TriCore™ TC1.8 architecture manual volume 2

32-bit microcontroller

Instruction Set
Volume 2 (of 2)

About this document

Scope and purpose

The TriCore™ architecture manual describes the core architecture and instruction set for Infineon Technologies TriCore™ microcontroller architecture. 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 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 product 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 document is specific to the version(s) identified on the cover page
- Information specific to a particular version of the architecture only, will be labeled as such

Additional documentation

For the latest documentation and additional TriCore™ information, please visit the TriCore™ home page at:

http://www.infineon.com/TriCore

The following additional documents are also available for download from the TriCore™ architecture and core section:

TriCore™ DSP optimization guide

TriCore™ EABI (Embedded ABI) user's manual

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 start-up 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/bits 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
  - Quad-word = 128-bit quantity

In tables where register bit-fields are defined, the conventions shown below 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, must be written with 0

***Note**: In register layout tables, a ‘Reserved Field’ is indicated with ‘RES’ in the field column and ‘-’ in the type column.*

Table of contents

Table of contents

Table of contents

Table of contents

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
- Addition and subtraction
- Multiply and multiply-add
- Division
- Absolute value, absolute difference
- Minimum, maximum, saturate
- Conditional arithmetic instructions
- Logical
- Count leading zeros, ones and signs
- Shift
- Bit-field extract and insert

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 or extended 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-bit or 64-bit) or MOV.U (which zero-extends to 32-bit).

A 4-bit constant is created using MOV (16-bit) (which sign-extends the value to 64-bit).

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 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)
- MUL.U (Multiply unsigned)
- 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, …15), or an extended data register E[n] (where n = 0, 2, …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 with single instructions (DIV, DIV.U). These instructions compute both quotient and remainder.

In addition division is supported with a sequence of instructions (see below DVINIT/DVSTEP/DVADJ). The sequence of instructions can be used to keep the interrupt latency low, and also in when larger dividends are required. In these cases 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.

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 Minimum, maximum, 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

Conditional arithmetic instructions are:

- 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.

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 zeros, 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 the figure, 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:

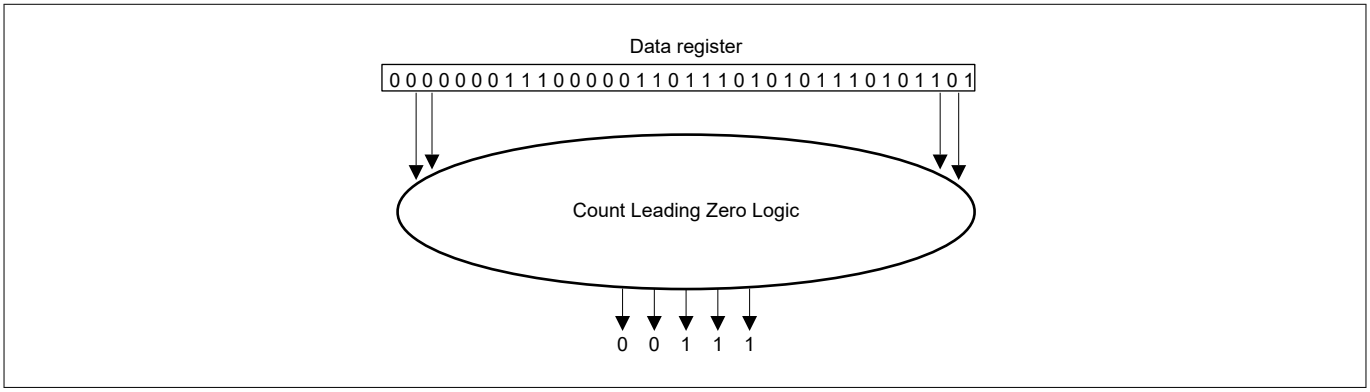


Figure 1 Operation of the CLZ instruction

1.1.10 Shift

The shift instructions support multi-bit shifts.

The shift amount is specified by a signed integer (n), which may be the contents of a register or a sign-extended constant in the instruction.

If n >= 0, the data is shifted left by n[4:0]; otherwise, the data is shifted right by (-n)[4:0].

The (logical) shift instruction SH, shifts in 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)

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.

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).

EXTR.U zero-fills the most significant (32-width) bits of the result.

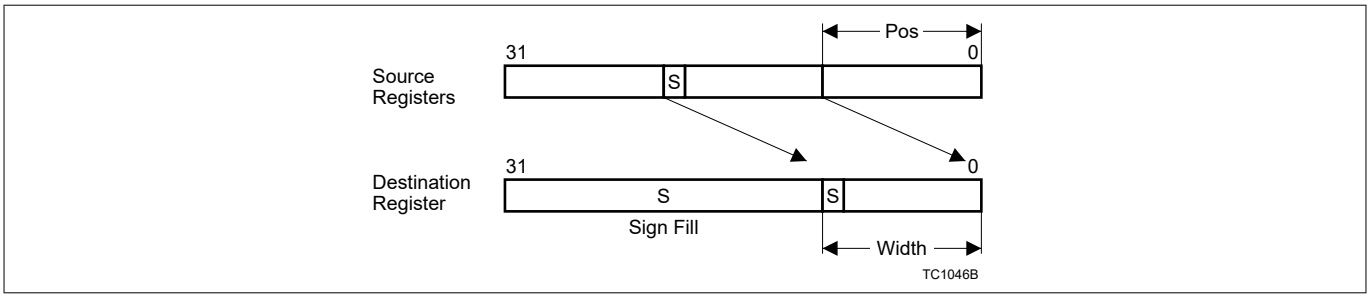


Figure 2 EXTR operation

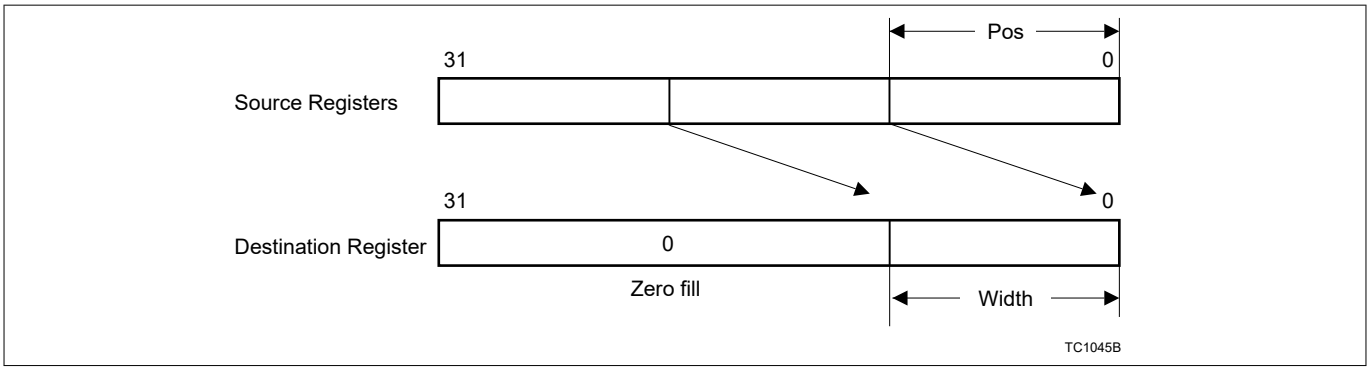


Figure 3 EXTR.U operation

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.

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).

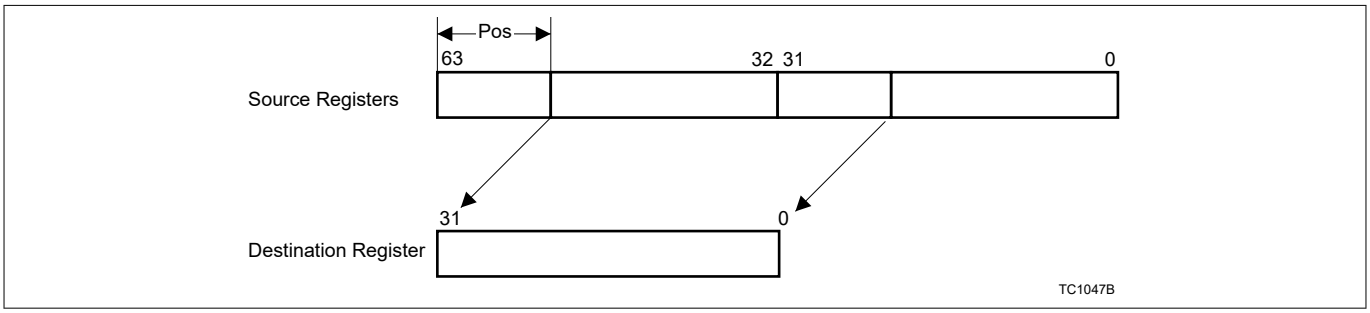


Figure 4 DEXTR operation

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-width) 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.

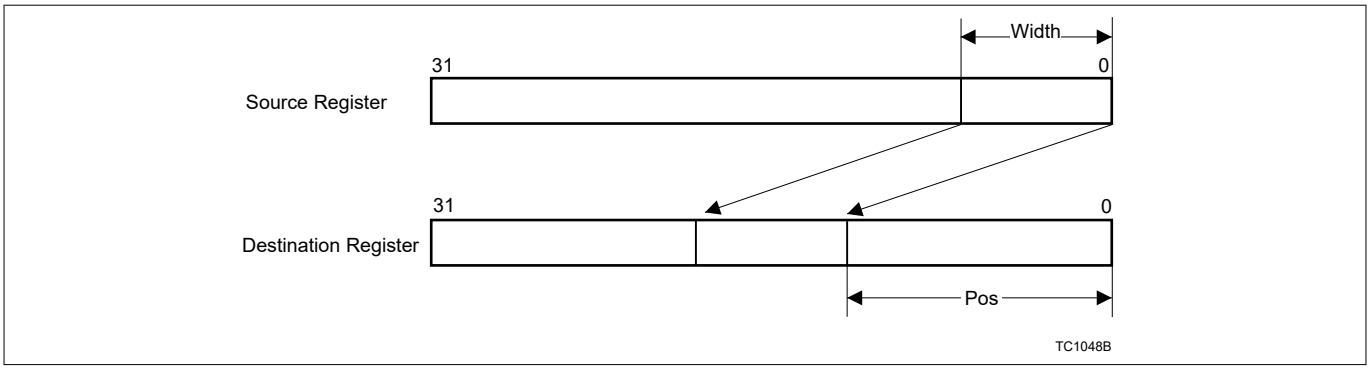


Figure 5 INSERT operation

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.

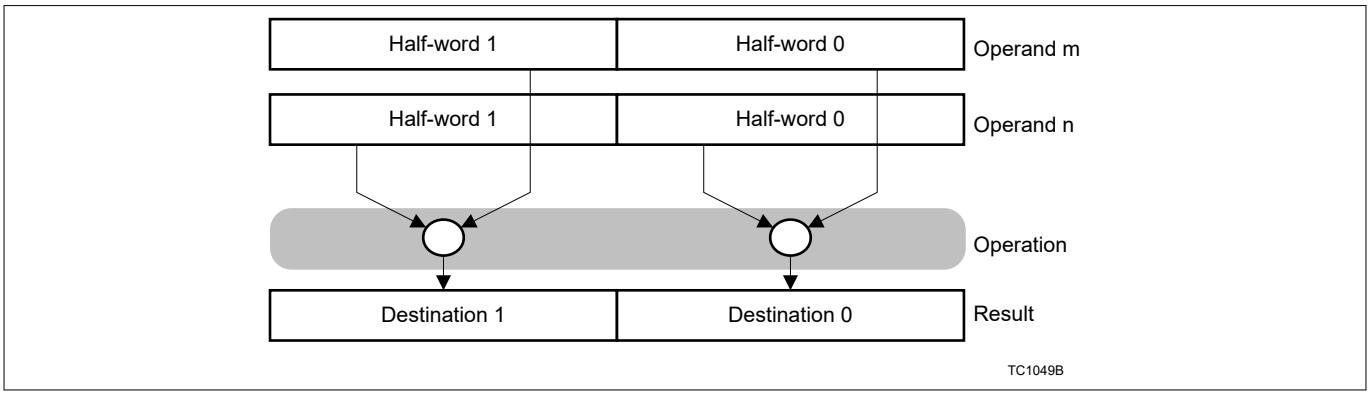


Figure 6 Packed half-word data format

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.

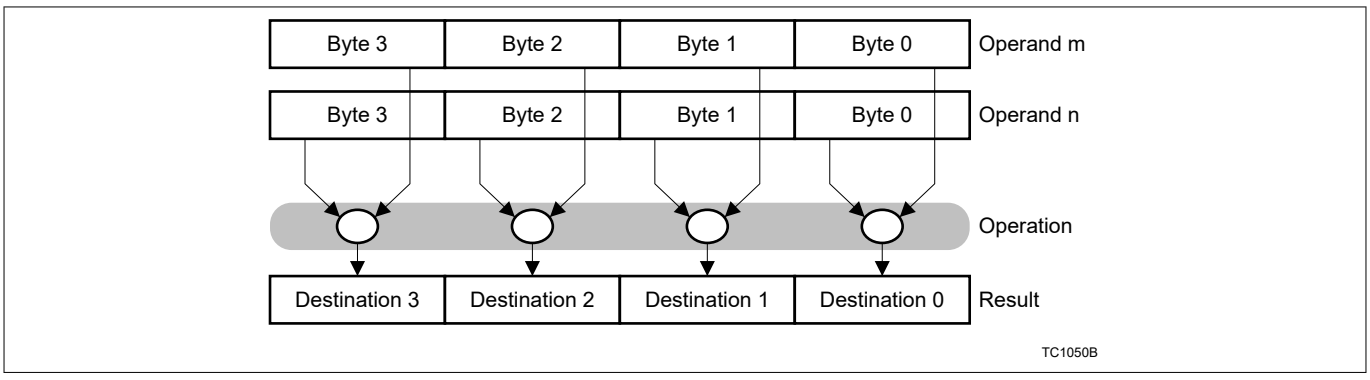


Figure 7 Packed byte data format

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.

Compare instructions are described in Compare instructions.

1.3 Program status word 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**: The PSW can also be read as a register pair {PSW, PCXI} using the move from core register pair (MFDCR) instruction and can be written using the move to core register pair (MTDCR) instruction.*

***Note**: MTCR and MTDCR are 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**: 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 (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).

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
- and so on

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 (16 by 16-bit multiply)

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

```
8000H * 8000H = 4000 0000H
```

This is correctly interpreted in Q format as:

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

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

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

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

```
8000H * 8000H = 7FFF FFFFH
```

This is correctly interpreted in Q format as:

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

This operation is completely transparent to the user and does not set the overflow flags. It applies only to 16-bit by 16-bit multiplies and does not apply to 16 by 32-bit or 32 by 32-bit multiplies.

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-bit) 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-bit 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 (that means 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

The sticky advance overflow (SAV) bit is set whenever an overflow ‘almost’ occurred. It can be used in block scaling of intermediate results, for example 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]). This gives 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 in 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

The following figure illustrates the operation of the LT (Less Than) compare instruction:

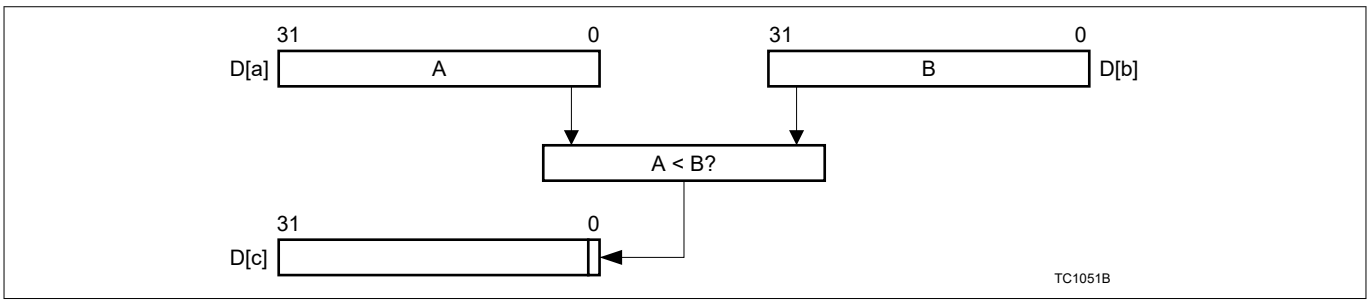


Figure 8 LT (Less Than) comparison

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.

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.

The figure below illustrates the combination of the LT instruction with a Boolean operation.

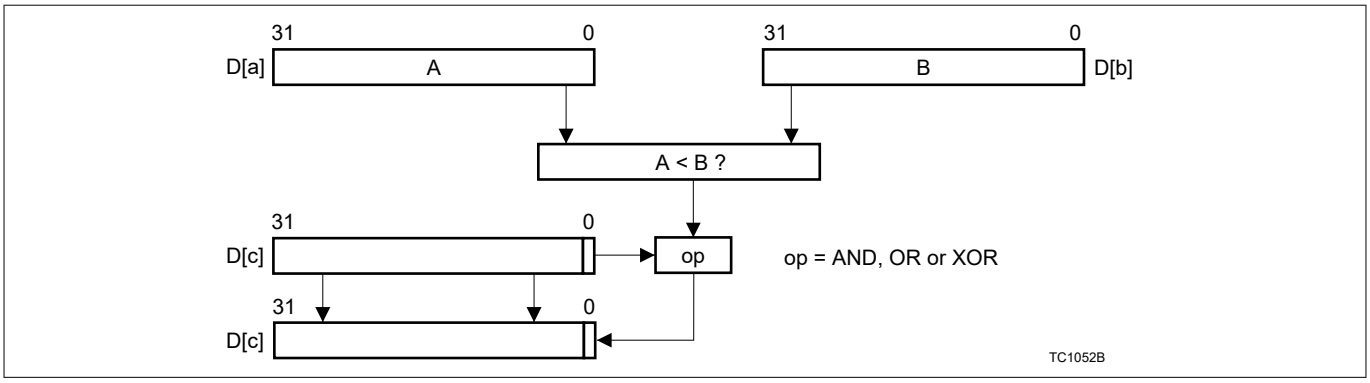


Figure 9 Combining LT comparison with Boolean operation

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

```
d5 = (d1 < d2) OR (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.

The figure below illustrates the operation of the SH.LT (Shift Less Than) instruction.

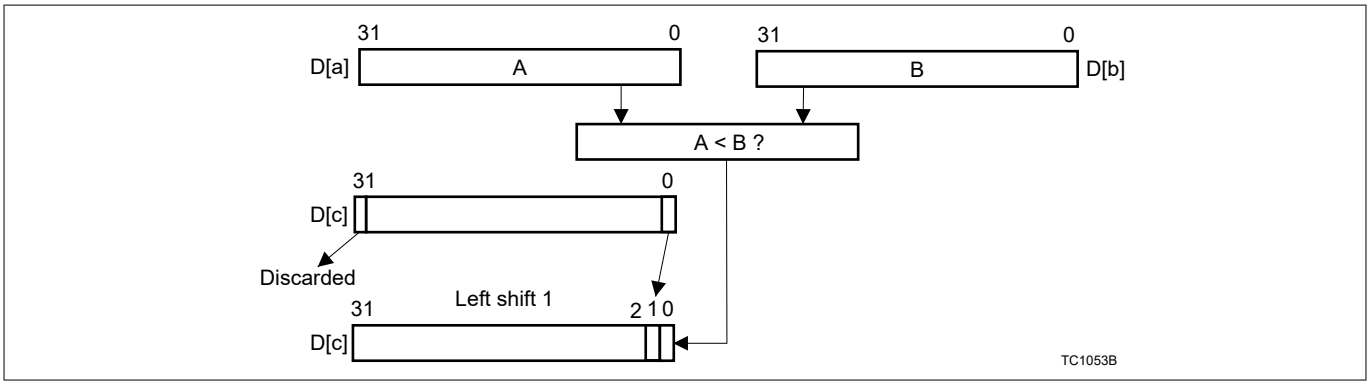


Figure 10 SH.LT instruction

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_H for a match, or 00_H 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:$FFFF_H$ for a match

and 0000_H 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 $FFFFFFFF_H$ for match and 00000000_H 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. The figure below shows an example of the EQ.B instruction.

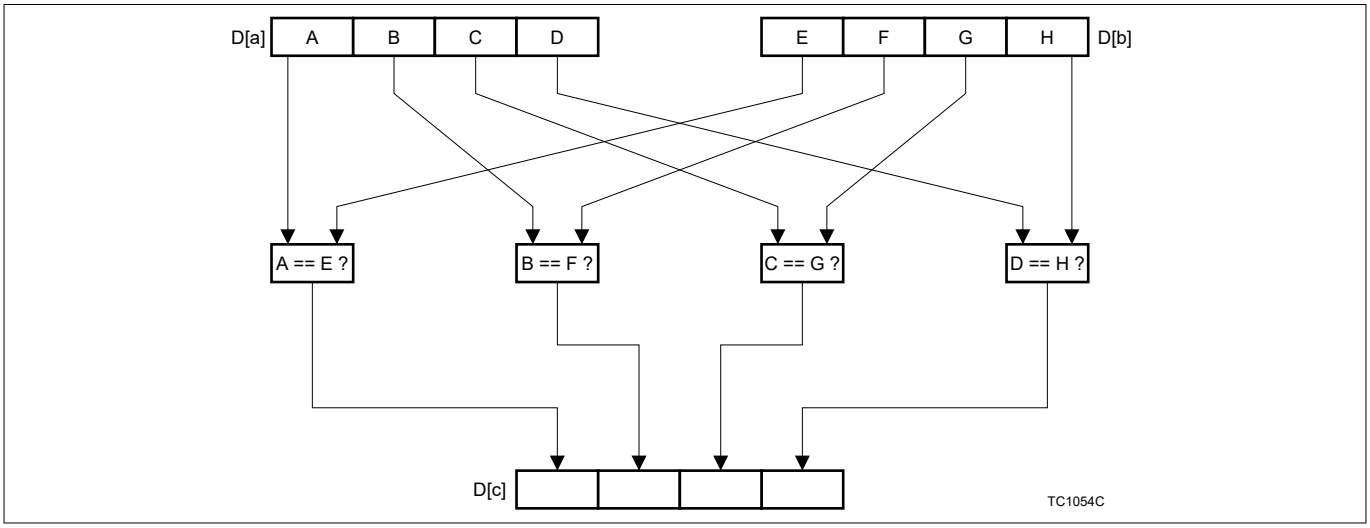


Figure 11 EQ.B instruction operation

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 the figure below. The available Boolean operations are:

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

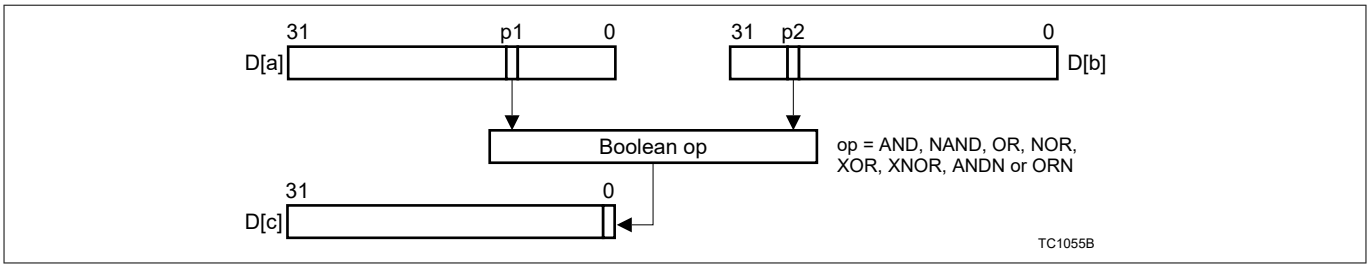


Figure 12 Boolean operations

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.

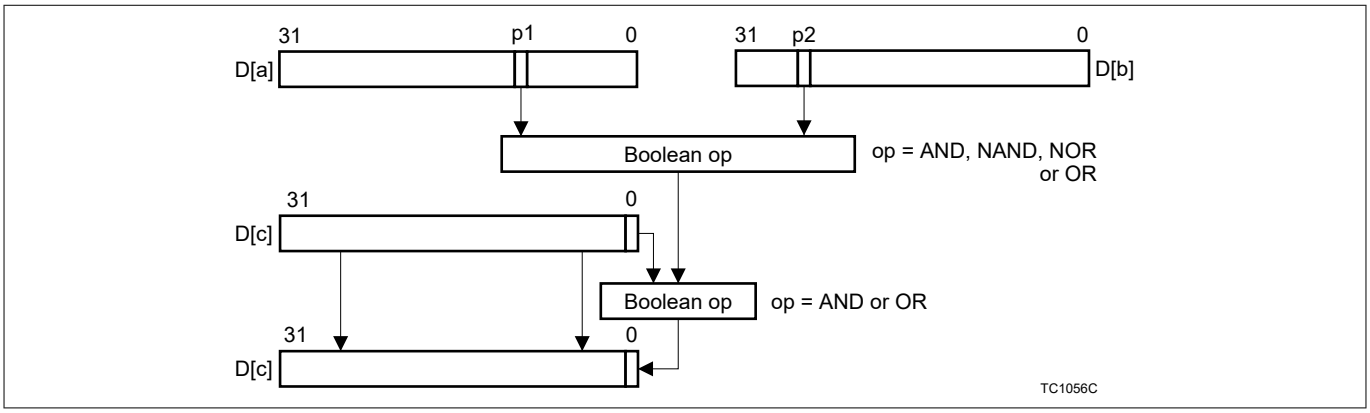


Figure 13 Three-input Boolean operation

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.

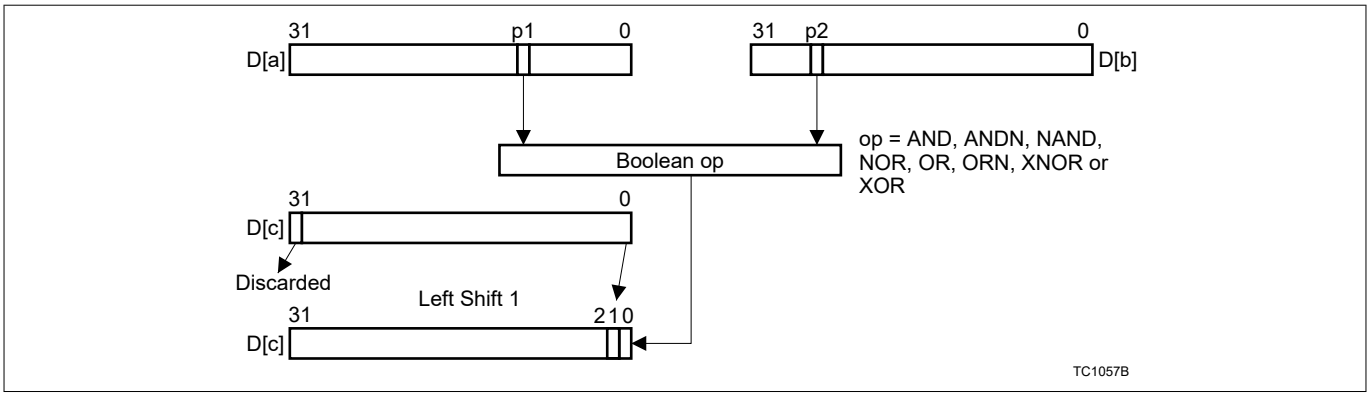


Figure 14 Shift plus Boolean operation

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), 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 the figure below.

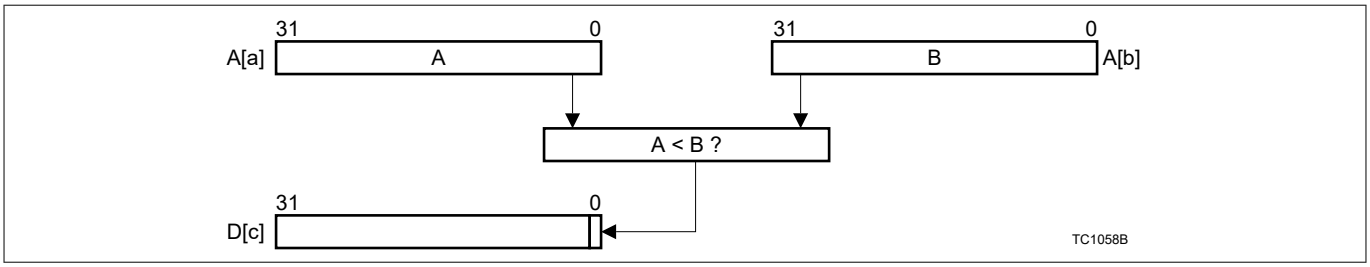


Figure 15 LT.A comparison operation

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 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.

***Note**: FCALL, FCALLA, FCALLI differ from normal call instructions as they do not save the caller's registers upper context.*

The return instruction, in addition to performing the return jump, restores the upper context.

***Note**: FRET differs from a normal return instruction and does not restore 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, JRI, CALLI, RET, RFE and RFH).

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 this figure.

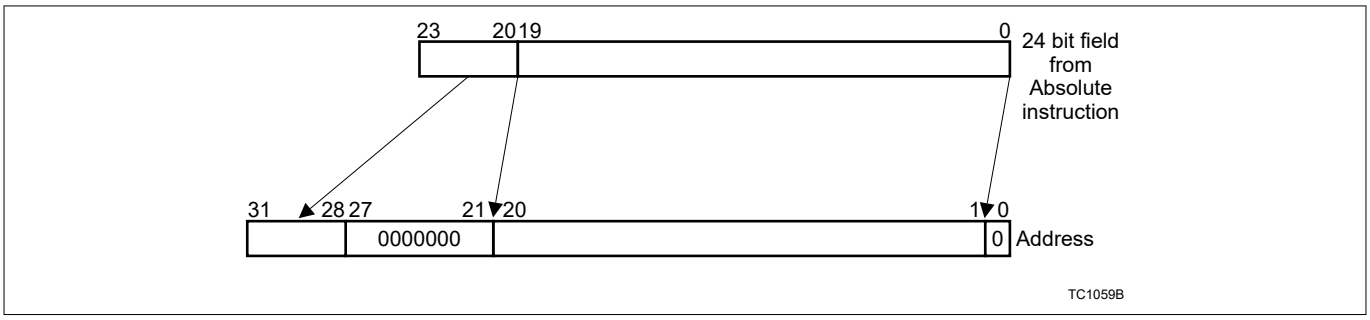


Figure 16 Calculation of absolute address

1.9.2 Conditional branch

The conditional branch instructions use the relative addressing mode, with a displacement value encoded in 4 (and in two cases 5), 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-bit 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 or a data register. The jump displacement field is limited to a 5-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
- LOOPU

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:

```
    mov    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 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	{offset 18[17:14], 14'b00000000000000, 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+1]+r	A[n]+A[n+1][15:0] (n is even)	BO

1.10.1 Load and store basic data types

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

Note that when the data loaded from memory is smaller than the destination register (that means 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.

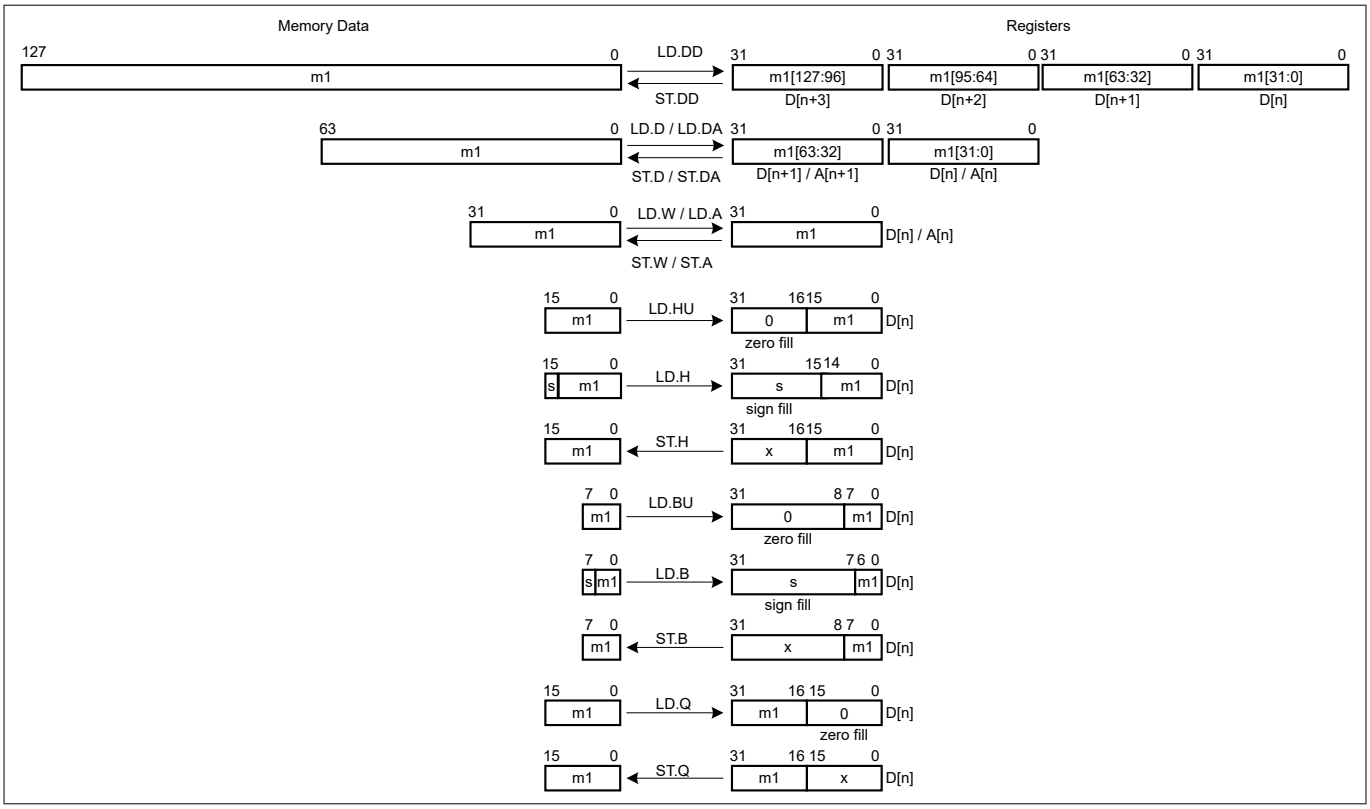


Figure 17 Load and store basic data type

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.

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; that means 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 to the CSA 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 STLCX 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™ 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 (DSYNC, LSYNC and ISYNC)

The TriCore™ architecture provides three synchronization primitives, DSYNC, LSYNC and ISYNC. These primitives provide a mechanism to software through which it can guarantee the ordering of various events within the system.

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.

LSYNC

The LSYNC primitive provides a mechanism through which a data memory barrier can be implemented.

The LSYNC instruction guarantees that all data accesses associated with instructions semantically prior to the LSYNC instruction are commenced on the CPU interconnect interface before any data memory accesses associated with an instruction semantically after LSYNC are initiated. This includes all accesses to the system bus and local data memory. As opposed to a DSYNC instruction, LSYNC will not wait for writes to complete at their target destination.

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 four 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 (that means User-1, User-0 or Supervisor mode)
- MFDCR
  - The move from dual core register instruction moves the contents of two naturally aligned addressed CSFRs into an extended data register. MFDCR can be executed in any mode (that means User-1, User-0 or Supervisor mode). The behavior of this instruction when only one CSFR exists is implementation specific
- 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
- MTDCR
  - The move to dual core register instruction moves the contents of an extended data register to the naturally aligned addressed CSFRs. To prevent unauthorized writes to the CSFRs the MTDCR instruction can only be executed in Supervisor mode. The behavior of this instruction when only one CSFR exists is implementation specific

Attention**: **An MTCR, or MTDCR or sequence of non-dependent (on side-effects) MTCR or MTDCR instructions 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 (that means 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, MTCR, MFDCR and MTDCR. The following figure summarizes TriCore™ core behavior when accessing CSFRs.

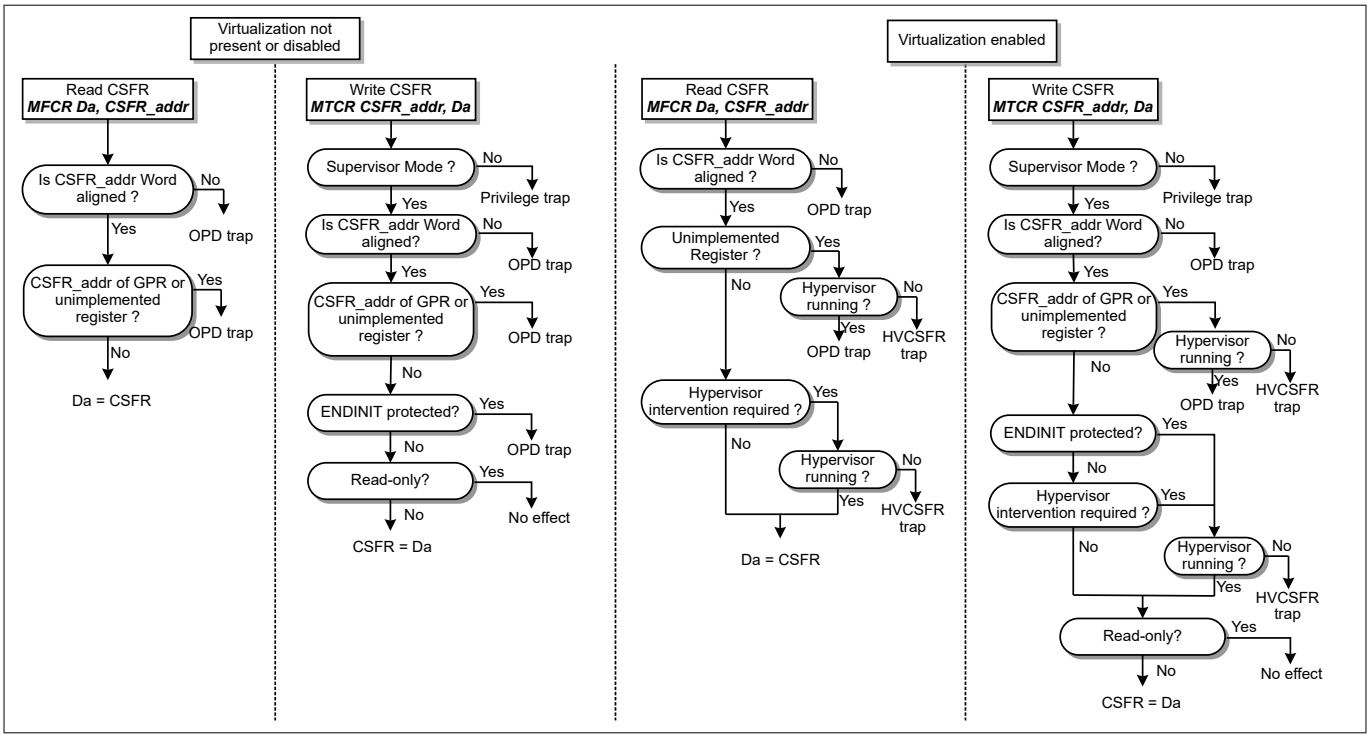


Figure 18 TriCore™ core behavior accessing CSFRs

1.12.4 Enabling and disabling the interrupt system

For control of the interruptibility of operations, the ENABLE, DISABLE and RESTORE instructions allow the explicit enabling, disabling or restoring of interrupts in both User-1 and Supervisor mode.

Virtualization extension not present or disabled

While interrupts are 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), as it is presented to the CPU as a trap.

If a User or system process erroneously disables interrupts for longer than a specified time, watchdog timers in the system or the TPS in the CPU 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).

Virtualization extension present and enabled

The ENABLE, DISABLE and RESTORE instructions only affect the currently running virtual machine (VM).

While interrupts are disabled, an interrupt for the current guest VM will not be taken by the guest VM regardless of the relative priorities of the guest VM and the highest interrupt pending targeting the currently running guest VM. An interrupt for a non running guest VM is not affected an could still be taken based on its own relative priority unless the hypervisor is currently running[1)].

