GE.U (cont'd)**

|                  |  |
|------------------|--|
| <b>Operation</b> | $D[c] = \{D[c][31:1], D[c][0] \text{ AND } (D[a] \geq D[b])\}$ ; signed<br>$D[c] = \{D[c][31:1], D[c][0] \text{ AND } (D[a] \geq \text{sign\_ext}(\text{const9}))\}$ ; signed<br><br>$D[c] = \{D[c][31:1], D[c][0] \text{ AND } (D[a] \geq D[b])\}$ ; unsigned<br>$D[c] = \{D[c][31:1], D[c][0] \text{ AND } (D[a] \geq \text{zero\_ext}(\text{const9}))\}$ ; unsigned |
| <b>Status</b>    | -  |
| <b>Examples</b>  | <pre>and.ge    d3, d1, d2 and.ge    d3, d1, 126 and.ge.u  d3, d1, d2 and.ge.u  d3, d1, 126</pre>   |
| <b>See also</b>  | OR.GE, OR.GE.U, XOR.GE, XOR.GE.U   |

#### 10.4.25 Less Than Accumulating .....AND.LT Less Than Accumulating Unsigned .....AND.LT.U

**Table 10-32**  
**AND.LT & AND.LT.U**

|                    |   |
|--------------------|---|
| <b>Syntax</b>      | <pre>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)</pre>  |
| <b>Description</b> | <p>Calculate the logical AND of <math>Dc[0]</math> and the Boolean result of the LT operation on the contents of data register <math>Da</math> and data register <math>Db/\text{const9}</math>. Put the result in <math>Dc[0]</math>. All other bits in <math>Dc</math> are unchanged. <math>Da</math> and <math>Db</math> are treated as 32-bit signed integers. The <math>\text{const9}</math> value is sign-extended to 32 bits.</p> <p>Calculate the logical AND of <math>Dc[0]</math> and the Boolean result of the LT.U operation on the contents of data register <math>Da</math> and data register <math>Db/\text{const9}</math>. Put the result in <math>Dc[0]</math>. All other bits in <math>Dc</math> are unchanged. <math>Da</math> and <math>Db</math> are treated as 32-bit unsigned integers. The <math>\text{const9}</math> value is zero-extended to 32 bits.</p> |

2001-04-30 @ 15:16

**Table 10-32**  
**AND.LT & AND.LT.U**

|                  |  |
|------------------|--|
| <b>Operation</b> | $D[c] = \{D[c][31:1], D[c][0] \text{ AND } (D[a] < D[b])\}$ ; signed<br>$D[c] = \{D[c][31:1], D[c][0] \text{ AND } (D[a] < \text{sign\_ext}(\text{const9}))\}$ ; signed<br><br>$D[c] = \{D[c][31:1], D[c][0] \text{ AND } (D[a] < D[b])\}$ ; unsigned<br>$D[c] = \{D[c][31:1], D[c][0] \text{ AND } (D[a] < \text{zero\_ext}(\text{const9}))\}$ ; unsigned |
| <b>Status</b>    | -  |
| <b>Examples</b>  | <pre>and.lt    d3, d1, d2 and.lt    d3, d1, 126 and.lt.u  d3, d1, d2 and.lt.u  d3, d1, 126</pre>   |
| <b>See also</b>  | OR.LT, OR.LT.U, XOR.LT, XOR.LT.U   |

#### 10.4.26 Not Equal Accumulating. . . . . AND.NE

**Table 10-33**  
**AND.NE**

|                    |   |
|--------------------|---|
| <b>Syntax</b>      | <pre>and.ne  Dc, Da, Db (RR) and.ne  Dc, Da, const9 (RC)</pre>  |
| <b>Description</b> | 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 $Db/\text{const9}$ . Put the result in $Dc[0]$ . All other bits in $Dc$ are unchanged. The $\text{const9}$ value is sign-extended to 32 bits. |
| <b>Operation</b>   | $D[c] = \{D[c][31:1], D[c][0] \text{ AND } (D[a] \neq D[b])\}$<br>$D[c] = \{D[c][31:1], D[c][0] \text{ AND } (D[a] \neq \text{sign\_ext}(\text{const9}))\}$   |
| <b>Status</b>      | -   |
| <b>Examples</b>    | <pre>and.ne    d3, d1, d2 and.ne    d3, d2, 126</pre>   |
| <b>See also</b>    | OR.NE, XOR.NE   |

#### 10.4.27 Bit Logical AND. . . . . AND.T

**Table 10-34**  
**AND.T**

|                    |  |
|--------------------|--|
| <b>Syntax</b>      | <pre>and.t    Dc, Da, p1, Db, p2 (BIT)</pre>   |
| <b>Description</b> | 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. |



**Table 10-34**
**AND.T**

|                  |   |
|------------------|---|
| <b>Operation</b> | $D[c] = D[a][p1] \text{ AND } D[b][p2]$           |
| <b>Status</b>    | -   |
| <b>Examples</b>  | <code>and.t d3, d1, 7, d2, 2</code>               |
| <b>See also</b>  | ANDN.T, NAND.T, NOR.T, OR.T, ORN.T, XNOR.T, XOR.T |

**10.4.28 AND-Not . . . . . ANDN**
**Table 10-35**
**AND.N**

|                    |   |
|--------------------|---|
| <b>Syntax</b>      | <code>andn Dc, Da, Db (RR)</code><br><code>andn Dc, Da, const9 (RC)</code>  |
| <b>Description</b> | Compute the bitwise logical AND of the contents of data register <i>Da</i> and the ones-complement of the contents of data register <i>Db/const9</i> and put the result in data register <i>Dc</i> . The <i>const9</i> value is zero-extended to 32 bits. |
| <b>Operation</b>   | $D[c] = D[a] \text{ AND } !D[b]$<br>$D[c] = D[a] \text{ AND } !\text{zero\_ext}(\text{const9})$   |
| <b>Status</b>      | -   |
| <b>Examples</b>    | <code>andn d3, d1, d2</code><br><code>andn d3, d1, 126</code>   |
| <b>See also</b>    | AND, NAND, NOR, NOT, OR, ORN, XNOR, XOR   |

**10.4.29 Bit Logical AND-Not . . . . . ANDN.T**
**Table 10-36**
**ANDN.T**

|                    |   |
|--------------------|---|
| <b>Syntax</b>      | <code>andn.t Dc, Da, p1, Db, p2 (BIT)</code>  |
| <b>Description</b> | Compute the logical AND of bit <i>p1</i> of data register <i>Da</i> and the inverse of bit <i>p2</i> of data register <i>Db</i> . Put the result in the least-significant bit of data register <i>Dc</i> and clear the remaining bits of <i>Dc</i> to zero. |
| <b>Operation</b>   | $D[c] = D[a][p1] \text{ AND } !D[b][p2]$  |
| <b>Status</b>      | -   |
| <b>Examples</b>    | <code>andn.t d3, d1, 2, d2, 5</code>  |
| <b>See also</b>    | AND.T, NAND.T, NOR.T, OR.T, ORN.T, XNOR.T, XOR.T  |

2001-04-30 @ 15:16

### 10.4.30 Begin ISR ..... BISR

**Table 10-37**  
**BISR**

|                    |   |
|--------------------|---|
| <b>Syntax</b>      | bisr    const9 (RC)   |
|                    | bisr    const8 (SC)   |
| <b>Description</b> | <p>Save the lower context by storing contents of A2 – A7, D0 – D7, and current return address (A11) to the current memory location pointed to by the FCX. Set the current CPU priority number (ICR.CCPN) to the value of <i>const9</i>[7:0]/<i>const8</i>, and enable interrupts (set ICR.IE to one). Note that BISR can be executed only in supervisor privilege mode.</p> <p>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.</p> <p>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.</p> |
| <b>Operation</b>   | <p>Save lower context;<br/>ICR.IE = 1<br/>ICR.CCPN = const9[7:0]</p>  |
|                    | <p>Save lower context;<br/>ICR.IE = 1<br/>ICR.CCPN = const8</p>   |
| <b>Status</b>      | -   |
| <b>Examples</b>    | bisr    126   |
|                    | bisr    126   |
| <b>See also</b>    | DISABLE, ENABLE, LDLCX, LDUCX, RET, RFE, RSLCX, STLCX, STUCX, SVLCX   |

#### 10.4.31 Cache Address, Invalidate . . . . . CACHEA.I

Table 10-38

CACHEA.I

|                    |  |
|--------------------|--|
| <b>Syntax</b>      | cachea.i <mode>  |
| <b>Description</b> | <p>If the cache line containing the memory location specified by the addressing mode is present in the L1 data cache invalidate the line. Note no write back is performed of any dirty data in the cache line prior to the invalidation.</p> <p>If the cache line containing the memory location specified by the addressing mode is not present in the L1 data cache then no operation should be performed in the L1 data cache. Specifically a refill of the line containing the byte pointed to by the effective address should not be performed.</p> <p>Note any address register updates associated with the addressing mode are always performed regardless of the cache operation.</p> <p>This instruction can be executed in supervisor mode only.</p> |
| <b>Operation</b>   | See Table 10-39.   |
| <b>Status</b>      | -  |
| <b>Examples</b>    | <pre>cachea.i    [a3]4 cachea.i    [+a3]4 cachea.i    [a3+]4 cachea.i    [a3/a4+c]4 cachea.i    [a3/a4+r]</pre>  |
| <b>See also</b>    | CACHEA.W, CACHEA.WI  |

Table 10-39

CACHEA.I Operation

| <mode>                     | Syntax      | Effective Address             | Instruction Format |
|----------------------------|-------------|-------------------------------|--------------------|
| <b>Base + Short Offset</b> | [An]offset  | A[b]+sign_ext(offset10)       | BO                 |
| <b>Pre-increment</b>       | [+An]offset | A[b]+sign_ext(offset10)       | BO                 |
| <b>Post-increment</b>      | [An+]offset | A[b]                          | BO                 |
| <b>Circular</b>            | [An/An+1+c] | A[b]+A[b+1][15:0] (b is even) | BO                 |
| <b>Bit-reverse</b>         | [An/An+r]   | A[b]+A[b+1][15:0] (b is even) | BO                 |

2001-04-30 @ 15:16

#### 10.4.32 Cache Address, Writeback. . . . . CACHEA.W

**Table 10-40**  
**CACHEA.W**

|                    |   |
|--------------------|---|
| <b>Syntax</b>      | cachea.w <mode>   |
| <b>Description</b> | <p>If the cache line containing the memory location specified by the addressing mode is present in the L1 data cache, write back any modified data. The line will still be present in the L1 data cache and will be marked as unmodified.</p> <p>If the cache line containing the memory location specified by the addressing mode is not present in the L1 data cache then no operation should be performed in the L1 data cache. Specifically a refill of the line containing the byte pointed to by the effective address should not be performed.</p> <p>Note any address register updates associated with the addressing mode are always performed regardless of the cache operation.</p> <p>This instruction can be executed in supervisor mode only.</p> |
| <b>Operation</b>   | See <a href="#">Table 10-41</a>   |
| <b>Status</b>      | -   |
| <b>Examples</b>    | <pre>cachea.w    [a3]4 cachea.w    [+a3]4 cachea.w    [a3+]4 cachea.w    [a3/a4+c]4 cachea.w    [a3/a4+r]</pre>   |
| <b>See also</b>    | CACHEA.I, CACHEA.WI   |

**Table 10-41**  
**CACHEA.W Operation**

| <mode>                     | Syntax      | Effective Address             | Instruction Format |
|----------------------------|-------------|-------------------------------|--------------------|
| <b>Base + Short Offset</b> | [An]offset  | A[b]+sign_ext(offset10)       | BO                 |
| <b>Pre-increment</b>       | [+An]offset | A[b]+sign_ext(offset10)       | BO                 |
| <b>Post-increment</b>      | [An+]offset | A[b]                          | BO                 |
| <b>Circular</b>            | [An/An+1+c] | A[b]+A[b+1][15:0] (b is even) | BO                 |
| <b>Bit-reverse</b>         | [An/An+r]   | A[b]+A[b+1][15:0] (b is even) | BO                 |

**10.4.33 Cache Address, Writeback and Invalidate . . . . . CACHEA.WI**
**Table 10-42  
CACHEA.WI**

|                    |   |
|--------------------|---|
| <b>Syntax</b>      | cachea.wi <mode>  |
| <b>Description</b> | <p>If the cache line containing the memory location specified by the addressing mode is present in the L1 data cache, write back any modified data and then invalidate the line in the L1 data cache.</p> <p>If the cache line containing the memory location specified by the addressing mode is not present in the L1 data cache then no operation should be performed in the L1 data cache. Specifically a refill of the line containing the byte pointed to by the effective address should not be performed.</p> <p>Note any address register updates associated with the addressing mode are always performed regardless of the cache operation.</p> <p>This instruction can be executed in supervisor mode only.</p> |
| <b>Operation</b>   | See <a href="#">Table 10-43</a> .   |
| <b>Status</b>      | -   |
| <b>Examples</b>    | <pre>cachea.wi    [a3]4 cachea.wi    [+a3]4 cachea.wi    [a3+]4 cachea.wi    [a3/a4+c]4 cachea.wi    [a3/a4+r]</pre>  |
| <b>See also</b>    | CACHEA.I, CACHEA.W  |

**Table 10-43  
CACHEA.WI Operation**

| <mode>                     | Syntax      | Effective Address             | Instruction Format |
|----------------------------|-------------|-------------------------------|--------------------|
| <b>Base + Short Offset</b> | [An]offset  | A[b]+sign_ext(offset10)       | BO                 |
| <b>Pre-increment</b>       | [+An]offset | A[b]+sign_ext(offset10)       | BO                 |
| <b>Post-increment</b>      | [An+]offset | A[b]                          | BO                 |
| <b>Circular</b>            | [An/An+1+c] | A[b]+A[b+1][15:0] (b is even) | BO                 |
| <b>Bit-reverse</b>         | [An/An+r]   | A[b]+A[b+1][15:0] (b is even) | BO                 |

2001-04-30 @ 15:16

#### 10.4.34 Conditional Add ..... CADD

**Table 10-44**  
**CADD**

|                    |   |
|--------------------|---|
| <b>Syntax</b>      | cadd Dc, Dd, Da, Db (RRR)   |
|                    | cadd Dc, Dd, Da, const9 (RCR)   |
|                    | cadd Da, D15, const4 (SCR)  |
| <b>Description</b> | If contents of data register <i>Dd</i> are non-zero, add contents of data register <i>Da</i> and contents of register <i>Db/const9</i> and put the result in data register <i>Dc</i> ; otherwise, put contents of <i>Da</i> in <i>Dc</i> . The <i>const9</i> value is sign-extended to 32 bits. |
|                    | If contents of data register D15 are non-zero, add contents of data register <i>Da</i> and the contents of <i>const4</i> and put the result in data register <i>Da</i> ; otherwise, the contents of <i>Da</i> is unchanged. The <i>const4</i> value is sign-extended to 32 bits.                |
| <b>Operation</b>   | $D[c] = ((D[d] \neq 0) ? D[a] + D[b] : D[a])$   |
|                    | $D[c] = ((D[d] \neq 0) ? D[a] + \text{sign\_ext}(\text{const9}) : D[a])$  |
|                    | $D[a] = ((D[15] \neq 0) ? D[a] + \text{sign\_ext}(\text{const4}) : D[a])$   |
| <b>Status</b>      | V, SV, AV, SAV  |
| <b>Examples</b>    | cadd d3, d4, d1, d2   |
|                    | cadd d3, d4, d1, 126  |
|                    | cadd d1, d15, 6   |
| <b>See also</b>    | CADDN, CMOV, CMOVN, CSUB, CSUBN, SEL, SELN  |

**Note** : Status is modified only when the contents of *Dd* are not zero, else these bits are unchanged.

#### 10.4.35 Conditional Add-Not ..... CADDN

**Table 10-45**  
**CADDN**

|                    |   |
|--------------------|---|
| <b>Syntax</b>      | caddn Dc, Dd, Da, Db (RRR)  |
|                    | caddn Dc, Dd, Da, const9 (RCR)  |
|                    | caddn Da, D15, const4 (SRC)   |
| <b>Description</b> | If contents of data register <i>Dd</i> are zero, add the contents of data register <i>Da</i> and the contents of register <i>Db/const9</i> and put the result in data register <i>Dc</i> ; otherwise, put the contents of <i>Da</i> in <i>Dc</i> . The <i>const9</i> value is sign-extended to 32 bits. |
|                    | If contents of data register D15 are zero, add the contents of data register <i>Da</i> and the contents of <i>const4</i> and put the result in data register <i>Da</i> ; otherwise, the contents of <i>Da</i> is unchanged. The <i>const4</i> value is sign-extended to 32 bits.                        |
| <b>Operation</b>   | $D[c] = ((D[d] == 0) ? D[a] + D[b] : D[a])$   |
|                    | $D[c] = ((D[d] == 0) ? D[a] + \text{sign\_ext}(\text{const9}) : D[a])$  |
|                    | $D[a] = ((D[15] == 0) ? D[a] + \text{sign\_ext}(\text{const4}) : D[a])$   |

**Table 10-45**
**CADDN**

|                 |   |
|-----------------|---|
| <b>Status</b>   | V, SV, AV, SAV                            |
| <b>Examples</b> | caddn d3, d4, d1, d2                      |
|                 | caddn d3, d4, d1, 126                     |
|                 | caddn d1, d15, 6                          |
| <b>See also</b> | CADD, CMOV, CMOVN, CSUB, CSUBN, SEL, SELN |

**Note :** Status is modified only when the contents of Dd are zero, else these bits are unchanged.

**10.4.36 Call ..... CALL**
**Table 10-46**
**CALL**

|                    |   |
|--------------------|---|
| <b>Syntax</b>      | call disp24 (B)   |
|                    | call disp8 (SB)   |
| <b>Description</b> | Add the value specified by <i>disp24</i> , 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 $\pm 16$ 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. |
|                    | Add the value specified by <i>disp8</i> , 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 $\pm 256$ bytes 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.  |
| <b>Operation</b>   | ret_addr = PC + 4;<br>PC = PC + sign_ext(2 * disp24);<br>Save upper context;<br>A[11] = ret_addr;   |
|                    | ret_addr = PC + 2;<br>PC = PC + sign_ext(2 * disp8);<br>Save upper context;<br>A[11] = ret_addr;  |
| <b>Status</b>      | -   |
| <b>Examples</b>    | call foobar   |
|                    | call foobar   |
| <b>See also</b>    | CALLA, CALLI, RET   |

**Note :** After CALL, upper context registers other than A10 and A11 are undefined.

2001-04-30 @ 15:16

10.4.37 Call Absolute ..... CALLA

Table 10-47  
CALLA

|             |  |
|-------------|--|
| Syntax      | calla    disp24 (B)  |
| Description | Jump to the address specified by <i>disp24</i> (See <a href="#">Figure 10-1</a> ). 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. |
| Operation   | ret_addr = PC+4<br>PC = {disp24[23:20], 7'b0000000, disp24[19:0], 1'b0};<br>Save upper context;<br>A[11] = ret_addr;   |
| Status      | -  |
| Examples    | calla    foobar  |
| See also    | CALL, CALLI, JL, JLA, RET  |

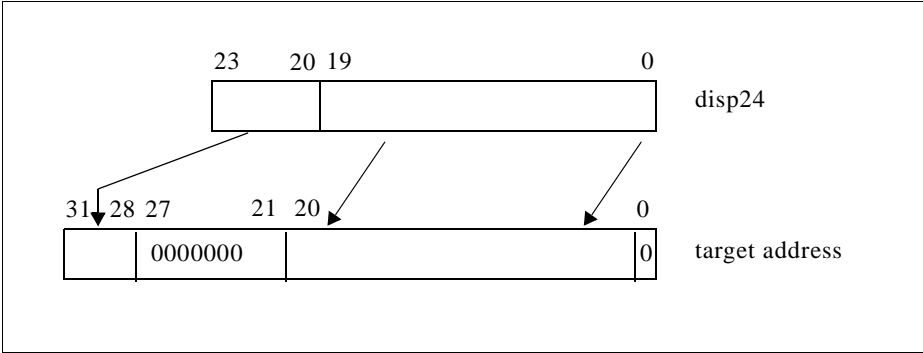


Figure 10-1  
CALLA jump description



2001-04-30 @ 15:16

**10.4.38 Call Indirect ..... CALLI**
**Table 10-48**
**CALLI**

|                    |   |
|--------------------|---|
| <b>Syntax</b>      | calli    Aa (RR)  |
| <b>Description</b> | Jump to address specified by contents of address register <i>Aa</i> . 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. |
| <b>Operation</b>   | ret_addr = PC + 4;<br>PC = {A[a][31:1],1b0};<br>Save upper context;<br>A[11]= ret_addr;   |
| <b>Status</b>      | -   |
| <b>Examples</b>    | calli    a2   |
| <b>See also</b>    | CALL, CALLA, RET  |

**10.4.39 Count Leading Ones ..... CLO**
**Table 10-49**
**CLO**

|                    |   |
|--------------------|---|
| <b>Syntax</b>      | clo       Dc, Da (RR)   |
| <b>Description</b> | Count number of consecutive ones in <i>Da</i> , starting with bit 31, and put result in <i>Dc</i> . |
| <b>Operation</b>   | D[c] = #leading_ones (D[a])   |
| <b>Status</b>      | -   |
| <b>Examples</b>    | clo       d3, d1  |
| <b>See also</b>    | CLS, CLZ  |

**10.4.40 Count Leading Ones in Packed Halfwords. ....CLO.H**
**Table 10-50**
**CLO.H**

|                    |   |
|--------------------|---|
| <b>Syntax</b>      | clo.h    Dc, Da (RR)  |
| <b>Description</b> | Count number of consecutive ones in each halfword of <i>Da</i> , starting with the most-significant bit, and put each result in the corresponding halfword of <i>Dc</i> . |
| <b>Operation</b>   | clo.h : D[c][:(n+15):n] = #leading_ones(D[a][:(n+15):n]); n = 0, 16   |
| <b>Status</b>      | TBD   |
| <b>Examples</b>    | clo.h    d3, d1   |
| <b>See also</b>    | CLO, CLS, CLS.H, CLZ, CLZ.H   |

2001-04-30 @ 15:16

#### 10.4.41 Count Leading Signs .....CLS

Table 10-51  
CLS

|                    |  |
|--------------------|--|
| <b>Syntax</b>      | cls Dc, Da (RR)  |
| <b>Description</b> | 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. |
| <b>Operation</b>   | $D[c] = \#leading\_signs(D[a]) - 1$  |
| <b>Status</b>      | -  |
| <b>Examples</b>    | cls d3, d1   |
| <b>See also</b>    | CLO, CLZ   |

#### 10.4.42 Count Leading Signs in Packed Halfwords .....CLS.H

Table 10-52  
CLS.H

|                    |   |
|--------------------|---|
| <b>Syntax</b>      | cls.h Dc, Da (RR)   |
| <b>Description</b> | Count number of consecutive bits in each halfword in data register Da, which have the same state as the most-significant bit (msb) in that halfword, starting with the next bit right of the msb. Put each result in the corresponding halfword of Dc. The results are the number of leading sign bits minus one in each halfword, giving the number of redundant sign bits in the halfwords of Da. |
| <b>Operation</b>   | $cls.h : D[c][(n+15):n] = \#leading\_signs(D[a][(n+15):n]) - 1 ; n = 0, 16$   |
| <b>Status</b>      | TBD   |
| <b>Examples</b>    | cls.h d3, d1  |
| <b>See also</b>    | CLO, CLO.H, CLS, CLZ, CLZ.H   |

#### 10.4.43 Count Leading Zeroes .....CLZ

Table 10-53  
CLZ

|                    |  |
|--------------------|--|
| <b>Syntax</b>      | clz Dc, Da (RR)  |
| <b>Description</b> | Count number of consecutive zeroes in Da starting with bit 31, and put result in Dc. |
| <b>Operation</b>   | $D[c] = \#leading\_zeroes(D[a])$   |
| <b>Status</b>      | -  |
| <b>Examples</b>    | clz d3, d1   |
| <b>See also</b>    | CLO, CLS   |

2001-04-30 @ 15:16

#### 10.4.44 Count Leading Zeroes in Packed Halfwords ..... CLZ.H

**Table 10-54**  
**CLZ.H**

|                    |   |
|--------------------|---|
| <b>Syntax</b>      | clz.h    Dc, Da (RR)  |
| <b>Description</b> | Count the number of consecutive zeroes in each byte/halfword of <i>Da</i> , starting with the most-significant bit of each byte/halfword, and put each result in the corresponding byte/halfword of <i>Dc</i> . |
| <b>Operation</b>   | clz.h : $D[c] \ll [(n+15):n] = \#leading\_zeroes(D[a] \ll [(n+15):n]); n = 0, 16$   |
| <b>Status</b>      | TBD   |
| <b>Examples</b>    | clz.h    d3, d1   |
| <b>See also</b>    | CLO, CLO.H, CLS, CLS.H, CLZ   |

#### 10.4.45 Conditional Move ..... CMOV

**Table 10-55**  
**CMOV**

|                    |   |
|--------------------|---|
| <b>Syntax</b>      | cmov    Da, D15, Db (SRR)<br>cmov    Da, D15, const4 (SRC)  |
| <b>Description</b> | If the contents of data register D15 are non-zero, copy the contents of data register <i>Db/const4</i> to data register <i>Da</i> ; otherwise, the contents of <i>Da</i> is unchanged. The <i>const4</i> value is sign-extended to 32 bits. |
| <b>Operation</b>   | $D[a] = ((D[15] \neq 0) ? D[b] : D[a])$<br>$D[a] = ((D[15] \neq 0) ? sign\_ext(const4) : D[a])$   |
| <b>Status</b>      | -   |
| <b>Examples</b>    | cmov    d1, d15, d2<br>cmov    d1, d15, 6   |
| <b>See also</b>    | CADD, CADDN, CMOVN, CSUB, CSUBN, SEL, SELN  |

#### 10.4.46 Conditional Move-Not ..... CMOVN

**Table 10-56**  
**CMOVN**

|                    |   |
|--------------------|---|
| <b>Syntax</b>      | cmovn   Da, D15, Db (SRR)<br>cmovn   Da, D15, const4 (SRC)  |
| <b>Description</b> | If the contents of data register D15 are zero, copy the contents of data register <i>Db/const4</i> to data register <i>Da</i> ; otherwise, the contents of <i>Da</i> is unchanged. The <i>const4</i> value is sign-extended to 32 bits. |

2001-04-30 @ 15:16

**Table 10-56**  
**CMOVN**

|                  |   |
|------------------|---|
| <b>Operation</b> | $D[a] = ((D[15] == 0) ? D[b] : D[a])$<br>$D[a] = ((D[15] == 0) ? \text{sign\_ext}(\text{const4}) : D[a])$ |
| <b>Status</b>    | -   |
| <b>Examples</b>  | <code>cmovn    d1, d15, d2</code><br><code>cmovn    d1, d15, 6</code>                                     |
| <b>See also</b>  | CADD, CADDN, CMOV, CSUB, CSUBN, SEL, SELN   |

#### 10.4.47 Conditional Subtract. . . . . **CSUB**

**Table 10-57**  
**CSUB**

|                    |  |
|--------------------|--|
| <b>Syntax</b>      | <code>csub    Dc, Dd, Da, Db (RRR)</code>  |
| <b>Description</b> | If the contents of data register <i>Dd</i> are non-zero, subtract the contents of data register <i>Db</i> from the contents of data register <i>Da</i> and put the result in data register <i>Dc</i> ; otherwise, put the contents of <i>Da</i> in <i>Dc</i> . |
| <b>Operation</b>   | $D[c] = ((D[d] \neq 0) ? D[a] - D[b] : D[a])$  |
| <b>Status</b>      | V, SV, AV, SAV   |
| <b>Examples</b>    | <code>csub    d3, d4, d1, d2</code>  |
| <b>See also</b>    | CADD, CADDN, CMOV, CMOVN, CSUBN, SEL, SELN   |

**Note** : Status is modified only when the contents of *Dd* are not zero, else these bits are unchanged.

#### 10.4.48 Conditional Subtract-Not . . . . . **CSUBN**

**Table 10-58**  
**CSUBN**

|                    |   |
|--------------------|---|
| <b>Syntax</b>      | <code>csubn   Dc, Dd, Da, Db (RRR)</code>   |
| <b>Description</b> | If the contents of data register <i>Dd</i> are zero, subtract the contents of data register <i>Db/const9</i> from the contents of data register <i>Da</i> and put the result in data register <i>Dc</i> ; otherwise, put the contents of <i>Da</i> in <i>Dc</i> . |
| <b>Operation</b>   | $D[c] = ((D[d] == 0) ? D[a] - D[b] : D[a])$   |
| <b>Status</b>      | V, SV, AV, SAV  |
| <b>Examples</b>    | <code>csubn   d3, d4, d1, d2</code>   |
| <b>See also</b>    | CADD, CADDN, CMOV, CMOVN, CSUB, SEL, SELN.A   |

**Note** : Status is modified only when the contents of *Dd* are not zero, else these bits are unchanged.

**10.4.49 Debug.....DEBUG**
**Table 10-59  
DEBUG**

|                    |  |
|--------------------|--|
| <b>Syntax</b>      | debug (SYS)  |
|                    | debug (SR)   |
| <b>Description</b> | 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. |
| <b>Operation</b>   | -  |
|                    | -  |
| <b>Status</b>      | -  |
| <b>Examples</b>    | debug  |
|                    | debug  |
| <b>See also</b>    | -  |

**10.4.50 Extract from Double Register .....DEXTR**
**Table 10-60  
DEXTR**

|                    |   |
|--------------------|---|
| <b>Syntax</b>      | dextr Dc, Da, Db, Dd (RRRR)   |
|                    | dextr Dc, Da, Db, p (RRPW)  |
| <b>Description</b> | Extract 32 bits from registers $\{Da, Db\}$ (where $Da$ contains the most-significant 32 bits of the value) starting at bit number specified by $63 - Dd[4:0]/p$ . Put result in $Dc$ .<br>Note: $Da$ and $Db$ can be any two data registers or the same register. For this instruction they are treated as a 64-bit entity where $Da$ contributes the high order bits and $Db$ the low order bits. |
| <b>Operation</b>   | $pos = D[d][4:0] / p;$<br>$D[c] = (\{D[a], D[b]\} \ll pos)[63:32];$   |
| <b>Status</b>      | -   |
| <b>Examples</b>    | dextr d1, d3, d5, d7  |
|                    | dextr d1, d3, d5, 11  |
| <b>See also</b>    | EXTR, EXTR.U, INSERT, INS.T, INSN.T   |

2001-04-30 @ 15:16

### 10.4.51 Disable Interrupts . . . . . DISABLE

**Table 10-61**  
**DISABLE**

|                    |   |
|--------------------|---|
| <b>Syntax</b>      | disable (SYS)   |
| <b>Description</b> | Disable interrupts by clearing Interrupt Enable bit (ICR.IE) in the Interrupt Control Register. |
| <b>Operation</b>   | -   |
| <b>Status</b>      | -   |
| <b>Examples</b>    | disable   |
| <b>See also</b>    | ENABLE  |

**Note :** DISABLE can be executed only in User1 or Supervisor privilege mode

### 10.4.52 Synchronize Data . . . . . DSYNC

**Table 10-62**  
**DSYNC**

|                    |   |
|--------------------|---|
| <b>Syntax</b>      | dsync (SYS)   |
| <b>Description</b> | Forces all data accesses to complete before any data accesses associated with an instruction semantically after the DSYNC are initiated.<br><b>Note :</b> dcache is not invalidated by dsync. |
| <b>Operation</b>   | -   |
| <b>Status</b>      | -   |
| <b>Examples</b>    | dsync   |
| <b>See also</b>    | ISYNC   |

### 10.4.53 Divide-Adjust. . . . . DVADJ

**Table 10-63**  
**DVADJ**

|                    |  |
|--------------------|--|
| <b>Syntax</b>      | <code>dvadj Ec, Ed, Db (RRR)</code>  |
| <b>Description</b> | <p>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.</p> <p>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.</p> <p>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.</p> <p>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.</p> |
| <b>Operation</b>   | $E[c] = \text{divide\_adjust}(E[d], D[b])$   |
| <b>Status</b>      | V, SV  |
| <b>Examples</b>    | -  |

2001-04-30 @ 15:16

|  |                  |
|--|------------------|
| <b>10.4.54 Divide-Initialization Word</b> .....      | <b>DVINIT</b>    |
| <b>Divide-Initialization Word Unsigned</b> .....     | <b>DVINIT.U</b>  |
| <b>Divide-Initialization Byte</b> .....              | <b>DVINIT.B</b>  |
| <b>Divide-Initialization Byte Unsigned</b> .....     | <b>DVINIT.BU</b> |
| <b>Divide-Initialization Halfword</b> .....          | <b>DVINIT.H</b>  |
| <b>Divide-Initialization Halfword Unsigned</b> ..... | <b>DVINIT.HU</b> |

**Table 10-64**

**DVINIT, DVINIT.U, DVINIT.B, DVINIT.BU, DVINIT.H, & DVINIT.HU**

|                    |   |
|--------------------|---|
| <b>Syntax</b>      | <div> <div>dvinit Ec, Da, Db (RR)</div> <div>dvinit.u Ec, Da, Db (RR)</div> <div>dvinit.b Ec, Da, Db (RR)</div> <div>dvinit.buEc, Da, Db (RR)</div> <div>dvinit.h Ec, Da, Db (RR)</div> <div>dvinit.huEc, Da, Db (RR)</div> </div>  |
| <b>Description</b> | <p>Sign-extend (DVINIT, 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.</p> <p>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.</p> |
| <b>Operation</b>   | <div> <div>E[c] = divide_init(D[a], D[b])</div> <div>E[c] = divide_init_u(D[a], D[b])</div> <div>E[c] = divide_init_b(D[a], D[b])</div> <div>E[c] = divide_init_b_u(D[a], D[b])</div> <div>E[c] = divide_init_h(D[a], D[b])</div> <div>E[c] = divide_init_h_u(D[a], D[b])</div> </div>  |
| <b>Status</b>      | V, SV   |
| <b>Examples</b>    | –   |
| <b>See also</b>    | DVADJ, DVSTEP, DVSTEP.U   |



**10.4.55 Divide-Step ..... DVSTEP**  
**Divide-Step Unsigned ..... DVSTEP.U**

**Table 10-65**  
**DVSTEP & DVSTEP.U**

|                    |   |
|--------------------|---|
| <b>Syntax</b>      | <div>dvstep <i>Ec</i>, <i>Ed</i>, <i>Db</i> (RRR)</div> <div>dvstep.u<i>Ec</i>, <i>Ed</i>, <i>Db</i> (RRR)</div>  |
| <b>Description</b> | <p>Divide the contents of extended register <i>Ed</i>, 8 bits at a time, by data register <i>Db</i>, and put the result in extended register <i>Ec</i>. <i>Ed</i> contains the result of a previous divide-initialization (DVINIT or similar) or divide-step (DVSTEP or DVSTEP.U) instruction. <i>Db</i> contains the divisor for the current divide operation</p> <p>The most-significant word of <i>Ed</i> (data register <i>D[d+1]</i>) 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 <i>Ed</i> (data register <i>Dd</i>) 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 <i>Dd</i>, 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.</p> <p>Each divide-step instruction processes 8 additional dividend bits, and develops 8 additional bits of quotient. A divide operation yielding a 32-bit quotient value requires four divide-step instructions. (Refer to the description of the DVINIT and 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, (or equivalent set-up).</p> <p>The unsigned divide-step instructions treat the dividend and partial remainders as unsigned, 32-bit values and develop 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.</p> |
| <b>Operation</b>   | <div><math>E[c] = \text{divide\_step}(E[d], D[b])</math></div> <div><math>E[c] = \text{divide\_step\_u}(E[d], D[b])</math></div>  |
| <b>Status</b>      | -   |
| <b>Examples</b>    | -   |
| <b>See also</b>    | DVADJ, DVINIT, DVINIT.B, DVINIT.BU, DVINIT.H, DVINIT.HU, DVINIT.U   |

2001-04-30 @ 15:16

**10.4.56 Enable Interrupts. . . . .ENABLE**

**Table 10-66**  
**ENABLE**

|                    |  |
|--------------------|--|
| <b>Syntax</b>      | enable (SYS)   |
| <b>Description</b> | Enable interrupts by setting Interrupt Enable bit (ICR.IE) in Interrupt Control Register to 1. |
| <b>Operation</b>   | -  |
| <b>Status</b>      | -  |
| <b>Examples</b>    | enable   |
| <b>See also</b>    | DISABLE  |

**Note :** ENABLE can be executed only in User1 or Supervisor privilege mode

**10.4.57 Equal . . . . .EQ**

**Table 10-67**  
**EQUAL**

|                    |   |                       |
|--------------------|---|-----------------------|
| <b>Syntax</b>      | eq  | Dc, Da, Db (RR)       |
|                    | eq  | Dc, Da, const9 (RC)   |
|                    | eq  | D15, Da, Db (SRR)     |
|                    | eq  | D15, Da, const4 (SRC) |
| <b>Description</b> | If the contents of data register <i>Da</i> are equal to the contents of data register <i>Db</i> /<br><i>const9</i> , set the least-significant bit of <i>Dc</i> to 1 and clear the remaining bits to zero;<br>otherwise, clear all bits in <i>Dc</i> . The <i>const9</i> value is sign-extended to 32 bits. |                       |
|                    | If the contents of data register <i>Da</i> are equal to the contents of data register <i>Db</i> /<br><i>const4</i> , set the least-significant bit of D15 to 1 and clear the remaining bits to zero;<br>otherwise, clear all bits in D15. The <i>const4</i> value is sign-extended to 32 bits.              |                       |
| <b>Operation</b>   | D[c] = (D[a] == D[b])<br>D[c] = (D[a] == sign_ext(const9))  |                       |
|                    | D[15] = (D[a] == D[b])<br>D[15] = (D[a] == sign_ext(const4))  |                       |
| <b>Status</b>      | -   |                       |
| <b>Examples</b>    | eq  | d3, d1, d2            |
|                    | eq  | d3, d1, 126           |
|                    | eq  | d15, d1, d2           |
|                    | eq  | d15, d1, 6            |
| <b>See also</b>    | GE, GE.U, LT, LT.U, NE  |                       |

**10.4.58 Equal to Address. ....EQ.A**
**Table 10-68**
**EQ.A**

|                    |  |
|--------------------|--|
| <b>Syntax</b>      | eq.a    Dc, Aa, Ab (RR)  |
| <b>Description</b> | If the contents of address registers <i>Aa</i> and <i>Ab</i> are equal, set the least-significant bit of <i>Dc</i> to 1 and clear the remaining bits to zero; otherwise, clear all bits in <i>Dc</i> . |
| <b>Operation</b>   | $D[c] = (A[a] == A[b])$  |
| <b>Status</b>      | -  |
| <b>Examples</b>    | eq.a    d3, a4, a2   |
| <b>See also</b>    | EQZ.A, GE.A, LT.A, NE.A, NEZ.A   |

**10.4.59 Equal Packed Byte .....EQ.B**
**Equal Packed Halfword .....EQ.H**
**Equal Packed Word .....EQ.W**
**Table 10-69**
**EQ.B, EQ.H, & EQ.W**

|                    |  |
|--------------------|--|
| <b>Syntax</b>      | eq.b    Dc, Da, Db (RR)<br>eq.h    Dc, Da, Db (RR)<br>eq.w    Dc, Da, Db (RR)  |
| <b>Description</b> | Compare each byte/halfword/word of <i>Da</i> with corresponding byte/halfword/word of <i>Db</i> . In each case, if the two are equal, set the corresponding byte/halfword/word of <i>Dc</i> to all 1's; otherwise, set the corresponding byte/halfword/word of <i>Dc</i> to all 0's.   |
| <b>Operation</b>   | if $(D[a][(n+7):n] == D[b][(n+7):n])$<br>then $D[c][(n+7):n] = 8'h\ FF$<br>else $D[c][(n+7):n] = 8'h\ 00$ ; $n = 0, 8, 16, 24$<br><br>if $(D[a][(n+15):n] == D[b][(n+15):n])$<br>then $D[c][(n+15):n] = 16'h\ FFFF$<br>else $D[c][(n+15):n] = 16'h\ 0000$ ; $n = 0, 16$<br><br>if $(D[a] == D[b])$<br>then $D[c] = 32'h\ FFFFFFFF$<br>else $D[c] = 32'h\ 00000000$ |
| <b>Status</b>      | -  |
| <b>Examples</b>    | eq.b    d3, d1, d2<br>eq.h    d3, d1, d2<br>eq.w    d3, d1, d2   |
| <b>See also</b>    | LT.B, LT.BU, LT.H, LT.HU, LT.W, LT.WU  |

2001-04-30 @ 15:16

**10.4.60 Equal Any Byte ..... EQANY.B**  
**Equal Any Halfword ..... EQANY.H**

**Table 10-70**  
**EQANY.B & EQANY.H**

|                    |  |
|--------------------|--|
| <b>Syntax</b>      | eqany.b Dc, Da, Db (RR)<br>eqany.b Dc, Da, const9 (RC)<br>eqany.h Dc, Da, Db (RR)<br>eqany.h Dc, Da, const9 (RC)   |
| <b>Description</b> | Compare each byte/halfword of <i>Da</i> with the corresponding byte/halfword of <i>Db/const9</i> . If the logical OR of the Boolean results from each comparison is TRUE, set the least-significant bit of <i>Dc</i> to 1 and clear the remaining bits to zero; otherwise, clear all bits in <i>Dc</i> . <i>Const9</i> is sign-extended to 32 bits.  |
| <b>Operation</b>   | $D[c] = (D[a][31:24] == D[b][31:24])$ $\text{OR } (D[a][23:16] == D[b][23:16])$ $\text{OR } (D[a][15:8] == D[b][15:8])$ $\text{OR } (D[a][7:0] == D[b][7:0])$<br>$D[c] = (D[a][31:24] == \text{sign\_ext}(\text{const9})[31:24])$ $\text{OR } (D[a][23:16] == \text{sign\_ext}(\text{const9})[23:16])$ $\text{OR } (D[a][15:8] == \text{sign\_ext}(\text{const9})[15:8])$ $\text{OR } (D[a][7:0] == \text{sign\_ext}(\text{const9})[7:0])$ |
| <b>Status</b>      | -  |
| <b>Examples</b>    | eqany.b      d3, d1, d2<br><br>eqany.b      d3, d1, 126<br><br>eqany.h      d3, d1, d2<br><br>eqany.h      d3, d1, 126   |
| <b>See also</b>    | EQ, GE, GE.U, LT, LT.U, NE   |

**10.4.61 Equal Zero Address ..... EQZ.A**

**Table 10-71**  
**EQZ.A**

|                    |   |
|--------------------|---|
| <b>Syntax</b>      | eqz.a Dc, Aa (RR)   |
| <b>Description</b> | If the contents of address register <i>Aa</i> are equal to zero, set the least significant bit of <i>Dc</i> to 1 and clear the remaining bits to zero; otherwise, clear all bits in <i>Dc</i> . |
| <b>Operation</b>   | $D[c] = (A[a] == 0)$  |
| <b>Status</b>      | -   |
| <b>Examples</b>    | eqz.a d3, a4  |
| <b>See also</b>    | EQ.A, GE.A, LT.A, NE.A, NEZ.A   |

2001-04-30 @ 15:16

## 10.4.62 Extract Bit Field ..... EXTR

### Extract Bit Field Unsigned ..... EXTR.U

**Table 10-72**
**EXTR & EXTR.U**

|                    |   |
|--------------------|---|
| <b>Syntax</b>      | extr     Dc, Da, Ed (RRRR)<br>extr     Dc, Da, Dd, w (RRRW)<br>extr     Dc, Da, p, w (RRPW)<br>extr.u   Dc, Da, Ed (RRRR)<br>extr.u   Dc, Da, Dd, w (RRRW)<br>extr.u   Dc, Da, p, w (RRPW)  |
| <b>Description</b> | Extract from <i>Da</i> the number of consecutive bits specified by <i>Ed(upper)/w</i> , starting at the bit number specified by <i>Ed(lower)/Dd/p</i> , and put the result, sign-extended (extr) or zero-extended (extr.u) to 32 bits, in <i>Dc</i> .   |
| <b>Operation</b>   | <pre>pos = E[d](lower) [4:0] / D[d][4:0]/p; width = E[d](upper)[4:0] / w D[c] = sign_ext((D[a]&gt;&gt;pos)[width-1:0])  pos = E[d](lower) [4:0] / D[d][4:0]/p; width = E[d](upper)[4:0] / w D[c] = zero_ext ((D[a]&gt;&gt;pos)[width-1:0])</pre> <p>Note: For EXTR and EXTR.U and either (pos+width &gt; 32) or (width = 0) the results of the instruction are undefined.</p> |
| <b>Status</b>      | -   |
| <b>Examples</b>    | <pre>extr     d3, d1, e2 extr     d3, d1, d2, 4 extr     d3, d1, 2, 4 extr.u   d3, d1, e2 extr.u   d3, d1, d2, 4 extr.u   d3, d1, 2, 4</pre>  |
| <b>See also</b>    | DEXTR, INSERT, INS.T, INS.N.T   |

