

See Also Addendum for TriCore Arch Manual, Vol2, V1.3.8, pages 493/494

# TriCore<sup>®</sup> 1

## 32-bit Unified Processor Core

### Volume 2

### Instruction Set

### V1.3 & V1.3.1 Architecture

# 32bit

## Microcontrollers



Never stop thinking

**Edition 2008-01**

**Published by  
Infineon Technologies AG  
81726 Munich, Germany**

**© 2008 Infineon Technologies AG  
All Rights Reserved.**

#### **Legal Disclaimer**

The information given in this document shall in no event be regarded as a guarantee of conditions or characteristics. With respect to any examples or hints given herein, any typical values stated herein and/or any information regarding the application of the device, Infineon Technologies hereby disclaims any and all warranties and liabilities of any kind, including without limitation warranties of non-infringement of intellectual property rights of any third party.

#### **Information**

For further information on technology, delivery terms and conditions and prices please contact your nearest Infineon Technologies Office ([www.infineon.com](http://www.infineon.com)).

#### **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 be used in life-support devices or systems only 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.

See Also Addendum for TriCore Arch Manual, Vol2, V1.3.8, pages 493/494

# TriCore<sup>®</sup> 1

## 32-bit Unified Processor Core

### Volume 2

### Instruction Set

### V1.3 & V1.3.1 Architecture

**Microcontrollers**



Never stop thinking

Previous Version - none

Version	Subjects (major changes since last revision)
V1.3.8	The Instruction Set Overview chapter was missing from Volume 2 of the initial release of this document (v1.3.8), dated 2007-11. This version (dated 2008-01) supercedes that release. There are no other changes to the document (vol1 or vol2) aside from the inclusion of the Instruction Set Overview chapter.

TriCore® is a Registered trademark of Infineon Technologies AG.

**We Listen to Your Comments**

Is there any information in this document that you feel is wrong, unclear or missing? Your feedback will help us to continuously improve the quality of our documentation. Please send feedback (including a reference to this document) to:

**[ipdoc@infineon.com](mailto:ipdoc@infineon.com)**



Table of Contents		Page
<b>1</b>	<b>Instruction Set Overview</b> .....	<b>1</b>
1.1	Integer Arithmetic .....	1
1.1.1	Move .....	1
1.1.2	Addition and Subtraction .....	1
1.1.3	Multiply and Multiply-Add .....	2
1.1.4	Division .....	2
1.1.5	Absolute Value, Absolute Difference .....	3
1.1.6	Min, Max, Saturate .....	3
1.1.7	Conditional Arithmetic Instructions .....	3
1.1.8	Logical .....	4
1.1.9	Count Leading Zeroes, Ones, and Signs .....	4
1.1.10	Shift .....	5
1.1.11	Bit-Field Extract and Insert .....	5
1.2	Packed Arithmetic .....	8
1.3	PSW Status Flags and Arithmetic Instructions .....	10
1.3.1	Usage .....	10
1.3.2	Saturation .....	10
1.4	DSP Arithmetic .....	10
1.4.1	Scaling .....	11
1.4.2	Special Case = $-1 * -1$ .....	11
1.4.3	Guard bits .....	11
1.4.4	Rounding .....	12
1.4.5	Overflow and Saturation .....	12
1.4.6	Sticky Advance Overflow and Block Scaling in FFT .....	12
1.4.7	Multiply and MAC .....	12
1.4.8	Packed Multiply and Packed MAC .....	13
1.5	Compare Instructions .....	14
1.5.1	Simple Compare .....	14
1.5.2	Accumulating Compare .....	15
1.5.3	Compare with Shift .....	16
1.5.4	Packed Compare .....	17
1.6	Bit Operations .....	18
1.6.1	Simple Bit Operations .....	18
1.6.2	Accumulating Bit Operations .....	19
1.6.3	Shifting Bit Operations .....	20
1.7	Address Arithmetic .....	21
1.8	Address Comparison .....	22
1.9	Branch Instructions .....	23
1.9.1	Unconditional Branch .....	23
1.9.2	Conditional Branch .....	24
1.9.3	Loop Instructions .....	25
1.10	Load and Store Instructions .....	27

1.10.1	Load/Store Basic Data Types	27
1.10.2	Load Bit	28
1.10.3	Store Bit and Bit Field	29
1.11	Context Related Instructions	30
1.11.1	Lower Context Saving and Restoring	30
1.11.2	Context Loading and Storing	30
1.12	System Instructions	31
1.12.1	System Call	31
1.12.2	Synchronization Primitives (DYSNC and ISYNC)	31
1.12.3	Access to the Core Special Function Registers (CSFRs)	32
1.12.4	Enabling and Disabling the Interrupt System	33
1.12.5	Return (RET) and Return From Exception (RFE) Instructions	34
1.12.6	Trap Instructions	34
1.12.7	No-operation (NOP)	34
1.13	Coprocessor Instructions	34
1.14	16-bit Instructions	35
<b>2</b>	<b>Instruction Set Information</b>	<b>1</b>
2.1	Instruction Syntax	2
2.1.1	Operand Definitions	2
2.1.2	Instruction Mnemonic	2
2.1.3	Operation Modifiers	3
2.1.4	Data Type Modifiers	4
2.2	Opcode Formats	5
2.2.1	16-bit Opcode Formats	5
2.2.2	32-bit Opcode Formats	5
2.2.3	Opcode Field Description	7
2.3	Instruction Operation Syntax	8
2.3.1	RTL Functions	11
2.3.2	Cache RTL Functions	12
2.3.3	Floating Point Operation Syntax	14
2.4	Coprocessor Instructions	17
2.5	PSW Status Flags (User Status Bits)	18
2.6	List of OS and I/O Privileged Instructions	19
<b>3</b>	<b>Instruction Set</b>	<b>21</b>
3.1	CPU Instructions	21
3.2	FPU Instructions	466
3.3	MMU Instructions	496
<b>4</b>	<b>Summary Tables of LS and IP Instructions</b>	<b>507</b>

## Preface

TriCore® is a unified, 32-bit microcontroller-DSP, single-core architecture optimized for real-time embedded systems.

This document has been written for system developers and programmers, and hardware and software engineers.

- Volume 1 provides a detailed description of the Core Architecture and system interaction.
- Volume 2 (this volume) gives a complete description of the TriCore Instruction Set including optional extensions for the Memory Management Unit (MMU) and Floating Point Unit (FPU).

It is important to note that this document describes the TriCore architecture, not an implementation. An implementation may have features and resources which are not part of the Core Architecture. The documentation for that implementation will describe all implementation specific features.

When working with a specific TriCore based product always refer to the appropriate supporting documentation.

### TriCore versions

There have been several versions of the TriCore Architecture implemented in production devices. This manual documents the following architectures: TriCore 1.3, TriCore 1.3.1.

- Unless defined otherwise in the text, or in the margin, all descriptions are common to all TriCore versions.
- If the text refers to TriCore 1 specifically, then it is relevant to both the TriCore 1.3 and the TriCore 1.3.1 architecture unless stated otherwise.
- Information unique to the TriCore 1.3.1 architecture is always labelled.

### Additional Information

For information and links to documentation for Infineon products that use TriCore, visit:  
<http://www.infineon.com/32-bit-microcontrollers>

## Text Conventions

This document uses the following text conventions:

- The default radix is decimal.
  - Hexadecimal constants are suffixed with a subscript letter 'H', as in:  $FFC_H$ .
  - Binary constants are suffixed with a subscript letter 'B', as in:  $111_B$ .
- Register reset values are not generally architecturally defined, but require setting on startup in a given implementation of the architecture. Only those reset values that are architecturally defined are shown in this document. Where no value is shown, the reset value is not defined. Refer to the documentation for a specific TriCore implementation.
- Bit field and bits in registers are in general referenced as 'Register name.Bit field', for example PSW.IS. The Interrupt Stack Control bit of the PSW register.
- Units are abbreviated as follows:
  - MHz = Megahertz.
  - kBaud, kBit = 1000 characters/bits per second.
  - MBaud, MBit = 1,000,000 characters per second.
  - KByte = 1024 bytes.
  - MByte = 1048576 bytes of memory.
  - GByte = 1,024 megabytes.
- Data format quantities referenced are as follows:
  - Byte = 8-bit quantity.
  - Half-word = 16-bit quantity.
  - Word = 32-bit quantity.
  - Double-word = 64-bit quantity.
- Pins using negative logic are indicated by an overbar:  $\overline{\text{BRKOUT}}$ .

In tables where register bit fields are defined, the conventions shown in the following table are used in this document.

**Table 1 Bit Type Abbreviations**

Abbreviation	Description
r	Read-only. The bit or bit field can only be read.
w	Write-only. The bit or bit field can only be written.
rw	The bit or bit field can be read and written.
h	The bit or bit field can be modified by hardware (such as a status bit). 'h' can be combined with 'rw' or 'r' bits to form 'rwh' or 'rh' bits.
-	Reserved Field. Read value is undefined, should be written with 0.

*Note: In register layout tables, a 'Reserved Field' is indicated with '-' in the Field and Type column.*



## 1 Instruction Set Overview

This chapter provides an overview of the TriCore® Instruction Set Architecture (ISA). The basic properties and use of each instruction type are described, together with a description of the selection and use of the 16-bit (short) instructions.

### 1.1 Integer Arithmetic

This section covers the following topics:

- [Move, page 1-1.](#)
- [Addition and Subtraction, page 1-1.](#)
- [Multiply and Multiply-Add, page 1-2.](#)
- [Division, page 1-2.](#)
- [Absolute Value, Absolute Difference, page 1-3.](#)
- [Min, Max, Saturate, page 1-3.](#)
- [Conditional Arithmetic Instructions, page 1-3.](#)
- [Logical, page 1-4.](#)
- [Count Leading Zeroes, Ones, and Signs, page 1-4.](#)
- [Shift, page 1-5.](#)
- [Bit-Field Extract and Insert, page 1-5.](#)

#### 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 High-word) instruction loads a 16-bit constant into the most-significant 16 bits of the register and zero fills the least-significant 16-bits. This is useful for loading a left-justified constant fraction.

Loading a 32-bit constant is achieved by using a MOVH instruction followed by an ADDI (Add Immediate), or a MOV.U followed by ADDIH (Add Immediate High-word).

#### 1.1.2 Addition and Subtraction

The addition instructions have three versions:

- ADD (No saturation).
- ADDS (Signed saturation).
- ADDS.U (Unsigned saturation).

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

## Instruction Set Overview

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.

It is often necessary to add 16-bit 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 is carried out 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.

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

### 1.1.3 Multiply and Multiply-Add

For the multiplication of 32-bit integers, the available mnemonics are:

- MUL (Multiply Signed).
- MULS (Multiply Signed with Saturation).
- MULS.U (Multiply Unsigned with Saturation).

These translate to machine instructions producing either 32-bit or 64-bit results, depending on whether the destination operand encoded in the assembly instruction is a single data register D[n] (where  $n = 0, 1, \dots, 15$ ), or an extended data register E[n] (where  $n = 0, 2, \dots, 14$ ).

In those cases where the number of bits in the destination is 32-bit, the result is taken from the lower bits of the product. This corresponds to the standard 'C' multiplication of two integers.

The MAC instructions (Multiplication with Accumulation) follow the instruction forms for multiplication; MADD, MADDS, MADD.U, MADDS.U, and MSUB, MSUBS, MSUB.U, MSUBS.U.

In all cases a third source operand register is specified, which provides the accumulator to which the multiplier results are added.

### 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 interrupt latency down. The divide step sequence allows the divide time to be proportional to the number of significant quotient bits expected.

## Instruction Set Overview

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. It simultaneously shifts in that many copies of the quotient sign bit to the low-order bit positions. 4, 2 or 1 Divide-Step instructions (DVSTEP or DVSTEP.U) then follow. 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.

### 1.1.5 Absolute Value, Absolute Difference

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

### 1.1.6 Min, Max, Saturate

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

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

### 1.1.7 Conditional Arithmetic Instructions

- CADD (Conditional Add) and CADDN (Conditional Add-Not).
- CSUB (Conditional Subtract) and CSUBN (Conditional Subtract-Not).
- SEL (Select) and SELN (Select-Not).

The conditional instructions 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. For the CADDN and CSUBN instructions it is performed if the condition is FALSE.

## Instruction Set Overview

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

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

If one of the two source operands in a SEL instruction is the same as the destination operand, then the SEL instruction implements a simple conditional move. This occurs 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.

### 1.1.8 Logical

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

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

To provide efficient support for normalization of numerical results, prioritization, and certain graphics operations, three Count Leading instructions are provided:

- CLZ (Count Leading Zeros).
- CLO (Count Leading Ones).
- CLS (Count Leading Signs).

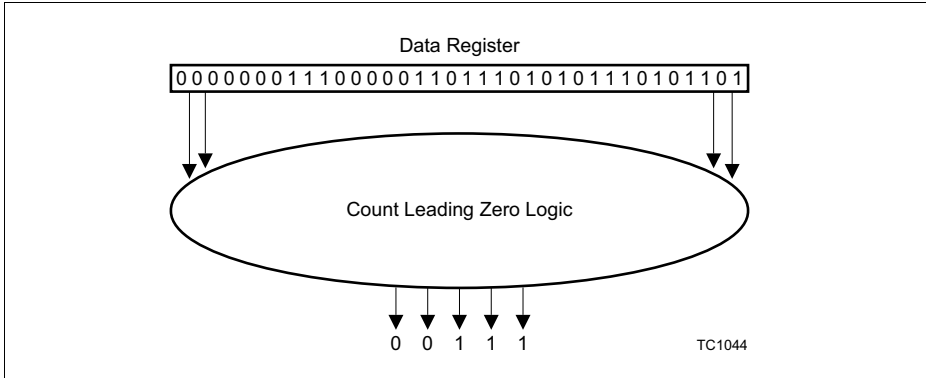
These instructions are used to determine the amount of left shifting necessary to remove redundant zeros, ones, or signs.

*Note: The CLS instruction returns the number of leading redundant signs, which is the number of leading signs minus one.*

The following special cases are defined:

- CLZ(0) = 32, CLO(-1) = 32.
- CLS(0) = CLS(-1) = 31.

For example, CLZ returns the number of consecutive zeros starting from the most significant bit of the value in the source data register. In the example shown in [Figure 1](#), there are seven 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 1 Operation of the 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).

### 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 zeros for both right and left shifts. The arithmetic shift instruction SHA, shifts in sign bits for right shifts and zeros 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.

### 1.1.11 Bit-Field Extract and Insert

The TriCore architecture supports three bit-field extract instructions.

- EXTR (Extract bit field).
- EXTR.U (Extract bit field unsigned).
- DEXTR (Extract from Double Register).

The INSERT instruction is described on [page 1-7](#).

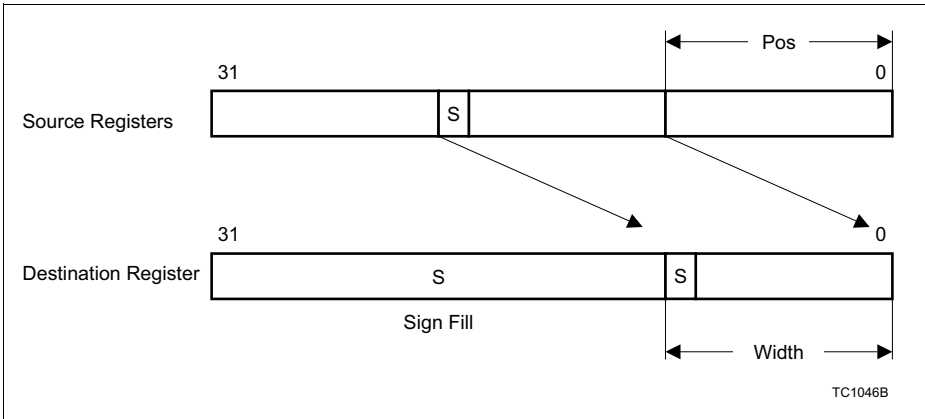
### EXTR and EXTR.U

The EXTR and EXTR.U instructions extract *width* consecutive bits from the source, beginning with the bit number specified by the *pos* (position) operand. The *width* and *pos* can be specified by two immediate values, by an immediate value and a data register, or by a data register pair.

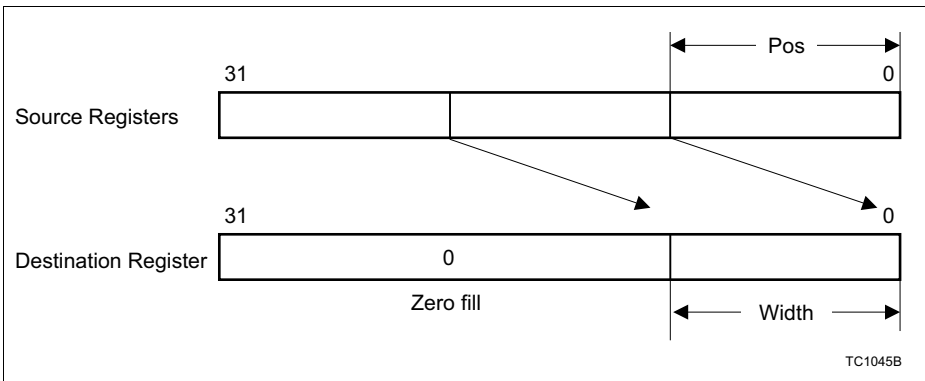
Instruction Set Overview

The EXTR instruction fills the most-significant bits of the result by sign-extending the bit field extracted (duplicating the most-significant bit of the bit field). See [Figure 2](#).

EXTR.U zero-fills the most significant (32-w) bits of the result. See [Figure 3](#).



**Figure 2** Operation of the EXTR Instruction



**Figure 3** Operation of the EXTR.U Instruction

**DEXTR**

The DEXTR instruction 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.

See Also Addendum for TriCore Arch Manual, Vol2, V1.3.8, pages 493/494

Instruction Set Overview

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).

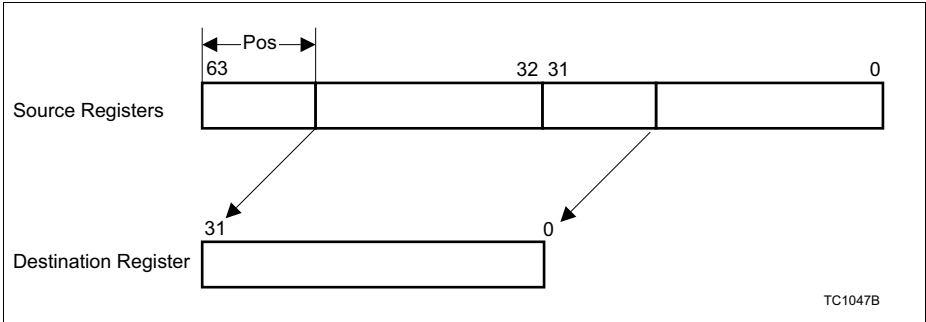


Figure 4 Operation of the DEXTR Instruction

INSERT

The INSERT instruction takes the *width* 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 *width* and *pos* can be specified by two immediate values, by an immediate value and a data register, or by a data register pair.

There is also an alternative form of INSERT that allows a zero-extended 4-bit constant to be the value which is inserted.

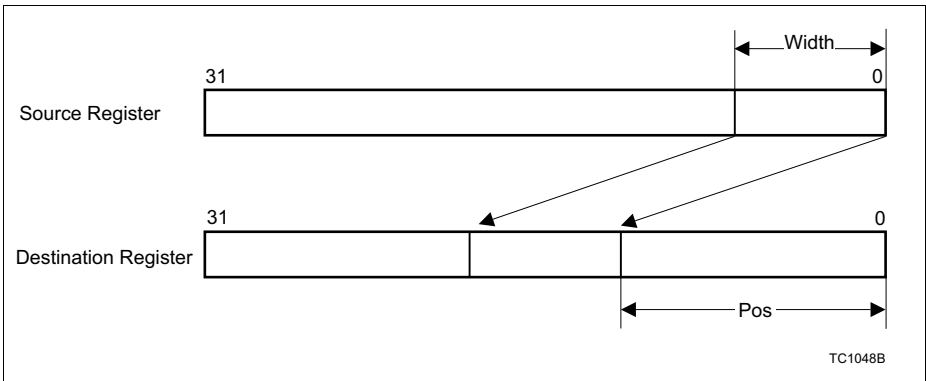


Figure 5 Operation of the INSERT Instruction

See Also Addendum for TriCore Arch Manual, Vol2, V1.3.8, pages 493/494

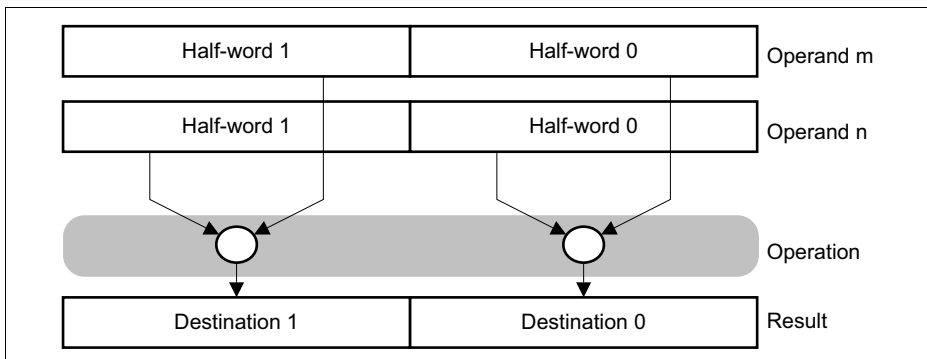
## 1.2 Packed Arithmetic

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:

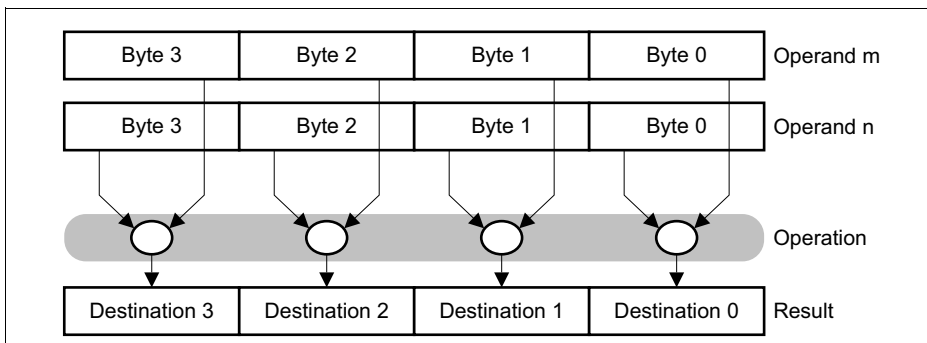
- Packed Half-word Data Format
- Packed Byte Data Format

The Packed Half-word Data format divides the 32-bit word into two, 16-bit (half-word) values. Instructions which operate on data in this way are denoted in the instruction mnemonic by the .H and .HU modifiers.



**Figure 6 Packed Half-word Data Format**

The Packed Byte Data format divides the 32-bit word into four, 8-bit values. Instructions which operate on the data in this way are denoted by the .B and .BU data type modifiers.



**Figure 7 Packed Byte Data Format**



## Instruction Set Overview

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, and so on.

Addition is performed on individual packed bytes or half-words using the ADD.B and ADD.H instructions. The saturating variation (ADDS.H) only exists for half-words.

The ADD.H instruction ignores overflow or underflow within individual half-words. ADDS.H will saturate individual half-words 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.HU instruction. Similarly, all packed addition operations have an equivalent subtraction.

Besides addition and subtraction, arithmetic on packed data includes absolute value, absolute difference, shift, and count leading operations.

Packed multiplication is described in the section [Packed Multiply and Packed MAC, page 1-13](#).

Compare instructions are described in [Compare Instructions, page 1-14](#).

### **1.3 PSW Status Flags and 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.

#### **1.3.1 Usage**

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 (Note that MTCR is only available in Supervisor mode). The Trap on Overflow (TRAPV) and Trap on Sticky Overflow (TRAPSV) instructions can be used to cause a trap if the respective V (overflow) and SV (sticky overflow) bits 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 it is only necessary to determine if an overflow occurred somewhere in an entire block of computation, then the SV bit is reset before the block (using the RSTV instruction) and tested after completion of the block (using MFCR).

Jumping based on the overflow result is achieved by using a MFCR instruction followed by a JZ.T or JNZ.T (conditional jump on the value of a bit) instruction.

#### **1.3.2 Saturation**

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.

### **1.4 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).

### 1.4.1 Scaling

The multiplier result can be treated in one of two ways:

- Left shifted by 1: One sign bit is suppressed and the result is left-aligned, so conserving the input format.
- Not shifted: The result retains its two sign bits (2.30 format). This format can be used with IIR (Infinite Impulse Response) filters for example, in which some of the coefficients are between 1 and 2, and to have one guard bit for accumulation.

### 1.4.2 Special Case = -1 \* -1

When multiplying two maximum-negative 1.15 format values (-1), the result is the positive number (+1). For example:

$$8000_{\text{H}} * 8000_{\text{H}} = 4000\ 0000_{\text{H}}$$

This is correctly interpreted in Q format as:

$$-1(1.15 \text{ format}) * -1(1.15 \text{ format}) = +1 (2.30 \text{ format})$$

However, when the result is shifted left by 1 (left-justified), the result is 8000 0000<sub>H</sub>. This is incorrectly interpreted as:

$$-1(1.15 \text{ format}) * -1(1.15 \text{ format}) = -1 (1.31 \text{ format})$$

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

$$8000_{\text{H}} * 8000_{\text{H}} = 7FFF\ FFFF_{\text{H}}$$

This is correctly interpreted in Q format as:

$$-1(1.15 \text{ format}) * -1(1.15 \text{ format}) = (\text{nearest representation of})+1 (1.31 \text{ format})$$

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.

### 1.4.3 Guard bits

When accumulating sums (in filter calculations for example), guard bits are often required to prevent overflow. The instruction set directly supports the use of one guard bit when using a 32-bit accumulator (2.30 format, where left shift by 1-bit of result is not requested). When more guard bits are required a register pair (64-bits) can be used. In that instance the intermediate result (also in 2.30 format, where left shift by 1-bit is not performed) is left shifted by 16-bits giving effectively a 18.46 format.

#### 1.4.4 Rounding

Rounding is used to retain the 16 most-significant bits of a 32-bit result. Rounding is implemented by adding 1 to bit 15 of a 32-bit intermediate result. If the operation writes a full 32-bit register (i.e. is not a component of a packed half-word operation), it then clears the lower 16-bits.

#### 1.4.5 Overflow and Saturation

Saturation on overflow is available on all DSP instructions.

#### 1.4.6 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. 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.

#### 1.4.7 Multiply and MAC

The available instructions for multiplication include:

- MUL.Q (Multiply Q format).
- MULR.Q (Multiply Q format with Rounding).

The operand encodings for the MUL.Q instruction distinguish between 16-bit source operands in either the upper D[n]U or lower half D[n]L of a data register, 32-bit source operands (D[n]), and 32-bit or 64-bit destination operands (D[n] or E[n]), giving a total of eight different cases:

- 16U \* 16U → 32
- 16L \* 16L → 32
- 16U \* 32 → 32
- 16L \* 32 → 32
- 32 \* 32 → 32
- 16U \* 32 → 64
- 16L \* 32 → 64
- 32 \* 32 → 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.

The MAC instructions consist of all the MUL combinations described above, followed by addition (MADD.Q, MADDS.Q,) and the rounding versions (MADDR.Q, MADDRS.Q). For the subtract versions of these instructions, ADD is replaced by SUB.

### 1.4.8 Packed Multiply and Packed MAC

There are three instructions for various forms of multiplication on packed 16-bit fractional values:

- MUL.H (Packed Multiply Q format).
- MULR.H (Packed Multiply Q format with Rounding).
- MULM.H (Packed Multiply Q format, Multi-precision).

These instructions 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 result that is stored into an extended register.

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

There is also a large group of Packed MAC instructions. These consist of all the MUL combinations described above, followed by addition, subtraction or a combination of both. Typical examples are MADD.H, MADDR.H, and MADDM.H.

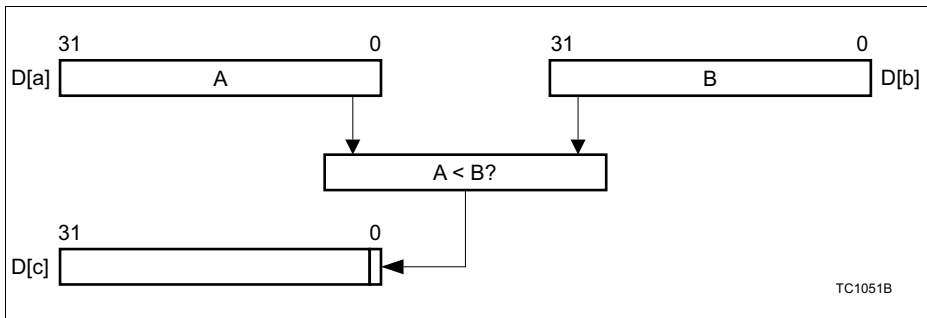
All combinations are found as either MADxxx.H or MSUxxx.H instructions.

## 1.5 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. The remaining bits in the register are cleared to zero.

### 1.5.1 Simple Compare

**Figure 8** illustrates the operation of the LT (Less Than) compare instruction:



**Figure 8 LT (Less Than) Comparison**

The comparison instructions are:

- EQ (Equal).
- NE (Not Equal).
- LT (Less Than).
- GE (Greater than or Equal to).

Versions for both signed and unsigned integers are available.

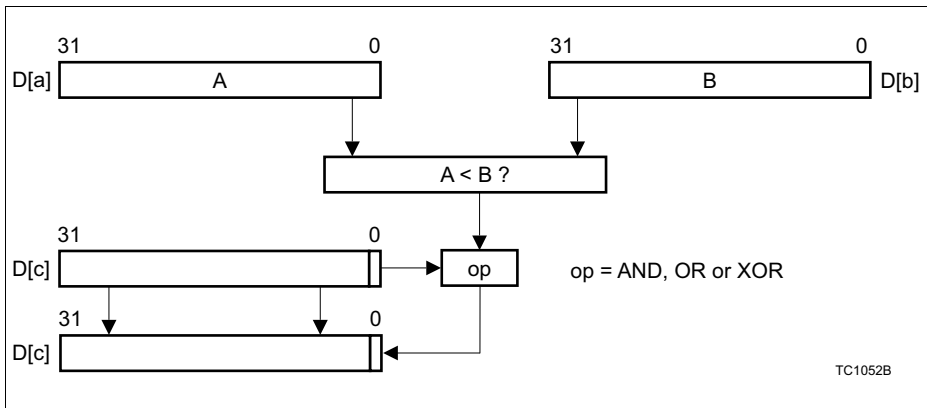
Comparison conditions not explicitly provided in the instruction set can be obtained by either swapping the operands when comparing two registers, or by incrementing the constant by one when comparing a register and a constant (See [Table 2](#)).

**Table 2 Equivalent Comparison Operations**

Implicit Comparison Operation		TriCore Equivalent Comparison Operation	
LE	D[c], D[a], D[b]	GE	D[c], D[b], D[a]
LE	D[c], D[a], const	LT	D[c], D[a], (const+1)
GT	D[c], D[a], D[b]	LT	D[c], D[b], D[a]
GT	D[c], D[a], const	GE	D[c], D[a], (const+1)

### 1.5.2 Accumulating Compare

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** illustrates the combination of the LT instruction with a Boolean operation.



**Figure 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);
```

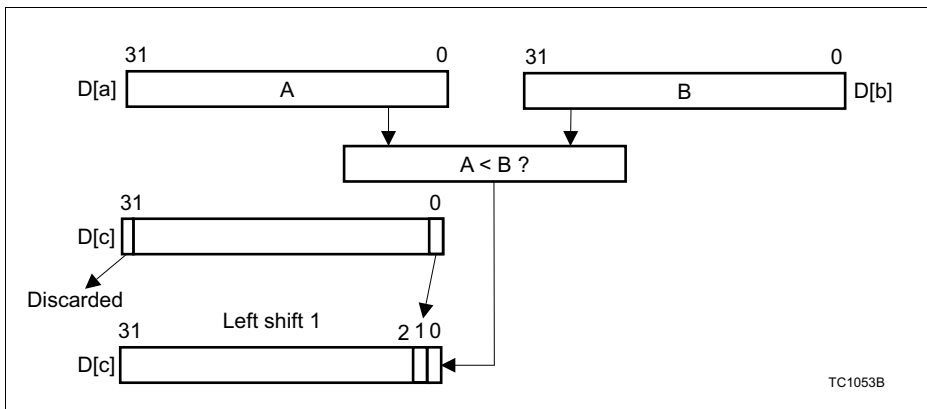
Assuming all variables are in registers, the following two instructions compute the value in d5:

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

### 1.5.3 Compare with Shift

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



**Figure 10 SH.LT Instruction**



### 1.5.4 Packed Compare

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 FF<sub>H</sub> for a match, or 00<sub>H</sub> for no match. Similarly for packed half-words there are special compare instructions that perform two individual half-word comparisons and produce two extended Booleans. The EQ.H instruction results in two extended Booleans: FFFF<sub>H</sub> for a match and 0000<sub>H</sub> for no match. There are also 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 FFFFFFFF<sub>H</sub> for match and 00000000<sub>H</sub> for no match.

Extended Booleans are useful as masks, which can be used by subsequent bit-wise logic operations. CLZ (Count Leading Zeros) or CLO (Count Leading Ones) can also be used on the result to quickly find the position of the left-most match. Figure 11 shows an example of the EQ.B instruction.

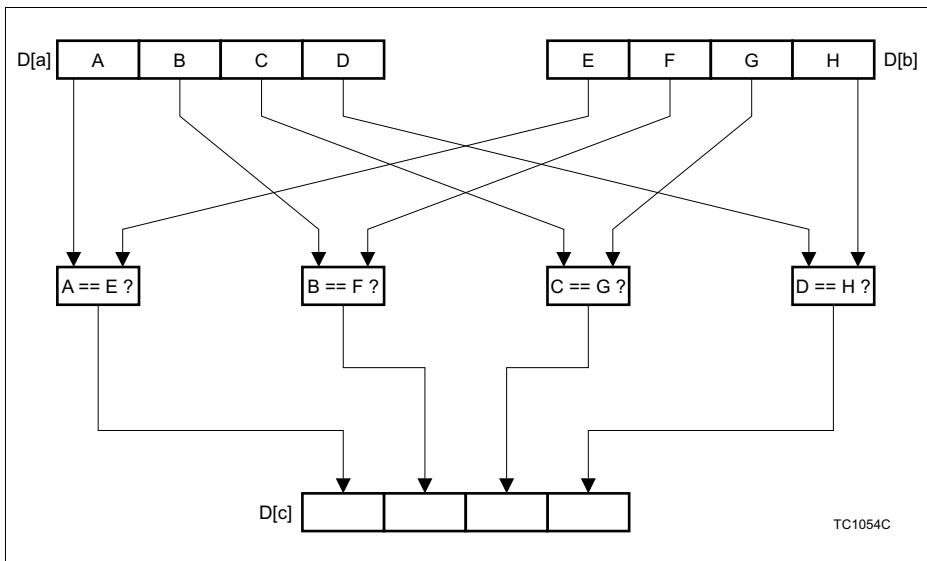


Figure 11 EQ.B Instruction Operation

See Also Addendum for TriCore Arch Manual, Vol2, V1.3.8, pages 493/494

## 1.6 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.

### 1.6.1 Simple Bit Operations

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 12](#). The available Boolean operations are:

- AND
- NAND
- OR
- NOR
- XOR
- XNOR
- ANDN
- ORN

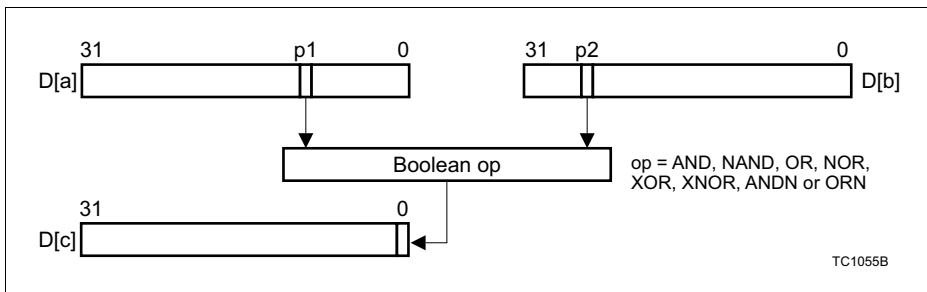
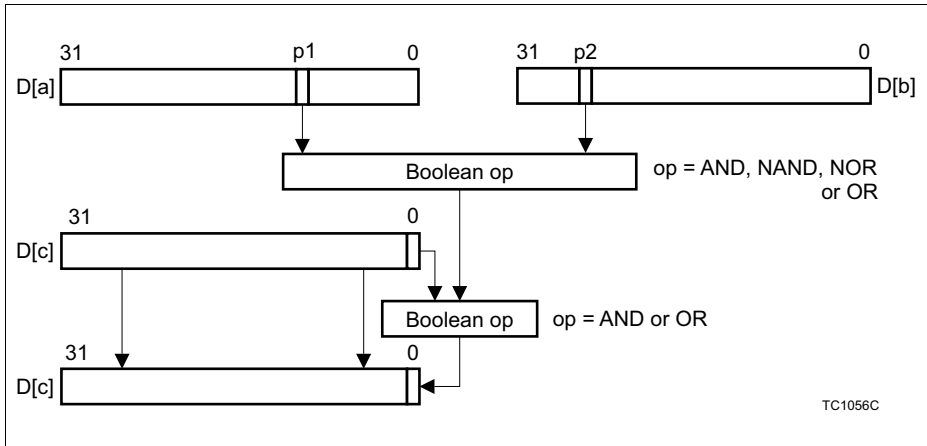


Figure 12 Boolean Operations

### 1.6.2 Accumulating Bit Operations

Evaluation of complex Boolean equations can use the 3-input Boolean operations, in which the output of a two-input instruction, together with the least-significant bit of a third data register, forms the input to a further operation. The result is written to bit 0 of the third data register, with the remaining bits unchanged (Figure 13).



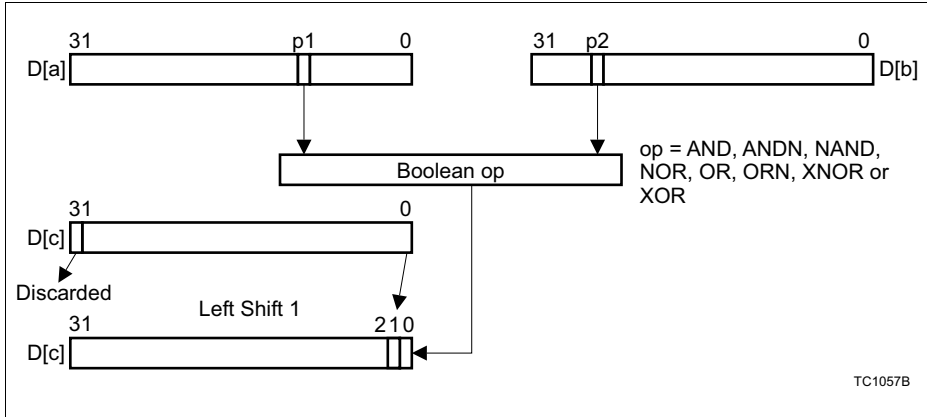
**Figure 13 Three-input Boolean Operation**

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
- OR.OR.T

### 1.6.3 Shifting Bit Operations

As with 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 14](#), [page 1-20](#)).



**Figure 14** Shift Plus Boolean Operation

See Also Addendum for TriCore Arch Manual, Vol2, V1.3.8, pages 493/494

## 1.7 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 a constant to be loaded into an address register. When the base pointer is in the first 16-KBytes of each segment this can be achieved using the Load Effective Address (LEA) instruction, using the absolute addressing mode.

Loading a 32-bit constant into an address register is accomplished using MOVH.A followed by an LEA that uses the base plus 16-bit offset addressing mode. For example:

```
movh.a    a5, ((ADDRESS+8000H)>>16) & FFFFH
lea      a5, [a5](ADDRESS & FFFFH)
```

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.

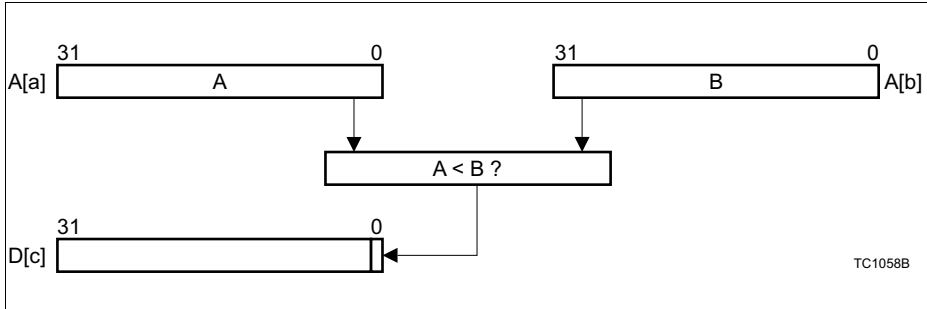
A 16-bit constant can be added to an address register by using the LEA instruction with the base plus offset addressing mode. A 32-bit constant can be added to an address register in two instructions: an Add Immediate High-word (ADDIH.A), which adds a 16-bit immediate to the most-significant 16-bits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

```
addih.a   a8, ((OFFSET+8000H)>>16) & FFFFH
lea      a8, [a8](OFFSET & FFFFH)
```

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

## 1.8 Address Comparison

As with the comparison instructions that use the data registers (see [Compare Instructions, page 1-14](#)), 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 15](#):



**Figure 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 3](#)).

**Table 3** Operation Equivalents

Implicit Comparison Operation	TriCore Equivalent Comparison Operation
LE.A D[c], A[a], A[b]	GE.A D[c], A[b], A[a]
GT.A D[c], A[a], A[b]	LT.A D[c], A[b], A[a]

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, a frequent operation when dealing with linked lists and complex data structures.

## 1.9 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 a conditional branch is taken depends on the result of a Boolean compare operation.

### 1.9.1 Unconditional Branch

There are three groups of unconditional branch instructions:

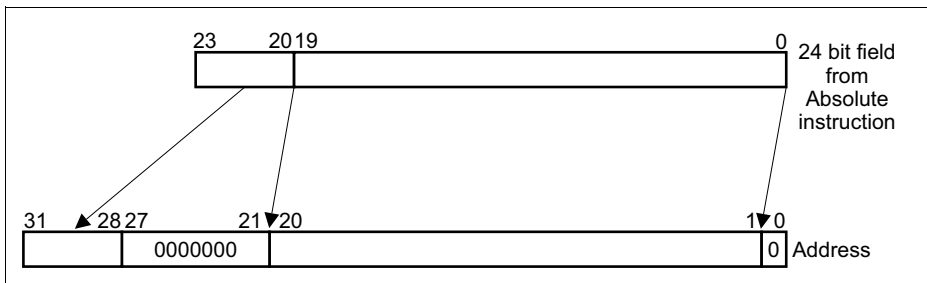
- Jump instructions.
- Jump and Link instructions.
- Call and Return instructions.

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 (RA) register A[11]. 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 the call instructions save the caller's registers upper context in a dynamically-allocated save area. The Return instruction, in addition to performing the return jump, restores the upper context.

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 a 24-bit field 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 field as an absolute address is shown in [Figure 16](#).



**Figure 16 Calculation of Absolute Address**

## 1.9.2 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 address. The scaled displacement is sign-extended to 32-bits before it is added to the program counter, unless otherwise noted.

### Conditional Jumps on Data Registers

Six of the conditional jump instructions use a 15-bit signed displacement field:

- JEQ (Comparison for Equality).
- JNE (Non-Equality).
- JLT (Less Than).
- JLT.U (Less Than Unsigned).
- JGE (Greater Than or Equal).
- JGE.U (Greater Than or Equal Unsigned).

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 D[15] register is equal to zero (JZ) or not equal to zero (JNZ). The displacement is 8-bit in this case.

The 16-bit instructions JEQ and JNE compare the implicit D[15] register with a 4-bit, sign-extended constant. The jump displacement field is limited to a 4-bit zero extended constant for these two instructions.

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 a 4-bit zero-extended constant in this case.

### 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:

- JEQ.A (Comparison for Equality).
- JNE.A (Non-Equality).
- JZ.A (Equal to Zero).
- JNZ.A (Non-Equal to Zero).

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 a 4-bit zero extended constant.



## 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 D[15] and have a displacement field of 4-bit zero extended constant.

### 1.9.3 Loop Instructions

Four special versions of conditional jump instructions are intended for efficient implementation of loops: JNEI, JNED, LOOP and LOOPU. These are described in this section.

#### Loop Instructions with Auto Incrementing/Decrementing Counter

The JNEI (Jump if Not Equal and Increment) and JNED (Jump if Not Equal and Decrement) instructions are similar to 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 D[3] = 3, 4, 5 ... 10, can be implemented as follows:

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

#### Loop Instructions with Reduced Execution Overhead

The LOOP instruction is a special kind of jump which utilizes the 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.

A loop that should be executed 100 times for example, may be implemented as:

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

This LOOP instruction (in the example 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

## **Instruction Set Overview**

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 also provided. This instruction 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.

## 1.10 Load and Store Instructions

The load (LD.x) and store (ST.x) instructions move data between registers and memory using seven addressing modes (Table 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 4 Addressing Modes**

Addressing Mode	Syntax	Effective Address	Instruction Format
Absolute	Constant	{offset18[17:14], 14'b0000000000000000, offset 18[13:0]}	ABS
Base + Short Offset	A[n]offset	A[n]+sign_ext(offset10)	BO
Base + Long Offset	A[n]offset	A[n]+sign_ext(offset16)	BOL
Pre-increment	+A[n]offset	A[n]+sign_ext(offset10)	BO
Post-increment	A[n+]offset	A[n]	BO
Circular	A[n] / A[n+1+c]	A[n]+A[n+1][15:0] (n is even)	BO
Bit-reverse	A[n] / A[n+r]	A[n]+A[n+1][15:0] (n is even)	BO

### 1.10.1 Load/Store Basic Data Types

The TriCore architecture defines loads and stores for the basic data types (corresponding to bytes, half-words, words, and double-words), as well as for signed fractions and addresses.

Note that when the data loaded from memory is smaller than the destination register (i.e. 8-bit 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.

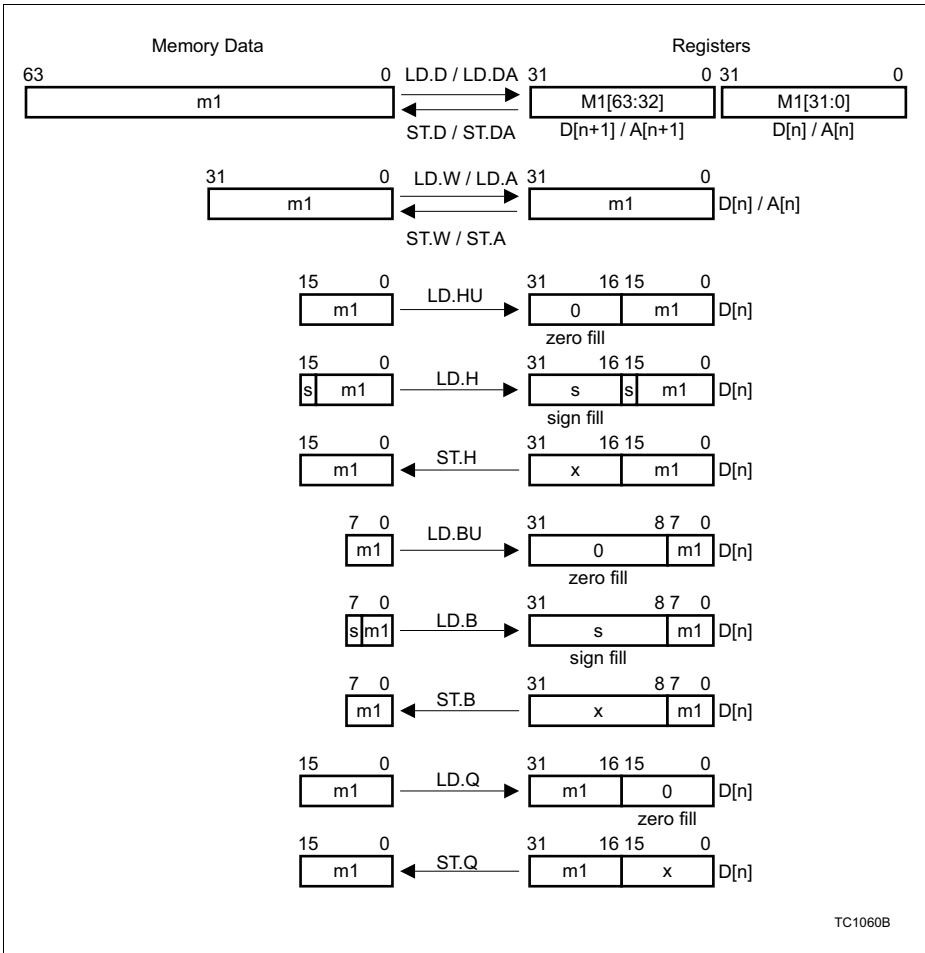


Figure 17 Load/Store Basic Data Type

### 1.10.2 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. It is then possible to extract, logically operate on, or jump to any bit in a register.

## Instruction Set Overview

Loading a single bit with a variable bit offset from a word-aligned byte pointer, is performed 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 zeros out the two lower bits, so 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; i.e. the bits that were either shifted out or masked out above. For example:

```
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
```

### 1.10.3 Store Bit and Bit Field

The ST.T (Store Bit) instruction can clear or set single memory or peripheral bits, resulting in reduced code size. ST.T 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 ST.T 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  $1011_B$  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 0000B    MASK
0000 0000 0000 0000 0000 0101 1000 0000B    VALUE
```

To store a single bit with a variable bit offset from a word-aligned byte pointer, the word address is first determined in the same way as for the load above. 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. Finally it zeros out the two lower bits, so 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
```

## 1.11 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 here.

### 1.11.1 Lower 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 explicitly saved or restored.

The SVLCX instruction (Save Lower Context) saves registers A[2] through A[7] and D[0] through D[7], together with the return address (RA) in register A[11] 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 A[2] through A[7] and D[0] through D[7] from the CSA. It also loads A[11] (Return Address) 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 = 1), allows the modification of the CPU priority number (CCPN), and saves the lower context in the same manner as the SVLCX instruction.

### 1.11.2 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 A[10] to A[15] and D[8] to D[15], and the current PSW and PCXI.

The LDUCX instruction (Load Upper Context) loads registers A[10] to A[15] and D[8] to D[15]. The PSW and link word fields in the saved context in memory are ignored. The PSW, FCX, and PCXI are unaffected.

The STLXC instruction (Store Lower Context) stores the same context information that is saved with an explicit lower context save operation: Registers A[2] to A[7] and D[0] to D[7], together with the Return Address (RA) in A[11] and the PCXI. The LDLCX instruction (Load Lower Context) loads registers A[2] through A[7] and D[0] through D[7]. The saved return address and the link word fields in the context stored in memory are ignored. Registers A[11] (Return Address), FCX, and PCXI are not affected.

## 1.12 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.

### 1.12.1 System Call

The SYSCALL instruction generates a system call trap, providing a secure mechanism for User mode application code to request Supervisor mode services. The system call trap, like other traps, vectors to the trap handler table, using the three-bit hardware-furnished trap class ID as an index. The trap class ID for system call traps is six. The Trap Identification Number (TIN) is specified by an immediate constant in the SYSCALL instruction and serves to identify the specific Supervisor mode service that is being requested.

### 1.12.2 Synchronization Primitives (DYSNC and ISYNC)

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

#### DSYNC

The DSYNC primitive 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.

#### ISYNC

The ISYNC primitive 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 guarantees 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 is 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. Re-Fetch the next instruction after the ISYNC.

### 1.12.3 Access to the Core Special Function Registers (CSFRs)

The core accesses the CSFRs through two instructions:

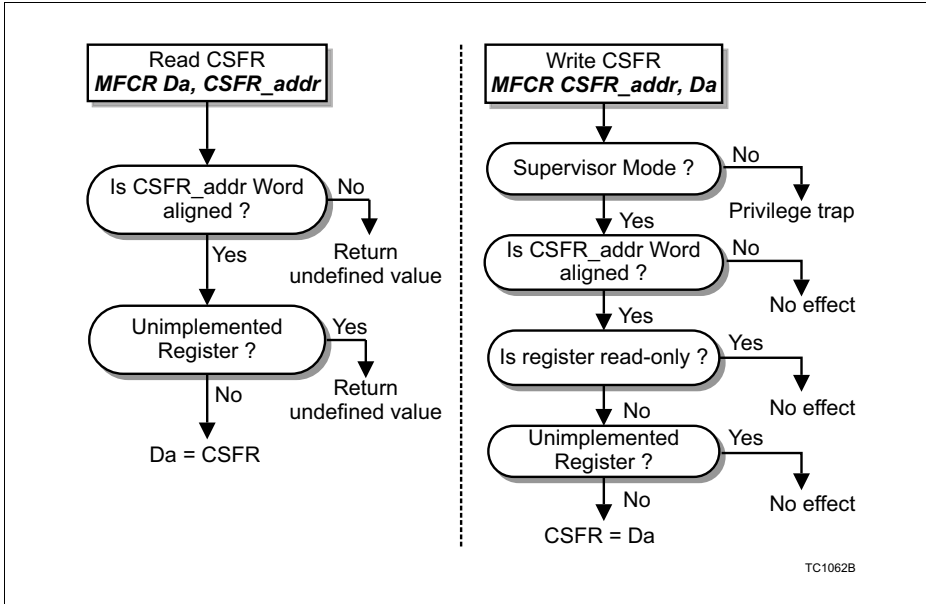
- MFCR:
  - The Move From Core Register instruction moves the contents of the addressed CSFR into a data register. MFCR can be executed in any mode (i.e. User-1, User-0 or Supervisor mode).
- MTCR:
  - The Move To Core Register instruction moves the contents of a data register to the addressed CSFR. To prevent unauthorized writes to the CSFRs the MTCR instruction can only be executed in Supervisor mode. An MTCR instruction should be followed by an ISYNC instruction. This ensures that all instructions following the MTCR see the effects of the CSFR update.

There are no instructions allowing bit, bit-field or load-modify-store accesses to the CSFRs. The RSTV instruction (Reset Overflow Flags) only resets the overflow flags in the PSW without modifying any of the other PSW bits. This instruction can be executed in any mode (i.e. User-1, User-0 or Supervisor mode).

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 that it is not permitted for the core to access the CSFRs through this mechanism. The core must use MFCR and MTCR.

**Figure 18** summarizes TriCore core behaviour when accessing CSFRs.





TC1062B

Figure 18 TriCore Core Behaviour Accessing CSFRs

### 1.12.4 Enabling and Disabling the Interrupt System

For non-interruptible operations, the ENABLE and DISABLE instructions allow the explicit enabling and disabling of interrupts in both User and Supervisor mode. 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 is serviced while interrupts are disabled is the NMI (Non-Maskable Interrupt), because it bypasses the normal interrupt structure.

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 BISR instruction (Begin Interrupt Service Routine) 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).

### 1.12.5 Return (RET) and Return From Exception (RFE) Instructions

The RET (Return) instruction is used to return from a function that was invoked via a CALL instruction. The RFE (Return From Exception) instruction is used to return from an interrupt or trap handler.

These 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 A[11] (prior to the context restore operation).

The two 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 non-zero, 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.

### 1.12.6 Trap Instructions

The Trap on Overflow (TRAPV) and Trap on Sticky Overflow (TRAPSV) instructions can be used to cause a trap if the PSWs V and SV bits respectively, are set. See [PSW Status Flags and Arithmetic Instructions, page 1-10](#).

### 1.12.7 No-operation (NOP)

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.

## 1.13 Coprocessor Instructions

The TriCore instruction set architecture may be extended with implementation defined, application specific instructions. These instructions are executed on dedicated coprocessor hardware attached to the coprocessor interface.

The coprocessors operate in a similar manner to the integer instructions, receiving operands from the general purpose data registers and able to return a result to the same registers.

The architecture supports the operation of up to four concurrent coprocessors ( $n = 0, 1, 2, 3$ ). Two of these ( $n = 0, 1$ ) are reserved for use by the TriCore CPU allowing two ( $n = 2, 3$ ) for use by the application hardware.

## 1.14 16-bit Instructions

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

The 16-bit instructions are formed by imposing one or more of the following format constraints:

- Smaller constants.
- Smaller displacements.
- Smaller offsets.
- Implicit source, destination, or base address registers.
- 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 D[15] and A[15] 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 D[15], and a 16-bit conditional move instruction (CMOV) which is controlled by the Boolean in D[15].

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. The offset is a scaled offset. It is scaled up to 10-bit by the type of instruction (byte, half-word, word).



See Also Addendum for TriCore Arch Manual, Vol2, V1.3.8, pages 493/494

## 2 Instruction Set Information

This chapter contains descriptions of all TriCore® instructions. The instruction mnemonics are grouped into families of similar or related instructions, then listed in alphabetical order within those groups.

*Note: All instructions and operators are signed unless stated 'unsigned'.*

*Note: Information specific to 16-bit instructions is shown in a box with a grey background.*

## 2.1 Instruction Syntax

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

### 2.1.1 Operand Definitions

**Table 5 Explicit Operand Description**

Operand	Description
D[n]	Data register n.
A[n]	Address register n.
E[n]	Extended data register n containing a 64-bit value made from an even/odd pair of registers (D[n], D[n+1]). The format is little endian. E[n][63:32] = D[n+1][31:0]; E[n][31:0] = D[n][31:0].
dispn	Displacement value of <i>n</i> bits used to form the effective address in branch instructions.
constn	Constant value of <i>n</i> bits used as instruction operand.
offn	Offset value of <i>n</i> bits used to form the effective address in load and store instructions.
pos1, pos2	Used to specify the position in a bit field instructions.
pos	Pos (position) is used with width to define a field.
width	Specifies the width of the bit field in bit field instructions.

### 2.1.2 Instruction Mnemonic

An instruction mnemonic is composed of up to three basic parts:

- A base operation.
  - Specifies the instructions basic operation. For example: ADD for addition, J for jump and LD for memory load. Some instructions such as OR.EQ, have more than one base operation, separated by a period (.).
- An operation modifier.
  - Specifies the operation more precisely. For example: ADDI for addition using an immediate value, or JL for a jump that includes a link. More than one operation modifier may be used for some instructions (ADDIH for example).
- An operand (data type) modifier.
  - Gives 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 half-word load. The data type modifier is separated by a period (.).

Using the ADDS.U instruction as an example:

- ‘ADD’ is the base operation.
- ‘S’ is an operation modifier specifying that the result is saturated.
- ‘U’ is a data type modifier specifying that the operands are unsigned.

Some instructions, typically 16-bit instructions, use a General Purpose Register (GPR) as an implicit source or destination.

**Table 6 Implicit Operand Descriptions**

Operand	Description
D[15]	Implicit Data Register for many 16-bit instructions.
A[10]	Stack Pointer (SP).
A[11]	Return Address (RA) register for CALL, JL, JLA, and JLI instructions, and Return PC value on interrupts.
A[15]	Implicit Address Register for many 16-bit load/store instructions.

*Note: In the syntax section of the instruction descriptions, the implicit registers are included as explicit operands. However they are not explicitly encoded in the instructions.*

### 2.1.3 Operation Modifiers

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

**Table 7 Operation Modifiers**

Operation Modifier	Name	Description	Example
C	Carry	Use and update PSW carry bit.	ADDC
I	Immediate	Large immediate.	ADDI
H	High word	Immediate value put in most-significant bits.	ADDIH
S	Saturation	Saturate result.	ADDS
X	Carry out	Update PSW carry bit.	ADDX
EQ	Equal	Comparison equal.	JEQ
GE	Greater than	Comparison greater than or equal.	JGE
L	Link	Record link (jump subroutine).	JL
A	Absolute	Absolute (jump).	JLA

Instruction Set Information

**Table 7**      **Operation Modifiers** (continued)

Operation Modifier	Name	Description	Example
I	Indirect	Register indirect (jump).	JLI
LT	Less than	Comparison less than.	JLT
NE	Not equal	Comparison not equal.	JNE
D	Decrement	Decrement counter.	JNED
I	Increment	Increment counter.	JNEI
Z	Zero	Use zero immediate.	JNZ
M	Multi-precision	Multi-precision result (>32-bit) in Q format	MULM
R	Round	Round result.	MULR
N	Not	Logical NOT.	SELN

**2.1.4**      **Data Type Modifiers**

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

**Table 8**      **Data Type Modifiers**

Data Type Modifier	Name	Description	Example
D	Data	32-bit data.	MOV.D
D	Double-word	64-bit data/address.	LD.D
W	Word	32-bit (word) data.	EQ.W
A	Address	32-bit address.	ADD.A
Q	Q Format	16-bit signed fraction (Q format).	MADD.Q
H	Half-word	16-bit data or two packed half-words.	ADD.H
B	Byte	8-bit data or four packed bytes.	ADD.B
T	Bit	1-bit data.	AND.T
U	Unsigned	Unsigned data type.	ADDS.U

*Note: Q format can be used as signed half-word multipliers.*

See Also Addendum for TriCore Arch Manual, Vol2, V1.3.8, pages 493/494



## 2.2 Opcode Formats

### 2.2.1 16-bit Opcode Formats

Note: Bit[0] of the op1 field is always 0 for 16-bit instructions.

**Table 9 16-bit Opcode Formats**

	15-14	13-12	11-10	09-08	07-06	05-04	03-02	01-00
SB	disp8				op1			
SBC	const4		disp4		op1			
SBR	s2		disp4		op1			
SBRN	n		disp4		op1			
SC	const8				op1			
SLR	s2		d		op1			
SLRO	off4		d		op1			
SR	op2		s1/d		op1			
SRC	const4		s1/d		op1			
SRO	s2		off4		op1			
SRR	s2		s1/d		op1			
SRRS	s2		s1/d		n	op1		
SSR	s2		s1		op1			
SSRO	off4		s1		op1			

### 2.2.2 32-bit Opcode Formats

Note: Bit[0] of the op1 field is always 1 for 32-bit instructions.

**Table 10 32-bit Opcode Formats**

	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	5-0
ABS	off18 [9:6]		op2	off18 [13:10]		off18 [5:0]			off18 [17:14]		s1/d		op1	
ABSB	off18 [9:6]		op2	off18 [13:10]		off18 [5:0]			off18 [17:14]		b	bpos 3		op1
B	disp24[15:0]								disp24[23:16]				op1	
BIT	d		pos2		op2	pos1			s2		s1		op1	

Instruction Set Information

**Table 10 32-bit Opcode Formats (continued)**

	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	5-0
BO	off10 [9:6]		op2			off10 [5:0]			s2	s1/d	op1			
BOL	off16 [9:6]		off16 [15:10]			off16 [5:0]			s2	s1/d	op1			
BRC	op2	disp15							const4	s1	op1			
BRN	op2	disp15							n[3:0]	s1	n[4]	op1		
BRR	op2	disp15							s2	s1	op1			
RC	d	op2				const9					s1	op1		
RCPW	d	pos		op2	width		const4		s1	op1				
RCR	d	s3		op2	const9					s1	op1			
RCRR	d	s3		op2	-		const4		s1	op1				
RCRW	d	s3		op2	width		const4		s1	op1				
RLC	d	const16									s1	op1		
RR	d	op2				-	n	s2		s1	op1			
RR1	d	op2						n	s2	s1	op1			
RR2	d	op2							s2	s1	op1			
RRPW	d	pos		op2	width			s2	s1	op1				
RRR	d	s3		op2	-	n	s2		s1	op1				
RRR1	d	s3		op2			n	s2	s1	op1				
RRR2	d	s3		op2				s2	s1	op1				
RRRR	d	s3		op2	-			s2	s1	op1				
RRRW	d	s3		op2	width			s2	s1	op1				
SYS	-	op2				-					s1/d	op1		

## 2.2.3 Opcode Field Description

**Table 11 Opcode Field Descriptions**

Name	Width	Description
s1	4	Source register(s) one.
s2	4	Source register(s) number two.
s3	4	Source register(s) number three.
d	4	Destination register. In case of register pair (E) the coding follows the register number. E[0] = 0000 <sub>B</sub> , E[2] = 0010 <sub>B</sub> , E[4] = 0100 <sub>B</sub> , etc.
b	1	Bit value.
bpos3	3	Bit position in a byte.
pos	5	Bit position in a register.
pos1	5	Bit position in a register.
pos2	5	Bit position in a register.
width	5	Bit position in a register.
n	2	<ul style="list-style-type: none"> <li>- Multiplication result shift value (only 00<sub>B</sub> and 01<sub>B</sub> are valid).</li> <li>- Address shift value in add scale.</li> <li>- Default to zero in all other operations using the format.</li> <li>- Coprocessor number for coprocessor instructions.</li> </ul>
const4	4	4-bit constant.
const9	9	9-bit constant.
const16	16	16-bit constant.
disp4	4	4-bit displacement.
disp8	8	8-bit displacement.
disp15	15	15-bit displacement
disp24	24	24-bit displacement.
off4	4	4-bit offset.
off10	10	10-bit offset.
off16	16	16-bit offset.
-		Reserved Field. Read value is undefined; should be written with 0. Must be set to zero to allow for future compatibility.
op1		Primary Opcode.
op2		Secondary Opcode.

## 2.3 Instruction Operation Syntax

The operation of each instruction is described using a 'C-like' Register Transfer Level (RTL) notation.

*Note: The numbering of bits begins with bit zero, which is the least-significant bit of the word.*

*Note: All intermediate 'result' values are assumed to have infinite precision unless otherwise indicated.*

**Table 12 RTL Syntax Description**

Syntax	Description
bpos3	Bit position.
constn	Constant value of $n$ bits used as instruction operand.
dispn	Displacement value of $n$ bits used to form the effective address in branch instructions.
(expression)[p]	A single bit, with ordinal index 'p' in the bit field '(expression)'.
$n$ 'bx	Constant bit string, where $n$ is the number of bits in the constant and $x$ is the constant in binary. For example; 2'b11.
$n$ 'hx	Constant bit string, where $n$ is the number of bits in the constant and $x$ is the constant in hexadecimal. For example; 16'hFFFF.
offn	Offset value of $n$ bits used to form the effective address in load and store instructions.
pos	Single bit position.
signed	A value that can be positive, negative or zero.
ssov	Saturation on signed overflow.
suov	Saturation on unsigned overflow.
unsigned	A value that can be positive or zero.
{x, y}	A bit string where $x$ and $y$ are expressions representing a bit or bit field. Any number of expressions can be concatenated, for example; {x,y,z}.
A[n]	Address register $n$ .
CR	Core Register.
D[n]	Data register $n$ .
EA	Effective Address.

**Table 12 RTL Syntax Description (continued)**

<b>Syntax</b>	<b>Description</b>
E[n]	Data register containing a 64-bit value, constructed by pairing two data registers. The least-significant bit is in the even register D[n] and the most-significant bit is in the odd register D[n+1].
M(EA, data_size)	Memory locations beginning at the specified byte location EA, and extending to EA + data_size - 1. data_size = byte, half-word, word, double-word, 16-word.
<mode>	An addressing mode.
P[n]	Address register containing a 64-bit value, constructed by pairing two address registers. The least-significant bit is in the even register A[n] and the most-significant bit is in the odd register A[n+1].
PC	The address of the instruction in memory.
[x:y]	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.
TRUE	Boolean true. Equivalent to integer 1.
FALSE	Boolean false. Equivalent to integer 0.
AND	Logical AND. Returns a boolean result.
OR	Logical OR. Returns a boolean result.
XOR	Logical XOR. Returns a boolean result.
!	Logical NOT. Returns a boolean result.
^	Bitwise XOR.
&	Bitwise AND.
	Bitwise OR.
~	Bitwise NOT.
<	Less than. Returns a boolean result.
>	Greater than. Returns a boolean result.
<=	Less than or equal to. Returns a boolean result.
>=	Greater than or equal to. Returns a boolean result.
>>	Right shift. High order bits shifted in are 0's.
<<	Left shift. Low order bits shifted in are 0's.
+	Add.
-	Subtract.

See Also Addendum for TriCore Arch Manual, Vol2, V1.3.8, pages 493/494

**Table 12** RTL Syntax Description (continued)

Syntax	Description
*	Multiply.
/	Divide.
%	Modulo.
=	Equal to (assignment).
==	Is equal to (comparison). Returns a boolean result.
!=	Not equal to. Returns a boolean result.
<sup>a</sup>	Approximately equal to.
	Parallel operation.
? :	Conditional expression (Ternary operator).
∞	Infinity.
//	Comment.

See Also Addendum for TriCore Arch Manual, Vol2, V1.3.8, pages 493/494

### 2.3.1 RTL Functions

Table 13 RTL Functions

Function	Description
abs(x)	abs(x) returns $((x < 0) ? (0 - x) : x)$ ;
cache_address_ivld(EA)	Defined in the 'Cache RTL Functions' section, which follows.
cache_address_wb(EA)	Defined in the 'Cache RTL Functions' section, which follows.
cache_address_wi(EA)	Defined in the 'Cache RTL Functions' section, which follows.
cache_index_wb(EA)	Defined in the 'Cache RTL Functions' section, which follows.. (TriCore 1.3.1)
cache_index_wi(EA)	Defined in the 'Cache RTL Functions' section, which follows. (TriCore 1.3.1)
carry(a,b,c)	carry(a,b,c) { result = a + b + c; // unsigned additions return result[32];
cdc_decrement()	If PSW.CDC == 7'b11111111 returns FALSE, otherwise decrements PSW.CDC.COUNT and returns TRUE if PSW.CDC.COUNT underflows, otherwise returns FALSE.
cdc_increment()	If PSW.CDC == 7'b11111111 returns FALSE, otherwise increments PSW.CDC.COUNT and returns TRUE if PSW.CDC.COUNT overflows, otherwise returns FALSE.
cdc_zero()	Returns TRUE if PCW.CDC.COUNT == 0 or if PSW.CDC == 7'b11111111, otherwise returns FALSE.
leading_ones(x)	Returns the number of leading ones of 'x'.
leading_signs(x)	Returns the number of leading sign bits of 'x'.
leading_zeros(x)	Returns the number of leading zeros of 'x'.
reverse16(n)	{n[0], n[1], n[2], n[3], n[4], n[5], n[6], n[7], n[8], n[9], n[10], n[11], n[12], n[13], n[14], n[15]}.
round16(x)	= $\{(x + 32'h00008000)[31:16], 16'h0000\}$ ;
ssov(x,y)	max_pos = $(1 \ll (y - 1)) - 1$ ; max_neg = $-(1 \ll (y - 1))$ ; return $((x > \text{max\_pos}) ? \text{max\_pos} : ((x < \text{max\_neg}) ? \text{max\_neg} : x))$ ;
suov(x,y)	max_pos = $(1 \ll y) - 1$ ; return $((x > \text{max\_pos}) ? \text{max\_pos} : ((x < 0) ? 0 : x))$ ;
sign_ext(x)	Sign extension; high-order bit of x is left extended.

**Table 13** RTL Functions (continued)

Function	Description
trap(x)	Instruction will take trap x.
zero_ext(x)	Zero extensions; high-order bits are set to 0.

### 2.3.2 Cache RTL Functions

CACHE[ ] is a syntactic structure which hides the implementation characteristics of the cache implemented.

CACHE can be associatively accessed either by:

- A single argument which is an address.
- Two arguments consisting of implementation defined ranges for set\_index and set\_element.

In either case the CACHE[ ] access returns a structure with:

- Boolean validity information (CACHE[ ].valid).
- Boolean data modification information (CACHE[ ].modified).
- Physical address of the copied location (CACHE[ ].physical\_address).
- Stored data associated with the address (CACHE[ ].data).

The cache function descriptions are given in the following table.

*Note: 'cacheline', which appears in the cache function descriptions, is the size of the cache line in bytes and is implementation dependent.*

*Note: 'index' and 'elem', which appear in the cache function descriptions, are the set\_index and set\_element values. These values are implementation dependent.*

**Table 14** Cache Functions

Function	Definition
cache_address_ivld(EA)	if (CACHE[EA].valid==1) then CACHE [EA].valid=0;
cache_address_wb(EA)	if ((CACHE[EA].valid==1) AND (CACHE[EA].modified==1)) then { pa = CACHE[EA].physical_address; M[pa,cacheline] = CACHE[EA].data; CACHE[EA].modified = 0; }



Table 14 Cache Functions (continued)

Function	Definition
<b>cache_address_wi(EA)</b>	<pre> if (CACHE[EA].valid==1) then { if (CACHE[EA].modified==1) then { pa = CACHE[EA].physical_address; M[pa,cacheline] = CACHE[EA].data; } CACHE[EA].modified = 0; CACHE[EA].valid = 0; } </pre>
<b>cache_index_wb(location)</b> TriCore 1.3.1	<pre> if ((CACHE[index,elem].valid==1) AND (CACHE[index,elem].modified==1)) then { pa = CACHE[index,elem].physical_address; M[pa,cacheline] = CACHE[index,elem].data; CACHE[index,elem].modified = 0; } </pre>
<b>cache_index_wi(location)</b> TriCore 1.3.1	<pre> if (CACHE[index,elem].valid==1) then { if (CACHE[index,elem].modified==1) then { pa = CACHE[index,elem].physical_address; M[pa,cacheline] = CACHE[index,elem].data; } CACHE[index,elem].modified = 0; CACHE[index,elem].valid = 0; } </pre>

### 2.3.3 Floating Point Operation Syntax

Table 2-1 Floating Point Syntax Definitions

Syntax	Definition
QUIET_NAN	7FC00000 <sub>H</sub>
ADD_NAN	7CF00001 <sub>H</sub>
MUL_NAN	7CF00002 <sub>H</sub>
SQRT_NAN	7CF00004 <sub>H</sub>
DIV_NAN	7CF00008 <sub>H</sub>
POS_INFINITY	7F800000 <sub>H</sub>
NEG_INFINITY	FF800000 <sub>H</sub>
is_s_nan(x)	the IEEE754 32-bit single precision floating point format value x and returns the boolean result of the expression: (x[31:22] == 10'b011111110) AND (x[21:0] != 0);
is_q_nan(x)	Takes the IEEE754 32-bit single precision floating point format value x and returns the boolean result of the expression: (x[31:0] == QUIET_NAN);
is_nan(x)	Takes the IEEE754 32-bit single precision floating point format value x and returns the boolean result of the expression: (is_s_nan(x) OR is_q_nan(x));
is_pos_inf(x)	Takes the IEEE754 32-bit single precision floating point format value x and returns the boolean result of the expression: (x[31:0] == POS_INFINITY);
is_neg_inf(x)	Takes the IEEE754 32-bit single precision floating point format value x and returns the boolean result of the expression: (x[31:0] == NEG_INFINITY);
is_inf(x)	Takes the IEEE754 32-bit single precision floating point format value x and returns the boolean result of the expression: (is_neg_inf(x) OR is_pos_inf(x));
is_zero(x)	Takes the IEEE754 32-bit single precision floating point format value x and returns the boolean result of the expression: (x[30:0] == 0);
is_denorm(x)	Takes the IEEE754 32-bit single precision floating point format value x and returns the boolean result of the expression: (x[30:23] == 0) AND (x[22:0] != 0);

**Table 2-1 Floating Point Syntax Definitions** (continued)

Syntax	Definition
denorm_to_zero(x)	If the IEEE754 32-bit single precision floating point format value x is a denormal value return the appropriately signed infinitely accurate real value 0. Otherwise return x as an infinitely accurate real value; i.e. if((x < 0) AND (x > -2 <sup>-126</sup> )) then return -0.0; else if((x > 0) AND (x < 2 <sup>126</sup> )) then return +0.0; else return f_real(x);
round_to_integer(x,y)	Returns a signed integer result of infinite width by rounding the IEEE754 32-bit single precision floating point format value x to an integer value using the IEEE754 mode specified by y.
round_to_unsigned(x,y)	Returns an unsigned integer result of infinite width by rounding the IEEE754 32-bit single precision floating point format value x to an integer value using the IEEE754 mode specified by y.
round_to_q31(x,y)	Returns a Q format result of infinite width by rounding the real value x to a Q format value using the IEEE754 mode specified by y.
i_real(x)	Returns a infinitely accurate real number of equal value to the 32-bit signed integer value x.
u_real(x)	Returns a infinitely accurate real number of equal value to the 32-bit unsigned integer value x.
f_real(x)	Returns the IEEE754 32-bit single precision floating point format value x as an infinitely accurate real value.
q_real(x)	Returns the Q31 format value x as an infinitely accurate real value.
add(x,y)	Adds the real value x to the real value y and returns an infinitely accurate real result.
mul(x,y)	Multiply the real value x by the real value y and return an infinitely accurate real result.
divide(x,y)	Divides the real value x by the real value y and returns an infinitely accurate real result.
ieee754_round(x,y)	Rounds the real value x using the type of rounding specified by y compliant with IEEE754.
ieee754_32bit_format(x)	Returns the real value x in the standard 32-bit single precision IEEE754 floating point format. x is converted to the correct IEEE754 result on overflow or underflow.

See Also Addendum for TriCore Arch Manual, Vol2, V1.3.8, pages 493/494

**Table 2-1 Floating Point Syntax Definitions** (continued)

Syntax	Definition
ieee754_lt(x,y)	Returns TRUE if x is less than y according to the IEEE754 rules for 32-bit single precision floating point numbers otherwise returns FALSE.
ieee754_gt(x,y)	Returns TRUE if x is greater than y according to the IEEE754 rules for 32-bit single precision floating point numbers otherwise returns FALSE.
ieee754_eq(x,y)	Returns TRUE if x is equal to y according to the IEEE754 rules for 32-bit single precision floating point numbers otherwise returns FALSE.
fp_abs(x)	Returns the infinitely accurate absolute value of the real value x; i.e. $(x < 0.0) ? (0.0 - x) : x$ ;
approx_inv_sqrt(x)	Takes the real argument x and returns the approximate inverse square root ( $x^{-0.5}$ ) to at least 6.75 bits of precision.

## 2.4 Coprocessor Instructions

The TriCore® instruction set architecture may be extended with implementation defined, application specific coprocessor instructions. These instructions are executed on dedicated coprocessor hardware attached to the coprocessor interface.

The coprocessors operate in a similar manner to the integer instructions, receiving operands from the general purpose data registers, returning a result to the same registers.

The architecture supports the operation of up to four concurrent coprocessors ( $n = 0, 1, 2, 3$ ). Two of these ( $n = 0, 1$ ) are reserved for use by the TriCore CPU allowing two ( $n = 2, 3$ ) for use by the application hardware.

### COPROCESSORD[d], D[a], D[b] (I)

31	28 27	20 19 18 17 16 15	12 11	8 7	0
c	op2	-	n	b	a
					4B <sub>H</sub>

$D[c] = \text{op2}[n] (D[a], D[b]);$

### COPROCESSORD[c], D[d], D[a], D[b] (R)

31	28 27	24 23	20 19 18 17 16 15	12 11	8 7	0
c	d	op2	-	n	b	a
						6B <sub>H</sub>

$D[c] = \text{op2}[n] (D[d], D[a], D[b]);$

### Status Flags

C	Not set by this instruction.
V	Not set by this instruction.
SV	Not set by this instruction.
AV	Not set by this instruction.
SAV	Not set by this instruction.

## 2.5 PSW Status Flags (User Status Bits)

The Status section of a given instruction description lists the five status flags that may be affected by the operation. The PSW logically groups the five user bits together as shown below.

*Note: In the following table, 'result' for 32-bit instructions is D[c]. For 16-bit instructions it is D[a] or D[15](when implicit).*

**Table 15 PSW Status Flags**

Field	PSW Bit	Type	Description
C	31	rw	Carry The result has generated a carry_out. if (carry_out) then PSW.C = 1 else PSW.C = 0;
V	30	rw	Overflow * The result exceeds the maximum or minimum signed or unsigned value, as appropriate. if (overflow) then PSW.V = 1 else PSW.V = 0;
SV	29	rw	Sticky Overflow A memorized overflow. Overflow is defined by V, above. if (overflow) then PSW.SV = 1 else PSW.SV = PSW.SV;
AV	28	rw	Advance Overflow * if (advanced_overflow) then PSW.AV = 1 else PSW.AV = 0;
SAV	27	rw	Sticky Advance Overflow A memorized advanced overflow. Advanced_overflow is defined by AV, above. if (advanced_overflow) then PSW.SAV = 1 else PSW.SAV = PSW.SAV;

*Note: The PSW register is defined in Volume 1, Core Registers.*

### \* Programming Note: V (Overflow) and AV (Advanced Overflow) Status Bits

Because the TriCore Instruction Set contains many compound instructions (MULR, MAC, ABSDIF), the question arises; *when are the overflow flags computed?*

The AV and V flags are computed on the final operation, except in the case of instructions with saturation, when it is always before saturation. Saturation is not part of the operation as such, but is the resulting effect (chosen by the user) of an overflow situation.

## 2.6 List of OS and I/O Privileged Instructions

The following is a list of operating system Input/Output privileged instructions:

Kernel (Supervisor Mode)	User-1 Mode	User-0 Mode
BISR MTCR CACHEA.I TLBMAP TLBDEMAP TLBFLUSH.A TLBFLUSH.B TLBPROBE.A TLBPROBE.I RFM	ENABLE DISABLE	all others (including DEBUG)

See Also Addendum for TriCore Arch Manual, Vol2, V1.3.8, pages 493/494



### 3 Instruction Set

The instruction mnemonics which follow are grouped into families of similar or related instructions, then listed in alphabetical order within those groups.

For explanations of the syntax used, please refer to the previous chapter.

#### 3.1 CPU Instructions

Each page for this group of instructions is laid out as follows:

**J** 1

**Jump Unconditional** 2

**Description** 3

Add the value specified by disp24, multiplied by two and sign-extended to 32-bits, to the contents of PC and jump to that address.

Add the value specified by disp8, multiplied by two and sign-extended to 32-bits, to the contents of PC and jump to that address. 4

**J** 5 **disp24 (B)**

31	16	15	8	7	0
disp24[15:0]		di	3:16]		1D <sub>H</sub>

PC = PC + sign\_ext(2 \* disp24); 7

**J** 8 **disp8 (SB)**

15	8	7	0
disp8		3C <sub>H</sub>	<span style="border: 1px solid black; border-radius: 50%; padding: 2px;">9</span>

PC = PC + sign\_ext(2 \* disp8); 10

**Status Flags**

C	Not set by this instruction. <span style="border: 1px solid black; border-radius: 50%; padding: 2px;">11</span>
V	Not set by this instruction.
SV	Not set by this instruction.
AV	Not set by this instruction.
SAV	Not set by this instruction.

**Examples**

j foobar 12

j foobar 13

**See Also**

[JA](#), [JI](#), [JL](#), [JLA](#), [JLI](#) 14

**Key:**

- 1 Instruction Mnemonic
- 2 Instruction Longname
- 3 Description (32-bit)
- 4 Description (16-bit)
- 5 Syntax (32-bit), and Instruction format in parentheses. Note also 15
- 6 Opcodes (32-bit)
- 7 Operation in RTL format (32-bit)
- 8 Syntax (16-bit)
- 9 Opcodes (16-bit)
- 10 Operation (RTL) (16-bit)
- 11 Status Flags (User Status Bits)
- 12 Instruction Examples (32-bit)
- 13 Instruction Examples (16-bit)
- 14 Related instructions
- 15 Operation quick reference following Syntax; see 5 (MAC instructions only)

**MSUB**      **D[c], D[d]**      **const9 (RCR)**  
32 - (32 \* K3) --> 32 signed 15

TC1066

See Also Addendum for TriCore Arch Manual, Vol2, V1.3.8, pages 493/494

## ABS Absolute Value

### Description

Put the absolute value of data register D[b] in data register D[c]: If the contents of D[b] are greater than or equal to zero then copy it to D[c], otherwise change the sign of D[b] and copy it to D[c].

The operands are treated as signed 32-bit signed integers.

### ABS                      D[c], D[b] (RR)

31	28 27	20 19 18 17 16 15	12 11	8 7	0
c	1C <sub>H</sub>	-	-	b	0B <sub>H</sub>

result = (D[b] >= 0) ? D[b] : (0 - D[b]);

D[c] = result[31:0];

### Status Flags

C	Not set by this instruction.
V	overflow = (result > 7FFFFFFF <sub>H</sub> ) OR (result < -80000000 <sub>H</sub> ); if (overflow) then PSW.V = 1 else PSW.V = 0;
SV	if (overflow) then PSW.SV = 1 else PSW.SV = PSW.SV;
AV	advanced_overflow = result[31] ^ result[30]; if (advanced_overflow) then PSW.AV = 1 else PSW.AV = 0;
SAV	if (advanced_overflow) then PSW.SAV = 1 else PSW.SV = PSW.SV;

### Examples

abs    d3, d1

### See Also

[ABSDIF](#), [ABSDIFS](#), [ABSS](#)

## ABS.B

Absolute Value Packed Byte

## ABS.H

Absolute Value Packed Half-word

### Description

Put the absolute value of each byte (ABS.B) or half-word (ABS.H) in data register D[b] into the corresponding byte or half-word of data register D[c]. The operands are treated as signed, 8-bit or 16-bit integers.

The overflow condition is calculated for each byte or half-word of the packed quantity.

### ABS.B D[c], D[b] (RR)

31	28 27	20 19 18 17 16 15	12 11	8 7	0
c	5C <sub>H</sub>	-	-	b	0B <sub>H</sub>

result\_byte3 = (D[b][31:24] >= 0) ? D[b][31:24] : (0 - D[b][31:24]);

result\_byte2 = (D[b][23:16] >= 0) ? D[b][23:16] : (0 - D[b][23:16]);

result\_byte1 = (D[b][15:8] >= 0) ? D[b][15:8] : (0 - D[b][15:8]);

result\_byte0 = (D[b][7:0] >= 0) ? D[b][7:0] : (0 - D[b][7:0]);

D[c] = {result\_byte3[7:0], result\_byte2[7:0], result\_byte1[7:0], result\_byte0[7:0]};

### ABS.H D[c], D[b] (RR)

31	28 27	20 19 18 17 16 15	12 11	8 7	0
c	7C <sub>H</sub>	-	-	b	0B <sub>H</sub>

result\_halfword1 = (D[b][31:16] >= 0) ? D[b][31:16] : (0 - D[b][31:16]);

result\_halfword0 = (D[b][15:0] >= 0) ? D[b][15:0] : (0 - D[b][15:0]);

D[c] = {result\_halfword1[15:0], result\_halfword0[15:0]};

### Status Flags

C	Not set by these instructions.
---	--------------------------------

V	<p>ABS.B  <math>ov\_byte3 = (result\_byte3 &gt; 7F_H) \text{ OR } (result\_byte3 &lt; -80_H);</math>  <math>ov\_byte2 = (result\_byte2 &gt; 7F_H) \text{ OR } (result\_byte2 &lt; -80_H);</math>  <math>ov\_byte1 = (result\_byte1 &gt; 7F_H) \text{ OR } (result\_byte1 &lt; -80_H);</math>  <math>ov\_byte0 = (result\_byte0 &gt; 7F_H) \text{ OR } (result\_byte0 &lt; -80_H);</math>  <math>overflow = ov\_byte3 \text{ OR } ov\_byte2 \text{ OR } ov\_byte1 \text{ OR } ov\_byte0;</math>                      if (overflow) then PSW.V = 1 else PSW.V = 0;                      ABS.H  <math>ov\_halfword1 = (result\_halfword1 &gt; 7FFF_H) \text{ OR } (result\_halfword1 &lt; -8000_H);</math>  <math>ov\_halfword0 = (result\_halfword0 &gt; 7FFF_H) \text{ OR } (result\_halfword0 &lt; -8000_H);</math>  <math>overflow = ov\_halfword1 \text{ OR } ov\_halfword0;</math>                      if (overflow) then PSW.V = 1 else PSW.V = 0;</p>
SV	<p>if (overflow) then PSW.SV = 1 else PSW.SV = PSW.SV;</p>
AV	<p>ABS.B  <math>aov\_byte3 = result\_byte3[7] \wedge result\_byte3[6];</math>  <math>aov\_byte2 = result\_byte2[7] \wedge result\_byte2[6];</math>  <math>aov\_byte1 = result\_byte1[7] \wedge result\_byte1[6];</math>  <math>aov\_byte0 = result\_byte0[7] \wedge result\_byte0[6];</math>  <math>advanced\_overflow = aov\_byte3 \text{ OR } aov\_byte2 \text{ OR } aov\_byte1 \text{ OR } aov\_byte0;</math>                      if (advanced_overflow) then PSW.AV = 1 else PSW.AV = 0;                      ABS.H  <math>aov\_halfword1 = result\_halfword1[15] \wedge result\_halfword1[14];</math>  <math>aov\_halfword0 = result\_halfword0[15] \wedge result\_halfword0[14];</math>  <math>advanced\_overflow = aov\_halfword1 \text{ OR } aov\_halfword0;</math>                      if (advanced_overflow) then PSW.AV = 1 else PSW.AV = 0;</p>
SAV	<p>if (advanced_overflow) then PSW.SAV = 1 else PSW.SAV = PSW.SAV;</p>

### Examples

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

### See Also

[ABSS.H](#), [ABSDIF.B](#), [ABSDIF.H](#), [ABSDIFS.H](#)

## ABSDIF Absolute Value of Difference

### Description

Put the absolute value of the difference between D[a] and either D[b] (instruction format RR) or const9 (instruction format RC) in D[c]; i.e. if the contents of data register D[a] are greater than either D[b] (format RR) or const9 (format RC), then subtract D[b] (format RR) or const9 (format RC) from D[a] and put the result in data register D[c]; otherwise subtract D[a] from either D[b] (format RR) or const9 (format RC) and put the result in D[c]. The operands are treated as signed 32-bit integers, and the const9 value is sign-extended.

### ABSDIF D[c], D[a], const9 (RC)

31	28 27	21 20	12 11	8 7	0
c	0EH	const9	a	8BH	

result = (D[a] > sign\_ext(const9)) ? D[a] - sign\_ext(const9) : sign\_ext(const9) - D[a];

D[c] = result[31:0];

### ABSDIF D[c], D[a], D[b] (RR)

31	28 27	20 19	18 17	16 15	12 11	8 7	0
c	0EH	-	-	b	a	0BH	

result = (D[a] > D[b]) ? D[a] - D[b] : D[b] - D[a];

D[c] = result[31:0];

### Status Flags

C	Not set by this instruction.
V	overflow = (result > 7FFFFFFFH) OR (result < -80000000H); if (overflow) then PSW.V = 1 else PSW.V = 0;
SV	if (overflow) then PSW.SV = 1 else PSW.SV = PSW.SV;
AV	advanced_overflow = result[31] ^ result[30]; if (advanced_overflow) then PSW.AV = 1 else PSW.AV = PSW.0;
SAV	if (advanced_overflow) then PSW.SAV = 1 else PSW.SAV = PSW.SAV;

### Examples

absdif d3, d1, d2

absdif d3, d1, #126

See Also

[ABS](#), [ABSS](#), [ABSDIFS](#)

See Also Addendum for TriCore Arch Manual, Vol2, V1.3.8, pages 493/494

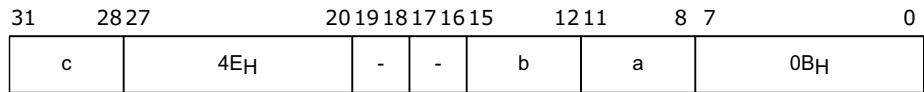
**ABSDIF.B**  
Absolute Value of Difference Packed Byte  
**ABSDIF.H**  
Absolute Value of Difference Packed Half-word

**Description**

Compute the absolute value of the difference between the corresponding bytes (ABSDIF.B) or half-words (ABSDIF.H) of D[a] and D[b], and put each result in the corresponding byte or half-word of D[c]. The operands are treated as signed, 8-bit or 16-bit integers.

The overflow condition is calculated for each byte (ABSDIF.B) or half-word (ABSDIF.H) of the packed register.

**ABSDIF.B**                      **D[c], D[a], D[b] (RR)**



$$\text{result\_byte3} = (D[a][31:24] > D[b][31:24]) ? (D[a][31:24] - D[b][31:24]) : (D[b][31:24] - D[a][31:24]);$$

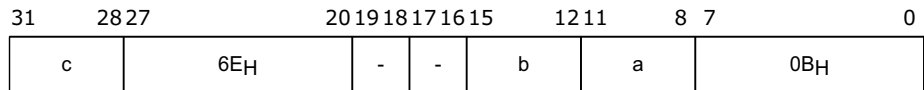
$$\text{result\_byte2} = (D[a][23:16] > D[b][23:16]) ? (D[a][23:16] - D[b][23:16]) : (D[b][23:16] - D[a][23:16]);$$

$$\text{result\_byte1} = (D[a][15:8] > D[b][15:8]) ? (D[a][15:8] - D[b][15:8]) : (D[b][15:8] - D[a][15:8]);$$

$$\text{result\_byte0} = (D[a][7:0] > D[b][7:0]) ? (D[a][7:0] - D[b][7:0]) : (D[b][7:0] - D[a][7:0]);$$

$$D[c] = \{\text{result\_byte3}[7:0], \text{result\_byte2}[7:0], \text{result\_byte1}[7:0], \text{result\_byte0}[7:0]\};$$

**ABSDIF.H**                      **D[c], D[a], D[b] (RR)**



$$\text{result\_halfword1} = (D[a][31:16] > D[b][31:16]) ? (D[a][31:16] - D[b][31:16]) : (D[b][31:16] - D[a][31:16]);$$

$$\text{result\_halfword0} = (D[a][15:0] > D[b][15:0]) ? (D[a][15:0] - D[b][15:0]) : (D[b][15:0] - D[a][15:0]);$$

$$D[c] = \{\text{result\_halfword1}[15:0], \text{result\_halfword0}[15:0]\};$$

**Status Flags**

C	Not set by these instructions.
---	--------------------------------

V	<p>ABSDIF.B  <math>ov\_byte3 = (result\_byte3 &gt; 7F_H) \text{ OR } (result\_byte3 &lt; -80_H);</math>  <math>ov\_byte2 = (result\_byte2 &gt; 7F_H) \text{ OR } (result\_byte2 &lt; -80_H);</math>  <math>ov\_byte1 = (result\_byte1 &gt; 7F_H) \text{ OR } (result\_byte1 &lt; -80_H);</math>  <math>ov\_byte0 = (result\_byte0 &gt; 7F_H) \text{ OR } (result\_byte0 &lt; -80_H);</math>  <math>overflow = ov\_byte3 \text{ OR } ov\_byte2 \text{ OR } ov\_byte1 \text{ OR } ov\_byte0;</math>                      if (overflow) then PSW.V = 1 else PSW.V = 0;                      ABSDIF.H  <math>ov\_halfword1 = (result\_halfword1 &gt; 7FFF_H) \text{ OR } (result\_halfword1 &lt; -8000_H);</math>  <math>ov\_halfword0 = (result\_halfword0 &gt; 7FFF_H) \text{ OR } (result\_halfword0 &lt; -8000_H);</math>  <math>overflow = ov\_halfword1 \text{ OR } ov\_halfword0;</math>                      if (overflow) then PSW.V = 1 else PSW.V = 0;</p>
SV	<p>if (overflow) then PSW.SV = 1 else PSW.SV = PSW.SV;</p>
AV	<p>ABSDIF.B  <math>aov\_byte3 = result\_byte3[7] \wedge result\_byte3[6];</math>  <math>aov\_byte2 = result\_byte2[7] \wedge result\_byte2[6];</math>  <math>aov\_byte1 = result\_byte1[7] \wedge result\_byte1[6];</math>  <math>aov\_byte0 = result\_byte0[7] \wedge result\_byte0[6];</math>  <math>advanced\_overflow = aov\_byte3 \text{ OR } aov\_byte2 \text{ OR } aov\_byte1 \text{ OR } aov\_byte0;</math>                      if (advanced_overflow) then PSW.AV = 1 else PSW.AV = 0;                      ABSDIF.H  <math>aov\_halfword1 = result\_halfword1[15] \wedge result\_halfword1[14];</math>  <math>aov\_halfword0 = result\_halfword0[15] \wedge result\_halfword0[14];</math>  <math>advanced\_overflow = aov\_halfword1 \text{ OR } aov\_halfword0;</math>                      if (advanced_overflow) then PSW.AV = 1 else PSW.AV = 0;</p>
SAV	<p>if (advanced_overflow) then PSW.SAV = 1 else PSW.SAV = PSW.SAV;</p>

**Examples**

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

**See Also**

**[ABS.B](#), [ABS.H](#), [ABSS.H](#), [ABSDIFS.H](#)**



## ABSDIFS

### Absolute Value of Difference with Saturation

#### Description

Put the absolute value of the difference between D[a] and either D[b] (instruction format RR) or const9 (instruction format RC) in D[c]; i.e. if the contents of data register D[a] are greater than either D[b] (format RR) or const9 (format RC), then subtract D[b] (format RR) or const9 (format RC) from D[a] and put the result in data register D[c]; otherwise, subtract D[a] from either D[b] (format RR) or const9 (format RC) and put the result in D[c]. The operands are treated as signed, 32-bit integers, with saturation on signed overflow (ssov). The const9 value is sign-extended.

#### ABSDIFS D[c], D[a], const9 (RC)

31	28 27	21 20	12 11	8 7	0
c	0FH	const9	a	8BH	

result = (D[a] > sign\_ext(const9)) ? D[a] - sign\_ext(const9) : sign\_ext(const9) - D[a];

D[c] = ssov(result, 32);

#### ABSDIFS D[c], D[a], D[b] (RR)

31	28 27	20 19 18 17 16 15	12 11	8 7	0
c	0FH	-	-	b	a
				8BH	

result = (D[a] > D[b]) ? D[a] - D[b] : D[b] - D[a];

D[c] = ssov(result, 32);

#### Status Flags

C	Not set by this instruction.
V	overflow = (result > 7FFFFFFFH) OR (result < -80000000H); if (overflow) then PSW.V = 1 else PSW.V = 0;
SV	if (overflow) then PSW.SV = 1 else PSW.SV = PSW.SV;
AV	advanced_overflow = result[31] ^ result[30]; if (advanced_overflow) then PSW.AV = 1 else PSW.AV = 0;
SAV	if (advanced_overflow) then PSW.SAV = 1 else PSW.SAV = PSW.SAV;

#### Examples

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

See Also

[ABS](#), [ABSDIF](#), [ABSS](#)

See Also Addendum for TriCore Arch Manual, Vol2, V1.3.8, pages 493/494

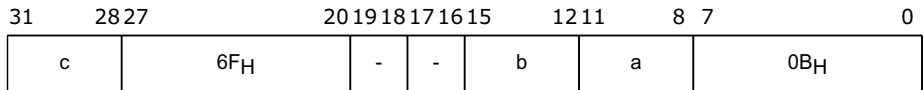
## ABSDIFS.H

### Absolute Value of Difference Packed Half-word with Saturation

#### Description

Compute the absolute value of the difference of the corresponding half-words of D[a] and D[b] and put each result in the corresponding half-word of D[c]. The operands are treated as signed 16-bit integers, with saturation on signed overflow. The overflow condition is calculated for each half-word of the packed quantity.

#### ABSDIFS.H                      D[c], D[a], D[b] (RR)



result\_halfword1 = (D[a][31:16] > D[b][31:16]) ? (D[a][31:16] - D[b][31:16]) : (D[b][31:16] - D[a][31:16]);

result\_halfword0 = (D[a][15:0] > D[b][15:0]) ? (D[a][15:0] - D[b][15:0]) : (D[b][15:0] - D[a][15:0]);

D[c] = {ssov(result\_halfword1, 16), ssov(result\_halfword0, 16)};

#### Status Flags

C	Not set by this instruction.
V	ov_halfword1 = (result_halfword1 > 7FFF <sub>H</sub> ) OR (result_halfword1 < -8000 <sub>H</sub> ); ov_halfword0 = (result_halfword0 > 7FFF <sub>H</sub> ) OR (result_halfword0 < -8000 <sub>H</sub> ); overflow = ov_halfword1 OR ov_halfword0; if (overflow) then PSW.V = 1 else PSW.V = 0;
SV	if (overflow) then PSW.SV = 1 else PSW.SV = PSW.SV;
AV	aov_halfword1 = result_halfword1[15] ^ result_halfword1[14]; aov_halfword0 = result_halfword0[15] ^ result_halfword0[14]; advanced_overflow = aov_halfword1 OR aov_halfword0; if (advanced_overflow) then PSW.AV = 1 else PSW.AV = 0;
SAV	if (advanced_overflow) then PSW.SAV = 1 else PSW.SAV = PSW.SAV;

#### Examples

```
absdifs.h    d3, d1, d2
```

#### See Also

[ABS.B](#), [ABS.H](#), [ABSS.H](#), [ABSDIFS.H](#)

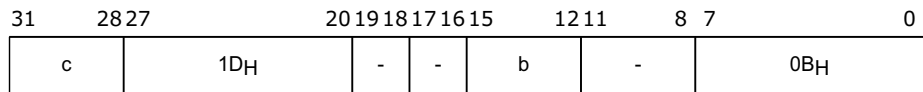
## ABSS Absolute Value with Saturation

### Description

Put the absolute value of data register D[b] in data register D[c]; If the contents of D[b] are greater than or equal to zero, then copy it to D[c]; otherwise change the sign of D[b] and copy it to D[c]. The operands are treated as signed, 32-bit integers, with saturation on sign overflow.

If  $D[b] = 80000000_H$  (the maximum negative value), then  $D[c] = 7FFFFFFF_H$ .

### ABSS                      D[c], D[b] (RR)



result = (D[b] >= 0) ? D[b] : (0 - D[b]);

D[c] = ssov(result, 32);

### Status Flags

C	Not set by this instruction.
V	overflow = (result > 7FFFFFFF <sub>H</sub> ) OR (result < -80000000 <sub>H</sub> ); if (overflow) then PSW.V = 1 else PSW.V = 0;
SV	if (overflow) then PSW.SV = 1 else PSW.SV = PSW.SV;
AV	advanced_overflow = result[31] ^ result[30]; if (advanced_overflow) then PSW.AV = 1 else PSW.AV = 0;
SAV	if (advanced_overflow) then PSW.SAV = 1 else PSW.SAV = PSW.SAV;

### Examples

abss d3, d1

### See Also

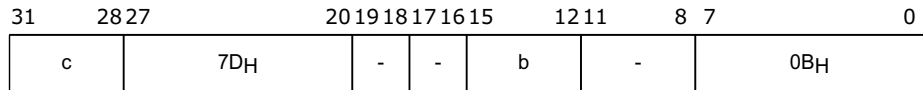
[ABS](#), [ABSDIF](#), [ABSDIFS](#)

## ABSS.H Absolute Value Packed Half-word with Saturation

### Description

Put the absolute value of each byte or half-word in data register D[b] in the corresponding byte or half-word of data register D[c]. The operands are treated as signed 8-bit or 16-bit integers, with saturation on signed overflow. The overflow condition is calculated for each byte or half-word of the packed register. Overflow occurs only if D[b][31:16] or D[b][15:0] has the maximum negative value of 8000<sub>H</sub> and the saturation yields 7FFF<sub>H</sub>.

### ABSS.H                      D[c], D[b] (RR)



result\_halfword1 = (D[b][31:16] >= 0) ? D[b][31:16] : (0 - D[b][31:16]);

result\_halfword0 = (D[b][15:0] >= 0) ? D[b][15:0] : (0 - D[b][15:0]);

D[c] = {ssov(result\_halfword1, 16), ssov(result\_halfword0, 16)};

### Status Flags

C	Not set by this instruction.
V	ov_halfword1 = (result_halfword1 > 7FFF <sub>H</sub> ) OR (result_halfword1 < -8000 <sub>H</sub> ); ov_halfword0 = (result_halfword0 > 7FFF <sub>H</sub> ) OR (result_halfword0 < -8000 <sub>H</sub> ); overflow = ov_halfword1 OR ov_halfword0; if (overflow) then PSW.V = 1 else PSW.V = 0;
SV	if (overflow) then PSW.SV = 1 else PSW.SV = PSW.SV;
AV	aov_halfword1 = result_halfword1[15] ^ result_halfword1[14]; aov_halfword0 = result_halfword0[15] ^ result_halfword0[14]; advanced_overflow = aov_halfword1 OR aov_halfword0; if (advanced_overflow) then PSW.AV = 1 else PSW.AV = 0;
SAV	if (advanced_overflow) then PSW.SAV = 1 else PSW.SAV = PSW.SAV;

### Examples

abss.h     d3, d1

### See Also

**ABS.B, ABS.H, ABSDIF.B, ABSDIF.H, ABSDIFS.H**

## ADD

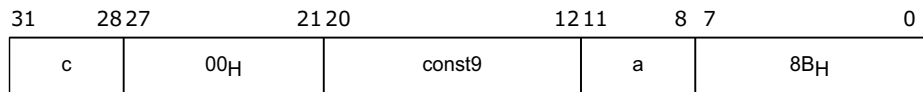
### Add

#### Description

Add the contents of data register D[a] to the contents of either data register D[b] (instruction format RR) or const9 (instruction format RC) and put the result in data register D[c]. The operands are treated as 32-bit integers, and the const9 value is sign-extended before the addition is performed.

Add the contents of either data register D[a] or D[15] to the contents of data register D[b] or const4, and put the result in either data register D[a] or D[15]. The operands are treated as 32-bit signed integers, and the const4 value is sign-extended before the addition is performed.

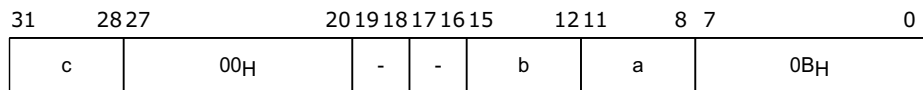
#### ADD D[c], D[a], const9 (RC)



result = D[a] + sign\_ext(const9);

D[c] = result[31:0];

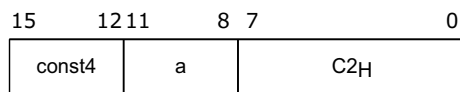
#### ADD D[c], D[a], D[b] (RR)



result = D[a] + D[b];

D[c] = result[31:0];

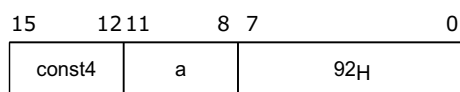
#### ADD D[a], const4 (SRC)



result = D[a] + sign\_ext(const4);

D[a] = result[31:0];

#### ADD D[a], D[15], const4 (SRC)