The only ‘interrupt’ that is serviced while interrupts are disabled by the hypervisor is the NMI (Non-maskable interrupt), as it is presented to the CPU as a trap.

If code running in VM0 erroneously disables interrupts for longer than a specified time, watchdog timers in the system or the TPS in VM0 can be used to recover.

1 Transition from the hypervisor (VM0) to a guest VM always requires software intervention. There is no hardware mechanism to transition to a guest VM based on interrupt requests when the hypervisor is running

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 through a CALL operation. 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 Hypervisor call (HVCALL) and return from hypervisor (RFH)

When the Virtualization extension is present and enabled, two instructions are available to switch from a guest Virtual Machine (VM) to the hypervisor and from the hypervisor to a guest VM.

Hypervisor call (HVCALL)

The HVCALL instruction generates a hypervisor call trap, providing a secure mechanism for operating system code to request Hypervisor mode services.

The hypervisor call trap, like other hypervisor traps, vectors to the hypervisor trap handler table, using the three-bit hardware-furnished hypervisor trap class ID as an index.

The hypervisor trap class ID for hypervisor call traps is zero.

The hypervisor Trap Identification Number (HVTIN) is specified by an immediate constant in the HVCALL instruction and serves to identify the specific hypervisor mode service that is being requested.

Return from hypervisor (RFH)

The RFH instruction is used to return from an hypervisor interrupt or hypervisor trap handler to a guest VM.

This instruction perform similar operation to the return from exception (RFE); it restores the upper context of the calling virtual machine function or interrupted virtual machine task and branch to the return address contained in register A[11] (prior to the context restore operation and prior to a hardware resource switch).

1.12.7 Trap instructions

The trap on overflow (TRAPV) and trap on sticky overflow (TRAPSV) instructions can be used to cause a trap if the PSW.V and PSW.SV bits respectively, are set. See Program status word status flags and arithmetic instructions.

The trap invalid (TRAPINV) instruction can be used to cause an illegal opcode trap (class 2, TIN 1).

1.12.8 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 (COP) instructions

The TriCore™ instruction set architecture may be extended with optional architectural features and 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).

Three of these (n = 0, 1, 2) are reserved for use by the TriCore™ CPU, allowing one (n = 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).

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.

***Notes**:*

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

2. *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

The operand definitions are detailed in the following table.

Table 5 Operand definitions

Operand	Definition
A[n]	Address register n
D[n]	Data register n
E[n]	Extended 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]. The format is little endian and 'n' must be even. E[n][63:32] = D[n+1][31:0] E[n][31:0] = D[n][31:0] n must be even.
P[n]	Extended 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]. The format is little endian and 'n' must be even. P[n][63:32] = A[n+1][31:0] P[n][31:0] = A[n][31:0]
Q[n]	Quad data register containing a 128-bit value, constructed by pairing two extended data registers. The least significant bit is register D[n]. The format is little endian and 'n' is restricted to values 0, 4, 8 or 12. Q[n][127:96] = D[n+3][31:0] Q[n][95:64] = D[n+2][31:0] Q[n][63:32] = D[n+1][31:0] Q[n][31:0] = D[n][31:0]
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
offn	Offset value of 'n' bits, used to form the effective address in load and store instructions
pos1, pos2	Used to specify the position in a bit-field instruction
pos	Pos (position) is used with width to define a field

(table continues...)

Table 5 (continued) Operand definitions

Operand	Definition
width	Specifies the width of the bit-field in bit-field instructions

Operand modifiers

Instructions may operate on parts of input registers (for example in the case of packed arithmetic). The following table outlines the types of modifiers that are applied to data registers:

Table 6 Operand modifiers

Operand	Definition
L	Lower part of the data register (D[n][15:0])
U	Upper part of the data register (D[n][31:16])
UU, UL, LU, LL	Multiply-accumulate (MAC) source operand modifier. Specifies the register usage for a given instruction. The details for each instruction is provided in the quick reference section following the instruction syntax. For example: Packed multiply-add Q format MADD.H E[c], E[d], D[a], D[b] **UL**, n (RRR1) // instruction syntax 32||32 +||+ (16**U** * 16**U** || 16**L** * 16**L**)--> 32||32 // quick reference

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 7 Implicit operand definitions

Operand	Definition
D[15]	Implicit data register for many 16-bit instructions
A[10]	Stack Pointer (SP)

(table continues...)

Table 7 (continued) Implicit operand definitions

Operand	Definition
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 8 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
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 9 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/LD.DA, ST.DA
DD	Quad-word	128-bit data	LD.DD
W	Word	32-bit (word) data	EQ.W
A	Address	32-bit address	ADD.A, MOV.AA
Q	Q Format	16-bit/32-bit signed fraction	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
F	Float	IEEE 754-2019 single precision floating point	ADD.F
DF	Double	IEEE 754-2019 double precision floating point	ADD.DF

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

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 10 16-bit opcode formats

	15:14	13:12	11:10	9:8	7:6	5:4	3:2	1:0
SB	disp8				op1			
SBC	const4		disp4		op1			
SBR	s2		disp4		op1			
SBRN	n		disp4		op1			
SC	const8				op1			
SLR	s2		d		op1			
SLRO	off4		d		op1			

(table continues...)

Table 10 (continued) 16-bit opcode formats

	15:14	13:12	11:10	9:8	7:6	5:4	3:2	1:0
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 11 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	6:0
ABS	off18[9:6]		op2	off18[13:10]		off18[5:0]			off18[17:14]		s1/d		op1	
ABSB	off18[9:6]		op2	off18[13:10]		off18[5:0]			off18[17:14]		b	bpos3	op1	
B	disp24[15:0]								disp24[23:16]				op1	
BIT	d		pos2		op2	pos1			s2		s1		op1	
BO	off10[9:6]		op2			off10[5:0]			s2		s1/d		op1	
BOL	off16[9:6]		off16[15:10]			off16[5:0]			s2		s1/d		op1	
BRC	op2	disp15							const4		s1		op1	
BRN	op2	disp15							n[3:0]		s1		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	

(table continues...)

Table 11 (continued) 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	6:0
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 definitions

Table 12 Opcode field definitions

Name	Width	Definition
s1	4	Source register(s) number one[1)]
s2	4	Source register(s) number two[1)]
s3	4	Source register(s) number three[1)]
d	4	Destination register[1)]
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	• Multiplication result shift value (only 00_B and 01_B are valid) • Address shift value in add scale • Default to zero in all other operations using the RR format • Coprocessor number for coprocessor instructions
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

(table continues...)

Table 12 (continued) Opcode field definitions

Name	Width	Definition
off16	16	16-bit offset
-	-	Reserved field Read value is undefined Must be set to zero (0)
op1		Primary opcode
op2		Secondary opcode

1) For an extended register (E or P) and for a quad register (Q) the coding follows the register number (for example E[0] = 0000_B, E[2] = 0010_B, and so on

2.3 Instruction operation syntax

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

***Notes**:*

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

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

Table 13 RTL syntax

Syntax	Definition
A[n]	Address register ‘n’
bpos3	Bit position
const’n’	Constant value of ‘n’bits, used as instruction operand
CR	Core register
D[n]	Data register ‘n’
disp’n’	Displacement value of ‘n’ bits, used to form the effective address in branch instructions
EA	Effective address
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] • The most significant bit is in the odd register D[n + 1]
(expression)[p]	A single bit, with ordinal index ‘p’ in the bit-field ‘expression’
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, quad-word, 16-word
<mode>	Any addressing mode
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
off’n’	Offset value of ‘n’ bits, used to form the effective address in load and store instructions
PC	The address of the instruction in memory
pos	Single bit position
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] • The most significant bit is in the odd register A[n+1]

(table continues...)

Table 13 (continued) RTL syntax

Syntax	Definition
Q[n]	Data register containing a 128-bit value, constructed by pairing two extended registers • The least significant bit is in the register E[n] • The most significant bit is in the register E[n+2]
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}
[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
*	Multiply
/	Divide
%	Modulo

(table continues...)

Table 13 (continued) RTL syntax

Syntax	Definition
=	Assignment
==	Is equal to (comparison). Returns a Boolean result
!=	Not equal to (comparison). Returns a Boolean result
≈	Approximately equal to
||	Parallel operation
? :	Conditional expression (ternary operator)
∞	Infinity
//	Comment

2.3.1 RTL functions

Register transfer level functions are defined in the table which follows.

Table 14 RTL functions

Function	Definition
abs(x)	abs(x) returns `((x <0) ? (0 - x) : x);`
byte_select(x,s)	Returns a naturally aligned byte value from the 32-bit value ‘x’. ‘s’ contains the index of the selected byte, where: • The least significant byte has index zero • The most significant byte has index three
cache_address_ivld(EA)	Defined in ‘Cache RTL functions’ section
cache_address_ivld(EA, VMN)	Defined in ‘Cache RTL functions’ section
cache_address_wb(EA)	Defined in ‘Cache RTL functions’ section
cache_address_wb(EA, VMN)	Defined in ‘Cache RTL functions’ section
cache_address_wi(EA)	Defined in ‘Cache RTL functions’ section
cache_address_wi(EA, VMN)	Defined in ‘Cache RTL functions’ section
cache_index_ivld(entry)	Defined in ‘Cache RTL functions’ section
cache_index_ivld(entry, VMN)	Defined in ‘Cache RTL functions’ section
cache_index_wi(entry)	Defined in ‘Cache RTL functions’ section
cache_index_wi(entry, VMN)	Defined in ‘Cache RTL functions’ section
cache_index_wi(entry)	Defined in ‘Cache RTL functions’ section
cache_index_wi(entry, VMN)	Defined in ‘Cache RTL functions’ section

carry(a,b,c)	`carry(a,b,c) {`
	`result = a + b + c; // unsigned additions`
	`return result[32];`
	`}`

cdc_decrement()	If PSW.CDC == 7’b1111111 returns FALSE, otherwise decrements PSW.CDC.COUNT and returns TRUE if PSW.CDC.COUNT underflows, otherwise returns FALSE
cdc_increment()	If PSW.CDC == 7'b1111111 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'b1111111, otherwise returns FALSE

(table continues...)

Table 14 (continued) RTL functions

Function	Definition
crc_div(crc_in, gen, crc_width, data_width)	'crc_in' and 'gen' each contain the coefficients of polynomials defined over GF(2). In the polynomial f(x), represented by the binary value v, bit v[n] is the coefficient of the term x^n. The value returned by this function is the binary number representing the coefficients of the remainder from the following polymomial division: `(crc_in << shift) % (gen | (1 << crc_width) )` where `shift = min(crc_width, data_width)`
hvtrap(hvclass, hvtin)	Instruction will take hypervisor trap 'hvclass/hvtin'
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'
min(a, b)	Minimum value returns `((a < b) ? a : b)`
population_count(x)	Returns the number of bits set to one in 'x'
reverse(x,n)	Reverse the n-bit binary value x. Returns `{x[0], x[1], ..., x[n-2], x[n-1]}`
reverse16(x)	reverse(x, 16)
round16(x)	`= {(x + 32'h00008000)[31:16],16'h0000};`
sign_ext(x)	Sign extension; high-order bit of x is left extended

ssov(x,y)	Saturation on signed overflow
	`max_pos = (1 << (y - 1)) - 1;`
	`max_neg = -(1 << (y - 1));`
	`return ((x > max_pos) ? max_pos : ((x < max_neg) ? max_neg : x ));`

suov(x,y)	Saturation on unsigned overflow
	`max_pos = (1 << y) - 1;`
	`return ((x > max_pos) ? max_pos : ((x < 0) ? 0 : x));`

trap(class, tin)	Instruction will take trap 'class/tin'
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.

When the virtualization extension is present and enabled, the cache operations will only operate on content belonging to the currently running virtual machine. This information is also contained in the structure:

- Virtual machine number associated with the address (CACHE[ ].vmn)

***Notes**:*

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

2. *'index' and 'elem', which appear in the cache function descriptions, are the set_index and set_element values. These values are implementation dependent*

Table 15 Cache functions

Function	Definition

cache_address_ivld(EA)	`if (CACHE[EA].valid == 1) then {`
	`  CACHE[EA].valid=0;`
	`}`

cache_address_ivld(EA, VMN)	`if (CACHE[EA].valid == 1) then {`
	`  if (CACHE[EA].vmn == VMN) then {`
	`  CACHE[EA].valid=0;`
	`  }`
	`}`

cache_address_wb(EA)	`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;`
	`  }`
	`}`

(table continues...)

Table 15 (continued) Cache functions

Function	Definition

cache_address_wb(EA, VMN)	`if (CACHE[EA].valid == 1) then {`
	`  if (CACHE[EA].vmn == VMN) then {`
	`  if (CACHE[EA].modified == 1) then {`
	`  pa = CACHE[EA].physical_address;`
	`  M[pa,cacheline] = CACHE[EA].data;`
	`  CACHE[EA].modified = 0;`
	`  }`
	`  }`
	`}`

cache_address_wi(EA)	`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;`
	`}`

cache_address_wi(EA, VMN)	`if (CACHE[EA].valid == 1) then {`
	`  if (CACHE[EA].vmn == VMN) 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;`
	`  }`
	`}`

cache_index_ivld(entry)	`index=entry.index`
	`way=entry.way`
	`if (CACHE[index,way].valid == 1) then {`
	`  CACHE [index,way].valid=0;`
	`}`

cache_index_ivld(entry, VMN)	`way=entry.way`
	`index=entry.index`
	`if (CACHE[index,way].valid == 1) then {`
	`  if (CACHE[index,way].vmn == VMN) then {`
	`  CACHE [index,way].valid=0;`
	`  }`
	`}`

(table continues...)

Table 15 (continued) Cache functions

Function	Definition

cache_index_wb(entry)	`index=entry.index`
	`way=entry.way`
	`if (CACHE[index,way].valid == 1) then {`
	`  if (CACHE[index,way].modified == 1) then {`
	`  pa = CACHE[index,way].physical_address;`
	`  M[pa,cacheline] = CACHE[index,way].data;`
	`  CACHE[index,way].modified = 0;`
	`  }`
	`}`

cache_index_wb(entry,VMN)	`index=entry.index`
	`way=entry.way`
	`if (CACHE[index,way].valid == 1) then {`
	`  if (CACHE[index,way].vmn == VMN) then {`
	`  if(CACHE[index,way].modified == 1) then {`
	`  pa = CACHE[index,way].physical_address;`
	`  M[pa,cacheline] = CACHE[index,way].data;`
	`  CACHE[index,way].modified = 0;`
	`  }`
	`  }`
	`}`

cache_index_wi(entry)	`index=entry.index`
	`way=entry.way`
	`if (CACHE[index,way].valid == 1) then {`
	`  if (CACHE[index,way].modified == 1) then {`
	`  pa = CACHE[index,way].physical_address;`
	`  M[pa,cacheline] = CACHE[index,way].data;`
	`  }`
	`  CACHE[index,way].modified = 0;`
	`  CACHE[index,way].valid = 0;`
	`}`

cache_index_wi(entry,VMN)	`index=entry.index`
	`way=entry.way`
	`if (CACHE[index,way].valid == 1) then {`
	`  if (CACHE[index,way].vmn == VMN) then {`
	`  if (CACHE[index,way].modified == 1) then {`
	`  pa = CACHE[index,way].physical_address;`
	`  M[pa,cacheline] = CACHE[index,way].data;`
	`  }`
	`  CACHE[index,way].modified = 0;`
	`  CACHE[index,way].valid = 0;`
	`  }`
	`}`

2.3.3 Floating point operation syntax

The following tables define the floating point operation syntax.

Table 16 Floating point values

16-bit half precision		32-bit single precision		64-bit double precision	
HP_ADD_NAN	-	SP_ADD_NAN	$7FC00001_H$	DP_ADD_NAN	$7FF8000000000001_H$
HP_DIV_NAN	-	SP_DIV_NAN	$7FC00008_H$	DP_DIV_NAN	$7FF8000000000008_H$
HP_MUL_NAN	-	SP_MUL_NAN	$7FC00002_H$	DP_MUL_NAN	$7FF8000000000002_H$
HP_MAX_VALUE	$2^{16}*(1.0-2^{-11})$	SP_MAX_VALUE	$2^{128}*(1.0-2^{-24})$	DP_MAX_VALUE	$2^{1024}*(1.0-2^{-53})$
HP_MIN_NORMAL	2^{-14}	SP_MIN_NORMAL	2^{-126}	DP_MIN_NORMAL	2^{-1022}
HP_NEG_INFINITY	$FC00_H$	SP_NEG_INFINITY	$FF800000_H$	DP_NEG_INFINITY	$FFF0000000000000_H$
HP_POS_INFINITY	$7C00_H$	SP_POS_INFINITY	$7F800000_H$	DP_POS_INFINITY	$7FF0000000000000_H$
HP_QUIET_NAN	$7E00_H$	SP_QUIET_NAN	$7FC00000_H$	DP_QUIET_NAN	$7FF8000000000000_H$
HP_SQRT_NAN	-	SP_SQRT_NAN	$7FC00004_H$	DP_SQRT_NAN	$7FF8000000000004_H$

Table 17 Floating point operation functions

Function	Definition
add(x,y)	Adds the real value ‘x’ to the real value ‘y’ and returns an infinitely precise real result
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
divide(x,y)	Divides the real value ‘x’ by the real value ‘y’ and returns an infinitely precise real result
fp_abs(x)	Returns the infinitely precise absolute value of the real value ‘x’ `(x < 0.0) ? (0.0 - x) : x;`
i_real(x)	Returns an infinitely precise real number of equal value to the 32-bit signed integer value ‘x’
l_real(x)	Returns an infinitely precise real number of equal value to the 64-bit signed long integer value ‘x’
mul(x,y)	Multiply the real value ‘x’ by the real value ‘y’ and return an infinitely precise real result
u_real(x)	Returns an infinitely precise real number of equal value to the 32-bit unsigned integer value ‘x’
ul_real(x)	Returns an infinitely precise real number of equal value to the 64-bit unsigned long integer value ‘x’

Table 18 Half-precision floating point operation functions

Function	Definition
hp_real(x)	Returns the IEEE-754-2019 16-bit half precision floating point format value ‘x’ as an infinitely precise real value

(table continues...)

Table 18 (continued) Half-precision floating point operation functions

Function	Definition
hp_round(f, m)	Rounds the real value ‘f’ to another real value which is exactly representable by a 16-bit half precision format number using the type of rounding mode specified by ‘m’, compliant with IEEE-754-2019. Note that the result may be representable only as a subnormal half precision value
hp_sign_bit(x)	Returns the 1-bit value x[15]
ieee754_hp_format(f)	Returns the real value ‘f’ in the 16-bit IEEE-754-2019 half precision floating point format. ‘f’ is converted to the correct IEEE-754-2019 result on overflow or underflow. The return value may be a 16-bit half precision subnormal value
is_hp_inf(x)	Takes the IEEE-754-2019 16-bit half precision floating point format value ‘x’ and returns the Boolean result of the expression: `(x[15:0] == HP_POS_INFINITY) OR (x[15:0] == HP_NEG_INFINITY);`
is_hp_nan(x)	Takes the IEEE-754-2019 16-bit half precision floating point format value ‘x’ and returns the Boolean result of the expression: `(is_hp_s_nan(x) OR is_hp_q_nant(x));`
is_hp_q_nan(x)	Takes the IEEE-754-2019 16-bit half precision floating point format value ‘x’ and returns the Boolean result of the expression: $(x[14:10] == 11111_B)$ AND `(x[9]==1);`
is_hp_s_nan(x)	Takes the IEEE-754-2019 16-bit half precision floating point format value ‘x’ and returns the Boolean result of the expression: $(x[14:10] == 11111_B)$ AND `(x[9] == 0) AND (x[8:0] !=0);`

Table 19 Single-precision floating point operation functions

Function	Definition
ieee754_sp_format(x)	Returns the real value ‘x’ in the standard 32-bit single precision IEEE-754-2019 floating point format. ‘x’ is converted to the correct IEEE-754-2019 result on overflow or underflow
is_sp_inf(x)	Takes the IEEE-754-2019 32-bit single precision floating point format value ‘x’ and returns the Boolean result of the expression: `(is_sp_neg_inf(x) OR is_sp_pos_inf(x));`
is_sp_nan(x)	Takes the IEEE-754-2019 32-bit single precision floating point format value ‘x’ and returns the Boolean result of the expression: `(is_sp_s_nan(x) OR is_sp_q_nan(x));`
is_sp_neg_inf(x)	Takes the IEEE-754-2019 32-bit single precision floating point format value ‘x’ and returns the Boolean result of the expression: `(x[31:0] == NEG_INFINITY);`
is_sp_pos_inf(x)	Takes the IEEE-754-2019 32-bit single precision floating point format value ‘x’ and returns the Boolean result of the expression: `(x[31:0] == POS_INFINITY);`

(table continues...)

Table 19 (continued) Single-precision floating point operation functions

Function	Definition
is_sp_q_nan(x)	Takes the IEEE-754-2019 32-bit single precision floating point format value ‘x’ and returns the Boolean result of the expression: `(x[30:23] == 8’b11111111) AND (x[22] == 1’b1);`
is_sp_s_nan(x)	Takes the IEEE-754-2019 32-bit single precision floating point format value ‘x’ and returns the Boolean result of the expression: `(x[30:23] == 8'b11111111) AND (x[22] == 1’b0) AND (x[21:0] != 0);`
is_sp_subnorm(x)	Takes the IEEE-754-2019 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);`
is_sp_zero(x)	Takes the IEEE-754-2019 32-bit single precision floating point format value ‘x’ and returns the Boolean result of the expression: `(x[30:0] == 0);`
sp_eq(x,y)	Returns TRUE if ‘x’ is equal to ‘y’ according to the IEEE-754-2019 comparison predicates for 32-bit single precision floating point numbers otherwise returns FALSE
sp_gt(x,y)	Returns TRUE if ‘x’ is greater than ‘y’ according to the IEEE-754-2019 comparison predicates for 32-bit single precision floating point numbers otherwise returns FALSE
sp_lt(x,y)	Returns TRUE if ‘x’ is less than ‘y’ according to the IEEE-754-2019 comparison predicates for 32-bit single precision floating point numbers otherwise returns FALSE
sp_real(x)	Returns the IEEE-754-2019 32-bit single precision floating point format value ‘x’ as an infinitely precise real value
sp_round_to_integer(x,m)	Rounds the IEEE-754-2019 32-bit single precision floating point format value ‘x’, using the type of round mode specified by ‘m’ compliant with IEEE-754-2019, to a signed integer value of infinite width.
sp_round_to_q31(x,m)	Rounds the real value ‘x’, using the type of round mode specified by ‘m’ compliant with IEEE-754-2019, to a Q format result of infinite width.
sp_round_to_unsigned(x,m)	Rounds the IEEE-754-2019 32-bit single precision floating point format value ‘x’, using the type of round mode specified by ‘m’ compliant with IEEE-754-2019, to an unsigned integer value of infinite width.
sp_round(x,m)	Rounds the real value ‘x’ using the type of rounding mode specified by ‘m’ compliant with IEEE-754-2019 for 32-bit single precision floating point numbers
sp_sign_bit(x)	Returns the 1-bit value x[31]

(table continues...)

Table 19 (continued) Single-precision floating point operation functions

Function	Definition

sp_subnorm_to_zero(x)	If the IEEE-754-2019 32-bit single precision floating point format value ‘x’ is a subnormal, return the appropriately signed exact zero. Otherwise return ‘x’ as an infinitely precise real value.
	`if (is_sp_subnorm(x)) then`
	`  if (x < 0) then`
	`  return -0.0;`
	`  else`
	`  return +0.0;`
	`else`
	`  return sp_real(x);`

Table 20 Double-precision floating point operation functions

Function	Definition
dp_eq(x,y)	Returns TRUE if ‘x’ is equal to ‘y’ according to the IEEE-754-2019 comparison predicates for 64-bit double precision floating point numbers otherwise returns FALSE
dp_gt(x,y)	Returns TRUE if ‘x’ is greater than ‘y’ according to the IEEE-754-2019 comparison predicates for 64-bit double precision floating point numbers otherwise returns FALSE
dp_lt(x,y)	Returns TRUE if ‘x’ is less than ‘y’ according to the IEEE-754-2019 comparison predicates for 64-bit double precision floating point numbers otherwise returns FALSE
dp_real(x)	Returns the IEEE-754-2019 64-bit double precision floating point format value ‘x’ as an infinitely precise real value
dp_round_to_integer(x,m)	Rounds the IEEE-754-2019 64-bit double precision floating point format value ‘x’, using the type of round mode specified by ‘m’ compliant with IEEE-754-2019, to a signed integer value of infinite width
dp_round_to_long_integer(x,m)	Rounds the IEEE-754-2019 64-bit double precision floating point format value ‘x’, using the type of round mode specified by ‘m’ compliant with IEEE-754-2019, to a signed long integer value of infinite width
dp_round_to_unsigned_long(x, m)	Rounds the IEEE-754-2019 64-bit double precision floating point format value ‘x’, using the type of round mode specified by ‘m’ compliant with IEEE-754-2019, to an unsigned long integer value of infinite width
dp_round_to_unsigned(x,m)	Rounds the IEEE-754-2019 64-bit double precision floating point format value ‘x’, using the type of round mode specified by ‘m’ compliant with IEEE-754-2019, to an unsigned integer value of infinite width
dp_round(x,m)	Rounds the real value ‘x’, using the type of rounding mode specified by ‘m’ compliant with IEEE-754-2019, for 64-bit double precision floating point numbers
dp_sign_bit(x)	Returns the 1-bit value x[63]

(table continues...)

Table 20 (continued) Double-precision floating point operation functions

Function	Definition
ieee754_dp_format(x)	Returns the real value ‘x’ in the standard 64-bit double precision IEEE-754-2019 floating point format. ‘x’ is converted to the correct IEEE-754-2019 result on overflow or underflow
is_dp_inf(x)	Takes the IEEE-754-2019 64-bit double precision floating point format value ‘x’ and returns the Boolean result of the expression: `(is_dp_neg_inf(x) OR is_dp_pos_inf(x));`
is_dp_nan(x)	Takes the IEEE-754-2019 64-bit double precision floating point format value ‘x’ and returns the Boolean result of the expression: `(is_dp_s_nan(x) OR is_dp_q_nan(x));`
is_dp_neg_inf(x)	Takes the IEEE-754-2019 64-bit double precision floating point format value ‘x’ and returns the Boolean result of the expression: `(x[63:0] == DP_NEG_INFINITY);`
is_dp_pos_inf(x)	Takes the IEEE-754-2019 32-bit single precision floating point format value ‘x’ and returns the Boolean result of the expression: `(x[63:0] == DP_POS_INFINITY);`
is_dp_q_nan(x)	Takes the IEEE-754-2019 64-bit double precision floating point format value ‘x’ and returns the Boolean result of the expression: `(x[62:52] == 11’b11111111111) AND (x[51] == 1’b1);`
is_dp_s_nan(x)	Takes the IEEE-754-2019 64-bit double precision floating point format value ‘x’ and returns the Boolean result of the expression: `(x[62:52] == 11'b11111111111) AND (x[51] == 1’b0) AND (x[50:0] != 0);`
is_dp_subnorm(x)	Takes the IEEE-754-2019 64-bit double precision floating point format value ‘x’ and returns the Boolean result of the expression: `(x[62:52] == 0) AND (x[51:0] != 0);`
is_dp_zero(x)	Takes the IEEE-754-2019 64-bit double precision floating point format value ‘x’ and returns the Boolean result of the expression: `(x[62:0] == 0);`

2.4 Coprocessor instructions

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

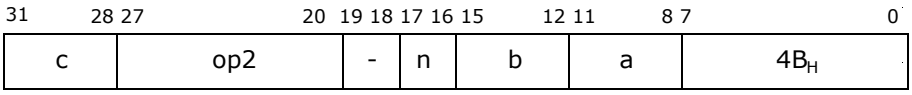
The coprocessors operate in a similar manner to the integer instructions, receiving operands from the general purpose data registers, returning a result to the same registers.

The architecture supports the operation of up to four concurrent coprocessors (n = 0, 1, 2, 3).

Three of these (n = 0, 1, 2) are reserved for use by the TriCore™ CPU, allowing one (n = 3) for use by the application hardware.

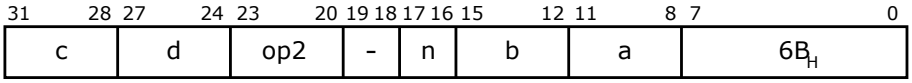
***Note**: Support for external coprocessors is implementation specific.*

COPROCESSOR D[c], D[a], D[b] (RR)




```
// Potential usages
op2[n](D[a]);                // no result
D[c] = op2[n](D[a]);
D[c] = op2[n](D[a], D[b]);
D[c] = op2[n](D[b], D[a]);
D[c] = op2[n](E[a]);
D[c] = op2[n](E[a], E[b]);
E[c] = op2[n](D[a]);
E[c] = op2[n](D[a], D[b]);
E[c] = op2[n](E[a]);
E[c] = op2[n](E[a], E[b]);
```

COPROCESSOR D[c], D[d], D[a], D[b] (RRR)




```
// Potential usages
D[c] = op2[n](D[d], D[a]);
D[c] = op2[n](D[d], D[a], D[b]);
D[c] = op2[n](E[d], D[a]);
E[c] = op2[n](E[d], D[b]);
E[c] = op2[n](E[d], E[a]);
E[c] = op2[n](E[d], E[a], E[b]);
```

Figure 19 Coprocessor instructions

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.

***Notes**:*

1. *In the following table, 'result' for 32-bit instructions is D[c] or E[c]. For 16-bit instructions it is D[a] or D[15](when implicit)*
2. *The PSW register is defined in volume 1, core registers*

Table 21 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;

*** Programming note: V (overflow) and AV (advanced overflow) status bits**

Because the TriCore™ instruction set contains many compound instructions (MULR, MAC, ABSDIF), it is necessary to understand when the overflow flags are 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:

Table 22 OS and I/O privileged instructions

Hypervisor[1)]	Supervisor	User-1 mode	User-0 mode
RFH	BISR	ENABLE	All others (including DEBUG)
CACHEA.I.VM	CACHEA.I	DISABLE	
CACHEA.W.VM	CACHEI.I	RESTORE	
CACHEA.WI.VM	HVCALL[1)]		
CACHEI.I.VM	MTCR		
CACHEI.W.VM	MTDCR		
CACHEI.WI.VM	RFM		

1) Attempting to execute these instructions when the virtualization extension is not present will result in a UOPC trap

3 Instruction set

3.1 CPU instructions

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

3.x.xxx J (1)

Jump Unconditional (2)

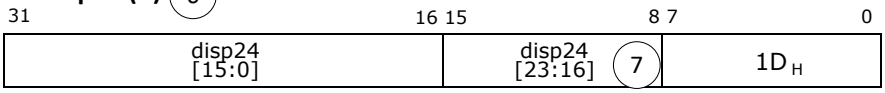
Description

Note: *Execution Mode - This instruction can only be executed in the following modes: Supervisor* (3)

Add the value specified by disp24, sign-extended and multiplied by 2, to the contents of PC and jump to that address. (4)

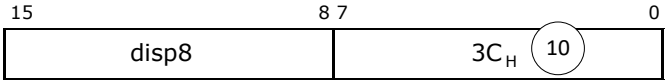
Add the value specified by disp8, sign-extended and multiplied by 2, to the contents of PC and jump to that address. (5)

J disp24 (B) (6)




```
PC = PC + sign_ext(disp24) * 2;
```
(8)

J disp8 (SB) (9)




```
PC = PC + sign_ext(disp8) * 2;
```
(11)

Status flags

C	Not set by this instruction. (12)
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
```
(13)

```
j     foobar
```
(14)

See also

JA, JI, JL, JLA, JLI, LOOPU (15)

Key:

1. Instruction Mnemonic
2. Instruction Longname
3. Example of execution mode restrictions for instructions requiring elevated privileges.
4. Description (32-bit) – Including potential execution mode restrictions
5. Description (16-bit)
6. Syntax (32-bit) and instruction format in parentheses. Note also (16)
7. Opcodes (32-bit)
8. Operation in RTL format (32-bit)
9. Syntax (16-bit)
10. Opcodes (16-bit)
11. Operation in RTL format (16-bit)
12. Status flags (User Status Bits)
13. Instruction examples (32-bit)
14. Instruction examples (16-bit)
15. Related instructions
16. Operation quick reference following syntax (MAC instructions only)

MSUB D[c], D[d], D[a], const9 (RCR)

32 - (32 * K9) --> 64 signed (16)

3.1.1 ABS

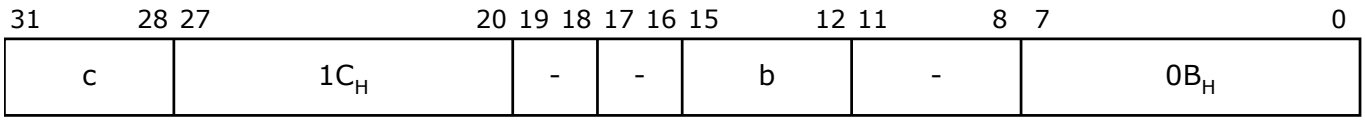
Absolute Value

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 32-bit signed integers.

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




```
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 > $7FFFFFFF_H$); 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

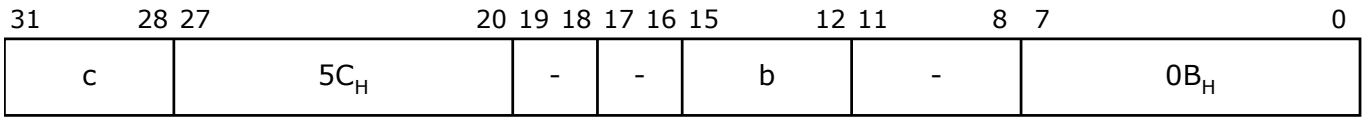
3.1.2 ABS.B

Absolute Value Packed Byte

Description

Put the absolute value of each byte in data register D[b] into the corresponding byte of data register D[c]. The operands are treated as 8-bit signed integers.
The overflow condition is calculated for each byte of the packed quantity.

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




```
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]};
```

Status Flags

C	Not set by this instruction.

V	ov_byte3 = (result_byte3 > $7F_H$);
	ov_byte2 = (result_byte2 > $7F_H$);
	ov_byte1 = (result_byte1 > $7F_H$);
	ov_byte0 = (result_byte0 > $7F_H$);
	overflow = ov_byte3 OR ov_byte2 OR ov_byte1 OR ov_byte0;
	if (overflow) then PSW.V = 1 else PSW.V = 0;

SV	if (overflow) then PSW.SV = 1 else PSW.SV = PSW.SV;

AV	aov_byte3 = result_byte3[7] ^ result_byte3[6];
	aov_byte2 = result_byte2[7] ^ result_byte2[6];
	aov_byte1 = result_byte1[7] ^ result_byte1[6];
	aov_byte0 = result_byte0[7] ^ result_byte0[6];
	advanced_overflow = aov_byte3 OR aov_byte2 OR aov_byte1 OR aov_byte0;
	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

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

See Also

ABS.H, ABSS.H, ABSDIF.B, ABSDIF.H, ABSDIFS.H

3.1.3 ABS.H

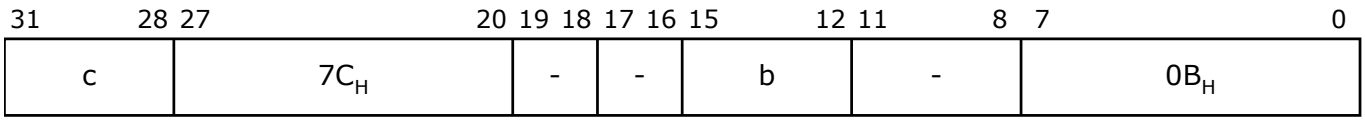
Absolute Value Packed Half-word

Description

Put the absolute value of each half-word in data register D[b] into the corresponding half-word of data register D[c]. The operands are treated as 16-bit signed integers.

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

ABS.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] = {result_halfword1[15:0], result_halfword0[15:0]};
```

Status Flags

C	Not set by this instruction.

V	ov_halfword1 = (result_halfword1 > $7FFF_H$);
	ov_halfword0 = (result_halfword0 > $7FFF_H$);
	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[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

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

See Also

ABS.B, ABSS.H, ABSDIF.B, ABSDIF.H, ABSDIFS.H

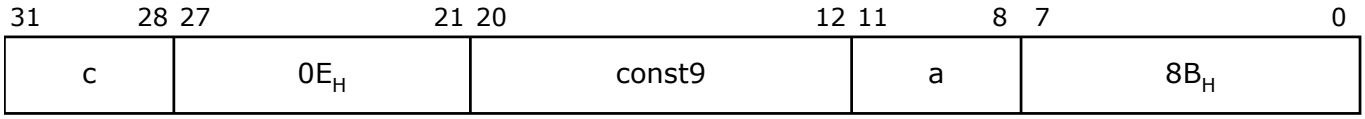
3.1.4 ABSDIF

Absolute Value of Difference

Description

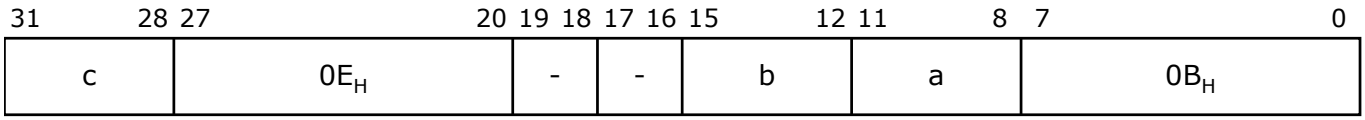
Put the absolute value of the difference between D[a] and either D[b] (instruction format RR) or const9 (instruction format RC) in D[c]; i.e. if the contents of data register D[a] are greater than either D[b] (format RR) or const9 (format RC), then subtract D[b] (format RR) or const9 (format RC) from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

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




```
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)




```
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 > $7FFFFFFF_H$) OR (result < -80000000_H); 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, #126
absdif    d3, d1, d2
```

See Also

ABS, ABSS, ABSDIFS

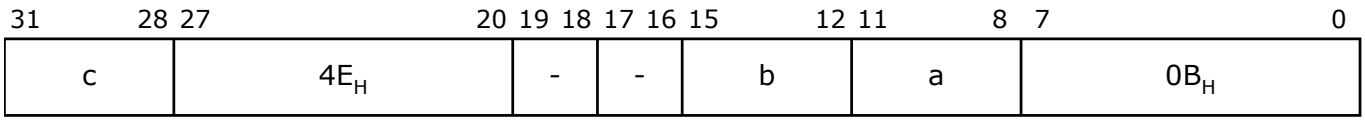
3.1.5 ABSDIF.B

Absolute Value of Difference Packed Byte

Description

Compute the absolute value of the difference between the corresponding bytes of D[a] and D[b], and put each result in the corresponding byte of D[c]. The operands are treated as 8-bit signed integers.
The overflow condition is calculated for each byte of the packed register.

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




```
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]);
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]);
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]);
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] = {result_byte3[7:0], result_byte2[7:0], result_byte1[7:0], result_byte0[7:0]};
```

Status Flags

C	Not set by this instruction.