2001-04-30 @ 15:16

**10.4.63 Greater Than or Equal . . . . . GE**  
**Greater Than or Equal Unsigned. . . . .GE.U**

**Table 10-73**  
**GE & GE.U**

|                    |  |
|--------------------|--|
| <b>Syntax</b>      | ge Dc, Da, Db (RR)<br>ge Dc, Da, const9 (RC)<br>ge.u Dc, Da, Db (RR)<br>ge.u Dc, Da, const9 (RC)   |
| <b>Description</b> | If the contents of data register <i>Da</i> are greater than or equal to the contents of data register <i>Db/const9</i> , set the least-significant bit of <i>Dc</i> to 1 and clear the remaining bits to zero; otherwise, clear all bits in <i>Dc</i> . <i>Da</i> and <i>Db</i> are treated as 32-bit signed integers, and the <i>const9</i> value is sign-extended to 32 bits.<br><br>If the contents of data register <i>Da</i> are greater than or equal to the contents of data register <i>Db/const9</i> , set the least-significant bit of <i>Dc</i> to 1 and clear the remaining bits to zero; otherwise, clear all bits in <i>Dc</i> . <i>Da</i> and <i>Db</i> are treated as 32-bit unsigned integers, and the <i>const9</i> value is zero-extended to 32 bits. |
| <b>Operation</b>   | D[c] = (D[a] >= D[b]); signed<br>D[c] = (D[a] >= sign_ext(const9)); signed<br><br>D[c] = (D[a] >= D[b]); unsigned<br>D[c] = (D[a] >= zero_ext(const9)); unsigned   |
| <b>Status</b>      | -  |
| <b>Examples</b>    | ge d3, d1, d2<br>ge d3, d1, 126<br>ge.u d3, d1, d2<br>ge.u d3, d1, 126   |
| <b>See also</b>    | EQ, LT, LT.U, NE   |

**10.4.64 Greater Than or Equal Address . . . . .GE.A**

**Table 10-74**  
**GE.A**

|                    |  |
|--------------------|--|
| <b>Syntax</b>      | ge.a Dc, Aa, Ab (RR)   |
| <b>Description</b> | If contents of address register <i>Aa</i> are greater than or equal to contents of address register <i>Ab</i> , set the least-significant bit of <i>Dc</i> to 1 and clear remaining bits to zero; otherwise, clear all bits in <i>Dc</i> . Operands are treated as unsigned 32-bit integers. |
| <b>Operation</b>   | D[c] = (A[a] >= A[b]); unsigned  |
| <b>Status</b>      | -  |
| <b>Examples</b>    | ge.a d3, a4, a2  |
| <b>See also</b>    | EQ.A, EQZ.A, LT.A, NE.A, NEZ.A   |

**10.4.65 Insert Mask ..... IMASK**
**Table 10-75**
**IMASK**

|                    |   |
|--------------------|---|
| <b>Syntax</b>      | imask Ec, Db, Dd, w (RRRW)<br>imask Ec, Db, p, w (RRPW)<br>imask Ec, const4, Dd, w (RCRW)<br>imask Ec, const4, p, w (RCPW)  |
| <b>Description</b> | Create a mask containing the number of bits specified by <i>w</i> , starting at the bit number specified by <i>Dd[4:0]/p</i> , and put the mask in data register <i>Ec(upper)</i> . Left-shift the value in <i>Db/const4</i> by the amount specified by <i>Dd[4:0]/p</i> and put the result value in <i>Ec(lower)</i> . The value <i>const4</i> 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. |
| <b>Operation</b>   | $pos = D[d][4:0] / p;$<br>$E[c](upper) = ((2^w - 1) \ll pos);$<br>$E[c](lower) = (D[b] \ll pos);$<br>zero_ext(const4) may replace <i>D[b]</i>   |
| <b>Status</b>      | -   |
| <b>Examples</b>    | imask e2, d1, d2, 11<br>imask e2, d1, 5, 11<br>imask e2, 6, d2, 11<br>imask e2, 6, 5, 11  |
| <b>See also</b>    | LDMST, ST.T   |

**10.4.66 Insert Bit .....INS.T**
**Insert Bit-Not ..... INSN.T**
**Table 10-76**
**INS.T & INSN.T**

|                    |   |
|--------------------|---|
| <b>Syntax</b>      | ins.t Dc, Da, p1, Db, p2 (BIT)<br>insn.t Dc, Da, p1, Db, p2 (BIT)   |
| <b>Description</b> | Move value of <i>Da</i> , with bit <i>p1</i> of this value replaced with bit <i>p2</i> of register <i>Db</i> , to <i>Dc</i> .<br>Move value of <i>Da</i> , with bit <i>p1</i> of this value replaced with inverse of bit <i>p2</i> of register <i>Db</i> , to <i>Dc</i> . |
| <b>Operation</b>   | $D[c] = \{D[a][31:(p1+1)], D[b][p2], D[a][(p1-1):0]\}$<br>$D[c] = \{D[a][31:(p1+1)], !D[b][p2], D[a][(p1-1):0]\}$   |
| <b>Status</b>      | -   |
| <b>Examples</b>    | ins.t d3, d1, 5, d2, 7<br>insn.t d3, d1, 5, d2, 7   |
| <b>See also</b>    | DEXTR, EXTR, EXTR.U, INSERT   |

2001-04-30 @ 15:16

**10.4.67 Insert Bit Field . . . . .INSERT**

**Table 10-77**  
**INSERT**

|                    |  |
|--------------------|--|
| <b>Syntax</b>      | insert Dc, Da, Db, Ed (RRRR)<br>insert Dc, Da, Db, Dd, w (RRRW)<br>insert Dc, Da, Db, p, w (RRPW)<br>insert Dc, Da, const4, Ed (RCRR)<br>insert Dc, Da, const4, Dd, w (RCRW)<br>insert Dc, Da, const4, p, w (RCPW)   |
| <b>Description</b> | Extract from <i>Db/const4</i> , starting at bit 0, the number of consecutive bits specified by <i>Ed(upper)/w</i> , and shift the result left by the number of bits specified by <i>Ed(upper)/Dd/p</i> ; extract a copy of <i>Da</i> , clearing the bits starting at the bit position specified by <i>Ed(upper)/Dd/p</i> , and extending for the number of bits specified by <i>Ed(upper)/w</i> . Put the logical OR of the two extracted words into <i>Dc</i> . |
| <b>Operation</b>   | $pos = E[d](lower)[4:0] / D[d][4:0]/p;$<br>$width = E[d](upper)[4:0] / w;$<br>$m = (2^{width}-1) << pos;$<br>$D[c] = (D[a] \text{ and } !m) \text{ or } ((D[b] << pos) \text{ and } m)$<br>zero_ext(const4) may replace D[b]<br>Note: When $pos+width>31$ some of the high order bits selected from D[b] will not be inserted into D[a].   |
| <b>Status</b>      | -  |
| <b>Examples</b>    | insert d3, d1, d2, e4<br>insert d3, d1, d2, d4, 8<br>insert d3, d1, d2, 16, 8<br>insert d3, d1, 0, e4<br>insert d3, d1, 0, d4, 8<br>insert d3, d1, 0, 16, 8  |
| <b>See also</b>    | DEXTR, EXTR, EXTR.U, INS.T, INSN.T   |

**10.4.68 Synchronize Instructions . . . . .ISYNC**

**Table 10-78**  
**ISYNC**

|                    |  |
|--------------------|--|
| <b>Syntax</b>      | isync (SYS)  |
| <b>Description</b> | Forces completion of all previous instructions, then flushes the CPU pipelines, and invalidates any cached pipeline state before proceeding to the next instruction.<br>Note: I-cache is not invalidated by ISYNC. |
| <b>Operation</b>   | -  |
| <b>Status</b>      | -  |
| <b>Examples</b>    | isync  |
| <b>See also</b>    | DSYNC  |



2001-04-30 @ 15:16

#### 10.4.69 Jump Unconditional ..... J

Table 10-79

J

|             |   |
|-------------|---|
| Syntax      | j      disp24 (B)   |
|             | j      disp8 (SB)   |
| Description | Add the value specified by <i>disp24</i> , multiplied by two and sign-extended to 32 bits, to the contents of the PC, and jump to that address. |
|             | Add the value specified by <i>disp8</i> , multiplied by two and sign-extended to 32 bits, to the contents of the PC, and jump to that address.  |
| Operation   | PC = PC + sign_ext(2 * disp24)  |
|             | PC = PC + sign_ext(2 * disp8)   |
| Status      | -   |
| Examples    | j      foobar   |
|             | j      foobar   |
| See also    | JA, JI, JL, JLA, JLI  |

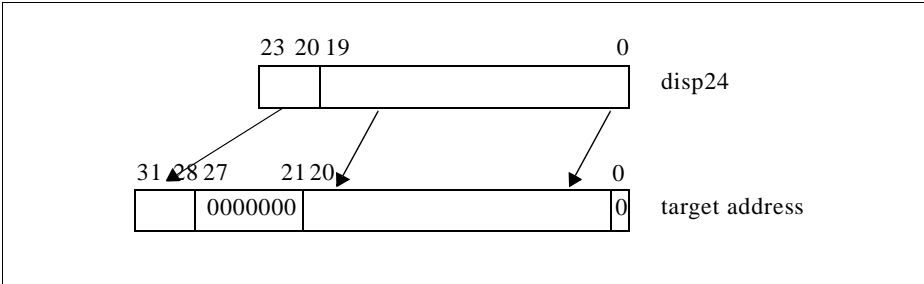
#### 10.4.70 Jump Unconditional Absolute ..... JA

Table 10-80

JA

|             |   |
|-------------|---|
| Syntax      | ja      disp24 (B)  |
| Description | Load the value specified by <i>disp24</i> into the PC and jump to that address. The value <i>disp24</i> is used to form the effective address (See <a href="#">Figure 10-2</a> ). |
| Operation   | PC = {disp24[23:20], 7'b00000000, disp24[19:0], 1'b0};  |
| Status      | -   |
| Examples    | ja      foobar  |
| See also    | JI, JL, JLA, JLI  |

2001-04-30 @ 15:16



**Figure 10-2**  
**JA Jump Description**

#### 10.4.71 Jump if Equal. ....JEQ

**Table 10-81**  
**JEQ**

|                    |  |
|--------------------|--|
| <b>Syntax</b>      | jeq Da, Db, disp15 (BRR)<br>jeq Da, const4, disp15 (BRC)   |
|                    | jeq D15, Db, disp4 (SBR)<br>jeq D15, const4, disp4 (SBC)   |
| <b>Description</b> | <p>If the contents of <i>Da</i> are equal to the contents <i>Db/const4</i>, then add the value specified by <i>disp15</i>, multiplied by two and sign-extended to 32 bits, to the contents of the PC, and jump to that address. The <i>const4</i> value is sign-extended to 32 bits.</p> <p>If the contents of D15 are equal to the contents <i>Db/const4</i>, then add the value specified by <i>disp4</i>, multiplied by two and zero-extended to 32 bits, to the contents of the PC, and jump to that address. The <i>const4</i> value is sign-extended to 32 bits.</p> |
| <b>Operation</b>   | if (D[a]==D[b]) then PC = PC+sign_ext(2 * disp15)<br>if (D[a]==sign_ext(const4)) then PC = PC+sign_ext(2 * disp15)<br>if (D[15]==D[b]) then PC = PC+zero_ext(2 * disp4)<br>if (D[15]==sign_ext(const4)) then PC = PC+zero_ext(2 * disp4)   |
| <b>Status</b>      | -  |
| <b>Examples</b>    | jeq d1, d2, foobar<br>jeq d1, 6, foobar<br>jeq d15, d2, foobar<br>jeq d15, 6 foobar  |
| <b>See also</b>    | JGE, JGE.U, JLT, JLT.U, JNE  |

**10.4.72 Jump if Equal Address . . . . . JEQ.A**
**Table 10-82**
**JEQ.A**

|                    |   |
|--------------------|---|
| <b>Syntax</b>      | <code>jeq.a Aa, Ab, disp15 (BRR)</code>   |
| <b>Description</b> | If the contents of <i>Aa</i> are equal to the contents <i>Ab</i> , then add the value specified by <i>disp15</i> , multiplied by two and sign-extended to 32 bits, to the contents of the PC, and jump to that address. |
| <b>Operation</b>   | if ( $A[a] == A[b]$ ) then ( $PC = PC + \text{sign\_ext}(2 * \text{disp15})$ )  |
| <b>Status</b>      | -   |
| <b>Examples</b>    | <code>jeq.a a4, a2, foobar</code>   |
| <b>See also</b>    | JNE.A   |

**10.4.73 Jump if Greater Than or Equal . . . . . JGE**
**Jump if Greater Than or Equal Unsigned . . . . . JGE.U**
**Table 10-83**
**JGE & JGE.U**

|                    |  |
|--------------------|--|
| <b>Syntax</b>      | <code>jge Da, Db, disp15 (BRR)</code><br><code>jge Da, const4, disp15 (BRC)</code><br><code>jge.u Da, Db, disp15 (BRR)</code><br><code>jge.u Da, const4, disp15 (BRC)</code>   |
| <b>Description</b> | If contents of <i>Da</i> are greater than or equal to contents of <i>Db/const4</i> , then add value specified by <i>disp15</i> , multiplied by two and sign-extended to 32 bits, to contents of PC, and jump to that address. Operands are treated as signed/unsigned, 32-bit integers. The <i>const4</i> value is sign-extended/zero-extended to 32 bits.   |
| <b>Operation</b>   | if ( $D[a] \geq D[b]$ ) then ( $PC = PC + \text{sign\_ext}(2 * \text{disp15})$ ); signed<br>if ( $D[a] \geq \text{sign\_ext}(\text{const4})$ ) then ( $PC = PC + \text{sign\_ext}(2 * \text{disp15})$ ); signed<br><br>if ( $D[a] \geq D[b]$ ) then ( $PC = PC + \text{sign\_ext}(2 * \text{disp15})$ ); unsigned<br>if ( $D[a] \geq \text{zero\_ext}(\text{const4})$ ) then ( $PC = PC + \text{sign\_ext}(2 * \text{disp15})$ ); unsigned |
| <b>Status</b>      | -  |
| <b>Examples</b>    | <code>jge d1, d2, foobar</code><br><code>jge d1, 6, foobar</code><br><code>jge.u d1, d2, foobar</code><br><code>jge.u d1, 6, foobar</code>   |
| <b>See also</b>    | JEQ, JLT, JLT.U, JNE   |

2001-04-30 @ 15:16

**10.4.74 Jump if Greater Than or Equal to Zero ..... JGEZ**

**Table 10-84**  
**JGEZ**

|                    |   |
|--------------------|---|
| <b>Syntax</b>      | <code>jgez Db, disp4 (SBR)</code>   |
| <b>Description</b> | If the contents of <i>Db</i> are greater than or equal to zero, then add the value specified by <i>disp4</i> , multiplied by two and zero-extended to 32 bits, to the contents of the PC, and jump to that address. |
| <b>Operation</b>   | if ( $D[b] \geq 0$ ) then ( $PC = PC + \text{zero\_ext}(2 * \text{disp4})$ )  |
| <b>Status</b>      | -   |
| <b>Examples</b>    | <code>jgez d2, foobar</code>  |
| <b>See also</b>    | JGTZ, JLEZ, JLTZ, JNZ, JZ   |

**10.4.75 Jump if Greater Than Zero .....JGTZ**

**Table 10-85**  
**JGTZ**

|                    |   |
|--------------------|---|
| <b>Syntax</b>      | <code>jgtz Db, disp4 (SBR)</code>   |
| <b>Description</b> | If the contents of <i>Db</i> are greater than zero, then add the value specified by <i>disp4</i> , multiplied by two and zero-extended to 32 bits, to the contents of the PC, and jump to that address. |
| <b>Operation</b>   | if ( $D[b] > 0$ ) then $PC = PC + \text{zero\_ext}(2 * \text{disp4})$   |
| <b>Status</b>      | -   |
| <b>Examples</b>    | <code>jgtz d2, foobar</code>  |
| <b>See also</b>    | JGEZ, JLEZ, JLTZ, JNZ, JZ   |

**10.4.76 Jump Indirect. ....JI**

**Table 10-86**  
**JI**

|                    |   |
|--------------------|---|
| <b>Syntax</b>      | <code>ji Aa (RR)</code>   |
|                    | <code>ji Aa (SBR)</code>  |
| <b>Description</b> | Load the contents of address register <i>Aa</i> into the PC and jump to that address. The least-significant bit is always set to 0. |
|                    | Load the contents of address register <i>Aa</i> into the PC and jump to that address. The least-significant bit is always set to 0. |
| <b>Operation</b>   | $PC = \{A[a][31:1], 1'b 0\}$  |
|                    | $PC = \{A[a][31:1], 1'b 0\}$  |

2001-04-30 @ 15:16

**Table 10-86**
**Jl**

|                 |                     |
|-----------------|---------------------|
| <b>Status</b>   | -                   |
| <b>Examples</b> | jl     a2           |
|                 | jl     a2           |
| <b>See also</b> | J, JA, JL, JLA, JLI |

#### 10.4.77 Jump and Link. .... JL

**Table 10-87**
**JL**

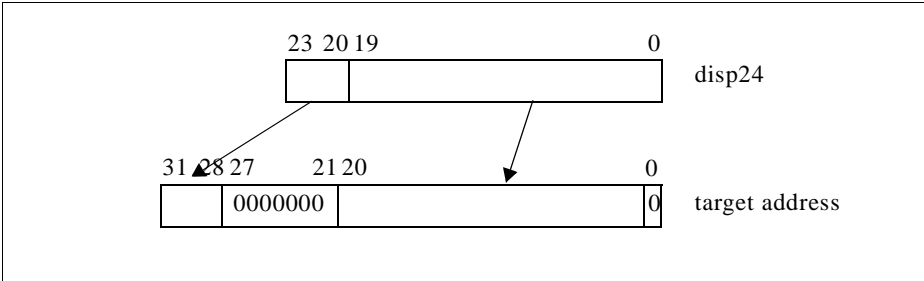
|                    |  |
|--------------------|--|
| <b>Syntax</b>      | jl         disp24 (B)  |
| <b>Description</b> | Store the address of the next instruction in A11. Then add the value specified by <i>disp24</i> , scaled by two and sign-extended to 32 bits, to the contents of the PC, and jump to that address. |
| <b>Operation</b>   | $A[11] = PC + 4$ ; $PC = PC + \text{sign\_ext}(2 * \text{disp24})$ ;   |
| <b>Status</b>      | -  |
| <b>Examples</b>    | jl     foobar  |
| <b>See also</b>    | J, JI, JA, JLA, JLI  |

#### 10.4.78 Jump and Link Absolute ..... JLA

**Table 10-88**
**JLA**

|                    |  |
|--------------------|--|
| <b>Syntax</b>      | jla         disp24 (B)   |
| <b>Description</b> | Store the address of the next instruction in A11. Then load the value specified by <i>disp24</i> into the PC and jump to that address. The value <i>disp24</i> is used to form the effective address (See <a href="#">Figure 10-3</a> ). |
| <b>Operation</b>   | $A[11] = PC + 4$ ; $PC = \{\text{disp24}[23:20], 7'b00000000, \text{disp24}[19:0], 1'b\ 0\}$ ;   |
| <b>Status</b>      | -  |
| <b>Examples</b>    | jla     foobar   |
| <b>See also</b>    | J, JI, JA, JL, JLI   |

2001-04-30 @ 15:16



**Figure 10-3**  
**JLA Jump Description**

#### 10.4.79 Jump if Less Than or Equal to Zero .....JLEZ

**Table 10-89**  
**JLEZ**

|                    |  |
|--------------------|--|
| <b>Syntax</b>      | <code>jlez Db, disp4 (SBR)</code>  |
| <b>Description</b> | If the contents of <i>Db</i> are less than or equal to zero, then add the value specified by <i>disp4</i> , multiplied by two and zero-extended to 32 bits, to the contents of the PC, and jump to that address. |
| <b>Operation</b>   | If ( $D[b] \leq 0$ ) then $PC = PC + \text{zero\_ext}(2 * \text{disp4})$   |
| <b>Status</b>      | -  |
| <b>Examples</b>    | <code>jlez d2, foobar</code>   |
| <b>See also</b>    | JGEZ, JGTZ, JLTZ, JNZ, JZ  |

#### 10.4.80 Jump and Link Indirect. ....JLI

**Table 10-90**  
**JLI**

|                    |   |
|--------------------|---|
| <b>Syntax</b>      | <code>jli Aa (RR)</code>  |
| <b>Description</b> | Store the address of the next instruction in A11. Then load the contents of address register <i>Aa</i> into the PC and jump to that address. The least-significant bit is set to 0. |
| <b>Operation</b>   | $A[11] = PC + 4$ ; $PC = \{A[a][31:1], 1'b\ 0\}$  |
| <b>Status</b>      | -   |
| <b>Examples</b>    | <code>jli a2</code>   |
| <b>See also</b>    | J, JI, JA, JL, JLA  |

#### 10.4.81 Jump if Less Than. .... JLT

#### Jump if Less Than Unsigned. .... JLT.U

**Table 10-91**
**JLT & JLT.U**

|                    |  |
|--------------------|--|
| <b>Syntax</b>      | jlt      Da, Db, disp15 (BRR)<br>jlt      Da, const4, disp15 (BRC)<br>jlt.u    Da, Db, disp15 (BRR)<br>jlt.u    Da, const4, disp15 (BRC)   |
| <b>Description</b> | If the contents of <i>Da</i> are less than the contents <i>Db/const4</i> , then add the value specified by <i>disp15</i> , multiplied by two and sign-extended to 32 bits, to the contents of the PC, and jump to that address. The operands are treated as signed/unsigned, 32-bit integers. The <i>const4</i> value is sign-extended/zero-extended to 32 bits. |
| <b>Operation</b>   | if (D[a]<D[b]) then PC = PC+sign_ext(2 * disp15); signed<br>if (D[a]<sign_ext(const4)) then PC = PC+sign_ext(2 * disp15); signed<br><br>if (D[a]<D[b]) then PC = PC+sign_ext(2 * disp15); unsigned<br>if (D[a]<zero_ext(const4)) then PC = PC+sign_ext(2 * disp15); unsigned   |
| <b>Status</b>      | -  |
| <b>Examples</b>    | jlt    d1, d2, foobar<br>jlt    d1, 6, foobar<br>jlt.u   d1, d2, foobar<br>jlt.u   d1, 6, foobar   |
| <b>See also</b>    | JEQ, JGE, JGE.U, JNE   |

#### 10.4.82 Jump if Less Than Zero .... JLTZ

**Table 10-92**
**JLTZ**

|                    |  |
|--------------------|--|
| <b>Syntax</b>      | jltz      Db, disp4 (SBR)  |
| <b>Description</b> | If the contents of <i>Db</i> are less than zero, then add the value specified by <i>disp4</i> , multiplied by two and zero-extended to 32 bits, to the contents of the PC, and jump to that address. |
| <b>Operation</b>   | if (D[b] < 0) then (PC = PC + zero_ext(2 * disp4))   |
| <b>Status</b>      | -  |
| <b>Examples</b>    | jltz    d2, foobar   |
| <b>See also</b>    | JGEZ, JGTZ, JLEZ, JNZ, JZ  |

2001-04-30 @ 15:16

#### 10.4.83 Jump if Not Equal .....JNE

**Table 10-93**  
**JNE**

|                    |   |                          |
|--------------------|---|--------------------------|
| <b>Syntax</b>      | jne   | Da, Db, disp15 (BRR)     |
|                    | jne   | Da, const4, disp15 (BRC) |
|                    | jne   | D15, Db, disp4 (SBR)     |
|                    | jne   | D15, const4, disp4 (SBC) |
| <b>Description</b> | If the contents of <i>Da</i> are not equal to the contents <i>Db/const4</i> , then add the value specified by <i>disp15</i> , multiplied by two and sign-extended to 32 bits, to the contents of the PC, and jump to that address. The <i>const4</i> value is sign-extended to 32 bits. |                          |
|                    | If the contents of D15 are not equal to the contents <i>Db/const4</i> , then add the value specified by <i>disp4</i> , multiplied by two and sign-extended to 32 bits, to the contents of the PC, and jump to that address. The <i>const4</i> value is sign-extended to 32 bits.        |                          |
| <b>Operation</b>   | 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)   |                          |
| <b>Status</b>      | -   |                          |
| <b>Examples</b>    | jne   | d1, d2, foobar           |
|                    | jne   | d1, 6, foobar            |
|                    | jne   | d15, d2, foobar          |
|                    | jne   | d15, 6, foobar           |
| <b>See also</b>    | JEQ, JGE, JGE.U, JLT, JLT.U   |                          |

#### 10.4.84 Jump if Not Equal Address .....JNE.A

**Table 10-94**  
**JNE.A**

|                    |   |                      |
|--------------------|---|----------------------|
| <b>Syntax</b>      | jne.a   | Aa, Ab, disp15 (BRR) |
| <b>Description</b> | If the contents of <i>Aa</i> are not equal to the contents <i>Ab</i> , then add the value specified by <i>disp15</i> , multiplied by two and sign-extended to 32 bits, to the contents of the PC, and jump to that address. |                      |
| <b>Operation</b>   | if (A[a] != A[b]) then PC = PC+sign_ext(2 * disp15)   |                      |
| <b>Status</b>      | -   |                      |
| <b>Examples</b>    | jne.a a4, a2, foobar  |                      |
| <b>See also</b>    | JEQ.A   |                      |



**10.4.85 Jump if Not Equal and Decrement. . . . . JNED**
**Table 10-95  
JNED**

|                    |  |
|--------------------|--|
| <b>Syntax</b>      | jned     Da, Db, disp15 (BRR)<br>jned     Da, const4, disp15 (BRC)   |
| <b>Description</b> | If the contents of <i>Da</i> are not equal to the contents <i>Db/const4</i> , then add the value specified by <i>disp15</i> , multiplied by two and sign-extended to 32 bits, to the contents of the PC, and jump to that address. Decrement the value in <i>Da</i> by 1. The <i>const4</i> value is sign-extended to 32 bits. |
| <b>Operation</b>   | if (D[a] != D[b]) then PC = PC+sign_ext(2 * disp15); D[a] = D[a]-1<br>if (D[a] != sign_ext(const4)) then PC = PC+sign_ext(2 * disp15); D[a] = D[a]-1   |
| <b>Status</b>      | -  |
| <b>Examples</b>    | jned d1, d2,     foobar<br>jned d1, 6,     foobar  |
| <b>See also</b>    | JNEI, LOOP, LOOPU  |

**Note :** The decrement is unconditional.

**10.4.86 Jump if Not Equal and Increment . . . . . JNEI**
**Table 10-96  
JNEI**

|                    |  |
|--------------------|--|
| <b>Syntax</b>      | jnei     Da, Db, disp15 (BRR)<br>jnei     Da, const4, disp15 (BRC)   |
| <b>Description</b> | If the contents of <i>Da</i> are not equal to the contents <i>Db/const4</i> , then add the value specified by <i>disp15</i> , multiplied by two and sign-extended to 32 bits, to the contents of the PC, and jump to that address. Increment the value in <i>Da</i> by 1. The <i>const4</i> value is sign-extended to 32 bits. |
| <b>Operation</b>   | if (D[a] != D[b]) then PC = PC+ sign_ext(2 * disp15); D[a] = D[a]+1<br>if (D[a] != sign_ext(const4)) then PC = PC + sign_ext(2 * disp15); D[a] = D[a]+1  |
| <b>Status</b>      | -  |
| <b>Examples</b>    | jnei     d1, d2, foobar<br>jnei     d1, 6, foobar  |
| <b>See also</b>    | JNED, LOOP, LOOPU  |

**Note :** The increment is unconditional.

2001-04-30 @ 15:16

**10.4.87 Jump if Not Equal to Zero ..... JNZ**

**Table 10-97**  
**JNZ**

|                    |   |
|--------------------|---|
| <b>Syntax</b>      | jnz Db, disp4 (SBR)<br>jnz D15, disp8 (SB)  |
| <b>Description</b> | If contents of <i>Db/D15</i> are not equal to zero, then add value specified by <i>disp4/disp8</i> , multiplied by two and zero-/sign-extended to 32 bits, to contents of PC, and jump to that address. |
| <b>Operation</b>   | if (D[b] != 0) then PC = PC + zero_ext(2 * disp4)<br>if (D[15] != 0) then PC = PC + sign_ext(2 * disp8)   |
| <b>Status</b>      | -   |
| <b>Examples</b>    | jnz d2, foobar<br>jnz d15, foobar   |
| <b>See also</b>    | JGEZ, JGTZ, JLEZ, JLTZ, JZ  |

**10.4.88 Jump if Not Equal to Zero Address ..... JNZ.A**

**Table 10-98**  
**JNZ.A**

|                    |   |
|--------------------|---|
| <b>Syntax</b>      | jnz.a Aa, disp15 (BRR)<br>jnz.a Aa, disp4 (SBR)   |
| <b>Description</b> | If the contents of <i>Aa</i> are not equal to zero, then add the value specified by <i>disp15</i> , multiplied by two and sign-extended to 32 bits, to the contents of the PC, and jump to that address.<br><br>If the contents of <i>Aa</i> are not equal to zero, then add the value specified by <i>disp4</i> , multiplied by two and zero-extended to 32 bits, to the contents of the PC, and jump to that address. |
| <b>Operation</b>   | if (A[b] != 0) then PC = PC + sign_ext(2 * disp15)<br>if (A[b] != 0) then PC = PC + zero_ext(2 * disp4)   |
| <b>Status</b>      | -   |
| <b>Examples</b>    | jnz.a a4, foobar<br>jnz.a a4, foobar  |
| <b>See also</b>    | JZ.A  |

2001-04-30 @ 15:16

#### 10.4.89 Jump if Not Equal to Zero Bit. . . . . JNZ.T

**Table 10-99  
JNZ.T**

|                    |   |
|--------------------|---|
| <b>Syntax</b>      | jnz.t    Da, n, disp15 (BRN)  |
|                    | jnz.t    D15, n, disp4 (SBRN)   |
| <b>Description</b> | If bit <i>n</i> of register <i>Da</i> is not equal to zero, then add the value specified by <i>disp15</i> , multiplied by two and sign-extended to 32 bits, to the contents of the PC and jump to that address. |
|                    | if bit <i>n</i> of register <i>D15</i> is not equal to zero, then add the value specified by <i>disp4</i> , multiplied by two and zero-extended to 32 bits, to the contents of the PC and jump to that address. |
| <b>Operation</b>   | 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 – 15)   |
| <b>Status</b>      | -   |
| <b>Examples</b>    | jnz.t    d1, 1, foobar<br>Note: some assemblers require the format:<br>jnz.t    d1: 1, foobar   |
|                    | jnz.t    d15, 1, foobar   |
| <b>See also</b>    | JZ.T  |

#### 10.4.90 Jump if Zero. . . . . JZ

**Table 10-100  
JZ**

|                    |  |
|--------------------|--|
| <b>Syntax</b>      | jz        Db, disp4 (SBR)  |
|                    | jz        D15, disp8 (SB)  |
| <b>Description</b> | If contents of D15/ <i>Db</i> are equal to zero, then add the value specified by <i>disp8/disp4</i> , multiplied by two and sign-extended/zero-extended to 32 bits, to the contents of the PC, and jump to that address. |
| <b>Operation</b>   | if (D[b] == 0) then PC = PC + zero_ext(2 * disp4)<br>if (D[15] == 0) then PC = PC + sign_ext(2 * disp8)  |
| <b>Status</b>      | -  |
| <b>Examples</b>    | jz        d2, foobar   |
|                    | jz        d15, foobar  |
| <b>See also</b>    | JGEZ, JGTZ, JLEZ, JLTZ, JNZ  |

2001-04-30 @ 15:16

#### 10.4.91 Jump if Zero Address . . . . . JZ.A

**Table 10-101**  
**JZ.A**

|                    |   |
|--------------------|---|
| <b>Syntax</b>      | jz.a    Aa, disp15 (BRR)  |
|                    | jz.a    Ab, disp4 (SBR)   |
| <b>Description</b> | If contents of <i>Aa</i> are equal to zero, then add value specified by <i>disp15</i> , multiplied by two and sign-extended to 32 bits, to contents of PC and jump to that address. |
|                    | If contents of <i>Ab</i> are equal to zero, then add value specified by <i>disp4</i> , multiplied by two and zero-extended to 32 bits, to contents of PC and jump to that address.  |
| <b>Operation</b>   | if (A[a] == 0) then PC = PC + sign_ext(2 * disp15)  |
|                    | if (A[b] == 0) then PC = PC + zero_ext(2 * disp4)   |
| <b>Status</b>      | -   |
| <b>Examples</b>    | jz.a    a4, foobar  |
|                    | jz.a    a2, foobar  |
| <b>See also</b>    | JNZ.A   |

#### 10.4.92 Jump if Zero Bit. . . . . JZ.T

**Table 10-102**  
**JZ.T**

|                    |  |
|--------------------|--|
| <b>Syntax</b>      | jz.t    Da, n, disp15 (BRN)  |
|                    | jz.t    D15, n, disp4 (SBRN)   |
| <b>Description</b> | If bit <i>n</i> of register <i>Da</i> is equal to zero, then add value specified by <i>disp15</i> , multiplied by two and sign-extended to 32 bits, to contents of PC, and jump to that address. |
|                    | If bit <i>n</i> of register <i>D15</i> is equal to zero, then add value specified by <i>disp4</i> , multiplied by two and zero-extended to 32 bits, to contents of PC, and jump to that address. |
| <b>Operation</b>   | 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 – 15)   |
| <b>Status</b>      | -  |
| <b>Examples</b>    | jz.t    d1, 1, foobar  |
|                    | Note: some assemblers require the format:<br>jz.t    d1: 1, foobar   |
|                    | jz.t    d15, 1, foobar   |
| <b>See also</b>    | JNZ.T  |

**10.4.93 Load Word to Address Register ..... LD.A**
**Table 10-103**
**LD.A**

|                    |  |
|--------------------|--|
| <b>Syntax</b>      | ld.a    Aa, <mode>   |
| <b>Description</b> | Load word contents of memory location specified by addressing mode into address register Aa. |
| <b>Operation</b>   | $A[a] = M(EA, \text{word})$  |
| <b>Status</b>      | -  |
| <b>Examples</b>    | -  |
| <b>See also</b>    | LD.B, LD.BU, LD.D, LD.DA, LD.H, LD.HU, LD.Q, LD.W,   |

**Table 10-104**
**LD.A Operation**

| <mode>                     | Syntax       | Effective Address                          | Instruction Format |
|----------------------------|--------------|--|--------------------|
| <b>Absolute</b>            | offset       | {offset18[17:14], 14b'0, offset18[13:0]}   | ABS                |
| <b>Base + Short Offset</b> | [An]offset   | $A[b] + \text{sign\_ext}(\text{offset}10)$ | BO                 |
| <b>Base + Long Offset</b>  | [An]offset   | $A[b] + \text{sign\_ext}(\text{offset}16)$ | BOL                |
| <b>Pre-increment</b>       | [+An]offset  | $A[b] + \text{sign\_ext}(\text{offset}10)$ | BO                 |
| <b>Post-increment</b>      | [An+]offset  | $A[b]$                                     | BO                 |
| <b>Circular</b>            | [An+c]offset | $A[b] + A[b+1][15:0]$ (b is even)          | BO                 |
| <b>Bit-reverse</b>         | [An+r]       | $A[b] + A[b+1][15:0]$ (b is even)          | BO                 |

**Note :** If the target register is modified by the addressing mode, the result is undefined.

2001-04-30 @ 15:16

**10.4.94 Load Word to Address Register (16-bit) . . . . . LD.A**

**Table 10-105**  
**LD.A (16-bit)**

|                    |   |
|--------------------|---|
| <b>Syntax</b>      | ld.a Aa, [Ab] (SLR) Register indirect<br>ld.a Aa, [A15] offset4 (SLRO) Implicit base + offset<br>ld.a A15, [Ab] offset4 (SRO) Implicit destination register<br>ld.a A15, [A10] offset8 (SC) Stack pointer + offset<br>ld.a Aa, [Ab+] (SLR) Post-increment   |
| <b>Description</b> | Load word contents of memory location specified by addressing mode into address register <i>Aa/A15</i> .  |
| <b>Operation</b>   | $A[a] = M(A[b], \text{word})$<br>$A[a] = M(A[15] + \text{zero\_ext}(4 * \text{offset4}), \text{word})$<br>$A[15] = M(A[b] + \text{zero\_ext}(4 * \text{offset4}), \text{word})$<br>$A[15] = M(A[10] + \text{zero\_ext}(4 * \text{offset8}), \text{word})$<br>$A[a] = M(A[b], \text{word}); A[b] = A[b] + 4$ |
| <b>Status</b>      | -   |
| <b>Examples</b>    | -   |
| <b>See also</b>    | LD.BU, LD.H, LD.W   |

**Note :** If the target register is modified by the addressing mode, the result is undefined.

**10.4.95 Load Byte . . . . . LD.B**  
**Load Byte Unsigned . . . . . LD.BU**

**Table 10-106**  
**LD.B & LD.BU**

|                    |  |
|--------------------|--|
| <b>Syntax</b>      | ld.b Da, <mode><br>ld.bu Da, <mode>  |
| <b>Description</b> | Load the byte contents of the memory location specified by the addressing mode, sign-extended/zero-extended to 32 bits, into data register <i>Da</i> . |
| <b>Operation</b>   | $D[a] = \text{sign\_ext}(M(EA, \text{byte}))$<br>$D[a] = \text{zero\_ext}(M(EA, \text{byte}))$<br>(See <a href="#">Table 10-107</a> )                  |
| <b>Status</b>      | -  |
| <b>Examples</b>    | -  |
| <b>See also</b>    | LD.A, LD.D, LD.DA, LD.H, LD.HU, LD.Q, LD.W   |

2001-04-30 @ 15:16

**Table 10-107**  
**LD.B & LD.BU Operation**

| <mode>                     | Syntax       | Effective Address                        | Instruction Format |
|----------------------------|--------------|--|--------------------|
| <b>Absolute</b>            | offset       | {offset18[17:14], 14'b0, offset18[13:0]} | ABS                |
| <b>Base + Short Offset</b> | [An]offset   | A[b]+sign_ext(offset10)                  | BO                 |
| <b>Pre-increment</b>       | [+An]offset  | A[b]+sign_ext(offset10)                  | BO                 |
| <b>Post-increment</b>      | [An+]offset  | A[b]                                     | BO                 |
| <b>Circular</b>            | [An+c]offset | A[b]+A[b+1][15:0] (b is even)            | BO                 |
| <b>Bit-reverse</b>         | [An+r]       | A[b]+A[b+1][15:0] (b is even)            | BO                 |

#### 10.4.96 Load Byte Unsigned (16-bit) .....LD.BU

**Table 10-108**  
**LD.BU (16-bit)**

|                    |  |
|--------------------|--|
| <b>Syntax</b>      | ld.bu    Da, [Ab]                      (SLR)                      Register indirect<br>id.bu    Da, [A15] offset4        (SLRO)                  Implicit base + offset<br>id.bu    D15, [Ab] offset4        (SRO)                    Implicit destination register<br>id.bu    Da, [Ab+]                    (SLR)                    Post-increment |
| <b>Description</b> | Load the byte contents of the memory location specified by the addressing mode, zero-extended to 32 bits, into data register <i>Da/D15</i> .   |
| <b>Operation</b>   | D[a] = zero_ext(M(A[b], byte, byte)<br>D[a] = zero_ext(M(A[15]+ zero_ext (offset4), byte)<br>D[15] = zero_ext(M(A[b] +zero_ext (offset4), byte)<br>D[a] = zero_ext(M(A[b], byte);A[b]+A[b]+1   |
| <b>Status</b>      | -  |
| <b>Examples</b>    | -  |
| <b>See also</b>    | LD.A, LD.H, LD.W   |

#### 10.4.97 Load Doubleword .....LD.D

**Table 10-109**  
**LD.D**

|                    |   |
|--------------------|---|
| <b>Syntax</b>      | ld.d    Ea, <mode>  |
| <b>Description</b> | Load doubleword contents of memory location specified by addressing mode into extended data register <i>Ea</i> . Least-significant word of doubleword value is loaded into even register ( <i>Dn</i> ) and most-significant word is loaded into odd register ( <i>Dn+1</i> ). |

2001-04-30 @ 15:16

**Table 10-109**  
**LD.D**

|                  |   |
|------------------|---|
| <b>Operation</b> | $E[a] = M(EA, \text{doubleword})$ (See <a href="#">Table 10-110</a> ) |
| <b>Status</b>    | -   |
| <b>Examples</b>  | -   |
| <b>See also</b>  | LD.A, LD.B, LD.BU, LD.DA, LD.H, LD.HU, LD.Q, LD.W                     |

**Table 10-110**  
**LD.D Operation**

| <mode>                     | Syntax       | Effective Address                          | Instruction Format |
|----------------------------|--------------|--|--------------------|
| <b>Absolute</b>            | offset       | {offset18[17:14], 14'b0, offset18[13:0]}   | ABS                |
| <b>Base + Short Offset</b> | [An]offset   | $A[b] + \text{sign\_ext}(\text{offset}10)$ | BO                 |
| <b>Pre-increment</b>       | [+An]offset  | $A[b] + \text{sign\_ext}(\text{offset}10)$ | BO                 |
| <b>Post-increment</b>      | [An+]offset  | $A[b]$                                     | BO                 |
| <b>Circular</b>            | [An+c]offset | $A[b] + A[b+1][15:0]$ (b is even)          | BO                 |
| <b>Bit-reverse</b>         | [An+r]       | $A[b] + A[b+1][15:0]$ (b is even)          | BO                 |

### 10.4.98 Load Doubleword to Address Register. . . . .LD.DA

**Table 10-111**  
**LD.DA**

|                    |  |
|--------------------|--|
| <b>Syntax</b>      | ld.da Aa, <mode>   |
| <b>Description</b> | Load doubleword contents of memory location specified by addressing mode into address register pair Aa. Least-significant word of doubleword value is loaded into even register (An) and most-significant word is loaded into odd register (An+1). |
| <b>Operation</b>   | $A[a](\text{pair}) = M(EA, \text{doubleword})$ (See <a href="#">Table 10-112</a> )   |
| <b>Status</b>      | -  |
| <b>Examples</b>    | -  |
| <b>See also</b>    | LD.A, LD.B, LD.BU, LD.D, LD.H, LD.HU, LD.Q, LD.W   |

**Table 10-112**  
**LD.DA Operation**

| <mode>                     | Syntax     | Effective Address                          | Instruction Format |
|----------------------------|------------|--|--------------------|
| <b>Absolute</b>            | offset     | {offset18[17:14], 14'b0, offset18[13:0]}   | ABS                |
| <b>Base + Short Offset</b> | [An]offset | $A[b] + \text{sign\_ext}(\text{offset}10)$ | BO                 |



**Table 10-112**  
**LD.DA Operation**

| <mode>                | Syntax       | Effective Address             | Instruction Format |
|-----------------------|--------------|-------------------------------|--------------------|
| <b>Pre-increment</b>  | [+An]offset  | A[b]+sign_ext(offset10)       | BO                 |
| <b>Post-increment</b> | [An+]offset  | A[b]                          | BO                 |
| <b>Circular</b>       | [An+c]offset | A[b]+A[b+1][15:0] (b is even) | BO                 |
| <b>Bit-reverse</b>    | [An+r]       | A[b]+A[b+1][15:0] (b is even) | BO                 |