```
result = D[15] + sign_ext(const4);
D[a] = result[31:0];
```

**ADD D[15], D[a], const4 (SRC)**

15 12 11 8 7 0

const4	a	9AH
--------	---	-----

```
result = D[a] + sign_ext(const4);
D[15] = result[31:0];
```

**ADD D[a], D[b] (SRR)**

15 12 11 8 7 0

b	a	42H
---	---	-----

```
result = D[a] + D[b];
D[a] = result[31:0];
```

**ADD D[a], D[15], D[b] (SRR)**

15 12 11 8 7 0

b	a	12H
---	---	-----

```
result = D[15] + D[b];
D[a] = result[31:0];
```

**ADD D[15], D[a], D[b] (SRR)**

15 12 11 8 7 0

b	a	1AH
---	---	-----

```
result = D[a] + D[b];
D[15] = result[31:0];
```

**Status Flags**

C	Not set by this instruction.
V	overflow = (result > 7FFFFFFFH) OR (result < -80000000H); if (overflow) then PSW.V = 1 else PSW.V = 0;
SV	if (overflow) then PSW.SV = 1 else PSW.SV = PSW.SV;

AV	advanced_overflow = result[31] ^ result[30]; if (advanced_overflow) then PSW.AV = 1 else PSW.AV = 0;
SAV	if (advanced_overflow) then PSW.SAV = 1 else PSW.SAV = PSW.SAV;

### Examples

```
add d3, d1, d2
add d3, d1, #126
```

```
add d1, d2
add d1, #6
add d15, d1, d2
add d15, d1, #6
add d1, d15, d2
add d1, d15, #6
```

### See Also

[ADDC](#), [ADDI](#), [ADDIH](#), [ADDS](#), [ADDS.U](#), [ADDX](#)

See Also Addendum for TriCore Arch Manual, Vol2, V1.3.8, pages 493/494



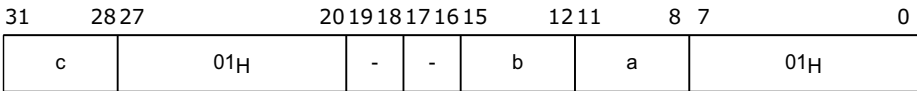
## ADD.A Add Address

### Description

Add the contents of address register A[a] to the contents of address register A[b] and put the result in address register A[c].

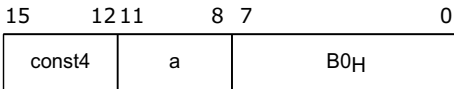
Add the contents of address register A[a] to the contents of either address register A[b] or const4 and put the result in address register A[a].

### ADD.A                      A[c], A[a], A[b] (RR)



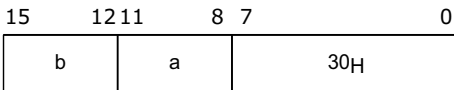
$A[c] = A[a] + A[b];$

### ADD.A                      A[a], const4 (SRC)



$A[a] = A[a] + \text{sign\_ext}(\text{const4});$

### ADD.A                      A[a], A[b] (SRR)



$A[a] = A[a] + A[b];$

### Status Flags

C	Not set by this instruction.
V	Not set by this instruction.
SV	Not set by this instruction.
AV	Not set by this instruction.
SAV	Not set by this instruction.

### Examples

add.a    a3, a4, a2

```
add.a    a1, a2  
add.a    a3, 6
```

**See Also**

[ADDIH.A](#), [ADDSC.A](#), [ADDSC.AT](#), [SUB.A](#)

See Also Addendum for TriCore Arch Manual, Vol2, V1.3.8, pages 493/494

### ADD.B

Add Packed Byte

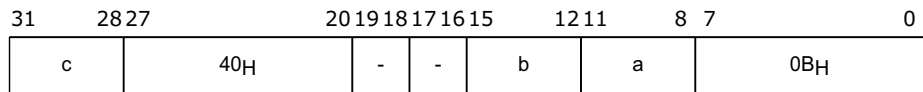
### ADD.H

Add Packed Half-word

#### Description

Add the contents of each byte (ADD.B) or half-word (ADD.H) of D[a] and D[b] and put the result in each corresponding byte or half-word of D[c]. The overflow condition is calculated for each byte or half-word of the packed quantity.

#### ADD.B D[c], D[a], D[b] (RR)



result\_byte3 = D[a][31:24] + D[b][31:24];

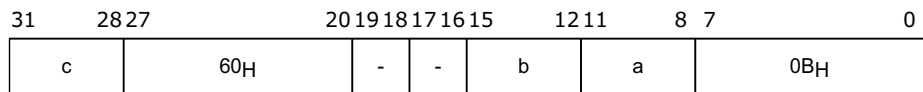
result\_byte2 = D[a][23:16] + D[b][23:16];

result\_byte1 = D[a][15:8] + D[b][15:8];

result\_byte0 = D[a][7:0] + D[b][7:0];

D[c] = {result\_byte3[7:0], result\_byte2[7:0], result\_byte1[7:0], result\_byte0[7:0]};

#### ADD.H D[c], D[a], D[b] (RR)



result\_halfword1 = D[a][31:16] + D[b][31:16];

result\_halfword0 = D[a][15:0] + D[b][15:0];

D[c] = {result\_halfword1[15:0], result\_halfword0[15:0]};

#### Status Flags

C	Not set by these instructions.
---	--------------------------------

V	<p>ADD.B  <math>ov\_byte3 = (result\_byte3 &gt; 7F_H) \text{ OR } (result\_byte3 &lt; -80_H);</math>  <math>ov\_byte2 = (result\_byte2 &gt; 7F_H) \text{ OR } (result\_byte2 &lt; -80_H);</math>  <math>ov\_byte1 = (result\_byte1 &gt; 7F_H) \text{ OR } (result\_byte1 &lt; -80_H);</math>  <math>ov\_byte0 = (result\_byte0 &gt; 7F_H) \text{ OR } (result\_byte0 &lt; -80_H);</math>  <math>overflow = ov\_byte3 \text{ OR } ov\_byte2 \text{ OR } ov\_byte1 \text{ OR } ov\_byte0;</math>                      if (overflow) then PSW.V = 1 else PSW.V = 0;</p> <p>ADD.H  <math>ov\_halfword1 = (result\_halfword1 &gt; 7FFF_H) \text{ OR } (result\_halfword1 &lt; -8000_H);</math>  <math>ov\_halfword0 = (result\_halfword0 &gt; 7FFF_H) \text{ OR } (result\_halfword0 &lt; -8000_H);</math>  <math>overflow = ov\_halfword1 \text{ OR } ov\_halfword0;</math>                      if (overflow) then PSW.V = 1 else PSW.V = 0;</p>
SV	<p>if (overflow) then PSW.SV = 1 else PSW.SV = PSW.SV;</p>
AV	<p>ADD.B  <math>aov\_byte3 = result\_byte3[7] \wedge result\_byte3[6];</math>  <math>aov\_byte2 = result\_byte2[7] \wedge result\_byte2[6];</math>  <math>aov\_byte1 = result\_byte1[7] \wedge result\_byte1[6];</math>  <math>aov\_byte0 = result\_byte0[7] \wedge result\_byte0[6];</math>  <math>advanced\_overflow = aov\_byte3 \text{ OR } aov\_byte2 \text{ OR } aov\_byte1 \text{ OR } aov\_byte0;</math>                      if (advanced_overflow) then PSW.AV = 1 else PSW.AV = 0;</p> <p>ADD.H  <math>aov\_halfword1 = result\_halfword1[15] \wedge result\_halfword1[14];</math>  <math>aov\_halfword0 = result\_halfword0[15] \wedge result\_halfword0[14];</math>  <math>advanced\_overflow = aov\_halfword1 \text{ OR } aov\_halfword0;</math>                      if (advanced_overflow) then PSW.AV = 1 else PSW.AV = 0;</p>
SAV	<p>if (advanced_overflow) then PSW.SAV = 1 else PSW.SAV = PSW.SAV;</p>

**Examples**

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

**See Also**

**[ADD.H](#), [ADDS.H](#), [ADDS.HU](#)**

## ADDC

### Add with Carry

#### Description

Add the contents of data register D[a] to the contents of either data register D[b] (instruction format RR) or const9 (instruction format RC) plus the carry bit, and put the result in data register D[c]. The operands are treated as 32-bit integers. The value const9 is sign-extended before the addition is performed. The PSW carry bit is set to the value of the ALU carry out.

#### ADDC D[c], D[a], const9 (RC)

31	28 27	21 20	12 11	8 7	0
c	05 <sub>H</sub>	const9	a	8B <sub>H</sub>	

result = D[a] + sign\_ext(const9) + PSW.C;

D[c] = result[31:0];

carry\_out = carry(D[a], sign\_ext(const9), PSW.C);

#### ADDC D[c], D[a], D[b] (RR)

31	28 27	20 19 18 17 16 15	12 11	8 7	0
c	05 <sub>H</sub>	- -	b	a	0B <sub>H</sub>

result = D[a] + D[b] + PSW.C;

D[c] = result[31:0];

carry\_out = carry(D[a], D[b], PSW.C);

#### Status Flags

C	PSW.C = carry_out;
V	overflow = (result > 7FFFFFFF <sub>H</sub> ) OR (result < -80000000 <sub>H</sub> ); if (overflow) then PSW.V = 1 else PSW.V = 0;
SV	if (overflow) then PSW.SV = 1 else PSW.SV = PSW.SV;
AV	advanced_overflow = result[31] ^ result[30]; if (advanced_overflow) then PSW.AV = 1 else PSW.AV = 0;
SAV	if (advanced_overflow) then PSW.SAV = 1 else PSW.SAV = PSW.SAV;

#### Examples

```
addc d3, d1, d2
```

addc d3, d1, #126

**See Also**

[ADD](#), [ADDI](#), [ADDIH](#), [ADDS](#), [ADDS.U](#), [ADDX](#)

See Also Addendum for TriCore Arch Manual, Vol2, V1.3.8, pages 493/494

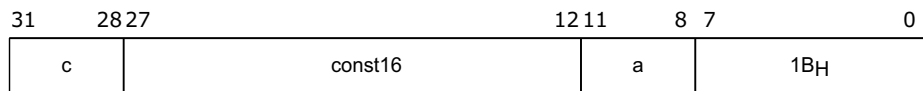
## ADDI

### Add Immediate

#### Description

Add the contents of data register D[a] to the value const16, and put the result in data register D[c]. The operands are treated as 32-bit signed integers. The value const16 is sign-extended before the addition is performed.

#### ADDI                      D[c], D[a], const16 (RLC)



result = D[a] + sign\_ext(const16);

D[c] = result[31:0];

#### Status Flags

C	Not set by this instruction.
V	overflow = (result > 7FFFFFFF <sub>H</sub> ) OR (result < -80000000 <sub>H</sub> ); if (overflow) then PSW.V = 1 else PSW.V = 0;
SV	if (overflow) then PSW.SV = 1 else PSW.SV = PSW.SV;
AV	advanced_overflow = result[31] ^ result[30]; if (advanced_overflow) then PSW.AV = 1 else PSW.AV = 0;
SAV	if (advanced_overflow) then PSW.SAV = 1 else PSW.SAV = PSW.SAV;

#### Examples

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

#### See Also

[ADD](#), [ADDC](#), [ADDIH](#), [ADDS](#), [ADDS.U](#), [ADDX](#)

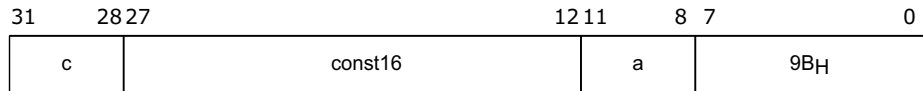
## ADDIH

### Add Immediate High

#### Description

Left-shift const16 by 16 bits, add the contents of data register D[a], and put the result in data register D[c]. The operands are treated as signed integers.

#### ADDIH                      D[c], D[a], const16 (RLC)



result = D[a] + {const16, 16'h0000};

D[c] = result[31:0];

#### Status Flags

C	Not set by this instruction.
V	overflow = (result > 7FFFFFFF <sub>H</sub> ) OR (result < -80000000 <sub>H</sub> ); if (overflow) then PSW.V = 1 else PSW.V = 0;
SV	if (overflow) then PSW.SV = 1 else PSW.SV = PSW.SV;
AV	advanced_overflow = result[31] ^ result[30]; if (advanced_overflow) then PSW.AV = 1 else PSW.AV = 0;
SAV	if (advanced_overflow) then PSW.SAV = 1 else PSW.SAV = PSW.SAV;

#### Examples

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

#### See Also

[ADD](#), [ADDC](#), [ADDI](#), [ADDS](#), [ADDS.U](#), [ADDX](#)



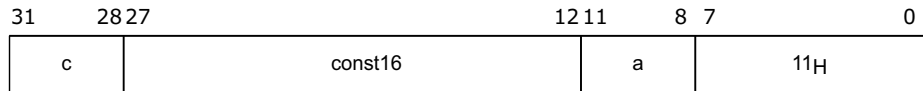
## ADDIH.A

### Add Immediate High to Address

#### Description

Left-shift const16 by 16 bits, add the contents of address register A[a], and put the result in address register A[c].

#### ADDIH.A                    A[c], A[a], const16 (RLC)



$$A[c] = A[a] + \{const16, 16'h0000\};$$

#### Status Flags

C	Not set by this instruction.
V	Not set by this instruction.
SV	Not set by this instruction.
AV	Not set by this instruction.
SAV	Not set by this instruction.

#### Examples

```
addih.a    a3, a4, -14526
```

#### See Also

[ADD.A](#), [ADDSC.A](#), [ADDSC.AT](#), [SUB.A](#)

## ADDS

### Add Signed with Saturation

#### Description

Add the contents of data register D[a] to the value in either data register D[b] (instruction format RR) or const9 (instruction format RC) and put the result in data register D[c]. The operands are treated as signed, 32-bit integers, with saturation on signed overflow. The value const9 is sign-extended before the addition is performed.

Add the contents of data register D[b] to the contents of data register D[a] and put the result in data register D[a]. The operands are treated as signed 32-bit integers, with saturation on signed overflow.

#### ADDS                      D[c], D[a], const9 (RC)

	31	28 27	21 20	12 11	8	7	0
	c	02H	const9	a	8BH		

result = D[a] + sign\_ext(const9);

D[c] = ssov(result, 32);

#### ADDS                      D[c], D[a], D[b] (RR)

	31	28 27	20	19	18	17	16	15	12 11	8	7	0
	c	02H	-	-	b	a	0BH					

result = D[a] + D[b];

D[c] = ssov(result, 32);

#### ADDS                      D[a], D[b], (SRR)

	15	12 11	8	7	0	
	b	a	22H			

result = D[a] + D[b];

D[a] = ssov(result, 32);

#### Status Flags

C	Not set by this instruction.
V	overflow = (result > 7FFFFFFFH) OR (result < -80000000H); if (overflow) then PSW.V = 1 else PSW.V = 0;

SV	if (overflow) then PSW.SV = 1 else PSW.SV = PSW.SV;
AV	advanced_overflow = result[31] ^ result[30]; if (advanced_overflow) then PSW.AV = 1 else PSW.AV = 0;
SAV	if (advanced_overflow) then PSW.SAV = 1 else PSW.SAV = PSW.SAV;

### Examples

```
adds    d3, d1, d2
adds    d3, d1, #126
```

```
adds    d3, d1
```

### See Also

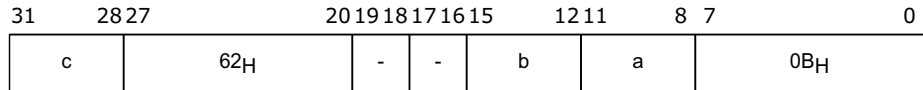
[ADD](#), [ADDC](#), [ADDI](#), [ADDIH](#), [ADDS.U](#), [ADDX](#)

**ADDS.H**  
Add Signed Packed Half-word with Saturation  
**ADDS.HU**  
Add Unsigned Packed Half-word with Saturation

**Description**

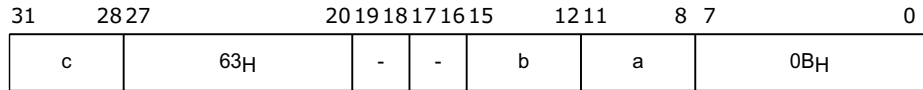
Add the contents of each half-word of D[a] and D[b] and put the result in each corresponding half-word of D[c], with saturation on signed overflow (ADDS.H) or saturation on unsigned overflow (ADDS.HU). The overflow (PSW.V) and advance overflow (PSW.AV) conditions are calculated for each half-word of the packed quantity.

**ADDS.H**                      **D[c], D[a], D[b] (RR)**



result\_halfword1 = D[a][31:16] + D[b][31:16];  
 result\_halfword0 = D[a][15:0] + D[b][15:0];  
 D[c] = {ssov(result\_halfword1, 16), ssov(result\_halfword0, 16)};

**ADDS.HU**                      **D[c], D[a], D[b] (RR)**



result\_halfword1 = D[a][31:16] + D[b][31:16]; // unsigned addition  
 result\_halfword0 = D[a][15:0] + D[b][15:0]; // unsigned addition  
 D[c] = {suov(result\_halfword1, 16), suov(result\_halfword0, 16)};

**Status Flags**

C	Not set by these instructions.
---	--------------------------------

See Also Addendum for TriCore Arch Manual, Vol2, V1.3.8, pages 493/494

V	<p>ADDS.H  <math>ov\_halfword1 = (result\_halfword1 &gt; 7FFF_H) \text{ OR } (result\_halfword1 &lt; -8000_H);</math>  <math>ov\_halfword0 = (result\_halfword0 &gt; 7FFF_H) \text{ OR } (result\_halfword0 &lt; -8000_H);</math>  <math>overflow = ov\_halfword1 \text{ OR } ov\_halfword0;</math>                      if (overflow) then PSW.V = 1 else PSW.V = 0;</p> <p>ADDS.HU  <math>ov\_halfword1 = (result\_halfword1 &gt; FFFF_H) \text{ OR } (result\_halfword1 &lt; 0000_H);</math>  <math>ov\_halfword0 = (result\_halfword0 &gt; FFFF_H) \text{ OR } (result\_halfword0 &lt; 0000_H);</math>  <math>overflow = ov\_halfword1 \text{ OR } ov\_halfword0;</math>                      if (overflow) then PSW.V = 1 else PSW.V = 0;</p>
SV	<p>if (overflow) then PSW.SV = 1 else PSW.SV = PSW.SV;</p>
AV	<p><math>aov\_halfword1 = result\_halfword1[15] \wedge result\_halfword1[14];</math>  <math>aov\_halfword0 = result\_halfword0[15] \wedge result\_halfword0[14];</math>  <math>advanced\_overflow = aov\_halfword1 \text{ OR } aov\_halfword0;</math>                      if (advanced_overflow) then PSW.AV = 1 else PSW.AV = 0;</p>
SAV	<p>if (advanced_overflow) then PSW.SAV = 1 else PSW.SAV = PSW.SAV;</p>

**Examples**

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

**See Also**

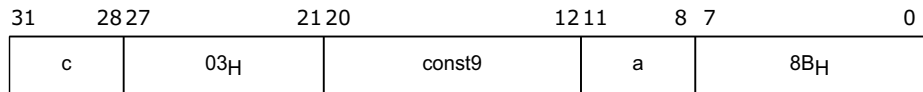
[ADD.B](#), [ADD.H](#)

## ADDS.U Add Unsigned with Saturation

### Description

Add the contents of data register D[a] to the contents of either data register D[b] (instruction format RR) or const9 (instruction format RC) and put the result in data register D[c]. The operands are treated as unsigned 32-bit integers, with saturation on unsigned overflow. The const9 value is sign-extended.

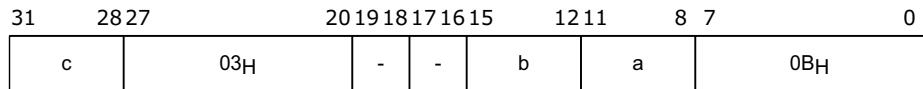
### ADDS.U                    D[c], D[a], const9 (RC)



result = D[a] + sign\_ext(const9); // unsigned addition

D[c] = suov(result, 32);

### ADDS.U                    D[c], D[a], D[b] (RR)



result = D[a] + D[b]; // unsigned addition

D[c] = suov(result, 32);

### Status Flags

C	Not set by this instruction.
V	overflow = (result > FFFFFFFF <sub>H</sub> ) OR (result < 00000000 <sub>H</sub> ); if (overflow) then PSW.V = 1 else PSW.V = 0;
SV	if (overflow) then PSW.SV = 1 else PSW.SV = PSW.SV;
AV	advanced_overflow = result[31] ^ result[30]; if (advanced_overflow) then PSW.AV = 1 else PSW.AV = 0;
SAV	if (advanced_overflow) then PSW.SAV = 1 else PSW.SAV = PSW.SAV;

### Examples

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

**See Also**

**ADD, ADDC, ADDI, ADDIH, ADDS, ADDX**

See Also Addendum for TriCore Arch Manual, Vol2, V1.3.8, pages 493/494

**ADDSC.A**  
Add Scaled Index to Address

**ADDSC.AT**  
Add Bit-Scaled Index to Address

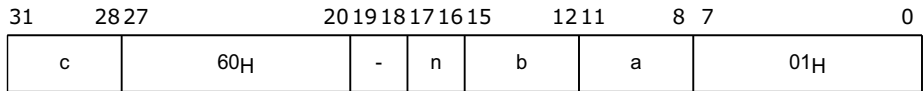
**Description**

For ADDSC.A, left-shift the contents of data register D[a] by the amount specified by n, where n can be 0, 1, 2, or 3. Add that value to the contents of address register A[b] and put the result in address register A[c].

For ADDSC.AT, right-shift the contents of D[a] by three (with sign fill). Add that value to the contents of address register A[b] and clear the bottom two bits to zero. Put the result in A[c]. The ADDSC.AT instruction generates the address of the word containing the bit indexed by D[a], starting from the base address in A[b].

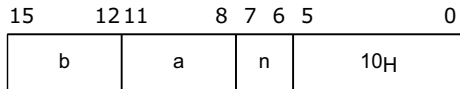
Left-shift the contents of data register D[15] by the amount specified by n, where n can be 0, 1, 2, or 3. Add that value to the contents of address register A[b] and put the result in address register A[a].

**ADDSC.A**                      **A[c], A[b], D[a], n (RR)**



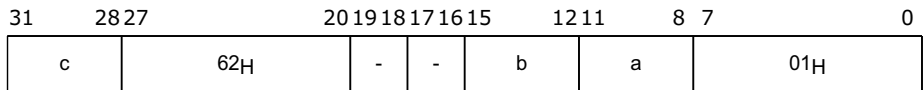
$A[c] = A[b] + (D[a] \ll n);$

**ADDSC.A**                      **A[a], A[b], D[15], n (SRRS)**



$A[a] = (A[b] + (D[15] \ll n));$

**ADDSC.AT**                      **A[c], A[b], D[a] (RR)**



$A[c] = (A[b] + (D[a] \gg 3)) \& 32'hFFFFFFFC;$

**Status Flags**

C	Not set by these instructions.
---	--------------------------------



V	Not set by these instructions.
SV	Not set by these instructions.
AV	Not set by these instructions.
SAV	Not set by these instructions.

### Examples

```
addsc.a    a3, a4, d2, #2
```

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

```
addsc.a    a3, a4, d15, #2
```

### See Also

[ADD.A](#), [ADDIH.A](#), [SUB.A](#)

## ADDX Add Extended

### Description

Add the contents of data register D[a] to the contents of either data register D[b] (instruction format RR) or const9 (instruction format RC) and put the result in data register D[c]. The operands are treated as 32-bit signed integers. The const9 value is sign-extended before the addition is performed. The PSW carry bit is set to the value of the ALU carry out.

### ADDX D[c], D[a], const9 (RC)

31	28 27	21 20	12 11	8 7	0
c	04 <sub>H</sub>	const9	a	8B <sub>H</sub>	

result = D[a] + sign\_ext(const9);

D[c] = result[31:0];

carry\_out = carry(D[a],sign\_ext(const9),0);

### ADDX D[c], D[a], D[b] (RR)

31	28 27	20 19 18 17 16 15	12 11	8 7	0
c	04 <sub>H</sub>	- -	b	a	0B <sub>H</sub>

result = D[a] + D[b];

D[c] = result[31:0];

carry\_out = carry(D[a],D[b],0);

### Status Flags

C	PSW.C = carry_out;
V	overflow = (result > 7FFFFFFF <sub>H</sub> ) OR (result < -80000000 <sub>H</sub> ); if (overflow) then PSW.V = 1 else PSW.V = 0;
SV	if (overflow) then PSW.SV = 1 else PSW.SV = PSW.SV;
AV	advanced_overflow = result[31] ^ result[30]; if (advanced_overflow) then PSW.AV = 1 else PSW.AV = 0;
SAV	if (advanced_overflow) then PSW.SAV = 1 else PSW.SAV = PSW.SAV;

### Examples

addx d3, d1, d2

addx d3, d1, #126

**See Also**

[ADD](#), [ADDC](#), [ADDI](#), [ADDIH](#), [ADDS](#), [ADDS.U](#)

See Also Addendum for TriCore Arch Manual, Vol2, V1.3.8, pages 493/494

## AND

### Bitwise AND

#### Description

Compute the bitwise AND of the contents of data register D[a] and the contents of either data register D[b] (instruction format RR) or const9 (instruction format RC) and put the result in data register D[c]. The const9 value is zero-extended.

Compute the bitwise AND of the contents of either data register D[a] (instruction format SRR) or D[15] (instruction format SC) and the contents of either data register D[b] (format SRR) or const8 (format SC), and put the result in data register D[a] (format SRR) or D[15] (format SC). The const8 value is zero-extended.

#### AND D[c], D[a], const9 (RC)

31	28 27	21 20	12 11	8 7	0
c	08H	const9	a	8FH	

D[c] = D[a] & zero\_ext(const9);

#### AND D[c], D[a], D[b] (RR)

31	28 27	20 19	18 17	16 15	12 11	8 7	0
c	08H	-	-	b	a	0FH	

D[c] = D[a] & D[b];

#### AND D[15], const8 (SC)

15	8 7	0
const8	16H	

D[15] = D[15] & zero\_ext(const8);

#### AND D[a], D[b] (SRR)

15	12 11	8 7	0
b	a	26H	

D[a] = D[a] & D[b];

### Status Flags

C	Not set by this instruction.
V	Not set by this instruction.
SV	Not set by this instruction.
AV	Not set by this instruction.
SAV	Not set by this instruction.

### Examples

```
and d3, d1, d2  
and d3, d1, #126
```

```
and d1, d2  
and d15, #126
```

### See Also

[ANDN](#), [NAND](#), [NOR](#), [NOT \(16-bit\)](#), [OR](#), [ORN](#), [XNOR](#), [XOR](#)

### AND.AND.T

Accumulating Bit Logical AND-AND

### AND.ANDN.T

Accumulating Bit Logical AND-AND-Not

### AND.NOR.T

Accumulating Bit Logical AND-NOR

### AND.OR.T

Accumulating Bit Logical AND-OR

#### Description

Compute the logical AND, ANDN, NOR or OR of the value in bit pos1 of data register D[a] and bit pos2 of D[b]. Then compute the logical AND of that result and bit 0 of D[c], and put the result in bit 0 of D[c]. All other bits in D[c] are unchanged.

#### AND.AND.T            D[c], D[a], pos1, D[b], pos2 (BIT)

31	28 27	23 22 21 20	16 15	12 11	8 7	0
c	pos2	00 <sub>H</sub>	pos1	b	a	47 <sub>H</sub>

$D[c] = \{D[c][31:1], D[c][0] \text{ AND } (D[a][pos1] \text{ AND } D[b][pos2])\};$

#### AND.ANDN.T            D[c], D[a], pos1, D[b], pos2 (BIT)

31	28 27	23 22 21 20	16 15	12 11	8 7	0
c	pos2	03 <sub>H</sub>	pos1	b	a	47 <sub>H</sub>

$D[c] = \{D[c][31:1], D[c][0] \text{ AND } (D[a][pos1] \text{ AND } !D[b][pos2])\};$

#### AND.NOR.T            D[c], D[a], pos1, D[b], pos2 (BIT)

31	28 27	23 22 21 20	16 15	12 11	8 7	0
c	pos2	02 <sub>H</sub>	pos1	b	a	47 <sub>H</sub>

$D[c] = \{D[c][31:1], D[c][0] \text{ AND } !(D[a][pos1] \text{ OR } D[b][pos2])\};$

#### AND.OR.T            D[c], D[a], pos1, D[b], pos2 (BIT)

31	28 27	23 22 21 20	16 15	12 11	8 7	0
c	pos2	01 <sub>H</sub>	pos1	b	a	47 <sub>H</sub>

$D[c] = \{D[c][31:1], D[c][0] \text{ AND } (D[a][pos1] \text{ OR } D[b][pos2])\};$

### Status Flags

C	Not set by these instructions.
V	Not set by these instructions.
SV	Not set by these instructions.
AV	Not set by these instructions.
SAV	Not set by these instructions.

### Examples

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

### See Also

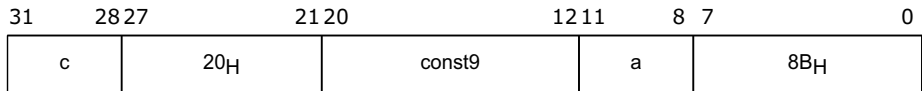
[OR.AND.T](#), [OR.ANDN.T](#), [OR.NOR.T](#), [OR.OR.T](#), [SH.AND.T](#), [SH.ANDN.T](#), [SH.NAND.T](#), [SH.NOR.T](#), [SH.OR.T](#), [SH.ORN.T](#), [SH.XNOR.T](#), [SH.XOR.T](#)

## AND.EQ Equal Accumulating

### Description

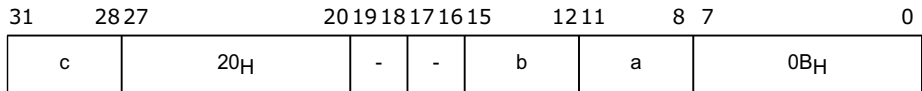
Compute the logical AND of D[c][0] and the boolean result of the equality comparison operation on the contents of data register D[a] and either data register D[b] (instruction format RR) or const9 (instruction RC). Put the result in D[c][0]. All other bits in D[c] are unchanged. The const9 value is sign-extended.

### AND.EQ D[c], D[a], const9 (RC)



$D[c] = \{D[c][31:1], D[c][0] \text{ AND } (D[a] == \text{sign\_ext}(\text{const9}))\};$

### AND.EQ D[c], D[a], D[b] (RR)



$D[c] = \{D[c][31:1], D[c][0] \text{ AND } (D[a] == D[b])\};$

### Status Flags

C	Not set by this instruction.
V	Not set by this instruction.
SV	Not set by this instruction.
AV	Not set by this instruction.
SAV	Not set by this instruction.

### Examples

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

### See Also

[OR.EQ](#), [XOR.EQ](#)

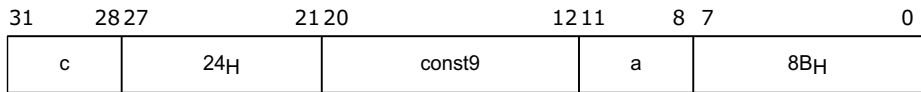


**AND.GE**  
Greater Than or Equal Accumulating  
**AND.GE.U**  
Greater Than or Equal Accumulating Unsigned

**Description**

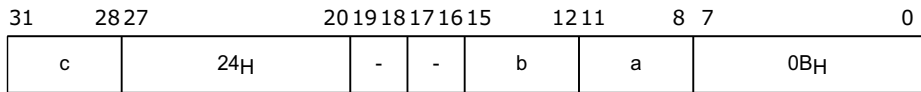
Calculate the logical AND of D[c][0] and the boolean result of the GE or GE.U operation on the contents of data register D[a] and either data register D[b] (instruction format RR) or const9 (instruction format RC). Put the result in D[c][0]. All other bits in D[c] are unchanged. D[a] and D[b] are treated as either 32-bit signed (AND.GE) or unsigned (AND.GE.U) integers. The const9 value is either sign-extended (AND.GE) or zero-extended (AND.GE.U) to 32-bits.

**AND.GE**                      **D[c], D[a], const9 (RC)**



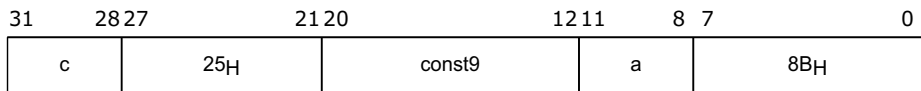
D[c] = {D[c][31:1], D[c][0] AND (D[a] >= sign\_ext(const9))};

**AND.GE**                      **D[c], D[a], D[b] (RR)**



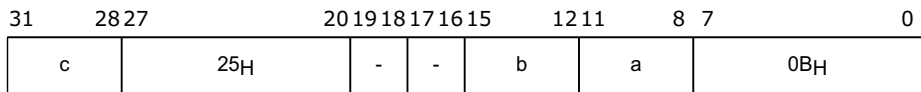
D[c] = {D[c][31:1], D[c][0] AND (D[a] >= D[b])};

**AND.GE.U**                      **D[c], D[a], const9 (RC)**



D[c] = {D[c][31:1], D[c][0] AND (D[a] >= zero\_ext(const9))}; // unsigned

**AND.GE.U**                      **D[c], D[a], D[b] (RR)**



D[c] = {D[c][31:1], D[c][0] AND (D[a] >= D[b])}; // unsigned

See Also Addendum for TriCore Arch Manual, Vol2, V1.3.8, pages 493/494

### Status Flags

C	Not set by these instructions.
V	Not set by these instructions.
SV	Not set by these instructions.
AV	Not set by these instructions.
SAV	Not set by these instructions.

### Examples

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

### See Also

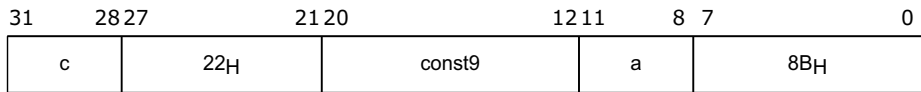
[OR.GE](#), [OR.GE.U](#), [XOR.GE](#), [XOR.GE.U](#)

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

**Description**

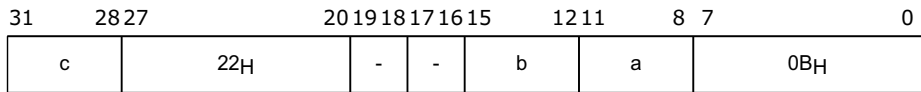
Calculate the logical AND of D[c][0] and the boolean result of the LT or LT.U operation on the contents of data register D[a] and either data register D[b] (instruction format RR) or const9 (instruction format RC). Put the result in D[c][0]. All other bits in D[c] are unchanged. D[a] and D[b] are treated as either 32-bit signed (AND.LT) or unsigned (AND.LT.U) integers. The const9 value is either sign-extended (AND.LT) or zero-extended (AND.LT.U) to 32-bits.

**AND.LT**                      **D[c], D[a], const9 (RC)**



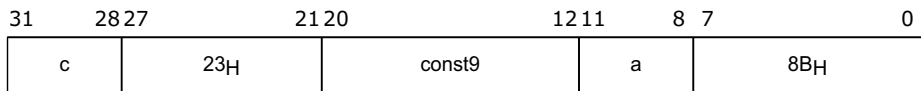
D[c] = {D[c][31:1], D[c][0] AND (D[a] < sign\_ext(const9))};

**AND.LT**                      **D[c], D[a], D[b] (RR)**



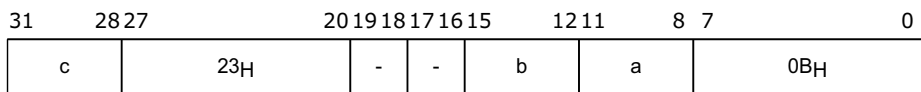
D[c] = {D[c][31:1], D[c][0] AND (D[a] < D[b])};

**AND.LT.U**                      **D[c], D[a], const9 (RC)**



D[c] = {D[c][31:1], D[c][0] AND (D[a] < zero\_ext(const9))}; // unsigned

**AND.LT.U**                      **D[c], D[a], D[b] (RR)**



D[c] = {D[c][31:1], D[c][0] AND (D[a] < D[b])}; // unsigned

See Also Addendum for TriCore Arch Manual, Vol2, V1.3.8, pages 493/494

### Status Flags

C	Not set by these instructions.
V	Not set by these instructions.
SV	Not set by these instructions.
AV	Not set by these instructions.
SAV	Not set by these instructions.

### Examples

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

### See Also

[OR.LT](#), [OR.LT.U](#), [XOR.LT](#), [XOR.LT.U](#)

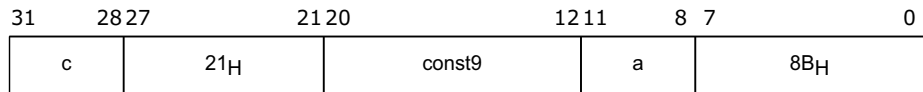
## AND.NE

### Not Equal Accumulating

#### Description

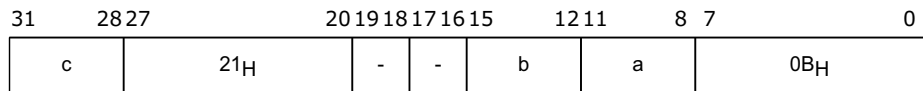
Calculate the logical AND of  $D[c][0]$  and the boolean result of the NE operation on the contents of data register  $D[a]$  and either data register  $D[b]$  (instruction format RR) or  $const9$  (instruction format RC). Put the result in  $D[c][0]$ . All other bits in  $D[c]$  are unchanged. The  $const9$  value is sign-extended.

#### AND.NE $D[c], D[a], const9$ (RC)



$$D[c] = \{D[c][31:1], D[c][0] \text{ AND } (D[a] \neq \text{sign\_ext}(const9))\};$$

#### AND.NE $D[c], D[a], D[b]$ (RR)



$$D[c] = \{D[c][31:1], D[c][0] \text{ AND } (D[a] \neq D[b])\};$$

#### Status Flags

C	Not set by this instruction.
V	Not set by this instruction.
SV	Not set by this instruction.
AV	Not set by this instruction.
SAV	Not set by this instruction.

#### Examples

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

#### See Also

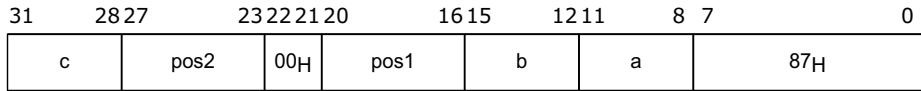
[OR.NE](#), [XOR.NE](#)

## AND.T Bit Logical AND

### Description

Compute the logical AND of bit pos1 of data register D[a] and bit pos2 of data register D[b]. Put the result in the least-significant bit of data register D[c] and clear the remaining bits of D[c] to zero.

### AND.T                      D[c], D[a], pos1, D[b], pos2 (BIT)



result = D[a][pos1] AND D[b][pos2];

D[c] = zero\_ext(result);

### Status Flags

C	Not set by this instruction.
V	Not set by this instruction.
SV	Not set by this instruction.
AV	Not set by this instruction.
SAV	Not set by this instruction.

### Examples

```
and.t    d3, d1, 7, d2, 2
```

### See Also

[ANDN.T](#), [NAND.T](#), [NOR.T](#), [OR.T](#), [ORN.T](#), [XNOR.T](#), [XOR.T](#)

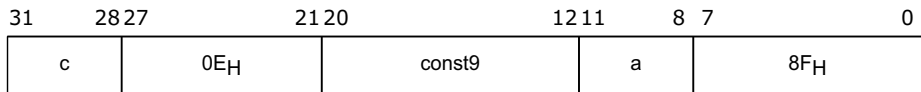
See Also Addendum for TriCore Arch Manual, Vol2, V1.3.8, pages 493/494

## ANDN Bitwise AND-Not

### Description

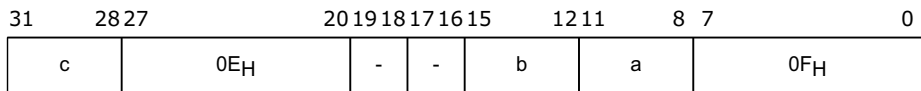
Compute the bitwise AND of the contents of data register D[a] and the ones complement of the contents of either data register D[b] (instruction format RR) or const9 (instruction format RC). Put the result in data register D[c]. The const9 value is zero-extended to 32-bits.

### ANDN D[c], D[a], const9 (RC)



$D[c] = D[a] \& \sim\text{zero\_ext}(\text{const9});$

### ANDN D[c], D[a], D[b] (RR)



$D[c] = D[a] \& \sim D[b];$

### Status Flags

C	Not set by this instruction.
V	Not set by this instruction.
SV	Not set by this instruction.
AV	Not set by this instruction.
SAV	Not set by this instruction.

### Examples

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

### See Also

[AND](#), [NAND](#), [NOR](#), [NOT \(16-bit\)](#), [OR](#), [ORN](#), [XNOR](#), [XOR](#)

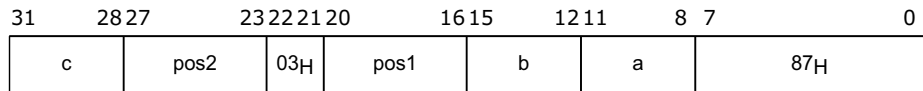
## ANDN.T

### Bit Logical AND-Not

#### Description

Compute the logical AND of bit pos1 of data register D[a] and the inverse of bit pos2 of data register D[b]. Put the result in the least-significant bit of data register D[c] and clear the remaining bits of D[c] to zero.

#### ANDN.T                      D[c], D[a], pos1, D[b], pos2 (BIT)



result = D[a][pos1] AND !D[b][pos2];

D[c] = zero\_ext(result);

#### Status Flags

C	Not set by this instruction.
V	Not set by this instruction.
SV	Not set by this instruction.
AV	Not set by this instruction.
SAV	Not set by this instruction.

#### Examples

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

#### See Also

[AND.T](#), [NAND.T](#), [NOR.T](#), [OR.T](#), [ORN.T](#), [XNOR.T](#), [XOR.T](#)



## BISR

### Begin Interrupt Service Routine

#### Description

*Note: BISR can only be executed in Supervisor mode.*

Save the lower context by storing the contents of A[2]-A[7], D[0]-D[7], and the current A[11] (return address) to the current memory location pointed to by the FCX. Set the current CPU priority number (ICR.CCPN) to the value of either const9[7:0] (instruction format RC) or const8 (instruction format SC), and enable interrupts (set ICR.IE to one).

This instruction is intended to be one of the first executed instructions in an interrupt routine. If the interrupt routine has not altered the lower context, the saved lower context is from the interrupted task. If a BISR instruction is issued at the beginning of an interrupt, then an RSLCX instruction should be performed before returning with the RFE instruction.

BISR		const9 (RC)			
31	28 27	21 20	12 11	8 7	0
-	00H	const9	-	ADH	

if (FCX == 0) trap(FCU);

tmp\_FCX = FCX;

EA = {FCX.FCXS, 6'b0, FCX.FCXO, 6'b0};

new\_FCX = M(EA, word);

M(EA, 16 \* word) = {PCXI, A[11], A[2], A[3], D[0], D[1], D[2], D[3], A[4], A[5], A[6], A[7], D[4], D[5], D[6], D[7]};

PCXI.PCPN = ICR.CCPN;

PCXI.PIE = ICR.IE;

PCXI.UJL = 0;

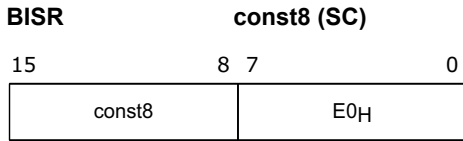
PCXI[19:0] = FCX[19:0];

FCX[19:0] = new\_FCX[19:0];

ICR.IE = 1;

ICR.CCPN = const9[7:0];

if (tmp\_FCX == LCX) trap(FCD);



tmp\_FCX = FCX;

```

if (FCX == 0) trap(FCU);
EA = {FCX.FCXS, 6'b0, FCX.FCXO, 6'b0};
new_FCX = M(EA, word);
M(EA, 16 * word) = {PCXI, A[11], A[2], A[3], D[0], D[1], D[2], D[3], A[4], A[5], A[6], A[7],
D[4], D[5], D[6], D[7]};
PCXI.PCPN = ICR.CCPN;
PCXI.PIE = ICR.IE;
PCXI.UL = 0;
PCXI[19:0] = FCX[19:0];
FCX[19:0] = new_FCX[19:0];
ICR.IE = 1;
ICR.CCPN = const8;
if (tmp_FCX == LCX) trap(FCD);
    
```

**Status Flags**

C	Not set by this instruction.
V	Not set by this instruction.
SV	Not set by this instruction.
AV	Not set by this instruction.
SAV	Not set by this instruction.

**Examples**

bisr #126

```
bisr #126
```

**See Also**

**DISABLE, ENABLE, LDLCX, LDU CX, STLCX, STUCX, SVLCX, RET, RFE, RSLCX, RSTV**

See Also Addendum for TriCore Arch Manual, Vol2, V1.3.8, pages 493/494

## BMERGE

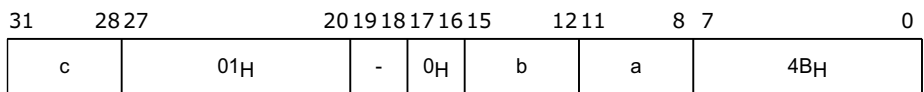
### Bit Merge

#### Description

Take the lower 16-bits of data register D[a] and move them to the odd bit positions of data register D[c]. The lower 16-bits of data register D[b] are moved to the even bit positions of data register D[c]. The upper 16-bits of D[a] and D[b] are not used.

This instruction is typically used to merge two bit streams such as commonly found in a convolutional coder.

#### BMERGE                      D[c], D[a], D[b] (RR)



D[c][31:24] = {D[a][15], D[b][15], D[a][14], D[b][14], D[a][13], D[b][13], D[a][12], D[b][12]};

D[c][23:16] = {D[a][11], D[b][11], D[a][10], D[b][10], D[a][9], D[b][9], D[a][8], D[b][8]};

D[c][15:8] = {D[a][7], D[b][7], D[a][6], D[b][6], D[a][5], D[b][5], D[a][4], D[b][4]};

D[c][7:0] = {D[a][3], D[b][3], D[a][2], D[b][2], D[a][1], D[b][1], D[a][0], D[b][0]};

#### Status Flags

C	Not set by this instruction.
V	Not set by this instruction.
SV	Not set by this instruction.
AV	Not set by this instruction.
SAV	Not set by this instruction.

#### Examples

bmerge    d0, d1, d2

#### See Also

[BSPLIT](#)

## BSPLIT Bit Split

### Description

Split data register D[a] into a data register pair E[c] such that all the even bits of D[a] are in the even register and all the odd bits of D[a] are in the odd register.

### BSPLIT                      E[c], D[a] (RR)

31	28 27		20 19 18 17 16 15		12 11	8 7	0
c	09 <sub>H</sub>	-	0 <sub>H</sub>	-	a	4B <sub>H</sub>	

E[c][63:48] = 0000<sub>H</sub>;

E[c][47:40] = {D[a][31], D[a][29], D[a][27], D[a][25], D[a][23], D[a][21], D[a][19], D[a][17]};

E[c][39:32] = {D[a][15], D[a][13], D[a][11], D[a][9], D[a][7], D[a][5], D[a][3], D[a][1]};

E[c][31:16] = 0000<sub>H</sub>;

E[c][15:8] = {D[a][30], D[a][28], D[a][26], D[a][24], D[a][22], D[a][20], D[a][18], D[a][16]};

E[c][7:0] = {D[a][14], D[a][12], D[a][10], D[a][8], D[a][6], D[a][4], D[a][2], D[a][0]};

### Status Flags

C	Not set by this instruction.
V	Not set by this instruction.
SV	Not set by this instruction.
AV	Not set by this instruction.
SAV	Not set by this instruction.

### Examples

```
bsplit e2, d5
```

### See Also

**BMERGE**

See Also Addendum for TriCore Arch Manual, Vol2, V1.3.8, pages 493/494

## CACHEA.I

### Cache Address, Invalidate

#### Description

*Note: This instruction can only be executed in Supervisor mode.*

If the cache line containing the byte memory location specified by the addressing mode is present in the L1 data cache, invalidate the line. Note that there is no writeback of any dirty data in the cache line prior to the invalidation.

If the cache line containing the byte 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. Any address register updates associated with the addressing mode are always performed regardless of the cache operation. The effective address is a virtual address when operating in virtual mode.

#### CACHEA.I                      A[b], off10 (BO)                      (Base + Short Offset Addressing Mode)

31	28 27	22 21	16 15	12 11	8 7	0
off10[9:6]	2EH	off10[5:0]	b	-		89H

EA = A[b] + sign\_ext(off10);

cache\_address\_ivld(EA);

#### CACHEA.I                      P[b] (BO)                      (Bit Reverse Addressing Mode)

31	28 27	22 21	16 15	12 11	8 7	0
-	0EH	-	b	-		A9H

index = zero\_ext(A[b+1][15:0]);

incr = zero\_ext(A[b+1][31:16]);

EA = A[b] + index;

cache\_address\_ivld(EA);

new\_index = reverse16(reverse16(index) + reverse16(incr));

A[b+1] = {incr[15:0], new\_index[15:0]};

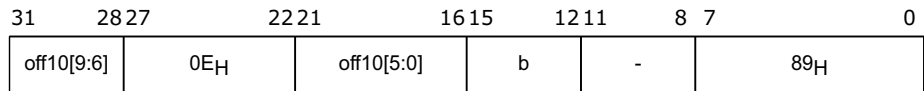
#### CACHEA.I                      P[b], off10 (BO)                      (Circular Addressing Mode)

31	28 27	22 21	16 15	12 11	8 7	0
off10[9:6]	1EH	off10[5:0]	b	-		A9H

index = zero\_ext(A[b+1][15:0]);

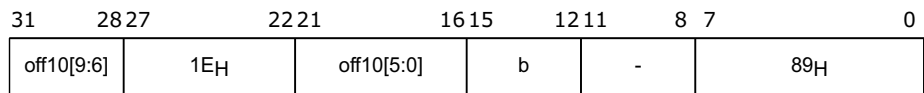
```
length = zero_ext(A[b+1][31:16]);
EA0 = A[b] + index;
cache_address_ivld(EA);
new_index = index + sign_ext(off10);
new_index = new_index < 0 ? new_index + length : new_index % length;
A[b+1] = {length[15:0],new_index[15:0]};
```

**CACHEA.I                      A[b], off10 (BO)                      (Post-increment Addressing Mode)**



```
EA = A[b];
cache_address_ivld(EA);
A[b] = EA + sign_ext(off10);
```

**CACHEA.I                      A[b], off10 (BO)                      (Pre-increment Addressing Mode)**



```
EA = A[b] + sign_ext(off10);
cache_address_ivld(EA);
A[b] = EA;
```

**Status Flags**

C	Not set by this instruction.
V	Not set by this instruction.
SV	Not set by this instruction.
AV	Not set by this instruction.
SAV	Not set by this instruction.

**Examples**

```
cachea.i    [a3] 4
cachea.i    [+a3] 4
cachea.i    [a3+] 4
cachea.i    [a4/a5+c] 4
cachea.i    [a4/a5+r]
```

See Also

[CACHEA.W](#), [CACHEA.WI](#)

See Also Addendum for TriCore Arch Manual, Vol2, V1.3.8, pages 493/494

## CACHEA.W

### Cache Address, Writeback

#### Description

If the cache line containing the byte 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.

If the cache line containing the byte 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. Any address register updates associated with the addressing mode are always performed regardless of the cache operation. The effective address is a virtual address when operating in virtual mode.

#### CACHEA.W      A[b], off10 (BO)      (Base + Short Offset Addressing Mode)

31	28 27	22 21	16 15	12 11	8 7	0
off10[9:6]	2C <sub>H</sub>	off10[5:0]	b	-	89 <sub>H</sub>	

EA = A[b] + sign\_ext(off10);

cache\_address\_wb(EA);

#### CACHEA.W      P[b] (BO)      (Bit-reverse Addressing Mode)

31	28 27	22 21	16 15	12 11	8 7	0
-	0C <sub>H</sub>	-	b	-	A9 <sub>H</sub>	

index = zero\_ext(A[b+1][15:0]);

incr = zero\_ext(A[b+1][31:16]);

EA = A[b] + index;

cache\_address\_wb(EA);

new\_index = reverse16(reverse16(index) + reverse16(incr));

A[b+1] = {incr[15:0], new\_index[15:0]};

#### CACHEA.W      P[b], off10 (BO)      (Circular Addressing Mode)

31	28 27	22 21	16 15	12 11	8 7	0
off10[9:6]	1C <sub>H</sub>	off10[5:0]	b	-	A9 <sub>H</sub>	

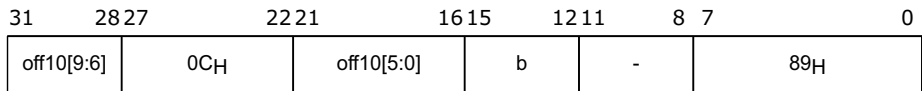
index = zero\_ext(A[b+1][15:0]);

length = zero\_ext(A[b+1][31:16]);



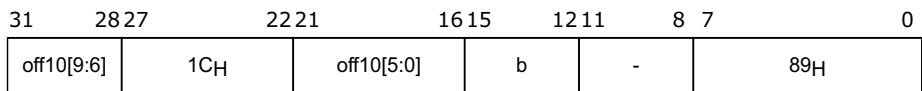
EA = A[b] + index;  
 cache\_address\_wb(EA);  
 new\_index = index + sign\_ext(off10);  
 new\_index = new\_index < 0 ? new\_index+length : new\_index % length;  
 A[b+1] = {length[15:0], new\_index[15:0]};

**CACHEA.W                      A[b], off10 (BO)                      (Post-increment Addressing Mode)**



EA = A[b];  
 cache\_address\_wb(EA);  
 A[b] = EA + sign\_ext(off10);

**CACHEA.W                      A[b], off10 (BO)                      (Pre-increment Addressing Mode)**



EA = A[b] + sign\_ext(off10);  
 cache\_address\_wb(EA);  
 A[b] = EA;

**Status Flags**

C	Not set by this instruction.
V	Not set by this instruction.
SV	Not set by this instruction.
AV	Not set by this instruction.
SAV	Not set by this instruction.

**Examples**

```
cachea.w    [a3] 4
cachea.w    [+a3] 4
cachea.w    [a3+] 4
cachea.w    [a4/a5+c] 4
cachea.w    [a4/a5+r]
```

See Also

[CACHEA.I](#), [CACHEA.WI](#)

See Also Addendum for TriCore Arch Manual, Vol2, V1.3.8, pages 493/494

## CACHEA.WI

### Cache Address, Writeback and Invalidate

#### Description

If the cache line containing the byte 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.

If the cache line containing the byte 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. Any address register updates associated with the addressing mode are always performed regardless of the cache operation. The effective address is a virtual address when operating in virtual mode.

#### CACHEA.WI      A[b], off10 (BO)      (Base + Short Offset Addressing Mode)

31	28 27	22 21	16 15	12 11	8 7	0
off10[9:6]	2D <sub>H</sub>	off10[5:0]	b	-		89 <sub>H</sub>

EA = A[b] + sign\_ext(off10);

cache\_address\_wi(EA);

#### CACHEA.WI      P[b] (BO)      (Bit-reverse Addressing Mode)

31	28 27	22 21	16 15	12 11	8 7	0
-	0D <sub>H</sub>	-	b	-		A9 <sub>H</sub>

index = zero\_ext(A[b+1][15:0]);

incr = zero\_ext(A[b+1][31:16]);

EA = A[b] + index;

cache\_address\_wi(EA);

new\_index = reverse16(reverse16(index) + reverse16(incr));

A[b+1] = {incr[15:0], new\_index[15:0]};

#### CACHEA.WI      P[b], off10 (BO)      (Circular Addressing Mode)

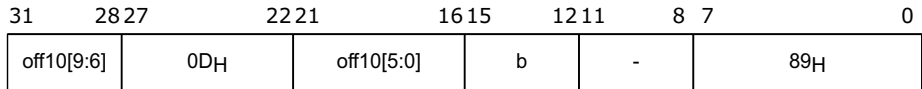
31	28 27	22 21	16 15	12 11	8 7	0
off10[9:6]	1D <sub>H</sub>	off10[5:0]	b	-		A9 <sub>H</sub>

index = zero\_ext(A[b+1][15:0]);

length = zero\_ext(A[b+1][31:16]);

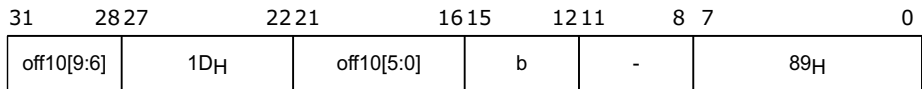
EA = A[b] + index;  
 cache\_address\_wi(EA);  
 new\_index = index + sign\_ext(off10);  
 new\_index = new\_index < 0 ? new\_index + length : new\_index % length;  
 A[b+1] = {length[15:0], new\_index[15:0]};

**CACHEA.WI      A[b], off10 (BO)      (Post-increment Addressing Mode)**



EA = A[b];  
 cache\_address\_wi(EA);  
 A[b] = EA + sign\_ext(off10);

**CACHEA.WI      A[b], off10 (BO)      (Pre-increment Addressing Mode)**



EA = A[b] + sign\_ext(off10);  
 cache\_address\_wi(EA);  
 A[b] = EA;

**Status Flags**

C	Not set by this instruction.
V	Not set by this instruction.
SV	Not set by this instruction.
AV	Not set by this instruction.
SAV	Not set by this instruction.

**Examples**

```

cachea.wi [a3]4
cachea.wi [+a3]4
cachea.wi [a3+]4
cachea.wi [a4/a5+c]4
cachea.wi [a4/a5+r]
```

See Also Addendum for TriCore Arch Manual, Vol2, V1.3.8, pages 493/494

See Also

[CACHEA.I](#), [CACHEA.W](#)

See Also Addendum for TriCore Arch Manual, Vol2, V1.3.8, pages 493/494

See Also Addendum for TriCore Arch Manual, Vol2, V1.3.8, pages 493-494

## CACHEI.W

### Cache Index, Writeback

1.3.1

#### Description

If any modified cache line at the memory index/way specified by address register A[b] is present in the L1 data cache, writeback the modified data. The line will still be present within the L1 data cache but will be marked as unmodified.

The address specified by the address register A[b] undergoes standard protection checks. Address register updates associated with the addressing mode are performed regardless of the cache operation.

The location of way/index within A[b] is implementation dependent.

#### CACHEI.W      A[b], off10 (BO)      (Base + Short Offset Addressing Mode)

31	28 27	22 21	16 15	12 11	8 7	0
off10[9:6]	2BH	off10[5:0]	b	-	89H	

index\_way = A[b] + sign\_ext(off10);

cache\_index\_wb(index\_way);

#### CACHEI.W      A[b], off10 (BO)      (Post-increment Addressing Mode)

31	28 27	22 21	16 15	12 11	8 7	0
off10[9:6]	0BH	off10[5:0]	b	-	89H	

index\_way = A[b];

cache\_index\_wb(index\_way);

A[b] = index\_way + sign\_ext(off10);

#### CACHEI.W      A[b], off10 (BO)      (Pre-increment Addressing Mode)

31	28 27	22 21	16 15	12 11	8 7	0
off10[9:6]	1BH	off10[5:0]	b	-	89H	

index\_way = A[b] + sign\_ext(off10);

cache\_index\_wb(index\_way);

A[b] = index\_way;

#### Status Flags

C	Not set by this instruction.
---	------------------------------

Instruction Set

V	Not set by this instruction.
SV	Not set by this instruction.
AV	Not set by this instruction.
SAV	Not set by this instruction.

**Examples**

```
cachei.w [a3]4  
cachei.w [+a3]4  
cachei.w [a3+]4
```

**See Also**

[CACHEA.I](#), [CACHEA.W](#), [CACHEA.WI](#), [CACHEI.I](#), [CACHEI.WI](#)

See Also Addendum for TriCore Arch Manual, Vol2, V1.3.8, pages 493-494

## CACHEI.WI

### Cache Index, Writeback, Invalidate

1.3.1

#### Description

If the cache line at the memory index/way specified by the address register A[b] is present in the L1 data cache, write back the modified data and then invalidate the line in the L1 data cache.

The address specified by the address register A[b] undergoes standard protection checks. Address register updates associated with the addressing mode are performed regardless of the cache operation.

The location of way/index within A[b] is implementation dependent.

#### CACHEI.WI      A[b], off10 (BO)      (Base + Short Offset Addressing Mode)

31	28 27	22 21	16 15	12 11	8 7	0
off10[9:6]	2FH	off10[5:0]	b	-		89H

index\_way = A[b] + sign\_ext(off10);

cache\_index\_wi(index\_way);

#### CACHEI.WI      A[b], off10 (BO)      (Post-increment Addressing Mode)

31	28 27	22 21	16 15	12 11	8 7	0
off10[9:6]	0FH	off10[5:0]	b	-		89H

index\_way = A[b];

cache\_index\_wi(index\_way);

A[b] = index\_way + sign\_ext(off10);

#### CACHEI.WI      A[b], off10 (BO)      (Pre-increment Addressing Mode)

31	28 27	22 21	16 15	12 11	8 7	0
off10[9:6]	1FH	off10[5:0]	b	-		89H

index\_way = A[b] + sign\_ext(off10);

cache\_index\_wi(index\_way);

A[b] = index\_way;

#### Status Flags

C	Not set by this instruction.
---	------------------------------



V	Not set by this instruction.
SV	Not set by this instruction.
AV	Not set by this instruction.
SAV	Not set by this instruction.

**Examples**

```
cachei.wi [a3]4  
cachei.wi [+a3]4  
cachei.wi [a3+]4
```

**See Also**

[CACHEA.I](#), [CACHEA.W](#), [CACHEA.WI](#), [CACHEI.I](#)

## CADD Conditional Add

### Description

If the contents of data register D[d] are non-zero, then add the contents of data register D[a] and the contents of either register D[b] (instruction format RRR) or const9 (instruction format RCR) and put the result in data register D[c]; otherwise put contents of D[a] in D[c]. The const9 value is sign-extended.

If the contents of data register D[15] are non-zero, then add contents of data register D[a] and the contents of const4 and put the result in data register D[a]; otherwise the contents of D[a] is unchanged. The const4 value is sign-extended.

### CADD                      D[c], D[d], D[a], const9 (RCR)

	31	28 27	24 23	21 20		12 11	8 7	0
	c	d	00H	const9		a	ABH	

condition = D[d] != 0;

result = ((condition) ? D[a] + sign\_ext(const9) : D[a]);

D[c] = result[31:0];

### CADD                      D[c], D[d], D[a], D[b] (RRR)

	31	28 27	24 23	20 19	18 17	16 15		12 11	8 7	0
	c	d	00H	-	-	b	a	2BH		

condition = (D[d] != 0);

result = ((condition) ? D[a] + D[b] : D[a]);

D[c] = result[31:0];

### CADD                      D[a], D[15], const4 (SRC)

	15	12 11	8 7	0
	const4	a	8AH	

condition = (D[15] != 0);  
result = ((condition) ? D[a] + sign\_ext(const4) : D[a]);  
D[a] = result[31:0];

### Status Flags

C	Not set by this instruction.
V	overflow = (result > 7FFFFFFF <sub>H</sub> ) OR (result < -80000000 <sub>H</sub> ); if (condition) then PSW.V = overflow else PSW.V = PSW.V;
SV	if (condition AND overflow) then PSW.SV = 1 else PSW.SV = PSW.SV;
AV	advanced_overflow = result[31] ^ result[30]; if (condition) then PSW.AV = advanced_overflow else PSW.AV = PSW.AV;
SAV	if (condition AND advanced_overflow) then PSW.SAV = 1 else PSW.SAV = PSW.SAV;

### Examples

```
cadd d3, d4, d1, d2
cadd d3, d4, d1, #126
```

```
cadd d1, d15, 6
```

### See Also

[CADDN](#), [CMOV \(16-bit\)](#), [CMOVN \(16-bit\)](#), [CSUB](#), [CSUBN](#), [SEL](#), [SELN](#)

## CADDN

### Conditional Add-Not

#### Description

If the contents of data register D[d] are zero, then add the contents of data register D[a] and the contents of either register D[b] (instruction format RRR) or const9 (instruction format RCR) and put the result in data register D[c]; otherwise put the contents of D[a] in D[c]. The const9 value is sign-extended.

If the contents of data register D[15] are zero, then add the contents of data register D[a] and the contents of const4 and put the result in data register D[a]; otherwise the contents of D[a] is unchanged. The const4 value is sign-extended.

#### CADDN                      D[c], D[d], D[a], const9 (RCR)

	31	28 27	24 23	21 20		12 11	8 7		0
	c	d	01 <sub>H</sub>	const9		a	AB <sub>H</sub>		

condition = (D[d] == 0);

result = ((condition) ? D[a] + sign\_ext(const9) : D[a]);

D[c] = result[31:0];

#### CADDN                      D[c], D[d], D[a], D[b] (RRR)

	31	28 27	24 23	20 19	18 17	16 15		12 11	8 7		0
	c	d	01 <sub>H</sub>	-	-	b	a	2B <sub>H</sub>			

condition = (D[d] == 0);

result = ((condition) ? D[a] + D[b] : D[a]);

D[c] = result[31:0];

#### CADDN                      D[a], D[15], const4 (SRC)

	15	12 11	8 7		0
	const4	a	CA <sub>H</sub>		

condition = (D[15] == 0);  
result = ((condition) ? D[a] + sign\_ext(const4) : D[a]);  
D[a] = result[31:0];

### Status Flags

C	Not set by this instruction.
V	overflow = (result > 7FFFFFFF <sub>H</sub> ) OR (result < -80000000 <sub>H</sub> ); if (condition) then PSW.V = overflow else PSW.V = PSW.V;
SV	if (condition AND overflow) then PSW.SV = 1 else PSW.SV = PSW.SV;
AV	advanced_overflow = result[31] ^ result[30]; if (condition) then PSW.AV = advanced_overflow else PSW.AV = PSW.AV;
SAV	if (condition AND advanced_overflow) then PSW.SAV = 1 else PSW.SAV = PSW.SAV;

### Examples

```
caddn d3, d4, d1, d2
caddn d3, d4, d1, #126
```

```
caddn d1, d15, #6
```

### See Also

**CADD**, **CMOV (16-bit)**, **CMOVN (16-bit)**, **CSUB**, **CSUBN**, **SEL**, **SELN**

## CALL

### Call

#### Description

Add the value specified by disp24, multiplied by two and sign-extended, 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). Set register A[11] (return address) to the address of the next instruction beyond the call.

*Note: After CALL, upper context registers are undefined except for A[10] and A[11].*

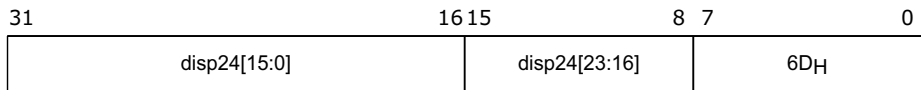
*Note: When the PSW is saved, the CDE bit is forced to '1'.*

Add the value specified by disp8, multiplied by two and sign-extended, 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). Set register A[11] (return address) to the address of the next instruction beyond the call.

*Note: After CALL, upper context registers are undefined except for A[10] and A[11].*

*Note: When the PSW is saved, the CDE bit is forced to '1'.*

#### CALL                      disp24 (B)



if (FCX == 0) trap(FCU);

if (PSW.CDE) then if (cdc\_increment()) then trap(CDO);

PSW.CDE = 1;

ret\_addr = PC + 4;

tmp\_FCX = FCX;

EA = {FCX.FCXS, 6'b0, FCX.FCXO, 6'b0};

new\_FCX = M(EA, word);

M(EA, 16 \* word) = {PCXI, PSW, A[10], A[11], D[8], D[9], D[10], D[11], A[12], A[13], A[14], A[15], D[12], D[13], D[14], D[15]};

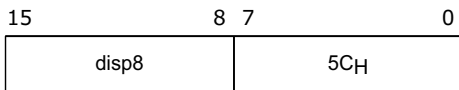
PCXI.PCPN = ICR.CCPN;

PCXI.PIE = ICR.IE;

PCXI.UL = 1;

```
PCXI[19:0] = FCX[19:0];
FCX[19:0] = new_FCX[19:0];
PC = PC + sign_ext(2 * disp24);
A[11] = ret_addr[31:0];
if (tmp_FCX == LCX) trap(FCD);
```

**CALL** **disp8 (SB)**



```
if (FCX == 0) trap(FCU);
if (PSW.CDE) then if(cdc_increment()) then trap(CDO);
PSW.CDE = 1;
ret_addr = PC + 2 ;
tmp_FCX = FCX;
EA = {FCX.FCXS, 6'b0, FCX.FCXO, 6'b0};
new_FCX = M(EA, word);
M(EA, 16 * word) = {PCXI, PSW, A[10], A[11], D[8], D[9], D[10], D[11], A[12], A[13],
A[14], A[15], D[12], D[13], D[14], D[15]};
PCXI.PCPN = ICR.CCPN;
PCXI.PIE = ICR.IE;
PCXI.UL = 1;
PCXI[19:0] = FCX[19:0];
FCX[19:0] = new_FCX[19:0];
PC = PC + sign_ext(2 * disp8);
A[11] = ret_addr[31:0];
if (tmp_FCX == LCX) trap(FCD);
```

**Status Flags**

C	Not changed by this instruction but read by the instruction.
V	Not changed by this instruction but read by the instruction.
SV	Not changed by this instruction but read by the instruction.
AV	Not changed by this instruction but read by the instruction.
SAV	Not changed by this instruction but read by the instruction.

**Examples**

```
call foobar
```

```
call foobar
```

See Also

[CALLA](#), [CALLI](#), [RET](#)

See Also Addendum for TriCore Arch Manual, Vol2, V1.3.8, pages 493/494



## CALLA Call Absolute

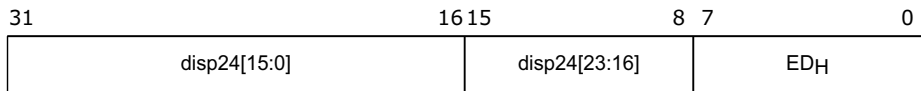
### Description

Jump to the address specified by disp24. In parallel with the jump, save the caller's Upper Context to an available Context Save Area (CSA). Set register A[11] (return address) to the address of the next instruction beyond the call.

*Note: After CALLA, upper context registers are undefined except for A[10] and A[11].*

*Note: When the PSW is saved, the CDE bit is forced to '1'.*

### CALLA                      disp24 (B)



if (FCX == 0) trap(FCU);

if (PSW.CDE) then if (cdc\_increment()) then trap(CDO);

PSW.CDE = 1;

ret\_addr = PC + 4;

tmp\_FCX = FCX;

EA = {FCX.FCXS, 6'b0, FCX.FCXO, 6'b0};

new\_FCX = M(EA, word);

M(EA, 16 \* word) = {PCXI, PSW, A[10], A[11], D[8], D[9], D[10], D[11], A[12], A[13], A[14], A[15], D[12], D[13], D[14], D[15]};

PCXI.PCPN = ICR.CCPN;

PCXI.PIE = ICR.IE;

PCXI.UL = 1;

PCXI[19:0] = FCX[19:0];

FCX[19:0] = new\_FCX[19:0];

PC = {disp24[23:20], 7'b0, disp24[19:0], 1'b0};

A[11] = ret\_addr[31:0];

if (tmp\_FCX == LCX) trap(FCD);

### Status Flags

C	Not changed by this instruction but read by the instruction.
V	Not changed by this instruction but read by the instruction.

SV	Not changed by this instruction but read by the instruction.
AV	Not changed by this instruction but read by the instruction.
SAV	Not changed by this instruction but read by the instruction.

**Examples**

```
calla foobar
```

**See Also**

[CALL](#), [CALLI](#), [JL](#), [JLA](#), [RET](#)

See Also Addendum for TriCore Arch Manual, Vol2, V1.3.8, pages 493/494

## CALLI

### Call Indirect

#### Description

Jump to the address specified by the contents of address register A[a]. In parallel with the jump save the caller's Upper Context to an available Context Save Area (CSA). Set register A[11] (return address) to the address of the next instruction beyond the call.

*Note: After CALLI, upper context registers are undefined except for A[10] and A[11].*

*Note: When the PSW is saved, the CDE bit is forced to '1'.*

#### CALLI                      A[a] (RR)

31	28 27	20 19 18 17 16 15	12 11	8 7	0
-	00H	-   -	-	a	2DH

if (FCX == 0) trap(FCU);

if (PSW.CDE) then if(cdc\_increment()) then trap(CDO);

PSW.CDE = 1;

ret\_addr = PC + 4;

tmp\_FCX = FCX;

EA = {FCX.FCXS, 6'b0, FCX.FCXO, 6'b0};

new\_FCX = M(EA, word);

M(EA, 16 \* word) = {PCXI, PSW, A[10], A[11], D[8], D[9], D[10], D[11], A[12], A[13], A[14], A[15], D[12], D[13], D[14], D[15]};

PCXI.PCPN = ICR.CCPN;

PCXI.PIE = ICR.IE;

PCXI.UL = 1;

PCXI[19:0] = FCX[19:0];

FCX[19:0] = new\_FCX[19:0];

PC = {A[a][31:1], 1'b0};

A[11] = ret\_addr[31:0];

if (tmp\_FCX == LCX) trap(FCD);

#### Status Flags

C	Not changed by this instruction but read by the instruction.
V	Not changed by this instruction but read by the instruction.

SV	Not changed by this instruction but read by the instruction.
AV	Not changed by this instruction but read by the instruction.
SAV	Not changed by this instruction but read by the instruction.

**Examples**

```
calli    a2
```

**See Also**

[CALL](#), [CALLA](#), [RET](#)

See Also Addendum for TriCore Arch Manual, Vol2, V1.3.8, pages 493/494

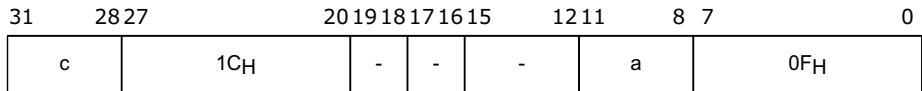
## CLO

### Count Leading Ones

#### Description

Count the number of consecutive ones in D[a], starting with bit 31, and put the result in D[c].

#### CLO                      D[c], D[a] (RR)



result = leading\_ones(D[a]);

D[c] = zero\_ext(result);

#### Status Flags

C	Not set by this instruction.
V	Not set by this instruction.
SV	Not set by this instruction.
AV	Not set by this instruction.
SAV	Not set by this instruction.

#### Examples

clo    d3, d1

#### See Also

[CLS](#), [CLZ](#), [CLO.H](#), [CLS.H](#), [CLZ.H](#)

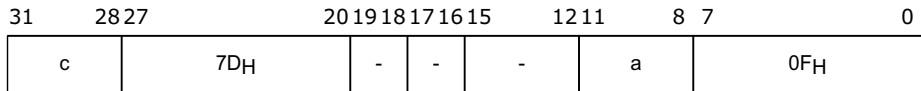
## CLO.H

### Count Leading Ones in Packed Half-words

#### Description

Count the number of consecutive ones in each half-word of D[a], starting with the most significant bit, and put each result in the corresponding half-word of D[c].

#### CLO.H                      D[c], D[a] (RR)



result\_halfword1 = zero\_ext(leading\_ones(D[a][31:16]));

result\_halfword0 = zero\_ext(leading\_ones(D[a][15:0]));

D[c] = {result\_halfword1[15:0],result\_halfword0[15:0]};

#### Status Flags

C	Not set by this instruction.
V	Not set by this instruction.
SV	Not set by this instruction.
AV	Not set by this instruction.
SAV	Not set by this instruction.

#### Examples

```
clo.h    d3, d1
```

#### See Also

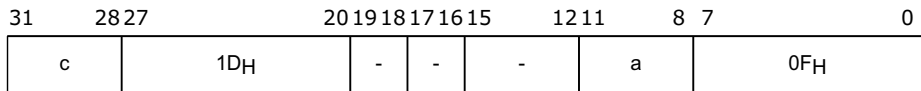
[CLO](#), [CLS](#), [CLS.H](#), [CLZ](#), [CLZ.H](#)

## CLS Count Leading Signs

### Description

Count the number of consecutive bits which have the same value as bit 31 in D[a], starting with bit 30, and put the result in D[c]. The result is the number of leading sign bits minus one, giving the number of redundant sign bits in D[a].

### CLS                      D[c], D[a] (RR)



result = leading\_signs(D[a]) - 1;

D[c] = zero\_ext(result);

### Status Flags

C	Not set by this instruction.
V	Not set by this instruction.
SV	Not set by this instruction.
AV	Not set by this instruction.
SAV	Not set by this instruction.

### Examples

```
cls    d3, d1
```

### See Also

[CLO](#), [CLO.H](#), [CLZ](#), [CLZ.H](#), [CLS.H](#)

See Also Addendum for TriCore Arch Manual, Vol2, V1.3.8, pages 493/494

## CLS.H Count Leading Signs in Packed Half-words

### Description

Count the number of consecutive bits in each half-word in data register D[a] which have the same value as the most-significant bit in that half-word, starting with the next bit right of the most-significant bit. Put each result in the corresponding half-word of D[c].

The results are the number of leading sign bits minus one in each half-word, giving the number of redundant sign bits in the half-words of D[a].

### CLS.H                      D[c], D[a] (RR)

31	28 27	20 19 18 17 16 15	12 11	8 7	0
c	7EH	-	-	a	0FH

result\_halfword1 = zero\_ext(leading\_signs(D[a][31:16]) - 1);

result\_halfword0 = zero\_ext(leading\_signs(D[a][15:0]) - 1);

D[c] = {result\_halfword1[15:0], result\_halfword0[15:0]};

### Status Flags

C	Not set by this instruction.
V	Not set by this instruction.
SV	Not set by this instruction.
AV	Not set by this instruction.
SAV	Not set by this instruction.

### Examples

cls.h    d3, d1

### See Also

[CLO](#), [CLO.H](#), [CLS](#), [CLZ](#), [CLZ.H](#)



## CLZ Count Leading Zeros

### Description

Count the number of consecutive zeros in D[a] starting with bit 31, and put result in D[c].

### CLZ                      D[c], D[a] (RR)

31	28 27	20 19 18 17 16 15	12 11	8 7	0
c	1BH	-	-	-	a
					0FH

result = leading\_zeros(D[a]);

D[c] = zero\_ext(result);

### Status Flags

C	Not set by this instruction.
V	Not set by this instruction.
SV	Not set by this instruction.
AV	Not set by this instruction.
SAV	Not set by this instruction.

### Examples

```
clz    d3, d1
```

### See Also

[CLO](#), [CLO.H](#), [CLS](#), [CLS.H](#), [CLZ.H](#)

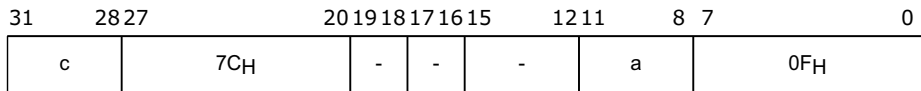
## CLZ.H

### Count Leading Zeros in Packed Half-words

#### Description

Count the number of consecutive zeros in each half-word of D[a], starting with the most significant bit of each half-word, and put each result in the corresponding half-word of D[c].

#### CLZ.H                      D[c], D[a] (RR)



result\_halfword1 = zero\_ext(leading\_zeros(D[a][31:16]));

result\_halfword0 = zero\_ext(leading\_zeros(D[a][15:0]));

D[c] = {result\_halfword1[15:0], result\_halfword0[15:0]};

#### Status Flags

C	Not set by this instruction.
V	Not set by this instruction.
SV	Not set by this instruction.
AV	Not set by this instruction.
SAV	Not set by this instruction.

#### Examples

clz.h    d3, d1

#### See Also

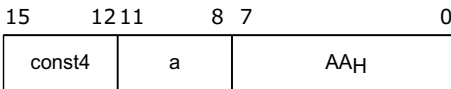
[CLO](#), [CLO.H](#), [CLS](#), [CLS.H](#), [CLZ](#)

## CMOV (16-bit) Conditional Move (16-bit)

### Description

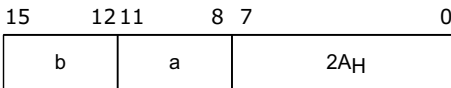
If the contents of data register D[15] are not zero, copy the contents of either data register D[b] (instruction format SRR) or const4 (instruction format SRC) to data register D[a]; otherwise the contents of D[a] is unchanged. The const4 value is sign-extended.

### CMOV D[a], D[15], const4 (SRC)



$D[a] = ((D[15] \neq 0) ? \text{sign\_ext}(\text{const4}) : D[a]);$

### CMOV D[a], D[15], D[b] (SRR)



$D[a] = ((D[15] \neq 0) ? D[b] : D[a]);$

### Status Flags

C	Not set by this instruction.
V	Not set by this instruction.
SV	Not set by this instruction.
AV	Not set by this instruction.
SAV	Not set by this instruction.

### Examples

```
cmov d1, d15, d2
cmov d1, d15, #6
```

### See Also

[CADD](#), [CADDN](#), [CMOVN \(16-bit\)](#), [CSUB](#), [CSUBN](#), [SEL](#), [SELN](#)

## CMOVN (16-bit) Conditional Move-Not (16-bit)

### Description

If the contents of data register D[15] are zero, copy the contents of either data register D[b] (instruction format SRR) or const4 (instruction format SRC) to data register D[a]; otherwise the contents of D[a] is unchanged. The const4 value is sign-extended to 32-bits.

### CMOVN                      D[a], D[15], const4 (SRC)

15      12 11              8 7                      0

const4	a	EAH
--------	---	-----

$D[a] = ((D[15] == 0) ? \text{sign\_ext}(\text{const4}) : D[a]);$

### CMOVN                      D[a], D[15], D[b] (SRR)

15      12 11              8 7                      0

b	a	6AH
---	---	-----

$D[a] = ((D[15] == 0) ? D[b] : D[a]);$

### Status Flags

C	Not set by this instruction.
V	Not set by this instruction.
SV	Not set by this instruction.
AV	Not set by this instruction.
SAV	Not set by this instruction.

### Examples

```
cmovn    d1, d15, d2
cmovn    d1, d15, #6
```

### See Also

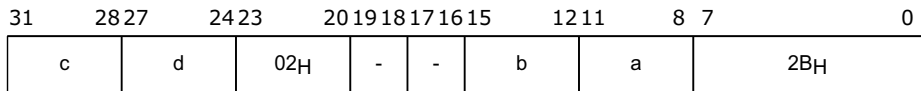
[CADD](#), [CADDN](#), [CMOV \(16-bit\)](#), [CSUB](#), [CSUBN](#), [SEL](#), [SELN](#)

## CSUB Conditional Subtract

### Description

If the contents of data register D[d] are not zero, subtract the contents of data register D[b] from the contents of data register D[a] and put the result in data register D[c]; otherwise put the contents of D[a] in D[c].

### CSUB                    D[c], D[d], D[a], D[b] (RRR)



condition = (D[d] != 0);

result = ((condition) ? D[a] - D[b] : D[a]);

D[c] = result[31:0];

### Status Flags

C	Not set by this instruction.
V	overflow = (result > 7FFFFFFF <sub>H</sub> ) OR (result < -80000000 <sub>H</sub> ); if(condition) then PSW.V = overflow else PSW.V = PSW.V;
SV	if (condition AND overflow) then PSW.SV = 1 else PSW.SV = PSW.SV;
AV	advanced_overflow = result[31] ^ result[30]; if (condition) then PSW.AV = advanced_overflow else PSW.AV = PSW.AV;
SAV	if (condition AND advanced_overflow) then PSW.SAV = 1 else PSW.SAV = PSW.SAV;

### Examples

```
csub d3, d4, d1, d2
```

### See Also

**CADD, CADDN, CMOV (16-bit), CMOVN (16-bit), CSUBN, SEL, SELN**

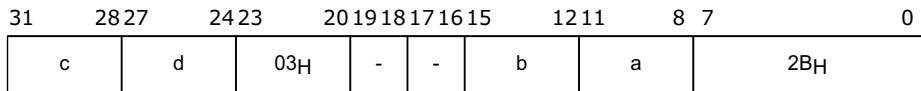
## CSUBN

### Conditional Subtract-Not

#### Description

If the contents of data register D[d] are zero, subtract the contents of data register D[b] from the contents of data register D[a] and put the result in data register D[c]; otherwise put the contents of D[a] in D[c].

#### CSUBN                      D[c], D[d], D[a], D[b] (RRR)



condition = (D[d] == 0);

result = ((condition) ? D[a] - D[b] : D[a]);

D[c] = result[31:0];

#### Status Flags

C	Not set by this instruction.
V	overflow = (result > 7FFFFFFF <sub>H</sub> ) OR (result < -80000000 <sub>H</sub> ); if (condition) then PSW.V = overflow else PSW.V = PSW.V;
SV	if (condition AND overflow) then PSW.SV = 1 else PSW.SV = PSW.SV;
AV	advanced_overflow = result[31] ^ result[30]; if (condition) then PSW.AV = advanced_overflow else PSW.AV = PSW.AV;
SAV	if (condition AND advanced_overflow) then PSW.SAV = 1 else PSW.SAV = PSW.SAV;

#### Examples

```
csubn d3, d4, d1, d2
```

#### See Also

**CADD, CADDN, CMOV (16-bit), CMOVN (16-bit), CSUB, SEL, SELN**

## DEBUG

### Debug

#### Description

If the Debug mode is enabled (DBGSR.DE == 1), cause a Debug Event; otherwise execute a NOP.

If the Debug mode is enabled (DBGSR.DE == 1), cause a Debug event; otherwise execute a NOP.

#### DEBUG (SR)

15 12 11 8 7 0

0AH	-	00H	
-----	---	-----	--

-

#### DEBUG (SYS)

31 28 27 22 21 12 11 8 7 0

-	04H	-	-	0DH
---	-----	---	---	-----

-

#### Status Flags

C	Not set by this instruction.
V	Not set by this instruction.
SV	Not set by this instruction.
AV	Not set by this instruction.
SAV	Not set by this instruction.

#### Examples

debug

debug

#### See Also

[RFM](#)

## DEXTR

### Extract from Double Register

#### Description

Extract 32-bits from registers {D[a], D[b]}, where D[a] contains the most-significant 32-bits of the value, starting at the bit number specified by either 32 - D[d][4:0] (instruction format RRRR) or 32 - pos (instruction format RRPW). Put the result in D[c].

*Note: D[a] and D[b] can be any two data registers or the same register. For this instruction they are treated as a 64-bit entity where D[a] contributes the high order bits and D[b] the low order bits.*

#### DEXTR D[c], D[a], D[b], pos (RRPW)

31	28 27	23 22 21 20	16 15	12 11	8 7	0
c	pos	00H	-	b	a	77H

$D[c] = (\{D[a], D[b]\} \ll \text{pos})[63:32];$

#### DEXTR D[c], D[a], D[b], D[d] (RRRR)

31	28 27	24 23	21 20	16 15	12 11	8 7	0
c	d	04H	-	b	a	17H	

$D[c] = (\{D[a], D[b]\} \ll D[d][4:0])[63:32];$

If  $D[d] > 31$  the result is undefined.

#### Status Flags

C	Not set by this instruction.
V	Not set by this instruction.
SV	Not set by this instruction.
AV	Not set by this instruction.
SAV	Not set by this instruction.

#### Examples

```
dextr d1, d3, d5, d7
dextr d1, d3, d5, #11
```

#### See Also

[EXTR](#), [EXTR.U](#), [INSERT](#), [INS.T](#), [INSN.T](#)



## DISABLE

### Disable Interrupts

#### Description

*Note: DISABLE can only be executed in User-1 mode or Supervisor mode.*

Disable interrupts by clearing Interrupt Enable bit (ICR.IE) in the Interrupt Control Register.

#### DISABLE (SYS)

31	28 27	22 21	12 11	8 7	0
-	0DH	-	-	-	0DH

ICR.IE = 0; // disables all interrupts

#### Status Flags

C	Not set by this instruction.
V	Not set by this instruction.
SV	Not set by this instruction.
AV	Not set by this instruction.
SAV	Not set by this instruction.

#### Examples

```
disable
```

#### See Also

[ENABLE](#), [BISR](#), [RSTV](#)

## DSYNC

### Synchronize Data

#### Description

Forces all data accesses to complete before any data accesses associated with an instruction, semantically after the DSYNC is initiated.

*Note: The Data Cache (DCACHE) is not invalidated by DSYNC.*

*Note: To ensure memory coherency, a DSYNC instruction must be executed prior to any access to an active CSA memory location.*

#### DSYNC (SYS)

31	28 27	22 21	12 11	8 7	0
-	12H	-	-	0DH	

#### Status Flags

C	Not set by this instruction.
V	Not set by this instruction.
SV	Not set by this instruction.
AV	Not set by this instruction.
SAV	Not set by this instruction.

#### Examples

dsync

#### See Also

[ISYNC](#)

## DVADJ Divide-Adjust

### Description

Divide-adjust the contents of the formatted data register E[d] using the divisor in D[b] and store the result in E[c]. E[d][63:32] contains the sign-extended final remainder from a previous DVSTEP instruction and E[d][31:0] contains the sign-extended final quotient in ones complement format. The DVADJ instruction converts the final quotient to twos complement format by adding one if the final quotient is negative, and corrects for a corner case that occurs when dividing a negative dividend that is an integer multiple of the divisor. The corner case is resolved by setting the remainder E[d][63:32] to zero and increasing the magnitude of the quotient E[d][31:0] by one. Note that the increment for converting a negative ones complement quotient to twos complement, and the decrement of a negative quotient in the corner case (described above), cancel out.

*Note: This operation must not be performed at the end of an unsigned divide sequence.*

### DVADJ                      E[c], E[d], D[b] (RRR)

31	28 27	24 23	20 19	18 17	16 15	12 11	8 7	0
c	d	0D <sub>H</sub>	-	0 <sub>H</sub>	b	-	6B <sub>H</sub>	

```
if ((abs(E[d][63:32]) == abs(D[b])) AND E[d][63]) then {
    result = E[d][31] ? {32'b0, E[d][31:0]} : {32'b0, E[d][31:0]+1};
} else {
    result = E[d][31] ? {E[d][63:32], E[d][31:0] + 1} : E[d];
}
E[c] = result[63:0];
```

### Status Flags

C	Not set by this instruction.
V	Not set by this instruction.
SV	Not set by this instruction.
AV	Not set by this instruction.
SAV	Not set by this instruction.

### Examples

-

**See Also**

-

See Also Addendum for TriCore Arch Manual, Vol2, V1.3.8, pages 493/494

## DVINIT

Divide-Initialization Word

### DVINIT.U

Divide-Initialization Word Unsigned

### DVINIT.B

Divide-Initialization Byte

### DVINIT.BU

Divide-Initialization Byte Unsigned

### DVINIT.H

Divide-Initialization Half-word

### DVINIT.HU

Divide-Initialization Half-word Unsigned

## Description

The DVINIT group of instructions prepare the operands for a subsequent DVSTEP instruction (see DVSTEP) from the dividend  $D[a]$  and divisor  $D[b]$ , and also check for conditions that will cause overflow of the final quotient result. After a DVINIT instruction  $E[c]$  contains the partially calculated remainder (equal to the sign-extended dividend) and partially calculated quotient (equal to + or - zero in ones complement format, depending on the signs of the dividend and divisor).

For signed operands DVINIT, DVINIT.H or DVINIT.B must be used. For unsigned operands DVINIT.U, DVINIT.HU and DVINIT.BU are used.

The size of the remainder and quotient bit fields in  $E[c]$  depend upon the variant of DVINIT used, which in turn depends upon the number of subsequent DVSTEP instructions required to calculate the final remainder and quotient results.

If the final quotient result is guaranteed to fit into 8 bits then a DVINIT.B(U) can be used, but must be followed by only one DVSTEP instruction.

If the quotient result is guaranteed to fit into 16 bits then a DVINIT.H(U) can be used but must be followed by two DVSTEP instructions.

For a quotient result of 32 bits a DVINIT(.U) must be used, followed by four DVSTEP instructions.

The resultant bit fields in  $E[c]$  are as follows:

- DVINIT(.U)  $E[c][63:0]$  = partially calculated remainder.
- DVINIT.H(U)  $E[c][63:16]$  = partially calculated remainder,  $E[c][15:0]$  = partially calculated quotient.
- DVINIT.B(.U)  $E[c][63:24]$  = partially calculated remainder,  $E[c][23:0]$  = partially calculated quotient.

The .B(U) and .H(U) suffixes of the DVINIT group of instructions indicate an 8-bit and 16-bit quotient result, not 8-bit and 16-bit operands as in other instructions. The operands

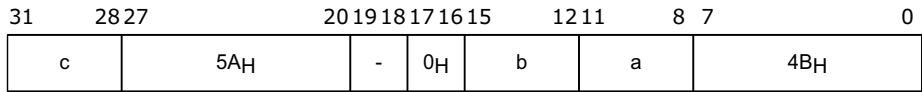
Instruction Set

supplied to a DVINIT, DVINIT.H or DVINIT.B instruction are required to be 32-bit sign-extended values. The operands supplied to the DVINIT.U, DVINIT.HU and DVINIT.BU instructions are 32-bit zero-extended values.

1.3.0 Overflow occurs if the expected quotient can not be represented in 32, 16 or 8-bits, depending on the DVINIT variant used. The magnitude of the remainder is not considered in the overflow calculation.

1.3.1 Overflow occurs if the divisor is zero, or if the dividend is the maximum negative value for the instruction variant and the divisor is minus one. No check is performed to ensure that the expected quotient can be represented in 32, 16, 8 bits, depending on the DVINIT variant used.

**DVINIT.B E[c], D[a], D[b] (RR)**



quotient\_sign = !(D[a][31] == D[b][31]);

abs\_sig\_dividend = abs(D[a]) >> 7;

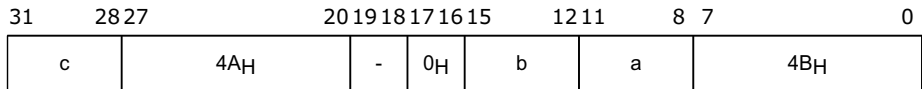
abs\_base\_dividend = abs(D[a]) & 32'h7F;

abs\_divisor = abs(D[b]);

E[c][63:24] = sign\_ext(D[a]);

E[c][23:0] = quotient\_sign ? 24'b111111111111111111111111 : 24'b0;

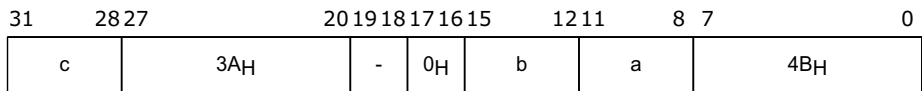
**DVINIT.BU E[c], D[a], D[b] (RR)**



E[c][63:24] = zero\_ext(D[a]);

E[c][23:0] = 0;

**DVINIT.H E[c], D[a], D[b] (RR)**



quotient\_sign = !(D[a][31] == D[b][31]);

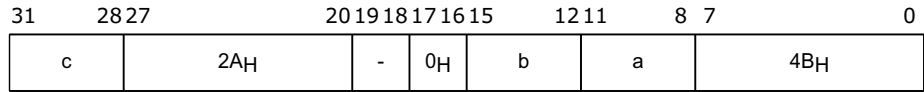
abs\_sig\_dividend = abs(D[a]) >> 15;

abs\_base\_dividend = abs(D[a]) & 32'h7FFF;

See Also Addendum for TriCore Arch Manual, Vol2, V1.3.8, pages 493/494

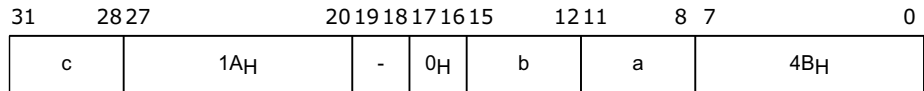
abs\_divisor = abs (D[b]);  
 E[c][63:16] = sign\_ext(D[a]);  
 E[c][15:0] = quotient\_sign ? 16'b1111111111111111 : 16'b0;

**DVINIT.HU E[c], D[a], D[b] (RR)**



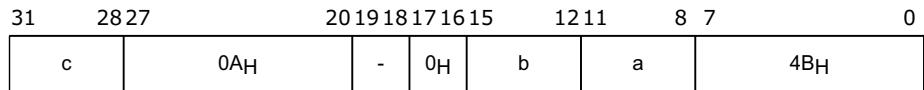
E[c][63:16] = zero\_ext(D[a]);  
 E[c][15:0] = 0;

**DVINIT E[c], D[a], D[b] (RR)**



E[c] = sign\_ext(D[a]);

**DVINIT.U E[c], D[a], D[b] (RR)**



E[c] = {00000000<sub>H</sub>, D[a]};

**Status Flags**

C	Not set by these instructions.
---	--------------------------------

See Also Addendum for TriCore Arch Manual, Vol2, V1.3.8, pages 493/494

V	<p>DVINIT</p> <p>if ((D[b] == 0) OR ((D[b] == 32'hFFFFFFFF) AND (D[a] == 32'h80000000)))  then overflow = 1 else overflow = 0;</p> <p>DVINIT.U</p> <p>if (D[b] == 0) then overflow = 1 else overflow = 0;</p> <p>DVINIT.B</p> <p>overflow = 0;</p> <p>if ((quotient_sign) AND (abs_dvisor)) then {</p> <p>overflow = (((abs_sig_dividend == abs_dvisor) AND (abs_base_dividend  &gt;= abs_dvisor)) OR (abs_sig_dividend &gt; abs_dvisor));</p> <p>} else {</p> <p>overflow = (abs_sig_dividend &gt;= abs_dvisor);</p> <p>}</p> <p>if ((D[b] == 0) OR ((D[b] == 32'hFFFFFFFF) AND (D[a] == 32'hFFFFFF80)))  then overflow = 1 else overflow = 0;</p> <p>DVINIT.BU</p> <p>if (abs(E[c][63:32]) &gt;= abs(D[b])) then overflow = 1 else overflow = 0;</p> <p>if (D[b]==0) then overflow = 1 else overflow = 0;</p> <p>DVINIT.H</p> <p>overflow = 0;</p> <p>if ((quotient_sign) AND (abs_dvisor)) then {</p> <p>overflow = (((abs_sig_dividend == abs_dvisor) AND (abs_base_dividend  &gt;= abs_dvisor)) OR (abs_sig_dividend &gt; abs_dvisor));</p> <p>} else {</p> <p>overflow = (abs_sig_dividend &gt;= abs_dvisor);</p> <p>}</p> <p>if ((D[b] == 0) OR ((D[b] == 32'hFFFFFFFF) AND (D[a] == 32'hFFFF8000)))  then overflow = 1 else overflow=0;</p> <p>DVINIT.HU</p> <p>if (abs(E[c][63:32]) &gt;= abs(D[b])) then overflow = 1 else overflow = 0;</p> <p>if (D[b] == 0) then overflow = 1 else overflow = 0;</p> <p>For all the DVINIT variations:  if (overflow) then PSW.V = 1 else PSW.V = 0;</p>	<p>1.3.0</p> <p>1.3.0</p> <p>1.3.0</p> <p>1.3.0</p> <p>1.3.0</p> <p>1.3.0</p> <p>1.3.0</p> <p>1.3.1</p> <p>1.3.0</p> <p>1.3.1</p> <p>1.3.0</p> <p>1.3.0</p> <p>1.3.0</p> <p>1.3.0</p> <p>1.3.0</p> <p>1.3.0</p> <p>1.3.0</p> <p>1.3.1</p> <p>1.3.0</p> <p>1.3.1</p> <p>1.3.0</p> <p>1.3.1</p> <p>1.3.0</p> <p>1.3.1</p>
SV	if (overflow) then PSW.SV = 1 else PSW.SV = PSW.SV;	
AV	PSW.AV = 0;	
SAV	Not set by these instructions.	

**Examples**

-



See Also

[DVADJ](#), [DVSTEP](#), [DVSTEP.U](#)

See Also Addendum for TriCore Arch Manual, Vol2, V1.3.8, pages 493/494

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

**Description**

The DVSTEP(.U) instruction divides the contents of the formatted data register E[d] by the divisor in D[b], producing 8-bits of quotient at a time. E[d] contains a partially calculated remainder and partially calculated quotient (in ones complement format) in bit fields that depend on the number of DVSTEP instructions required to produce a final result (see DVSTEP).

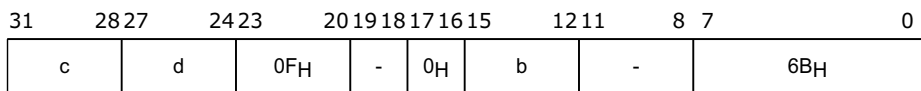
DVSTEP uses a modified restoring division algorithm to calculate 8-bits of the final remainder and quotient results. The size of the bit fields of the output register E[c] depend on the size of the bit fields in the input register E[d].

Resultant bit field sizes of E[c]:

- If E[d][63:0] = partially calculated remainder then E[c][63:8] = partially calculated remainder and E[c][7:0] = partially calculated quotient.
- If E[d][63:8] = partially calculated remainder then E[c][63:16] = partially calculated remainder and E[c][15:0] = partially calculated quotient.
- If E[d][63:16] = partially calculated remainder then E[c][63:24] = partially calculated remainder and E[c][23:0] = partially calculated quotient.
- If E[d][63:24] = partially calculated remainder then E[c][63:32] = final remainder and E[c][31:0] = final quotient.

The DVSTEP and DVSTEP.U operate on signed and unsigned operands respectively. A DVSTEP instruction that yields the final remainder and final quotient should be followed by a DVADJ instruction (see DVADJ).

**DVSTEP**                      **E[c], E[d], D[b] (RRR)**



```
dividend_sign = E[d][63];
divisor_sign = D[b][31];
quotient_sign = dividend_sign != divisor_sign;
addend = quotient_sign ? D[b] : 0 - D[b];
dividend_quotient = E[d][31:0];
remainder = E[d][63:32];
for i = 0 to 7 {
```

See Also Addendum for TriCore Arch Manual, Vol2, V1.3.8, pages 493/494

```

remainder = (remainder << 1) | dividend_quotient[31];
dividend_quotient <<= 1;
temp = remainder + addend;
remainder = ((temp < 0) == dividend_sign) ? temp :: remainder;
dividend_quotient = dividend_quotient | (((temp < 0) == dividend_sign) ?
!quotient_sign : quotient_sign);
}
E[c] = {remainder[31:0], dividend_quotient[31:0]};

```

**DVSTEP.U                    E[c], E[d], D[b] (RRR)**

31	28 27	24 23	20 19 18 17 16 15	12 11	8 7	0	
c	d	0EH	-	0H	b	-	6BH