V	ov_byte3 = (result_byte3 > $7F_H$) OR (result_byte3 < -80_H);
	ov_byte2 = (result_byte2 > $7F_H$) OR (result_byte2 < -80_H);
	ov_byte1 = (result_byte1 > $7F_H$) OR (result_byte1 < -80_H);
	ov_byte0 = (result_byte0 > $7F_H$) OR (result_byte0 < -80_H);
	overflow = ov_byte3 OR ov_byte2 OR ov_byte1 OR ov_byte0;
	if (overflow) then PSW.V = 1 else PSW.V = 0;

SV	if (overflow) then PSW.SV = 1 else PSW.SV = PSW.SV;

AV	aov_byte3 = result_byte3[7] ^ result_byte3[6];
	aov_byte2 = result_byte2[7] ^ result_byte2[6];
	aov_byte1 = result_byte1[7] ^ result_byte1[6];
	aov_byte0 = result_byte0[7] ^ result_byte0[6];
	advanced_overflow = aov_byte3 OR aov_byte2 OR aov_byte1 OR aov_byte0;
	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

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

See Also

ABS.B, ABS.H, ABSS.H, ABSDIF.H, ABSDIFS.H

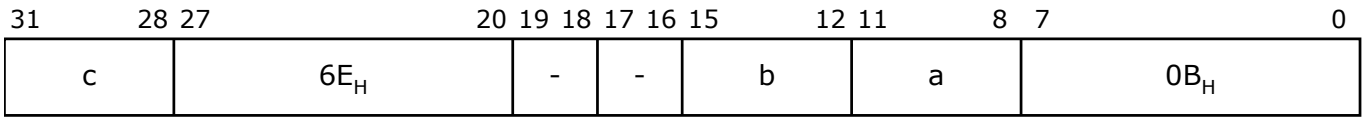
3.1.6 ABSDIF.H

Absolute Value of Difference Packed Half-word

Description

Compute the absolute value of the difference between 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 16-bit signed integers.
The overflow condition is calculated for each half-word of the packed register.

ABSDIF.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] = {result_halfword1[15:0], result_halfword0{15:0]};
```

Status Flags

C	Not set by this instruction.

V	ov_halfword1 = (result_halfword1 > $7FFF_H$) OR (result_halfword1 < -8000_H);
	ov_halfword0 = (result_halfword0 > $7FFF_H$) OR (result_halfword0 < -8000_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[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

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

See Also

ABS.B, ABS.H, ABSS.H, ABSDIF.B, ABSDIFS.H

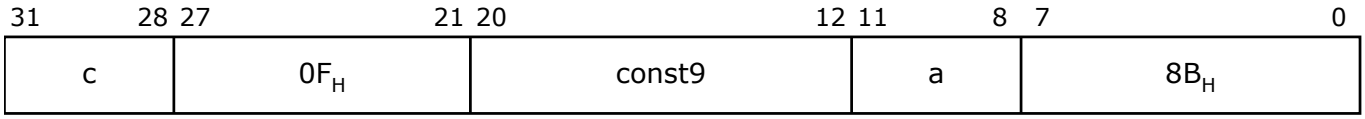
3.1.7 ABSDIFS

Absolute Value of Difference with Saturation

Description

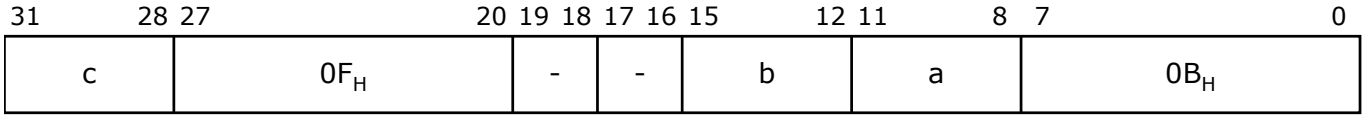
Put the saturated absolute value of the difference between D[a] and either D[b] (instruction format RR) or const9 (instruction format RC) in D[c]; i.e. if the contents of data register D[a] are greater than either D[b] (format RR) or const9 (format RC), then subtract D[b] (format RR) or const9 (format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

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




```
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)




```
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 > 7FFFFFFF$_H$) OR (result < -80000000$_H$); 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, #126
absdifs    d3, d1, d2
```

See Also

ABS, ABSDIF, ABSS

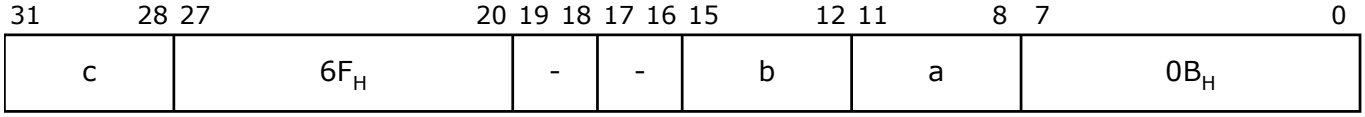
3.1.8 ABSDIFS.H

Absolute Value of Difference Packed Half-word with Saturation

Description

Compute the saturated 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 16-bit signed 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 > $7FFF_H$) OR (result_halfword1 < -8000_H);
	ov_halfword0 = (result_halfword0 > $7FFF_H$) OR (result_halfword0 < -8000_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[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

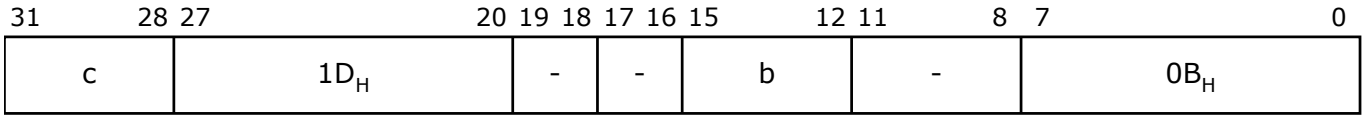
3.1.9 ABSS

Absolute Value with Saturation

Description

Put the saturated 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 32-bit signed integers with saturation on signed 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 > $7FFFFFFF_H$); 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

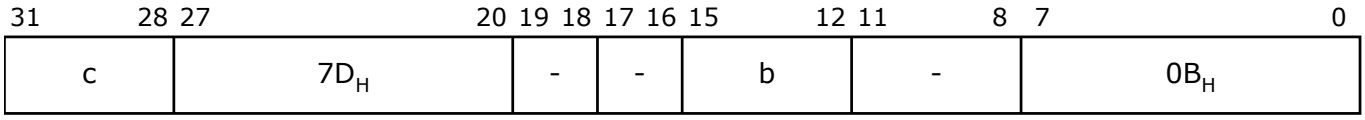
3.1.10 ABSS.H

Absolute Value Packed Half-word with Saturation

Description

Put the saturated 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 8-bit or 16-bit signed 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_H and the saturation yields $7FFF_H$.

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 > $7FFF_H$);
	ov_halfword0 = (result_halfword0 > $7FFF_H$);
	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[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

3.1.11 ADD

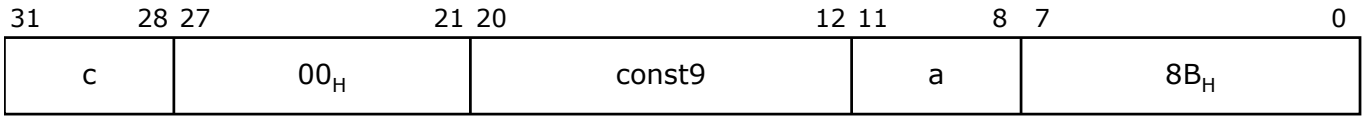
Add

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 signed integers, and the const9 value is sign-extended before the addition is performed.

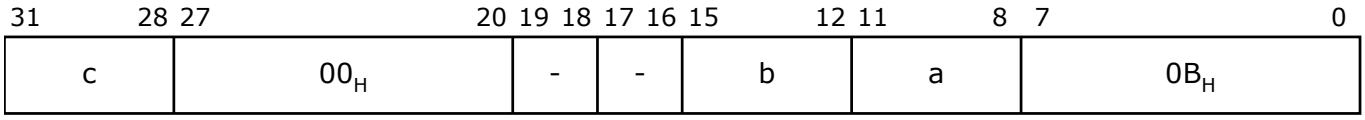
Add the contents of either data register D[a] or D[15] to the contents of data register D[b] or const4, and put the result in either data register D[a] or D[15]. The operands are treated as 32-bit signed integers, and the const4 value is sign-extended before the addition is performed.

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



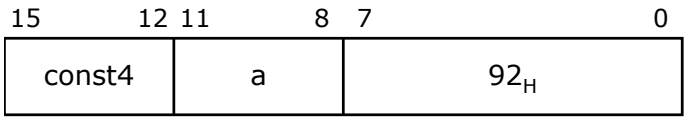

```
result = D[a] + sign_ext(const9);
D[c] = result[31:0];
```

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



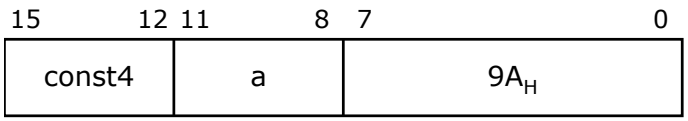

```
result = D[a] + D[b];
D[c] = result[31:0];
```

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



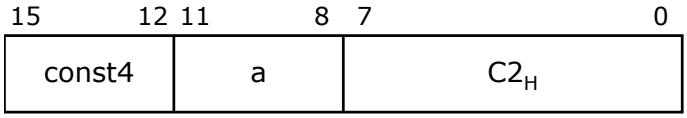

```
result = D[15] + sign_ext(const4);
D[a] = result[31:0];
```

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



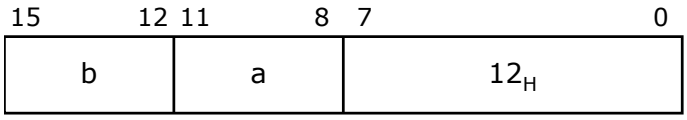

```
result = D[a] + sign_ext(const4);
D[15] = result[31:0];
```

ADD D[a], const4 (SRC)



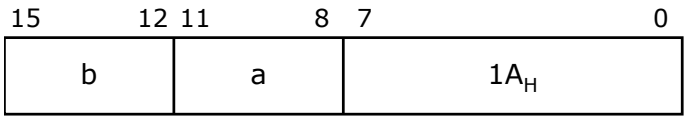

```
result = D[a] + sign_ext(const4);
D[a] = result[31:0];
```

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



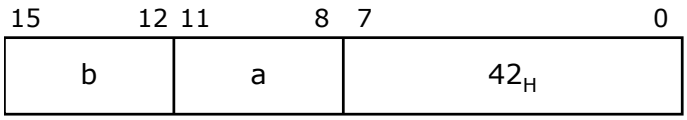

```
result = D[15] + D[b];
D[a] = result[31:0];
```

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




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

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




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

Status Flags

C	Not set by this instruction.
V	overflow = (result > 7FFFFFFF$_H$) OR (result < -80000000$_H$); 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, #126
add     d3, d1, d2
```

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

See Also

ADDC, ADDI, ADDIH, ADDS, ADDS.U, ADDX

3.1.12 ADD.A

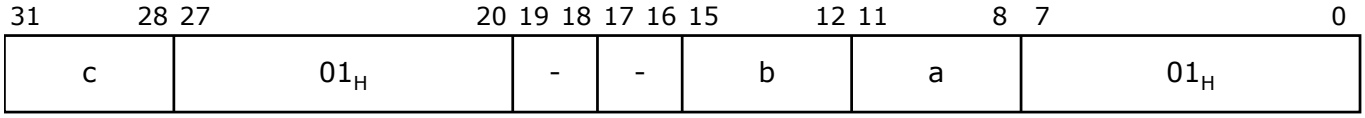
Add Address

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].

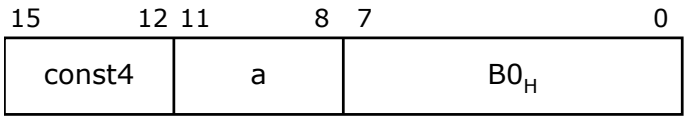
Add the contents of address register A[a] to the contents of either address register A[b] or const4 and put the result in address register A[a].

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



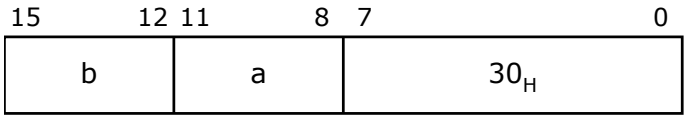

```
A[c] = A[a] + A[b];
```

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




```
A[a] = A[a] + sign_ext(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    a3, #6
add.a    a1, a2
```

See Also

ADDIH.A, ADDSC.A, ADDSC.AT, SUB.A

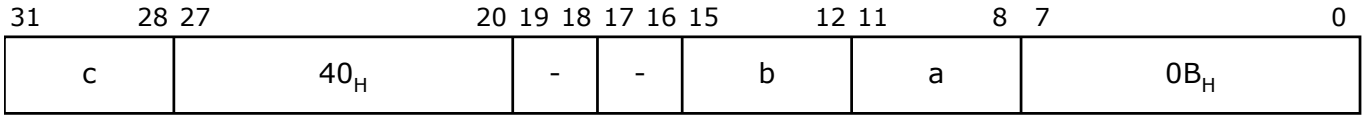
3.1.13 ADD.B

Add Packed Byte

Description

Add the contents of each byte of D[a] and D[b] and put the result in each corresponding byte of D[c]. The overflow condition is calculated for each byte 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]};
```

Status Flags

C	Not set by this instruction.

V	ov_byte3 = (result_byte3 > $7F_H$) OR (result_byte3 < -80_H);
	ov_byte2 = (result_byte2 > $7F_H$) OR (result_byte2 < -80_H);
	ov_byte1 = (result_byte1 > $7F_H$) OR (result_byte1 < -80_H);
	ov_byte0 = (result_byte0 > $7F_H$) OR (result_byte0 < -80_H);
	overflow = ov_byte3 OR ov_byte2 OR ov_byte1 OR ov_byte0;
	if (overflow) then PSW.V = 1 else PSW.V = 0;

SV	if (overflow) then PSW.SV = 1 else PSW.SV = PSW.SV;

AV	aov_byte3 = result_byte3[7] ^ result_byte3[6];
	aov_byte2 = result_byte2[7] ^ result_byte2[6];
	aov_byte1 = result_byte1[7] ^ result_byte1[6];
	aov_byte0 = result_byte0[7] ^ result_byte0[6];
	advanced_overflow = aov_byte3 OR aov_byte2 OR aov_byte1 OR aov_byte0;
	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.b    d3, d1, d2
```

See Also

ADD.H, ADDS.H, ADDS.HU

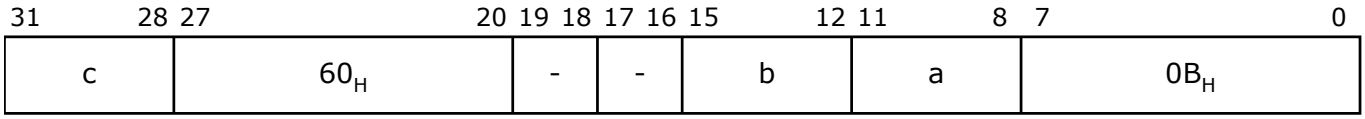
3.1.14 ADD.H

Add Packed Half-word

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]. The overflow condition is calculated for half-word of the packed quantity.

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 this instruction.

V	ov_halfword1 = (result_halfword1 > $7FFF_H$) OR (result_halfword1 < -8000_H);
	ov_halfword0 = (result_halfword0 > $7FFF_H$) OR (result_halfword0 < -8000_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[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

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

See Also

ADD.B, ADDS.H, ADDS.HU

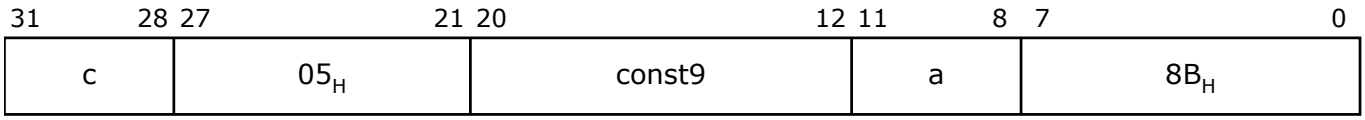
3.1.15 ADDC

Add with Carry

Description

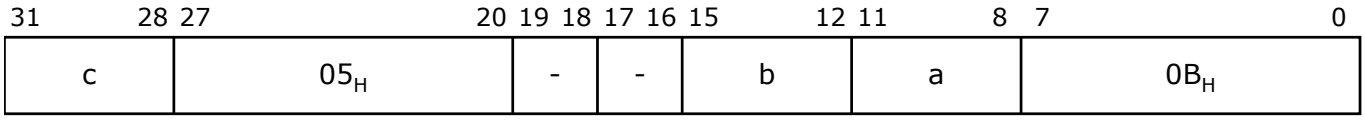
Add the contents of data register D[a] to the contents of either data register D[b] (instruction format RR) or const9 (instruction format RC) plus the carry bit, and put the result in data register D[c]. The operands are treated as 32-bit signed integers. The value const9 is sign-extended before the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

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




```
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)




```
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$_H$) OR (result < -80000000$_H$); 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, #126
addc    d3, d1, d2
```

See Also

ADD, ADDI, ADDIH, ADDS, ADDS.U, ADDX, SUBC

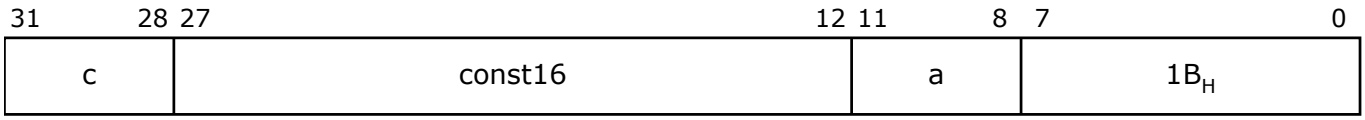
3.1.16 ADDI

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 > $7FFFFFFF_H$) OR (result < -80000000_H); 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

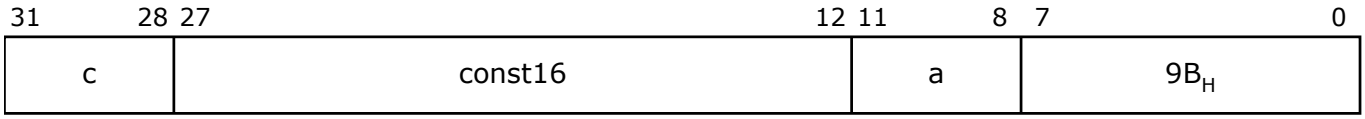
3.1.17 ADDIH

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$_H$) OR (result < -80000000$_H$); 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

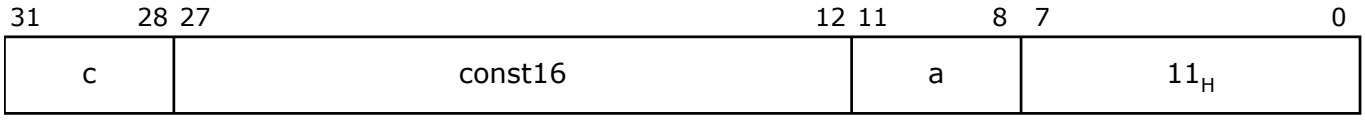
3.1.18 ADDIH.A

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

3.1.19 ADDS

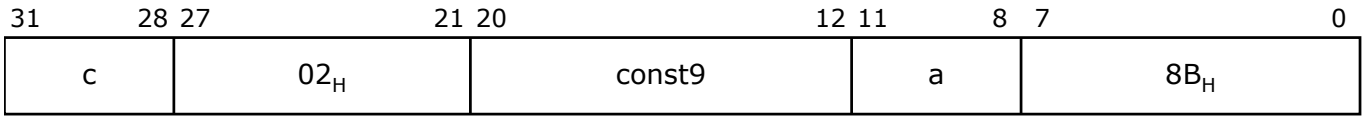
Add with Saturation

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 32-bit signed integers, with saturation on signed overflow. The value const9 is sign-extended before the addition is performed.

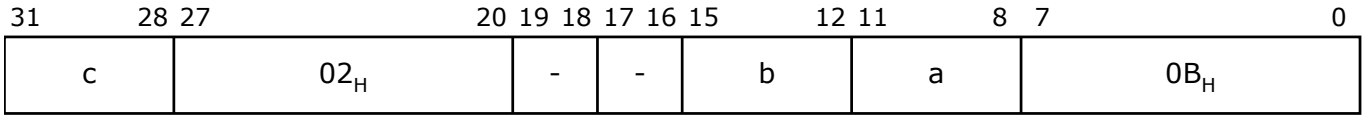
Add the contents of data register D[b] to the contents of data register D[a] and put the result in data register D[a]. The operands are treated as 32-bit signed integers, with saturation on signed overflow.

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



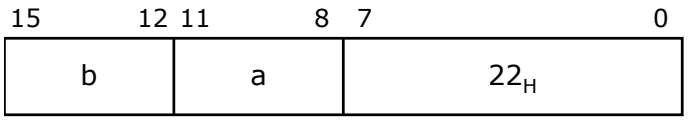

```
result = D[a] + sign_ext(const9);
D[c] = ssov(result, 32);
```

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




```
result = D[a] + D[b];
D[c] = ssov(result, 32);
```

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




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

Status Flags

C	Not set by this instruction.
V	overflow = (result > 7FFFFFFF$_H$) OR (result < -80000000$_H$); 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, #126
```

```
adds    d3, d1, d2
adds    d3, d1
```

See Also

ADD, ADDC, ADDI, ADDIH, ADDS.U, ADDX

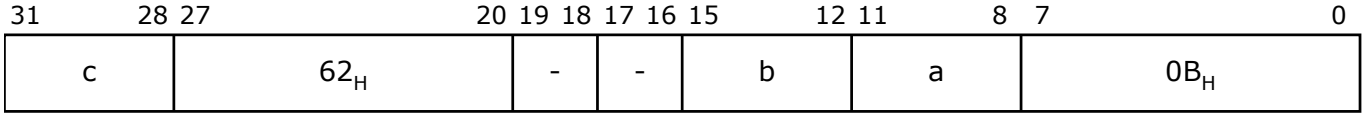
3.1.20 ADDS.H

Add 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. 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)};
```

Status Flags

C	Not set by this instruction.

V	ov_halfword1 = (result_halfword1 > $7FFF_H$) OR (result_halfword1 < -8000_H);
	ov_halfword0 = (result_halfword0 > $7FFF_H$) OR (result_halfword0 < -8000_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[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

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

See Also

ADD.B, ADD.H, ADDS.HU

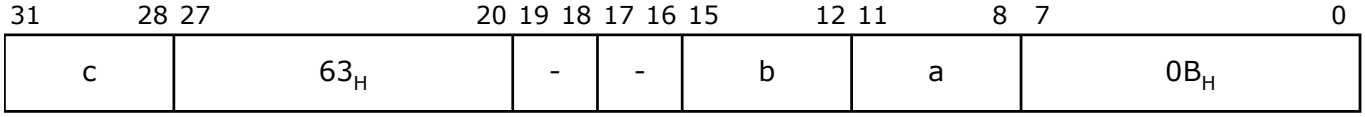
3.1.21 ADDS.HU

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 unsigned overflow. The overflow (PSW.V) and advance overflow (PSW.AV) conditions are calculated for each half-word of the packed quantity.

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 this instruction.

V	ov_halfword1 = (result_halfword1 > $FFFF_H$);
	ov_halfword0 = (result_halfword0 > $FFFF_H$);
	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[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

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

See Also

ADD.B, ADD.H, ADDS.H

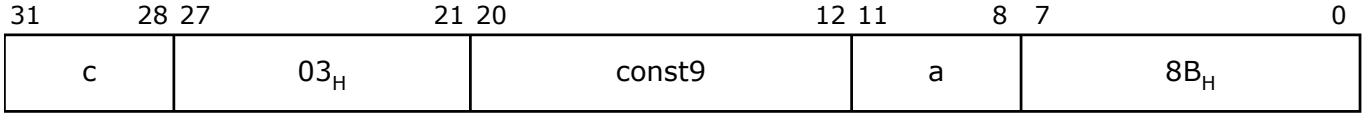
3.1.22 ADDS.U

Add Unsigned with Saturation

Description

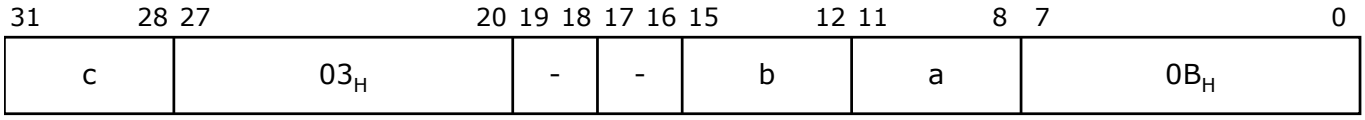
Add the contents of data register D[a] to the contents of either data register D[b] (instruction format RR) or const9 (instruction format RC) and put the result in data register D[c]. The operands are treated as 32-bit unsigned integers, with saturation on unsigned overflow. The const9 value is sign-extended.

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$_H$); 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, #126
adds.u    d3, d1, d2
```

See Also

ADD, ADDC, ADDI, ADDIH, ADDS, ADDX

3.1.23 ADDSC.A

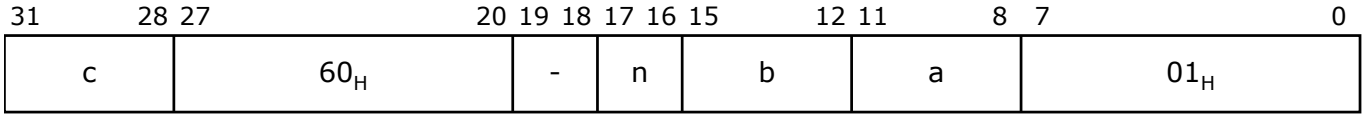
Add Scaled Index to Address

Description

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].

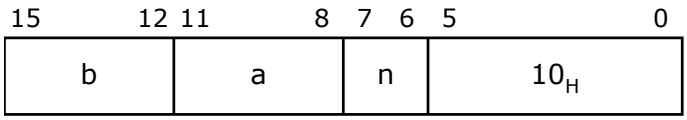
Left-shift the contents of data register D[15] by the amount specified by n, where n can be 0, 1, 2, or 3. Add that value to the contents of address register A[b] and put the result in address register A[a].

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




```
A[c] = A[b] + (D[a] << n);
```

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




```
A[a] = A[b] + (D[15] << n);
```

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

```
addsc.a    a3, a4, d2, #2
addsc.a    a3, a4, d15, #2
```

See Also

ADD.A, ADDIH.A, ADDSC.AT, SUB.A

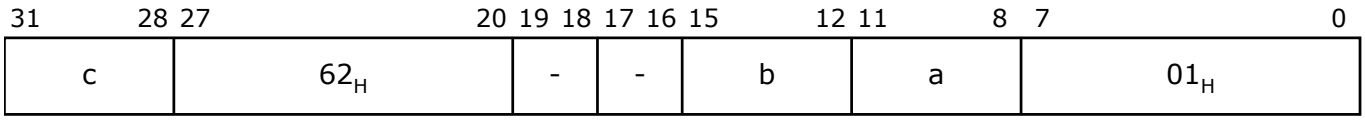
3.1.24 ADDSC.AT

Add Bit-Scaled Index to Address

Description

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].

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




```
A[c] = (A[b] + (D[a] >> 3)) & 32'hFFFFFFFC;
```

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

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

See Also

ADD.A, ADDIH.A, ADDSC.A, SUB.A

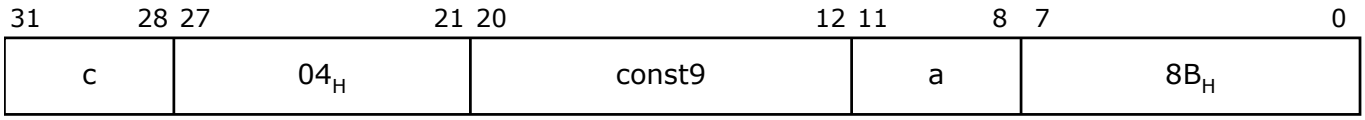
3.1.25 ADDX

Add Extended

Description

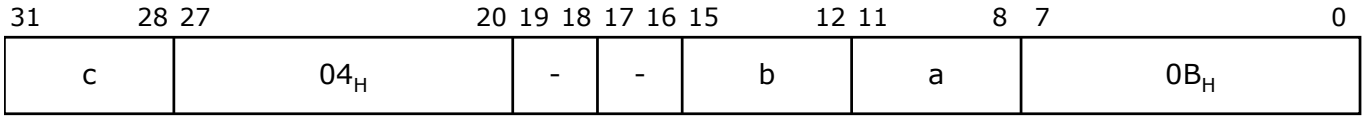
Add the contents of data register D[a] to the contents of either data register D[b] (instruction format RR) or const9 (instruction format RC) and put the result in data register D[c]. The operands are treated as 32-bit signed integers. The const9 value is sign-extended before the addition is performed.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

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




```
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)




```
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$_H$) OR (result < -80000000$_H$); 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, #126
addx    d3, d1, d2
```

See Also

ADD, ADDC, ADDI, ADDIH, ADDS, ADDS.U, SUBX

3.1.26 AND

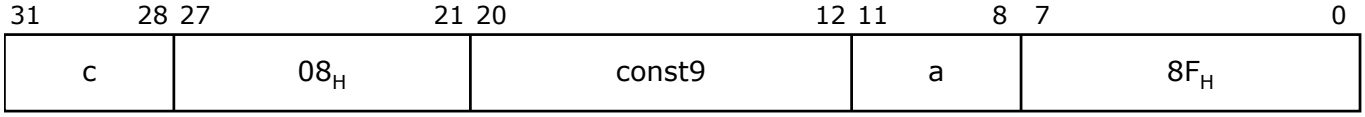
Bitwise AND

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.

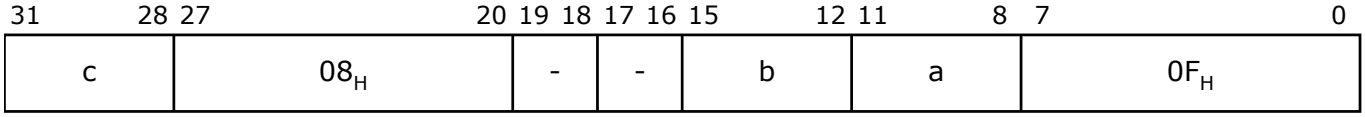
Compute the bitwise AND of the contents of either data register D[a] (instruction format SRR) or D[15] (instruction format SC) and the contents of either data register D[b] (format SRR) or const8 (format SC), and put the result in data register D[a] (format SRR) or D[15] (format SC). The const8 value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

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



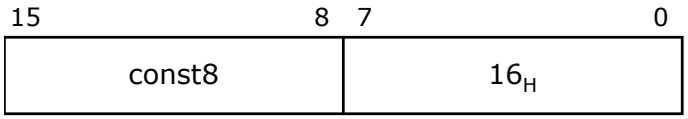

```
D[c] = D[a] & zero_ext(const9);
```

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



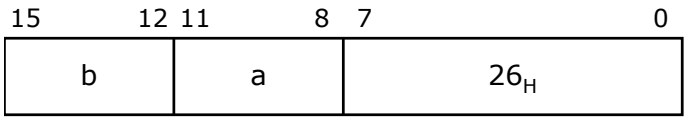

```
D[c] = D[a] & D[b];
```

AND D[15], const8 (SC)




```
D[15] = D[15] & zero_ext(const8);
```

AND 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

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

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

See Also

ANDN, NAND, NOR, NOT, OR, ORN, XNOR, XOR

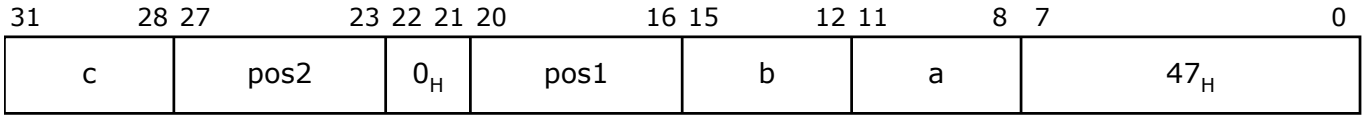
3.1.27 AND.AND.T

Accumulating Bit Logical AND-AND

Description

Compute the logical AND 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)




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

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.and.t    d3, d1, #4, d2, #9
```

See Also

AND.ANDN.T, AND.NOR.T, AND.OR.T, 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

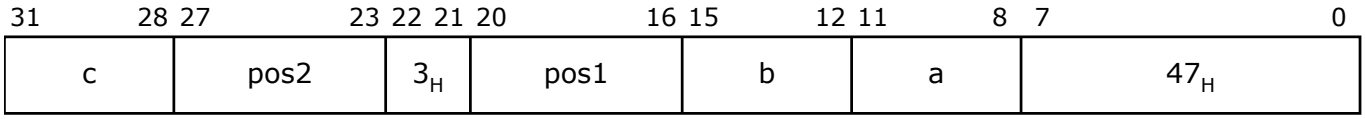
3.1.28 AND.ANDN.T

Accumulating Bit Logical AND-AND-Not

Description

Compute the logical ANDN 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.ANDN.T D[c], D[a], pos1, D[b], pos2 (BIT)




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

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.andn.t    d3, d1, #6, d2, #15
```

See Also

AND.AND.T, AND.NOR.T, AND.OR.T, 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

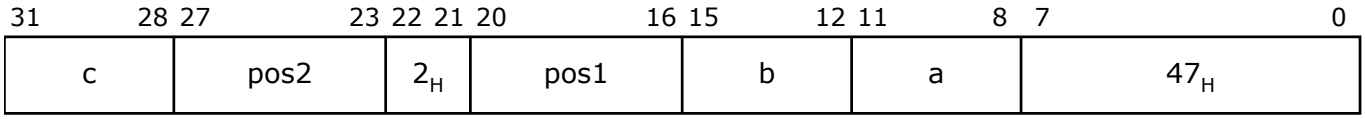
3.1.29 AND.NOR.T

Accumulating Bit Logical AND-NOR

Description

Compute the logical NOR 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.NOR.T D[c], D[a], pos1, D[b], pos2 (BIT)




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

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.nor.t    d3, d1, #5, d2, #9
```

See Also

AND.AND.T, AND.ANDN.T, AND.OR.T, 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

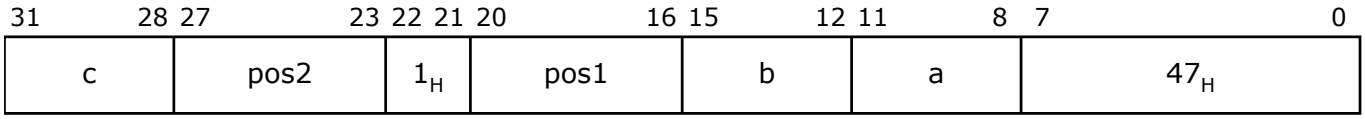
3.1.30 AND.OR.T

Accumulating Bit Logical AND-OR

Description

Compute the logical 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.OR.T D[c], D[a], pos1, D[b], pos2 (BIT)




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

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.or.t    d3, d1, #4, d2, #6
```

See Also

AND.AND.T, AND.ANDN.T, AND.NOR.T, 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

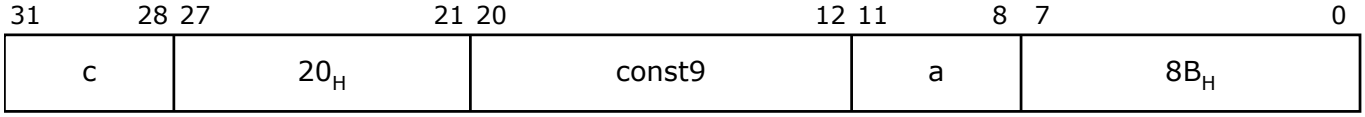
3.1.31 AND.EQ

Equal, AND Accumulating

Description

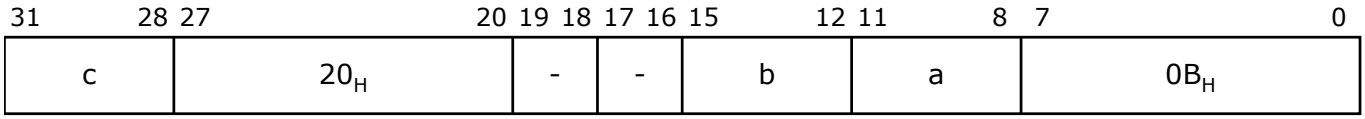
Compute the logical AND of D[c][0] and the boolean result of the equality comparison operation on the contents of data register D[a] and either data register D[b] (instruction format RR) or const9 (instruction RC). Put the result in D[c][0]. All other bits in D[c] are unchanged. The const9 value is sign-extended.