**10.4.99 Load Halfword . . . . . LD.H**  
**Load Halfword Unsigned . . . . . LD.HU**

**Table 10-113**  
**LD.H & LD.HU**

|                    |  |
|--------------------|--|
| <b>Syntax</b>      | ld.h     Da, <mode><br>ld.hu    Da, <mode>   |
| <b>Description</b> | Load the halfword contents of the memory location specified by the addressing mode, sign-extended/zero-extended to 32 bits, into data register Da. |
| <b>Operation</b>   | D[a] = sign_ext(M(EA, halfword))<br>D[a] = zero_ext(M(EA, halfword))   |
| <b>Status</b>      | -  |
| <b>Examples</b>    | -  |
| <b>See also</b>    | LD.A, LD.B, LD.BU, LD.D, LD.DA, LD.Q, LD.W   |

**Table 10-114**  
**LD.H & LD.HU Operation**

| <mode>                     | Syntax       | Effective Address                        | Instruction Format |
|----------------------------|--------------|--|--------------------|
| <b>Absolute</b>            | offset       | {offset18[17:14], 14'b0, offset18[13:0]} | ABS                |
| <b>Base + Short Offset</b> | [An]offset   | A[b]+sign_ext(offset10)                  | BO                 |
| <b>Pre-increment</b>       | [+An]offset  | A[b]+sign_ext(offset10)                  | BO                 |
| <b>Post-increment</b>      | [An+]offset  | A[b]                                     | BO                 |
| <b>Circular</b>            | [An+c]offset | A[b]+A[b+1][15:0] (b is even)            | BO                 |
| <b>Bit-reverse</b>         | [An+r]       | A[b]+A[b+1][15:0] (b is even)            | BO                 |

2001-04-30 @ 15:16

#### 10.4.100 Load Halfword (16-bit) ..... LD.H

Table 10-115  
LD.H (16-bit)

|                    |  |
|--------------------|--|
| <b>Syntax</b>      | ld.h    Da, [Ab]            (SLR)            Register indirect<br>id.h    Da, [A15] offset4 (SLRO)          Implicit base + offset<br>id.h    D15, [Ab] offset4 (SRO)            Implicit destination register<br>id.h    Da, [Ab+]            (SLR)            Post-increment |
| <b>Description</b> | Load the halfword contents of the memory location specified by the addressing mode, zero-extended to 32 bits, into data register <i>Da/D15</i>   |
| <b>Operation</b>   | D[a] = sign_ext(M(A[b], halfword)<br>D[a] = sign_ext(M(A[15]+ zero_ext (2*offset4), halfword)<br>D[15] = sign_ext(M(A[b] +zero_ext (2*offset4), halfword)<br>D[a] = sign_ext(M(A[b], halfword);A[b]=A[b]+2   |
| <b>Status</b>      | -  |
| <b>Examples</b>    | -  |
| <b>See also</b>    | LD.A, LD.H, LD.W   |

#### 10.4.101 Load Halfword Signed Fraction ..... LD.Q

Table 10-116  
LD.Q

|                    |  |
|--------------------|--|
| <b>Syntax</b>      | ld.q    Da, <mode>   |
| <b>Description</b> | Load halfword contents of memory location specified by addressing mode into most-significant halfword of data register <i>Da</i> , setting 16 least-significant bits of <i>Da</i> to zero. |
| <b>Operation</b>   | D[a] = {M(EA, halfword), 16'h 0000} (See <a href="#">Table 10-117</a> )  |
| <b>Status</b>      | -  |
| <b>Examples</b>    | -  |
| <b>See also</b>    | LD.A, LD.D, LD.DA, LD.B, LD.BU, LD.H, LD.HU, LD.W  |

Table 10-117  
LD.Q Operation

| <mode>                     | Syntax     | Effective Address                        | Instruction Format |
|----------------------------|------------|--|--------------------|
| <b>Absolute</b>            | offset     | {offset18[17:14], 14'b0, offset18[13:0]} | ABS                |
| <b>Base + Short Offset</b> | [An]offset | A[b]+sign_ext(offset10)                  | BO                 |

2001-04-30 @ 15:16

**Table 10-117  
LD.Q Operation**

| <mode>                | Syntax       | Effective Address             | Instruction Format |
|-----------------------|--------------|-------------------------------|--------------------|
| <b>Pre-increment</b>  | [+An]offset  | A[b]+sign_ext(offset10)       | BO                 |
| <b>Post-increment</b> | [An+]offset  | A[b]                          | BO                 |
| <b>Circular</b>       | [An+c]offset | A[b]+A[b+1][15:0] (b is even) | BO                 |
| <b>Bit-reverse</b>    | [An+r]       | A[b]+A[b+1][15:0] (b is even) | BO                 |

**10.4.102 Load Word .....LD.W**
**Table 10-118  
LD.W**

|                    |   |
|--------------------|---|
| <b>Syntax</b>      | ld.w    Da,<mode>   |
| <b>Description</b> | Load word contents of memory location specified by addressing mode into data register <i>Da</i> . |
| <b>Operation</b>   | D[a] = M(EA, word)  |
| <b>Status</b>      | -   |
| <b>Examples</b>    | -   |
| <b>See also</b>    | LD.A, LD.D, LD.DA, LD.B, LD.BU, LD.H, LD.HU, LD.Q   |

**Table 10-119  
LD.W Operation**

| <mode>                     | Syntax       | Effective Address                        | Instruction Format |
|----------------------------|--------------|--|--------------------|
| <b>Absolute</b>            | offset       | {offset18[17:14], 14'b0, offset18[13:0]} | ABS                |
| <b>Base + Short Offset</b> | [An]offset   | A[b]+sign_ext(offset10)                  | BO                 |
| <b>Base + Long Offset</b>  | [An]offset   | A[b]+sign_ext(offset16)                  | BOL                |
| <b>Pre-increment</b>       | [+An]offset  | A[b]+sign_ext(offset10)                  | BO                 |
| <b>Post-increment</b>      | [An+]offset  | A[b]                                     | BO                 |
| <b>Circular</b>            | [An+c]offset | A[b]+A[b+1][15:0] (b is even)            | BO                 |
| <b>Bit-reverse</b>         | [An+r]       | A[b]+A[b+1][15:0] (b is even)            | BO                 |

2001-04-30 @ 15:16

#### 10.4.103 Load Word (16-bit) .....LD.W

Table 10-120

##### LD.W (16-bit)

|                    |   |
|--------------------|---|
| <b>Syntax</b>      | ld.w Da, [Ab] (SCR) Register indirect<br>ld.w Da, [A15] offset4 (SLRO) Implicit base + offset<br>ld.w D15, [Ab] offset4 (SRO) Implicit destination register<br>ld.w D15, [A10] offset8 (SC) Stack pointer + offset<br>ld.w Da, [Ab+] (SLR) Post-increment |
| <b>Description</b> | Load word contents of memory location specified by addressing mode into data register <i>Da/D15</i> .   |
| <b>Operation</b>   | D[a] = M(A[b], word)<br>D[a] = M(A[15]+ zero_ext(4*offset4), word)<br>D[a] = M(A[b]+ zero_ext(4*offset4), word)<br>D[a] = M(A[10]+ zero_ext(4*offset8), word)<br>D[a] = M(A[b], word); A[b]=A[b]+4  |
| <b>Status</b>      | -   |
| <b>Examples</b>    | -   |
| <b>See also</b>    | LD.A, LD.BU, LD.H   |

#### 10.4.104 Load Lower Context .....LDLCX

Table 10-121

##### LDLCX

|                    |   |
|--------------------|---|
| <b>Syntax</b>      | ldlcx <mode>  |
| <b>Description</b> | 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.<br><br>Note that the effective address specified by the addressing mode must be aligned on a 16-word boundary. For this instruction, the addressing mode is restricted to absolute (ABS) or base plus short offset (BO). |
| <b>Operation</b>   | {dummy, dummy, A[2:3], D[0:3], A[4:7], D[4:7]} = M(EA, 16-word)   |
| <b>Status</b>      | -   |
| <b>Examples</b>    | -   |
| <b>See also</b>    | LDUCX, RSLCX, STLCX, STUCX, SVLCX   |

Table 10-122

##### LDLCX Operation

| <mode>                     | Syntax     | Effective Address                        | Instruction Format |
|----------------------------|------------|--|--------------------|
| <b>Absolute</b>            | offset     | {offset18[17:14], 14'b0, offset18[13:0]} | ABS                |
| <b>Base + Short Offset</b> | [An]offset | A[b]+sign_ext(offset10)                  | BO                 |

#### 10.4.105 Load-Modify-Store ..... LDMST

Table 10-123

##### LDMST

|                    |  |
|--------------------|--|
| <b>Syntax</b>      | ldmst <mode>, Ea   |
| <b>Description</b> | The atomic Load-Modify-Store implements a store under a mask of a value to the memory word, whose address is specified by the addressing mode. Only those bits of the value Ea(lower), where the corresponding bits in the mask Ea(upper) are set are stored into memory. The value and mask may be generated using the IMASK instruction. |
| <b>Operation</b>   | $M(Ea, \text{word}) = (M(Ea, \text{word}) \text{ AND } !Ea(\text{upper})) \text{ OR } (Ea(\text{lower}) \text{ AND } Ea(\text{upper}))$<br>(See Table 10-124)  |
| <b>Status</b>      | -  |
| <b>Examples</b>    | -  |
| <b>See also</b>    | IMASK, ST.T  |

Table 10-124

##### LDMST Operation

| <mode>                     | Syntax       | Effective Address                        | Instruction Format |
|----------------------------|--------------|--|--------------------|
| <b>Absolute</b>            | offset       | {offset18[17:14], 14'b0, offset18[13:0]} | ABS                |
| <b>Base + Short Offset</b> | [An]offset   | A[b]+sign_ext(offset10)                  | BO                 |
| <b>Pre-increment</b>       | [+An]offset  | A[b]+sign_ext(offset10)                  | BO                 |
| <b>Post-increment</b>      | [An+]offset  | A[b]                                     | BO                 |
| <b>Circular</b>            | [An+c]offset | A[b]+A[b+1][15:0] (b is even)            | BO                 |
| <b>Bit-reverse</b>         | [An+r]       | A[b]+A[b+1][15:0] (b is even)            | BO                 |

2001-04-30 @ 15:16

**10.4.106 Load Upper Context ..... LDUCX**

**Table 10-125**  
**LDUCX**

|                    |   |
|--------------------|---|
| <b>Syntax</b>      | lducx <mode>  |
| <b>Description</b> | <p>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.</p> <p>Note that the effective address specified by the addressing mode must be aligned on a 16-word boundary. For this instruction, the addressing mode is restricted to absolute (ABS) or base plus short offset (BO).</p> |
| <b>Operation</b>   | {dummy, dummy, A[10:11], D[8:11], A[12:15], D[12:15]} = M(EA, 16-word)    See <a href="#">Table 10-126</a>  |
| <b>Status</b>      | -   |
| <b>Examples</b>    | -   |
| <b>See also</b>    | LDLCX , RSLCX, STLCX, STUCX, SVLCX  |

**Table 10-126**  
**LDUCX Operation**

| <mode>                     | Syntax     | Effective Address                        | Instruction Format |
|----------------------------|------------|--|--------------------|
| <b>Absolute</b>            | constant   | {offset18[17:14], 14'b0, offset18[13:0]} | ABS                |
| <b>Base + Short Offset</b> | [An]offset | A[a]+sign_ext(offset10)                  | BO                 |

**10.4.107 Load Effective Address ..... LEA**

**Table 10-127**  
**LEA**

|                    |  |
|--------------------|--|
| <b>Syntax</b>      | lea Aa, <mode>   |
| <b>Description</b> | <p>Compute the absolute (effective) address defined by the addressing mode and put the result in address register Aa.</p> <p>Note: For this instruction the auto-increment addressing modes are not supported.</p> |
| <b>Operation</b>   | A[a] = EA (See <a href="#">Table 10-128</a> )  |
| <b>Status</b>      | -  |
| <b>Examples</b>    | -  |
| <b>See also</b>    | MOV.A, MOV.D, MOVH.A   |

2001-04-30 @ 15:16

**Table 10-128**  
**LEA Operation**

| <mode>                     | Syntax     | Effective Address                        | Instruction Format |
|----------------------------|------------|--|--------------------|
| <b>Absolute</b>            | constant   | {offset18[17:14], 14'b0, offset18[13:0]} | ABS                |
| <b>Base + Short Offset</b> | [Ab]offset | A[b]+sign_ext(offset10)                  | BO                 |
| <b>Base + Long Offset</b>  | [Ab]offset | A[b]+sign_ext(offset16)                  | BOL                |

#### 10.4.108 Loop ..... LOOP

**Table 10-129**  
**LOOP**

|                    |   |
|--------------------|---|
| <b>Syntax</b>      | loop Aa, disp15 (BRR)   |
|                    | loop Aa, disp4 (SBR)  |
| <b>Description</b> | <p>If address register Aa is not equal to zero, then add value specified by <i>disp15</i>, multiplied by two and sign-extended to 32 bits, to contents of PC, and jump to that address. The address register is decremented unconditionally</p> <p>If address register Aa is not equal to zero, then add value specified by <i>disp4</i>, multiplied by two and one-extended to a 32-bit negative number, to contents of PC, and jump to that address. Address register is decremented unconditionally.</p> |
| <b>Operation</b>   | <p>if (A[a] != 0) then PC = PC + sign_ext(2 * disp15); A[a] = A[a]-1</p> <p>if (A[a] != 0) then PC = PC + one_ext(2 * disp4); A[a] = A[a]-1</p>   |
| <b>Status</b>      | -   |
| <b>Examples</b>    | <p>loop a4, iloop</p> <p>loop a4, iloop</p>   |
| <b>See also</b>    | JNED, JNEI, LOOPU   |

#### 10.4.109 Loop Unconditional ..... LOOPU

**Table 10-130**  
**LOOPU**

|                    |   |
|--------------------|---|
| <b>Syntax</b>      | loopu disp15 (BRR)  |
| <b>Description</b> | Add value specified by <i>disp15</i> multiplied by two and sign-extended to 32 bits, to contents of PC, and jump to that address. |
| <b>Operation</b>   | PC = PC + sign_ext(2 * disp15)  |
| <b>Status</b>      | -   |
| <b>Examples</b>    | loopu iloop   |
| <b>See also</b>    | J, JA, JI, JL, JLA, JLI   |

2001-04-30 @ 15:16

**10.4.110 Less Than .....LT**  
**Less Than Unsigned. .... LT.U**

**Table 10-131**

**LT & LT.U**

|                    |   |                       |
|--------------------|---|-----------------------|
| <b>Syntax</b>      | lt  | Dc, Da, Db (RR)       |
|                    | lt  | Dc, Da, const9 (RC)   |
|                    | lt.u  | Dc, Da, Db (RR)       |
|                    | lt.u  | Dc, Da, const9 (RC)   |
|                    | lt  | D15, Da, Db (SRR)     |
|                    | lt  | D15, Da, const4 (SRC) |
| <b>Description</b> | If the contents of data register <i>Da</i> are less than the contents of data register <i>Db/const9</i> , set the least-significant bit of <i>Dc</i> to 1 and clear the remaining bits to zero; otherwise, clear all bits in <i>Dc</i> . The operands are treated as signed integers, and the <i>const9</i> value is sign-extended to 32 bits.          |                       |
|                    | If the contents of data register <i>Da</i> are less than the contents of data register <i>Db/const9</i> , set the least-significant bit of <i>Dc</i> . The operands are treated as unsigned integers, and the <i>const9</i> value is zero-extended to 32 bits.  |                       |
|                    | If the contents of data register <i>Da</i> are less than the contents of data register <i>Db/const4</i> , set the least-significant bit of <i>D15</i> to 1 and clear the remaining bits to zero; otherwise, clear all bits in <i>D15</i> . The operands are treated as signed 32-bit integers, and the <i>const4</i> value is sign-extended to 32 bits. |                       |
| <b>Operation</b>   | D[c] = (D[a] < D[b]); signed<br>D[c] = (D[a] < sign_ext(const9)); signed  |                       |
|                    | D[c] = (D[a] < D[b]); unsigned<br>D[c] = (D[a] < zero_ext(const9)); unsigned  |                       |
|                    | D[15] = (D[a] < D[b]); signed<br>D[15] = (D[a] < sign_ext(const4)); signed  |                       |
| <b>Status</b>      | -   |                       |
| <b>Examples</b>    | lt  | d3, d1, d2            |
|                    | lt  | d3, d1, 126           |
|                    | lt.u  | d3, d1, d2            |
|                    | lt.u  | d3, d1, 253           |
|                    | lt  | d15, d1, d2           |
|                    | lt  | d15, d1, 6            |
| <b>See also</b>    | EQ, GE, GE.U, NE  |                       |



2001-04-30 @ 15:16

**10.4.111 Less Than Address . . . . . LT.A**
**Table 10-132**
**LT.A**

|                    |   |
|--------------------|---|
| <b>Syntax</b>      | lt.a      Dc, Aa, Ab (RR)   |
| <b>Description</b> | If contents of address register <i>Aa</i> are less than contents of address register <i>Ab</i> , set least-significant bit of <i>Dc</i> to 1 and clear remaining bits to zero; otherwise, clear all bits in <i>Dc</i> . The operands are treated as unsigned 32-bit integers. |
| <b>Operation</b>   | $D[c] = (A[a] < A[b]); \text{ unsigned}$  |
| <b>Status</b>      | -   |
| <b>Examples</b>    | lt.a      d3, a4, a2  |
| <b>See also</b>    | EQ.A, EQZ.A, GE.A, NE.A, NEZ.A  |

**10.4.112 Less Than Packed Byte . . . . . LT.B**
**Less Than Packed Byte Unsigned . . . . . LT.BU**
**Table 10-133**
**LT.B & LT.BU**

|                    |  |
|--------------------|--|
| <b>Syntax</b>      | lt.b      Dc, Da, Db (RR)<br>lt.bu      Dc, Da, Db (RR)  |
| <b>Description</b> | Compare each byte of data register <i>Da</i> with the corresponding byte of <i>Db</i> . In each case, if the value of the byte in <i>Da</i> is less than the value of the byte in <i>Db</i> , set all bits in the corresponding byte of <i>Dc</i> to 1; otherwise, clear all the bits. The operands are treated as signed 8-bit integers.<br><br>Compare each byte of data register <i>Da</i> with the corresponding byte of <i>Db</i> . In each case, if the value of the byte in <i>Da</i> is less than the value of the byte in <i>Db</i> , set all bits in the corresponding byte of <i>Dc</i> to 1; otherwise, clear all the bits. The operands are treated as unsigned 8-bit integers. |
| <b>Operation</b>   | if ( $D[a][(n+7):n] < D[b][(n+7):n]$ )<br>then $D[c][(n+7):n] = 8'h\ FF$<br>else $D[c][(n+7):n] = 8'h\ 00$ ; $n = 0, 8, 16, 24$ ;signed<br><br>if ( $D[a][(n+7):n] < D[b][(n+7):n]$ )<br>then $D[c][(n+7):n] = 8'h\ FF$<br>else $D[c][(n+7):n] = 8'h\ 00$ ; $n = 0, 8, 16, 24$ ;unsigned   |
| <b>Status</b>      | -  |
| <b>Examples</b>    | lt.b      d3, d1, d2<br><br>lt.bu      d3, d1, d2  |
| <b>See also</b>    | EQ.B, EQ.H, EQ.W, LT.H, LT.HU, LT.W, LT.WU   |

2001-04-30 @ 15:16

### 10.4.113 Less Than Packed Halfword ..... LT.H

### Less Than Packed Halfword Unsigned. .... LT.HU

**Table 10-134**  
**LT.H & LT.HU**

|                    |   |
|--------------------|---|
| <b>Syntax</b>      | lt.h      Dc, Da, Db (RR)<br>lt.hu     Dc, Da, Db (RR)  |
| <b>Description</b> | <p>Compare each halfword of data register <i>Da</i> with the corresponding halfword of <i>Db</i>. In each case, if the value of the halfword in <i>Da</i> is less than the value of the corresponding halfword in <i>Db</i>, set all bits of the corresponding halfword of <i>Dc</i> to 1; otherwise, clear all the bits. The operands are treated as signed 16-bit integers.</p> <p>Compare each halfword of data register <i>Da</i> with the corresponding halfword of <i>Db</i>. In each case, if the value of the halfword in <i>Da</i> is less than the value of the corresponding halfword in <i>Db</i>, set all bits of the corresponding halfword of <i>Dc</i> to 1; otherwise, clear all the bits. The operands are treated as unsigned 16-bit integers.</p> |
| <b>Operation</b>   | <p>if (<math>D[a][(n+15):n] &lt; D[b][(n+15):n]</math>)<br/> then <math>D[c][(n+15):n] = 16'h\ FFFF</math><br/> else <math>D[c][(n+15):n] = 16'h\ 0000</math>; n = 0, 16;signed</p> <p>if (<math>D[a][(n+15):n] &lt; D[b][(n+15):n]</math>)<br/> then <math>D[c][(n+15):n] = 16'h\ FFFF</math><br/> else <math>D[c][(n+15):n] = 16'h\ 0000</math>; n = 0, 16;unsigned</p>   |
| <b>Status</b>      | -   |
| <b>Examples</b>    | lt.h      d3, d1, d2<br>lt.hu     d3, d1, d2  |
| <b>See also</b>    | EQ.B, EQ.H, EQ.W, LT.B, LT.BU, LT.W, LT.WU  |

2001-04-30 @ 15:16

**10.4.114 Less Than Packed Word .....LT.W**  
**Less Than Packed Word Unsigned. .... LT.WU**

**Table 10-135**
**LT.H & LT.HU**

|                    |  |
|--------------------|--|
| <b>Syntax</b>      | lt.w     Dc, Da, Db (RR)<br>lt.wu    Dc, Da, Db (RR)   |
| <b>Description</b> | <p>If the contents of data register <i>Da</i> are less than the contents of data register <i>Db</i>, set all bits in <i>Dc</i> to 1 and clear the remaining bits to zero; otherwise, clear all bits in <i>Dc</i>. <i>Da</i> and <i>Db</i> are treated as signed 32-bit integers.</p> <p>If the contents of data register <i>Da</i> are less than the contents of data register <i>Db</i>, set all bits in <i>Dc</i> to 1 and clear the remaining bits to zero; otherwise, clear all bits in <i>Dc</i>. The operands are treated as unsigned 32-bit integers.</p> |
| <b>Operation</b>   | <p>if (D[a] &lt; D[b])<br/> then D[c] = 32'h FFFFFFFF<br/> else D[c] = 32'h 00000000;signed</p> <p>if (D[a] &lt; D[b])<br/> then D[c] = 32'h FFFFFFFF<br/> else D[c] = 32'h 00000000;unsigned</p>  |
| <b>Status</b>      | -  |
| <b>Examples</b>    | lt.w     d3, d1, d2<br><br>lt.wu    d3, d1, d2   |
| <b>See also</b>    | EQ.B, EQ.H, EQ.W, LT.B, LT.BU, LT.H, LT.HU   |

2001-04-30 @ 15:16

### 10.4.115 Multiply-Add ..... MADD(S)

**Table 10-136**

**MADD(S)**

|                    |  |
|--------------------|--|
| <b>Syntax</b>      | <ol style="list-style-type: none"> <li>1. madd Dc,Dd,Da,Db ; 32 + (32*32)--&gt; 32 signed</li> <li>2. madd Dc,Dd,Da,const9 ; 32 + (32*K9)--&gt; 32 signed</li> <li>3. madd Ec,Ed,Da,Db ; 64 + (32*32)--&gt; 64 signed</li> <li>4. madd Ec,Ed,Da,const9 ; 64 + (32*K9)--&gt; 64 signed</li> <li>5. madds Dc,Dd,Da,Db ; 32 + (32*32)--&gt; 32 signed sat</li> <li>6. madds Dc,Dd,Da,const9 ; 32 + (32*K9)--&gt; 32 signed sat</li> <li>7. madds Ec,Ed,Da,Db ; 64 + (32*32)--&gt; 64 signed sat</li> <li>8. madds Ec,Ed,Da,const9 ; 64 + (32*K9)--&gt; 64 signed sat</li> </ol>   |
| <b>Description</b> | <p>Multiply 2 signed 32-bit integers, add the product to a signed 32-bit or 64-bit integer and put the result into a 32-bit or 64-bit register.</p> <p>The value const9 is sign-extended to 32 bits before the multiplication is performed. Overflow and advanced overflow are calculated on the final result.</p> <p><b>(S)</b> On overflow the result is saturated</p>   |
| <b>Operation</b>   | <ol style="list-style-type: none"> <li>1. <math>D[c][31:0] = D[d][31:0] + (D[a][31:0] * D[b][31:0]); \text{signed}</math></li> <li>2. <math>D[c][31:0] = D[d][31:0] + (D[a][31:0] * \text{sign\_ext}(\text{const9})); \text{signed}</math></li> <li>3. <math>E[c][63:0] = E[d][63:0] + (D[a][31:0] * D[b][31:0]); \text{signed}</math></li> <li>4. <math>E[c][63:0] = E[d][63:0] + (D[a][31:0] * \text{sign\_ext}(\text{const9})); \text{signed}</math></li> <li>5. <math>D[c][31:0] = D[d][31:0] + (D[a][31:0] * D[b][31:0]); \text{signed}; \text{ssov}</math></li> <li>6. <math>D[c][31:0] = D[d][31:0] + (D[a][31:0] * \text{sign\_ext}(\text{const9})); \text{signed}; \text{ssov}</math></li> <li>7. <math>E[c][63:0] = E[d][63:0] + (D[a][31:0] * D[b][31:0]); \text{signed}; \text{ssov}</math></li> <li>8. <math>E[c][63:0] = E[d][63:0] + (D[a][31:0] * \text{sign\_ext}(\text{const9})); \text{signed}; \text{ssov}</math></li> </ol> |
| <b>Status</b>      | V, SV, AV, SAV   |
| <b>Examples</b>    |  |

2001-04-30 @ 15:16

#### 10.4.116 Packed Multiply-Add Q Format .....MADD(S).H

**Table 10-137**

**MADD(S).H**

|                    |  |
|--------------------|--|
| <b>Syntax</b>      | <p>1. madd.h Ec,Ed,Da,DbUL,n ; 32  32 +  + (16U*16U    16L*16L)--&gt; 32  32</p> <p>2. madd.h Ec,Ed,Da,DbLU,n ; 32  32 +  + (16U*16L    16L*16U)--&gt; 32  32</p> <p>3. madd.h Ec,Ed,Da,DbLL,n ; 32  32 +  + (16U*16L    16L*16L)--&gt; 32  32</p> <p>4. madd.h Ec,Ed,Da,DbUU,n ; 32  32 +  + (16L*16U    16U*16U)--&gt; 32  32</p> <p>5. madds.h Ec,Ed,Da,DbUL,n ; 32  32 +  + (16U*16U    16L*16L)--&gt; 32  32 sat</p> <p>6. madds.h Ec,Ed,Da,DbLU,n ; 32  32 +  + (16U*16L    16L*16U)--&gt; 32  32 sat</p> <p>7. madds.h Ec,Ed,Da,DbLL,n ; 32  32 +  + (16U*16L    16L*16L)--&gt; 32  32 sat</p> <p>8. madds.h Ec,Ed,Da,DbUU,n ; 32  32 +  + (16L*16U    16U*16U)--&gt; 32  32 sat</p>  |
| <b>Description</b> | <p>This operation is duplicated.</p> <p>Multiply 2 signed 16-bit (halfword) values, add the product ( left justified if n=1) to a signed 32-bit value and put the result into a 32-bit register.</p> <p>There are 4 cases of halfword multiplication: (1) upper*upper and lower*lower, (2) upper*lower and lower*upper, (3) upper*lower and lower*lower, (4) lower*upper and upper*upper.</p> <p>Overflow and advanced overflow are calculated on the final result.</p> <p>If n=1, 0x8000 * 0x8000 = 0x7FFFFFFF</p> <p><b>(S)</b> On overflow each result is independently saturated.</p>  |
| <b>Operation</b>   | <p>1. <math>E[c][63:32] = E[d][63:32] + (((D[a][31:16] * D[b][31:16]) &lt; n)[31:0]) ;</math><br/> <math>E[c][31:0] = E[d][31:0] + (((D[a][15:0] * D[b][15:0]) &lt; n)[31:0]) ;</math></p> <p>2. <math>E[c][63:32] = E[d][63:32] + (((D[a][31:16] * D[b][15:0]) &lt; n)[31:0]) ;</math><br/> <math>E[c][31:0] = E[d][31:0] + (((D[a][15:0] * D[b][31:16]) &lt; n)[31:0]) ;</math></p> <p>3. <math>E[c][63:32] = E[d][63:32] + (((D[a][31:16] * D[b][15:0]) &lt; n)[31:0]) ;</math><br/> <math>E[c][31:0] = E[d][31:0] + (((D[a][15:0] * D[b][15:0]) &lt; n)[31:0]) ;</math></p> <p>4. <math>E[c][63:32] = E[d][63:32] + (((D[a][15:0] * D[b][31:16]) &lt; n)[31:0]) ;</math><br/> <math>E[c][31:0] = E[d][31:0] + (((D[a][31:16] * D[b][31:16]) &lt; n)[31:0]) ;</math></p> <p>5. <math>E[c][63:32] = E[d][63:32] + (((D[a][31:16] * D[b][31:16]) &lt; n)[31:0]) ;</math>ssov<br/> <math>E[c][31:0] = E[d][31:0] + (((D[a][15:0] * D[b][15:0]) &lt; n)[31:0]) ;</math>ssov</p> <p>6. <math>E[c][63:32] = E[d][63:32] + (((D[a][31:16] * D[b][15:0]) &lt; n)[31:0]) ;</math>ssov<br/> <math>E[c][31:0] = E[d][31:0] + (((D[a][15:0] * D[b][31:16]) &lt; n)[31:0]) ;</math>ssov</p> <p>7. <math>E[c][63:32] = E[d][63:32] + (((D[a][31:16] * D[b][15:0]) &lt; n)[31:0]) ;</math>ssov<br/> <math>E[c][31:0] = E[d][31:0] + (((D[a][15:0] * D[b][15:0]) &lt; n)[31:0]) ;</math>ssov</p> <p>8. <math>E[c][63:32] = E[d][63:32] + (((D[a][15:0] * D[b][31:16]) &lt; n)[31:0]) ;</math>ssov<br/> <math>E[c][31:0] = E[d][31:0] + (((D[a][31:16] * D[b][31:16]) &lt; n)[31:0]) ;</math>ssov</p> <p>for all operations = signed; n=0,1;</p> |
| <b>Status</b>      | V, SV, AV, SAV   |
| <b>Examples</b>    |  |

2001-04-30 @ 15:16

### 10.4.117 Multiply-Add Q Format ..... MADD(S).Q

**Table 10-138**  
**MADD(S).Q**

|                    |   |                 |                  |        |     |
|--------------------|---|-----------------|------------------|--------|-----|
| <b>Syntax</b>      | 1. madd.q   | Dc,Dd,Da,Db,n   | ;32 + (32*32)Up  | --> 32 |     |
|                    | 2. madd.q   | Dc,Dd,Da,DbU,n  | ;32 + (16U*32)Up | --> 32 |     |
|                    | 3. madd.q   | Dc,Dd,Da,DbL,n  | ;32 + (16L*32)Up | --> 32 |     |
|                    | 4. madd.q   | Dc,Dd,DaU,DbU,n | ;32 + (16U*16U)  | --> 32 |     |
|                    | 5. madd.q   | Dc,Dd,DaL,DbL,n | ;32 + (16L*16L)  | --> 32 |     |
|                    | 6. madd.q   | Ec,Ed,Da,Db,n   | ;64 + (32*32)    | --> 64 |     |
|                    | 7. madd.q   | Ec,Ed,Da,DbU,n  | ;64 + (16U* 32)  | --> 64 |     |
|                    | 8. madd.q   | Ec,Ed,Da,DbL,n  | ;64 + (16L* 32)  | --> 64 |     |
|                    | 9. madd.q   | Ec,Ed,DaU,DbU,n | ;64 + (16U*16U)  | --> 64 |     |
|                    | 10. madds.q   | Ec,Ed,DaL,DbL,n | ;64 + (16L*16L)  | --> 64 |     |
|                    | 11. madds.q   | Dc,Dd,Da,Db,n   | ;32 + (32*32)Up  | --> 32 | sat |
|                    | 12. madds.q   | Dc,Dd,Da,DbU,n  | ;32 + (16U*32)Up | --> 32 | sat |
|                    | 13. madds.q   | Dc,Dd,Da,DbL,n  | ;32 + (16L*32)Up | --> 32 | sat |
|                    | 14. madds.q   | Dc,Dd,DaU,DbU,n | ;32 + (16U*16U)  | --> 32 | sat |
|                    | 15. madds.q   | Dc,Dd,DaL,DbL,n | ;32 + (16L*16L)  | --> 32 | sat |
|                    | 16. madds.q   | Ec,Ed,Da,Db,n   | ;64 + (32*32)    | --> 64 | sat |
|                    | 17. madds.q   | Ec,Ed,Da,DbU,n  | ;64 + (16U* 32)  | --> 64 | sat |
|                    | 18. madds.q   | Ec,Ed,Da,DbL,n  | ;64 + (16L* 32)  | --> 64 | sat |
|                    | 19. madds.q   | Ec,Ed,DaU,DbU,n | ;64 + (16U*16U)  | --> 64 | sat |
|                    | 20. madds.q   | Ec,Ed,DaL,DbL,n | ;64 + (16L*16L)  | --> 64 | sat |
| <b>Description</b> | <p>Multiply 2 signed 16-bit or 32-bit values, add the product (left justified if n=1) to a signed 32-bit or 64-bit value and put the result into a 32-bit or 64-bit register.</p> <p>There are 8 cases of 16*16 operations, 8 cases of 16x32 operations and 4 cases of 32*32 operations.</p> <p>Overflow and advanced overflow are calculated on the final result.</p> <p>If n=1, 0x8000 * 0x8000 = 0x7FFFFFFF (only for 16 by 16-bit operations).</p> <p><b>(S)</b> On overflow the result is saturated.</p> |                 |                  |        |     |

**Table 10-138**  
**MADD(S).Q (cont'd)**

|                  |   |
|------------------|---|
| <b>Operation</b> | <ol style="list-style-type: none"> <li>1. <math>D[c][31:0] = D[d][31:0] + (((D[a][31:0] * D[b][31:0]) &lt;&lt; n)[63:32])</math> ;</li> <li>2. <math>D[c][31:0] = D[d][31:0] + (((D[a][31:0] * D[b][31:16]) &lt;&lt; n)[47:16])</math> ;</li> <li>3. <math>D[c][31:0] = D[d][31:0] + (((D[a][31:0] * D[b][15:0]) &lt;&lt; n)[47:16])</math> ;</li> <li>4. <math>D[c][31:0] = D[d][31:0] + (((D[a][31:16] * D[b][31:16]) &lt;&lt; n)[31:0])</math> ;</li> <li>5. <math>D[c][31:0] = D[d][31:0] + (((D[a][15:0] * D[b][15:0]) &lt;&lt; n)[31:0])</math> ;</li> <li>6. <math>E[c][63:0] = E[d][63:0] + (((D[a][31:0] * D[b][31:0]) &lt;&lt; n)[63:0])</math> ;</li> <li>7. <math>E[c][63:0] = E[d][63:0] + (((D[a][31:0] * D[b][31:16]) &lt;&lt; n)[47:0])</math> ;</li> <li>8. <math>E[c][63:0] = E[d][63:0] + (((D[a][31:0] * D[b][15:0]) &lt;&lt; n)[47:0])</math> ;</li> <li>9. <math>E[c][63:0] = E[d][63:0] + (((D[a][31:16] * D[b][31:16]) &lt;&lt; n)[31:0] &lt;&lt; 16)</math> ;</li> <li>10. <math>E[c][63:0] = E[d][63:0] + (((D[a][15:0] * D[b][15:0]) &lt;&lt; n)[31:0] &lt;&lt; 16)</math> ;</li> <li>11. <math>D[c][31:0] = D[d][31:0] + (((D[a][31:0] * D[b][31:0]) &lt;&lt; n)[63:32])</math> ;ssov</li> <li>12. <math>D[c][31:0] = D[d][31:0] + (((D[a][31:0] * D[b][31:16]) &lt;&lt; n)[47:16])</math> ;ssov</li> <li>13. <math>D[c][31:0] = D[d][31:0] + (((D[a][31:0] * D[b][15:0]) &lt;&lt; n)[47:16])</math> ;ssov</li> <li>14. <math>D[c][31:0] = D[d][31:0] + (((D[a][31:16] * D[b][31:16]) &lt;&lt; n)[31:0])</math> ;ssov</li> <li>15. <math>D[c][31:0] = D[d][31:0] + (((D[a][15:0] * D[b][15:0]) &lt;&lt; n)[31:0])</math> ;ssov</li> <li>16. <math>E[c][63:0] = D[d][63:0] + (((D[a][31:0] * D[b][31:0]) &lt;&lt; n)[63:0])</math> ;ssov</li> <li>17. <math>E[c][63:0] = D[d][63:0] + (((D[a][31:0] * D[b][31:16]) &lt;&lt; n)[47:0])</math> ;ssov</li> <li>18. <math>E[c][63:0] = D[d][63:0] + (((D[a][31:0] * D[b][15:0]) &lt;&lt; n)[47:0])</math> ;ssov</li> <li>19. <math>E[c][63:0] = D[d][63:0] + (((D[a][31:0] * D[b][31:0]) &lt;&lt; n)[31:0] &lt;&lt; 16)</math> ;ssov</li> <li>20. <math>E[c][63:0] = D[d][63:0] + (((D[a][15:0] * D[b][15:0]) &lt;&lt; n)[31:0] &lt;&lt; 16)</math> ;ssov</li> </ol> <p>for all operations = signed ; n=0,1;</p> |
| <b>Status</b>    | V, SV, AV, SAV  |
| <b>Examples</b>  |   |

2001-04-30 @ 15:16

### 10.4.118 Multiply-Add Unsigned. ....MADD(S).U

**Table 10-139**  
**MADD(S).U**

|                    |   |
|--------------------|---|
| <b>Syntax</b>      | 1. madd.u     Dc,Dd,Da,Db     ;32 + (32*32)--> 32   unsigned<br>2. madd.u     Dc,Dd,Da,const9   ;32 + (32*K9)--> 32   unsigned<br>3. madd.u     Ec,Ed,Da,Db     ;64 + (32*32)--> 64   unsigned<br>4. madd.u     Ec,Ed,Da,const9   ;64 + (32*K9)--> 64   unsigned<br>5. madds.u     Dc,Dd,Da,Db     ;32 + (32*32)--> 32   unsigned sat<br>6. madds.u     Dc,Dd,Da,const9   ;32 + (32*K9)--> 32   unsigned sat<br>7. madds.u     Ec,Ed,Da,Db     ;64 + (32*32)--> 64   unsigned sat<br>8. madds.u     Ec,Ed,Da,const9   ;64 + (32*K9)--> 64   unsigned sat  |
| <b>Description</b> | <p>Multiply 2 unsigned 32-bit integers, add the product to an unsigned 32-bit or 64-bit integer, and put the result into a 32-bit or 64-bit register.</p> <p>The value const9 is zero-extended to 32 bits before the multiplication is performed. Overflow and advanced overflow are calculated on the final result.</p> <p><b>(S)</b> On overflow the result is saturated</p>  |
| <b>Operation</b>   | 1. $D[c][31:0] = D[d][31:0] + (D[a][31:0] * D[b][31:0])$ ;signed<br>2. $D[c][31:0] = D[d][31:0] + (D[a][31:0] * \text{zero\_ext}(\text{const9}))$ ;signed<br>3. $E[c][63:0] = E[d][63:0] + (D[a][31:0] * D[b][31:0])$ ;signed<br>4. $E[c][63:0] = E[d][63:0] + (D[a][31:0] * \text{zero\_ext}(\text{const9}))$ ;signed<br>5. $D[c][31:0] = D[d][31:0] + (D[a][31:0] * D[b][31:0])$ ;signed;suov<br>6. $D[c][31:0] = D[d][31:0] + (D[a][31:0] * \text{zero\_ext}(\text{const9}))$ ;signed;suov<br>7. $E[c][63:0] = E[d][63:0] + (D[a][31:0] * D[b][31:0])$ ;signed;suov<br>8. $E[c][63:0] = E[d][63:0] + (D[a][31:0] * \text{zero\_ext}(\text{const9}))$ ;signed ;suov |
| <b>Status</b>      | V, SV, AV, SAV  |
| <b>Examples</b>    |   |



**10.4.119 Packed Multiply-Add Q Format-Multiprecision ..... MADDM(S).H**
**Table 10-140  
MADDM(S).H**

|                    |  |
|--------------------|--|
| <b>Syntax</b>      | 1. maddm.h Ec,Ed,Da,DbUL,n ;64 + 16U*16U + 16L*16L --> 64<br>2. maddm.h Ec,Ed,Da,DbLU,n ; ;64 + 16U*16L + 16L*16U --> 64<br>3. maddm.h Ec,Ed,Da,DbLL,n ;64 + 16U*16L + 16L*16L --> 64<br>4. maddm.h Ec,Ed,Da,DbUU,n ;64 + 16L*16U + 16U*16U --> 64<br>5. maddms.h Ec,Ed,Da,DbUL,n ;64 + 16U*16U + 16L*16L --> 64 sat<br>6. maddms.h Ec,Ed,Da,DbLU,n ;64 + 16U*16L + 16L*16U --> 64 sat<br>7. maddms.h Ec,Ed,Da,DbLL,n ;64 + 16U*16L + 16L*16L --> 64 sat<br>8. maddms.h Ec,Ed,Da,DbUU,n ;64 + 16L*16U + 16U*16U --> 64 sat   |
| <b>Description</b> | <p>Perform 2 multiplications of 2 signed 16-bit (halfword). Add the 2 products ( left justified if n=1) <b>(left-shifted by 16)</b> to a signed 64-bit value and put the result in a 64-bit register.</p> <p>There are 4 cases of halfword multiplication: (1) upper * upper and lower*lower, (2) upper*lower and lower*upper, (3) upper*lower and lower*lower, (4) lower*upper and upper*upper.</p> <p>Overflow and advanced overflow are calculated on the final result.</p> <p>If n=1, 0x8000 * 0x8000 = 0x7FFFFFFF</p> <p><b>(S)</b> On overflow the result is saturated.</p>  |
| <b>Operation</b>   | 1. $E[c][63:0] = E[d][63:0] + (((D[a][31:16] * D[b][31:16]) \ll n) + ((D[a][15:0] * D[b][15:0]) \ll n) \ll 16) ; \text{signed}, n=0,1$<br>2. $E[c][63:0] = E[d][63:0] + (((D[a][31:16] * D[b][15:0]) \ll n) + ((D[a][15:0] * D[b][31:16]) \ll n) \ll 16) ; \text{signed}, n=0,1$<br>3. $E[c][63:0] = E[d][63:0] + (((D[a][31:16] * D[b][15:0]) \ll n) + ((D[a][15:0] * D[b][15:0]) \ll n) \ll 16) ; \text{signed}, n=0,1$<br>4. $E[c][63:0] = E[d][63:0] + (((D[a][15:0] * D[b][31:16]) \ll n) + ((D[a][31:16] * D[b][31:16]) \ll n) \ll 16) ; \text{signed}, n=0,1$<br>5. $E[c][63:0] = E[d][63:0] + (((D[a][31:16] * D[b][31:16]) \ll n) + ((D[a][15:0] * D[b][15:0]) \ll n) \ll 16) ; \text{signed}, n=0,1 ; \text{ssov}$<br>6. $E[c][63:0] = E[d][63:0] + (((D[a][31:16] * D[b][15:0]) \ll n) + ((D[a][15:0] * D[b][31:16]) \ll n) \ll 16) ; \text{signed}, n=0,1 ; \text{ssov}$<br>7. $E[c][63:0] = E[d][63:0] + (((D[a][31:16] * D[b][15:0]) \ll n) + ((D[a][15:0] * D[b][15:0]) \ll n) \ll 16) ; \text{signed}, n=0,1 ; \text{ssov}$<br>8. $E[c][63:0] = E[d][63:0] + (((D[a][15:0] * D[b][31:16]) \ll n) + ((D[a][31:16] * D[b][31:16]) \ll n) \ll 16) ; \text{signed}, n=0,1 ; \text{ssov}$ |
| <b>Status</b>      | V, SV, AV, SAV   |
| <b>Examples</b>    |  |