```

divisor = D[b];
dividend_quotient = E[d][31:0];
remainder = E[d][63:32];
for i = 0 to 7 {
    remainder = (remainder << 1) | dividend_quotient[31];
    dividend_quotient <<= 1;
    temp = remainder - divisor;
    remainder = (temp < 0) ? remainder : temp;
    dividend_quotient = dividend_quotient | !(temp < 0);
}
E[c] = {remainder[31:0], dividend_quotient[31:0]};

```

**Status Flags**

C	Not set by these instructions.
V	Not set by these instructions.
SV	Not set by these instructions.
AV	Not set by these instructions.
SAV	Not set by these instructions.

**Examples**

-

**See Also**

[DVADJ](#), [DVINIT](#), [DVINIT.B](#), [DVINIT.BU](#), [DVINIT.H](#), [DVINIT.HU](#), [DVINIT.U](#)

See Also Addendum for TriCore Arch Manual, Vol2, V1.3.8, pages 493/494

## ENABLE

### Enable Interrupts

#### Description

*Note: ENABLE can only be executed in User-1 or Supervisor mode.*

Enable interrupts by setting the Interrupt Enable bit (ICR.IE) in the Interrupt Control Register (ICR) to one.

#### ENABLE (SYS)

31	28 27	22 21	12 11	8 7	0
-	0C <sub>H</sub>	-	-	-	0D <sub>H</sub>

ICR.IE = 1;

#### Status Flags

C	Not set by this instruction.
V	Not set by this instruction.
SV	Not set by this instruction.
AV	Not set by this instruction.
SAV	Not set by this instruction.

#### Examples

enable

#### See Also

**BISR, DISABLE, RSTV**

## EQ Equal

### Description

If the contents of data register D[a] are equal to the contents of either data register D[b] (instruction format RR) or const9 (instruction format RC), set the least-significant bit of D[c] to one and clear the remaining bits to zero; otherwise clear all bits in D[c]. The const9 value is sign-extended.

If the contents of data register D[a] are equal to the contents of either data register D[b] (instruction format SRR) or const4 (instruction format SRC), set the least-significant bit of D[15] to 1 and clear the remaining bits to zero; otherwise clear all bits in D[15]. The const4 value is sign-extended.

### EQ                      D[c], D[a], const9 (RC)

	31	28 27	21 20	12 11	8 7	0
	c	10 <sub>H</sub>	const9	a	8B <sub>H</sub>	

result = (D[a] == sign\_ext(const9));

D[c] = zero\_ext(result);

### EQ                      D[c], D[a], D[b] (RR)

	31	28 27	20 19 18 17 16 15	12 11	8 7	0	
	c	10 <sub>H</sub>	-	-	b	a	0B <sub>H</sub>

result = (D[a] == D[b]);

D[c] = zero\_ext(result);

### EQ                      D[15], D[a], const4 (SRC)

	15	12 11	8 7	0
	const4	a	BA <sub>H</sub>	

result = (D[a] == sign\_ext(const4));  
D[15] = zero\_ext(result);

### EQ                      D[15], D[a], D[b] (SRR)

	15	12 11	8 7	0
	b	a	3A <sub>H</sub>	

```
result = (D[a] == D[b]);
D[15] = zero_ext(result);
```

### Status Flags

C	Not set by this instruction.
V	Not set by this instruction.
SV	Not set by this instruction.
AV	Not set by this instruction.
SAV	Not set by this instruction.

### Examples

```
eq d3, d1, d2
eq d3, d1, #126
```

```
eq d15, d1, d2
eq d15, d1, #6
```

### See Also

[GE](#), [GE.U](#), [LT](#), [LT.U](#), [NE](#), [EQANY.B](#), [EQANY.H](#)

## EQ.A Equal to Address

### Description

If the contents of address registers A[a] and A[b] are equal, set the least-significant bit of D[c] to one and clear the remaining bits to zero; otherwise clear all bits in D[c].

### EQ.A                      D[c], A[a], A[b] (RR)

31	28 27	20 19 18 17 16 15	12 11	8 7	0
c	40H	-	-	b	a
01H					

$D[c] = (A[a] == A[b]);$

### Status Flags

C	Not set by this instruction.
V	Not set by this instruction.
SV	Not set by this instruction.
AV	Not set by this instruction.
SAV	Not set by this instruction.

### Examples

eq.a    d3, a4, a2

### See Also

[EQZ.A](#), [GE.A](#), [LT.A](#), [NE](#), [NEZ.A](#)



**EQ.B**  
Equal Packed Byte  
**EQ.H**  
Equal Packed Half-word  
**EQ.W**  
Equal Packed Word

**Description**

Compare each byte (EQ.B), half-word (EQ.H) or word (EQ.W) of D[a] with the corresponding byte, half-word or word of D[b].

In each case, if the two are equal, set the corresponding byte, half-word or word of D[c] to all ones; otherwise set the corresponding byte, half-word or word of D[c] to all zeros.

**EQ.B**                      **D[c], D[a], D[b] (RR)**

31	28 27	20 19 18 17 16 15	12 11	8 7	0	
c	50 <sub>H</sub>	-	-	b	a	0B <sub>H</sub>

$$D[c][31:24] = (D[a][31:24] == D[b][31:24]) ? 8'hFF : 8'h00;$$

$$D[c][23:16] = (D[a][23:16] == D[b][23:16]) ? 8'hFF : 8'h00;$$

$$D[c][15:8] = (D[a][15:8] == D[b][15:8]) ? 8'hFF : 8'h00;$$

$$D[c][7:0] = (D[a][7:0] == D[b][7:0]) ? 8'hFF : 8'h00;$$

**EQ.H**                      **D[c], D[a], D[b] (RR)**

31	28 27	20 19 18 17 16 15	12 11	8 7	0	
c	70 <sub>H</sub>	-	-	b	a	0B <sub>H</sub>

$$D[c][31:16] = (D[a][31:16] == D[b][31:16]) ? 16'hFFFF : 16'h0000;$$

$$D[c][15:0] = (D[a][15:0] == D[b][15:0]) ? 16'hFFFF : 16'h0000;$$

**EQ.W**                      **D[c], D[a], D[b] (RR)**

31	28 27	20 19 18 17 16 15	12 11	8 7	0	
c	90 <sub>H</sub>	-	-	b	a	0B <sub>H</sub>

$$D[c] = (D[a] == D[b]) ? 32'hFFFFFFFF : 32'h00000000;$$

**Status Flags**

C	Not set by these instructions.
---	--------------------------------

See Also Addendum for TriCore Arch Manual, Vol2, V1.3.8, pages 493/494

Instruction Set

V	Not set by these instructions.
SV	Not set by these instructions.
AV	Not set by these instructions.
SAV	Not set by these instructions.

**Examples**

eq.b d3, d1, d2  
eq.h d3, d1, d2  
eq.w d3, d1, d2

**See Also**

[LT.B](#), [LT.BU](#), [LT.H](#), [LT.HU](#), [LT.W](#), [LT.WU](#)

See Also Addendum for TriCore Arch Manual, Vol2, V1.3.8, pages 493/494

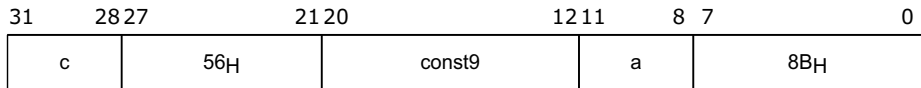
**EQANY.B**  
Equal Any Byte

**EQANY.H**  
Equal Any Half-word

**Description**

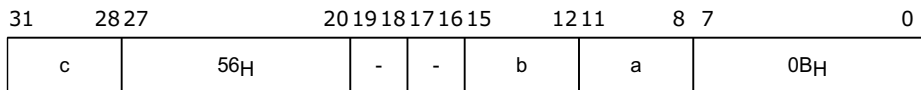
Compare each byte (EQANY.B) or half-word (EQANY.H) of D[a] with the corresponding byte or half-word of either D[b] (instruction format RR) or const9 (instruction format RC). If the logical OR of the Boolean results from each comparison is TRUE, set the least-significant bit of D[c] to 1 and clear the remaining bits to zero; otherwise clear all bits in D[c]. Const9 is sign-extended.

**EQANY.B                      D[c], D[a], const9 (RC)**



```
result_byte3 = (D[a][31:24] == sign_ext(const9)[31:24]);
result_byte2 = (D[a][23:16] == sign_ext(const9)[23:16]);
result_byte1 = (D[a][15:8] == sign_ext(const9)[15:8]);
result_byte0 = (D[a][7:0] == sign_ext(const9)[7:0]);
result = result_byte3 OR result_byte2 OR result_byte1 OR result_byte0;
D[c] = zero_ext(result);
```

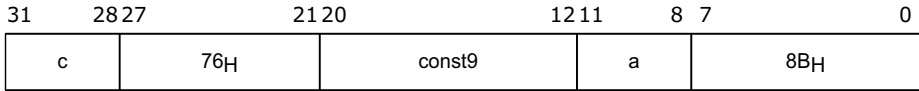
**EQANY.B                      D[c], D[a], D[b] (RR)**



```
result_byte3 = (D[a][31:24] == D[b][31:24]);
result_byte2 = (D[a][23:16] == D[b][23:16]);
result_byte1 = (D[a][15:8] == D[b][15:8]);
result_byte0 = (D[a][7:0] == D[b][7:0]);
result = result_byte3 OR result_byte2 OR result_byte1 OR result_byte0;
D[c] = zero_ext(result);
```

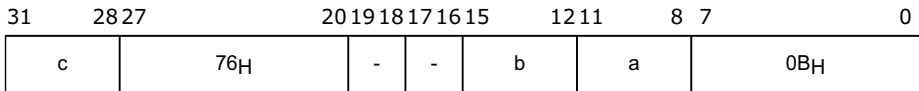
See Also Addendum for TriCore Arch Manual, Vol2, V1.3.8, pages 493/494

**EQANY.H**                      **D[c], D[a], const9 (RC)**



```
result_halfword1 = (D[a][31:16] == sign_ext(const9)[31:16]);
result_halfword0 = (D[a][15:0] == sign_ext(const9)[15:0]);
result = result_halfword1 OR result_halfword0;
D[c] = zero_ext(result);
```

**EQANY.H**                      **D[c], D[a], D[b] (RR)**



```
result_halfword1 = (D[a][31:16] == D[b][31:16]);
result_halfword0 = (D[a][15:0] == D[b][15:0]);
result = result_halfword1 OR result_halfword0;
D[c] = zero_ext(result);
```

**Status Flags**

C	Not set by these instructions.
V	Not set by these instructions.
SV	Not set by these instructions.
AV	Not set by these instructions.
SAV	Not set by these instructions.

**Examples**

```
eqany.b    d3, d1, d2
eqany.b    d3, d1, #126
eqany.h    d3, d1, d2
eqany.h    d3, d1, #126
```

**See Also**

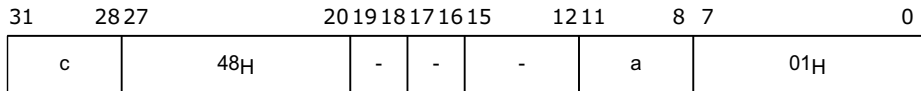
[EQ](#), [GE.U](#), [LT](#), [LT.U](#), [NE](#)

## EQZ.A Equal Zero Address

### Description

If the contents of address register A[a] are equal to zero, set the least significant bit of D[c] to one and clear the remaining bits to zero; otherwise clear all bits in D[c].

### EQZ.A                      D[c], A[a] (RR)



$D[c] = (A[a] == 0);$

### Status Flags

C	Not set by this instruction.
V	Not set by this instruction.
SV	Not set by this instruction.
AV	Not set by this instruction.
SAV	Not set by this instruction.

### Examples

`eqz.a d3, a4`

### See Also

[EQ.A](#), [GE.A](#), [LT.A](#), [NE](#), [NEZ.A](#)

**EXTR**  
Extract Bit Field  
**EXTR.U**  
Extract Bit Field Unsigned

**Description**

Extract the number of consecutive bits specified by either E[d][36:32] (instruction format RRRR) or width (instruction formats RRRW and RRPW) from D[a], starting at the bit number specified by either E[d][4:0] (instruction format RRRR), D[d][4:0] (instruction format RRRW) or pos (instruction format RRPW). Put the result in D[c], sign-extended (EXTR) or zero-extended (EXTR.U).

**EXTR**                      **D[c], D[a], pos, width (RRPW)**

31	28 27	23 22 21 20	16 15	12 11	8 7	0
c	pos	02H	width	-	a	37H

$D[c] = \text{sign\_ext}((D[a] \gg \text{pos})[\text{width}-1:0]);$

If  $\text{pos} + \text{width} > 32$  or if  $\text{width} = 0$ , then the results are undefined.

**EXTR**                      **D[c], D[a], E[d] (RRRR)**

31	28 27	24 23	21 20	16 15	12 11	8 7	0
c	d	02H	-	-	a	17H	

$\text{width} = E[d][36:32];$

$D[c] = \text{sign\_ext}((D[a] \gg E[d][4:0])[\text{width}-1:0]);$

If  $E[d][4:0] + \text{width} > 32$  or if  $\text{width} = 0$ , then the results are undefined.

**EXTR**                      **D[c], D[a], D[d], width (RRRW)**

31	28 27	24 23	21 20	16 15	12 11	8 7	0
c	d	02H	width	-	a	57H	

$D[c] = \text{sign\_ext}((D[a] \gg D[d][4:0])[\text{width}-1:0]);$

If  $D[d][4:0] + \text{width} > 32$  or if  $\text{width} = 0$ , then the results are undefined.

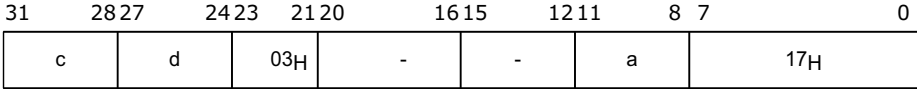
**EXTR.U**                      **D[c], D[a], pos, width (RRPW)**

31	28 27	23 22 21 20	16 15	12 11	8 7	0
c	pos	03H	width	-	a	37H

$D[c] = \text{zero\_ext}((D[a] \gg \text{pos})[\text{width}-1:0]);$

If  $\text{pos} + \text{width} > 32$  or if  $\text{width} = 0$ , then the results are undefined.

**EXTR.U**                      **D[c], D[a], E[d] (RRRR)**

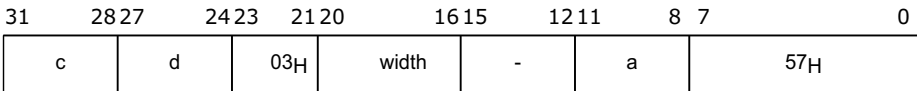


$\text{width} = \text{E}[d][36:32]$ ;

$\text{D}[c] = \text{zero\_ext}((\text{D}[a] \gg \text{E}[d][4:0])[\text{width}-1:0])$ ;

If  $\text{E}[d][4:0] + \text{width} > 32$  or if  $\text{width} = 0$ , then the results are undefined.

**EXTR.U**                      **D[c], D[a], D[d], width (RRRW)**



$\text{D}[c] = \text{zero\_ext}((\text{D}[a] \gg \text{D}[d][4:0])[\text{width}-1:0])$ ;

If  $\text{D}[d][4:0] + \text{width} > 32$  or if  $\text{width} = 0$ , then the results are undefined.

**Status Flags**

C	Not set by these instructions.
V	Not set by these instructions.
SV	Not set by these instructions.
AV	Not set by these instructions.
SAV	Not set by these instructions.

**Examples**

```
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
```

**See Also**

[DEXTR](#), [INSERT](#), [INS.T](#), [INSN.T](#)

## GE

Greater Than or Equal

## GE.U

Greater Than or Equal Unsigned

### Description

If the contents of data register D[a] are greater than or equal to the contents of either data register D[b] (instruction format RR) or const9 (instruction format RC), set the least-significant bit of D[c] to one and clear the remaining bits to zero; otherwise clear all bits in D[c]. D[a] and D[b] are treated as 32-bit signed (GE) or unsigned (GE.U) integers. The const9 value is sign-extended (GE) or zero-extended (GE.U).

### GE D[c], D[a], const9 (RC)

31	28 27	21 20	12 11	8 7	0
c	14H	const9	a	8BH	

result = (D[a] >= sign\_ext(const9));

D[c] = zero\_ext(result);

### GE D[c], D[a], D[b] (RR)

31	28 27	20 19 18 17 16 15	12 11	8 7	0
c	14H	- -	b	a	0BH

result = (D[a] >= D[b]);

D[c] = zero\_ext(result);

### GE.U D[c], D[a], const9 (RC)

31	28 27	21 20	12 11	8 7	0
c	15H	const9	a	8BH	

result = (D[a] >= sign\_ext(const9)); // unsigned

D[c] = zero\_ext(result);

### GE.U D[c], D[a], D[b] (RR)

31	28 27	20 19 18 17 16 15	12 11	8 7	0
c	15H	- -	b	a	0BH

result = (D[a] >= D[b]); // unsigned

D[c] = zero\_ext(result);



### Status Flags

C	Not set by these instructions.
V	Not set by these instructions.
SV	Not set by these instructions.
AV	Not set by these instructions.
SAV	Not set by these instructions.

### Examples

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

### See Also

[EQ](#), [LT](#), [LT.U](#), [NE](#), [EQANY.B](#), [EQANY.H](#)

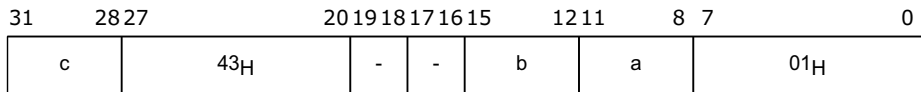
See Also Addendum for TriCore Arch Manual, Vol2, V1.3.8, pages 493/494

## GE.A Greater Than or Equal Address

### Description

If the contents of address register A[a] are greater than or equal to the contents of address register A[b], set the least-significant bit of D[c] to one and clear the remaining bits to zero; otherwise clear all bits in D[c]. Operands are treated as unsigned 32-bit integers.

### GE.A                      D[c], A[a], A[b] (RR)



D[c] = (A[a] >= A[b]); // unsigned

### Status Flags

C	Not set by this instruction.
V	Not set by this instruction.
SV	Not set by this instruction.
AV	Not set by this instruction.
SAV	Not set by this instruction.

### Examples

ge.a    d3, a4, a2

### See Also

[EQ.A](#), [EQZ.A](#), [LT.A](#), [NE](#), [NEZ.A](#)

## IMASK Insert Mask

### Description

Create a mask containing the number of bits specified by width, starting at the bit number specified by either D[d][4:0] (instruction formats RRRW and RCRW) or pos (instruction formats RRPW and RCPW), and put the mask in data register E[c][63:32].

Left-shift the value in either D[b] (formats RRRW and RRPW) or const4 (formats RCRW and RCPW) by the amount specified by either D[d][4:0] (formats RRRW and RCRW) or pos (formats RRPW and RCPW) and put the result value in data register E[c][31:0].

The value const4 is zero-extended. 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.

### IMASK                      E[c], const4, pos, width (RCPW)

31	28 27	23 22 21 20	16 15	12 11	8 7	0
c	pos	01 <sub>H</sub>	width	const4	-	B7 <sub>H</sub>

$$E[c][63:32] = ((2^{\text{width}} - 1) \ll \text{pos});$$

$$E[c][31:0] = (\text{zero\_ext}(\text{const4}) \ll \text{pos});$$

If pos + width > 31 the result is undefined.

### IMASK                      E[c], const4, D[d], width (RCRW)

31	28 27	24 23	21 20	16 15	12 11	8 7	0
c	d	01 <sub>H</sub>	width	const4	-	D7 <sub>H</sub>	

$$E[c][63:32] = ((2^{\text{width}} - 1) \ll D[d][4:0]);$$

$$E[c][31:0] = (\text{zero\_ext}(\text{const4}) \ll D[d][4:0]);$$

If (D[d][4:0] + width) > 31 the result is undefined.

### IMASK                      E[c], D[b], pos, width (RRPW)

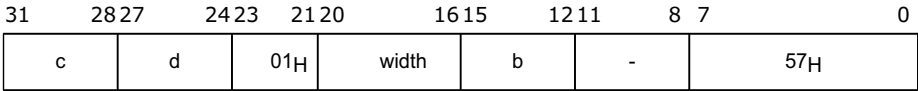
31	28 27	23 22 21 20	16 15	12 11	8 7	0
c	pos	01 <sub>H</sub>	width	b	-	37 <sub>H</sub>

$$E[c][63:32] = ((2^{\text{width}} - 1) \ll \text{pos});$$

$$E[c][31:0] = (D[b][31:0] \ll \text{pos});$$

If (pos + width) > 31 the result is undefined.

**IMASK**                      E[c], D[b], D[d], width (RRRW)



$E[c][63:32] = ((2^{\text{width}} - 1) \ll D[d][4:0]);$

$E[c][31:0] = (D[b] \ll D[d][4:0]);$

If  $(D[d][4:0] + \text{width}) > 31$  the result is undefined.

**Status Flags**

C	Not set by this instruction.
V	Not set by this instruction.
SV	Not set by this instruction.
AV	Not set by this instruction.
SAV	Not set by this instruction.

**Examples**

```
imask    e2, d1, d2, #11
imask    e2, d1, #5, #11
imask    e2, #6, d2, #11
imask    e2, #6, #5, #11
```

**See Also**

[LDMST](#), [ST.T](#)



## INSERT Insert Bit Field

### Description

Starting at bit zero, extract from either D[b] (instruction formats RRRR, RRRW, RRPW) or const4 (instruction formats RCRR, RCRW, RCPW) the number of consecutive bits specified by either E[d][36:32] (formats RRRR, RCRR) or width (formats RRRW, RRPW, RCRW, RCPW).

Shift the result left by the number of bits specified by either E[d][4:0] (formats RRRR, RCRR), D[d] (formats RRRW, RCRW) or pos (formats RRPW, RCPW); extract a copy of D[a], clearing the bits starting at the bit position specified by either E[d][4:0] (formats RRRR, RCRR), D[d] (formats RRRW, RCRW) or pos (formats RRPW, RCPW), and extending for the number of bits specified by either E[d][36:32] (formats RRRR, RCRR) or width (formats RRRW, RRPW, RCRW, RCPW). Put the bitwise OR of the two extracted words into D[c].

### INSERT D[c], D[a], const4, pos, width (RCPW)

31	28 27	23 22 21 20	16 15	12 11	8 7	0
c	pos	00H	width	const4	a	B7H

mask = (2<sup>width</sup> - 1) << pos;

D[c] = (D[a] & ~mask) | ((zero\_ext(const4) << pos) & mask);

If pos + width > 32, then the result is undefined.

### INSERT D[c], D[a], const4, E[d] (RCRR)

31	28 27	24 23	21 20	16 15	12 11	8 7	0
c	d	00H	-	const4	a	97H	

width = E[d][36:32];

mask = (2<sup>width</sup> - 1) << E[d][4:0];

D[c] = (D[a] & ~mask) | ((zero\_ext(const4) << E[d][4:0]) & mask);

If E[d][4:0] + E[d][36:32] > 32, then the result is undefined.

### INSERT D[c], D[a], const4, D[d], width (RCRW)

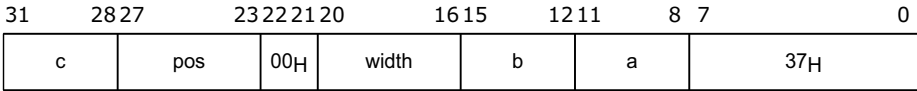
31	28 27	24 23	21 20	16 15	12 11	8 7	0
c	d	00H	width	const4	a	D7H	

mask = (2<sup>width</sup> - 1) << D[d][4:0];

D[c] = (D[a] & ~mask) | ((zero\_ext(const4) << D[d][4:0]) & mask);

If  $D[d][4:0] + \text{width} > 32$ , then the result is undefined.

**INSERT                      D[c], D[a], D[b], pos, width (RRPW)**

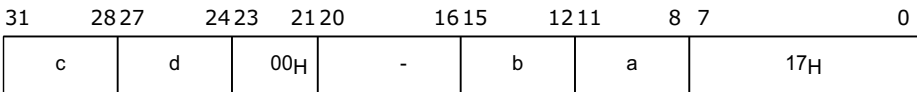


$\text{mask} = (2^{\text{width}} - 1) \ll \text{pos};$

$D[c] = (D[a] \& \sim\text{mask}) \mid ((D[b] \ll \text{pos}) \& \text{mask});$

If  $\text{pos} + \text{width} > 32$ , then the result is undefined.

**INSERT                      D[c], D[a], D[b], E[d] (RRRR)**



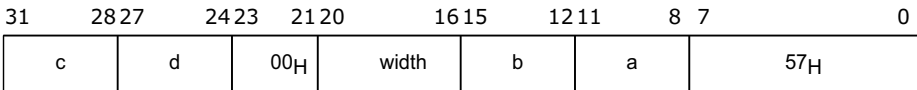
$\text{width} = E[d][36:32];$

$\text{mask} = (2^{\text{width}} - 1) \ll E[d][4:0];$

$D[c] = (D[a] \& \sim\text{mask}) \mid ((D[b] \ll E[d][4:0]) \& \text{mask});$

If  $E[d][4:0] + E[d][36:32] > 32$ , then the result is undefined.

**INSERT                      D[c], D[a], D[b], D[d], width (RRRW)**



$\text{mask} = (2^{\text{width}} - 1) \ll D[d][4:0];$

$D[c] = (D[a] \& \sim\text{mask}) \mid ((D[b] \ll D[d][4:0]) \& \text{mask});$

If  $D[d][4:0] + \text{width} > 32$ , then the result is undefined.

**Status Flags**

C	Not set by this instruction.
V	Not set by this instruction.
SV	Not set by this instruction.
AV	Not set by this instruction.
SAV	Not set by this instruction.

**Examples**

```
insert d3, d1, d2, e4
```

```
insert d3, d1, d2, d4, #8  
insert d3, d1, d2, #16, #8  
insert d3, d1, 0, e4  
insert d3, d1, 0, d4, #8  
insert d3, d1, 0, #16, #8
```

**See Also**

[DEXTR](#), [EXTR](#), [EXTR.U](#), [INS.T](#), [INSN.T](#)

See Also Addendum for TriCore Arch Manual, Vol2, V1.3.8, pages 493/494



## ISYNC

### Synchronize Instructions

#### Description

The ISYNC instruction forces completion of all previous instructions, then flushes the CPU pipelines and invalidates any cached pipeline state before proceeding to the next instruction.

*Note: I-cache is not invalidated by ISYNC.*

*Note: An ISYNC instruction should follow a MTRC instruction. This ensures that all instructions following the MTRC see the effects of the CSFR update.*

#### ISYNC (SYS)

31	28 27	22 21	12 11	8 7	0
-	13H	-	-	0DH	

#### Status Flags

C	Not set by this instruction.
V	Not set by this instruction.
SV	Not set by this instruction.
AV	Not set by this instruction.
SAV	Not set by this instruction.

#### Examples

```
isync
```

#### See Also

[DSYNC](#)

## IXMAX

Find Maximum Index

### IXMAX.U

Find Maximum Index (unsigned)

#### Description

Enables a search of maximum value and its related index in a vector of 16-bit signed (IXMAX) or unsigned (IXMAX.U) values.

The IXMAX and IXMAX.U instructions are not available in the TriCore 1.2 Architecture.

For all operations:

- E[d][15:0] Working index.
- E[d][31:16] Current index of maximum.
- E[d][47:32] Current value of maximum.
- E[d][63:48] 00<sub>H</sub>.
- D[b][15:0] First compare value.
- D[b][31:16] Second compare value.
- E[c][15:0] Update working index.
- E[c][31:16] Update index of maximum.
- E[c][47:32] Update value of maximum.
- E[c][63:48] 00<sub>H</sub>.

#### IXMAX

E[c], E[d], D[b] (RRR)

31	28 27	24 23	20 19 18 17 16 15	12 11	8 7	0	
c	d	0A <sub>H</sub>	-	0 <sub>H</sub>	b	-	6B <sub>H</sub>

E[c][15:0] = E[d][15:0] + 2;

E[c][63:48] = 00<sub>H</sub>;

if (D[b][15:0] >= D[b][31:16]) AND (D[b][15:0] > E[d][47:32]) then {

E[c][47:32] = D[b][15:0];

E[c][31:16] = E[d][15:0];

} else if (D[b][31:16] > D[b][15:0]) AND (D[b][31:16] > E[d][47:32]) then {

E[c][47:32] = D[b][31:16];

E[c][31:16] = E[d][15:0]+1;

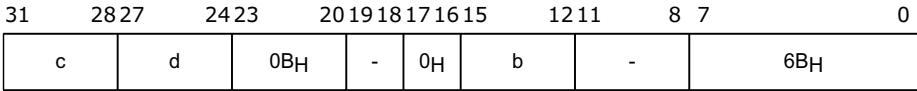
} else {

E[c][47:32] = E[d][47:32];

E[c][31:16] = E[d][31:16];

}

**IXMAX.U**                      **E[c], E[d], D[b] (RRR)**



For IXMAX.U, the comparison is on unsigned numbers.

$$E[c][15:0] = E[d][15:0] + 2;$$

$$E[c][63:48] = 00_H;$$

if (D[b][15:0] >= D[b][31:16]) AND (D[b][15:0] > E[d][47:32]) then {

$$E[c][47:32] = D[b][15:0];$$

$$E[c][31:16] = E[d][15:0];$$

} else if (D[b][31:16] > D[b][15:0]) AND (D[b][31:16] > E[d][47:32]) then {

$$E[c][47:32] = D[b][31:16];$$

$$E[c][31:16] = E[d][15:0] + 1;$$

} else {

$$E[c][47:32] = E[d][47:32];$$

$$E[c][31:16] = E[d][31:16];$$

}

For all index additions, on overflow: wrapping, no trap.

If the 1st compare value and 2nd compare value and current maximum value are the same, the priority select is: current maximum is the highest priority then 1st compare value and 2nd compare value is in the lowest priority.

**Status Flags**

C	Not set by these instructions.
V	Not set by these instructions.
SV	Not set by these instructions.
AV	Not set by these instructions.
SAV	Not set by these instructions.

**Examples**

ixmax            e2, e8, d6

ixmax.u        e2, e0, d4

**See Also**

**IXMIN**

See Also Addendum for TriCore Arch Manual, Vol2, V1.3.8, pages 493/494

## IXMIN

Find Minimum Index

### IXMIN.U

Find Minimum Index (unsigned)

#### Description

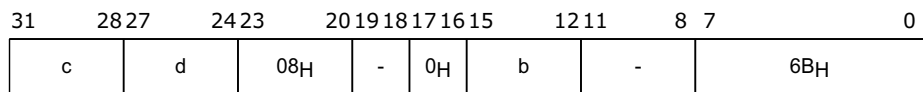
Enables search of minimum value and its related index in a vector of 16-bit signed (IXMIN) or unsigned (IXMIN.U) values.

The IXMIN and IXMIN.U instructions are not available in the TriCore 1.2 Architecture.

For all operations:

- E[d][15:0] Working index.
- E[d][31:16] Current index of minimum.
- E[d][47:32] Current value of minimum.
- E[d][63:48] 00<sub>H</sub>.
- D[b][15:0] First compare value.
- D[b][31:16] Second compare value.
- E[c][15:0] Update working index.
- E[c][31:16] Update index of minimum.
- E[c][47:32] Update value of minimum.
- E[c][63:48] 00<sub>H</sub>.

#### IXMIN                      E[c], E[d], D[b] (RRR)



E[c][15:0] = E[d][15:0] + 2;

E[c][63:48] = 00<sub>H</sub>;

if (D[b][15:0] <= D[b][31:16]) AND (D[b][15:0] < E[d][47:32]) then {

E[c][47:32] = D[b][15:0];

E[c][31:16] = E[d][15:0];

} else if (D[b][31:16] < D[b][15:0]) AND (D[b][31:16] < E[d][47:32]) then {

E[c][47:32] = D[b][31:16];

E[c][31:16] = E[d][15:0]+1;

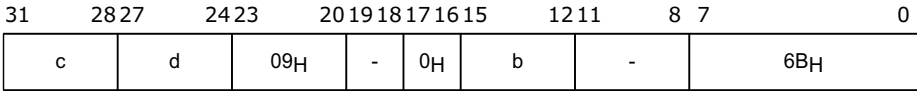
} else {

E[c][47:32] = E[d][47:32];

E[c][31:16] = E[d][31:16];

}

**IXMIN.U**                      **E[c], E[d], D[b] (RRR)**



For IXMIN.U, the comparison is on unsigned numbers.

$E[c][15:0] = E[d][15:0] + 2;$

$E[c][63:48] = 00_H;$

if  $(D[b][15:0] \leq D[b][31:16])$  AND  $(D[b][15:0] < E[d][47:32])$  then {

$E[c][47:32] = D[b][15:0];$

$E[c][31:16] = E[d][15:0];$

} else if  $(D[b][31:16] < D[b][15:0])$  AND  $(D[b][31:16] < E[d][47:32])$  then {

$E[c][47:32] = D[b][31:16];$

$E[c][31:16] = E[d][15:0] + 1;$

} else {

$E[c][47:32] = E[d][47:32];$

$E[c][31:16] = E[d][31:16];$

}

For all index additions, on overflow: wrapping, no trap

If the 1st compare value and 2nd compare value and current minimum value are the same, the priority select is: current minimum is the highest priority then 1st compare value and 2nd compare value is in the lowest priority.

**Status Flags**

C	Not set by these instructions.
V	Not set by these instructions.
SV	Not set by these instructions.
AV	Not set by these instructions.
SAV	Not set by these instructions.

**Examples**

`ixmin            e10, e2, d0`

`ixmin.u        e14, e2, d7`

**See Also**

**IXMAX**

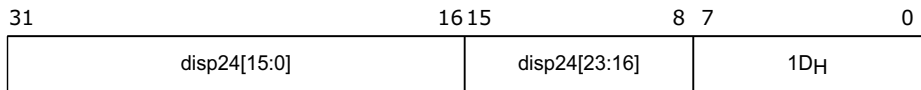
## J Jump Unconditional

### Description

Add the value specified by disp24, sign-extended and multiplied by 2, to the contents of PC and jump to that address.

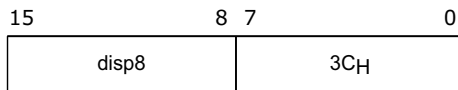
Add the value specified by disp8, sign-extended and multiplied by 2, to the contents of PC and jump to that address.

### J                      disp24 (B)



PC = PC + sign\_ext(disp24) \* 2;

### J                      disp8 (SB)



PC = PC + sign\_ext(disp8) \* 2;

### Status Flags

C	Not set by this instruction.
V	Not set by this instruction.
SV	Not set by this instruction.
AV	Not set by this instruction.
SAV	Not set by this instruction.

### Examples

j foobar

j foobar

### See Also

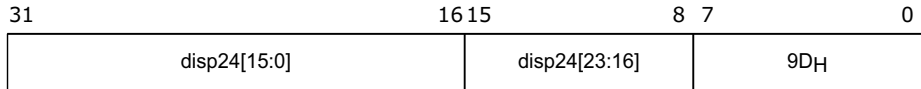
[JA](#), [JI](#), [JL](#), [JLA](#), [JLI](#), [LOOPU](#)

## JA Jump Unconditional Absolute

### Description

Load the value specified by disp24 into PC and jump to that address.  
The value disp24 is used to form the Effective Address (EA).

### JA                      disp24 (B)



PC = {disp24[23:20], 7'b0000000, disp24[19:0], 1'b0};

### Status Flags

C	Not set by this instruction.
V	Not set by this instruction.
SV	Not set by this instruction.
AV	Not set by this instruction.
SAV	Not set by this instruction.

### Examples

```
ja            foobar
```

### See Also

[J](#), [JI](#), [JL](#), [JLA](#), [JLI](#), [LOOPU](#)

See Also Addendum for TriCore Arch Manual, Vol2, V1.3.8, pages 493/494

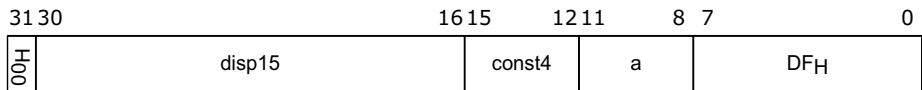
## JEQ Jump if Equal

### Description

If the contents of D[a] are equal to the contents of either D[b] or const4, then add the value specified by disp15, sign-extended and multiplied by 2, to the contents of PC and jump to that address. The const4 value is sign-extended.

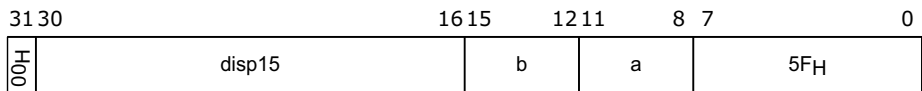
If the contents of D[15] are equal to the contents of either D[b] or const4, then add the value specified by disp4, zero-extended and multiplied by 2, to the contents of PC and jump to that address. The const4 value is sign-extended.

### JEQ D[a], const4, disp15 (BRC)



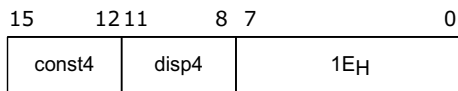
if (D[a] == sign\_ext(const4)) then PC = PC + sign\_ext(disp15) \* 2;

### JEQ D[a], D[b], disp15 (BRR)



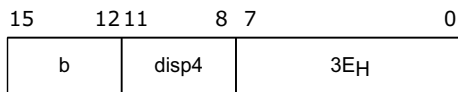
if (D[a] == D[b]) then PC = PC + sign\_ext(disp15) \* 2;

### JEQ D[15], const4, disp4 (SBC)



if (D[15] == sign\_ext(const4)) then PC = PC + zero\_ext(disp4) \* 2;

### JEQ D[15], D[b], disp4 (SBR)



if (D[15] == D[b]) then PC = PC + zero\_ext(disp4) \* 2;

### Status Flags

C	Not set by this instruction.
---	------------------------------



Instruction Set

V	Not set by this instruction.
SV	Not set by this instruction.
AV	Not set by this instruction.
SAV	Not set by this instruction.

**Examples**

```
jeq d1, d2, foobar  
jeq d1, #6, foobar
```

```
jeq d15, d2, foobar  
jeq d15, #6, foobar
```

**See Also**

[JGE](#), [JGE.U](#), [JLT](#), [JLT.U](#), [JNE](#)

See Also Addendum for TriCore Arch Manual, Vol2, V1.3.8, pages 493/494

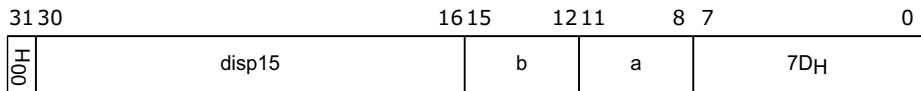
## JEQ.A

### Jump if Equal Address

#### Description

If the contents of A[a] are equal to the contents of A[b], then add the value specified by disp15, sign-extended and multiplied by 2, to the contents of PC and jump to that address.

#### JEQ.A                      A[a], A[b], disp15 (BRR)



if (A[a] == A[b]) then PC = PC + sign\_ext(disp15) \* 2;

#### Status Flags

C	Not set by this instruction.
V	Not set by this instruction.
SV	Not set by this instruction.
AV	Not set by this instruction.
SAV	Not set by this instruction.

#### Examples

```
jeq.a    a4, a2, foobar
```

#### See Also

[JNE.A](#)

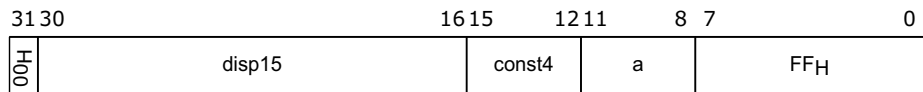
**JGE**  
**Jump if Greater Than or Equal**  
**JGE.U**  
**Jump if Greater Than or Equal Unsigned**

**Description**

If the contents of D[a] are greater than or equal to the contents of either D[b] (instruction format BRR) or const4 (instruction format BRC), then add the value specified by disp15, sign-extended and multiplied by 2, to the contents of PC and jump to that address.

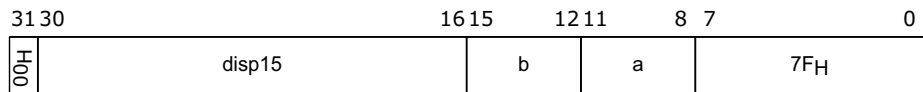
Operands are treated as signed (JGE) or unsigned (JGE.U), 32-bit integers. The const4 value is sign-extended (JGE) or zero-extended (JGE.U).

**JGE D[a], const4, disp15 (BRC)**



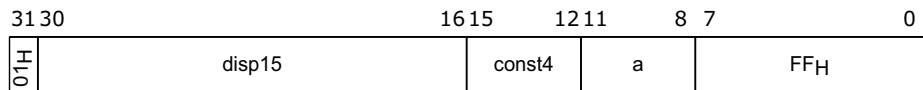
if (D[a] >= sign\_ext(const4)) then PC = PC + sign\_ext(disp15) \* 2;

**JGE D[a], D[b], disp15 (BRR)**



if (D[a] >= D[b]) then PC = PC + sign\_ext(disp15) \* 2;

**JGE.U D[a], const4, disp15 (BRC)**

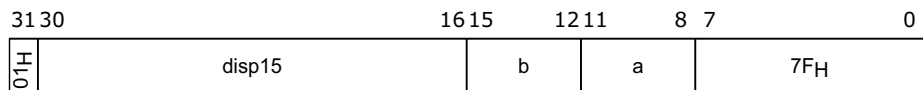


if (D[a] >= zero\_ext(const4)) then { // unsigned comparison

PC = PC + sign\_ext(disp15) \* 2;

}

**JGE.U D[a], D[b], disp15 (BRR)**



if (D[a] >= D[b]) then PC = PC + sign\_ext(disp15) \* 2; // unsigned comparison

### Status Flags

C	Not set by these instructions.
V	Not set by these instructions.
SV	Not set by these instructions.
AV	Not set by these instructions.
SAV	Not set by these instructions.

### Examples

```
jge    d1, d2, foobar
jge    d1, #6, foobar
jge.u  d1, d2, foobar
jge.u  d1, #6, foobar
```

### See Also

[JEQ](#), [JLT](#), [JLT.U](#), [JNE](#)

## JGEZ (16-bit)

### Jump if Greater Than or Equal to Zero (16-bit)

#### Description

If the contents of D[b] are greater than or equal to zero, then add the value specified by disp4, zero-extended and multiplied by 2, to the contents of PC and jump to that address.

#### JGEZ                      D[b], disp4 (SBR)

15      12 11              8 7                              0

b	disp4	CE <sub>H</sub>
---	-------	-----------------

if (D[b] >= 0) then PC = PC + zero\_ext(disp4) \* 2;

#### Status Flags

C	Not set by this instruction.
V	Not set by this instruction.
SV	Not set by this instruction.
AV	Not set by this instruction.
SAV	Not set by this instruction.

#### Examples

```
jgez d2, foobar
```

#### See Also

[JGTZ \(16-bit\)](#), [JLEZ \(16-bit\)](#), [JLTZ \(16-bit\)](#), [JNZ \(16-bit\)](#), [JZ \(16-bit\)](#)



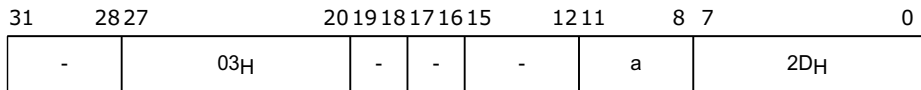
## JI Jump Indirect

### Description

Load the contents of address register A[a] into PC and jump to that address. The least-significant bit is always set to 0.

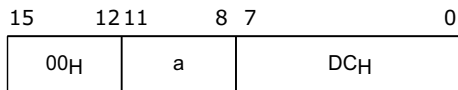
Load the contents of address register A[a] into PC and jump to that address. The least-significant bit is always set to 0.

### JI A[a] (RR)



PC = {A[a][31:1], 1'b0};

### JI A[a] (SR)



PC = {A[a][31:1], 1'b0};

### Status Flags

C	Not set by this instruction.
V	Not set by this instruction.
SV	Not set by this instruction.
AV	Not set by this instruction.
SAV	Not set by this instruction.

### Examples

ji a2

ji a2

### See Also

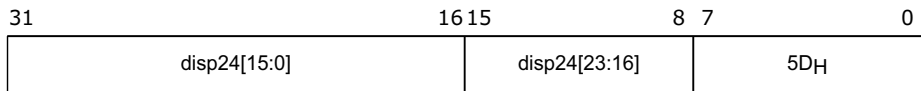
[J](#), [JA](#), [JL](#), [JLA](#), [JLI](#), [LOOPU](#)

## JL Jump and Link

### Description

Store the address of the next instruction in A[11] (return address). Add the value specified by disp24, sign-extended and multiplied by 2, to the contents of PC and jump to that address.

### JL                      disp24 (B)



A[11] = PC + 4;

PC = PC + sign\_ext(disp24) \* 2;

### Status Flags

C	Not set by this instruction.
V	Not set by this instruction.
SV	Not set by this instruction.
AV	Not set by this instruction.
SAV	Not set by this instruction.

### Examples

```
jl    foobar
```

### See Also

[J](#), [JI](#), [JA](#), [JLA](#), [JLI](#), [CALLA](#), [LOOPU](#)



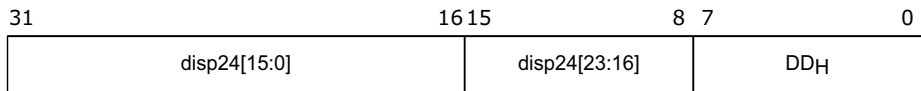
## JLA

### Jump and Link Absolute

#### Description

Store the address of the next instruction in A[11] (return address). Load the value specified by disp24 into PC and jump to that address. The value disp24 is used to form the effective address (EA).

#### JLA                      disp24 (B)



A[11] = PC + 4;

PC = {disp24[23:20], 7'b0000000, disp24[19:0], 1'b0};

#### Status Flags

C	Not set by this instruction.
V	Not set by this instruction.
SV	Not set by this instruction.
AV	Not set by this instruction.
SAV	Not set by this instruction.

#### Examples

```
jla foobar
```

#### See Also

[J](#), [JI](#), [JA](#), [JL](#), [JLI](#), [CALLA](#), [LOOPU](#)



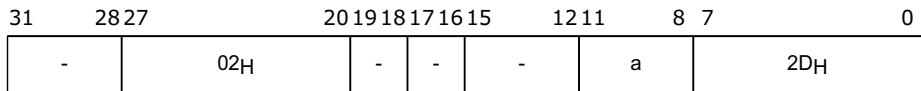
## JLI

### Jump and Link Indirect

#### Description

Store the address of the next instruction in A[11] (return address). Load the contents of address register A[a] into PC and jump to that address. The least-significant bit is set to zero.

#### JLI                      A[a] (RR)



A[11] = PC + 4;

PC = {A[a][31:1], 1'b0};

#### Status Flags

C	Not set by this instruction.
V	Not set by this instruction.
SV	Not set by this instruction.
AV	Not set by this instruction.
SAV	Not set by this instruction.

#### Examples

```
jli    a2
```

#### See Also

[J](#), [JI](#), [JA](#), [JL](#), [JLA](#), [LOOPU](#)

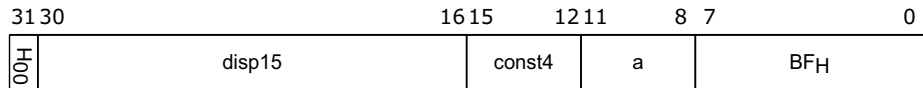
See Also Addendum for TriCore Arch Manual, Vol2, V1.3.8, pages 493/494

**JLT**  
**Jump if Less Than**  
**JLT.U**  
**Jump if Less Than Unsigned**

**Description**

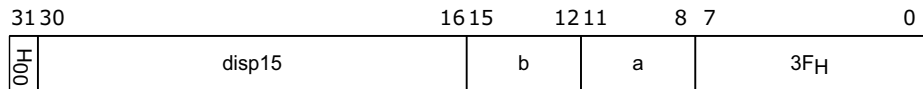
If the contents of D[a] are less than the contents of either D[b] (instruction format BRR) or const4 (instruction format BRC), then add the value specified by disp15, sign-extended and multiplied by 2, to the contents of PC and jump to that address. The operands are treated as signed (JLT) or unsigned (JLT.U), 32-bit integers. The const4 value is sign-extended (JLT) or zero-extended (JLT.U).

**JLT** **D[a], const4, disp15 (BRC)**



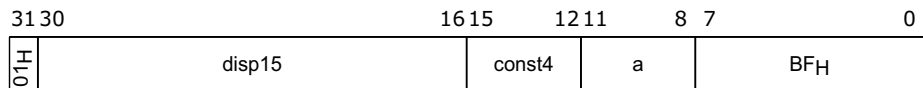
if (D[a] < sign\_ext(const4)) then PC = PC + sign\_ext(disp15) \* 2;

**JLT** **D[a], D[b], disp15 (BRR)**



if (D[a] < D[b]) then PC = PC + sign\_ext(disp15) \* 2;

**JLT.U** **D[a], const4, disp15 (BRC)**

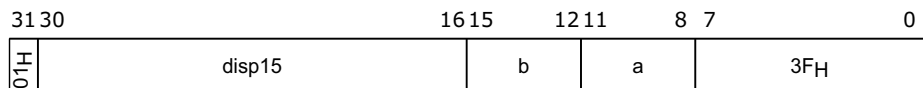


if (D[a] < zero\_ext(const4)) then { // unsigned comparison

PC = PC + sign\_ext(disp15) \* 2;

}

**JLT.U** **D[a], D[b], disp15 (BRR)**



if (D[a] < D[b]) then PC = PC + sign\_ext(disp15) \* 2; // unsigned comparison

### Status Flags

C	Not set by these instructions.
V	Not set by these instructions.
SV	Not set by these instructions.
AV	Not set by these instructions.
SAV	Not set by these instructions.

### Examples

```
jlt    d1, d2, foobar  
jlt    d1, #6, foobar  
jlt.u  d1, d2, foobar  
jlt.u  d1, #6, foobar
```

### See Also

[JEQ](#), [JGE](#), [JGE.U](#), [JNE](#)

## JLTZ (16-bit)

### Jump if Less Than Zero (16-bit)

#### Description

If the contents of D[b] are less than zero then add the value specified by disp4, zero-extended and multiplied by 2, to the contents of PC and jump to that address.

#### JLTZ                      D[b], disp4 (SBR)

15      12 11              8 7                      0

b	disp4	0EH
---	-------	-----

if (D[b] < 0) then PC = PC + zero\_ext(disp4) \* 2;

#### Status Flags

C	Not set by this instruction.
V	Not set by this instruction.
SV	Not set by this instruction.
AV	Not set by this instruction.
SAV	Not set by this instruction.

#### Examples

```
jltz d2, foobar
```

#### See Also

[JGEZ \(16-bit\)](#), [JGTZ \(16-bit\)](#), [JLEZ \(16-bit\)](#), [JNZ \(16-bit\)](#), [JZ \(16-bit\)](#)

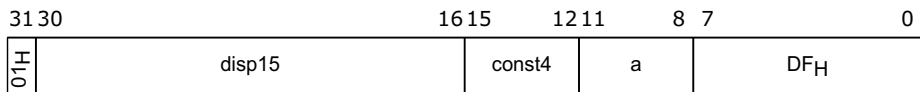
## JNE Jump if Not Equal

### Description

If the contents of D[a] are not equal to the contents of either D[b] (instruction format BRR) or const4 (instruction format BRC), then add the value specified by disp15, sign-extended and multiplied by 2, to the contents of PC and jump to that address. The const4 value is sign-extended.

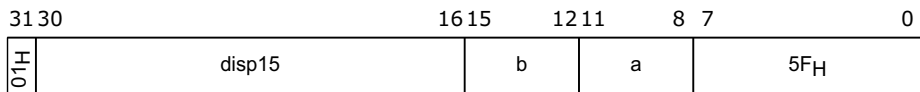
If the contents of D[15] are not equal to the contents of either D[b] (instruction format SBR) or const4 (instruction format SBC), then add the value specified by disp4, zero-extended and multiplied by 2, to the contents of PC and jump to that address. The const4 value is sign-extended.

### JNE D[a], const4, disp15 (BRC)



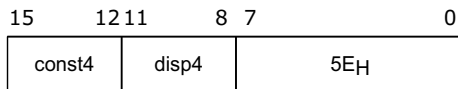
if (D[a] != sign\_ext(const4)) then PC = PC + sign\_ext(disp15) \* 2;

### JNE D[a], D[b], disp15 (BRR)



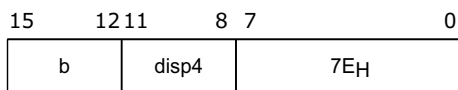
if (D[a] != D[b]) then PC = PC + sign\_ext(disp15) \* 2;

### JNE D[15], const4, disp4 (SBC)



if (D[15] != sign\_ext(const4)) then PC = PC + zero\_ext(disp4) \* 2;

### JNE D[15], D[b], disp4 (SBR)



if (D[15] != D[b]) then PC = PC + zero\_ext(disp4) \* 2;

### Status Flags

C	Not set by this instruction.
V	Not set by this instruction.
SV	Not set by this instruction.
AV	Not set by this instruction.
SAV	Not set by this instruction.

### Examples

```
jne d1, d2, foobar  
jne d1, #6, foobar
```

```
jne d15, d2, foobar  
jne d15, #6, foobar
```

### See Also

[JEQ](#), [JGE](#), [JGE.U](#), [JLT](#), [JLT.U](#)



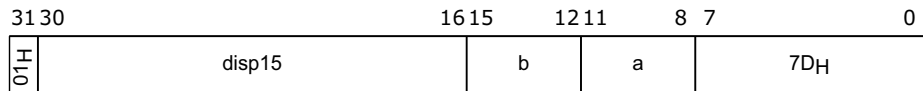
## JNE.A

### Jump if Not Equal Address

#### Description

If the contents of A[a] are not equal to the contents of A[b] then add the value specified by disp15, sign-extended and multiplied by 2, to the contents of PC and jump to that address.

#### JNE.A                      A[a], A[b], disp15 (BRR)



if (A[a] != A[b]) then PC = PC + sign\_ext(disp15) \* 2;

#### Status Flags

C	Not set by this instruction.
V	Not set by this instruction.
SV	Not set by this instruction.
AV	Not set by this instruction.
SAV	Not set by this instruction.

#### Examples

```
jne.a    a4, a2, foobar
```

#### See Also

[JEQ.A](#)

## JNED Jump if Not Equal and Decrement

### Description

If the contents of D[a] are not equal to the contents of either D[b] (instruction format BRR) or const4 (instruction format BRC), then add the value specified by disp15, sign-extended and multiplied by 2, to the contents of PC and jump to that address. Decrement the value in D[a] by one. The const4 value is sign-extended.

### JNED D[a], const4, disp15 (BRC)

31	30	16	15	12	11	8	7	0
01H		disp15			const4		a	9FH

if (D[a] != sign\_ext(const4)) then PC = PC + sign\_ext(disp15) \* 2;

D[a] = D[a] - 1;

The decrement is unconditional.

### JNED D[a], D[b], disp15 (BRR)

31	30	16	15	12	11	8	7	0
01H		disp15			b	a	1FH	

if (D[a] != D[b]) then PC = PC + sign\_ext(disp15) \* 2;

D[a] = D[a] - 1;

The decrement is unconditional.

### Status Flags

C	Not set by this instruction.
V	Not set by this instruction.
SV	Not set by this instruction.
AV	Not set by this instruction.
SAV	Not set by this instruction.

### Examples

```
jned d1, d2, foobar
jned d1, #6, foobar
```

See Also

[JNEI](#), [LOOP](#), [LOOPU](#)

See Also Addendum for TriCore Arch Manual, Vol2, V1.3.8, pages 493/494

## JNEI

### Jump if Not Equal and Increment

#### Description

If the contents of D[a] are not equal to the contents of either D[b] (instruction format BRR) or const4 (instruction format BRC), then add the value specified by disp15, sign-extended and multiplied by 2, to the contents of PC and jump to that address. Increment the value in D[a] by one. The const4 value is sign-extended.

#### JNEI D[a], const4, disp15 (BRC)

31 30	16 15	12 11	8 7	0
H OO	disp15	const4	a	9FH

if (D[a] != sign\_ext(const4)) then PC = PC + sign\_ext(disp15) \* 2;

D[a] = D[a] + 1;

The increment is unconditional.

#### JNEI D[a], D[b], disp15 (BRR)

31 30	16 15	12 11	8 7	0
H OO	disp15	b	a	1FH

if (D[a] != D[b]) then PC = PC + sign\_ext(disp15) \* 2;

D[a] = D[a] + 1;

The increment is unconditional.

#### Status Flags

C	Not set by this instruction.
V	Not set by this instruction.
SV	Not set by this instruction.
AV	Not set by this instruction.
SAV	Not set by this instruction.

#### Examples

```
jnei d1, d2, foobar
jnei d1, #6, foobar
```

See Also

[JNED](#), [LOOP](#), [LOOPU](#)

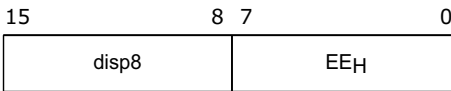
See Also Addendum for TriCore Arch Manual, Vol2, V1.3.8, pages 493/494

## JNZ (16-bit) Jump if Not Equal to Zero (16-bit)

### Description

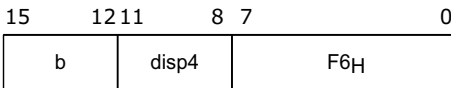
If contents of either D[b] (instruction format SBR) or D[15] (instruction format SB) are not equal to zero, then add value specified by either disp4 (format SBR) or disp8 (format SB), zero-extended (disp4) or sign-extended (disp8) and multiplied by 2, to the contents of PC and jump to that address.

### JNZ D[15], disp8 (SB)



if (D[15] != 0) then PC = PC + sign\_ext(disp8) \* 2;

### JNZ D[b], disp4 (SBR)



if (D[b] != 0) then PC = PC + zero\_ext(disp4) \* 2;

### Status Flags

C	Not set by this instruction.
V	Not set by this instruction.
SV	Not set by this instruction.
AV	Not set by this instruction.
SAV	Not set by this instruction.

### Examples

```
jnz d2, foobar
jnz d15, foobar
```

### See Also

**JGEZ (16-bit), JGTZ (16-bit), JLEZ (16-bit), JLTZ (16-bit), JZ (16-bit)**

## JNZ.A

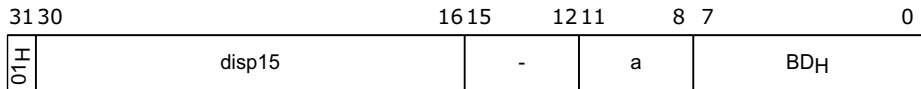
### Jump if Not Equal to Zero Address

#### Description

If the contents of A[a] are not equal to zero, then add the value specified by disp15, sign-extended and multiplied by 2, to the contents of PC and jump to that address.

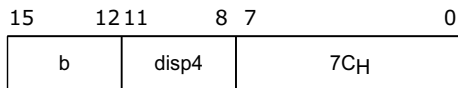
If the contents of A[b] are not equal to zero, then add the value specified by disp4, zero-extended and multiplied by 2, to the contents of PC and jump to that address.

#### JNZ.A                      A[a], disp15 (BRR)



if (A[a] != 0) then PC = PC + sign\_ext(disp15) \* 2;

#### JNZ.A                      A[b], disp4 (SBR)



if (A[b] != 0) then PC = PC + zero\_ext(disp4) \* 2;

#### Status Flags

C	Not set by this instruction.
V	Not set by this instruction.
SV	Not set by this instruction.
AV	Not set by this instruction.
SAV	Not set by this instruction.

#### Examples

```
jnz.a a4, foobar
```

```
jnz.a a4, foobar
```

#### See Also

[JZ.A](#)

## JNZ.T

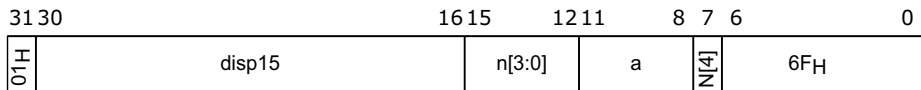
### Jump if Not Equal to Zero Bit

#### Description

If bit  $n$  of register  $D[a]$  is not equal to zero, then add the value specified by  $disp15$ , sign-extended and multiplied by 2, to the contents of PC and jump to that address.

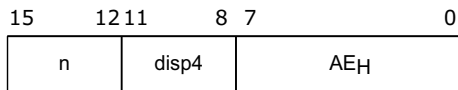
if bit  $n$  of register  $D[15]$  is not equal to zero, then add the value specified by  $disp4$ , zero-extended and multiplied by 2, to the contents of PC and jump to that address.

#### JNZ.T                      $D[a], n, disp15$ (BRN)



if  $(D[a][n])$  then  $PC = PC + sign\_ext(disp15) * 2;$

#### JNZ.T                      $D[15], n, disp4$ (SBRN)



if  $(D[15][n])$  then  $PC = PC + zero\_ext(disp4) * 2;$

#### Status Flags

C	Not set by this instruction.
V	Not set by this instruction.
SV	Not set by this instruction.
AV	Not set by this instruction.
SAV	Not set by this instruction.

#### Examples

```
jnz.t            d1, 1, foobar
```

```
jnz.t            d15, 1, foobar
```

#### See Also

[JZ.T](#)

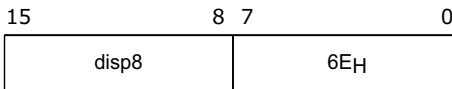


## JZ (16-bit) Jump if Zero (16-bit)

### Description

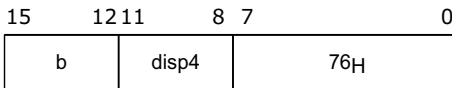
If the contents of either D[15] (instruction format SB) or D[b] (instruction format SBR) are equal to zero, then add the value specified by either disp8 (format SB) or disp4 (format SBR), sign-extended (disp8) or zero-extended (disp4) and multiplied by 2, to the contents of PC, and jump to that address.

### JZ D[15], disp8 (SB)



if (D[15] == 0) then PC = PC + sign\_ext(disp8) \* 2;

### JZ D[b], disp4 (SBR)



if (D[b] == 0) then PC = PC + zero\_ext(disp4) \* 2;

### Status Flags

C	Not set by this instruction.
V	Not set by this instruction.
SV	Not set by this instruction.
AV	Not set by this instruction.
SAV	Not set by this instruction.

### Examples

```
jz d2, foobar
jz d15, foobar
```

### See Also

[JGEZ \(16-bit\)](#), [JGTZ \(16-bit\)](#), [JLEZ \(16-bit\)](#), [JLTZ \(16-bit\)](#), [JNZ \(16-bit\)](#)

## JZ.A

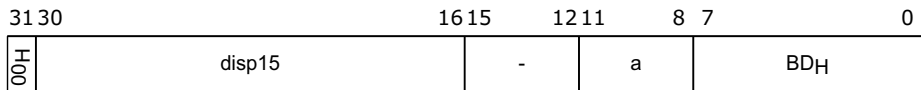
### Jump if Zero Address

#### Description

If the contents of A[a] are equal to zero then add the value specified by disp15, sign-extended and multiplied by 2, to the contents of PC and jump to that address.

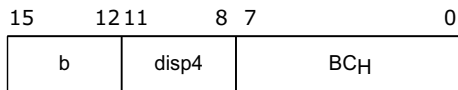
If the contents of A[b] are equal to zero then add the value specified by disp4, zero-extended and multiplied by 2, to the contents of PC and jump to that address.

#### JZ.A                      A[a], disp15 (BRR)



if (A[a] == 0) then PC = PC + sign\_ext(disp15) \* 2;

#### JZ.A                      A[b], disp4 (SBR)



if (A[b] == 0) then PC = PC + zero\_ext(disp4) \* 2;

#### Status Flags

C	Not set by this instruction.
V	Not set by this instruction.
SV	Not set by this instruction.
AV	Not set by this instruction.
SAV	Not set by this instruction.

#### Examples

```
jz.a    a4, foobar
```

```
jz.a    a2, foobar
```

#### See Also

[JNZ.A](#)

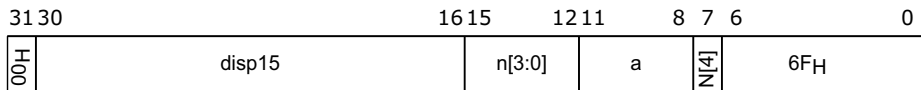
## JZ.T Jump if Zero Bit

### Description

If bit *n* of register *D[a]* is equal to zero then add the value specified by *disp15*, sign-extended and multiplied by 2, to the contents of PC and jump to that address.

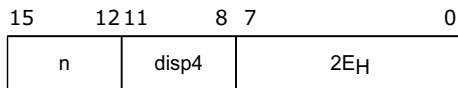
If bit *n* of register *D[15]* is equal to zero then add the value specified by *disp4*, zero-extended and multiplied by 2, to the contents of PC and jump to that address.

### JZ.T                      D[a], n, disp15 (BRN)



if (!D[a][n]) then PC = PC + sign\_ext(disp15) \* 2;

### JZ.T                      D[15], n, disp4 (SBRN)



if (!D[15][n]) then PC = PC + zero\_ext(disp4) \* 2;

### Status Flags

C	Not set by this instruction.
V	Not set by this instruction.
SV	Not set by this instruction.
AV	Not set by this instruction.
SAV	Not set by this instruction.

### Examples

```
jz.t    d1, 1, foobar
```

```
jz.t    d15, 1, foobar
```

### See Also

[JNZ.T](#)

## LD.A

### Load Word to Address Register

#### Description

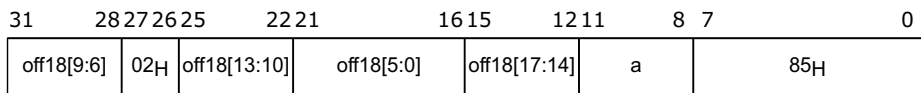
Load the word contents of the memory location specified by the addressing mode into address register A[a].

*Note: If the target register is modified by the addressing mode, the result is undefined.*

Load the word contents of the memory location specified by the addressing mode into either address register A[a] or A[15].

*Note: If the target register is modified by the addressing mode, the result is undefined.*

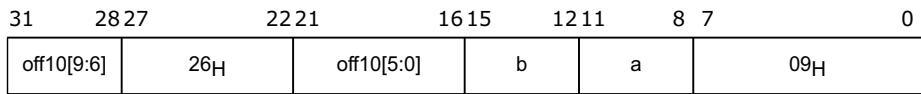
#### LD.A                      A[a], off18 (ABS)                      (Absolute Addressing Mode)



EA = {off18[17:14], 14b'0, off18[13:0]};

A[a] = M(EA, word);

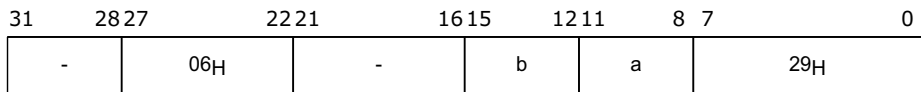
#### LD.A                      A[a], A[b], off10 (BO) (Base + Short Offset Addressing Mode)



EA = A[b] + sign\_ext(off10);

A[a] = M(EA, word);

#### LD.A                      A[a], P[b] (BO)                      (Bit-reverse Addressing Mode)



index = zero\_ext(A[b+1][15:0]);

incr = zero\_ext(A[b+1][31:16]);

EA = A[b] + index;

A[a] = M(EA, word);

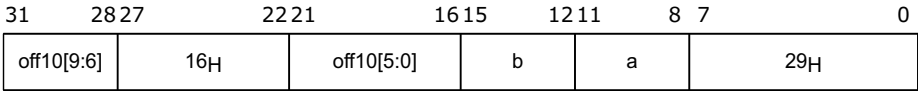
new\_index = reverse16(reverse16(index) + reverse16(incr));

A[b+1] = {incr[15:0], new\_index[15:0]};

See Also Addendum for TriCore Arch Manual, Vol2, V1.3.8, pages 493/494

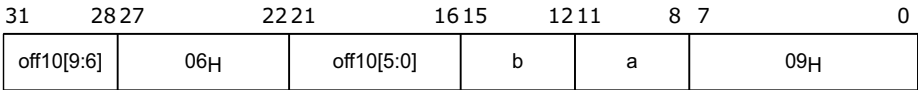
Instruction Set

**LD.A                      A[a], P[b], off10 (BO)                      (Circular Addressing Mode)**



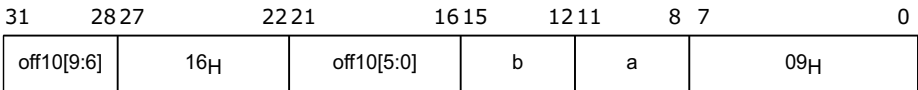
index = zero\_ext(A[b+1][15:0]);  
length = zero\_ext(A[b+1][31:16]);  
EA = A[b] + index;  
A[a] = M(EA, word);  
new\_index = index + sign\_ext(off10);  
new\_index = new\_index < 0 ? new\_index+length : new\_index % length;  
A[b+1] = {length[15:0], new\_index[15:0]};

**LD.A                      A[a], A[b], off10 (BO)                      (Post-increment Addressing Mode)**



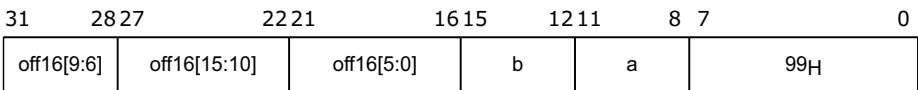
EA = A[b];  
A[a] = M(EA, word);  
A[b] = EA + sign\_ext(off10);

**LD.A                      A[a], A[b], off10 (BO)                      (Pre-increment Addressing Mode)**



EA = A[b] + sign\_ext(off10);  
A[a] = M(EA, word);  
A[b] = EA;

**LD.A                      A[a], A[b], off16 (BOL)(Base + Long Offset Addressing Mode)**

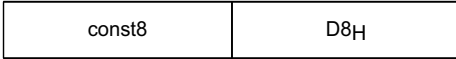


EA = A[b] + sign\_ext(off16);  
A[a] = M(EA, word);

See Also Addendum for TriCore Arch Manual, Vol2, V1.3.8, pages 493/494

**LD.A                    A[15], A[10], const8 (SC)**

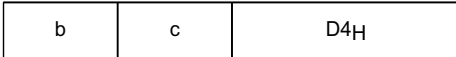
15                    8 7                    0



$A[15] = M(A[10] + \text{zero\_ext}(4 * \text{const8}), \text{word});$

**LD.A                    A[c], A[b] (SLR)**

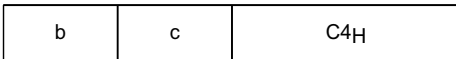
15    12 11            8 7                    0



$A[c] = M(A[b], \text{word});$

**LD.A                    A[c], A[b] (SLR)                    (Post-increment Addressing Mode)**

15    12 11            8 7                    0

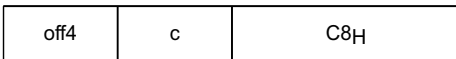


$A[c] = M(A[b], \text{word});$

$A[b] = A[b] + 4;$

**LD.A                    A[c], A[15], off4 (SLRO)**

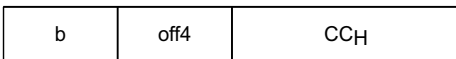
15    12 11            8 7                    0



$A[c] = M(A[15] + \text{zero\_ext}(4 * \text{off4}), \text{word});$

**LD.A                    A[15], A[b], off4 (SRO)**

15    12 11            8 7                    0



$A[15] = M(A[b] + \text{zero\_ext}(4 * \text{off4}), \text{word});$

**Status Flags**

C	Not set by this instruction.
V	Not set by this instruction.
SV	Not set by this instruction.
AV	Not set by this instruction.

---

SAV	Not set by this instruction.
-----	------------------------------

---

**Examples**

```
ld.a    a0, [a0]
ld.a    a5, [a0+]4
```

**See Also**

[LD.B](#), [LD.BU](#), [LD.D](#), [LD.DA](#), [LD.H](#), [LD.Q](#), [LD.W](#)

See Also Addendum for TriCore Arch Manual, Vol2, V1.3.8, pages 493/494

## LD.B

Load Byte

## LD.BU

Load Byte Unsigned

### Description

Load the byte contents of the memory location specified by the addressing mode, sign-extended or zero-extended, into data register D[a].

Load the byte contents of the memory location specified by the addressing mode, zero-extended, into either data register D[a] or D[15].

### LD.B                      D[a], off18 (ABS)                      (Absolute Addressing Mode)

31	28 27 26 25	22 21	16 15	12 11	8 7	0
off18[9:6]	00 <sub>H</sub>	off18[13:10]	off18[5:0]	off18[17:14]	a	05 <sub>H</sub>

EA = {off18[17:14], 14b'0, off18[13:0]};

D[a] = sign\_ext(M(EA, byte));

### LD.B                      D[a], A[b], off10 (BO) (Base + Short Offset Addressing Mode)

31	28 27	22 21	16 15	12 11	8 7	0
off10[9:6]	20 <sub>H</sub>	off10[5:0]	b	a	09 <sub>H</sub>	

EA = A[b] + sign\_ext(off10);

D[a] = sign\_ext(M(EA, byte));

### LD.B                      D[a], P[b] (BO)                      (Bit-reverse Addressing Mode)

31	28 27	22 21	16 15	12 11	8 7	0
-	00 <sub>H</sub>	-	b	a	29 <sub>H</sub>	

index = zero\_ext(A[b+1][15:0]);

incr = zero\_ext(A[b+1][31:16]);

EA = A[b] + index;

D[a] = sign\_ext(M(EA, byte));

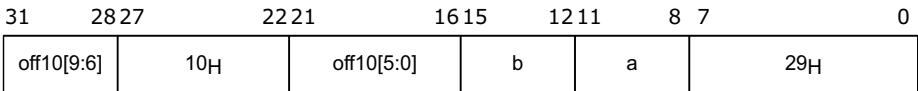
new\_index = reverse16(reverse16(index) + reverse16(incr));

A[b+1] = {incr[15:0], new\_index[15:0]};



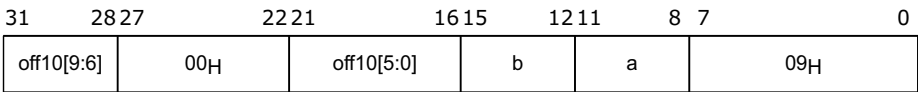
Instruction Set

**LD.B                                    D[a], P[b], off10 (BO)                                    (Circular Addressing Mode)**



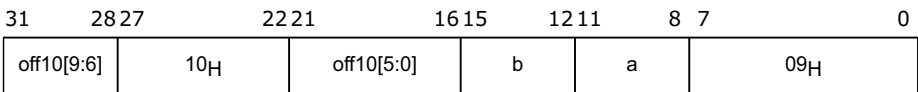
index = zero\_ext(A[b+1][15:0]);  
length = zero\_ext(A[b+1][31:16]);  
EA = A[b] + index;  
D[a] = sign\_ext(M(EA, byte));  
new\_index = index + sign\_ext(off10);  
new\_index = new\_index < 0 ? new\_index+length : new\_index % length;  
A[b+1] = {length[15:0], new\_index[15:0]};

**LD.B                                    D[a], A[b], off10 (BO)                                    (Post-increment Addressing Mode)**



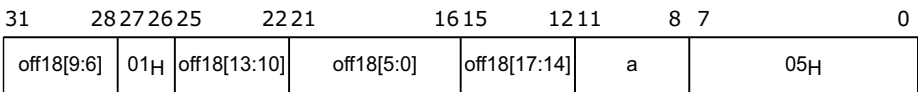
EA = A[b];  
D[a] = sign\_ext(M(EA, byte));  
A[b] = EA + sign\_ext(off10);

**LD.B                                    D[a], A[b], off10 (BO)                                    (Pre-increment Addressing Mode)**



EA = A[b] + sign\_ext(off10);  
D[a] = sign\_ext(M(EA, byte));  
A[b] = EA;

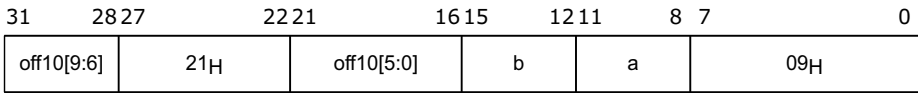
**LD.BU                                    D[a], off18 (ABS)                                    (Absolute Addressing Mode)**



EA = {off18[17:14], 14b'0, off18[13:0]};  
D[a] = zero\_ext(M(EA, byte));

See Also Addendum for TriCore Arch Manual, Vol2, V1.3.8, pages 493/494

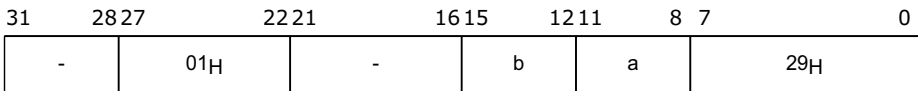
**LD.BU D[a], A[b], off10 (BO) (Base + Short Offset Addressing Mode)**



EA = A[b] + sign\_ext(off10);

D[a] = zero\_ext(M(EA, byte));

**LD.BU D[a], P[b] (BO) (Bit-reverse Addressing Mode)**



index = zero\_ext(A[b+1][15:0]);

incr = zero\_ext(A[b+1][31:16]);

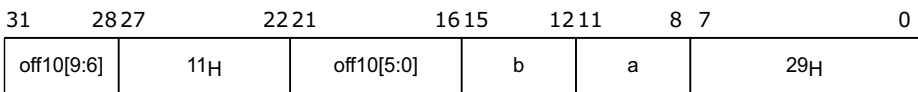
EA = A[b] + index;

D[a] = zero\_ext(M(EA, byte));

new\_index = reverse16(reverse16(index) + reverse16(incr));

A[b+1] = {incr[15:0], new\_index[15:0]};

**LD.BU D[a], P[b], off10 (BO) (Circular Addressing Mode)**



index = zero\_ext(A[b+1][15:0]);

length = zero\_ext(A[b+1][31:16]);

EA = A[b] + index;

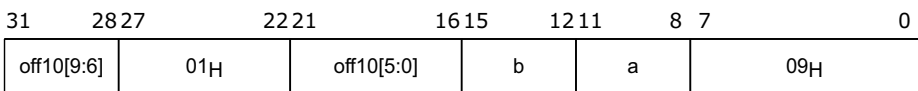
D[a] = zero\_ext(M(EA, byte));

new\_index = index + sign\_ext(off10);

new\_index = new\_index < 0 ? new\_index+length : new\_index % length;

A[b+1] = {length[15:0], new\_index[15:0]};

**LD.BU D[a], A[b], off10 (BO) (Post-increment Addressing Mode)**

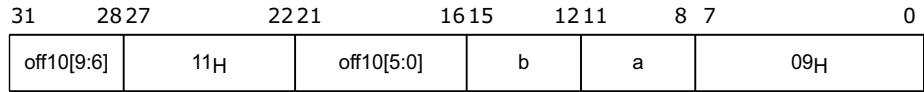


EA = A[b];

D[a] = zero\_ext(M(EA, byte));

$A[b] = EA + \text{sign\_ext}(\text{off10});$

**LD.BU**                      **D[a], A[b], off10 (BO)**                      **(Pre-increment Addressing Mode)**

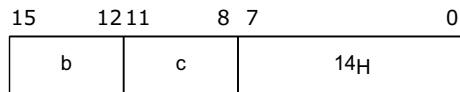


$EA = A[b] + \text{sign\_ext}(\text{off10});$

$D[a] = \text{zero\_ext}(M(EA, \text{byte}));$

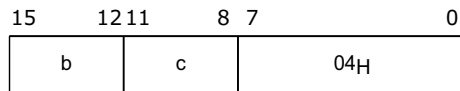
$A[b] = EA;$

**LD.BU**                      **D[c], A[b] (SLR)**



$D[c] = \text{zero\_ext}(M(A[b], \text{byte}));$

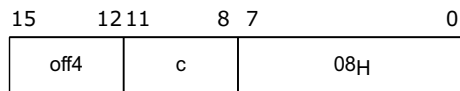
**LD.BU**                      **D[c], A[b] (SLR)**                      **(Post-increment Addressing Mode)**



$D[c] = \text{zero\_ext}(M(A[b], \text{byte}));$

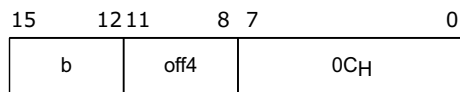
$A[b] = A[b] + 1;$

**LD.BU**                      **D[c], A[15], off4 (SLRO)**



$D[c] = \text{zero\_ext}(M(A[15] + \text{zero\_ext}(\text{off4}), \text{byte}));$

**LD.BU**                      **D[15], A[b], off4 (SRO)**



$D[15] = \text{zero\_ext}(M(A[b] + \text{zero\_ext}(\text{off4}), \text{byte}));$

**Status Flags**

C	Not set by these instructions.
---	--------------------------------

See Also Addendum for TriCore Arch Manual, Vol2, V1.3.8, pages 493/494

V	Not set by these instructions.
SV	Not set by these instructions.
AV	Not set by these instructions.
SAV	Not set by these instructions.

### Examples

```
ld.b    d0, [a0]
ld.bu   d5, [a0+]4
```

### See Also

[LD.A](#), [LD.D](#), [LD.DA](#), [LD.H](#), [LD.Q](#), [LD.W](#)

## LD.D

### Load Double-word

#### Description

Load the double-word contents of the memory location specified by the addressing mode into the extended data register E[a]. The least-significant word of the double-word value is loaded into the even register (D[n]) and the most-significant word is loaded into the odd register (D[n+1]).

#### LD.D E[a], off18 (ABS) (Absolute Addressing Mode)

31	28	27	26	25	22	21	16	15	12	11	8	7	0
off18[9:6]			01 <sub>H</sub>		off18[13:10]			off18[5:0]		a		85 <sub>H</sub>	

EA = {off18[17:14], 14b'0, off18[13:0]};

E[a] = M(EA, doubleword);

#### LD.D E[a], A[b], off10 (BO) (Base + Short Offset Addressing Mode)

31	28	27	22	21	16	15	12	11	8	7	0	
off10[9:6]			25 <sub>H</sub>		off10[5:0]			b		a		09 <sub>H</sub>

EA = A[b] + sign\_ext(off10);

E[a] = M(EA, doubleword);

#### LD.D E[a], P[b] (BO) (Bit-reverse Addressing Mode)

31	28	27	22	21	16	15	12	11	8	7	0	
-			05 <sub>H</sub>		-			b		a		29 <sub>H</sub>

index = zero\_ext(A[b+1][15:0]);

incr = zero\_ext(A[b+1][31:16]);

EA = A[b] + index;

D[a] = zero\_ext(M(EA, doubleword));

new\_index = reverse16(reverse16(index) + reverse16(incr));

A[b+1] = {incr[15:0], new\_index[15:0]};

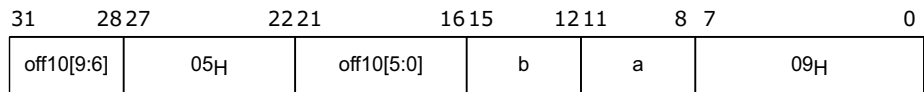
#### LD.D E[a], P[b], off10 (BO) (Circular Addressing Mode)

31	28	27	22	21	16	15	12	11	8	7	0	
off10[9:6]			15 <sub>H</sub>		off10[5:0]			b		a		29 <sub>H</sub>

```

index = zero_ext(A[b+1][15:0]);
length = zero_ext(A[b+1][31:16]);
EA0 = A[b] + index;
EA2 = A[b] + (index + 2) % length;
EA4 = A[b] + (index + 4) % length;
EA6 = A[b] + (index + 6) % length;
EA = {M(EA6, halfword), M(EA4, halfword), M(EA2, halfword), M(EA0, halfword)};
new_index = index + sign_ext(off10);
new_index = new_index < 0 ? new_index+length : new_index % length;
A[b+1] = {length[15:0], new_index[15:0]};
    
```

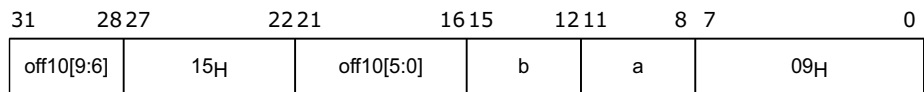
**LD.D E[a], A[b], off10 (BO) (Post-increment Addressing Mode)**



```

EA = A[b];
E[a] = M(EA, doubleword);
A[b] = EA + sign_ext(off10);
    
```

**LD.D E[a], A[b], off10 (BO) (Pre-increment Addressing Mode)**



```

EA = A[b] + sign_ext(off10);
E[a] = M(EA, doubleword);
A[b] = EA;
    
```

**Status Flags**

C	Not set by this instruction.
V	Not set by this instruction.
SV	Not set by this instruction.
AV	Not set by this instruction.
SAV	Not set by this instruction.

See Also Addendum for TriCore Arch Manual, Vol2, V1.3.8, pages 493/494

### Examples

```
ld.d    e0, [a0]
ld.d    d0/d1, [a0]
ld.d    e4, [a10+]4
```

### See Also

[LD.A](#), [LD.B](#), [LD.BU](#), [LD.DA](#), [LD.H](#), [LD.Q](#), [LD.W](#)

See Also Addendum for TriCore Arch Manual, Vol2, V1.3.8, pages 493/494

## LD.DA Load Double-word to Address Register

### Description

Load the double-word contents of the memory location specified by the addressing mode into an address register pair A[a]. The least-significant word of the double-word value is loaded into the even register (A[a]) and the most-significant word is loaded into the odd register (A[a+1]).

*Note: If the target register is modified by the addressing mode, the result is undefined.*

### LD.DA P[a], off18 (ABS) (Absolute Addressing Mode)

31	28 27 26 25	22 21	16 15	12 11	8 7	0
off18[9:6]	03 <sub>H</sub>	off18[13:10]	off18[5:0]	off18[17:14]	a	85 <sub>H</sub>

EA = {off18[17:14], 14b'0, off18[13:0]};

P[a] = M(EA, doubleword);

### LD.DA P[a], A[b], off10 (BO) (Base + Short Offset Addressing Mode)

31	28 27	22 21	16 15	12 11	8 7	0
off10[9:6]	27 <sub>H</sub>	off10[5:0]	b	a	09 <sub>H</sub>	

EA = A[b] + sign\_ext(off10);

P[a] = M(EA, doubleword);

### LD.DA P[a], P[b] (BO) (Bit-reverse Addressing Mode)

31	28 27	22 21	16 15	12 11	8 7	0
-	07 <sub>H</sub>	-	b	a	29 <sub>H</sub>	

index = zero\_ext(A[b+1][15:0]);

incr = zero\_ext(A[b+1][31:16]);

EA = A[b] + index;

P[a] = M(EA, doubleword);

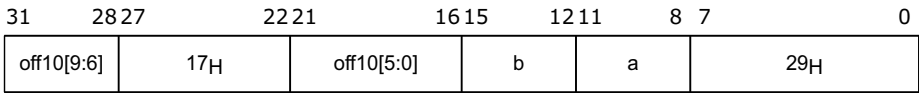
new\_index = reverse16(reverse16(index) + reverse16(incr));

A[b+1] = {incr[15:0], new\_index[15:0]};



Instruction Set

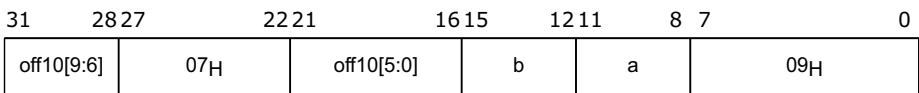
**LD.DA**                      **P[a], P[b], off10 (BO)**                      **(Circular Addressing Mode)**



```

index = zero_ext(A[b+1][15:0]);
length = zero_ext(A[b+1][31:16]);
EA0 = A[b] + index;
EA4 = A[b] + (index + 4) % length;
P[a] = {M(EA4, word), M(EA0, word)};
new_index = index + sign_ext(off10);
new_index = new_index < 0 ? new_index + length : new_index % length;
A[b+1] = {length[15:0], new_index[15:0]};
    
```

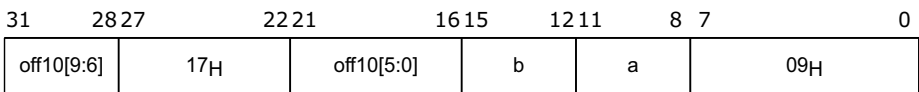
**LD.DA**                      **P[a], A[b], off10 (BO)**                      **(Post-increment Addressing Mode)**



```

EA = A[b];
P[a] = M(EA, doubleword);
A[b] = EA + sign_ext(off10);
    
```

**LD.DA**                      **P[a], A[b], off10 (BO)**                      **(Pre-increment Addressing Mode)**



```

EA = A[b] + sign_ext(off10);
P[a] = M(EA, doubleword);
A[b] = EA;
    
```

**Status Flags**

C	Not set by this instruction.
V	Not set by this instruction.
SV	Not set by this instruction.
AV	Not set by this instruction.
SAV	Not set by this instruction.

**Examples**

```
ld.da    a4/a5, [a6]+8
ld.da    a0/a1, _savedPointerBuffer
```

**See Also**

[LD.A](#), [LD.B](#), [LD.BU](#), [LD.D](#), [LD.H](#), [LD.Q](#), [LD.W](#)

See Also Addendum for TriCore Arch Manual, Vol2, V1.3.8, pages 493/494

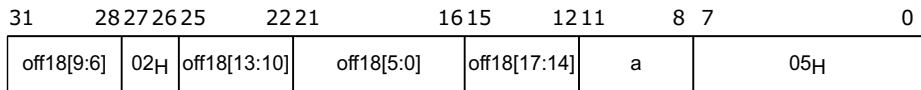
## LD.H Load Half-word

### Description

Load the half-word contents of the memory location specified by the addressing mode, sign-extended, into data register D[a].

Load the half-word contents of the memory location specified by the addressing mode, sign-extended, into either data register D[a] or D[15].

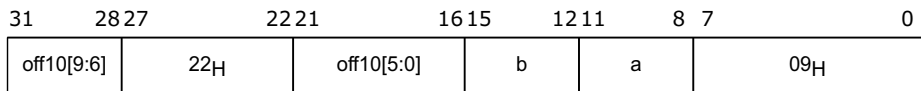
### LD.H                      D[a], off18 (ABS)                      (Absolute Addressing Mode)



EA = {off18[17:14], 14b'0, off18[13:0]};

D[a] = sign\_ext(M(EA, halfword));

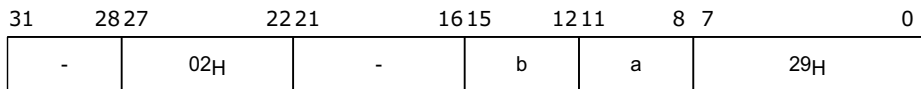
### LD.H                      D[a], A[b], off10 (BO) (Base + Short Offset Addressing Mode)



EA = A[b] + sign\_ext(off10);

D[a] = sign\_ext(M(EA, halfword));

### LD.H                      D[a], P[b] (BO)                      (Bit-reverse Addressing Mode)



index = zero\_ext(A[b+1][15:0]);

incr = zero\_ext(A[b+1][31:16]);

EA = A[b] + index;

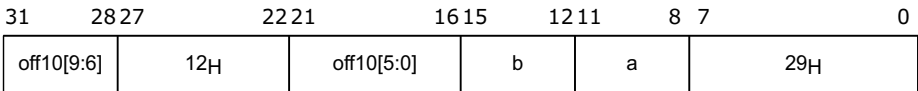
D[a] = sign\_ext(M(EA, halfword));

new\_index = reverse16(reverse16(index) + reverse16(incr));

A[b+1] = {incr[15:0], new\_index[15:0]};

Instruction Set

**LD.H D[a], P[b], off10 (BO) (Circular Addressing Mode)**

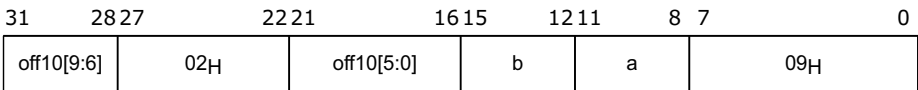


```

index = zero_ext(A[b+1][15:0]);
length = zero_ext(A[b+1][31:16]);
EA = A[b] + index;
D[a] = sign_ext(M(EA, halfword));
new_index = index + sign_ext(off10);
new_index = new_index < 0 ? new_index + length : new_index % length;
A[b+1] = {length[15:0], new_index[15:0]};

```

**LD.H D[a], A[b], off10 (BO) (Post-increment Addressing Mode)**

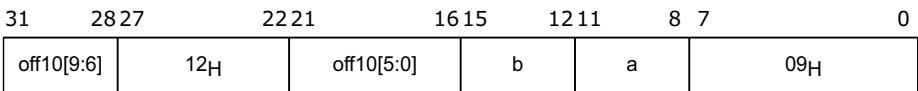


```

EA = A[b];
D[a] = sign_ext(M(EA, halfword));
A[b] = EA + sign_ext(off10);

```

**LD.H D[a], A[b], off10 (BO) (Pre-increment Addressing Mode)**

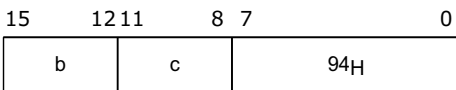


```

EA = A[b] + sign_ext(off10);
D[a] = sign_ext(M(EA, halfword));
A[b] = EA;

```

**LD.H D[c], A[b] (SLR)**



D[c] = sign\_ext(M(A[b], halfword));

Instruction Set

**LD.H D[c], A[b] (SLR) (Post-increment Addressing Mode)**

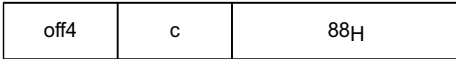
15 12 11 8 7 0



$D[c] = \text{sign\_ext}(M(A[b], \text{half-word}))$ ;  
 $A[b] = A[b] + 2$ ;

**LD.H D[c], A[15], off4 (SLRO)**

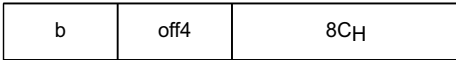
15 12 11 8 7 0



$D[c] = \text{sign\_ext}(M(A[15] + \text{zero\_ext}(2 * \text{off4}), \text{half-word}))$ ;

**LD.H D[15], A[b], off4 (SRO)**

15 12 11 8 7 0



$D[15] = \text{sign\_ext}(M(A[b] + \text{zero\_ext}(2 * \text{off4}), \text{half-word}))$ ;

**Status Flags**

C	Not set by this instruction.
V	Not set by this instruction.
SV	Not set by this instruction.
AV	Not set by this instruction.
SAV	Not set by this instruction.

**Examples**

```
ld.h    d0, [a0]
ld.hu   d1, [a0]
```

**See Also**

[LD.A](#), [LD.B](#), [LD.BU](#), [LD.D](#), [LD.DA](#), [LD.Q](#), [LD.W](#)

## LD.HU Load Half-word Unsigned

### Description

Load the half-word contents of the memory location specified by the addressing mode, zero-extended, into data register D[a].

### LD.HU                      D[a], off18 (ABS)                      (Absolute Addressing Mode)

31	28 27 26 25	22 21	16 15	12 11	8 7	0
off18[9:6]	03 <sub>H</sub>	off18[13:10]	off18[5:0]	off18[17:14]	a	05 <sub>H</sub>

EA = {off18[17:14], 14b'0, off18[13:0]};

D[a] = zero\_ext(M(EA, halfword));

### LD.HU                      D[a], A[b], off10 (BO) (Base + Short Offset Addressing Mode)

31	28 27	22 21	16 15	12 11	8 7	0
off10[9:6]	23 <sub>H</sub>	off10[5:0]	b	a	09 <sub>H</sub>	

EA = A[b] + sign\_ext(off10);

D[a] = zero\_ext(M(EA, halfword));

### LD.HU                      D[a], P[b] (BO)                      (Bit-reverse Addressing Mode)

31	28 27	22 21	16 15	12 11	8 7	0
-	03 <sub>H</sub>	-	b	a	29 <sub>H</sub>	

index = zero\_ext(A[b+1][15:0]);

incr = zero\_ext(A[b+1][31:16]);

EA = A[b] + index;

D[a] = zero\_ext(M(EA, halfword));

new\_index = reverse16(reverse16(index) + reverse16(incr));

A[b+1] = {incr[15:0], new\_index[15:0]};

### LD.HU                      D[a], P[b], off10 (BO)                      (Circular Addressing Mode)

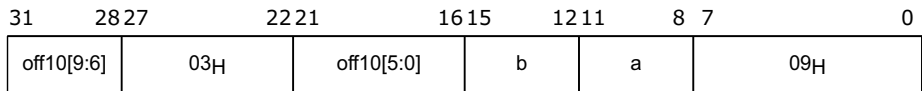
31	28 27	22 21	16 15	12 11	8 7	0
off10[9:6]	13 <sub>H</sub>	off10[5:0]	b	a	29 <sub>H</sub>	

index = zero\_ext(A[b+1][15:0]);

length = zero\_ext(A[b+1][31:16]);

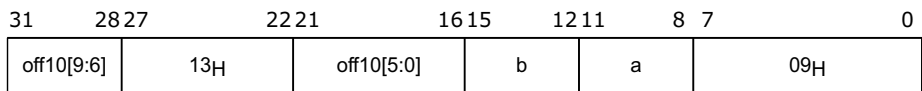
EA0 = A[b] + index;  
D[a] = zero\_ext(EA, halfword);  
new\_index = index + sign\_ext(off10);  
new\_index = new\_index < 0 ? new\_index + length : new\_index % length;  
A[b+1] = {length[15:0], new\_index[15:0]};

**LD.HU                      D[a], A[b], off10 (BO)                      (Post-increment Addressing Mode)**



EA = A[b];  
D[a] = zero\_ext(M(EA, halfword));  
A[b] = EA + sign\_ext(off10);

**LD.HU                      D[a], A[b], off10 (BO)                      (Pre-increment Addressing Mode)**



EA = A[b] + sign\_ext(off10);  
D[a] = zero\_ext(M(EA, halfword));  
A[b] = EA;

**Status Flags**

C	Not set by this instruction.
V	Not set by this instruction.
SV	Not set by this instruction.
AV	Not set by this instruction.
SAV	Not set by this instruction.

**Examples**

```
ld.h    d0, [a0]
ld.hu   d1, [a0]
```

**See Also**

[LD.A](#), [LD.B](#), [LD.BU](#), [LD.D](#), [LD.DA](#), [LD.Q](#), [LD.W](#)

See Also Addendum for TriCore Arch Manual, Vol2, V1.3.8, pages 493/494

## LD.Q Load Half-word Signed Fraction

### Description

Load the half-word contents of the memory location specified by the addressing mode into the most-significant half-word of data register D[a], setting the 16 least-significant bits of D[a] to zero.

### LD.Q                      D[a], off18 (ABS)                      (Absolute Addressing Mode)

31	28 27 26 25	22 21	16 15	12 11	8 7	0
off18[9:6]	00 <sub>H</sub>	off18[13:10]	off18[5:0]	off18[17:14]	a	45 <sub>H</sub>

EA = {off18[17:14], 14b'0, off18[13:0]};

D[a] = {M(EA, halfword), 16'h0000};

### LD.Q                      D[a], A[b], off10 (BO) (Base + Short Offset Addressing Mode)

31	28 27	22 21	16 15	12 11	8 7	0
off10[9:6]	28 <sub>H</sub>	off10[5:0]	b	a	09 <sub>H</sub>	

EA = A[b] + sign\_ext(off10);

D[a] = {M(EA, halfword), 16'h0000};

### LD.Q                      D[a], P[b] (BO)                      (Bit-reverse Addressing Mode)

31	28 27	22 21	16 15	12 11	8 7	0
-	08 <sub>H</sub>	-	b	a	29 <sub>H</sub>	

index = zero\_ext(A[b+1][15:0]);

incr = zero\_ext(A[b+1][31:16]);

EA = A[b] + index;

D[a] = {M(EA, halfword), 16'h0000};

new\_index = reverse16(reverse16(index) + reverse16(incr));

A[b+1] = {incr[15:0], new\_index[15:0]};

### LD.Q                      D[a], P[b], off10 (BO)                      (Circular Addressing Mode)

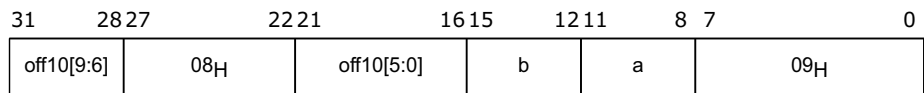
31	28 27	22 21	16 15	12 11	8 7	0
off10[9:6]	18 <sub>H</sub>	off10[5:0]	b	a	29 <sub>H</sub>	

index = zero\_ext(A[b+1][15:0]);



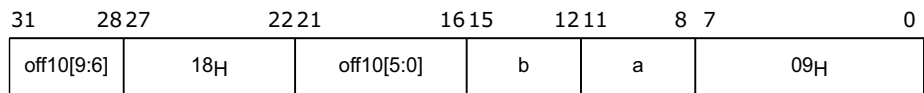
length = zero\_ext(A[b+1][31:16]);  
 EA0 = A[b] + index;  
 D[a] = {M(EA, halfword), 16'h0000};  
 new\_index = index + sign\_ext(off10);  
 new\_index = new\_index < 0 ? new\_index + length : new\_index % length;  
 A[b+1] = {length[15:0], new\_index[15:0]};

**LD.Q D[a], A[b], off10 (BO) (Post-increment Addressing Mode)**



EA = A[b];  
 D[a] = {M(EA, halfword), 16'h0000};  
 A[b] = EA + sign\_ext(off10);

**LD.Q D[a], A[b], off10 (BO) (Pre-increment Addressing Mode)**



EA = A[b] + sign\_ext(off10);  
 D[a] = {M(EA, halfword), 16'h0000};  
 A[b] = EA;

**Status Flags**

C	Not set by this instruction.
V	Not set by this instruction.
SV	Not set by this instruction.
AV	Not set by this instruction.
SAV	Not set by this instruction.

**Examples**

```
ld.q    d4, [a0+]2
ld.q    d2, [a2+]22
```

**See Also**

[LD.A](#), [LD.D](#), [LD.DA](#), [LD.B](#), [LD.BU](#), [LD.H](#), [LD.W](#)

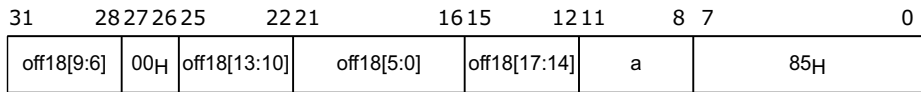
## LD.W Load Word

### Description

Load word contents of the memory location specified by the addressing mode into data register D[a].

Load word contents of the memory location specified by the addressing mode into data register either D[a] or D[15].

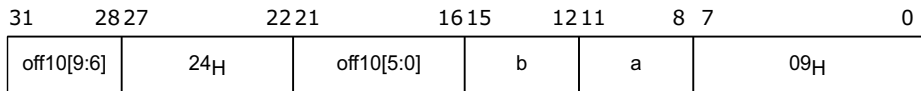
### LD.W                      D[a], off18 (ABS)                      (Absolute Addressing Mode)



EA = {off18[17:14], 14b'0, off18[13:0]};

D[a] = M(EA, word);

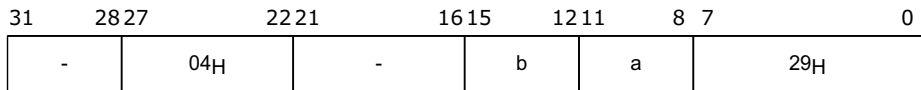
### LD.W                      D[a], A[b], off10 (BO) (Base + Short Offset Addressing Mode)



EA = A[b] + sign\_ext(off10);

D[a] = M(EA, word);

### LD.W                      D[a], P[b] (BO)                      (Bit-reverse Addressing Mode)



index = zero\_ext(A[b+1][15:0]);

incr = zero\_ext(A[b+1][31:16]);

EA = A[b] + index;

D[a] = M(EA, word);

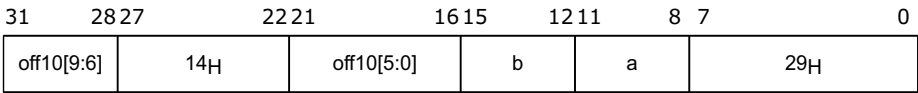
new\_index = reverse16(reverse16(index) + reverse16(incr));

A[b+1] = {incr[15:0], new\_index[15:0]};

See Also Addendum for TriCore Arch Manual, Vol2, V1.3.8, pages 493/494

Instruction Set

**LD.W                      D[a], P[b], off10 (BO)                      (Circular Addressing Mode)**

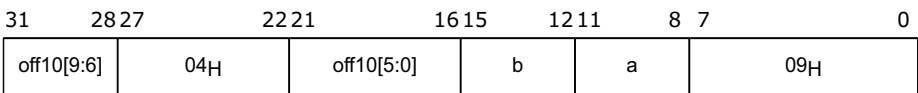


```

index = zero_ext(A[b+1][15:0]);
length = zero_ext(A[b+1][31:16]);
EA0 = A[b] + index;
EA2 = A[b] + (index + 2% length);
D[a] = {M(EA2, halfword), M(EA0, halfword)};
new_index = index + sign_ext(off10);
new_index = new_index < 0 ? new_index + length : new_index % length;
A[b+1] = {length[15:0], new_index[15:0]};

```

**LD.W                      D[a], A[b], off10 (BO)                      (Post-increment Addressing Mode)**

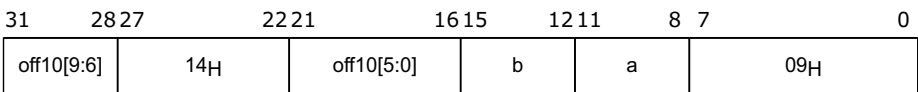