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




```
D[c] = {D[c][31:1], D[c][0] AND (D[a] == sign_ext(const9))};
```

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




```
D[c] = {D[c][31:1], D[c][0] 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, #126
and.eq    d3, d1, d2
```

See Also

OR.EQ, XOR.EQ

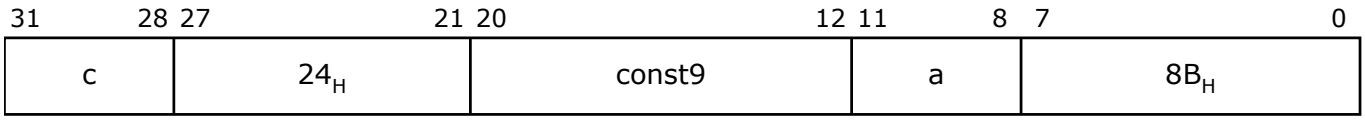
3.1.32 AND.GE

Greater Than or Equal, AND Accumulating

Description

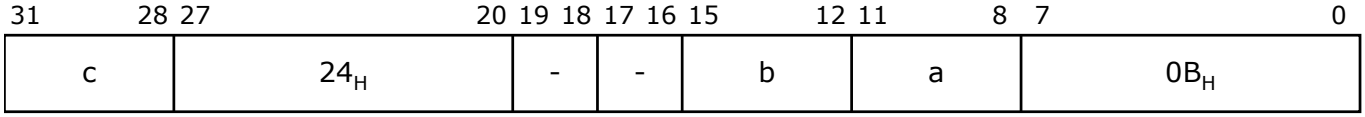
Calculate the logical AND of D[c][0] and the boolean result of the GE operation on the contents of data register D[a] and either data register D[b] (instruction format RR) or const9 (instruction format RC). Put the result in D[c][0]. All other bits in D[c] are unchanged. D[a] and D[b] are treated as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

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])};
```

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

See Also

AND.GE.U, OR.GE, OR.GE.U, XOR.GE, XOR.GE.U

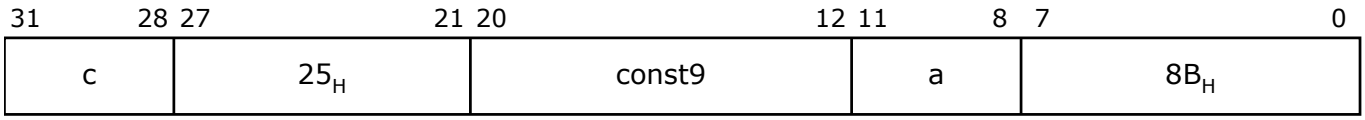
3.1.33 AND.GE.U

Greater Than or Equal, AND Accumulating Unsigned

Description

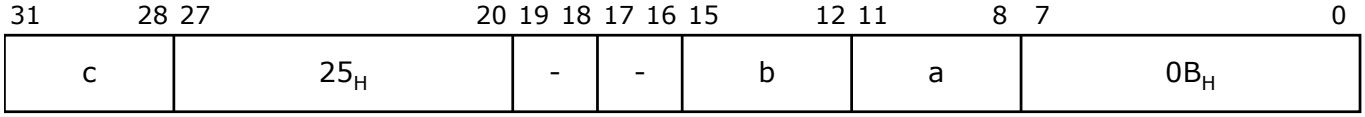
Calculate the logical AND of D[c][0] and the boolean result of the GE.U operation on the contents of data register D[a] and either data register D[b] (instruction format RR) or const9 (instruction format RC). Put the result in D[c][0]. All other bits in D[c] are unchanged. D[a] and D[b] are treated as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

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
```

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

See Also

AND.GE, OR.GE, OR.GE.U, XOR.GE, XOR.GE.U

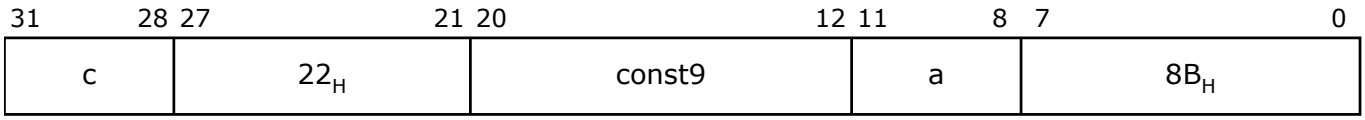
3.1.34 AND.LT

Less Than, AND Accumulating

Description

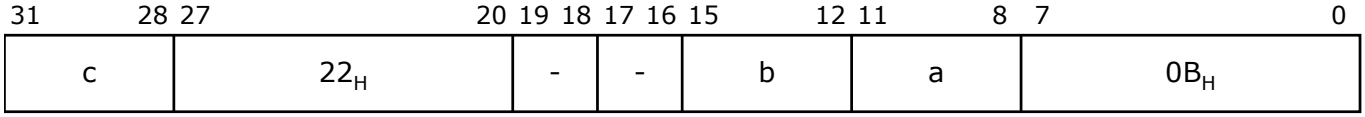
Calculate the logical AND of D[c][0] and the boolean result of the LT operation on the contents of data register D[a] and either data register D[b] (instruction format RR) or const9 (instruction format RC). Put the result in D[c][0]. All other bits in D[c] are unchanged. D[a] and D[b] are treated as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

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])};
```

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

See Also

AND.LT.U, OR.LT, OR.LT.U, XOR.LT, XOR.LT.U

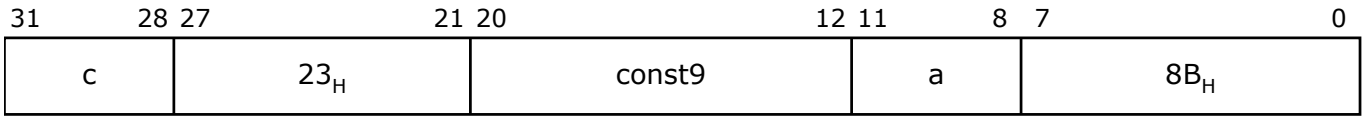
3.1.35 AND.LT.U

Less Than, AND Accumulating Unsigned

Description

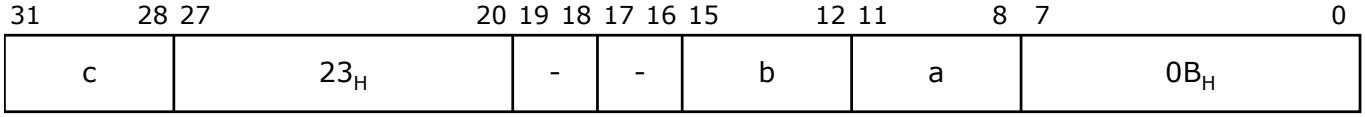
Calculate the logical AND of D[c][0] and the boolean result of the LT.U operation on the contents of data register D[a] and either data register D[b] (instruction format RR) or const9 (instruction format RC). Put the result in D[c][0]. All other bits in D[c] are unchanged. D[a] and D[b] are treated as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

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
```

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

See Also

AND.LT, OR.LT, OR.LT.U, XOR.LT, XOR.LT.U

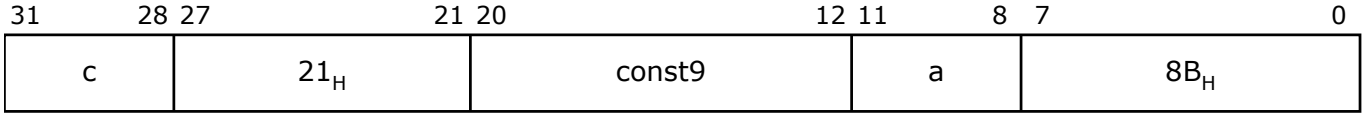
3.1.36 AND.NE

Not Equal, AND Accumulating

Description

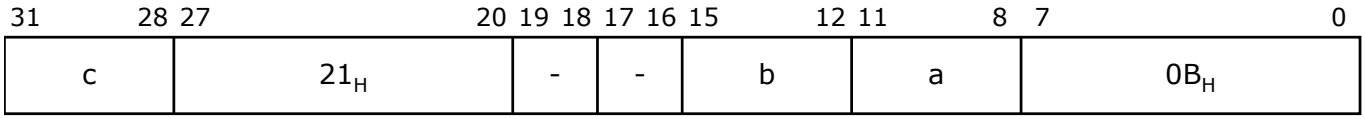
Calculate the logical AND of D[c][0] and the boolean result of the NE operation on the contents of data register D[a] and either data register D[b] (instruction format RR) or const9 (instruction format RC). Put the result in D[c][0]. All other bits in D[c] are unchanged. The const9 value is sign-extended.

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




```
D[c] = {D[c][31:1], D[c][0] AND (D[a] != sign_ext(const9))};
```

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




```
D[c] = {D[c][31:1], D[c][0] 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.ne    d3, d2, #126
and.ne    d3, d1, d2
```

See Also

OR.NE, XOR.NE

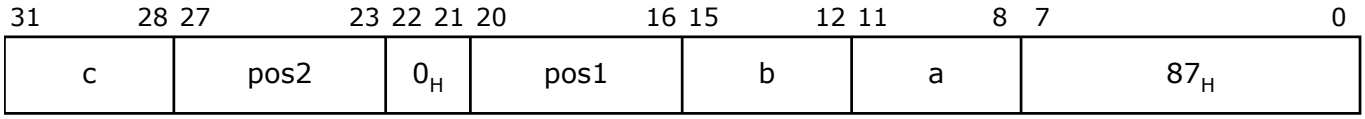
3.1.37 AND.T

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

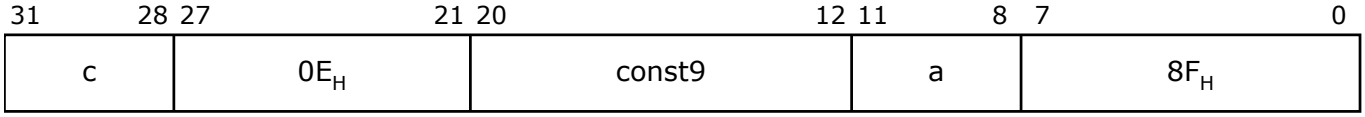
3.1.38 ANDN

Bitwise AND-Not

Description

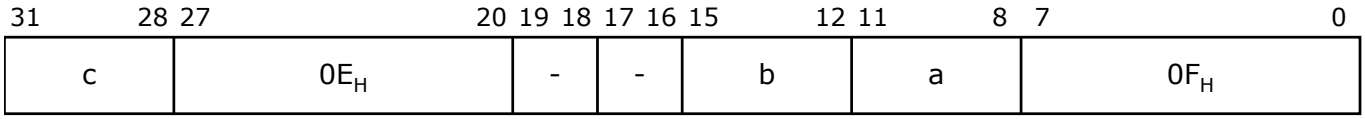
Compute the bitwise AND of the contents of data register D[a] and the ones complement of the contents of either data register D[b] (instruction format RR) or const9 (instruction format RC). Put the result in data register D[c]. The const9 value is zero-extended to 32-bit.

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




```
D[c] = D[a] & ~zero_ext(const9);
```

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




```
D[c] = 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

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

See Also

AND, NAND, NOR, NOT, OR, ORN, XNOR, XOR

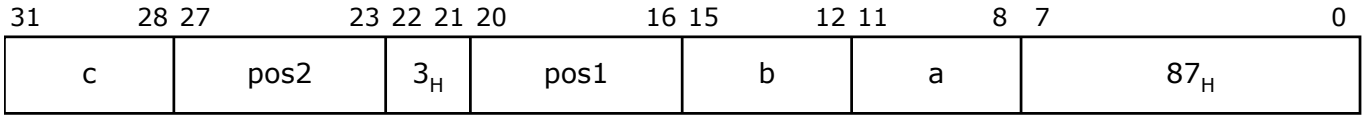
3.1.39 ANDN.T

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

3.1.40 BISR

Begin Interrupt Service Routine

Description

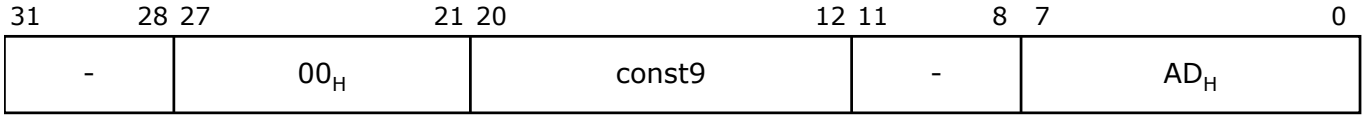
***Note**: Execution Mode - This instruction can only be executed in the following mode: Supervisor.*

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).

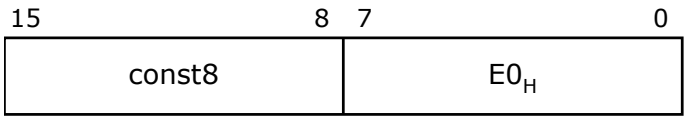
This instruction is intended to be one of the first executed instructions in an interrupt routine. If the interrupt routine has not altered the lower context, the saved lower context is from the interrupted task. If a BISR instruction is issued at the beginning of an interrupt, then an RSLCX instruction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

BISR const9 (RC)




```
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];
ICR.IE = 1;
ICR.CCPN = const9[7:0];
if (tmp_FCX == LCX) trap(FCD);
```

BISR const8 (SC)



```
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];
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, LDUCX, STLCX, STUCX, SVLCX, RET, RFE, RSLCX, RSTV

3.1.41 BMERGE

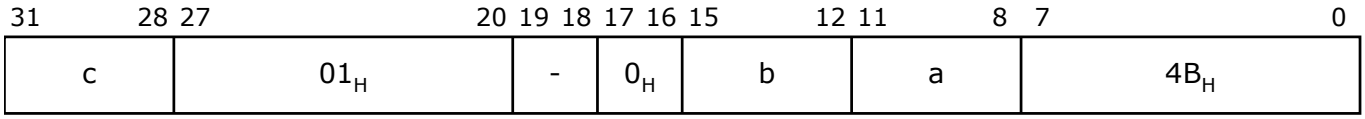
Bit Merge

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, SHUFFLE

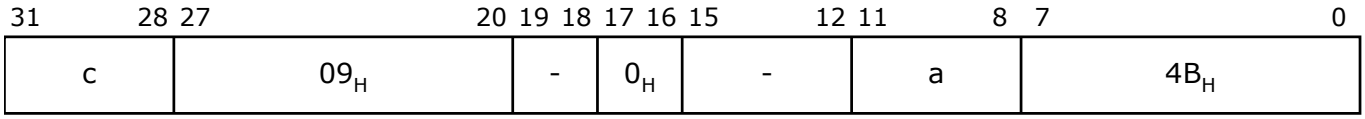
3.1.42 BSPLIT

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)




```
E[c][63:48] = 0000H;
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] = 0000H;
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, SHUFFLE

3.1.43 CACHEA.I

Cache Address, Invalidate

Description

***Note**: Execution Mode - This instruction can only be executed in the following mode: Supervisor.*

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

The effective address undergoes standard protection checks and requires write permission

If the cache line containing the byte memory location specified by the effective address 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.

***Note**: If the virtualization extension is present and enabled the instruction will only operate on data cache entries that belong to the virtual machine (VM) number defined in VCON2.VMN. This ensures that a VM can only operate on its own data cache entries. This also allows the Hypervisor to selectively operate on a particular VM cache lines.*

***Note**: If the virtualization extension is not present or disabled the instruction operates on effective address alone.*

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

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

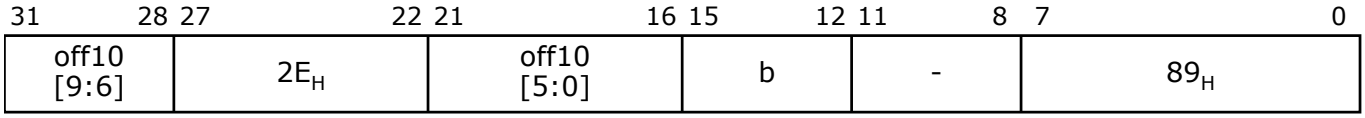
```
EA = A[b];
if (TCCON.VIRT==1 AND VCON0.EN==1) then
    cache_address_ivld(EA, VCON2.VMN);
else
    cache_address_ivld(EA);
A[b] = EA + sign_ext(off10);
```

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

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

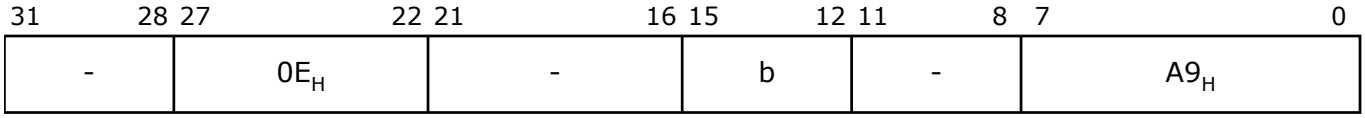
```
EA = A[b] + sign_ext(off10);
if (TCCON.VIRT==1 AND VCON0.EN==1) then
    cache_address_ivld(EA, VCON2.VMN);
else
    cache_address_ivld(EA);
A[b] = EA;
```

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



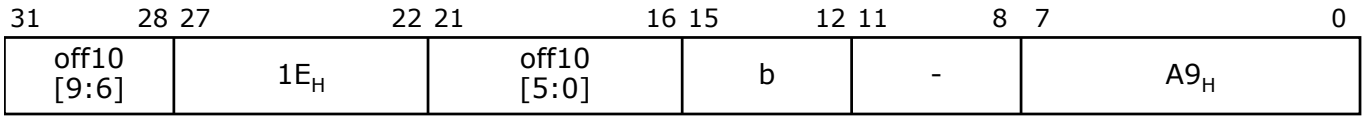

```
EA = A[b] + sign_ext(off10);
if (TCCON.VIRT==1 AND VCON0.EN==1) then
    cache_address_ivld(EA, VCON2.VMN);
else
    cache_address_ivld(EA);
```

CACHEA.I 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;
if (TCCON.VIRT==1 AND VCON0.EN==1) then
    cache_address_ivld(EA, VCON2.VMN);
else
    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)




```
index = zero_ext(A[b+1][15:0]);
length = zero_ext(A[b+1][31:16]);
EA = A[b] + index;
if (TCCON.VIRT==1 AND VCON0.EN==1) then
    cache_address_ivld(EA, VCON2.VMN);
else
    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]};
```

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+r]
cachea.i    [p4+r]
cachea.i    [a4/a5+c]4
cachea.i    [p4+c]4
```

See Also

CACHEA.W, CACHEA.WI, CACHEI.I, CACHEI.W, CACHEI.WI, CACHEA.I.VM, CACHEA.W.VM, CACHEA.WI.VM, CACHEI.I.VM, CACHEI.W.VM, CACHEI.WI.VM

3.1.44 CACHEA.W

Cache Address, Writeback

Description

If the cache line containing the byte memory location specified by the effective address 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 effective address 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.

***Note**: If the virtualization extension is present and enabled the instruction will only operate on data cache entries that belong to the virtual machine (VM) number defined in VCON2.VMN. This ensures that a VM can only operate on its own data cache entries. This also allows the Hypervisor to selectively operate on a particular VM cache lines.*

***Note**: If the virtualization extension is not present or disabled the instruction operates on effective address alone.*

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

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

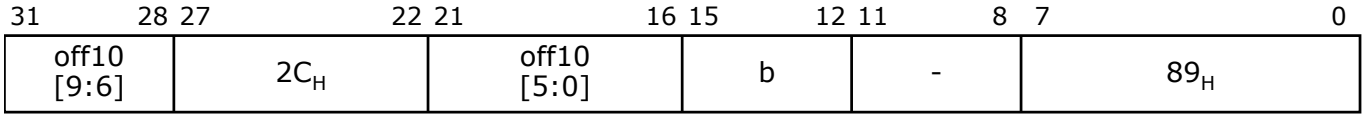
```
EA = A[b];
if (TCCON.VIRT==1 AND VCON0.EN==1) then
    cache_address_wb(EA, VCON2.VMN);
else
    cache_address_wb(EA);
A[b] = EA + sign_ext(off10);
```

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

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

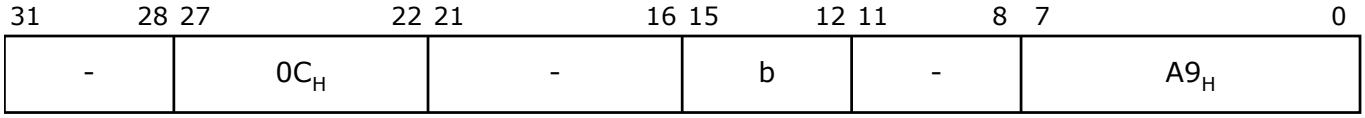
```
EA = A[b] + sign_ext(off10);
if (TCCON.VIRT==1 AND VCON0.EN==1) then
    cache_address_wb(EA, VCON2.VMN);
else
    cache_address_wb(EA);
A[b] = EA;
```

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



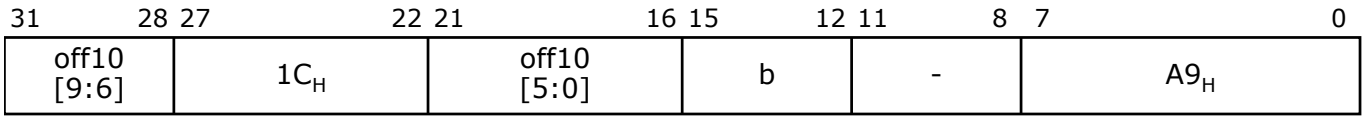

```
EA = A[b] + sign_ext(off10);
if (TCCON.VIRT==1 AND VCON0.EN==1) then
    cache_address_wb(EA, VCON2.VMN);
else
    cache_address_wb(EA);
```

CACHEA.W 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;
if (TCCON.VIRT==1 AND VCON0.EN==1) then
    cache_address_wb(EA, VCON2.VMN);
else
    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)




```
index = zero_ext(A[b+1][15:0]);
length = zero_ext(A[b+1][31:16]);
EA = A[b] + index;
if (TCCON.VIRT==1 AND VCON0.EN==1) then
    cache_address_wb(EA, VCON2.VMN);
else
    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]};
```

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+r]
cachea.w    [p4+r]
cachea.w    [a4/a5+c]4
cachea.w    [p4+c]4
```

See Also

CACHEA.I, CACHEA.WI, CACHEI.I, CACHEI.W, CACHEI.WI, CACHEA.W.VM, CACHEA.WI.VM, CACHEI.I.VM, CACHEI.W.VM, CACHEI.WI.VM

3.1.45 CACHEA.WI

Cache Address, Writeback and Invalidate

Description

If the cache line containing the byte memory location specified by the effective address 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 effective address 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.

***Note**: If the virtualization extension is present and enabled the instruction will only operate on data cache entries that belong to the virtual machine (VM) number defined in VCON2.VMN. This ensures that a VM can only operate on its own data cache entries. This also allows the Hypervisor to selectively operate on a particular VM cache lines.*

***Note**: If the virtualization extension is not present or disabled the instruction operates on effective address alone.*

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

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

```
EA = A[b];
if (TCCON.VIRT==1 AND VCON0.EN==1) then
    cache_address_wi(EA, VCON2.VMN);
else
    cache_address_wi(EA);
A[b] = EA + sign_ext(off10);
```

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

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

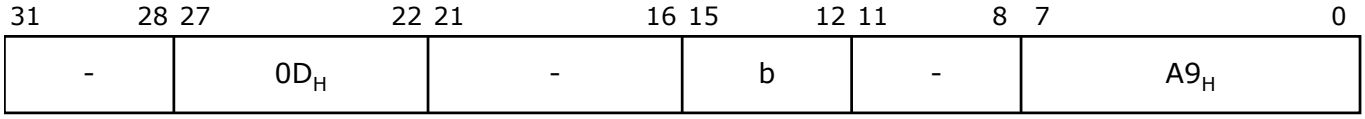
```
EA = A[b] + sign_ext(off10);
if (TCCON.VIRT==1 AND VCON0.EN==1) then
    cache_address_wi(EA, VCON2.VMN);
else
    cache_address_wi(EA);
A[b] = EA;
```

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]	$2D_H$	off10 [5:0]	b	-	89_H

```
EA = A[b] + sign_ext(off10);
if (TCCON.VIRT==1 AND VCON0.EN==1) then
    cache_address_wi(EA, VCON2.VMN);
else
    cache_address_wi(EA);
```

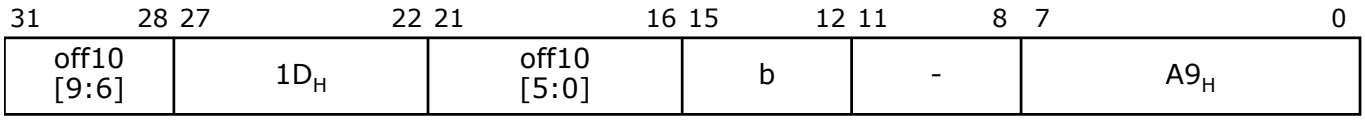
CACHEA.WI P[b] (BO) (Bit-reverse Addressing Mode)



31 28	27 22	21 16	15 12	11 8	7 0
-	$0D_H$	-	b	-	$A9_H$

```
index = zero_ext(A[b+1][15:0]);
incr = zero_ext(A[b+1][31:16]);
EA = A[b] + index;
if (TCCON.VIRT==1 AND VCON0.EN==1) then
    cache_address_wi(EA, VCON2.VMN);
else
    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]	$1D_H$	off10 [5:0]	b	-	$A9_H$

```
index = zero_ext(A[b+1][15:0]);
length = zero_ext(A[b+1][31:16]);
EA = A[b] + index;
if (TCCON.VIRT==1 AND VCON0.EN==1) then
    cache_address_wi(EA, VCON2.VMN);
else
    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]};
```

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+r]
cachea.wi    [p4+r]
cachea.wi    [a4/a5+c]4
cachea.wi    [p4+c]4
```

See Also

CACHEA.I, CACHEA.W, CACHEI.I, CACHEI.W, CACHEI.WI, CACHEA.W.VM, CACHEA.WI.VM, CACHEI.I.VM, CACHEI.W.VM, CACHEI.WI.VM

3.1.46 CACHEI.I

Cache Index, Invalidate

Description

***Note**: Execution Mode - This instruction can only be executed in the following mode: Supervisor.*

If the cache line at the index and way specified by the address register A[b] is valid in the L1 data cache, then invalidate the line. Note that there is no writeback of any modified data in the cache line prior to invalidation.

The effective address undergoes standard protection checks and requires write permission. Address register updates associated with the addressing mode are performed regardless of the cache operation.

***Note**: If the virtualization extension is present and enabled the instruction will only operate on data cache entries that belong to the virtual machine (VM) number defined in VCON2.VMN. This ensures that a VM can only operate on its own data cache entries. This also allows the Hypervisor to selectively operate on a particular VM cache lines.*

***Note**: If the virtualization extension is not present or disabled the instruction operates on index and way alone.*

***Note**: The location of the cache index and cache way fields within the effective address is implementation dependent.*

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

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

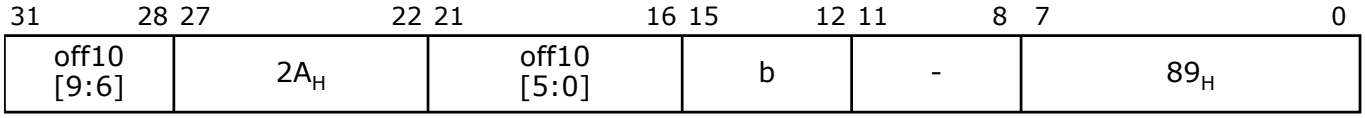
```
EA = A[b];
index_way = EA;
if (TCCON.VIRT==1 AND VCON0.EN==1) then
    cache_index_ivld(index_way, VCON2.VMN);
else
    cache_index_ivld(index_way);
A[b] = EA + sign_ext(off10);
```

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

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

```
EA = A[b] + sign_ext(off10);
index_way = EA;
if (TCCON.VIRT==1 AND VCON0.EN==1) then
    cache_index_ivld(index_way, VCON2.VMN);
else
    cache_index_ivld(index_way);
A[b] = EA;
```

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




```
EA = A[b] + sign_ext(off10);
index_way = EA;
if (TCCON.VIRT==1 AND VCON0.EN==1) then
    cache_index_ivld(index_way, VCON2.VMN);
else
    cache_index_ivld(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.i    [a3+]4
cachei.i    [+a3]4
cachei.i    [a3]4
```

See Also

CACHEA.I, CACHEA.W, CACHEA.WI, CACHEI.W, CACHEI.WI, CACHEA.W.VM, CACHEA.WI.VM, CACHEI.I.VM, CACHEI.W.VM, CACHEI.WI.VM

3.1.47 CACHEI.W

Cache Index, Writeback

Description

If any modified cache line at the memory index and way specified by the address register A[b] is valid 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 effective address undergoes standard protection checks. Address register updates associated with the addressing mode are performed regardless of the cache operation.

***Note**: If the virtualization extension is present and enabled the instruction will only operate on data cache entries that belong to the virtual machine (VM) number defined in VCON2.VMN. This ensures that a VM can only operate on its own data cache entries. This also allows the Hypervisor to selectively operate on a particular VM cache lines.*

***Note**: If the virtualization extension is not present or disabled the instruction operates on index and way alone.*

***Note**: The location of the cache index and cache way fields within the effective address is implementation dependent.*

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]	$0B_H$	off10 [5:0]	b	-	89_H

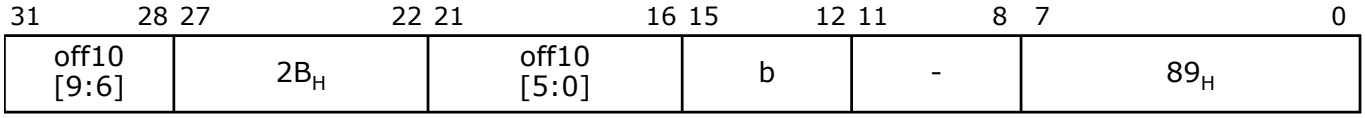
```
EA = A[b];
index_way = EA;
if (TCCON.VIRT==1 AND VCON0.EN==1) then
    cache_index_wb(index_way, VCON2.VMN);
else
    cache_index_wb(index_way);
A[b] = EA + sign_ext(off10);
```

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]	$1B_H$	off10 [5:0]	b	-	89_H

```
EA = A[b] + sign_ext(off10);
index_way = EA;
if (TCCON.VIRT==1 AND VCON0.EN==1) then
    cache_index_wb(index_way, VCON2.VMN);
else
    cache_index_wb(index_way);
A[b] = EA;
```

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




```
EA = A[b] + sign_ext(off10);
index_way = EA;
if (TCCON.VIRT==1 AND VCON0.EN==1) then
    cache_index_wb(index_way, VCON2.VMN);
else
    cache_index_wb(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.w    [a3+]4
cachei.w    [+a3]4
cachei.w    [a3]4
```

See Also

CACHEA.I, CACHEA.W, CACHEA.WI, CACHEI.I, CACHEI.WI, CACHEA.W.VM, CACHEA.WI.VM, CACHEI.I.VM, CACHEI.W.VM, CACHEI.WI.VM

3.1.48 CACHEI.WI

Cache Index, Writeback, Invalidate

Description

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

The effective address undergoes standard protection checks. Address register updates associated with the addressing mode are performed regardless of the cache operation.

***Note**: If the virtualization extension is present and enabled the instruction will only operate on data cache entries that belong to the virtual machine (VM) number defined in VCON2.VMN. This ensures that a VM can only operate on its own data cache entries. This also allows the Hypervisor to selectively operate on a particular VM cache lines.*

***Note**: If the virtualization extension is not present or disabled the instruction operates on index and way alone.*

***Note**: The location of the cache index and cache way fields within the effective address is implementation dependent.*

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]	$0F_H$	off10 [5:0]	b	-	89_H

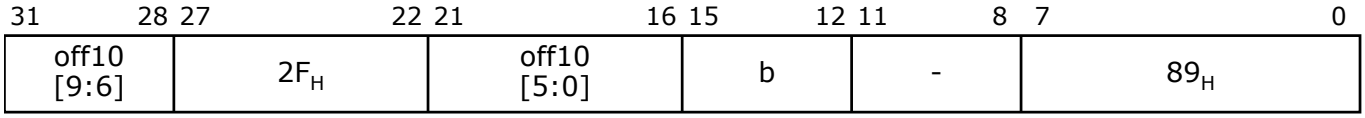
```
EA = A[b];
index_way = EA;
if (TCCON.VIRT==1 AND VCON0.EN==1) then
    cache_index_wi(index_way, VCON2.VMN);
else
    cache_index_wi(index_way);
A[b] = EA + sign_ext(off10);
```

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]	$1F_H$	off10 [5:0]	b	-	89_H

```
EA = A[b] + sign_ext(off10);
index_way = EA;
if (TCCON.VIRT==1 AND VCON0.EN==1) then
    cache_index_wi(index_way, VCON2.VMN);
else
    cache_index_wi(index_way);
A[b] = EA;
```

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




```
EA = A[b] + sign_ext(off10);
index_way = EA;
if (TCCON.VIRT==1 AND VCON0.EN==1) then
    cache_index_wi(index_way, VCON2.VMN);
else
    cache_index_wi(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, CACHEI.WI, CACHEA.W.VM, CACHEA.WI.VM, CACHEI.I.VM, CACHEI.W.VM, CACHEI.WI.VM

3.1.49 CADD

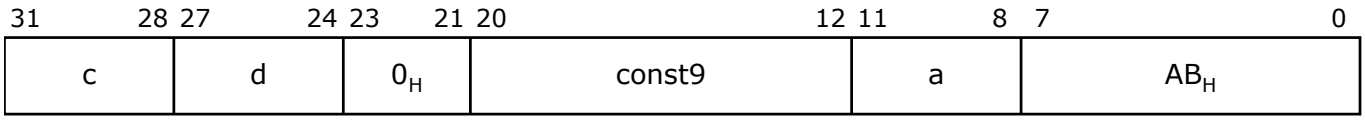
Conditional Add

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.

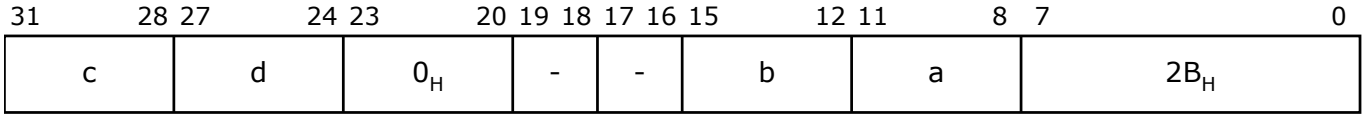
If the contents of data register D[15] are non-zero, then add contents of data register D[a] and the contents of const4 and put the result in data register D[a]; otherwise the contents of D[a] is unchanged. The const4 value is sign-extended.

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



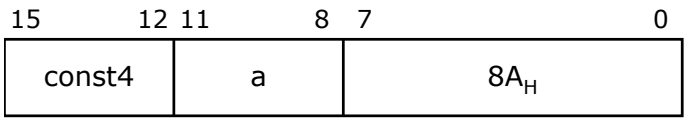

```
condition = (D[d] != 0);
result = condition ? (D[a] + sign_ext(const9)) : D[a];
D[c] = result[31:0];
```

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




```
condition = (D[d] != 0);
result = condition ? (D[a] + D[b]) : D[a];
D[c] = result[31:0];
```

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




```
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 > $7FFFFFFF_H$) OR (result < -80000000_H); 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, #126
cadd    d3, d4, d1, d2
cadd    d1, d15, #6
```

See Also

CADDN, CMOV, CMOVN, CSUB, CSUBN, SEL, SELN

3.1.50 CADDN

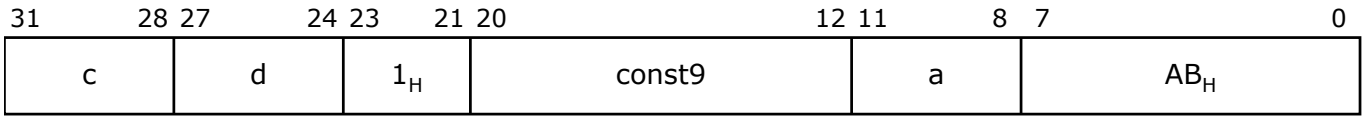
Conditional Add-Not

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.

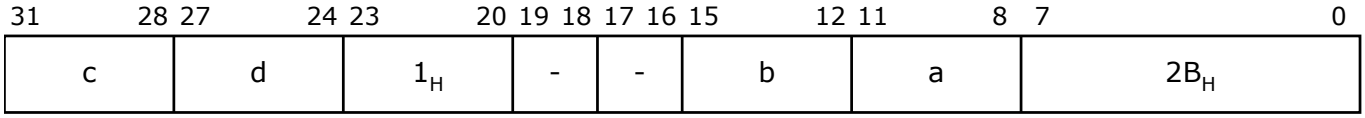
If the contents of data register D[15] are zero, then add the contents of data register D[a] and the contents of const4 and put the result in data register D[a]; otherwise the contents of D[a] is unchanged. The const4 value is sign-extended.

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



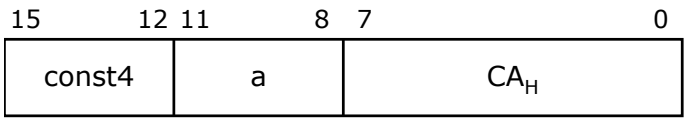

```
condition = (D[d] == 0);
result = condition ? (D[a] + sign_ext(const9)) : D[a];
D[c] = result[31:0];
```

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




```
condition = (D[d] == 0);
result = condition ? (D[a] + D[b]) : D[a];
D[c] = result[31:0];
```

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




```
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 > $7FFFFFFF_H$) OR (result < -80000000_H); 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, #126
caddn    d3, d4, d1, d2
```

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

See Also

CADD, CMOV, CMOVN, CSUB, CSUBN, SEL, SELN

3.1.51 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 ±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, the upper context registers are preserved and can be used to pass arguments to the called function (except for A[11] which is updated with the return address).*

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

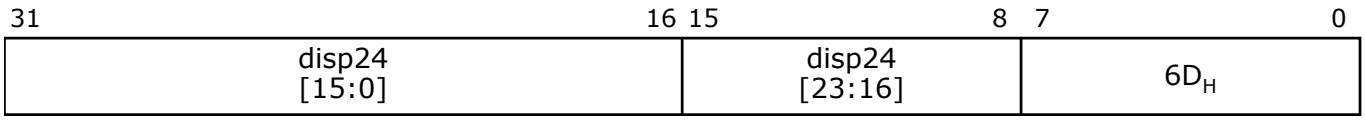
Add the value specified by disp8, multiplied by two and sign-extended, to the address of the CALL instruction, and jump to the resulting address. The target address range is ±256 bytes relative to the current PC.