2001-04-30 @ 15:16

#### 10.4.120 Packed Multiply-Add Q Format w/ Rounding . . . . . MADDR(S).H

**Table 10-141**  
**MADDR(S).H**

|                    |   |
|--------------------|---|
| <b>Syntax</b>      | <ol style="list-style-type: none"> <li>1. maddr.h Dc,Dd,Da,DbUL,n ;16U   16L +  + (16U*16U    16L*16L) R --&gt; 16  16</li> <li>2. maddr.h Dc,Dd,Da,DbLU,n ;16U   16L +  + (16U*16L    16L*16U) R --&gt; 16  16</li> <li>3. maddr.h Dc,Dd,Da,DbLL,n ;16U   16L +  + (16U*16L    16L*16L) R --&gt; 16  16</li> <li>4. maddr.h Dc,Dd,Da,DbUU,n ;16U   16L +  + (16L*16U    16U*16U) R --&gt; 16  16</li> <li>5. maddr.h Dc,Ed,Da,DbUL,n ; 32   32 +  + (16U*16U    16L*16L) R --&gt; 16  16</li> <li>6. maddrs.h Dc,Dd,Da,DbUL,n ;16U   16L +  + (16U*16U    16L*16L) R --&gt; 16  16 sat</li> <li>7. maddrs.h Dc,Dd,Da,DbLU,n ;16U   16L +  + (16U*16L    16L*16U) R --&gt; 16  16 sat</li> <li>8. maddrs.h Dc,Dd,Da,DbLL,n ;16U   16L +  + (16U*16L    16L*16L) R --&gt; 16  16 sat</li> <li>9. maddrs.h Dc,Dd,Da,DbUU,n ;16U   16L +  + (16L*16U    16U*16U) R --&gt; 16  16 sat</li> <li>10. maddrs.h Dc,Ed,Da,DbUL,n ; 32    32 +  + (16U*16U    16L*16L) R --&gt; 16  16 sat</li> </ol> |
| <b>Description</b> | <p>This operation is duplicated.</p> <p>Multiply 2 signed 16-bit (halfword) values, add the product ( left justified if n=1) to a signed 16-bit value and put the <b>rounded</b> result into <b>half</b> of a 32-bit register.</p> <p>(Note: since there are 2 results the two register halves are used) .</p> <p>There are 4 cases of halfword multiplication: (1) upper * upper and lower*lower, (2) upper*lower and lower*upper, (3) upper*lower and lower*lower, (4) lower*upper and upper*upper.</p> <p>There is a special case: add the product ( left justified if n=1) to a signed 32-bit value and put the <b>rounded</b> result into <b>half</b> of a 32-bit register.</p> <p>Overflow and advanced overflow are calculated on the final results.</p> <p>If n=1, 0x8000 * 0x8000 = 0x7FFFFFFF</p> <p><b>(S)</b> On overflow each result is independently saturated.</p>   |

2001-04-30 @ 15:16

**Table 10-141**  
**MADDR(S).H (cont'd)**

|                  |   |
|------------------|---|
| <b>Operation</b> | <ol style="list-style-type: none"> <li>1. <math>D[c][31:16] = \text{round16}(D[d][31:16] + ((D[a][31:16] * D[b][31:16]) \ll n)[31:0])</math>;<br/> <math>D[c][15:0] = \text{round16}(D[d][15:0] + ((D[a][15:0] * D[b][15:0]) \ll n)[31:0])</math> ;</li> <li>2. <math>D[c][31:16] = \text{round16}(D[d][31:16] + ((D[a][31:16] * D[b][15:0]) \ll n)[31:0])</math>;<br/> <math>D[c][15:0] = \text{round16}(D[d][15:0] + ((D[a][15:0] * D[b][31:16]) \ll n)[31:0])</math> ;</li> <li>3. <math>D[c][31:16] = \text{round16}(D[d][31:16] + ((D[a][31:16] * D[b][15:0]) \ll n)[31:0])</math> ;<br/> <math>D[c][15:0] = \text{round16}(D[d][15:0] + ((D[a][15:0] * D[b][15:0]) \ll n)[31:0])</math> ;</li> <li>4. <math>D[c][31:16] = \text{round16}(D[d][31:16] + ((D[a][15:0] * D[b][31:16]) \ll n)[31:0])</math>;<br/> <math>D[c][15:0] = \text{round16}(D[d][15:0] + ((D[a][31:16] * D[b][31:16]) \ll n)[31:0])</math> ;</li> <li>5. <math>D[c][31:16] = \text{round16}(E[d][63:32] + ((D[a][31:16] * D[b][31:16]) \ll n)[31:0])</math> ;<br/> <math>D[c][15:0] = \text{round16}(E[d][31:0] + ((D[a][15:0] * D[b][15:0]) \ll n)[31:0])</math> ;</li> <li>6. <math>D[c][31:16] = \text{round16}(D[d][31:16] + ((D[a][31:16] * D[b][31:16]) \ll n)[31:0])</math> ;ssov<br/> <math>D[c][15:0] = \text{round16}(D[d][15:0] + ((D[a][31:16] * D[b][31:16]) \ll n)[31:0])</math> ;ssov</li> <li>7. <math>D[c][31:16] = \text{round16}(D[d][31:16] + ((D[a][31:16] * D[b][15:0]) \ll n)[31:0])</math> ;ssov<br/> <math>D[c][15:0] = \text{round16}(D[d][15:0] + ((D[a][15:0] * D[b][31:16]) \ll n)[31:0])</math> ;ssov</li> <li>8. <math>D[c][31:16] = \text{round16}(D[d][31:16] + ((D[a][31:16] * D[b][15:0]) \ll n)[31:0])</math> ;ssov<br/> <math>D[c][15:0] = \text{round16}(D[d][15:0] + ((D[a][15:0] * D[b][15:0]) \ll n)[31:0])</math> ;ssov</li> <li>9. <math>D[c][31:16] = \text{round16}(D[d][31:16] + ((D[a][15:0] * D[b][31:16]) \ll n)[31:0])</math> ;ssov<br/> <math>D[c][15:0] = \text{round16}(D[d][15:0] + ((D[a][31:16] * D[b][31:16]) \ll n)[31:0])</math> ;ssov</li> <li>10. <math>D[c][31:16] = \text{round16}(E[d][63:32] + ((D[a][31:16] * D[b][31:16]) \ll n)[31:0])</math> ;ssov<br/> <math>D[c][15:0] = \text{round16}(E[d][31:0] + ((D[a][15:0] * D[b][15:0]) \ll n)[31:0])</math> ;ssov</li> </ol> <p>for all operations = signed ; n=0,1;<br/> round16 == add 0x8000 to 32bit value and clear bits[15:0]</p> |
| <b>Status</b>    | V, SV, AV, SAV  |
| <b>Examples</b>  |   |

2001-04-30 @ 15:16

### 10.4.121 Multiply-Add Q Format with Rounding ..... MADDR(S).Q

Table 10-142

MADDR(S).Q

|                    |  |
|--------------------|--|
| <b>Syntax</b>      | 1. maddr.q Dc,Dd,DaU,DbU,n ;32 + (16U*16U) R--> 32<br>2. maddr.q Dc,Dd,DaL,DbL,n ;32 + (16L*16L) R--> 32<br>3. maddrs.q Dc,Dd,DaU,DbU,n ;32 + (16U*16U) R--> 32 sat<br>4. maddrs.q Dc,Dd,DaL,DbL,n ;32 + (16L*16L) R--> 32 sat   |
| <b>Description</b> | <p>Multiply 2 signed 16-bit (halfword) values, add the product (left justified if n=1) to a 32-bit signed value, put the <b>rounded</b> result in a 32-bit register.</p> <p>The lower halfword is cleared.</p> <p>Overflow and advanced overflow are calculated on the final results.</p> <p>If n=1, 0x8000 * 0x8000 = 0x7FFFFFFF (only for 16 by 16-bit operations).</p> <p><b>(S)</b> On overflow the result is saturated.</p>   |
| <b>Operation</b>   | 1. $D[c][31:0] = \text{round16}(D[d][31:0] + ((D[a][31:16] * D[b][31:16]) \ll n)[31:0])$<br>2. $D[c][31:0] = \text{round16}(D[d][31:0] + ((D[a][15:0] * D[b][15:0]) \ll n)[31:0])$<br>3. $D[c][31:0] = \text{round16}(D[d][31:0] + ((D[a][31:16] * D[b][31:16]) \ll n)[31:0]); \text{ssov}$<br>4. $D[c][31:0] = \text{round16}(D[d][31:0] + ((D[a][15:0] * D[b][15:0]) \ll n)[31:0]); \text{ssov}$<br>for all operations = signed; n=0,1;<br>round16 == add 0x8000 to 32bit value and clear bits[15:0] |
| <b>Status</b>      | V, SV, AV, SAV   |
| <b>Examples</b>    |  |

**10.4.122 Packed Multiply-Add/Sub Q Format ..... MADDSU(S).H**
**Table 10-143  
MADDSU(S).H**

|                    |  |
|--------------------|--|
| <b>Syntax</b>      | 1. maddsu.h Ec,Ed,Da,DbUL,n ;32  32 +  - (16U*16U    16L*16L)--> 32  32<br>2. maddsu.h Ec,Ed,Da,DbLU,n ;32  32 +  - (16U*16L    16L*16U)--> 32  32<br>3. maddsu.h Ec,Ed,Da,DbLL,n ;32  32 +  - (16U*16L    16L*16L)--> 32  32<br>4. maddsu.h Ec,Ed,Da,DbUU,n ;32  32 +  - (16L*16U    16U*16U)--> 32  32<br>5. maddsus.h Ec,Ed,Da,DbUL,n ;32  32 +  - (16U*16U    16L*16L)--> 32  32sat<br>6. maddsus.h Ec,Ed,Da,DbLU,n ;32  32 +  - (16U*16L    16L*16U)--> 32  32sat<br>7. maddsus.h Ec,Ed,Da,DbLL,n ;32  32 +  - (16U*16L    16L*16L)--> 32  32sat<br>8. maddsus.h Ec,Ed,Da,DbUU,n ;32  32 +  - (16L*16U    16U*16U)--> 32  32sat   |
| <b>Description</b> | <p>This operation is duplicated.</p> <p>Multiply 2 signed 16-bit (halfword) values, add (or subtract) the product ( left justified if n=1) to a signed 32-bit value and put the result into a 32-bit register.</p> <p>There are 4 cases of halfword multiplication: (1) upper * upper and lower*lower, (2) upper*lower and lower*upper, (3) upper*lower and lower*lower, (4) lower*upper and upper*upper.</p> <p>Overflow and advanced overflow are calculated on the final results.</p> <p>If n=1, 0x8000 * 0x8000 = 0x7FFFFFFF</p> <p><b>(S)</b> On overflow each result is independently saturated.</p>   |
| <b>Operation</b>   | 1. $E[c][63:32] = E[d][63:32] + (((D[a][31:16] * D[b][31:16]) << n)[31:0])$ ;<br>$E[c][31:0] = E[d][31:0] - (((D[a][15:0] * D[b][15:0]) << n)[31:0])$ ;<br>2. $E[c][63:32] = E[d][63:32] + (((D[a][31:16] * D[b][15:0]) << n)[31:0])$ ;<br>$E[c][31:0] = E[d][31:0] - (((D[a][15:0] * D[b][31:16]) << n)[31:0])$ ;<br>3. $E[c][63:32] = E[d][63:32] + (((D[a][31:16] * D[b][15:0]) << n)[31:0])$ ;<br>$E[c][31:0] = E[d][31:0] - (((D[a][15:0] * D[b][15:0]) << n)[31:0])$ ;<br>4. $E[c][63:32] = E[d][63:32] + (((D[a][15:0] * D[b][31:16]) << n)[31:0])$ ;<br>$E[c][31:0] = E[d][31:0] - (((D[a][31:16] * D[b][31:16]) << n)[31:0])$ ;<br>5. $E[c][63:32] = E[d][63:32] + (((D[a][31:16] * D[b][31:16]) << n)[31:0])$ ;ssov<br>$E[c][31:0] = E[d][31:0] - (((D[a][31:16] * D[b][31:16]) << n)[31:0])$ ;ssov<br>6. $E[c][63:32] = E[d][63:32] + (((D[a][31:16] * D[b][15:0]) << n)[31:0])$ ;ssov<br>$E[c][31:0] = E[d][31:0] - (((D[a][15:0] * D[b][31:16]) << n)[31:0])$ ;ssov<br>7. $E[c][63:32] = E[d][63:32] + (((D[a][31:16] * D[b][15:0]) << n)[31:0])$ ;ssov<br>$E[c][31:0] = E[d][31:0] - (((D[a][15:0] * D[b][15:0]) << n)[31:0])$ ;ssov<br>8. $E[c][63:32] = E[d][63:32] + (((D[a][15:0] * D[b][31:16]) << n)[31:0])$ ;ssov<br>$E[c][31:0] = E[d][31:0] - (((D[a][31:16] * D[b][31:16]) << n)[31:0])$ ;ssov<br>for all operations = signed ; n=0,1; |
| <b>Status</b>      | V, SV, AV, SAV   |
| <b>Examples</b>    |  |

2001-04-30 @ 15:16

#### 10.4.123 Packed Multiply-Add/Sub Q Format-Multiprecision .....MADDSUM(S).H

Table 10-144  
MADDSUM(S).H

|                    |   |
|--------------------|---|
| <b>Syntax</b>      | <p>1. maddsum.h Ec,Ed,Da,DbUL,n ; 64 + 16U*16U - 16L*16L --&gt; 64</p> <p>2. maddsum.h Ec,Ed,Da,DbLU,n ; 64 + 16U*16L - 16L*16U --&gt; 64</p> <p>3. maddsum.h Ec,Ed,Da,DbLL,n ; 64 + 16U*16L - 16L*16L --&gt; 64</p> <p>4. maddsum.h Ec,Ed,Da,DbUU,n ; 64 + 16L*16U - 16U*16U --&gt; 64</p> <p>5. maddsums.h Ec,Ed,Da,DbUL,n ; 64 + 16U*16U - 16L*16L --&gt; 64 sat</p> <p>6. maddsums.h Ec,Ed,Da,DbLU,n ; 64 + 16U*16L - 16L*16U --&gt; 64 sat</p> <p>7. maddsums.h Ec,Ed,Da,DbLL,n ; 64 + 16U*16L - 16L*16L --&gt; 64 sat</p> <p>8. maddsums.h Ec,Ed,Da,DbUU,n ; 64 + 16L*16U - 16U*16U --&gt; 64 sat</p>   |
| <b>Description</b> | <p>Perform 2 multiplications of 2 signed 16-bit (halfword). Add 1 product and subtract the other product (left justified if n=1) (<b>left-shifted by 16</b>) to/from a signed 64-bit value and put the result in a 64-bit register.</p> <p>There are 4 cases of halfword multiplication: (1) upper * upper and lower*lower, (2) upper*lower and lower*upper, (3) upper*lower and lower*lower, (4) lower*upper and upper*upper.</p> <p>Overflow and advanced overflow are calculated on the final result.</p> <p>If n=1, 0x8000 * 0x8000 = 0x7FFFFFFF</p> <p><b>(S)</b> On overflow the result is saturated.</p>   |
| <b>Operation</b>   | <p>1. <math>E[c][63:0] = E[d][63:0] + (((D[a][31:16] * D[b][31:16]) \ll n) - ((D[a][15:0] * D[b][15:0]) \ll n) \ll 16)</math>; signed, n=0,1</p> <p>2. <math>E[c][63:0] = E[d][63:0] + (((D[a][31:16] * D[b][15:0]) \ll n) - ((D[a][15:0] * D[b][31:16]) \ll n) \ll 16)</math>; signed, n=0,1</p> <p>3. <math>E[c][63:0] = E[d][63:0] + (((D[a][31:16] * D[b][15:0]) \ll n) - ((D[a][15:0] * D[b][15:0]) \ll n) \ll 16)</math>; signed, n=0,1</p> <p>4. <math>E[c][63:0] = E[d][63:0] + (((D[a][15:0] * D[b][31:16]) \ll n) - ((D[a][31:16] * D[b][31:16]) \ll n) \ll 16)</math>; signed, n=0,1</p> <p>5. <math>E[c][63:0] = E[d][63:0] + (((D[a][31:16] * D[b][31:16]) \ll n) - ((D[a][15:0] * D[b][15:0]) \ll n) \ll 16)</math>; signed, n=0,1; ssov</p> <p>6. <math>E[c][63:0] = E[d][63:0] + (((D[a][31:16] * D[b][15:0]) \ll n) - ((D[a][15:0] * D[b][31:16]) \ll n) \ll 16)</math>; signed, n=0,1; ssov</p> <p>7. <math>E[c][63:0] = E[d][63:0] + (((D[a][31:16] * D[b][15:0]) \ll n) - ((D[a][15:0] * D[b][15:0]) \ll n) \ll 16)</math>; signed, n=0,1; ssov</p> <p>8. <math>E[c][63:0] = E[d][63:0] + (((D[a][15:0] * D[b][31:16]) \ll n) - ((D[a][31:16] * D[b][31:16]) \ll n) \ll 16)</math>; signed, n=0,1; ssov</p> |
| <b>Status</b>      | V, SV, AV, SAV  |
| <b>Examples</b>    |   |

2001-04-30 @ 15:16

#### 10.4.124 Packed Multiply-Add/Sub Q Format w/Rounding .....MADDSUR(S).H

**Table 10-145**  
**MADDSUR(S).H**

|                    |  |
|--------------------|--|
| <b>Syntax</b>      | <p>1. maddsur.h Dc,Dd,Da,DbUL,n ;16U  16L + - (16U*16U    16L*16L) R --&gt; 16  16</p> <p>2. maddsur.h Dc,Dd,Da,DbLU,n ;16U  16L + - (16U*16L    16L*16U) R --&gt; 16  16</p> <p>3. maddsur.h Dc,Dd,Da,DbLL,n ;16U  16L + - (16U*16L    16L*16L) R --&gt; 16  16</p> <p>4. maddsur.h Dc,Dd,Da,DbUU,n ;16U  16L + - (16L*16U    16U*16U) R --&gt; 16  16</p> <p>5. maddsur.h Dc,Dd,Da,DbUL,n ;16U  16L + - (16U*16U    16L*16L) R --&gt;16  16 sat</p> <p>6. maddsur.h Dc,Dd,Da,DbLU,n ;16U  16L + - (16U*16L    16L*16U) R --&gt; 16  16 sat</p> <p>7. maddsur.h Dc,Dd,Da,DbLL,n ;16U  16L + - (16U*16L    16L*16L) R --&gt; 16  16 sat</p> <p>8. maddsur.h Dc,Dd,Da,DbUU,n ;16U  16L + - (16L*16U    16U*16U) R --&gt;16  16 sat</p>  |
| <b>Description</b> | <p>This operation is duplicated.</p> <p>Multiply 2 signed 16-bit (halfword) values, add (subtract) the product ( left justified if n=1) from a signed 16-bit value and put the <b>rounded</b> result into <b>half</b> of a 32-bit register.(Note: since there are 2 results the two register halves are used) .</p> <p>There are 4 cases of halfword multiplication: (1) upper * upper and lower*lower, (2) upper*lower and lower*upper, (3) upper*lower and lower*lower, (4) lower*upper and upper*upper.</p> <p>Overflow and advanced overflow are calculated on the final results.</p> <p>If n=1, 0x8000 * 0x8000 = 0x7FFFFFFF</p> <p><b>(S)</b> On overflow each result is independently saturated.</p>  |
| <b>Operation</b>   | <p>1. D[c][31:16] = round16(D[d][31:16] + (((D[a][31:16]* D[b][31:16])&lt;&lt;n)[31:0]);<br/>D[c][15:0] = round16(D[d][15:0] - (((D[a][15:0] * D[b][15:0]) &lt;&lt;n)[31:0])</p> <p>2. D[c][31:16] = round16(D[d][31:16] + (((D[a][31:16]* D[b][15:0]) &lt;&lt;n)[31:0])<br/>D[c][15:0] = round16(D[d][15:0] - (((D[a][15:0] * D[b][31:16])&lt;&lt;n)[31:0])</p> <p>3. D[c][31:16] = round16(D[d][31:16] + (((D[a][31:16]* D[b][15:0]) &lt;&lt;n)[31:0])<br/>D[c][15:0] = round16(D[d][15:0] - (((D[a][15:0] * D[b][15:0]) &lt;&lt;n)[31:0])</p> <p>4. D[c][31:16] = round16(D[d][31:16] + (((D[a][15:0] * D[b][31:16])&lt;&lt;n)[31:0])<br/>D[c][15:0] = round16(D[d][15:0] - (((D[a][31:16]* D[b][31:16])&lt;&lt;n)[31:0])</p> <p>5. D[c][31:16] = round16(D[d][31:16] + (((D[a][31:16]* D[b][31:16])&lt;&lt;n)[31:0]);ssov<br/>D[c][15:0] = round16(D[d][15:0] - (((D[a][15:0] * D[b][15:0]) &lt;&lt;n)[31:0]);ssov</p> <p>6. D[c][31:16] = round16(D[d][31:16] + (((D[a][31:16]* D[b][15:0]) &lt;&lt;n)[31:0]);ssov<br/>D[c][15:0] = round16(D[d][15:0] - (((D[a][15:0] * D[b][31:16])&lt;&lt;n)[31:0]);ssov</p> <p>7. D[c][31:16] = round16(D[d][31:16] + (((D[a][31:16]* D[b][15:0]) &lt;&lt;n)[31:0]);ssov<br/>D[c][15:0] = round16(D[d][15:0] - (((D[a][15:0] * D[b][15:0]) &lt;&lt;n)[31:0]);ssov</p> <p>8. D[c][31:16] = round16(D[d][31:16] + (((D[a][15:0] * D[b][31:16])&lt;&lt;n)[31:0]);ssov<br/>D[c][15:0] = round16(D[d][15:0] - (((D[a][31:16]* D[b][31:16])&lt;&lt;n)[31:0]);ssov</p> <p>for all operations = signed ; n=0,1;<br/>round16 == add 0x8000 to 32bit value and clear bits[15:0]</p> |
| <b>Status</b>      | V, SV, AV, SAV   |
| <b>Examples</b>    |  |

2001-04-30 @ 15:16

#### 10.4.125 Maximum Value ..... MAX

**Table 10-146**  
**MAX**

|                    |   |
|--------------------|---|
| <b>Syntax</b>      | max Dc, Da, Db (RR)<br>max Dc, Da, const9 (RC)  |
| <b>Description</b> | If the contents of data register <i>Da</i> are greater than the contents of data register <i>Db</i> /<br><i>const9</i> , put the contents of <i>Da</i> in data register <i>Dc</i> ; otherwise, put the contents of <i>Db</i> /<br><i>const9</i> in <i>Dc</i> . The operands are treated as signed, 32-bit integers. |
| <b>Operation</b>   | if ( $D[a] > D[b]$ ) then $D[c] = D[a]$<br>else $D[c] = D[b]$ ; signed<br><br>if ( $D[a] > \text{sign\_ext}(\text{const9})$ ) then $D[c] = D[a]$<br>else $D[c] = \text{sign\_ext}(\text{const9})$ ; signed  |
| <b>Status</b>      | -   |
| <b>Examples</b>    | max d3, d1, d2<br><br>max d3, d1, 126   |
| <b>See also</b>    | MAX.U, MIN, MIN.U   |

#### 10.4.126 Maximum Value Packed Byte ..... MAX.B Maximum Value Packed Byte Unsigned ..... MAX.BU

**Table 10-147**  
**MAX.B & MAX.BU**

|                    |  |
|--------------------|--|
| <b>Syntax</b>      | max.b Dc, Da, Db (RR)<br>max.bu Dc, Da, Db (RR)  |
| <b>Description</b> | Compute the maximum value of the corresponding bytes in <i>Da</i> and <i>Db</i> and put<br>each result in the corresponding byte of <i>Dc</i> . The operands are treated as signed/<br>unsigned, 8-bit integers.   |
| <b>Operation</b>   | if ( $D[a][(n+7):n] > D[b][(n+7):n]$ )<br>then $D[c][(n+7):n] = D[a][(n+7):n]$<br>else $D[c][(n+7):n] = D[b][(n+7):n]$ ; $n = 0, 8, 16, 24$ ; signed<br><br>if ( $D[a][(n+7):n] > D[b][(n+7):n]$ )<br>then $D[c][(n+7):n] = D[a][(n+7):n]$<br>else $D[c][(n+7):n] = D[b][(n+7):n]$ ; $n = 0, 8, 16, 24$ ; unsigned |
| <b>Status</b>      | -  |
| <b>Examples</b>    | max.b d3, d1, d2<br><br>max.bu d3, d1, d2  |
| <b>See also</b>    | MAX.H, MAX.HU, MIN.B, MIN.BU, MIN.H, MIN.HU  |



**10.4.127 Maximum Value Packed Halfword ..... MAX.H**  
**Maximum Value Packed Halfword Unsigned ..... MAX.HU**

**Table 10-148**
**MAX.H & MAX.HU**

|                    |  |
|--------------------|--|
| <b>Syntax</b>      | max.h Dc, Da, Db (RR)<br>max.hu Dc, Da, Db (RR)  |
| <b>Description</b> | Compute the maximum value of the corresponding halfwords in <i>Da</i> and <i>Db</i> and put each result in the corresponding halfword of <i>Dc</i> . The operands are treated as signed/unsigned, 16-bit integers.   |
| <b>Operation</b>   | if (D[a][(n+15):n] > D[b][(n+15):n])<br>then D[c][(n+15):n] = D[a][(n+15):n]<br>else D[c][(n+15):n] = D[b][(n+15):n]; n = 0, 16; signed<br><br>if (D[a][(n+15):n] > D[b][(n+15):n])<br>then D[c][(n+15):n] = D[a][(n+15):n]<br>else D[c][(n+15):n] = D[b][(n+15):n]; n = 0, 16; unsigned |
| <b>Status</b>      | -  |
| <b>Examples</b>    | max.h d3, d1, d2<br>max.hu d3, d1, d2  |
| <b>See also</b>    | MAX.B, MAX.BU, MIN.B, MIN.BU, MIN.H, MIN.HU  |

**10.4.128 Maximum Value Unsigned ..... MAX.U**

**Table 10-149**
**MAX.U**

|                    |   |
|--------------------|---|
| <b>Syntax</b>      | max.u Dc, Da, Db (RR)<br>max.u Dc, Da, const9 (RC)  |
| <b>Description</b> | If the contents of data register <i>Da</i> are greater than the contents of data register <i>Db/const9</i> , put the contents of <i>Da</i> in data register <i>Dc</i> ; otherwise, put the contents of <i>Db/const9</i> in <i>Dc</i> . The operands are treated as unsigned, 32-bit integers. |
| <b>Operation</b>   | if (D[a] > D[b]) then D[c] = D[a]<br>else D[c] = D[b]; unsigned<br><br>if (D[a] > zero_ext(const9)) then D[c] = D[a]<br>else D[c] = zero_ext(const9); unsigned  |
| <b>Status</b>      | -   |
| <b>Examples</b>    | max.u d3, d1, d2<br>max.u d3, d1, 126   |
| <b>See also</b>    | MAX, MIN, MIN.U   |

2001-04-30 @ 15:16

### 10.4.129 Move From Core Register ..... MFCR

**Table 10-150**  
**MFCR**

|                    |  |
|--------------------|--|
| <b>Syntax</b>      | <code>mfcrr Dc, const16 (RLC)</code>   |
| <b>Description</b> | <p>Move the contents of the core SFR register, selected by the value <i>const16</i>, to data register <i>Dc</i>. The core SFR address is a <i>const16</i> 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.</p> <p>This instruction may not be used to access GPRs, attempting to access a GPR with this instruction will return an undefined value.</p> |
| <b>Operation</b>   | $D[c] = CR[const16]$   |
| <b>Status</b>      | -  |
| <b>Examples</b>    | <code>mfcrr d3, 0xfe04</code>  |
| <b>See also</b>    | MTCR   |

### 10.4.130 Minimum Value ..... MIN

**Table 10-151**  
**MIN**

|                    |  |
|--------------------|--|
| <b>Syntax</b>      | <code>min Dc, Da, Db (RR)</code><br><code>min Dc, Da, const9 (RC)</code>   |
| <b>Description</b> | <p>If the contents of data register <i>Da</i> are less than the contents of data register <i>Db</i>/<i>const9</i>, put the contents of <i>Da</i> in data register <i>Dc</i>; otherwise, put the contents of <i>Db</i>/<i>const9</i> in <i>Dc</i>. The operands are treated as signed, 32-bit integers.</p> |
| <b>Operation</b>   | <p>if (<math>D[a] &lt; D[b]</math>) then <math>D[c] = D[a]</math><br/> else <math>D[c] = D[b]</math>; signed</p> <p>if (<math>D[a] &lt; \text{sign\_ext}(const9)</math>) then <math>D[c] = D[a]</math><br/> else <math>D[c] = \text{sign\_ext}(const9)</math>; signed</p>                                  |
| <b>Status</b>      | -  |
| <b>Examples</b>    | <code>min d3, d1, d2</code><br><code>min d3, d1, 126</code>  |
| <b>See also</b>    | MAX, MAX.U, MIN.U  |

2001-04-30 @ 15:16

**10.4.131 Minimum Value Packed Byte ..... MIN.B**  
**Minimum Value Packed Byte Unsigned ..... MIN.BU**

**Table 10-152**  
**MIN.B & MIN.BU**

|                    |  |
|--------------------|--|
| <b>Syntax</b>      | min.b Dc, Da, Db (RR)<br>min.bu Dc, Da, Db (RR)  |
| <b>Description</b> | Compute the minimum value of the corresponding bytes in <i>Da</i> and <i>Db</i> and put each result in the corresponding byte of <i>Dc</i> . The operands are treated as signed/unsigned, 8-bit integers.  |
| <b>Operation</b>   | if ( $D[a][(n+7):n] < D[b][(n+7):n]$ )<br>then $D[c][(n+7):n] = D[a][(n+7):n]$<br>else $D[c][(n+7):n] = D[b][(n+7):n]$ ; $n = 0, 8, 16, 24$ ; signed<br><br>if ( $D[a][(n+7):n] < D[b][(n+7):n]$ )<br>then $D[c][(n+7):n] = D[a][(n+7):n]$<br>else $D[c][(n+7):n] = D[b][(n+7):n]$ ; $n = 0, 8, 16, 24$ ; unsigned |
| <b>Status</b>      | -  |
| <b>Examples</b>    | min.b d3, d1, d2<br>min.bu d3, d1, d2  |
| <b>See also</b>    | MAX.B, MAX.BU, MAX.H, MAX.HU, MIN.H, MIN.HU  |

**10.4.132 Minimum Value Packed Halfword ..... MIN.H**  
**Minimum Value Packed Halfword Unsigned ..... MIN.HU**

**Table 10-153**  
**MIN. H & MIN.HU**

|                    |  |
|--------------------|--|
| <b>Syntax</b>      | min.h Dc, Da, Db (RR)<br>min.hu Dc, Da, Db (RR)  |
| <b>Description</b> | Compute the minimum value of the corresponding halfwords in <i>Da</i> and <i>Db</i> and put each result in the corresponding halfword of <i>Dc</i> . The operands are treated as signed/unsigned, 16-bit integers. |

2001-04-30 @ 15:16

**Table 10-153**  
**MIN. H & MIN.HU**

|                  |   |
|------------------|---|
| <b>Operation</b> | <p>if (<math>D[a][(n+15):n] &lt; D[b][(n+15):n]</math>)<br/> then <math>D[c][(n+15):n] = D[a][(n+15):n]</math><br/> else <math>D[c][(n+15):n] = D[b][(n+15):n]</math>; <math>n = 0, 16</math>; signed</p> <p>if (<math>D[a][(n+15):n] &lt; D[b][(n+15):n]</math>)<br/> then <math>D[c][(n+15):n] = D[a][(n+15):n]</math><br/> else <math>D[c][(n+15):n] = D[b][(n+15):n]</math>; <math>n = 0, 16</math>; unsigned</p> |
| <b>Status</b>    | -   |
| <b>Examples</b>  | <pre>min.h    d3, d1, d2 min.hu   d3, d1, d2</pre>  |
| <b>See also</b>  | MAX.B, MAX.BU, MAX.H, MAX.HU, MIN.B, MIN.BU   |

### 10.4.133 Minimum Value Unsigned ..... MIN.U

**Table 10-154**  
**MIN. U**

|                    |   |
|--------------------|---|
| <b>Syntax</b>      | <pre>min.u    Dc, Da, Db (RR) min.u    Dc, Da, const9 (RC)</pre>  |
| <b>Description</b> | If the contents of data register <i>Da</i> are less than the contents of data register <i>Db/const9</i> , put the contents of <i>Da</i> in data register <i>Dc</i> ; otherwise, put the contents of <i>Db/const9</i> in <i>Dc</i> . The operands are treated as unsigned, 32-bit integers.  |
| <b>Operation</b>   | <p>if (<math>D[a] &lt; D[b]</math>) then <math>D[c] = D[a]</math><br/> else <math>D[c] = D[b]</math>; unsigned</p> <p>if (<math>D[a] &lt; \text{zero\_ext}(\text{const9})</math>) then <math>D[c] = D[a]</math><br/> else <math>D[c] = \text{zero\_ext}(\text{const9})</math>; unsigned</p> |
| <b>Status</b>      | -   |
| <b>Examples</b>    | <pre>min.u d3, d1, d2 min.u d3, d1, 126</pre>   |
| <b>See also</b>    | MAX, MAX.U, MIN   |

2001-04-30 @ 15:16

**10.4.134 Move. .... MOV**
**Table 10-155  
MOV**

|                    |  |
|--------------------|--|
| <b>Syntax</b>      | mov Dc, Db (RR)  |
|                    | mov Dc, const16 (RLC)  |
|                    | mov Da, Db (SRR)   |
|                    | mov Da, const4 (SRC)   |
|                    | mov D15, const8 (SC)   |
| <b>Description</b> | Move the contents of data register Db/const16 to data register Dc. 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 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. |
| <b>Operation</b>   | D[c] = D[b]<br>D[c] = sign_ext(const16)  |
|                    | D[a] = D[b]<br>D[a] = sign_ext(const4)<br>D[15] = zero_ext(const8)   |
| <b>Status</b>      | -  |
| <b>Examples</b>    | mov d3, d1   |
|                    | mov d3, -30000   |
|                    | mov d1, d2   |
|                    | mov d1, 6  |
|                    | mov d15, 126   |
| <b>See also</b>    | MOV.U, MOVH  |

**10.4.135 Move Value to Address Register. .... MOV.A**
**Table 10-156  
MOV.A**

|                    |  |
|--------------------|--|
| <b>Syntax</b>      | mov.a Ac, Db (RR)  |
|                    | mov.a Aa, Db (SRR)   |
|                    | mov.a Aa, const4 (SRC)   |
| <b>Description</b> | Move the contents of data register Db to address register Ac.  |
|                    | Move the contents of data register Db/const4 to address register Aa. The value const4 is zero-extended to 32 bits before it is moved |
| <b>Operation</b>   | A[c] = D[b]  |
|                    | A[a] = D[b]  |
|                    | A[a] = zero_ext(const4)  |

2001-04-30 @ 15:16

**Table 10-156**  
**MOV.A**

|                 |                            |
|-----------------|----------------------------|
| <b>Status</b>   | -                          |
| <b>Examples</b> | mov.a a3, d1               |
|                 | mov.a a4, d2               |
|                 | mov.a a4, 7                |
| <b>See also</b> | LEA, MOV.AA, MOV.D, MOVH.A |

### 10.4.136 Move Address from Address Register . . . . . MOV.AA

**Table 10-157**  
**MOV.AA**

|                    |   |
|--------------------|---|
| <b>Syntax</b>      | mov.aa Ac, Ab (RR)  |
|                    | mov.aa Aa, Ab (SRR)   |
| <b>Description</b> | Move the contents of address register <i>Ab</i> to address register <i>Ac</i> . |
|                    | Move the contents of address register <i>Ab</i> to address register <i>Aa</i> . |
| <b>Operation</b>   | $A[c] = A[b]$   |
|                    | $A[a] = A[b]$   |
| <b>Status</b>      | -   |
| <b>Examples</b>    | mov.aa a3, a4   |
|                    | mov.aa a4, a2   |
| <b>See also</b>    | LEA, MOV.A, MOV.D, MOVH.A   |

### 10.4.137 Move Address to Data Register . . . . . MOV.D

**Table 10-158**  
**MOV.D**

|                    |  |
|--------------------|--|
| <b>Syntax</b>      | mov.d Dc, Ab (RR)  |
|                    | mov.d Da, Ab (SRR)   |
| <b>Description</b> | Move the contents of address register <i>Ab</i> to data register <i>Dc</i> . |
|                    | Move the contents of address register <i>Ab</i> to data register <i>Da</i> . |
| <b>Operation</b>   | $D[c] = A[b]$  |
|                    | $D[a] = A[b]$  |
| <b>Status</b>      | -  |
| <b>Examples</b>    | mov.d d3, a4   |
|                    | mov.d d1, a2   |
| <b>See also</b>    | LEA, MOV.A, MOV.AA, MOVH.A   |

2001-04-30 @ 15:16

#### 10.4.138 Move Unsigned ..... MOV.U

Table 10-159

MOV. U

|                    |  |
|--------------------|--|
| <b>Syntax</b>      | mov.u Dc, const16 (RLC)  |
| <b>Description</b> | Move the zero-extended value <i>const16</i> to data register <i>Dc</i> . |
| <b>Operation</b>   | D[c] = zero_ext(const16)   |
| <b>Status</b>      | -  |
| <b>Examples</b>    | mov.u d3, 526  |
| <b>See also</b>    | MOV, MOVH  |

#### 10.4.139 Move High .....MOVH

Table 10-160

MOVH

|                    |  |
|--------------------|--|
| <b>Syntax</b>      | movh Dc, const16 (RLC)   |
| <b>Description</b> | Move the value <i>const16</i> to the most-significant halfword of data register <i>Dc</i> and set the least-significant 16 bits to zero. |
| <b>Operation</b>   | D[c] = {const16, 16'h 0000}  |
| <b>Status</b>      | -  |
| <b>Examples</b>    | movh d3, 526   |
| <b>See also</b>    | MOV, MOV.U   |

#### 10.4.140 Move High to Address ..... MOVH.A

Table 10-161

MOVH.A

|                    |   |
|--------------------|---|
| <b>Syntax</b>      | movh.a Ac, const16 (RLC)  |
| <b>Description</b> | Move the value <i>const16</i> to the most-significant halfword of address register <i>Ac</i> and set the least-significant 16 bits to zero. |
| <b>Operation</b>   | A[c] = {const16, 16'h 0000}   |
| <b>Status</b>      | -   |
| <b>Examples</b>    | movh.a a3, 526  |
| <b>See also</b>    | LEA, MOV.A, MOV.AA, MOV.D   |

2001-04-30 @ 15:16

### 10.4.141 Multiply-Sub. .... MSUB(S)

**Table 10-162**  
**MSUB(S)**

|                    |  |
|--------------------|--|
| <b>Syntax</b>      | 1. msub Dc,Dd,Da,Db ; 32 - (32*32)--> 32 signed<br>2. msub Dc,Dd,Da,const9 ; 32 - (32*K9)--> 32 signed<br>3. msub Ec,Ed,Da,Db ; 64 - (32*32)--> 64 signed<br>4. msub Ec,Ed,Da,const9 ; 64 - (32*K9)--> 64 signed<br>5. msubs Dc,Dd,Da,Db ; 32 - (32*32)--> 32 signed sat<br>6. msubs Dc,Dd,Da,const9 ; 32 - (32*K9)--> 32 signed sat<br>7. msubs Ec,Ed,Da,Db ; 64 - (32*32)--> 64 signed sat<br>8. msubs Ec,Ed,Da,const9 ; 64 - (32*K9)--> 64 signed sat   |
| <b>Description</b> | <p>Multiply 2 signed 32-bit integers, subtract the product from a signed 32-bit or 64-bit integer and put the result into a 32-bit or 64-bit register.</p> <p>The value const9 is sign-extended to 32 bits before the multiplication is performed.</p> <p>Overflow and advanced overflow are calculated on the final result.</p> <p><b>(S)</b> On overflow the result is saturated</p>   |
| <b>Operation</b>   | 1. $D[c][31:0] = D[d][31:0] - (D[a][31:0] * D[b][31:0])$ ;signed<br>2. $D[c][31:0] = D[d][31:0] - (D[a][31:0] * \text{sign\_ext}(\text{const9}))$ ;signed<br>3. $E[c][63:0] = E[d][63:0] - (D[a][31:0] * D[b][31:0])$ ;signed<br>4. $E[c][63:0] = E[d][63:0] - (D[a][31:0] * \text{sign\_ext}(\text{const9}))$ ;signed<br>5. $D[c][31:0] = D[d][31:0] - (D[a][31:0] * D[b][31:0])$ ;signed;ssov<br>6. $D[c][31:0] = D[d][31:0] - (D[a][31:0] * \text{sign\_ext}(\text{const9}))$ ;signed;ssov<br>7. $E[c][63:0] = E[d][63:0] - (D[a][31:0] * D[b][31:0])$ ;signed;ssov<br>8. $E[c][63:0] = E[d][63:0] - (D[a][31:0] * \text{sign\_ext}(\text{const9}))$ ;signed;ssov |
| <b>Status</b>      | V, SV, AV, SAV   |
| <b>Examples</b>    |  |



**10.4.142 Packed Multiply-Sub Q Format . . . . .MSUB(S).H**
**Table 10-163**
**MSUB(S).H**

|                    |   |
|--------------------|---|
| <b>Syntax</b>      | 1. msub.h      Ec,Ed,Da,DbUL,n    ; 32  32 -  - (16U*16U    16L*16L)--> 32  32<br>2. msub.h      Ec,Ed,Da,DbLU,n    ; 32  32 -  - (16U*16L    16L*16U)--> 32  32<br>3. msub.h      Ec,Ed,Da,DbLL,n    ; 32  32 -  - (16U*16L    16L*16L)--> 32  32<br>4. msub.h      Ec,Ed,Da,DbUU,n    ; 32  32 -  - (16L*16U    16U*16U)--> 32  32<br>5. msubs.h     Ec,Ed,Da,DbUL,n    ; 32  32 -  - (16U*16U    16L*16L)--> 32  32 sat<br>6. msubs.h     Ec,Ed,Da,DbLU,n    ; 32  32 -  - (16U*16L    16L*16U)--> 32  32 sat<br>7. msubs.h     Ec,Ed,Da,DbLL,n    ; 32  32 -  - (16U*16L    16L*16L)--> 32  32 sat<br>8. msubs.h     Ec,Ed,Da,DbUU,n    ; 32  32 -  - (16L*16U    16U*16U)--> 32  32 sat  |
| <b>Description</b> | <p>This operation is duplicated.</p> <p>Multiply 2 signed 16-bit (halfword) values, subtract the product ( left justified if n=1) from a signed 32-bit value and put the result into a 32-bit register.</p> <p>There are 4 cases of halfword multiplication: (1) upper * upper and lower*lower, (2) upper*lower and lower*upper, (3) upper*lower and lower*lower, (4) lower*upper and upper*upper.</p> <p>Overflow and advanced overflow are calculated on the final result.</p> <p>If n=1, 0x8000 * 0x8000 = 0x7FFFFFFF</p> <p><b>(S)</b> On overflow each result is independently saturated.</p>  |
| <b>Operation</b>   | 1. $E[c][63:32] = E[d][63:32] - (((D[a][31:16] * D[b][31:16]) << n)[31:0])$ ;<br>$E[c][31:0] = E[d][31:0] - (((D[a][15:0] * D[b][15:0]) << n)[31:0])$ ;<br>2. $E[c][63:32] = E[d][63:32] - (((D[a][31:16] * D[b][15:0]) << n)[31:0])$ ;<br>$E[c][31:0] = E[d][31:0] - (((D[a][15:0] * D[b][31:16]) << n)[31:0])$ ;<br>3. $E[c][63:32] = E[d][63:32] - (((D[a][31:16] * D[b][15:0]) << n)[31:0])$ ;<br>$E[c][31:0] = E[d][31:0] - (((D[a][15:0] * D[b][15:0]) << n)[31:0])$ ;<br>4. $E[c][63:32] = E[d][63:32] - (((D[a][15:0] * D[b][31:16]) << n)[31:0])$ ;<br>$E[c][31:0] = E[d][31:0] - (((D[a][31:16] * D[b][31:16]) << n)[31:0])$ ;<br>5. $E[c][63:32] = E[d][63:32] - (((D[a][31:16] * D[b][31:16]) << n)[31:0])$ ;ssov<br>$E[c][31:0] = E[d][31:0] - (((D[a][15:0] * D[b][15:0]) << n)[31:0])$ ;ssov<br>6. $E[c][63:32] = E[d][63:32] - (((D[a][31:16] * D[b][15:0]) << n)[31:0])$ ;ssov<br>$E[c][31:0] = E[d][31:0] - (((D[a][15:0] * D[b][31:16]) << n)[31:0])$ ;sssov<br>7. $E[c][63:32] = E[d][63:32] - (((D[a][31:16] * D[b][15:0]) << n)[31:0])$ ;ssov<br>$E[c][31:0] = E[d][31:0] - (((D[a][15:0] * D[b][15:0]) << n)[31:0])$ ;ssov<br>8. $E[c][63:32] = E[d][63:32] - (((D[a][15:0] * D[b][31:16]) << n)[31:0])$ ;ssov<br>$E[c][31:0] = E[d][31:0] - (((D[a][31:16] * D[b][31:16]) << n)[31:0])$ ;ssov<br>for all operations = signed; n=0,1 |
| <b>Status</b>      | V, SV, AV, SAV  |
| <b>Examples</b>    |   |