```

EA = A[b];
D[a] = M(EA, word);
A[b] = EA + sign_ext(off10);

```

**LD.W                      D[a], A[b], off10 (BO)                      (Pre-increment Addressing Mode)**

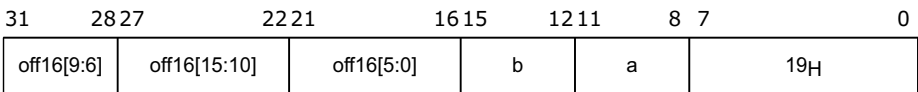


```

EA = A[b] + sign_ext(off10);
D[a] = M(EA, word);
A[b] = EA;

```

**LD.W                      D[a], A[b], off16 (BOL)(Base + Long Offset Addressing Mode)**



```

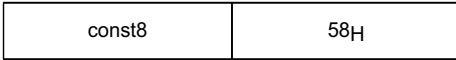
EA = A[b] + sign_ext(off16);
D[a] = M(EA, word);

```

See Also Addendum for TriCore Arch Manual, Vol2, V1.3.8, pages 493/494

**LD.W D[15], A[10], const8 (SC)**

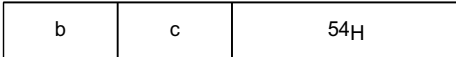
15 8 7 0



$D[15] = M(A[10] + \text{zero\_ext}(4 * \text{const8}), \text{word});$

**LD.W D[c], A[b] (SLR)**

15 12 11 8 7 0



$D[c] = M(A[b], \text{word});$

**LD.W D[c], A[b] (SLR) (Post-increment Addressing Mode)**

15 12 11 8 7 0

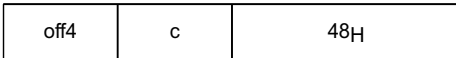


$D[c] = M(A[b], \text{word});$

$A[b] = A[b] + 4;$

**LD.W D[c], A[15], off4 (SLRO)**

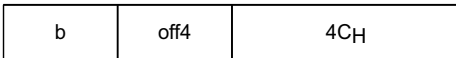
15 12 11 8 7 0



$D[c] = M(A[15] + \text{zero\_ext}(4 * \text{off4}), \text{word});$

**LD.W D[15], A[b], off4 (SRO)**

15 12 11 8 7 0



$D[15] = M(A[b] + \text{zero\_ext}(4 * \text{off4}), \text{word});$

**Status Flags**

C	Not set by this instruction.
V	Not set by this instruction.
SV	Not set by this instruction.
AV	Not set by this instruction.

---

SAV	Not set by this instruction.
-----	------------------------------

---

### Examples

```
ld.w    d4, [a0+]2  
ld.w    d2, [a2+]22
```

### See Also

[LD.A](#), [LD.D](#), [LD.DA](#), [LD.B](#), [LD.BU](#), [LD.H](#), [LD.Q](#), [LD.B](#), [LD.D](#)

## LDLCX

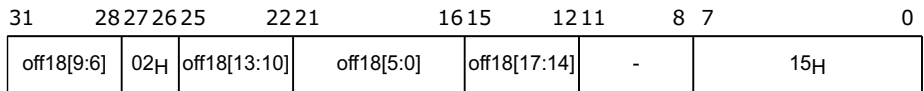
### Load Lower Context

#### Description

Load the contents of the memory block specified by the addressing mode into registers A[2]-A[7] and D[0]-D[7]. This operation is normally used to restore GPR values that were saved previously by an STLCX instruction.

*Note: 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).*

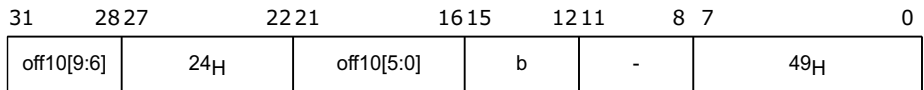
#### LDLCX                      off18 (ABS)                      (Absolute Addressing Mode)



EA = {off18[17:14], 14b'0, off18[13:0]};

{dummy, dummy, A[2:3], D[0:3], A[4:7], D[4:7]} = M(EA, 16-word);

#### LDLCX                      A[b], off10 (BO)                      (Base + Short Index Addressing Mode)



EA = A[b] + sign\_ext(off10);

{dummy, dummy, A[2:3], D[0:3], A[4:7], D[4:7]} = M(EA, 16-word);

#### Status Flags

C	Not set by this instruction.
V	Not set by this instruction.
SV	Not set by this instruction.
AV	Not set by this instruction.
SAV	Not set by this instruction.

#### Examples

-

#### See Also

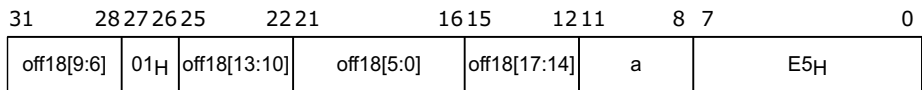
[LDUCX](#), [RSLCX](#), [STLCX](#), [STUCX](#), [SVLCX](#), [BISR](#)

## LDMST Load-Modify-Store

### Description

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  $E[a][31:0]$  where the corresponding bits in the mask  $E[a][63:32]$  are set, are stored into memory. The value and mask may be generated using the **IMASK** instruction.

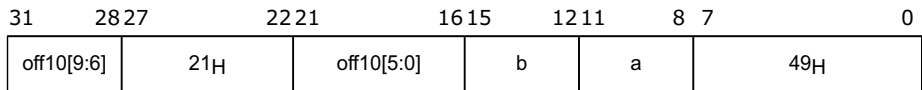
### LDMST                      off18, E[a] (ABS)                      (Absolute Addressing Mode)



$EA = \{off18[17:14], 14b'0, off18[13:0]\};$

$M(EA, word) = (M(EA, word) \& \sim E[a][63:32]) | (E[a][31:0] \& E[a][63:32]);$

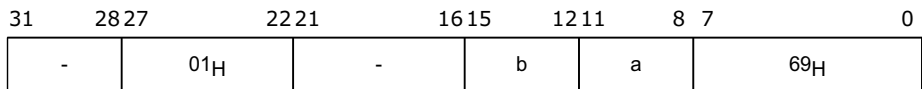
### LDMST                      A[b], off10, E[a] (BO) (Base + Short Offset Addressing Mode)



$EA = A[b] + sign\_ext(off10);$

$M(EA, word) = (M(EA, word) \& \sim E[a][63:32]) | (E[a][31:0] \& E[a][63:32]);$

### LDMST                      P[b], E[a] (BO)                      (Bit-reverse Addressing Mode)



$index = zero\_ext(A[b+1][15:0]);$

$incr = zero\_ext(A[b+1][31:16]);$

$EA = A[b] + index;$

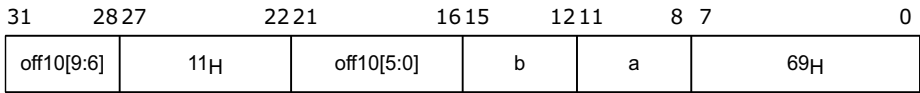
$M(EA, word) = (M(EA, word) \& \sim E[a][63:32]) | (E[a][31:0] \& E[a][63:32]);$

$new\_index = reverse16(reverse16(index) + reverse16(incr));$

$A[b+1] = \{incr[15:0], new\_index[15:0]\};$

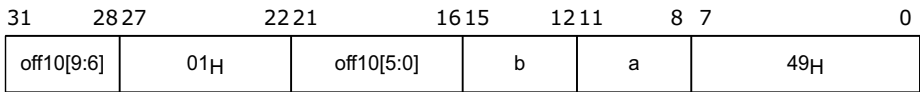
Instruction Set

**LDMST                      P[b], off10, E[a] (BO)                      (Circular Addressing Mode)**



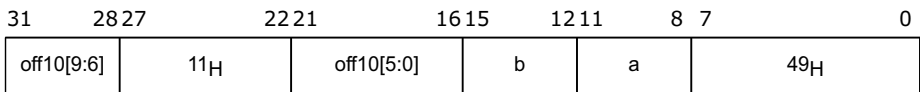
```
index = zero_ext(A[b+1][15:0]);
length = zero_ext(A[b+1][31:16]);
EA = A[b] + index;
M(EA, word) = (M(EA, word) & ~E[a][63:32]) | (E[a][31:0] & E[a][63:32]);
new_index = index + sign_ext(off10);
new_index = new_index < 0 ? new_index + length : new_index % length;
A[b+1] = {length[15:0], new_index[15:0]};
```

**LDMST                      A[b], off10, E[a] (BO)                      (Post-increment Addressing Mode)**



```
EA = A[b];
M(EA, word) = (M(EA, word) & ~E[a][63:32]) | (E[a][31:0] & E[a][63:32]);
A[b] = EA + sign_ext(off10);
```

**LDMST                      A[b], off10, E[a] (BO)                      (Pre-increment Addressing Mode)**



```
EA = A[b] + sign_ext(off10);
M(EA, word) = (M(EA, word) & ~E[a][63:32]) | (E[a][31:0] & E[a][63:32]);
A[b] = EA;
```

**Status Flags**

C	Not set by this instruction.
V	Not set by this instruction.
SV	Not set by this instruction.
AV	Not set by this instruction.
SAV	Not set by this instruction.



**Examples**

-

**See Also**

**IMASK, ST.T, SWAP.W**

See Also Addendum for TriCore Arch Manual, Vol2, V1.3.8, pages 493/494

## LDUCX

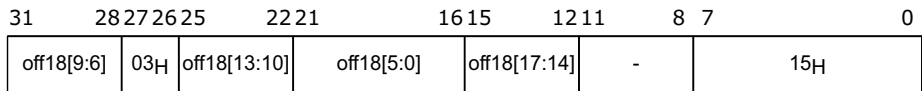
### Load Upper Context

#### Description

Load the contents of the memory block specified by the addressing mode into registers A[10] to A[15] and D[8] to D[15]. This operation is used normally to restore GPR values that were saved previously by a **STUCX** instruction.

*Note: The effective address (EA) 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).*

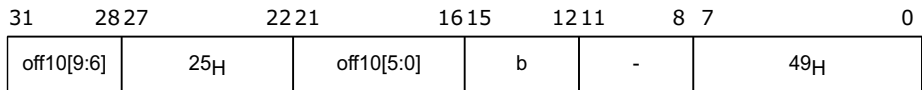
#### LDUCX                      off18 (ABS)                      (Absolute Addressing Mode)



EA = {off18[17:14], 14b'0, off18[13:0]};

{dummy, dummy, A[10:11], D[8:11], A[12:15], D[12:15]} = M(EA, 16-word);

#### LDUCX                      A[b], off10 (BO)                      (Base + Short Index Addressing Mode)



EA = A[b][31:0] + sign\_ext(off10);

{dummy, dummy, A[10:11], D[8:11], A[12:15], D[12:15]} = M(EA, 16-word);

#### Status Flags

C	Not set by this instruction.
V	Not set by this instruction.
SV	Not set by this instruction.
AV	Not set by this instruction.
SAV	Not set by this instruction.

#### Examples

-

#### See Also

**LDLCX, RSLCX, STLCX, STUCX, SVLCX, LDUCX**

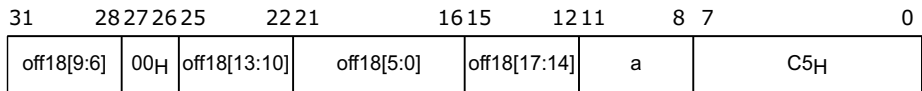
## LEA Load Effective Address

### Description

Compute the absolute (effective) address defined by the addressing mode and put the result in address register A[a].

*Note: The auto-increment addressing modes are not supported for this instruction.*

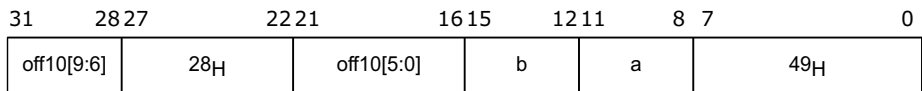
### LEA                      A[a], off18 (ABS)                      (Absolute Addressing Mode)



EA = {off18[17:14], 14b'0, off18[13:0]};

A[a] = EA[31:0];

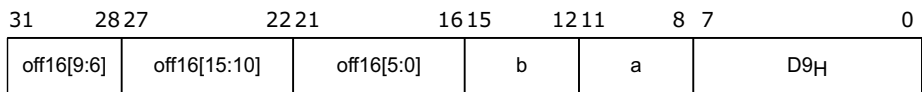
### LEA                      A[a], A[b], off10 (BO) (Base + Short Offset Addressing Mode)



EA = A[b] + sign\_ext(off10);

A[a] = EA[31:0];

### LEA                      A[a], A[b], off16 (BOL)(Base + Long Offset Addressing Mode)



EA = A[b] + sign\_ext(off16);

A[a] = EA[31:0];

### Status Flags

C	Not set by this instruction.
V	Not set by this instruction.
SV	Not set by this instruction.
AV	Not set by this instruction.
SAV	Not set by this instruction.

See Also Addendum for TriCore Arch Manual, Vol2, V1.3.8, pages 493/494

**Examples**

```
lea a0, _absadd  
lea a7, NumberOfLoops
```

**See Also**

[MOV.A](#), [MOV.D](#), [MOVH.A](#)

See Also Addendum for TriCore Arch Manual, Vol2, V1.3.8, pages 493/494

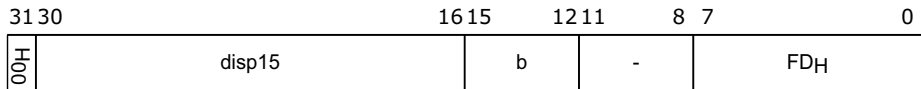
## LOOP Loop

### Description

If address register A[b] is not equal to zero, then add the value specified by disp15, multiplied by two and sign-extended, to the contents of PC and jump to that address. The address register is decremented unconditionally.

If address register A[b] is not equal to zero then add value specified by disp4, multiplied by two and one-extended to a 32-bit negative number, to the contents of PC and jump to that address. The address register is decremented unconditionally.

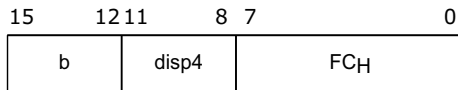
### LOOP                    A[b], disp15 (BRR)



if (A[b] != 0) then PC = PC + sign\_ext(2 \* disp15);

A[b] = A[b] - 1;

### LOOP                    A[b], disp4 (SBR)



if (A[b] != 0) then PC = PC + {27b'11111111111111111111111111111111, disp4, 0};  
A[b] = A[b] - 1;

### Status Flags

C	Not set by this instruction.
V	Not set by this instruction.
SV	Not set by this instruction.
AV	Not set by this instruction.
SAV	Not set by this instruction.

### Examples

```
loop    a4, iloop
```

```
loop    a4, iloop
```

See Also

[JNED](#), [JNEI](#), [LOOPU](#)

See Also Addendum for TriCore Arch Manual, Vol2, V1.3.8, pages 493/494

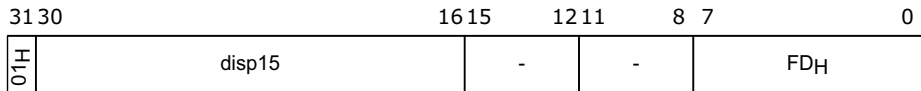
## LOOPU

### Loop Unconditional

#### Description

Add the value specified by disp15, multiplied by two and sign-extended, to the contents of PC and jump to that address.

#### LOOPU                      disp15 (BRR)



PC = PC + sign\_ext(2 \* disp15);

#### Status Flags

C	Not set by this instruction.
V	Not set by this instruction.
SV	Not set by this instruction.
AV	Not set by this instruction.
SAV	Not set by this instruction.

#### Examples

```
loopu    iloop
```

#### See Also

[J](#), [JA](#), [JI](#), [JL](#), [JLA](#), [JLI](#), [JNED](#), [JNEI](#), [LOOP](#)





**LT D[15], D[a], D[b] (SRR)**

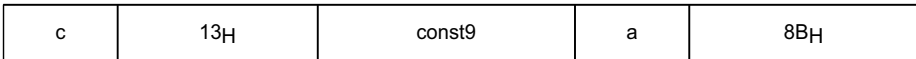
15 12 11 8 7 0



```
result = (D[a] < D[b]);
D[15] = zero_ext(result);
```

**LT.U D[c], D[a], const9 (RC)**

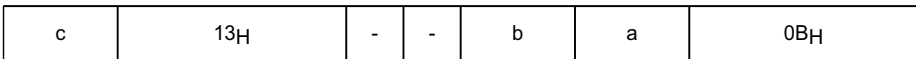
31 28 27 21 20 12 11 8 7 0



```
result = (D[a] < sign_ext(const9)); // unsigned
D[c] = zero_ext(result);
```

**LT.U D[c], D[a], D[b] (RR)**

31 28 27 20 19 18 17 16 15 12 11 8 7 0



```
result = (D[a] < D[b]); // unsigned
D[c] = zero_ext(result);
```

**Status Flags**

C	Not set by these instructions.
V	Not set by these instructions.
SV	Not set by these instructions.
AV	Not set by these instructions.
SAV	Not set by these instructions.

**Examples**

```
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
```

**See Also**

[EQ](#), [GE](#), [GE.U](#), [NE](#), [EQANY.B](#), [EQANY.H](#)

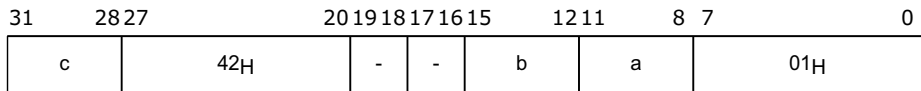
See Also Addendum for TriCore Arch Manual, Vol2, V1.3.8, pages 493/494

## LT.A Less Than Address

### Description

If the contents of address register A[a] are less than the contents of address register A[b], set the least-significant bit of D[c] to one and clear the remaining bits to zero; otherwise clear all bits in D[c]. The operands are treated as unsigned 32-bit integers.

### LT.A                      D[c], A[a], A[b] (RR)



D[c] = (A[a] < A[b]); // unsigned

### Status Flags

C	Not set by this instruction.
V	Not set by this instruction.
SV	Not set by this instruction.
AV	Not set by this instruction.
SAV	Not set by this instruction.

### Examples

```
lt.a    d3, a4, a2
```

### See Also

[EQ.A](#), [EQZ.A](#), [GE.A](#), [NE](#), [NEZ.A](#)

## LT.B

Less Than Packed Byte

## LT.BU

Less Than Packed Byte Unsigned

### Description

Compare each byte of data register D[a] with the corresponding byte of D[b]. In each case, if the value of the byte in D[a] is less than the value of the byte in D[b], set all bits in the corresponding byte of D[c] to one; otherwise clear all the bits. The operands are treated as signed (LT.B) or unsigned (LT.BU) 8-bit integers.

### LT.B D[c], D[a], D[b] (RR)

31	28 27	20 19 18 17 16 15	12 11	8 7	0	
c	52 <sub>H</sub>	-	-	b	a	0B <sub>H</sub>

$D[c][31:24] = (D[a][31:24] < D[b][31:24]) ? 8'hFF : 8'h00;$

$D[c][23:16] = (D[a][23:16] < D[b][23:16]) ? 8'hFF : 8'h00;$

$D[c][15:8] = (D[a][15:8] < D[b][15:8]) ? 8'hFF : 8'h00;$

$D[c][7:0] = (D[a][7:0] < D[b][7:0]) ? 8'hFF : 8'h00;$

### LT.BU D[c], D[a], D[b] (RR)

31	28 27	20 19 18 17 16 15	12 11	8 7	0	
c	53 <sub>H</sub>	-	-	b	a	0B <sub>H</sub>

$D[c][31:24] = (D[a][31:24] < D[b][31:24]) ? 8'hFF : 8'h00; // \text{unsigned}$

$D[c][23:16] = (D[a][23:16] < D[b][23:16]) ? 8'hFF : 8'h00; // \text{unsigned}$

$D[c][15:8] = (D[a][15:8] < D[b][15:8]) ? 8'hFF : 8'h00; // \text{unsigned}$

$D[c][7:0] = (D[a][7:0] < D[b][7:0]) ? 8'hFF : 8'h00; // \text{unsigned}$

### Status Flags

C	Not set by these instructions.
V	Not set by these instructions.
SV	Not set by these instructions.
AV	Not set by these instructions.
SAV	Not set by these instructions.

**Examples**

lt.b     d3, d1, d2  
lt.bu    d3, d1, d2

**See Also**

[EQ.B](#), [EQ.H](#), [EQ.W](#), [LT.H](#), [LT.HU](#), [LT.W](#), [LT.WU](#)

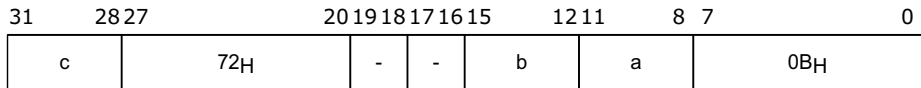
See Also Addendum for TriCore Arch Manual, Vol2, V1.3.8, pages 493/494

**LT.H**  
Less Than Packed Half-word  
**LT.HU**  
Less Than Packed Half-word Unsigned

**Description**

Compare each half-word of data register D[a] with the corresponding half-word of D[b]. In each case, if the value of the half-word in D[a] is less than the value of the corresponding half-word in D[b], set all bits of the corresponding half-word of D[c] to one; otherwise clear all the bits. Operands are treated as signed (LT.H) or unsigned (LT.HU) 16-bit integers.

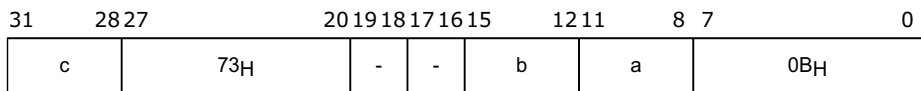
**LT.H**                      **D[c], D[a], D[b] (RR)**



$D[c][31:16] = (D[a][31:16] < D[b][31:16]) ? 16'hFFFF : 16'h0000;$

$D[c][15:0] = (D[a][15:0] < D[b][15:0]) ? 16'hFFFF : 16'h0000;$

**LT.HU**                      **D[c], D[a], D[b] (RR)**



$D[c][31:16] = (D[a][31:16] < D[b][31:16]) ? 16'hFFFF : 16'h0000; // \text{unsigned}$

$D[c][15:0] = (D[a][15:0] < D[b][15:0]) ? 16'hFFFF : 16'h0000; // \text{unsigned}$

**Status Flags**

C	Not set by these instructions.
V	Not set by these instructions.
SV	Not set by these instructions.
AV	Not set by these instructions.
SAV	Not set by these instructions.

**Examples**

lt.h      d3, d1, d2

lt.hu    d3, d1, d2

See Also Addendum for TriCore Arch Manual, Vol2, V1.3.8, pages 493/494

**See Also**

**[EQ.B](#), [EQ.H](#), [EQ.W](#), [LT.B](#), [LT.BU](#), [LT.W](#), [LT.WU](#)**

See Also Addendum for TriCore Arch Manual, Vol2, V1.3.8, pages 493/494

## LT.W

Less Than Packed Word

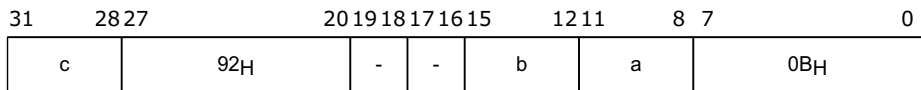
## LT.WU

Less Than Packed Word Unsigned

### Description

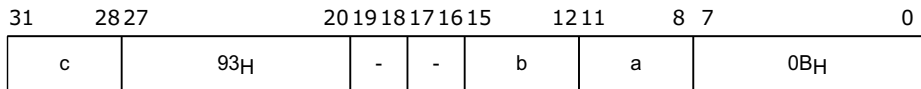
If the contents of data register D[a] are less than the contents of data register D[b], set all bits in D[c] to one; otherwise clear all bits in D[c]. D[a] and D[b] are treated as either signed (LT.W) or unsigned (LT.WU) 32-bit integers.

### LT.W D[c], D[a], D[b] (RR)



D[c] = (D[a] < D[b]) ? 32'hFFFFFFFF : 32'h00000000;

### LT.WU D[c], D[a], D[b] (RR)



D[c] = (D[a] < D[b]) ? 32'hFFFFFFFF : 32'h00000000; // unsigned

### Status Flags

C	Not set by these instructions.
V	Not set by these instructions.
SV	Not set by these instructions.
AV	Not set by these instructions.
SAV	Not set by these instructions.

### Examples

```
lt.w    d3, d1, d2
lt.wu   d3, d1, d2
```

### See Also

[EQ.B](#), [EQ.H](#), [EQ.W](#), [LT.B](#), [LT.BU](#), [LT.H](#), [LT.HU](#)



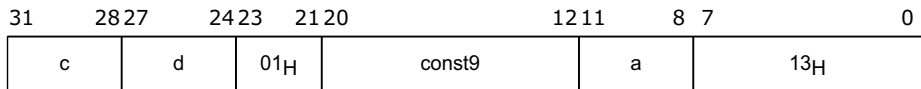
**MADD**  
Multiply-Add  
**MADDS**  
Multiply-Add, Saturated

**Description**

Multiply two 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. The value const9 is sign-extended before the multiplication is performed. The MADDS result is saturated on overflow.

**MADD**                      **D[c], D[d], D[a], const9 (RCR)**

32 + (32 \* K9)--> 32 signed

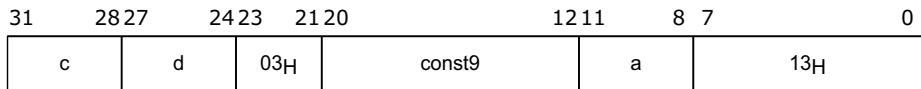


result = D[d] + (D[a] \* sign\_ext(const9));

D[c] = result[31:0];

**MADD**                      **E[c], E[d], D[a], const9 (RCR)**

64 + (32 \* K9)--> 64 signed

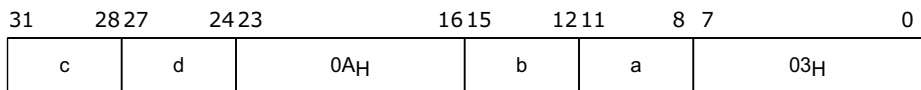


result = E[d] + (D[a] \* sign\_ext(const9));

E[c] = result[63:0];

**MADD**                      **D[c], D[d], D[a], D[b] (RRR2)**

32 + (32 \* 32)--> 32 signed



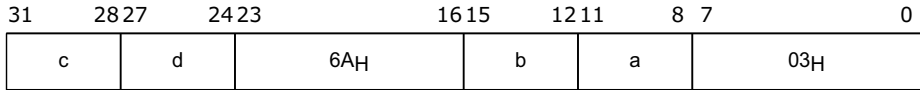
result = D[d] + (D[a] \* D[b]);

D[c] = result[31:0];

**MADD**                      **E[c], E[d], D[a], D[b] (RRR2)**

64 + (32 \* 32)--> 64 signed

See Also Addendum for TriCore Arch Manual, Vol2, V1.3.8, pages 493/494

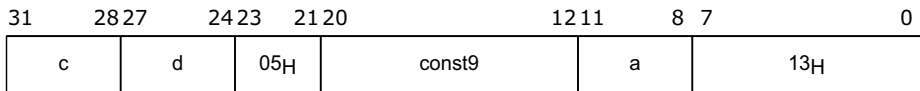


result = E[d] + (D[a] \* D[b]);

E[c] = result[63:0];

**MADDS**                      **D[c], D[d], D[a], const9 (RCR)**

32 + (32 \* K9)--> 32 signed saturated

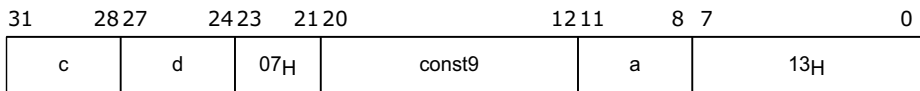


result = D[d] + (D[a] \* sign\_ext(const9));

D[c] = ssov(result, 32);

**MADDS**                      **E[c], E[d], D[a], const9 (RCR)**

64 + (32 \* K9)--> 64 signed saturated

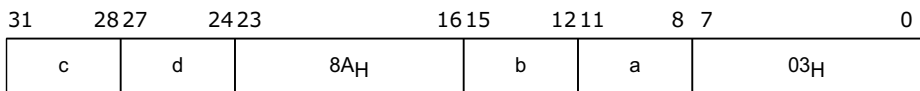


result = E[d] + (D[a] \* sign\_ext(const9));

E[c] = ssov(result, 64);

**MADDS**                      **D[c], D[d], D[a], D[b] (RRR2)**

32 + (32 \* 32)--> 32 signed saturated

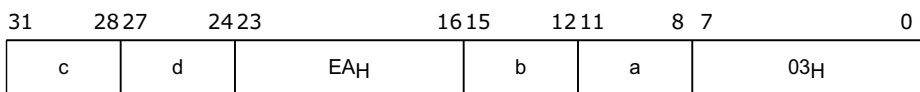


result = D[d] + (D[a] \* D[b]);

D[c] = ssov(result, 32);

**MADDS**                      **E[c], E[d], D[a], D[b] (RRR2)**

64 + (32 \* 32)--> 64 signed saturated



result = E[d] + (D[a] \* D[b]);

E[c] = ssov(result, 64);

### Status Flags

C	Not set by these instructions.
V	32-bit result: overflow = (result > 7FFFFFFF <sub>H</sub> ) OR (result < -80000000 <sub>H</sub> ); if (overflow) then PSW.V = 1 else PSW.V = 0; 64-bit result: overflow = (result > 7FFFFFFFFFFFFFFF <sub>H</sub> ) OR (result < -8000000000000000 <sub>H</sub> ); if (overflow) then PSW.V = 1 else PSW.V = 0;
SV	if (overflow) then PSW.SV = 1 else PSW.SV = PSW.SV;
AV	32-bit result: advanced_overflow = result[31] ^ result[30]; if (advanced_overflow) then PSW.AV = 1 else PSW.AV = 0; 64-bit result: advanced_overflow = result[63] ^ result[62]; if (advanced_overflow) then PSW.AV = 1 else PSW.AV = 0;
SAV	if (advanced_overflow) then PSW.SAV = 1 else PSW.SAV = PSW.SAV;

### Examples

```

madd    d0, d1, d2, d3
madd    d0, d1, d2, #7
madd    e0, e2, d6, d11
madd    e0, e0, d3, #80
madds   d5, d1, d2, d2
madds   d11, d1, d2, #7
madds   e0, e2, d6, d11
madds   e8, e10, d3, #80
    
```

### See Also

-

## MADD.H

Packed Multiply-Add Q Format

## MADDS.H

Packed Multiply-Add Q Format, Saturated

### Description

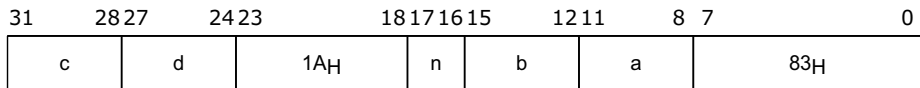
Multiply two signed 16-bit (half-word) values, add the product (left justified if  $n == 1$ ) to a signed 32-bit value and put the result into a 32-bit register. There are four cases of half-word multiplication.

Each MADDS.H result is independently saturated on overflow.

Note that  $n$  should only take the values 0 or 1, any other value returns an undefined result. If ( $n == 1$ ) then  $8000_H * 8000_H = 7FFFFFFF_H$  (for signed 16-bit \* 16-bit multiplications only).

### MADD.H                      E[c], E[d], D[a], D[b] LL, n (RRR1)

$32||32 +||+ (16U * 16L || 16L * 16L) \rightarrow 32||32$



$sc1 = (D[a][31:16] == 8000_H) \text{ AND } (D[b][15:0] == 8000_H) \text{ AND } (n == 1);$

$sc0 = (D[a][15:0] == 8000_H) \text{ AND } (D[b][15:0] == 8000_H) \text{ AND } (n == 1);$

$mul\_res1 = sc1 ? 7FFFFFFF_H : ((D[a][31:16] * D[b][15:0]) \ll n);$

$mul\_res0 = sc0 ? 7FFFFFFF_H : ((D[a][15:0] * D[b][15:0]) \ll n);$

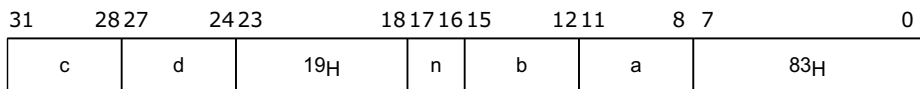
$result\_word1 = E[d][63:32] + mul\_res1;$

$result\_word0 = E[d][31:0] + mul\_res0;$

$E[c] = \{result\_word1[31:0], result\_word0[31:0]\}; // \text{ Packed fraction}$

### MADD.H                      E[c], E[d], D[a], D[b] LU, n (RRR1)

$32||32 +||+ (16U * 16L || 16L * 16U) \rightarrow 32||32$



$sc1 = (D[a][31:16] == 8000_H) \text{ AND } (D[b][15:0] == 8000_H) \text{ AND } (n == 1);$

$sc0 = (D[a][15:0] == 8000_H) \text{ AND } (D[b][31:16] == 8000_H) \text{ AND } (n == 1);$

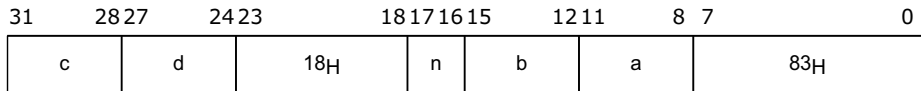
$mul\_res1 = sc1 ? 7FFFFFFF_H : ((D[a][31:16] * D[b][15:0]) \ll n);$

$mul\_res0 = sc0 ? 7FFFFFFF_H : ((D[a][15:0] * D[b][31:16]) \ll n);$

result\_word1 = E[d][63:32] + mul\_res1;  
result\_word0 = E[d][31:0] + mul\_res0;  
E[c] = {result\_word1[31:0], result\_word0[31:0]}; // Packed fraction

**MADD.H**                    **E[c], E[d], D[a], D[b] UL, n (RRR1)**

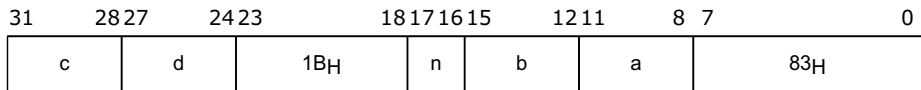
32||32 +||+ (16U \* 16U || 16L \* 16L)--> 32||32



sc1 = (D[a][31:16] == 8000<sub>H</sub>) AND (D[b][31:16] == 8000<sub>H</sub>) AND (n == 1);  
sc0 = (D[a][15:0] == 8000<sub>H</sub>) AND (D[b][15:0] == 8000<sub>H</sub>) AND (n == 1);  
mul\_res1 = sc1 ? 7FFFFFFF<sub>H</sub> : ((D[a][31:16] \* D[b][31:16]) << n);  
mul\_res0 = sc0 ? 7FFFFFFF<sub>H</sub> : ((D[a][15:0] \* D[b][15:0]) << n);  
result\_word1 = E[d][63:32] + mul\_res1;  
result\_word0 = E[d][31:0] + mul\_res0;  
E[c] = {result\_word1[31:0], result\_word0[31:0]}; // Packed fraction

**MADD.H**                    **E[c], E[d], D[a], D[b] UU, n (RRR1)**

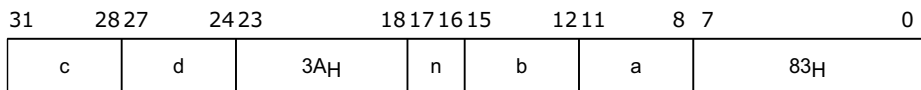
32||32 +||+ (16L \* 16U || 16U \* 16U)--> 32||32



sc1 = (D[a][15:0] == 8000<sub>H</sub>) AND (D[b][31:16] == 8000<sub>H</sub>) AND (n == 1);  
sc0 = (D[a][31:16] == 8000<sub>H</sub>) AND (D[b][31:16] == 8000<sub>H</sub>) AND (n == 1);  
mul\_res1 = sc1 ? 7FFFFFFF<sub>H</sub> : ((D[a][15:0] \* D[b][31:16]) << n);  
mul\_res0 = sc0 ? 7FFFFFFF<sub>H</sub> : ((D[a][31:16] \* D[b][31:16]) << n);  
result\_word1 = E[d][63:32] + mul\_res1;  
result\_word0 = E[d][31:0] + mul\_res0;  
E[c] = {result\_word1[31:0], result\_word0[31:0]}; // Packed fraction

**MADDS.H**                    **E[c], E[d], D[a], D[b] LL, n (RRR1)**

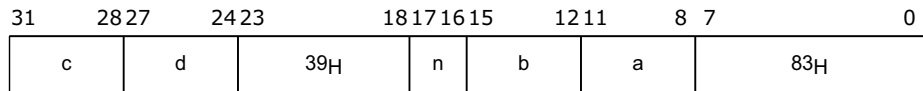
32||32 +||+ (16U \* 16L || 16L \* 16L)--> 32||32 saturated



```
sc1 = (D[a][31:16] == 8000H) AND (D[b][15:0] == 8000H) AND (n == 1);
sc0 = (D[a][15:0] == 8000H) AND (D[b][15:0] == 8000H) AND (n == 1);
mul_res1 = sc1 ? 7FFFFFFFH : ((D[a][31:16] * D[b][15:0]) << n);
mul_res0 = sc0 ? 7FFFFFFFH : ((D[a][15:0] * D[b][15:0]) << n);
result_word1 = E[d][63:32] + mul_res1;
result_word0 = E[d][31:0] + mul_res0;
E[c] = {ssov(result_word1, 32), ssov(result_word0, 32)}; // Packed fraction
```

**MADDS.H**                    **E[c], E[d], D[a], D[b] LU, n (RRR1)**

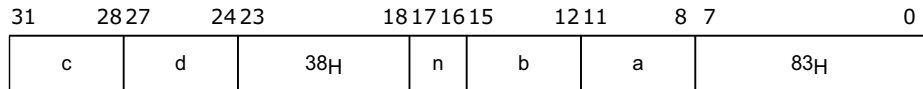
32||32 +||+ (16U \* 16L || 16L \* 16U)--> 32||32 saturated



```
sc1 = (D[a][31:16] == 8000H) AND (D[b][15:0] == 8000H) AND (n == 1);
sc0 = (D[a][15:0] == 8000H) AND (D[b][31:16] == 8000H) AND (n == 1);
mul_res1 = sc1 ? 7FFFFFFFH : ((D[a][31:16] * D[b][15:0]) << n);
mul_res0 = sc0 ? 7FFFFFFFH : ((D[a][15:0] * D[b][31:16]) << n);
result_word1 = E[d][63:32] + mul_res1;
result_word0 = E[d][31:0] + mul_res0;
E[c] = {ssov(result_word1, 32), ssov(result_word0, 32)}; // Packed fraction
```

**MADDS.H**                    **E[c], E[d], D[a], D[b] UL, n (RRR1)**

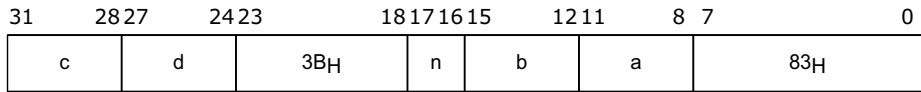
32||32 +||+ (16U \* 16U || 16L \* 16L)--> saturated



```
sc1 = (D[a][31:16] == 8000H) AND (D[b][31:16] == 8000H) AND (n == 1);
sc0 = (D[a][15:0] == 8000H) AND (D[b][15:0] == 8000H) AND (n == 1);
mul_res1 = sc1 ? 7FFFFFFFH : ((D[a][31:16] * D[b][31:16]) << n);
mul_res0 = sc0 ? 7FFFFFFFH : ((D[a][15:0] * D[b][15:0]) << n);
result_word1 = E[d][63:32] + mul_res1;
result_word0 = E[d][31:0] + mul_res0;
E[c] = {ssov(result_word1, 32), ssov(result_word0, 32)}; // Packed fraction
```

**MADDS.H**                    **E[c], E[d], D[a], D[b] UU, n (RRR1)**

32||32 +||+ (16L \* 16U || 16U \* 16U)--> 32||32 saturated



```

sc1 = (D[a][15:0] == 8000H) AND (D[b][31:16] == 8000H) AND (n == 1);
sc0 = (D[a][31:16] == 8000H) AND (D[b][31:16] == 8000H) AND (n == 1);
mul_res1 = sc1 ? 7FFFFFFFH : ((D[a][15:0] * D[b][31:16]) << n);
mul_res0 = sc0 ? 7FFFFFFFH : ((D[a][31:16] * D[b][31:16]) << n);
result_word1 = E[d][63:32] + mul_res1;
result_word0 = E[d][31:0] + mul_res0;
E[c] = {ssov(result_word1, 32), ssov(result_word0, 32)}; // Packed fraction
    
```

**Status Flags**

C	Not set by these instructions.
V	ov_word1 = (result_word1 > 7FFFFFFF <sub>H</sub> ) OR (result_word1 < -80000000 <sub>H</sub> ); ov_word0 = (result_word0 > 7FFFFFFF <sub>H</sub> ) OR (result_word0 < -80000000 <sub>H</sub> ); overflow = ov_word1 OR ov_word0; if (overflow) then PSW.V = 1 else PSW.V = 0;
SV	if (overflow) then PSW.SV = 1 else PSW.SV = PSW.SV;
AV	aov_word1 = result_word1[31] ^ result_word1[30]; aov_word0 = result_word0[31] ^ result_word0[30]; advanced_overflow = aov_word1 OR aov_word0; if (advanced_overflow) then PSW.AV = 1 else PSW.AV = 0;
SAV	if (advanced_overflow) then PSW.SAV = 1 else PSW.SAV = PSW.SAV;

**Examples**

-

**See Also**

-

See Also Addendum for TriCore Arch Manual, Vol2, V1.3.8, pages 493/494

## MADD.Q

Multiply-Add Q Format

## MADDS.Q

Multiply-Add Q Format, Saturated

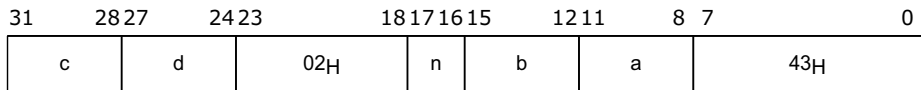
### Description

Multiply two 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. There are eight cases of 16\*16 operations, eight cases of 16\*32 operations and four cases of 32\*32 operations. On overflow the MADDS.Q result is saturated.

Note that  $n$  should only take the values 0 or 1, any other value returns an undefined result. If ( $n == 1$ ) then  $8000_H * 8000_H = 7FFFFFFF_H$  (for signed 16-bit \* 16-bit multiplications only).

### MADD.Q D[c], D[d], D[a], D[b], n (RRR1)

$32 + (32 * 32)_{Up} \rightarrow 32$

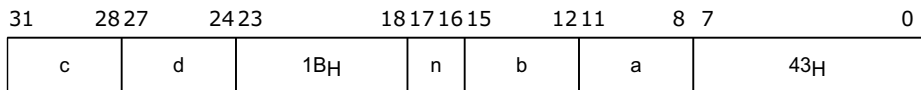


result = D[d] + (((D[a] \* D[b]) << n) >> 32);

D[c] = result[31:0]; // Fraction

### MADD.Q E[c], E[d], D[a], D[b], n (RRR1)

$64 + (32 * 32) \rightarrow 64$

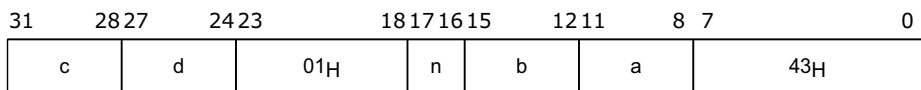


result = E[d] + ((D[a] \* D[b]) << n);

E[c] = result[63:0]; // Multi-precision fraction

### MADD.Q D[c], D[d], D[a], D[b] L, n (RRR1)

$32 + (16L * 32)_{Up} \rightarrow 32$



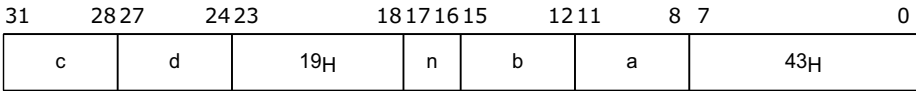
result = D[d] + (((D[a] \* D[b][15:0]) << n) >> 16);

D[c] = result[31:0]; // Fraction



**MADD.Q**                    **E[c], E[d], D[a], D[b] L, n (RRR1)**

64 + (16L \* 32) --> 64

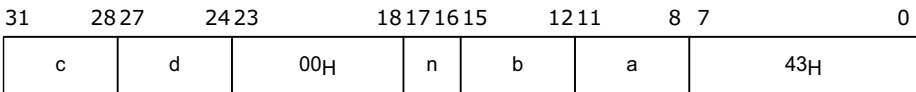


result = E[d] + ((D[a] \* D[b][15:0]) << n);

E[c] = result[63:0]; // Multi-precision accumulator

**MADD.Q**                    **D[c], D[d], D[a], D[b] U, n (RRR1)**

32 + (16U \* 32)<sub>Up</sub> --> 32

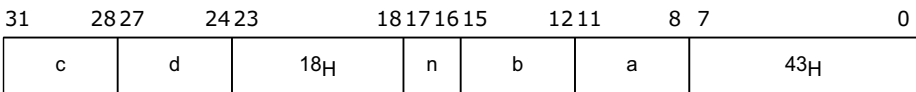


result = D[d] + (((D[a] \* D[b][31:16]) << n) >> 16);

D[c] = result[31:0]; // Fraction

**MADD.Q**                    **E[c], E[d], D[a], D[b] U, n (RRR1)**

64 + (16U \* 32) --> 64

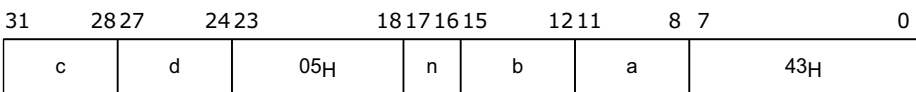


result = E[d] + ((D[a] \* D[b][31:16]) << n);

E[c] = result[63:0]; // Multi-precision accumulator

**MADD.Q**                    **D[c], D[d], D[a] L, D[b] L, n (RRR1)**

32 + (16L \* 16L) --> 32



sc = (D[a][15:0] == 8000<sub>H</sub>) AND (D[b][15:0] == 8000<sub>H</sub>) AND (n == 1);

mul\_res = sc ? 7FFFFFFF<sub>H</sub> : ((D[a][15:0] \* D[b][15:0]) << n);

result = D[d] + mul\_res;

D[c] = result[31:0]; // Fraction

**MADD.Q**                    **E[c], E[d], D[a] L, D[b] L, n (RRR1)**

64 + (16L \* 16L) --> 64

31	28 27	24 23	18 17 16 15	12 11	8 7	0
c	d	1D <sub>H</sub>	n	b	a	43 <sub>H</sub>

sc = (D[a][15:0] == 8000<sub>H</sub>) AND (D[b][15:0] == 8000<sub>H</sub>) AND (n == 1);

mul\_res = sc ? 7FFFFFFF<sub>H</sub> : ((D[a][15:0] \* D[b][15:0]) << n);

result = E[d] + (mul\_res << 16);

E[c] = result[63:0]; // Multi-precision accumulator

**MADD.Q**                      **D[c], D[d], D[a] U, D[b] U, n (RRR1)**

32 + (16U \* 16U) --> 32

31	28 27	24 23	18 17 16 15	12 11	8 7	0
c	d	04 <sub>H</sub>	n	b	a	43 <sub>H</sub>

sc = (D[a][31:16] == 8000<sub>H</sub>) AND (D[b][31:16] == 8000<sub>H</sub>) AND (n == 1);

mul\_res = sc ? 7FFFFFFF<sub>H</sub> : ((D[a][31:16] \* D[b][31:16]) << n);

result = D[d] + mul\_res;

D[c] = result[31:0]; // Fraction

**MADD.Q**                      **E[c], E[d], D[a] U, D[b] U, n (RRR1)**

64 + (16U \* 16U) --> 64

31	28 27	24 23	18 17 16 15	12 11	8 7	0
c	d	1C <sub>H</sub>	n	b	a	43 <sub>H</sub>

sc = (D[a][31:16] == 8000<sub>H</sub>) AND (D[b][31:16] == 8000<sub>H</sub>) AND (n == 1);

mul\_res = sc ? 7FFFFFFF<sub>H</sub> : ((D[a][31:16] \* D[b][31:16]) << n);

result = E[d] + (mul\_res << 16);

E[c] = result[63:0]; // Multi-precision accumulator

**MADDS.Q**                      **D[c], D[d], D[a], D[b], n (RRR1)**

32 + (32 \* 32)Up --> 32 saturated

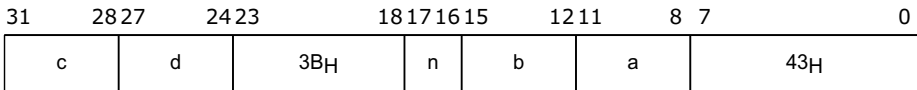
31	28 27	24 23	18 17 16 15	12 11	8 7	0
c	d	22 <sub>H</sub>	n	b	a	43 <sub>H</sub>

result = D[d] + (((D[a] \* D[b]) << n) >> 32);

D[c] = ssov(result, 32); // Fraction

**MADDS.Q**                    **E[c], E[d], D[a], D[b], n (RRR1)**

64 + (32 \* 32) --> 64 saturated

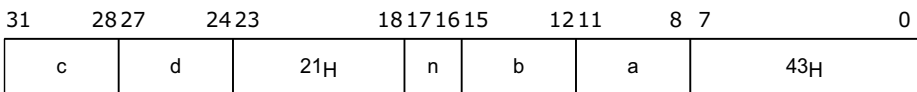


result = E[d] + ((D[a] \* D[b]) << n);

E[c] = ssov(result, 64) // Multi-precision fraction

**MADDS.Q**                    **D[c], D[d], D[a], D[b] L, n (RRR1)**

32 + (16L \* 32)<sub>Up</sub> --> 32 saturated

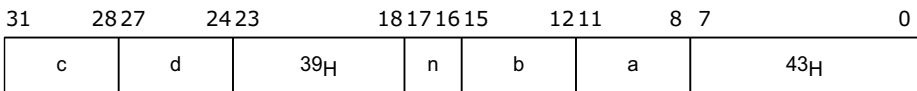


result = D[d] + (((D[a] \* D[b][15:0]) << n) >> 16);

D[c] = ssov(result, 32); // Fraction

**MADDS.Q**                    **E[c], E[d], D[a], D[b] L, n (RRR1)**

64 + (16L \* 32) --> 64 saturated

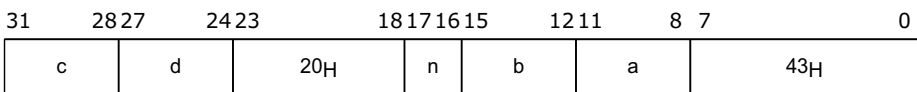


result = E[d] + ((D[a] \* D[b][15:0]) << n);

E[c] = ssov(result, 64); // Multi-precision accumulator

**MADDS.Q**                    **D[c], D[d], D[a], D[b] U, n (RRR1)**

32 + (16U \* 32)<sub>Up</sub> --> 32 saturated

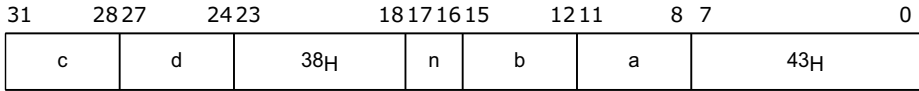


result = D[d] + (((D[a] \* D[b][31:16]) << n) >> 16);

D[c] = ssov(result, 32); // Fraction

**MADDS.Q**                    **E[c], E[d], D[a], D[b] U, n (RRR1)**

64 + (16U \* 32) --> 64 saturated

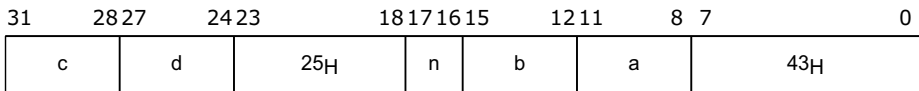


result = E[d] + ((D[a] \* D[b][31:16]) << n);

E[c] = ssov(result, 64); // Multi-precision accumulator

**MADDS.Q**                    **D[c], D[d], D[a] L, D[b] L, n (RRR1)**

32 + (16L \* 16L) --> 32 saturated



sc = (D[a][15:0] == 8000<sub>H</sub>) AND (D[b][15:0] == 8000<sub>H</sub>) AND (n == 1);

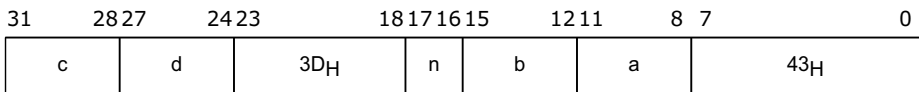
mul\_res = sc ? 7FFFFFFF<sub>H</sub> : ((D[a][15:0] \* D[b][15:0]) << n);

result = D[d] + mul\_res;

D[c] = ssov(result, 32); // Fraction

**MADDS.Q**                    **E[c], E[d], D[a] L, D[b] L, n (RRR1)**

64 + (16L \* 16L) --> 64 saturated



sc = (D[a][15:0] == 8000<sub>H</sub>) AND (D[b][15:0] == 8000<sub>H</sub>) AND (n == 1);

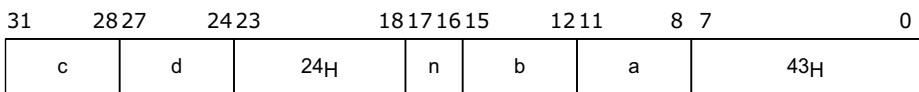
mul\_res = sc ? 7FFFFFFF<sub>H</sub> : ((D[a][15:0] \* D[b][15:0]) << n);

result = E[d] + (mul\_res << 16);

E[c] = ssov(result, 64); // Multi-precision accumulator

**MADDS.Q**                    **D[c], D[d], D[a] U, D[b] U, n (RRR1)**

32 + (16U \* 16U) --> 32 saturated



sc = (D[a][31:16] == 8000<sub>H</sub>) AND (D[b][31:16] == 8000<sub>H</sub>) AND (n == 1);

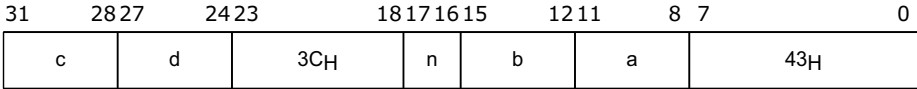
mul\_res = sc ? 7FFFFFFF<sub>H</sub> : ((D[a][31:16] \* D[b][31:16]) << n);

result = D[d] + mul\_res;

D[c] = ssov(result, 32); // Fraction

**MADDS.Q**            **E[c], E[d], D[a] U, D[b] U, n (RRR1)**

64 + (16U \* 16U) --> 64 saturated



sc = (D[a][31:16] == 8000<sub>H</sub>) AND (D[b][31:16] == 8000<sub>H</sub>) AND (n == 1);

mul\_res = sc ? 7FFFFFFF<sub>H</sub> : ((D[a][31:16] \* D[b][31:16]) << n);

result = E[d] + (mul\_res << 16);

E[c] = ssov(result, 64); // Multi-precision accumulator

**Status Flags**

<b>C</b>	Not set by these instructions.
<b>V</b>	32-bit result: overflow = (result > 7FFFFFFF <sub>H</sub> ) OR (result < -80000000 <sub>H</sub> ); if (overflow) then PSW.V = 1 else PSW.V = 0; 64-bit result: overflow = (result > 7FFFFFFFFFFFFFFF <sub>H</sub> ) OR (result < -8000000000000000 <sub>H</sub> ); if (overflow) then PSW.V = 1 else PSW.V = 0;
<b>SV</b>	if (overflow) then PSW.SV = 1 else PSW.SV = PSW.SV;
<b>AV</b>	32-bit result: advanced_overflow = result[31] ^ result[30]; if (advanced_overflow) then PSW.AV = 1 else PSW.AV = 0; 64-bit result: advanced_overflow = result[63] ^ result[62]; if (advanced_overflow) then PSW.AV = 1 else PSW.AV = 0;
<b>SAV</b>	if (advanced_overflow) then PSW.SAV = 1 else PSW.SAV = PSW.SAV;

**Examples**

```

madd.q    d0, d1, d2, d3, #1
madd.q    d0, d1, d2, d6U, #1
madd.q    d0, d2, d1, d3L, #1
madd.q    d2, d0, d3U, d4U, #1
madd.q    d2, d0, d4L, d4L, #1
madd.q    e2, e2, d3, d7, #1
madd.q    e2, e2, d4, d6U, #1
madd.q    e2, e2, d5, d6L, #1
madd.q    e2, e2, d6U, d7U, #1
    
```

See Also Addendum for TriCore Arch Manual, Vol2, V1.3.8, pages 493/494

```
madd.q    e2, e2, d8L, d0L, #1
madds.q   d0, d1, d2, d3, #1
madds.q   d0, d1, d2, d6U, #1
madds.q   d0, d2, d1, d3L, #1
madds.q   d2, d0, d3U, d4U, #1
madds.q   d2, d0, d4L, d4L, #1
madds.q   e2, e2, d3, d7, #1
madds.q   e2, e2, d4, d6U, #1
madds.q   e2, e2, d5, d6L, #1
madds.q   e2, e2, d6U, d7U, #1
madds.q   e2, e0, d11L, d4L, #1
```

**See Also**

-

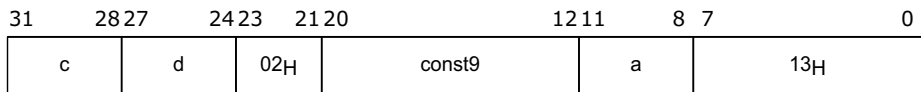
**MADD.U**  
Multiply-Add Unsigned  
**MADDS.U**  
Multiply-Add Unsigned, Saturated

**Description**

Multiply two 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. The value const9 is zero-extended before the multiplication is performed. The MADDS.U result is saturated on overflow.

**MADD.U**                    **E[c], E[d], D[a], const9 (RCR)**

64 + (32 \* K9) --> 64 unsigned

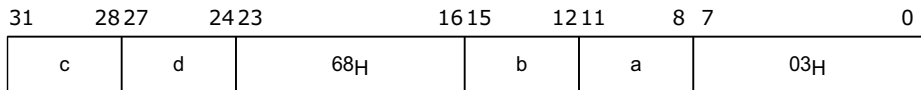


result = E[d] + (D[a] \* zero\_ext(const9)); // unsigned operators

E[c] = result[63:0];

**MADD.U**                    **E[c], E[d], D[a], D[b] (RRR2)**

32 + (32 \* 32) --> 32 unsigned

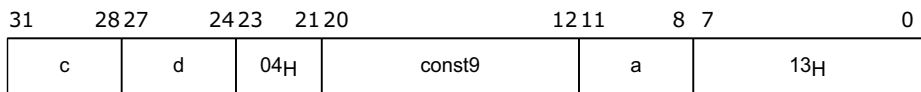


result = E[d] + (D[a] \* D[b]); // unsigned operators

E[c] = result[63:0];

**MADDS.U**                    **D[c], D[d], D[a], const9 (RCR)**

32 + (32 \* K9) --> 32 unsigned saturated



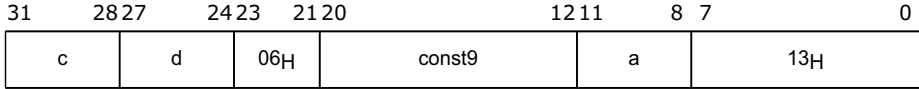
result = D[d] + (D[a] \* zero\_ext(const9)); // unsigned operators

D[c] = suov(result, 32);

**MADDS.U**                    **E[c], E[d], D[a], const9 (RCR)**

64 + (32 \* K9) --> 64 unsigned saturated

See Also Addendum for TriCore Arch Manual, Vol2, V1.3.8, pages 493/494

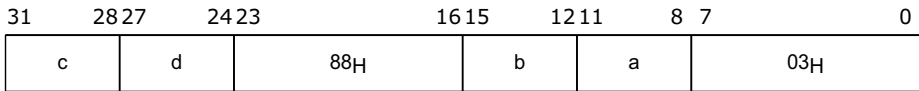


result = E[d] + (D[a] \* zero\_ext(const9)); // unsigned operators

E[c] = suov(result, 64);

**MADDS.U                    D[c], D[d], D[a], D[b] (RRR2)**

32 + (32 \* 32) --> unsigned saturated

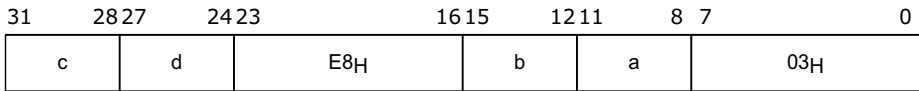


result = D[d] + (D[a] \* D[b]); // unsigned operators

D[c] = suov(result, 32);

**MADDS.U                    E[c], E[d], D[a], D[b] (RRR2)**

64 + (32 \* 32) --> 64 unsigned saturated



result = E[d] + (D[a] \* D[b]); // unsigned operators

E[c] = suov(result, 64);

**Status Flags**

C	Not set by these instructions.
V	32-bit result: overflow = (result > FFFFFFFFH); if (overflow) then PSW.V = 1 else PSW.V = 0; 64-bit result: overflow = (result > FFFFFFFFFFFFFFFFH); if (overflow) then PSW.V = 1 else PSW.V = 0;
SV	if (overflow) then PSW.SV = 1 else PSW.SV = PSW.SV;
AV	32-bit result: advanced_overflow = result[31] ^ result[30]; if (advanced_overflow) then PSW.AV = 1 else PSW.AV = 0; 64-bit result: advanced_overflow = result[63] ^ result[62]; if (advanced_overflow) then PSW.AV = 1 else PSW.AV = 0;

See Also Addendum for TriCore Arch Manual, Vol2, V1.3.8, pages 493/494



---

SAV	if (advanced_overflow) then PSW.SAV = 1 else PSW.SAV = PSW.SAV;
-----	---

---

**Examples**

```
madd.u    e0, e2, d6, d11
madd.u    e0, e0, d3, #56
madds.u   d5, d1, d2, d2
madds.u   d11, d1, d2, #7
madds.u   e0, e2, d6, d11
madds.u   e8, e0, d0, #80
```

**See Also**

-

## MADDM.H

Packed Multiply-Add Q Format Multi-precision

## MADDMS.H

Packed Multiply-Add Q Format Multi-precision, Saturated

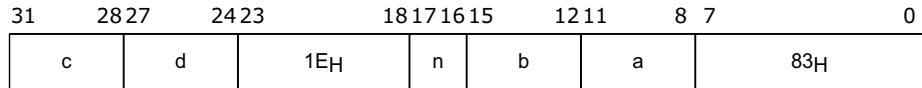
### Description

Perform two multiplications of two signed 16-bit (half-word) values. Add the two products (left justified if  $n == 1$ ) left-shifted by 16, to a signed 64-bit value and put the result in a 64-bit register. The MADDMS.H result is saturated on overflow. There are four cases of half-word multiplication.

Note that  $n$  should only take the values 0 or 1, any other value returns an undefined result. If ( $n == 1$ ) then  $8000_H * 8000_H = 7FFFFFFF_H$  (for signed 16-bit \* 16-bit multiplications only).

### MADDM.H                    E[c], E[d], D[a], D[b] LL, n (RRR1)

$64 + (16U * 16L) + (16L * 16L) \rightarrow 64$



$sc1 = (D[a][31:16] == 8000_H) \text{ AND } (D[b][15:0] == 8000_H) \text{ AND } (n == 1);$

$sc0 = (D[a][15:0] == 8000_H) \text{ AND } (D[b][15:0] == 8000_H) \text{ AND } (n == 1);$

$result\_word1 = sc1 ? 7FFFFFFF_H : ((D[a][31:16] * D[b][15:0]) \ll n);$

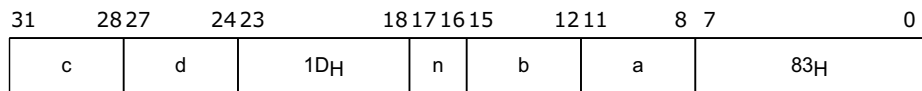
$result\_word0 = sc0 ? 7FFFFFFF_H : ((D[a][15:0] * D[b][15:0]) \ll n);$

$result = E[d] + ((result\_word1 + result\_word0) \ll 16);$

$E[c] = result[63:0];$  // Multi-precision accumulator

### MADDM.H                    E[c], E[d], D[a], D[b] LU, n (RRR1)

$64 + (16U * 16L) + (16L * 16U) \rightarrow 64$



$sc1 = (D[a][31:16] == 8000_H) \text{ AND } (D[b][15:0] == 8000_H) \text{ AND } (n == 1);$

$sc0 = (D[a][15:0] == 8000_H) \text{ AND } (D[b][31:16] == 8000_H) \text{ AND } (n == 1);$

$result\_word1 = sc1 ? 7FFFFFFF_H : ((D[a][31:16] * D[b][15:0]) \ll n);$

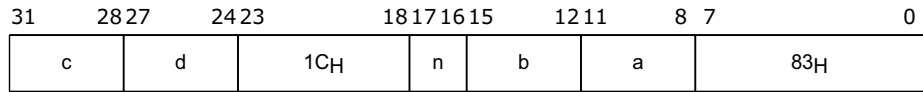
$result\_word0 = sc0 ? 7FFFFFFF_H : ((D[a][15:0] * D[b][31:16]) \ll n);$

$result = E[d] + ((result\_word1 + result\_word0) \ll 16);$

$E[c] = result[63:0];$  // Multi-precision accumulator

**MADDM.H E[c], E[d], D[a], D[b] UL, n (RRR1)**

64 + (16U \* 16U) + (16L \* 16L) --> 64



sc1 = (D[a][31:16] == 8000<sub>H</sub>) AND (D[b][31:16] == 8000<sub>H</sub>) AND (n == 1);

sc0 = (D[a][15:0] == 8000<sub>H</sub>) AND (D[b][15:0] == 8000<sub>H</sub>) AND (n == 1);

result\_word1 = sc1 ? 7FFFFFFF<sub>H</sub> : ((D[a][31:16] \* D[b][31:16]) << n);

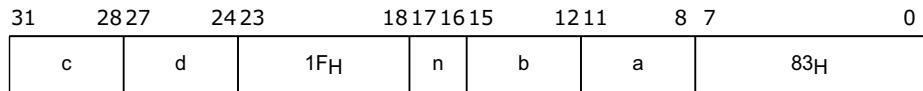
result\_word0 = sc0 ? 7FFFFFFF<sub>H</sub> : ((D[a][15:0] \* D[b][15:0]) << n);

result = E[d] + ((result\_word1 + result\_word0) << 16);

E[c] = result[63:0]; // Multi-precision accumulator

**MADDM.H E[c], E[d], D[a], D[b] UU, n (RRR1)**

64 + (16L \* 16U) + (16U \* 16U) --> 64



sc1 = (D[a][15:0] == 8000<sub>H</sub>) AND (D[b][31:16] == 8000<sub>H</sub>) AND (n == 1);

sc0 = (D[a][31:16] == 8000<sub>H</sub>) AND (D[b][31:16] == 8000<sub>H</sub>) AND (n == 1);

result\_word1 = sc1 ? 7FFFFFFF<sub>H</sub> : ((D[a][15:0] \* D[b][31:16]) << n);

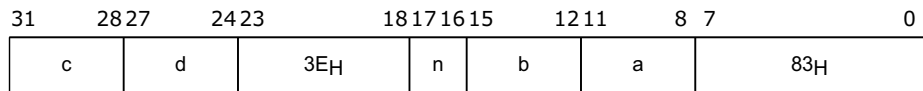
result\_word0 = sc0 ? 7FFFFFFF<sub>H</sub> : ((D[a][31:16] \* D[b][31:16]) << n);

result = E[d] + ((result\_word1 + result\_word0) << 16);

E[c] = result[63:0]; // Multi-precision accumulator

**MADDMS.H E[c], E[d], D[a], D[b] LL, n (RRR1)**

64 + (16U \* 16L) + (16L \* 16L) --> 64 saturated



sc1 = (D[a][31:16] == 8000<sub>H</sub>) AND (D[b][15:0] == 8000<sub>H</sub>) AND (n == 1);

sc0 = (D[a][15:0] == 8000<sub>H</sub>) AND (D[b][15:0] == 8000<sub>H</sub>) AND (n == 1);

result\_word1 = sc1 ? 7FFFFFFF<sub>H</sub> : ((D[a][31:16] \* D[b][15:0]) << n);

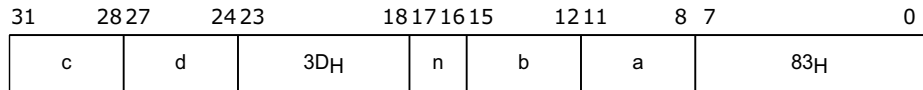
result\_word0 = sc0 ? 7FFFFFFF<sub>H</sub> : ((D[a][15:0] \* D[b][15:0]) << n);

result = E[d] + ((result\_word1 + result\_word0) << 16);

E[c] = ssov(result, 64); // Multi-precision accumulator

**MADDMS.H            E[c], E[d], D[a], D[b] LU, n (RRR1)**

64 + (16U \* 16L) + (16L \* 16U) --> 64 saturated



sc1 = (D[a][31:16] == 8000<sub>H</sub>) AND (D[b][15:0] == 8000<sub>H</sub>) AND (n == 1);

sc0 = (D[a][15:0] == 8000<sub>H</sub>) AND (D[b][31:16] == 8000<sub>H</sub>) AND (n == 1);

result\_word1 = sc1 ? 7FFFFFFF<sub>H</sub> : ((D[a][31:16] \* D[b][15:0]) << n);

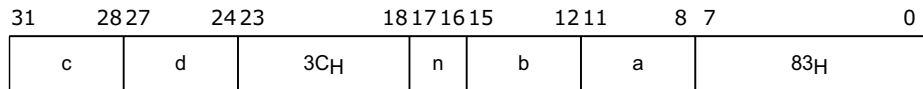
result\_word0 = sc0 ? 7FFFFFFF<sub>H</sub> : ((D[a][15:0] \* D[b][31:16]) << n);

result = E[d] + ((result\_word1 + result\_word0) << 16);

E[c] = ssov(result, 64); // Multi-precision accumulator

**MADDMS.H            E[c], E[d], D[a], D[b] UL, n (RRR1)**

64 + (16U \* 16U) + (16L \* 16L) --> 64 saturated



sc1 = (D[a][31:16] == 8000<sub>H</sub>) AND (D[b][31:16] == 8000<sub>H</sub>) AND (n == 1);

sc0 = (D[a][15:0] == 8000<sub>H</sub>) AND (D[b][15:0] == 8000<sub>H</sub>) AND (n == 1);

result\_word1 = sc1 ? 7FFFFFFF<sub>H</sub> : ((D[a][31:16] \* D[b][31:16]) << n);

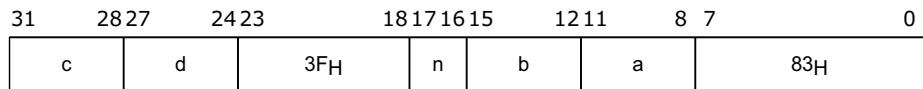
result\_word0 = sc0 ? 7FFFFFFF<sub>H</sub> : ((D[a][15:0] \* D[b][15:0]) << n);

result = E[d] + ((result\_word1 + result\_word0) << 16);

E[c] = ssov(result, 64); // Multi-precision accumulator

**MADDMS.H            E[c], E[d], D[a], D[b] UU, n (RRR1)**

64 + (16L \* 16U) + (16U \* 16U) --> 64 saturated



sc1 = (D[a][15:0] == 8000<sub>H</sub>) AND (D[b][31:16] == 8000<sub>H</sub>) AND (n == 1);

sc0 = (D[a][31:16] == 8000<sub>H</sub>) AND (D[b][31:16] == 8000<sub>H</sub>) AND (n == 1);

result\_word1 = sc1 ? 7FFFFFFF<sub>H</sub> : ((D[a][15:0] \* D[b][31:16]) << n);

result\_word0 = sc0 ? 7FFFFFFF<sub>H</sub> : ((D[a][31:16] \* D[b][31:16]) << n);

result = E[d] + ((result\_word1 + result\_word0) << 16);

E[c] = ssov(result, 64); // Multi-precision accumulator

**Status Flags**

C	Not set by these instructions.
V	overflow = (result > 7FFFFFFFFFFFFFFF <sub>H</sub> ) OR (result < -8000000000000000 <sub>H</sub> ); if (overflow) then PSW.V = 1 else PSW.V = 0;
SV	if (overflow) then PSW.SV = 1 else PSW.SV = PSW.SV;
AV	advanced_overflow = result[63] ^ result[62]; if (advanced_overflow) then PSW.AV = 1 else PSW.AV = 0;
SAV	if (advanced_overflow) then PSW.SAV = 1 else PSW.SAV = PSW.SAV;

**Examples**

-

**See Also**

-

## MADDR.H

### Packed Multiply-Add Q Format with Rounding

## MADDRS.H

### Packed Multiply-Add Q Format with Rounding, Saturated

#### Description

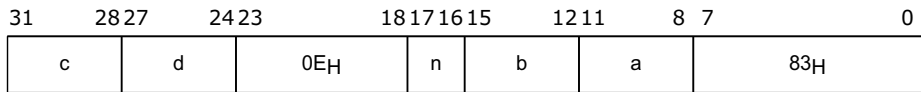
Multiply two signed 16-bit (half-word) values, add the product (left justified if  $n == 1$ ) to a signed 16-bit or 32-bit value and put the rounded result into half of a 32-bit register (Note that since there are two results the two register halves are used). There are four cases of half-word multiplication:

- $16U * 16U, 16L * 16L$
- $16U * 16L, 16L * 16U$
- $16U * 16L, 16L * 16L$
- $16L * 16U, 16U * 16U$

Note that  $n$  should only take the values 0 or 1, any other value returns an undefined result. If  $(n == 1)$  then  $8000_H * 8000_H = 7FFFFFFF_H$  (for signed 16-bit \* 16-bit multiplications only).

#### MADDR.H                      D[c], D[d], D[a], D[b] LL, n (RRR1)

$16U \parallel 16L + \parallel + (16U * 16L \parallel 16L * 16L)$  rounded  $\rightarrow 16 \parallel 16$



$sc1 = (D[a][31:16] == 8000_H) \text{ AND } (D[b][15:0] == 8000_H) \text{ AND } (n == 1);$

$sc0 = (D[a][15:0] == 8000_H) \text{ AND } (D[b][15:0] == 8000_H) \text{ AND } (n == 1);$

$mul\_res1 = sc1 ? 7FFFFFFF_H : ((D[a][31:16] * D[b][15:0]) \ll n);$

$mul\_res0 = sc0 ? 7FFFFFFF_H : ((D[a][15:0] * D[b][15:0]) \ll n);$

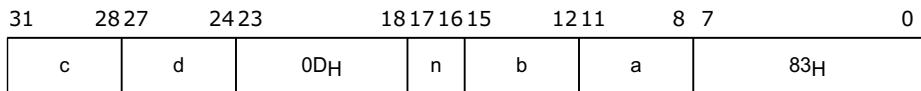
$result\_halfword1 = \{D[d][31:16], 16'b0\} + mul\_res1 + 8000_H;$

$result\_halfword0 = \{D[d][15:0], 16'b0\} + mul\_res0 + 8000_H;$

$D[c] = \{result\_halfword1[31:16], result\_halfword0[31:16]\}; // \text{Packed short fraction}$

#### MADDR.H                      D[c], D[d], D[a], D[b] LU, n (RRR1)

$16U \parallel 16L + \parallel + (16U * 16L \parallel 16L * 16U)$  rounded  $\rightarrow 16 \parallel 16$

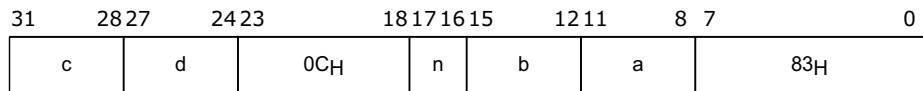


$sc1 = (D[a][31:16] == 8000_H) \text{ AND } (D[b][15:0] == 8000_H) \text{ AND } (n == 1);$

sc0 = (D[a][15:0] == 8000<sub>H</sub>) AND (D[b][31:16] == 8000<sub>H</sub>) AND (n == 1);  
mul\_res1 = sc1 ? 7FFFFFFF<sub>H</sub> : ((D[a][31:16] \* D[b][15:0]) << n);  
mul\_res0 = sc0 ? 7FFFFFFF<sub>H</sub> : ((D[a][15:0] \* D[b][31:16]) << n);  
result\_halfword1 = {D[d][31:16], 16'b0} + mul\_res1 + 8000<sub>H</sub>;  
result\_halfword0 = {D[d][15:0], 16'b0} + mul\_res0 + 8000<sub>H</sub>;  
D[c] = {result\_halfword1[31:16], result\_halfword0[31:16]}; // Packed short fraction

**MADDR.H                    D[c], D[d], D[a], D[b] UL, n (RRR1)**

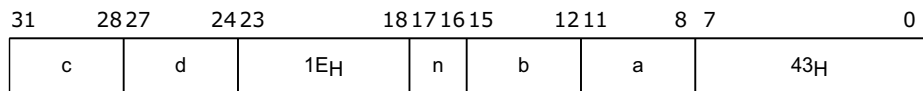
16U || 16L +||+ (16U \* 16U || 16L \* 16L) rounded --> 16||16



sc1 = (D[a][31:16] == 8000<sub>H</sub>) AND (D[b][31:16] == 8000<sub>H</sub>) AND (n == 1);  
sc0 = (D[a][15:0] == 8000<sub>H</sub>) AND (D[b][15:0] == 8000<sub>H</sub>) AND (n == 1);  
mul\_res1 = sc1 ? 7FFFFFFF<sub>H</sub> : ((D[a][31:16] \* D[b][31:16]) << n);  
mul\_res0 = sc0 ? 7FFFFFFF<sub>H</sub> : ((D[a][15:0] \* D[b][15:0]) << n);  
result\_halfword1 = {D[d][31:16], 16'b0} + mul\_res1 + 8000<sub>H</sub>;  
result\_halfword0 = {D[d][15:0], 16'b0} + mul\_res0 + 8000<sub>H</sub>;  
D[c] = {result\_halfword1[31:16], result\_halfword0[31:16]}; // Packed short fraction

**MADDR.H                    D[c], E[d], D[a], D[b] UL, n (RRR1)**

32 || 32 +||+ (16U \* 16U || 16L \* 16L) rounded --> 16||16



sc1 = (D[a][31:16] == 8000<sub>H</sub>) AND (D[b][31:16] == 8000<sub>H</sub>) AND (n == 1);  
sc0 = (D[a][15:0] == 8000<sub>H</sub>) AND (D[b][15:0] == 8000<sub>H</sub>) AND (n == 1);  
mul\_res1 = sc1 ? 7FFFFFFF<sub>H</sub> : ((D[a][31:16] \* D[b][31:16]) << n);  
mul\_res0 = sc0 ? 7FFFFFFF<sub>H</sub> : ((D[a][15:0] \* D[b][15:0]) << n);  
result\_halfword1 = E[d][63:32] + mul\_res1 + 8000<sub>H</sub>;  
result\_halfword0 = E[d][31:0] + mul\_res0 + 8000<sub>H</sub>;  
D[c] = {result\_halfword1[31:16], result\_halfword0[31:16]}; // Packed short fraction

**MADDR.H                    D[c], D[d], D[a], D[b] UU, n (RRR1)**

16U || 16L +||+ (16L \* 16U || 16U \* 16U) rounded --> 16||16

31	28 27	24 23	18 17 16 15	12 11	8 7	0
c	d	0FH	n	b	a	83H

```

sc1 = (D[a][15:0] == 8000H) AND (D[b][31:16] == 8000H) AND (n == 1);
sc0 = (D[a][31:16] == 8000H) AND (D[b][31:16] == 8000H) AND (n == 1);
mul_res1 = sc1 ? 7FFFFFFFH : ((D[a][15:0] * D[b][31:16]) << n);
mul_res0 = sc0 ? 7FFFFFFFH : ((D[a][31:16] * D[b][31:16]) << n);
result_halfword1 = {D[d][31:16], 16'b0} + mul_res1 + 8000H;
result_halfword0 = {D[d][15:0], 16'b0} + mul_res0 + 8000H;
D[c] = {result_halfword1[31:16], result_halfword0[31:16]}; // Packed short fraction

```

**MADDRS.H            D[c], D[d], D[a], D[b] LL, n (RRR1)**

16U || 16L +||+ (16U \* 16L || 16L \* 16L) rounded --> 16||16 saturated

31	28 27	24 23	18 17 16 15	12 11	8 7	0
c	d	2EH	n	b	a	83H

```

sc1 = (D[a][31:16] == 8000H) AND (D[b][15:0] == 8000H) AND (n == 1);
sc0 = (D[a][15:0] == 8000H) AND (D[b][15:0] == 8000H) AND (n == 1);
mul_res1 = sc1 ? 7FFFFFFFH : ((D[a][31:16] * D[b][15:0]) << n);
mul_res0 = sc0 ? 7FFFFFFFH : ((D[a][15:0] * D[b][15:0]) << n);
result_halfword1 = {D[d][31:16], 16'b0} + mul_res1 + 8000H;
result_halfword0 = {D[d][15:0], 16'b0} + mul_res0 + 8000H;
D[c] = {ssov(result_halfword1, 32)[31:16], ssov(result_halfword0, 32)[31:16]};
// Packed short fraction result

```

**MADDRS.H            D[c], D[d], D[a], D[b] LU, n (RRR1)**

16U || 16L +||+ (16U \* 16L || 16L \* 16U) rounded --> 16||16 saturated

31	28 27	24 23	18 17 16 15	12 11	8 7	0
c	d	2DH	n	b	a	83H

```

sc1 = (D[a][31:16] == 8000H) AND (D[b][15:0] == 8000H) AND (n == 1);
sc0 = (D[a][15:0] == 8000H) AND (D[b][31:16] == 8000H) AND (n == 1);
mul_res1 = sc1 ? 7FFFFFFFH : ((D[a][31:16] * D[b][15:0]) << n);
mul_res0 = sc0 ? 7FFFFFFFH : ((D[a][15:0] * D[b][31:16]) << n);
result_halfword1 = {D[d][31:16], 16'b0} + mul_res1 + 8000H;

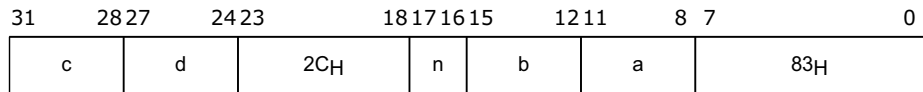
```



```
result_halfword0 = {D[d][15:0], 16'b0} + mul_res0 + 8000H;  
D[c] = {ssov(result_halfword1, 32)[31:16], ssov(result_halfword0, 32)[31:16]};  
// Packed short fraction result
```

**MADDRS.H            D[c], D[d], D[a], D[b] UL, n (RRR1)**

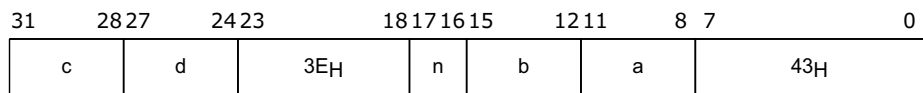
16U || 16L +||+ (16U \* 16U || 16L \* 16L) rounded --> 16||16 saturated



```
sc1 = (D[a][31:16] == 8000H) AND (D[b][31:16] == 8000H) AND (n == 1);  
sc0 = (D[a][15:0] == 8000H) AND (D[b][15:0] == 8000H) AND (n == 1);  
mul_res1 = sc1 ? 7FFFFFFFH : ((D[a][31:16] * D[b][31:16]) << n);  
mul_res0 = sc0 ? 7FFFFFFFH : ((D[a][15:0] * D[b][15:0]) << n);  
result_halfword1 = {D[d][31:16], 16'b0} + mul_res1 + 8000H;  
result_halfword0 = {D[d][15:0], 16'b0} + mul_res0 + 8000H;  
D[c] = {ssov(result_halfword1, 32)[31:16], ssov(result_halfword0, 32)[31:16]};  
// Packed short fraction result  
// Packed short fraction result
```

**MADDRS.H            D[c], E[d], D[a], D[b] UL, n (RRR1)**

32 || 32 +||+ (16U \* 16U || 16L \* 16L) rounded --> 16||16 saturated

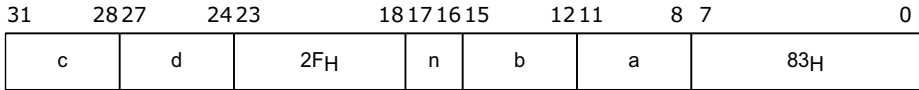


```
sc1 = (D[a][31:16] == 8000H) AND (D[b][31:16] == 8000H) AND (n == 1);  
sc0 = (D[a][15:0] == 8000H) AND (D[b][15:0] == 8000H) AND (n == 1);  
mul_res1 = sc1 ? 7FFFFFFFH : ((D[a][31:16] * D[b][31:16]) << n);  
mul_res0 = sc0 ? 7FFFFFFFH : ((D[a][15:0] * D[b][15:0]) << n);  
result_halfword1 = E[d][63:32] + mul_res1 + 8000H;  
result_halfword0 = E[d][31:0] + mul_res0 + 8000H;  
D[c] = {ssov(result_halfword1, 32)[31:16], ssov(result_halfword0, 32)[31:16]};
```

**MADDRS.H            D[c], D[d], D[a], D[b] UU, n (RRR1)**

16U || 16L +||+ (16L \* 16U || 16U \* 16U) rounded --> 16||16 saturated

**Instruction Set**



```

sc1 = (D[a][15:0] == 8000H) AND (D[b][31:16] == 8000H) AND (n == 1);
sc0 = (D[a][31:16] == 8000H) AND (D[b][31:16] == 8000H) AND (n == 1);
mul_res1 = sc1 ? 7FFFFFFFH : ((D[a][15:0] * D[b][31:16]) << n);
mul_res0 = sc0 ? 7FFFFFFFH : ((D[a][31:16] * D[b][31:16]) << n);
result_halfword1 = {D[d][31:16], 16'b0} + mul_res1 + 8000H;
result_halfword0 = {D[d][15:0], 16'b0} + mul_res0 + 8000H;
D[c] = {ssov(result_halfword1, 32)[31:16], ssov(result_halfword0, 32)[31:16]};
// Packed short fraction result
  
```

**Status Flags**

C	Not set by these instructions.
V	$ov\_halfword1 = (result\_halfword1 > 7FFFFFFF_{H}) \text{ OR } (result\_halfword1 < -80000000_{H});$ $ov\_halfword0 = (result\_halfword0 > 7FFFFFFF_{H}) \text{ OR } (result\_halfword0 < -80000000_{H});$ $overflow = ov\_halfword1 \text{ OR } ov\_halfword0;$ $\text{if } (overflow) \text{ then } PSW.V = 1 \text{ else } PSW.V = 0;$
SV	$\text{if } (overflow) \text{ then } PSW.SV = 1 \text{ else } PSW.SV = PSW.SV;$
AV	$aov\_halfword1 = result\_halfword1[31] \wedge result\_halfword1[30];$ $aov\_halfword0 = result\_halfword0[31] \wedge result\_halfword0[30];$ $advanced\_overflow = aov\_halfword1 \text{ OR } aov\_halfword0;$ $\text{if } (advanced\_overflow) \text{ then } PSW.AV = 1 \text{ else } PSW.AV = 0;$
SAV	$\text{if } (advanced\_overflow) \text{ then } PSW.SAV = 1 \text{ else } PSW.SAV = PSW.SAV;$

**Examples**

-

**See Also**

-

See Also Addendum for TriCore Arch Manual, Vol2, V1.3.8, pages 493/494

**MADDR.Q**  
Multiply-Add Q Format with Rounding

**MADDRS.Q**  
Multiply-Add Q Format with Rounding, Saturated

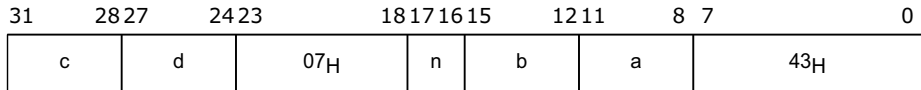
**Description**

Multiply two signed 16-bit (half-word) values, add the product (left justified if  $n == 1$ ) to a 32-bit signed value, and put the rounded result in a 32-bit register. The lower half-word is cleared. Overflow and advanced overflow are calculated on the final results.

Note that  $n$  should only take the values 0 or 1, any other value returns an undefined result. If ( $n == 1$ ) then  $8000_H * 8000_H = 7FFFFFFF_H$  (for signed 16-bit \* 16-bit multiplications only).

**MADDR.Q**                    **D[c], D[d], D[a] L, D[b] L, n (RRR1)**

32 + (16L \* 16L) rounded --> 32



$sc = (D[a][15:0] == 8000_H) \text{ AND } (D[b][15:0] == 8000_H) \text{ AND } (n == 1);$

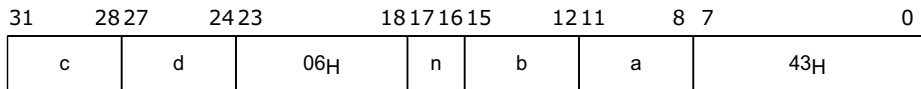
$mul\_res = sc ? 7FFFFFFF_H : ((D[a][15:0] * D[b][15:0]) << n);$

$result = D[d] + mul\_res + 8000_H;$

$D[c] = \{result[31:16], 16'b0\}; // \text{ Short fraction}$

**MADDR.Q**                    **D[c], D[d], D[a] U, D[b] U, n (RRR1)**

32 + (16U \* 16U) rounded --> 32



$sc = (D[a][31:16] == 8000_H) \text{ AND } (D[b][31:16] == 8000_H) \text{ AND } (n == 1);$

$mul\_res = sc ? 7FFFFFFF_H : ((D[a][31:16] * D[b][31:16]) << n);$

$result = D[d] + mul\_res + 8000_H;$

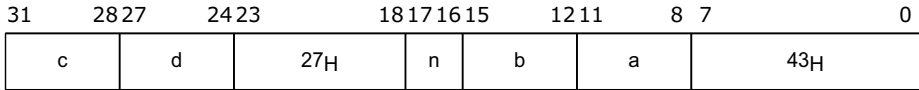
$D[c] = \{result[31:16], 16'b0\}; // \text{ Short fraction}$

**MADDRS.Q**                    **D[c], D[d], D[a] L, D[b] L, n (RRR1)**

32 + (16L \* 16L) rounded --> 32 saturated

See Also Addendum for TriCore Arch Manual, Vol2, V1.3.8, pages 493/494

Instruction Set



sc = (D[a][15:0] == 8000<sub>H</sub>) AND (D[b][15:0] == 8000<sub>H</sub>) AND (n == 1);

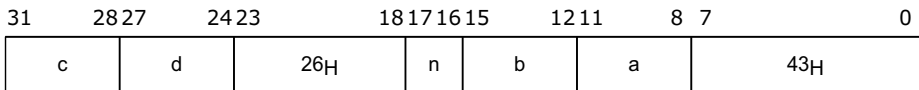
mul\_res = sc ? 7FFFFFFF<sub>H</sub> : ((D[a][15:0] \* D[b][15:0]) << n);

result = D[d] + mul\_res + 8000<sub>H</sub>;

D[c] = {ssov(result,32)[31:16]}, 16'b0; // Short fraction

**MADDRS.Q**                    **D[c], D[d], D[a] U, D[b] U, n (RRR1)**

32 + (16U \* 16U) rounded --> 32 saturated



sc = (D[a][31:16] == 8000<sub>H</sub>) AND (D[b][31:16] == 8000<sub>H</sub>) AND (n == 1);

mul\_res = sc ? 7FFFFFFF<sub>H</sub> : ((D[a][31:16] \* D[b][31:16]) << n);

result = D[d] + mul\_res + 8000<sub>H</sub>;

D[c] = {ssov(result,32)[31:16]}, 16'b0; // Short fraction

**Status Flags**

C	Not set by these instructions.
V	overflow = (result > 7FFFFFFF <sub>H</sub> ) OR (result < -80000000 <sub>H</sub> ); if (overflow) then PSW.V = 1 else PSW.V = 0;
SV	if (overflow) then PSW.SV = 1 else PSW.SV = PSW.SV;
AV	advanced_overflow = result[31] ^ result[30]; if (advanced_overflow) then PSW.AV = 1 else PSW.AV = 0;
SAV	if (advanced_overflow) then PSW.SAV = 1 else PSW.SAV = PSW.SAV;

**Examples**

-

**See Also**

-

## MADDSU.H

Packed Multiply-Add/Subtract Q Format

## MADDSUS.H

Packed Multiply-Add/Subtract Q Format Saturated

### Description

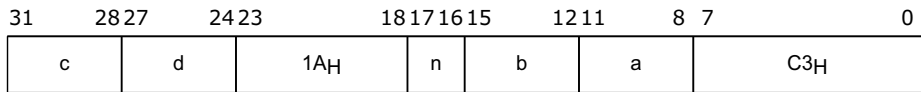
Multiply two signed 16-bit (half-word) values. Add (or subtract) the product (left justified if  $n == 1$ ) to a signed 32-bit value and put the result into a 32-bit register. Each MADDSUS.H result is independently saturated on overflow. There are four cases of half-word multiplication:

- $16U * 16U, 16L * 16L$
- $16U * 16L, 16L * 16U$
- $16U * 16L, 16L * 16L$
- $16L * 16U, 16U * 16U$

Note that  $n$  should only take the values 0 or 1, any other value returns an undefined result. If  $(n == 1)$  then  $8000_H * 8000_H = 7FFFFFFF_H$  (for signed 16-bit \* 16-bit multiplications only).

### MADDSU.H                      E[c], E[d], D[a], D[b] LL, n (RRR1)

$32||32 +||- (16U * 16L || 16L * 16L) \rightarrow 32||32$



$sc1 = (D[a][31:16] == 8000_H) \text{ AND } (D[b][15:0] == 8000_H) \text{ AND } (n == 1);$

$sc0 = (D[a][15:0] == 8000_H) \text{ AND } (D[b][15:0] == 8000_H) \text{ AND } (n == 1);$

$mul\_res1 = sc1 ? 7FFFFFFF_H : ((D[a][31:16] * D[b][15:0]) \ll n);$

$mul\_res0 = sc0 ? 7FFFFFFF_H : ((D[a][15:0] * D[b][15:0]) \ll n);$

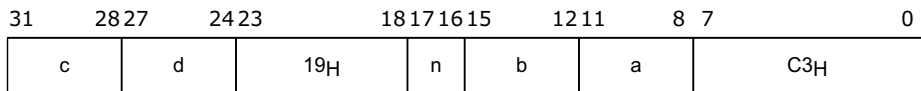
$result\_word1 = E[d][63:32] + mul\_res1;$

$result\_word0 = E[d][31:0] - mul\_res0;$

$E[c] = \{result\_word1[31:0], result\_word0[31:0]\}; // \text{ Packed fraction}$

### MADDSU.H                      E[c], E[d], D[a], D[b] LU, n (RRR1)

$32||32 +||- (16U * 16L || 16L * 16U) \rightarrow 32||32$

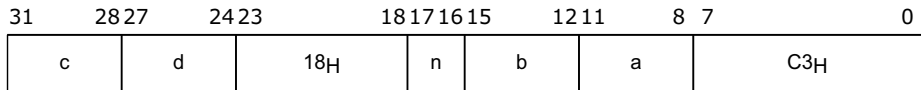


$sc1 = (D[a][31:16] == 8000_H) \text{ AND } (D[b][15:0] == 8000_H) \text{ AND } (n == 1);$

```
sc0 = (D[a][15:0] == 8000H) AND (D[b][31:16] == 8000H) AND (n == 1);
mul_res1 = sc1 ? 7FFFFFFFH : ((D[a][31:16] * D[b][15:0]) << n);
mul_res0 = sc0 ? 7FFFFFFFH : ((D[a][15:0] * D[b][31:16]) << n);
result_word1 = E[d][63:32] + mul_res1;
result_word0 = E[d][31:0] - mul_res0;
E[c] = {result_word1[31:0], result_word0[31:0]}; // Packed fraction
```

**MADDSU.H**      **E[c], E[d], D[a], D[b] UL, n (RRR1)**

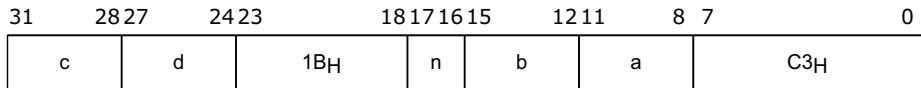
32||32 +|- (16U \* 16U || 16L \* 16L) --> 32||32



```
sc1 = (D[a][31:16] == 8000H) AND (D[b][31:16] == 8000H) AND (n == 1);
sc0 = (D[a][15:0] == 8000H) AND (D[b][15:0] == 8000H) AND (n == 1);
mul_res1 = sc1 ? 7FFFFFFFH : ((D[a][31:16] * D[b][31:16]) << n);
mul_res0 = sc0 ? 7FFFFFFFH : ((D[a][15:0] * D[b][15:0]) << n);
result_word1 = E[d][63:32] + mul_res1;
result_word0 = E[d][31:0] - mul_res0;
E[c] = {result_word1[31:0], result_word0[31:0]}; // Packed fraction
```

**MADDSU.H**      **E[c], E[d], D[a], D[b] UU, n (RRR1)**

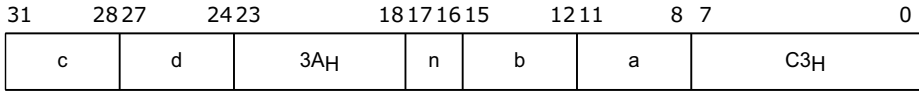
32||32 +|- (16L \* 16U || 16U \* 16U) --> 32||32



```
sc1 = (D[a][15:0] == 8000H) AND (D[b][31:16] == 8000H) AND (n == 1);
sc0 = (D[a][31:16] == 8000H) AND (D[b][31:16] == 8000H) AND (n == 1);
mul_res1 = sc1 ? 7FFFFFFFH : ((D[a][15:0] * D[b][31:16]) << n);
mul_res0 = sc0 ? 7FFFFFFFH : ((D[a][31:16] * D[b][31:16]) << n);
result_word1 = E[d][63:32] + mul_res1;
result_word0 = E[d][31:0] - mul_res0;
E[c] = {result_word1[31:0], result_word0[31:0]}; // Packed fraction
```

**MADDSUS.H**      **E[c], E[d], D[a], D[b] LL, n (RRR1)**

32||32 +|- (16U \* 16L || 16L \* 16L) --> 32||32 saturated

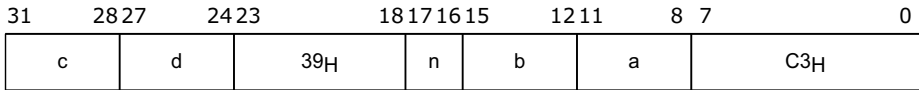


```

sc1 = (D[a][31:16] == 8000H) AND (D[b][15:0] == 8000H) AND (n == 1);
sc0 = (D[a][15:0] == 8000H) AND (D[b][15:0] == 8000H) AND (n == 1);
mul_res1 = sc1 ? 7FFFFFFFH : ((D[a][31:16] * D[b][15:0]) << n);
mul_res0 = sc0 ? 7FFFFFFFH : ((D[a][15:0] * D[b][15:0]) << n);
result_word1 = E[d][63:32] + mul_res1;
result_word0 = E[d][31:0] - mul_res0;
E[c] = {ssov(result_word1, 32), ssov(result_word0, 32)}; // Packed fraction
    
```

**MADDSUS.H      E[c], E[d], D[a], D[b] LU, n (RRR1)**

32||32 +|- (16U \* 16L || 16L \* 16U) --> 32||32 saturated

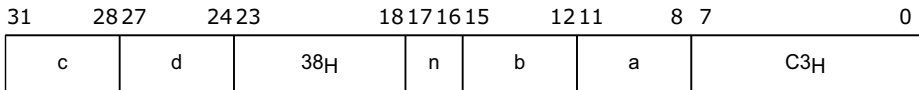


```

sc1 = (D[a][31:16] == 8000H) AND (D[b][15:0] == 8000H) AND (n == 1);
sc0 = (D[a][15:0] == 8000H) AND (D[b][31:16] == 8000H) AND (n == 1);
mul_res1 = sc1 ? 7FFFFFFFH : ((D[a][31:16] * D[b][15:0]) << n);
mul_res0 = sc0 ? 7FFFFFFFH : ((D[a][15:0] * D[b][31:16]) << n);
result_word1 = E[d][63:32] + mul_res1;
result_word0 = E[d][31:0] - mul_res0;
E[c] = {ssov(result_word1, 32), ssov(result_word0, 32)}; // Packed fraction
    
```

**MADDSUS.H      E[c], E[d], D[a], D[b] UL, n (RRR1)**

32||32 +|- (16U \* 16U || 16L \* 16L) --> 32||32 saturated



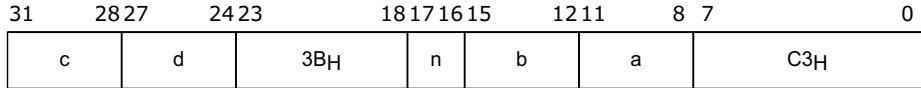
```

sc1 = (D[a][31:16] == 8000H) AND (D[b][31:16] == 8000H) AND (n == 1);
sc0 = (D[a][15:0] == 8000H) AND (D[b][15:0] == 8000H) AND (n == 1);
mul_res1 = sc1 ? 7FFFFFFFH : ((D[a][31:16] * D[b][31:16]) << n);
mul_res0 = sc0 ? 7FFFFFFFH : ((D[a][15:0] * D[b][15:0]) << n);
result_word1 = E[d][63:32] + mul_res1;
result_word0 = E[d][31:0] - mul_res0;
    