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, the upper context registers are preserved and can be used to pass arguments to the called function (except for A[11] which is updated with the return address).*

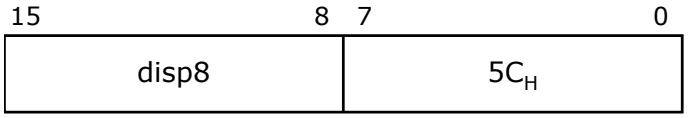
***Note**: When the PSW is saved, the CDE bit is forced to '1'.*

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);
csa_PSW = {PSW[31:16], PPRS[2], PSW[14], PPRS[1:0], PSW[11:0]}
PPRS = PSW.PRS
M(EA, 16-word) = {PCXI, csa_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);
csa_PSW = {PSW[31:16], PPRS[2], PSW[14], PPRS[1:0], PSW[11:0]}
PPRS = PSW.PRS
M(EA, 16-word) = {PCXI, csa_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	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

```
call    foobar
```

```
call    foobar
```

See Also

CALLA, CALLI, RET, FCALL, FRET

3.1.52 CALLA

Call Absolute

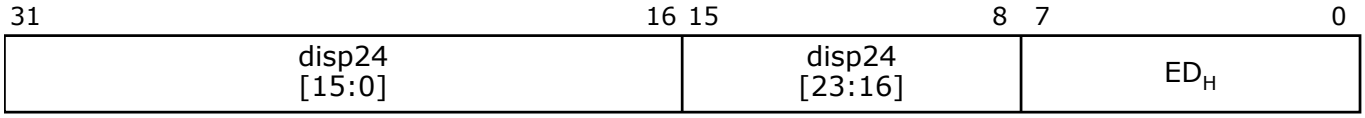
Description

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, the upper context registers are preserved and can be used to pass arguments to the called function (except for A[11] which is updated with the return address).*

***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);
csa_PSW = {PSW[31:16], PPRS[2], PSW[14], PPRS[1:0], PSW[11:0]}
PPRS = PSW.PRS
M(EA, 16-word) = {PCXI, csa_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	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

```
calla    foobar
```

See Also

CALL, CALLI, JL, JLA, RET, FCALLA, FRET

3.1.53 CALLI

Call Indirect

Description

Jump to the address specified by the contents of address register A[a]. The least-significant bit is always set to 0. 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, the upper context registers are preserved and can be used to pass arguments to the called function (except for A[11] which is updated with the return address).*

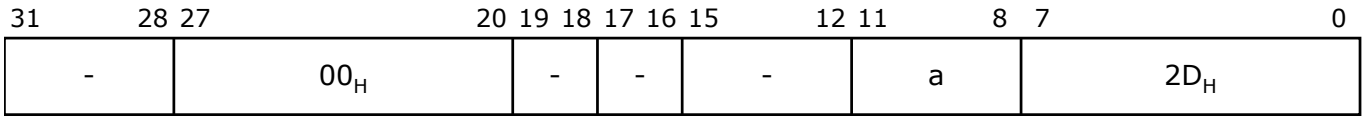
***Note**: When the PSW is saved, the CDE bit is forced to '1'.*

Jump to the address specified by the contents of address register A[a]. The least-significant bit is always set to 0. 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, the upper context registers are preserved and can be used to pass arguments to the called function (except for A[11] which is updated with the return address).*

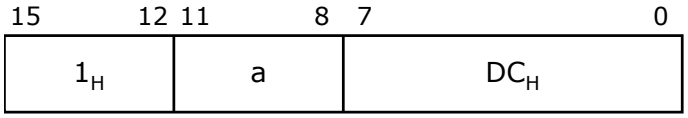
***Note**: When the PSW is saved, the CDE bit is forced to '1'.*

CALLI A[a] (RR)




```
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);
csa_PSW = {PSW[31:16], PPRS[2], PSW[14], PPRS[1:0], PSW[11:0]}
PPRS = PSW.PRS
M(EA, 16-word) = {PCXI, csa_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);
```

CALLI A[a] (SR)




```
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);
csa_PSW = {PSW[31:16], PPRS[2], PSW[14], PPRS[1:0], PSW[11:0]}
PPRS = PSW.PRS
M(EA, 16-word) = {PCXI, csa_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	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

```
calli    a2
```

See Also

CALL, CALLA, RET, FCALLI, FRET

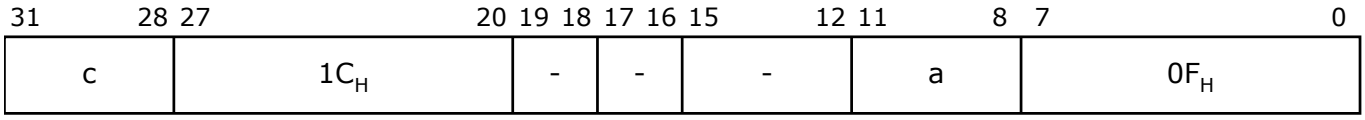
3.1.54 CLO

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, POPCNT.W

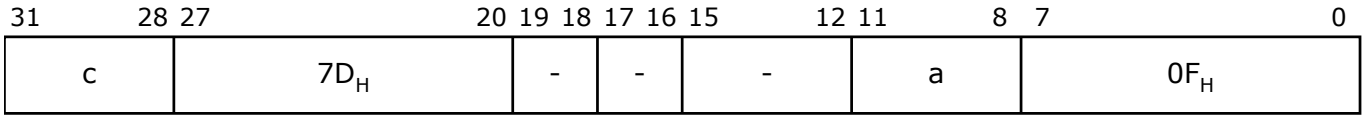
3.1.55 CLO.H

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, POPCNT.W

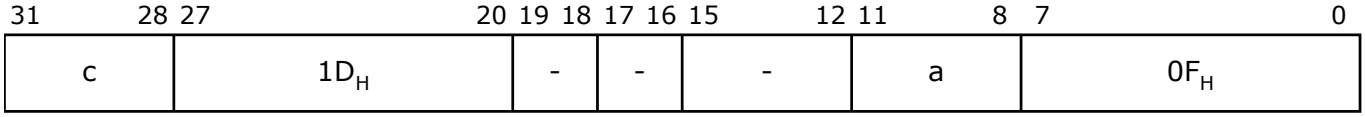
3.1.56 CLS

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, POPCNT.W

3.1.57 CLS.H

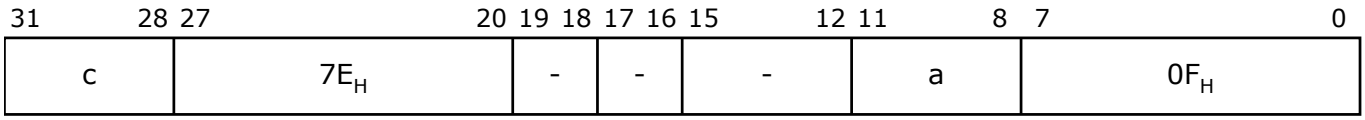
Count Leading Signs in Packed Half-words

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)




```
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, POPCNT.W

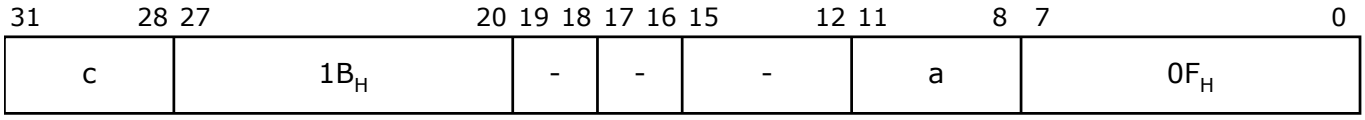
3.1.58 CLZ

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)




```
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, POPCNT.W

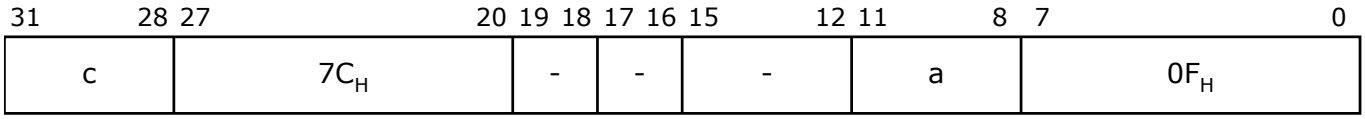
3.1.59 CLZ.H

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, POPCNT.W

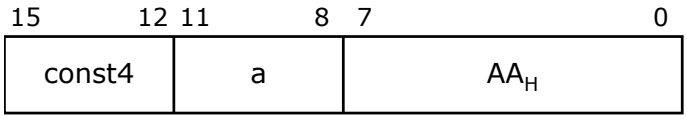
3.1.60 CMOV

Conditional Move (16-bit)

Description

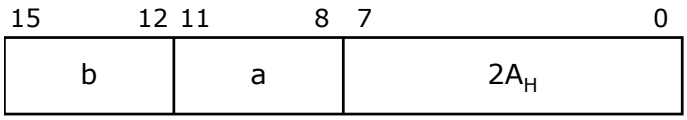
If the contents of data register D[15] are not zero, copy the contents of either data register D[b] (instruction format SRR) or const4 (instruction format SRC) to data register D[a]; otherwise the contents of D[a] is unchanged. The const4 value is sign-extended.

CMOV D[a], D[15], const4 (SRC)




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

CMOV D[a], D[15], D[b] (SRR)




```
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

```
cmov    d1, d15, #6
cmov    d1, d15, d2
```

See Also

CADD, CADDN, CMOVN, CSUB, CSUBN, SEL, SELN

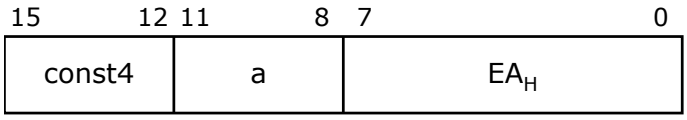
3.1.61 CMOVN

Conditional Move-Not (16-bit)

Description

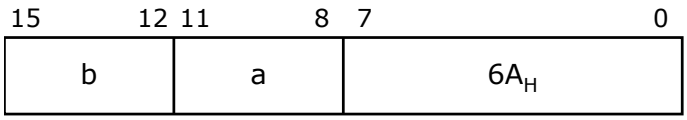
If the contents of data register D[15] are zero, copy the contents of either data register D[b] (instruction format SRR) or const4 (instruction format SRC) to data register D[a]; otherwise the contents of D[a] is unchanged. The const4 value is sign-extended to 32-bit.

CMOVN D[a], D[15], const4 (SRC)




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

CMOVN D[a], D[15], D[b] (SRR)




```
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, #6
cmovn    d1, d15, d2
```

See Also

CADD, CADDN, CMOV, CSUB, CSUBN, SEL, SELN

3.1.62 CMPSWAP.W

Compare and Swap

Description

The CMPSWAP.W instruction conditionally swaps the data register D[a] and the contents of the memory word specified by the effective address.

If the contents of the memory word specified by the effective address is equal to the contents of register D[a+1] then swap the contents of the memory word with the register D[a]. Register D[a] is unconditionally updated with the contents of the memory word specified by the effective address.

The M(EA, word) read and the M(EA, word) write are atomic.

CMPSWAP.W 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]	03_H	off10 [5:0]	b	a	49_H

```
EA = A[b];
tmp = M(EA, word);
M(EA, word) = (tmp == D[a+1]) ? D[a] : tmp;
D[a] = tmp;
A[b] = EA + sign_ext(off10);
```

CMPSWAP.W 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]	13_H	off10 [5:0]	b	a	49_H

```
EA = A[b] + sign_ext(off10);
tmp = M(EA, word);
M(EA, word) = (tmp == D[a+1]) ? D[a] : tmp;
D[a] = tmp;
A[b] = EA;
```

CMPSWAP.W 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]	23_H	off10 [5:0]	b	a	49_H

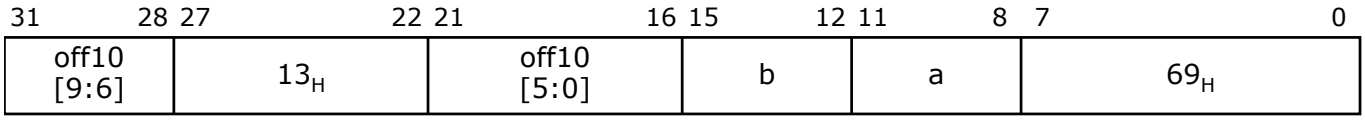
```
EA = A[b] + sign_ext(off10);
tmp = M(EA, word);
M(EA, word) = (tmp == D[a+1]) ? D[a] : tmp;
D[a] = tmp;
```

CMPSWAP.W P[b], E[a] (BO) (Bit-reverse Addressing Mode)

31 28	27 22	21 16	15 12	11 8	7 0
-	03_H	-	b	a	69_H

```
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) = (tmp == D[a+1]) ? D[a] : tmp;
D[a] = tmp;
new_index = reverse16(reverse16(index) + reverse16(incr));
A[b+1] = {incr[15:0], new_index[15:0]};
```

CMPSWAP.W 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;
tmp = M(EA, word);
M(EA, word) = (tmp == D[a+1]) ? D[a] : tmp;
D[a] = tmp;
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]};
```

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

```
cmpswap.w    [a0+]4, e0
cmpswap.w    [+a0]4, e0
cmpswap.w    [a0+4], e0
cmpswap.w    [a0/a1+r], e0
cmpswap.w    [p0+r], e0
cmpswap.w    [a0/a1+c], e0
cmpswap.w    [p0+c], e0
```

See Also

LDMST, ST.T, SWAP.W, SWAPMSK.W

3.1.63 CRC32.B

CRC32 Byte

Description

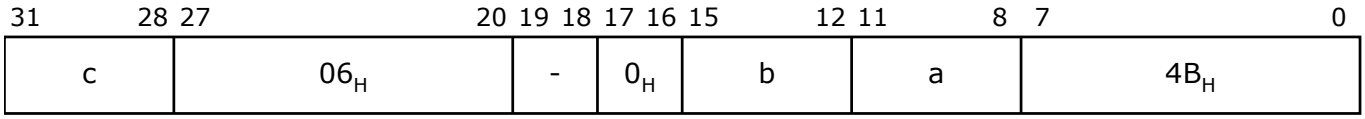
Calculate the CRC of 8 bits of input data from register D[a] and put the result in register D[c]. Register D[b] contains either an initial seed value, or the cumulative CRC result from a previous sequence of data.

The initial seed value of D[b] should be zero. By passing the CRC result to a subsequent CRC32 instruction, data larger than 8 bits can be processed.

The CRC polynomial used is the CRC-32 polynomial as defined in the IEEE 802.3 standard. The CRC result produced is bit-reversed and inverted as per the IEEE 802.3 standard.

The CRC algorithm treats input data as a stream of bits. Exactly 8 bits of input data in D[a][7:0] are processed. The least significant bit is processed first, and the most significant bit is processed last.

CRC32.B D[c], D[b], D[a] (RR)




```
A = reverse(D[a][7:0], 8);
crc_in = (A << 24) ^ ~reverse(D[b][31:0], 32);

D[c][31:0] = ~reverse(crc_div(crc_in, CRC_32_GENERATOR, 32, 8), 32);
```

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

```
;; Calculate CRC for string "12345"
mov.u        d1, #0x3231
addih        d1, d1, #0x3433
mov          d2, #0x35
mov          d0, #0x0
crc32l.w     d0, d0, d1
crc32.b      d0, d0, d2
; CRC result is stored in d0
```

See Also

CRC32B.W, CRC32L.W, CRCN, PARITY

3.1.64 CRC32B.W

CRC32 Word Big-Endian

Description

Calculate the CRC of register D[a] and put the result in register D[c]. Register D[b] contains either an initial seed value, or the cumulative CRC result from a previous sequence of data.

The initial seed value of D[b] should be zero. By passing the CRC result to a subsequent CRC32 instruction, data larger than 32 bits can be processed.

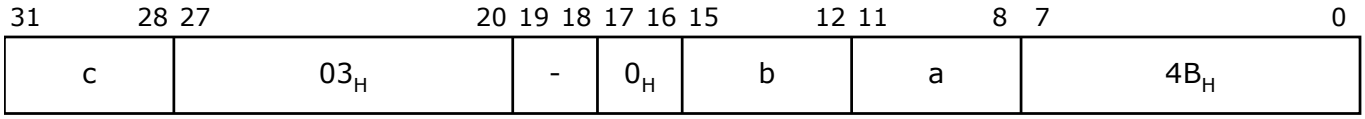
The CRC polynomial used is the CRC-32 polynomial as defined in the IEEE 802.3 standard. The CRC result is bit-reversed and inverted as per the IEEE 802.3 standard.

The CRC algorithm treats input data as a stream of bits. Exactly 32 bits of input data from D[a] are processed.

The four bytes in the input word are processed in big-endian order.

Within each 8-bit byte, the least significant bit is processed first and the most significant bit is processed last, as per the standard.

CRC32B.W D[c], D[b], D[a] (RR)




```
A = reverse(D[a][31:24], 8);
B = reverse(D[a][23:16], 8);
C = reverse(D[a][15:8], 8);
D = reverse(D[a][7:0], 8);
crc_in = {A,B,C,D} ^ ~reverse(D[b][31:0], 32);

D[c][31:0] = ~reverse(crc_div(crc_in, CRC_32_GENERATOR, 32, 32), 32);
```

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

```
;; Calculate CRC for string "12345678"
mov.u        d1, #0x3334
addih        d1, d1, #0x3132
mov.u        d2, #0x3738
addih        d2, d2, #0x3536
mov          d0, #0x0
crc32b.w     d0, d0, d1
crc32b.w     d0, d0, d2
; CRC result is stored in d0
```

See Also

CRC32L.W, CRC32.B, CRCN, PARITY

3.1.65 CRC32L.W

CRC32 Word Little-Endian

Description

Calculate the CRC of register D[a] and put the result in register D[c]. Register D[b] contains either an initial seed value, or the cumulative CRC result from a previous sequence of data.

The initial seed value of D[b] should be zero. By passing the CRC result to a subsequent CRC32 instruction, data larger than 32 bits can be processed.

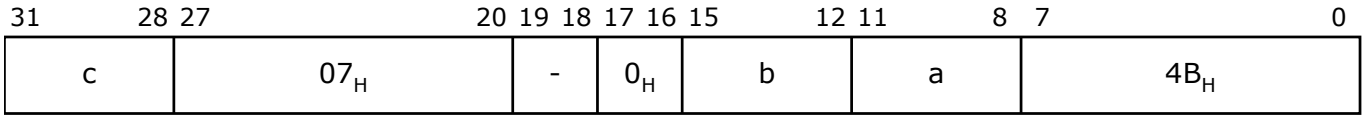
The CRC polynomial used is the CRC-32 polynomial as defined in the IEEE 802.3 standard. The CRC result is bit-reversed and inverted as per the IEEE 802.3 standard.

The CRC algorithm treats input data as a stream of bits. Exactly 32 bits of input data from D[a] are processed.

The four bytes in the input word are processed in little-endian order.

Within each 8-bit byte, the least significant bit is processed first and the most significant bit is processed last, as per the standard.

CRC32L.W D[c], D[b], D[a] (RR)




```
A = reverse(D[a][7:0], 8);
B = reverse(D[a][15:8], 8);
C = reverse(D[a][23:16], 8);
D = reverse(D[a][31:24], 8);
crc_in = {A,B,C,D} ^ ~reverse(D[b][31:0], 32);

D[c][31:0] = ~reverse(crc_div(crc_in, CRC_32_GENERATOR, 32, 32), 32);
```

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

```
;; Calculate CRC for string "12345678"
mov.u        d1, #0x3334
addih        d1, d1, #0x3132
mov.u        d2, #0x3738
addih        d2, d2, #0x3536
mov          d0, #0x0
crc32l.w     d0, d0, d1
crc32l.w     d0, d0, d2
; CRC result is stored in d0
```

See Also

CRC32.B, CRC32B.W, CRCN, PARITY

3.1.66 CRCN

User-Defined CRC

Description

Calculate the CRC value for 1 to 8 bits of input data using a user-defined CRC algorithm with a CRC width from 1 up to 16 bits. Register D[d] contains an initial seed value. The register D[a] specifies all parameters of the CRC algorithm. Register D[b] contains the input data. The result is stored in register D[c].

The field D[a][15:12] contains N-1, where N is the CRC width in the range [1,16].

If the bit D[a][8] is set then input data bit order is treated as little-endian, otherwise input data bit order is treated as big-endian. For little-endian bit order, the bit D[b][0] is processed first. For big-endian bit order, the bit D[b][0] is processed last.

If the bit D[a][9] is set, a bitwise logical inversion is applied to both the result and seed values.

D[d][N-1:0] contains either an initial seed value, or the cumulative CRC result from a previous sequence of data. The seed value should be initialized according to the chosen CRC algorithm. Note that the result invert bit, D[a][9], must be taken into account when specifying the seed as the inversion is applied to the initial seed value as well as the final result.

The field D[a][16+N-1:16] encodes the coefficients of the generator polynomial. The coefficient of the most significant term is omitted as it must be 1 by definition. D[a][16+N-1:16] contains the remaining coefficients, stored with the highest term in D[a][16+N-1] and lowest term in D[a][16]. For example, the CRC-8-SAE J1850 polynomial is encoded as $1D_H$.

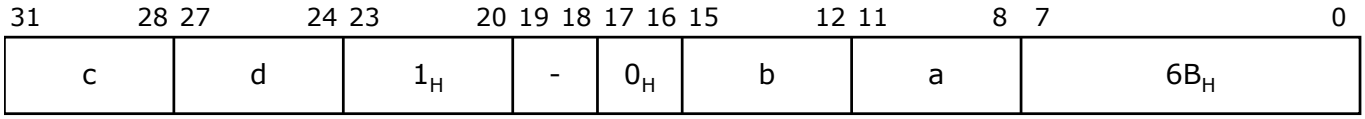
All unused bits in the register D[a] must be zero.

The field D[a][2:0] contains M-1, where M is the input data width in the range [1,8].

The field D[b][M-1:0] contains the input data. All other bits in D[b] are ignored.

The CRC result is stored in the register D[c]. D[c][N-1:0] contains the CRC result and all other bits are set to zero. The result register can be used as the seed input for a subsequent CRCN instruction. By chaining CRCN instructions in this way data larger than 8 bits in length can be processed.

CRCN D[c], D[d], D[a], D[b] (RRR)




```
// CRC width
N = D[a][15:12] + 1;
GEN = D[a][16+N-1:16];
{INV, LITTLE_E} = D[a][9:8];

// Data width
M = D[a][2:0] + 1;

data = D[b][M-1:0];
if (LITTLE_E) then {
    data = reverse(data, M);
}
seed = D[d][N-1:0];
if (INV) then {
    seed = ~seed;
}
// Combine seed and data
```

```
if (M > N) then {
    crc_in[N-1:0] = (data >> (M-N)) ^ seed;
} else {
    crc_in[N-1:0] = (data << (N-M)) ^ seed;
}
result = crc_div(crc_in, GEN, N, M);

if (INV) then {
    result = ~result;
}
D[c][N-1:0] = result;
D[c][31:N]  = 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

```
; CRC-15-CAN     - Size = 15-bit, Generator = 0x4599
;                - Seed = 0x7FFF, Invert = 1
;                - Input Data = 0x345 (10-bit)
;                - Big-Endian Input Bit Order
mov.u  d0, #0x0000           ; Inverted Seed Value
mov.u  d1, #0xE200           ; CRC Width & Invert Flag
addih  d1, d1, #0x4599       ; Generator Polynomial

; First 2 bits
insert d1, d1, #0x1, #0, #3  ; Data Size Feild  (2 bits)
mov.u  d2, #0x03             ; 2 bit Input Data (0x3)
crcn   d0, d0, d1, d2

; Remaining 8 bits
insert d1, d1, #0x7, #0, #3  ; Data Size Feild  (8 bits)
mov.u  d2, #0x45             ; 8 bit Input Data (0x45)
crcn   d0, d0, d1, d2

; Check Result in d0
mov.u  d3, 0x6DF9            ; Check Value = 0x6DF9
jne    d0, d3, crc_mismatch
```

See Also

CRC32B.W, CRC32L.W, CRC32.B, PARITY

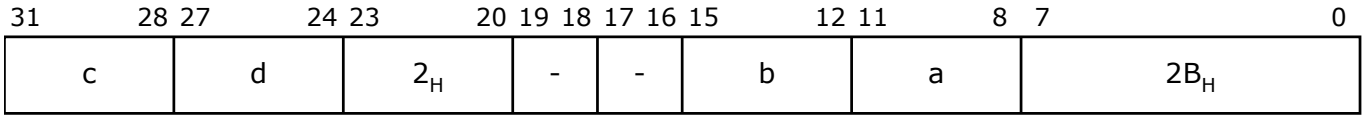
3.1.67 CSUB

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$_H$) OR (result < -80000000$_H$); 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, CMOVN, CSUBN, SEL, SELN

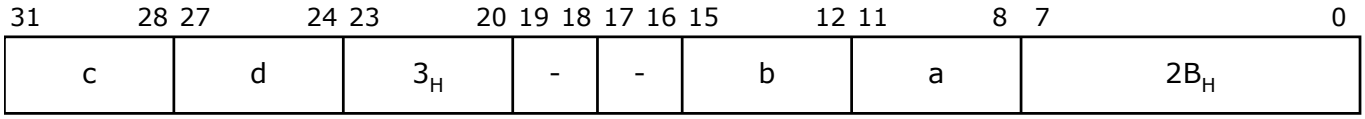
3.1.68 CSUBN

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$_H$) OR (result < -80000000$_H$); 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, CMOVN, CSUB, SEL, SELN

3.1.69 DEBUG

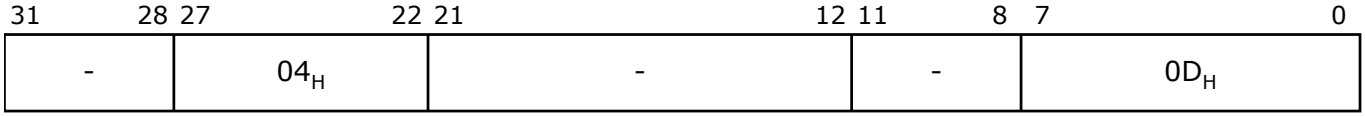
Debug

Description

If the Debug mode is enabled (DBGSR.DE == 1), cause a Debug Event; otherwise execute a NOP.

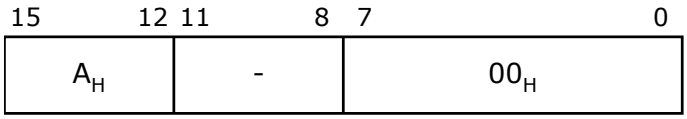
If the Debug mode is enabled (DBGSR.DE == 1), cause a Debug event; otherwise execute a NOP.

DEBUG (SYS)



-

DEBUG (SR)



-

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

3.1.70 DEXTR

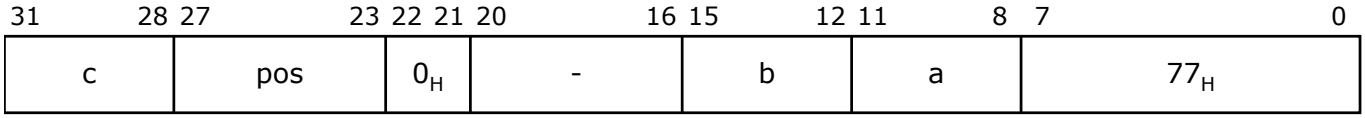
Extract from Double Register

Description

Extract 32 bits from registers {D[a], D[b]}, where D[a] contains the most-significant 32-bit 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].

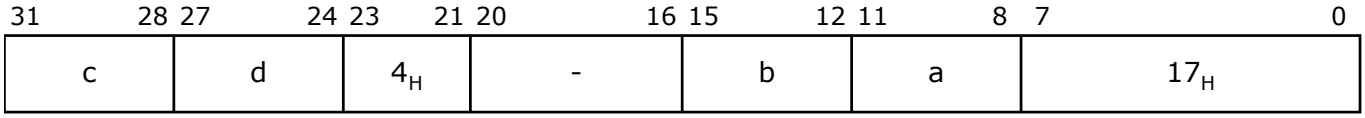
***Note**: D[a] and D[b] can be any two data registers or the same register. For this instruction they are treated as a 64-bit entity where D[a] contributes the high order bits and D[b] the low order bits.*

DEXTR D[c], D[a], D[b], pos (RRPW)




```
D[c] = ({D[a], D[b]} << pos)[63:32];
```

DEXTR D[c], D[a], D[b], D[d] (RRRR)




```
D[c] = ({D[a], D[b]} << D[d][4:0])[63:32];
```

***Note**: 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

3.1.71 DISABLE

Disable Interrupts

Description

***Note**: Execution Mode - This instruction can only be executed in the following modes: User-1, Supervisor.*

If the virtualization extension is not present or disabled

- Disable interrupts by clearing Interrupt Enable bit (ICR.IE) in the Interrupt Control Register. Optionaly update D[a] with the ICR.IE value prior to clearing.

If the virtualization extension is present and enabled

- Disable interrupts for the current virtual machine (VMn) by clearing Interrupt Enable bit (ICR.IE) in the Interrupt Control Register (ICR and VMn_ICR are aliases). Optionaly update D[a] with the ICR.IE value prior to clearing.

DISABLE (SYS)

31 28	27 22	21 12	11 8	7 0
-	$0D_H$	-	-	$0D_H$

```
if (TCCON.VIRT==1 AND VCON0.EN==1) then {
    // disables interrupts for virutal machine n
    // ICR is aliased to VMn_ICR
    VMn_ICR.IE = 0;
} else {
    ICR.IE = 0;
}
```

DISABLE D[a] (SYS)

31 28	27 22	21 12	11 8	7 0
-	$0F_H$	-	a	$0D_H$

```
D[a][31:1] = 0H;
if (TCCON.VIRT==1 AND VCON0.EN==1) then {
    // disables interrupts for virutal machine n
    // ICR is aliased to VMn_ICR
    D[a][0] = VMn_ICR.IE;
    VMn_ICR.IE = 0;
} else {
    D[a][0] = ICR.IE;
    ICR.IE = 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

```
disable
disable    d7
```

See Also

ENABLE, BISR, RESTORE

3.1.72 DSYNC

Synchronize Data

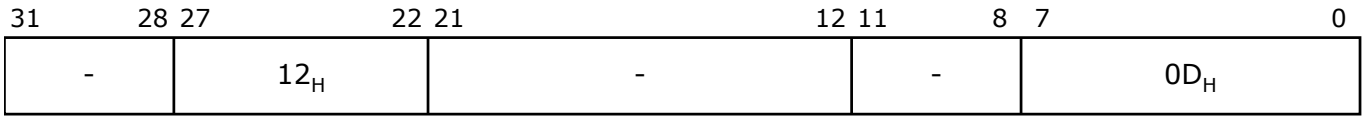
Description

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 consistency, a DSYNC instruction must be executed whenever write completion needs to be guaranteed.*

DSYNC (SYS)



-

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

LSYNC, ISYNC

3.1.73 DVADJ

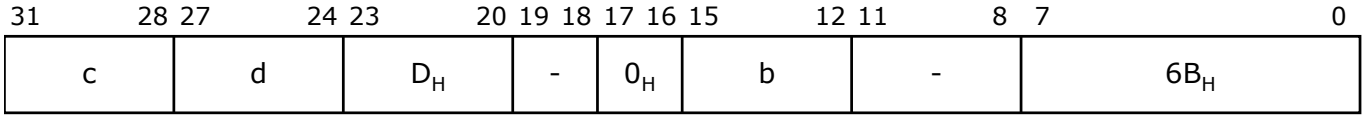
Divide-Adjust

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)




```
q_sign = E[d][63] ^ D[b][31];
x_sign = E[d][63];
eq_pos = x_sign & (E[d][63:32] == D[b]);
eq_neg = x_sign & (E[d][63:32] == -D[b]);

if((q_sign & ~eq_neg) | eq_pos) {
    quotient = E[d][31:0] + 1;
} else {
    quotient = E[d][31:0];
}
if(eq_pos | eq_neg) {
    remainder = 0;
} else {
    remainder = E[d][63:32];
}

gt = abs(E[d][63:32]) > abs(D[b]);
eq = !E[d][63] AND (abs(E[d][63:32] == abs(D[b]));
overflow = eq | gt;
if(overflow) {
    // Result is undefined
} else {
    E[c] = {remainder[31:0],quotient[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.

Examples

```
dvadj    e2, e4, d0
```

See Also

DVINIT, DVSTEP

3.1.74 DIV

Divide

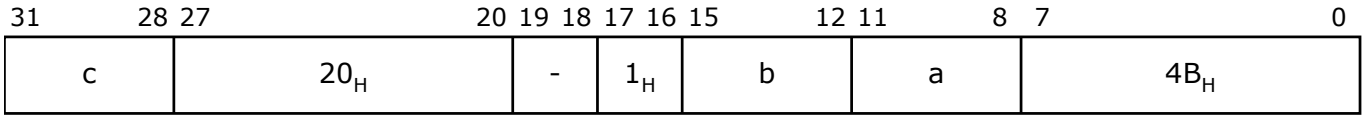
Description

Divide the contents of register D[a] by the contents of register D[b]. Put the resulting quotient in E[c][31:0] and the remainder in E[c][63:32].

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

Overflow occurs if the divisor (D[b]) is zero or if the dividend (D[a]) is the minimum negative number and the divisor is minus 1.

DIV E[c], D[a], D[b] (RR)




```
dividend = D[a];
divisor = D[b];
if (divisor == 0) then {
    if (dividend >= 0) then {
        quotient = 0x7fffffff;
        remainder = 0x00000000;
    } else {
        quotient = 0x80000000;
        remainder = 0x00000000;
    }
} else if ((divisor == 0xffffffff) AND (dividend == 0x80000000)) then {
    quotient = 0x7fffffff;
    remainder = 0x00000000;
} else {
    remainder = dividend % divisor
    quotient = (dividend - remainder)/divisor
}
E[c][31:0] = quotient;
E[c][63:32] = remainder;
```

Status Flags

C	Not set by this instruction.
V	if ((D[b] == 0) OR ((D[b] == 32'hFFFFFFFF) AND (D[a] == 32'h80000000))) then overflow = 1 else overflow = 0; if overflow then PSW.V = 1 else PSW.V = 0;
SV	if (overflow) then PSW.SV = 1 else PSW.SV = PSW.SV;
AV	PSW.AV = 0;
SAV	Not set by this instruction.

Examples

```
div    e2, d4, d0
```

See Also

DIV.U, DIV64, DIV64.U, REM64, REM64.U

3.1.75 DIV64

Divide 64-bit Long

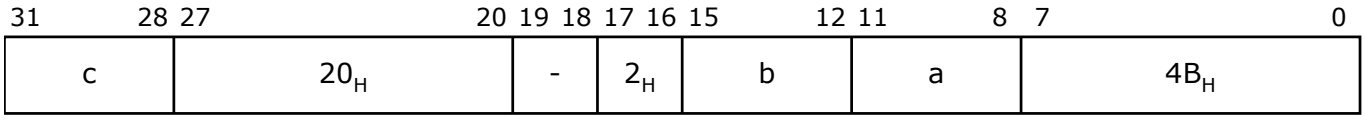
Description

Divide the contents of register E[a] by the contents of register E[b]. Put the resulting quotient in E[c][63:0]. This instruction does not provide a remainder.

The operands and results are treated as 64-bit signed integers.

Overflow occurs if the divisor (E[b]) is zero or if the dividend (E[a]) is the minimum negative number and the divisor is minus 1.

DIV64 E[c], E[a], E[b] (RR)




```
dividend = E[a];
divisor = E[b];
if (divisor == 0) then {
    if (dividend >= 0) then {
        quotient = 0x7fffffff_ffffffff;
    } else {
        quotient = 0x80000000_00000000;
    }
} else if ((divisor == 0xffffffff_ffffffff) AND (dividend == 0x80000000_00000000)) then {
    quotient = 0x7fffffff_ffffffff;
} else {
    remainder = dividend % divisor
    quotient = (dividend - remainder) / divisor
}
E[c][63:0] = quotient;
```

Status Flags

C	Not set by this instruction.
V	if ((E[b] == 0) OR ((E[b] == 64'hFFFFFFFFFFFFFFFF) AND (E[a] == 64'h8000000000000000))) then overflow = 1 else overflow = 0; if overflow then PSW.V = 1 else PSW.V = 0;
SV	if (overflow) then PSW.SV = 1 else PSW.SV = PSW.SV;
AV	PSW.AV = 0;
SAV	Not set by this instruction.

Examples

```
div64    e2, e4, e0
```

See Also

DIV, DIV.U, DIV64.U, REM64, REM64.U

3.1.76 DIV.U

Divide Unsigned

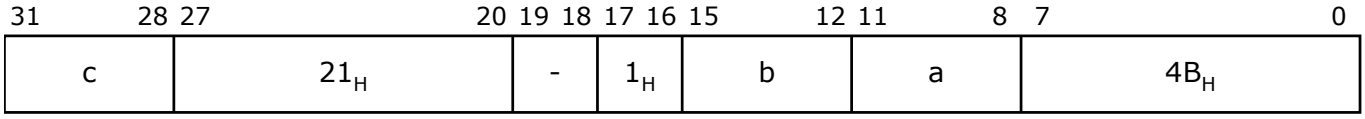
Description

Divide the contents of register D[a] by the contents of register D[b]. Put the resulting quotient in E[c][31:0] and the remainder in E[c][63:32].

The operands and results are treated as 32-bit unsigned integers.

Overflow occurs if the divisor (D[b]) is zero.

DIV.U E[c], D[a], D[b] (RR)




```
dividend = D[a];
divisor = D[b];
if (divisor == 0) then {
    quotient = 0xffffffff;
    remainder = 0x00000000;
} else {
    remainder = dividend % divisor
    quotient = (dividend - remainder)/divisor
}
E[c][31:0] = quotient;
E[c][63:32] = remainder;
```

Status Flags

C	Not set by this instruction.
V	if (D[b] == 0) then overflow = 1 else overflow = 0; if overflow then PSW.V = 1 else PSW.V = 0;
SV	if (overflow) then PSW.SV = 1 else PSW.SV = PSW.SV;
AV	PSW.AV = 0;
SAV	Not set by this instruction.

Examples

```
div.u    e2, d4, d0
```

See Also

DIV, DIV64, DIV64.U, REM64, REM64.U

3.1.77 DIV64.U

Divide 64-bit Unsigned Long

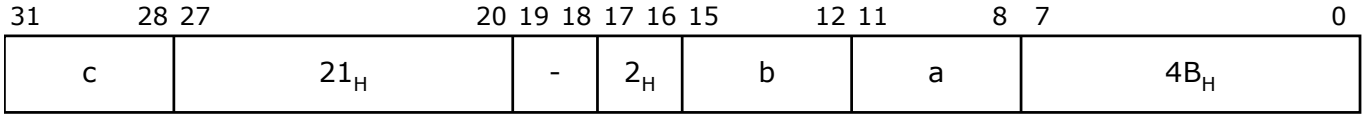
Description

Divide the contents of register E[a] by the contents of register E[b]. Put the resulting quotient in E[c][63:0]. This instruction does not provide a remainder.

The operands and results are treated as 64-bit unsigned long integers.

Overflow occurs if the divisor (E[b]) is zero.

DIV64.U E[c], E[a], E[b] (RR)




```
dividend = E[a];
divisor = E[b];
if (divisor == 0) then {
    quotient = 0xffffffff_ffffffff;
} else {
    remainder = dividend % divisor
    quotient = (dividend - remainder)/divisor
}
E[c][63:0] = quotient;
```

Status Flags

C	Not set by this instruction.
V	if (E[b] == 0) then overflow = 1 else overflow = 0; if overflow then PSW.V = 1 else PSW.V = 0;
SV	if (overflow) then PSW.SV = 1 else PSW.SV = PSW.SV;
AV	PSW.AV = 0;
SAV	Not set by this instruction.

Examples

```
div64.u    e2, e4, e0
```

See Also

DIV, DIV.U, DIV64, REM64, REM64.U

3.1.78 DVINIT

Divide-Initialization Word

Description

The DVINIT group of instructions prepare the operands for a subsequent DVSTEP instruction 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.

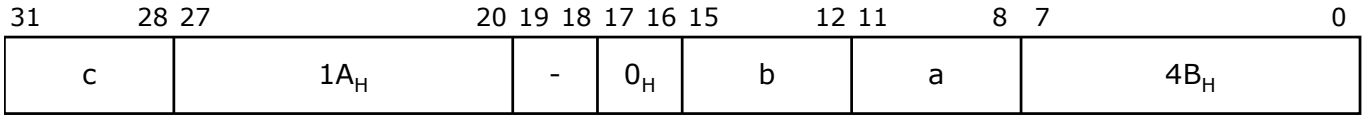
For a quotient result of 32 bits a DVINIT must be used, followed by four DVSTEP instructions.

The resultant bit fields in E[c] are as follows:

- DVINIT E[c][63:0] = partially calculated remainder.

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 bits.

DVINIT E[c], D[a], D[b] (RR)




```
E[c] = sign_ext(D[a]);
```

Status Flags

C	Not set by this instruction.
V	if ((D[b] == 0) OR ((D[b] == 32'hFFFFFFFF) AND (D[a] == 32'h80000000))) then overflow = 1 else overflow = 0; if (overflow) then PSW.V = 1 else PSW.V = 0;
SV	if (overflow) then PSW.SV = 1 else PSW.SV = PSW.SV;
AV	PSW.AV = 0;
SAV	Not set by this instruction.

Examples

```
dvinit    e2, d0, d4
```

See Also

DVADJ, DVINIT.U, DVINIT.B, DVINIT.BU, DVINIT.H, DVINIT.HU, DVSTEP, DVSTEP.U

3.1.79 DVINIT.U

Divide-Initialization Word Unsigned

Description

The DVINIT group of instructions prepare the operands for a subsequent DVSTEP instruction 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.

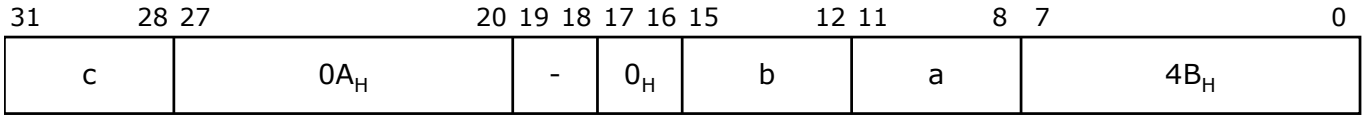
For a quotient result of 32 bits a DVINIT.U must be used, followed by four DVSTEP instructions.

The resultant bit fields in E[c] are as follows:

- DVINIT.U E[c][63:0] = partially calculated remainder.

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 bits.

DVINIT.U E[c], D[a], D[b] (RR)




```
E[c] = {00000000H, D[a]};
```

Status Flags

C	Not set by this instruction.
V	if (D[b] == 0) then overflow = 1 else overflow = 0; if (overflow) then PSW.V = 1 else PSW.V = 0;
SV	if (overflow) then PSW.SV = 1 else PSW.SV = PSW.SV;
AV	PSW.AV = 0;
SAV	Not set by this instruction.

Examples

```
dvinit.u    e2, d0, d4
```

See Also

DVINIT, DVINIT.B, DVINIT.BU, DVINIT.H, DVINIT.HU, DVSTEP, DVSTEP.U

3.1.80 DVINIT.B

Divide-Initialization Byte

Description

The DVINIT group of instructions prepare the operands for a subsequent DVSTEP instruction 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 can be used, but must be followed by only one DVSTEP instruction.

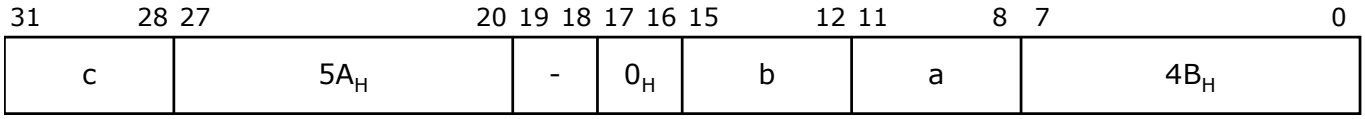
The resultant bit fields in E[c] are as follows:

- DVINIT.B E[c][63:24] = partially calculated remainder, E[c][23:0] = partially calculated quotient.

The .B suffix of the DVINIT group of instructions indicate an 8-bit quotient result, not 8-bit operands as in other instructions. The operands supplied to a DVINIT.B instruction are required to be 32-bit sign-extended values.

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 8 bits.