2001-04-30 @ 15:16

#### 10.4.143 Multiply-Sub Q Format . . . . .MSUB(S).Q

**Table 10-164**  
**MSUB(S).Q**

|                    |   |  |  |  |  |
|--------------------|---|--|--|--|--|
| <b>Syntax</b>      | <ol style="list-style-type: none"> <li>1. msub.q      Dc,Dd,Da,Db,n      ; 32 - (32*32)Up      --&gt; 32</li> <li>2. msub.q      Dc,Dd,Da,DbU,n      ; 32 - (16U*32)Up      --&gt; 32</li> <li>3. msub.q      Dc,Dd,Da,DbL,n      ; 32 - (16L*32)Up      --&gt; 32</li> <li>4. msub.q      Dc,Dd,DaU,DbU,n      ; 32 - (16U*16U)      --&gt; 32</li> <li>5. msub.q      Dc,Dd,DaL,DbL,n      ; 32 - (16L*16L)      --&gt; 32</li> <li>6. msub.q      Ec,Ed,Da,Db,n      ; 64 - (32*32)      --&gt; 64</li> <li>7. msub.q      Ec,Ed,Da,DbU,n      ; 64 - (16U* 32)      --&gt; 64</li> <li>8. msub.q      Ec,Ed,Da,DbL,n      ; 64 - (16L* 32)      --&gt; 64</li> <li>9. msub.q      Ec,Ed,DaU,DbU,n      ; 64 - (16U*16U)      --&gt; 64</li> <li>10. msub.q      Ec,Ed,DaL,DbL,n      ; 64 - (16L*16L)      --&gt; 64</li> <li>11. msubs.q      Dc,Dd,Da,Db,n      ; 32 - (32*32)Up      --&gt; 32      sat</li> <li>12. msubs.q      Dc,Dd,Da,DbU,n      ; 32 - (16U*32)Up      --&gt; 32      sat</li> <li>13. msubs.q      Dc,Dd,Da,DbL,n      ; 32 - (16L*32)Up      --&gt; 32      sat</li> <li>14. msubs.q      Dc,Dd,DaU,DbU,n      ; 32 - (16U*16U)      --&gt; 32      sat</li> <li>15. msubs.q      Dc,Dd,DaL,DbL,n      ; 32 - (16L*16L)      --&gt; 32      sat</li> <li>16. msubs.q      Ec,Ed,Da,Db,n      ; 64 - (32*32)      --&gt; 64      sat</li> <li>17. msubs.q      Ec,Ed,Da,DbU,n      ; 64 - (16U* 32)      --&gt; 64      sat</li> <li>18. msubs.q      Ec,Ed,Da,DbL,n      ; 64 - (16L* 32)      --&gt; 64      sat</li> <li>19. msubs.q      Ec,Ed,DaU,DbU,n      ; 64 - (16U*16U)      --&gt; 64      sat</li> <li>20. msubs.q      Ec,Ed,DaL,DbL,n      ; 64 - (16L*16L)      --&gt; 64      sat</li> </ol> |  |  |  |  |
| <b>Description</b> | <p>Multiply 2 signed 16-bit or 32-bit values, subtract the product (left justified if n=1) from a signed 32-bit or 64-bit value and put the result into a 32-bit or 64-bit register. There are 8 cases of 16*16 operations, 8 cases of 16x32 operations and 4 cases of 32*32 operations.</p> <p>Overflow and advanced overflow are calculated on the final result.</p> <p>If n=1, 0x8000 * 0x8000 = 0x7FFFFFFF.</p> <p><b>(S)</b> On overflow the result is saturated.</p>  |  |  |  |  |

**Table 10-164**  
**MSUB(S).Q (cont'd)**

|                  |  |
|------------------|--|
| <b>Operation</b> | <ol style="list-style-type: none"> <li>1. <math>D[c][31:0] = D[d][31:0] - (((D[a][31:0] * D[b][31:0]) &lt;&lt; n)[63:32])</math> ;</li> <li>2. <math>D[c][31:0] = D[d][31:0] - (((D[a][31:0] * D[b][31:16]) &lt;&lt; n)[47:16])</math></li> <li>3. <math>D[c][31:0] = D[d][31:0] - (((D[a][31:0] * D[b][15:0]) &lt;&lt; n)[47:16])</math> ;</li> <li>4. <math>D[c][31:0] = D[d][31:0] - (((D[a][31:16] * D[b][31:16]) &lt;&lt; n)[31:0])</math> ;</li> <li>5. <math>D[c][31:0] = D[d][31:0] - (((D[a][15:0] * D[b][15:0]) &lt;&lt; n)[31:0])</math> ;</li> <li>6. <math>E[c][63:0] = E[d][63:0] - (((D[a][31:0] * D[b][31:0]) &lt;&lt; n)[63:0])</math> ;</li> <li>7. <math>E[c][63:0] = E[d][63:0] - (((D[a][31:0] * D[b][31:16]) &lt;&lt; n)[47:0])</math> ;</li> <li>8. <math>E[c][63:0] = E[d][63:0] - (((D[a][31:0] * D[b][15:0]) &lt;&lt; n)[47:0])</math> ;</li> <li>9. <math>E[c][63:0] = E[d][63:0] - (((D[a][31:16] * D[b][31:16]) &lt;&lt; n)[31:0] &lt;&lt; 16)</math>;</li> <li>10. <math>E[c][63:0] = E[d][63:0] - (((D[a][15:0] * D[b][15:0]) &lt;&lt; n)[31:0] &lt;&lt; 16)</math>;</li> <li>11. <math>D[c][31:0] = D[d][31:0] - (((D[a][31:0] * D[b][31:0]) &lt;&lt; n)[63:32])</math></li> <li>12. <math>D[c][31:0] = D[d][31:0] - (((D[a][31:0] * D[b][31:16]) &lt;&lt; n)[47:16])</math> ssov</li> <li>13. <math>D[c][31:0] = D[d][31:0] - (((D[a][31:0] * D[b][15:0]) &lt;&lt; n)[47:16])</math> ; ssov</li> <li>14. <math>D[c][31:0] = D[d][31:0] - (((D[a][31:16] * D[b][31:16]) &lt;&lt; n)[31:0])</math> ; ssov</li> <li>15. <math>D[c][31:0] = D[d][31:0] - (((D[a][15:0] * D[b][15:0]) &lt;&lt; n)[31:0])</math> ; ssov</li> <li>16. <math>E[c][63:0] = E[d][63:0] - (((D[a][31:0] * D[b][31:0]) &lt;&lt; n)[63:0])</math> ; ssov</li> <li>17. <math>E[c][63:0] = E[d][63:0] - (((D[a][31:0] * D[b][31:16]) &lt;&lt; n)[47:0])</math> ; ssov</li> <li>18. <math>E[c][63:0] = E[d][63:0] - (((D[a][31:0] * D[b][15:0]) &lt;&lt; n)[47:0])</math> ; ssov</li> <li>19. <math>E[c][63:0] = E[d][63:0] - (((D[a][31:0] * D[b][31:0]) &lt;&lt; n)[31:0] &lt;&lt; 16)</math> ; ssov</li> <li>20. <math>E[c][63:0] = E[d][63:0] - (((D[a][15:0] * D[b][15:0]) &lt;&lt; n)[31:0] &lt;&lt; 16)</math> ; ssov</li> </ol> <p>for all operations = signed; n=0,1</p> |
| <b>Status</b>    | V, SV, AV, SAV   |
| <b>Examples</b>  |  |

2001-04-30 @ 15:16

#### 10.4.144 Multiply-Sub Unsigned . . . . .MSUB(S).U

**Table 10-165**  
**MSUB(S).U**

|                    |   |
|--------------------|---|
| <b>Syntax</b>      | 1. msub.u     Dc,Dd,Da,Db     ; 32 - (32*32)--> 32   unsigned<br>2. msub.u     Dc,Dd,Da,const9   ; 32 - (32*K9)--> 32   unsigned<br>3. msub.u     Ec,Ed,Da,Db     ; 64 - (32*32)--> 64   unsigned<br>4. msub.u     Ec,Ed,Da,const9   ; 64 - (32*K9)--> 64   unsigned<br>5. msubs.u     Dc,Dd,Da,Db     ; 32 - (32*32)--> 32   unsigned sat<br>6. msubs.u     Dc,Dd,Da,const9   ; 32 - (32*K9)--> 32   unsigned sat<br>7. msubs.u     Ec,Ed,Da,Db     ; 64 - (32*32)--> 64   unsigned sat<br>8. msubs.u     Ec,Ed,Da,const9   ; 64 - (32*K9)--> 64   unsigned sat  |
| <b>Description</b> | <p>Multiply 2 unsigned 32-bit integers, subtract the product from an unsigned 32-bit or 64-bit integer, and put the result into a 32-bit or 64-bit register.</p> <p>The value const9 is zero-extended to 32 bits before the multiplication is performed.</p> <p>Overflow and advanced overflow are calculated on the final result.</p> <p><b>(S)</b> On overflow the result is saturated</p>  |
| <b>Operation</b>   | 1. $D[c][31:0] = D[d][31:0] - (D[a][31:0] * D[b][31:0])$ ;signed<br>2. $D[c][31:0] = D[d][31:0] - (D[a][31:0] * \text{zero\_ext}(\text{const9}))$ ;signed<br>3. $E[c][63:0] = E[d][63:0] - (D[a][31:0] * D[b][31:0])$ ;signed<br>4. $E[c][63:0] = E[d][63:0] - (D[a][31:0] * \text{zero\_ext}(\text{const9}))$ ;signed<br>5. $D[c][31:0] = D[d][31:0] - (D[a][31:0] * D[b][31:0])$ ;signed;suov<br>6. $D[c][31:0] = D[d][31:0] - (D[a][31:0] * \text{zero\_ext}(\text{const9}))$ ;signed ;suov<br>7. $E[c][63:0] = E[d][63:0] - (D[a][31:0] * D[b][31:0])$ ;signed;suov<br>8. $E[c][63:0] = E[d][63:0] - (D[a][31:0] * \text{zero\_ext}(\text{const9}))$ ;signed;suov |
| <b>Status</b>      | V, SV, AV, SAV  |
| <b>Examples</b>    |   |

**10.4.145 Packed Multiply-Sub/Add Q Format . . . . . MSUBAD(S).H**
**Table 10-166  
MSUBAD(S).H**

|                    |   |
|--------------------|---|
| <b>Syntax</b>      | 1. msubad.h Ec,Ed,Da,DbUL,n ;32  32 -  + (16U*16U    16L*16L)--> 32  32<br>2. msubad.h Ec,Ed,Da,DbLU,n ;32  32 -  + (16U*16L    16L*16U)--> 32  32<br>3. msubad.h Ec,Ed,Da,DbLL,n ;32  32 -  + (16U*16L    16L*16L) --> 32  32<br>4. msubad.h Ec,Ed,Da,DbUU,n ;32  32 -  + (16L*16U    16U*16U) --> 32  32<br>5. msubads.h Ec,Ed,Da,DbUL,n ;32  32 -  + (16U*16U    16L*16L)--> 32  32 sat<br>6. msubads.h Ec,Ed,Da,DbLU,n ;32  32 -  + (16U*16L    16L*16U)--> 32  32 sat<br>7. msubads.h Ec,Ed,Da,DbLL,n ;32  32 -  + (16U*16L    16L*16L)--> 32  32 sat<br>8. msubads.h Ec,Ed,Da,DbUU,n ;32  32 -  + (16L*16U    16U*16U) --> 32  32 sat   |
| <b>Description</b> | <p>This operation is duplicated.</p> <p>Multiply 2 signed 16-bit (halfword) values, subtract(or add) the product (left justified if n=1) from a signed 32-bit value and put the result into a 32-bit register. There are 4 cases of halfword multiplication: (1) upper * upper and lower*lower, (2) upper*lower and lower*upper, (3) upper*lower and lower*lower, (4) lower*upper and upper*upper.</p> <p>Overflow and advanced overflow are calculated on the final results.</p> <p>If n=1, 0x8000 * 0x8000 = 0x7FFFFFFF</p> <p><b>(S)</b> On overflow each result is independently saturated.</p>   |
| <b>Operation</b>   | 1. $E[c][63:32] = E[d][63:32] - (((D[a][31:16] * D[b][31:16]) < n)[31:0]);$<br>$E[c][31:0] = E[d][31:0] + (((D[a][15:0] * D[b][15:0]) < n)[31:0]);$ ;<br>2. $E[c][63:32] = E[d][63:32] - (((D[a][31:16] * D[b][15:0]) < n)[31:0]);$<br>$E[c][31:0] = E[d][31:0] + (((D[a][15:0] * D[b][31:16]) < n)[31:0]);$ ;<br>3. $E[c][63:32] = E[d][63:32] - (((D[a][31:16] * D[b][15:0]) < n)[31:0]);$<br>$E[c][31:0] = E[d][31:0] + (((D[a][15:0] * D[b][15:0]) < n)[31:0]);$ ;<br>4. $E[c][63:32] = E[d][63:32] - (((D[a][15:0] * D[b][31:16]) < n)[31:0]);$<br>$E[c][31:0] = E[d][31:0] + (((D[a][31:16] * D[b][31:16]) < n)[31:0]);$ ;<br>5. $E[c][63:32] = E[d][63:32] - (((D[a][31:16] * D[b][31:16]) < n)[31:0]);$ ;ssov<br>$E[c][31:0] = E[d][31:0] + (((D[a][15:0] * D[b][15:0]) < n)[31:0]);$ ;ssov<br>6. $E[c][63:32] = E[d][63:32] - (((D[a][31:16] * D[b][15:0]) < n)[31:0]);$ ;ssov<br>$E[c][31:0] = E[d][31:0] + (((D[a][15:0] * D[b][31:16]) < n)[31:0]);$ ;ssov<br>7. $E[c][63:32] = E[d][63:32] - (((D[a][31:16] * D[b][15:0]) < n)[31:0]);$ ;ssov<br>$E[c][31:0] = E[d][31:0] + (((D[a][15:0] * D[b][15:0]) < n)[31:0]);$ ;ssov<br>8. $E[c][63:32] = E[d][63:32] - (((D[a][15:0] * D[b][31:16]) < n)[31:0]);$ ;ssov<br>$E[c][31:0] = E[d][31:0] + (((D[a][31:16] * D[b][31:16]) < n)[31:0]);$ ;ssov<br>for all operations = signed ; n= 0,1; |
| <b>Status</b>      | V, SV, AV, SAV  |
| <b>Examples</b>    |   |

2001-04-30 @ 15:16

#### 10.4.146 Packed Multiply-Sub/Add Q Format-Multiprecision .....MSUBADM(S).H

Table 10-167  
MSUBADM(S).H

|                    |  |
|--------------------|--|
| <b>Syntax</b>      | 1. msubadm.h Ec,Ed,Da,DbUL,n ; 64 - 16U*16U + 16L*16L --> 64<br>2. msubadm.h Ec,Ed,Da,DbLU,n ; 64 - 16U*16L + 16L*16U --> 64<br>3. msubadm.h Ec,Ed,Da,DbLL,n ; 64 - 16U*16L + 16L*16L --> 64<br>4. msubadm.h Ec,Ed,Da,DbUU,n ; 64 - 16L*16U + 16U*16U --> 64<br>5. msubadms.h Ec,Ed,Da,DbUL,n ; 64 - 16U*16U + 16L*16L --> 64 sat<br>6. msubadms.h Ec,Ed,Da,DbLU,n ; 64 - 16U*16L + 16L*16U --> 64 sat<br>7. msubadms.h Ec,Ed,Da,DbLL,n ; 64 - 16U*16L + 16L*16L --> 64 sat<br>8. msubadms.h Ec,Ed,Da,DbUU,n ; 64 - 16L*16U + 16U*16U--> 64 sat  |
| <b>Description</b> | <p>Perform 2 multiplications of 2 signed 16-bit (halfword). Subtract 1 product and add the other product (left justified if n=1) (<b>left-shifted by 16</b>) from/to a signed 64-bit value and put the result in a 64-bit register.</p> <p>There are 4 cases of halfword multiplication: (1) upper * upper and lower*lower, (2) upper*lower and lower*upper, (3) upper*lower and lower*lower, (4) lower*upper and upper*upper.</p> <p>Overflow and advanced overflow are calculated on the final result.</p> <p>If n=1, 0x8000 * 0x8000 = 0x7FFFFFFF</p> <p><b>(S)</b> On overflow the result is saturated.</p>  |
| <b>Operation</b>   | 1. $E[c][63:0] = E[d][63:0] - (((D[a][31:16] * D[b][31:16]) << n) - ((D[a][31:16] * D[b][31:16]) << n) << 16); \text{signed}, n=0,1$<br>2. $E[c][63:0] = E[d][63:0] - (((D[a][31:16] * D[b][15:0]) << n) - ((D[a][15:0] * D[b][31:16]) << n) << 16); \text{signed}, n=0,1$<br>3. $E[c][63:0] = E[d][63:0] - (((D[a][31:16] * D[b][15:0]) << n) - ((D[a][15:0] * D[b][15:0]) << n) << 16); \text{signed}, n=0,1$<br>4. $E[c][63:0] = E[d][63:0] - (((D[a][15:0] * D[b][31:16]) << n) - ((D[a][31:16] * D[b][31:16]) << n) << 16); \text{signed}, n=0,1$<br>5. $E[c][63:0] = E[d][63:0] - (((D[a][31:16] * D[b][31:16]) << n) - ((D[a][31:16] * D[b][31:16]) << n) << 16); \text{signed}, n=0,1; \text{ssov}$<br>6. $E[c][63:0] = E[d][63:0] - (((D[a][31:16] * D[b][15:0]) << n) - ((D[a][15:0] * D[b][31:16]) << n) << 16); \text{signed}, n=0,1; \text{ssov}$<br>7. $E[c][63:0] = E[d][63:0] - (((D[a][31:16] * D[b][15:0]) << n) - ((D[a][15:0] * D[b][15:0]) << n) << 16); \text{signed}, n=0,1; \text{ssov}$<br>8. $E[c][63:0] = E[d][63:0] - (((D[a][15:0] * D[b][31:16]) << n) - ((D[a][31:16] * D[b][31:16]) << n) << 16); \text{signed}, n=0,1; \text{ssov}$ |
| <b>Status</b>      | V, SV, AV, SAV   |
| <b>Examples</b>    |  |

**10.4.147 Packed Multiply-Sub/Add Q Format w/Rounding .....MSUBADR(S).H**
**Table 10-168**
**MSUBADR(S).H**

|                    |   |
|--------------------|---|
| <b>Syntax</b>      | 1. msubadr.h Dc,Dd,Da,DbUL,n ;16U  16L -  + (16U*16U    16L*16L) R--> 16  16<br>2. msubadr.h Dc,Dd,Da,DbLU,n ;16U  16L -  + (16U*16L    16L*16U) R--> 16  16<br>3. msubadr.h Dc,Dd,Da,DbLL,n ;16U  16L -  + (16U*16L    16L*16L) R --> 16  16<br>4. msubadr.h Dc,Dd,Da,DbUU,n ;16U  16L -  + (16L*16U  16U*16U) R -->16  16<br>5. msubadrs.h Dc,Dd,Da,DbUL,n ;16U  16L -  + (16U*16U   16L*16L) R --> 16  16 sat<br>6. msubadrs.h Dc,Dd,Da,DbLU,n ;16U  16L -  + (16U*16L    16L*16U) R --> 16  16 sat<br>7. msubadrs.h Dc,Dd,Da,DbLL,n ;16U  16L -  + (16U*16L    16L*16L) R--> 16  16 sat<br>8. msubadrs.h Dc,Dd,Da,DbUU,n ;16U  16L -  + (16L*16U   16U*16U) R--> 16  16 sat   |
| <b>Description</b> | <p>This operation is duplicated.</p> <p>Multiply 2 signed 16-bit (halfword) values, subtract (or add) the product (left justified if n=1) from a signed 16-bit value and put the <b>rounded</b> result into <b>half</b> of a 32-bit register.(Note: since there are 2 results the two register halves are used) . There are 4 cases of halfword multiplication: (1) upper * upper and lower*lower, (2) upper*lower and lower*upper, (3) upper*lower and lower*lower, (4) lower*upper and upper*upper.</p> <p>Overflow and advanced overflow are calculated on the final results.</p> <p>If n=1, 0x8000 * 0x8000 = 0x7FFFFFFF</p> <p><b>(S)</b> On overflow each result is independently saturated.</p>  |
| <b>Operation</b>   | 1. D[c][31:16] = round16(D[d][31:16] - ((D[a][31:16]* D[b][31:16])<<n)[31:0])<br>D[c][15:0] = round16(D[d][15:0] + ((D[a][15:0] * D[b][15:0]) <<n)[31:0])<br>2. D[c][31:16] = round16(D[d][31:16] - ((D[a][31:16]* D[b][15:0]) <<n)[31:0])<br>D[c][15:0] = round16(D[d][15:0] + ((D[a][15:0] * D[b][31:16])<<n)[31:0])<br>3. D[c][31:16] = round16(D[d][31:16] - ((D[a][31:16]* D[b][15:0]) <<n)[31:0])<br>D[c][15:0] = round16(D[d][15:0] + ((D[a][15:0] * D[b][15:0]) <<n)[31:0])<br>4. D[c][31:16] = round16(D[d][31:16] - ((D[a][15:0] * D[b][31:16])<<n)[31:0])<br>D[c][15:0] = round16(D[d][15:0] + ((D[a][31:16]* D[b][31:16])<<n)[31:0])<br>5. D[c][31:16] = round16(D[d][31:16] - ((D[a][31:16]* D[b][31:16])<<n)[31:0]);ssov<br>D[c][15:0] = round16(D[d][15:0] + ((D[a][15:0] * D[b][15:0]) <<n)[31:0]);ssov<br>6. D[c][31:16] = round16(D[d][31:16] - ((D[a][31:16]* D[b][15:0]) <<n)[31:0]);ssov<br>D[c][15:0] = round16(D[d][15:0] + ((D[a][15:0] * D[b][31:16])<<n)[31:0]);ssov<br>7. D[c][31:16] = round16(D[d][31:16] - ((D[a][31:16]* D[b][15:0]) <<n)[31:0]);ssov<br>D[c][15:0] = round16(D[d][15:0] + ((D[a][15:0] * D[b][15:0]) <<n)[31:0]);ssov<br>8. D[c][31:16] = round16(D[d][31:16] - ((D[a][15:0] * D[b][31:16])<<n)[31:0]);ssov<br>D[c][15:0] = round16(D[d][15:0] + ((D[a][31:16]* D[b][31:16])<<n)[31:0]);ssov<br>for all operations = signed ; n=0,1;<br>round16 == add 0x8000 to 32bit value and clear bits[15:0] |
| <b>Status</b>      | V, SV, AV, SAV  |
| <b>Examples</b>    |   |

2001-04-30 @ 15:16

#### 10.4.148 Packed Multiply-Sub Q Format-Multiprecision ..... MSUBM(S).H

Table 10-169  
MSUBM(S).H

|                    |  |
|--------------------|--|
| <b>Syntax</b>      | 1. msubm.h Ec,Ed,Da,DbUL,n ;64 - 16U*16U - 16L*16L --> 64<br>2. msubm.h Ec,Ed,Da,DbLU,n ;64 - 16U*16L - 16L*16U --> 64<br>3. msubm.h Ec,Ed,Da,DbLL,n ;64 - 16U*16L - 16L*16L --> 64<br>4. msubm.h Ec,Ed,Da,DbUU,n ;64 - 16L*16U - 16U*16U --> 64<br>5. msubms.h Ec,Ed,Da,DbUL,n ;64 - 16U*16U - 16L*16L --> 64 sat<br>6. msubms.h Ec,Ed,Da,DbLU,n ;64 - 16U*16L - 16L*16U --> 64 sat<br>7. msubms.h Ec,Ed,Da,DbLL,n ;64 - 16U*16L - 16L*16L --> 64 sat<br>8. msubms.h Ec,Ed,Da,DbUU,n ;64 - 16L*16U - 16U*16U --> 64 sat   |
| <b>Description</b> | <p>Perform 2 multiplications of 2 signed 16-bit (halfword). Subtract the 2 products (left justified if n=1) (<b>left-shifted by 16</b>) from a signed 64-bit value and put the result in a 64-bit register.</p> <p>There are 4 cases of halfword multiplication: (1) upper*upper and lower*lower, (2) upper*lower and lower*upper, (3) upper*lower and lower*lower, (4) lower*upper and upper*upper.</p> <p>Overflow and advanced overflow are calculated on the final result.</p> <p>If n=1, 0x8000 * 0x8000 = 0x7FFFFFFF</p> <p><b>(S)</b> On overflow the result is saturated.</p>  |
| <b>Operation</b>   | 1. $E[c][63:0] = E[d][63:0] - (((D[a][31:16] * D[b][31:16]) << n) + ((D[a][15:0] * D[b][15:0]) << n) << 16); \text{signed}, n=0,1$<br>2. $E[c][63:0] = E[d][63:0] - (((D[a][31:16] * D[b][15:0]) << n) + ((D[a][15:0] * D[b][31:16]) << n) << 16); \text{signed}, n=0,1$<br>3. $E[c][63:0] = E[d][63:0] - (((D[a][31:16] * D[b][15:0]) << n) + ((D[a][15:0] * D[b][15:0]) << n) << 16); \text{signed}, n=0,1$<br>4. $E[c][63:0] = E[d][63:0] - (((D[a][15:0] * D[b][31:16]) << n) + ((D[a][31:16] * D[b][31:16]) << n) << 16); \text{signed}, n=0,1$<br>5. $E[c][63:0] = E[d][63:0] - (((D[a][31:16] * D[b][31:16]) << n) + ((D[a][15:0] * D[b][15:0]) << n) << 16); \text{signed}, n=0,1; \text{ssov}$<br>6. $E[c][63:0] = E[d][63:0] - (((D[a][31:16] * D[b][15:0]) << n) + ((D[a][15:0] * D[b][31:16]) << n) << 16); \text{signed}, n=0,1; \text{ssov}$<br>7. $E[c][63:0] = E[d][63:0] - (((D[a][31:16] * D[b][15:0]) << n) + ((D[a][15:0] * D[b][15:0]) << n) << 16); \text{signed}, n=0,1; \text{ssov}$<br>8. $E[c][63:0] = E[d][63:0] - (((D[a][15:0] * D[b][31:16]) << n) + ((D[a][31:16] * D[b][31:16]) << n) << 16); \text{signed}, n=0,1; \text{ssov}$ |
| <b>Status</b>      | V, SV, AV, SAV   |
| <b>Examples</b>    |  |



10.4.149 Packed Multiply-Sub Q Format w/Rounding ..... MSUBR(S).H

Table 10-170  
MSUBR(S).H

| Syntax       |   |
|--------------|---|
| 1. msubr.h   | Dc,Dd,Da,DbUL,n ;16U  16L -  - (16U*16U    16L*16L) R-->16  16  |
| 2. msubr.h   | Dc,Dd,Da,DbLU,n ;16U  16L -  - (16U*16L    16L*16U) R-->16  16  |
| 3. msubr.h   | Dc,Dd,Da,DbLL,n ;16U  16L -  - (16U*16L    16L*16L) R-->16  16  |
| 4. msubr.h   | Dc,Dd,Da,DbUU,n ;16U  16L -  - (16L*16U    16U*16U) R -->16  16   |
| 5. msubr.h   | Dc,Ed,Da,DbUL,n ;32   32 -  - (16U*16U    16L*16L) R--> 16  16  |
| 6. msubrs.h  | Dc,Dd,Da,DbUL,n ;16U  16L -  - (16U*16U    16L*16L) R-->16  16 sat  |
| 7. msubrs.h  | Dc,Dd,Da,DbLU,n ;16U  16L -  - (16U*16L    16L*16U) R-->16  16sat   |
| 8. msubrs.h  | Dc,Dd,Da,DbLL,n ;16U  16L -  - (16U*16L    16L*16L) R-->16  16sat   |
| 9. msubrs.h  | Dc,Dd,Da,DbUU,n ;16U  16L -  - (16L*16U    16U*16U) R-->16  16sat   |
| 10. msubrs.h | Dc,Ed,Da,DbUL,n ;32  32 -  - (16U*16U    16L*16L) R-->16  16sat   |
| Description  | <p>This operation is duplicated.</p> <p>Multiply 2 signed 16-bit (halfword) values, subtract the product ( left justified if n=1) from a signed 16-bit value and put the <b>rounded</b> result into <b>half</b> of a 32-bit register.(Note= since there are 2 results the two register halves are used) .</p> <p>There are 4 cases of halfword multiplication: (1) upper * upper and lower*lower, (2) upper*lower and lower*upper, (3) upper*lower and lower*lower, (4) lower*upper and upper*upper.</p> <p>There is a special case: subtract ( left justified if n=1) from a <b>32-bit value</b> and put the rounded result into half of a 32-bit register.</p> <p>Overflow and advanced overflow are calculated on the final results.</p> <p>If n=1, 0x8000 * 0x8000 = 0x7FFFFFFF</p> <p><b>(S)</b> On overflow each result is independently saturated.</p> |

2001-04-30 @ 15:16

**Table 10-170**  
**MSUBR(S).H (cont'd)**

|                  |   |
|------------------|---|
| <b>Operation</b> | <ol style="list-style-type: none"> <li>1. <math>D[c][31:16] = \text{round16}(D[d][31:16] - ((D[a][31:16] * D[b][31:16]) \ll n)[31:0])</math> ;<br/> <math>D[c][15:0] = \text{round16}(D[d][15:0] - ((D[a][15:0] * D[b][15:0]) \ll n)[31:0])</math> ;</li> <li>2. <math>D[c][31:16] = \text{round16}(D[d][31:16] - ((D[a][31:16] * D[b][15:0]) \ll n)[31:0])</math> ;<br/> <math>D[c][15:0] = \text{round16}(D[d][15:0] - ((D[a][15:0] * D[b][31:16]) \ll n)[31:0])</math> ;</li> <li>3. <math>D[c][31:16] = \text{round16}(D[d][31:16] - ((D[a][31:16] * D[b][15:0]) \ll n)[31:0])</math> ;<br/> <math>D[c][15:0] = \text{round16}(D[d][15:0] - ((D[a][15:0] * D[b][15:0]) \ll n)[31:0])</math> ;</li> <li>4. <math>D[c][31:16] = \text{round16}(D[d][31:16] - ((D[a][15:0] * D[b][31:16]) \ll n)[31:0])</math> ;<br/> <math>D[c][15:0] = \text{round16}(D[d][15:0] - ((D[a][31:16] * D[b][31:16]) \ll n)[31:0])</math> ;</li> <li>5. <math>D[c][31:16] = \text{round16}(E[d][63:32] - ((D[a][31:16] * D[b][31:16]) \ll n)[31:0])</math> ;<br/> <math>D[c][15:0] = \text{round16}(E[d][31:0] - ((D[a][15:0] * D[b][15:0]) \ll n)[31:0])</math> ;</li> <li>6. <math>D[c][31:16] = \text{round16}(D[d][31:16] - ((D[a][31:16] * D[b][31:16]) \ll n)[31:0])</math> ;ssov<br/> <math>D[c][15:0] = \text{round16}(D[d][15:0] - ((D[a][15:0] * D[b][15:0]) \ll n)[31:0])</math> ;ssov</li> <li>7. <math>D[c][31:16] = \text{round16}(D[d][31:16] - ((D[a][31:16] * D[b][15:0]) \ll n)[31:0])</math> ;ssov<br/> <math>D[c][15:0] = \text{round16}(D[d][15:0] - ((D[a][15:0] * D[b][31:16]) \ll n)[31:0])</math> ;ssov</li> <li>8. <math>D[c][31:16] = \text{round16}(D[d][31:16] - ((D[a][31:16] * D[b][15:0]) \ll n)[31:0])</math> ;ssov<br/> <math>D[c][15:0] = \text{round16}(D[d][15:0] - ((D[a][15:0] * D[b][15:0]) \ll n)[31:0])</math> ;ssov</li> <li>9. <math>D[c][31:16] = \text{round16}(D[d][31:16] - ((D[a][15:0] * D[b][31:16]) \ll n)[31:0])</math> ;ssov<br/> <math>D[c][15:0] = \text{round16}(D[d][15:0] - ((D[a][31:16] * D[b][31:16]) \ll n)[31:0])</math> ;ssov</li> <li>10. <math>D[c][31:16] = \text{round16}(E[d][63:32] - ((D[a][31:16] * D[b][31:16]) \ll n)[31:0])</math> ;ssov<br/> <math>D[c][15:0] = \text{round16}(E[d][31:0] - (((D[a][15:0] * D[b][15:0]) \ll n)[31:0])</math> ;ssov</li> </ol> <p>for all operations = signed ; n=0,1;<br/> round16 == add 0x8000 to 32bit value and clear bits[15:0]</p> |
| <b>Status</b>    | V, SV, AV, SAV  |
| <b>Examples</b>  |   |

2001-04-30 @ 15:16

#### 10.4.150 Multiply-Sub Q Format with Rounding ..... MSUBR(S).Q

**Table 10-171**  
**MSUBR(S).Q**

|                    |   |
|--------------------|---|
| <b>Syntax</b>      | 1. msubr.q    Dc,Dd,DaU,DbU,n   ; 32 - (16U*16U) R--> 32<br>2. msubr.q    Dc,Dd,DaL,DbL,n   ; 32 - (16L*16L) R--> 32<br>3. msubrs.q    Dc,Dd,DaU,DbU,n   ; 32 - (16U*16U) R--> 32 sat<br>4. msubrs.q    Dc,Dd,DaL,DbL,n   ; 32 - (16L*16L) R--> 32 sat  |
| <b>Description</b> | <p>Multiply 2 signed 16-bit (halfword) values, subtract the product (left justified if n=1) from a 32-bit signed value, put the <b>rounded</b> result in a 32-bit register.<br/> The lower halfword is cleared.<br/> Overflow and advanced overflow are calculated on the final results.<br/> If n=1, 0x8000 * 0x8000 = 0x7FFFFFFF (only for 16 by 16-bit operations).<br/> <b>(S)</b> On overflow the result is saturated.</p>   |
| <b>Operation</b>   | 1. $D[c][31:0] = \text{round16}(D[d][31:0] - ((D[a][31:16] * D[b][31:16]) << n)[31:0]); \text{ssov}$<br>2. $D[c][31:0] = \text{round16}(D[d][31:0] - ((D[a][15:0] * D[b][15:0]) << n)[31:0]); \text{ssov}$<br>3. $D[c][31:0] = \text{round16}(D[d][31:0] - ((D[a][31:16] * D[b][31:16]) << n)[31:0]); \text{ssov}$<br>4. $D[c][31:0] = \text{round16}(D[d][31:0] - ((D[a][15:0] * D[b][15:0]) << n)[31:0]); \text{ssov}$<br>for all operations = signed ; n=0,1;<br>round16 == add 0x8000 to 32bit value and clear bits[15:0] |
| <b>Status</b>      | V, SV, AV, SAV  |
| <b>Examples</b>    |   |

#### 10.4.151 Move To Core Register. .... MTCR

**Table 10-172**  
**MTCR**

|                    |   |
|--------------------|---|
| <b>Syntax</b>      | mtcr    const16, Da (RLC)   |
| <b>Description</b> | <p>Move the value in data register Da to the core SFR register selected by the value <i>const16</i>. The core SFR address is a <i>const16</i> byte offset from the core SFR base address. It must be word-aligned (the least-significant 2 bits are zero). Non-aligned address have an undefined effect.<br/> This instruction may not be used to access GPRs. Attempting to update a GPR with this instruction will have no effect.<br/> This instruction can be executed in supervisor mode only.</p> |
| <b>Operation</b>   | $CR[\text{const16}] = D[a]$   |
| <b>Status</b>      | modified by write to PSW core SFR   |
| <b>Examples</b>    | mtcr    4, d1   |
| <b>See also</b>    | MFCR  |

2001-04-30 @ 15:16

### 10.4.152 Multiply.....MUL(S)

**Table 10-173**  
**MUL(S)**

|                    |  |
|--------------------|--|
| <b>Syntax</b>      | 1. mul      Dc,Da,Db            ; (32*32) --> 32    signed<br>2. mul      Dc,Da,const9       ; (32*K9) --> 32    signed<br>3. mul      Ec,Da,Db            ; (32*32) --> 64    signed<br>4. mul      Ec,Da,const9       ; (32*K9) --> 64    signed<br>5. muls     Dc,Da,Db            ; (32*32) --> 32    signed sat   |
| <b>Description</b> | <p>Multiply 2 signed 32-bit integers and put the product into a 32-bit or 64-bit register. The value const9 is sign-extended to 32 bits before the multiplication is performed. Overflow and advanced overflow are calculated on the final result.</p> <p><b>(S)</b> On overflow the result is saturated</p>   |
| <b>Operation</b>   | 1. $D[c][31:0] = D[a][31:0] * D[b][31:0] \quad ;\text{signed}$<br>2. $D[c][31:0] = D[a][31:0] * \text{sign\_ext}(\text{const9}) ;\text{signed}$<br>3. $E[c][63:0] = D[a][31:0] * D[b][31:0] \quad ;\text{signed}$<br>4. $E[c][63:0] = D[a][31:0] * \text{sign\_ext}(\text{const9}) ;\text{signed}$<br>5. $D[c][31:0] = D[a][31:0] * D[b][31:0] \quad ;\text{signed};\text{ssov}$ |
| <b>Status</b>      | V, SV, AV, SAV   |
| <b>Examples</b>    |  |

**10.4.153 Packed Multiply Q Format ..... MUL.H**
**Table 10-174  
MUL.H**

|                    |  |
|--------------------|--|
| <b>Syntax</b>      | 1. mul.h      Ec,Da,DbUL,n      (16U*16U    16L*16L) --> 32  32<br>2. mul.h      Ec,Da,DbLU,n      (16U*16L    16L*16U) --> 32  32<br>3. mul.h      Ec,Da,DbLL,n      (16U*16L    16L*16L) --> 32  32<br>4. mul.h      Ec,Da,DbUU,n      (16L*16U    16U*16U) --> 32  32   |
| <b>Description</b> | <p>This operation is duplicated.</p> <p>Multiply 2 signed 16-bit (halfword) values and put the product ( left justified if n=1) into a 32-bit register.</p> <p>There are 4 cases of halfword multiplication: (1) upper * upper and lower*lower, (2) upper*lower and lower*upper, (3) upper*lower and lower*lower, (4) lower*upper and upper*upper.</p> <p>Overflow =0 ;</p> <p>If n=1, 0x8000 * 0x8000 = 0x7FFFFFFF</p>  |
| <b>Operation</b>   | 1. $E[c][63:32] = (D[a][31:16] * D[b][31:16]) \ll n ; \text{signed}, n=0,1$<br>$E[c][31:0] = (D[a][15:0] * D[b][15:0]) \ll n ; \text{signed}, n=0,1$<br>2. $E[c][63:32] = (D[a][31:16] * D[b][15:0]) \ll n ; \text{signed}, n=0,1$<br>$E[c][31:0] = (D[a][15:0] * D[b][31:16]) \ll n ; \text{signed}, n=0,1$<br>3. $E[c][63:32] = (D[a][31:16] * D[b][15:0]) \ll n ; \text{signed}, n=0,1$<br>$E[c][31:0] = (D[a][15:0] * D[b][15:0]) \ll n ; \text{signed}, n=0,1$<br>4. $E[c][63:32] = (D[a][15:0] * D[b][31:16]) \ll n ; \text{signed}, n=0,1$<br>$E[c][31:0] = (D[a][31:16] * D[b][31:16]) \ll n ; \text{signed}, n=0,1$ |
| <b>Status</b>      | V, SV, AV, SAV   |
| <b>Examples</b>    |  |

2001-04-30 @ 15:16

#### 10.4.154 Multiply Q Format . . . . . MUL.Q

**Table 10-175**  
**MUL.Q**

|                    |  |
|--------------------|--|
| <b>Syntax</b>      | 1. mul.q      Dc,DaU,DbU,n      ;(16U*16U) --> 32<br>2. mul.q      Dc,DaL,DbL,n      ;(16L*16L) --> 32<br>3. mul.q      Dc,Da,DbU,n      ;(16U*32)Up--> 32<br>4. mul.q      Dc,Da,DbL,n      ;(16L*32)Up --> 32<br>5. mul.q      Dc,Da,Db,n      ;(32*32) Up --> 32<br>6. mul.q      Ec,Da,DbU,n      ;(16U*32) --> 64<br>7. mul.q      Ec,Da,DbL,n      ;(16L*32) --> 64<br>8. mul.q      Ec,Da,Db,n      ;(32*32) --> 64   |
| <b>Description</b> | <p>Multiply 2 signed 16-bit or 32-bit values and put the product (left justified if n=1) into a 32-bit or 64-bit register.</p> <p>There are 2 cases of 16*16 operations, 4 cases of 16x32 operations and 2 cases of 32*32 operations.</p> <p>Overflow =0.</p> <p>If n=1, 0x8000 * 0x8000 = 0x7FFFFFFF (only for 16 by 16-bit operations).</p>  |
| <b>Operation</b>   | 1. $D[c][31:0] = (D[a][31:16] * D[b][31:16]) \ll n$ ;signed;n=0,1<br>2. $D[c][31:0] = (D[a][15:0] * D[b][15:0]) \ll n$ ;signed;n=0,1<br>3. $D[c][31:0] = ((D[a][31:0] * D[b][31:16]) \ll n)[47:16]$ ;signed;n=0,1<br>4. $D[c][31:0] = ((D[a][31:0] * D[b][15:0]) \ll n)[47:16]$ ;signed;n=0,1<br>5. $D[c][31:0] = ((D[a][31:0] * D[b][31:0]) \ll n)[63:32]$ ;signed;n=0,1<br>6. $E[c][63:0] = ((D[a][31:0] * D[b][31:16]) \ll n)[47:0]$ ;signed;n=0,1 ; (1)<br>7. $E[c][63:0] = ((D[a][31:0] * D[b][15:0]) \ll n)[47:0]$ ;signed;n=0,1 ; (1)<br>8. $E[c][63:0] = ((D[a][31:0] * D[b][31:0]) \ll n)[63:0]$ ;signed;n=0,1<br>(1) 16*32 --> 64    note : Ec[63:48]=sign; Ec=[47:0]= product |
| <b>Status</b>      | V, SV, AV, SAV   |
| <b>Examples</b>    |  |

**10.4.155 Multiply Unsigned . . . . . MUL(S).U**
**Table 10-176**
**MUL(S).U**

|                    |   |
|--------------------|---|
| <b>Syntax</b>      | 1. mul.u      Ec,Da,Db      ;(32*32) --> 64      unsigned<br>2. mul.u      Ec,Da,const9      ;(32*K9) --> 64      unsigned<br>3. muls.u      Dc,Da,Db      ;(32*32) --> 32      unsigned sat  |
| <b>Description</b> | <p>Multiply 2 unsigned 32-bit integers and put the product into a 32-bit or 64-bit register.</p> <p>The value const9 is zero-extended to 32 bits before the multiplication is performed. Overflow and advanced overflow are calculated on the final result.</p> <p><b>(S)</b> On overflow the result is saturated.</p> <p><b>Note :</b> (32 x 32) -&gt; 32 unsigned is equivalent to (32 by 32) -&gt; signed, so MUL can be used.</p> |
| <b>Operation</b>   | 1. E[c][63:0] = D[a][31:0] * D[b][31:0]      ;unsigned;<br>2. E[c][63:0] = D[a][31:0] * zero_ext(const9)      ;unsigned<br>3. D[c][31:0] = D[a][31:0] * D[b][31:0]      ;unsigned;ssuv  |
| <b>Status</b>      | V, SV, AV, SAV  |
| <b>Examples</b>    |   |