```

$E[c] = \{ssov(result\_word1, 32), ssov(result\_word0, 32)\}; //$  Packed fraction

**MADDSUS.H      E[c], E[d], D[a], D[b] UU, n (RRR1)**

32||32 +|- (16L \* 16U || 16U \* 16U) --> 32||32 saturated



$sc1 = (D[a][15:0] == 8000_{H}) \text{ AND } (D[b][31:16] == 8000_{H}) \text{ AND } (n == 1);$

$sc0 = (D[a][31:16] == 8000_{H}) \text{ AND } (D[b][31:16] == 8000_{H}) \text{ AND } (n == 1);$

$mul\_res1 = sc1 ? 7FFFFFFF_{H} : ((D[a][15:0] * D[b][31:16]) \ll n);$

$mul\_res0 = sc0 ? 7FFFFFFF_{H} : ((D[a][31:16] * D[b][31:16]) \ll n);$

$result\_word1 = E[d][63:32] + mul\_res1;$

$result\_word0 = E[d][31:0] - mul\_res0;$

$E[c] = \{ssov(result\_word1, 32), ssov(result\_word0, 32)\}; //$  Packed fraction

**Status Flags**

C	Not set by these instructions.
V	$ov\_word1 = (result\_word1 > 7FFFFFFF_{H}) \text{ OR } (result\_word1 < -80000000_{H});$ $ov\_word0 = (result\_word0 > 7FFFFFFF_{H}) \text{ OR } (result\_word0 < -80000000_{H});$ $overflow = ov\_word1 \text{ OR } ov\_word0;$ if (overflow) then PSW.V = 1 else PSW.V = 0;
SV	if (overflow) then PSW.SV = 1 else PSW.SV = PSW.SV;
AV	$aov\_word1 = result\_word1[31] \wedge result\_word1[30];$ $aov\_word0 = result\_word0[31] \wedge result\_word0[30];$ $advanced\_overflow = aov\_word1 \text{ OR } aov\_word0;$ if (advanced_overflow) then PSW.AV = 1 else PSW.AV = 0;
SAV	if (advanced_overflow) then PSW.SAV = 1 else PSW.SAV = PSW.SAV;

**Examples**

-

**See Also**

-



## MADDSUM.H

Packed Multiply-Add/Subtract Q Format Multi-precision

## MADDSUMS.H

Packed Multiply-Add/Subtract Q Format Multi-precision Saturated

### Description

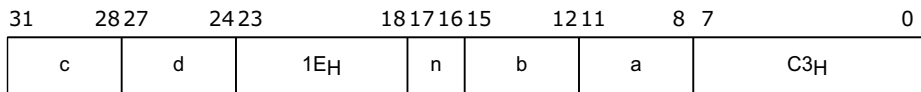
Perform two multiplications of two signed 16-bit (half-word) values. Add one product and subtract the other product (left justified if  $n == 1$ ) left-shifted by 16, to/from a signed 64-bit value and put the result in a 64-bit register. The MADDSUMS.H result is saturated on overflow. There are four cases of half-word multiplication:

- $16U * 16U, 16L * 16L$
- $16U * 16L, 16L * 16U$
- $16U * 16L, 16L * 16L$
- $16L * 16U, 16U * 16U$

Note that  $n$  should only take the values 0 or 1, any other value returns an undefined result. If  $(n == 1)$  then  $8000_H * 8000_H = 7FFFFFFF_H$  (for signed 16-bit \* 16-bit multiplications only).

### MADDSUM.H      E[c], E[d], D[a], D[b] LL, n (RRR1)

$64 + (16U * 16L) - (16L * 16L) \rightarrow 64$



$sc1 = (D[a][31:16] == 8000_H) \text{ AND } (D[b][15:0] == 8000_H) \text{ AND } (n == 1);$

$sc0 = (D[a][15:0] == 8000_H) \text{ AND } (D[b][15:0] == 8000_H) \text{ AND } (n == 1);$

$result\_word1 = sc1 ? 7FFFFFFF_H : ((D[a][31:16] * D[b][15:0]) \ll n);$

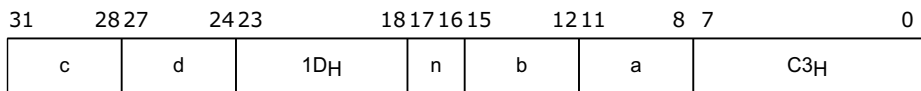
$result\_word0 = sc0 ? 7FFFFFFF_H : ((D[a][15:0] * D[b][15:0]) \ll n);$

$result = E[d] + ((result\_word1 - result\_word0) \ll 16);$

$E[c] = result[63:0];$  // Multi-precision accumulator

### MADDSUM.H      E[c], E[d], D[a], D[b] LU, n (RRR1)

$64 + (16U * 16L) - (16L * 16U) \rightarrow 64$



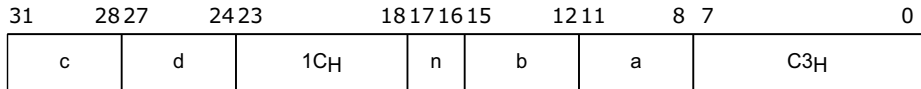
$sc1 = (D[a][31:16] == 8000_H) \text{ AND } (D[b][15:0] == 8000_H) \text{ AND } (n == 1);$

$sc0 = (D[a][15:0] == 8000_H) \text{ AND } (D[b][31:16] == 8000_H) \text{ AND } (n == 1);$

result\_word1 = sc1 ? 7FFFFFFF<sub>H</sub> : ((D[a][31:16] \* D[b][15:0]) << n);  
 result\_word0 = sc0 ? 7FFFFFFF<sub>H</sub> : ((D[a][15:0] \* D[b][31:16]) << n);  
 result = E[d] + ((result\_word1 - result\_word0) << 16);  
 E[c] = result[63:0]; // Multi-precision accumulator

**MADDSUM.H E[c], E[d], D[a], D[b] UL, n (RRR1)**

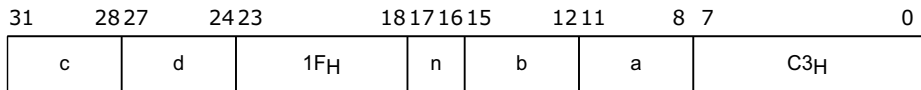
64 + (16U \* 16U) - (16L \* 16L) --> 64



sc1 = (D[a][31:16] == 8000<sub>H</sub>) AND (D[b][31:16] == 8000<sub>H</sub>) AND (n == 1);  
 sc0 = (D[a][15:0] == 8000<sub>H</sub>) AND (D[b][15:0] == 8000<sub>H</sub>) AND (n == 1);  
 result\_word1 = sc1 ? 7FFFFFFF<sub>H</sub> : ((D[a][31:16] \* D[b][31:16]) << n);  
 result\_word0 = sc0 ? 7FFFFFFF<sub>H</sub> : ((D[a][15:0] \* D[b][15:0]) << n);  
 result = E[d] + ((result\_word1 - result\_word0) << 16);  
 E[c] = result[63:0]; // Multi-precision accumulator

**MADDSUM.H E[c], E[d], D[a], D[b] UU, n (RRR1)**

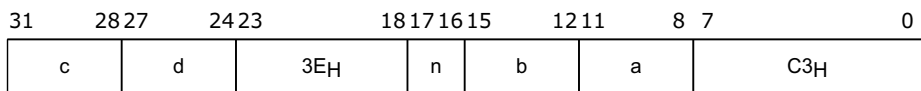
64 + (16L \* 16U) - (16U \* 16U) --> 64



sc1 = (D[a][15:0] == 8000<sub>H</sub>) AND (D[b][31:16] == 8000<sub>H</sub>) AND (n == 1);  
 sc0 = (D[a][31:16] == 8000<sub>H</sub>) AND (D[b][31:16] == 8000<sub>H</sub>) AND (n == 1);  
 result\_word1 = sc1 ? 7FFFFFFF<sub>H</sub> : ((D[a][15:0] \* D[b][31:16]) << n);  
 result\_word0 = sc0 ? 7FFFFFFF<sub>H</sub> : ((D[a][31:16] \* D[b][31:16]) << n);  
 result = E[d] + ((result\_word1 - result\_word0) << 16);  
 E[c] = result[63:0]; // Multi-precision accumulator

**MADDSUMS.H E[c], E[d], D[a], D[b] LL, n (RRR1)**

64 + (16U \* 16L) - (16L \* 16L) --> 64 saturated

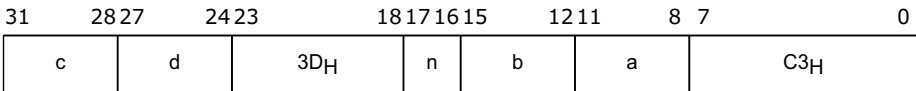


sc1 = (D[a][31:16] == 8000<sub>H</sub>) AND (D[b][15:0] == 8000<sub>H</sub>) AND (n == 1);

sc0 = (D[a][15:0] == 8000<sub>H</sub>) AND (D[b][15:0] == 8000<sub>H</sub>) AND (n == 1);  
 result\_word1 = sc1 ? 7FFFFFFF<sub>H</sub> : ((D[a][31:16] \* D[b][15:0]) << n);  
 result\_word0 = sc0 ? 7FFFFFFF<sub>H</sub> : ((D[a][15:0] \* D[b][15:0]) << n);  
 result = E[d] + ((result\_word1 - result\_word0) << 16);  
 E[c] = ssov(result, 64); // Multi-precision accumulator

**MADDSUMS.H E[c], E[d], D[a], D[b] LU, n (RRR1)**

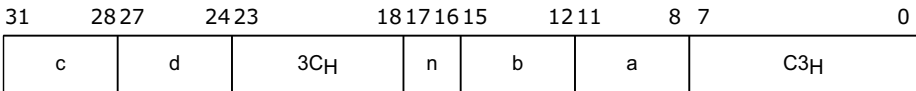
64 + (16U \* 16L) - (16L \* 16U) --> 64 saturated



sc1 = (D[a][31:16] == 8000<sub>H</sub>) AND (D[b][15:0] == 8000<sub>H</sub>) AND (n == 1);  
 sc0 = (D[a][15:0] == 8000<sub>H</sub>) AND (D[b][31:16] == 8000<sub>H</sub>) AND (n == 1);  
 result\_word1 = sc1 ? 7FFFFFFF<sub>H</sub> : ((D[a][31:16] \* D[b][15:0]) << n);  
 result\_word0 = sc0 ? 7FFFFFFF<sub>H</sub> : ((D[a][15:0] \* D[b][31:16]) << n);  
 result = E[d] + ((result\_word1 - result\_word0) << 16);  
 E[c] = ssov(result, 64); // Multi-precision accumulator

**MADDSUMS.H E[c], E[d], D[a], D[b] UL, n (RRR1)**

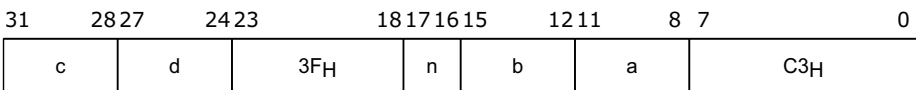
64 + (16U \* 16U) - (16L \* 16L) --> 64 saturated



sc1 = (D[a][31:16] == 8000<sub>H</sub>) AND (D[b][31:16] == 8000<sub>H</sub>) AND (n == 1);  
 sc0 = (D[a][15:0] == 8000<sub>H</sub>) AND (D[b][15:0] == 8000<sub>H</sub>) AND (n == 1);  
 result\_word1 = sc1 ? 7FFFFFFF<sub>H</sub> : ((D[a][31:16] \* D[b][31:16]) << n);  
 result\_word0 = sc0 ? 7FFFFFFF<sub>H</sub> : ((D[a][15:0] \* D[b][15:0]) << n);  
 result = E[d] + ((result\_word1 - result\_word0) << 16);  
 E[c] = ssov(result, 64); // Multi-precision accumulator

**MADDSUMS.H E[c], E[d], D[a], D[b] UU, n (RRR1)**

64 + (16L \* 16U) - (16U \* 16U) --> 64 saturated



**Instruction Set**

```
sc1 = (D[a][15:0] == 8000H) AND (D[b][31:16] == 8000H) AND (n == 1);
sc0 = (D[a][31:16] == 8000H) AND (D[b][31:16] == 8000H) AND (n == 1);
result_word1 = sc1 ? 7FFFFFFFH : ((D[a][15:0] * D[b][31:16]) << n);
result_word0 = sc0 ? 7FFFFFFFH : ((D[a][31:16] * D[b][31:16]) << n);
result = E[d] + ((result_word1 - result_word0) << 16);
E[c] = ssov(result, 64); // Multi-precision accumulator
```

**Status Flags**

C	Not set by these instructions.
V	overflow = (result > 7FFFFFFFFFFFFFFF <sub>H</sub> ) OR (result < -8000000000000000 <sub>H</sub> ); if (overflow) then PSW.V = 1 else PSW.V = 0;
SV	if (overflow) then PSW.SV = 1 else PSW.SV = PSW.SV;
AV	advanced_overflow = result[63] ^ result[62]; if (advanced_overflow) then PSW.AV = 1 else PSW.AV = 0;
SAV	if (advanced_overflow) then PSW.SAV = 1 else PSW.SAV = PSW.SAV;

**Examples**

-

**See Also**

-

## MADDSUR.H

Packed Multiply-Add/Subtract Q Format with Rounding

## MADDSURS.H

Packed Multiply-Add/Subtract Q Format with Rounding Saturated

### Description

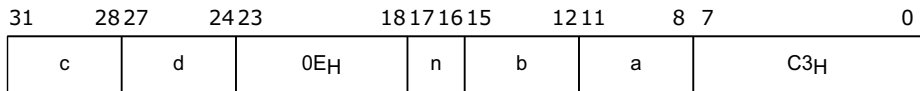
Multiply two signed 16-bit (half-word) values. Add (subtract) the product (left justified if  $n == 1$ ) to (from) a signed 16-bit value and put the rounded result into half of a 32-bit register (Note that since there are two results, the two register halves are used). There are four cases of half-word multiplication:

- $16U * 16U, 16L * 16L$
- $16U * 16L, 16L * 16U$
- $16U * 16L, 16L * 16L$
- $16L * 16U, 16U * 16U$

Note that  $n$  should only take the values 0 or 1, any other value returns an undefined result. If  $(n == 1)$  then  $8000_H * 8000_H = 7FFFFFFF_H$  (for signed 16-bit \* 16-bit multiplications only).

### MADDSUR.H      D[c], D[d], D[a], D[b] LL, n (RRR1)

$16U * 16L +|- (16U * 16L || 16L * 16L)$  rounded  $\rightarrow 16||16$



$sc1 = (D[a][31:16] == 8000_H) \text{ AND } (D[b][15:0] == 8000_H) \text{ AND } (n == 1);$

$sc0 = (D[a][15:0] == 8000_H) \text{ AND } (D[b][15:0] == 8000_H) \text{ AND } (n == 1);$

$mul\_res1 = sc1 ? 7FFFFFFF_H : ((D[a][31:16] * D[b][15:0]) \ll n);$

$mul\_res0 = sc0 ? 7FFFFFFF_H : ((D[a][15:0] * D[b][15:0]) \ll n);$

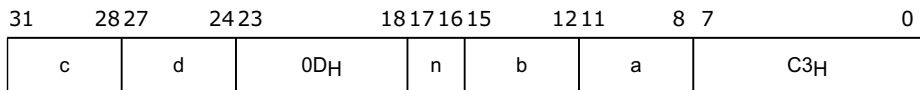
$result\_halfword1 = \{D[d][31:16], 16'b0\} + mul\_res1 + 8000_H;$

$result\_halfword0 = \{D[d][15:0], 16'b0\} - mul\_res0 + 8000_H;$

$D[c] = \{result\_halfword1[31:16], result\_halfword0[31:16]\}; // \text{Packed short fraction}$

### MADDSUR.H      D[c], D[d], D[a], D[b] LU, n (RRR1)

$16U || 16L +|- (16U * 16L || 16L * 16U)$  rounded  $\rightarrow 16||16$

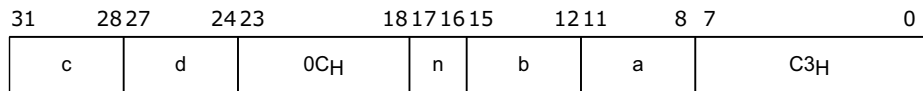


$sc1 = (D[a][31:16] == 8000_H) \text{ AND } (D[b][15:0] == 8000_H) \text{ AND } (n == 1);$

```
sc0 = (D[a][15:0] == 8000H) AND (D[b][31:16] == 8000H) AND (n == 1);
mul_res1 = sc1 ? 7FFFFFFFH : ((D[a][31:16] * D[b][15:0]) << n);
mul_res0 = sc0 ? 7FFFFFFFH : ((D[a][15:0] * D[b][31:16]) << n);
result_halfword1 = {D[d][31:16], 16'b0} + mul_res1 + 8000H;
result_halfword0 = {D[d][15:0], 16'b0} - mul_res0 + 8000H;
D[c] = {result_halfword1[31:16], result_halfword0[31:16]}; // Packed short fraction
```

**MADDSUR.H      D[c], D[d], D[a], D[b] UL, n (RRR1)**

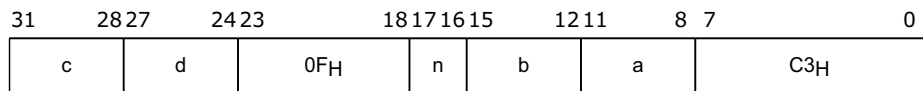
16U || 16L +|- (16U \* 16U || 16L \* 16L) rounded --> 16||16



```
sc1 = (D[a][31:16] == 8000H) AND (D[b][31:16] == 8000H) AND (n == 1);
sc0 = (D[a][15:0] == 8000H) AND (D[b][15:0] == 8000H) AND (n == 1);
mul_res1 = sc1 ? 7FFFFFFFH : ((D[a][31:16] * D[b][31:16]) << n);
mul_res0 = sc0 ? 7FFFFFFFH : ((D[a][15:0] * D[b][15:0]) << n);
result_halfword1 = {D[d][31:16], 16'b0} + mul_res1 + 8000H;
result_halfword0 = {D[d][15:0], 16'b0} - mul_res0 + 8000H;
D[c] = {result_halfword1[31:16], result_halfword0[31:16]}; // Packed short fraction
```

**MADDSUR.H      D[c], D[d], D[a], D[b] UU, n (RRR1)**

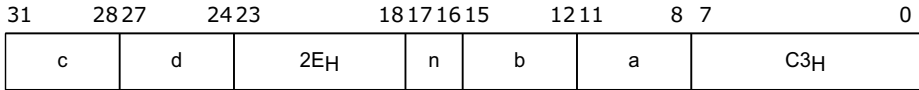
16U || 16L +|- (16L \* 16U || 16U \* 16U) rounded --> 16||16



```
sc1 = (D[a][15:0] == 8000H) AND (D[b][31:16] == 8000H) AND (n == 1);
sc0 = (D[a][31:16] == 8000H) AND (D[b][31:16] == 8000H) AND (n == 1);
mul_res1 = sc1 ? 7FFFFFFFH : ((D[a][15:0] * D[b][31:16]) << n);
mul_res0 = sc0 ? 7FFFFFFFH : ((D[a][31:16] * D[b][31:16]) << n);
result_halfword1 = {D[d][31:16], 16'b0} + mul_res1 + 8000H;
result_halfword0 = {D[d][15:0], 16'b0} - mul_res0 + 8000H;
D[c] = {result_halfword1[31:16], result_halfword0[31:16]}; // Packed short fraction
```

**MADDSURS.H      D[c], D[d], D[a], D[b] LL, n (RRR1)**

16U || 16L +|- (16U \* 16L || 16L \* 16L) rounded --> 16||16 saturated

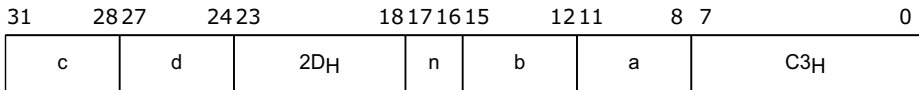


```

sc1 = (D[a][31:16] == 8000H) AND (D[b][15:0] == 8000H) AND (n == 1);
sc0 = (D[a][15:0] == 8000H) AND (D[b][15:0] == 8000H) AND (n == 1);
mul_res1 = sc1 ? 7FFFFFFFH : ((D[a][31:16] * D[b][15:0]) << n);
mul_res0 = sc0 ? 7FFFFFFFH : ((D[a][15:0] * D[b][15:0]) << n);
result_halfword1 = {D[d][31:16], 16'b0} + mul_res1 + 8000H;
result_halfword0 = {D[d][15:0], 16'b0} - mul_res0 + 8000H;
D[c] = {ssov(result_halfword1, 32)[31:16], ssov(result_halfword0, 32)[31:16]};
// Packed short fraction result
    
```

**MADDSURS.H      D[c], D[d], D[a], D[b] LU, n (RRR1)**

16U || 16L +|- (16U \* 16L || 16L \* 16U) rounded --> 16||16 saturated

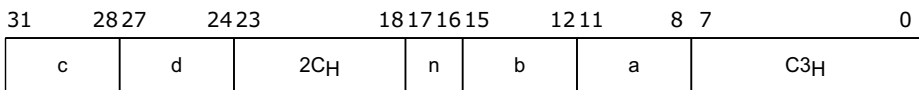


```

sc1 = (D[a][31:16] == 8000H) AND (D[b][15:0] == 8000H) AND (n == 1);
sc0 = (D[a][15:0] == 8000H) AND (D[b][31:16] == 8000H) AND (n == 1);
mul_res1 = sc1 ? 7FFFFFFFH : ((D[a][31:16] * D[b][15:0]) << n);
mul_res0 = sc0 ? 7FFFFFFFH : ((D[a][15:0] * D[b][31:16]) << n);
result_halfword1 = {D[d][31:16], 16'b0} + mul_res1 + 8000H;
result_halfword0 = {D[d][15:0], 16'b0} - mul_res0 + 8000H;
D[c] = {ssov(result_halfword1, 32)[31:16], ssov(result_halfword0, 32)[31:16]};
// Packed short fraction result
    
```

**MADDSURS.H      D[c], D[d], D[a], D[b] UL, n (RRR1)**

16U || 16L +|- (16U \* 16U || 16L \* 16L) rounded --> 16||16 saturated



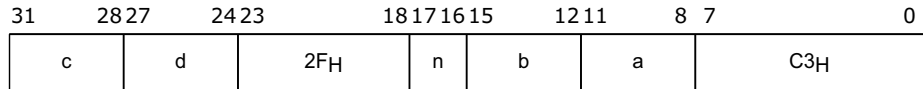
```

sc1 = (D[a][31:16] == 8000H) AND (D[b][31:16] == 8000H) AND (n == 1);
sc0 = (D[a][15:0] == 8000H) AND (D[b][15:0] == 8000H) AND (n == 1);
mul_res1 = sc1 ? 7FFFFFFFH : ((D[a][31:16] * D[b][31:16]) << n);
mul_res0 = sc0 ? 7FFFFFFFH : ((D[a][15:0] * D[b][15:0]) << n);
    
```

```
result_halfword1 = {D[d][31:16], 16'b0} + mul_res1 + 8000H;
result_halfword0 = {D[d][15:0], 16'b0} - mul_res0 + 8000H;
D[c] = {ssov(result_halfword1, 32)[31:16], ssov(result_halfword0, 32)[31:16]};
// Packed short fraction result
```

**MADDSURS.H      D[c], D[d], D[a], D[b] UU, n (RRR1)**

16U || 16L +|- (16L \* 16U || 16U \* 16U) rounded -> 16||16 saturated



```
sc1 = (D[a][15:0] == 8000H) AND (D[b][31:16] == 8000H) AND (n == 1);
sc0 = (D[a][31:16] == 8000H) AND (D[b][31:16] == 8000H) AND (n == 1);
mul_res1 = sc1 ? 7FFFFFFFH : ((D[a][15:0] * D[b][31:16]) << n);
mul_res0 = sc0 ? 7FFFFFFFH : ((D[a][31:16] * D[b][31:16]) << n);
result_halfword1 = {D[d][31:16], 16'b0} + mul_res1 + 8000H;
result_halfword0 = {D[d][15:0], 16'b0} - mul_res0 + 8000H;
D[c] = {ssov(result_halfword1, 32)[31:16], ssov(result_halfword0, 32)[31:16]};
// Packed short fraction result
```

**Status Flags**

C	Not set by these instructions.
V	ov_halfword1 = (result_halfword1 > 7FFFFFFF <sub>H</sub> ) OR (result_halfword1 < -80000000 <sub>H</sub> ); ov_halfword0 = (result_halfword0 > 7FFFFFFF <sub>H</sub> ) OR (result_halfword0 < -80000000 <sub>H</sub> ); overflow = ov_halfword1 OR ov_halfword0; if (overflow) then PSW.V = 1 else PSW.V = 0;
SV	if (overflow) then PSW.SV = 1 else PSW.SV = PSW.SV;
AV	aov_halfword1 = result_halfword1[31] ^ result_halfword1[30]; aov_halfword0 = result_halfword0[31] ^ result_halfword0[30]; advanced_overflow = aov_halfword1 OR aov_halfword0; if (advanced_overflow) then PSW.AV = 1 else PSW.AV = 0;
SAV	if (advanced_overflow) then PSW.SAV = 1 else PSW.SAV = PSW.SAV;

**Examples**

-



**See Also**

-

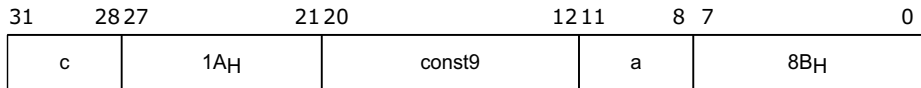
See Also Addendum for TriCore Arch Manual, Vol2, V1.3.8, pages 493/494

**MAX**  
Maximum Value  
**MAX.U**  
Maximum Value Unsigned

**Description**

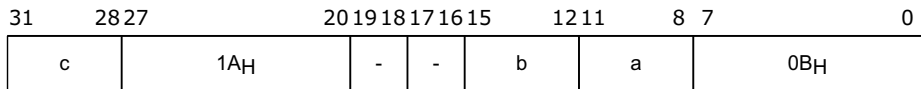
If the contents of data register D[a] are greater than the contents of either data register D[b] (instruction format RR) or const9 (instruction format RC), then put the contents of D[a] in data register D[c]; otherwise put the contents of either D[b] (format RR) or const9 (format RC) in D[c]. The operands are treated as either signed (MAX) or unsigned (MAX.U) 32-bit integers.

**MAX**                                    **D[c], D[a], const9 (RC)**



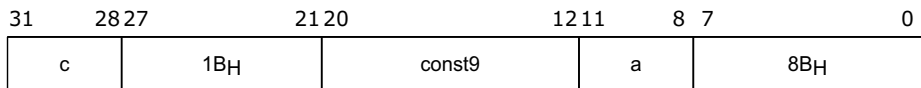
$$D[c] = (D[a] > \text{sign\_ext}(\text{const9})) ? D[a] : \text{sign\_ext}(\text{const9});$$

**MAX**                                    **D[c], D[a], D[b] (RR)**



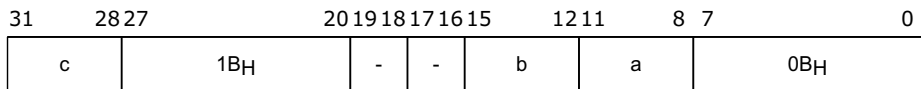
$$D[c] = (D[a] > D[b]) ? D[a] : D[b];$$

**MAX.U**                                    **D[c], D[a], const9 (RC)**



$$D[c] = (D[a] > \text{zero\_ext}(\text{const9})) ? D[a] : \text{zero\_ext}(\text{const9}); // \text{unsigned}$$

**MAX.U**                                    **D[c], D[a], D[b] (RR)**



$$D[c] = (D[a] > D[b]) ? D[a] : D[b]; // \text{unsigned}$$

**Status Flags**

C	Not set by these instructions.
---	--------------------------------

Instruction Set

V	Not set by these instructions.
SV	Not set by these instructions.
AV	Not set by these instructions.
SAV	Not set by these instructions.

**Examples**

```
max d3, d1, d2
max d3, d1, #126
max.u d3, d1, d2
max.u d3, d1, #126
```

**See Also**

**MIN, MOV**

See Also Addendum for TriCore Arch Manual, Vol2, V1.3.8, pages 493/494

## MAX.B

Maximum Value Packed Byte

## MAX.BU

Maximum Value Packed Byte Unsigned

### Description

Compute the maximum value of the corresponding bytes in D[a] and D[b] and put each result in the corresponding byte of D[c]. The operands are treated as either signed (MAX.B) or unsigned (MAX.BU), 8-bit integers.

### MAX.B D[c], D[a], D[b] (RR)

31	28 27	20 19 18 17 16 15	12 11	8 7	0	
c	5AH	-	-	b	a	0BH

$D[c][31:24] = (D[a][31:24] > D[b][31:24]) ? D[a][31:24] : D[b][31:24];$

$D[c][23:16] = (D[a][23:16] > D[b][23:16]) ? D[a][23:16] : D[b][23:16];$

$D[c][15:8] = (D[a][15:8] > D[b][15:8]) ? D[a][15:8] : D[b][15:8];$

$D[c][7:0] = (D[a][7:0] > D[b][7:0]) ? D[a][7:0] : D[b][7:0];$

### MAX.BU D[c], D[a], D[b] (RR)

31	28 27	20 19 18 17 16 15	12 11	8 7	0	
c	5BH	-	-	b	a	0BH

$D[c][31:24] = (D[a][31:24] > D[b][31:24]) ? D[a][31:24] : D[b][31:24]; // \text{unsigned}$

$D[c][23:16] = (D[a][23:16] > D[b][23:16]) ? D[a][23:16] : D[b][23:16]; // \text{unsigned}$

$D[c][15:8] = (D[a][15:8] > D[b][15:8]) ? D[a][15:8] : D[b][15:8]; // \text{unsigned}$

$D[c][7:0] = (D[a][7:0] > D[b][7:0]) ? D[a][7:0] : D[b][7:0]; // \text{unsigned}$

### Status Flags

C	Not set by these instructions.
V	Not set by these instructions.
SV	Not set by these instructions.
AV	Not set by these instructions.
SAV	Not set by these instructions.

**Examples**

max.b     d3, d1, d2  
max.bu    d3, d1, d2

**See Also**

[MAX.H](#), [MAX.HU](#), [MIN.B](#), [MIN.BU](#), [MIN.H](#), [MIN.HU](#)

See Also Addendum for TriCore Arch Manual, Vol2, V1.3.8, pages 493/494

## MAX.H

Maximum Value Packed Half-word

## MAX.HU

Maximum Value Packed Half-word Unsigned

### Description

Compute the maximum value of the corresponding half-words in D[a] and D[b] and put each result in the corresponding half-word of D[c]. The operands are treated as either signed (MAX.H) or unsigned (MAX.HU), 16-bit integers.

### MAX.H D[c], D[a], D[b] (RR)

31	28 27	20 19 18 17 16 15	12 11	8 7	0
c	7AH	-	-	b	a
0BH					

$D[c][31:16] = (D[a][31:16] > D[b][31:16]) ? D[a][31:16] : D[b][31:16];$

$D[c][15:0] = (D[a][15:0] > D[b][15:0]) ? D[a][15:0] : D[b][15:0];$

### MAX.HU D[c], D[a], D[b] (RR)

31	28 27	20 19 18 17 16 15	12 11	8 7	0
c	7BH	-	-	b	a
0BH					

$D[c][31:16] = (D[a][31:16] > D[b][31:16]) ? D[a][31:16] : D[b][31:16]; // \text{unsigned}$

$D[c][15:0] = (D[a][15:0] > D[b][15:0]) ? D[a][15:0] : D[b][15:0]; // \text{unsigned}$

### Status Flags

C	Not set by these instructions.
V	Not set by these instructions.
SV	Not set by these instructions.
AV	Not set by these instructions.
SAV	Not set by these instructions.

### Examples

max.h d3, d1, d2

max.hu d3, d1, d2

### See Also

[MAX.B](#), [MAX.BU](#), [MIN.B](#), [MIN.BU](#), [MIN.H](#), [MIN.HU](#)

## MFCR

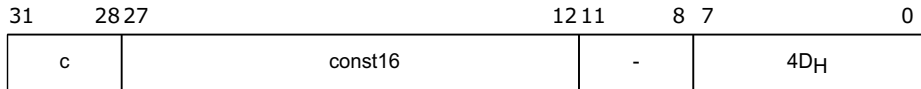
### Move From Core Register

#### Description

Move the contents of the Core Special Function Register (CSFR), selected by the value const16, to data register D[c]. The CSFR address is a const16 byte offset from the CSFR base address. It must be word-aligned (the least-significant two bits equal zero). Nonaligned addresses have an undefined effect.

MFCR can be executed on any privilege level. This instruction may not be used to access GPRs. Attempting to access a GPR with this instruction will return an undefined value.

#### MFCR                      D[c], const16 (RLC)



D[c] = CR[const16];

#### Status Flags

C	Read by the instruction but not changed.
V	Read by the instruction but not changed.
SV	Read by the instruction but not changed.
AV	Read by the instruction but not changed.
SAV	Read by the instruction but not changed.

#### Examples

```
mfcr    d3, fe04H
```

#### See Also

[MTCR](#)

See Also Addendum for TriCore Arch Manual, Vol2, V1.3.8, pages 493/494

## MIN

Minimum Value

## MIN.U

Minimum Value Unsigned

### Description

If the contents of data register D[a] are less than the contents of either data register D[b] (instruction format RR) or const9 (instruction format RC), then put the contents of D[a] in data register D[c]; otherwise put the contents of either D[b] (format RR) or const9 (format RC) in to D[c]. The operands are treated as either signed (MIN) or unsigned (MIN.U) 32-bit integers.

### MIN D[c], D[a], const9 (RC)

31	28 27	21 20	12 11	8 7	0
c	18 <sub>H</sub>	const9	a	8B <sub>H</sub>	

$D[c] = (D[a] < \text{sign\_ext}(\text{const9})) ? D[a] : \text{sign\_ext}(\text{const9});$

### MIN D[c], D[a], D[b] (RR)

31	28 27	20 19	18 17	16 15	12 11	8 7	0
c	18 <sub>H</sub>	-	-	b	a	0B <sub>H</sub>	

$D[c] = (D[a] < D[b]) ? D[a] : D[b];$

### MIN.U D[c], D[a], const9 (RC)

31	28 27	21 20	12 11	8 7	0
c	19 <sub>H</sub>	const9	a	8B <sub>H</sub>	

$D[c] = (D[a] < \text{zero\_ext}(\text{const9})) ? D[a] : \text{zero\_ext}(\text{const9}); // \text{unsigned}$

### MIN.U D[c], D[a], D[b] (RR)

31	28 27	20 19	18 17	16 15	12 11	8 7	0
c	19 <sub>H</sub>	-	-	b	a	0B <sub>H</sub>	

$D[c] = (D[a] < D[b]) ? D[a] : D[b]; // \text{unsigned}$

### Status Flags

C	Not set by these instructions.
---	--------------------------------



Instruction Set

V	Not set by these instructions.
SV	Not set by these instructions.
AV	Not set by these instructions.
SAV	Not set by these instructions.

**Examples**

```
min d3, d1, d2
min d3, d1, #126
min.u d3, d1, d2
min.u d3, d1, #126
```

**See Also**

**MAX, MAX.U**

See Also Addendum for TriCore Arch Manual, Vol2, V1.3.8, pages 493/494

## MIN.B

Minimum Value Packed Byte

## MIN.BU

Minimum Value Packed Byte Unsigned

### Description

Compute the minimum value of the corresponding bytes in D[a] and D[b] and put each result in the corresponding byte of D[c]. The operands are treated as either signed (MIN.B) or unsigned (MIN.BU), 8-bit integers.

### MIN.B D[c], D[a], D[b] (RR)

31	28 27	20 19 18 17 16 15	12 11	8 7	0
c	58 <sub>H</sub>	-	-	b	a
					0B <sub>H</sub>

$D[c][31:24] = (D[a][31:24] < D[b][31:24]) ? D[a][31:24] : D[b][31:24];$

$D[c][23:16] = (D[a][23:16] < D[b][23:16]) ? D[a][23:16] : D[b][23:16];$

$D[c][15:8] = (D[a][15:8] < D[b][15:8]) ? D[a][15:8] : D[b][15:8];$

$D[c][7:0] = (D[a][7:0] < D[b][7:0]) ? D[a][7:0] : D[b][7:0];$

### MIN.BU D[c], D[a], D[b] (RR)

31	28 27	20 19 18 17 16 15	12 11	8 7	0
c	59 <sub>H</sub>	-	-	b	a
					0B <sub>H</sub>

$D[c][31:24] = (D[a][31:24] < D[b][31:24]) ? D[a][31:24] : D[b][31:24]; // \text{unsigned}$

$D[c][23:16] = (D[a][23:16] < D[b][23:16]) ? D[a][23:16] : D[b][23:16]; // \text{unsigned}$

$D[c][15:8] = (D[a][15:8] < D[b][15:8]) ? D[a][15:8] : D[b][15:8]; // \text{unsigned}$

$D[c][7:0] = (D[a][7:0] < D[b][7:0]) ? D[a][7:0] : D[b][7:0]; // \text{unsigned}$

### Status Flags

C	Not set by these instructions.
V	Not set by these instructions.
SV	Not set by these instructions.
AV	Not set by these instructions.
SAV	Not set by these instructions.

**Examples**

min.b     d3, d1, d2  
min.bu    d3, d1, d2

**See Also**

[MAX.B](#), [MAX.BU](#), [MAX.H](#), [MAX.HU](#), [MIN.H](#), [MIN.HU](#)

See Also Addendum for TriCore Arch Manual, Vol2, V1.3.8, pages 493/494

## MIN.H

Minimum Value Packed Half-word

## MIN.HU

Minimum Value Packed Half-word Unsigned

### Description

Compute the minimum value of the corresponding half-words in D[a] and D[b] and put each result in the corresponding half-word of D[c]. The operands are treated as either signed (MIN.H) or unsigned (MIN.HU), 16-bit integers.

### MIN.H D[c], D[a], D[b] (RR)

31	28 27	20 19 18 17 16 15	12 11	8 7	0	
c	78 <sub>H</sub>	-	-	b	a	0B <sub>H</sub>

$D[c][31:16] = (D[a][31:16] < D[b][31:16]) ? D[a][31:16] : D[b][31:16];$

$D[c][15:0] = (D[a][15:0] < D[b][15:0]) ? D[a][15:0] : D[b][15:0];$

### MIN.HU D[c], D[a], D[b] (RR)

31	28 27	20 19 18 17 16 15	12 11	8 7	0	
c	79 <sub>H</sub>	-	-	b	a	0B <sub>H</sub>

$D[c][31:16] = (D[a][31:16] < D[b][31:16]) ? D[a][31:16] : D[b][31:16]; // \text{unsigned}$

$D[c][15:0] = (D[a][15:0] < D[b][15:0]) ? D[a][15:0] : D[b][15:0]; // \text{unsigned}$

### Status Flags

C	Not set by these instructions.
V	Not set by these instructions.
SV	Not set by these instructions.
AV	Not set by these instructions.
SAV	Not set by these instructions.

### Examples

min.h d3, d1, d2

min.hu d3, d1, d2

### See Also

[MAX.B](#), [MAX.BU](#), [MAX.H](#), [MAX.HU](#), [MIN.B](#), [MIN.BU](#)

## MOV

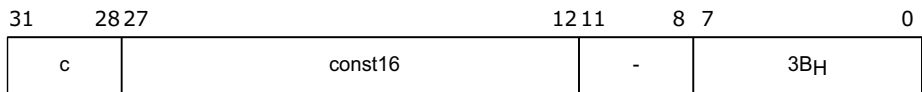
### Move

#### Description

Move the contents of either data register D[b] (instruction format RR) or const16 (instruction format RLC), to data register D[c]. The value const16 is sign-extended to 32-bits before it is moved.

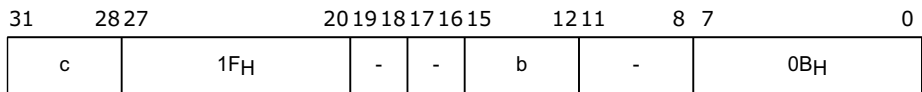
Move the contents of either data register D[b] (instruction format SRR), const4 (instruction format SRC) or const8 (instruction format SC) to either data register D[a] (formats SRR, SRC) or D[15] (format SC). The value const4 is sign-extended before it is moved. The value const8 is zero-extended before it is moved.

#### MOV D[c], const16 (RLC)



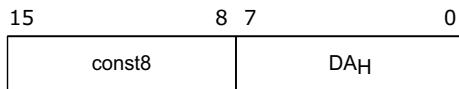
D[c] = sign\_ext(const16);

#### MOV D[c], D[b] (RR)



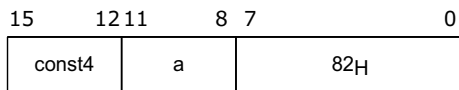
D[c] = D[b];

#### MOV D[15], const8 (SC)



D[15] = zero\_ext(const8);

#### MOV D[a], const4 (SRC)



D[a] = sign\_ext(const4);

**MOV**                      **D[a], D[b] (SRR)**

15      12 11              8 7                      0



D[a] = D[b];

**Status Flags**

C	Not set by this instruction.
V	Not set by this instruction.
SV	Not set by this instruction.
AV	Not set by this instruction.
SAV	Not set by this instruction.

**Examples**

```
mov d3, d1
mov d3, #-30000
```

```
mov d1, d2
mov d1, #6
mov d15, #126
```

**See Also**

[MAX](#), [MAX.U](#), [MOV.U](#), [MOVH](#)

See Also Addendum for TriCore Arch Manual, Vol2, V1.3.8, pages 493/494

## MOV.A

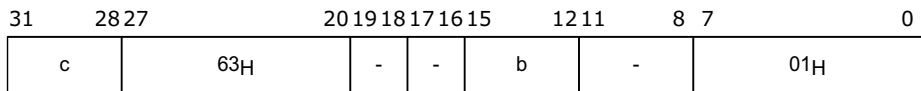
### Move Value to Address Register

#### Description

Move the contents of data register D[b] to address register A[c].

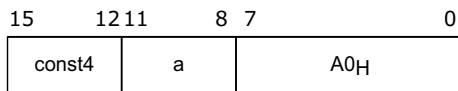
Move the contents of either data register D[b] (format SRR) or const4 (format SRC) to address register A[a]. The value const4 is zero-extended before it is moved.

#### MOV.A                      A[c], D[b] (RR)



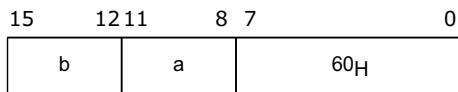
A[c] = D[b];

#### MOV.A                      A[a], const4 (SRC)



A[a] = zero\_ext(const4);

#### MOV.A                      A[a], D[b] (SRR)



A[a] = D[b];

#### Status Flags

C	Not set by this instruction.
V	Not set by this instruction.
SV	Not set by this instruction.
AV	Not set by this instruction.
SAV	Not set by this instruction.

#### Examples

```
mov.a    a3, d1
```

```
mov.a    a4, d2  
mov.a    a4, 7
```

**See Also**

[LEA](#), [MOV.AA](#), [MOV.D](#), [MOVH.A](#)

See Also Addendum for TriCore Arch Manual, Vol2, V1.3.8, pages 493/494



## MOV.AA

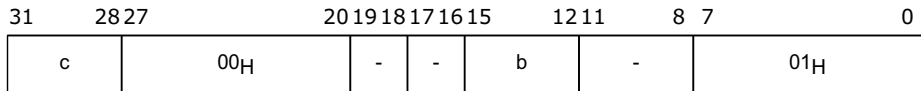
### Move Address from Address Register

#### Description

Move the contents of address register A[b] to address register A[c].

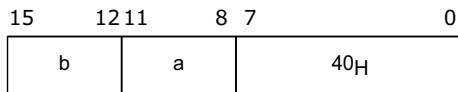
Move the contents of address register A[b] to address register A[a].

#### MOV.AA                    A[c], A[b] (RR)



A[c] = A[b];

#### MOV.AA                    A[a], A[b] (SRR)



A[a] = A[b];

#### Status Flags

C	Not set by this instruction.
V	Not set by this instruction.
SV	Not set by this instruction.
AV	Not set by this instruction.
SAV	Not set by this instruction.

#### Examples

mov.aa    a3, a4

mov.aa    a4, a2

#### See Also

[LEA](#), [MOVH.A](#), [MOV.D](#), [MOVH.A](#)

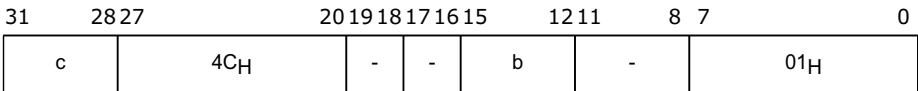
## MOV.D Move Address to Data Register

### Description

Move the contents of address register A[b] to data register D[c].

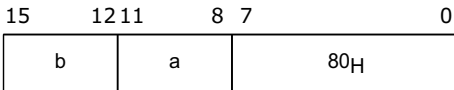
Move the contents of address register A[b] to data register D[a].

### MOV.D D[c], A[b] (RR)



D[c] = A[b];

### MOV.D D[a], A[b] (SRR)



D[a] = A[b];

### Status Flags

C	Not set by this instruction.
V	Not set by this instruction.
SV	Not set by this instruction.
AV	Not set by this instruction.
SAV	Not set by this instruction.

### Examples

```
mov.d d3, a4
```

```
mov.d d1, a2
```

### See Also

[LEA](#), [MOV.A](#), [MOV.AA](#), [MOVH.A](#)

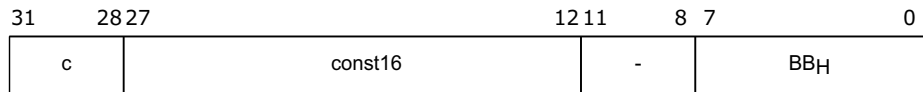
## MOV.U

### Move Unsigned

#### Description

Move the zero-extended value const16 to data register D[c].

#### MOV.U                      D[c], const16 (RLC)



D[c] = zero\_ext(const16);

#### Status Flags

C	Not set by this instruction.
V	Not set by this instruction.
SV	Not set by this instruction.
AV	Not set by this instruction.
SAV	Not set by this instruction.

#### Examples

```
mov.u   d3, #526
```

#### See Also

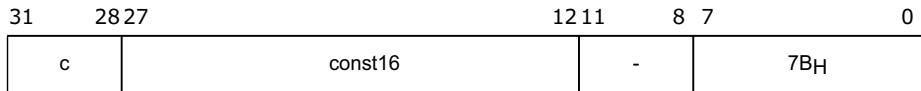
[MOV](#), [MOVH](#)

## MOVH Move High

### Description

Move the value const16 to the most-significant half-word of data register D[c] and set the least-significant 16-bits to zero.

### MOVH                      D[c], const16 (RLC)



D[c] = {const16, 16'h0000};

### Status Flags

C	Not set by this instruction.
V	Not set by this instruction.
SV	Not set by this instruction.
AV	Not set by this instruction.
SAV	Not set by this instruction.

### Examples

```
movh    d3, #526
```

### See Also

[MOV](#), [MOV.U](#)

See Also Addendum for TriCore Arch Manual, Vol2, V1.3.8, pages 493/494

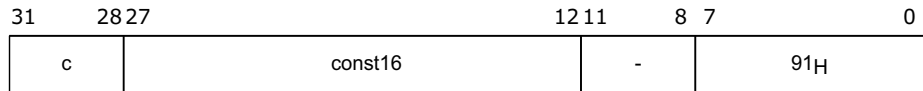
## MOVH.A

### Move High to Address

#### Description

Move the value const16 to the most-significant half-word of address register A[c] and set the least-significant 16-bits to zero.

#### MOVH.A                      A[c], const16 (RLC)



A[c] = {const16, 16'h0000};

#### Status Flags

C	Not set by this instruction.
V	Not set by this instruction.
SV	Not set by this instruction.
AV	Not set by this instruction.
SAV	Not set by this instruction.

#### Examples

```
movh.a    a3, #526
```

#### See Also

[LEA](#), [MOV.A](#), [MOV.AA](#), [MOV.D](#)

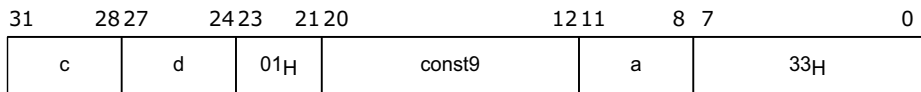
**MSUB**  
Multiply-Subtract  
**MSUBS**  
Multiply-Subtract, Saturated

**Description**

Multiply two 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. The value const9 is sign-extended before the multiplication is performed. The MSUBS result is saturated on overflow.

**MSUB**                      **D[c], D[d], D[a], const9 (RCR)**

32 - (32 \* K9) --> 32 signed

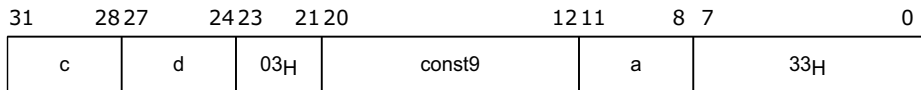


result = D[d] - (D[a] \* sign\_ext(const9));

D[c] = result[31:0];

**MSUB**                      **E[c], E[d], D[a], const9 (RCR)**

64 - (32 \* K9) --> 64 signed

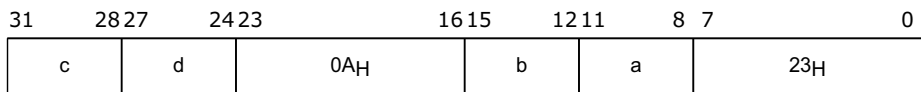


result = E[d] - (D[a] \* sign\_ext(const9));

E[c] = result[63:0];

**MSUB**                      **D[c], D[d], D[a], D[b] (RRR2)**

32 - (32 \* 32) --> 32 signed



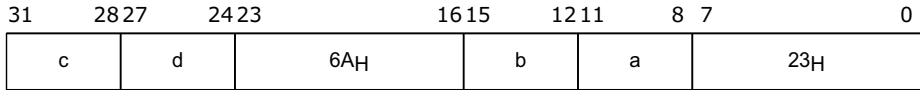
result = D[d] - (D[a] \* D[b]);

D[c] = result[31:0];

**MSUB**                      **E[c], E[d], D[a], D[b] (RRR2)**

64 - (32 \* 32) --> 64 signed

See Also Addendum for TriCore Arch Manual, Vol2, V1.3.8, pages 493/494

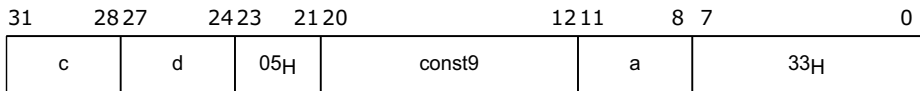


result = E[d] - (D[a] \* D[b]);

E[c] = result[63:0];

**MSUBS**                      **D[c], D[d], D[a], const9 (RCR)**

32 - (32 \* K9) --> 32 signed saturated

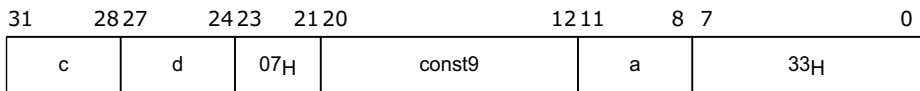


result = D[d] - (D[a] \* sign\_ext(const9));

D[c] = ssov(result, 32);

**MSUBS**                      **E[c], E[d], D[a], const9 (RCR)**

64 - (32 \* K9) --> 64 signed saturated

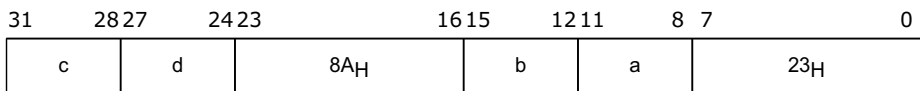


result = E[d] - (D[a] \* sign\_ext(const9));

E[c] = ssov(result, 64);

**MSUBS**                      **D[c], D[d], D[a], D[b] (RRR2)**

32 - (32 \* 32) --> 32 signed saturated

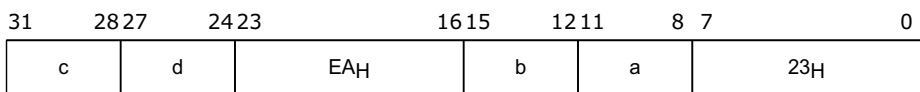


result = D[d] - (D[a] \* D[b]);

D[c] = ssov(result, 32);

**MSUBS**                      **E[c], E[d], D[a], D[b] (RRR2)**

64 - (32 \* 32) --> 64 signed saturated



result = E[d] - (D[a] \* D[b]);

See Also Addendum for TriCore Arch Manual, Vol2, V1.3.8, pages 493/494

E[c] = ssov(result, 64)

### Status Flags

C	Not set by these instructions.
V	32-bit result: overflow = (result > 7FFFFFFF <sub>H</sub> ) OR (result < -80000000 <sub>H</sub> ); if (overflow) then PSW.V = 1 else PSW.V = 0; 64-bit result: overflow = (result > 7FFFFFFFFFFFFFFF <sub>H</sub> ) OR (result < -8000000000000000 <sub>H</sub> ); if (overflow) then PSW.V = 1 else PSW.V = 0;
SV	if (overflow) then PSW.SV = 1 else PSW.SV = PSW.SV;
AV	32-bit result: advanced_overflow = result[31] ^ result[30]; if (advanced_overflow) then PSW.AV = 1 else PSW.AV = 0; 64-bit result: advanced_overflow = result[63] ^ result[62]; if (advanced_overflow) then PSW.AV = 1 else PSW.AV = 0;
SAV	if (advanced_overflow) then PSW.SAV = 1 else PSW.SAV = PSW.SAV;

### Examples

```

msub  d0, d1, d2, d3
msub  d0, d1, d2, #7
msub  e0, e2, d6, d11
msub  e0, e0, d3, #80
msubs d5, d1, d2, d2
msubs d1, d1, d2, #7
msubs e0, e2, d6, d11
msubs e8, e4, d3, #80
    
```

### See Also

[MUL](#)



**MSUB.H**  
Packed Multiply-Subtract Q Format  
**MSUBS.H**  
Packed Multiply-Subtract Q Format, Saturated

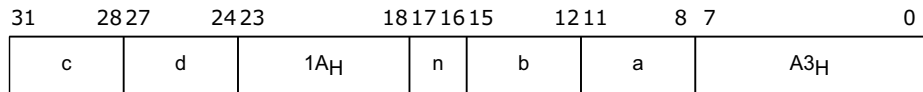
**Description**

Multiply two signed 16-bit (half-word) values. Subtract the product (left justified if n == 1) from a signed 32-bit value and put the result into a 32-bit register. There are four cases of half-word multiplication.

Note that n should only take the values 0 or 1, any other value returns an undefined result. If (n == 1) then  $8000_H * 8000_H = 7FFFFFFF_H$  (for signed 16-bit \* 16-bit multiplications only).

**MSUB.H**                      **E[c], E[d], D[a], D[b] LL, n (RRR1)**

32||32 -||- (16U \* 16L || 16L \* 16L) --> 32||32



sc1 = (D[a][31:16] == 8000<sub>H</sub>) AND (D[b][15:0] == 8000<sub>H</sub>) AND (n == 1);

sc0 = (D[a][15:0] == 8000<sub>H</sub>) AND (D[b][15:0] == 8000<sub>H</sub>) AND (n == 1);

mul\_res1 = sc1 ? 7FFFFFFF<sub>H</sub> : ((D[a][31:16] \* D[b][15:0]) << n);

mul\_res0 = sc0 ? 7FFFFFFF<sub>H</sub> : ((D[a][15:0] \* D[b][15:0]) << n);

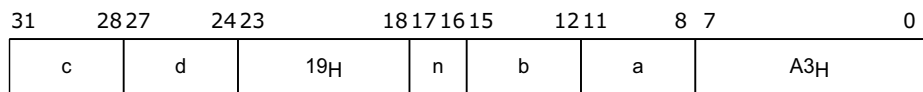
result\_word1 = E[d][63:32] - mul\_res1;

result\_word0 = E[d][31:0] - mul\_res0;

E[c] = {result\_word1[31:0], result\_word0[31:0]}; // Packed fraction

**MSUB.H**                      **E[c], E[d], D[a], D[b] LU, n (RRR1)**

32||32 -||- (16U \* 16L || 16L \* 16U) --> 32||32



sc1 = (D[a][31:16] == 8000<sub>H</sub>) AND (D[b][15:0] == 8000<sub>H</sub>) AND (n == 1);

sc0 = (D[a][15:0] == 8000<sub>H</sub>) AND (D[b][31:16] == 8000<sub>H</sub>) AND (n == 1);

mul\_res1 = sc1 ? 7FFFFFFF<sub>H</sub> : ((D[a][31:16] \* D[b][15:0]) << n);

mul\_res0 = sc0 ? 7FFFFFFF<sub>H</sub> : ((D[a][15:0] \* D[b][31:16]) << n);

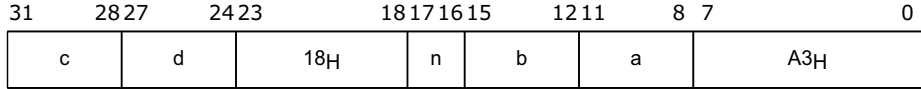
result\_word1 = E[d][63:32] - mul\_res1;

See Also Addendum for TriCore Arch Manual, Vol2, V1.3.8, pages 493/494

```
result_word0 = E[d][31:0] - mul_res0;
E[c] = {result_word1[31:0], result_word0[31:0]}; // Packed fraction
```

**MSUB.H**                    **E[c], E[d], D[a], D[b] UL, n (RRR1)**

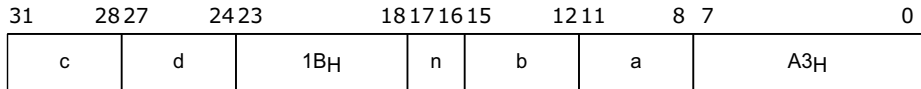
32||32 -|- (16U \* 16U || 16L \* 16L) --> 32||32



```
sc1 = (D[a][31:16] == 8000H) AND (D[b][31:16] == 8000H) AND (n == 1);
sc0 = (D[a][15:0] == 8000H) AND (D[b][15:0] == 8000H) AND (n == 1);
mul_res1 = sc1 ? 7FFFFFFFH : ((D[a][31:16] * D[b][31:16]) << n);
mul_res0 = sc0 ? 7FFFFFFFH : ((D[a][15:0] * D[b][15:0]) << n);
result_word1 = E[d][63:32] - mul_res1;
result_word0 = E[d][31:0] - mul_res0;
E[c] = {result_word1[31:0], result_word0[31:0]}; // Packed fraction
```

**MSUB.H**                    **E[c], E[d], D[a], D[b] UU, n (RRR1)**

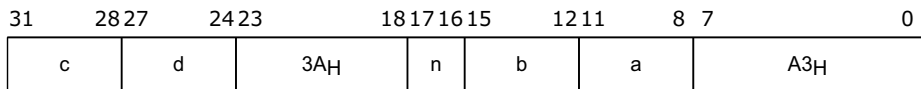
32||32 -|- (16L \* 16U || 16U \* 16U) --> 32||32



```
sc1 = (D[a][15:0] == 8000H) AND (D[b][31:16] == 8000H) AND (n == 1);
sc0 = (D[a][31:16] == 8000H) AND (D[b][31:16] == 8000H) AND (n == 1);
mul_res1 = sc1 ? 7FFFFFFFH : ((D[a][15:0] * D[b][31:16]) << n);
mul_res0 = sc0 ? 7FFFFFFFH : ((D[a][31:16] * D[b][31:16]) << n);
result_word1 = E[d][63:32] - mul_res1;
result_word0 = E[d][31:0] - mul_res0;
E[c] = {result_word1[31:0], result_word0[31:0]}; // Packed fraction
```

**MSUBS.H**                    **E[c], E[d], D[a], D[b] LL, n (RRR1)**

32||32 -|- (16U \* 16L || 16L \* 16L) --> 32||32 saturated

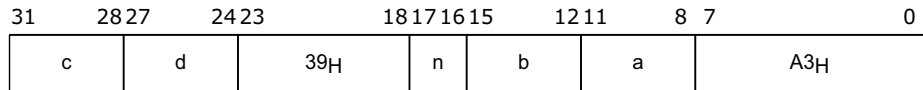


```
sc1 = (D[a][31:16] == 8000H) AND (D[b][15:0] == 8000H) AND (n == 1);
```

```
sc0 = (D[a][15:0] == 8000H) AND (D[b][15:0] == 8000H) AND (n == 1);
mul_res1 = sc1 ? 7FFFFFFFH : ((D[a][31:16] * D[b][15:0]) << n);
mul_res0 = sc0 ? 7FFFFFFFH : ((D[a][15:0] * D[b][15:0]) << n);
result_word1 = E[d][63:32] - mul_res1;
result_word0 = E[d][31:0] - mul_res0;
E[c] = {ssov(result_word1, 32), ssov(result_word0, 32)}; // Packed fraction
```

**MSUBS.H E[c], E[d], D[a], D[b] LU, n (RRR1)**

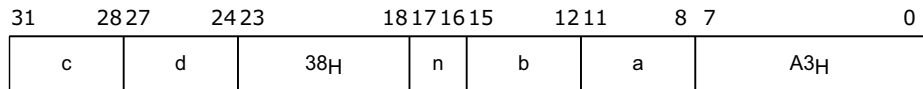
32||32 -||- (16U \* 16L || 16L \* 16U) --> 32||32 saturated



```
sc1 = (D[a][31:16] == 8000H) AND (D[b][15:0] == 8000H) AND (n == 1);
sc0 = (D[a][15:0] == 8000H) AND (D[b][31:16] == 8000H) AND (n == 1);
mul_res1 = sc1 ? 7FFFFFFFH : ((D[a][31:16] * D[b][15:0]) << n);
mul_res0 = sc0 ? 7FFFFFFFH : ((D[a][15:0] * D[b][31:16]) << n);
result_word1 = E[d][63:32] - mul_res1;
result_word0 = E[d][31:0] - mul_res0;
E[c] = {ssov(result_word1, 32), ssov(result_word0, 32)}; // Packed fraction
```

**MSUBS.H E[c], E[d], D[a], D[b] UL, n (RRR1)**

32||32 -||- (16U \* 16U || 16L \* 16L) --> 32||32 saturated

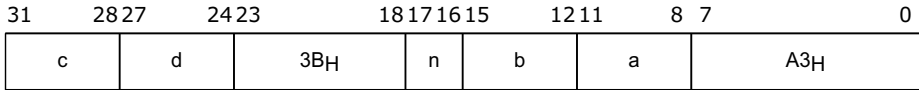


```
sc1 = (D[a][31:16] == 8000H) AND (D[b][31:16] == 8000H) AND (n == 1);
sc0 = (D[a][15:0] == 8000H) AND (D[b][15:0] == 8000H) AND (n == 1);
mul_res1 = sc1 ? 7FFFFFFFH : ((D[a][31:16] * D[b][31:16]) << n);
mul_res0 = sc0 ? 7FFFFFFFH : ((D[a][15:0] * D[b][15:0]) << n);
result_word1 = E[d][63:32] - mul_res1;
result_word0 = E[d][31:0] - mul_res0;
E[c] = {ssov(result_word1, 32), ssov(result_word0, 32)}; // Packed fraction
```

**MSUBS.H E[c], E[d], D[a], D[b] UU, n (RRR1)**

32||32 -||- (16L \* 16U || 16U \* 16U) --> 32||32 saturated

**Instruction Set**



```

sc1 = (D[a][15:0] == 8000H) AND (D[b][31:16] == 8000H) AND (n == 1);
sc0 = (D[a][31:16] == 8000H) AND (D[b][31:16] == 8000H) AND (n == 1);
mul_res1 = sc1 ? 7FFFFFFFH : ((D[a][15:0] * D[b][31:16]) << n);
mul_res0 = sc0 ? 7FFFFFFFH : ((D[a][31:16] * D[b][31:16]) << n);
result_word1 = E[d][63:32] - mul_res1;
result_word0 = E[d][31:0] - mul_res0;
E[c] = {ssov(result_word1, 32), ssov(result_word0, 32)}; // Packed fraction
  
```

**Status Flags**

C	Not set by these instructions.
V	$ov\_word1 = (result\_word1 > 7FFFFFFF_H) \text{ OR } (result\_word1 < -80000000_H);$ $ov\_word0 = (result\_word0 > 7FFFFFFF_H) \text{ OR } (result\_word0 < -80000000_H);$ $overflow = ov\_word1 \text{ OR } ov\_word0;$ $\text{if } (overflow) \text{ then PSW.V} = 1 \text{ else PSW.V} = 0;$
SV	$\text{if } (overflow) \text{ then PSW.SV} = 1 \text{ else PSW.SV} = \text{PSW.SV};$
AV	$aov\_word1 = result\_word1[31] \wedge result\_word1[30];$ $aov\_word0 = result\_word0[31] \wedge result\_word0[30];$ $advanced\_overflow = aov\_word1 \text{ OR } aov\_word0;$ $\text{if } (advanced\_overflow) \text{ then PSW.AV} = 1 \text{ else PSW.AV} = 0;$
SAV	$\text{if } (advanced\_overflow) \text{ then PSW.SAV} = 1 \text{ else PSW.SAV} = \text{PSW.SAV};$

**Examples**

-

**See Also**

-

See Also Addendum for TriCore Arch Manual, Vol2, V1.3.8, pages 493/494

## MSUB.Q

### Multiply-Subtract Q Format

## MSUBS.Q

### Multiply-Subtract Q Format, Saturated

#### Description

Multiply two 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.

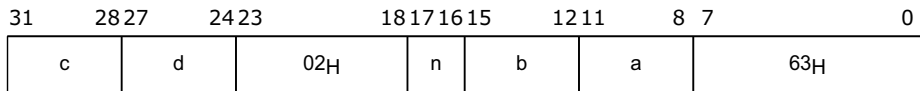
There are eight cases of 16\*16 operations, eight cases of 16\*32 operations and four cases of 32\*32 operations.

The MSUBS.Q result is saturated on overflow.

Note that  $n$  should only take the values 0 or 1, any other value returns an undefined result. If ( $n == 1$ ) then  $8000_H * 8000_H = 7FFFFFFF_H$  (for signed 16-bit \* 16-bit multiplications only).

#### MSUB.Q                      D[c], D[d], D[a], D[b], n (RRR1)

32 - (32 \* 32)Up --> 32

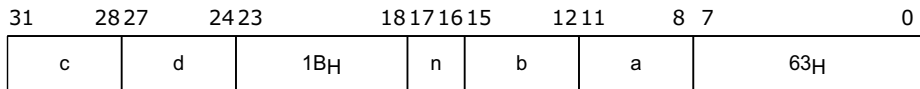


result = D[d] - (((D[a] \* D[b]) << n) >> 32);

D[c] = result[31:0]; // Fraction

#### MSUB.Q                      E[c], E[d], D[a], D[b], n (RRR1)

64 - (32 \* 32) --> 64

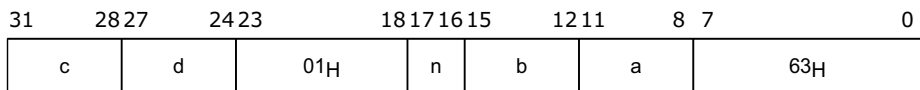


result = E[d] - ((D[a] \* D[b]) << n);

E[c] = result[63:0]; // Multi-precision fraction

#### MSUB.Q                      D[c], D[d], D[a], D[b] L, n (RRR1)

32 - (32 \* 16L)Up --> 32

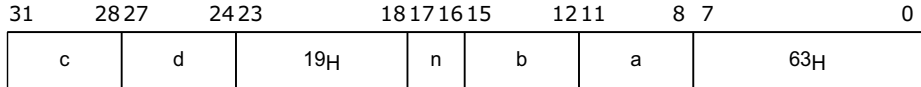


result = D[d] - (((D[a] \* D[b][15:0]) << n) >> 16);

D[c] = result[31:0]; // Fraction

**MSUB.Q**                    **E[c], E[d], D[a], D[b] L, n (RRR1)**

64 - (32 \* 16L) --> 64

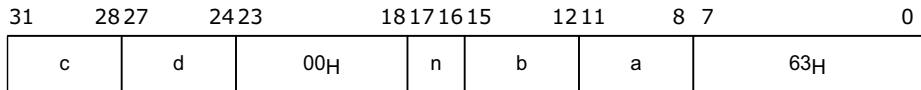


result = E[d] - ((D[a] \* D[b][15:0]) << n);

E[c] = result[63:0]; // Multi-precision accumulator

**MSUB.Q**                    **D[c], D[d], D[a], D[b] U, n (RRR1)**

32 - (32 \* 16U)<sub>Up</sub> --> 32

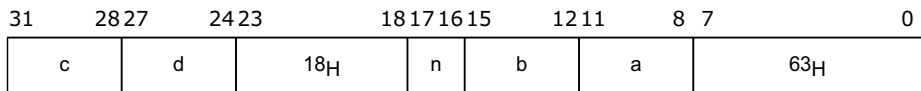


result = D[d] - (((D[a] \* D[b][31:16]) << n) >> 16);

D[c] = result[31:0]; // Fraction

**MSUB.Q**                    **E[c], E[d], D[a], D[b] U, n (RRR1)**

64 - (32 \* 16U) --> 64

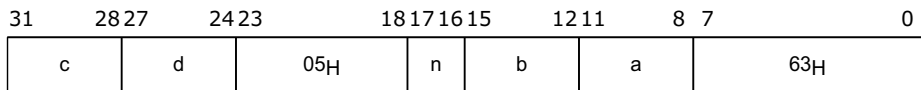


result = E[d] - ((D[a] \* D[b][31:16]) << n);

E[c] = result[63:0]; // Multi-precision accumulator

**MSUB.Q**                    **D[c], D[d], D[a] L, D[b] L, n (RRR1)**

32 - (16L \* 16L) --> 32



sc = (D[a][15:0] == 8000<sub>H</sub>) AND (D[b][15:0] == 8000<sub>H</sub>) AND (n == 1);

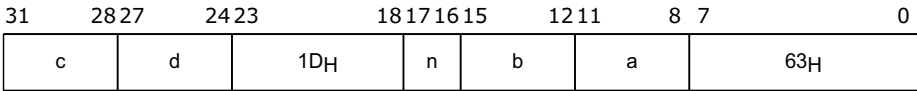
mul\_res = sc ? 7FFFFFFF<sub>H</sub> : ((D[a][15:0] \* D[b][15:0]) << n);

result = D[d] - mul\_res;

D[c] = result[31:0]; // Fraction

**MSUB.Q**                    **E[c], E[d], D[a] L, D[b] L, n (RRR1)**

64 - (16L \* 16L) --> 64



sc = (D[a][15:0] == 8000<sub>H</sub>) AND (D[b][15:0] == 8000<sub>H</sub>) AND (n == 1);

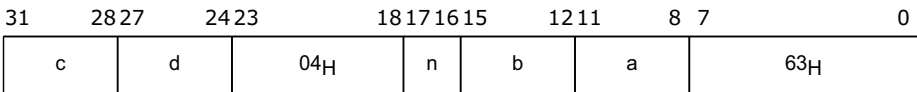
mul\_res = sc ? 7FFFFFFF<sub>H</sub> : ((D[a][15:0] \* D[b][15:0]) << n);

result = E[d] - (mul\_res << 16);

E[c] = result[63:0]; // Multi-precision accumulator

**MSUB.Q**                    **D[c], D[d], D[a] U, D[b] U, n (RRR1)**

32 - (16U \* 16U) --> 32



sc = (D[a][31:16] == 8000<sub>H</sub>) AND (D[b][31:16] == 8000<sub>H</sub>) AND (n == 1);

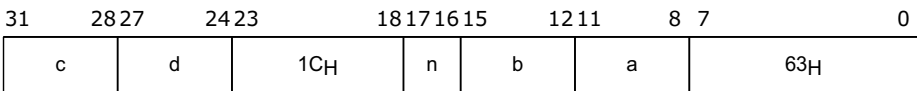
mul\_res = sc ? 7FFFFFFF<sub>H</sub> : ((D[a][31:16] \* D[b][31:16]) << n);

result = D[d] - mul\_res;

D[c] = result[31:0]; // Fraction

**MSUB.Q**                    **E[c], E[d], D[a] U, D[b] U, n (RRR1)**

64 - (16U \* 16U) --> 64



sc = (D[a][31:16] == 8000<sub>H</sub>) AND (D[b][31:16] == 8000<sub>H</sub>) AND (n == 1);

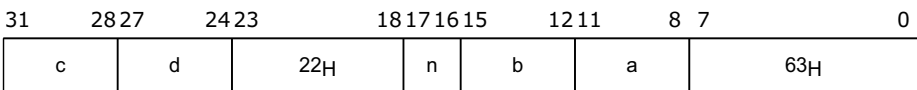
mul\_res = sc ? 7FFFFFFF<sub>H</sub> : ((D[a][31:16] \* D[b][31:16]) << n);

result = E[d] - (mul\_res << 16);

E[c] = result[63:0]; // Multi-precision accumulator

**MSUBS.Q**                    **D[c], D[d], D[a], D[b], n (RRR1)**

32 - (32 \* 32)Up --> 32 saturated

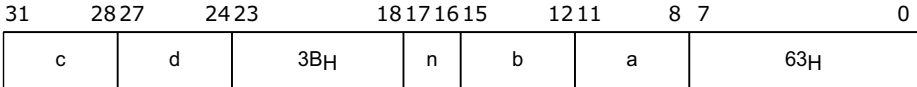


result = D[d] - (((D[a] \* D[b]) << n) >> 32);

D[c] = ssov(result, 32); // Fraction

**MSUBS.Q**            **E[c], E[d], D[a], D[b], n (RRR1)**

64 - (32 \* 32) --> 64 saturated

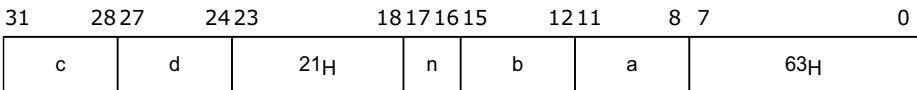


result = E[d] - ((D[a] \* D[b]) << n);

E[c] = ssov(result, 64); // Multi-precision fraction

**MSUBS.Q**            **D[c], D[d], D[a], D[b] L, n (RRR1)**

32 - (32 \* 16L)Up --> 32 saturated

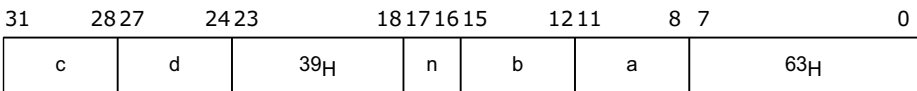


result = D[d] - (((D[a] \* D[b][15:0]) << n) >> 16);

D[c] = ssov(result, 32); // Fraction

**MSUBS.Q**            **E[c], E[d], D[a], D[b] L, n (RRR1)**

64 - (32 \* 16L) --> 64 saturated

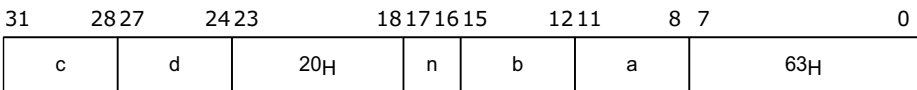


result = E[d] - ((D[a] \* D[b][15:0]) << n);

E[c] = ssov(result, 64); // Multi-precision accumulator

**MSUBS.Q**            **D[c], D[d], D[a], D[b] U, n (RRR1)**

32 - (32 \* 16U)Up --> 32 saturated



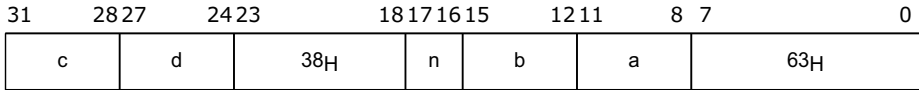
result = D[d] - (((D[a] \* D[b][31:16]) << n) >> 16);

D[c] = ssov(result, 32); // Fraction

**MSUBS.Q**            **E[c], E[d], D[a], D[b] U, n (RRR1)**

64 - (32 \* 16U) --> 64 saturated



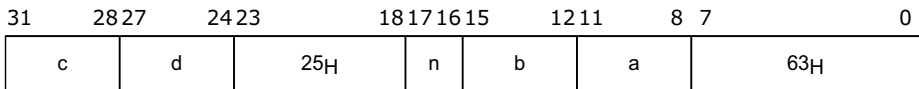


result = E[d] - ((D[a] \* D[b][31:16]) << n);

E[c] = ssov(result, 64); // Multi-precision accumulator

**MSUBS.Q**                    **D[c], D[d], D[a] L, D[b] L, n (RRR1)**

32 - (16L \* 16L) --> 32 saturated



sc = (D[a][15:0] == 8000<sub>H</sub>) AND (D[b][15:0] == 8000<sub>H</sub>) AND (n == 1);

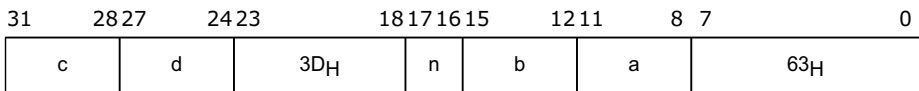
mul\_res = sc ? 7FFFFFFF<sub>H</sub> : ((D[a][15:0] \* D[b][15:0]) << n);

result = D[d] - mul\_res;

D[c] = ssov(result, 32); // Fraction

**MSUBS.Q**                    **E[c], E[d], D[a] L, D[b] L, n (RRR1)**

64 - (16L \* 16L) --> 64 saturated



sc = (D[a][15:0] == 8000<sub>H</sub>) AND (D[b][15:0] == 8000<sub>H</sub>) AND (n == 1);

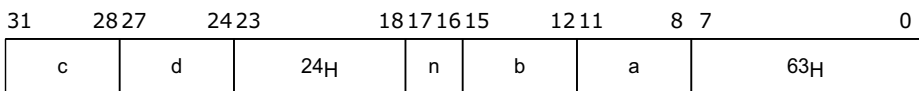
mul\_res = sc ? 7FFFFFFF<sub>H</sub> : ((D[a][15:0] \* D[b][15:0]) << n);

result = E[d] - (mul\_res << 16);

E[c] = ssov(result, 64); // Multi-precision accumulator

**MSUBS.Q**                    **D[c], D[d], D[a] U, D[b] U, n (RRR1)**

32 - (16U \* 16U) --> 32 saturated



sc = (D[a][31:16] == 8000<sub>H</sub>) AND (D[b][31:16] == 8000<sub>H</sub>) AND (n == 1);

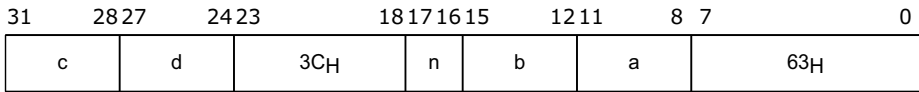
mul\_res = sc ? 7FFFFFFF<sub>H</sub> : ((D[a][31:16] \* D[b][31:16]) << n);

result = D[d] - mul\_res;

D[c] = ssov(result, 32); // Fraction

**MSUBS.Q**            **E[c], E[d], D[a] U, D[b] U, n (RRR1)**

64 - (16U \* 16U) --> 64 saturated



sc = (D[a][31:16] == 8000<sub>H</sub>) AND (D[b][31:16] == 8000<sub>H</sub>) AND (n == 1);

mul\_res = sc ? 7FFFFFFF<sub>H</sub> : ((D[a][31:16] \* D[b][31:16]) << n);

result = E[d] - (mul\_res << 16);

E[c] = ssov(result, 64); // Multi-precision accumulator

**Status Flags**

C	Not set by these instructions.
V	32-bit result: overflow = (result > 7FFFFFFF <sub>H</sub> ) OR (result < -80000000 <sub>H</sub> ) if (overflow) then PSW.V = 1 else PSW.V = 0; 64-bit result: overflow = (result > 7FFFFFFFFFFFFFFF <sub>H</sub> ) OR (result < -8000000000000000 <sub>H</sub> ); if (overflow) then PSW.V = 1 else PSW.V = 0;
SV	if (overflow) then PSW.SV = 1 else PSW.SV = PSW.SV;
AV	32-bit result: advanced_overflow = result[31] ^ result[30]; if (advanced_overflow) then PSW.AV = 1 else PSW.AV = 0; 64-bit result: advanced_overflow = result[63] ^ result[62]; if (advanced_overflow) then PSW.AV = 1 else PSW.AV = 0;
SAV	if (advanced_overflow) then PSW.SAV = 1 else PSW.SAV = PSW.SAV;

**Examples**

```

msub.q    d0, d1, d2, d3, #1
msub.q    d0, d1, d2, d6U, #1
msub.q    d0, d2, d1, d3L, #1
msub.q    d2, d0, d3U, d4U, #1
msub.q    d2, d0, d4L, d4L, #1
msub.q    e2, e2, d3, d7, #1
msub.q    e2, e2, d4, d6U, #1
msub.q    e2, e2, d5, d6L, #1
msub.q    e2, e2, d6U, d7U, #1
    
```

See Also Addendum for TriCore Arch Manual, Vol2, V1.3.8, pages 493/494

```
msub.q    e2, e2, d8L, d0L, #1
msubs.q   d0, d1, d2, d3, #1
msubs.q   d0, d1, d2, d6U, #1
msubs.q   d0, d2, d1, d3L, #1
msubs.q   d2, d0, d3U, d4U, #1
msubs.q   d2, d0, d4L, d4L, #1
msubs.q   e2, e2, d3, d7, #1
msubs.q   e2, e2, d4, d6U, #1
msubs.q   e2, e2, d5, d6L, #1
msubs.q   e2, e2, d6U, d7U, #1
msubs.q   e2, e0, d11L, d4L, #1
```

**See Also**

-

## MSUB.U

Multiply-Subtract Unsigned

## MSUBS.U

Multiply-Subtract Unsigned, Saturated

### Description

Multiply two 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. The value const9 is zero-extended before the multiplication is performed. The MSUBS.U results are saturated on overflow.

### MSUB.U E[c], E[d], D[a], const9 (RCR)

64 - (32 \* K9) --> 64 unsigned

31	28 27	24 23	21 20	12 11	8 7	0
c	d	02H	const9	a	33H	

result = E[d] - (D[a] \* zero\_ext(const9)); // unsigned operators

E[c] = result[63:0];

### MSUB.U E[c], E[d], D[a], D[b] (RRR2)

64 - (32 \* 32) --> 64 unsigned

31	28 27	24 23	16 15	12 11	8 7	0
c	d	68H		b	a	23H

result = E[d] - (D[a] \* D[b]); // unsigned operators

E[c] = result[63:0];

### MSUBS.U D[c], D[d], D[a], const9 (RCR)

32 - (32 \* K9) --> 32 unsigned saturated

31	28 27	24 23	21 20	12 11	8 7	0
c	d	04H	const9	a	33H	

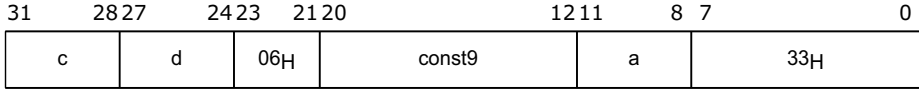
result = D[d] - (D[a] \* zero\_ext(const9)); // unsigned operators

D[c] = suov(result, 32);

### MSUBS.U E[c], E[d], D[a], const9 (RCR)

64 - (32 \* K9) --> 64 unsigned saturated

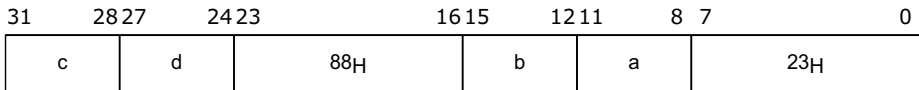
Instruction Set



result = E[d] - (D[a] \* zero\_ext(const9)); // unsigned operators  
E[c] = suov(result, 64);

**MSUBS.U**                    **D[c], D[d], D[a], D[b] (RRR2)**

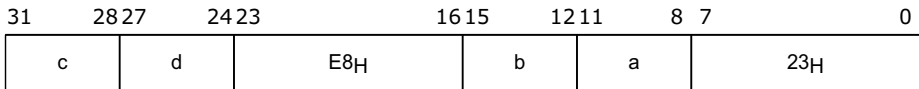
32 - (32 \* 32) --> 32 unsigned saturated



result = D[d] - (D[a] \* D[b]); // unsigned operators  
D[c]= suov(result, 32);

**MSUBS.U**                    **E[c], E[d], D[a], D[b] (RRR2)**

64 - (32 \* 32) --> 64 unsigned saturated



result = E[d] - (D[a] \* D[b]); // unsigned operators  
E[c] = suov(result, 64);

**Status Flags**

C	Not set by these instructions.
V	32-bit result: overflow = (result > FFFFFFFFH) OR (result < 00000000H); if (overflow) then PSW.V = 1 else PSW.V = 0; 64-bit result: overflow = (result > FFFFFFFFFFFFFFFFH) OR (result < 0000000000000000H); if (overflow) then PSW.V = 1 else PSW.V = 0;
SV	if (overflow) then PSW.SV = 1 else PSW.SV = PSW.SV;

See Also Addendum for TriCore Arch Manual, Vol2, V1.3.8, pages 493/494

AV	<p>32-bit result:  <code>advanced_overflow = result[31] ^ result[30];</code>  <code>if (advanced_overflow) then PSW.AV = 1 else PSW.AV = 0;</code>            64-bit result:  <code>advanced_overflow = result[63] ^ result[62];</code>  <code>if (advanced_overflow) then PSW.AV = 1 else PSW.AV = 0;</code></p>
SAV	<p><code>if (advanced_overflow) then PSW.SAV = 1 else PSW.SAV = PSW.SAV;</code></p>

**Examples**

```

msub.u   e0, e2, d6, d11
msub.u   e0, e0, d3, #80
msubs.u  d5, d1, d2, d2
msubs.u  d1, d1, d2, #7
msubs.u  e0, e2, d6, d11
msubs.u  e8, e4, d3, #80
  
```

**See Also**

-

## MSUBAD.H

Packed Multiply-Subtract/Add Q Format

## MSUBADS.H

Packed Multiply-Subtract/Add Q Format, Saturated

### Description

Multiply two signed 16-bit (half-word) 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 four cases of half-word multiplication:

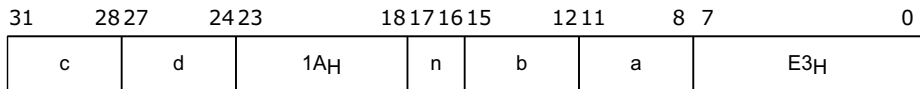
- $16U * 16U, 16L * 16L$
- $16U * 16L, 16L * 16U$
- $16U * 16L, 16L * 16L$
- $16L * 16U, 16U * 16U$

On overflow each MSUBADS.H result is independently saturated.

Note that  $n$  should only take the values 0 or 1, any other value returns an undefined result. If ( $n == 1$ ) then  $8000_H * 8000_H = 7FFFFFFF_H$  (for signed 16-bit \* 16-bit multiplications only).

### MSUBAD.H                      E[c], E[d], D[a], D[b] LL, n (RRR1)

$32||32 -||+ (16U * 16L || 16L * 16L) \rightarrow 32||32$



$sc1 = (D[a][31:16] == 8000_H) \text{ AND } (D[b][15:0] == 8000_H) \text{ AND } (n == 1);$

$sc0 = (D[a][15:0] == 8000_H) \text{ AND } (D[b][15:0] == 8000_H) \text{ AND } (n == 1);$

$mul\_res1 = sc1 ? 7FFFFFFF_H : ((D[a][31:16] * D[b][15:0]) \ll n);$

$mul\_res0 = sc0 ? 7FFFFFFF_H : ((D[a][15:0] * D[b][15:0]) \ll n);$

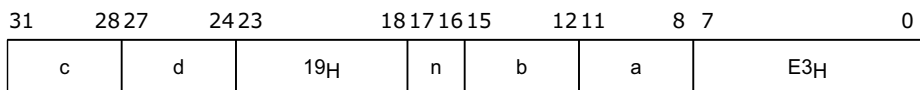
$result\_word1 = E[d][63:32] - mul\_res1;$

$result\_word0 = E[d][31:0] + mul\_res0;$

$E[c] = \{result\_word1[31:0], result\_word0[31:0]\}; // \text{ Packed fraction}$

### MSUBAD.H                      E[c], E[d], D[a], D[b] LU, n (RRR1)

$32||32 -||+ (16U * 16L || 16L * 16U) \rightarrow 32||32$

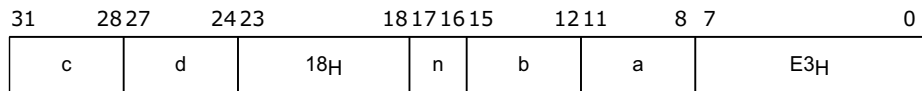


$sc1 = (D[a][31:16] == 8000_H) \text{ AND } (D[b][15:0] == 8000_H) \text{ AND } (n == 1);$

```
sc0 = (D[a][15:0] == 8000H) AND (D[b][31:16] == 8000H) AND (n == 1);
mul_res1 = sc1 ? 7FFFFFFFH : ((D[a][31:16] * D[b][15:0]) << n);
mul_res0 = sc0 ? 7FFFFFFFH : ((D[a][15:0] * D[b][31:16]) << n);
result_word1 = E[d][63:32] - mul_res1;
result_word0 = E[d][31:0] + mul_res0;
E[c] = {result_word1[31:0], result_word0[31:0]}; // Packed fraction
```

**MSUBAD.H**      **E[c], E[d], D[a], D[b] UL, n (RRR1)**

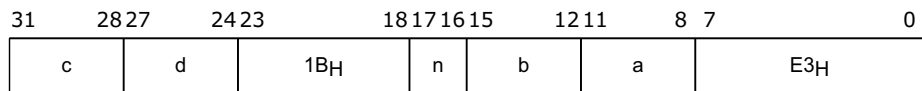
32||32 -||+ (16U \* 16U || 16L \* 16L) --> 32||32



```
sc1 = (D[a][31:16] == 8000H) AND (D[b][31:16] == 8000H) AND (n == 1);
sc0 = (D[a][15:0] == 8000H) AND (D[b][15:0] == 8000H) AND (n == 1);
mul_res1 = sc1 ? 7FFFFFFFH : ((D[a][31:16] * D[b][31:16]) << n);
mul_res0 = sc0 ? 7FFFFFFFH : ((D[a][15:0] * D[b][15:0]) << n);
result_word1 = E[d][63:32] - mul_res1;
result_word0 = E[d][31:0] + mul_res0;
E[c] = {result_word1[31:0], result_word0[31:0]}; // Packed fraction
```

**MSUBAD.H**      **E[c], E[d], D[a], D[b] UU, n (RRR1)**

32||32 -||+ (16L \* 16U || 16U \* 16U) --> 32||32

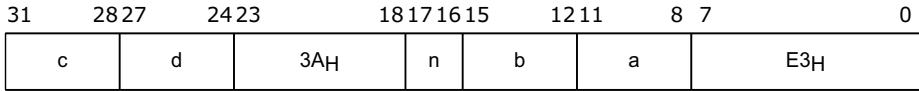


```
sc1 = (D[a][15:0] == 8000H) AND (D[b][31:16] == 8000H) AND (n == 1);
sc0 = (D[a][31:16] == 8000H) AND (D[b][31:16] == 8000H) AND (n == 1);
mul_res1 = sc1 ? 7FFFFFFFH : ((D[a][15:0] * D[b][31:16]) << n);
mul_res0 = sc0 ? 7FFFFFFFH : ((D[a][31:16] * D[b][31:16]) << n);
result_word1 = E[d][63:32] - mul_res1;
result_word0 = E[d][31:0] + mul_res0;
E[c] = {result_word1[31:0], result_word0[31:0]}; // Packed fraction
```

**MSUBADS.H**      **E[c], E[d], D[a], D[b] LL, n (RRR1)**

32||32 -||+ (16U \* 16L || 16L \* 16L) --> 32||32 saturated

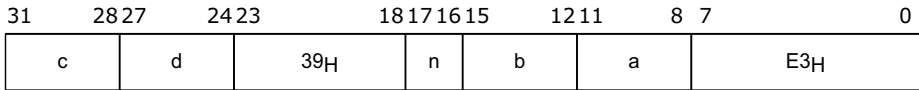




$sc1 = (D[a][31:16] == 8000_H) \text{ AND } (D[b][15:0] == 8000_H) \text{ AND } (n == 1);$   
 $sc0 = (D[a][15:0] == 8000_H) \text{ AND } (D[b][15:0] == 8000_H) \text{ AND } (n == 1);$   
 $mul\_res1 = sc1 ? 7FFFFFFF_H : ((D[a][31:16] * D[b][15:0]) \ll n);$   
 $mul\_res0 = sc0 ? 7FFFFFFF_H : ((D[a][15:0] * D[b][15:0]) \ll n);$   
 $result\_word1 = E[d][63:32] - mul\_res1;$   
 $result\_word0 = E[d][31:0] + mul\_res0;$   
 $E[c] = \{ssov(result\_word1, 32), ssov(result\_word0, 32)\}; // \text{Packed fraction}$

**MSUBADS.H      E[c], E[d], D[a], D[b] LU, n (RRR1)**

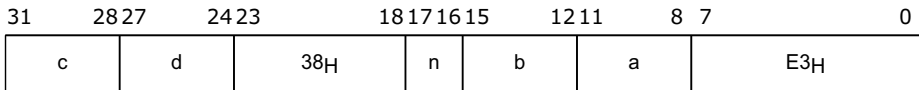
32||32 -||+ (16U \* 16L || 16L \* 16U) --> 32||32 saturated



$sc1 = (D[a][31:16] == 8000_H) \text{ AND } (D[b][15:0] == 8000_H) \text{ AND } (n == 1);$   
 $sc0 = (D[a][15:0] == 8000_H) \text{ AND } (D[b][31:16] == 8000_H) \text{ AND } (n == 1);$   
 $mul\_res1 = sc1 ? 7FFFFFFF_H : ((D[a][31:16] * D[b][15:0]) \ll n);$   
 $mul\_res0 = sc0 ? 7FFFFFFF_H : ((D[a][15:0] * D[b][31:16]) \ll n);$   
 $result\_word1 = E[d][63:32] - mul\_res1;$   
 $result\_word0 = E[d][31:0] + mul\_res0;$   
 $E[c] = \{ssov(result\_word1, 32), ssov(result\_word0, 32)\}; // \text{Packed fraction}$

**MSUBADS.H      E[c], E[d], D[a], D[b] UL, n (RRR1)**

32||32 -||+ (16U \* 16U || 16L \* 16L) --> 32||32 saturated

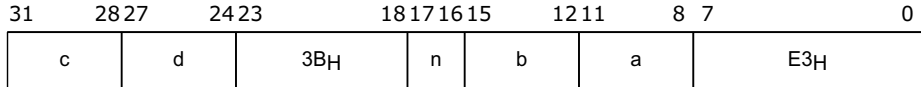


$sc1 = (D[a][31:16] == 8000_H) \text{ AND } (D[b][31:16] == 8000_H) \text{ AND } (n == 1);$   
 $sc0 = (D[a][15:0] == 8000_H) \text{ AND } (D[b][15:0] == 8000_H) \text{ AND } (n == 1);$   
 $mul\_res1 = sc1 ? 7FFFFFFF_H : ((D[a][31:16] * D[b][31:16]) \ll n);$   
 $mul\_res0 = sc0 ? 7FFFFFFF_H : ((D[a][15:0] * D[b][15:0]) \ll n);$   
 $result\_word1 = E[d][63:32] - mul\_res1;$   
 $result\_word0 = E[d][31:0] + mul\_res0;$

$E[c] = \{ssov(result\_word1, 32), ssov(result\_word0, 32)\}; //$  Packed fraction

**MSUBADS.H      E[c], E[d], D[a], D[b] UU, n (RRR1)**

32||32 -||+ (16L \* 16U || 16U \* 16U) --> 32||32 saturated



$sc1 = (D[a][15:0] == 8000_{H}) \text{ AND } (D[b][31:16] == 8000_{H}) \text{ AND } (n == 1);$

$sc0 = (D[a][31:16] == 8000_{H}) \text{ AND } (D[b][31:16] == 8000_{H}) \text{ AND } (n == 1);$

$mul\_res1 = sc1 ? 7FFFFFFF_{H} : ((D[a][15:0] * D[b][31:16]) \ll n);$

$mul\_res0 = sc0 ? 7FFFFFFF_{H} : ((D[a][31:16] * D[b][31:16]) \ll n);$

$result\_word1 = E[d][63:32] - mul\_res1;$

$result\_word0 = E[d][31:0] + mul\_res0;$

$E[c] = \{ssov(result\_word1, 32), ssov(result\_word0, 32)\}; //$  Packed fraction

**Status Flags**

C	Not set by these instructions.
V	$ov\_word1 = (result\_word1 > 7FFFFFFF_{H}) \text{ OR } (result\_word1 < -80000000_{H});$ $ov\_word0 = (result\_word0 > 7FFFFFFF_{H}) \text{ OR } (result\_word0 < -80000000_{H});$ $overflow = ov\_word1 \text{ OR } ov\_word0;$ if (overflow) then PSW.V = 1 else PSW.V = 0;
SV	if (overflow) then PSW.SV = 1 else PSW.SV = PSW.SV;
AV	$aov\_word1 = result\_word1[31] \wedge result\_word1[30];$ $aov\_word0 = result\_word0[31] \wedge result\_word0[30];$ $advanced\_overflow = aov\_word1 \text{ OR } aov\_word0;$ if (advanced_overflow) then PSW.AV = 1 else PSW.AV = 0;
SAV	if (advanced_overflow) then PSW.SAV = 1 else PSW.SAV = PSW.SAV;

**Examples**

-

**See Also**

-

## MSUBADM.H

Packed Multiply-Subtract/Add Q Format-Multi-precision

## MSUBADMS.H

Packed Multiply-Subtract/Add Q Format-Multi-precision, Saturated

### Description

Perform two multiplications of two signed 16-bit (half-word) values. Subtract one product and add the other product (left justified if  $n == 1$ ) left-shifted by 16, from/to a signed 64-bit value and put the result in a 64-bit register. There are four cases of half-word multiplication:

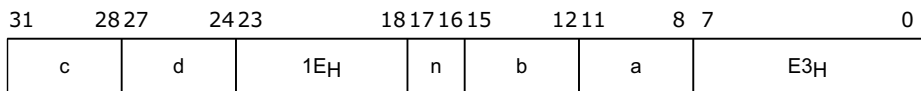
- $16U * 16U, 16L * 16L$
- $16U * 16L, 16L * 16U$
- $16U * 16L, 16L * 16L$
- $16L * 16U, 16U * 16U$

On overflow the MSUBADMS.H result is saturated.

Note that  $n$  should only take the values 0 or 1, any other value returns an undefined result. If ( $n == 1$ ) then  $8000_H * 8000_H = 7FFFFFFF_H$  (for signed 16-bit \* 16-bit multiplications only).

### MSUBADM.H      E[c], E[d], D[a], D[b] LL, n (RRR1)

$64 - (16U * 16L) + (16L * 16L) \rightarrow 64$



$sc1 = (D[a][31:16] == 8000_H) \text{ AND } (D[b][15:0] == 8000_H) \text{ AND } (n == 1);$

$sc0 = (D[a][15:0] == 8000_H) \text{ AND } (D[b][15:0] == 8000_H) \text{ AND } (n == 1);$

$result\_word1 = sc1 ? 7FFFFFFF_H : ((D[a][31:16] * D[b][15:0]) \ll n);$

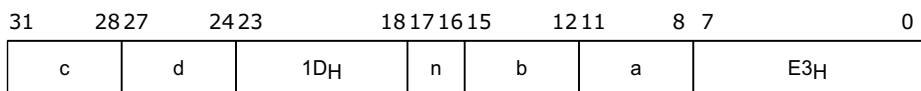
$result\_word0 = sc0 ? 7FFFFFFF_H : ((D[a][15:0] * D[b][15:0]) \ll n);$

$result = E[d] - ((result\_word1 - result\_word0) \ll 16);$

$E[c] = result[63:0]; // \text{ Multi-precision accumulator}$

### MSUBADM.H      E[c], E[d], D[a], D[b] LU, n (RRR1)

$64 - (16U * 16L) + (16L * 16U) \rightarrow 64$

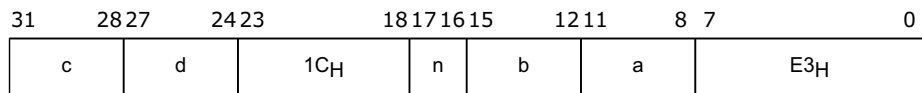


$sc1 = (D[a][31:16] == 8000_H) \text{ AND } (D[b][15:0] == 8000_H) \text{ AND } (n == 1);$

sc0 = (D[a][15:0] == 8000<sub>H</sub>) AND (D[b][31:16] == 8000<sub>H</sub>) AND (n == 1);  
 result\_word1 = sc1 ? 7FFFFFFF<sub>H</sub> : ((D[a][31:16] \* D[b][15:0]) << n);  
 result\_word0 = sc0 ? 7FFFFFFF<sub>H</sub> : ((D[a][15:0] \* D[b][31:16]) << n);  
 result = E[d] - ((result\_word1 - result\_word0) << 16);  
 E[c] = result[63:0]; // Multi-precision accumulator

**MSUBADM.H      E[c], E[d], D[a], D[b] UL, n (RRR1)**

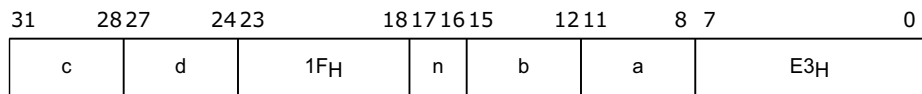
64 - (16U \* 16U) + (16L \* 16L) --> 64



sc1 = (D[a][31:16] == 8000<sub>H</sub>) AND (D[b][31:16] == 8000<sub>H</sub>) AND (n == 1);  
 sc0 = (D[a][15:0] == 8000<sub>H</sub>) AND (D[b][15:0] == 8000<sub>H</sub>) AND (n == 1);  
 result\_word1 = sc1 ? 7FFFFFFF<sub>H</sub> : ((D[a][31:16] \* D[b][31:16]) << n);  
 result\_word0 = sc0 ? 7FFFFFFF<sub>H</sub> : ((D[a][15:0] \* D[b][15:0]) << n);  
 result = E[d] - ((result\_word1 - result\_word0) << 16);  
 E[c] = result[63:0]; // Multi-precision accumulator

**MSUBADM.H      E[c], E[d], D[a], D[b] UU, n (RRR1)**

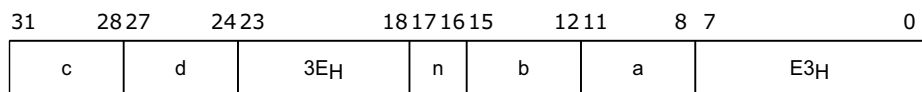
64 - (16L \* 16U) + (16U \* 16U) -> 64



sc1 = (D[a][15:0] == 8000<sub>H</sub>) AND (D[b][31:16] == 8000<sub>H</sub>) AND (n == 1);  
 sc0 = (D[a][31:16] == 8000<sub>H</sub>) AND (D[b][31:16] == 8000<sub>H</sub>) AND (n == 1);  
 result\_word1 = sc1 ? 7FFFFFFF<sub>H</sub> : ((D[a][15:0] \* D[b][31:16]) << n);  
 result\_word0 = sc0 ? 7FFFFFFF<sub>H</sub> : ((D[a][31:16] \* D[b][31:16]) << n);  
 result = E[d] - ((result\_word1 - result\_word0) << 16);  
 E[c] = result[63:0]; // Multi-precision accumulator

**MSUBADMS.H      E[c], E[d], D[a], D[b] LL, n (RRR1)**

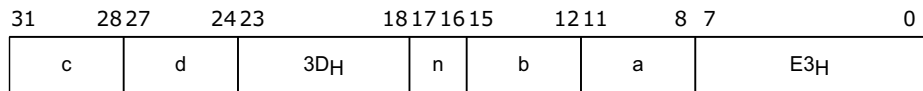
64 - (16U \* 16L) + (16L \* 16L) --> 64 saturated



```
sc1 = (D[a][31:16] == 8000H) AND (D[b][15:0] == 8000H) AND (n == 1);
sc0 = (D[a][15:0] == 8000H) AND (D[b][15:0] == 8000H) AND (n == 1);
result_word1 = sc1 ? 7FFFFFFFH : ((D[a][31:16] * D[b][15:0]) << n);
result_word0 = sc0 ? 7FFFFFFFH : ((D[a][15:0] * D[b][15:0]) << n);
result = E[d] - ((result_word1 - result_word0) << 16);
E[c] = ssov(result, 64); // Multi-precision accumulator
```

**MSUBADMS.H E[c], E[d], D[a], D[b] LU, n (RRR1)**

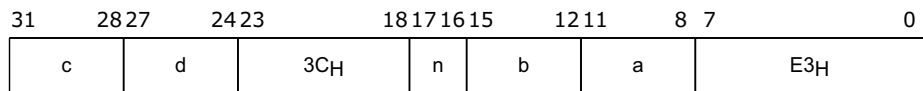
64 - (16U \* 16L) + (16L \* 16U) --> 64 saturated



```
sc1 = (D[a][31:16] == 8000H) AND (D[b][15:0] == 8000H) AND (n == 1);
sc0 = (D[a][15:0] == 8000H) AND (D[b][31:16] == 8000H) AND (n == 1);
result_word1 = sc1 ? 7FFFFFFFH : ((D[a][31:16] * D[b][15:0]) << n);
result_word0 = sc0 ? 7FFFFFFFH : ((D[a][15:0] * D[b][31:16]) << n);
result = E[d] - ((result_word1 - result_word0) << 16);
E[c] = ssov(result, 64); // Multi-precision accumulator
```

**MSUBADMS.H E[c], E[d], D[a], D[b] UL, n (RRR1)**

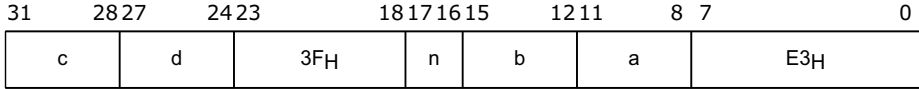
64 - (16U \* 16U) + (16L \* 16L) --> 64 saturated