DVINIT.B E[c], D[a], D[b] (RR)




```
quotient_sign = !(D[a][31] == D[b][31]);
E[c][63:24] = sign_ext(D[a]);
E[c][23:0] = quotient_sign ? 24'b111111111111111111111111 : 24'b0;
```

Status Flags

C	Not set by this instruction.
V	if ((D[b] == 0) OR ((D[b] == 32'hFFFFFFFF AND (D[a] == 32'hFFFFFF80)) then overflow = 1 else overflow = 0; if (overflow) then PSW.V = 1 else PSW.V = 0;
SV	if (overflow) then PSW.SV = 1 else PSW.SV = PSW.SV;
AV	PSW.AV = 0;
SAV	Not set by this instruction.

Examples

```
dvinit.b    e2, d0, d4
```

See Also

DVADJ, DVINIT, DVINIT.U, DVINIT.BU, DVINIT.H, DVINIT.HU, DVSTEP, DVSTEP.U

3.1.81 DVINIT.BU

Divide-Initialization Byte Unsigned

Description

The DVINIT group of instructions prepare the operands for a subsequent DVSTEP instruction 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.BU can be used, but must be followed by only one DVSTEP instruction.

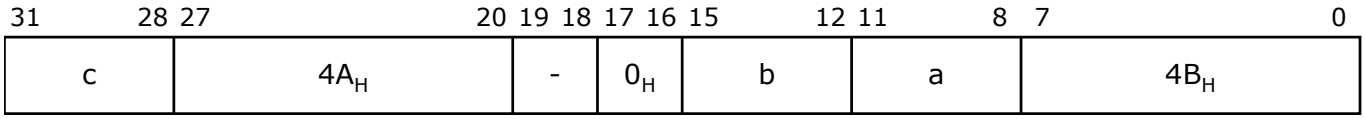
The resultant bit fields in E[c] are as follows:

- DVINIT.BU E[c][63:24] = partially calculated remainder, E[c][23:0] = partially calculated quotient.

The .BU suffix of the DVINIT group of instructions indicate an 8-bit quotient result, not 8-bit operands as in other instructions. The operands supplied to the DVINIT.BU instructions are 32-bit zero-extended values.

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 8 bits.

DVINIT.BU E[c], D[a], D[b] (RR)




```
E[c][63:24] = zero_ext(D[a]);
E[c][23:0] = 0;
```

Status Flags

C	Not set by this instruction.
V	if (D[b]==0) then overflow = 1 else overflow = 0; if (overflow) then PSW.V = 1 else PSW.V = 0;
SV	if (overflow) then PSW.SV = 1 else PSW.SV = PSW.SV;
AV	PSW.AV = 0;
SAV	Not set by this instruction.

Examples

```
dvinit.bu    e2, d0, d4
```

See Also

DVINIT, DVINIT.U, DVINIT.B, DVINIT.H, DVINIT.HU, DVSTEP, DVSTEP.U

3.1.82 DVINIT.H

Divide-Initialization Half-word

Description

The DVINIT group of instructions prepare the operands for a subsequent DVSTEP instruction 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 quotient result is guaranteed to fit into 16 bits then a DVINIT.H can be used but must be followed by two DVSTEP instructions.

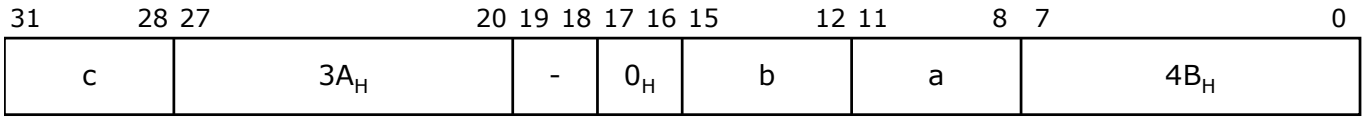
The resultant bit fields in E[c] are as follows:

- DVINIT.H E[c][63:16] = partially calculated remainder, E[c][15:0] = partially calculated quotient.

The .H suffix of the DVINIT group of instructions indicate an 16-bit quotient result, not an 16-bit operands as in other instructions. The operands supplied to a DVINIT.H instruction are required to be 32-bit sign-extended values.

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 16 bits.

DVINIT.H E[c], D[a], D[b] (RR)




```
quotient_sign = !(D[a][31] == D[b][31];
E[c][63:16] = sign_ext(D[a]);
E[c][15:0] = quotient_sign ? 16'b1111111111111111 : 16'b0;
```

Status Flags

C	Not set by this instruction.
V	if ((D[b] == 0) OR ((D[b] == 32'hFFFFFFFF AND (D[a] == 32'hFFFF8000))) then overflow = 1 else overflow=0; if (overflow) then PSW.V = 1 else PSW.V = 0;
SV	if (overflow) then PSW.SV = 1 else PSW.SV = PSW.SV;
AV	PSW.AV = 0;
SAV	Not set by this instruction.

Examples

```
dvinit.h    e2, d0, d4
```

See Also

DVADJ, DVINIT, DVINIT.U, DVINIT.B, DVINIT.BU, DVINIT.HU, DVSTEP, DVSTEP.U

3.1.83 DVINIT.HU

Divide-Initialization Half-word Unsigned

Description

The DVINIT group of instructions prepare the operands for a subsequent DVSTEP instruction 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 quotient result is guaranteed to fit into 16 bits then a DVINIT.HU can be used but must be followed by two DVSTEP instructions.

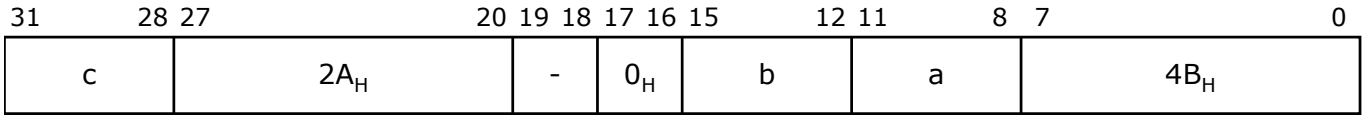
The resultant bit fields in E[c] are as follows:

- DVINIT.HU E[c][63:16] = partially calculated remainder, E[c][15:0] = partially calculated quotient.

The .HU suffix of the DVINIT group of instructions indicate an 16-bit quotient result, not an 16-bit operands as in other instructions. The operands supplied to the DVINIT.HU instructions are 32-bit zero-extended values.

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 16 bits.

DVINIT.HU E[c], D[a], D[b] (RR)




```
E[c][63:16] = zero_ext(D[a]);
E[c][15:0] = 0;
```

Status Flags

C	Not set by this instruction.
V	if (D[b] == 0) then overflow = 1 else overflow = 0; if (overflow) then PSW.V = 1 else PSW.V = 0;
SV	if (overflow) then PSW.SV = 1 else PSW.SV = PSW.SV;
AV	PSW.AV = 0;
SAV	Not set by this instruction.

Examples

```
dvinit.hu    e2, d0, d4
```

See Also

DVINIT, DVINIT.U, DVINIT.B, DVINIT.BU, DVINIT.H, DVSTEP, DVSTEP.U

3.1.84 DVSTEP

Divide-Step

Description

The DVSTEP 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).

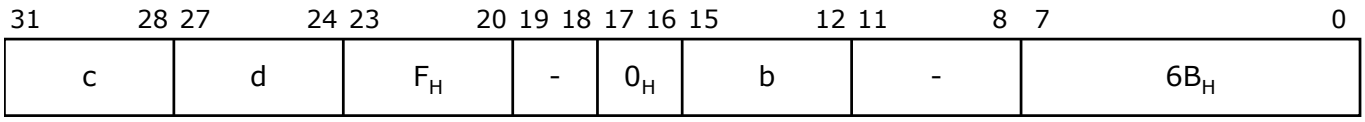
DVSTEP uses a modified restoring division algorithm to calculate 8 bits of the final remainder and quotient results. The size of the bit fields of the output register E[c] depend on the size of the bit fields in the input register E[d].

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 operates on signed operands. A DVSTEP instruction that yields the final remainder and final quotient should be followed by a DVADJ instruction.

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 {
    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]};
```

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

```
dvstep    e2, e4, d0
```

See Also

DVADJ, DVINIT, DVINIT.B, DVINIT.BU, DVINIT.H, DVINIT.HU, DVINIT.U, DVSTEP.U

3.1.85 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).

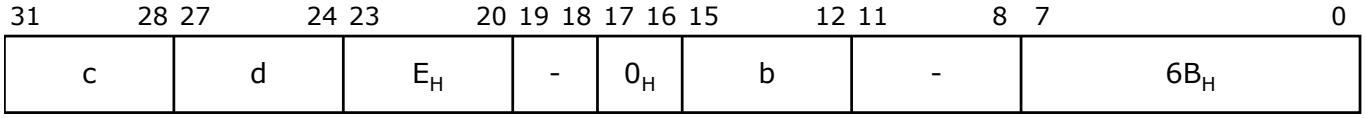
DVSTEP uses a modified restoring division algorithm to calculate 8 bits of the final remainder and quotient results. The size of the bit fields of the output register E[c] depend on the size of the bit fields in the input register E[d].

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.U operates on unsigned operands.

DVSTEP.U E[c], E[d], D[b] (RRR)




```
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 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

```
dvstep.u    e2, e4, d0
```

See Also

DVINIT, DVINIT.B, DVINIT.BU, DVINIT.H, DVINIT.HU, DVINIT.U, DVSTEP

3.1.86 ENABLE

Enable Interrupts

Description

***Note**: Execution Mode - This instruction can only be executed in the following modes: User-1, Supervisor.*

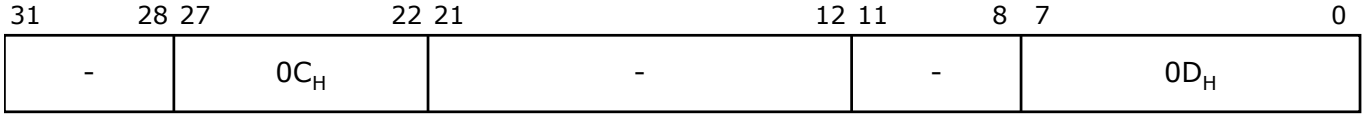
If the virtualization extension is not present or disabled

- Enable interrupts by setting the Interrupt Enable bit (ICR.IE) in the Interrupt Control Register (ICR) to one.

If the virtualization extension is present and enabled

- Enable interrupts for the current vitual machine (VMn) by setting the Interrupt Enable bit (ICR.IE) in the Interrupt Control Register to one (ICR and VMn_ICR are aliases).

ENABLE (SYS)




```
if (TCCON.VIRT==1 AND VCON0.EN==1) then {
    // enables interrupts for virutal machine n
    // ICR is aliased to VMn_ICR
    VMn_ICR.IE = 1;
} else {
    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, RESTORE

3.1.87 EQ

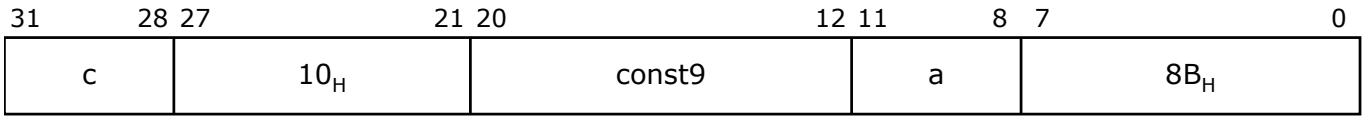
Equal

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.

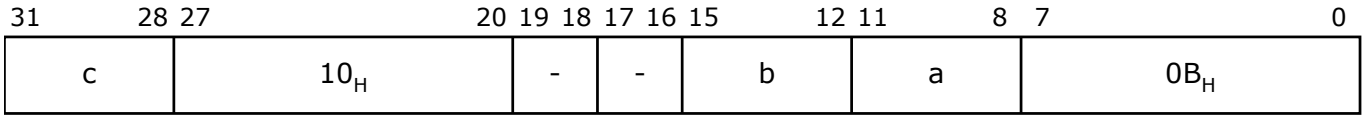
If the contents of data register D[a] are equal to the contents of either data register D[b] (instruction format SRR) or const4 (instruction format SRC), set the least-significant bit of D[15] to 1 and clear the remaining bits to zero; otherwise clear all bits in D[15]. The const4 value is sign-extended.

EQ D[c], D[a], const9 (RC)



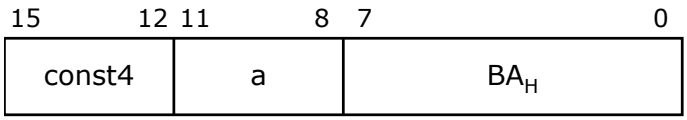

```
result = (D[a] == sign_ext(const9));
D[c] = zero_ext(result);
```

EQ D[c], D[a], D[b] (RR)



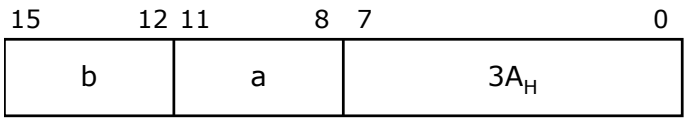

```
result = (D[a] == D[b]);
D[c] = zero_ext(result);
```

EQ D[15], D[a], const4 (SRC)




```
result = (D[a] == sign_ext(const4));
D[15] = zero_ext(result);
```

EQ D[15], D[a], D[b] (SRR)




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

```
eq    d15, d1, #6
eq    d15, d1, d2
```

See Also

GE, GE.U, LT, LT.U, NE, EQANY.B, EQANY.H

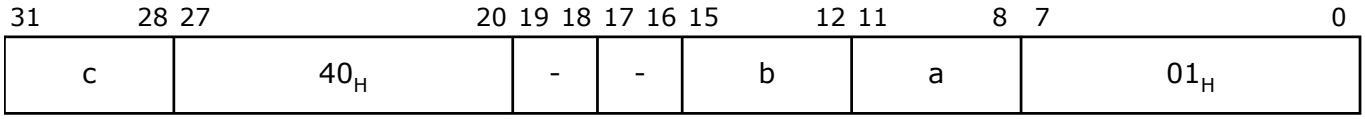
3.1.88 EQ.A

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)




```
result = (A[a] == A[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

```
eq.a    d3, a4, a2
```

See Also

EQZ.A, GE.A, LT.A, NE.A, NEZ.A

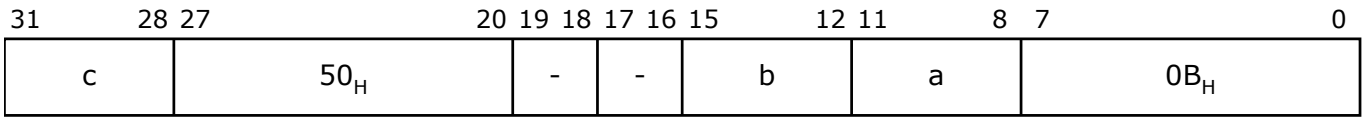
3.1.89 EQ.B

Equal Packed Byte

Description

Compare each byte of D[a] with the corresponding byte of D[b].
In each case, if the two are equal, set the corresponding byte of D[c] to all ones; otherwise set the corresponding byte of D[c] to all zeros.

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




```
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;
```

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.b    d3, d1, d2
```

See Also

EQ.H, EQ.W, LT.B, LT.BU, LT.H, LT.HU, LT.W, LT.WU

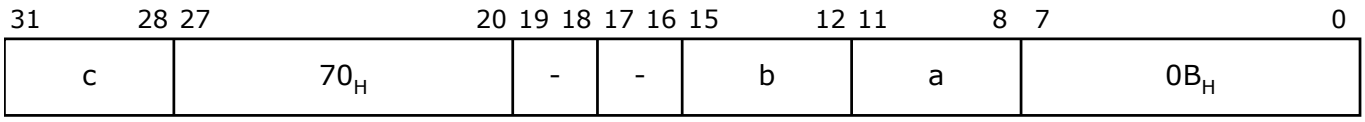
3.1.90 EQ.H

Equal Packed Half-word

Description

Compare each half-word of D[a] with the corresponding half-word of D[b].
In each case, if the two are equal, set the corresponding half-word of D[c] to all ones; otherwise set the corresponding half-word of D[c] to all zeros.

EQ.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;
```

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.h    d3, d1, d2
```

See Also

EQ.B, EQ.W, LT.BU, LT.H, LT.HU, LT.W, LT.WU

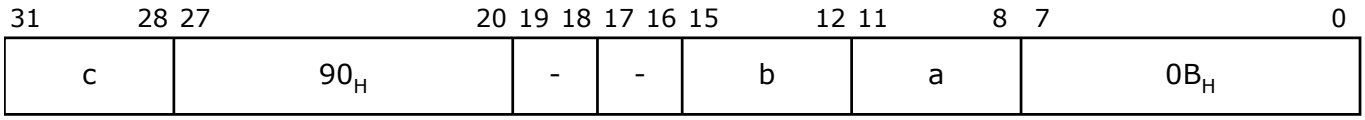
3.1.91 EQ.W

Equal Packed Word

Description

Compare D[a] with D[b].
If the two are equal, set D[c] to all ones; otherwise set the corresponding D[c] to all zeros.

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




```
D[c] = (D[a] == D[b]) ? 32'hFFFFFFFF ? 32'h00000000;
```

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.w    d3, d1,  d2
```

See Also

EQ.B, EQ.H, LT.B, LT.BU, LT.H, LT.HU, LT.W, LT.WU

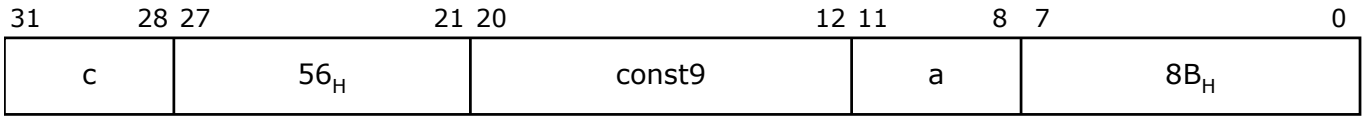
3.1.92 EQANY.B

Equal Any Byte

Description

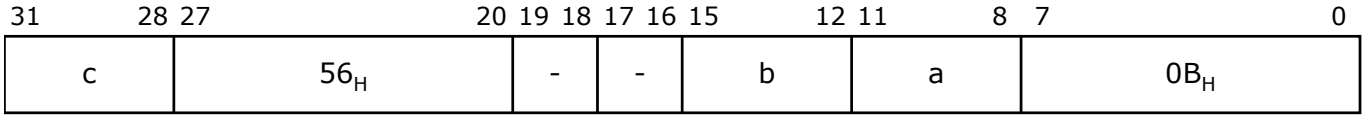
Compare each byte of D[a] with the corresponding byte of either D[b] (instruction format RR) or const9 (instruction format RC). If the logical OR of the Boolean results from each comparison is TRUE, set the least-significant bit of D[c] to 1 and clear the remaining bits to zero; otherwise clear all bits 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);
```

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

```
eqany.b    d3, d1, #126
eqany.b    d3, d1, d2
```

See Also

EQ, EQANY.H, GE, GE.U, LT, LT.U, NE

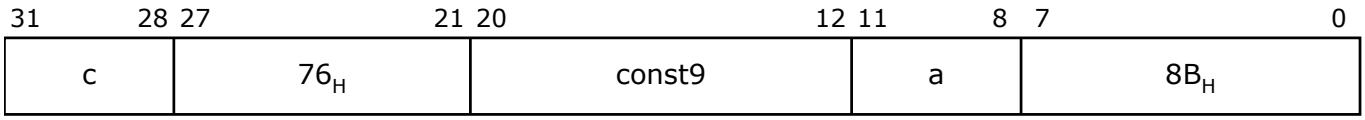
3.1.93 EQANY.H

Equal Any Half-word

Description

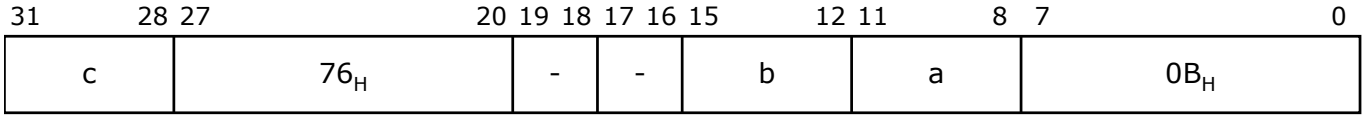
Compare each half-word of D[a] with the corresponding half-word of either D[b] (instruction format RR) or const9 (instruction format RC). If the logical OR of the Boolean results from each comparison is TRUE, set the least-significant bit of D[c] to 1 and clear the remaining bits to zero; otherwise clear all bits in D[c]. Const9 is sign-extended.

EQANY.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_halfword1;
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_halfword1;
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

```
eqany.h    d3, d1, #126
eqany.h    d3, d1, d2
```

See Also

EQ, EQANY.B, GE, GE.U, LT, LT.U, NE

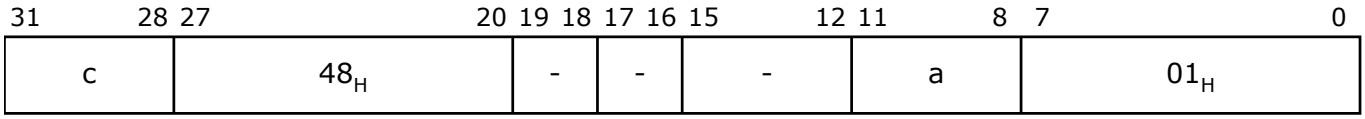
3.1.94 EQZ.A

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)




```
result = (A[a] == 0);
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

```
eqz.a    d3, a4
```

See Also

EQ.A, GE.A, LT.A, NE, NEZ.A

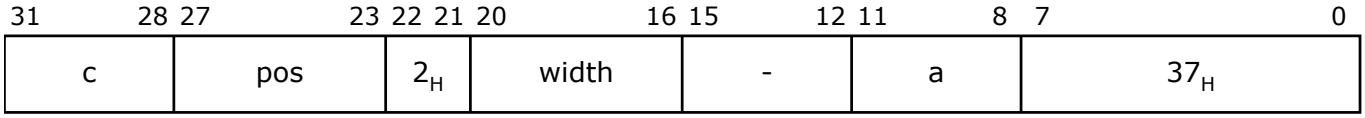
3.1.95 EXTR

Extract Bit Field

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 sign-extended result in D[c].

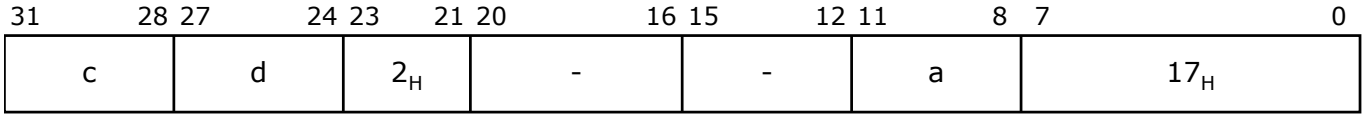
EXTR D[c], D[a], pos, width (RRPW)




```
D[c] = sign_ext((D[a] >> pos)[width-1:0]);
```

***Note**: If pos + width > 32 or if width = 0, then the results are undefined.*

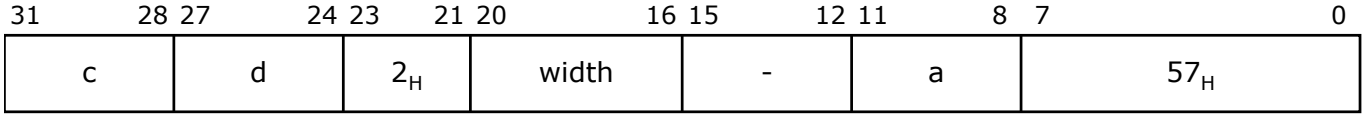
EXTR D[c], D[a], E[d] (RRRR)




```
width = E[d][36:32];
D[c] = sign_ext((D[a] >> E[d][4:0])[width-1:0]);
```

***Note**: If E[d][4:0] + width > 32 or if width = 0, then the results are undefined.*

EXTR D[c], D[a], D[d], width (RRRW)




```
D[c] = sign_ext((D[a] >> D[d][4:0])[width-1:0]);
```

***Note**: If D[d][4:0] + width > 32 or if width = 0, then the results are 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

```
extr    d3, d1, #2, #4
```

```
extr    d3, d1, e2
extr    d3, d1, d2, #4
```

See Also

DEXTR, EXTR.U, INSERT, INS.T, INSN.T, SHUFFLE

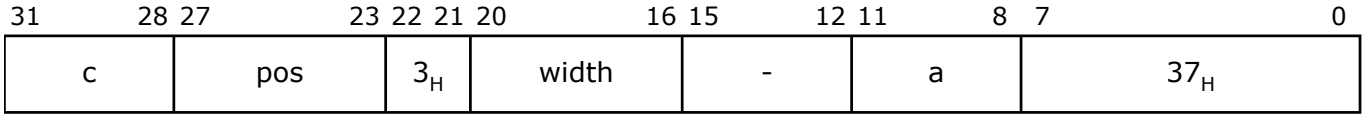
3.1.96 EXTR.U

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 zero-extended result in D[c].

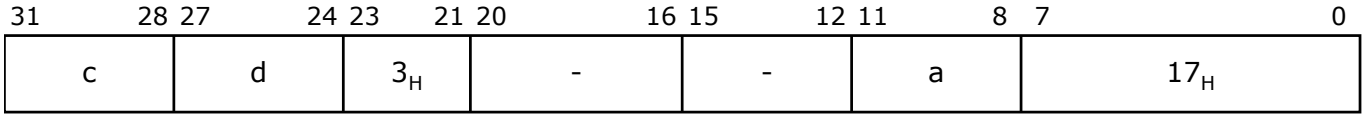
EXTR.U D[c], D[a], pos, width (RRPW)




```
D[c] = zero_ext((D[a] >> pos)[width-1:0]);
```

***Note**: If pos + width > 32 or if width = 0, then the results are undefined.*

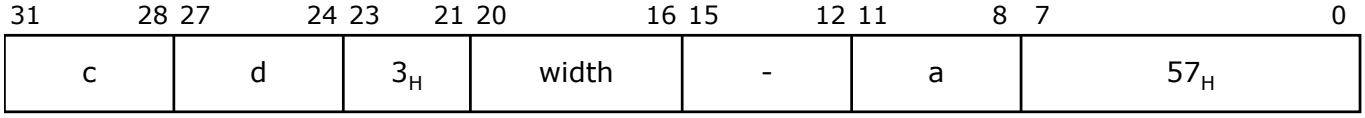
EXTR.U D[c], D[a], E[d] (RRRR)




```
width = E[d][36:32];
D[c] = zero_ext((D[a] >> E[d][4:0])[width-1:0]);
```

***Note**: If E[d][4:0] + width > 32 or if width = 0, then the results are undefined.*

EXTR.U D[c], D[a], D[d], width (RRRW)




```
D[c] = zero_ext((D[a] >> D[d][4:0])[width-1:0]);
```

***Note**: If D[d][4:0] + width > 32 or if width = 0, then the results are 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

```
extr.u    d3, d1, #2, #4
```

```
extr.u    d3, d1, e2
extr.u    d3, d1, d2, #4
```

See Also

DEXTR, EXTR, INSERT, INS.T, INSN.T, SHUFFLE

3.1.97 FCALL

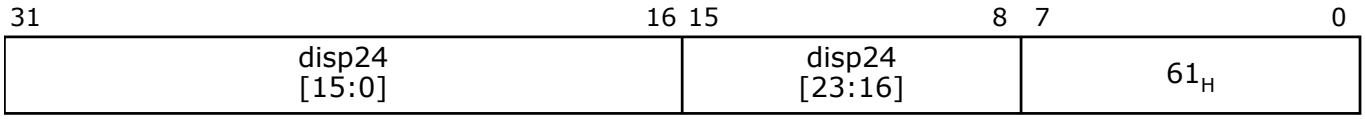
Fast Call

Description

Add the value specified by disp24, multiplied by two and sign-extended to 32-bit, to the address of the FCALL instruction and jump to the resulting address. The target address range is ±16 MBytes relative to the current PC.

Store A[11] to the memory address specified by A[10] pre-decremented by 4. Store the address of the next instruction in A[11].

FCALL disp24 (B)




```
ret_addr = PC + 4;
EA = A[10] - 4;
M(EA,word) = A[11];
PC = PC + sign_ext(2 * disp24);
A[11] = ret_addr[31:0];
A[10] = EA[31:0];
```

Status Flags

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

Examples

```
fcall    foobar
```

See Also

CALL, FCALLA, FCALLI, JL, JLA, RET, FRET

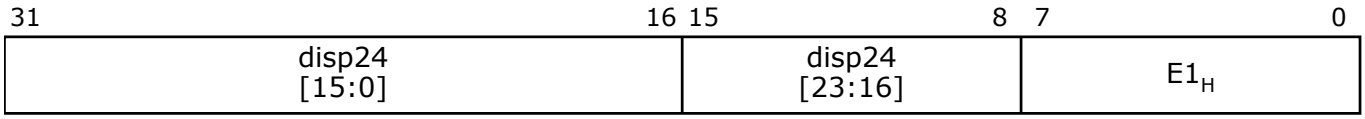
3.1.98 FCALLA

Fast Call Absolute

Description

Jump to the address specified by disp24. Store A[11] to the memory address specified by A[10] pre-decremented by 4. Store the address of the next instruction in A[11].

FCALLA disp24 (B)




```
ret_addr = PC + 4;
EA = A[10] - 4;
M(EA,word) = A[11];
PC = {disp24[23:20], 7'b0, disp24[19:0], 1'b0};
A[11] = ret_addr[31:0];
A[10] = EA[31:0];
```

Status Flags

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

Examples

```
fcalla    foobar
```

See Also

CALLA, FCALL, FCALLI, JL, JLA, RET, FRET

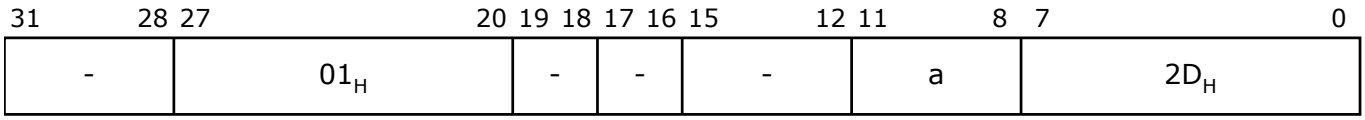
3.1.99 FCALLI

Fast Call Indirect

Description

Jump to the address specified by the contents of address register A[a]. The least-significant bit is always set to 0. Store A[11] to the memory address specified by A[10] pre-decremented by 4. Store the address of the next instruction in A[11].

FCALLI A[a] (RR)




```
ret_addr = PC + 4;
EA = A[10] - 4;
M(EA,word) = A[11];
PC = {A[a][31:1], 1'b0};
A[11] = ret_addr[31:0];
A[10] = EA[31:0];
```

Status Flags

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

Examples

```
fcalli    a2
```

See Also

CALLI, FCALL, FCALLA, JL, JLA, RET, FRET

3.1.100 FRET

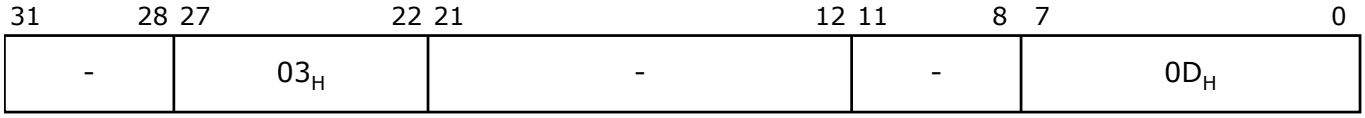
Return from Fast Call

Description

Return from a function that was invoked with an FCALL instruction. Jump to the address specified by A[11]. Load A[11] from the address specified by A[10] then increment A[10] by 4.

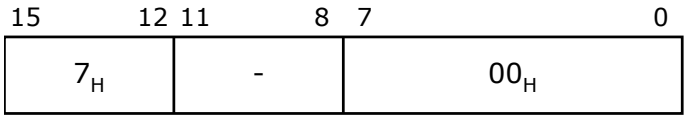
Return from a function that was invoked with an FCALL instruction. Jump to the address specified by A[11]. Load A[11] from the address specified by A[10] then increment A[10] by 4.

FRET (SYS)




```
PC = {A[11] [31:1], 1'b0};
EA = A[10];
A[11] = M(EA, word);
A[10] = A[10] + 4;
```

FRET (SR)




```
PC = {A[11] [31:1], 1'b0};
EA = A[10];
A[11] = M(EA, word);
A[10] = A[10] + 4;
```

Status Flags

C	Not changed by this instruction
V	Not changed by this instruction
SV	Not changed by this instruction
AV	Not changed by this instruction
SAV	Not changed by this instruction

Examples

```
fret
```

```
fret
```

See Also

CALL, CALLA, CALLI, FCALL, FCALLA, FCALLI, JL, JLA

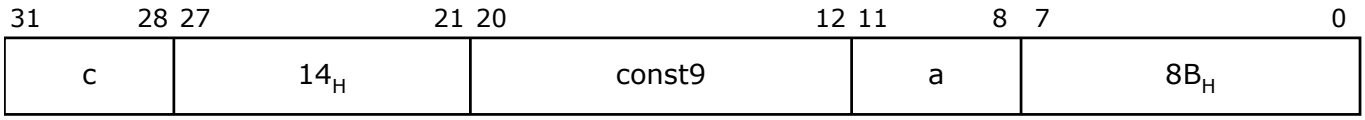
3.1.101 GE

Greater Than or Equal

Description

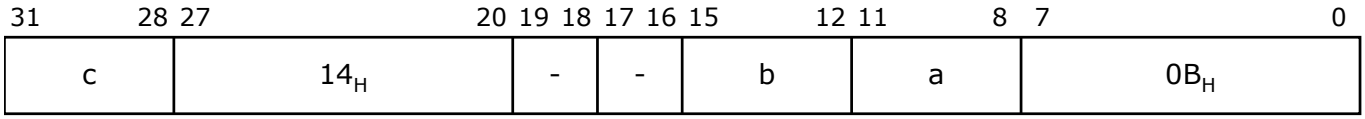
If the contents of data register D[a] are greater than or equal to the contents of either data register D[b] (instruction format RR) or const9 (instruction format RC), set the least-significant bit of D[c] to one and clear the remaining bits to zero; otherwise clear all bits in D[c]. D[a] and D[b] are treated as 32-bit signed. The const9 value is sign-extended.

GE D[c], D[a], const9 (RC)




```
result = (D[a] >= sign_ext(const9));
D[c] = zero_ext(result);
```

GE 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

```
ge    d3, d1, #126
ge    d3, d1, d2
```

See Also

EQ, GE.U, LT, LT.U, NE, EQANY.B, EQANY.H

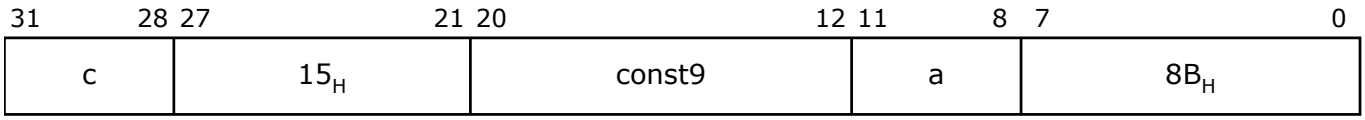
3.1.102 GE.U

Greater Than or Equal Unsigned

Description

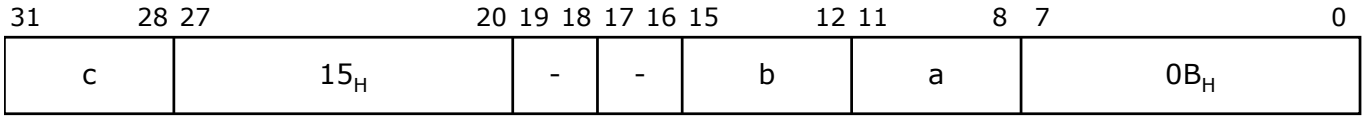
If the contents of data register D[a] are greater than or equal to the contents of either data register D[b] (instruction format RR) or const9 (instruction format RC), set the least-significant bit of D[c] to one and clear the remaining bits to zero; otherwise clear all bits in D[c]. D[a] and D[b] are treated as 32-bit unsigned integers. The const9 value is zero-extended.

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




```
result = (D[a] >= zero_ext(const9)); // unsigned
D[c] = zero_ext(result);
```

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




```
result = (D[a] >= D[b]); // unsigned
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

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

See Also

EQ, GE, LT, LT.U, NE, EQANY.B, EQANY.H

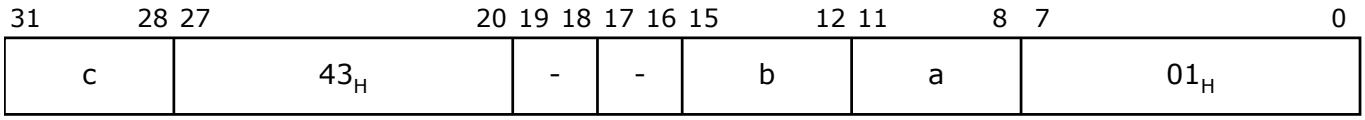
3.1.103 GE.A

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 32-bit unsigned integers.

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




```
result = (A[a] >= A[b]); // unsigned
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

```
ge.a    d3, a4, a2
```

See Also

EQ.A, EQZ.A, LT.A, NE, NEZ.A

3.1.104 IMASK

Insert Mask

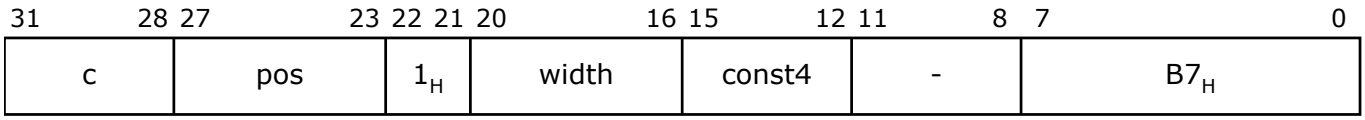
Description

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.

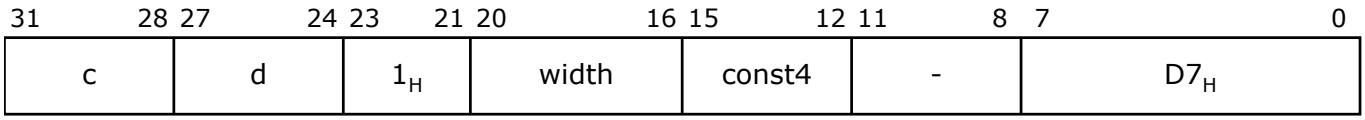
IMASK E[c], const4, pos, width (RCPW)




```
E[c][63:32] = ((2^width -1) << pos);
E[c][31:0] = (zero_ext(const4) << pos);
```

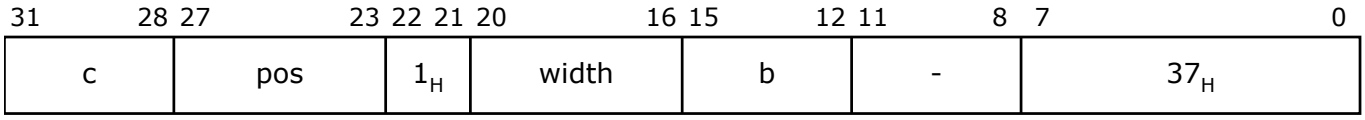
***Note**: If pos + width > 32 the result is undefined.*

IMASK E[c], const4, D[d], width (RCRW)



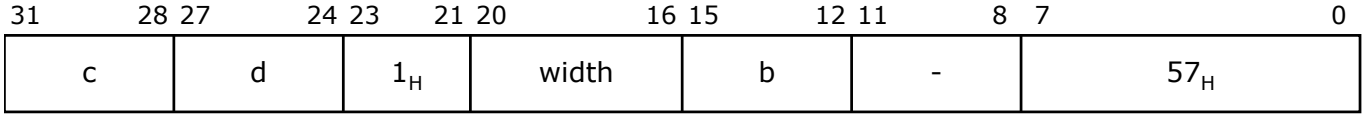

```
E[c][63:32] = ((2^width-1) << D[d][4:0]);
E[c][31:0] = (zero_ext(const4) << D[d][4:0]);
If (D[d][4:0] + width) > 32 the result is undefined.
```

IMASK E[c], D[b], pos, width (RRPW)




```
E[c][63:32] = ((2^width-1) << pos);
E[c][31:0] = (D[b][31:0] << pos);
If (pos + width) > 32 the result is undefined.
```

IMASK E[c], D[b], D[d], width (RRRW)




```
E[c][63:32] = ((2^width-1) << D[d][4:0]);
E[c][31:0] = (D[b] << D[d][4:0]);
```

```
If (D[d][4:0] + width) > 32 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, #6, #5, #11
imask    e2, #6, d2, #11
imask    e2, d1, #5, #11
imask    e2, d1, d2, #11
```

See Also

LDMST, ST.T

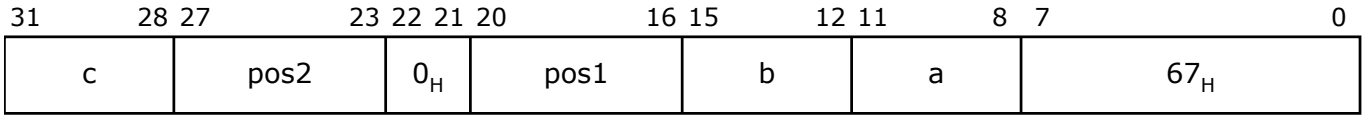
3.1.105 INS.T

Insert Bit

Description

Move the value of D[a] to D[c] with bit pos1 of this value replaced with bit pos2 of register D[b].