2001-04-30 @ 15:16

**10.4.156 Packed Multiply Q Format-Multiprecision ..... MULM.H**

**Table 10-177**  
**MULM.H**

|                    |   |
|--------------------|---|
| <b>Syntax</b>      | 1. mulm.h      Ec,Da,DbUL,n      ; 16U*16U + 16L*16L    --> 64<br>2. mulm.h      Ec,Da,DbLU,n      ; 16U*16L + 16L*16U    --> 64<br>3. mulm.h      Ec,Da,DbLL,n      ; 16U*16L + 16L*16L    --> 64<br>4. mulm.h      Ec,Da,DbUU,n      ; 16L*16U + 16U*16U    --> 64  |
| <b>Description</b> | Perform 2 multiplications of 2 signed 16-bit (halfword). Add the 2 products ( left justified if n=1) <b>(left-shifted by 16)</b> in a 64-bit register.<br>There are 4 cases of halfword multiplication: (1) upper * upper and lower*lower, (2) upper*lower and lower*upper, (3) upper*lower and lower*lower, (4) lower*upper and upper*upper.<br>Overflow and advanced overflow=0<br>If n=1, 0x8000 * 0x8000 = 0x7FFFFFFF   |
| <b>Operation</b>   | 1. $E[c][63:0] = (((D[a][31:16] * D[b][31:16]) \ll n) + ((D[a][15:0] * D[b][15:0]) \ll n)) \ll 16$ ;<br>2. $E[c][63:0] = (((D[a][31:16] * D[b][15:0]) \ll n) + ((D[a][15:0] * D[b][31:16]) \ll n)) \ll 16$ ;<br>3. $E[c][63:0] = (((D[a][31:16] * D[b][15:0]) \ll n) + ((D[a][15:0] * D[b][15:0]) \ll n)) \ll 16$ ;<br>4. $E[c][63:0] = (((D[a][15:0] * D[b][31:16]) \ll n) + ((D[a][31:16] * D[b][31:16]) \ll n)) \ll 16$ ;<br>for all operations = signed ; n=0,1 |
| <b>Status</b>      | V, SV, AV, SAV  |
| <b>Examples</b>    |   |



**10.4.157 Packed Multiply Q Format with Rounding ..... MULR.H**
**Table 10-178**
**MULR.H**

|                    |   |
|--------------------|---|
| <b>Syntax</b>      | 1. mulr.h      Dc,Da,DbUL,n      ; (16U*16U    16L*16L) R --> 16  16<br>2. mulr.h      Dc,Da,DbLU,n      ; (16U*16L    16L*16U) R --> 16  16<br>3. mulr.h      Dc,Da,DbLL,n      ; (16U*16L    16L*16L) R --> 16  16<br>4. mulr.h      Dc,Da,DbUU,n      ; (16L*16U    16U*16U) R --> 16  16  |
| <b>Description</b> | <p>This operation is duplicated.</p> <p>Multiply 2 signed 16-bit (halfword) values, add the product ( left justified if n=1) to a signed 16-bit value and put the <b>rounded</b> result into <b>half</b> of a 32-bit register.</p> <p>(Note: since there are 2 results the two register halves are used) .</p> <p>There are 4 cases of halfword multiplication: (1) upper * upper and lower*lower, (2) upper*lower and lower*upper, (3) upper*lower and lower*lower, (4) lower*upper and upper*upper.</p> <p>Overflow=0 ;</p> <p>If n=1, 0x8000 * 0x8000 = 0x7FFFFFFF</p>   |
| <b>Operation</b>   | 1. D[c][31:16] = round16(((D[a][31:16] * D[b][31:16])<< n)[31:0])<br>D[c][15:0] = round16(((D[a][15:0] * D[b][15:0] )<< n)[31:0])<br>2. D[c][31:16] = round16(((D[a][31:16] * D[b][15:0] )<< n)[31:0])<br>D[c][15:0] = round16(((D[a][15:0] * D[b][31:16])<< n)[31:0])<br>3. D[c][31:16] = round16(((D[a][31:16] * D[b][15:0] )<< n)[31:0])<br>D[c][15:0] = round16(((D[a][15:0] * D[b][15:0] )<< n)[31:0])<br>4. D[c][31:16] = round16(((D[a][15:0] * D[b][31:16])<< n)[31:0])<br>D[c][15:0] = round16(((D[a][31:16] * D[b][31:16])<< n)[31:0])<br>for all operations = signed ; n=0,1;<br>round16 == add 0x8000 to 32bit value and clear bits[15:0] |
| <b>Status</b>      | V, SV, AV, SAV  |
| <b>Examples</b>    |   |

2001-04-30 @ 15:16

### 10.4.158 Multiply Q Format with Rounding ..... MULR.Q

**Table 10-179**  
**MULR.Q**

|                    |   |
|--------------------|---|
| <b>Syntax</b>      | 1. mulr.q      Dc,DaU,DbU,n      ; (16U*16U) Round --> 32<br>2. mulr.q      Dc,DaL,DbL,n      ; (16L*16L) Round --> 32  |
| <b>Description</b> | Multiply 2 signed 16-bit (halfword) values and put the rounded result (left justified if n=1) into a 32-bit register. The lower halfword is cleared.<br>Overflow =0.<br>If n=1, 0x8000 * 0x8000 = 0x7FFFFFFF (==7FFF0000 after clearing lower half) |
| <b>Operation</b>   | 1. D[c][31:0] = round16(((D[a][31:16]*D[b][31:16])<< n)[31:0])<br>2. D[c][31:0] = round16(((D[a][15:0] *D[b][15:0] )<< n) [31:0])<br>for all operations = signed ; n=0,1;<br>round16 == add 0x8000 to 32bit value and clear bits[15:0]              |
| <b>Status</b>      | V, SV, AV, SAV  |
| <b>Examples</b>    |   |

### 10.4.159 Logical NAND ..... NAND

**Table 10-180**  
**NAND**

|                    |   |
|--------------------|---|
| <b>Syntax</b>      | nand      Dc, Da, Db (RR)<br>nand      Dc, Da, const9 (RC)  |
| <b>Description</b> | Compute the bitwise logical NAND of the contents of data register <i>Da</i> and data register <i>Db/const9</i> and put the result in data register <i>Dc</i> . The <i>const9</i> value is zero-extended to 32 bits. |
| <b>Operation</b>   | D[c] = !(D[a] AND D[b])<br>D[c] = !(D[a] AND zero_ext(const9))  |
| <b>Status</b>      | -   |
| <b>Examples</b>    | nand d3, d1, d2<br>nand d3, d1, 126   |
| <b>See also</b>    | AND, ANDN, NOR, NOT, OR, ORN, XNOR, XOR   |

2001-04-30 @ 15:16

#### 10.4.160 Bit Logical NAND ..... NAND.T

**Table 10-181**  
**NAND.T**

|                    |  |
|--------------------|--|
| <b>Syntax</b>      | nand.t Dc, Da, p1, Db, p2 (BIT)  |
| <b>Description</b> | 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. |
| <b>Operation</b>   | $D[c] = \neg(D[a][p1] \text{ AND } D[b][p2])$  |
| <b>Status</b>      | -  |
| <b>Examples</b>    | nand.t d3, d1, 2, d2, 4  |
| <b>See also</b>    | AND.T, ANDN.T, OR.T, ORN.T, XNOR.T, XOR.T  |

#### 10.4.161 Not Equal ..... NE

**Table 10-182**  
**NE**

|                    |   |
|--------------------|---|
| <b>Syntax</b>      | ne Dc, Da, Db (RR)<br>ne Dc, Da, const9 (RC)  |
| <b>Description</b> | 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. |
| <b>Operation</b>   | $D[c] = (D[a] \neq D[b])$<br>$D[c] = (D[a] \neq \text{sign\_ext}(\text{const9}))$   |
| <b>Status</b>      | -   |
| <b>Examples</b>    | ne d3, d1, d2<br>ne d3, d1, 126   |
| <b>See also</b>    | EQ, GE, GE.U, LT, LT.U  |

#### 10.4.162 Not Equal Address ..... NE.A

**Table 10-183**  
**NE.A**

|                    |   |
|--------------------|---|
| <b>Syntax</b>      | ne.a Dc, Aa, Ab (RR)  |
| <b>Description</b> | 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. |
| <b>Operation</b>   | $D[c] = (A[a] \neq A[b])$   |
| <b>Status</b>      | -   |
| <b>Examples</b>    | ne.a d3, a4, a2   |
| <b>See also</b>    | EQ.A, EQZ.A, GE.A, LT.A, NEZ.A  |

2001-04-30 @ 15:16

**10.4.163 Not Equal Zero Address . . . . . NEZ.A**

**Table 10-184**  
**NEZ.A**

|                    |   |
|--------------------|---|
| <b>Syntax</b>      | nez.a Dc, Aa (RR)   |
| <b>Description</b> | If the contents of address register <i>Aa</i> are not equal to zero, set the least significant bit of <i>Dc</i> to 1 and clear the remaining bits to zero; otherwise, clear all bits in <i>Dc</i> . |
| <b>Operation</b>   | $D[c] = (A[a] \neq 0)$  |
| <b>Status</b>      | -   |
| <b>Examples</b>    | nez.a d3, a4  |
| <b>See also</b>    | EQ.A, EQZ.A, GE.A, LT.A, NE.A   |

**10.4.164 No Operation . . . . . NOP**

**Table 10-185**  
**NOP**

|                    |  |
|--------------------|--|
| <b>Syntax</b>      | nop (SYS)  |
|                    | nop (SR)   |
| <b>Description</b> | NOP is used to implement efficient low-power non-operational instructions. |
|                    | NOP is used to implement efficient low-power non-operational instructions. |
| <b>Operation</b>   | no operation   |
|                    | no operation   |
| <b>Status</b>      | -  |
| <b>Examples</b>    | nop  |
|                    | nop  |
| <b>See also</b>    | -  |

**10.4.165 Logical NOR . . . . . NOR**

**Table 10-186**  
**NOR**

|                    |  |
|--------------------|--|
| <b>Syntax</b>      | nor Dc, Da, Db (RR)  |
|                    | nor Dc, Da, const9 (RC)  |
|                    | nor Da, const_zero (SR)  |
| <b>Description</b> | Compute the bitwise logical NOR of the contents of data register <i>Da</i> and the contents of data register <i>Db/const9</i> and put the result in data register <i>Dc</i> . The <i>const9</i> value is zero-extended to 32 bits. |
|                    | Compute the bitwise NOT of the contents of register <i>Da</i> and put the result in data register <i>Da</i> . The operation is performed by a bitwise NOR of <i>Da</i> and <i>const_zero</i> value is zero-extended to 32 bits.    |

**Table 10-186**
**NOR**

|                  |  |
|------------------|--|
| <b>Operation</b> | $D[c] = !(D[a] \text{ OR } D[b])$                            |
|                  | $D[c] = !(D[a] \text{ OR } \text{zero\_ext}(\text{const9}))$ |
|                  | $D[a] = !(D[a] \text{ OR } \text{zero\_ext}(\text{zero}))$   |
| <b>Status</b>    | -  |
| <b>Examples</b>  | <code>nor d3, d1, d2</code>                                  |
|                  | <code>nor d3, d1, 126</code>                                 |
|                  | <code>nor d2, 0</code>                                       |
| <b>See also</b>  | AND, ANDN, NAND, NOT, OR, ORN, XNOR, XOR                     |

**10.4.166 Bit Logical NOR. . . . .NOR.T**
**Table 10-187**
**NOR.T**

|                    |  |
|--------------------|--|
| <b>Syntax</b>      | <code>nor.t Dc, Da, p1, Db, p2 (BIT)</code>  |
| <b>Description</b> | Compute the logical NOR of bit <i>p1</i> of data register <i>Da</i> and bit <i>p2</i> of data register <i>Db</i> . Put the result in the least-significant bit of data register <i>Dc</i> and clear the remaining bits of <i>Dc</i> to zero. |
| <b>Operation</b>   | $D[c] = !(D[a][p1] \text{ OR } D[b][p2])$  |
| <b>Status</b>      | -  |
| <b>Examples</b>    | <code>nor.t d3, d1, 5, d2, 3</code>  |
| <b>See also</b>    | AND.T, ANDN.T, NAND.T, OR.T, ORN.T, XNOR.T, XOR.T  |

**10.4.167 Bitwise Complement. . . . . NOT**
**Table 10-188**
**NOT**

|                    |   |
|--------------------|---|
| <b>Syntax</b>      | <code>not Da (Alias to nor16)</code>  |
| <b>Description</b> | Compute the bitwise complement of the contents of data register <i>Da</i> . |
| <b>Operation</b>   | $D[a] = !D[a]$  |
| <b>Status</b>      | -   |
| <b>Examples</b>    | <code>not d15</code>  |
| <b>See also</b>    | XNOR  |

**Note :** The 32-bit equivalent of the NOT instruction is a NOR with a constant of zero.

2001-04-30 @ 15:16

### 10.4.168 Logical OR ..... OR

**Table 10-189**  
**OR**

|                    |   |                     |
|--------------------|---|---------------------|
| <b>Syntax</b>      | or  | Dc, Da, Db (RR)     |
|                    | or  | Dc, Da, const9 (RC) |
|                    | or  | Da, Db (SRR)        |
|                    | or  | D15, const8 (SC)    |
| <b>Description</b> | Compute the bitwise logical OR of the contents of data register <i>Da</i> and the contents of data register <i>Db/const9</i> and put the result in data register <i>Dc</i> . The <i>const9</i> value is zero-extended to 32 bits. |                     |
|                    | Compute the logical OR of the contents of data register <i>Da/D15</i> and the contents of data register <i>Db/const8</i> and put the result in data register <i>Da/D15</i> . The <i>const8</i> value is zero-extended to 32 bits. |                     |
| <b>Operation</b>   | D[c] = D[a] OR D[b]<br>D[c] = D[a] OR zero_ext(const9)  |                     |
|                    | D[a] = D[a] or D[b]<br>D[15] = D[15] or zero_ext(const8)  |                     |
| <b>Status</b>      | -   |                     |
| <b>Examples</b>    | or  | d3, d1, d2          |
|                    | or  | d3, d1, 126         |
|                    | or  | d1, d2              |
|                    | or  | d15, 126            |
| <b>See also</b>    | AND, ANDN, NAND, NOR, NOT, ORN, XNOR, XOR   |                     |

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

**Table 10-190**
**OR.AND.T, OR.ANDN.T, OR.NOR.T & OR.OR.T**

|                    |  |
|--------------------|--|
| <b>Syntax</b>      | or.and.t Dc, Da, p1, Db, p2 (BIT)<br>or.andn.t Dc, Da, p1, Db, p2 (BIT)<br>or.nor.t Dc, Da, p1, Db, p2 (BIT)<br>or.or.t Dc, Da, p1, Db, p2 (BIT)   |
| <b>Description</b> | Compute the logical AND/ANDN/NOR/OR of the value of bit <i>p1</i> of data register <i>Da</i> and bit <i>p2</i> of <i>Db</i> . Then compute the bitwise logical OR of that result and bit 0 of <i>Dc</i> , and put the result back in bit 0 of <i>Dc</i> . All other bits in <i>Dc</i> are unchanged.                                 |
| <b>Operation</b>   | $D[c] = \{D[c][31:1], D[c][0] \text{ OR } (D[a][p1] \text{ AND } D[b][p2])\}$<br>$D[c] = \{D[c][31:1], D[c][0] \text{ OR } (D[a][p1] \text{ AND } !D[b][p2])\}$<br><br>$D[c] = \{D[c][31:1], D[c][0] \text{ OR } !(D[a][p1] \text{ OR } D[b][p2])\}$<br>$D[c] = \{D[c][31:1], D[c][0] \text{ OR } (D[a][p1] \text{ OR } D[b][p2])\}$ |
| <b>Status</b>      | -  |
| <b>Examples</b>    | or.and.t d3, d1, 3, d2, 5<br>or.andn.t d3, d1, 3, d2, 5<br>or.nor.t d3, d1, 3, d2, 5<br>or.or.t d3, d1, 3, d2, 5   |
| <b>See also</b>    | AND.AND.T, AND.ANDN.T, AND.NOR.T, AND.OR.T, SH.AND.T,<br>SH.ANDN.T, SH.NAND.T, SH.NOR.T, SH.OR.T, SH.ORN.T, SH.XNOR.T,<br>SH.XOR.T   |

**10.4.170 Equal Accumulating . . . . . OR.EQ**

**Table 10-191**
**OR.EQ**

|                    |   |
|--------------------|---|
| <b>Syntax</b>      | or.eq Dc, Da, Db (RR)<br>or.eq Dc, Da, const9 (RC)  |
| <b>Description</b> | Compute the logical OR of <i>Dc</i> [0] and the Boolean result of the EQ operation on the contents of data register <i>Da</i> and data register <i>Db/const9</i> . Put the result in <i>Dc</i> [0]. All other bits in <i>Dc</i> are unchanged. The <i>const9</i> value is sign-extended to 32 bits. |

2001-04-30 @ 15:16

**Table 10-191**  
**OR.EQ**

|                  |   |
|------------------|---|
| <b>Operation</b> | $D[c] = \{D[c][31:1], D[c][0] \text{ OR } (D[a] == D[b])\}$<br>$D[c] = \{D[c][31:1], D[c][0] \text{ OR } (D[a] == \text{sign\_ext}(\text{const9}))\}$ |
| <b>Status</b>    | -   |
| <b>Examples</b>  | <pre>or.eq    d3, d1, d2 or.eq    d3, d1, 126</pre>   |
| <b>See also</b>  | AND.EQ, XOR.EQ  |

### 10.4.171 Greater Than or Equal Accumulating ..... OR.GE Greater Than or Equal Accumulating Unsigned .....OR.GE.U

**Table 10-192**  
**OR.GE & OR.GE.U**

|                    |   |
|--------------------|---|
| <b>Syntax</b>      | <pre>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)</pre>  |
| <b>Description</b> | <p>Calculate the logical OR of <math>Dc[0]</math> and the Boolean result of the GE operation on the contents of data register <math>Da</math> and data register <math>Db/\text{const9}</math>. Put the result in <math>Dc[0]</math>. All other bits in <math>Dc</math> are unchanged. <math>Da</math> and <math>Db</math> are treated as 32-bit signed integers. The <math>\text{const9}</math> value is sign-extended to 32 bits.</p> <p>Calculate the logical OR of <math>Dc[0]</math> and the Boolean result of the GE.U operation on the contents of data register <math>Da</math> and data register <math>Db/\text{const9}</math>. Put the result in <math>Dc[0]</math>. All other bits in <math>Dc</math> are unchanged. <math>Da</math> and <math>Db</math> are treated as 32-bit unsigned integers. The <math>\text{const9}</math> value is zero-extended to 32 bits.</p> |
| <b>Operation</b>   | $D[c] = \{D[c][31:1], D[c][0] \text{ OR } (D[a] >= D[b])\}$ ; signed<br>$D[c] = \{D[c][31:1], D[c][0] \text{ OR } (D[a] >= \text{sign\_ext}(\text{const9}))\}$ ; signed<br><br>$D[c] = \{D[c][31:1], D[c][0] \text{ OR } (D[a] >= D[b])\}$ ; unsigned<br>$D[c] = \{D[c][31:1], D[c][0] \text{ OR } (D[a] >= \text{zero\_ext}(\text{const9}))\}$ ; unsigned  |
| <b>Status</b>      | -   |
| <b>Examples</b>    | <pre>or.ge    d3, d1, d2 or.ge    d3, d1, 126 or.ge.u   d3, d1, d2 or.ge.u   d3, d1, 126</pre>  |
| <b>See also</b>    | AND.GE, AND.GE.U, XOR.GE, XOR.GE.U  |



2001-04-30 @ 15:16

#### 10.4.172 Less Than Accumulating ..... OR.LT Less Than Accumulating Unsigned ..... OR.LT.U

**Table 10-193**
**OR.LT & OR.LT.U**

|                    |   |
|--------------------|---|
| <b>Syntax</b>      | or.lt     Dc, Da, Db (RR)<br>or.lt     Dc, Da, const9 (RC)<br>or.lt.u   Dc, Da, Db (RR)<br>or.lt.u   Dc, Da, const9 (RC)  |
| <b>Description</b> | <p>Calculate the logical OR of <math>Dc[0]</math> and the Boolean result of the LT operation on the contents of data register <math>Da</math> and data register <math>Db/const9</math>. Put the result in <math>Dc[0]</math>. All other bits in <math>Dc</math> are unchanged. <math>Da</math> and <math>Db</math> are treated as 32-bit signed integers. The <math>const9</math> value is sign-extended to 32 bits.</p> <p>Calculate the logical OR of <math>Dc[0]</math> and the Boolean result of the LT.U operation on the contents of data register <math>Da</math> and data register <math>Db/const9</math>. Put the result in <math>Dc[0]</math>. All other bits in <math>Dc</math> are unchanged. <math>Da</math> and <math>Db</math> are treated as 32-bit unsigned integers. The <math>const9</math> value is zero-extended to 32 bits.</p> |
| <b>Operation</b>   | $D[c] = \{D[c][31:1], D[c][0] \text{ OR } (D[a] < D[b])\}$ ; signed<br>$D[c] = \{D[c][31:1], D[c][0] \text{ OR } (D[a] < \text{sign\_ext}(const9))\}$ ; signed<br><br>$D[c] = \{D[c][31:1], D[c][0] \text{ OR } (D[a] < D[b])\}$ ; unsigned<br>$D[c] = \{D[c][31:1], D[c][0] \text{ OR } (D[a] < \text{zero\_ext}(const9))\}$ ; unsigned  |
| <b>Status</b>      | -   |
| <b>Examples</b>    | or.lt     d3, d1, d2<br>or.lt     d3, d1, 126<br>or.lt.u   d3, d1, d2<br>or.lt.u   d3, d1, 126  |
| <b>See also</b>    | AND.LT, AND.LT.U, XOR.LT, XOR.LT.U  |

#### 10.4.173 Not Equal Accumulating. .... OR.NE

**Table 10-194**
**OR.NE**

|                    |  |
|--------------------|--|
| <b>Syntax</b>      | or.ne     Dc, Da, Db (RR)<br>or.ne     Dc, Da, const9 (RC)   |
| <b>Description</b> | 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. |

2001-04-30 @ 15:16

**Table 10-194**  
**OR.NE**

|                  |   |
|------------------|---|
| <b>Operation</b> | $D[c] = \{D[c][31:1], D[c][0] \text{ OR } (D[a] \neq D[b])\}$<br>$D[c] = \{D[c][31:1], D[c][0] \text{ OR } (D[a] \neq \text{sign\_ext}(\text{const9}))\}$ |
| <b>Status</b>    | -   |
| <b>Examples</b>  | <code>or.ne d3, d1, d2</code><br><code>or.ne d3, d1, 126</code>   |
| <b>See also</b>  | AND.NE, XOR.NE  |

#### 10.4.174 Bit Logical OR ..... OR.T

**Table 10-195**  
**OR.T**

|                    |   |
|--------------------|---|
| <b>Syntax</b>      | <code>or.t Dc, Da, p1, Db, p2 (BIT)</code>  |
| <b>Description</b> | Compute the logical OR of bit <i>p1</i> of data register <i>Da</i> and bit <i>p2</i> of data register <i>Db</i> . Put the result in the least-significant bit of data register <i>Dc</i> and clear the remaining bits of <i>Dc</i> to zero. |
| <b>Operation</b>   | $D[c] = D[a][p1] \text{ OR } D[b][p2]$  |
| <b>Status</b>      | -   |
| <b>Examples</b>    | <code>or.t d3, d1, 7, d2, 9</code>  |
| <b>See also</b>    | AND.T, ANDN.T, NAND.T, NOR.T, ORN.T, XNOR.T, XOR.T  |

#### 10.4.175 Logical OR-Not ..... ORN

**Table 10-196**  
**ORN**

|                    |   |
|--------------------|---|
| <b>Syntax</b>      | <code>orn Dc, Da, Db (RR)</code><br><code>orn Dc, Da, const9 (RC)</code>  |
| <b>Description</b> | Compute the bitwise logical OR of the contents of data register <i>Da</i> and the one's complement of the contents of data register <i>Db/const9</i> and put the result in data register <i>Dc</i> . The <i>const9</i> value is zero-extended to 32 bits. |
| <b>Operation</b>   | $D[c] = D[a] \text{ OR } !D[b]$<br>$D[c] = D[a] \text{ OR } !\text{zero\_ext}(\text{const9})$   |
| <b>Status</b>      | -   |
| <b>Examples</b>    | <code>orn d3, d1, d2</code><br><code>orn d3, d1, 126</code>   |
| <b>See also</b>    | AND, ANDN, NAND, NOR, NOT, OR, XNOR, XOR  |

2001-04-30 @ 15:16

#### 10.4.176 Bit Logical OR-Not ..... ORN.T

**Table 10-197**  
**ORN.T**

|                    |  |
|--------------------|--|
| <b>Syntax</b>      | orn.t    Dc, Da, p1, Db, p2 (BIT)  |
| <b>Description</b> | Compute the logical OR of bit p1 of data register <i>Da</i> and the inverse of bit p2 of data register <i>Db</i> . Put the result in the least-significant bit of data register <i>Dc</i> and clear the remaining bits of <i>Dc</i> to zero. |
| <b>Operation</b>   | $D[c] = D[a][p1] \text{ OR } !D[b][p2]$  |
| <b>Status</b>      | -  |
| <b>Examples</b>    | orn.t    d3, d1, 2, d2, 5  |
| <b>See also</b>    | AND.T, ANDN.T, NAND.T, NOR.T, OR.T, XNOR.T, XOR.T  |

#### 10.4.177 Return from Call .....RET

**Table 10-198**  
**RET**

|                    |  |
|--------------------|--|
| <b>Syntax</b>      | ret            (SYS)   |
|                    | ret            (SR)  |
| <b>Description</b> | 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. |
|                    | 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. |
| <b>Operation</b>   | if (call_depth_counter == 0) then trap(CDU);<br>PC = A[11];<br>Restore upper context;  |
|                    | if (call_depth_counter == 0) then trap(CDU);<br>PC = A[11];<br>Restore upper context;  |
| <b>Status</b>      | -  |
| <b>Examples</b>    | ret  |
|                    | ret  |
| <b>See also</b>    | CALL, CALLA, CALLI, RFE, SYSCALL   |

2001-04-30 @ 15:16

**10.4.178 Return from Exception . . . . .RFE**

**Table 10-199**  
**RFE**

|                    |  |
|--------------------|--|
| <b>Syntax</b>      | rfe (SYS)  |
|                    | rfe (SR)   |
| <b>Description</b> | 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 PCXI 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.  |
|                    | 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 PCXI register to cause another task to be dispatched. |
| <b>Operation</b>   | if (call_depth_counter !=0) then trap (NEST);<br>PC = A[11];<br>ICR.CCPN = PCXI.PCPN;<br>ICR.IE = PCXI.PIE;<br>Restore upper context;  |
|                    | if (call_depth_counter !=0) then trap (NEST);<br>PC = A[11];<br>ICR.CCPN = PCXI.PCPN;<br>ICR.IE = PCXI.PIE;<br>Restore upper context;  |
| <b>Status</b>      | -  |
| <b>Examples</b>    | rfe  |
|                    | rfe  |
| <b>See also</b>    | CALL, CALLA, CALLI, RET, SYSCALL   |

2001-04-30 @ 15:16

#### 10.4.179 Restore Lower Context. .... RSLCX

**Table 10-200**  
**RSLCX**

|                    |  |
|--------------------|--|
| <b>Syntax</b>      | rslcx            (SYS)   |
| <b>Description</b> | 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. |
| <b>Operation</b>   | Restore lower context  |
| <b>Status</b>      | -  |
| <b>Examples</b>    | RSLCX  |
| <b>See also</b>    | LDLCX, LDUCX, STLCX, STUCX, SVLCX  |

#### 10.4.180 Reset Overflow Bits ..... RSTV

**Table 10-201**  
**RSTV**

|                    |                                     |
|--------------------|-------------------------------------|
| <b>Syntax</b>      | rstv            (SYS)               |
| <b>Description</b> | Reset overflow status flags in PSW. |
| <b>Operation</b>   | PSW.{V, SV, AV, SAV} = {0, 0, 0, 0} |
| <b>Status</b>      | V, SV, AV, SAV                      |
| <b>Examples</b>    | rstv                                |
| <b>See also</b>    | BISR, MTCR, ENABLE, DISABLE         |

#### 10.4.181 Reverse-Subtract ..... RSUB

**Table 10-202**  
**RSUB**

|                    |  |
|--------------------|--|
| <b>Syntax</b>      | rsub Dc, Da, const9 (RC)   |
|                    | rsub Da, (SR)  |
| <b>Description</b> | Subtract the contents of data register <i>Da</i> from the value <i>const9</i> and put the result in data register <i>Dc</i> . The operands are treated as 32-bit integers. The value <i>const9</i> is sign-extended to 32 bits before the subtraction is performed |
|                    | Subtract the contents of data register <i>Da</i> from zero and put the result in data register <i>Da</i> . The operand is treated as a 32-bit integer.   |

2001-04-30 @ 15:16

**Table 10-202**  
**RSUB**

|                  |   |
|------------------|---|
| <b>Operation</b> | $D[c] = \text{sign\_ext}(\text{const9}) - D[a]$ |
|                  | $D[a] = 0 - D[a]$                               |
| <b>Status</b>    | V, SV, AV, SAV                                  |
| <b>Examples</b>  | <pre>rsub    d3, d1, 126 rsub    d1</pre>       |
| <b>See also</b>  | RSUBS, RSUBS.U                                  |

**10.4.182 Reverse-Subtract with Saturation . . . . . RSUBS**  
**Reverse-Subtract Unsigned with Saturation . . . . . RSUBS.U**

**Table 10-203**  
**RSUBS & RSUBS.U**

|                    |   |
|--------------------|---|
| <b>Syntax</b>      | <pre>rsubs   Dc, Da, const9 (RC) rsubs.u Dc, Da, const9 (RC)</pre>  |
| <b>Description</b> | Subtract the contents of data register <i>Da</i> from the value <i>const9</i> and put the result in data register <i>Dc</i> . The operands are treated as signed/unsigned, 32-bit integers, with saturation on signed/unsigned overflow. The value <i>const9</i> is sign-extended/zero-extended to 32 bits before the operation is performed. |
| <b>Operation</b>   | $D[c] = \text{sign\_ext}(\text{const9}) - D[a]; \text{signed}; \text{ssov}$   |
|                    | $D[c] = \text{zero\_ext}(\text{const9}) - D[a]; \text{unsigned}; \text{suov}$   |
| <b>Status</b>      | V, SV, AV, SAV  |
| <b>Examples</b>    | <pre>rsubs    d3, d1, 126 rsubs.u  d3, d1, 126</pre>  |
| <b>See also</b>    | RSUB  |

**10.4.183 Saturate Byte. . . . . SAT.B**

**Table 10-204**  
**SAT.B**

|                    |   |
|--------------------|---|
| <b>Syntax</b>      | sat.b    Dc, Da (RR)  |
|                    | sat.b    Da (SR)  |
| <b>Description</b> | If the signed 32-bit value in <i>Da</i> is less than $-128$ , then store the value $-128$ in <i>Dc</i> . If <i>Da</i> is greater than $127$ , then store the value $127$ in <i>Dc</i> . Otherwise, copy <i>Da</i> to <i>Dc</i> .              |
|                    | If the signed 32-bit value in <i>Da</i> is less than $-128$ , then store the value $-128$ in <i>Da</i> . If <i>Da</i> is greater than $127$ , then store the value $127$ in <i>Da</i> . Otherwise, leave the contents of <i>Da</i> unchanged. |

**Table 10-204**
**SAT.B**

|                  |   |
|------------------|---|
| <b>Operation</b> | $D[c] = (D[a] < -128) ? -128 : ((D[a] > 127) ? 127 : D[a]);$ signed |
|                  | $D[a] = (D[a] < -128) ? -128 : ((D[a] > 127) ? 127 : D[a]);$ signed |
| <b>Status</b>    | -   |
| <b>Examples</b>  | sat.b     d3, d1  |
|                  | sat.b     d1  |
| <b>See also</b>  | SAT.BU, SAT.H, SAT.HU   |

**10.4.184 Saturate Byte Unsigned ..... SAT.BU**
**Table 10-205**
**SAT.BU**

|                    |  |
|--------------------|--|
| <b>Syntax</b>      | sat.bu   Dc, Da (RR)   |
|                    | sat.bu   Da (SR)   |
| <b>Description</b> | If the unsigned 32-bit value in <i>Da</i> is greater than 255, then store the value 255 in <i>Dc</i> . Otherwise, copy <i>Da</i> to <i>Dc</i> .              |
|                    | If the unsigned 32-bit value in <i>Da</i> is greater than 255, then store the value 255 in <i>Da</i> . Otherwise, leave the contents of <i>Da</i> unchanged. |
| <b>Operation</b>   | $D[c] = (D[a] > 255) ? 255 : D[a];$ unsigned   |
|                    | $D[a] = (D[a] > 255) ? 255 : D[a];$ unsigned   |
| <b>Status</b>      | -  |
| <b>Examples</b>    | sat.bu     d3, d1  |
|                    | sat.bu     d1  |
| <b>See also</b>    | SAT.B, SAT.H, SAT.HU   |

**10.4.185 Saturate Halfword ..... SAT.H**
**Table 10-206**
**SAT.H**

|                    |   |
|--------------------|---|
| <b>Syntax</b>      | sat.h     Dc, Da (RR)   |
|                    | sat.h     Da (SR)   |
| <b>Description</b> | If the signed 32-bit value in <i>Da</i> is less than -32,768, then store the value -32,768 in <i>Dc</i> . If <i>Da</i> is greater than 32,767, then store the value 32,767 in <i>Dc</i> . Otherwise, copy <i>Da</i> to <i>Dc</i> .              |
|                    | If the signed 32-bit value in <i>Da</i> is less than -32,768, then store the value -32,768 in <i>Da</i> . If <i>Da</i> is greater than 32,767, then store the value 32,767 in <i>Da</i> . Otherwise, leave the contents of <i>Da</i> unchanged. |

2001-04-30 @ 15:16

**Table 10-206**  
**SAT.H**

|                  |   |
|------------------|---|
| <b>Operation</b> | $D[c] = (D[a] < -2^{15}) ? -2^{15} : ((D[a] > 2^{15}-1) ? 2^{15}-1 : D[a]);$ signed |
|                  | $D[a] = (D[a] < -2^{15}) ? -2^{15} : ((D[a] > 2^{15}-1) ? 2^{15}-1 : D[a]);$ signed |
| <b>Status</b>    | -   |
| <b>Examples</b>  | sat.h    d3, d1   |
|                  | sat.h    d1   |
| <b>See also</b>  | SAT.B, SAT.BU, SAT.HU   |

#### 10.4.186 Saturate Halfword Unsigned ..... SAT.HU

**Table 10-207**  
**SAT.HU**

|                    |  |
|--------------------|--|
| <b>Syntax</b>      | sat.hu   Dc, Da (RR)   |
|                    | sat.hu   Da (SR)   |
| <b>Description</b> | If the unsigned 32-bit value in <i>Da</i> is greater than 65,535, then store the value 65,535 in <i>Dc</i> ; otherwise, copy <i>Da</i> to <i>Dc</i> .              |
|                    | If the unsigned 32-bit value in <i>Da</i> is greater than 65,535, then store the value 65,535 in <i>Da</i> ; otherwise, leave the contents of <i>Da</i> unchanged. |
| <b>Operation</b>   | $D[c] = (D[a] > 2^{16}-1) ? 2^{16}-1 : D[a];$ unsigned   |
|                    | $D[a] = (D[a] > 2^{16}-1) ? 2^{16}-1 : D[a];$ unsigned   |
| <b>Status</b>      | -  |
| <b>Examples</b>    | sat.hu    d3, d1   |
|                    | sat.hu    d1   |
| <b>See also</b>    | SAT.B, SAT.BU, SAT.H   |

#### 10.4.187 Select .....SEL

**Table 10-208**  
**SEL**

|                    |   |
|--------------------|---|
| <b>Syntax</b>      | sel       Dc, Dd, Da, Db (RRR)  |
|                    | sel       Dc, Dd, Da, const9 (RCR)  |
| <b>Description</b> | If the contents of data register <i>Dd</i> are non-zero, copy the contents of data register <i>Da</i> to data register <i>Dc</i> ; otherwise, copy the contents of <i>Db/const9</i> to <i>Dc</i> . The value <i>const9</i> is sign-extended to 32 bits. |



2001-04-30 @ 15:16

Table 10-208

## SEL

|                  |   |
|------------------|---|
| <b>Operation</b> | $D[c] = ((D[d] \neq 0) ? D[a] : D[b])$<br>$D[c] = ((D[d] \neq 0) ? D[a] : \text{sign\_ext}(\text{const9}))$ |
| <b>Status</b>    | -   |
| <b>Examples</b>  | <code>sel d3, d4, d1, d2</code><br><code>sel d3, d4, d1, 126</code>   |
| <b>See also</b>  | CADD, CADDN, CMOV, CMOVN, CSUB, CSUBN, SELN   |

## 10.4.188 Select-Not ..... SELN

Table 10-209

## SELN

|                    |   |
|--------------------|---|
| <b>Syntax</b>      | <code>seln Dc, Dd, Da, Db (RRR)</code><br><code>seln Dc, Dd, Da, const9 (RCR)</code>  |
| <b>Description</b> | If the contents of data register <i>Dd</i> are zero, copy the contents of data register <i>Da</i> to data register <i>Dc</i> ; otherwise, copy the contents of <i>Db/const9</i> to <i>Dc</i> . The value <i>const9</i> is sign-extended to 32 bits. |
| <b>Operation</b>   | $D[c] = ((D[d] == 0) ? D[a] : D[b])$<br>$D[c] = ((D[d] == 0) ? D[a] : \text{sign\_ext}(\text{const9}))$   |
| <b>Status</b>      | -   |
| <b>Examples</b>    | <code>seln d3, d4, d1, d2</code><br><code>seln d3, d4, d1, 126</code>   |
| <b>See also</b>    | CADD, CADDN, CMOV, CMOVN, CSUB, CSUBN, SEL  |

2001-04-30 @ 15:16

**10.4.189 Shift .....SH.H**

**Table 10-210**  
**SH**

|                    |   |
|--------------------|---|
| <b>Syntax</b>      | <div>sh      Dc, Da, Db (RR)</div> <div>sh      Dc, Da, const9 (RC)</div> <div>sh      Da, const4 (SRC)</div>   |
| <b>Description</b> | <p>If the shift count specified through the contents of <i>Db/const9</i> is greater than or equal to zero, then left-shift the value in <i>Da</i> by the amount specified by shift count. Otherwise, right-shift the value in <i>Da</i> by the absolute value of the shift count. Put the result in <i>Dc</i>. 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 <i>Db[5:0]</i> or <i>const9[5:0]</i>. 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).</p> <p>If the shift count specified through the value <i>const4</i> is greater than or equal to zero, then left-shift the value in <i>Da</i> by the amount specified by the shift count. Otherwise, right-shift the value in <i>Da</i> by the absolute value of the shift count. Put the result in <i>Da</i>. 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 <i>const4[3:0]</i>. 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.</p> |
| <b>Operation</b>   | <div>shift_count = D[b][5:0] or const9[5:0];<br/>if (shift_count &gt;= 0) then D[c] = D[a] &lt;&lt; shift_count; zero-fill<br/>else D[c] = D[a] &gt;&gt; (–shift_count); zero-fill</div> <div>shift_count = sign_ext(const4[3:0]);<br/>if (shift_count &gt;= 0) then D[a] = D[a] &lt;&lt; shift_count; zero-fill<br/>else D[a] = D[a] &gt;&gt; (–shift_count); zero-fill</div>  |
| <b>Status</b>      | <div>-</div> <div>-</div>   |
| <b>Examples</b>    | <div>sh    d3, d1, d2</div> <div>sh    d3, d1, 26</div> <div>sh    d1, 6</div>  |
| <b>See also</b>    | SH.H, SHA, SHA.H, SHAS  |

**10.4.190 Shift Packed Halfwords .....SH**
**Table 10-211**
**SH.H**

|                    |   |
|--------------------|---|
| <b>Syntax</b>      | sh.h    Dc, Da, Db (RR)<br>sh.h    Dc, Da, const9 (RC)  |
| <b>Description</b> | <p>If the shift count specified through the contents of <i>Db/const9</i> is greater than or equal to zero, then left-shift each halfword in <i>Da</i> by the amount specified by shift count. Otherwise, right-shift each halfword in <i>Da</i> by the absolute value of the shift count. Put the result in <i>Dc</i>. In both cases, the vacated bits are filled with zeroes and bits shifted out are discarded. Note that for these shifts, each halfword is treated individually, and bits shifted out of a halfword are not shifted in to the next halfword.</p> <p>The shift count is a signed number, derived from the sign-extension of <i>Db[4:0]</i> or <i>const9[4:0]</i>. The range for the shift count therefore is <math>-16</math> to <math>+15</math>. The result for a shift count of <math>-16</math> for halfwords is zero.</p> |
| <b>Operation</b>   | sh.h : if (shift_count $\geq$ 0) then $D[c] = D[a][(n+15):n] \ll \text{shift\_count}$ ; zero-fill<br>else $D[c] = D[a][(n+15):n] \gg (-\text{shift\_count})$ ; zero-fill<br>shift_count = sign_ext( <i>Db[4:0]</i> ) or sign_ext( <i>const9[4:0]</i> ); $n = 0, 16$   |
| <b>Status</b>      | TBD   |
| <b>Examples</b>    | sh.h    d3, d1, d2<br>sh.h    d3, d1, 12  |
| <b>See also</b>    | SH, SHA, SHA.H, SHAS  |

**10.4.191 Shift Equal .....SH.EQ**
**Table 10-212**
**SH.EQ**

|                    |  |
|--------------------|--|
| <b>Syntax</b>      | sh.eq    Dc, Da, Db (RR)<br>sh.eq    Dc, Da, const9 (RC)   |
| <b>Description</b> | <p>Left shift <i>Dc</i> by 1. If the contents of data register <i>Da</i> are equal to the contents of data register <i>Db/const9</i>, set the least-significant bit of <i>Dc</i> to 1; otherwise, set the least-significant bit of <i>Dc</i> to 0.</p> <p>The value <i>const9</i> is sign-extended to 32 bits.</p> |
| <b>Operation</b>   | $D[c] = \{D[c][30:0], (D[a] == D[b])\}$<br>$D[c] = \{D[c][30:0], (D[a] == \text{sign\_ext}(\text{const9}))\}$  |
| <b>Status</b>      | -  |
| <b>Examples</b>    | sh.eq    d3, d1, d2<br>sh.eq    d3, d1, 126  |
| <b>See also</b>    | SH.GE, SH.GE.U, SH.LT, SH.LT.U, SH.NE  |