```
sc1 = (D[a][31:16] == 8000H) AND (D[b][31:16] == 8000H) AND (n == 1);
sc0 = (D[a][15:0] == 8000H) AND (D[b][15:0] == 8000H) AND (n == 1);
result_word1 = sc1 ? 7FFFFFFFH : ((D[a][31:16] * D[b][31:16]) << n);
result_word0 = sc0 ? 7FFFFFFFH : ((D[a][15:0] * D[b][15:0]) << n);
result = E[d] - ((result_word1 - result_word0) << 16);
E[c] = ssov(result, 64); // Multi-precision accumulator
```

**MSUBADMS.H E[c], E[d], D[a], D[b] UU, n (RRR1)**

64 - (16L \* 16U) + (16U \* 16U) --> 64 saturated



sc1 = (D[a][15:0] == 8000<sub>H</sub>) AND (D[b][31:16] == 8000<sub>H</sub>) AND (n == 1);

sc0 = (D[a][31:16] == 8000<sub>H</sub>) AND (D[b][31:16] == 8000<sub>H</sub>) AND (n == 1);

result\_word1 = sc1 ? 7FFFFFFF<sub>H</sub> : ((D[a][15:0] \* D[b][31:16]) << n);

result\_word0 = sc0 ? 7FFFFFFF<sub>H</sub> : ((D[a][31:16] \* D[b][31:16]) << n);

result = E[d] - ((result\_word1 - result\_word0) << 16);

E[c] = ssov(result, 64); // Multi-precision accumulator

**Status Flags**

C	Not set by these instructions.
V	overflow = (result > 7FFFFFFF <sub>H</sub> ) OR (result < -8000000000000000 <sub>H</sub> ); if (overflow) then PSW.V = 1 else PSW.V = 0;
SV	if (overflow) then PSW.SV = 1 else PSW.SV = PSW.SV;
AV	advanced_overflow = result[63] ^ result[62]; if (advanced_overflow) then PSW.AV = 1 else PSW.AV = 0;
SAV	if (advanced_overflow) then PSW.SAV = 1 else PSW.SAV = PSW.SAV;

**Examples**

-

**See Also**

-

See Also Addendum for TriCore Arch Manual, Vol2, V1.3.8, pages 493/494

## MSUBADR.H

Packed Multiply-Subtract/Add Q Format with Rounding

## MSUBADRS.H

Packed Multiply-Subtract/Add Q Format with Rounding, Saturated

### Description

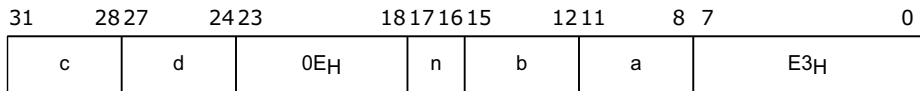
Multiply two signed 16-bit (half-word) values. Subtract (or add) the product (left justified if  $n == 1$ ) from (to) a signed 16-bit value and put the rounded result into half of a 32-bit register (Note that since there are two results the two register halves are used). There are four cases of half-word multiplication:

- $16U * 16U, 16L * 16L$
- $16U * 16L, 16L * 16U$
- $16U * 16L, 16L * 16L$
- $16L * 16U, 16U * 16U$

Note that  $n$  should only take the values 0 or 1, any other value returns an undefined result. If  $(n == 1)$  then  $8000_H * 8000_H = 7FFFFFFF_H$  (for signed 16-bit \* 16-bit multiplications only).

### MSUBADR.H      D[c], D[d], D[a], D[b] LL, n (RRR1)

$16U \parallel 16L -||+ (16U * 16L \parallel 16L * 16L)$  rounded  $\rightarrow 16||16$



$sc1 = (D[a][31:16] == 8000_H) \text{ AND } (D[b][15:0] == 8000_H) \text{ AND } (n == 1);$

$sc0 = (D[a][15:0] == 8000_H) \text{ AND } (D[b][15:0] == 8000_H) \text{ AND } (n == 1);$

$mul\_res1 = sc1 ? 7FFFFFFF_H : ((D[a][31:16] * D[b][15:0]) \ll n);$

$mul\_res0 = sc0 ? 7FFFFFFF_H : ((D[a][15:0] * D[b][15:0]) \ll n);$

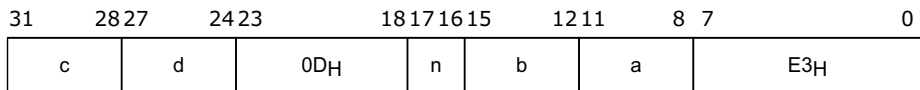
$result\_halfword1 = \{D[d][31:16], 16'b0\} - mul\_res1 + 8000_H;$

$result\_halfword0 = \{D[d][15:0], 16'b0\} + mul\_res0 + 8000_H;$

$D[c] = \{result\_halfword1[31:16], result\_halfword0[31:16]\}; // \text{Packed short fraction}$

### MSUBADR.H      D[c], D[d], D[a], D[b] LU, n (RRR1)

$16U \parallel 16L -||+ (16U * 16L \parallel 16L * 16U)$  rounded  $\rightarrow 16||16$

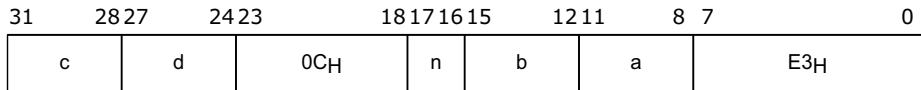


$sc1 = (D[a][31:16] == 8000_H) \text{ AND } (D[b][15:0] == 8000_H) \text{ AND } (n == 1);$

sc0 = (D[a][15:0] == 8000<sub>H</sub>) AND (D[b][31:16] == 8000<sub>H</sub>) AND (n == 1);  
mul\_res1 = sc1 ? 7FFFFFFF<sub>H</sub> : ((D[a][31:16] \* D[b][15:0]) << n);  
mul\_res0 = sc0 ? 7FFFFFFF<sub>H</sub> : ((D[a][15:0] \* D[b][31:16]) << n);  
result\_halfword1 = {D[d][31:16], 16'b0} - mul\_res1 + 8000<sub>H</sub>;  
result\_halfword0 = {D[d][15:0], 16'b0} + mul\_res0 + 8000<sub>H</sub>;  
D[c] = {result\_halfword1[31:16], result\_halfword0[31:16]}; // Packed short fraction

**MSUBADR.H      D[c], D[d], D[a], D[b] UL, n (RRR1)**

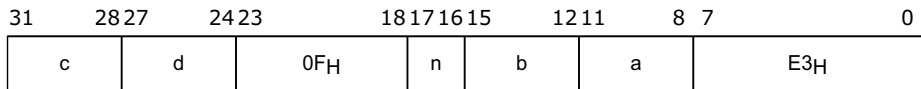
16U||16L -||+ (16U \* 16U || 16L \* 16L) rounded --> 16||16



sc1 = (D[a][31:16] == 8000<sub>H</sub>) AND (D[b][31:16] == 8000<sub>H</sub>) AND (n == 1);  
sc0 = (D[a][15:0] == 8000<sub>H</sub>) AND (D[b][15:0] == 8000<sub>H</sub>) AND (n == 1);  
mul\_res1 = sc1 ? 7FFFFFFF<sub>H</sub> : ((D[a][31:16] \* D[b][31:16]) << n);  
mul\_res0 = sc0 ? 7FFFFFFF<sub>H</sub> : ((D[a][15:0] \* D[b][15:0]) << n);  
result\_halfword1 = {D[d][31:16], 16'b0} - mul\_res1 + 8000<sub>H</sub>;  
result\_halfword0 = {D[d][15:0], 16'b0} + mul\_res0 + 8000<sub>H</sub>;  
D[c] = {result\_halfword1[31:16], result\_halfword0[31:16]}; // Packed short fraction

**MSUBADR.H      D[c], D[d], D[a], D[b] UU, n (RRR1)**

16U || 16L -||+ (16L \* 16U || 16U \* 16U) rounded --> 16||16

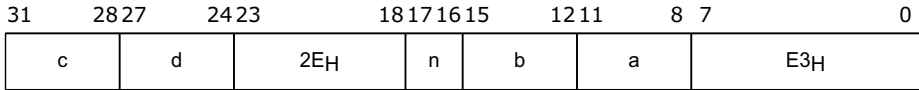


sc1 = (D[a][15:0] == 8000<sub>H</sub>) AND (D[b][31:16] == 8000<sub>H</sub>) AND (n == 1);  
sc0 = (D[a][31:16] == 8000<sub>H</sub>) AND (D[b][31:16] == 8000<sub>H</sub>) AND (n == 1);  
mul\_res1 = sc1 ? 7FFFFFFF<sub>H</sub> : ((D[a][15:0] \* D[b][31:16]) << n);  
mul\_res0 = sc0 ? 7FFFFFFF<sub>H</sub> : ((D[a][31:16] \* D[b][31:16]) << n);  
result\_halfword1 = {D[d][31:16], 16'b0} - mul\_res1 + 8000<sub>H</sub>;  
result\_halfword0 = {D[d][15:0], 16'b0} + mul\_res0 + 8000<sub>H</sub>;  
D[c] = {result\_halfword1[31:16], result\_halfword0[31:16]}; // Packed short fraction

**MSUBADRS.H      D[c], D[d], D[a], D[b] LL, n (RRR1)**

16U || 16L -||+ (16U \* 16L || 16L \* 16L) rounded --> 16||16 saturated



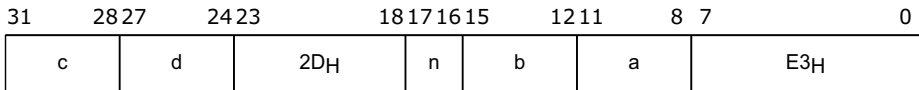


```

sc1 = (D[a][31:16] == 8000H) AND (D[b][15:0] == 8000H) AND (n == 1);
sc0 = (D[a][15:0] == 8000H) AND (D[b][15:0] == 8000H) AND (n == 1);
mul_res1 = sc1 ? 7FFFFFFFH : ((D[a][31:16] * D[b][15:0]) << n);
mul_res0 = sc0 ? 7FFFFFFFH : ((D[a][15:0] * D[b][15:0]) << n);
result_halfword1 = {D[d][31:16], 16'b0} - mul_res1 + 8000H;
result_halfword0 = {D[d][15:0], 16'b0} + mul_res0 + 8000H;
D[c] = {ssov(result_halfword1, 32)[31:16], ssov(result_halfword0, 32)[31:16]};
// Packed short fraction result
    
```

**MSUBADRS.H D[c], D[d], D[a], D[b] LU, n (RRR1)**

16U || 16L -||+ (16U \* 16L || 16L \* 16U) rounded --> 16||16 saturated

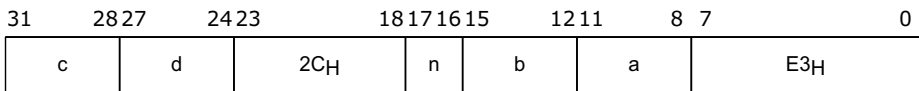


```

sc1 = (D[a][31:16] == 8000H) AND (D[b][15:0] == 8000H) AND (n == 1);
sc0 = (D[a][15:0] == 8000H) AND (D[b][31:16] == 8000H) AND (n == 1);
mul_res1 = sc1 ? 7FFFFFFFH : ((D[a][31:16] * D[b][15:0]) << n);
mul_res0 = sc0 ? 7FFFFFFFH : ((D[a][15:0] * D[b][31:16]) << n);
result_halfword1 = {D[d][31:16], 16'b0} - mul_res1 + 8000H;
result_halfword0 = {D[d][15:0], 16'b0} + mul_res0 + 8000H;
D[c] = {ssov(result_halfword1, 32)[31:16], ssov(result_halfword0, 32)[31:16]};
// Packed short fraction result
    
```

**MSUBADRS.H D[c], D[d], D[a], D[b] UL, n (RRR1)**

16U || 16L -||+ (16U \* 16U || 16L \* 16L) rounded --> 16||16 saturated



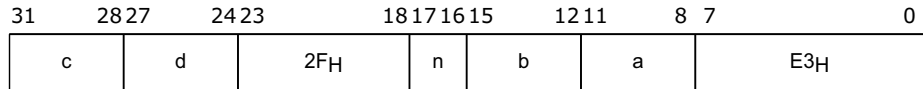
```

sc1 = (D[a][31:16] == 8000H) AND (D[b][31:16] == 8000H) AND (n == 1);
sc0 = (D[a][15:0] == 8000H) AND (D[b][15:0] == 8000H) AND (n == 1);
mul_res1 = sc1 ? 7FFFFFFFH : ((D[a][31:16] * D[b][31:16]) << n);
mul_res0 = sc0 ? 7FFFFFFFH : ((D[a][15:0] * D[b][15:0]) << n);
    
```

```
result_halfword1 = {D[d][31:16], 16'b0} - mul_res1 + 8000H;
result_halfword0 = {D[d][15:0], 16'b0} + mul_res0 + 8000H;
D[c] = {ssov(result_halfword1, 32)[31:16], ssov(result_halfword0, 32)[31:16]};
// Packed short fraction result
```

### MSUBADRS.H      D[c], D[d], D[a], D[b] UU, n (RRR1)

16U || 16L -||+ (16L \* 16U || 16U \* 16U) rounded > 16||16 saturated



```
sc1 = (D[a][15:0] == 8000H) AND (D[b][31:16] == 8000H) AND (n == 1);
sc0 = (D[a][31:16] == 8000H) AND (D[b][31:16] == 8000H) AND (n == 1);
mul_res1 = sc1 ? 7FFFFFFFH : ((D[a][15:0] * D[b][31:16]) << n);
mul_res0 = sc0 ? 7FFFFFFFH : ((D[a][31:16] * D[b][31:16]) << n);
result_halfword1 = {D[d][31:16], 16'b0} - mul_res1 + 8000H;
result_halfword0 = {D[d][15:0], 16'b0} + mul_res0 + 8000H;
D[c] = {ssov(result_halfword1, 32)[31:16], ssov(result_halfword0, 32)[31:16]};
// Packed short fraction result
```

### Status Flags

C	Not set by these instructions.
V	$ov\_halfword1 = (result\_halfword1 > 7FFFFFFF_H) OR (result\_halfword1 < -80000000_H);$ $ov\_halfword0 = (result\_halfword0 > 7FFFFFFF_H) OR (result\_halfword0 < -80000000_H);$ overflow = $ov\_halfword1 OR ov\_halfword0;$ if (overflow) then PSW.V = 1 else PSW.V = 0;
SV	if (overflow) then PSW.SV = 1 else PSW.SV = PSW.SV;
AV	$aov\_halfword1 = result\_halfword1[31] ^ result\_halfword1[30];$ $aov\_halfword0 = result\_halfword0[31] ^ result\_halfword0[30];$ advanced_overflow = $aov\_halfword1 OR aov\_halfword0;$ if (advanced_overflow) then PSW.AV = 1 else PSW.AV = 0;
SAV	if (advanced_overflow) then PSW.SAV = 1; else PSW.SAV = PSW.SAV;

### Examples

-

**See Also**

-

See Also Addendum for TriCore Arch Manual, Vol2, V1.3.8, pages 493/494

## MSUBM.H

Packed Multiply-Subtract Q Format-Multi-precision

## MSUBMS.H

Packed Multiply-Subtract Q Format-Multi-precision, Saturated

### Description

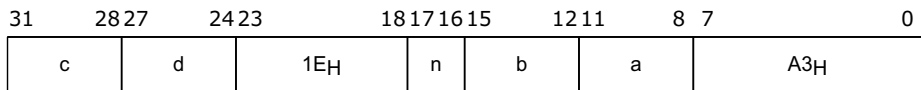
Perform two multiplications of two signed 16-bit (half-word) values. Subtract the two products (left justified if  $n == 1$ ) left-shifted by 16, from a signed 64-bit value and put the result in a 64-bit register. There are four cases of half-word multiplication:

- $16U * 16U, 16L * 16L$
- $16U * 16L, 16L * 16U$
- $16U * 16L, 16L * 16L$
- $16L * 16U, 16U * 16U$

Note that  $n$  should only take the values 0 or 1, any other value returns an undefined result. If  $(n == 1)$  then  $8000_H * 8000_H = 7FFFFFFF_H$  (for signed 16-bit \* 16-bit multiplications only).

### MSUBM.H                      E[c], E[d], D[a], D[b] LL, n (RRR1)

$64 - (16U * 16L) - (16L * 16L) \rightarrow 64$



$sc1 = (D[a][31:16] == 8000_H) \text{ AND } (D[b][15:0] == 8000_H) \text{ AND } (n == 1);$

$sc0 = (D[a][15:0] == 8000_H) \text{ AND } (D[b][15:0] == 8000_H) \text{ AND } (n == 1);$

$result\_word1 = sc1 ? 7FFFFFFF_H : ((D[a][31:16] * D[b][15:0]) \ll n);$

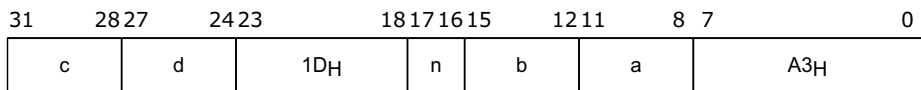
$result\_word0 = sc0 ? 7FFFFFFF_H : ((D[a][15:0] * D[b][15:0]) \ll n);$

$result = E[d] - ((result\_word1 + result\_word0) \ll 16);$

$E[c] = result[63:0]; // \text{ Multi-precision accumulator}$

### MSUBM.H                      E[c], E[d], D[a], D[b] LU, n (RRR1)

$64 - (16U * 16L) - (16L * 16U) \rightarrow 64$



$sc1 = (D[a][31:16] == 8000_H) \text{ AND } (D[b][15:0] == 8000_H) \text{ AND } (n == 1);$

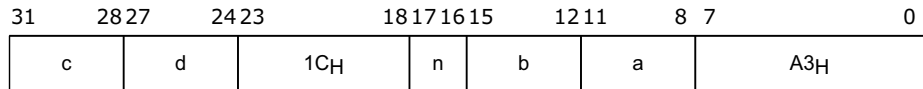
$sc0 = (D[a][15:0] == 8000_H) \text{ AND } (D[b][31:16] == 8000_H) \text{ AND } (n == 1);$

$result\_word1 = sc1 ? 7FFFFFFF_H : ((D[a][31:16] * D[b][15:0]) \ll n);$

result\_word0 = sc0 ? 7FFFFFFF<sub>H</sub> : ((D[a][15:0] \* D[b][31:16]) << n);  
 result = E[d] - ((result\_word1 + result\_word0) << 16);  
 E[c] = result[63:0]; // Multi-precision accumulator

**MSUBM.H                    E[c], E[d], D[a], D[b] UL, n (RRR1)**

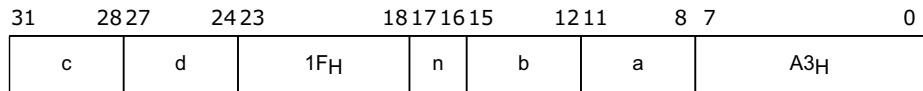
64 - (16U \* 16U) - (16L \* 16L) --> 64



sc1 = (D[a][31:16] == 8000<sub>H</sub>) AND (D[b][31:16] == 8000<sub>H</sub>) AND (n == 1);  
 sc0 = (D[a][15:0] == 8000<sub>H</sub>) AND (D[b][15:0] == 8000<sub>H</sub>) AND (n == 1);  
 result\_word1 = sc1 ? 7FFFFFFF<sub>H</sub> : ((D[a][31:16] \* D[b][31:16]) << n);  
 result\_word0 = sc0 ? 7FFFFFFF<sub>H</sub> : ((D[a][15:0] \* D[b][15:0]) << n);  
 result = E[d] - ((result\_word1 + result\_word0) << 16);  
 E[c] = result[63:0]; // Multi-precision accumulator

**MSUBM.H                    E[c], E[d], D[a], D[b] UU, n (RRR1)**

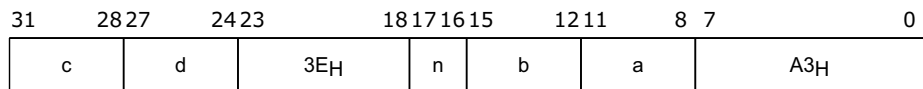
64 - (16L \* 16U) - (16U \* 16U) --> 64



sc1 = (D[a][15:0] == 8000<sub>H</sub>) AND (D[b][31:16] == 8000<sub>H</sub>) AND (n == 1);  
 sc0 = (D[a][31:16] == 8000<sub>H</sub>) AND (D[b][31:16] == 8000<sub>H</sub>) AND (n == 1);  
 result\_word1 = sc1 ? 7FFFFFFF<sub>H</sub> : ((D[a][15:0] \* D[b][31:16]) << n);  
 result\_word0 = sc0 ? 7FFFFFFF<sub>H</sub> : ((D[a][31:16] \* D[b][31:16]) << n);  
 result = E[d] - ((result\_word1 + result\_word0) << 16);  
 E[c] = result[63:0]; // Multi-precision accumulator

**MSUBMS.H                    E[c], E[d], D[a], D[b] LL, n (RRR1)**

64 - (16U \* 16L) - (16L \* 16L) --> 64 saturated

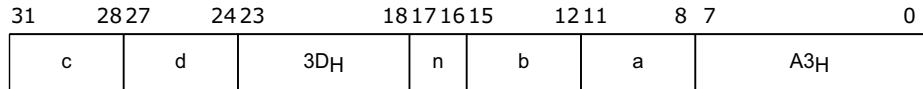


sc1 = (D[a][31:16] == 8000<sub>H</sub>) AND (D[b][15:0] == 8000<sub>H</sub>) AND (n == 1);  
 sc0 = (D[a][15:0] == 8000<sub>H</sub>) AND (D[b][15:0] == 8000<sub>H</sub>) AND (n == 1);

```
result_word1 = sc1 ? 7FFFFFFFH : ((D[a][31:16] * D[b][15:0]) << n);
result_word0 = sc0 ? 7FFFFFFFH : ((D[a][15:0] * D[b][15:0]) << n);
result = E[d] - ((result_word1 + result_word0) << 16);
E[c] = ssov(result, 64); // Multi-precision accumulator
```

**MSUBMS.H E[c], E[d], D[a], D[b] LU, n (RRR1)**

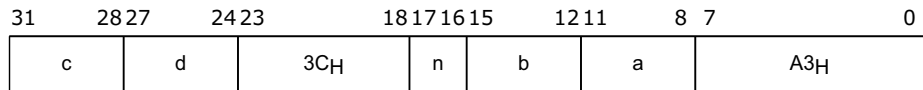
64 - (16U \* 16L) - (16L \* 16U) --> saturated



```
sc1 = (D[a][31:16] == 8000H) AND (D[b][15:0] == 8000H) AND (n == 1);
sc0 = (D[a][15:0] == 8000H) AND (D[b][31:16] == 8000H) AND (n == 1);
result_word1 = sc1 ? 7FFFFFFFH : ((D[a][31:16] * D[b][15:0]) << n);
result_word0 = sc0 ? 7FFFFFFFH : ((D[a][15:0] * D[b][31:16]) << n);
result = E[d] - ((result_word1 + result_word0) << 16);
E[c] = ssov(result, 64); // Multi-precision accumulator
```

**MSUBMS.H E[c], E[d], D[a], D[b] UL, n (RRR1)**

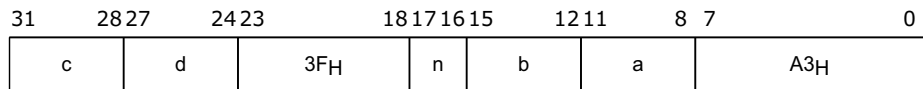
64 - (16U \* 16U) - (16L \* 16L) > 64 saturated



```
sc1 = (D[a][31:16] == 8000H) AND (D[b][31:16] == 8000H) AND (n == 1);
sc0 = (D[a][15:0] == 8000H) AND (D[b][15:0] == 8000H) AND (n == 1);
result_word1 = sc1 ? 7FFFFFFFH : ((D[a][31:16] * D[b][31:16]) << n);
result_word0 = sc0 ? 7FFFFFFFH : ((D[a][15:0] * D[b][15:0]) << n);
result = E[d] - ((result_word1 + result_word0) << 16);
E[c] = ssov(result, 64); // Multi-precision accumulator
```

**MSUBMS.H E[c], E[d], D[a], D[b] UU, n (RRR1)**

64 - (16L \* 16U) - (16U \* 16U) --> 64 saturated



```
sc1 = (D[a][15:0] == 8000H) AND (D[b][31:16] == 8000H) AND (n == 1);
```

**Instruction Set**

```
sc0 = (D[a][31:16] == 8000H) AND (D[b][31:16] == 8000H) AND (n == 1);
result_word1 = sc1 ? 7FFFFFFFH : ((D[a][15:0] * D[b][31:16]) << n);
result_word0 = sc0 ? 7FFFFFFFH : ((D[a][31:16] * D[b][31:16]) << n);
result = E[d] - ((result_word1 + result_word0) << 16);
E[c] = ssov(result, 64); // Multi-precision accumulator
```

**Status Flags**

C	Not set by these instructions.
V	overflow = (result > 7FFFFFFFFFFFFFFF <sub>H</sub> ) OR (result < -8000000000000000 <sub>H</sub> ); if (overflow) then PSW.V = 1 else PSW.V = 0;
SV	if (overflow) then PSW.SV = 1 else PSW.SV = PSW.SV;
AV	advanced_overflow = result[63] ^ result[62]; if (advanced_overflow) then PSW.AV = 1 else PSW.AV = 0;
SAV	if (advanced_overflow) then PSW.SAV = 1 else PSW.SAV = PSW.SAV;

**Examples**

-

**See Also**

-

## MSUBR.H

Packed Multiply-Subtract Q Format with Rounding

## MSUBRS.H

Packed Multiply-Subtract Q Format with Rounding, Saturated

### Description

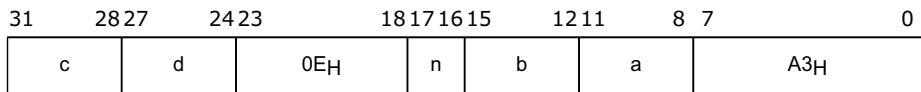
Multiply two signed 16-bit (half-word) values. Subtract the product (left justified if  $n == 1$ ) from a signed 16-bit or 32-bit value and put the rounded result into half of a 32-bit register. Note that since there are two results the two register halves are used. There are four cases of half-word multiplication:

- $16U * 16U, 16L * 16L$
- $16U * 16L, 16L * 16U$
- $16U * 16L, 16L * 16L$
- $16L * 16U, 16U * 16U$

Note that  $n$  should only take the values 0 or 1. Any other value returns an undefined result. If  $(n == 1)$  then  $8000_H * 8000_H = 7FFFFFFF_H$  (for signed 16-bit \* 16-bit multiplications only).

### MSUBR.H                      D[c], D[d], D[a], D[b] LL, n (RRR1)

$16U \parallel 16L$  -|- ( $16U * 16L \parallel 16L * 16L$ ) rounded  $\rightarrow 16 \parallel 16$



$sc1 = (D[a][31:16] == 8000_H) \text{ AND } (D[b][15:0] == 8000_H) \text{ AND } (n == 1);$

$sc0 = (D[a][15:0] == 8000_H) \text{ AND } (D[b][15:0] == 8000_H) \text{ AND } (n == 1);$

$mul\_res1 = sc1 ? 7FFFFFFF_H : ((D[a][31:16] * D[b][15:0]) \ll n);$

$mul\_res0 = sc0 ? 7FFFFFFF_H : ((D[a][15:0] * D[b][15:0]) \ll n);$

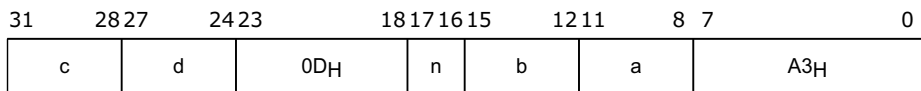
$result\_halfword1 = \{D[d][31:16], 16'b0\} - mul\_res1 + 8000_H;$

$result\_halfword0 = \{D[d][15:0], 16'b0\} - mul\_res0 + 8000_H;$

$D[c] = \{result\_halfword1[31:16], result\_halfword0[31:16]\}; // \text{Packed short fraction}$

### MSUBR.H                      D[c], D[d], D[a], D[b] LU, n (RRR1)

$16U \parallel 16L$  -|- ( $16U * 16L \parallel 16L * 16U$ ) rounded  $\rightarrow 16 \parallel 16$



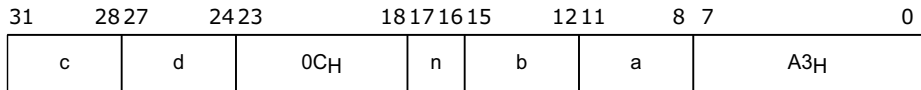
$sc1 = (D[a][31:16] == 8000_H) \text{ AND } (D[b][15:0] == 8000_H) \text{ AND } (n == 1);$



sc0 = (D[a][15:0] == 8000<sub>H</sub>) AND (D[b][31:16] == 8000<sub>H</sub>) AND (n == 1);  
mul\_res1 = sc1 ? 7FFFFFFF<sub>H</sub> : ((D[a][31:16] \* D[b][15:0]) << n);  
mul\_res0 = sc0 ? 7FFFFFFF<sub>H</sub> : ((D[a][15:0] \* D[b][31:16]) << n);  
result\_halfword1 = {D[d][31:16], 16'b0} - mul\_res1 + 8000<sub>H</sub>;  
result\_halfword0 = {D[d][15:0], 16'b0} - mul\_res0 + 8000<sub>H</sub>;  
D[c] = {result\_halfword1[31:16], result\_halfword0[31:16]}; // Packed short fraction

**MSUBR.H                    D[c], D[d], D[a], D[b] UL, n (RRR1)**

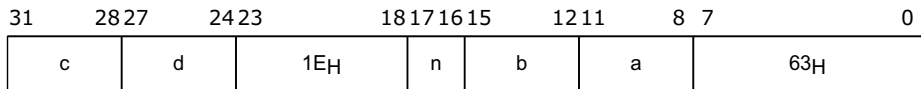
16U || 16L -||- (16U \* 16U || 16L \* 16L) rounded --> 16||16



sc1 = (D[a][31:16] == 8000<sub>H</sub>) AND (D[b][31:16] == 8000<sub>H</sub>) AND (n == 1);  
sc0 = (D[a][15:0] == 8000<sub>H</sub>) AND (D[b][15:0] == 8000<sub>H</sub>) AND (n == 1);  
mul\_res1 = sc1 ? 7FFFFFFF<sub>H</sub> : ((D[a][31:16] \* D[b][31:16]) << n);  
mul\_res0 = sc0 ? 7FFFFFFF<sub>H</sub> : ((D[a][15:0] \* D[b][15:0]) << n);  
result\_halfword1 = {D[d][31:16], 16'b0} - mul\_res1 + 8000<sub>H</sub>;  
result\_halfword0 = {D[d][15:0], 16'b0} - mul\_res0 + 8000<sub>H</sub>;  
D[c] = {result\_halfword1[31:16], result\_halfword0[31:16]}; // Packed short fraction

**MSUBR.H                    D[c], E[d], D[a], D[b] UL, n (RRR1)**

32 || 32 -||- (16U \* 16U || 16L \* 16L) rounded > 16 || 16



sc1 = (D[a][31:16] == 8000<sub>H</sub>) AND (D[b][31:16] == 8000<sub>H</sub>) AND (n == 1);  
sc0 = (D[a][15:0] == 8000<sub>H</sub>) AND (D[b][15:0] == 8000<sub>H</sub>) AND (n == 1);  
mul\_res1 = sc1 ? 7FFFFFFF<sub>H</sub> : ((D[a][31:16] \* D[b][31:16]) << n);  
mul\_res0 = sc0 ? 7FFFFFFF<sub>H</sub> : ((D[a][15:0] \* D[b][15:0]) << n);  
result\_halfword1 = E[d][63:32] - mul\_res1 + 8000<sub>H</sub>;  
result\_halfword0 = E[d][31:0] - mul\_res0 + 8000<sub>H</sub>;  
D[c] = {result\_halfword1[31:16], result\_halfword0[31:16]}; // Packed short fraction

**MSUBR.H                    D[c], D[d], D[a], D[b] UU, n (RRR1)**

16U || 16L -||- (16L \* 16U || 16U \* 16U) rounded --> 16||16

31	28 27	24 23	18 17 16 15	12 11	8 7	0
c	d	0FH	n	b	a	A3H

$sc1 = (D[a][15:0] == 8000_H) \text{ AND } (D[b][31:16] == 8000_H) \text{ AND } (n == 1);$   
 $sc0 = (D[a][31:16] == 8000_H) \text{ AND } (D[b][31:16] == 8000_H) \text{ AND } (n == 1);$   
 $mul\_res1 = sc1 ? 7FFFFFFF_H : ((D[a][15:0] * D[b][31:16]) \ll n);$   
 $mul\_res0 = sc0 ? 7FFFFFFF_H : ((D[a][31:16] * D[b][31:16]) \ll n);$   
 $result\_halfword1 = \{D[d][31:16], 16'b0\} - mul\_res1 + 8000_H;$   
 $result\_halfword0 = \{D[d][15:0], 16'b0\} - mul\_res0 + 8000_H;$   
 $D[c] = \{result\_halfword1[31:16], result\_halfword0[31:16]\}; // \text{ Packed short fraction}$

**MSUBRS.H            D[c], D[d], D[a], D[b] LL, n (RRR1)**

16U || 16L -||- (16U \* 16L || 16L \* 16L) rounded --> 16||16 saturated

31	28 27	24 23	18 17 16 15	12 11	8 7	0
c	d	2EH	n	b	a	A3H

$sc1 = (D[a][31:16] == 8000_H) \text{ AND } (D[b][15:0] == 8000_H) \text{ AND } (n == 1);$   
 $sc0 = (D[a][15:0] == 8000_H) \text{ AND } (D[b][15:0] == 8000_H) \text{ AND } (n == 1);$   
 $mul\_res1 = sc1 ? 7FFFFFFF_H : ((D[a][31:16] * D[b][15:0]) \ll n);$   
 $mul\_res0 = sc0 ? 7FFFFFFF_H : ((D[a][15:0] * D[b][15:0]) \ll n);$   
 $result\_halfword1 = \{D[d][31:16], 16'b0\} - mul\_res1 + 8000_H;$   
 $result\_halfword0 = \{D[d][15:0], 16'b0\} - mul\_res0 + 8000_H;$   
 $D[c] = \{ssov(result\_halfword1, 32)[31:16], ssov(result\_halfword0, 32)[31:16]\};$   
// Packed short fraction result

**MSUBRS.H            D[c], D[d], D[a], D[b] LU, n (RRR1)**

16U || 16L -||- (16U \* 16L || 16L \* 16U) rounded --> 16||16 saturated

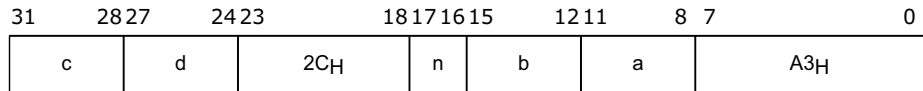
31	28 27	24 23	18 17 16 15	12 11	8 7	0
c	d	2DH	n	b	a	A3H

$sc1 = (D[a][31:16] == 8000_H) \text{ AND } (D[b][15:0] == 8000_H) \text{ AND } (n == 1);$   
 $sc0 = (D[a][15:0] == 8000_H) \text{ AND } (D[b][31:16] == 8000_H) \text{ AND } (n == 1);$   
 $mul\_res1 = sc1 ? 7FFFFFFF_H : ((D[a][31:16] * D[b][15:0]) \ll n);$   
 $mul\_res0 = sc0 ? 7FFFFFFF_H : ((D[a][15:0] * D[b][31:16]) \ll n);$   
 $result\_halfword1 = \{D[d][31:16], 16'b0\} - mul\_res1 + 8000_H;$

```
result_halfword0 = {D[d][15:0], 16'b0} - mul_res0 + 8000H;
D[c] = {ssov(result_halfword1, 32)[31:16], ssov(result_halfword0, 32)[31:16]};
// Packed short fraction result
```

**MSUBRS.H**            **D[c], D[d], D[a], D[b] UL, n (RRR1)**

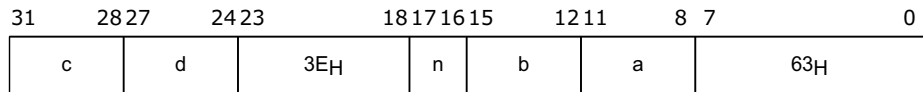
16U || 16L -||- (16U \* 16U || 16L \* 16L) rounded --> 16||16 saturated



```
sc1 = (D[a][31:16] == 8000H) AND (D[b][31:16] == 8000H) AND (n == 1);
sc0 = (D[a][15:0] == 8000H) AND (D[b][15:0] == 8000H) AND (n == 1);
mul_res1 = sc1 ? 7FFFFFFFH : ((D[a][31:16] * D[b][31:16]) << n);
mul_res0 = sc0 ? 7FFFFFFFH : ((D[a][15:0] * D[b][15:0]) << n);
result_halfword1 = {D[d][31:16], 16'b0} - mul_res1 + 8000H;
result_halfword0 = {D[d][15:0], 16'b0} - mul_res0 + 8000H;
D[c] = {ssov(result_halfword1, 32)[31:16], ssov(result_halfword0, 32)[31:16]};
// Packed short fraction result
```

**MSUBRS.H**            **D[c], E[d], D[a], D[b] UL, n (RRR1)**

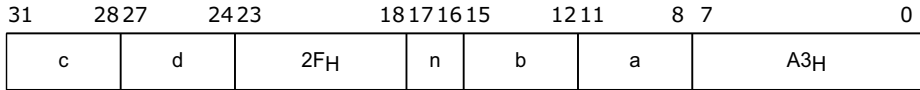
32||32 -||- (16U \* 16U || 16L \* 16L) rounded --> 16||16 saturated



```
sc1 = (D[a][31:16] == 8000H) AND (D[b][31:16] == 8000H) AND (n == 1);
sc0 = (D[a][15:0] == 8000H) AND (D[b][15:0] == 8000H) AND (n == 1);
mul_res1 = sc1 ? 7FFFFFFFH : ((D[a][31:16] * D[b][31:16]) << n);
mul_res0 = sc0 ? 7FFFFFFFH : ((D[a][15:0] * D[b][15:0]) << n);
result_halfword1 = E[d][63:32] - mul_res1 + 8000H;
result_halfword0 = E[d][31:0] - mul_res0 + 8000H;
D[c] = {ssov(result_halfword1, 32)[31:16], ssov(result_halfword0, 32)[31:16]};
// Packed short fraction result
```

**MSUBRS.H**            **D[c], D[d], D[a], D[b] UU, n (RRR1)**

16U || 16L -||- (16L \* 16U || 16U \* 16U) rounded --> 16||16 saturated



```

sc1 = (D[a][15:0] == 8000H) AND (D[b][31:16] == 8000H) AND (n == 1);
sc0 = (D[a][31:16] == 8000H) AND (D[b][31:16] == 8000H) AND (n == 1);
mul_res1 = sc1 ? 7FFFFFFFH : ((D[a][15:0] * D[b][31:16]) << n);
mul_res0 = sc0 ? 7FFFFFFFH : ((D[a][31:16] * D[b][31:16]) << n);
result_halfword1 = {D[d][31:16], 16'b0} - mul_res1 + 8000H;
result_halfword0 = {D[d][15:0], 16'b0} - mul_res0 + 8000H;
D[c] = {ssov(result_halfword1, 32)[31:16], ssov(result_halfword0, 32)[31:16]};
// Packed short fraction result
  
```

**Status Flags**

C	Not set by these instructions.
V	$ov\_halfword1 = (result\_halfword1 > 7FFFFFFFH) \text{ OR } (result\_halfword1 < -80000000H);$ $ov\_halfword0 = (result\_halfword0 > 7FFFFFFFH) \text{ OR } (result\_halfword0 < -80000000H);$ $overflow = ov\_halfword1 \text{ OR } ov\_halfword0;$ $\text{if } (overflow) \text{ then } PSW.V = 1 \text{ else } PSW.V = 0;$
SV	$\text{if } (overflow) \text{ then } PSW.SV = 1 \text{ else } PSW.SV = PSW.SV;$
AV	$aov\_overflow1 = result\_halfword1[31] \wedge result\_halfword1[30];$ $aov\_overflow0 = result\_halfword0[31] \wedge result\_halfword0[30];$ $advanced\_overflow = aov\_overflow1 \text{ OR } aov\_overflow0;$ $\text{if } (advanced\_overflow) \text{ then } PSW.AV = 1 \text{ else } PSW.AV = 0;$
SAV	$\text{if } (advanced\_overflow) \text{ then } PSW.SAV = 1 \text{ else } PSW.SAV = PSW.SAV;$

**Examples**

-

**See Also**

-

See Also Addendum for TriCore Arch Manual, Vol2, V1.3.8, pages 493/494

## MSUBR.Q

### Multiply-Subtract Q Format with Rounding

## MSUBRS.Q

### Multiply-Subtract Q Format with Rounding, Saturated

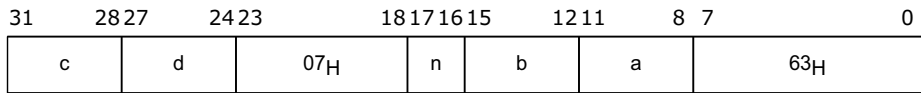
#### Description

Multiply two signed 16-bit (half-word) values. Subtract the product (left justified if  $n == 1$ ) from a 32-bit signed value, and put the rounded result in a 32-bit register. The lower half-word is cleared. Overflow and advanced overflow are calculated on the final results.

Note that  $n$  should only take the values 0 or 1, any other value returns an undefined result. If ( $n == 1$ ) then  $8000_H * 8000_H = 7FFFFFFF_H$  (for signed 16-bit \* 16-bit multiplications only).

#### MSUBR.Q                    D[c], D[d], D[a] L, D[b] L, n (RRR1)

32 - (16L \* 16L) rounded --> 32



$sc = (D[a][15:0] == 8000_H) \text{ AND } (D[b][15:0] == 8000_H) \text{ AND } (n == 1);$

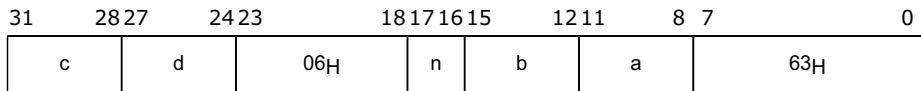
$mul\_res = sc ? 7FFFFFFF_H : ((D[a][15:0] * D[b][15:0]) << n);$

$result = D[d] - mul\_res + 8000_H;$

$D[c] = \{result[31:16], 16'b0\}; // \text{Short fraction}$

#### MSUBR.Q                    D[c], D[d], D[a] U, D[b] U, n (RRR1)

32 - (16U \* 16U) rounded --> 32



$sc = (D[a][31:16] == 8000_H) \text{ AND } (D[b][31:16] == 8000_H) \text{ AND } (n == 1);$

$mul\_res = sc ? 7FFFFFFF_H : ((D[a][31:16] * D[b][31:16]) << n);$

$result = D[d] - mul\_res + 8000_H;$

$D[c] = \{result[31:16], 16'b0\}; // \text{Short fraction}$

#### MSUBRS.Q                    D[c], D[d], D[a] L, D[b] L, n (RRR1)

32 - (16L \* 16L) rounded -->32 saturated

Instruction Set

31	28 27	24 23	18 17 16 15	12 11	8 7	0
c	d	27 <sub>H</sub>	n	b	a	63 <sub>H</sub>

sc = (D[a][15:0] == 8000<sub>H</sub>) AND (D[b][15:0] == 8000<sub>H</sub>) AND (n == 1);

mul\_res = sc ? 7FFFFFFF<sub>H</sub> : ((D[a][15:0] \* D[b][15:0]) << n);

result = D[d] - mul\_res + 8000<sub>H</sub>;

D[c] = {ssov(result,32)[31:16]}, 16'b0; // Short fraction

**MSUBRS.Q**                    **D[c], D[d], D[a] U, D[b] U, n (RRR1)**

32 - (16U \* 16U) rounded --> 32 saturated

31	28 27	24 23	18 17 16 15	12 11	8 7	0
c	d	26 <sub>H</sub>	n	b	a	63 <sub>H</sub>

sc = (D[a][31:16] == 8000<sub>H</sub>) AND (D[b][31:16] == 8000<sub>H</sub>) AND (n == 1);

mul\_res = sc ? 7FFFFFFF<sub>H</sub> : ((D[a][31:16] \* D[b][31:16]) << n);

result = D[d] - mul\_res + 8000<sub>H</sub>;

D[c] = {ssov(result,32)[31:16]}, 16'b0; // Short fraction

**Status Flags**

C	Not set by these instructions.
V	overflow = (result > 7FFFFFFF <sub>H</sub> ) OR (result < -80000000 <sub>H</sub> ); if (overflow) then PSW.V = 1 else PSW.V = 0;
SV	if (overflow) then PSW.SV = 1 else PSW.SV = PSW.SV;
AV	advanced_overflow = result[31] ^ result[30]; if (advanced_overflow) then PSW.AV = 1 else PSW.AV = 0;
SAV	if (advanced_overflow) then PSW.SAV = 1 else PSW.SAV = PSW.SAV;

**Examples**

-

**See Also**

-

## MTCR Move To Core Register

### Description

*Note: This instruction can only be executed in Supervisor mode.*

Move the value in data register D[a] to the Core Special Function Register (CSFR) selected by the value const16. The CSFR address is a const16 byte offset from the CSFR base address. It must be word-aligned (the least-significant two bits are zero). Non-aligned address have an undefined effect.

The MTCR instruction can not be used to access GPRs. Attempting to update a GPR with this instruction will have no effect.

An MTCR instruction should be followed by an ISYNC instruction. This ensures that all instructions following the MTCR see the effects of the CSFR update.

### MTCR                      const16, D[a] (RLC)

31	28 27	12 11	8 7	0
-	const16	a	CD <sub>H</sub>	

CR[const16] = D[a];

### Status Flags

C	if (const16 == FE04 <sub>H</sub> ) then PSW.C = D[a][31];
V	if (const16 == FE04 <sub>H</sub> ) then PSW.V = D[a][30];
SV	if (const16 == FE04 <sub>H</sub> ) then PSW.SV = D[a][29];
AV	if (const16 == FE04 <sub>H</sub> ) then PSW.AV = D[a][28];
SAV	if (const16 == FE04 <sub>H</sub> ) then PSW.SAV = D[a][27];

### Examples

```
mtcr 4, d1
```

### See Also

**MFCR**, **RSTV**

**MUL**  
Multiply  
**MULS**  
Multiply, Saturated

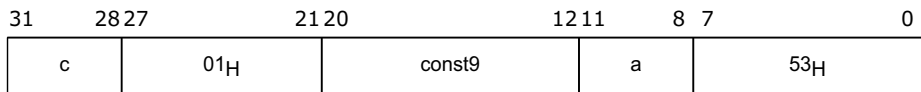
**Description**

Multiply two signed 32-bit integers and put the product into a 32-bit or 64-bit register. The value const9 is sign-extended before the multiplication is performed. The MULS result is saturated on overflow.

Multiply D[a] by D[b] (two signed 32-bit integers) and put the product into D[a].

**MUL**                                    **D[c], D[a], const9 (RC)**

(32 \* K9) --> 32 signed

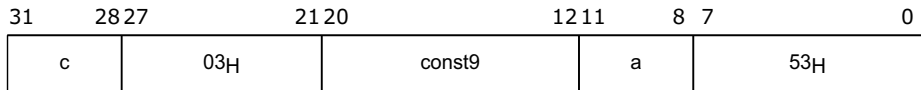


result = D[a] \* sign\_ext(const9);

D[c] = result[31:0];

**MUL**                                    **E[c], D[a], const9 (RC)**

(32 \* K9) --> 64 signed

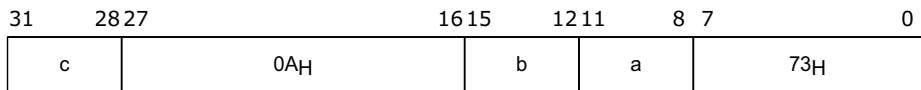


result = D[a] \* sign\_ext(const9);

E[c] = result[63:0];

**MUL**                                    **D[c], D[a], D[b] (RR2)**

(32 \* 32) --> 32 signed



result = D[a] \* D[b];

D[c] = result[31:0];

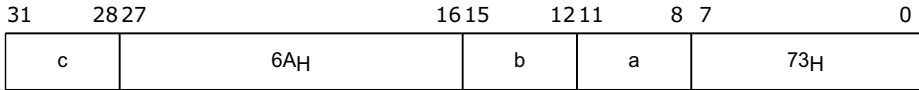
**MUL**                                    **E[c], D[a], D[b] (RR2)**

(32 \* 32) --> 64 signed

See Also Addendum for TriCore Arch Manual, Vol2, V1.3.8, pages 493/494



Instruction Set

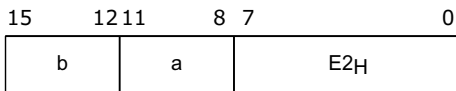


result = D[a] \* D[b];

E[c] = result[63:0];

**MUL**                                    **D[a], D[b] (SRR)**

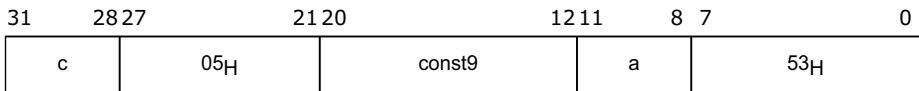
(32 \* 32) --> 32 signed



result = D[a] \* D[b];  
D[a] = result[31:0];

**MULS**                                    **D[c], D[a], const9 (RC)**

(32 \* K9) --> 32 signed saturated

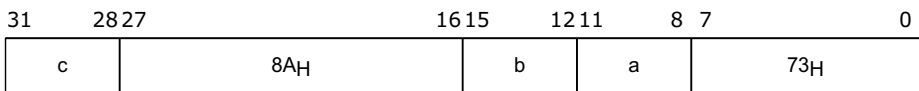


result = D[a] \* sign\_ext(const9);

D[c] = ssov(result, 32);

**MULS**                                    **D[c], D[a], D[b] (RR2)**

(32 \* 32) --> 32 signed saturated



result = D[a] \* D[b];

D[c] = ssov(result, 32);

**Status Flags**

C	Not set by these instructions.
---	--------------------------------

See Also Addendum for TriCore Arch Manual, Vol2, V1.3.8, pages 493/494

V	<p>32-bit result:  overflow = (result &gt; 7FFFFFFF<sub>H</sub>) OR (result &lt; -80000000<sub>H</sub>);  if (overflow) then PSW.V = 1 else PSW.V = 0;  64-bit result:  It is mathematically impossible to generate an overflow when multiplying two 32-bit numbers and storing the result in a 64-bit register.</p>
SV	<p>if (overflow) then PSW.SV = 1 else PSW.SV = PSW.SV;</p>
AV	<p>32-bit result:  advanced_overflow = result[31] ^ result[30];  if (advanced_overflow) then PSW.AV = 1 else PSW.AV = 0;  64-bit result:  advanced_overflow = result[63] ^ result[62];  if (advanced_overflow) then PSW.AV = 1 else PSW.AV = 0;</p>
SAV	<p>if (advanced_overflow) then PSW.SAV = 1 else PSW.SAV = PSW.SAV;</p>

### Examples

```

mul    d3, d1, d2
mul    d2, d4, #21H
mul    e2, d5, d1
muls   d2, d0, d0

mul    d3, d11

```

### See Also

[MUL.U](#), [MADD](#), [MSUB](#)

## MUL.H Packed Multiply Q Format

### Description

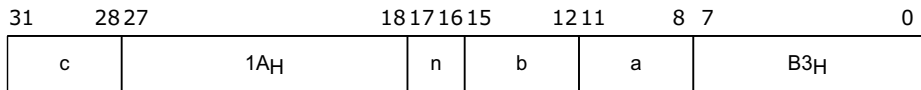
Multiply two signed 16-bit (half-word) values and put the product (left justified if  $n == 1$ ) into a 32-bit register. Note that since there are two results both halves of an extended data register are used. There are four cases of half-word multiplication:

- $16U * 16U, 16L * 16L$
- $16U * 16L, 16L * 16U$
- $16U * 16L, 16L * 16L$
- $16L * 16U, 16U * 16U$

Note that  $n$  should only take the values 0 or 1, any other value returns an undefined result. If  $(n == 1)$  then  $8000_H * 8000_H = 7FFFFFFF_H$  (for signed 16-bit \* 16-bit multiplications only).

### MUL.H E[c], D[a], D[b] LL, n (RR1)

$(16U * 16L || 16L * 16L) \rightarrow 32||32$



$sc1 = (D[a][31:16] == 8000_H) \text{ AND } (D[b][15:0] == 8000_H) \text{ AND } (n == 1);$

$sc0 = (D[a][15:0] == 8000_H) \text{ AND } (D[b][15:0] == 8000_H) \text{ AND } (n == 1);$

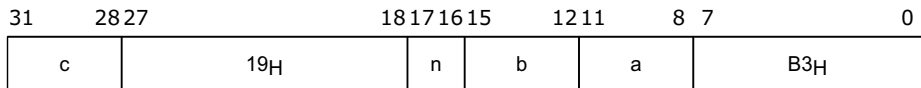
$result\_word1 = sc1 ? 7FFFFFFF_H : ((D[a][31:16] * D[b][15:0]) \ll n);$

$result\_word0 = sc0 ? 7FFFFFFF_H : ((D[a][15:0] * D[b][15:0]) \ll n);$

$E[c] = \{result\_word1[31:0], result\_word0[31:0]\}; // \text{ Packed fraction}$

### MUL.H E[c], D[a], D[b] LU, n (RR1)

$(16U * 16L || 16L * 16U) \rightarrow 32||32$



$sc1 = (D[a][31:16] == 8000_H) \text{ AND } (D[b][15:0] == 8000_H) \text{ AND } (n == 1);$

$sc0 = (D[a][15:0] == 8000_H) \text{ AND } (D[b][31:16] == 8000_H) \text{ AND } (n == 1);$

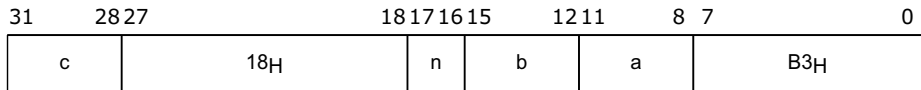
$result\_word1 = sc1 ? 7FFFFFFF_H : ((D[a][31:16] * D[b][15:0]) \ll n);$

$result\_word0 = sc0 ? 7FFFFFFF_H : ((D[a][15:0] * D[b][31:16]) \ll n);$

$E[c] = \{result\_word1[31:0], result\_word0[31:0]\}; // \text{ Packed fraction}$

**MUL.H** **E[c], D[a], D[b] UL, n (RR1)**

(16U \* 16U || 16L \* 16L) --> 32||32



sc1 = (D[a][31:16] == 8000<sub>H</sub>) AND (D[b][31:16] == 8000<sub>H</sub>) AND (n == 1);

sc0 = (D[a][15:0] == 8000<sub>H</sub>) AND (D[b][15:0] == 8000<sub>H</sub>) AND (n == 1);

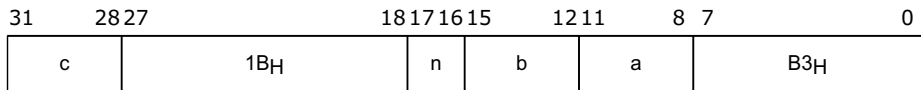
result\_word1 = sc1 ? 7FFFFFFF<sub>H</sub> : ((D[a][31:16] \* D[b][31:16]) << n);

result\_word0 = sc0 ? 7FFFFFFF<sub>H</sub> : ((D[a][15:0] \* D[b][15:0]) << n);

E[c] = {result\_word1[31:0], result\_word0[31:0]}; // Packed fraction

**MUL.H** **E[c], D[a], D[b] UU, n (RR1)**

(16L \* 16U || 16U \* 16U) --> 32||32



sc1 = (D[a][15:0] == 8000<sub>H</sub>) AND (D[b][31:16] == 8000<sub>H</sub>) AND (n == 1);

sc0 = (D[a][31:16] == 8000<sub>H</sub>) AND (D[b][31:16] == 8000<sub>H</sub>) AND (n == 1);

result\_word1 = sc1 ? 7FFFFFFF<sub>H</sub> : ((D[a][15:0] \* D[b][31:16]) << n);

result\_word0 = sc0 ? 7FFFFFFF<sub>H</sub> : ((D[a][31:16] \* D[b][31:16]) << n);

E[c] = {result\_word1[31:0], result\_word0[31:0]}; // Packed fraction

**Status Flags**

C	Not set by this instruction.
V	The PSW.V status bit is cleared.
SV	Not set by this instruction.
AV	aov_word1 = result_word1[31] ^ result_word1[30]; aov_word0 = result_word0[31] ^ result_word0[30]; advanced_overflow = aov_word1 OR aov_word0; if (advanced_overflow) then PSW.AV = 1 else PSW.AV = 0;
SAV	if (advanced_overflow) then PSW.SV = 1 else PSW.SV = PSW.SV;

**Examples**

-

**See Also**

-

See Also Addendum for TriCore Arch Manual, Vol2, V1.3.8, pages 493/494

## MUL.Q Multiply Q Format

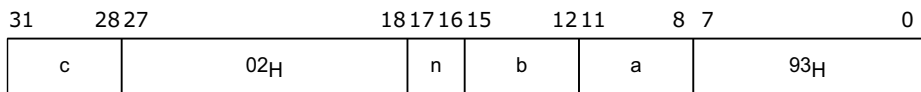
### Description

Multiply two signed 16-bit or 32-bit values and put the product (left justified if  $n == 1$ ) into a 32-bit or 64-bit register. There are two cases of  $16 * 16$  operations, four cases of  $16 * 32$  operations and two cases of  $32 * 32$  operations.

Note that  $n$  should only take the values 0 or 1, any other value returns an undefined result. If  $(n == 1)$  then  $8000_H * 8000_H = 7FFFFFFF_H$  (for signed 16-bit \* 16-bit multiplications only).

### MUL.Q                    D[c], D[a], D[b], n (RR1)

(32 \* 32)Up --> 32

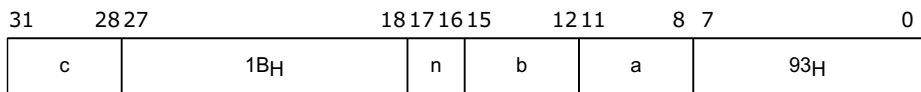


result = ((D[a] \* D[b]) << n) >> 32;

D[c] = result[31:0]; // Fraction

### MUL.Q                    E[c], D[a], D[b], n (RR1)

(32 \* 32) --> 64

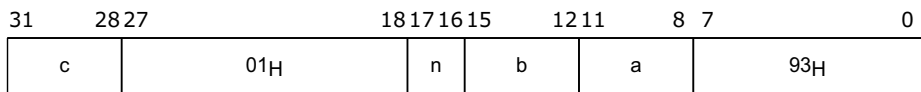


result = (D[a] \* D[b]) << n;

E[c] = result[63:0]; // Multi-precision fraction

### MUL.Q                    D[c], D[a], D[b] L, n (RR1)

(32 \* 16L)Up --> 32



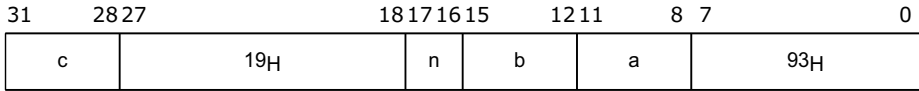
result = ((D[a] \* D[b][15:0]) << n) >> 16;

D[c] = result[31:0]; // Fraction

### MUL.Q                    E[c], D[a], D[b] L, n (RR1)

(32 \* 16L) --> 64

See Also Addendum for TriCore Arch Manual, Vol2, V1.3.8, pages 493/494

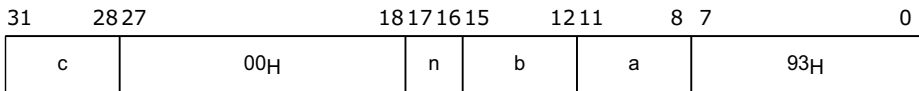


result = (D[a] \* D[b][15:0]) << n;

E[c] = result[63:0]; // Multi-precision accumulator

**MUL.Q**                      **D[c], D[a], D[b] U, n (RR1)**

(32 \* 16U)Up --> 32

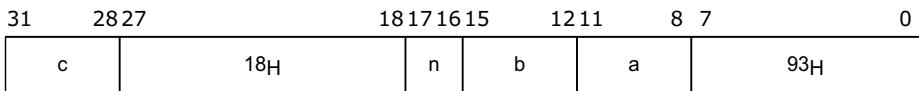


result = ((D[a] \* D[b][31:16]) << n) >> 16;

D[c] = result[31:0]; // Fraction

**MUL.Q**                      **E[c], D[a], D[b] U, n (RR1)**

(32 \* 16U) --> 64

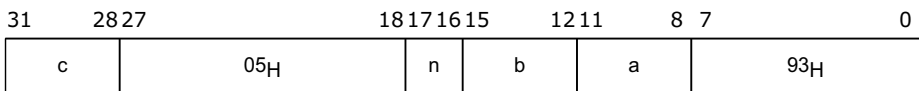


result = (D[a] \* D[b][31:16]) << n;

E[c] = result[63:0]; // Multi-precision accumulator

**MUL.Q**                      **D[c], D[a] L, D[b] L, n (RR1)**

(16L \* 16L) --> 32



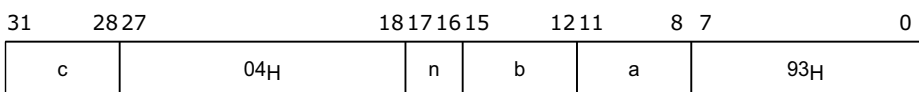
sc = (D[a][15:0] == 8000<sub>H</sub>) AND (D[b][15:0] == 8000<sub>H</sub>) AND (n == 1);

result = sc ? 7FFFFFFF<sub>H</sub> : ((D[a][15:0] \* D[b][15:0]) << n);

D[c] = result[31:0]; // Fraction

**MUL.Q**                      **D[c], D[a] U, D[b] U, n (RR1)**

(16U \* 16U) --> 32



```
sc = (D[a][31:16] == 8000H) AND (D[b][31:16] == 8000H) AND (n == 1);
result = sc ? 7FFFFFFFH : ((D[a][31:16] * D[b][31:16]) << n);
D[c] = result[31:0]; // Fraction
```

### Status Flags

C	Not set by this instruction.
V	32-bit result: overflow = (result > 7FFFFFFF <sub>H</sub> ) OR (result < -80000000 <sub>H</sub> ); if (overflow) then PSW.V = 1 else PSW.V = 0; 64-bit result: overflow = (result > 7FFFFFFFFFFFFFFF <sub>H</sub> ) OR (result < -8000000000000000 <sub>H</sub> ); if (overflow) then PSW.V = 1 else PSW.V = 0;
SV	if (overflow) then PSW.SV = 1 else PSW.SV = PSW.SV;
AV	32-bit result: advanced_overflow = result[31] ^ result[30]; if (advanced_overflow) then PSW.AV = 1 else PSW.AV = 0; 64-bit result: advanced_overflow = result[63] ^ result[62]; if (advanced_overflow) then PSW.AV = 1 else PSW.AV = 0;
SAV	if (advanced_overflow) then PSW.SAV = 1 else PSW.SAV = PSW.SAV;

### Examples

```
mul.q d3, d1U, d2U, #1
mul.q d3, d1L, d2L, #1
mul.q d3, d1, d2U, #0
mul.q d3, d1, d2L, #1
mul.q d2, d1, d2, #1
mul.q e2, d1, d0U, #1
mul.q e2, d1, d0L, #1
mul.q e2, d1, d7, #1
```

### See Also

-



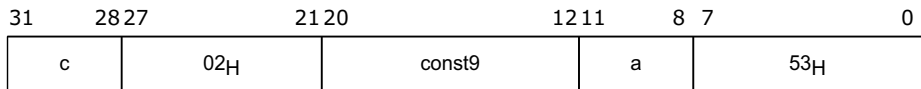
**MUL.U**  
Multiply Unsigned  
**MULS.U**  
Multiply Unsigned, Saturated

**Description**

Multiply two unsigned 32-bit integers and put the product into a 32-bit or 64-bit register. The value const9 (instruction format RC) is zero-extended before the multiplication is performed. The MULS.U result is saturated on overflow.

**MUL.U**                    **E[c], D[a], const9 (RC)**

(32 \* K9) --> 64 unsigned

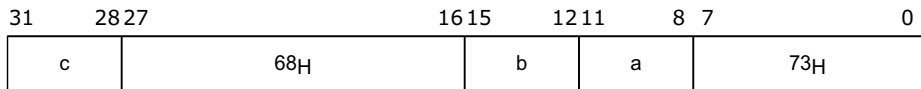


result = D[a] \* zero\_ext(const9); // unsigned

E[c] = result[63:0];

**MUL.U**                    **E[c], D[a], D[b] (RR2)**

(32 \* 32) --> 64 unsigned

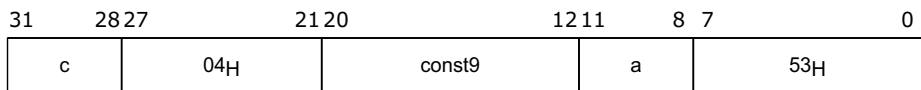


result = D[a] \* D[b]; // unsigned

E[c] = result[63:0];

**MULS.U**                    **D[c], D[a], const9 (RC)**

(32 \* K9) --> 32 unsigned saturated



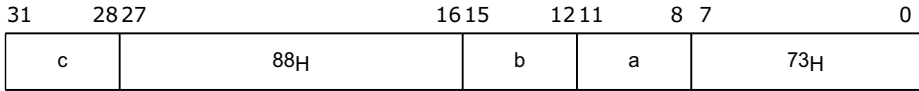
result = D[a] \* zero\_ext(const9); // unsigned

D[c] = suov(result, 32);

**MULS.U**                    **D[c], D[a], D[b] (RR2)**

(32 \* 32) --> 32 unsigned saturated

See Also Addendum for TriCore Arch Manual, Vol2, V1.3.8, pages 493/494



result = D[a] \* D[b]; // unsigned

D[c] = suov(result, 32);

### Status Flags

C	Not set by this instruction.
V	32-bit result: overflow = (result > FFFFFFFF <sub>H</sub> ) OR (result < 00000000 <sub>H</sub> ); if (overflow) then PSW.V = 1 else PSW.V = 0; 64-bit result: It is mathematically impossible to generate an overflow when multiplying two 32-bit numbers and storing the result in a 64-bit register.
SV	if (overflow) then PSW.SV = 1 else PSW.SV = PSW.SV;
AV	32-bit result: advanced_overflow = result[31] ^ result[30]; if (advanced_overflow) then PSW.AV = 1 else PSW.AV = 0; 64-bit result: advanced_overflow = result[63] ^ result[62]; if (advanced_overflow) then PSW.AV = 1 else PSW.AV = 0;
SAV	if (advanced_overflow) then PSW.SAV = 1 else PSW.SAV = PSW.SAV;

### Examples

```
mul.u    e0, d2, d3
muls.u   d3, d5, d9
```

### See Also

[MUL](#)

## MULM.H

### Packed Multiply Q Format-Multi-precision

#### Description

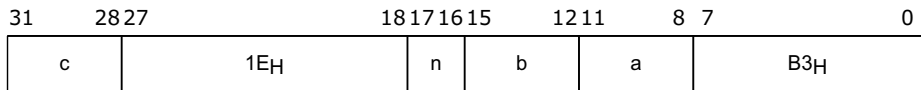
Perform two multiplications of two signed 16-bit (half-word) values. Add the two products (left justified if  $n == 1$ ) left-shifted by 16, in a 64-bit register. There are four cases of half-word multiplication:

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

Note that  $n$  should only take the values 0 or 1, any other value returns an undefined result. If ( $n == 1$ ) then  $8000_H * 8000_H = 7FFFFFFF_H$  (for signed 16-bit \* 16-bit multiplications only).

#### MULM.H                      E[c], D[a], D[b] LL, n (RR1)

16U \* 16L + 16L \* 16L --> 64



$sc1 = (D[a][31:16] == 8000_H) \text{ AND } (D[b][15:0] == 8000_H) \text{ AND } (n == 1);$

$sc0 = (D[a][15:0] == 8000_H) \text{ AND } (D[b][15:0] == 8000_H) \text{ AND } (n == 1);$

$result\_word1 = sc1 ? 7FFFFFFF_H : ((D[a][31:16] * D[b][15:0]) \ll n);$

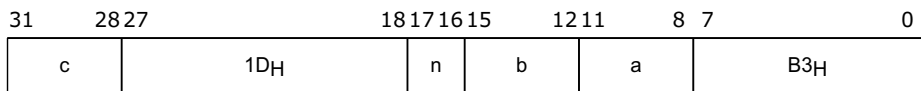
$result\_word0 = sc0 ? 7FFFFFFF_H : ((D[a][15:0] * D[b][15:0]) \ll n);$

$result = (result\_word1 + result\_word0) \ll 16;$

$E[c] = result[63:0]; // \text{ Multi-precision accumulator}$

#### MULM.H                      E[c], D[a], D[b] LU, n (RR1)

16U \* 16L + 16L \* 16U --> 64



$sc1 = (D[a][31:16] == 8000_H) \text{ AND } (D[b][15:0] == 8000_H) \text{ AND } (n == 1);$

$sc0 = (D[a][15:0] == 8000_H) \text{ AND } (D[b][31:16] == 8000_H) \text{ AND } (n == 1);$

$result\_word1 = sc1 ? 7FFFFFFF_H : ((D[a][31:16] * D[b][15:0]) \ll n);$

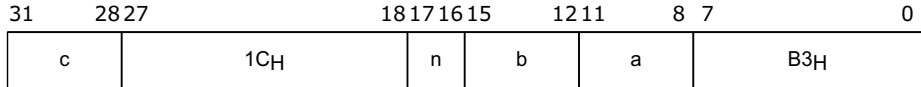
$result\_word0 = sc0 ? 7FFFFFFF_H : ((D[a][15:0] * D[b][31:16]) \ll n);$

$result = (result\_word1 + result\_word0) \ll 16;$

E[c] = result[63:0]; // Multi-precision accumulator

**MULM.H**                    **E[c], D[a], D[b] UL, n (RR1)**

16U \* 16U + 16L \* 16L --> 64



sc1 = (D[a][31:16] == 8000<sub>H</sub>) AND (D[b][31:16] == 8000<sub>H</sub>) AND (n == 1);

sc0 = (D[a][15:0] == 8000<sub>H</sub>) AND (D[b][15:0] == 8000<sub>H</sub>) AND (n == 1);

result\_word1 = sc1 ? 7FFFFFFF<sub>H</sub> : ((D[a][31:16] \* D[b][31:16]) << n);

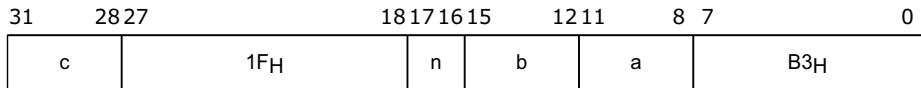
result\_word0 = sc0 ? 7FFFFFFF<sub>H</sub> : ((D[a][15:0] \* D[b][15:0]) << n);

result = (result\_word1 + result\_word0) << 16;

E[c] = result[63:0]; // Multi-precision accumulator

**MULM.H**                    **E[c], D[a], D[b] UU, n (RR1)**

16L \* 16U + 16U \* 16U --> 64



sc1 = (D[a][15:0] == 8000<sub>H</sub>) AND (D[b][31:16] == 8000<sub>H</sub>) AND (n == 1);

sc0 = (D[a][31:16] == 8000<sub>H</sub>) AND (D[b][31:16] == 8000<sub>H</sub>) AND (n == 1);

result\_word1 = sc1 ? 7FFFFFFF<sub>H</sub> : ((D[a][15:0] \* D[b][31:16]) << n);

result\_word0 = sc0 ? 7FFFFFFF<sub>H</sub> : ((D[a][31:16] \* D[b][31:16]) << n);

result = (result\_word1 + result\_word0) << 16;

E[c] = result[63:0]; // Multi-precision accumulator

**Status Flags**

C	Not set by this instruction.
V	The PSW.V status bit is cleared.
SV	Not set by this instruction.
AV	The PSW.AV status bit is cleared.
SAV	Not set by this instruction.

**Examples**

-

**See Also**

-

See Also Addendum for TriCore Arch Manual, Vol2, V1.3.8, pages 493/494

## MULR.H Packed Multiply Q Format with Rounding

### Description

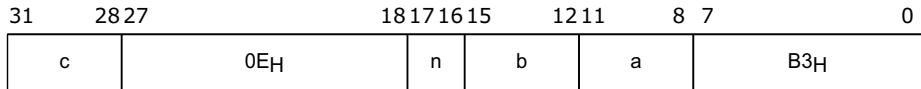
Multiply two signed 16-bit (half-word) values. Add the product (left justified if  $n == 1$ ) to a signed 16-bit value and put the rounded result into half of a 32-bit register. Note that since there are two results the two register halves are used). There are four cases of half-word multiplication:

- $16U * 16U, 16L * 16L$
- $16U * 16L, 16L * 16U$
- $16U * 16L, 16L * 16L$
- $16L * 16U, 16U * 16U$

Note that  $n$  should only take the values 0 or 1, any other value returns an undefined result. If  $(n == 1)$  then  $8000_H * 8000_H = 7FFFFFFF_H$  (for signed 16-bit \* 16-bit multiplications only).

### MULR.H                      D[c], D[a], D[b] LL, n (RR1)

$(16U * 16L \parallel 16L * 16L)$  rounded  $\rightarrow 16 \parallel 16$



$sc1 = (D[a][31:16] == 8000_H) \text{ AND } (D[b][15:0] == 8000_H) \text{ AND } (n == 1);$

$sc0 = (D[a][15:0] == 8000_H) \text{ AND } (D[b][15:0] == 8000_H) \text{ AND } (n == 1);$

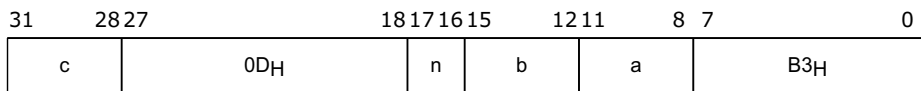
$result\_halfword1 = sc1 ? 7FFFFFFF_H : (((D[a][31:16] * D[b][15:0]) << n) + 8000_H);$

$result\_halfword0 = sc0 ? 7FFFFFFF_H : (((D[a][15:0] * D[b][15:0]) << n) + 8000_H);$

$D[c] = \{result\_halfword1[31:16], result\_halfword0[31:16]\}; // \text{Packed short fraction}$

### MULR.H                      D[c], D[a], D[b] LU, n (RR1)

$(16U * 16L \parallel 16L * 16U)$  rounded  $\rightarrow 16 \parallel 16$



$sc1 = (D[a][31:16] == 8000_H) \text{ AND } (D[b][15:0] == 8000_H) \text{ AND } (n == 1);$

$sc0 = (D[a][15:0] == 8000_H) \text{ AND } (D[b][31:16] == 8000_H) \text{ AND } (n == 1);$

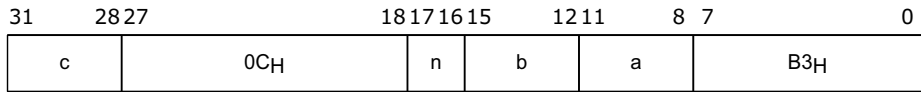
$result\_halfword1 = sc1 ? 7FFFFFFF_H : (((D[a][31:16] * D[b][15:0]) << n) + 8000_H);$

$result\_halfword0 = sc0 ? 7FFFFFFF_H : (((D[a][15:0] * D[b][31:16]) << n) + 8000_H);$

$D[c] = \{result\_halfword1[31:16], result\_halfword0[31:16]\}; // \text{Packed short fraction}$

**MULR.H**                      **D[c], D[a], D[b] UL, n (RR1)**

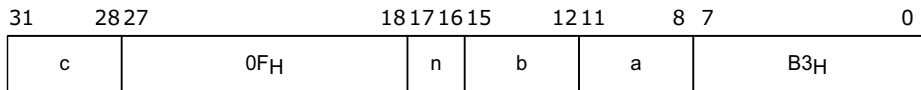
(16U \* 16U || 16L \* 16L) rounded --> 16||16



sc1 = (D[a][31:16] == 8000<sub>H</sub>) AND (D[b][31:16] == 8000<sub>H</sub>) AND (n == 1);  
 sc0 = (D[a][15:0] == 8000<sub>H</sub>) AND (D[b][15:0] == 8000<sub>H</sub>) AND (n == 1);  
 result\_halfword1 = sc1 ? 7FFFFFFF<sub>H</sub> : (((D[a][31:16] \* D[b][31:16]) << n) + 8000<sub>H</sub>);  
 result\_halfword0 = sc0 ? 7FFFFFFF<sub>H</sub> : (((D[a][15:0] \* D[b][15:0]) << n) + 8000<sub>H</sub>);  
 D[c] = {result\_halfword1[31:16], result\_halfword0[31:16]}; // Packed short fraction

**MULR.H**                      **D[c], D[a], D[b] UU, n (RR1)**

(16L \* 16U || 16U \* 16U) rounded --> 16||16



sc1 = (D[a][15:0] == 8000<sub>H</sub>) AND (D[b][31:16] == 8000<sub>H</sub>) AND (n == 1);  
 sc0 = (D[a][31:16] == 8000<sub>H</sub>) AND (D[b][31:16] == 8000<sub>H</sub>) AND (n == 1);  
 result\_halfword1 = sc1 ? 7FFFFFFF<sub>H</sub> : (((D[a][15:0] \* D[b][31:16]) << n) + 8000<sub>H</sub>);  
 result\_halfword0 = sc0 ? 7FFFFFFF<sub>H</sub> : (((D[a][31:16] \* D[b][31:16]) << n) + 8000<sub>H</sub>);  
 D[c] = {result\_halfword1[31:16], result\_halfword0[31:16]}; // Packed short fraction

**Status Flags**

C	Not set by this instruction.
V	The PSW.V status bit is cleared.
SV	Not set by this instruction.
AV	aov_halfword1 = result_halfword1[31] ^ result_halfword1[30]; aov_halfword0 = result_halfword0[31] ^ result_halfword0[30]; advanced_overflow = aov_halfword1 OR aov_halfword0; if (advanced_overflow) then PSW.AV = 1 else PSW.AV = 0;
SAV	if (advanced_overflow) then PSW.SAV = 1 else PSW.SAV = PSW.SAV;

**Examples**

-

**See Also**

-

See Also Addendum for TriCore Arch Manual, Vol2, V1.3.8, pages 493/494



## MULR.Q

### Multiply Q Format with Rounding

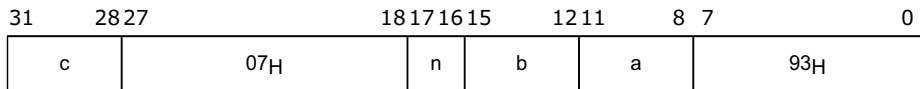
#### Description

Multiply two signed 16-bit (half-word) values and put the rounded result (left justified if  $n == 1$ ) into a 32-bit register. The lower half-word is cleared.

Note that  $n$  should only take the values 0 or 1, any other value returns an undefined result. If ( $n == 1$ ) then  $8000_H * 8000_H = 7FFFFFFF_H$  (for signed 16-bit \* 16-bit multiplications only).

#### MULR.Q                      D[c], D[a] L, D[b] L, n (RR1)

(16L \* 16L) rounded --> 32



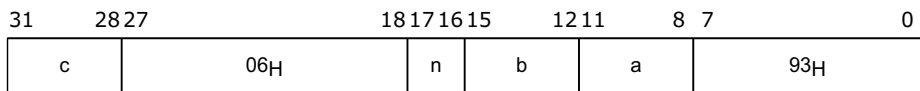
$sc = (D[a][15:0] == 8000_H) \text{ AND } (D[b][15:0] == 8000_H) \text{ AND } (n == 1);$

$result = sc ? 7FFFFFFF_H : (((D[a][15:0] * D[b][15:0]) << n) + 8000_H);$

$D[c] = \{result[31:16], 16'b0\}; // \text{ Short fraction}$

#### MULR.Q                      D[c], D[a] U, D[b] U, n (RR1)

(16U \* 16U) rounded --> 32



$sc = (D[a][31:16] == 8000_H) \text{ AND } (D[b][31:16] == 8000_H) \text{ AND } (n == 1);$

$result = sc ? 7FFFFFFF_H : (((D[a][31:16] * D[b][31:16]) << n) + 8000_H);$

$D[c] = \{result[31:16], 16'b0\}; // \text{ Short fraction}$

#### Status Flags

C	Not set by this instruction.
V	The PSW.V status bit is cleared.
SV	Not set by this instruction.
AV	$advanced\_overflow = result[31] \wedge result[30];$ if (advanced_overflow) then PSW.AV = 1 else PSW.AV = 0;
SAV	if (advanced_overflow) then PSW.SAV = 1 else PSW.SAV = PSW.SAV;

**Examples**

-

**See Also**

-

See Also Addendum for TriCore Arch Manual, Vol2, V1.3.8, pages 493/494

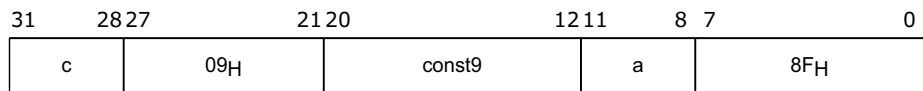
## NAND

### Bitwise NAND

#### Description

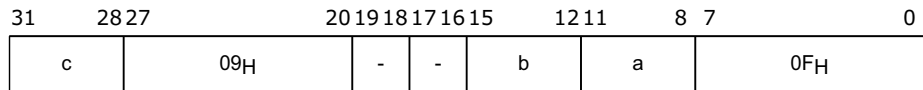
Compute the bitwise NAND of the contents of data register D[a] and either data register D[b] (instruction format RR) or const9 (instruction format RC). Put the result in data register D[c]. The const9 value is zero-extended.

#### NAND D[c], D[a], const9 (RC)



$D[c] = \sim(D[a] \& \text{zero\_ext}(\text{const9}));$

#### NAND D[c], D[a], D[b] (RR)



$D[c] = \sim(D[a] \& D[b]);$

#### Status Flags

C	Not set by this instruction.
V	Not set by this instruction.
SV	Not set by this instruction.
AV	Not set by this instruction.
SAV	Not set by this instruction.

#### Examples

```
nand d3, d1, d2
nand d3, d1, #126
```

#### See Also

[AND](#), [ANDN](#), [NOR](#), [NOT \(16-bit\)](#), [OR](#), [ORN](#), [XNOR](#), [XOR](#)

## NAND.T

### Bit Logical NAND

#### Description

Compute the logical NAND of bit pos1 of data register D[a], and bit pos2 of data register D[b]. Put the result in the least-significant bit of data register D[c] and clear the remaining bits of D[c] to zero.

#### NAND.T                      D[c], D[a], pos1, D[b], pos2 (BIT)

31	28 27	23 22 21 20	16 15	12 11	8 7	0
c	pos2	00H	pos1	b	a	07H

result = !(D[a][pos1] AND D[b][pos2]);

D[c] = zero\_ext(result);

#### Status Flags

C	Not set by this instruction.
V	Not set by this instruction.
SV	Not set by this instruction.
AV	Not set by this instruction.
SAV	Not set by this instruction.

#### Examples

```
nand.t    d3, d1, 2, d2, #4
```

#### See Also

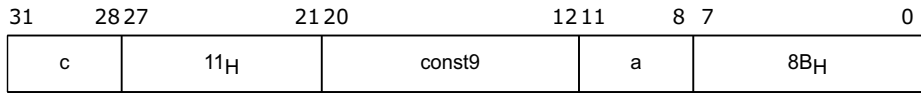
[AND.T](#), [ANDN.T](#), [OR.T](#), [ORN.T](#), [XNOR.T](#), [XOR.T](#)

**NE**  
**Not Equal**

**Description**

If the contents of data register D[a] are not equal to the contents of either data register D[b] (instruction format RR) or const9 (instruction format RC), set the least-significant bit of D[c] to one and clear the remaining bits to zero; otherwise clear all bits in D[c]. The const9 value is sign-extended.

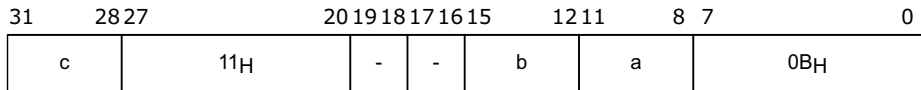
**NE D[c], D[a], const9 (RC)**



result = (D[a] != sign\_ext(const9));

D[c] = zero\_ext(result);

**NE D[c], D[a], D[b] (RR)**



result = (D[a] != D[b]);

D[c] = zero\_ext(result);

**Status Flags**

C	Not set by this instruction.
V	Not set by this instruction.
SV	Not set by this instruction.
AV	Not set by this instruction.
SAV	Not set by this instruction.

**Examples**

```
ne d3, d1, d2
ne d3, d1, #126
```

**See Also**

[EQ](#), [GE](#), [GE.U](#), [LT](#), [LT.U](#), [EQANY.B](#), [EQANY.H](#), [NEZ.A](#)

See Also Addendum for TriCore Arch Manual, Vol2, V1.3.8, pages 493/494

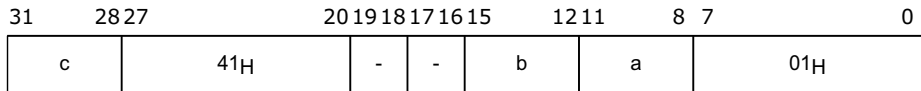
## NE.A

### Not Equal Address

#### Description

If the contents of address registers A[a] and A[b] are not equal, set the least-significant bit of D[c] to one and clear the remaining bits to zero; otherwise clear all bits in D[c].

#### NE.A                      D[c], A[a], A[b] (RR)



D[c] = (A[a] != A[b]);

#### Status Flags

C	Not set by this instruction.
V	Not set by this instruction.
SV	Not set by this instruction.
AV	Not set by this instruction.
SAV	Not set by this instruction.

#### Examples

ne.a    d3, a4, a2

#### See Also

[EQ.A](#), [EQZ.A](#), [GE.A](#), [LT.A](#), [NEZ.A](#)

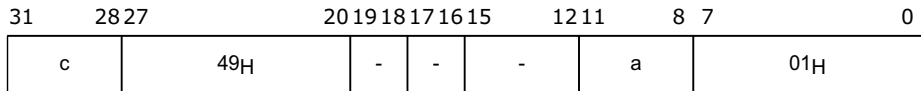
## NEZ.A

### Not Equal Zero Address

#### Description

If the contents of address register A[a] are not equal to zero, set the least significant bit of D[c] to one and clear the remaining bits to zero; otherwise clear all bits in D[c].

#### NEZ.A                      D[c], A[a] (RR)



D[c] = (A[a] != 0);

#### Status Flags

C	Not set by this instruction.
V	Not set by this instruction.
SV	Not set by this instruction.
AV	Not set by this instruction.
SAV	Not set by this instruction.

#### Examples

nez.a    d3, a4

#### See Also

[EQ.A](#), [EQZ.A](#), [GE.A](#), [LT.A](#), [NE](#)

See Also Addendum for TriCore Arch Manual, Vol2, V1.3.8, pages 493/494

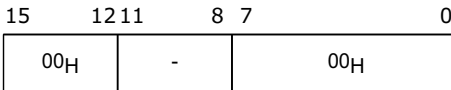
**NOP**  
**No Operation**

**Description**

Used to implement efficient low-power, non-operational instructions.

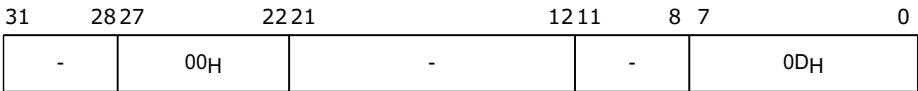
Used to implement efficient low-power, non-operational instructions.

**NOP (SR)**



No operation.

**NOP (SYS)**



No operation.

**Status Flags**

C	Not set by this instruction.
V	Not set by this instruction.
SV	Not set by this instruction.
AV	Not set by this instruction.
SAV	Not set by this instruction.

**Examples**

nop

nop

**See Also**

-

See Also Addendum for TriCore Arch Manual, Vol2, V1.3.8, pages 493/494



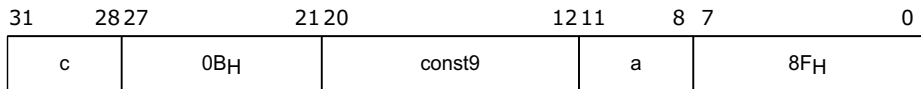
## NOR

### Bitwise NOR

#### Description

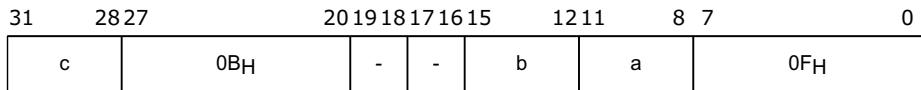
Compute the bitwise NOR of the contents of data register D[a] and the contents of either data register D[b] (instruction format RR) or const9 (instruction format RC) and put the result in data register D[c]. The const9 value is zero-extended.

#### NOR D[c], D[a], const9 (RC)



$$D[c] = \sim(D[a] \mid \text{zero\_ext}(\text{const9}));$$

#### NOR D[c], D[a], D[b] (RR)



$$D[c] = \sim(D[a] \mid D[b]);$$

#### Status Flags

C	Not set by this instruction.
V	Not set by this instruction.
SV	Not set by this instruction.
AV	Not set by this instruction.
SAV	Not set by this instruction.

#### Examples

```
nor d3, d1, d2
nor d3, d1, #126
```

#### See Also

[AND](#), [ANDN](#), [NAND](#), [NOT \(16-bit\)](#), [OR](#), [ORN](#), [XNOR](#), [XOR](#)

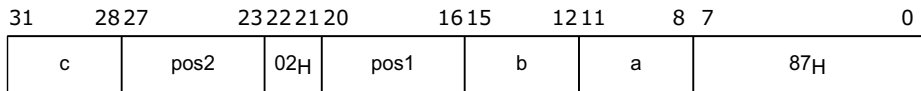
## NOR.T

### Bit Logical NOR

#### Description

Compute the logical NOR of bit pos1 of data register D[a] and bit pos2 of data register D[b]. Put the result in the least-significant bit of data register D[c] and clear the remaining bits of D[c] to zero.

#### NOR.T                      D[c], D[a], pos1, D[b], pos2 (BIT)



result = !(D[a][pos1] OR D[b][pos2]);

D[c] = zero\_ext(result);

#### Status Flags

C	Not set by this instruction.
V	Not set by this instruction.
SV	Not set by this instruction.
AV	Not set by this instruction.
SAV	Not set by this instruction.

#### Examples

```
nor.t    d3, d1, 5, d2, #3
```

#### See Also

[AND.T](#), [ANDN.T](#), [NAND.T](#), [OR.T](#), [ORN.T](#), [XNOR.T](#), [XOR.T](#)

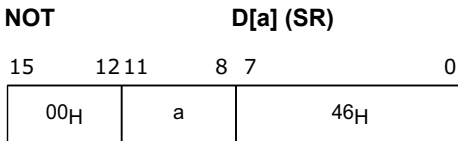
See Also Addendum for TriCore Arch Manual, Vol2, V1.3.8, pages 493/494

## NOT (16-bit)

### Bitwise Complement NOT (16-bit)

#### Description

Compute the bitwise NOT of the contents of register D[a] and put the result in data register D[a].



$D[a] = \sim D[a];$

#### Status Flags

C	Not set by this instruction.
V	Not set by this instruction.
SV	Not set by this instruction.
AV	Not set by this instruction.
SAV	Not set by this instruction.

#### Examples

```
not d2
```

#### See Also

[AND](#), [ANDN](#), [NAND](#), [NOR](#), [ORN](#), [XNOR](#), [XOR](#)

## OR

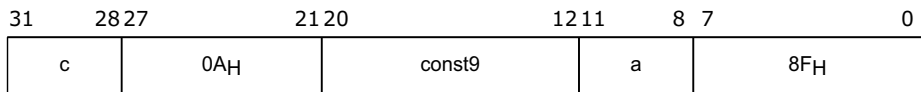
### Bitwise OR

#### Description

Compute the bitwise OR of the contents of data register D[a] and the contents of either data register D[b] (instruction format RR) or const9 (instruction format RC). Put the result in data register D[c]. The const9 value is zero-extended.

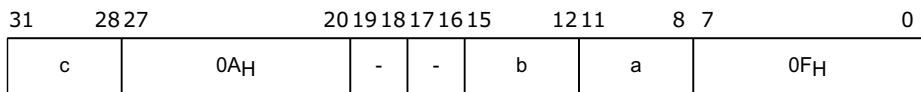
Compute the bitwise OR of the contents of either data register D[a] (instruction format SRR) or D[15] (instruction format SC) and the contents of either data register D[b] (format SRR) or const8 (format SC). Put the result in either data register D[a] (format SRR) or D[15] (format SC). The const8 value is zero-extended.

#### OR D[c], D[a], const9 (RC)



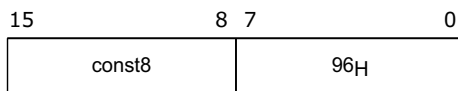
D[c] = D[a] | zero\_ext(const9);

#### OR D[c], D[a], D[b] (RR)



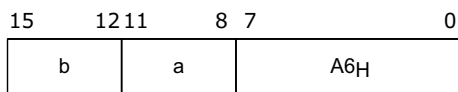
D[c] = D[a] | D[b];

#### OR D[15], const8 (SC)



D[15] = D[15] | zero\_ext(const8);

#### OR D[a], D[b] (SRR)



D[a] = D[a] | D[b];

### Status Flags

C	Not set by this instruction.
V	Not set by this instruction.
SV	Not set by this instruction.
AV	Not set by this instruction.
SAV	Not set by this instruction.

### Examples

```
or    d3, d1, d2
or    d3, d1, #126
```

```
or    d1, d2
or    d15, #126
```

### See Also

[AND](#), [ANDN](#), [NAND](#), [NOR](#), [NOT \(16-bit\)](#), [ORN](#), [XNOR](#), [XOR](#)

See Also Addendum for TriCore Arch Manual, Vol2, V1.3.8, pages 493/494

**OR.AND.T**

Accumulating Bit Logical OR-AND

**OR.ANDN.T**

Accumulating Bit Logical OR-AND-Not

**OR.NOR.T**

Accumulating Bit Logical OR-NOR

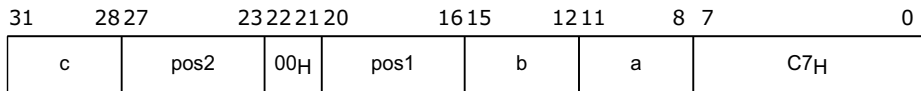
**OR.OR.T**

Accumulating Bit Logical OR-OR

**Description**

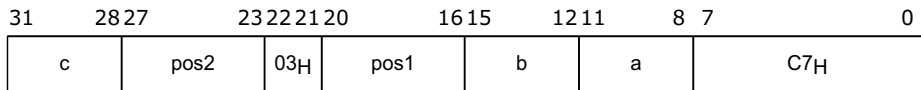
Compute the logical operation (AND, ANDN, NOR or OR as appropriate) of the value of bit pos1 of data register D[a], and bit pos2 of D[b]. Compute the logical OR of that result and bit [0] of D[c]. Put the result back in bit [0] of D[c]. All other bits in D[c] are unchanged.

**OR.AND.T                    D[c], D[a], pos1, D[b], pos2 (BIT)**



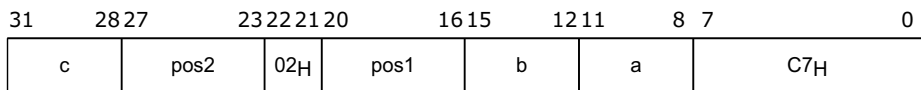
$$D[c] = \{D[c][31:1], D[c][0] \text{ OR } (D[a][pos1] \text{ AND } D[b][pos2])\};$$

**OR.ANDN.T                    D[c], D[a], pos1, D[b], pos2 (BIT)**



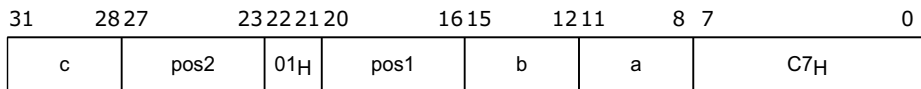
$$D[c] = \{D[c][31:1], D[c][0] \text{ OR } (D[a][pos1] \text{ AND } !D[b][pos2])\};$$

**OR.NOR.T                    D[c], D[a], pos1, D[b], pos2 (BIT)**



$$D[c] = \{D[c][31:1], D[c][0] \text{ OR } !(D[a][pos1] \text{ OR } D[b][pos2])\};$$

**OR.OR.T                    D[c], D[a], pos1, D[b], pos2 (BIT)**



$$D[c] = \{D[c][31:1], D[c][0] \text{ OR } (D[a][pos1] \text{ OR } D[b][pos2])\};$$

### Status Flags

C	Not set by these instructions.
V	Not set by these instructions.
SV	Not set by these instructions.
AV	Not set by these instructions.
SAV	Not set by these instructions.

### Examples

```

or.and.t      d3, d1, 3, d2, 5
or.andn.t    d3, d1, 3, d2, 5
or.nor.t     d3, d1, 3, d2, 5
or.or.t      d3, d1, 3, d2, 5

```

### See Also

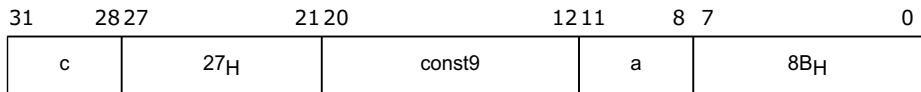
[AND.AND.T](#), [AND.ANDN.T](#), [AND.NOR.T](#), [AND.OR.T](#), [SH.AND.T](#), [SH.ANDN.T](#), [SH.NAND.T](#), [SH.NOR.T](#), [SH.OR.T](#), [SH.ORN.T](#), [SH.XNOR.T](#)

## OR.EQ Equal Accumulating

### Description

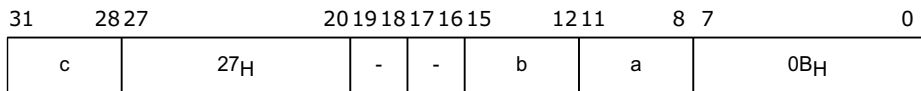
Compute the logical OR of D[c][0] and the Boolean result of the EQ operation on the contents of data register D[a] and either data register D[b] (instruction format RR) or const9 (instruction format RC). Put the result in D[c][0]. All other bits in D[c] are unchanged. The const9 value is sign-extended.

### OR.EQ D[c], D[a], const9 (RC)



$D[c] = \{D[c][31:1], D[c][0] \text{ OR } (D[a] == \text{sign\_ext}(\text{const9}))\};$

### OR.EQ D[c], D[a], D[b] (RR)



$D[c] = \{D[c][31:1], D[c][0] \text{ OR } (D[a] == D[b])\};$

### Status Flags

C	Not set by this instruction.
V	Not set by this instruction.
SV	Not set by this instruction.
AV	Not set by this instruction.
SAV	Not set by this instruction.

### Examples

```
or.eq d3, d1, d2
or.eq d3, d1, #126
```

### See Also

[AND.EQ](#), [XOR.EQ](#)





Instruction Set

V	Not set by these instructions.
SV	Not set by these instructions.
AV	Not set by these instructions.
SAV	Not set by these instructions.

**Examples**

```
or.ge    d3, d1, d2
or.ge    d3, d1, #126
or.ge.u  d3, d1, d2
or.ge.u  d3, d1, #126
```

**See Also**

[AND.GE](#), [AND.GE.U](#), [XOR.GE](#), [XOR.GE.U](#)



Instruction Set

V	Not set by these instructions.
SV	Not set by these instructions.
AV	Not set by these instructions.
SAV	Not set by these instructions.

**Examples**

```
or.lt    d3, d1, d2
or.lt    d3, d1, #126
or.lt.u  d3, d1, d2
or.lt.u  d3, d1, #126
```

**See Also**

[AND.LT](#), [AND.LT.U](#), [XOR.LT](#), [XOR.LT.U](#)

## OR.NE Not Equal Accumulating

### Description

Calculate the logical OR of D[c][0] and the Boolean result of the NE operation on the contents of data register D[a] and either data register D[b] (instruction format RR) or const9 (instruction format RC). Put the result in D[c][0]. All other bits in D[c] are unchanged.

### OR.NE D[c], D[a], const9 (RC)

31	28 27	21 20	12 11	8 7	0
c	28 <sub>H</sub>	const9	a	8B <sub>H</sub>	

D[c] = {D[c][31:1], D[c][0] OR (D[a] != sign\_ext(const9))};

### OR.NE D[c], D[a], D[b] (RR)

31	28 27	20 19 18 17 16 15	12 11	8 7	0
c	28 <sub>H</sub>	- -	b	a	0B <sub>H</sub>

D[c] = {D[c][31:1], D[c][0] OR (D[a] != D[b])};

### Status Flags

C	Not set by this instruction.
V	Not set by this instruction.
SV	Not set by this instruction.
AV	Not set by this instruction.
SAV	Not set by this instruction.

### Examples

```
or.ne d3, d1, d2
or.ne d3, d1, #126
```

### See Also

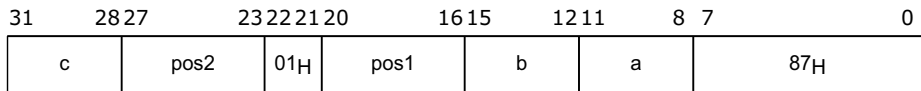
[AND.NE](#), [XOR.NE](#)

## OR.T Bit Logical OR

### Description

Compute the logical OR of bit pos1 of data register D[a] and bit pos2 of data register D[b]. Put the result in the least-significant bit of data register D[c]. Clear the remaining bits of D[c].

### OR.T                      D[c], D[a], pos1, D[b], pos2 (BIT)



result = D[a][pos1] OR D[b][pos2];

D[c] = zero\_ext(result);

### Status Flags

C	Not set by this instruction.
V	Not set by this instruction.
SV	Not set by this instruction.
AV	Not set by this instruction.
SAV	Not set by this instruction.

### Examples

```
or.t    d3, d1, 7, d2, #9
```

### See Also

[AND.T](#), [ANDN.T](#), [NAND.T](#), [NOR.T](#), [ORN.T](#), [XNOR.T](#), [XOR.T](#)

See Also Addendum for TriCore Arch Manual, Vol2, V1.3.8, pages 493/494

## ORN Bitwise OR-Not

### Description

Compute the bitwise OR of the contents of data register D[a] and the ones' complement of the contents of either data register D[b] (instruction format RR) or const9 (instruction format RC). Put the result in data register D[c]. The const9 value is zero-extended to 32-bits.

### ORN D[c], D[a], const9 (RC)

31	28 27	21 20	12 11	8 7	0
c	0FH	const9	a	8FH	

$D[c] = D[a] \mid \sim\text{zero\_ext}(\text{const9});$

### ORN D[c], D[a], D[b] (RR)

31	28 27	20 19	18 17	16 15	12 11	8 7	0
c	0FH	-	-	b	a	0FH	

$D[c] = D[a] \mid \sim D[b];$

### Status Flags

C	Not set by this instruction.
V	Not set by this instruction.
SV	Not set by this instruction.
AV	Not set by this instruction.
SAV	Not set by this instruction.

### Examples

```
orn d3, d1, d2
orn d3, d1, #126
```

### See Also

**AND, ANDN, NAND, NOR, NOT (16-bit), OR, XNOR, XOR**

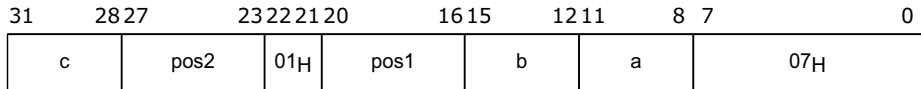
## ORN.T

### Bit Logical OR-Not

#### Description

Compute the logical OR of bit pos1 of data register D[a] and the inverse of bit pos2 of data register D[b]. Put the result in the least-significant bit of data register D[c] and clear the remaining bits of D[c] to zero.

#### ORN.T                      D[c], D[a], pos1, D[b], pos2 (BIT)



result = D[a][pos1] OR !D[b][pos2];

D[c] = zero\_ext(result);

#### Status Flags

C	Not set by this instruction.
V	Not set by this instruction.
SV	Not set by this instruction.
AV	Not set by this instruction.
SAV	Not set by this instruction.

#### Examples