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




```
D[c] = {D[a][31:(pos1+1)], D[b][pos2], D[a][(pos1-1):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

```
ins.t    d3, d1, #5, d2, #7
```

See Also

DEXTR, EXTR, EXTR.U, INSERT, INSN.T

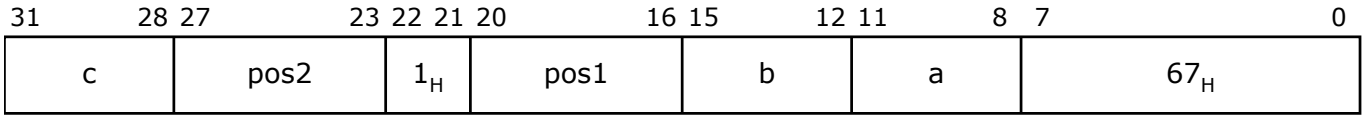
3.1.106 INSN.T

Insert Bit-Not

Description

Move the value of D[a] to D[c] with bit pos1 of this value replaced with the inverse of bit pos2 of register D[b].

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




```
D[c] = {D[a][31:(pos1+1)], !D[b][pos2], D[a][(pos1-1):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

```
insn.t    d3, d1, #5, d2, #7
```

See Also

DEXTR, EXTR, EXTR.U, INSERT, INS.T

3.1.107 INSERT

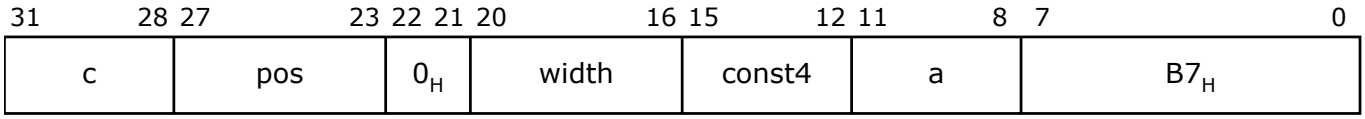
Insert Bit Field

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].

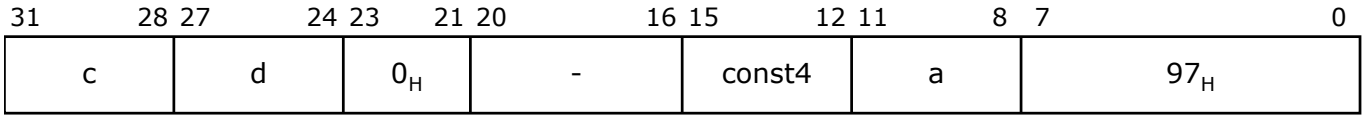
INSERT D[c], D[a], const4, pos, width (RCPW)




```
mask = (2^width -1) << pos;
D[c] = (D[a] & ~mask) | ((zero_ext(const4) << pos) & mask);
```

***Note**: If pos + width > 32, then the result is undefined.*

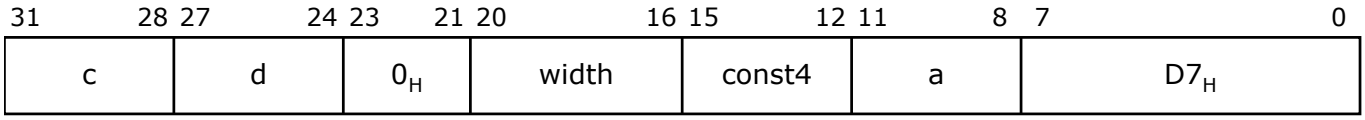
INSERT D[c], D[a], const4, E[d] (RCRR)




```
width = E[d][36:32];
mask = (2^width -1) << E[d][4:0];
D[c] = (D[a] & ~mask) | ((zero_ext(const4) << E[d][4:0]) & mask);
```

***Note**: If E[d][4:0] + E[d][36:32] > 32, then the result is undefined.*

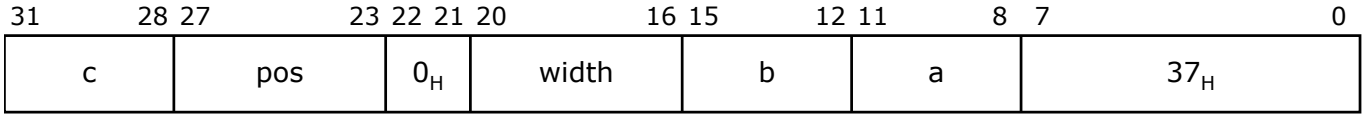
INSERT D[c], D[a], const4, D[d], width (RCRW)




```
mask = (2^width -1) << D[d][4:0];
D[c] = (D[a] & ~mask) | ((zero_ext(const4) << D[d][4:0]) & mask);
```

***Note**: If D[d][4:0] + width > 32, then the result is undefined.*

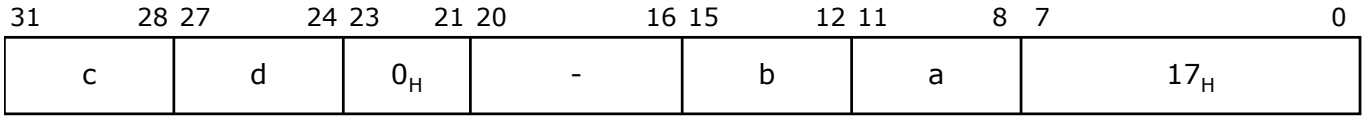
INSERT D[c], D[a], D[b], pos, width (RRPW)




```
mask = (2^width -1) << pos;
D[c] = (D[a] & ~mask) | ((D[b] << pos) & mask);
```

***Note**: If pos + width > 32, then the result is undefined.*

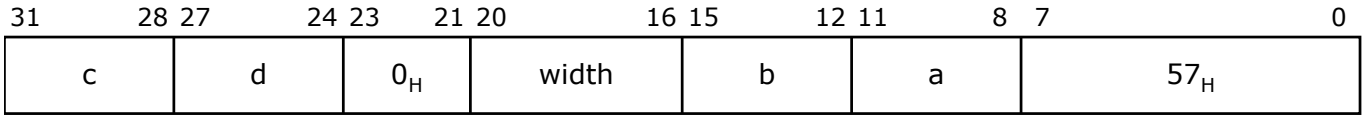
INSERT D[c], D[a], D[b], E[d] (RRRR)




```
width = E[d][36:32];
mask = (2^width -1) << E[d][4:0];
D[c] = (D[a] & ~mask) | ((D[b] << E[d][4:0]) & mask);
```

***Note**: 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)




```
mask = (2^width -1) << D[d][4:0];
D[c] = (D[a] & ~mask) | ((D[b] << D[d][4:0]) & mask);
```

***Note**: If D[d][4:0] + 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, 0, #16, #8
insert    d3, d1, 0, e4
insert    d3, d1, 0, d4, #8
insert    d3, d1, d2, #16, #8
insert    d3, d1, d2, e4
insert    d3, d1, d2, d4, #8
```

See Also

DEXTR, EXTR, EXTR.U, INS.T, INSN.T, SHUFFLE

3.1.108 ISYNC

Synchronize Instructions

Description

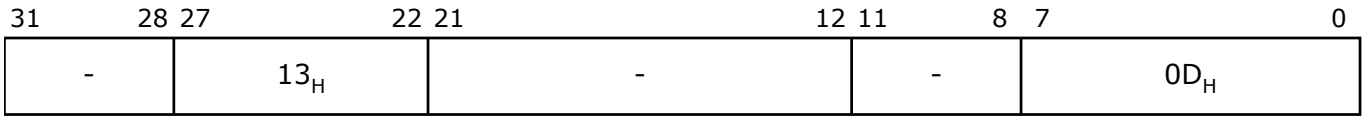
The ISYNC instruction forces completion of all previous instructions, then flushes the CPU pipelines and invalidates any cached pipeline state before proceeding to the next instruction.

***Note**: The Program Cache (PCACHE) is not invalidated by ISYNC.*

***Note**: An ISYNC instruction should follow a MTCR instruction. This ensures that all instructions following the ISYNC see the effects of the CSFR update.*

***Note**: An ISYNC instruction should follow a MTDCR instruction. This ensures that all instructions following the ISYNC see the effects of the CSFR update.*

ISYNC (SYS)



-

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, LSYNC, MTCR, MTDCR

3.1.109 IXMAX

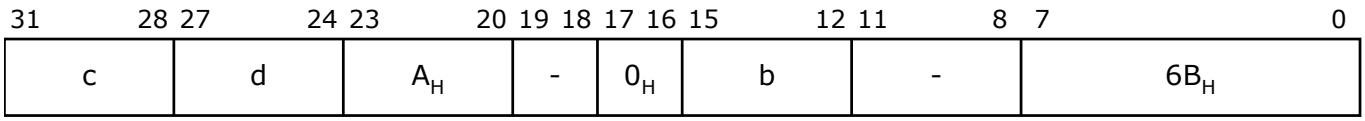
Find Maximum Index

Description

Enables a search of maximum value and its related index in a vector of 16-bit signed values.
For IXMAX:

- 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_H.
- D[b][15:0] First compare value.
- D[b][31:16] Second compare value.
- E[c][15:0] Updated working index.
- E[c][31:16] Updated index of maximum.
- E[c][47:32] Updated value of maximum.
- E[c][63:48] 00_H.

IXMAX E[c], E[d], D[b] (RRR)




```
E[c][15:0] = E[d][15:0] + 2;
E[c][63:48] = 00H;
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 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

```
ixmax    e2, e8, d6
```

See Also

IXMAX.U, IXMIN, IXMIN.U

3.1.110 IXMAX.U

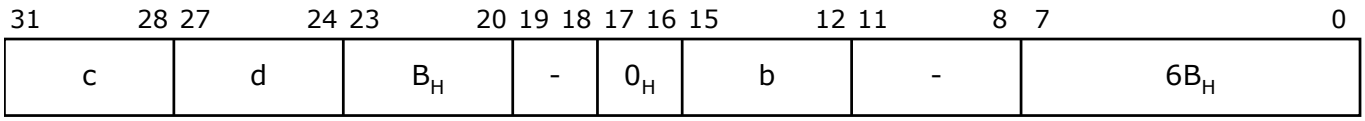
Find Maximum Index Unsigned

Description

Enables a search of maximum value and its related index in a vector of 16-bit unsigned values.
For IXMAX.U:

- 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_H.
- D[b][15:0] First compare value.
- D[b][31:16] Second compare value.
- E[c][15:0] Updated working index.
- E[c][31:16] Updated index of maximum.
- E[c][47:32] Updated value of maximum.
- E[c][63:48] 00_H.

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] = 00H;
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 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

```
ixmax.u    e2, e0, d4
```

See Also

IXMAX, IXMIN, IXMIN.U

3.1.111 IXMIN

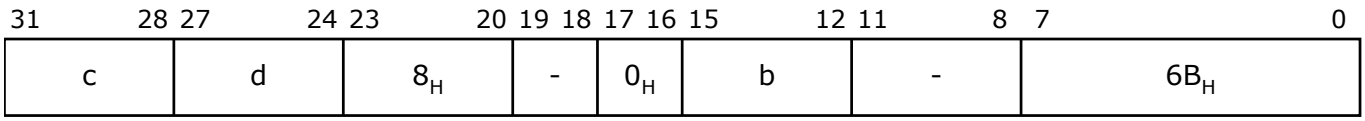
Find Minimum Index

Description

Enables search of minimum value and its related index in a vector of 16-bit signed values.
For IXMIN:

- 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_H.
- D[b][15:0] First compare value.
- D[b][31:16] Second compare value.
- E[c][15:0] Updated working index.
- E[c][31:16] Updated index of minimum.
- E[c][47:32] Updated value of minimum.
- E[c][63:48] 00_H.

IXMIN E[c], E[d], D[b] (RRR)




```
E[c][15:0] = E[d][15:0] + 2;
E[c][63:48] = 00H;
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 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

```
ixmin    e10, e2, d0
```

See Also

IXMAX, IXMAX.U, IXMIN.U

3.1.112 IXMIN.U

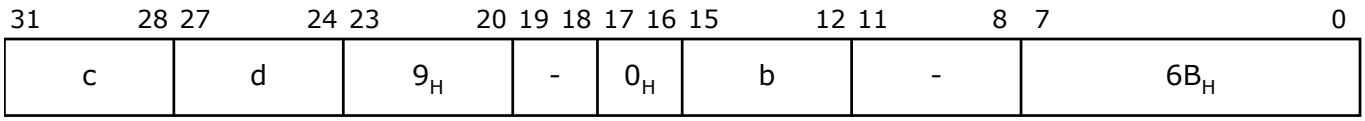
Find Minimum Index Unsigned

Description

Enables search of minimum value and its related index in a vector of 16-bit unsigned values.
For IXMIN.U:

- 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_H.
- D[b][15:0] First compare value.
- D[b][31:16] Second compare value.
- E[c][15:0] Updated working index.
- E[c][31:16] Updated index of minimum.
- E[c][47:32] Updated value of minimum.
- E[c][63:48] 00_H.

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] = 00H;
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 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 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

```
ixmin.u    e14, e2, d7
```

See Also

IXMAX, IXMAX.U, IXMIN

3.1.113 J

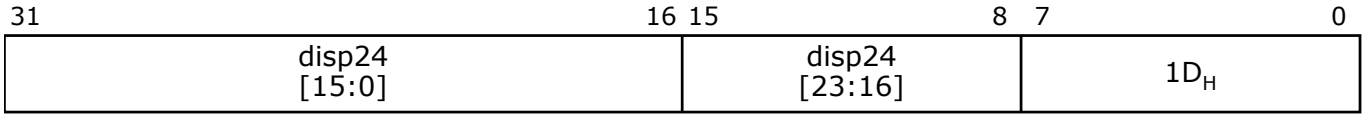
Jump Unconditional

Description

Add the value specified by disp24, multiplied by 2 and sign-extended, to the contents of PC and jump to that address.

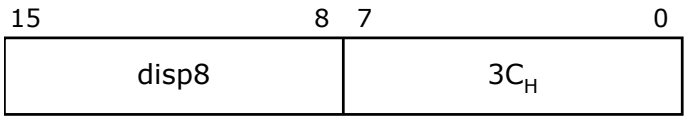
Add the value specified by disp8, multiplied by 2 and sign-extended, to the contents of PC and jump to that address.

J disp24 (B)




```
PC = PC + sign_ext(2 * disp24);
```

J disp8 (SB)




```
PC = PC + sign_ext(2 * disp8);
```

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, JRI, LOOPU

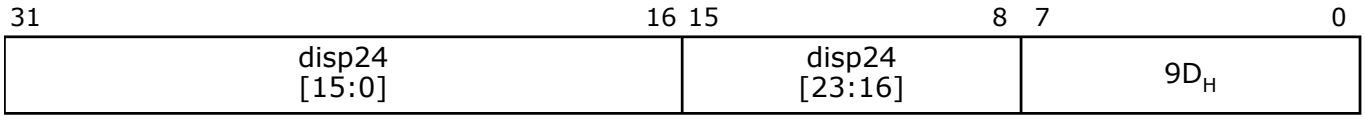
3.1.114 JA

Jump Unconditional Absolute

Description

Jump to the address specified by disp24.

JA disp24 (B)




```
PC = {disp24[23:20], 7'b0, 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, JRI, LOOPU

3.1.115 JEQ

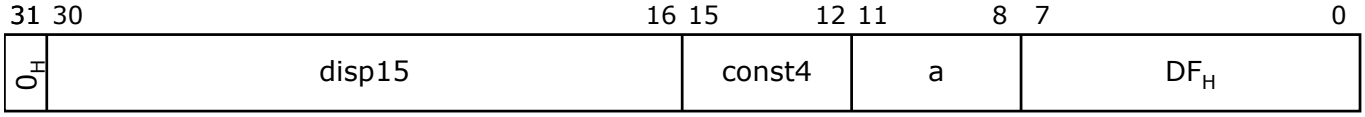
Jump if Equal

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, multiplied by 2 and sign-extended, to the contents of PC and jump to that address. The const4 value is sign-extended.

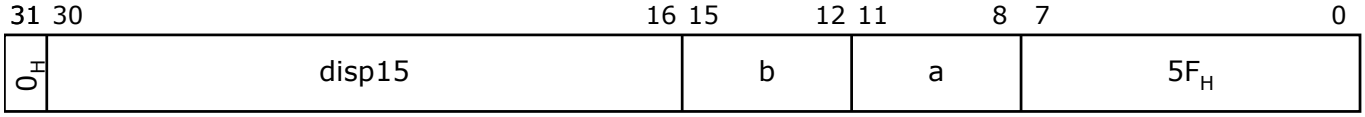
If the contents of D[15] are equal to the contents of either D[b] or const4, then add the value specified by either disp4 or disp4 + 16, multiplied by 2 and zero-extended, to the contents of PC and jump to that address. The const4 value is sign-extended.

JEQ D[a], const4, disp15 (BRC)



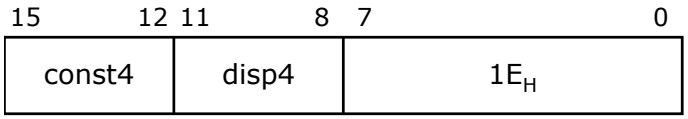

```
if (D[a] == sign_ext(const4)) then PC = PC + sign_ext(2 * disp15);
```

JEQ D[a], D[b], disp15 (BRR)



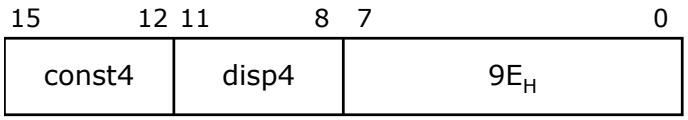

```
if (D[a] == D[b]) then PC = PC + sign_ext(2 * disp15);
```

JEQ D[15], const4, disp4 (SBC)



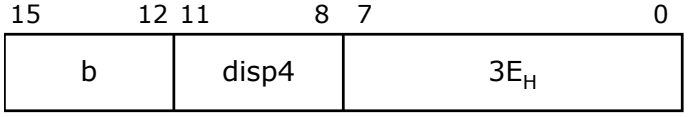

```
if (D[15] == sign_ext(const4)) then PC = PC + zero_ext(2 * disp4);
```

JEQ D[15], const4, disp4 (SBC)



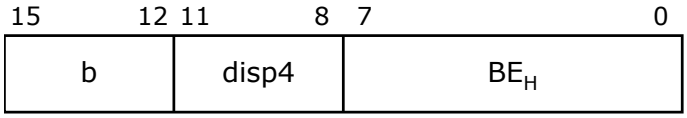

```
if (D[15] == sign_ext(const4)) then PC = PC + zero_ext(2 * (disp4 + 16));
```

JEQ D[15], D[b], disp4 (SBR)




```
if (D[15] == D[b]) then PC = PC + zero_ext(2 * disp4);
```

JEQ D[15], D[b], disp4 (SBR)




```
if (D[15] == D[b]) then PC = PC + zero_ext(2 * (disp4 + 16));
```

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    d1, #6, foobar
jeq    d1, d2, foobar
jeq    d15, #6, foobar
jeq    d15, d2, foobar
```

See Also

JGE, JGE.U, JLT, JLT.U, JNE

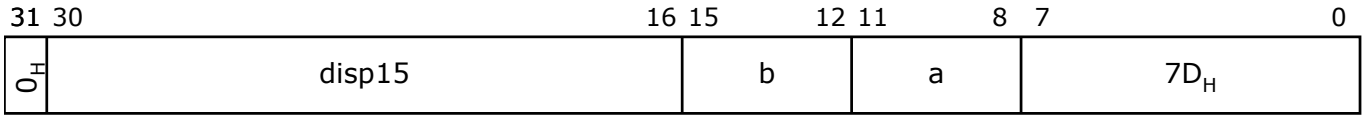
3.1.116 JEQ.A

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, multiplied by 2 and sign-extended, 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(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

```
jeq.a    a4, a2, foobar
```

See Also

JNE.A

3.1.117 JGE

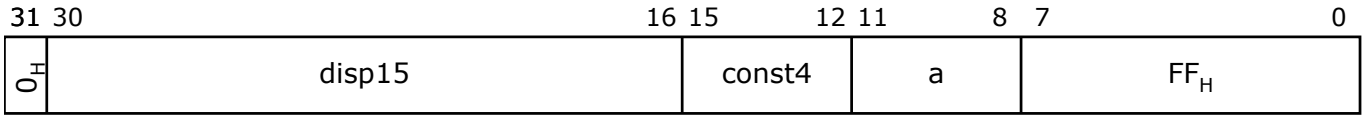
Jump if Greater Than or Equal

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, multiplied by 2 and sign-extended, to the contents of PC and jump to that address.

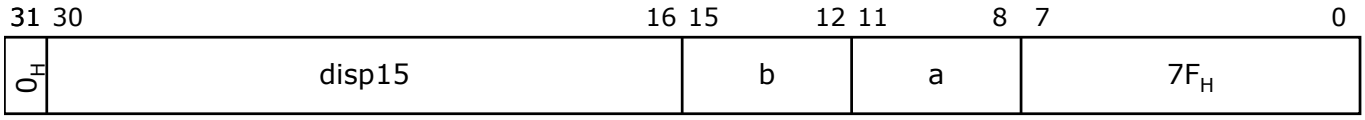
Operands are treated as 32-bit signed integers. The const4 value is sign-extended.

JGE D[a], const4, disp15 (BRC)




```
if (D[a] >= sign_ext(const4)) then PC = PC + sign_ext(2 * disp15);
```

JGE D[a], D[b], disp15 (BRR)




```
if (D[a] >= D[b]) then 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

```
jge    d1, d2, foobar
jge    d1, #6, foobar
```

See Also

JEQ, JGE.U, JLT, JLT.U, JNE

3.1.118 JGE.U

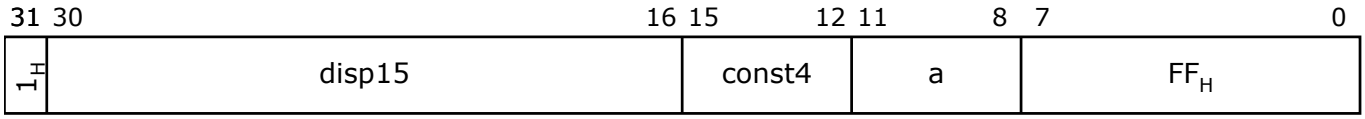
Jump if Greater Than or Equal Unsigned

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, multiplied by 2 and sign-extended, to the contents of PC and jump to that address.

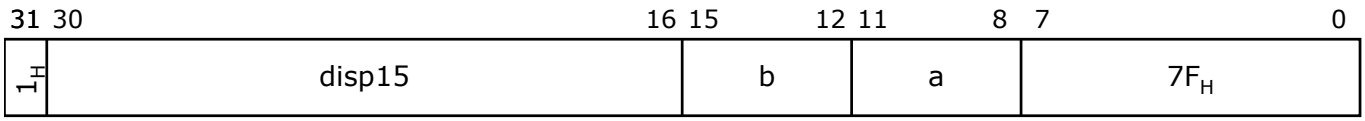
Operands are treated as 32-bit unsigned integers. The const4 value is zero-extended.

JGE.U D[a], const4, disp15 (BRC)




```
if (D[a] >= zero_ext(const4)) then PC = PC + sign_ext(2 * disp15); // unsigned comparison
```

JGE.U D[a], D[b], disp15 (BRR)




```
if (D[a] >= D[b]) then PC = PC + sign_ext(2 * disp15); // 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

```
jge.u    d1, d2, foobar
jge.u    d1, #6, foobar
```

See Also

JEQ, JGE, JLT, JLT.U, JNE

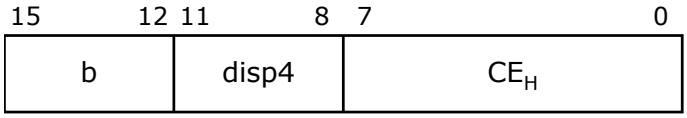
3.1.119 JGEZ

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, multiplied by 2 and zero-extended, to the contents of PC and jump to that address.

JGEZ D[b], disp4 (SBR)




```
if (D[b] >= 0) then PC = PC + zero_ext(2 * disp4);
```

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, JLEZ, JLTZ, JNZ, JZ

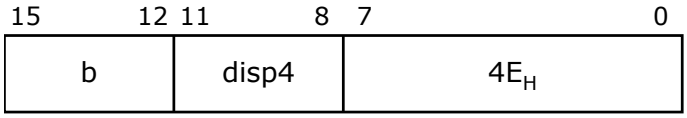
3.1.120 JGTZ

Jump if Greater Than Zero (16-bit)

Description

If the contents of D[b] are greater than zero, then add the value specified by disp4, multiplied by 2 and zero-extended, to the contents of PC and jump to that address.

JGTZ D[b], disp4 (SBR)




```
if (D[b] > 0) then PC = PC + zero_ext(2 * disp4);
```

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

```
jgtz    d2, foobar
```

See Also

JGEZ, JLEZ, JLTZ, JNZ, JZ

3.1.121 JI

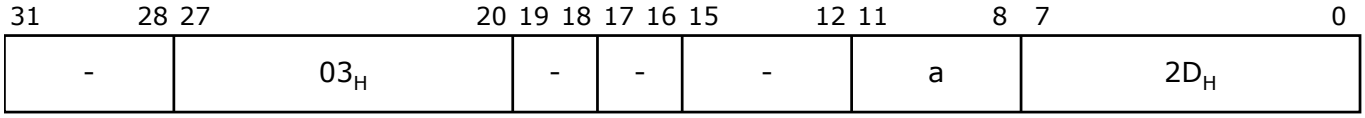
Jump Indirect

Description

Jump to the address specified by the contents of address register A[a]. The least-significant bit is always set to 0.

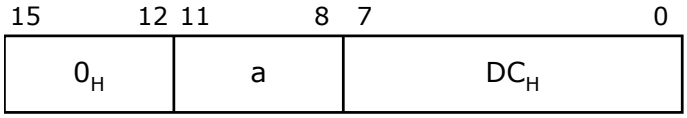
Jump to the address specified by the contents of address register A[a]. The least-significant bit is always set to 0.

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, JRI, LOOPU

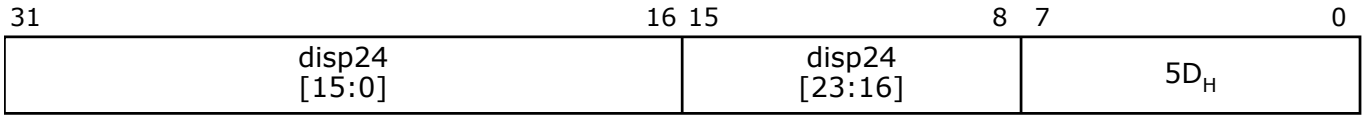
3.1.122 JL

Jump and Link

Description

Store the address of the next instruction in A[11] (return address). Add the value specified by disp24, multiplied by 2 and sign-extended, to the contents of PC and jump to that address.

JL disp24 (B)




```
A[11] = PC + 4;
PC = PC + sign_ext(2 * disp24);
```

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, JRI, CALLA, FCALL, FCALLA, LOOPU

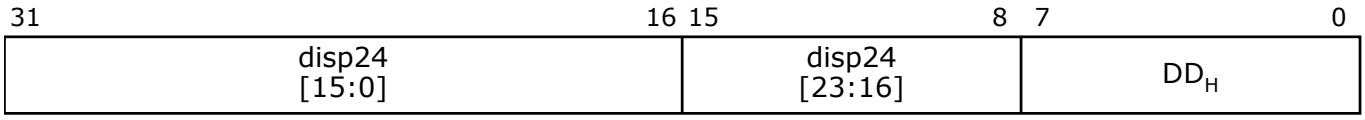
3.1.123 JLA

Jump and Link Absolute

Description

Store the address of the next instruction in A[11] (return address). Jump to the address specified by disp24.

JLA disp24 (B)




```
A[11] = PC + 4;
PC = {disp24[23:20], 7'b0, 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, JRI, CALLA, FCALL, FCALLA, LOOPU

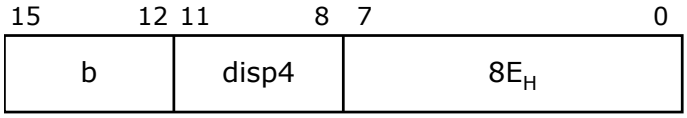
3.1.124 JLEZ

Jump if Less Than or Equal to Zero (16-bit)

Description

If the contents of D[b] are less than or equal to zero, then add the value specified by disp4, multiplied by 2 and zero-extended, to the contents of PC and jump to that address.

JLEZ D[b], disp4 (SBR)




```
If (D[b] <= 0) then PC = PC + zero_ext(2 * disp4);
```

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

```
jlez    d2, foobar
```

See Also

JGEZ, JGTZ, JLTZ, JNZ, JZ

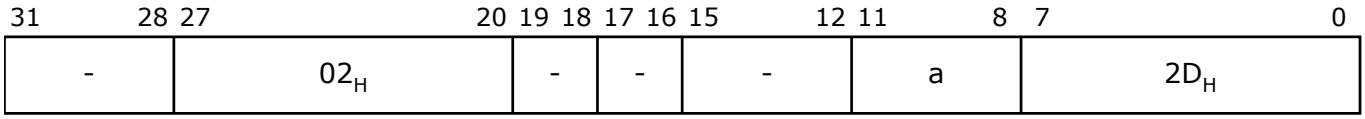
3.1.125 JLI

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, JRI, LOOPU

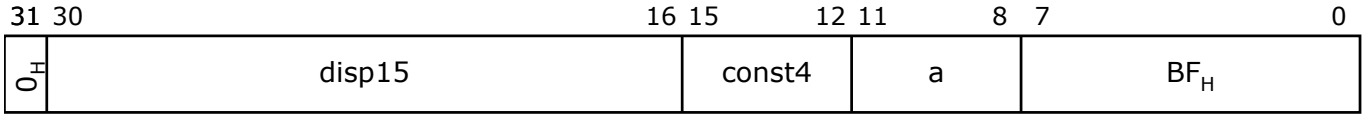
3.1.126 JLT

Jump if Less Than

Description

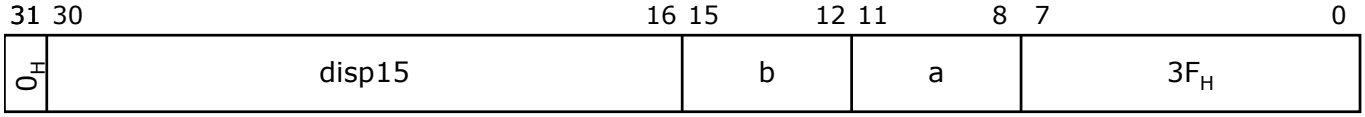
If the contents of D[a] are less than the contents of either D[b] (instruction format BRR) or const4 (instruction format BRC), then add the value specified by disp15, multiplied by 2 and sign-extended, to the contents of PC and jump to that address. The operands are treated as 32-bit signed integers. The const4 value is sign-extended.

JLT D[a], const4, disp15 (BRC)




```
if (D[a] < sign_ext(const4)) then PC = PC + sign_ext(2 * disp15);
```

JLT D[a], D[b], disp15 (BRR)




```
if (D[a] < D[b]) then 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

```
jlt    d1, #6, foobar
jlt    d1, d2, foobar
```

See Also

JEQ, JGE, JGE.U, JLT.U, JNE

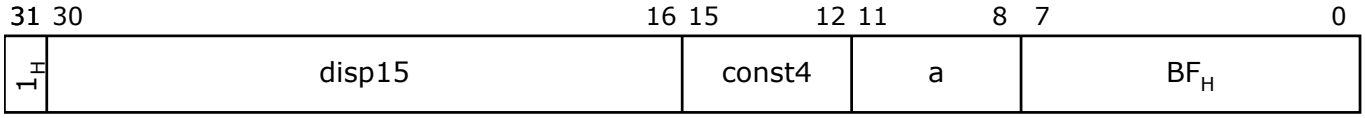
3.1.127 JLT.U

Jump if Less Than Unsigned

Description

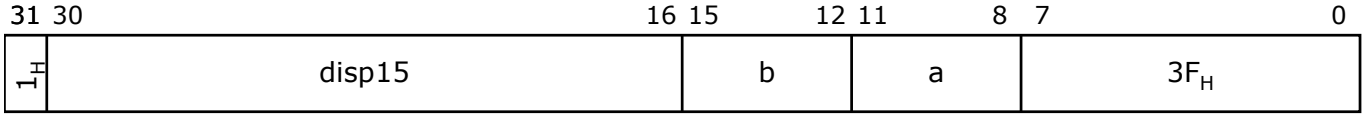
If the contents of D[a] are less than the contents of either D[b] (instruction format BRR) or const4 (instruction format BRC), then add the value specified by disp15, multiplied by 2 and sign-extended, to the contents of PC and jump to that address. The operands are treated as 32-bit unsigned integers. The const4 value is zero-extended.

JLT.U D[a], const4, disp15 (BRC)




```
if (D[a] < zero_ext(const4)) then PC = PC + sign_ext(2 * disp15); // unsigned comparison
```

JLT.U D[a], D[b], disp15 (BRR)




```
if (D[a] < D[b]) then PC = PC + sign_ext(2 * disp15); // 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

```
jlt.u    d1, #6, foobar
jlt.u    d1, d2, foobar
```

See Also

JEQ, JGE, JGE.U, JNE, JLT

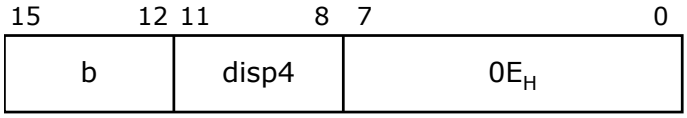
3.1.128 JLTZ

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, multiplied by 2 and zero-extended, to the contents of PC and jump to that address.

JLTZ D[b], disp4 (SBR)




```
if (D[b] < 0) then PC = PC + zero_ext(2 * disp4);
```

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, JGTZ, JLEZ, JNZ, JZ

3.1.129 JNE

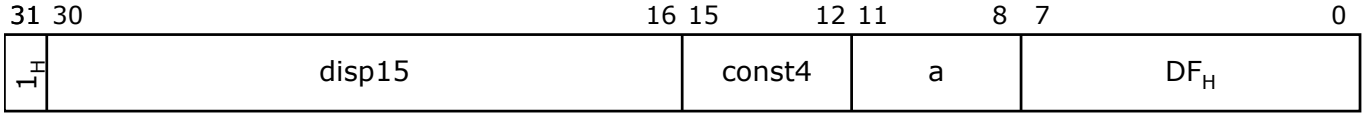
Jump if Not Equal

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, multiplied by 2 and sign-extended, to the contents of PC and jump to that address. The const4 value is sign-extended.

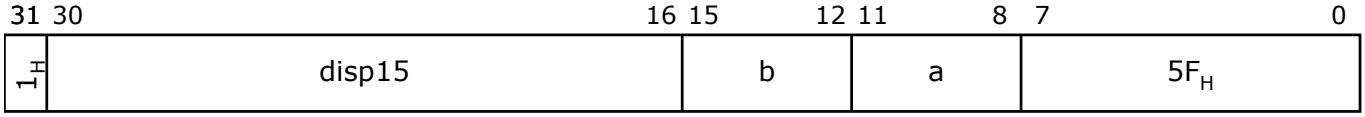
If the contents of D[15] are not equal to the contents of either D[b] (instruction format SBR) or const4 (instruction format SBC), then add the value specified by either disp4 or disp4 + 16, multiplied by 2 and zero-extended, to the contents of PC and jump to that address. The const4 value is sign-extended.