2001-04-30 @ 15:16

### 10.4.192 Shift Greater Than or Equal . . . . .SH.GE

### Shift Greater Than or Equal Unsigned . . . . .SH.GE.U

**Table 10-213**  
**SH.GE & SH.GE.U**

|                    |   |
|--------------------|---|
| <b>Syntax</b>      | sh.ge Dc, Da, Db (RR)<br>sh.ge Dc, Da, const9 (RC)<br>sh.ge.u Dc, Da, Db (RR)<br>sh.ge.u Dc, Da, const9 (RC)  |
| <b>Description</b> | <p>Left shift Dc by 1. If the contents of data register <i>Da</i> are greater than or equal to the contents of data register <i>Db/const9</i>, set the least-significant bit of <i>Dc</i> to 1; otherwise, set the least-significant bit of <i>Dc</i> to 0.</p> <p><i>Da</i> and <i>Db</i> are treated as signed integers. The value <i>const9</i> is sign-extended to 32 bits.</p> <p>Left shift Dc by 1. If the contents of data register <i>Da</i> are greater than or equal to the contents of data register <i>Db/const9</i>, set the least-significant bit of <i>Dc</i> to 1; otherwise, set the least-significant bit of <i>Dc</i> to 0.</p> <p><i>Da</i> and <i>Db</i> are treated as unsigned integers. The value <i>const9</i> is zero-extended to 32 bits.</p> |
| <b>Operation</b>   | <p>D[c] = {D[c][30:0], (D[a] &gt;= D[b])}; signed<br/>           D[c] = {D[c][30:0], (D[a] &gt;= sign_ext(const9))}; signed</p> <p>D[c] = (D[c][30:0], (D[a] &gt;= D[b])); unsigned<br/>           D[c] = (D[c][30:0], (D[a] &gt;= zero_ext(const9))); unsigned</p>   |
| <b>Status</b>      | -   |
| <b>Examples</b>    | sh.ged3, d1, d2<br>sh.ged3, d1, 126<br>sh.ge.ud3, d1, d2<br>sh.ge.ud3, d1, 126  |
| <b>See also</b>    | SH.EQ, SH.LT, SH.LT.U, SH.NE  |

2001-04-30 @ 15:16

### 10.4.193 Shift Less Than ..... SH.LT

### Shift Less Than Unsigned ..... SH.LT.U

Table 10-214

SH.LT &amp; SH.LT.U

|                    |   |
|--------------------|---|
| <b>Syntax</b>      | sh.lt Dc, Da, Db (RR)<br>sh.lt Dc, Da, const9 (RC)<br>sh.lt.u Dc, Da, Db (RR)<br>sh.lt.u Dc, Da, const9 (RC)  |
| <b>Description</b> | <p>Left shift Dc by 1. If the contents of data register <i>Da</i> are less the contents of data register <i>Db/const9</i>, set the least-significant bit of <i>Dc</i> to 1; otherwise, set the least-significant bit of <i>Dc</i> to 0.</p> <p><i>Da</i> and <i>Db/const9</i> are treated as signed integers. The value <i>const9</i> is sign-extended to 32 bits.</p> <p>Left shift Dc by 1. If the contents of data register <i>Da</i> are less the contents of data register <i>Db/const9</i>, set the least-significant bit of <i>Dc</i> to 1; otherwise, set the least-significant bit of <i>Dc</i> to 0.</p> <p><i>Da</i> and <i>Db/const9</i> are treated as unsigned integers. The value <i>const9</i> is zero-extended to 32 bits.</p> |
| <b>Operation</b>   | D[c] = {D[c][30:0], (D[a] < D[b])}; signed<br>D[c] = {D[c][30:0], (D[a] < sign_ext(const9))}; signed<br><br>D[c] = {D[c][30:0], (D[a] < D[b])}; unsigned<br>D[c] = {(D[c][30:0], (D[a] < zero_ext(const9))}; unsigned   |
| <b>Status</b>      | -   |
| <b>Examples</b>    | sh.lt d3, d1, d2<br>sh.lt d3, d1, 126<br>sh.lt.u d3, d1, d2<br>sh.lt.u d3, d1, 126  |
| <b>See also</b>    | SH.EQ, SH.GE, SH.GE.U, SH.NE  |

2001-04-30 @ 15:16

### 10.4.194 Shift Not Equal .....SH.NE

**Table 10-215**  
**SH.NE**

|                    |  |
|--------------------|--|
| <b>Syntax</b>      | sh.ne Dc, Da, Db (RR)<br>sh.ne Dc, Da, const9 (RC)   |
| <b>Description</b> | Left shift Dc by 1. 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 value const9 is sign-extended to 32 bits. |
| <b>Operation</b>   | D[c] = {D[c][30:0], (D[a] != D[b])}<br>D[c] = {D[c][30:0], (D[a] != sign_ext(const9))}   |
| <b>Status</b>      | -  |
| <b>Examples</b>    | sh.ne d3, d1, d2<br>sh.ne d3, d1, 126  |
| <b>See also</b>    | SH.EQ, SH.GE, SH.GE.U, SH.LT, SH.LT.U  |

### 10.4.195 Accumulating Shift-AND .....SH.AND.T

### Accumulating Shift-AND-Not .....SH.ANDN.T

### Accumulating Shift-N .....SH.NAND.T

### Accumulating Shift-NOR .....SH.NOR.T

### Accumulating Shift-OR .....SH.OR.T

### Accumulating Shift-OR-Not .....SH.ORN.T

### Accumulating Shift-XNOR .....SH.XNOR.T

### Accumulating Shift-XOR .....SH.XOR.T

**Table 10-216**

**SH.AND.T, SH.ANDN.T, SH.NAND.T, SH.NOR.T, SH.OR.T, SH.ORN.T, SH.XNOR.T, SH.XOR.T**

|                    |  |
|--------------------|--|
| <b>Syntax</b>      | sh.and.t Dc, Da, p1, Db, p2 (BIT)<br>sh.andn.t Dc, Da, p1, Db, p2 (BIT)<br>sh.nand.t Dc, Da, p1, Db, p2 (BIT)<br>sh.nor.t Dc, Da, p1, Db, p2 (BIT)<br>sh.or.t Dc, Da, p1, Db, p2 (BIT)<br>sh.orn.t Dc, Da, p1, Db, p2 (BIT)<br>sh.xnor.t Dc, Da, p1, Db, p2 (BIT)<br>sh.xor.t Dc, Da, p1, Db, p2 (BIT) |
| <b>Description</b> | 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].  |

2001-04-30 @ 15:16

**Table 10-216**
**SH.AND.T, SH.ANDN.T, SH.NAND.T, SH.NOR.T, SH.OR.T, SH.ORN.T, SH.XNOR.T, SH.XOR.T  
(cont'd) (cont'd)**

|                  |   |
|------------------|---|
| <b>Operation</b> | sh.and.t : $D[c] = \{D[c][30:0], (D[a][p1] \text{ AND } D[b][p2])\}$<br>sh.andn.t : $D[c] = \{D[c][30:0], (D[a][p1] \text{ AND } \neg(D[b][p2]))\}$<br>sh.nand.t : $D[c] = \{D[c][30:0], \neg(D[a][p1] \text{ AND } D[b][p2])\}$<br>sh.nor.t : $D[c] = \{D[c][30:0], \neg(D[a][p1] \text{ OR } D[b][p2])\}$<br>sh.or.t : $D[c] = \{D[c][30:0], (D[a][p1] \text{ OR } D[b][p2])\}$<br>sh.orn.t : $D[c] = \{D[c][30:0], (D[a][p1] \text{ OR } \neg(D[b][p2]))\}$<br>sh.xnor.t : $D[c] = \{D[c][30:0], \neg(D[a][p1] \text{ XOR } D[b][p2])\}$<br>sh.xor.t : $D[c] = \{D[c][30:0], (D[a][p1] \text{ XOR } D[b][p2])\}$ |
| <b>Status</b>    | -   |
| <b>Examples</b>  | sh.and.t     d3, d1, 4, d2, 7<br>sh.andn.t   d3, d1, 4, d2, 7<br>sh.nand.t   d3, d1, 4, d2, 7<br>sh.nor.t     d3, d1, 4, d2, 7<br>sh.or.t      d3, d1, 4, d2, 7<br>sh.orn.t     d3, d1, 4, d2, 7<br>sh.xnor.t    d3, d1, 4, d2, 7<br>sh.xor.t     d3, d1, 4, d2, 7  |
| <b>See also</b>  | AND.AND.T, AND.ANDN.T, AND.NOR.T, AND.OR.T, OR.AND.T, OR.ANDN.T, OR.NOR.T, OR.OR.T  |

2001-04-30 @ 15:16

**10.4.196 Arithmetic Shift ..... SHA**

**Table 10-217**  
**SHA**

|                    |   |
|--------------------|---|
| <b>Syntax</b>      | <div>sha     Dc, Da, Db (RR)</div> <div>sha     Dc, Da, const9 (RC)</div> <div>sha     Da, const4 (SRC)</div>   |
| <b>Description</b> | <p>If shift count specified through contents of <i>Db/const9</i> is greater than or equal to zero, then left-shift the value in <i>Da</i> 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 <i>Da</i> 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 <i>Dc</i>.</p> <p>Shift count is a 6-bit signed number, derived from <i>Db[5:0]</i> or <i>const9[5:0]</i>. The range for shift count therefore is <math>-32</math> to <math>+31</math>, allowing to shift left up to 31 bit positions &amp; 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. On all 1-bit or greater shifts (left or right), <i>PSW.C</i> is set to the logical-OR of shifted out bits. On zero-bit shifts, <i>C</i> is cleared.</p> <p>If shift count specified through the value <i>const4</i> is greater than or equal to zero, then left-shift the value in <i>Da</i> 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 <i>Da</i> 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 <i>Da</i>.</p> <p>The shift count is a 6-bit signed number, derived from the sign-extension of <i>const4[3:0]</i>. The resulting range for the shift count therefore is <math>-8</math> to <math>+7</math>, 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), <i>PSW.C</i> is set to the logical-OR of the shifted out bits. On zero-bit shifts, <i>C</i> is cleared.</p> |
| <b>Operation</b>   | <div> shift_count = Db[5:0] or const9[5:0];<br/> if (shift_count &gt;= 0) then D[c] = D[a] &lt;&lt; shift_count; zero-fill<br/> else D[c] = D[a] &gt;&gt; (-shift_count); sign-fill<br/> PSW.C = OR(bits shifted out) </div> <div> shift_count = sign_ext(const4[3:0]);<br/> if (shift_count &gt;= 0) then D[a] = D[a] &lt;&lt; shift_count; zero-fill<br/> else D[a] = D[a] &gt;&gt; (-shift_count); sign-fill<br/> PSW.C = OR(bits shifted out) </div>  |



2001-04-30 @ 15:16

**Table 10-217  
SHA (cont'd)**

|                 |                   |
|-----------------|-------------------|
| <b>Status</b>   | V, SV, AV, SAV, C |
|                 | V, SV, AV, SAV, C |
| <b>Examples</b> | sha d3, d1, d2    |
|                 | sha d3, d1, 26    |
|                 | sha d1, 6         |
| <b>See also</b> | SH, SH.H, SHAS    |

#### 10.4.197 Arithmetic Shift Packed Halfwords .....SHA.H

**Table 10-218  
SHA.H**

|                    |   |
|--------------------|---|
| <b>Syntax</b>      | sha.h Dc, Da, Db (RR)<br>sha.h Dc, Da, const9 (RC)  |
| <b>Description</b> | <p>If the shift count specified through the contents of <i>Db/const9</i> is greater than or equal to zero, then left-shift each halfword in <i>Da</i> 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 halfword in <i>Da</i> by the absolute value of the shift count. The vacated bits are filled with the sign-bit (MSB) of the respective halfword, and bits shifted out are discarded. Put the result in <i>Dc</i>. Note that for these shifts, each halfword is treated individually, and bits shifted out of a halfword are not shifted in to the next halfword.</p> <p>The shift count is a signed number, derived from the sign-extension of Db[4:0] or const9[4:0]. The range for the shift count therefore is –16 to +15. The result for each halfword for a shift count of –16 is either all zeros or all ones, depending on the sign-bit of the respective halfword.</p> |
| <b>Operation</b>   | sha.h : if (shift_count >= 0) then D[c] = D[a][(n+15):n] << shift_count; zero-fill<br>else D[c] = D[a][(n+15):n] >> (–shift_count); sign-fill<br>shift_count = sign_ext(Db[4:0]) or sign_ext(const9[4:0]); n = 0, 16  |
| <b>Status</b>      | V, SV, AV, SAV  |
| <b>Examples</b>    | sha.h d3, d1, d2  |
|                    | sha.h d3, d1, 12  |
| <b>See also</b>    | SH, SHA, SHAS, SH.H   |

2001-04-30 @ 15:16

**10.4.198 Arithmetic Shift with Saturation . . . . . SHAS**

**Table 10-219**  
**SHAS**

|                    |  |
|--------------------|--|
| <b>Syntax</b>      | shas Dc, Da, Db (RR)<br>shas Dc, Da, const9 (RC)   |
| <b>Description</b> | <p>If the shift count specified through the contents of <i>Db/const9</i> is greater than or equal to zero, then left-shift the value in <i>Da</i> 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 <i>Da</i> 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 <i>Dc</i>.</p> <p>The shift count is a 6-bit signed number, derived from <i>Db[5:0]</i> or <i>const9[5:0]</i>. The range for the shift count therefore is <math>-32</math> to <math>+31</math>, 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).</p> |
| <b>Operation</b>   | $\text{shift\_count} = D[b][5:0] \text{ or } \text{const9}[5:0];$<br>if ( $\text{shift\_count} \geq 0$ ) then $D[c] = D[a] \ll \text{shift\_count}$ ; zero-fill; ssov<br>else $D[c] = D[a] \gg (-\text{shift\_count})$ ; sign-fill   |
| <b>Status</b>      | V, SV, AV, SAV   |
| <b>Examples</b>    | shas d3, d1, d2<br>shas d3, d1, 26   |
| <b>See also</b>    | SH, SH.H, SHA, SHA.H   |

**10.4.199 Store Word from Address Register . . . . . ST.A**

**Table 10-220**  
**ST.A**

|                    |  |
|--------------------|--|
| <b>Syntax</b>      | st.a <mode>, Aa  |
| <b>Description</b> | Store the value in address register <i>Aa</i> to the memory location specified by the addressing mode. |
| <b>Operation</b>   | $M(EA, \text{word}) = A[a]$ (See <a href="#">Table 10-221</a> )  |
| <b>Status</b>      | -  |
| <b>Examples</b>    | -  |
| <b>See also</b>    | ST.B, ST.D, ST.DA, ST.H, ST.Q, ST.W  |

2001-04-30 @ 15:16

**Table 10-221  
ST.A Operation**

| <mode>                     | Syntax       | Effective Address                        | Instruction Format |
|----------------------------|--------------|--|--------------------|
| <b>Absolute</b>            | offset       | {offset18[17:14], 14'b0, offset18[13:0]} | ABS                |
| <b>Base + Short Offset</b> | [An]offset   | A[b]+sign_ext(offset10)                  | BO                 |
| <b>Pre-increment</b>       | [+An]offset  | A[b]+sign_ext(offset10)                  | BO                 |
| <b>Post-increment</b>      | [An+]offset  | A[b]                                     | BO                 |
| <b>Circular</b>            | [An+c]offset | A[b]+A[b+1][15:0] (b is even)            | BO                 |
| <b>Bit-reverse</b>         | [An+r]       | A[b]+A[b+1][15:0] (b is even)            | BO                 |

**10.4.200 Store Word from Address Register (16-bit) ..... ST.A**
**Table 10-222  
ST.A (16-bit)**

|                    |  |
|--------------------|--|
| <b>Syntax</b>      | st.a    [Ab],Aa    (SSR)    Register indirect<br>st.a    [A15] offset4, Aa    (SSRO)    Implicit base + offset<br>st.a    [Ab] offset4, A15    (SRO)    Implicit source<br>st.a    [A10]offset8, A15    (SC)    Stack pointer + offset<br>st.a    [Ab+], Aa    (SSR)    Post-increment |
| <b>Description</b> | Store the value in address register Aa/A15 to the memory location specified by the addressing mode.  |
| <b>Operation</b>   | M(A[b], word) = A[a]<br>M(A[15]+ zero_ext(4*offset4), word)=A[a]<br>M(A[b]+ zero_ext(4*offset4), word)=A[15]<br>M(A[10]+ zero_ext(4*offset8), word)=A[15]<br>M(A[b], word) = A[a]; A[b]=A[b]+4   |
| <b>Status</b>      | -  |
| <b>Examples</b>    | -  |
| <b>See also</b>    | ST.B, ST.H, ST.W   |

**10.4.201 Store Byte ..... ST.B**
**Table 10-223  
ST.B**

|                    |  |
|--------------------|--|
| <b>Syntax</b>      | st.b    <mode>, Da   |
| <b>Description</b> | Store the byte value in the 8 least-significant bits of data register Da to the byte memory location specified by the addressing mode. |

2001-04-30 @ 15:16

**Table 10-223**  
**ST.B**

|                  |  |
|------------------|--|
| <b>Operation</b> | $M(EA, \text{byte}) = D[a][7:0]$ (See <a href="#">Table 10-224</a> ) |
| <b>Status</b>    | -  |
| <b>Examples</b>  | -  |
| <b>See also</b>  | ST.A, ST.D, ST.DA, ST.H, ST.Q, ST.W                                  |

**Table 10-224**  
**ST.B Operation**

| <mode>                     | Syntax       | Effective Address                          | Instruction Format |
|----------------------------|--------------|--|--------------------|
| <b>Absolute</b>            | offset       | {offset18[17:14], 14'b0, offset18[13:0]}   | ABS                |
| <b>Base + Short Offset</b> | [An]offset   | $A[b] + \text{sign\_ext}(\text{offset}10)$ | BO                 |
| <b>Pre-increment</b>       | [+An]offset  | $A[b] + \text{sign\_ext}(\text{offset}10)$ | BO                 |
| <b>Post-increment</b>      | [An+]offset  | $A[b]$                                     | BO                 |
| <b>Circular</b>            | [An+c]offset | $A[b] + A[b+1][15:0]$ (b is even)          | BO                 |
| <b>Bit-reverse</b>         | [An+r]       | $A[b] + A[b+1][15:0]$ (b is even)          | BO                 |

### 10.4.202 Store Byte (16-bit) . . . . . ST.B

**Table 10-225**  
**ST.B (16-bit)**

|                    |  |
|--------------------|--|
| <b>Syntax</b>      | st.b $A[b], Da$ (SSR) Register indirect<br>st.b $A[15]\text{offset}4, Da$ (SSRO) Implicit base+offset<br>st.b $A[b]\text{offset}4, D15$ (SRO) Implicit source register<br>st.b $A[Ab+], Da$ (SSR) Post-increment                                 |
| <b>Description</b> | Store the byte value in the 8 least-significant bits of data register $Da/D15$ to the byte memory location specified by the addressing mode.   |
| <b>Operation</b>   | $M(A[b], \text{byte}) = D[a][7:0]$<br>$M(A[15] + \text{zero\_ext}(\text{offset}4), \text{byte}) = D[a][7:0]$<br>$M(A[b] + \text{zero\_ext}(2 * \text{offset}4), \text{byte}) = D15[15:0]$<br>$M(A[b], \text{byte}) = D[a][7:0]; A[b] = A[b] + 1$ |
| <b>Status</b>      | -  |
| <b>Examples</b>    | -  |
| <b>See also</b>    | ST.A, ST.H, ST.W   |

2001-04-30 @ 15:16

#### 10.4.203 Store Doubleword ..... ST.D

**Table 10-226**  
**ST.D**

|                    |   |
|--------------------|---|
| <b>Syntax</b>      | st.d <mode>, Ea   |
| <b>Description</b> | Store the value in the extended data register pair <i>Ea</i> to the memory location specified by the addressing mode. The value in the even register ( <i>Dn</i> ) is stored in the least-significant memory word, and the value in the odd register ( <i>Dn+1</i> ) is stored in the most-significant memory word. |
| <b>Operation</b>   | $M(EA, \text{doubleword}) = E[a]$ (See <a href="#">Table 10-227</a> )   |
| <b>Status</b>      | -   |
| <b>Examples</b>    | -   |
| <b>See also</b>    | ST.A, ST.B, ST.DA, ST.H, ST.Q, ST.W   |

**Table 10-227**  
**ST.D Operation**

| <mode>                     | Syntax       | Effective Address                          | Instruction Format |
|----------------------------|--------------|--|--------------------|
| <b>Absolute</b>            | offset       | {offset18[17:14], 14'b0, offset18[13:0]}   | ABS                |
| <b>Base + Short Offset</b> | [An]offset   | $A[b] + \text{sign\_ext}(\text{offset}10)$ | BO                 |
| <b>Pre-increment</b>       | [+An]offset  | $A[b] + \text{sign\_ext}(\text{offset}10)$ | BO                 |
| <b>Post-increment</b>      | [An+]offset  | $A[b]$                                     | BO                 |
| <b>Circular</b>            | [An+c]offset | $A[b] + A[b+1][15:0]$ (b is even)          | BO                 |
| <b>Bit-reverse</b>         | [An+r]       | $A[b] + A[b+1][15:0]$ (b is even)          | BO                 |

#### 10.4.204 Store Doubleword from Address Registers ..... ST.DA

**Table 10-228**  
**ST.DA**

|                    |   |
|--------------------|---|
| <b>Syntax</b>      | st.da <mode>, Aa  |
| <b>Description</b> | Store the value in the address register pair <i>Aa</i> to the memory location specified by the addressing mode. The value in the even register ( <i>An</i> ) is stored in the least-significant memory word, and the value in the odd register ( <i>An+1</i> ) is stored in the most-significant memory word. |
| <b>Operation</b>   | $M(EA, \text{doubleword}) = A[a](\text{pair})$ (See <a href="#">Table 10-229</a> )  |
| <b>Status</b>      | -   |
| <b>Examples</b>    | -   |
| <b>See also</b>    | ST.A, ST.B, ST.D, ST.H, ST.Q, ST.W  |

2001-04-30 @ 15:16

**Table 10-229**  
**ST.DA Operation**

| <mode>                     | Syntax       | Effective Address                        | Instruction Format |
|----------------------------|--------------|--|--------------------|
| <b>Absolute</b>            | offset       | {offset18[17:14], 14'b0, offset18[13:0]} | ABS                |
| <b>Base + Short Offset</b> | [An]offset   | A[b]+sign_ext(offset10)                  | BO                 |
| <b>Pre-increment</b>       | [+An]offset  | A[b]+sign_ext(offset10)                  | BO                 |
| <b>Post-increment</b>      | [An+]offset  | A[b]                                     | BO                 |
| <b>Circular</b>            | [An+c]offset | A[b]+A[b+1][15:0] (b is even)            | BO                 |
| <b>Bit-reverse</b>         | [An+r]       | A[b]+A[b+1][15:0] (b is even)            | BO                 |

#### 10.4.205 Store Halfword. . . . . ST.H

**Table 10-230**  
**ST.H**

|                    |   |
|--------------------|---|
| <b>Syntax</b>      | st.h <mode>, Da   |
| <b>Description</b> | Store the halfword value in the 16 least-significant bits of data register Da to the halfword memory location specified by the addressing mode. |
| <b>Operation</b>   | M(EA, halfword) = D[a][15:0] (See <a href="#">Table 10-231</a> )  |
| <b>Status</b>      | -   |
| <b>Examples</b>    | -   |
| <b>See also</b>    | ST.A, ST.B, ST.D, ST.DA, ST.Q, ST.W   |

**Table 10-231**  
**ST.H Operation**

| <mode>                     | Syntax       | Effective Address                        | Instruction Format |
|----------------------------|--------------|--|--------------------|
| <b>Absolute</b>            | offset       | {offset18[17:14], 14'b0, offset18[13:0]} | ABS                |
| <b>Base + Short Offset</b> | [An]offset   | A[b]+sign_ext(offset10)                  | BO                 |
| <b>Pre-increment</b>       | [+An]offset  | A[b]+sign_ext(offset10)                  | BO                 |
| <b>Post-increment</b>      | [An+]offset  | A[b]                                     | BO                 |
| <b>Circular</b>            | [An+c]offset | A[b]+A[b+1][15:0] (b is even)            | BO                 |
| <b>Bit-reverse</b>         | [An+r]       | A[b]+A[b+1][15:0] (b is even)            | BO                 |

2001-04-30 @ 15:16

**10.4.206 Store Halfword (16-bit) . . . . . ST.H**
**Table 10-232  
ST.H (16-bit)**

|                    |   |
|--------------------|---|
| <b>Syntax</b>      | st.h [Ab],Da (SSR) Register indirect<br>st.h [A15]offset4,Da (SSRO) Implicit base+offset<br>st.h [Ab]offset4,D15 (SRO) Implicit source register<br>st.h [Ab+],Da (SSR) Post-increment   |
| <b>Description</b> | Store the halfword value in the 16 least-significant bits of data register Da/D15 to the halfword memory location specified by the addressing mode.   |
| <b>Operation</b>   | $M(A[b], \text{halfword}) = D[a][15:0]$<br>$M([A15] + \text{zero\_ext}(2 * \text{offset4}), \text{halfword}) = D[a][15:0]$<br>$M(A[b] + \text{zero\_ext}(2 * \text{offset4}), \text{halfword}) = D[15][15:0]$<br>$M(A[b] \text{ halfword}) = D[a][15:0]; A[b] = A[b] + 2$ |
| <b>Status</b>      | -   |
| <b>Examples</b>    | -   |
| <b>See also</b>    | ST.A, ST.B, ST.W  |

**10.4.207 Store Halfword Signed Fraction . . . . . ST.Q**
**Table 10-233  
ST.Q**

|                    |   |
|--------------------|---|
| <b>Syntax</b>      | st.q <mode>, Da   |
| <b>Description</b> | Store the value in the most-significant halfword of data register Da to the memory location specified by the addressing mode. |
| <b>Operation</b>   | $M(EA, \text{halfword}) = D[a][31:16]$ (See <a href="#">Table 10-234</a> )  |
| <b>Status</b>      | -   |
| <b>Examples</b>    | -   |
| <b>See also</b>    | ST.A, ST.B, ST.D, ST.DA, ST.H, ST.W   |

**Table 10-234  
ST.Q Operation**

| <mode>                     | Syntax       | Effective Address                          | Instruction Format |
|----------------------------|--------------|--|--------------------|
| <b>Absolute</b>            | offset       | {offset18[17:14], 14'b0, offset18[13:0]}   | ABS                |
| <b>Base + Short Offset</b> | [An]offset   | $A[b] + \text{sign\_ext}(\text{offset10})$ | BO                 |
| <b>Pre-increment</b>       | [+An]offset  | $A[b] + \text{sign\_ext}(\text{offset10})$ | BO                 |
| <b>Post-increment</b>      | [An+]offset  | $A[b]$                                     | BO                 |
| <b>Circular</b>            | [An+c]offset | $A[b] + A[b+1][15:0]$ (b is even)          | BO                 |
| <b>Bit-reverse</b>         | [An+r]       | $A[b] + A[b+1][15:0]$ (b is even)          | BO                 |

2001-04-30 @ 15:16

### 10.4.208 Store Bit . . . . . ST.T

**Table 10-235**  
**ST.T**

|                    |   |
|--------------------|---|
| <b>Syntax</b>      | st.t      offset18, bpos3, b  |
| <b>Description</b> | Store the bit value <i>b</i> to the byte at the memory address specified by <i>offset18</i> in the bit position specified by <i>bpos3</i> . The other bits of the byte are unchanged. |
| <b>Operation</b>   | $M(EA, \text{byte}) = (M(EA, \text{byte}) \text{ AND } !(1 \ll bpos3)) \text{ OR } (b \ll bpos3)$ (See <a href="#">Table 10-236</a> )   |
| <b>Status</b>      | -   |
| <b>Examples</b>    | -   |
| <b>See also</b>    | IMASK   |

**Table 10-236**  
**ST.T Operation**

| <mode>          | Syntax | Effective Address                        | Instruction Format |
|-----------------|--------|--|--------------------|
| <b>Absolute</b> | offset | {offset18[17:14], 14'b0, offset18[13:0]} | ABSB               |

### 10.4.209 Store Word . . . . . ST.W

**Table 10-237**  
**ST.W**

|                    |  |
|--------------------|--|
| <b>Syntax</b>      | st.w      <mode>, Da   |
| <b>Description</b> | Store the word value in data register <i>Da</i> to the memory location specified by the addressing mode. |
| <b>Operation</b>   | $M(EA, \text{word}) = D[a]$ (See <a href="#">Table 10-238</a> )  |
| <b>Status</b>      | -  |
| <b>Examples</b>    | -  |
| <b>See also</b>    | ST.B, ST.D, ST.DA, ST.H, ST.Q  |

**Table 10-238**  
**ST.W Operation**

| <mode>                     | Syntax     | Effective Address                        | Instruction Format |
|----------------------------|------------|--|--------------------|
| <b>Absolute</b>            | offset     | {offset18[17:14], 14'b0, offset18[13:0]} | ABS                |
| <b>Base + Short Offset</b> | [An]offset | A[b]+sign_ext(offset10)                  | BO                 |
| <b>Base + Long Offset</b>  | [An]offset | A[b]+sign_ext(offset16)                  | BOL                |



2001-04-30 @ 15:16

**Table 10-238**  
**ST.W Operation (cont'd)**

| <mode>                | Syntax       | Effective Address             | Instruction Format |
|-----------------------|--------------|-------------------------------|--------------------|
| <b>Pre-increment</b>  | [+An]offset  | A[b]+sign_ext(offset10)       | BO                 |
| <b>Post-increment</b> | [An+]offset  | A[b]                          | BO                 |
| <b>Circular</b>       | [An+c]offset | A[b]+a[b+1][15:0] (b is even) | BO                 |
| <b>Bit-reverse</b>    | [An+r]       | A[b]+a[b+1][15:0] (b is even) | BO                 |

#### 10.4.210 Store Word (16-bit) ..... ST.W

**Table 10-239**  
**ST.W (16-bit)**

|                    |   |
|--------------------|---|
| <b>Syntax</b>      | st.w [Ab], Da (SSR) Register indirect<br>st.w [A15]offset4, Da (SSRO) Implicit base + offset<br>st.w [Ab]offset4, D15 (SRO) Implicit source register<br>st.w [A10]offset8, D15 (SC) Stack pointer + offset<br>st.w [Ab+], Da (SSR) Post-increment |
| <b>Description</b> | Store the word value in data register <i>Da/D15</i> to the memory location specified by the addressing mode.  |
| <b>Operation</b>   | M(A[b], word) = D[a]<br>M(A[15]+zero_ext (4*offset4), word) = D[a]<br>M(A[b]+zero_ext (4*offset4), word) = D[15]<br>M(A[10]+zero_ext (4*offset8), word) = D[15]<br>M(A[b],word= D[a]; A[b]+A[b]+4   |
| <b>Status</b>      | -   |
| <b>Examples</b>    | -   |
| <b>See also</b>    | ST.A, ST.B, ST.H  |

**10.4.211 Store Lower Context . . . . . STLCX**

**Table 10-240**  
**STLCX**

|                    |  |
|--------------------|--|
| <b>Syntax</b>      | stlcx <mode>   |
| <b>Description</b> | <p>Store the contents of registers A2 – A7, D0 – D7, and A11 to the memory block specified by the addressing mode. For this instruction, the addressing mode is limited to absolute (ABS) or base plus short offset (BO).</p> <p>Note that the effective address specified by the addressing mode must be aligned on a 16-word boundary. For this instruction, the addressing mode is restricted to absolute (ABS) or base plus short offset (BO).</p> |
| <b>Operation</b>   | M(EA, 16-word) = {PCXI, A[11], A[2:3], D[0:3], A[4:7], D[4:7]} See <a href="#">Table 10-241</a>  |
| <b>Status</b>      | -  |
| <b>Examples</b>    | -  |
| <b>See also</b>    | LDLCX, LDUCX, RSLCX, STUCX, SVLCX  |

**Table 10-241**  
**STLCX Operation**

| <mode>                     | Syntax     | Effective Address       | Instruction Format |
|----------------------------|------------|-------------------------|--------------------|
| <b>Absolute</b>            | constant   |                         | ABS                |
| <b>Base + Short Offset</b> | [An]offset | A[a]+sign_ext(offset10) | BO                 |

**Note :** The effective address specified by the addressing mode must be aligned on a 16-word boundary.

**10.4.212 Store Upper Context . . . . . STUCX**

**Table 10-242**  
**STUCX**

|                    |   |
|--------------------|---|
| <b>Syntax</b>      | stucx <mode>  |
| <b>Description</b> | <p>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. For this instruction, the addressing mode is limited to absolute (ABS) or base plus short offset (BO).</p> <p>Note that the effective address specified by the addressing mode must be aligned on a 16-word boundary. For this instruction, the addressing mode is restricted to absolute (ABS) or base plus short offset (BO).</p> |

2001-04-30 @ 15:16

**Table 10-242**
**STUCX**

|                  |   |
|------------------|---|
| <b>Operation</b> | M(EA, 16-word) = {PCXI, PSW, A[10:11], D[8:11], A[12:15], D[12:15]}<br>See <a href="#">Table 10-243</a> |
| <b>Status</b>    | -   |
| <b>Examples</b>  | -   |
| <b>See also</b>  | LDLCX, LDUCX, RSLCX, STLCX, SVLCX   |

**Table 10-243**
**STUCX Operation**

| <mode>                     | Syntax     | Effective Address       | Instruction Format |
|----------------------------|------------|-------------------------|--------------------|
| <b>Absolute</b>            | constant   |                         | ABS                |
| <b>Base + Short Offset</b> | [An]offset | A[a]+sign_ext(offset10) | BO                 |

**Note :** The effective address specified by the addressing mode must be aligned on a 16-word boundary.

**10.4.213 Subtract . . . . . SUB**
**Table 10-244**
**SUB**

|                    |   |                   |
|--------------------|---|-------------------|
| <b>Syntax</b>      | sub   | Dc, Da, Db (RR)   |
|                    | sub   | Da, Db (SRR)      |
|                    | sub   | D15, Da, Db (SRR) |
|                    | sub   | Da, D15, Db (SRR) |
| <b>Description</b> | Subtract the contents of data register <i>Db</i> from the contents of data register <i>Da</i> and put the result in data register <i>Dc</i> . The operaands are treated as unsigned, 32-bit integers.         |                   |
|                    | Subtract the contents of data register <i>Db</i> from the contents of data register <i>Da/D15</i> and put the result in data register <i>Da/D15</i> . The operaands are treated as unsigned, 32-bit integers. |                   |
| <b>Operation</b>   | $D[c] = D[a] - D[b]$  |                   |
|                    | $D[a] = D[a] - D[b]$  |                   |
|                    | $D[15] = D[a] - D[b]$   |                   |
|                    | $D[a] = D[15] - D[b]$   |                   |

2001-04-30 @ 15:16

**Table 10-244**  
**SUB**

|                 |                          |
|-----------------|--------------------------|
| <b>Status</b>   | V, SV, AV, SAV           |
|                 | V, SV, AV, SAV           |
| <b>Examples</b> | sub    d3, d1, d2        |
|                 | sub    d1, d2            |
|                 | sub    d15, d1, d2       |
|                 | sub    d1, d15, d2       |
| <b>See also</b> | SUBS, SUBS.U, SUBX, SUBC |

**10.4.214 Subtract Address .....SUB.A**

**Table 10-245**  
**SUB.A**

|                    |  |
|--------------------|--|
| <b>Syntax</b>      | sub.a    Ac, Aa, Ab (RR)   |
|                    | sub.a    SP, const8 (SC)   |
| <b>Description</b> | Subtract the contents of address register <i>Ab</i> from the contents of address register <i>Aa</i> and put the result in address register <i>Ac</i> . |
|                    | Decrement the Stack Pointer (A10) by the zero-extended value of <i>const8</i> (a range of 0 through 255).  |
| <b>Operation</b>   | $A[c] = A[a] - A[b]$   |
|                    | $A[10] = A[10] - \text{zero\_ext}(\text{const8})$  |
| <b>Status</b>      | -  |
|                    | -  |
| <b>Examples</b>    | sub.a    a3, a4, a2  |
|                    | sub.a    sp, 126   |
| <b>See also</b>    | ADD.A, ADDIH.A, ADDSC.A, ADDSC.AT  |

2001-04-30 @ 15:16

**10.4.215 Subtract Packed Byte . . . . .SUB.B**  
**Subtract Packed Halfword . . . . .SUB.H**

**Table 10-246**
**SUB.B & SUB.H**

|                    |   |
|--------------------|---|
| <b>Syntax</b>      | sub.b Dc, Da, Db (RR)<br>sub.h Dc, Da, Db (RR)  |
| <b>Description</b> | Subtract the contents of each byte/halfword of data register <i>Db</i> from the contents of data register <i>Da</i> and put the result in each corresponding byte/halfword of data register <i>Dc</i> |
| <b>Operation</b>   | $D[c][(n+7):n] = D[a][(n+7):n] + D[b][(n+7):n]; n = 0, 8, 16, 24;$<br>$D[c][(n+15):n] = D[a][(n+15):n] + D[b][(n+15):n]; n = 0, 16$   |
| <b>Status</b>      | V, SV, AV, SAV  |
| <b>Examples</b>    | sub.b d3, d1, d2<br>sub.h d3, d1, d2  |
| <b>See also</b>    | SUBS.H, SUBS.HU   |

**10.4.216 Subtract With Carry . . . . .SUBC**

**Table 10-247**
**SUBC**

|                    |  |
|--------------------|--|
| <b>Syntax</b>      | subc Dc, Da, Db (RR)   |
| <b>Description</b> | Subtract contents of data register <i>Db</i> from contents of data register <i>Da</i> plus the carry bit minus 1, and put the result in data register <i>Dc</i> . The operands are treated as 32-bit integers. The PSW carry bit is set to the value of the ALU carry out. |
| <b>Operation</b>   | $D[c] = D[a] - D[b] + \text{psw.C} - 1; \text{psw.C} = \text{carry\_out}$  |
| <b>Status</b>      | C, V, SV, AV, SAV  |
| <b>Examples</b>    | subc d3, d1, d2  |
| <b>See also</b>    | SUB, SUBS, SUBS.U, SUBX  |

2001-04-30 @ 15:16

**10.4.217 Subtract Signed with Saturation . . . . . SUBS**  
**Subtract Unsigned with Saturation . . . . . SUBS.U**

**Table 10-248**  
**SUBS & SUBS.U**

|                    |   |
|--------------------|---|
| <b>Syntax</b>      | subs    Dc, Da, Db (RR)   |
|                    | subs.u   Dc, Da, Db (RR)  |
|                    | subs    Da, Db (SRR)  |
| <b>Description</b> | Subtract the contents of data register <i>Db</i> from the contents of data register <i>Da</i> and put the result in data register <i>Dc</i> . The operands are treated as signed/unsigned 32-bit integers, with saturation on signed/unsigned overflow. |
|                    | Subtract the contents of data register <i>Db</i> from the contents of data register <i>Da</i> and put the result in data register <i>Da</i> . The operands are treated as signed 32-bit integers, with saturation on signed overflow.                   |
| <b>Operation</b>   | $D[c] = D[a] - D[b]$ ; signed; ssov   |
|                    | $D[c] = D[a] - D[b]$ ; unsigned; suov   |
|                    | $D[a] = D[a] - D[b]$ ; signed; ssov   |
| <b>Status</b>      | V, SV, AV, SAV  |
|                    | V, SV, AV, SAV  |
| <b>Examples</b>    | subs        d3, d1, d2  |
|                    | subs.u     d3, d1, d2   |
|                    | subs        d3, d1  |
| <b>See also</b>    | SUB, SUBX, SUBC   |

**10.4.218 Subtract Packed Halfword with Saturation. . . . . SUBS.H**  
**Subtract Packed Halfword Unsigned w/Saturat'n . . . . . SUBS.HU**

**Table 10-249**  
**SUBS.H & SUBS.HU**

|                    |  |
|--------------------|--|
| <b>Syntax</b>      | subs.h   Dc, Da, Db (RR)   |
|                    | subs.hu   Dc, Da, Db (RR)  |
| <b>Description</b> | Subtract the contents of each halfword of data register <i>Db</i> from the contents of data register <i>Da</i> and put the result in each corresponding halfword of data register <i>Dc</i> , with saturation on signed/unsigned overflow. |
| <b>Operation</b>   | $D[c][(n+15):n] = D[a][(n+15):n] - D[b][(n+15):n]$ ; $n = 0, 16$ ; signed; ssov  |
|                    | $D[c][(n+15):n] = D[a][(n+15):n] - D[b][(n+15):n]$ ; $n = 0, 16$ ; unsigned; suov  |
| <b>Status</b>      | V, SV, AV, SAV   |
| <b>Examples</b>    | subs.h     d3, d1, d2  |
|                    | subs.hu    d3, d1, d2  |
| <b>See also</b>    | SUB.B  |

2001-04-30 @ 15:16

#### 10.4.219 Subtract Extended ..... SUBX

**Table 10-250**  
**SUBX**

|                    |  |
|--------------------|--|
| <b>Syntax</b>      | subx Dc, Da, Db (RR)   |
| <b>Description</b> | Subtract the contents of data register <i>Db</i> from the contents of data register <i>Da</i> and put the result in data register <i>Dc</i> . The operands are treated as 32-bit integers. The PSW carry bit is set to the value of the ALU carry out. |
| <b>Operation</b>   | $D[c] = D[a] - D[b]$ ; psw.C = carry_out   |
| <b>Status</b>      | C, V, SV, AV, SAV  |
| <b>Examples</b>    | subx d3, d1, d2  |
| <b>See also</b>    | SUB, SUBC, SUBS, SUBS.U  |

#### 10.4.220 Save Lower Context ..... SVLCX

**Table 10-251**  
**SVLCX**

|                    |   |
|--------------------|---|
| <b>Syntax</b>      | svlcx (SYS)   |
| <b>Description</b> | 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. |
| <b>Operation</b>   | Save lower context  |
| <b>Status</b>      | -   |
| <b>Examples</b>    | svlcx   |
| <b>See also</b>    | LDLCX, LDUCX, RSLCX, STLCX, STUCX   |

#### 10.4.221 Swap with Data Register ..... SWAP.W

**Table 10-252**  
**SWAP.W**

|                    |   |
|--------------------|---|
| <b>Syntax</b>      | swap.w Da, <mode>   |
| <b>Description</b> | Swap atomically the contents of data register <i>Db</i> and the memory word specified by the addressing mode. |
| <b>Operation</b>   | tmp = M(EA, word);<br>M(EA, word) = D[a];<br>D[a] = tmp<br>(See <a href="#">Table 10-253</a> )                |
| <b>Status</b>      | -   |
| <b>Examples</b>    | -   |
| <b>See also</b>    | -   |

2001-04-30 @ 15:16

**Table 10-253**  
**SWAP.W Operation**

| <mode>                     | Syntax       | Effective Address                        | Instruction Format |
|----------------------------|--------------|--|--------------------|
| <b>Absolute</b>            | offset       | {offset18[17:14], 14'b0, offset18[13:0]} | ABS                |
| <b>Base + Short Offset</b> | [An]offset   | A[b]+sign_ext(offset10)                  | BO                 |
| <b>Pre-increment</b>       | [+An]offset  | A[b]+sign_ext(offset10)                  | BO                 |
| <b>Post-increment</b>      | [An+]offset  | A[b]                                     | BO                 |
| <b>Circular</b>            | [An+c]offset | A[b]+A[b+1][15:0] (b is even)            | BO                 |
| <b>Bit-reverse</b>         | [An+r]       | A[b]+A[b+1][15:0] (b is even)            | BO                 |

### 10.4.222 System Call .....SYSCALL

**Table 10-254**  
**SYSCALL**

|                    |   |
|--------------------|---|
| <b>Syntax</b>      | syscall const9 (RC)   |
| <b>Description</b> | Cause a system call trap, using Trap Identification Number (TIN) specified by <i>const9</i> . |
| <b>Operation</b>   | trap(SYS, const9[7:0])  |
| <b>Status</b>      | -   |
| <b>Examples</b>    | syscall 4   |
| <b>See also</b>    | RET, RFE, TRAPV, TRAPSV   |

**Note :** The trap return PC will be that of the instruction following the SYSCALL instruction.



2001-04-30 @ 15:16

#### 10.4.223 TLB Map . . . . . TLBMAP

**Table 10-255**  
**TLBMAP**

|                    |   |
|--------------------|---|
| <b>Syntax</b>      | tlbmap Ea   |
| <b>Description</b> | The TLBMAP instruction is used to install a mapping in the MMU. The TLBMAP instruction takes an extended data register ( <i>Ea</i> ) as a parameter. The even <i>Ea</i> register contains the VPN for the translation while the odd <i>Ea</i> register contains the page attributes and PPN. The ASI for the translation is obtained from the ASI register. The page attributes are contained in the most significant byte of the odd register with the format shown in <a href="#">Section 7.8.1, “TLBMAP (TLB Map).”</a><br>This instruction can be executed in supervisor mode only. |
| <b>Operation</b>   | TTE.ASI = ASI<br>TTE.VPN = E[a](lower)[21:0]<br>TTE.PPN = E[a](upper)[21:0]<br>TTE.(attributes) = E[a](upper)[31:24]  |
| <b>Status</b>      |   |
| <b>Examples</b>    | tlbmap e2   |
| <b>See also</b>    | TLBMAP, TLBDEMAP, TLBFLUSH, TLBPROBE, TLBPROBE.A, TLBPROBE.I and <a href="#">Section 7.8.1, “TLBMAP (TLB Map).”</a>   |