```
orn.t    d3, d1, 2, d2, #5
```

#### See Also

[AND.T](#), [ANDN.T](#), [NAND.T](#), [NOR.T](#), [OR.T](#), [XNOR.T](#), [XOR.T](#)

See Also Addendum for TriCore Arch Manual, Vol2, V1.3.8, pages 493/494



## PACK

### Pack

#### Description

Take the data register pair E[d] and bit 31 of data register D[a] and pack them into an IEEE-754 single precision floating point format number, in data register D[c]. The odd register E[d][63:32], holds the unbiased exponent. The even register E[d][31:0], holds the normalised mantissa in a fractional 1.31 format. Bit 31 of data register D[a] holds the sign bit.

To compute the floating point format number, the input number is first checked for special cases: Infinity, NAN, Overflow, Underflow and Zero. If the input number is not one of these special cases, it is either a normal or denormal number. In both cases, rounding of the input number is performed. First an intermediate biased exponent is calculated, by adding 128 to the unpacked exponent for normal numbers and set to zero for denormal numbers, and inserted into bits [30:23] of the intermediate result. Bits [30:8] of E[d] are inserted into bits [22:0] of the intermediate result. A round flag is calculated from bits [8:0] of E[d] using the IEEE-754 Round-to-Nearest rounding definition, with the PSW.C field acting as an additional sticky bit. If the round flag is set, the intermediate result is incremented by one. Bits [30:0] of the intermediate result are then inserted into bits [30:0] of D[c]. In all cases, bit 31 from D[a] is copied into bit 31 of D[c]. The special cases are handled

as described below.

#### PACK                      D[c], E[d], D[a] (RRR)

31	28 27	24 23	20 19 18 17 16 15	12 11	8 7	0	
c	d	00H	-	0H	-	a	6BH

```
int_exp = E[d][63:32];
```

```
int_mant = E[d][31:0];
```

```
flag_rnd = int_mant[7] AND (int_mant[8] OR int_mant[6:0] OR PSW.C);
```

```
if ((int_mant[31] == 0) AND (int_exp == +255)) then {
```

```
    // Infinity or NaN
```

```
    fp_exp = +255;
```

```
    fp_frac = int_mant[30:8];
```

```
} else if ((int_mant[31] == 1) AND (int_exp >= +127)) then {
```

```
    // Overflow → Infinity.
```

```
    fp_exp = +255;
```

```
    fp_frac = 0;
```

```

} else if ((int_mant[31] == 1) AND (int_exp <= -128)) then {
    // Underflow → Zero
    fp_exp = 0;
    fp_frac = 0;
} else if (int_mant == 0) then {
    // Zero
    fp_exp = 0;
    fp_frac = 0;
} else {
    if (int_mant[31] == 0) then {
        // Denormal
        temp_exp = 0;
    } else {
        // Normal
        temp_exp = int_exp + 128;
    }
    fp_exp_frac[30:0] = {tmp_exp[7:0], int_mant[30:8]} + flag_rnd;
    fp_exp = fp_exp_frac[30:23];
    fp_frac = fp_exp_frac[22:0];
}

```

D[c][31] = D[a][31];

D[c][30:23] = fp\_exp;

D[c][22:0] = fp\_frac;

### Status Flags

C	PSW.C is read by the instruction but not set by this instruction.
V	Not set by this instruction.
SV	Not set by this instruction.
AV	Not set by this instruction.
SAV	Not set by this instruction.

### Examples

pack d8, e2, d10

See Also

[UNPACK](#)

See Also Addendum for TriCore Arch Manual, Vol2, V1.3.8, pages 493/494

## PARITY

### Parity

#### Description

Compute the four byte parity bits of data register D[a]. Put each byte parity bit into every 8th bit of the data register D[c] and then clear the remaining bits of D[c]. A byte parity bit is set to one if the number of ones in a byte is an odd number.

#### PARITY D[c], D[a] (RR)

31	28 27	20 19	18 17	16 15	12 11	8 7	0
c	02H	-	0H	-	a	4BH	

$$D[c][31:24] = \{7'b0, D[a][31] \wedge D[a][30] \wedge D[a][29] \wedge D[a][28] \wedge D[a][27] \wedge D[a][26] \wedge D[a][25] \wedge D[a][24]\};$$

$$D[c][23:16] = \{7'b0, D[a][23] \wedge D[a][22] \wedge D[a][21] \wedge D[a][20] \wedge D[a][19] \wedge D[a][18] \wedge D[a][17] \wedge D[a][16]\};$$

$$D[c][15:8] = \{7'b0, D[a][15] \wedge D[a][14] \wedge D[a][13] \wedge D[a][12] \wedge D[a][11] \wedge D[a][10] \wedge D[a][9] \wedge D[a][8]\};$$

$$D[c][7:0] = \{7'b0, D[a][7] \wedge D[a][6] \wedge D[a][5] \wedge D[a][4] \wedge D[a][3] \wedge D[a][2] \wedge D[a][1] \wedge D[a][0]\};$$

#### Status Flags

C	Not set by this instruction.
V	Not set by this instruction.
SV	Not set by this instruction.
AV	Not set by this instruction.
SAV	Not set by this instruction.

#### Examples

```
parity    d3, d5
```

#### See Also

-

## RET Return from Call

### Description

Return from a function that was invoked with a CALL instruction. The return address is in register A[11] (return address). 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 A[11] (return address). The caller's upper context register values are restored as part of the return operation.

### RET (SR)

15      12 11      8 7                      0

09H	-	00H	
-----	---	-----	--

```

if (PSW.CDE) then if (cdc_decrement()) then trap(CDU);
if (PCXI[19:0] == 0) then trap(CSU);
if (PCXI.UL == 0) then trap(CTYP);
PC = {A11 [31:1], 1'b0};
EA = {PCXI.PCXS, 6'b0, PCXI.PCXO, 6'b0};
{new_PCXI, new_PSW, A[10], A[11], D[8], D[9], D[10], D[11], A[12], A[13], A[14],
A[15], D[12], D[13], D[14], D[15]} = M(EA, 16 * word);
{new_PCXI, PSW, A[10], A[11], D[8], D[9], D[10], D[11], A[12], A[13], A[14], A[15],
D[12], D[13], D[14], D[15]} = M(EA, 16 * word);
M(EA, word) = FCX;
FCX[19:0] = PCXI[19:0];
PCXI = new_PCXI;
PSW = {new_PSW[31:26], PSW[25:24], new_PSW[23:0]};
    
```

1.3.1

1.3.0

1.3.1

### RET (SYS)

31      28 27                      22 21                      12 11      8 7                      0

-	06H	-	-	0DH
---	-----	---	---	-----

```

if (PSW.CDE) then if (cdc_decrement()) then trap(CDU);
if (PCXI[19:0] == 0) then trap(CSU);
if (PCXI.UL == 0) then trap(CTYP);
PC = {A11 [31:1], 1'b0};
EA = {PCXI.PCXS, 6'b0, PCXI.PCXO, 6'b0};
    
```

1.3.1

See Also Addendum for TriCore Arch Manual, Vol2, V1.3.8, pages 493/494

**Instruction Set**

{new\_PCXI, new\_PSW, A[10], A[11], D[8], D[9], D[10], D[11], A[12], A[13], A[14], A[15], D[12], D[13], D[14], D[15]} = M(EA, 16 \* word);

{new\_PCXI, PSW, A[10], A[11], D[8], D[9], D[10], D[11], A[12], A[13], A[14], A[15], D[12], D[13], D[14], D[15]} = M(EA, 16 \* word);

M(EA, word) = FCX;

FCX[19:0] = PCXI[19:0];

PCXI = new\_PCXI;

PSW = {new\_PSW[31:26], PSW[25:24], new\_PSW[23:0]};

**Status Flags**

C	PSW.C is overwritten with the value restored from the Context Save Area (CSA).
V	PSW.V is overwritten with the value restored from the CSA.
SV	PSW.SV is overwritten with the value restored from the CSA.
AV	PSW.AV is overwritten with the value restored from the CSA.
SAV	PSW.SAV is overwritten with the value restored from the CSA.

**Examples**

ret

```
ret
```

**See Also**

[CALL](#), [CALLA](#), [CALLI](#), [RFE](#), [SYSCALL](#), [BISR](#)

See Also Addendum for TriCore Arch Manual, Vol2, V1.3.8, pages 493/494

3.0

1.3.0

3.1

1.3.1

## RFE Return From Exception

### Description

Return from an interrupt service routine or trap handler to the task whose saved upper context is specified by the contents of the Previous Context Information register (PCXI). The contents are normally the context of the task that was interrupted or that took a trap. However in some cases Task Management software may have altered the contents of the PCXI register to cause another task to be dispatched.

The return PC value is taken from register A[11] (register address) 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. The return PC value is taken from register A[11] (register address) 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.

### RFE (SR)

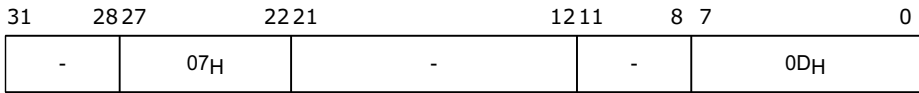
15	12 11	8 7	0
08H	-	00H	

```

if (PCXI[19:0] == 0) then trap(CSU);
if (PCXI.UL == 0) then trap(CTYP);
if (!cdc_zero() AND PSW.CDE) then trap(NEST);
PC = {A11 [31:1], 1'b0};
ICR.IE = PCXI.PIE;
ICR.CCPN = PCXI.PCPN;
EA = {PCXI.PCXS, 6'b0, PCXI.PCXO, 6'b0};
{new_PCXI, PSW, A[10], A[11], D[8], D[9], D[10], D[11], A[12], A[13], A[14], A[15],
D[12], D[13], D[14], D[15]} = M(EA, 16 * word);
M(EA, word) = FCX;
FCX[19:0] = PCXI[19:0];
PCXI = new_PCXI;
    
```

See Also Addendum for TriCore Arch Manual, Vol2, V1.3.8, pages 493/494

**RFE (SYS)**



```

if (PCXI[19:0] == 0) then trap(CSU);
if (PCXI.UL == 0) then trap(CTYP);
if (!lcdc_zero() AND PSW.CDE) then trap(NEST);
PC = {A11 [31:1], 1'b0};
ICR.IE = PCXI.PIE;
ICR.CCPN = PCXI.PCPN;
EA = {PCXI.PCXS, 6'b0, PCXI.PCXO, 6'b0};
{new_PCXI, PSW, A[10], A[11], D[8], D[9], D[10], D[11], A[12], A[13], A[14], A[15], D[12],
D[13], D[14], D[15]} = M(EA, 16 * word);
M(EA, word) = FCX;
FCX[19:0] = PCXI[19:0];
PCXI = new_PCXI;
  
```

**Status Flags**

C	PSW.C is overwritten with the value restored from the Context Save Area (CSA).
V	PSW.V is overwritten with the value restored from the CSA.
SV	PSW.SV is overwritten with the value restored from the CSA.
AV	PSW.AV is overwritten with the value restored from the CSA.
SAV	PSW.SAV is overwritten with the value restored from the CSA.

**Examples**

rfe

```
rfe
```

**See Also**

[CALL](#), [CALLA](#), [CALLI](#), [RET](#), [SYSCALL](#), [BISR](#), [RFM](#)

See Also Addendum for TriCore Arch Manual, Vol2, V1.3.8, pages 493/494



## RFM

### Return From Monitor

#### Description

*Note: The RFM instruction can only be executed in Supervisor mode.*

If the Debug mode is disabled (DBGSR.DE==0) execute as a NOP; otherwise return from a breakpoint monitor to the task whose saved debug context area is located at DCX (Debug Context Pointer).

The Debug Context Area is a four word subset of the context of the task that took a Debug trap, which is saved on entry to the monitor routine. The return PC value is taken from register A[11]. In parallel with the jump to the return PC address, the PCXI and PSW, together with the saved A[10] and A[11] values in the Debug Context Area, are restored to the original task.

1.3.1 The Debug Trap active bit (DBGTCR.DTA) is cleared. 1.3.1

#### RFM

#### (SYS)

31	28 27	22 21	12 11	8 7	0
-	05H	-	-	0DH	

The Debug Context Pointer (DCX) value is implementation dependent.

if (PSW.IO != 2'b10) then trap (PRIV);

if (DBGSR.DE) then{

    PC = {A11 [31:1], 1'b0};

    ICR.IE = PCXI.IE;

    ICR.CCPN = PCXI.PCPN;

    EA = DCX;

    {PCXI, PSW, A[10], A[11]} = M(EA, 4 \* word);

1.3.1 DBGTCCR.DTA = 0; 1.3.1

}else{

    NOP;

}

#### Status Flags

C	PSW.C is overwritten with the value restored from the Debug Context Area.
V	PSW.V is overwritten with the value restored from the Debug Context Area.

Instruction Set

SV	PSW.SV is overwritten with the value restored from the Debug Context Area.
AV	PSW.AV is overwritten with the value restored from the Debug Context Area.
SAV	PSW.SAV is overwritten with the value restored from the Debug Context Area.

**Examples**

r fm

**See Also**

**DEBUG, RFE**

See Also Addendum for TriCore Arch Manual, Vol2, V1.3.8, pages 493/494

## RSLCX

### Restore Lower Context

#### Description

Load the contents of the memory block pointed to by the PCX field in PCXI into registers A[2] to A[7], D[0] to D[7], A[11] (return address), and PCXI. This operation restores the register contents of a previously saved lower context.

#### RSLCX (SYS)

31	28 27	22 21	12 11	8 7	0
-	09H	-	-	-	0DH

if(PCXI[19:0] == 0) then trap(CSU);

if(PCXI.UL == 1) then trap(CTYP);

EA = {PCXI.PCXS, 6'b0, PCXI.PCXO, 6'b0};

{new\_PCXI, A[11], A[2], A[3], D[0], D[1], D[2], D[3], A[4], A[5], A[6], A[7], D[4], D[5], D[6], D[7]} = M(EA, 16\*word);

M(EA, word) = FCX;

FCX[19:0] = PCXI[19:0];

PCXI = new\_PCXI;

#### Status Flags

C	Not set by this instruction.
V	Not set by this instruction.
SV	Not set by this instruction.
AV	Not set by this instruction.
SAV	Not set by this instruction.

#### Examples

```
rslcx
```

#### See Also

[LDLCX](#), [LDUCX](#), [STLCX](#), [STUCX](#), [SVLCX](#), [BISR](#)

## RSTV

### Reset Overflow Bits

#### Description

Reset overflow status flags in the Program Status Word (PSW).

#### RSTV (SYS)

31	28 27	22 21	12 11	8 7	0
-	00H	-	-	2FH	

PSW.{V, SV, AV, SAV} = {0, 0, 0, 0};

#### Status Flags

C	Not set by this instruction.
V	The PSW.V status bit is cleared.
SV	The PSW.SV status bit is cleared.
AV	The PSW.AV status bit is cleared.
SAV	The PSW.SAV status bit is cleared.

#### Examples

```
rstv
```

#### See Also

[BISR](#), [DISABLE](#), [ENABLE](#), [MTCR](#), [TRAPV](#), [TRAPSV](#)

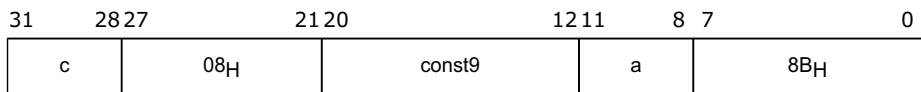
## RSUB Reverse-Subtract

### Description

Subtract the contents of data register D[a] from the value const9 and put the result in data register D[c]. The operands are treated as 32-bit integers. The value const9 is sign-extended before the subtraction is performed.

Subtract the contents of data register D[a] from zero and put the result in data register D[a]. The operand is treated as a 32-bit integer.

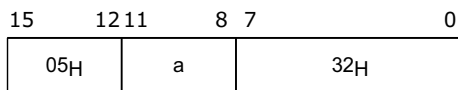
### RSUB                      D[c], D[a], const9 (RC)



result = sign\_ext(const9) - D[a];

D[c] = result[31:0];

### RSUB                      D[a] (SR)



result = 0 - D[a];  
D[a] = result[31:0];

### Status Flags

C	Not set by this instruction.
V	overflow = (result > 7FFFFFFF <sub>H</sub> ) OR (result < -80000000 <sub>H</sub> ); if (overflow) then PSW.V = 1 else PSW.V = 0;
SV	if (overflow) then PSW.SV = 1 else PSW.SV = PSW.SV;
AV	advanced_overflow = result[31] ^ result[30]; if (advanced_overflow) then PSW.AV = 1 else PSW.AV = 0;
SAV	if (advanced_overflow) PSW.SAV = 1 else PSW.SAV = PSW.SAV;

### Examples

```
rsub d3, d1, #126
```

```
rsub d1
```

See Also

[RSUBS](#), [RSUBS.U](#)

See Also Addendum for TriCore Arch Manual, Vol2, V1.3.8, pages 493/494

## RSUBS

Reverse-Subtract with Saturation

## RSUBS.U

Reverse-Subtract Unsigned with Saturation

### Description

Subtract the contents of data register D[a] from the value const9 and put the result in data register D[c]. The operands are treated as signed (RSUBS) or unsigned (RSUBS.U) 32-bit integers, with saturation on signed (RSUBS) or unsigned (RSUBS.U) overflow. The value const9 is sign-extended before the operation is performed.

### RSUBS D[c], D[a], const9 (RC)

31	28 27	21 20	12 11	8 7	0
c	0AH	const9	a	8BH	

result = sign\_ext(const9) - D[a];

D[c] = ssov(result, 32);

### RSUBS.U D[c], D[a], const9 (RC)

31	28 27	21 20	12 11	8 7	0
c	0BH	const9	a	8BH	

result = sign\_ext(const9) - D[a]; // unsigned

D[c] = suov(result, 32);

### Status Flags

C	Not set by these instructions.
V	signed: overflow = (result > 7FFFFFFFH) OR (result < -80000000H); if (overflow) then PSW.V = 1 else PSW.V = 0; unsigned: overflow = (result > FFFFFFFFH) OR (result < 00000000H); if (overflow) then PSW.V = 1 else PSW.V = 0;
SV	if (overflow) then PSW.SV = 1 else PSW.SV = PSW.SV;
AV	advanced_overflow = result[31] ^ result[30]; if (advanced_overflow) then PSW.AV = 1 else PSW.AV = 0;
SAV	if (advanced_overflow) then PSW.SAV = 1 else PSW.SAV = PSW.SAV;

**Examples**

```
rsubs    d3, d1, #126  
rsubs.u  d3, d1, #126
```

**See Also**

**RSUB**

See Also Addendum for TriCore Arch Manual, Vol2, V1.3.8, pages 493/494





See Also

[SAT.BU](#), [SAT.H](#), [SAT.HU](#)

See Also Addendum for TriCore Arch Manual, Vol2, V1.3.8, pages 493/494

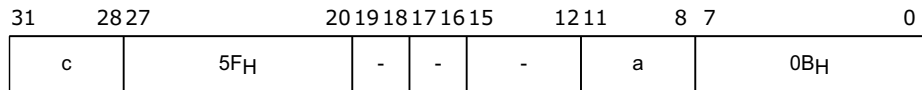
## SAT.BU Saturate Byte Unsigned

### Description

If the unsigned 32-bit value in D[a] is greater than 255, then store the value 255 in D[c]. Otherwise copy D[a] to D[c].

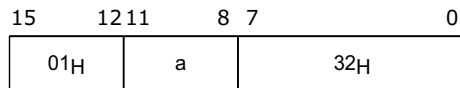
If the unsigned 32-bit value in D[a] is greater than 255, then store the value 255 in D[a]. Otherwise leave the contents of D[a] unchanged.

### SAT.BU                      D[c], D[a] (RR)



D[c] = (D[a] > FF<sub>H</sub>) ? FF<sub>H</sub> : D[a]; // unsigned comparison

### SAT.BU                      D[a] (SR)



D[a] = (D[a] > FF<sub>H</sub>) ? FF<sub>H</sub> : D[a]; // unsigned comparison

### Status Flags

C	Not set by this instruction.
V	Not set by this instruction.
SV	Not set by this instruction.
AV	Not set by this instruction.
SAV	Not set by this instruction.

### Examples

sat.bu    d3, d1

sat.bu    d1

### See Also

[SAT.B](#), [SAT.H](#), [SAT.HU](#)



See Also

[SAT.B](#), [SAT.BU](#), [SAT.HU](#)

See Also Addendum for TriCore Arch Manual, Vol2, V1.3.8, pages 493/494

## SAT.HU

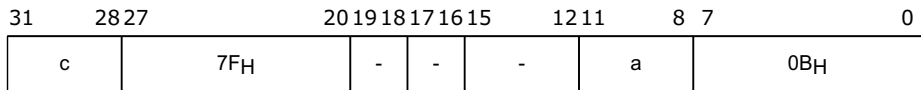
### Saturate Half-word Unsigned

#### Description

If the unsigned 32-bit value in D[a] is greater than 65,535, then store the value 65,535 in D[c]; otherwise copy D[a] to D[c].

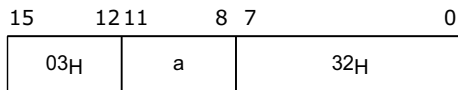
If the unsigned 32-bit value in D[a] is greater than 65,535, then store the value 65,535 in D[c]; otherwise leave the contents of D[a] unchanged.

#### SAT.HU                      D[c], D[a] (RR)



D[c] = (D[a] > FFFF<sub>H</sub>) ? FFFF<sub>H</sub> : D[a]; // unsigned comparison

#### SAT.HU                      D[a] (SR)



D[a] = (D[a] > FFFF<sub>H</sub>) ? FFFF<sub>H</sub> : D[a]; // unsigned comparison

#### Status Flags

C	Not set by this instruction.
V	Not set by this instruction.
SV	Not set by this instruction.
AV	Not set by this instruction.
SAV	Not set by this instruction.

#### Examples

sat.hu    d3, d1

sat.hu    d1

#### See Also

[SAT.B](#), [SAT.BU](#), [SAT.H](#)

## SEL Select

### Description

If the contents of data register D[d] are non-zero, copy the contents of data register D[a] to data register D[c]; otherwise copy the contents of either D[b] (instruction format RRR) or const9 (instruction format RCR), to D[c].

The value const9 (instruction format RCR) is sign-extended.

### SEL D[c], D[d], D[a], const9 (RCR)

31	28 27	24 23	21 20	12 11	8 7	0
c	d	04 <sub>H</sub>	const9	a	AB <sub>H</sub>	

$D[c] = ((D[d] \neq 0) ? D[a] : \text{sign\_ext}(\text{const9}))$ ;

### SEL D[c], D[d], D[a], D[b] (RRR)

31	28 27	24 23	20 19 18 17 16 15	12 11	8 7	0
c	d	04 <sub>H</sub>	- -	b	a	2B <sub>H</sub>

$D[c] = ((D[d] \neq 0) ? D[a] : D[b])$ ;

### Status Flags

C	Not set by this instruction.
V	Not set by this instruction.
SV	Not set by this instruction.
AV	Not set by this instruction.
SAV	Not set by this instruction.

### Examples

```
sel d3, d4, d1, d2
sel d3, d4, d1, #126
```

### See Also

[CADD](#), [CADDN](#), [CMOV \(16-bit\)](#), [CMOVN \(16-bit\)](#), [CSUB](#), [CSUBN](#), [SELN](#)

## SELN Select-Not

### Description

If the contents of data register D[d] are zero, copy the contents of data register D[a] to data register D[c]; otherwise copy the contents of either D[b] or const9 to D[c].

The value const9 (instruction format RCR) is sign-extended.

### SELN D[c], D[d], D[a], const9 (RCR)

31	28 27	24 23	21 20	12 11	8 7	0
c	d	05 <sub>H</sub>	const9	a	AB <sub>H</sub>	

$D[c] = ((D[d] == 0) ? D[a] : \text{sign\_ext}(\text{const9}));$

### SELN D[c], D[d], D[a], D[b] (RRR)

31	28 27	24 23	20 19	18 17	16 15	12 11	8 7	0
c	d	05 <sub>H</sub>	-	-	b	a	2B <sub>H</sub>	

$D[c] = ((D[d] == 0) ? D[a] : D[b]);$

### Status Flags

C	Not set by this instruction.
V	Not set by this instruction.
SV	Not set by this instruction.
AV	Not set by this instruction.
SAV	Not set by this instruction.

### Examples

```
seln d3, d4, d1, d2
seln d3, d4, d1, #126
```

### See Also

**CADD, CADDN, CMOV (16-bit), CMOVN (16-bit), CSUB, CSUBN, SEL**



## SH Shift

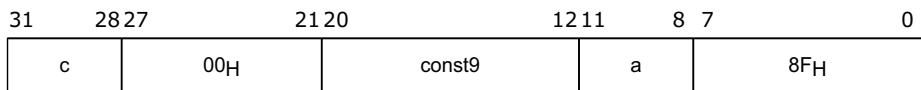
### Description

Shift the value in D[a] by the amount specified by shift count. If the shift count specified through the contents of either D[b] (instruction format RR) or const9 (instruction format RC) is greater than or equal to zero, then left-shift. Otherwise right-shift by the absolute value of the shift count. Put the result in D[c]. In both cases the vacated bits are filled with zeros and the bits shifted out are discarded.

The shift count is a 6-bit signed number, derived from either D[b][5:0] or const9[5:0]. The range for the shift count is therefore -32 to +31, allowing a shift left up to 31 bit positions and to shift right up to 32 bit positions (Note that a shift right by 32 bits leaves zeros in the result).

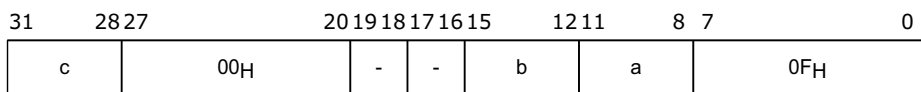
If the shift count specified through the value const4 is greater than or equal to zero, then left-shift the value in D[a] by the amount specified by the shift count. Otherwise right-shift the value in D[a] by the absolute value of the shift count. Put the result in D[a]. In both cases, the vacated bits are filled with zeros and bits shifted out are discarded. The shift count is a 4-bit signed number, derived from the sign-extension of const4[3:0]. The resulting range for the shift count therefore is -8 to +7, allowing a shift left up to 7-bit positions and to shift right up to 8-bit positions.

### SH                      D[c], D[a], const9 (RC)



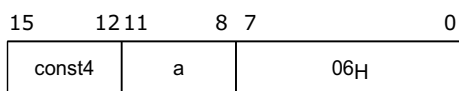
D[c] = (const9[5:0] >= 0) ? D[a] << const9[5:0] : D[a] >> (-const9[5:0]);

### SH                      D[c], D[a], D[b] (RR)



D[c] = (D[b][5:0] >= 0) ? D[a] << D[b][5:0] : D[a] >> (-D[b][5:0]);

### SH                      D[a], const4 (SRC)



shift\_count = sign\_ext(const4[3:0]);

D[a] = (shift\_count >= 0) ? D[a] << shift\_count : D[a] >> (-shift\_count);

### Status Flags

C	Not set by this instruction.
V	Not set by this instruction.
SV	Not set by this instruction.
AV	Not set by this instruction.
SAV	Not set by this instruction.

### Examples

```
sh d3, d1, d2  
sh d3, d1, #26
```

```
sh d1, #6
```

### See Also

[SH.H](#), [SHA](#), [SHA.H](#), [SHAS](#)

## SH.EQ Shift Equal

### Description

Left shift D[c] by one. If the contents of data register D[a] are equal to the contents of either data register D[b] (instruction format RR) or const9 (instruction format RC), set the least-significant bit of D[c] to one; otherwise set the least-significant bit of D[c] to 0.

The value const9 (format RC) is sign-extended.

### SH.EQ                      D[c], D[a], const9 (RC)

31	28 27	21 20	12 11	8 7	0
c	37 <sub>H</sub>	const9	a	8B <sub>H</sub>	

$D[c] = \{D[c][30:0], (D[a] == \text{sign\_ext}(\text{const9}))\};$

### SH.EQ                      D[c], D[a], D[b] (RR)

31	28 27	20 19	18 17	16 15	12 11	8 7	0
c	37 <sub>H</sub>	-	-	b	a	0B <sub>H</sub>	

$D[c] = \{D[c][30:0], (D[a] == D[b])\};$

### Status Flags

C	Not set by this instruction.
V	Not set by this instruction.
SV	Not set by this instruction.
AV	Not set by this instruction.
SAV	Not set by this instruction.

### Examples

```
sh.eq d3, d1, d2
sh.eq d3, d1, #126
```

### See Also

[SH.GE](#), [SH.GE.U](#), [SH.LT](#), [SH.LT.U](#), [SH.NE](#)

## SH.GE

Shift Greater Than or Equal

## SH.GE.U

Shift Greater Than or Equal Unsigned

### Description

Left shift D[c] by one. If the contents of data register D[a] are greater than or equal to the contents of either data register D[b] (instruction format RR) or const9 (instruction format RC), set the least-significant bit of D[c] to one; otherwise set the least-significant bit of D[c] to 0. D[a] and D[b] are treated as signed (SH.GE) or unsigned (SH.GE.U) integers. The value const9 is sign-extended (SH.GE) or zero-extended (SH.GE.U).

### SH.GE D[c], D[a], const9 (RC)

31	28 27	21 20	12 11	8 7	0
c	3BH	const9	a	8BH	

$D[c] = \{D[c][30:0], (D[a] \geq \text{sign\_ext}(\text{const9}))\};$

### SH.GE D[c], D[a], D[b] (RR)

31	28 27	20 19 18 17 16 15	12 11	8 7	0
c	3BH	- -	b	a	0BH

$D[c] = \{D[c][30:0], (D[a] \geq D[b])\};$

### SH.GE.U D[c], D[a], const9 (RC)

31	28 27	21 20	12 11	8 7	0
c	3CH	const9	a	8BH	

$D[c] = \{D[c][30:0], (D[a] \geq \text{zero\_ext}(\text{const9}))\}; // \text{unsigned}$

### SH.GE.U D[c], D[a], D[b] (RR)

31	28 27	20 19 18 17 16 15	12 11	8 7	0
c	3CH	- -	b	a	0BH

$D[c] = \{D[c][30:0], (D[a] \geq D[b])\}; // \text{unsigned}$

### Status Flags

C	Not set by these instructions.
---	--------------------------------

Instruction Set

V	Not set by these instructions.
SV	Not set by these instructions.
AV	Not set by these instructions.
SAV	Not set by these instructions.

**Examples**

```
sh.ge d3, d1, d2
sh.ge d3, d1, #126
sh.ge.u d3, d1, d2
sh.ge.u d3, d1, #126
```

**See Also**

[SH.EQ](#), [SH.LT](#), [SH.LT.U](#), [SH.NE](#)

See Also Addendum for TriCore Arch Manual, Vol2, V1.3.8, pages 493/494

## SH.H Shift Packed Half-words

### Description

If the shift count specified through the contents of either D[b] (instruction format RR) or const9 (instruction format RC) is greater than or equal to zero, then left-shift each half-word in D[a] by the amount specified by shift count. Otherwise, right-shift each half-word in D[a] by the absolute value of the shift count. Put the result in D[c]. In both cases the vacated bits are filled with zeros and bits shifted out are discarded. For these shifts, each half-word is treated individually, and bits shifted out of a half-word are not shifted in to the next half-word.

The shift count is a signed number, derived from the sign-extension of either D[b][4:0] (instruction format RR) or const9[4:0] (instruction format RC). The range for the shift count is therefore -16 to +15. The result for a shift count of -16 for half-words is zero.

### SH.H D[c], D[a], const9 (RC)

31	28 27	21 20	12 11	8 7	0
c	40 <sub>H</sub>	const9	a	8F <sub>H</sub>	

shift\_count = sign\_ext(const9[4:0]);

result\_halfword1 = (shift\_count >= 0) ? D[a][31:16] << shift\_count : D[a][31:16] >> (0 - shift\_count);

result\_halfword0 = (shift\_count >= 0) ? D[a][15:0] << shift\_count : D[a][15:0] >> (0 - shift\_count);

D[c] = {result\_halfword1[15:0], result\_halfword0[15:0]};

### SH.H D[c], D[a], D[b] (RR)

31	28 27	20 19	18 17	16 15	12 11	8 7	0
c	40 <sub>H</sub>	-	-	b	a	0F <sub>H</sub>	

shift\_count = sign\_ext(D[b][4:0]);

result\_halfword1 = (shift\_count >= 0) ? D[a][31:16] << shift\_count : D[a][31:16] >> (0 - shift\_count);

result\_halfword0 = (shift\_count >= 0) ? D[a][15:0] << shift\_count : D[a][15:0] >> (0 - shift\_count);

D[c] = {result\_halfword1[15:0], result\_halfword0[15:0]};

### Status Flags

C	Not set by this instruction.
V	Not set by this instruction.
SV	Not set by this instruction.
AV	Not set by this instruction.
SAV	Not set by this instruction.

### Examples

```
sh.h    d3, d1, d2  
sh.h    d3, d1, #12
```

### See Also

[SH](#), [SHA](#), [SHA.H](#), [SHAS](#)

See Also Addendum for TriCore Arch Manual, Vol2, V1.3.8, pages 493/494

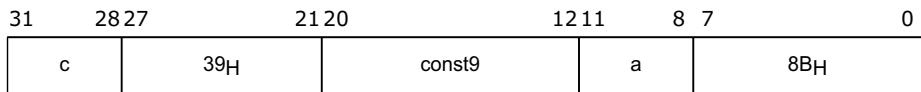
**SH.LT**  
**Shift Less Than**  
**SH.LT.U**  
**Shift Less Than Unsigned**

**Description**

Left shift D[c] by one. If the contents of data register D[a] are less than the contents of either data register D[b] (instruction format RR) or const9 (instruction format RC), set the least-significant bit of D[c] to one; otherwise set the least-significant bit of D[c] to zero.

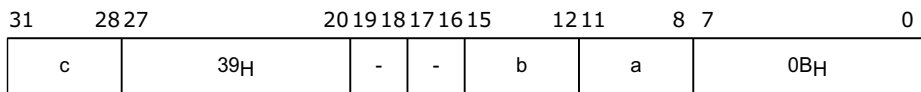
D[a] and either D[b] (format RR) or const9 (format RC) are treated as signed (SH.LT) or unsigned (SH.LT.U) integers. The value const9 is sign-extended (SH.LT) or zero-extended (SH.LT.U).

**SH.LT**                      **D[c], D[a], const9 (RC)**



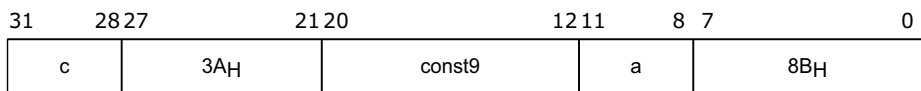
$$D[c] = \{D[c][30:0], (D[a] < \text{sign\_ext}(\text{const9}))\};$$

**SH.LT**                      **D[c], D[a], D[b] (RR)**



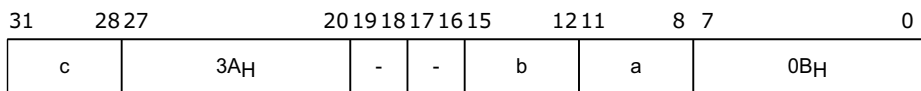
$$D[c] = \{D[c][30:0], (D[a] < D[b])\};$$

**SH.LT.U**                      **D[c], D[a], const9 (RC)**



$$D[c] = \{D[c][30:0], (D[a] < \text{zero\_ext}(\text{const9}))\}; // \text{unsigned}$$

**SH.LT.U**                      **D[c], D[a], D[b] (RR)**



$$D[c] = \{D[c][30:0], (D[a] < D[b])\}; // \text{unsigned}$$

See Also Addendum for TriCore Arch Manual, Vol2, V1.3.8, pages 493/494



### Status Flags

C	Not set by these instructions.
V	Not set by these instructions.
SV	Not set by these instructions.
AV	Not set by these instructions.
SAV	Not set by these instructions.

### Examples

```
sh.lt    d3, d1, d2
sh.lt    d3, d1, #126
sh.lt.u  d3, d1, d2
sh.lt.u  d3, d1, #126
```

### See Also

[SH.EQ](#), [SH.GE](#), [SH.GE.U](#), [SH.NE](#)



**SH.AND.T**

Accumulating Shift-AND

**SH.ANDN.T**

Accumulating Shift-AND-Not

**SH.NAND.T**

Accumulating Shift-NAND

**SH.NOR.T**

Accumulating Shift-NOR

**SH.OR.T**

Accumulating Shift-OR

**SH.ORN.T**

Accumulating Shift-OR-Not

**SH.XNOR.T**

Accumulating Shift-XNOR

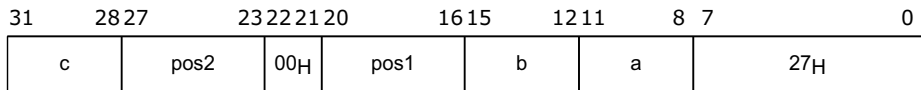
**SH.XOR.T**

Accumulating Shift-XOR

**Description**

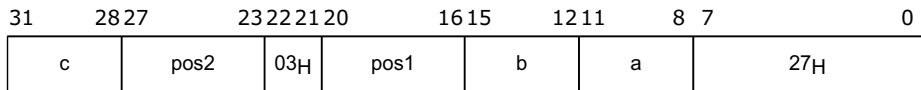
Left shift D[c] by one. The bit shifted out is discarded. Compute the logical operation (AND, ANDN, NAND, NOR, OR, ORN, XNOR or XOR) of the value of bit pos1 of data register D[a], and bit pos2 of D[b]. Put the result in D[c][0].

**SH.AND.T                    D[c], D[a], pos1, D[b], pos2 (BIT)**



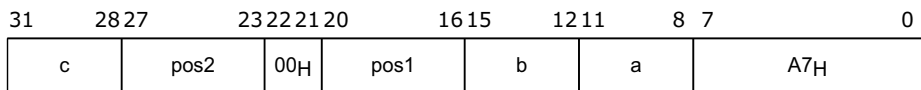
$$D[c] = \{D[c][30:0], (D[a][pos1] \text{ AND } D[b][pos2])\};$$

**SH.ANDN.T                    D[c], D[a], pos1, D[b], pos2 (BIT)**



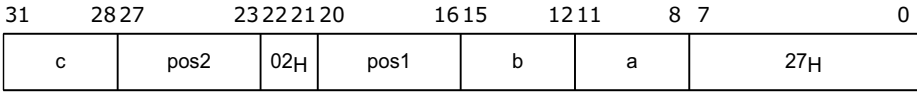
$$D[c] = \{D[c][30:0], (D[a][pos1] \text{ AND } \neg(D[b][pos2]))\};$$

**SH.NAND.T                    D[c], D[a], pos1, D[b], pos2 (BIT)**



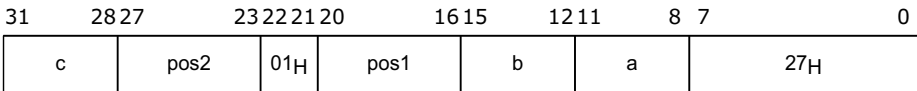
$$D[c] = \{D[c][30:0], \neg(D[a][pos1] \text{ AND } D[b][pos2])\};$$

**SH.NOR.T**                      **D[c], D[a], pos1, D[b], pos2 (BIT)**



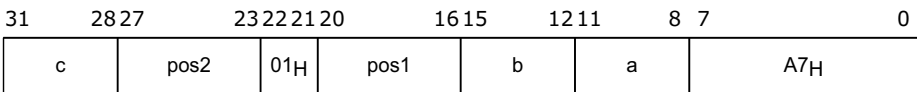
$D[c] = \{D[c][30:0], \!(D[a][pos1] \text{ OR } D[b][pos2])\};$

**SH.OR.T**                      **D[c], D[a], pos1, D[b], pos2 (BIT)**



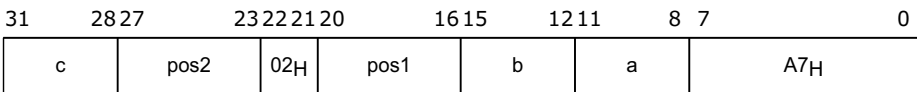
$D[c] = \{D[c][30:0], (D[a][pos1] \text{ OR } D[b][pos2])\};$

**SH.ORN.T**                      **D[c], D[a], pos1, D[b], pos2 (BIT)**



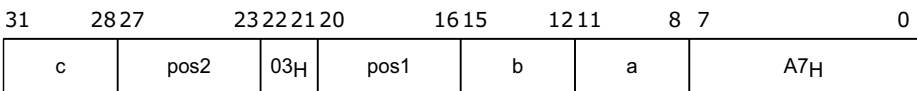
$D[c] = \{D[c][30:0], (D[a][pos1] \text{ OR } \!(D[b][pos2]))\};$

**SH.XNOR.T**                      **D[c], D[a], pos1, D[b], pos2 (BIT)**



$D[c] = \{D[c][30:0], \!(D[a][pos1] \text{ XOR } D[b][pos2])\};$

**SH.XOR.T**                      **D[c], D[a], pos1, D[b], pos2 (BIT)**



$D[c] = \{D[c][30:0], (D[a][pos1] \text{ XOR } D[b][pos2])\};$

**Status Flags**

C	Not set by these instructions.
V	Not set by these instructions.
SV	Not set by these instructions.
AV	Not set by these instructions.
SAV	Not set by these instructions.

### Examples

sh.and.t	d3, d1, 4, d2, 7
sh.andn.t	d3, d1, 4, d2, 7
sh.nand.t	d3, d1, 4, d2, 7
sh.nor.t	d3, d1, 4, d2, 7
sh.or.t	d3, d1, 4, d2, 7
sh.orn.t	d3, d1, 4, d2, 7
sh.xnor.t	d3, d1, 4, d2, 7
sh.xor.t	d3, d1, 4, d2, 7

### See Also

[AND.AND.T](#), [AND.ANDN.T](#), [AND.NOR.T](#), [AND.OR.T](#), [OR.AND.T](#), [OR.ANDN.T](#),  
[OR.NOR.T](#), [OR.OR.T](#)

## SHA Arithmetic Shift

### Description

If shift count specified through contents of either D[b] (instruction format RR) or const9 (instruction format RC) is greater than or equal to zero, then left-shift the value in D[a] 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 the value in D[a] by the absolute value of the shift count. The vacated bits are filled with the sign-bit (the most significant bit) and bits shifted out are discarded. Put the result in D[c].

The shift count is a 6-bit signed number, derived from either D[b][5:0] or const9[5:0]. The range for shift count is therefore -32 to +31, allowing a shift left up to 31 bit positions and a shift right up to 32 bit positions (a shift right by 32 bits leaves all zeros or all ones in the result, depending on the sign bit). On all 1-bit or greater shifts (left or right), PSW.C is set to the logical-OR of the shifted out bits. On zero-bit shifts C is cleared.

If shift count specified through the value const4 is greater than or equal to zero, then left-shift the value in D[a] by the amount specified by the 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 the value in D[a] by the absolute value of the shift count. The vacated bits are filled with the sign-bit (the most significant bit) and bits shifted out are discarded. Put the result in D[a].

The shift count is a 6-bit signed number, derived from the sign-extension of const4[3:0]. The resulting range for the shift count is therefore -8 to +7, allowing a shift left up to 7 bit positions, and a shift right up to 8 bit positions. On all shifts of 1-bit or greater (left or right), PSW.C is set to the logical-OR of the shifted out bits. On zero-bit shifts C is cleared.

### SHA                      D[c], D[a], const9 (RC)

31	28 27	21 20	12 11	8 7	0
c	01H	const9	a	8FH	

```

if (const9[5:0] >= 0) then {
    carry_out = const9[5:0] ? (D[a][31:32 - const9[5:0]] != 0) : 0;
    result = D[a] << const9[5:0];
} else {
    shift_count = 0 - const9[5:0];
    msk = D[a][31] ? (((1 << shift_count) - 1) << (32 - shift_count)) : 0;
    result = msk | (D[a] >> shift_count);
    carry_out = (D[a][shift_count - 1:0] != 0);

```

```
}
D[c] = result[31:0];
```

**SHA D[c], D[a], D[b] (RR)**

31	28 27	20 19	18 17	16 15	12 11	8 7	0
c	01H	-	-	b	a	0FH	

```
if (D[b][5:0] >= 0) then {
    carry_out = D[b][5:0] ? (D[a][31:32 - D[b][5:0]] != 0) : 0;
    result = D[a] << D[b][5:0];
} else {
    shift_count = 0 - D[b][5:0];
    msk = D[a][31] ? (((1 << shift_count) - 1) << (32 - shift_count)) : 0;
    result = msk | (D[a] >> shift_count);
    carry_out = (D[a][shift_count - 1:0] != 0);
}
D[c] = result[31:0];
```

**SHA D[a], const4 (SRC)**

15	12 11	8 7	0
const4	a	86H	

```
if (const4[4:0] >= 0) then {
    carry_out = const4[4:0] ? (D[a][31:32 - const4[4:0]] != 0) : 0;
    result = D[a] << const4[4:0];
} else {
    shift_count = 0 - const4[4:0];
    msk = D[a][31] ? (((1 << shift_count) - 1) << (32 - shift_count)) : 0;
    result = msk | (D[a] >> shift_count);
    carry_out = (D[a][shift_count - 1:0] != 0);
}
D[a] = result[31:0];
```

**Status Flags**

C	if (carry_out) then PSW.C = 1 else PSW.C = 0;
V	overflow = (result > 7FFFFFFFH) OR (result < -80000000H); if (overflow) then PSW.V = 1 else PSW.V = 0;

See Also Addendum for TriCore Arch Manual, Vol2, V1.3.8, pages 493/494

SV	if (overflow) then PSW.SV = 1 else PSW.SV = PSW.SV;
AV	advanced_overflow = D[c][31] ^ D[c][30]; if (advanced_overflow) then PSW.AV = 1 else PSW.AV = 0;
SAV	if (advanced_overflow) then PSW.SAV = 1 else PSW.SAV = PSW.SAV;

### Examples

```
sha d3, d1, d2
sha d3, d1, #26
```

```
sha d1, #6
```

### See Also

[SH](#), [SH.H](#), [SHAS](#), [SHA.H](#)



## SHA.H Arithmetic Shift Packed Half-words

### Description

If the shift count specified through the contents of either D[b] (instruction format RR) or const9 (instruction format RC) is greater than or equal to zero, then left-shift each half-word in D[a] 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 half-word in D[a] by the absolute value of the shift count. The vacated bits are filled with the sign-bit (the most significant bit) of the respective half-word, and bits shifted out are discarded. Put the result in D[c]. Note that for the shifts, each half-word is treated individually, and bits shifted out of a half-word are not shifted into the next half-word.

The shift count is a signed number, derived from the sign-extension of either D[b][4:0] (format RR) or const9[4:0] (format RC).

The range for the shift count is -16 to +15. The result for each half-word for a shift count of -16 is either all zeros or all ones, depending on the sign-bit of the respective half-word.

### SHA.H                      D[c], D[a], const9 (RC)

31	28 27	21 20	12 11	8 7	0
c	41H	const9	a	8FH	

if (const9[4:0] >= 0) then {

result\_halfword0 = D[a][15:0] << const9[4:0];

result\_halfword1 = D[a][31:16] << const9[4:0];

} else {

shift\_count = 0 - const9[4:0];

msk = D[a][31] ? (((1 << shift\_count) - 1) << (16 - shift\_count)) : 0;

result = msk | (D[a] >> shift\_count);

result\_halfword0 = msk | (D[a][15:0] >> shift\_count);

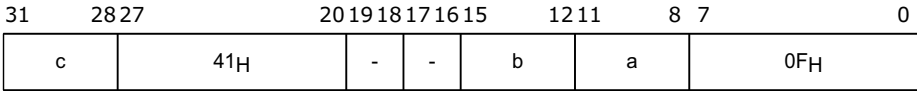
result\_halfword1 = msk | (D[a][31:16] >> shift\_count);

}

D[c][15:0] = result\_halfword0[15:0];

D[c][31:16] = result\_halfword1[15:0];

**SHA.H**                      **D[c], D[a], D[b] (RR)**



```

if (D[b][4:0] >= 0) then {
    result_halfword0 = D[a][15:0] << D[b][4:0];
    result_halfword1 = D[a][31:16] << D[b][4:0];
} else {
    shift_count = 0 - D[b][4:0];
    msk = D[a][31] ? (((1 << shift_count) - 1) << (16 - shift_count)) : 0;
    result_halfword0 = msk | (D[a][15:0] >> shift_count);
    result_halfword1 = msk | (D[a][31:16] >> shift_count);
}
D[c][15:0] = result_halfword0[15:0];
D[c][31:16] = result_halfword1[15:0];

```

**Status Flags**

C	Not set by this instruction.
V	Not set by this instruction.
SV	Not set by this instruction.
AV	Not set by this instruction.
SAV	Not set by this instruction.

**Examples**

```

sha.h   d3, d1, d2
sha.h   d3, d1, #12

```

**See Also**

**SH, SHA, SHAS, SH.H**

See Also Addendum for TriCore Arch Manual, Vol2, V1.3.8, pages 493/494

## SHAS Arithmetic Shift with Saturation

### Description

If the shift count specified through the contents of either D[b] (instruction format RR) or const9 (instruction format RC) is greater than or equal to zero, then left-shift the value in D[a] by the amount specified by shift count. The vacated bits are filled with zeros 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 D[a] by the absolute value of the shift count. The vacated bits are filled with the sign-bit (the most significant bit) and bits shifted out are discarded. Put the result in D[c]. The shift count is a 6-bit signed number, derived from D[b][5:0] (format RR) or const9[5:0] (format RC).

The range for the shift count is -32 to +31, allowing shift left up to 31 bit positions and to shift right up to 32 bit positions. Note that a shift right by 32 bits leaves all zeros or all ones in the result, depending on the sign-bit.

### SHAS                      D[c], D[a], const9 (RC)

31	28 27	21 20	12 11	8 7	0
c	02H	const9	a	8FH	

```
if (const9[5:0] >= 0) then {
    result = D[a] << const9[5:0];
} else {
    shift_count = 0 - const9[5:0];
    msk = D[a][31] ? (((1 << shift_count) - 1) << (32 - shift_count)) : 0;
    result = msk | (D[a] >> shift_count);
}
D[c] = ssov(result,32);
```

### SHAS                      D[c], D[a], D[b] (RR)

31	28 27	20 19 18 17 16 15	12 11	8 7	0	
c	02H	-	-	b	a	0FH

```
if (D[b][5:0] >= 0) then {
    result = D[a] << D[b][5:0];
} else {
    shift_count = 0 - D[b][5:0];
    msk = D[a][31] ? (((1 << shift_count) - 1) << (32 - shift_count)) : 0;
```

```

result = msk | (D[a] >> shift_count);
}
D[c] = ssov(result,32);

```

### Status Flags

C	Not set by this instruction.
V	overflow = (result > 7FFFFFFF <sub>H</sub> ) OR (result < -80000000 <sub>H</sub> ); if (overflow) then PSW.V = 1 else PSW.V = 0;
SV	if (overflow) then PSW.SV = 1 else PSW.SV = PSW.SV;
AV	advanced_overflow = result[31] ^ result[30]; if (advanced_overflow) then PSW.AV = 1 else PSW.AV = 0;
SAV	if (advanced_overflow) then PSW.SAV = 1 else PSW.SAV = PSW.SAV;

### Examples

```

shas d3, d1, d2
shas d3, d1, #26

```

### See Also

[SH](#), [SH.H](#), [SHA](#), [SHA.H](#)

## ST.A Store Word from Address Register

### Description

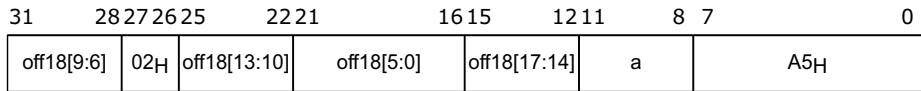
Store the value in address register A[a] to the memory location specified by the addressing mode.

*Note: If the source register is modified by the addressing mode, the value stored to memory is undefined.*

Store the value in address register A[a] (instruction format BO, SSR, SSRO or SSR) or A[15] (instruction format SRO or SC) to the memory location specified by the addressing mode.

*Note: If the source register is modified by the addressing mode, the value stored to memory is undefined.*

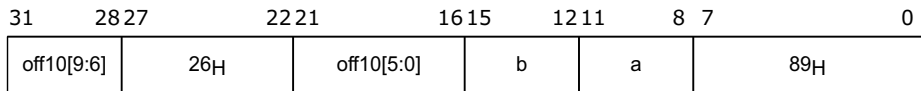
### ST.A                      off18, A[a] (ABS)                      (Absolute Addressing Mode)



EA = {off18[17:14], 14b'0, off18[13:0]};

M(EA, word) = A[a];

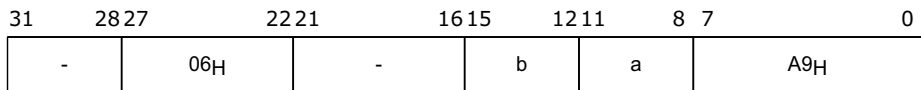
### ST.A                      A[b], off10, A[a] (BO) (Base + Short Offset Addressing Mode)



EA = A[b] + sign\_ext(off10);

M(EA, word) = A[a];

### ST.A                      P[b], A[a] (BO)                      (Bit-reverse Addressing Mode)



index = zero\_ext(A[b+1][15:0]);

incr = zero\_ext(A[b+1][31:16]);

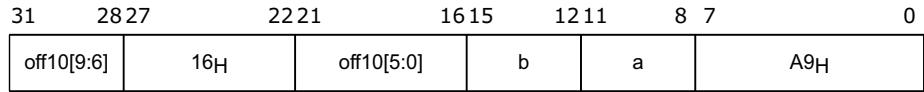
EA = A[b] + index;

M(EA, word) = A[a];

new\_index = reverse16(reverse16(index) + reverse16(incr));

$A[b+1] = \{\text{incr}[15:0], \text{new\_index}[15:0]\};$

**ST.A                      P[b], off10, A[a] (BO)                      (Circular Addressing Mode)**



$\text{index} = \text{zero\_ext}(A[b+1][15:0]);$

$\text{length} = \text{zero\_ext}(A[b+1][31:16]);$

$\text{EA} = A[b] + \text{index};$

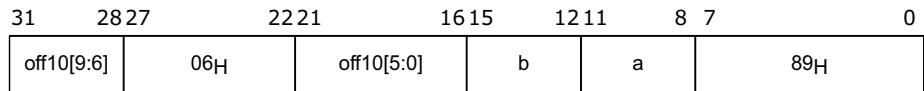
$M(\text{EA}, \text{word}) = A[a];$

$\text{new\_index} = \text{index} + \text{sign\_ext}(\text{off10});$

$\text{new\_index} = \text{new\_index} < 0 ? \text{new\_index} + \text{length} : \text{new\_index} \% \text{length};$

$A[b+1] = \{\text{length}[15:0], \text{new\_index}[15:0]\};$

**ST.A                      A[b], off10, A[a] (BO)                      (Post-increment Addressing Mode)**

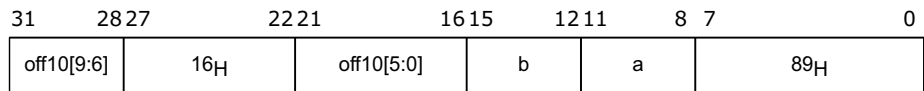


$\text{EA} = A[b];$

$M(\text{EA}, \text{word}) = A[a];$

$A[b] = \text{EA} + \text{sign\_ext}(\text{off10});$

**ST.A                      A[b], off10, A[a] (BO)                      (Pre-increment Addressing Mode)**

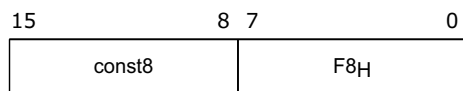


$\text{EA} = A[b] + \text{sign\_ext}(\text{off10});$

$M(\text{EA}, \text{word}) = A[a];$

$A[b] = \text{EA};$

**ST.A                      A[10], const8, A[15] (SC)**



$M(A[10] + \text{zero\_ext}(4 * \text{const8}), \text{word}) = A[15];$

See Also Addendum for TriCore Arch Manual, Vol2, V1.3.8, pages 493/494

**ST.A                      A[b], off4, A[15] (SRO)**

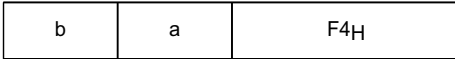
15      12 11              8 7                      0



$M(A[b] + \text{zero\_ext}(4 * \text{off4}), \text{word}) = A[15];$

**ST.A                      A[b], A[a] (SSR)**

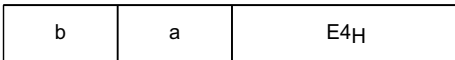
15      12 11              8 7                      0



$M(A[b], \text{word}) = A[a];$

**ST.A                      A[b], A[a] (SSR)                      (Post-increment Addressing Mode)**

15      12 11              8 7                      0

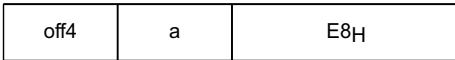


$M(A[b], \text{word}) = A[a];$

$A[b] = A[b] + 4;$

**ST.A                      A[15], off4, A[a] (SSRO)**

15      12 11              8 7                      0



$M(A[15] + \text{zero\_ext}(4 * \text{off4}), \text{word}) = A[a];$

**Status Flags**

C	Not set by this instruction.
V	Not set by this instruction.
SV	Not set by this instruction.
AV	Not set by this instruction.
SAV	Not set by this instruction.

**Examples**

-

```
st.a    [a0], a0
st.a    [a15]+4, a2
```

See Also Addendum for TriCore Arch Manual, Vol2, V1.3.8, pages 493/494

See Also

[ST.B](#), [ST.D](#), [ST.DA](#), [ST.H](#), [ST.Q](#), [ST.W](#)

See Also Addendum for TriCore Arch Manual, Vol2, V1.3.8, pages 493/494



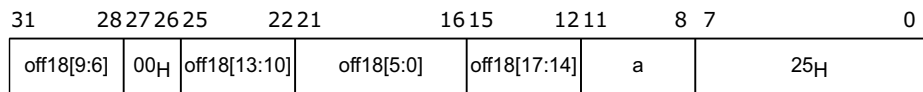
## ST.B Store Byte

### Description

Store the byte value in the eight least-significant bits of data register D[a] to the byte memory location specified by the addressing mode.

Store the byte value in the eight least-significant bits of either data register D[a] (instruction format SSR, SSR0 or BO) or D[15] (instruction format SRO) to the byte memory location specified by the addressing mode.

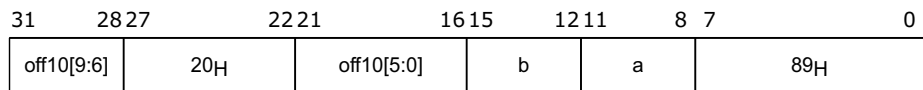
### ST.B                      off18, D[a] (ABS)                      (Absolute Addressing Mode)



EA = {off18[17:14], 14b'0, off18[13:0]};

M(EA, byte) = D[a][7:0];

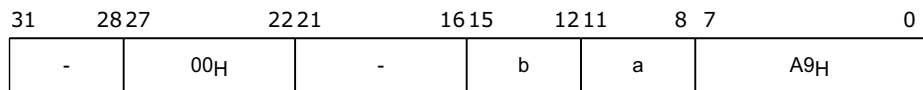
### ST.B                      A[b], off10, D[a] (BO) (Base + Short Offset Addressing Mode)



EA = A[b] + sign\_ext(off10);

M(EA, byte) = D[a][7:0];

### ST.B                      P[b], D[a] (BO)                      (Bit-reverse Addressing Mode)



index = zero\_ext(A[b+1][15:0]);

incr = zero\_ext(A[b+1][31:16]);

EA = A[b] + index;

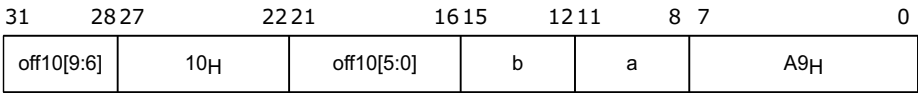
M(EA, byte) = D[a][7:0];

new\_index = reverse16(reverse16(index) + reverse16(incr));

A[b+1] = {incr[15:0], new\_index[15:0]};

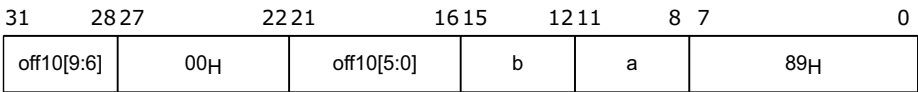
Instruction Set

**ST.B                      P[b], off10, D[a] (BO)                      (Circular Addressing Mode)**



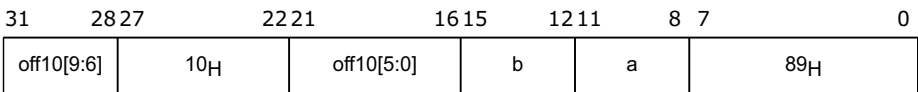
index = zero\_ext(A[b+1][15:0]);  
length = zero\_ext(A[b+1][31:16]);  
EA0 = A[b] + index;  
M(EA, byte) = D[a][7:0];  
new\_index = index + sign\_ext(off10);  
new\_index = new\_index < 0 ? new\_index + length : new\_index % length;  
A[b+1] = {length[15:0], new\_index[15:0]};

**ST.B                      A[b], off10, D[a] (BO)                      (Post-increment Addressing Mode)**



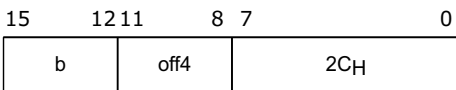
EA = A[b];  
M(EA, byte) = D[a][7:0];  
A[b] = EA + sign\_ext(off10);

**ST.B                      A[b], off10, D[a] (BO)                      (Pre-increment Addressing Mode)**



EA = A[b] + sign\_ext(off10);  
M(EA, byte) = D[a][7:0];  
A[b] = EA;

**ST.B                      A[b], off4, D[15] (SRO)**

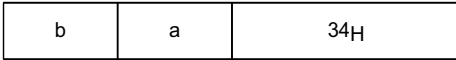


M(A[b] + zero\_ext(off4), byte) = D[15][7:0];

See Also Addendum for TriCore Arch Manual, Vol2, V1.3.8, pages 493/494

**ST.B                      A[b], D[a] (SSR)**

15      12 11              8 7                      0

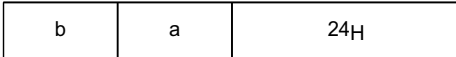


$M(A[b], \text{byte}) = D[a][7:0];$

**ST.B                      A[b], D[a] (SSR)**

**(Post-increment Addressing Mode)**

15      12 11              8 7                      0

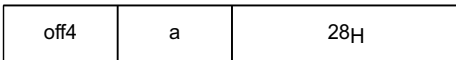


$M(A[b], \text{byte}) = D[a][7:0];$

$A[b] = A[b] + 1;$

**ST.B                      A[15], off4, D[a] (SSRO)**

15      12 11              8 7                      0



$M(A[15] + \text{zero\_ext}(\text{off4}), \text{byte}) = D[a][7:0];$

**Status Flags**

C	Not set by this instruction.
V	Not set by this instruction.
SV	Not set by this instruction.
AV	Not set by this instruction.
SAV	Not set by this instruction.

**Examples**

st.b [a0+]2, d0

st.b [a3]+24, d2

st.b [a0], d0

st.b [a15]+14, d2

**See Also**

[ST.A](#), [ST.D](#), [ST.DA](#), [ST.H](#), [ST.Q](#), [ST.W](#)

## ST.D Store Double-word

### Description

Store the value in the extended data register pair E[a] to the memory location specified by the addressing mode. The value in the even register D[n] is stored in the least-significant memory word, and the value in the odd register (D[n+1]) is stored in the most-significant memory word.

### ST.D                      off18, E[a] (ABS)                      (Absolute Addressing Mode)

31	28 27 26 25	22 21	16 15	12 11	8 7	0
off18[9:6]	01 <sub>H</sub>	off18[13:10]	off18[5:0]	off18[17:14]	a	A5 <sub>H</sub>

EA = {off18[17:14], 14b'0, off18[13:0]};

M(EA, doubleword) = E[a];

### ST.D                      A[b], off10, E[a] (BO) (Base + Short Offset Addressing Mode)

31	28 27	22 21	16 15	12 11	8 7	0
off10[9:6]	25 <sub>H</sub>	off10[5:0]	b	a		89 <sub>H</sub>

EA = A[b] + sign\_ext(off10);

M(EA, doubleword) = E[a];

### ST.D                      P[b], E[a] (BO)                      (Bit-reverse Addressing Mode)

31	28 27	22 21	16 15	12 11	8 7	0
-	05 <sub>H</sub>	-	b	a		A9 <sub>H</sub>

index = zero\_ext(A[b+1][15:0]);

incr = zero\_ext(A[b+1][31:16]);

EA = A[b] + index;

M(EA, doubleword) = E[a];

new\_index = reverse16(reverse16(index) + reverse16(incr));

A[b+1] = {incr[15:0], new\_index[15:0]};

### ST.D                      P[b], off10, E[a] (BO)                      (Circular Addressing Mode)

31	28 27	22 21	16 15	12 11	8 7	0
off10[9:6]	15 <sub>H</sub>	off10[5:0]	b	a		A9 <sub>H</sub>

```

index = zero_ext(A[b+1][15:0]);
length = zero_ext(A[b+1][31:16]);
EA0 = A[b] + index;
EA2 = A[b] + (index + 2) % length;
EA4 = A[b] + (index + 4) % length;
EA6 = A[b] + (index + 6) % length;
M(EA0, halfword) = D[a][15:0];
M(EA2, halfword) = D[a][31:16];
M(EA4, halfword) = D[a+1][15:0];
M(EA6, halfword) = D[a+1][31:16];
new_index = index + sign_ext(off10);
new_index = new_index < 0 ? new_index + length : new_index % length;
A[b+1] = {length[15:0], new_index[15:0]};
    
```

**ST.D                      A[b], off10, E[a] (BO)                      (Post-increment Addressing Mode)**

31	28 27	22 21	16 15	12 11	8 7	0
off10[9:6]	05H	off10[5:0]	b	a	89H	

```

EA = A[b];
M(EA, doubleword) = E[a];
A[b] = EA + sign_ext(off10);
    
```

**ST.D                      A[b], off10, E[a] (BO)                      (Pre-increment Addressing Mode)**

31	28 27	22 21	16 15	12 11	8 7	0
off10[9:6]	15H	off10[5:0]	b	a	89H	

```

EA = A[b] + sign_ext(off10);
M(EA, doubleword) = E[a];
A[b] = EA;
    
```

**Status Flags**

C	Not set by this instruction.
V	Not set by this instruction.
SV	Not set by this instruction.
AV	Not set by this instruction.

See Also Addendum for TriCore Arch Manual, Vol2, V1.3.8, pages 493/494

---

SAV	Not set by this instruction.
-----	------------------------------

---

**Examples**

```
st.d [a0], e0
st.d [a0], d0/d1
st.d [a15+]8, e12
st.d [a15+]8, d12/d13
```

**See Also**

[ST.A](#), [ST.B](#), [ST.DA](#), [ST.H](#), [ST.Q](#), [ST.W](#)

See Also Addendum for TriCore Arch Manual, Vol2, V1.3.8, pages 493/494

## ST.DA Store Double-word from Address Registers

### Description

Store the value in the address register pair  $A[a]/A[a+1]$  to the memory location specified by the addressing mode. The value in the even register  $A[a]$  is stored in the least-significant memory word, and the value in the odd register ( $A[a+1]$ ) is stored in the most-significant memory word.

### ST.DA                      off18, P[a] (ABS)                      (Absolute Addressing Mode)

31	28 27 26 25	22 21	16 15	12 11	8 7	0
off18[9:6]	03 <sub>H</sub>	off18[13:10]	off18[5:0]	off18[17:14]	a	A5 <sub>H</sub>

$EA = \{off18[17:14], 14b'0, off18[13:0]\};$

$M(EA, doubleword) = P[a];$

### ST.DA                      A[b], off10, P[a] (BO) (Base + Short Offset Addressing Mode)

31	28 27	22 21	16 15	12 11	8 7	0
off10[9:6]	27 <sub>H</sub>	off10[5:0]	b	a		89 <sub>H</sub>

$EA = A[b] + sign\_ext(off10);$

$M(EA, doubleword) = P[a];$

### ST.DA                      P[b], P[a] (BO)                      (Bit-reverse Addressing Mode)

31	28 27	22 21	16 15	12 11	8 7	0
-	07 <sub>H</sub>	-	b	a		A9 <sub>H</sub>

$index = zero\_ext(A[b+1][15:0]);$

$incr = zero\_ext(A[b+1][31:16]);$

$EA = A[b] + index;$

$M(EA, doubleword) = P[a];$

$new\_index = reverse16(reverse16(index) + reverse16(incr));$

$A[b+1] = \{incr[15:0], new\_index[15:0]\};$

### ST.DA                      P[b], off10, P[a] (BO)                      (Circular Addressing Mode)

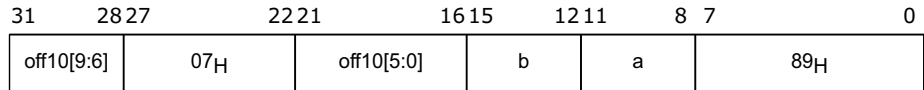
31	28 27	22 21	16 15	12 11	8 7	0
off10[9:6]	17 <sub>H</sub>	off10[5:0]	b	a		A9 <sub>H</sub>

```

index = zero_ext(A[b+1][15:0]);
length = zero_ext(A[b+1][31:16]);
EA0 = A[b] + index;
EA4 = A[b] + (index + 4) % length;
(M(EA0, word) = A[a];
(M(EA4, word) = A[a+1];
new_index = index + sign_ext(off10);
new_index = new_index < 0 ? new_index + length : new_index % length;
A[b+1] = {length[15:0], new_index[15:0]};

```

**ST.DA                      A[b], off10, P[a] (BO)                      (Post-increment Addressing Mode)**

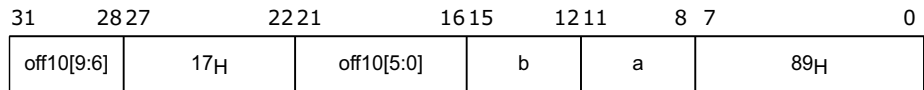


```

EA = A[b];
M(EA, doubleword) = P[a];
A[b] = EA + sign_ext(off10);

```

**ST.DA                      A[b], off10, P[a] (BO)                      (Pre-increment Addressing Mode)**



```

EA = A[b] + sign_ext(off10);
M(EA, doubleword) = P[a];
A[b] = EA;

```

**Status Flags**

C	Not set by this instruction.
V	Not set by this instruction.
SV	Not set by this instruction.
AV	Not set by this instruction.
SAV	Not set by this instruction.

**Examples**

```
st.da    [a6]+8, a4/a5
```



st.da `_savedPointerBuffer`, a0/a1

**See Also**

[ST.A](#), [ST.B](#), [ST.D](#), [ST.H](#), [ST.Q](#), [ST.W](#)

See Also Addendum for TriCore Arch Manual, Vol2, V1.3.8, pages 493/494

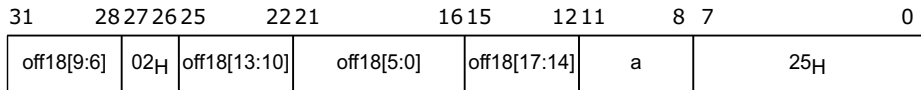
## ST.H Store Half-word

### Description

Store the half-word value in the 16 least-significant bits of data register D[a] to the half-word memory location specified by the addressing mode.

Store the half-word value in the 16 least-significant bits of either data register D[a] (instruction format or D[15]) to the half-word memory location specified by the addressing mode.

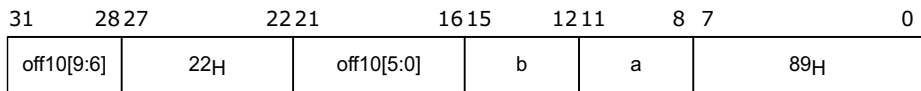
### ST.H                      off18, D[a] (ABS)                      (Absolute Addressing Mode)



EA = {off18[17:14], 14b'0, off18[13:0]};

M(EA, halfword) = D[a][15:0];

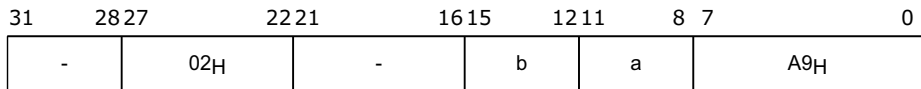
### ST.H                      A[b], off10, D[a] (BO) (Base + Short Offset Addressing Mode)



EA = A[b] + sign\_ext(off10);

M(EA, halfword) = D[a][15:0];

### ST.H                      P[b], D[a] (BO)                      (Bit-reverse Addressing Mode)



index = zero\_ext(A[b+1][15:0]);

incr = zero\_ext(A[b+1][31:16]);

EA = A[b] + index;

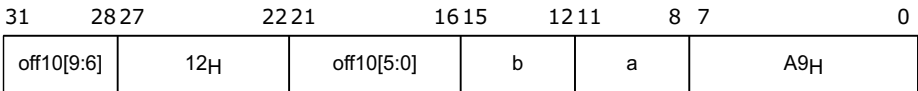
M(EA, halfword) = D[a][15:0];

new\_index = reverse16(reverse16(index) + reverse16(incr));

A[b+1] = {incr[15:0], new\_index[15:0]};

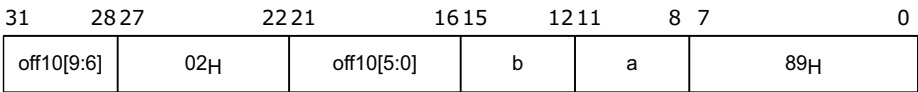
Instruction Set

**ST.H                      P[b], off10, D[a] (BO)                      (Circular Addressing Mode)**



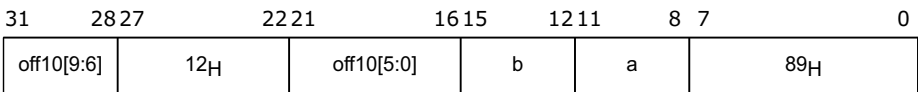
index = zero\_ext(A[b+1][15:0]);  
length = zero\_ext(A[b+1][31:16]);  
EA = A[b] + index;  
M(EA, halfword) = D[a][15:0];  
new\_index = index + sign\_ext(off10);  
new\_index = new\_index < 0 ? new\_index + length : new\_index % length;  
A[b+1] = {length[15:0], new\_index[15:0]};

**ST.H                      A[b], off10, D[a] (BO)                      (Post-increment Addressing Mode)**



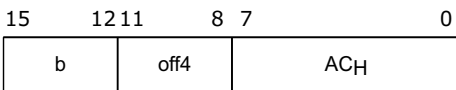
EA = A[b];  
M(EA, halfword) = D[a][15:0];  
A[b] = EA + sign\_ext(off10);

**ST.H                      A[b], off10, D[a] (BO)                      (Pre-increment Addressing Mode)**



EA = A[b] + sign\_ext(off10);  
M(EA, halfword) = D[a][15:0];  
A[b] = EA;

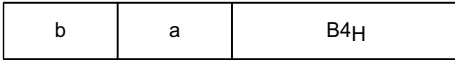
**ST.H                      A[b], off4, D[15] (SRO)**



M(A[b] + zero\_ext(2 \* off4), half-word) = D[15][15:0];

**ST.H                      A[b], D[a] (SSR)**

15      12 11              8 7                      0

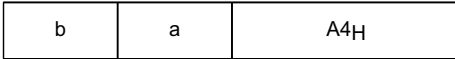


$M(A[b], \text{half-word}) = D[a][15:0];$

**ST.H                      A[b], D[a] (SSR)**

**(Post-increment Addressing Mode)**

15      12 11              8 7                      0

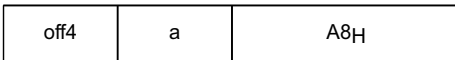


$M(A[b], \text{half-word}) = D[a][15:0];$

$A[b] = A[b] + 2;$

**ST.H                      A[15], off4, D[a] (SSRO)**

15      12 11              8 7                      0



$M(A[15] + \text{zero\_ext}(2 * \text{off4}), \text{half-word}) = D[a][15:0];$

**Status Flags**

C	Not set by this instruction.
V	Not set by this instruction.
SV	Not set by this instruction.
AV	Not set by this instruction.
SAV	Not set by this instruction.

**Examples**

st.h      [a0+]8, d0

st.h      [a0+]24, d2

st.h      [a0], d0

**See Also**

[ST.A](#), [ST.B](#), [ST.D](#), [ST.DA](#), [ST.Q](#), [ST.W](#)

## ST.Q Store Half-word Signed Fraction

### Description

Store the value in the most-significant half-word of data register D[a] to the memory location specified by the addressing mode.

### ST.Q                      off18, D[a] (ABS)                      (Absolute Addressing Mode)

31	28 27 26 25	22 21	16 15	12 11	8 7	0
off18[9:6]	00 <sub>H</sub>	off18[13:10]	off18[5:0]	off18[17:14]	a	65 <sub>H</sub>

EA = {off18[17:14], 14b'0, off18[13:0]};

M(EA, halfword) = D[a][31:16];

### ST.Q                      A[b], off10, D[a] (BO) (Base + Short Offset Addressing Mode)

31	28 27	22 21	16 15	12 11	8 7	0
off10[9:6]	28 <sub>H</sub>	off10[5:0]	b	a	89 <sub>H</sub>	

EA = A[b] + sign\_ext(off10);

M(EA, halfword) = D[a][31:16];

### ST.Q                      P[b], D[a] (BO)                      (Bit-reverse Addressing Mode)

31	28 27	22 21	16 15	12 11	8 7	0
-	08 <sub>H</sub>	-	b	a	A9 <sub>H</sub>	

index = zero\_ext(A[b+1][15:0]);

incr = zero\_ext(A[b+1][31:16]);

EA = A[b] + index;

M(EA, halfword) = D[a][31:16];

new\_index = reverse16(reverse16(index) + reverse16(incr));

A[b+1] = {incr[15:0], new\_index[15:0]};

### ST.Q                      P[b], off10, D[a] (BO)                      (Circular Addressing Mode)

31	28 27	22 21	16 15	12 11	8 7	0
off10[9:6]	18 <sub>H</sub>	off10[5:0]	b	a	A9 <sub>H</sub>	

index = zero\_ext(A[b+1][15:0]);

length = zero\_ext(A[b+1][31:16]);

EA = A[b] + index;

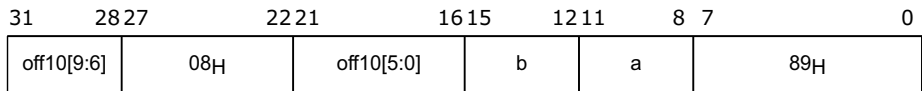
M(EA, halfword) = D[a][31:16];

new\_index = index + sign\_ext(off10);

new\_index = new\_index < 0 ? new\_index + length : new\_index % length;

A[b+1] = {length[15:0], new\_index[15:0]};

**ST.Q                      A[b], off10, D[a] (BO)                      (Post-increment Addressing Mode)**

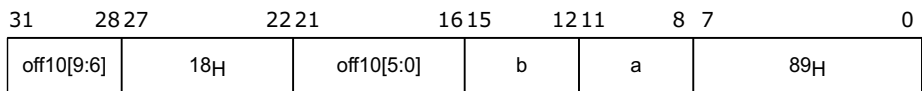


EA = A[b];

M(EA, halfword) = D[a][31:16];

A[b] = EA + sign\_ext(off10);

**ST.Q                      A[b], off10, D[a] (BO)                      (Pre-increment Addressing Mode)**



EA = A[b] + sign\_ext(off10);

M(EA, halfword) = D[a][31:16];

A[b] = EA;

**Status Flags**

C	Not set by this instruction.
V	Not set by this instruction.
SV	Not set by this instruction.
AV	Not set by this instruction.
SAV	Not set by this instruction.

**Examples**

st.q [a0+]2, d0

st.q [a0+]22, d0

**See Also**

[ST.A](#), [ST.B](#), [ST.D](#), [ST.DA](#), [ST.H](#), [ST.W](#)

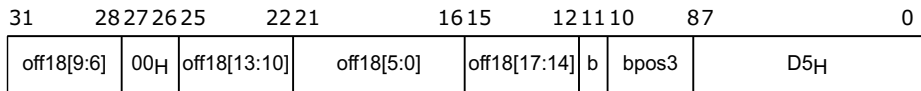
See Also Addendum for TriCore Arch Manual, Vol2, V1.3.8, pages 493/494

## ST.T Store Bit

### Description

Store the bit value *b* to the byte at the memory address specified by *off18*, in the bit position specified by *bpos3*. The other bits of the byte are unchanged. Individual bits can be used as semaphore.

### ST.T                      *off18, bpos3, b (ABSB)*



EA = {off18[17:14], 14'b0, off18[13:0]};

M(EA, byte) = (M(EA, byte) AND ~(1 << *bpos3*)) | (*b* << *bpos3*);

### Status Flags

C	Not set by this instruction.
V	Not set by this instruction.
SV	Not set by this instruction.
AV	Not set by this instruction.
SAV	Not set by this instruction.

### Examples

```
st.t 90000000H, #7H, #1H
```

### See Also

[IMASK](#), [LDMST](#), [SWAP.W](#)

See Also Addendum for TriCore Arch Manual, Vol2, V1.3.8, pages 493/494

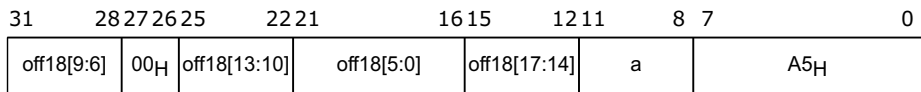
## ST.W Store Word

### Description

Store the word value in data register D[a] to the memory location specified by the addressing mode.

Store the word value in either data register D[a] (instruction format SSR, SSRO) or D[15] (instruction format SRO, SC) to the memory location specified by the addressing mode.

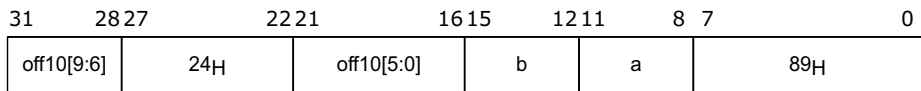
### ST.W                      off18, D[a] (ABS)                      (Absolute Addressing Mode)



EA = {off18[17:14], 14b'0, off18[13:0]};

M(EA, word) = D[a];

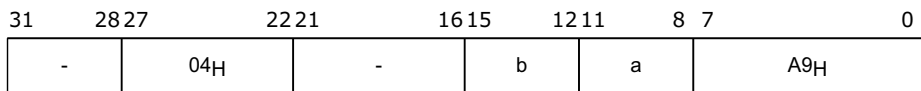
### ST.W                      A[b], off10, D[a] (BO) (Base + Short Offset Addressing Mode)



EA = A[b] + sign\_ext(off10);

M(EA, word) = D[a];

### ST.W                      P[b], D[a] (BO)                      (Bit-reverse Addressing Mode)



index = zero\_ext(A[b+1][15:0]);

incr = zero\_ext(A[b+1][31:16]);

EA = A[b] + index;

M(EA, word) = D[a];

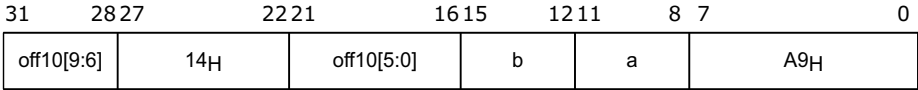
new\_index = reverse16(reverse16(index) + reverse16(incr));

A[b+1] = {incr[15:0], new\_index[15:0]};



Instruction Set

**ST.W                      P[b], off10, D[a] (BO)                      (Circular Addressing Mode)**

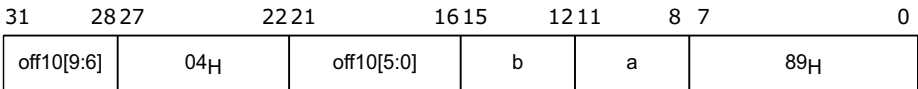


```

index = zero_ext(A[b+1][15:0]);
length = zero_ext(A[b+1][31:16]);
EA0 = A[b] + index;
EA2 = A[b] + (index +2) % length;
M(EA0, halfword) = D[a][15:0];
M(EA2, halfword) = D[a][31:16];
new_index = index + sign_ext(off10);
new_index = new_index < 0 ? new_index + length : new_index % length;
A[b+1] = {length[15:0], new_index[15:0]};

```

**ST.W                      A[b], off10, D[a] (BO)                      (Post-increment Addressing Mode)**

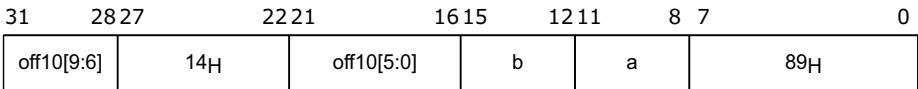


```

EA = A[b];
M(EA, word) = D[a];
A[b] = EA + sign_ext(off10);

```

**ST.W                      A[b], off10, D[a] (BO)                      (Pre-increment Addressing Mode)**

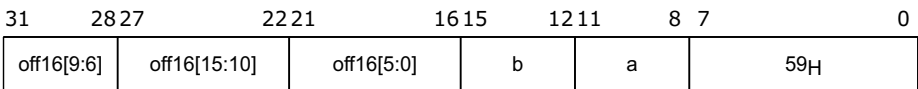


```

EA = A[b] + sign_ext(off10);
M(EA, word) = D[a];
A[b] = EA;

```

**ST.W                      A[b], off16, D[a] (BOL)(Base + Long Offset Addressing Mode)**



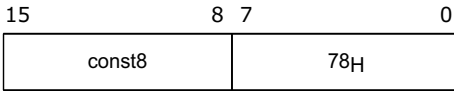
```

EA = A[b] + sign_ext(off16);
M(EA, word) = D[a];

```

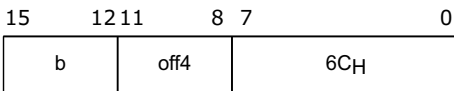
See Also Addendum for TriCore Arch Manual, Vol2, V1.3.8, pages 493/494

**ST.W                      A[10], const8, D[15] (SC)**



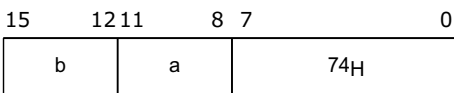
$M(A[10] + \text{zero\_ext}(4 * \text{const8}), \text{word}) = D[15];$

**ST.W                      A[b], off4, D[15] (SRO)**



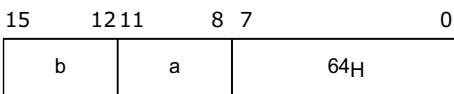
$M(A[b] + \text{zero\_ext}(4 * \text{off4}), \text{word}) = D[15];$

**ST.W                      A[b], D[a] (SSR)**



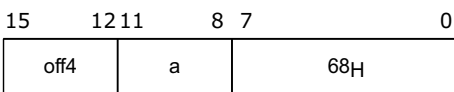
$M(A[b], \text{word}) = D[a];$

**ST.W                      A[b], D[a] (SSR)                      (Post-increment Addressing Mode)**



$M(A[b], \text{word}) = D[a];$   
 $A[b] = A[b] + 4;$

**ST.W                      A[15], off4, D[a] (SSRO)**



$M(A[15] + \text{zero\_ext}(4 * \text{off4}), \text{word}) = D[a];$

**Status Flags**

C	Not set by this instruction.
V	Not set by this instruction.
SV	Not set by this instruction.
AV	Not set by this instruction.

---

SAV	Not set by this instruction.
-----	------------------------------

---

**Examples**

```
st.w    [a0+]2, d0  
st.w    [a0+]22, d0
```

**See Also**

[ST.A](#), [ST.B](#), [ST.D](#), [ST.DA](#), [ST.H](#), [ST.Q](#)

See Also Addendum for TriCore Arch Manual, Vol2, V1.3.8, pages 493/494

## STLCX

### Store Lower Context

#### Description

Store the contents of registers A[2] to A[7], D[0] to D[7], A[11] (return address) and PCXI, 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).

*Note: The effective address (EA) specified by the addressing mode must be aligned on a 16-word boundary.*

#### STLCX                      off18 (ABS)                      (Absolute Addressing Mode)

31	28 27 26 25	22 21	16 15	12 11	8 7	0
off18[9:6]	00 <sub>H</sub>	off18[13:10]	off18[5:0]	off18[17:14]	-	15 <sub>H</sub>

EA = {off18[17:14], 14b'0, off18[13:0]};

M(EA, 16-word) = {PCXI, A[11], A[2:3], D[0:3], A[4:7], D[4:7]};

#### STLCX                      A[b], off10 (BO)                      (Base + Short Index Addressing Mode)

31	28 27	22 21	16 15	12 11	8 7	0
off10[9:6]	26 <sub>H</sub>	off10[5:0]	b	-	49 <sub>H</sub>	

EA = A[b] + sign\_ext(off10)[9:0];

M(EA, 16-word) = {PCXI, A[11], A[2:3], D[0:3], A[4:7], D[4:7]};

#### Status Flags

C	Not set by this instruction.
V	Not set by this instruction.
SV	Not set by this instruction.
AV	Not set by this instruction.
SAV	Not set by this instruction.

#### Examples

-

#### See Also

**LDLCX, LDUCX, RSLCX, STUCX, SVLCX, BISR**

## STUCX

### Store Upper Context

#### Description

Store the contents of registers A[10] to A[15], D[8] to D[15], 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).

*Note: The effective address (EA) specified by the addressing mode must be aligned on a 16-word boundary.*

#### STUCX                      off18 (ABS)                      (Absolute Addressing Mode)

31	28 27 26 25	22 21	16 15	12 11	8 7	0
off18[9:6]	01 <sub>H</sub>	off18[13:10]	off18[5:0]	off18[17:14]	-	15 <sub>H</sub>

EA = {off18[17:14], 14b'0, off18[13:0]};

M(EA, 16-word) = {PCXI, PSW, A[10:11], D[8:11], A[12:15], D[12:15]};

#### STUCX                      A[b], off10 (BO)                      (Base + Short Index Addressing Mode)

31	28 27	22 21	16 15	12 11	8 7	0
off10[9:6]	27 <sub>H</sub>	off10[5:0]	b	-		49 <sub>H</sub>

EA = A[b] + sign\_ext(off10)[9:0];

M(EA, 16-word) = {PCXI, PSW, A[10:11], D[8:11], A[12:15], D[12:15]};

#### Status Flags

C	PSW.C is read by the instruction but not changed.
V	PSW.V is read by the instruction but not changed.
SV	PSW.SV is read by the instruction but not changed.
AV	PSW.AV is read by the instruction but not changed.
SAV	PSW.SAV is read by the instruction but not changed.

#### Examples

-

#### See Also

[LDLCX](#), [LDUCX](#), [RSLCX](#), [STLCX](#), [SVLCX](#), [STUCX](#)

## SUB

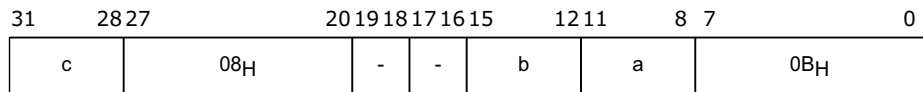
### Subtract

#### Description

Subtract the contents of data register D[b] from the contents of data register D[a] and put the result in data register D[c]. The operands are treated as 32-bit integers.

Subtract the contents of data register D[b] from the contents of either data register D[a] or D[15] and put the result in either data register D[a] or D[15]. The operands are treated as 32-bit integers.

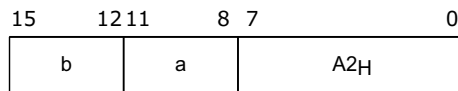
#### SUB D[c], D[a], D[b] (RR)



result = D[a] - D[b];

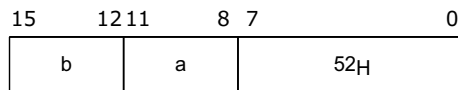
D[c] = result[31:0];

#### SUB D[a], D[b] (SRR)



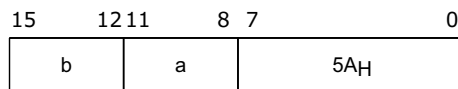
result = D[a] - D[b];  
D[a] = result[31:0];

#### SUB D[a], D[15], D[b] (SRR)



result = D[15] - D[b];  
D[a] = result[31:0];

#### SUB D[15], D[a], D[b] (SRR)



result = D[a] - D[b];  
D[15] = result[31:0];

### Status Flags

C	Not set by this instruction.
V	overflow = (result > 7FFFFFFF <sub>H</sub> ) OR (result < -80000000 <sub>H</sub> ); if (overflow) then PSW.V = 1 else PSW.V = 0;
SV	if (overflow) then PSW.SV = 1 else PSW.SV = PSW.SV;
AV	advanced_overflow = result[31] ^ result[30]; if (advanced_overflow) then PSW.AV = 1 else PSW.AV = 0;
SAV	if (advanced_overflow) then PSW.SAV = 1 else PSW.SAV = PSW.SAV;

### Examples

```
sub    d3, d1, d2
```

```
sub    d1, d2
sub    d15, d1, d2
sub    d1, d15, d2
```

### See Also

[SUBS](#), [SUBS.U](#), [SUBX](#), [SUBC](#)

## SUB.A

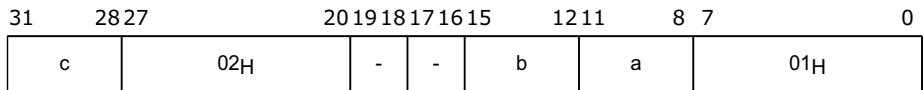
### Subtract Address

#### Description

Subtract the contents of address register A[b] from the contents of address register A[a] and put the result in address register A[c].

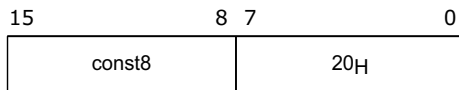
Decrement the Stack Pointer (A[10]) by the zero-extended value of const8 (a range of 0 through to 255).

#### SUB.A                      A[c], A[a], A[b] (RR)



A[c] = A[a] - A[b];

#### SUB.A                      A[10], const8 (SC)



A[10] = A[10] - zero\_ext(const8);

#### Status Flags

C	Not set by this instruction.
V	Not set by this instruction.
SV	Not set by this instruction.
AV	Not set by this instruction.
SAV	Not set by this instruction.

#### Examples

```
sub.a    a3, a4, a2
```

```
sub.a    sp, #126
```

#### See Also

[ADD.A](#), [ADDIH.A](#), [ADDSC.A](#), [ADDSC.AT](#)



## SUB.B

Subtract Packed Byte

## SUB.H

Subtract Packed Half-word

### Description

Subtract the contents of each byte or half-word of data register D[b] from the contents of data register D[a]. Put the result in each corresponding byte or half-word of data register D[c].

### SUB.B D[c], D[a], D[b] (RR)

31	28 27	20 19 18 17 16 15	12 11	8 7	0	
c	48 <sub>H</sub>	-	-	b	a	0B <sub>H</sub>

result\_byte3 = D[a][31:24] - D[b][31:24];

result\_byte2 = D[a][23:16] - D[b][23:16];

result\_byte1 = D[a][15:8] - D[b][15:8];

result\_byte0 = D[a][7:0] - D[b][7:0];

D[c] = {result\_byte3[7:0], result\_byte2[7:0], result\_byte1[7:0], result\_byte0[7:0]};

### SUB.H D[c], D[a], D[b] (RR)

31	28 27	20 19 18 17 16 15	12 11	8 7	0	
c	68 <sub>H</sub>	-	-	b	a	0B <sub>H</sub>

result\_halfword1 = D[a][31:16] - D[b][31:16];

result\_halfword0 = D[a][15:0] - D[b][15:0];

D[c] = {result\_halfword1[15:0], result\_halfword0[15:0]};

### Status Flags

C	Not set by these instructions.
---	--------------------------------

V	<p>SUB.B  <math>ov\_byte3 = (result\_byte3 &gt; 7F_H) \text{ OR } (result\_byte3 &lt; -80_H);</math>  <math>ov\_byte2 = (result\_byte2 &gt; 7F_H) \text{ OR } (result\_byte2 &lt; -80_H);</math>  <math>ov\_byte1 = (result\_byte1 &gt; 7F_H) \text{ OR } (result\_byte1 &lt; -80_H);</math>  <math>ov\_byte0 = (result\_byte0 &gt; 7F_H) \text{ OR } (result\_byte0 &lt; -80_H);</math>  <math>overflow = ov\_byte3 \text{ OR } ov\_byte2 \text{ OR } ov\_byte1 \text{ OR } ov\_byte0;</math>                      if (overflow) then PSW.V = 1 else PSW.V = 0;</p> <p>SUB.H  <math>ov\_halfword1 = (result\_halfword1 &gt; 7FFF_H) \text{ OR } (result\_halfword1 &lt; -8000_H);</math>  <math>ov\_halfword0 = (result\_halfword0 &gt; 7FFF_H) \text{ OR } (result\_halfword0 &lt; -8000_H);</math>  <math>overflow = ov\_halfword1 \text{ OR } ov\_halfword0;</math>                      if (overflow) then PSW.V = 1 else PSW.V = 0;</p>
SV	<p>if (overflow) then PSW.SV = 1 else PSW.SV = PSW.SV;</p>
AV	<p>SUB.B  <math>aov\_byte3 = result\_byte3[7] \wedge result\_byte3[6];</math>  <math>aov\_byte2 = result\_byte2[7] \wedge result\_byte2[6];</math>  <math>aov\_byte1 = result\_byte1[7] \wedge result\_byte1[6];</math>  <math>aov\_byte0 = result\_byte0[7] \wedge result\_byte0[6];</math>  <math>advanced\_overflow = aov\_byte3 \text{ OR } aov\_byte2 \text{ OR } aov\_byte1 \text{ OR } aov\_byte0;</math>                      if (advanced_overflow) then PSW.AV = 1 else PSW.AV = 0;</p> <p>SUB.H  <math>aov\_halfword1 = result\_halfword1[15] \wedge result\_halfword1[14];</math>  <math>aov\_halfword0 = result\_halfword0[15] \wedge result\_halfword0[14];</math>  <math>advanced\_overflow = aov\_halfword1 \text{ OR } aov\_halfword0;</math>                      if (advanced_overflow) then PSW.AV = 1 else PSW.AV = 0;</p>
SAV	<p>if (advanced_overflow) then PSW.SAV = 1 else PSW.SAV = PSW.SAV;</p>

**Examples**

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

**See Also**

**SUBS.H, SUBS.HU**

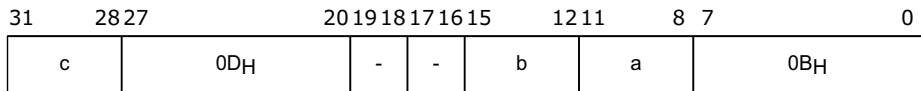
## SUBC

### Subtract With Carry

#### Description

Subtract the contents of data register D[b] from contents of data register D[a] plus the carry bit minus one. Put the result in data register D[c]. The operands are treated as 32-bit integers. The PSW carry bit is set to the value of the ALU carry out.

#### SUBC                      D[c], D[a], D[b] (RR)



result = D[a] - D[b] + PSW.C - 1;  
D[c] = result[31:0];  
carry\_out = carry(D[a], ~D[b], PSW.C);

#### Status Flags

C	PSW.C = carry_out;
V	overflow = (result > 7FFFFFFF <sub>H</sub> ) OR (result < -80000000 <sub>H</sub> ); if (overflow) then PSW.V = 1 else PSW.V = 0;
SV	if (overflow) then PSW.SV = 1 else PSW.SV = PSW.SV;
AV	advanced_overflow = result[31] ^ result[30]; if (advanced_overflow) then PSW.AV = 1 else PSW.AV = 0;
SAV	if (advanced_overflow) then PSW.SAV = 1 else PSW.SAV = PSW.SAV;

#### Examples

```
subc d3, d1, d2
```

#### See Also

[SUB](#), [SUBS](#), [SUBS.U](#), [SUBX](#)

See Also Addendum for TriCore Arch Manual, Vol2, V1.3.8, pages 493/494

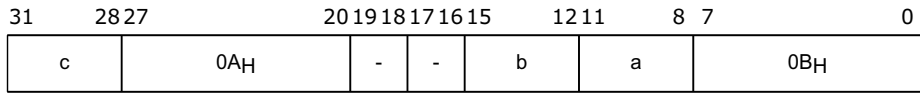
**SUBS**  
Subtract Signed with Saturation  
**SUBS.U**  
Subtract Unsigned with Saturation

**Description**

Subtract the contents of data register D[b] from the contents of data register D[a] and put the result in data register D[c]. The operands are treated as signed (SUBS) or unsigned (SUBS.U) 32-bit integers, with saturation on signed (SUBS) or (SUBS.U) unsigned overflow.

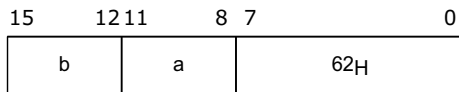
Subtract the contents of data register D[b] from the contents of data register D[a] and put the result in data register D[a]. The operands are treated as signed 32-bit integers, with saturation on signed overflow.

**SUBS**                      **D[c], D[a], D[b] (RR)**



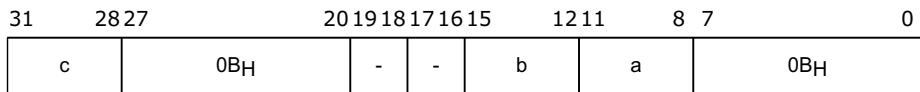
result = D[a] - D[b];  
D[c] = ssov(result, 32);

**SUBS**                      **D[a], D[b] (SRR)**



result = D[a] - D[b];  
D[a] = ssov(result, 32);

**SUBS.U**                      **D[c], D[a], D[b] (RR)**



result = D[a] - D[b];  
D[c] = suov(result, 32);

**Status Flags**

C	Not set by these instructions.
---	--------------------------------

See Also Addendum for TriCore Arch Manual, Vol2, V1.3.8, pages 493/494

V	signed: overflow = (result > 7FFFFFFF <sub>H</sub> ) OR (result < -80000000 <sub>H</sub> ); if (overflow) then PSW.V = 1 else PSW.V = 0; unsigned: overflow = (result > FFFFFFFF <sub>H</sub> ) OR (result < 00000000 <sub>H</sub> ); if (overflow) then PSW.V = 1 else PSW.V = 0;
SV	if (overflow) then PSW.SV = 1 else PSW.SV = PSW.SV;
AV	advanced_overflow = result[31] ^ result[30]; if (advanced_overflow) then PSW.AV = 1 else PSW.AV = 0;
SAV	if (advanced_overflow) then PSW.SAV = 1 else PSW.SAV = PSW.SAV;

### Examples

```
subs    d3, d1, d2
subs.u  d3, d1, d2
```

```
subs    d3, d1
```

### See Also

[SUB](#), [SUBX](#), [SUBC](#)

## SUBS.H

Subtract Packed Half-word with Saturation

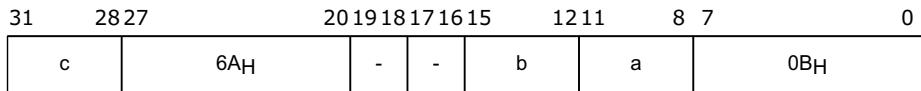
## SUBS.HU

Subtract Packed Half-word Unsigned with Saturation

### Description

Subtract the contents of each half-word of data register D[b] from the contents of data register D[a]. Put the result in each corresponding half-word of data register D[c], with saturation on signed (SUBS.H) or unsigned (SUBS.HU) overflow.

### SUBS.H D[c], D[a], D[b] (RR)

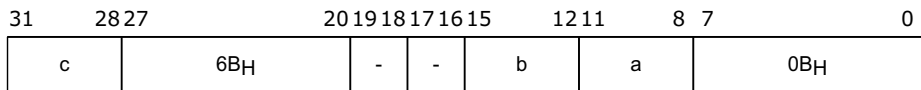


result\_halfword1 = D[a][31:16] - D[b][31:16];

result\_halfword0 = D[a][15:0] - D[b][15:0];

D[c] = {ssov(result\_halfword1, 16), ssov(result\_halfword0, 16)};

### SUBS.HU D[c], D[a], D[b] (RR)



result\_halfword1 = D[a][31:16] - D[b][31:16];

result\_halfword0 = D[a][15:0] - D[b][15:0];

D[c] = {suov(result\_halfword1, 16), suov(result\_halfword0, 16)};

### Status Flags

C	Not set by these instructions.
V	<p>signed:</p> <p>ov_halfword1 = (result_halfword1 &gt; 7FFF<sub>H</sub>) OR (result_halfword1 &lt; -8000<sub>H</sub>);</p> <p>ov_halfword0 = (result_halfword0 &gt; 7FFF<sub>H</sub>) OR (result_halfword0 &lt; -8000<sub>H</sub>);</p> <p>overflow = ov_halfword1 OR ov_halfword0;</p> <p>if (overflow) then PSW.V = 1 else PSW.V = 0;</p> <p>unsigned:</p> <p>ov_halfword1 = (result_halfword1 &gt; FFFF<sub>H</sub>) OR (result_halfword1 &lt; 0000<sub>H</sub>);</p> <p>ov_halfword0 = (result_halfword0 &gt; FFFF<sub>H</sub>) OR (result_halfword0 &lt; 0000<sub>H</sub>);</p> <p>overflow = ov_halfword1 OR ov_halfword0;</p> <p>if (overflow) then PSW.V = 1 else PSW.V = 0;</p>

SV	if (overflow) then PSW.SV = 1 else PSW.SV = PSW.SV;
AV	aov_halfword1 = result_halfword1[15] ^ result_halfword1[14]; aov_halfword0 = result_halfword0[15] ^ result_halfword0[14]; advanced_overflow = aov_halfword1 OR aov_halfword0; if (advanced_overflow) then PSW.AV = 1 else PSW.AV = 0;
SAV	if (advanced_overflow) then PSW.SAV = 1 else PSW.SAV = PSW.SAV;

### Examples

```
subs.h    d3, d1, d2
subs.hu   d3, d1, d2
```

### See Also

[SUB.B](#), [SUB.H](#)

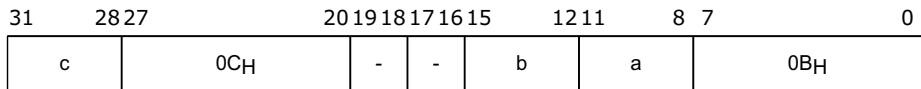
## SUBX

### Subtract Extended

#### Description

Subtract the contents of data register D[b] from the contents of data register D[a] and put the result in data register D[c]. The operands are treated as 32-bit integers. The PSW carry bit is set to the value of the ALU carry out.

#### SUBX D[c], D[a], D[b] (RR)



result = D[a] - D[b];

D[c] = result[31:0];

carry\_out = carry(D[a], ~D[b], 1);

#### Status Flags

C	PSW.C = carry_out;
V	overflow = (result > 7FFFFFFF <sub>H</sub> ) OR (result < -80000000 <sub>H</sub> ); if (overflow) then PSW.V = 1 else PSW.V = 0;
SV	if (overflow) then PSW.SV = 1 else PSW.SV = PSW.SV;
AV	advanced_overflow = result[31] ^ result[30]; if (advanced_overflow) then PSW.AV = 1 else PSW.AV = 0;
SAV	if (advanced_overflow) then PSW.SAV = 1 else PSW.SAV = PSW.SAV;

#### Examples

```
subx    d3, d1, d2
```

#### See Also

[SUB](#), [SUBC](#), [SUBS](#), [SUBS.U](#)



## SVLCX

### Save Lower Context

#### Description

Store the contents of registers A[2] to A[7], D[0] to D[7], A[11] (return address) and PCXI, to the memory location pointed to by the FCX register. This operation saves the lower context of the currently executing task.

#### SVLCX (SYS)

31	28 27	22 21	12 11	8 7	0
-	08H	-	-	-	0DH

if (FCX == 0) trap(FCU);

tmp\_FCX = FCX;

EA = {FCX.FCXS, 6'b0, FCX.FCXO, 6'b0};

new\_FCX = M(EA, word);

M(EA, 16 \* word) = {PCXI, A[11], A[2], A[3], D[0], D[1], D[2], D[3], A[4], A[5], A[6], A[7], D[4], D[5], D[6], D[7]};

PCXI.PCPN = ICR.CCPN

PCXI.PIE = ICR.IE;

PCXI.UL = 0;

PCXI[19:0] = FCX[19:0];

FCX[19:0] = new\_FCX[19:0];

if (tmp\_FCX == LCX) trap(FCD);

#### Status Flags

C	Not set by this instruction.
V	Not set by this instruction.
SV	Not set by this instruction.
AV	Not set by this instruction.
SAV	Not set by this instruction.

#### Examples

```
svlcx
```

**See Also**

**LDLCX, LDUCX, RSLCX, STLCX, STUCX, BISR**

See Also Addendum for TriCore Arch Manual, Vol2, V1.3.8, pages 493/494

## SWAP.W

### Swap with Data Register

#### Description

Swap atomically the contents of data register D[a] and the memory word specified by the addressing mode. Swap enables individual bits or bytes to be used as semaphore.

#### SWAP.W                      off18, D[a] (ABS)                      (Absolute Addressing Mode)

31	28 27 26 25	22 21	16 15	12 11	8 7	0
off18[9:6]	00H	off18[13:10]	off18[5:0]	off18[17:14]	a	E5H

EA = {off18[17:14], 14b'0, off18[13:0]};

tmp = M(EA, word);

M(EA, word) = D[a];

D[a] = tmp[31:0];

#### SWAP.W                      A[b], off10, D[a] (BO) (Base + Short Offset Addressing Mode)

31	28 27	22 21	16 15	12 11	8 7	0
off10[9:6]	20H	off10[5:0]	b	a	49H	

EA = A[b] + sign\_ext(off10);

tmp = M(EA, word);

M(EA, word) = D[a];

A[a] = M(EA, word);

D[a] = tmp[31:0];

#### SWAP.W                      P[b], D[a] (BO)                      (Bit-reverse Addressing Mode)

31	28 27	22 21	16 15	12 11	8 7	0
-	00H	-	b	a	69H	

index = zero\_ext(A[b+1][15:0]);

incr = zero\_ext(A[b+1][31:16]);

EA = A[b] + index;

tmp = M(EA, word);

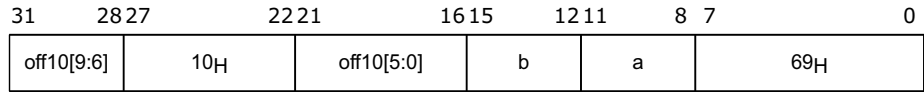
M(EA, word) = D[a];

D[a] = tmp[31:0];

new\_index = reverse16(reverse16(index) + reverse16(incr));

$A[b+1] = \{incr[15:0], new\_index[15:0]\};$

**SWAP.W**                      **P[b], off10, D[a] (BO)**                      **(Circular Addressing Mode)**



$index = zero\_ext(A[b+1][15:0]);$

$length = zero\_ext(A[b+1][31:16]);$

$EA = A[b] + index;$

$tmp = M(EA, word);$

$M(EA, word) = D[a];$

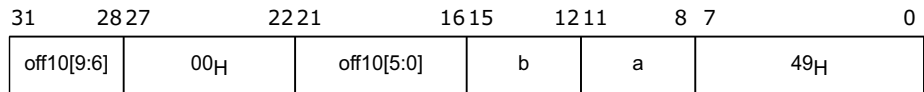
$D[a] = tmp[31:0];$

$new\_index = index + sign\_ext(off10);$

$new\_index = new\_index < 0 ? new\_index + length : new\_index \% length;$

$A[b+1] = \{length[15:0], new\_index[15:0]\};$

**SWAP.W**                      **A[b], off10, D[a] (BO)**                      **(Post-increment Addressing Mode)**



$EA = A[b];$

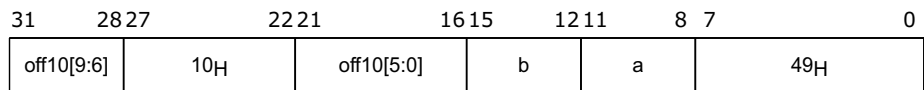
$tmp = M(EA, word);$

$M(EA, word) = D[a];$

$D[a] = tmp[31:0];$

$A[b] = EA + sign\_ext(off10);$

**SWAP.W**                      **A[b], off10, D[a] (BO)**                      **(Pre-increment Addressing Mode)**



$EA = A[b] + sign\_ext(off10);$

$tmp = M(EA, word);$

$M(EA, word) = D[a];$

$D[a] = tmp[31:0];$

$A[b] = EA;$

See Also Addendum for TriCore Arch Manual, Vol2, V1.3.8, pages 493/494

### Status Flags

C	Not set by this instruction.
V	Not set by this instruction.
SV	Not set by this instruction.
AV	Not set by this instruction.
SAV	Not set by this instruction.

### Examples

-

### See Also

[ST.T](#), [LDMST](#)

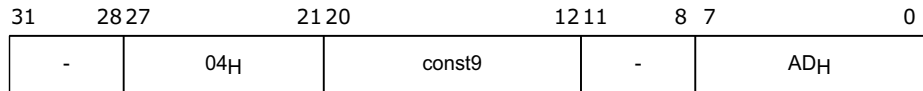
## SYSCALL System Call

### Description

Cause a system call trap, using Trap Identification Number (TIN) specified by const9.

*Note: The trap return PC will be the instruction following the SYSCALL instruction.*

### SYSCALL                      const9 (RC)



trap(SYS, const9[7:0]);

### Status Flags

C	PSW.C is read, but not set by the instruction.
V	PSW.V is read, but not set by the instruction.
SV	PSW.SV is read, but not set by the instruction.
AV	PSW.AV is read, but not set by the instruction.
SAV	PSW.SAV is read, but not set by the instruction.

### Examples

```
syscall 4
```

### See Also

[RET](#), [RFE](#), [TRAPV](#), [TRAPSV](#), [UNPACK](#)

## TRAPSV Trap on Sticky Overflow

### Description

If the PSW sticky overflow status flag (PSW.SV) is set, generate a trap to the vector entry for the sticky overflow trap handler (SOV-trap).

### TRAPSV (SYS)

31	28 27	22 21	12 11	8 7	0
-	15H	-	-	0DH	

if PSW.SV == 1 then trap(SOVF);

### Status Flags

C	Not set by this instruction.
V	Not set by this instruction.
SV	PSW.SV is read, but not set by this instruction.
AV	Not set by this instruction.
SAV	Not set by this instruction.

### Examples

```
trapsv
```

### See Also

[RSTV](#), [SYSCALL](#), [TRAPV](#)

## TRAPV

### Trap on Overflow

#### Description

If the PSW overflow status flag (PSW.V) is set, generate a trap to the vector entry for the overflow trap handler (OVF trap).

#### TRAPV (SYS)

31	28 27	22 21	12 11	8 7	0
-	14H	-	-	0DH	

if PSW.V then trap(OVF);

#### Status Flags

C	Not set by this instruction.
V	PSW.V is read, but not set by this instruction.
SV	Not set by this instruction.
AV	Not set by this instruction.
SAV	Not set by this instruction.

#### Examples

```
trapv
```

#### See Also

[RSTV](#), [SYSCALL](#), [TRAPSV](#), [UNPACK](#)



## UNPACK

### Unpack Floating Point

#### Description

Take an IEEE 754 single precision floating point number in data register D[a] and unpack it as exponent and mantissa into data register pair E[c], such that it can be more easily processed through regular instructions.

The odd register E[c][63:32] receives the unbiased exponent. The even register E[c][31:0] receives the mantissa. Note that the sign-bit of the floating point number is available in bit 31 of data register D[a].

To compute the mantissa and the exponent, the input number is first checked for special cases: Infinity, NAN, Zero & Denormalised. If the input number is not one of these special cases it is a normalised number. Bits [22:0] of D[a] are then copied to bits [29:7] of E[c], with bits [6:0] of E[c] cleared to 0. Bit 30 is set to one, as the implicit high order bit for a normalised mantissa. Bit 31 becomes zero, since the unpacked mantissa is always positive. The bias is removed from the exponent, by subtracting 127, and the result placed in bits [63:32] of E[c].

*Note: For both normalised and denormalised input numbers the output mantissa is in a fractional 2.30 format.*

The special cases are handled as shown in the operation, described below.

#### UNPACK                      E[c], D[a] (RR)

31	28 27	20 19 18 17 16 15	12 11	8 7	0
c	08H	-	0H	-	a                      4BH

fp\_exp[7:0] = D[a][30:23];

fp\_frac[22:0] = D[a][22:0];

```
if (fp_exp == 255) then {
    // Infinity or NaN
    int_exp = +255;
    int_mant = {2'b00, fp_frac[22:0], 7'b00000000};
} else if ((fp_exp == 0) AND (fp_frac == 0)) then {
    // Zero
    int_exp = -127;
    int_mant = 0;
} else if ((fp_exp == 0) AND (fp_frac != 0)) then {
    // Denormalised
```

```

int_exp = -126;
int_mant = {2'b00, fp_frac[22:0], 7'b0000000};
} else {
    // Normalized
    int_exp = fp_exp - 127;
    int_mant = {2'b01, fp_frac[22:0], 7'b0000000};
}

```

E[c][63:32] = int\_exp;

E[c][31:0] = int\_mant;

**Status Flags**

C	Not set by this instruction.
V	Not set by this instruction.
SV	Not set by this instruction.
AV	Not set by this instruction.
SAV	Not set by this instruction.

**Examples**

```
unpack    e2, d5
```

**See Also**

**PACK, SYSCALL, TRAPV**

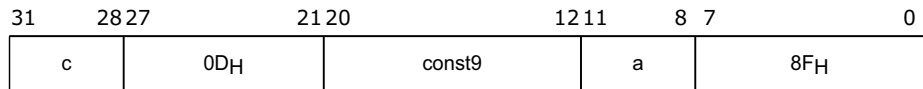
## XNOR

### Bitwise XNOR

#### Description

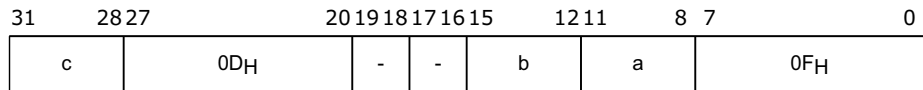
Compute the bitwise exclusive NOR of the contents of data register D[a] and the contents of either data register D[b] (instruction format RR) or const9 (instruction format RC). Put the result in to data register D[c]. The value const9 is zero-extended.

#### XNOR D[c], D[a], const9 (RC)



$$D[c] = \sim(D[a] \wedge \text{zero\_ext}(\text{const9}));$$

#### XNOR D[c], D[a], D[b] (RR)



$$D[c] = \sim(D[a] \wedge D[b]);$$

#### Status Flags

C	Not set by this instruction.
V	Not set by this instruction.
SV	Not set by this instruction.
AV	Not set by this instruction.
SAV	Not set by this instruction.

#### Examples

```
xnor d3, d1, d2
xnor d3, d1, #126
```

#### See Also

**AND, ANDN, NAND, NOR, NOT (16-bit), OR, ORN, XOR**

## XNOR.T

### Bit Logical XNOR

#### Description

Compute the logical exclusive NOR of bit pos1 of data register D[a] and bit pos2 of data register D[b]. Put the result in the least-significant bit of data register D[c] and clear the remaining bits of D[c] to zero.

#### XNOR.T                      D[c], D[a], pos1, D[b], pos2 (BIT)

31	28 27	23 22 21 20	16 15	12 11	8 7	0
c	pos2	02H	pos1	b	a	07H

result = !(D[a][pos1] XOR D[b][pos2]);

D[c] = zero\_ext(result);

#### Status Flags

C	Not set by this instruction.
V	Not set by this instruction.
SV	Not set by this instruction.
AV	Not set by this instruction.
SAV	Not set by this instruction.

#### Examples

```
xnor.t    d3, d1, 3, d2, 5
```

#### See Also

[AND.T](#), [ANDN.T](#), [NAND.T](#), [NOR.T](#), [OR.T](#), [ORN.T](#), [XOR.T](#)

## XOR

### Bitwise XOR

#### Description

Compute the bitwise exclusive OR of the contents of data register D[a] and the contents of either data register D[b] (instruction format RR) or const9 (instruction format RC). Put the result in data register D[c]. The value const9 is zero-extended to 32-bits.

Compute the bitwise exclusive OR of the contents of data register D[a] and the contents of data register D[b]. Put the result in data register D[a].

#### XOR D[c], D[a], const9 (RC)

	31	28 27	21 20	12 11	8 7	0
	c	0CH	const9	a	8FH	

$D[c] = D[a] \wedge \text{zero\_ext}(\text{const9});$

#### XOR D[c], D[a], D[b] (RR)

	31	28 27	20 19	18 17	16 15	12 11	8 7	0
	c	0CH	-	-	b	a	0FH	

$D[c] = D[a] \wedge D[b];$

#### XOR D[a], D[b] (SRR)

	15	12 11	8 7	0
	b	a	C6H	

$D[a] = D[a] \wedge D[b];$

#### Status Flags

C	Not set by this instruction.
V	Not set by this instruction.
SV	Not set by this instruction.
AV	Not set by this instruction.
SAV	Not set by this instruction.

#### Examples

```
xor d3, d1, d2
```

```
xor    d3, d1, #126
```

```
xor    d3, d2
```

**See Also**

[AND](#), [ANDN](#), [NAND](#), [NOR](#), [NOT \(16-bit\)](#), [OR](#), [ORN](#), [XNOR](#)

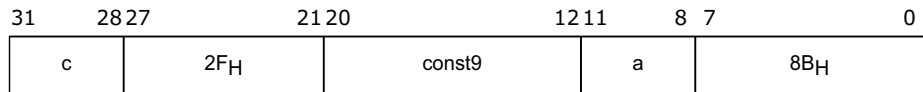
See Also Addendum for TriCore Arch Manual, Vol2, V1.3.8, pages 493/494

## XOR.EQ Equal Accumulating

### Description

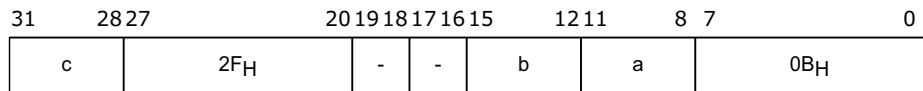
Compute the logical XOR of D[c][0] and the Boolean result of the EQ operation on the contents of data register D[a] and either data register D[b] (instruction format RR) or const9 (instruction format RC). Put the result in D[c][0]. All other bits in D[c] are unchanged. The value const9 is sign-extended.

### XOR.EQ                    D[c], D[a], const9 (RC)



$D[c] = \{D[c][31:1], D[c][0] \text{ XOR } (D[a] == \text{sign\_ext}(\text{const9}))\};$

### XOR.EQ                    D[c], D[a], D[b] (RR)



$D[c] = \{D[c][31:1], D[c][0] \text{ XOR } (D[a] == D[b])\};$

### Status Flags

C	Not set by this instruction.
V	Not set by this instruction.
SV	Not set by this instruction.
AV	Not set by this instruction.
SAV	Not set by this instruction.

### Examples

```
xor.eq  d3, d1, d2
xor.eq  d3, d1, #126
```

### See Also

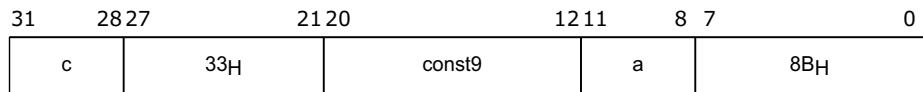
[AND.EQ](#), [OR.EQ](#)

**XOR.GE**  
Greater Than or Equal Accumulating  
**XOR.GE.U**  
Greater Than or Equal Accumulating Unsigned

**Description**

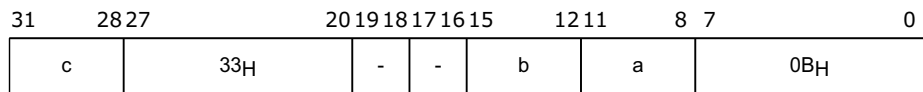
Calculate the logical XOR of D[c][0] and the Boolean result of the GE or GE.U operation on the contents of data register D[a] and either data register D[b] (instruction format RR) or const9 (instruction format RC). Put the result in D[c][0]. All other bits in D[c] are unchanged. D[a] and D[b] are treated as 32-bit signed (XOR.GE) or unsigned (XOR.GE.U) integers. The value const9 is sign-extended (XOR.GE) or zero-extended (XOR.GE.U).

**XOR.GE D[c], D[a], const9 (RC)**



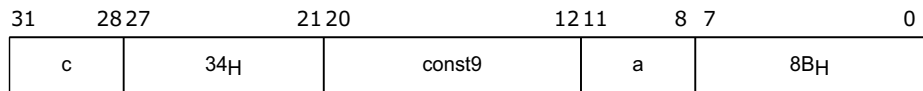
D[c] = {D[c][31:1], D[c][0] XOR (D[a] >= sign\_ext(const9))};

**XOR.GE D[c], D[a], D[b] (RR)**



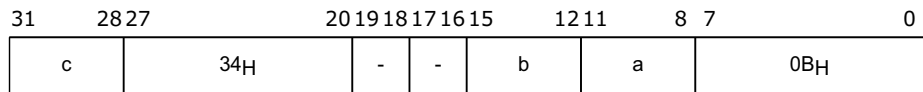
D[c] = {D[c][31:1], D[c][0] XOR (D[a] >= D[b])};

**XOR.GE.U D[c], D[a], const9 (RC)**



D[c] = {D[c][31:1], D[c][0] XOR (D[a] >= zero\_ext(const9))}; // unsigned

**XOR.GE.U D[c], D[a], D[b] (RR)**



D[c] = {D[c][31:1], D[c][0] XOR (D[a] >= D[b])}; // unsigned

See Also Addendum for TriCore Arch Manual, Vol2, V1.3.8, pages 493/494



### Status Flags

C	Not set by these instructions.
V	Not set by these instructions.
SV	Not set by these instructions.
AV	Not set by these instructions.
SAV	Not set by these instructions.

### Examples

```
xor.ge    d3, d1, d2
xor.ge    d3, d1, #126
xor.ge.u  d3, d1, d2
xor.ge.u  d3, d1, #126
```

### See Also

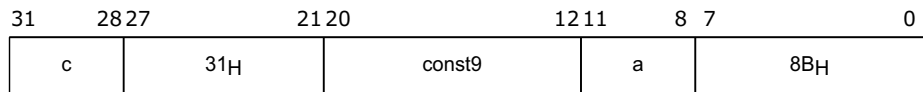
[AND.GE](#), [AND.GE.U](#), [OR.GE](#), [OR.GE.U](#)

**XOR.LT**  
Less Than Accumulating  
**XOR.LT.U**  
Less Than Accumulating Unsigned

**Description**

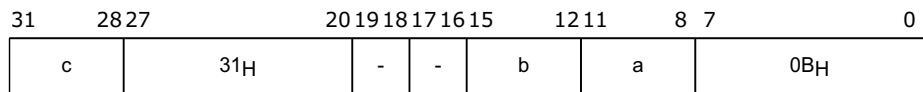
Calculate the logical XOR of D[c][0] and the Boolean result of the LT or LT.U operation on the contents of data register D[a] and either data register D[b] (instruction format RR) or const9 (instruction format RC). Put the result in D[c][0]. All other bits in D[c] are unchanged. D[a] and D[b] are treated as 32-bit signed (XOR.LT) or unsigned (XOR.LT.U) integers. The value const9 is sign-extended (XOR.LT) or zero-extended (XOR.LT.U).

**XOR.LT**                      **D[c], D[a], const9 (RC)**



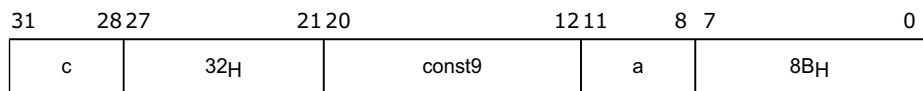
$$D[c] = \{D[c][31:1], D[c][0] \text{ XOR } (D[a] < \text{sign\_ext}(\text{const9}))\};$$

**XOR.LT**                      **D[c], D[a], D[b] (RR)**



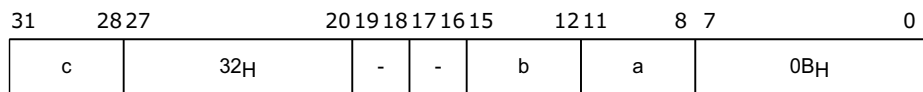
$$D[c] = \{D[c][31:1], D[c][0] \text{ XOR } (D[a] < D[b])\};$$

**XOR.LT.U**                      **D[c], D[a], const9 (RC)**



$$D[c] = \{D[c][31:1], D[c][0] \text{ XOR } (D[a] < \text{zero\_ext}(\text{const9}))\}; // \text{ unsigned}$$

**XOR.LT.U**                      **D[c], D[a], D[b] (RR)**



$$D[c] = \{D[c][31:1], D[c][0] \text{ XOR } (D[a] < D[b])\}; // \text{ unsigned}$$

See Also Addendum for TriCore Arch Manual, Vol2, V1.3.8, pages 493/494

### Status Flags

C	Not set by these instructions.
V	Not set by these instructions.
SV	Not set by these instructions.
AV	Not set by these instructions.
SAV	Not set by these instructions.

### Examples

```
xor.lt    d3, d1, d2
xor.lt    d3, d1, #126
xor.lt.u  d3, d1, d2
xor.lt.u  d3, d1, #126
```

### See Also

[AND.LT](#), [AND.LT.U](#), [OR.LT](#), [OR.LT.U](#)

## XOR.NE

### Not Equal Accumulating

#### Description

Calculate the logical XOR of D[c][0] and the Boolean result of the NE operation on the contents of data register D[a] and either data register D[b] (instruction format RR) or const9. (instruction format RC). Put the result in D[c][0]. All other bits in D[c] are unchanged. The value const9 is sign-extended.

#### XOR.NE D[c], D[a], const9 (RC)

31	28 27	21 20	12 11	8 7	0
c	30 <sub>H</sub>	const9	a	8B <sub>H</sub>	

$D[c] = \{D[c][31:1], D[c][0] \text{ XOR } (D[a] \text{ != sign\_ext(const9)})\};$

#### XOR.NE D[c], D[a], D[b] (RR)

31	28 27	20 19 18 17 16 15	12 11	8 7	0
c	30 <sub>H</sub>	- -	b	a	0B <sub>H</sub>

$D[c] = \{D[c][31:1], D[c][0] \text{ XOR } (D[a] \text{ != } D[b])\};$

#### Status Flags

C	Not set by this instruction.
V	Not set by this instruction.
SV	Not set by this instruction.
AV	Not set by this instruction.
SAV	Not set by this instruction.

#### Examples

```
xor.ne d3, d1, d2
xor.ne d3, d1, #126
```

#### See Also

[AND.NE](#), [OR.NE](#)

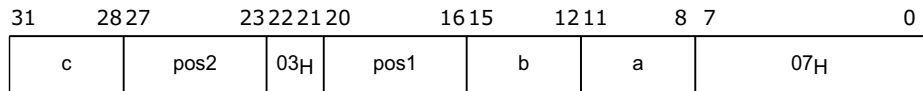
## XOR.T

### Bit Logical XOR

#### Description

Compute the logical XOR of bit pos1 of data register D[a] and bit pos2 of data register D[b]. Put the result in the least-significant bit of data register D[c] and clear the remaining bits of D[c] to zero.

#### XOR.T                      D[c], D[a], pos1, D[b], pos2 (BIT)



result = D[a][pos1] XOR D[b][pos2];

D[c] = zero\_ext(result);

#### Status Flags

C	Not set by this instruction.
V	Not set by this instruction.
SV	Not set by this instruction.
AV	Not set by this instruction.
SAV	Not set by this instruction.

#### Examples

```
xor.t    d3, d1, 3, d2, #7
```

#### See Also

[AND.T](#), [ANDN.T](#), [NAND.T](#), [NOR.T](#), [OR.T](#), [ORN.T](#), [XNOR.T](#)

### 3.2 FPU Instructions

Each page for this group of instructions is laid out as follows:

1
1

**UTOF** 2

**Unsigned to Floating-point**

**Description**

Converts the content of data register D[a] from 32-bit unsigned integer format to floating-point format. The rounded result is stored in D[c].

**UTOF** 4 **D[c], D[a] (RR)**

31	28	27	20	19	18	17	16	15	12	11	0
c	16 <sub>H</sub>			-	-	-	-	-	a	5	4B <sub>H</sub>

rounded\_result = ieee754\_round(u\_real(D[a], PSW.RM));  
 result = ieee754\_32bit\_format(rounded\_result);  
 D[c] = result[31:0];

**Exception Flags**

FS	if(set_FX) then PSW.FS = 1 else PSW.FS = 0;
FI	Not set by this instruction. <span style="border: 1px solid black; border-radius: 50%; padding: 2px 5px;">7</span>
FV	Not set by this instruction.
FZ	Not set by this instruction.
FU	Not set by this instruction.
FX	if(u_real(D[c]) != f_real(D[a])) then set_FX = 1 else set_FX = 0; if(set_FX) then PSW.FX = 1;

**Examples**

utof d2, d1 8

**See Also**

[FTOU](#) 9

Key:

- 1) Instruction Mnemonic
- 2) Instruction Longname
- 3) Description
- 4) Syntax, followed by Instruction Format in parentheses
- 5) Opcodes
- 6) Operation (RTL format)
- 7) Exception Flags. IEEE-754 Exceptions that can occur when using this Instruction
- 8) One or more Instruction examples
- 9) Links to related Instructions

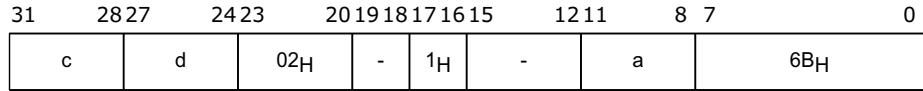
TC1068

## ADD.F Add Float

### Description

Add the contents of data register D[a] to the contents of data register D[d]. Put the result in data register D[c]. The operands and result are single precision IEEE-754 floating-point numbers. If either operand is a NaN (quiet or signalling), then the return result will be the quiet NaN 7FC00000<sub>H</sub>.

### ADD.F                      D[c], D[d], D[a] (RRR)



arg\_a = denorm\_to\_zero(f\_real(D[a]);

arg\_b = denorm\_to\_zero(f\_real(D[d]);

if(is\_nan(D[a]) OR is\_nan(D[d])) then result = QUIET\_NAN;

else if(is\_pos\_inf(D[a]) AND is\_neg\_inf(D[d])) then result = ADD\_NAN;

    else if(is\_neg\_inf(D[a]) AND is\_pos\_inf(D[d])) then result = ADD\_NAN;

    else {

        precise\_result = add(arg\_a, arg\_b);

        normal\_result = denorm\_to\_zero(precise\_result);

        rounded\_result = ieee754\_round(normal\_result, PSW.RM);

        result = ieee754\_32bit\_format(rounded\_result);

    }

D[c] = result[31:0];

### Exception Flags

FS	if(set_FI OR set_FV OR set_FU OR set_FX) then PSW.FS = 1 else PSW.FS = 0;
FI	if(is_s_nan(D[a]) OR is_s_nan(D[d])) then set_FI = 1 else set_FI = 0; if(set_FI) then PSW.FI = 1;
FV	if(rounded_result >= 2 <sup>128</sup> ) then set_FV = 1 else set_FV = 0; if(set_FV) then PSW.FV = 1;
FZ	Not set by this instruction.
FU	if(fp_abs(precise_result) < 2 <sup>-126</sup> ) then set_FU = 1 else set_FU = 0; if(set_FU) then PSW.FU = 1;

FX	if(precise_result != f_real(result)) then set_FX = 1 else set_FX = 0; if(set_FX) then PSW.FX = 1;
----	--

### Examples

add.f d3, d1, d2

### See Also

**SUB.F**

See Also Addendum for TriCore Arch Manual, Vol2, V1.3.8, pages 493/494



## CMP.F Compare Float

### Description

This instruction compares the IEEE-754 single-precision floating-point operands and asserts bits in the result if their associated condition is true:

bit [0]  $D[a] < D[b]$

bit [1]  $D[a] == D[b]$

bit [2]  $D[a] > D[b]$

bit [3] Unordered

bit [4]  $D[a]$  is denormal

bit [5]  $D[b]$  is denormal

bits[31:06] are cleared.

The 'unordered' bit is asserted if either operand is a NaN.

*Note: CMP.F is the only FPU instruction that does not substitute denormal operands for zero before computation.*

### CMP.F                      D[c], D[a], D[b] (RR)

31	28 27	20 19 18 17 16 15	12 11	8 7	0
c	00H	-	1H	b	a
4BH					

$D[c][0] = \text{ieee754\_lt}(D[a], D[b]);$

$D[c][1] = \text{ieee754\_eq}(D[a], D[b]);$

$D[c][2] = \text{ieee754\_gt}(D[a], D[b]);$

$D[c][3] = (\text{is\_nan}(D[a]) \text{ OR } \text{is\_nan}(D[b]));$

$D[c][4] = \text{is\_denorm}(D[a]);$

$D[c][5] = \text{is\_denorm}(D[b]);$

### Exception Flags

FS	if(set_FI) then PSW.FS = 1 else PSW.FS = 0;
FI	if(is_s_nan(D[a]) OR is_s_nan(D[b])) then set_FI = 1; if(set_FI) then PSW.FI = 1;
FV	Not set by this instruction.
FZ	Not set by this instruction.
FU	Not set by this instruction.

FX	Not set by this instruction.
----	------------------------------

**Examples**

cmp.f d3, d1, d2

**See Also**

-

See Also Addendum for TriCore Arch Manual, Vol2, V1.3.8, pages 493/494



FU	if(fp_abs(precise_result) < 2 <sup>-126</sup> ) then set_FU = 1 else set_FU = 0; if(set_FU) then PSW.FU = 1;
FX	if(precise_result != f_real(result)) then set_FX = 1 else set_FX = 0; if(set_FX) then PSW.FX = 1;

### Examples

div.f d3, d1, d2

### See Also

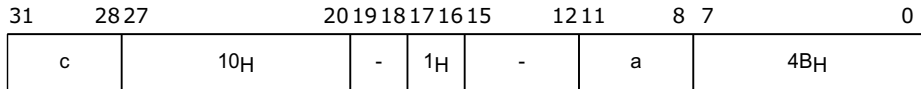
-

## FTOI Float to Integer

### Description

Converts the contents of data register D[a] from floating-point format to a 32-bit two's complement signed integer format. The rounded result is put in data register D[c]. The rounding mode used for the conversion is defined by the PSW.RM field.

### FTOI                      D[c], D[a] (RR)



if(is\_nan(D[a])) then result = 0;

else if(f\_real(D[a]) > 2<sup>31</sup>-1) then result = 7FFFFFFF<sub>H</sub>;

else if(f\_real(D[a]) < -2<sup>31</sup>) then result = 80000000<sub>H</sub>;

else result = round\_to\_integer(D[a], PSW.RM);

D[c] = result[31:0];

### Exception Flags

FS	if(set_FI OR set_FX) then PSW.FS = 1 else PSW.FS = 0;
FI	if((f_real(D[a]) > 2 <sup>31</sup> -1) OR (f_real(D[a]) < -2 <sup>31</sup> ) OR is_nan(D[a])) then set_FI = 1 else set_FI = 0; if(set_FI) then PSW.FI = 1;
FV	Not set by this instruction.
FZ	Not set by this instruction.
FU	Not set by this instruction.
FX	if(f_real(D[a]) != i_real(result)) then set_FX = 1 else set_FX = 0; if(set_FX) then PSW.FX = 1;

### Examples

```
ftoi d2, d1
```

### See Also

[ITOF](#), [FTOIZ](#)

See Also Addendum for TriCore Arch Manual, Vol2, V1.3.8, pages 493-494

## FTOIZ

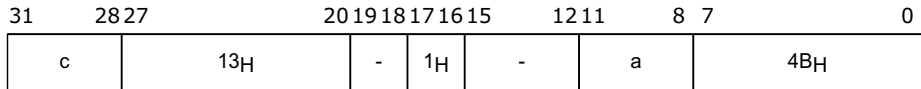
### Float to Integer, Round towards Zero

1.3.1

#### Description

Converts the contents of data register D[a] from floating-point format to a 32-bit two's complement signed integer format. The result is rounded towards zero and put in data register D[c].

#### FTOIZ                      D[c], D[a] (RR)



if(is\_nan(D[a])) then result = 0;

else if(f\_real(D[a]) > 2<sup>31</sup>-1) then result = 7FFFFFFF<sub>H</sub>;

else if(f\_real(D[a]) < -2<sup>31</sup>) then result = 80000000<sub>H</sub>;

else result = round\_to\_integer(D[a], 11<sub>B</sub>);

D[c] = result[31:0];

#### Exception Flags

FS	if(set_FI OR set_FX) then PSW.FS = 1 else PSW.FS = 0;
FI	if((f_real(D[a]) > 2 <sup>31</sup> -1) OR (f_real(D[a]) < -2 <sup>31</sup> ) OR is_nan(D[a])) then set_FI = 1 else set_FI = 0; if(set_FI) then PSW.FI = 1;
FV	Not set by this instruction.
FZ	Not set by this instruction.
FU	Not set by this instruction.
FX	if(f_real(D[a]) != i_real(result)) then set_FX = 1 else set_FX = 0; if(set_FX) then PSW.FX = 1;

#### Examples

```
ftoiz d2, d1
```

#### See Also

[ITOF](#), [FTOI](#)

## FTOQ31 Float to Fraction

### Description

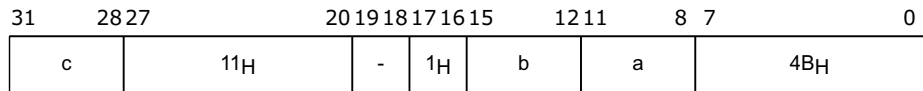
Subtracts D[b] from the exponent of the floating-point input value D[a] and converts the result to the Q31fraction format. The result is stored in D[c]. The rounding mode used for the conversion is defined by the PSW.RM field.

The exponent adjustment is a 9-bit two's complement number taken from D[b][8:0], with a value of [-256, 255]. D[b][31:9] is ignored.

Q31 fraction format is a 32-bit two's complement format which represents a value in the range [-1,1).

- Bit 31 represents -1
- Bit 30 represents +1/2
- Bit 29 represents +1/4
- Bit 28 represents +1/8
- etc.

### FTOQ31                      D[c], D[a], D[b] (RR)



arg\_a = denorm\_to\_zero(f\_real(D[a]));

if(is\_nan(D[a])) then result = 0;

else precise\_result = mul(arg\_a, 2<sup>-D[b][8:0]</sup>);

if(precise\_result > q\_real(7FFFFFFF<sub>H</sub>)) then result = 7FFFFFFF<sub>H</sub>;

else if(precise\_result < -1.0) then result = 80000000<sub>H</sub>;

else result = round\_to\_q31(precise\_result);

D[c] = result[31:0];

### Exception Flags

FS	if(set_FI OR set_FX) then PSW.FS = 1 else PSW.FS = 0;
FI	if((precise_result > q_real(7FFFFFFF <sub>H</sub> )) OR (precise_result < -1.0) OR is_nan(D[a])) then set_FI = 1 else set_FI = 0; if(set_FI) then PSW.FI = 1;
FV	Not set by this instruction.
FZ	Not set by this instruction.
FU	Not set by this instruction.

FX	if(f_real(D[a]) != q_real(result)) then set_FX = 1 else set_FX = 0; if(set_FX) then PSW.FX = 1;
----	--

### Examples

ftoq31      d3, d1, d2

### See Also

[Q31TOF](#), [FTOQ31Z](#)

See Also Addendum for TriCore Arch Manual, Vol2, V1.3.8, pages 493/494



## FTOQ31Z

### Float to Fraction, Round towards Zero

1.3.1

#### Description

Subtracts  $D[b]$  from the exponent of the floating-point input value  $D[a]$  and converts the result to the Q31fraction format. The result is rounded towards zero and stored in  $D[c]$ .

The exponent adjustment is a 9-bit two's complement number taken from  $D[b][8:0]$ , with a value of  $[-256, 255]$ .  $D[b][31:9]$  is ignored.

Q31 fraction format is a 32-bit two's complement format which represents a value in the range  $[-1, 1)$ .

Bit 31 represents -1

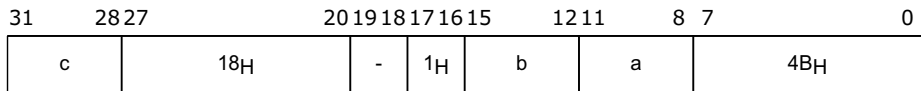
Bit 30 represents  $+1/2$

Bit 29 represents  $+1/4$

Bit 28 represents  $+1/8$

etc.

#### FTOQ31Z                      $D[c], D[a], D[b]$ (RR)



$arg\_a = denorm\_to\_zero(f\_real(D[a]));$

$if(is\_nan(D[a]))$  then result = 0;

else  $precise\_result = mul(arg\_a, 2^{-D[b][8:0]});$

$if(precise\_result > q\_real(7FFFFFFF_H))$  then result =  $7FFFFFFF_H;$

else  $if(precise\_result < -1.0)$  then result =  $80000000_H;$

else result =  $round\_to\_q31(precise\_result, 11_B);$

$D[c] = result[31:0];$

#### Exception Flags

FS	$if(set\_FI \text{ OR } set\_FX)$ then PSW.FS = 1 else PSW.FS = 0;
FI	$if(((precise\_result > q\_real(7FFFFFFF_H)) \text{ OR } (precise\_result < -1.0) \text{ OR } is\_nan(D[a]))$ then set_FI = 1 else set_FI = 0; $if(set\_FI)$ then PSW.FI = 1;
FV	Not set by this instruction.
FZ	Not set by this instruction.

FU	Not set by this instruction.
FX	if(f_real(D[a]) != q_real(result)) then set_FX = 1 else set_FX = 0; if(set_FX) then PSW.FX = 1;

### Examples

ftoq31z    d3, d1, d2

### See Also

[Q31TOF](#), [FTOQ31](#)

## FTOU

### Float to Unsigned

#### Description

Converts the contents of data register D[a] from floating-point format to a 32-bit unsigned integer format. The rounded result is put in data register D[c].

#### FTOU                      D[c], D[a] (RR)

31	28 27	20 19 18 17 16 15	12 11	8 7	0
c	12H	-	1H	-	a
4BH					

if(is\_nan(D[a])) then result = 0;

else if(f\_real(D[a]) > 2<sup>32</sup>-1) then result = FFFFFFFFH;

    else if(f\_real(D[a]) < 0.0) then result = 0;

        else result = round\_to\_unsigned(D[a], PSW.RM);

D[c] = result[31:0];

#### Exception Flags

FS	if(set_FI OR set_FX) then PSW.FS = 1 else PSW.FS = 0;
FI	if((f_real(D[a]) > 2 <sup>32</sup> -1) OR (f_real(D[a]) < 0.0) OR is_nan(D[a])) then set_FI = 1 else set_FI = 0; if(set_FI) then PSW.FI = 1;
FV	Not set by this instruction.
FZ	Not set by this instruction.
FU	Not set by this instruction.
FX	if(f_real(D[a]) != u_real(result)) then set_FX = 1 else set_FX = 0; if(set_FX) then PSW.FX = 1;

#### Examples

ftou                      d2, d1

#### See Also

[UTOF](#), [FTOUZ](#)

See Also Addendum for TriCore Arch Manual, Vol2, V1.3.8, pages 493-494

## FTOUZ

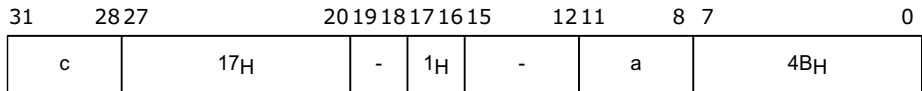
### Float to Unsigned, Round towards Zero

1.3.1

#### Description

Converts the contents of data register D[a] from floating-point format to a 32-bit unsigned integer format. The result is rounded towards zero and put in data register D[c].

#### FTOUZ                      D[c], D[a] (RR)



```
if(is_nan(D[a])) then result = 0;
else if(f_real(D[a]) > 232-1) then result = FFFFFFFFH;
    else if(f_real(D[a]) < 0.0) then result = 0;
        else result = round_to_unsigned(D[a], 11B);
D[c] = result[31:0];
```

#### Exception Flags

FS	if(set_FI OR set_FX) then PSW.FS = 1 else PSW.FS = 0;
FI	if((f_real(D[a]) > 2 <sup>32</sup> -1) OR (f_real(D[a]) < 0.0) OR is_nan(D[a])) then set_FI = 1 else set_FI = 0; if(set_FI) then PSW.FI = 1;
FV	Not set by this instruction.
FZ	Not set by this instruction.
FU	Not set by this instruction.
FX	if(f_real(D[a]) != u_real(result)) then set_FX = 1 else set_FX = 0; if(set_FX) then PSW.FX = 1;

#### Examples

```
ftouz            d2, d1
```

#### See Also

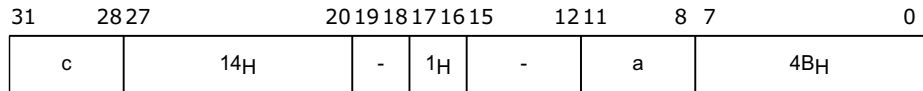
**UTOF, FTOU**

## ITOF Integer to Float

### Description

Converts the contents of data register D[a] from 32-bit two's complement signed integer format to floating-point format. The rounded result is put in data register D[c].

### ITOF                      D[c], D[a] (RR)



rounded\_result = ieee754\_round(i\_real(D[a]), PSW.RM);

result = ieee754\_32bit\_format(rounded\_result);

D[c] = result[31:0];

### Exception Flags

FS	if(set_FX) then PSW.FS = 1 else PSW.FS = 0;
FI	Not set by this instruction.
FV	Not set by this instruction.
FZ	Not set by this instruction.
FU	Not set by this instruction.
FX	if(f_real(result) != i_real(D[a])) then set_FX = 1 else set_FX = 0; if(set_FX) then PSW.FX = 1;

### Examples

```
itof    d2, d1
```

### See Also

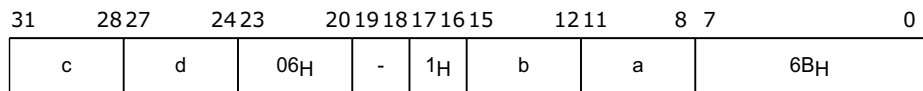
**FTOI, FTOIZ**

## MADD.F Multiply Add Float

### Description

Multiplies D[a] and D[b] and adds the product to D[d]. The result is put in D[c]. The operands and result are floating-point numbers. If an operand is a NaN (quiet or signalling), then the return result will be the quiet NaN 7FC00000<sub>H</sub>.

### MADD.F                    D[c], D[d], D[a], D[b] (RRR)



```

arg_a = denorm_to_zero(f_real(D[a]);
arg_b = denorm_to_zero(f_real(D[b]);
arg_c = denorm_to_zero(f_real(D[d]);
if(is_nan(D[a]) OR is_nan(D[b]) OR is_nan(D[d])) then result = QUIET_NAN;
  else if(is_inf(D[a]) AND is_zero(D[b])) then result = MUL_NAN;
  else if(is_zero(D[a]) AND is_inf(D[b])) then result = MUL_NAN;
  else if(((is_neg_inf(D[a]) AND is_neg_inf(D[b])) OR
    ((is_pos_inf(D[a]) AND is_pos_inf(D[b]))) AND
    is_neg_inf(D[d])) then result = ADD_NAN;
  else if(((is_neg_inf(D[a]) AND is_pos_inf(D[b])) OR
    ((is_pos_inf(D[a]) AND is_neg_inf(D[b]))) AND
    is_pos_inf(D[d])) then result = ADD_NAN;
  else {
    precise_mul_result = mul(arg_a, arg_b);
    precise_result = add(precise_mul_result, arg_c);
    normal_result = denorm_to_zero(precise_result);
    rounded_result = ieee754_round(normal_result,
    PSW.RM);
    result = ieee754_32bit_format(rounded_result);
  }
D[c] = result[31:0];

```

See Also Addendum for TriCore Arch Manual, Vol2, V1.3.8, pages 493/494

### Exception Flags

FS	if(set_FI OR set_FV OR set_FU or set_FX) then PSW.FS = 1 else PSW.FS = 0;
FI	if(is_s_nan(D[a]) OR is_s_nan(D[b]) OR is_s_nan(D[d]) OR (result == ADD_NAN) OR (result == MUL_NAN)) then set_FI = 1 else set_FI = 0; if(set_FI) then PSW.FI = 1;
FV	if(rounded_result >= 2 <sup>128</sup> ) then set_FV = 1 else set_FV = 0; if(set_FV) then PSW.FV = 1;
FZ	Not set by this instruction.
FU	if(fp_abs(precise_result) < 2 <sup>-126</sup> ) then set_FU = 1 else set_FU = 0; if(set_FU) then PSW.FU = 1;
FX	if(precise_result != f_real(result)) then set_FX = 1 else set_FX = 0; if(set_FX) then PSW.FX = 1;

### Examples

madd.f      d4, d3, d1, d2

### See Also

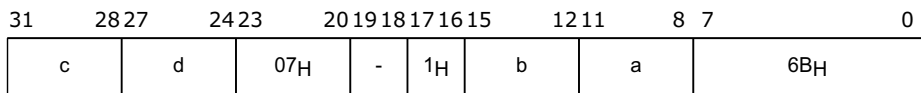
**MSUB.F, MUL**

## MSUB.F Multiply Subtract Float

### Description

Multiplies D[a] and D[b] and subtracts the product from D[d], putting the result in D[c]. The operands and result are floating-point numbers. If any operand is a NaN (quiet or signalling), then the return result will be the quiet NaN 7FC0 0000<sub>H</sub>.

### MSUB.F                      D[c], D[d], D[a], D[b] (RRR)



```

arg_a = denorm_to_zero(f_real(D[a]);
arg_b = denorm_to_zero(f_real(D[b]);
arg_c = denorm_to_zero(f_real(D[d]);
if(is_nan(D[a]) OR is_nan(D[b]) OR is_nan(D[d])) then result = QUIET_NAN;
else if(is_inf(D[a]) AND is_zero(D[b])) then result = MUL_NAN;
    else if(is_zero(D[a]) AND is_inf(D[b])) then result = MUL_NAN;
        else if(((is_neg_inf(D[a]) AND is_neg_inf(D[b])) OR
            ((is_pos_inf(D[a]) AND is_pos_inf(D[b])) AND
            is_pos_inf(D[d])) then result = ADD_NAN;
            else if(((is_neg_inf(D[a]) AND is_pos_inf(D[b])) OR
                ((is_pos_inf(D[a]) AND is_neg_inf(D[b])) AND
                is_neg_inf(D[b])) then result = ADD_NAN;
                else {
                    precise_mul_result = ieee754_mul(arg_a, arg_b);
                    precise_result = ieee754_add(-precise_mul_result, arg_c);
                    normal_result = denorm_to_zero(precise_result);
                    rounded_result = ieee754_round(normal_result, PSW.RM);
                    result = ieee754_32bit_format(rounded_result);
                }
D[c] = result[31:0];

```

See Also Addendum for TriCore Arch Manual, Vol2, V1.3.8, pages 493/494



### Exception Flags

FS	if(set_FI OR set_FV OR set_FU OR set_FX) then PSW.FS = 1 else PSW.FS = 0;
FI	if(is_s_nan(D[a]) OR is_s_nan(D[b]) OR is_s_nan(D[d]) OR (result == ADD_NAN) OR (result == MUL_NAN)) then set_FI = 1 else set_FI = 0; if(set_FI) then PSW.FI = 1;
FV	if(rounded_result >= 2 <sup>128</sup> ) then set_FV = 1 else set_FV = 0; if(set_FV) then PSW.FV = 1;
FZ	Not set by this instruction.
FU	if(fp_abs(precise_result) < 2 <sup>-126</sup> ) then set_FU = 1 else set_FU = 0; if(set_FU) then PSW.FU = 1;
FX	if(precise_result != f_real(result)) then set_FX = 1 else set_FX = 0; if(set_FX) then PSW.FX = 1;

### Examples

msub.f      d4, d3, d1, d2

### See Also

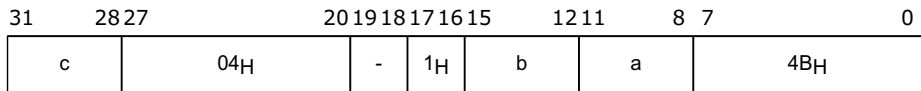
[MADD.F](#)

## MUL.F Multiply Float

### Description

Multiplies D[a] and D[b] and stores the result in D[c]. The operands and result are floating-point numbers. If an operand is a NaN (quiet or signalling), then the return result will be the quiet NaN 7FC00000<sub>H</sub>.

### MUL.F                      D[c], D[a], D[b] (RR)



arg\_a = denorm\_to\_zero(f\_real(D[a]);

arg\_b = denorm\_to\_zero(f\_real(D[b]);

if(is\_nan(D[a]) OR is\_nan(D[b])) then result = QUIET\_NAN;

else if(is\_inf(D[a]) AND is\_zero(D[b])) then result = MUL\_NAN;

else if(is\_inf(D[b]) AND is\_zero(D[a])) then result = MUL\_NAN;

else {

precise\_result = mul(arg\_a, arg\_b);

normal\_result = denorm\_to\_zero(precise\_result);

rounded\_result = ieee754\_round(normal\_result, PSW.RM);

result = ieee754\_32bit\_format(rounded\_result);

}

D[c] = result[31:0];

### Exception Flags

FS	if(set_FI OR set_FV OR set_FU OR set_FX) then PSW.FS = 1 else PSW.FS = 0;
FI	if(is_s_nan(D[a]) OR is_s_nan(D[b]) OR (result == MUL_NAN)) then set_FI = 1 else set_FI = 0; if(set_FI) then PSW.FI = 1;
FV	if(rounded_result >= 2 <sup>128</sup> ) then set_FV = 1 else set_FV = 0; if(set_FV) then PSW.FV = 1;
FZ	Not set by this instruction.
FU	if(fp_abs(precise_result) < 2 <sup>-126</sup> ) then set_FU = 1 else set_FU = 0; if(set_FU) then PSW.FU = 1;

---

FX	if(precise_result != f_real(result)) then set_FX = 1 else set_FX = 0; if(set_FX) then PSW.FX = 1;
----	--

---

**Examples**

mul.f        d3, d1, d2

**See Also**

-

See Also Addendum for TriCore Arch Manual, Vol2, V1.3.8, pages 493/494

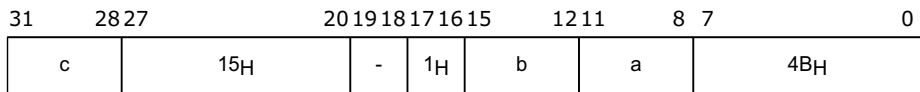
## Q31TOF Fraction to Floating-point

### Description

Converts the D[a] from Q31 fraction format to floating-point format, then adds D[b] to the exponent and stores the resulting value in D[c]. The exponent adjustment is a 9-bit two's complement number taken from D[b][8:0], with a value in the range [-256, 255]. D[b][31:9] is ignored. Q31 fraction format is a 32-bit two's complement format which represents a value in the range [-1,1).

- Bit 31 represents -1
- Bit 30 represents +1/2
- Bit 29 represents +1/4
- etc.

### Q31TOF                      D[c], D[a], D[b] (RR)



```

precise_result = mul(q_real(D[a]),2D[b][8:0]);
rounded_result = ieee754_round(precise_result, PSW.RM);
result = ieee754_32bit_format(rounded_result);
D[c] = result[31:0];

```

### Exception Flags

FS	if(set_FX) then PSW.FS = 1 else PSW.FS = 0;
FI	Not set by this instruction.
FV	Not set by this instruction.
FZ	Not set by this instruction.
FU	Not set by this instruction.
FX	if(precise_result != f_real(result)) then set_FX = 1 else set_FX = 0; if(set_FX) then PSW.FX = 1;

### Examples

```
q31tof        d3, d1, d2
```

### See Also

[FTOQ31](#), [FTOQ31Z](#)

See Also Addendum for TriCore Arch Manual, Vol2, V1.3.8, pages 493/494

## QSEED.F Inverse Square Root Seed

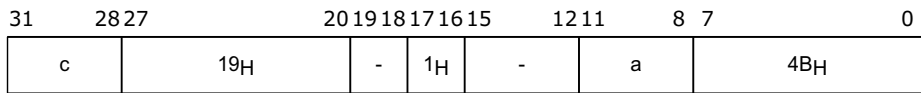
### Description

An approximation of the reciprocal of the square root of  $D[a]$  is stored in  $D[c]$ . The accuracy of the result is no less than 6.75 bits, and therefore always within  $\pm 1\%$  of the accurate result.

The operand and result are floating-point numbers. If the operand is  $\pm 0$  then the result will be the appropriately signed  $\infty$ . If the operand is a NaN (quiet or signalling), then the return result will be the quiet NaN 7FC00000<sub>H</sub>.

This instruction can be used to implement a floating-point square root function in software using the Newton-Raphson iterative method.

### QSEED.F                      D[c], D[a] (RR)



```
arg_a = denorm_to_zero(f_real(D[a]);
if(is_nan(D[a])) then result = QUIET_NAN;
else if(arg_a == +0.0) then result = POS_INFINITY;
    else if(arg_a == -0.0) then result = NEG_INFINITY;
        else if(arg_a < 0.0) then result = SQRT_NAN;
            else {
                normal_result = approx_inv_sqrt(arg_a);
                result = ieee754_32bit_format(normal_result);
            }
}
```

$D[c] = \text{result}[31:0];$

### Exception Flags

FS	if(set_FI) then PSW.FS = 1 else PSW.FS = 0;
FI	if(is_s_nan(D[a]) OR (D[c] == SQRT_NAN)) then set_FI = 1 else set_FI = 0; if(set_FI) then PSW.FI = 1;
FV	Not set by this instruction.
FZ	Not set by this instruction.
FU	Not set by this instruction.
FX	Not set by this instruction.

**Examples**

qseed.f      d2, d1

**See Also**

-

See Also Addendum for TriCore Arch Manual, Vol2, V1.3.8, pages 493/494

## SUB.F Subtract Float

### Description

Subtracts D[a] from D[d] and stores the result in D[c]. The operands and result are floating-point numbers.

If any operand is a NaN (quiet or signalling), then the return result will be the quiet NaN 7FC00000<sub>H</sub>.

### SUB.F                      D[c], D[d], D[a] (RRR)

31	28 27	24 23	20 19	18 17	16 15	12 11	8 7	0
c	d	03 <sub>H</sub>	-	1 <sub>H</sub>	-	a	6B <sub>H</sub>	

arg\_a = denorm\_to\_zero(f\_real(D[a]);

arg\_b = denorm\_to\_zero(f\_real(D[d]);

if(is\_nan(D[a]) OR is\_nan(D[b])) then result = QUIET\_NAN;

else if(is\_pos\_inf(D[a]) AND is\_pos\_inf(D[b])) then result = ADD\_NAN;

    else if(is\_neg\_inf(D[a]) AND is\_neg\_inf(D[b])) then result = ADD\_NAN;

    else {

        precise\_result = add(-arg\_a, arg\_b);

        normal\_result = denorm\_to\_zero(precise\_result);

        rounded\_result = ieee754\_round(normal\_result, PSW.RM);

        result = ieee754\_32bit\_format(rounded\_result);

    }

D[c] = result[31:0];

### Exception Flags

FS	if(set_FI OR set_FV OR set_FU OR set_FX) then PSW.FS = 1 else PSW.FS = 0;
FI	if((is_s_nan(D[a]) OR is_s_nan(D[b]))) then set_FI = 1 else set_FI = 0; if(set_FI) then PSW.FI = 1;
FV	if(rounded_result >= 2 <sup>128</sup> ) then set_FV = 1 else set_FV = 0; if(set_FV) then PSW.FV = 1;
FZ	Not set by this instruction.
FU	if(fp_abs(precise_result) < 2 <sup>-126</sup> ) then set_FU = 1 else set_FU = 0; if(set_FU) then PSW.FU = 1;

---

FX	if(precise_result != f_real(result)) then set_FX = 1 else set_FX = 0; if(set_FX) then PSW.FX = 1;
----	--

---

**Examples**

sub.f        d3, d1, d2

**See Also**

[ADD.F](#)

See Also Addendum for TriCore Arch Manual, Vol2, V1.3.8, pages 493/494



## UPDFL Update Flags

### Description

The UPDFL instruction takes two 8-bit data fields from D[a], and uses them to update the PSW user flag bits (PSW [31:24]) that the FPU uses to store its exception flags and rounding mode in. D[a][15:8] are the update mask field; a '1' in a given bit position indicates that the corresponding PSW user flag bit is to be updated. D[a][7:0] are the update value field. These bits supply the values to be written to the PSW user flags bits, in the positions specified by the mask field.

Example: Changing the current PSW[25:24] (Rounding mode) to round toward  $+\infty$ , without modifying any of the current exception flag settings, can be accomplished by loading the literal value 0301<sub>H</sub> into register D[0], and issuing the instruction, UPDFL D[0].

UPDFL can be used to create exceptions that cause asynchronous traps to be taken.

### UPDFL                      D[a] (RR)

31	28 27	20 19 18 17 16 15	12 11	8 7	0
-	0C <sub>H</sub>	-	1 <sub>H</sub>	a	4B <sub>H</sub>

set\_FS = (PSW.FS & ~D[a][15]) | (D[a][7] & D[a][15]);

set\_FI = (PSW.FI & ~D[a][14]) | (D[a][6] & D[a][14]);

set\_FV = (PSW.FV & ~D[a][13]) | (D[a][5] & D[a][13]);

set\_FZ = (PSW.FZ & ~D[a][12]) | (D[a][4] & D[a][12]);

set\_FU = (PSW.FU & ~D[a][11]) | (D[a][3] & D[a][11]);

set\_FX = (PSW.FX & ~D[a][10]) | (D[a][2] & D[a][10]);

set\_RM = (PSW.RM & ~D[a][9:8]) | (D[a][1:0] & D[a][9:8]);

PSW.[31:24] = {set\_FS, set\_FI, set\_FV, set\_FZ, set\_FU, set\_FX, set\_RM};

### Exception Flags

FS	PSW.FS = set_FS;
FI	PSW.FI = set_FI;
FV	PSW.FV = set_FV;
FZ	PSW.FZ = set_FZ;
FU	PSW.FU = set_FU;
FX	PSW.FX = set_FX;

**Examples**

updf1      d1

**See Also**

-

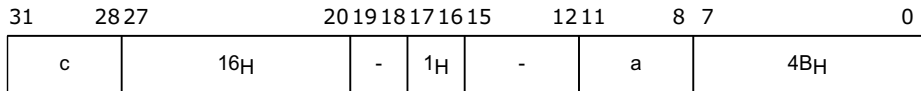
See Also Addendum for TriCore Arch Manual, Vol2, V1.3.8, pages 493/494

## UTOF Unsigned to Floating-point

### Description

Converts the contents of data register D[a] from 32-bit unsigned integer format to floating-point format. The rounded result is stored in D[c].

### UTOF                      D[c], D[a] (RR)



rounded\_result = ieee754\_round(u\_real(D[a]), PSW.RM);

result = ieee754\_32bit\_format(rounded\_result);

D[c] = result[31:0];

### Exception Flags

FS	if(set_FX) then PSW.FS = 1 else PSW.FS = 0;
FI	Not set by this instruction.
FV	Not set by this instruction.
FZ	Not set by this instruction.
FU	Not set by this instruction.
FX	if(u_real(D[c]) != f_real(D[a])) then set_FX = 1 else set_FX = 0; if(set_FX) then PSW.FX = 1;

### Examples

utof                      d2, d1

### See Also

**FTOU, FTOUZ**

See Also Addendum for TriCore Arch Manual, Vol2, V1.3.8, pages 493/494

### 3.3 MMU Instructions

Each page for this group of instructions is laid out as follows:

**J** 1

**Jump Unconditional** 2

**Description** 3

Add the value specified by disp24, multiplied by two and sign-extended to 32-bits, to the contents of PC and jump to that address.

Add the value specified by disp8, multiplied by two and sign-extended to 32-bits, to the contents of PC and jump to that address. 4

**J** 5 **disp24 (B)**

31	16	15	8	7	0
disp24[15:0]			d[3:16]		1D <sub>H</sub>

PC = PC + sign\_ext(2 \* disp24); 7

**J** 8 **disp8 (SB)**

15	8	7	0
disp8		3C <sub>H</sub>	

PC = PC + sign\_ext(2 \* disp8); 10

**Status Flags**

C	Not set by this instruction. <span style="border: 1px solid black; border-radius: 50%; padding: 2px;">11</span>
V	Not set by this instruction.
SV	Not set by this instruction.
AV	Not set by this instruction.
SAV	Not set by this instruction.

**Examples**

J foobar; 12

J foobar; 13

**See Also**

[JA](#), [JI](#), [JL](#), [JLA](#), [JLI](#) 14

Key:

- 1 Instruction Mnemonic
  - 2 Instruction Longname
  - 3 Description (32-bit)
  - 4 Description (16-bit)
  - 5 Syntax (32-bit), and Instruction format in parentheses. Note also 15
  - 6 Opcodes (32-bit)
  - 7 Operation in RTL format (32-bit)
  - 8 Syntax (16-bit)
  - 9 Opcodes (16-bit)
  - 10 Operation (RTL) (16-bit)
  - 11 Status Flags (User Status Bits)
  - 12 Instruction Examples (32-bit)
  - 13 Instruction Examples (16-bit)
  - 14 Related instructions
  - 15 Operation quick reference following Syntax; see 5 (MAC instructions only)
- MSUB**      **D(c), D(d)**      **const9 (RCR)**  
32 - (32 \* K9) --> 32 signed 15

TC1066

## TLBDEMAP

### Translation Lookaside Buffer Demap

#### Description

*Note: The TLBDEMAP instruction can only be executed in Supervisor mode.*

The TLBDEMAP instruction is used to uninstall a mapping in the MMU (Memory Management Unit). TLBDEMAP takes as a parameter, a data register that contains the virtual address whose mapping is to be removed. The Address Space Identifier (ASI) for the demap operation is obtained from the ASI register. Note that demapping a translation that does not exist in the MMU results in a NOP.

*Note: A TLBDEMAP instruction should be followed by an **ISYNC** before any access to an address in the demapped page is made.*

#### TLBDEMAP      D[a] (RR)

31	28 27	20 19 18 17 16 15	12 11	8 7	0
-	00H	-	-	a	75H

Removes the Page Table Entry (PTE) containing the virtual address specified by D[a] from either TLB-A or TLB-B. If the PTE does not exist then NOP.

#### Status Flags

C	Not set by this instruction.
V	Not set by this instruction.
SV	Not set by this instruction.
AV	Not set by this instruction.
SAV	Not set by this instruction.

#### Examples

```
tlbdemap    d2
```

#### See Also

[TLBMAP](#), [TLBFLUSH.A](#), [TLBFLUSH.B](#), [TLBPROBE.A](#), [TLBPROBE.I](#)

**TLBFLUSH.A**  
TLB-A Flush  
**TLBFLUSH.B**  
TLB-B Flush

**Description**

*Note: The TLBFLUSH instruction can only be executed in Supervisor mode.*

The TLBFLUSH instructions are used to flush mappings from the MMU (Memory Management Unit). There are two variants of the TLBFLUSH instruction:

- TLBFLUSH.A flushes all the mappings from TLB-A.
- TLBFLUSH.B flushes all mappings from TLB-B.

*Note: The TLBFLUSH instruction should be followed by an **ISYNC** before any access is made to a PTE translated virtual address.*

**TLBFLUSH.A (RR)**

31	28 27	20 19 18 17 16 15	12 11	8 7	0
-	04H	- - -	-	-	75H

Flushes all Page Table Entry (PTE) mappings from TLB-A.

**TLBFLUSH.B (RR)**

31	28 27	20 19 18 17 16 15	12 11	8 7	0
-	05H	- - -	-	-	75H

Flushes all Page Table Entry (PTE) mappings from TLB-B.

**Status Flags**

C	Not set by this instruction.
V	Not set by this instruction.
SV	Not set by this instruction.
AV	Not set by this instruction.
SAV	Not set by this instruction.

**Examples**

`tlbflush.a`

See Also

[TLBMAP](#), [TLBDEMAP](#), [TLBPROBE.A](#), [TLBPROBE.I](#)

See Also Addendum for TriCore Arch Manual, Vol2, V1.3.8, pages 493/494

## TLBMAP

### TLB Map

#### Description

*Note: This instruction can only be executed in Supervisor mode.*

The TLBMAP instruction is used to install a mapping in the MMU (Memory Management Unit). The data register pair E[a] issued as a parameter. The even E[a] register contains the VPN for the translation while the odd E[a] 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. Bits E[a][9:0] and E[a][55:54] are reserved and should be written with 0's. Bits E[a][15:10] and E[a][5:0] are reserved when unused and therefore should be set to 0 when unused. For example, if the page size (PSZ) is set to 4K (01<sub>B</sub>) then bits E[a][11:10] of the VPN are unused and should be set to 0, similarly bits E[a][33:32] of the PPN are also unused and should be set to 0.

*Note: A TLBMAP instruction is to be followed by an **ISYNC** instruction before attempting to use the installed mapping.*

*Note: Bits E[a][9:0] and E[a][55:54] are reserved.*

#### TLBMAP

#### E[a] (RR)

31	28 27	20 19 18 17 16 15	12 11	8 7	0
-	40 <sub>H</sub>	-	-	-	a
					75 <sub>H</sub>

TTE.ASI = ASI;

TTE.VPN = E[a][31:10];

TTE.PPN = E[a][53:32];

TTE.PSZ = E[a][57:56];

TTE.C = E[a][58];

TTE.G = E[a][59];

TTE.RE = E[a][60];

TTE.WE = E[a][61];

TTE.XE = E[a][62];

TTE.V = E[a][63];

#### Status Flags

C	Not set by this instruction.
---	------------------------------



V	Not set by this instruction.
SV	Not set by this instruction.
AV	Not set by this instruction.
SAV	Not set by this instruction.

**Examples**

```
tlbmap    e2  
isync
```

**See Also**

[TLBMAP](#), [TLBDEMAP](#), [TLBFLUSH.A](#), [TLBFLUSH.B](#), [TLBPROBE.A](#), [TLBPROBE.I](#)

See Also Addendum for TriCore Arch Manual, Vol2, V1.3.8, pages 493/494

## TLBPROBE.A

### TLB Probe Address

#### Description

*Note: The TLBPROBE.A instruction can only be executed in Supervisor mode.*

TLBPROBE.A is used to probe the MMU (Memory Management Unit) for a virtual address. Data register D[a] contains the virtual address for the probe. The Address Space Identifier (ASI) for the probe is obtained from the ASI register. The instruction returns:

- 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).
- 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.

#### TLBPROBE.A      D[a] (RR)

31	28 27	20 19 18 17 16 15	12 11	8 7	0
-	08H	-	-	a	75H

if (TLB contains an entry which matches the ASI / VPN in D[a]) {

    index = matched TLB entry;

    TVA.ASI = TLB[index].ASI;

    TVA.VPN = TLB[index].VPN;

    TPA.PPN = TLB[index].PPN;

    TPA.PSZ = TLB[index].PSZ;

    TPA.C = TLB[index].C;

    TPA.G = TLB[index].G;

    TPA.RE = TLB[index].RE;

    TPA.WE = TLB[index].WE;

    TPA.XE = TLB[index].XE;

    TPA.V = TLB[index].V;

    TPX = index;

} else {

    TPA.V = 0;

}

### Status Flags

C	Not set by this instruction.
V	Not set by this instruction.
SV	Not set by this instruction.
AV	Not set by this instruction.
SAV	Not set by this instruction.

### Examples

```
tlbprobe.a    d2
```

### See Also

[TLBMAP](#), [TLBDEMAP](#), [TLBFLUSH.A](#), [TLBFLUSH.B](#), [TLBPROBE.A](#), [TLBPROBE.I](#)

## TLBPROBE.I

### TLB Probe Index

#### Description

*Note: The TLBPROBE.I instruction can only be executed in Supervisor mode.*

The TLBPROBE.I instruction takes a data register (D[a]) as a parameter and is used to probe the TLB at a given index. The D[a] register contains the index for the probe. Bits D[a][31:8] are reserved and should be set to 0's.

*Note: This instruction is intended for diagnostic use only. The index set for the TLBs is implementation specific, and there is no architecturally defined way to predict what TLB index value will be associated with a given address mapping.*

#### TLBPROBE.I      D[a] (RR)

31	28 27	20 19 18 17 16 15	12 11	8 7	0
-	09H	-	-	-	a      75H

if (D[a] is a valid TLB entry index) {

    index = D[a];

    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;

}

#### Status Flags

C	Not set by this instruction.
V	Not set by this instruction.
SV	Not set by this instruction.
AV	Not set by this instruction.
SAV	Not set by this instruction.

#### Examples

```
tlbprobe.i    d2
```

See Also

[TLBMAP](#), [TLBDEMAP](#), [TLBFLUSH.A](#), [TLBFLUSH.B](#), [TLBPROBE.A](#), [TLBPROBE.I](#)

See Also Addendum for TriCore Arch Manual, Vol2, V1.3.8, pages 493/494



See Also Addendum for TriCore Arch Manual, Vol2, V1.3.8, pages 493/494

## 4 Summary Tables of LS and IP Instructions

This chapter contains two tables, one of the LS instructions and one of the IP instructions.

**Table 4-1 Load Store Instructions**

ADD.A	ADDIH.A	ADDSC.A	ADDSC.AT	BISR
CACHEA.I	CACHEA.W	CACHEA.WI	CACHEI.W	CACHEI.WI
CALL	CALLA	CALLI	DEBUG	DISABLE
DSYNC	ENABLE	EQ.A	EQZ.A	GE.A
ISYNC	J	JA	JEQ.A	JI
JL	JLA	JLI	JNE.A	JNZ.A
JZ.A	LD.A	LD.B	LD.BU	LD.D
LD.DA	LD.H	LD.HU	LD.Q	LD.W
LDLCX	LDMST	LDUCX	LEA	LOOP
LOOPU	LT.A	MFCR	MOV.A	MOV.AA
MOV.D	MOVH.A	MTCR	NE.A	NEZ.A
NOP	RET	RFE	RFM	RSLCX
ST.A	ST.B	ST.D	ST.DA	ST.H
ST.Q	ST.T	ST.W	STLCX	STUCX
SUB.A	SVLCX	SWAP.W	SYSCALL	TLBDEMAP
TLBFLUSH.A	TLBFLUSH.B	TLBMAP	TLBPROBE.A	TLBPROBE.I
TRAPSV	TRAPV			

Summary Tables of LS and IP Instructions

**Table 4-2 IP Instructions**

ABS.B	ABS.H	ABS	ABSDIF.B	ABSDIF.H
ABSDIF	ABSDIFS.H	ABSDIFS	ABSS.H	ABSS
ADD.B	ADD.F	ADD.H	ADD	ADDC
ADDI	ADDIH	ADDS.H	ADDS.HU	ADDS.U
ADDS	ADDX	AND.AND.T	AND.ANDN.T	AND.EQ
AND.GE.U	AND.GE	AND.LT.U	AND.LT	AND.NE
AND.NOR.T	AND.OR.T	AND.T	AND	ANDN.T
ANDN	BMERGE	BSPLIT	CADD	CADDN
CLO.H	CLO	CLS.H	CLS	CLZ.H
CLZ	CMOV	CMOVN	CMP.F	CSUB
CSUBN	DEXTR	DIV.F	DVADJ	DVINIT.B
DVINIT.BU	DVINIT.H	DVINIT.HU	DVINIT.U	DVINIT
DVSTEP.U	DVSTEP	EQ.B	EQ.H	EQ.W
EQ	EQANY.B	EQANY.H	EXTR.U	EXTR
FTOI	FTOIZ	FTOQ31	FTOQ31Z	FTOU
FTOUZ	GE.U	GE	IMASK	INS.T
INSERT	INSN.T	ITOF	IXMAX.U	IXMAX
IXMIN.U	IXMIN	JEQ	JGE.U	JGE
JGEZ	JGTZ	JLEZ	JLT.U	JLT
JLTZ	JNE	JNED	JNEI	JNZ.T
JNZ	JZ.T	JZ	LT.B	LT.BU
LT.H	LT.HU	LT.U	LT.W	LT.WU
LT	MADD.F	MADD.H	MADD.Q	MADD.U
MADD	MADDM.H	MADDMS.H	MADDR.H	MADDR.Q
MADDRS.H	MADDRS.Q	MADDS.H	MADDS.Q	MADDS.U
MADDS	MADDSU.H	MADDSUM.H	MADDSUMS.H	MADDSUR.H
MADDSURS. H	MADDSUS.H	MAX.B	MAX.BU	MAX.H
MAX.HU	MAX.U	MAX	MIN.B	MIN.BU
MIN.H	MIN.HU	MIN.U	MIN	MOV.U
MOV	MOVH	MSUB.F	MSUB.H	MSUB.Q



Summary Tables of LS and IP Instructions

**Table 4-2 IP Instructions**

MSUB.U	MSUB	MSUBAD.H	MSUBADM.H	MSUBADMS.H
MSUBADR.H	MSUBADRS.H	MSUBADS.H	MSUBM.H	MSUBMS.H
MSUBR.H	MSUBR.Q	MSUBRS.H	MSUBRS.Q	MSUBS.H
MSUBS.Q	MSUBS.U	MSUBS	MUL.F	MUL.H
MUL.Q	MUL.U	MUL	MULM.H	MULMS.H
MULR.H	MULR.Q	MULS.U	MULS	NAND.T
NAND	NE	NOR.T	NOR	OR.AND.T
OR.ANDN.T	OR.EQ	OR.GE.U	OR.GE	OR.LT.U
OR.LT	OR.NE	OR.NOR.T	OR.OR.T	OR.T
OR	ORN.T	ORN	PACK	PARITY
Q31TOF	QSEED.F	RSTV	RSUB	RSUBS.U
RSUBS	SAT.B	SAT.BU	SAT.H	SAT.HU
SEL	SELN	SH.AND.T	SH.ANDN.T	SH.EQ
SH.GE.U	SH.GE	SH.H	SH.LT.U	SH.LT
SH.NAND.T	SH.NE	SH.NOR.T	SH.OR.T	SH.ORN.T
SH.XNOR.T	SH.XOR.T	SH	SHA.H	SHA
SHAS	SUB.B	SUB.F	SUB.H	SUB
SUBC	SUBS.H	SUBS.HU	SUBS.U	SUBS
SUBX	UNPACK	UPDFL	UTOF	XNOR.T
XNOR	XOR.EQ	XOR.GE.U	XOR.GE	XOR.LT.U
XOR.LT	XOR.NE	XOR.T	XOR	

See Also Addendum for TriCore Arch Manual, Vol2, V1.3.8, pages 493/494

List of Instructions by Shortname	Chapter - Page
ABS	3 - 22
ABS.B	3 - 23
ABS.H	3 - 23
ABSDIF	3 - 25
ABSDIF.B	3 - 27
ABSDIF.H	3 - 27
ABSDIFS	3 - 29
ABSDIFS.H	3 - 31
ABSS	3 - 32
ABSS.H	3 - 33
ADD	3 - 34
ADD.A	3 - 37
ADD.B	3 - 39
ADD.F	3 - 467
ADD.H	3 - 39
ADDC	3 - 41
ADDI	3 - 43
ADDIH	3 - 44
ADDIH.A	3 - 45
ADDS	3 - 46
ADDS.H	3 - 48
ADDS.HU	3 - 48
ADDS.U	3 - 50
ADDSC.A	3 - 52
ADDSC.AT	3 - 52
ADDX	3 - 54
AND	3 - 56
AND.AND.T	3 - 58
AND.ANDN.T	3 - 58
AND.EQ	3 - 60
AND.GE	3 - 61
AND.GE.U	3 - 61
AND.LT	3 - 63
AND.LT.U	3 - 63
AND.NE	3 - 65
AND.NOR.T	3 - 58
AND.OR.T	3 - 58
AND.T	3 - 66
ANDN	3 - 67

ANDN.T	3 - 68
BISR	3 - 69
BMERGE	3 - 71
BSPLIT	3 - 72
CACHEA.I	3 - 73
CACHEA.W	3 - 76
CACHEA.WI	3 - 79
CACHEI.W	3 - 82
CACHEI.WI	3 - 84
CADD	3 - 86
CADDN	3 - 88
CALL	3 - 90
CALLA	3 - 93
CALLI	3 - 95
CLO	3 - 97
CLO.H	3 - 98
CLS	3 - 99
CLS.H	3 - 100
CLZ	3 - 101
CLZ.H	3 - 102
CMOV (16-bit)	3 - 103
CMOVN (16-bit)	3 - 104
CMP.F	3 - 469
CSUB	3 - 105
CSUBN	3 - 106
DEBUG	3 - 107
DEXTR	3 - 108
DISABLE	3 - 109
DIV.F	3 - 471
DSYNC	3 - 110
DVADJ	3 - 111
DVINIT	3 - 113
DVINIT.B	3 - 113
DVINIT.BU	3 - 113
DVINIT.H	3 - 113
DVINIT.HU	3 - 113
DVINIT.U	3 - 113
DVSTEP	3 - 118
DVSTEP.U	3 - 118

See Also Addendum for TriCore Arch Manual, Vol2, V1.3.8, pages 493/494

ENABLE	3 - 121
EQ	3 - 122
EQ.A	3 - 124
EQ.B	3 - 125
EQ.H	3 - 125
EQ.W	3 - 125
EQANY.B	3 - 127
EQANY.H	3 - 127
EQZ.A	3 - 129
EXTR	3 - 130
EXTR.U	3 - 130
FTOI	3 - 473
FTOIZ	3 - 474
FTOQ31	3 - 475
FTOQ31Z	3 - 477
FTOU	3 - 479
FTOUZ	3 - 480
GE	3 - 132
GE.A	3 - 134
GE.U	3 - 132
IMASK	3 - 135
INS.T	3 - 137
INSERT	3 - 138
INSN.T	3 - 137
ISYNC	3 - 141
ITOF	3 - 481
IXMAX	3 - 142
IXMAX.U	3 - 142
IXMIN	3 - 144
IXMIN.U	3 - 144
J	3 - 146
JA	3 - 147
JEQ	3 - 148
JEQ.A	3 - 150
JGE	3 - 151
JGE.U	3 - 151
JGEZ (16-bit)	3 - 153
JGTZ (16-bit)	3 - 154
JL	3 - 155

JL	3 - 156
JLA	3 - 157
JLEZ (16-bit)	3 - 158
JLI	3 - 159
JLT	3 - 160
JLT.U	3 - 160
JLTZ (16-bit)	3 - 162
JNE	3 - 163
JNE.A	3 - 165
JNED	3 - 166
JNEI	3 - 168
JNZ (16-bit)	3 - 170
JNZ.A	3 - 171
JNZ.T	3 - 172
JZ (16-bit)	3 - 173
JZ.A	3 - 174
JZ.T	3 - 175
LD.A	3 - 176
LD.B	3 - 180
LD.BU	3 - 180
LD.D	3 - 185
LD.DA	3 - 188
LD.H	3 - 191
LD.HU	3 - 194
LD.Q	3 - 196
LD.W	3 - 198
LDLCX	3 - 202
LDMST	3 - 203
LDUCX	3 - 206
LEA	3 - 207
LOOP	3 - 209
LOOPU	3 - 211
LT	3 - 212
LT.A	3 - 215
LT.B	3 - 216
LT.BU	3 - 216
LT.H	3 - 218
LT.HU	3 - 218
LT.U	3 - 212

LT.W	3 - 220
LT.WU	3 - 220
MADD	3 - 221
MADD.F	3 - 482
MADD.H	3 - 224
MADD.Q	3 - 228
MADD.U	3 - 235
MADDM.H	3 - 238
MADDMS.H	3 - 238
MADDR.H	3 - 242
MADDR.Q	3 - 247
MADDRS.H	3 - 242
MADDRS.Q	3 - 247
MADDS	3 - 221
MADDS.H	3 - 224
MADDS.Q	3 - 228
MADDS.U	3 - 235
MADDSU.H	3 - 249
MADDSUM.H	3 - 253
MADDSUMS.H	3 - 253
MADDSUR.H	3 - 257
MADDSURS.H	3 - 257
MADDSUS.H	3 - 249
MAX	3 - 262
MAX.B	3 - 264
MAX.BU	3 - 264
MAX.H	3 - 266
MAX.HU	3 - 266
MAX.U	3 - 262
MFCR	3 - 267
MIN	3 - 268
MIN.B	3 - 270
MIN.BU	3 - 270
MIN.H	3 - 272
MIN.HU	3 - 272
MIN.U	3 - 268
MOV	3 - 273
MOV.A	3 - 275
MOV.AA	3 - 277

See Also Addendum for TriCore Arch Manual, Vol2, V1.3.8, pages 493/494

MOV.D	3 - 278
MOV.U	3 - 279
MOVH	3 - 280
MOVH.A	3 - 281
MSUB	3 - 282
MSUB.F	3 - 484
MSUB.H	3 - 285
MSUB.Q	3 - 289
MSUB.U	3 - 296
MSUBAD.H	3 - 299
MSUBADM.H	3 - 303
MSUBADMS.H	3 - 303
MSUBADR.H	3 - 307
MSUBADRS.H	3 - 307
MSUBADS.H	3 - 299
MSUBM.H	3 - 312
MSUBMS.H	3 - 312
MSUBR.H	3 - 316
MSUBR.Q	3 - 321
MSUBRS.H	3 - 316
MSUBRS.Q	3 - 321
MSUBS	3 - 282
MSUBS.H	3 - 285
MSUBS.Q	3 - 289
MSUBS.U	3 - 296
MTCR	3 - 323
MUL	3 - 324
MUL.F	3 - 486
MUL.H	3 - 327
MUL.Q	3 - 330
MUL.U	3 - 333
MULM.H	3 - 335
MULR.H	3 - 338
MULR.Q	3 - 341
MULS	3 - 324
MULS.U	3 - 333
NAND	3 - 343
NAND.T	3 - 344
NE	3 - 345



See Also Addendum for TriCore Arch Manual, Vol2, V1.3.8, pages 493/494

NE.A	3 - 346
NEZ.A	3 - 347
NOP	3 - 348
NOR	3 - 349
NOR.T	3 - 350
NOT (16-bit)	3 - 351
OR	3 - 352
OR.AND.T	3 - 354
OR.ANDN.T	3 - 354
OR.EQ	3 - 356
OR.GE	3 - 357
OR.GE.U	3 - 357
OR.LT	3 - 359
OR.LT.U	3 - 359
OR.NE	3 - 361
OR.NOR.T	3 - 354
OR.OR.T	3 - 354
OR.T	3 - 362
ORN	3 - 363
ORN.T	3 - 364
PACK	3 - 365
PARITY	3 - 368
Q31TOF	3 - 488
QSEED.F	3 - 489
RET	3 - 369
RFE	3 - 371
RFM	3 - 373
RSLCX	3 - 375
RSTV	3 - 376
RSUB	3 - 377
RSUBS	3 - 379
RSUBS.U	3 - 379
SAT.B	3 - 381
SAT.BU	3 - 383
SAT.H	3 - 384
SAT.HU	3 - 386
SEL	3 - 387
SELN	3 - 388
SH	3 - 389

See Also Addendum for TriCore Arch Manual, Vol2, V1.3.8, pages 493/494

SH.AND.T	3 - 399
SH.ANDN.T	3 - 399
SH.EQ	3 - 391
SH.GE	3 - 392
SH.GE.U	3 - 392
SH.H	3 - 394
SH.LT	3 - 396
SH.LT.U	3 - 396
SH.NAND.T	3 - 399
SH.NE	3 - 398
SH.NOR.T	3 - 399
SH.OR.T	3 - 399
SH.ORN.T	3 - 399
SH.XNOR.T	3 - 399
SH.XOR.T	3 - 399
SHA	3 - 402
SHA.H	3 - 405
SHAS	3 - 407
ST.A	3 - 409
ST.B	3 - 413
ST.D	3 - 416
ST.DA	3 - 419
ST.H	3 - 422
ST.Q	3 - 425
ST.T	3 - 427
ST.W	3 - 428
STLCX	3 - 432
STUCX	3 - 433
SUB	3 - 434
SUB.A	3 - 436
SUB.B	3 - 437
SUB.F	3 - 491
SUB.H	3 - 437
SUBC	3 - 439
SUBS	3 - 440
SUBS.H	3 - 442
SUBS.HU	3 - 442
SUBS.U	3 - 440
SUBX	3 - 444

See Also Addendum for TriCore Arch Manual, Vol2, V1.3.8, pages 493/494

SVLCX .....	3 - 445
SWAP.W .....	3 - 447
SYSCALL .....	3 - 450
TLBDEMAP .....	3 - 497
TLBFLUSH.A .....	3 - 498
TLBFLUSH.B .....	3 - 498
TLBMAP .....	3 - 500
TLBPROBE.A .....	3 - 502
TLBPROBE.I .....	3 - 504
TRAPSV .....	3 - 451
TRAPV .....	3 - 452
UNPACK .....	3 - 453
UPDFL .....	3 - 493
UTOF .....	3 - 495
XNOR .....	3 - 455
XNOR.T .....	3 - 456
XOR .....	3 - 457
XOR.EQ .....	3 - 459
XOR.GE .....	3 - 460
XOR.GE.U .....	3 - 460
XOR.LT .....	3 - 462
XOR.LT.U .....	3 - 462
XOR.NE .....	3 - 464
XOR.T .....	3 - 465

See Also Addendum for TriCore Arch Manual, Vol2, V1.3.8, pages 493/494

List of Instructions by Longname	Chapter - Page
Absolute Value	3 - 22
Absolute Value of Difference	3 - 25
Absolute Value of Difference Packed Byte	3 - 27
Absolute Value of Difference Packed Half-word	3 - 27
Absolute Value of Difference Packed Half-word with Saturation	3 - 31
Absolute Value of Difference with Saturation	3 - 29
Absolute Value Packed Byte	3 - 23
Absolute Value Packed Half-word	3 - 23
Absolute Value Packed Half-word with Saturation	3 - 33
Absolute Value with Saturation	3 - 32
Accumulating Bit Logical AND-AND	3 - 58
Accumulating Bit Logical AND-AND-Not	3 - 58
Accumulating Bit Logical AND-NOR	3 - 58
Accumulating Bit Logical AND-OR	3 - 58
Accumulating Bit Logical OR-AND	3 - 354
Accumulating Bit Logical OR-AND-Not	3 - 354
Accumulating Bit Logical OR-NOR	3 - 354
Accumulating Bit Logical OR-OR	3 - 354
Accumulating Shift-AND	3 - 399
Accumulating Shift-AND-Not	3 - 399
Accumulating Shift-NAND	3 - 399
Accumulating Shift-NOR	3 - 399
Accumulating Shift-OR	3 - 399
Accumulating Shift-OR-Not	3 - 399
Accumulating Shift-XNOR	3 - 399
Accumulating Shift-XOR	3 - 399
Add	3 - 34
Add Address	3 - 37
Add Bit-Scaled Index to Address	3 - 52
Add Extended	3 - 54
Add Float	3 - 467
Add Immediate	3 - 43
Add Immediate High	3 - 44
Add Immediate High to Address	3 - 45
Add Packed Byte	3 - 39
Add Packed Half-word	3 - 39
Add Scaled Index to Address	3 - 52
Add Signed Packed Half-word with Saturation	3 - 48
Add Signed with Saturation	3 - 46

Add Unsigned Packed Half-word with Saturation . . . . .	3 - 48
Add Unsigned with Saturation . . . . .	3 - 50
Add with Carry . . . . .	3 - 41
Arithmetic Shift . . . . .	3 - 402
Arithmetic Shift Packed Half-words . . . . .	3 - 405
Arithmetic Shift with Saturation . . . . .	3 - 407
Begin Interrupt Service Routine . . . . .	3 - 69
Bit Logical AND . . . . .	3 - 66
Bit Logical AND-Not . . . . .	3 - 68
Bit Logical NAND . . . . .	3 - 344
Bit Logical NOR . . . . .	3 - 350
Bit Logical OR . . . . .	3 - 362
Bit Logical OR-Not . . . . .	3 - 364
Bit Logical XNOR . . . . .	3 - 456
Bit Logical XOR . . . . .	3 - 465
Bit Merge . . . . .	3 - 71
Bit Split . . . . .	3 - 72
Bitwise AND . . . . .	3 - 56
Bitwise AND-Not . . . . .	3 - 67
Bitwise Complement NOT (16-bit) . . . . .	3 - 351
Bitwise NAND . . . . .	3 - 343
Bitwise NOR . . . . .	3 - 349
Bitwise OR . . . . .	3 - 352
Bitwise OR-Not . . . . .	3 - 363
Bitwise XNOR . . . . .	3 - 455
Bitwise XOR . . . . .	3 - 457
Cache Address, Invalidate . . . . .	3 - 73
Cache Address, Writeback . . . . .	3 - 76
Cache Address, Writeback and Invalidate . . . . .	3 - 79
Cache Index, Writeback . . . . .	3 - 82
Cache Index, Writeback, Invalidate . . . . .	3 - 84
Call . . . . .	3 - 90
Call Absolute . . . . .	3 - 93
Call Indirect . . . . .	3 - 95
Compare Float . . . . .	3 - 469
Conditional Add . . . . .	3 - 86
Conditional Add-Not . . . . .	3 - 88
Conditional Move (16-bit) . . . . .	3 - 103
Conditional Move-Not (16-bit) . . . . .	3 - 104

Conditional Subtract	3 - 105
Conditional Subtract-Not	3 - 106
Count Leading Ones	3 - 97
Count Leading Ones in Packed Half-words	3 - 98
Count Leading Signs	3 - 99
Count Leading Signs in Packed Half-words	3 - 100
Count Leading Zeros	3 - 101
Count Leading Zeros in Packed Half-words	3 - 102
Debug	3 - 107
Disable Interrupts	3 - 109
Divide Float	3 - 471
Divide-Adjust	3 - 111
Divide-Initialization Byte	3 - 113
Divide-Initialization Byte Unsigned	3 - 113
Divide-Initialization Half-word	3 - 113
Divide-Initialization Half-word Unsigned	3 - 113
Divide-Initialization Word	3 - 113
Divide-Initialization Word Unsigned	3 - 113
Divide-Step	3 - 118
Divide-Step Unsigned	3 - 118
Enable Interrupts	3 - 121
Equal	3 - 122
Equal Accumulating	3 - 356
Equal Accumulating	3 - 459
Equal Accumulating	3 - 60
Equal Any Byte	3 - 127
Equal Any Half-word	3 - 127
Equal Packed Byte	3 - 125
Equal Packed Half-word	3 - 125
Equal Packed Word	3 - 125
Equal to Address	3 - 124
Equal Zero Address	3 - 129
Extract Bit Field	3 - 130
Extract Bit Field Unsigned	3 - 130
Extract from Double Register	3 - 108
Find Maximum Index	3 - 142
Find Maximum Index (unsigned)	3 - 142
Find Minimum Index	3 - 144
Find Minimum Index (unsigned)	3 - 144

Float to Fraction .....	3 - 475
Float to Fraction, Round towards Zero .....	3 - 477
Float to Integer .....	3 - 473
Float to Integer, Round towards Zero .....	3 - 474
Float to Unsigned .....	3 - 479
Float to Unsigned, Round towards Zero .....	3 - 480
Fraction to Floating-point .....	3 - 488
Greater Than or Equal .....	3 - 132
Greater Than or Equal Accumulating .....	3 - 357
Greater Than or Equal Accumulating .....	3 - 460
Greater Than or Equal Accumulating .....	3 - 61
Greater Than or Equal Accumulating Unsigned .....	3 - 357
Greater Than or Equal Accumulating Unsigned .....	3 - 460
Greater Than or Equal Accumulating Unsigned .....	3 - 61
Greater Than or Equal Address .....	3 - 134
Greater Than or Equal Unsigned .....	3 - 132
Insert Bit .....	3 - 137
Insert Bit Field .....	3 - 138
Insert Bit-Not .....	3 - 137
Insert Mask .....	3 - 135
Integer to Float .....	3 - 481
Inverse Square Root Seed .....	3 - 489
Jump and Link .....	3 - 156
Jump and Link Absolute .....	3 - 157
Jump and Link Indirect .....	3 - 159
Jump if Equal Address .....	3 - 150
Jump if Equal .....	3 - 148
Jump if Greater Than or Equal .....	3 - 151
Jump if Greater Than or Equal to Zero (16-bit) .....	3 - 153
Jump if Greater Than or Equal Unsigned .....	3 - 151
Jump if Greater Than Zero (16-bit) .....	3 - 154
Jump if Less Than .....	3 - 160
Jump if Less Than or Equal to Zero (16-bit) .....	3 - 158
Jump if Less Than Unsigned .....	3 - 160
Jump if Less Than Zero (16-bit) .....	3 - 162
Jump if Not Equal .....	3 - 163
Jump if Not Equal Address .....	3 - 165
Jump if Not Equal and Decrement .....	3 - 166
Jump if Not Equal and Increment .....	3 - 168



Jump if Not Equal to Zero (16-bit) . . . . .	3 - 170
Jump if Not Equal to Zero Address . . . . .	3 - 171
Jump if Not Equal to Zero Bit . . . . .	3 - 172
Jump if Zero (16-bit) . . . . .	3 - 173
Jump if Zero Address . . . . .	3 - 174
Jump if Zero Bit . . . . .	3 - 175
Jump Indirect . . . . .	3 - 155
Jump Unconditional . . . . .	3 - 146
Jump Unconditional Absolute . . . . .	3 - 147
Less Than . . . . .	3 - 212
Less Than Accumulating . . . . .	3 - 359
Less Than Accumulating . . . . .	3 - 462
Less Than Accumulating . . . . .	3 - 63
Less Than Accumulating Unsigned . . . . .	3 - 359
Less Than Accumulating Unsigned . . . . .	3 - 462
Less Than Accumulating Unsigned . . . . .	3 - 63
Less Than Address . . . . .	3 - 215
Less Than Packed Byte . . . . .	3 - 216
Less Than Packed Byte Unsigned . . . . .	3 - 216
Less Than Packed Half-word . . . . .	3 - 218
Less Than Packed Half-word Unsigned . . . . .	3 - 218
Less Than Packed Word . . . . .	3 - 220
Less Than Packed Word Unsigned . . . . .	3 - 220
Less Than Unsigned . . . . .	3 - 212
Load Byte . . . . .	3 - 180
Load Byte Unsigned . . . . .	3 - 180
Load Double-word . . . . .	3 - 185
Load Double-word to Address Register . . . . .	3 - 188
Load Effective Address . . . . .	3 - 207
Load Half-word . . . . .	3 - 191
Load Half-word Signed Fraction . . . . .	3 - 196
Load Half-word Unsigned . . . . .	3 - 194
Load Lower Context . . . . .	3 - 202
Load Upper Context . . . . .	3 - 206
Load Word . . . . .	3 - 198
Load Word to Address Register . . . . .	3 - 176
Load-Modify-Store . . . . .	3 - 203
Loop . . . . .	3 - 209
Loop Unconditional . . . . .	3 - 211

Maximum Value	3 - 262
Maximum Value Packed Byte	3 - 264
Maximum Value Packed Byte Unsigned	3 - 264
Maximum Value Packed Half-word	3 - 266
Maximum Value Packed Half-word Unsigned	3 - 266
Maximum Value Unsigned	3 - 262
Minimum Value	3 - 268
Minimum Value Packed Byte	3 - 270
Minimum Value Packed Byte Unsigned	3 - 270
Minimum Value Packed Half-word	3 - 272
Minimum Value Packed Half-word Unsigned	3 - 272
Minimum Value Unsigned	3 - 268
Move	3 - 273
Move Address from Address Register	3 - 277
Move Address to Data Register	3 - 278
Move From Core Register	3 - 267
Move High	3 - 280
Move High to Address	3 - 281
Move To Core Register	3 - 323
Move Unsigned	3 - 279
Move Value to Address Register	3 - 275
Multiply	3 - 324
Multiply Add Float	3 - 482
Multiply Float	3 - 486
Multiply Q Format	3 - 330
Multiply Q Format with Rounding	3 - 341
Multiply Subtract Float	3 - 484
Multiply Unsigned	3 - 333
Multiply Unsigned, Saturated	3 - 333
Multiply, Saturated	3 - 324
Multiply-Add	3 - 221
Multiply-Add Q Format	3 - 228
Multiply-Add Q Format with Rounding	3 - 247
Multiply-Add Q Format with Rounding, Saturated	3 - 247
Multiply-Add Q Format, Saturated	3 - 228
Multiply-Add Unsigned	3 - 235
Multiply-Add Unsigned, Saturated	3 - 235
Multiply-Add, Saturated	3 - 221
Multiply-Subtract	3 - 282

Multiply-Subtract Q Format	3 - 289
Multiply-Subtract Q Format with Rounding	3 - 321
Multiply-Subtract Q Format with Rounding, Saturated	3 - 321
Multiply-Subtract Q Format, Saturated	3 - 289
Multiply-Subtract Unsigned	3 - 296
Multiply-Subtract Unsigned, Saturated	3 - 296
Multiply-Subtract, Saturated	3 - 282
No Operation	3 - 348
Not Equal	3 - 345
Not Equal Accumulating	3 - 361
Not Equal Accumulating	3 - 464
Not Equal Accumulating	3 - 65
Not Equal Address	3 - 346
Not Equal Zero Address	3 - 347
Pack	3 - 365
Packed Multiply Q Format	3 - 327
Packed Multiply Q Format with Rounding	3 - 338
Packed Multiply Q Format-Multi-precision	3 - 335
Packed Multiply-Add Q Format	3 - 224
Packed Multiply-Add Q Format Multi-precision	3 - 238
Packed Multiply-Add Q Format Multi-precision, Saturated	3 - 238
Packed Multiply-Add Q Format with Rounding	3 - 242
Packed Multiply-Add Q Format with Rounding, Saturated	3 - 242
Packed Multiply-Add Q Format, Saturated	3 - 224
Packed Multiply-Add/Subtract Q Format	3 - 249
Packed Multiply-Add/Subtract Q Format Multi-precision	3 - 253
Packed Multiply-Add/Subtract Q Format Multi-precision Saturated	3 - 253
Packed Multiply-Add/Subtract Q Format Saturated	3 - 249
Packed Multiply-Add/Subtract Q Format with Rounding	3 - 257
Packed Multiply-Add/Subtract Q Format with Rounding Saturated	3 - 257
Packed Multiply-Subtract Q Format	3 - 285
Packed Multiply-Subtract Q Format with Rounding	3 - 316
Packed Multiply-Subtract Q Format with Rounding, Saturated	3 - 316
Packed Multiply-Subtract Q Format, Saturated	3 - 285
Packed Multiply-Subtract Q Format-Multi-precision	3 - 312
Packed Multiply-Subtract Q Format-Multi-precision, Saturated	3 - 312
Packed Multiply-Subtract/Add Q Format	3 - 299
Packed Multiply-Subtract/Add Q Format with Rounding	3 - 307
Packed Multiply-Subtract/Add Q Format with Rounding, Saturated	3 - 307

Packed Multiply-Subtract/Add Q Format, Saturated	3 - 299
Packed Multiply-Subtract/Add Q Format-Multi-precision	3 - 303
Packed Multiply-Subtract/Add Q Format-Multi-precision, Saturated	3 - 303
Parity	3 - 368
Reset Overflow Bits	3 - 376
Restore Lower Context	3 - 375
Return from Call	3 - 369
Return From Exception	3 - 371
Return From Monitor	3 - 373
Reverse-Subtract	3 - 377
Reverse-Subtract Unsigned with Saturation	3 - 379
Reverse-Subtract with Saturation	3 - 379
Saturate Byte	3 - 381
Saturate Byte Unsigned	3 - 383
Saturate Half-word	3 - 384
Saturate Half-word Unsigned	3 - 386
Save Lower Context	3 - 445
Select	3 - 387
Select-Not	3 - 388
Shift	3 - 389
Shift Equal	3 - 391
Shift Greater Than or Equal	3 - 392
Shift Greater Than or Equal Unsigned	3 - 392
Shift Less Than	3 - 396
Shift Less Than Unsigned	3 - 396
Shift Not Equal	3 - 398
Shift Packed Half-words	3 - 394
Store Bit	3 - 427
Store Byte	3 - 413
Store Double-word	3 - 416
Store Double-word from Address Registers	3 - 419
Store Half-word	3 - 422
Store Half-word Signed Fraction	3 - 425
Store Lower Context	3 - 432
Store Upper Context	3 - 433
Store Word	3 - 428
Store Word from Address Register	3 - 409
Subtract	3 - 434
Subtract Address	3 - 436

Subtract Extended . . . . .	3 - 444
Subtract Float . . . . .	3 - 491
Subtract Packed Byte . . . . .	3 - 437
Subtract Packed Half-word . . . . .	3 - 437
Subtract Packed Half-word Unsigned with Saturation . . . . .	3 - 442
Subtract Packed Half-word with Saturation . . . . .	3 - 442
Subtract Signed with Saturation . . . . .	3 - 440
Subtract Unsigned with Saturation . . . . .	3 - 440
Subtract With Carry . . . . .	3 - 439
Swap with Data Register . . . . .	3 - 447
Synchronize Data . . . . .	3 - 110
Synchronize Instructions . . . . .	3 - 141
System Call . . . . .	3 - 450
TLB Map . . . . .	3 - 500
TLB Probe Address . . . . .	3 - 502
TLB Probe Index . . . . .	3 - 504
TLB-A Flush . . . . .	3 - 498
TLB-B Flush . . . . .	3 - 498
Translation Lookaside Buffer Demap . . . . .	3 - 497
Trap on Overflow . . . . .	3 - 452
Trap on Sticky Overflow . . . . .	3 - 451
Unpack Floating Point . . . . .	3 - 453
Unsigned to Floating-point . . . . .	3 - 495
Update Flags . . . . .	3 - 493

See Also Addendum for TriCore Arch Manual, Vol2, V1.3.8, pages 493/494

See Also Addendum for TriCore Arch Manual, Vol2, V1.3.8, pages 493/494

See Also Addendum for TriCore Arch Manual, Vol2, V1.3.8, pages 493/494

[www.infineon.com](http://www.infineon.com)