#### 10.4.224 TLB Demap . . . . . TLBDEMAP

**Table 10-256**  
**TLBDEMAP**

|                    |   |
|--------------------|---|
| <b>Syntax</b>      | tlbdemap Da   |
| <b>Description</b> | The TLBDEMAP instruction is used to uninstall a mapping in the MMU. The TLBDEMAP instruction takes a data register ( <i>Da</i> ) as a parameter. The <i>Da</i> register contains a PSZ and a VPN for the demap operation. The address space identifier for the demap operation is obtained from the ASI register. Demapping a translation that does not exist in the MMU results in a NOP.<br>This instruction can be executed in supervisor mode only. |
| <b>Operation</b>   |   |
| <b>Status</b>      |   |
| <b>Examples</b>    | tlbdemap d2   |
| <b>See also</b>    | TLBMAP, TLBFLUSH, TLBPROBE, TLBPROBE.A, TLBPROBE.I and <a href="#">Section 7.8.2, “TLBDEMAP (TLB Demap).”</a>   |

2001-04-30 @ 15:16

**10.4.225 TLB-A Flush..... TLBFLUSH.A**  
**TLB-B Flush ..... TLBFLUSH.B**

**Table 10-257**  
**TLBFLUSH**

|                    |   |
|--------------------|---|
| <b>Syntax</b>      | tlbflush  |
| <b>Description</b> | <p>The TLBFLUSH instructions are used to flush mappings from the MMU. There are two variants of the tlbflush instruction. The tlbflush.a instruction flushes all the mappings from TLB-A while the tlbflush.b instruction flushes all the mappings from TLB-B.</p> <p>This instruction can be executed in supervisor mode only.</p> |
| <b>Operation</b>   |   |
| <b>Status</b>      |   |
| <b>Examples</b>    | <pre>tlbflush.a tlbflush.b</pre>  |
| <b>See also</b>    | TLBMAP, TLBDEMAP, TLBPROBE, TLBPROBE.A, TLBPROBE.I and <a href="#">Section 7.8.3, "TLBFLUSH (TLB Flush)."</a>   |

**10.4.226 TLB Probe Address..... TLBPROBE.A**

**Table 10-258**  
**TLBPROBE.A**

|                    |  |
|--------------------|--|
| <b>Syntax</b>      | tlbprobe.a   |
| <b>Description</b> | <p>The TLBPROBE.A (TLB Probe Address) instruction takes a data register (<i>Da</i>) as a parameter and is used to probe the MMU for a virtual address. The <i>Da</i> register contains the virtual address for the probe. The address space identifier for the probe is obtained from the ASI register.</p> <p>This instruction can be executed in supervisor mode only.</p> |
| <b>Operation</b>   | <pre>if ( TLB contains an entry which matches the ASI /VPN in Da ) {     index = matched TLB entry     TVA.ASI = TLB[index].ASI     TVA.VPN = TLB[index].VPN     TPA.PPN = TLB[index].PPN     TPA.{attributes} = TLB[index].{attributes}     TPX = index } else {     TPA.V = 0 }</pre>  |
| <b>Status</b>      |  |
| <b>Examples</b>    | tlbprobe.a d2  |
| <b>See also</b>    | TLBMAP, TLBDEMAP, TLBFLUSH, TLBPROBE, TLBPROBE.I and <a href="#">Section 7.8.4, "TLBPROBE (TLB Probe)."</a>  |

2001-04-30 @ 15:16

#### 10.4.227 TLB Probe Address. .... TLBPROBE.I

**Table 10-259**  
**TLBPROBE.I**

|                    |  |
|--------------------|--|
| <b>Syntax</b>      | tlbprobe.i   |
| <b>Description</b> | <p>The TLBPROBE.I (TLB Probe Index) instruction takes a data register (Da) as a parameter and is used to probe the TLB at a given index. The Da register contains the index for the probe.</p> <p>This instruction can be executed in supervisor mode only.</p>      |
| <b>Operation</b>   | <pre> if (Da is a valid TLB entry index ) {     index = Da     TVA.ASI = TLB[index].ASI     TVA.VPN = TLB[index].VPN     TPA.PPN = TLB[index].PPN     TPA.{attributes} = TLB[index].{attributes}     TPX = index } else {     TPA.V = 0 }                     </pre> |
| <b>Status</b>      |  |
| <b>Examples</b>    | tlbprobe.i d2  |
| <b>See also</b>    | TLBMAP, TLBDEMAP, TLBFLUSH, TLBPROBE, TLBPROBE.A and <a href="#">Section 7.8.4, "TLBPROBE (TLB Probe)."</a>  |

#### 10.4.228 Trap on Overflow. .... TRAPV

**Table 10-260**  
**TRAPV**

|                    |   |
|--------------------|---|
| <b>Syntax</b>      | trapv (SYS)   |
| <b>Description</b> | 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). |
| <b>Operation</b>   | if PSW.V then trap (OVF)  |
| <b>Status</b>      | -   |
| <b>Examples</b>    | trapv   |
| <b>See also</b>    | RSTV, SYSCALL, TRAPSV   |

2001-04-30 @ 15:16

### 10.4.229 Trap on Sticky Overflow . . . . . TRAPSV

Table 10-261

#### TRAPSV

|                    |   |
|--------------------|---|
| <b>Syntax</b>      | trapsv (SYS)  |
| <b>Description</b> | If the PSWs sticky overflow status flag (PSW.SV) is set, generate a trap to the vector entry for the sticky overflow trap handler (SOV-trap). |
| <b>Operation</b>   | if PSW.SV then trap (SOVF)  |
| <b>Status</b>      | -   |
| <b>Examples</b>    | trapsv  |
| <b>See also</b>    | RSTV, SYSCALL, TRAPV  |

### 10.4.230 Logical XNOR . . . . . XNOR

Table 10-262

#### XNOR

|                    |  |
|--------------------|--|
| <b>Syntax</b>      | xnor Dc, Da, Db (RR)<br>xnor Dc, Da, const9 (RC)   |
| <b>Description</b> | Compute the bitwise logical exclusive NOR of the contents of data register <i>Da</i> and the contents of data register <i>Db/const9</i> and put the result in data register <i>Dc</i> . The value <i>const9</i> is zero-extended to 32 bits. |
| <b>Operation</b>   | $D[c] = \neg(D[a] \text{ XOR } D[b])$<br>$D[c] = \neg(D[a] \text{ XOR } \text{zero\_ext}(\text{const9}))$  |
| <b>Status</b>      | -  |
| <b>Examples</b>    | xnor d3, d1, d2<br>xnor d3, d1, 126  |
| <b>See also</b>    | AND, ANDN, NAND, NOR, NOT, OR, ORN, XOR  |

### 10.4.231 Bit Logical XNOR . . . . . XNOR.T

Table 10-263

#### XNOR.T

|                    |  |
|--------------------|--|
| <b>Syntax</b>      | xnor.t Dc, Da, p1, Db, p2 (BIT)  |
| <b>Description</b> | Compute the logical exclusive NOR of bit <i>p1</i> of data register <i>Da</i> and bit <i>p2</i> of data register <i>Db</i> . Put the result in the least-significant bit of data register <i>Dc</i> and clear the remaining bits of <i>Dc</i> to zero. |
| <b>Operation</b>   | $D[c] = \neg(D[a][p1] \text{ XOR } D[b][p2])$  |
| <b>Status</b>      | -  |
| <b>Examples</b>    | xnor.t d3, d1, 3, d2, 5  |
| <b>See also</b>    | AND.T, ANDN.T, NAND.T, NOR.T, OR.T, ORN.T, XOR.T   |

2001-04-30 @ 15:16

#### 10.4.232 Logical XOR..... XOR

Table 10-264

##### XOR

|             |   |
|-------------|---|
| Syntax      | xor Dc, Da, Db (RR)   |
|             | xor Dc, Da, const9 (RC)   |
|             | xor Da, Db (SR)   |
| Description | Compute the bitwise logical exclusive OR of the contents of data register <i>Da</i> and the contents of data register <i>Db/const9</i> and put the result in data register <i>Dc</i> . The value <i>const9</i> is zero-extended to 32 bits. |
|             | Compute the bitwise logical exclusive OR of the contents of data register <i>Da</i> and the contents of data register <i>Db</i> and put the result in data register <i>Da</i> .   |
| Operation   | $D[c] = D[a] \text{ xor } D[b]$   |
|             | $D[c] = D[a] \text{ xor } \text{zero\_ext}(\text{const9})$  |
|             | $D[a] = D[a] \text{ xor } D[b]$   |
| Status      | -   |
| Examples    | xor d3, d1, d2  |
|             | xor d3, d1, 126   |
|             | xor d3, d2  |
| See also    | AND, ANDN, NAND, NOR, NOT, OR, ORN, XNOR  |

#### 10.4.233 Equal Accumulating..... XOR.EQ

Table 10-265

##### XOR.EQ

|             |   |
|-------------|---|
| Syntax      | xor.eq Dc, Da, Db (RR)  |
|             | xor.eq Dc, Da, const9 (RC)  |
| Description | Compute the logical XOR of <i>Dc[0]</i> and the Boolean result of the EQ operation on the contents of data register <i>Da</i> and data register <i>Db/const9</i> . Put the result in <i>Dc[0]</i> . All other bits in <i>Dc</i> are unchanged. The value <i>const9</i> is sign-extended to 32 bits. |
| Operation   | $D[c] = \{D[c][31:1], D[c][0] \text{ XOR } (D[a]==D[b])\}$  |
|             | $D[c] = \{D[c][31:1], D[c][0] \text{ XOR } (D[a]==\text{sign\_ext}(\text{const9}))\}$   |
| Status      | -   |
| Examples    | xor.eq d3, d1, d2   |
|             | xor.eq d3, d1, 126  |
| See also    | AND.EQ, OR.EQ   |

2001-04-30 @ 15:16

**10.4.234 Greater Than or Equal Accumulating . . . . . XOR.GE**  
**Greater Than or Equal Accumulating Unsigned . . . . . XOR.GE.U**

**Table 10-266**

**XOR.GE & XOR.GE.U**

|                    |   |
|--------------------|---|
| <b>Syntax</b>      | xor.ge Dc, Da, Db (RR)<br>xor.ge Dc, Da,const9 (RC)<br>xor.ge.u Dc, Da, Db (RR)<br>xor.ge.u Dc, Da,const9 (RC)  |
| <b>Description</b> | <p>Calculate the logical XOR of <i>Dc</i>[0] and the Boolean result of the GE operation on the contents of data register <i>Da</i> and data register <i>Db</i>/<i>const9</i>. Put the result in <i>Dc</i>[0]. All other bits in <i>Dc</i> are unchanged. <i>Da</i> and <i>Db</i> are treated as 32-bit signed integers. The value <i>const9</i> is sign-extended to 32 bits.</p> <p>Calculate the logical XOR of <i>Dc</i>[0] and the Boolean result of the GE.U operation on the contents of data register <i>Da</i> and data register <i>Db</i>/<i>const9</i>. Put the result in <i>Dc</i>[0]. All other bits in <i>Dc</i> are unchanged. <i>Da</i> and <i>Db</i> are treated as 32-bit unsigned integers. The value <i>const9</i> is zero-extended to 32 bits.</p> |
| <b>Operation</b>   | D[c] = {D[c][31:1], D[c][0] XOR (D[a]>=D[b])}; signed<br>D[c] = {D[c][31:1], D[c][0] XOR (D[a]>=sign_ext(const9))}; signed<br><br>D[c] = {D[c][31:1], D[c][0] XOR (D[a]>=D[b])}; unsigned<br>D[c] = {D[c][31:1], D[c][0] XOR (D[a]>=zero_ext(const9))}; unsigned  |
| <b>Status</b>      | -   |
| <b>Examples</b>    | xor.ge d3, d1, d2<br>xor.ge d3, d1, 126<br>xor.ge.u d3, d1, d2<br>xor.ge.u d3, d1, 126  |
| <b>See also</b>    | AND.GE, AND.GE.U, OR.GE, OR.GE.U  |

**10.4.235 Less Than Accumulating . . . . .XOR.LT**  
**Less Than Accumulating Unsigned . . . . .XOR.LT.U**

**Table 10-267**  
**XOR.LT & XOR.LT.U**

|                    |   |
|--------------------|---|
| <b>Syntax</b>      | xor.lt Dc, Da, Db (RR)<br>xor.lt Dc, Da,const9 (RC)<br>xor.lt.u Dc, Da, Db (RR)<br>xor.lt.u Dc, Da,const9 (RC)  |
| <b>Description</b> | <p>Calculate the logical XOR of <math>Dc[0]</math> and the Boolean result of the LT operation on the contents of data register <math>Da</math> and data register <math>Db/const9</math>. Put the result in <math>Dc[0]</math>. All other bits in <math>Dc</math> are unchanged. <math>Da</math> and <math>Db</math> are treated as 32-bit signed integers. The value <math>const9</math> is sign-extended to 32 bits.</p> <p>Calculate the logical XOR of <math>Dc[0]</math> and the Boolean result of the LT.U operation on the contents of data register <math>Da</math> and data register <math>Db/const9</math>. Put the result in <math>Dc[0]</math>. All other bits in <math>Dc</math> are unchanged. <math>Da</math> and <math>Db</math> are treated as 32-bit unsigned integers. The value <math>const9</math> is zero-extended to 32 bits.</p> |
| <b>Operation</b>   | <p><math>D[c] = \{D[c][31:1], D[c][0] \text{ XOR } (D[a] &lt; D[b])\}</math>; signed<br/> <math>D[c] = \{D[c][31:1], D[c][0] \text{ XOR } (D[a] &lt; \text{sign\_ext}(const9))\}</math>; signed</p> <p><math>D[c] = \{D[c][31:1], D[c][0] \text{ XOR } (D[a] &lt; D[b])\}</math>; unsigned<br/> <math>D[c] = \{D[c][31:1], D[c][0] \text{ XOR } (D[a] &lt; \text{zero\_ext}(const9))\}</math>; unsigned</p>   |
| <b>Status</b>      | -   |
| <b>Examples</b>    | xor.lt d3, d1, d2<br>xor.lt d3, d1, 126<br>xor.lt.u d3, d1, d2<br>xor.lt.u d3, d1, 126  |
| <b>See also</b>    | AND.LT, AND.LT.U, OR.LT, OR.LT.U  |

**10.4.236 Not Equal Accumulating. . . . .XOR.NE**

**Table 10-268**  
**XOR.NE**

|                    |  |
|--------------------|--|
| <b>Syntax</b>      | xor.ne Dc, Da, Db (RR)<br>xor.ne Dc, Da,const9 (RC)  |
| <b>Description</b> | <p>Calculate the logical XOR of <math>Dc[0]</math> and the Boolean result of the NE operation on the contents of data register <math>Da</math> and data register <math>Db/const9</math>. Put the result in <math>Dc[0]</math>. All other bits in <math>Dc</math> are unchanged. The value <math>const9</math> is sign-extended to 32 bits.</p> |

2001-04-30 @ 15:16

**Table 10-268**  
**XOR.NE**

|                  |   |
|------------------|---|
| <b>Operation</b> | $D[c] = \{D[c][31:1], D[c][0] \text{ XOR } (D[a] \neq D[b])\}$<br>$D[c] = \{D[c][31:1], D[c][0] \text{ XOR } (D[a] \neq \text{sign\_ext}(\text{const9}))\}$ |
| <b>Status</b>    | -   |
| <b>Examples</b>  | <code>xor.ne d3, d1, d2</code><br><code>xor.ne d3, d1, 126</code>   |
| <b>See also</b>  | AND.NE, OR.NE   |

### 10.4.237 Bit Logical XOR.....XOR.T

**Table 10-269**  
**XOR.T**

|                    |  |
|--------------------|--|
| <b>Syntax</b>      | <code>xor.t Dc, Da, p1, Db, p2 (BIT)</code>  |
| <b>Description</b> | Compute the logical XOR of bit <i>p1</i> of data register <i>Da</i> and bit <i>p2</i> of data register <i>Db</i> . Put the result in the least-significant bit of data register <i>Dc</i> and clear the remaining bits of <i>Dc</i> to zero. |
| <b>Operation</b>   | $D[c] = D[a][p1] \text{ XOR } D[b][p2]$  |
| <b>Status</b>      | -  |
| <b>Examples</b>    | <code>xor.t d3, d1, 3, d2, 7</code>  |
| <b>See also</b>    | AND.T, ANDN.T, NAND.T, NOR.T, OR.T, ORN.T, XNOR.T  |



---

# Global PartnerChip for Systems on Silicon

---

**Austria (A)**

**Siemens AG Österreich**  
Erdberger Lände 26  
A-1031 Wien  
Tel. (+43) 1-1707-356 11  
Fax (+43) 1-1707-559 73

**Australia (AUS)**

**Siemens Ltd.**  
885 Mountain Highway  
Bayswater Victoria 3153  
Tel: (+61) 3 9721 2111  
Fax: (+61) 3 9721 7275

**Belgium (B)**

**Siemens Electronic Components Benelux**  
Charleroisesteenweg 116/  
Chaussée de Charleroi 116  
B-1060 Brussel/Bruuxelles  
Tel. (+32) 2-536 69 05  
Fax (+32) 2-536 28 57

**Brazil (BR)**

**Siemens Ltda. Semiconductores**  
Avenida Mutinga, 3800 - Pirituba  
05110-901 São Paulo, SP  
Tel. (55) 11-3908 2564  
Fax (55) 11-3908 27 28

**Canada (CND)**

**Siemens Electric Ltd.  
Electronic Components Division**  
1180 Courtney Park Drive  
Mississauga, Ontario L5T 1P2  
Tel. (+1) 905-819-80 00  
Fax (+1) 905-819-57 44  
**Infineon Technologies Corporation**  
320 March Road, Suite 204  
Kanata, Ontario K2K 2E2  
Tel. (+1) 613-591 63 86  
Fax (+1) 613-591 63 89

**China, People's Republic of (PRC)**

**Infineon Technologies Hong Kong Ltd.**  
Room 2106, Building A  
Vantone New World Plaza  
No. 2, Fu Cheng Men Wai Da Jie  
100037 Beijing  
Tel. (+86) 10-6857 9006/7  
Fax (+86) 10-6857 90 08

**Infineon Technologies Hong Kong Ltd.**  
Room 1101, Lucky Target Square  
No. 500 Chengdu Road North  
North Shanghai 200003  
Tel. (+86) 21-6361 2618 / 19  
Fax (+86) 21-6361 11 67

**Infineon Technologies Hong Kong Ltd.**  
Suite 302, Level 3,  
Festival Walk,  
80 Tat Chee Avenue,  
Yau Yat Tsuen,  
Kowloon Tong, Hong Kong  
Tel. (+852) 2832 05 00  
Fax (+852) 2827 97 62

**Infineon Technologies Hong Kong Ltd.**  
Room 1502, Block A  
Tian An International Building  
Renmin South Road  
Shenzhen 518005  
Tel. (+86) 755-228 9104  
Fax (+86) 755-228 0217

**Infineon Technologies Hong Kong Ltd.**  
Room 14J1, Jinyang Mansion  
58 Tidu Street  
Chengdu, Sichuan Province 610 016  
Tel. (+86) ) 28-661 5446 / 7951  
Fax (+86) 28-661 0159

**Denmark (DK)**

**Siemens A/S**  
Borupvang 3  
DK-2750 Ballerup  
Tel. (+45) 4477-4477  
Fax (+45) 4477-4017

**France (F)**

**Infineon Technologies France**  
39/47, Bd. Ornano  
F-93527 Saint-Denis CEDEX 2  
Tel. (+33) 1-4922 31 00  
Fax (+33) 1-4922 28 01

**Finland (FIN)**

**Siemens Components Scandinavia**  
P.O.Box 60  
FIN-02601 Espoo (Helsinki)  
Tel. (+358) 10-511 51 51  
Fax (+358) 10-511 24 95

**Germany (D)**

**Infineon Technologies AG**  
Völklinger Str. 2  
D-40219 Düsseldorf  
Tel. (+49) 211 399 29 30  
Fax (+49) 211 399 14 81

**Infineon Technologies AG**  
Weissacher Straße 11  
D-70499 Stuttgart  
Tel. (+49) 711-137 33 14  
Fax (+49) 711-137 24 48

**Infineon Technologies AG**  
Werner von Siemens Platz 1  
D-30880 Laatzen (Hannover)  
Tel. (+49) 511-877 22 22  
Fax (+49) 511-877 15 20

**Infineon Technologies AG  
Halbleiter Distribution**  
Richard-Strauss-Straße 76  
D-81679 München  
<http://www.infineon.com/distribution>

2001-04-30 @ 15:16

**Infineon Technologies AG**

Von-der-Tann-Straße 30  
D-90439 Nürnberg  
Tel. (+49) 911-654 76 99  
Fax (+49) 911-654 76 24

**Hong Kong (HK)**

see China, People's Republic of (PRC)

**Hungary (H)**
**Simacomp Kft.**

Lajos u. 103  
H-1036 Budapest  
Tel. (+36) - 1 457 - 1690  
Fax (+36) 1 457 1692

**India (IND)**
**Siemens Ltd.**
**Components Division, 4th Floor**

130, Pandurang Budhkar Marg,  
Worli  
Mumbai 400 018  
Tel. (+91) 22-496 21 99  
Fax (+91) 22-496 22 01

**Siemens Ltd.**
**Components Division**

No. 84 Keonics Electronic City  
Hosur Road  
Bangalore 561 229  
Tel. (+91) 80-852 1122  
Fax (+91) 80-852 1180

**Siemens Ltd.**
**Components Division, 5th Floor**

4A Ring Road, IP Estate  
New Delhi 110 002  
Tel. (+91) 11-331 9912  
Fax (+91) 11-331 9604

**Ireland (IRL)**
**Siemens Ltd.**
**Electronic Components Division**

8, Raglan Road  
IRL-Dublin 4  
Tel. (+353) 1-216 23 42  
Fax (+353) 1-603 23 49

**Israel (IL)**
**Nisko Ltd.**
**Electronic Components Division**

2A, Habarzel St.  
P.O.Box 58151  
61580 Tel Aviv  
Tel. (+972) 3-76 57 300  
Fax (+972) 3-76 57 333

**Italy (I)**
**Siemens S.p.A.  
Semiconductor Sales**

Via Piero e Alberto Pirelli, 10  
I-20126 Milano  
Tel. (+39) 02-6676-1  
Fax (+39) 02-6676 43 95

**Japan (J)**
**Infineon Technologies Japan**
**K.K.,  
Tokyo**

Takanawa Park Tower 12F & 17F  
3-20-14 Higashi-Gotanda  
Shinagawa-ku  
Tokyo 141-0022  
Tel. (+81) 3-5449 6411  
Fax (+81) 3-5449 6401

**Korea (Republic of) (ROC)**
**Siemens Ltd.**

Asia Tower, 10th Floor  
726 Yeoksam-dong, Kang-nam  
Ku  
PO Box 3001  
Seoul 135-080  
Tel. (+82) 2-527 77 00  
Fax (+82) 2-527 77 79

**Malaysia (MAL)**
**Infineon Technologies AG Sdn  
Bhd**

Bayan Lepas Free Industrial  
Zone 1  
11900 Penang  
Tel. (+60) 4-644 9975  
Fax (+60) 4-641 4872

**Netherlands (NL)**
**Siemens Electronic Components**

Benelux  
Postbus 16068  
NL-2500 BB Den Haag  
Tel. (+31) 70-333 20 65  
Fax (+31) 70-333 28 15

**New Zealand (NZ)**
**Siemens Components**

300 Great South Road  
PO Box 17122  
Greenland, Auckland  
Tel. (+64) 9-520 30 33  
Fax (+64) 9-520 15 56

**Norway (N)**
**Siemens Components Scandinavia**

Østre Aker vei 24  
Postboks 10, Veitvet  
NO-0518 Oslo  
Tel. (+47) 22-63 30 00  
Fax (+47) 22-66 49 13

**Pakistan (PK)**
**Siemens Pakistan Engineering Co. Ltd.**

PO Box 1129, Islamabad 44000  
23 West Jinnah Ave  
Islamabad  
Tel. (+92) 51-21 22 00  
Fax (+35) 51-21 16 10

**Poland (PL)**
**Siemens Sp. z o.o.**

ul. Zupnicza 11  
PL-03-821 Warszawa  
Tel. (+48) 22-870 91 50  
Fax (+48) 22-870 91 59

**Portugal (P)**
**Siemens S.A.**

**An Componentes Electronicos**  
R. Irmaos Siemens, 1  
Alfragide  
P-2720-093 Amadora  
Tel. (+35) 1-417 85 90  
Fax (+35) 1-417 80 83

**Russia (RUS)**
**Siemens AG**

ul. Dubininskaya, 98A  
RUS-113 093 Moskva  
Tel (+7) 095 737 -1435, -1436  
Fax (+7) 095 737 14 39

**INTECH electronics**

Smolnaya ul. 24/1203  
RUS-125445 Moscow  
Tel (+7) 95-451-9737,-8608  
Fax (+7) 95-451-9737,-8608

**Singapore (SGP)**
**Infineon Technologies Asia Pacific, Pte. Ltd.**

168 Kallang Way  
Singapore 349 253  
Tel. (+65) 840 06 10  
Fax (+65) 742 62 39

2001-04-30 @ 15:16

**South Africa (ZA)****Siemens Ltd.****Components Division**

P.O.B. 3438

Halfway House 1685

Tel. (+27) 11-652-2000, -2700

Fax (+27) 11-652 20 42

**Spain (E)****Siemens S.A.****Division Componentes**

Ronda de Europa, 5

28760 Tres Cantos-Madrid

Tel. (+34) 1-514 71 51

Fax (+34) 1-514 70 13

**Sweden (S)****Siemens Components Scandina-  
navia**

Österögatan 1

Box 46

SE-164 93 Kista

Tel. (+46) 8-703 35 00

Fax (+46) 8-703 35 01

**Switzerland (CH)****Siemens Schweiz AG****Geschäftsbereich Bauele-  
mente**

Freilagerstraße 40

CH-8047 Zürich

Tel. (+41) 1-495 30 65

Fax (+41) 1-495 50 50

**Taiwan (Republic of China)  
(RC)****Infineon Technologies Asia Pa-  
cific Pte Ltd.**10F, No. 136, Nan King East  
Road, Sec. 3, Taipei

Tel. (+886) 2-2773 66 06

Fax (+886) 2-2771 20 76

**United Kingdom (GB)****Infineon Technologies**

Siemens House

Oldbury

GB-Bracknell, Berkshire RG12

8FZ

Tel. (+44) 1344-39 66 18

Fax (+44) 1344-39 66 32

**United States of America  
(USA)****Infineon Technologies Corpo-  
ration**

1730 North First St

San Jose, CA 95112

Tel. (+1) 408-501 60 00

Fax (+1) 408-501 24 24

<http://www.infineon.com>**Siemens Components, Inc.  
Optoelectronics Division**

19000 Homestead Road

Cupertino, CA 95014

Tel. (+1) 408-257 79 10

Fax (+1) 408-725 34 39

**Siemens Components, Inc.**

Special Products Division

186 Wood Avenue South

Iselin, NJ 08830-2770

Tel. (+1) 732-906 43 00

Fax (+1) 732-632 28 30

2001-04-30 @ 15:16

---

# Total Quality Management

---

Quality takes on an all-encompassing significance at Infineon Technologies Corp. 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 Infineon's 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.

2001-04-30 @ 15:16

# Index

## A

|                         |                                 |
|-------------------------|---------------------------------|
| absolute address        | 16                              |
| absolute difference     | 119                             |
| absolute value          | 119                             |
| access permissions      | 110                             |
| addition instructions   | 117                             |
| address                 |                                 |
| absolute                | 21                              |
| arithmetic              | 132                             |
| base, vector table      | 36                              |
| code                    | 21                              |
| comparison              | 132                             |
| effective               | 20, 30                          |
| general-purpose         | 34                              |
| halfword                | 36                              |
| indexed                 | 21                              |
| load effective          | 21                              |
| physical memory         | 31                              |
| ranges                  | 31                              |
| return                  | 26                              |
| address register        | 5, 17, 20, 21, 22, 25, 132, 134 |
| addressing modes        | 5                               |
| hardware                | 21                              |
| synthesized             | 21                              |
| alignment trap          | 19                              |
| ALN                     | 80                              |
| Architecture            | 1                               |
| architecture            |                                 |
| overview                | 2                               |
| support                 | 21                              |
| arithmetic instructions | 116, 123                        |
| arithmetic operations   | 132                             |
| array                   | 5                               |
| address                 | 20                              |
| index                   | 20                              |
| asynchronous trap       | 76                              |
| automatic switch        | 34                              |

## B

|         |    |
|---------|----|
| base    |    |
| address | 20 |

|                                |        |
|--------------------------------|--------|
| pointers                       | 132    |
| register                       | 21     |
| base+offset addressing         | 17     |
| basic data types               | 136    |
| Begin ISR                      | 48     |
| bit                            |        |
| enable/disable                 | 34     |
| field                          | 138    |
| operations                     | 130    |
| reverse addressing             | 19, 20 |
| string                         | 12     |
| bit-field extract instructions | 121    |
| bit-reversed                   |        |
| order                          | 19     |
| BIV register                   | 66     |
| boolean                        | 12     |
| Boolean equations              | 131    |
| branch instructions            | 133    |
| buffer                         |        |
| size                           | 21     |
| start                          | 19     |
| byte                           |        |
| indices                        | 21     |
| offset                         | 18     |
| ordering                       | 14     |

## C

|                         |          |
|-------------------------|----------|
| call depth counter      | 28, 50   |
| CALL instruction        | 48, 51   |
| CDO                     | 81       |
| CDU                     | 81       |
| circular                |          |
| addressing              | 17, 18   |
| buffer                  | 5, 18    |
| circular buffer         |          |
| restrictions            | 19       |
| circular buffers        | 17       |
| code                    |          |
| address                 | 21       |
| range                   | 42       |
| compare instruction     | 126, 129 |
| comparison instructions | 131, 132 |

2001-04-30 @ 15:16

|  |                       |  |                |
|--|-----------------------|--|----------------|
| conditiona                             |                       | Debug                                      |                |
| I jump instructions . . . . .          | 134                   | Control Unit . . . . .                     | 9              |
| conditional                            |                       | System . . . . .                           | 9              |
| branch instructions . . . . .          | 134                   | debug                                      |                |
| expressions . . . . .                  | 127                   | control unit . . . . .                     | 43             |
| instructions . . . . .                 | 120                   | register . . . . .                         | 43             |
| context                                |                       | signals . . . . .                          | 40             |
| current . . . . .                      | 50                    | depletion trap entry . . . . .             | 49             |
| definition of . . . . .                | 5                     | DSE . . . . .                              | 82             |
| instructions . . . . .                 | 138                   | <b>E</b>                                   |                |
| list . . . . .                         | 54                    | exception instruction . . . . .            | 141            |
| loading . . . . .                      | 139                   | extended-size registers . . . . .          | 25             |
| lower . . . . .                        | 5, 27, 30, 46, 47, 55 | <b>F</b>                                   |                |
| restore . . . . .                      | 51                    | Fast Context Switching . . . . .           | 7              |
| restore operation . . . . .            | 54                    | FCD . . . . .                              | 80             |
| restoring . . . . .                    | 139                   | FCU . . . . .                              | 81             |
| save . . . . .                         | 51                    | FCX Offset Address Field . . . . .         | 31             |
| save area . . . . .                    | 6, 30, 46             | FCX pointer . . . . .                      | 31             |
| saving . . . . .                       | 139                   | FCX Segment Address Field . . . . .        | 31             |
| storing . . . . .                      | 139                   | Feature Overview . . . . .                 | 3              |
| switching . . . . .                    | 2, 50                 | FFT algorithms . . . . .                   | 19             |
| upper . . . . .                        | 5, 27, 30, 46, 47     | filter calculations . . . . .              | 17             |
| core                                   |                       | floating-point registers . . . . .         | 27             |
| accesses . . . . .                     | 140                   | free context list . . . . .                | 49, 50, 51, 53 |
| register access . . . . .              | 25                    | Function Call . . . . .                    | 48, 51         |
| special function register . . . . .    | 3, 24                 | function call . . . . .                    | 51             |
| CSA list underflow . . . . .           | 33, 49                | <b>G</b>                                   |                |
| CSU . . . . .                          | 82                    | general purpose registers . . . . .        | 3, 24, 51      |
| CTYP . . . . .                         | 82                    | global                                     |                |
| Current CPU Priority Number . . . . .  | 34                    | data . . . . .                             | 16             |
| current task context . . . . .         | 50                    | registers . . . . .                        | 28             |
| <b>D</b>                               |                       | Global Register Write Permission . . . . . | 28, 29         |
| DAE . . . . .                          | 80, 82                | GRWP . . . . .                             | 79             |
| Data                                   |                       | guard bits . . . . .                       | 124            |
| Formats . . . . .                      | 13                    | <b>H</b>                                   |                |
| General Purpose Registers . . . . .    | 25                    | Hard Reset . . . . .                       | 9              |
| Memory Unit Control Register . . . . . | 37                    | <b>I</b>                                   |                |
| Protection Mode . . . . .              | 40                    | I/O Privilege . . . . .                    | 28, 29         |
| Types . . . . .                        | 4                     | index                                      |                |
| data                                   |                       | algorithm . . . . .                        | 18             |
| elements . . . . .                     | 5                     | bit-reverse . . . . .                      | 21             |
| memory . . . . .                       | 21, 140               | modifier . . . . .                         | 20             |
| range . . . . .                        | 40                    | indexed                                    |                |
| register . . . . .                     | 5, 25, 132            | addressing . . . . .                       | 21             |
| size . . . . .                         | 19                    | arrays . . . . .                           | 21             |
| structures . . . . .                   | 5                     | indexes                                    |                |
| values . . . . .                       | 17                    |  |                |



|                                     |                   |                                   |                   |
|-------------------------------------|-------------------|-----------------------------------|-------------------|
| table . . . . .                     | 25                | Load . . . . .                    | 47                |
| Instruction                         |                   | load                              |                   |
| Formats . . . . .                   | 5                 | bit . . . . .                     | 137               |
| Set Architecture . . . . .          | 2                 | effective address . . . . .       | 21                |
| instruction                         |                   | instructions . . . . .            | 136               |
| fetching . . . . .                  | 2                 | word . . . . .                    | 18                |
| jump and link . . . . .             | 22                | local                             |                   |
| load double word . . . . .          | 19                | variables . . . . .               | 17                |
| load word . . . . .                 | 19                | logic operations . . . . .        | 120               |
| on-chip . . . . .                   | 21                | loop instructions . . . . .       | 135               |
| set architecture . . . . .          | 2, 116            | lower address registers . . . . . | 4                 |
| word . . . . .                      | 18                | lower context . . . . .           | 5, 27, 30, 46, 47 |
| integer arithmetic . . . . .        | 117               | <b>M</b>                          |                   |
| integers . . . . .                  | 12                | MEM . . . . .                     | 80                |
| internal                            |                   | memory                            |                   |
| buffer . . . . .                    | 54                | access . . . . .                  | 18                |
| Interrupt                           |                   | memory access                     |                   |
| Control Register . . . . .          | 34                | boundary crossing . . . . .       | 113               |
| Service Request . . . . .           | 7                 | legality . . . . .                | 9                 |
| Service Routine . . . . .           | 5                 | memory protection . . . . .       | 37, 38            |
| Stack Pointer . . . . .             | 33                | model . . . . .                   | 105               |
| interrupt . . . . .                 | 34, 50            | registers . . . . .               | 27                |
| enable . . . . .                    | 50                | MMU . . . . .                     | 78                |
| handler . . . . .                   | 47, 50            | mode table entry . . . . .        | 40, 42            |
| latency . . . . .                   | 2                 | move . . . . .                    | 117               |
| pending . . . . .                   | 50                | MPN . . . . .                     | 79                |
| service routine . . . . .           | 5, 34, 50         | MPP . . . . .                     | 79                |
| system disabling . . . . .          | 141               | MPR . . . . .                     | 79                |
| system enabling . . . . .           | 141               | MPW . . . . .                     | 79                |
| vector table . . . . .              | 34, 35            | MPX . . . . .                     | 79                |
| interrupt priority number . . . . . | 66                | multiply                          |                   |
| interrupt service routine . . . . . | 69                | accumulate instructions . . . . . | 116               |
| interrupt vector table . . . . .    | 66                | add . . . . .                     | 118               |
| interrups . . . . .                 | 26                | <b>N</b>                          |                   |
| Interrupt Stack . . . . .           | 28                | NEST . . . . .                    | 82                |
| IOPC . . . . .                      | 79                | nesting . . . . .                 | 7, 75, 111        |
| <b>J</b>                            |                   | NMI . . . . .                     | 83                |
| jump instruction . . . . .          | 133               | non-maskable interrupt . . . . .  | 75                |
| <b>L</b>                            |                   | no-operation . . . . .            | 141               |
| LCX . . . . .                       | 80                | normalization . . . . .           | 120               |
| Link                                |                   | <b>O</b>                          |                   |
| Segment . . . . .                   | 46                | on-chip                           |                   |
| link                                |                   | instruction . . . . .             | 21                |
| instruction . . . . .               | 133               | OPD . . . . .                     | 80                |
| Link Index . . . . .                | 46                | ORed . . . . .                    | 35                |
| link word . . . . .                 | 6, 46, 49, 52, 54 | ORing . . . . .                   | 36                |
| little-endian . . . . .             | 14                |                                   |                   |

2001-04-30 @ 15:16

|                  |         |
|------------------|---------|
| overflow .....   | 75, 125 |
| conditions ..... | 116     |
| OVF .....        | 83      |

**P**

|                                    |                      |
|------------------------------------|----------------------|
| packed                             |                      |
| arithmetic instructions .....      | 125                  |
| bytes .....                        | 129                  |
| instructions .....                 | 116                  |
| values .....                       | 126                  |
| PC, Program Counter .....          | 5                    |
| relative addressing .....          | 21                   |
| return .....                       | 77                   |
| pending                            |                      |
| interrupt .....                    | 50                   |
| interrupt priority number .....    | 34, 50               |
| peripheral registers .....         | 5, 16                |
| permission levels .....            | 8                    |
| pointer                            |                      |
| function .....                     | 32                   |
| interrupt vector table .....       | 34                   |
| trap vector table .....            | 34                   |
| post-decrement .....               | 17                   |
| post-increment addressing .....    | 17                   |
| Power-On Reset .....               | 9                    |
| pre-increment addressing .....     | 17                   |
| previous context list .....        | 48, 49, 51, 53       |
| Previous Context Pointer .....     | 30, 32               |
| Previous CPU Priority Number ..... | 30                   |
| Previous Interrupt Enable .....    | 30                   |
| priority                           |                      |
| number .....                       | 30                   |
| priority number                    |                      |
| CPU .....                          | 50                   |
| interrupt .....                    | 66                   |
| pending interrupt .....            | 50                   |
| previous CPU .....                 | 50                   |
| PRIV .....                         | 78                   |
| program                            |                      |
| counter .....                      | 3, 26, 27            |
| memory .....                       | 21                   |
| status information .....           | 3, 5                 |
| Program State Registers .....      | 3, 4                 |
| Programming Model .....            | 12                   |
| protection                         |                      |
| register set .....                 | 9, 27, 105, 110, 112 |
| protection system .....            | 104                  |
| hardware operation .....           | 104                  |

|                        |     |
|------------------------|-----|
| major components ..... | 104 |
| purpose of .....       | 7   |
| PSE .....              | 82  |

**R**

|                                    |        |
|------------------------------------|--------|
| range                              |        |
| checking .....                     | 39     |
| table entry .....                  | 110    |
| Real Time Operating System .....   | 46     |
| record elements .....              | 17     |
| records .....                      | 5      |
| redundant zeros .....              | 120    |
| register .....                     | 40     |
| Code Protection Mode .....         | 42     |
| code segment protection .....      | 39     |
| data .....                         | 25     |
| data memory unit control .....     | 37     |
| data segment protection .....      | 39     |
| debug .....                        | 43     |
| global .....                       | 47     |
| maintenance of contents .....      | 47     |
| memory protection .....            | 38     |
| mode .....                         | 39     |
| non-volatile .....                 | 51     |
| scaled data .....                  | 21     |
| scratch .....                      | 51     |
| segment protection .....           | 39     |
| static hardware .....              | 47     |
| register sets .....                | 38     |
| registers                          |        |
| BIV .....                          | 66     |
| context management .....           | 30     |
| extended-size .....                | 25     |
| floating point .....               | 27     |
| general purpose .....              | 24, 51 |
| global .....                       | 28     |
| memory protection .....            | 27     |
| system configuration control ..... | 37     |
| system global .....                | 26     |
| Reset Overflow Flags .....         | 25     |
| Reset System .....                 | 9      |
| Restore .....                      | 47     |
| return                             |        |
| address .....                      | 26     |
| address register .....             | 4      |
| instruction .....                  | 141    |
| rounding .....                     | 124    |

## S

|   |         |
|---|---------|
| saturate instructions . . . . .                 | 120     |
| saturation . . . . .                            | 125     |
| Save Lower Context. . . . .                     | 48      |
| scale factor . . . . .                          | 21      |
| scaled  |         |
| data register . . . . .                         | 21      |
| offset instruction . . . . .                    | 137     |
| scaling . . . . .                               | 124     |
| service   |         |
| request . . . . .                               | 7       |
| shift instructions . . . . .                    | 121     |
| signed fraction . . . . .                       | 12      |
| Soft Reset. . . . .                             | 9       |
| software  |         |
| managed tasks . . . . .                         | 46, 50  |
| SOVF . . . . .                                  | 83      |
| stack   |         |
| management. . . . .                             | 33      |
| pointer. . . . .                                | 4       |
| Stack Pointer . . . . .                         | 26      |
| stacks . . . . .                                | 5       |
| state information. . . . .                      | 27, 29  |
| static data. . . . .                            | 17      |
| status flags . . . . .                          | 116     |
| sticky overflow . . . . .                       | 75      |
| store   |         |
| bit . . . . .                                   | 138     |
| instructions . . . . .                          | 136     |
| Store Lower Context. . . . .                    | 48      |
| Store Upper Context. . . . .                    | 48      |
| supervisor mode. . . . .                        | 110     |
| synchronization primitives . . . . .            | 140     |
| synchronous trap . . . . .                      | 76      |
| SYS . . . . .                                   | 83      |
| system  |         |
| call . . . . .                                  | 75, 139 |
| global registers . . . . .                      | 4, 26   |
| instructions . . . . .                          | 139     |
| System Configuration Control Register . . . . . | 37      |

## T

|                            |    |
|----------------------------|----|
| table indexes . . . . .    | 25 |
| task switching. . . . .    | 47 |
| tasks                      |    |
| software managed . . . . . | 50 |
| software-managed . . . . . | 46 |
| user. . . . .              | 5  |

|                                 |        |
|---------------------------------|--------|
| TIN . . . . .                   | 74     |
| Trap . . . . .                  | 78     |
| trap. . . . .                   | 30, 74 |
| default state . . . . .         | 78     |
| depletion entry . . . . .       | 49     |
| handler vector . . . . .        | 77     |
| identification number . . . . . | 36     |
| system . . . . .                | 7      |
| vector table. . . . .           | 36, 77 |
| vector table pointe . . . . .   | 34     |
| traps. . . . .                  | 26     |
| asynchronous. . . . .           | 76     |
| synchronous. . . . .            | 76     |
| TriCore supported . . . . .     | 74     |
| TriCore. . . . .                | 74     |

## U

|   |                   |
|---|-------------------|
| unconditional branch instructions . . . . . | 133               |
| UOPC . . . . .                              | 80                |
| upper address registers . . . . .           | 4                 |
| upper context . . . . .                     | 5, 27, 30, 46, 47 |
| user tasks . . . . .                        | 5                 |

## V

|                        |    |
|------------------------|----|
| VAF . . . . .          | 78 |
| VAP . . . . .          | 78 |
| vector table . . . . . | 36 |

## W

|                                |    |
|--------------------------------|----|
| Wake-up Reset . . . . .        | 9  |
| wrap around behavior . . . . . | 18 |