JNE D[a], const4, disp15 (BRC)



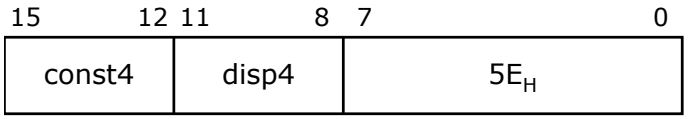

```
if (D[a] != sign_ext(const4)) then PC = PC + sign_ext(2 * disp15);
```

JNE D[a], D[b], disp15 (BRR)



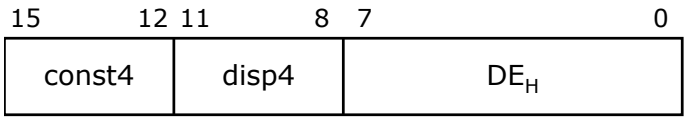

```
if (D[a] != D[b]) then PC = PC + sign_ext(2 * disp15);
```

JNE D[15], const4, disp4 (SBC)



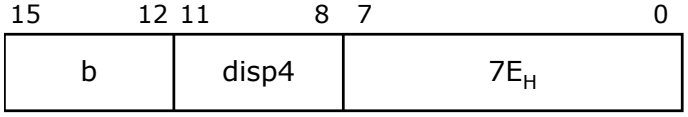

```
if (D[15] != sign_ext(const4)) then PC = PC + zero_ext(2 * disp4);
```

JNE D[15], const4, disp4 (SBC)



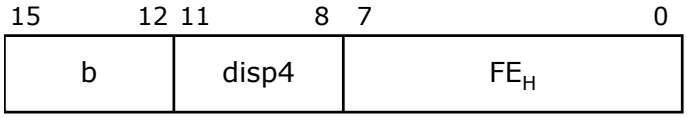

```
if (D[15] != sign_ext(const4)) then PC = PC + zero_ext(2 * (disp4 + 16));
```

JNE D[15], D[b], disp4 (SBR)




```
if (D[15] != D[b]) then PC = PC + zero_ext(2 * disp4);
```

JNE D[15], D[b], disp4 (SBR)




```
if (D[15] != D[b]) then PC = PC + zero_ext(2 * (disp4 + 16));
```

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, #6, foobar
jne    d1, d2, foobar
jne    d15, #6, foobar
jne    d15, d2, foobar
```

See Also

JEQ, JGE, JGE.U, JLT, JLT.U

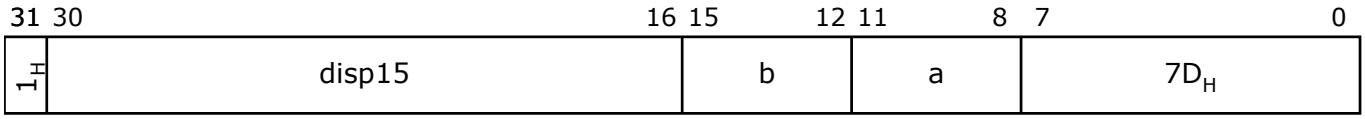
3.1.130 JNE.A

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, multiplied by 2 and sign-extended, 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(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

```
jne.a    a4, a2, foobar
```

See Also

JEQ.A

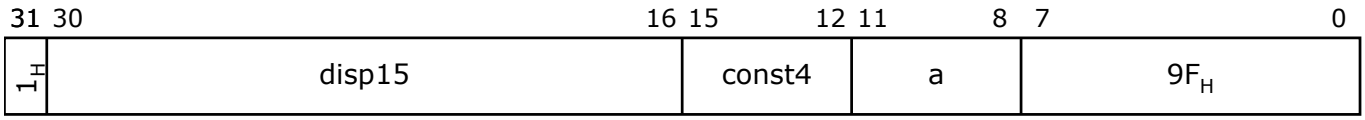
3.1.131 JNED

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, multiplied by 2 and sign-extended, to the contents of PC and jump to that address. Decrement the value in D[a] by one. The const4 value is sign-extended.

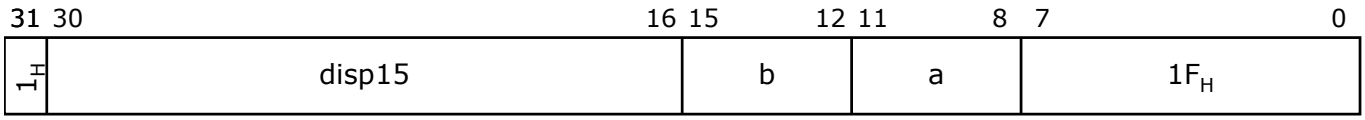
JNED D[a], const4, disp15 (BRC)




```
if (D[a] != sign_ext(const4)) then PC = PC + sign_ext(2 * disp15);
D[a] = D[a] - 1;
```

***Note**: The decrement is unconditional.*

JNED D[a], D[b], disp15 (BRR)




```
if (D[a] != D[b]) then PC = PC + sign_ext(2 * disp15);
D[a] = D[a] - 1;
```

***Note**: 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

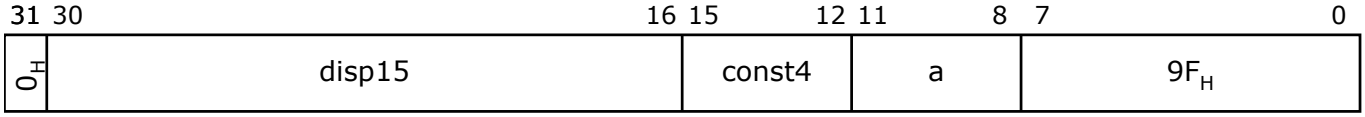
3.1.132 JNEI

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, multiplied by 2 and sign-extended, to the contents of PC and jump to that address. Increment the value in D[a] by one. The const4 value is sign-extended.

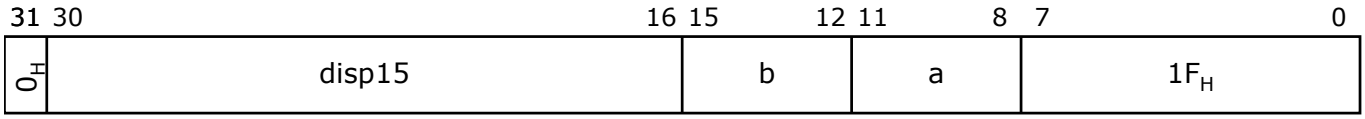
JNEI D[a], const4, disp15 (BRC)




```
if (D[a] != sign_ext(const4)) then PC = PC + sign_ext(2 * disp15);
D[a] = D[a] + 1;
```

***Note**: The increment is unconditional.*

JNEI D[a], D[b], disp15 (BRR)




```
if (D[a] != D[b]) then PC = PC + sign_ext(2 * disp15);
D[a] = D[a] + 1;
```

***Note**: 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

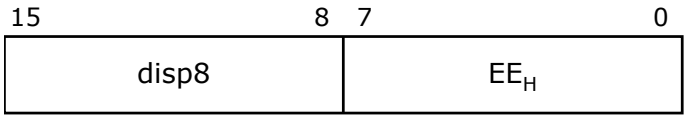
3.1.133 JNZ

Jump if Not Equal to Zero (16-bit)

Description

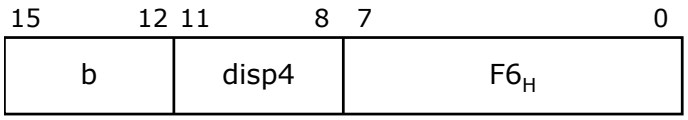
If contents of either D[b] (instruction format SBR) or D[15] (instruction format SB) are not equal to zero, then add value specified by either disp8 (format SB) or disp4 (format SBR), multiplied by 2 and sign-extended (disp8) or zero-extended (disp4), to the contents of PC and jump to that address.

JNZ D[15], disp8 (SB)




```
if (D[15] != 0) then PC = PC + sign_ext(2 * disp8);
```

JNZ D[b], disp4 (SBR)




```
if (D[b] != 0) then PC = PC + zero_ext(2 * disp4);
```

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    d15, foobar
jnz    d2,  foobar
```

See Also

JGEZ, JGTZ, JLEZ, JLTZ, JZ

3.1.134 JNZ.A

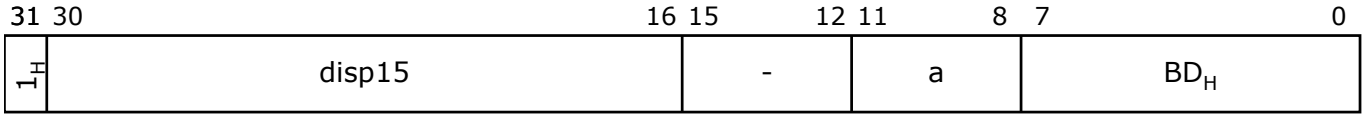
Jump if Not Equal to Zero Address

Description

If the contents of A[a] are not equal to zero, then add the value specified by disp15, multiplied by 2 and sign-extended, to the contents of PC and jump to that address.

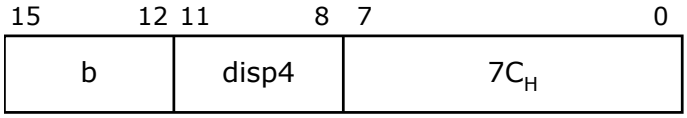
If the contents of A[b] are not equal to zero, then add the value specified by disp4, multiplied by 2 and zero-extended, to the contents of PC and jump to that address.

JNZ.A A[a], disp15 (BRR)




```
if (A[a] != 0) then PC = PC + sign_ext(2 * disp15);
```

JNZ.A A[b], disp4 (SBR)




```
if (A[b] != 0) then PC = PC + zero_ext(2 * disp4);
```

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

3.1.135 JNZ.T

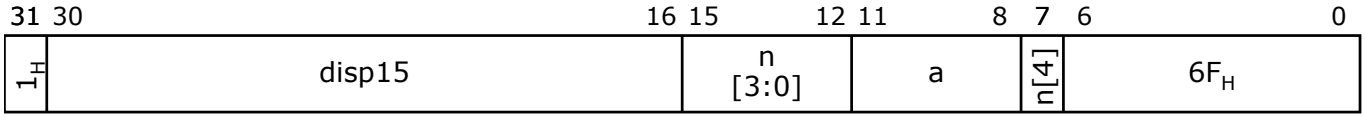
Jump if Not Equal to Zero Bit

Description

If bit n of register D[a] is not equal to zero, then add the value specified by disp15, multiplied by 2 and sign-extended, to the contents of PC and jump to that address.

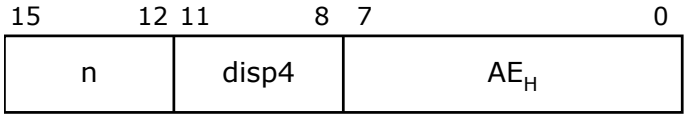
If bit n of register D[15] is not equal to zero, then add the value specified by disp4, multiplied by 2 and zero-extended, to the contents of PC and jump to that address.

JNZ.T D[a], n, disp15 (BRN)




```
if (D[a][n]) then PC = PC + sign_ext(2 * disp15);
```

JNZ.T D[15], n, disp4 (SBRN)




```
if (D[15][n]) then PC = PC + zero_ext(2 * disp4);
```

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

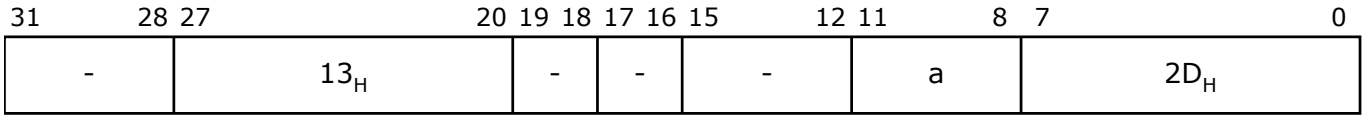
3.1.136 JRI

Jump Relative Indirect

Description

Add the contents of address register A[a] to the contents of PC and jump to that address. The least-significant bit is always set to 0.

JRI A[a] (RR)




```
target_PC = PC + A[a];
PC = {target_PC[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

```
jri    a2
```

See Also

J, JI, JA, JL, JLA, JLI, LOOPU

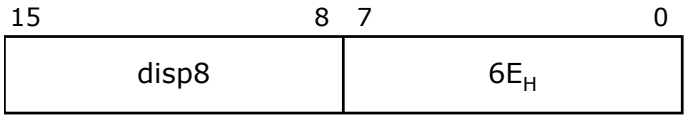
3.1.137 JZ

Jump if Zero (16-bit)

Description

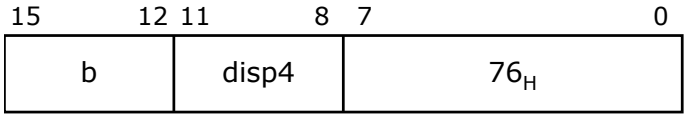
If the contents of either D[15] (instruction format SB) or D[b] (instruction format SBR) are equal to zero, then add the value specified by either disp8 (format SB) or disp4 (format SBR), multiplied by 2 and sign-extended (disp8) or zero-extended (disp4), to the contents of PC, and jump to that address.

JZ D[15], disp8 (SB)




```
if (D[15] == 0) then PC = PC + sign_ext(2 * disp8);
```

JZ D[b], disp4 (SBR)




```
if (D[b] == 0) then PC = PC + zero_ext(2 * disp4);
```

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    d15, foobar
jz    d2, foobar
```

See Also

JGEZ, JGTZ, JLEZ, JLTZ, JNZ

3.1.138 JZ.A

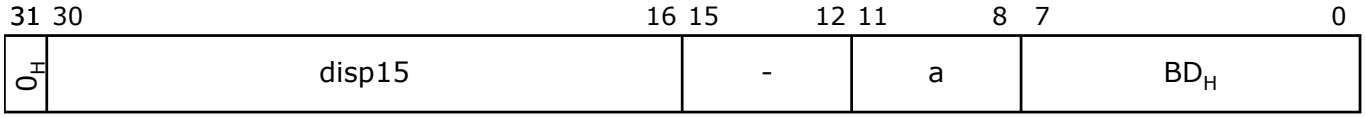
Jump if Zero Address

Description

If the contents of A[a] are equal to zero then add the value specified by disp15, multiplied by 2 and sign-extended, to the contents of PC and jump to that address.

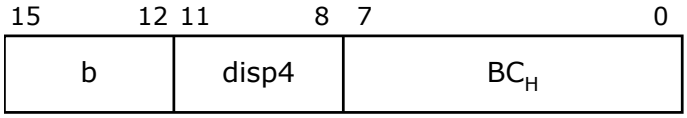
If the contents of A[b] are equal to zero then add the value specified by disp4, multiplied by 2 and zero-extended, to the contents of PC and jump to that address.

JZ.A A[a], disp15 (BRR)




```
if (A[a] == 0) then PC = PC + sign_ext(2 * disp15);
```

JZ.A A[b], disp4 (SBR)




```
if (A[b] == 0) then PC = PC + zero_ext(2 * disp4);
```

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

3.1.139 JZ.T

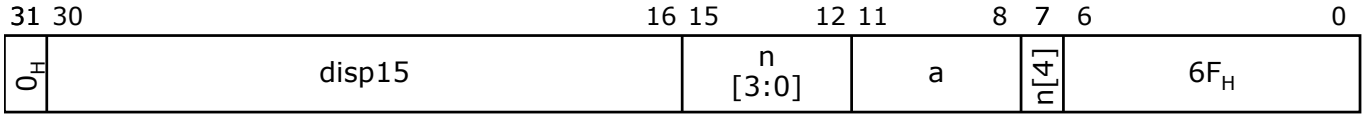
Jump if Zero Bit

Description

If bit n of register D[a] is equal to zero then add the value specified by disp15, multiplied by 2 and sign-extended, to the contents of PC and jump to that address.

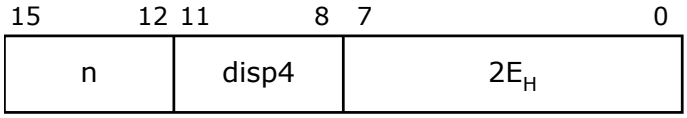
If bit n of register D[15] is equal to zero then add the value specified by disp4, multiplied by 2 and zero-extended, to the contents of PC and jump to that address.

JZ.T D[a], n, disp15 (BRN)




```
if (!D[a][n]) then PC = PC + sign_ext(2 * disp15);
```

JZ.T D[15], n, disp4 (SBRN)




```
if (!D[15][n]) then PC = PC + zero_ext(2 * disp4);
```

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

3.1.140 LD.A

Load Word to Address Register

Description

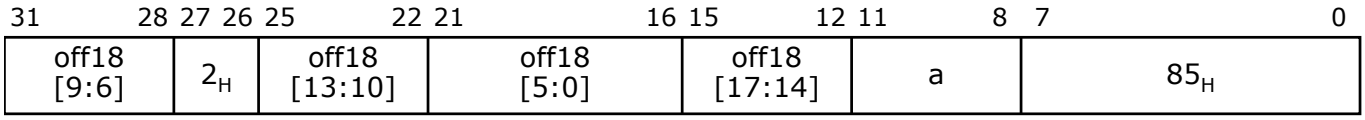
Load the word contents of the memory location specified by the effective address into address register A[a].

***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 effective address into either address register A[a] or A[15].

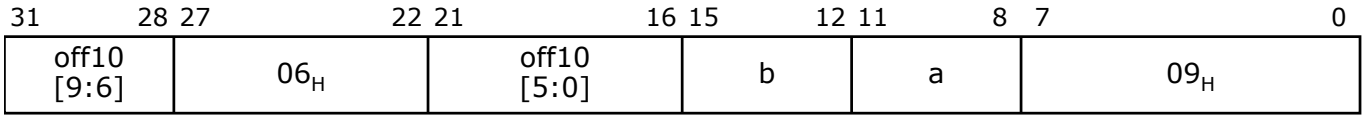
***Note**: If the target register is modified by the addressing mode, the result is undefined.*

LD.A A[a], off18 (ABS) (Absolute Addressing Mode)



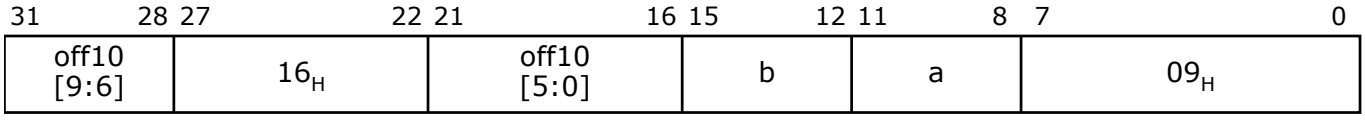

```
EA = {off18[17:14], 14b'0, off18[13:0]};
A[a] = M(EA, word);
```

LD.A A[a], A[b], off10 (BO) (Post-increment Addressing Mode)



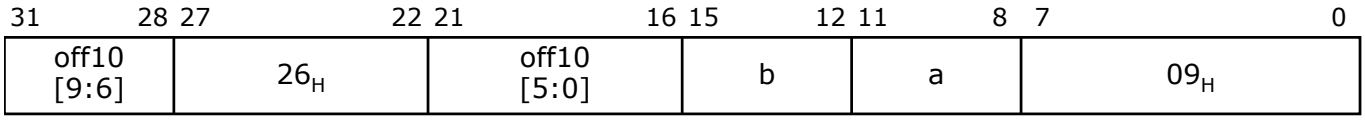

```
EA = A[b];
A[a] = M(EA, word);
A[b] = EA + sign_ext(off10);
```

LD.A A[a], A[b], off10 (BO) (Pre-increment Addressing Mode)



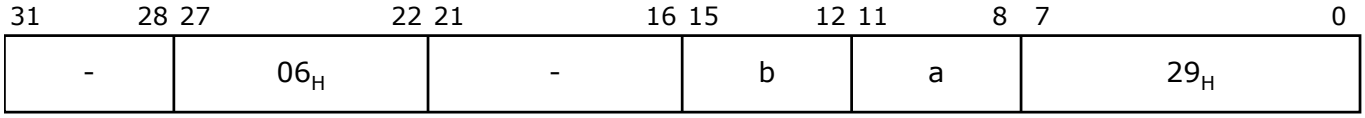

```
EA = A[b] + sign_ext(off10);
A[a] = M(EA, word);
A[b] = EA;
```

LD.A A[a], A[b], off10 (BO) (Base + Short Offset Addressing Mode)



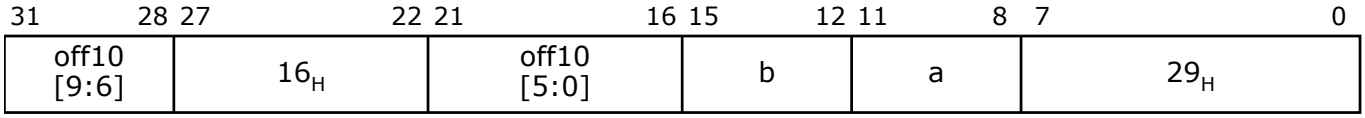

```
EA = A[b] + sign_ext(off10);
A[a] = M(EA, word);
```

LD.A A[a], P[b] (BO) (Bit-reverse Addressing Mode)



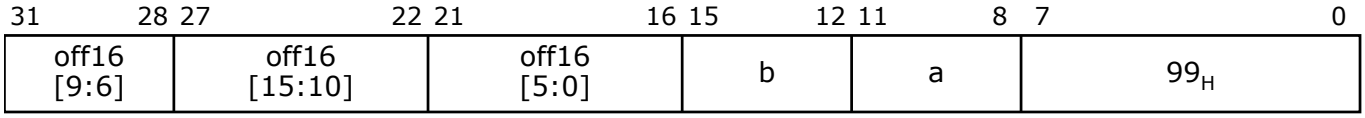

```
index = zero_ext(A[b+1][15:0]);
incr = zero_ext(A[b+1][31:16]);
EA = A[b] + index;
A[a] = M(EA, word);
new_index = reverse16(reverse16(index) + reverse16(incr));
A[b+1] = {incr[15:0], new_index[15:0]};
```

LD.A A[a], P[b], off10 (BO) (Circular Addressing Mode)



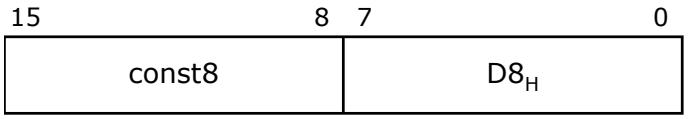

```
index = zero_ext(A[b+1][15:0]);
length = zero_ext(A[b+1][31:16]);
EA = A[b] + index;
A[a] = M(EA, word);
new_index = index + sign_ext(off10);
new_index = (new_index < 0) ? (new_index + length) : (new_index % length);
A[b+1] = {length[15:0], new_index[15:0]};
```

LD.A A[a], A[b], off16 (BOL) (Base + Long Offset Addressing Mode)



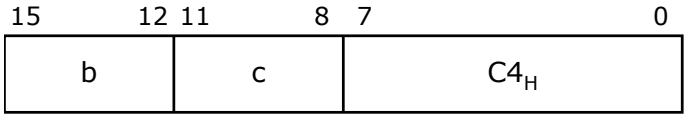

```
EA = A[b] + sign_ext(off16);
A[a] = M(EA, word);
```

LD.A A[15], A[10], const8 (SC)



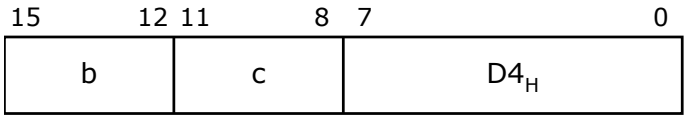

```
A[15] = M(A[10] + zero_ext(4 * const8), word);
```

LD.A A[c], A[b] (SLR) (Post-increment Addressing Mode)



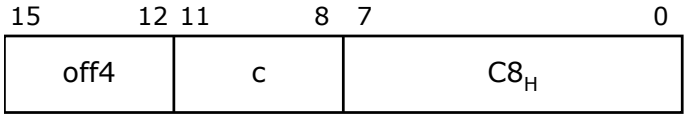
```
A[c] = M(A[b], word);
A[b] = A[b] + 4;
```

LD.A A[c], A[b] (SLR)



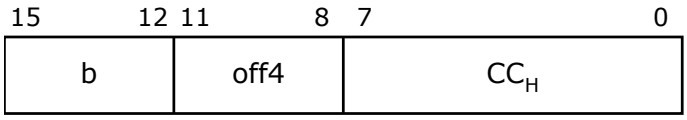

```
A[c] = M(A[b], word);
```

LD.A A[c], A[15], off4 (SLRO)




```
A[c] = M(A[15] + zero_ext(4 * off4), word);
```

LD.A A[15], A[b], off4 (SRO)




```
A[15] = M(A[b] + zero_ext(4 * off4), 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    a5, [a0+]4
ld.a    a2, [+a0]8
ld.a    a0, [a1]
ld.a    a3, [a6/a7+r]
ld.a    a3, [p6+r]
ld.a    a1, [a8/a9+c]
ld.a    a1, [p8+c]
```

```
ld.a    a15, [a10]2
ld.a    a5,  [a2+]
ld.a    a5,  [a2]
ld.a    a2,  [a15]4
ld.a    a15, [a2]4
```

See Also

LD.B, LD.BU, LD.D, LD.DA, LD.DD, LD.H, LD.HU, LD.Q, LD.W

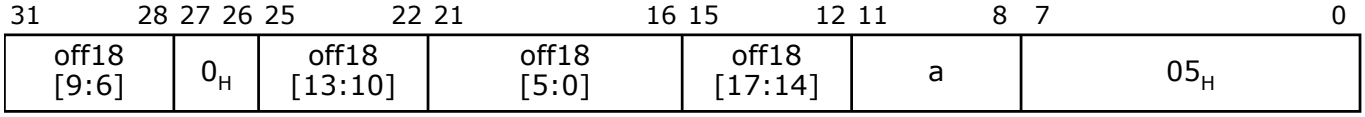
3.1.141 LD.B

Load Byte

Description

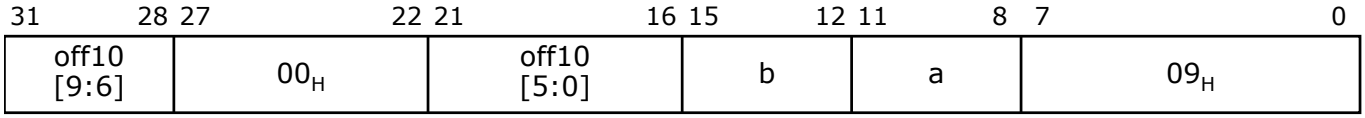
Load the byte contents of the memory location specified by the effective address, sign-extended into data register D[a].

LD.B D[a], off18 (ABS) (Absolute Addressing Mode)



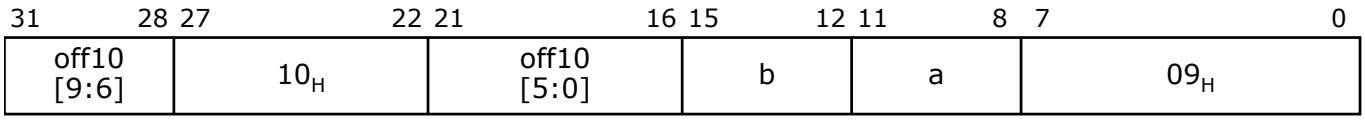

```
EA = {off18[17:14], 14b'0, off18[13:0]};
D[a] = sign_ext(M(EA, byte));
```

LD.B D[a], A[b], off10 (BO) (Post-increment Addressing Mode)



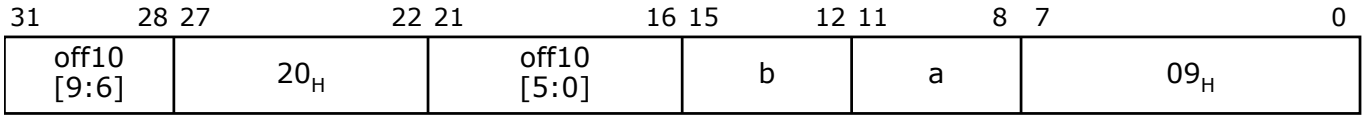

```
EA = A[b];
D[a] = sign_ext(M(EA, byte));
A[b] = EA + sign_ext(off10);
```

LD.B D[a], A[b], off10 (BO) (Pre-increment Addressing Mode)



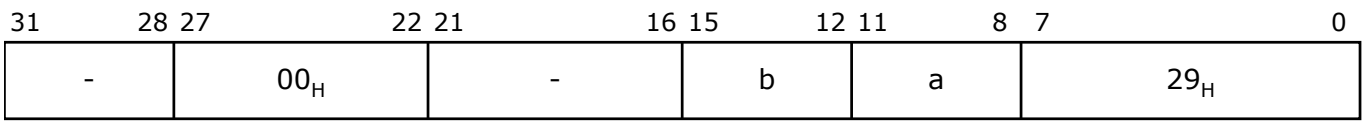

```
EA = A[b] + sign_ext(off10);
D[a] = sign_ext(M(EA, byte));
A[b] = EA;
```

LD.B D[a], A[b], off10 (BO) (Base + Short Offset Addressing Mode)




```
EA = A[b] + sign_ext(off10);
D[a] = sign_ext(M(EA, byte));
```

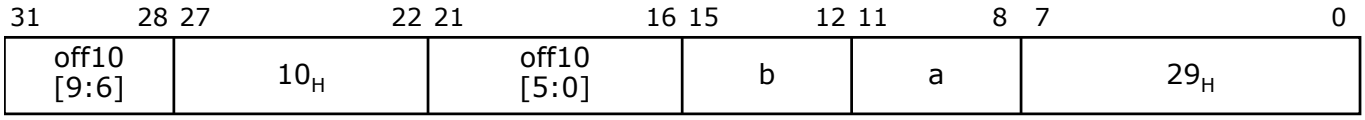
LD.B D[a], P[b] (BO) (Bit-reverse Addressing Mode)




```
index = zero_ext(A[b+1][15:0]);
incr = zero_ext(A[b+1][31:16]);
```

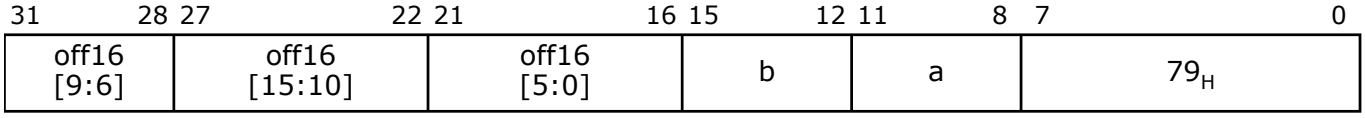
```
EA = A[b] + index;
D[a] = sign_ext(M(EA, byte));
new_index = reverse16(reverse16(index) + reverse16(incr));
A[b+1] = {incr[15:0], new_index[15:0]};
```

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], off16 (BOL) (Base + Long Offset Addressing Mode)




```
EA = A[b] + sign_ext(off16);
D[a] = sign_ext(M(EA, byte));
```

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.b    d0, [a0+]4
ld.b    d0, [+a0]8
ld.b    d0, [a0]2
ld.b    d0, [a0/a1+r]
ld.b    d0, [p0+r]
ld.b    d0, [a0/a1+c]
ld.b    d0, [p0+c]
```

See Also

LD.A, LD.BU, LD.D, LD.DA, LD.DD, LD.H, LD.HU, LD.Q, LD.W

3.1.142 LD.BU

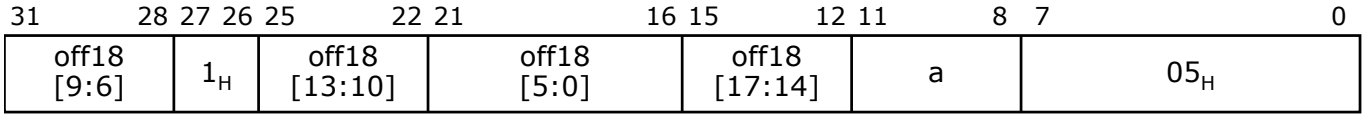
Load Byte Unsigned

Description

Load the byte contents of the memory location specified by the effective address, zero-extended, into data register D[a].

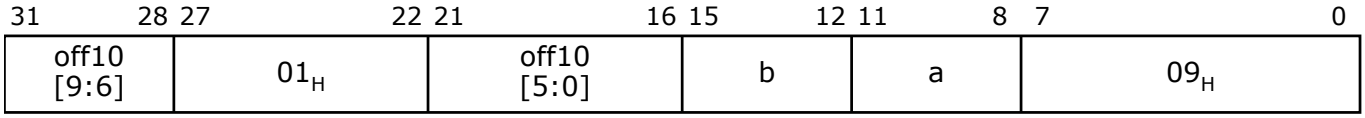
Load the byte contents of the memory location specified by the effective address, zero-extended, into either data register D[a] or D[15].

LD.BU D[a], off18 (ABS) (Absolute Addressing Mode)



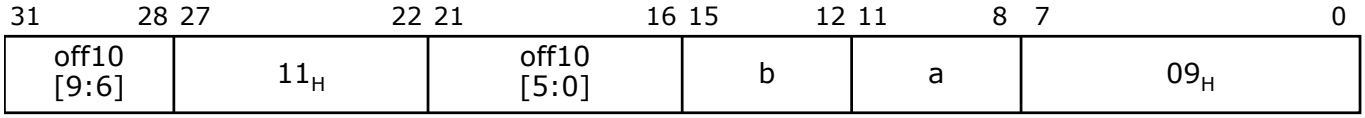

```
EA = {off18[17:14], 14b'0, off18[13:0]};
D[a] = zero_ext(M(EA, byte));
```

LD.BU D[a], A[b], off10 (BO) (Post-increment Addressing Mode)



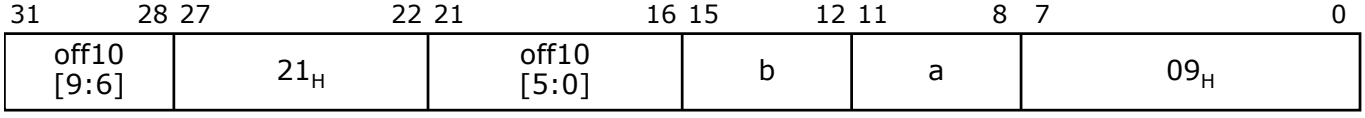

```
EA = A[b];
D[a] = zero_ext(M(EA, byte));
A[b] = EA+sign_ext(off10);
```

LD.BU D[a], A[b], off10 (BO) (Pre-increment Addressing Mode)



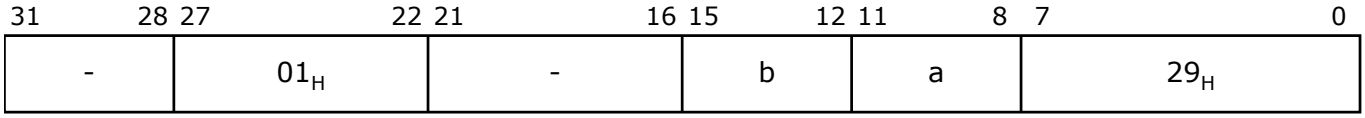

```
EA = A[b] + sign_ext(off10);
D[a] = zero_ext(M(EA, byte));
A[b] = EA;
```

LD.BU D[a], A[b], off10 (BO) (Base + Short Offset Addressing Mode)



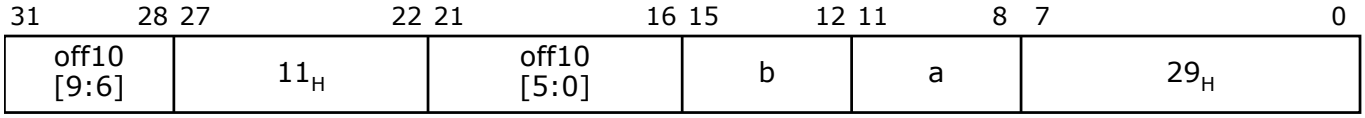

```
EA = A[b] + sign_ext(off10);
D[a] = zero_ext(M(EA, byte));
```

LD.BU D[a], P[b] (BO) (Bit-reverse Addressing Mode)



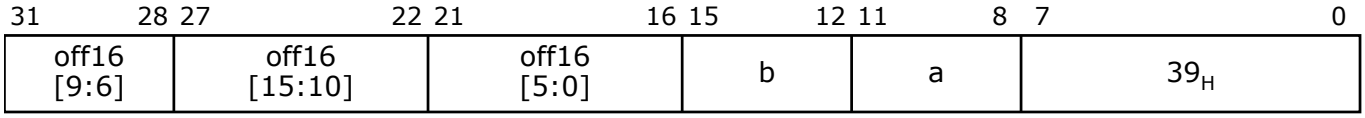

```
index = zero_ext(A[b+1][15:0]);
incr = zero_ext(A[b+1][31:16]);
EA = A[b] + index;
D[a] = zero_ext(M(EA, byte));
new_index = reverse16(reverse16(index) + reverse16(incr));
A[b+1] = {incr[15:0], new_index[15:0]};
```

LD.BU D[a], P[b], off10 (BO) (Circular Addressing Mode)



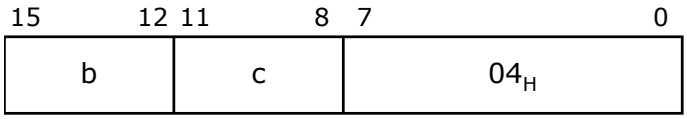

```
index = zero_ext(A[b+1][15:0]);
length = zero_ext(A[b+1][31:16]);
EA = A[b] + index;
D[a] = zero_ext(M(EA, byte));
new_index = index + sign_ext(off10);
new_index = (new_index < 0) ? (new_index + length) : (new_index % length);
A[b+1] = {length[15:0], new_index[15:0]};
```

LD.BU D[a], A[b], off16 (BOL) (Base + Long Offset Addressing Mode)



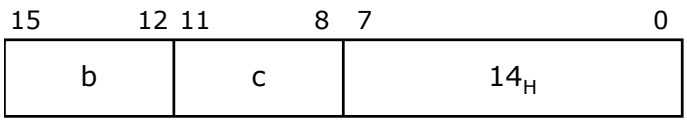

```
EA = A[b] + sign_ext(off16);
D[a] = zero_ext(M(EA, byte));
```

LD.BU D[c], A[b] (SLR) (Post-increment Addressing Mode)



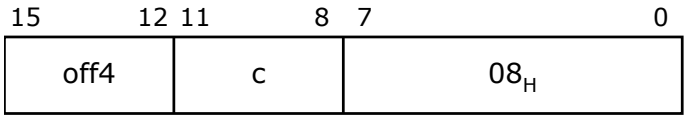

```
D[c] = zero_ext(M(A[b], byte));
A[b] = A[b] + 1;
```

LD.BU D[c], A[b] (SLR)



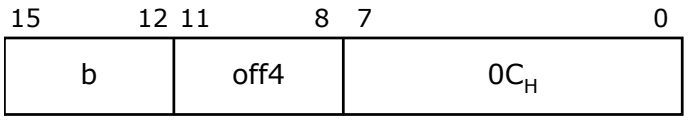
```
D[c] = zero_ext(M(A[b], byte));
```

LD.BU D[c], A[15], off4 (SLRO)




```
D[c] = zero_ext(M(A[15] + zero_ext(off4), byte));
```

LD.BU D[15], A[b], off4 (SRO)




```
D[15] = zero_ext(M(A[b] + zero_ext(off4), byte));
```

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.bu    d0, [a0+]4
ld.bu    d0, [+a0]8
ld.bu    d0, [a0]2
ld.bu    d0, [a0/a1+r]
ld.bu    d0, [p0+r]
ld.bu    d0, [a0/a1+c]
ld.bu    d0, [p0+c]
```

```
ld.bu    d0, [a0]
ld.bu    d0, [a15]8
ld.bu    d15, [a0]4
```

See Also

LD.A, LD.B, LD.D, LD.DA, LD.DD, LD.H, LD.HU, LD.Q, LD.W

3.1.143 LD.D

Load Double-word

Description

Load the double-word contents of the memory location specified by the effective address into the extended data register E[a]. The least-significant word of the double-word value is loaded into the even register (D[a]) and the most-significant word is loaded into the odd register (D[a+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]	1_H	off18 [13:10]	off18 [5:0]	off18 [17:14]	a	85_H

```
EA = {off18[17:14], 14b'0, off18[13:0]};
E[a] = M(EA, doubleword);
```

LD.D E[a], A[b], off10 (BO) (Post-increment Addressing Mode)

31 28	27 22	21 16	15 12	11 8	7 0
off10 [9:6]	05_H	off10 [5:0]	b	a	09_H

```
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)

31 28	27 22	21 16	15 12	11 8	7 0
off10 [9:6]	15_H	off10 [5:0]	b	a	09_H

```
EA = A[b] + sign_ext(off10);
E[a] = M(EA, doubleword);
A[b] = EA;
```

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_H	off10 [5:0]	b	a	09_H

```
EA = A[b] + sign_ext(off10);
E[a] = M(EA, doubleword);
```

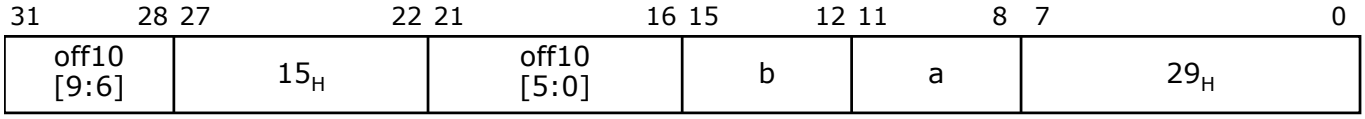
LD.D E[a], P[b] (BO) (Bit-reverse Addressing Mode)

31 28	27 22	21 16	15 12	11 8	7 0
-	05_H	-	b	a	29_H

```
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)




```
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]};
```

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.d    e0, [a10+]4
ld.d    e0, [+a0]8
ld.d    e0, [a0]2
ld.d    e0, [a0/a1+r]
ld.d    e0, [p0+r]
ld.d    e0, [a0/a1+c]
ld.d    e0, [p0+c]
```

See Also

LD.A, LD.B, LD.BU, LD.DA, LD.DD, LD.H, LD.HU, LD.Q, LD.W

3.1.144 LD.DA

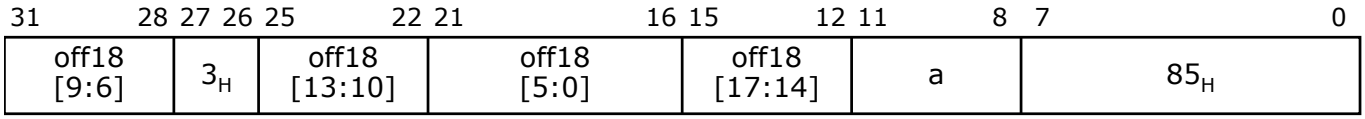
Load Double-word to Address Register

Description

Load the double-word contents of the memory location specified by the effective address into an address register pair P[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]).

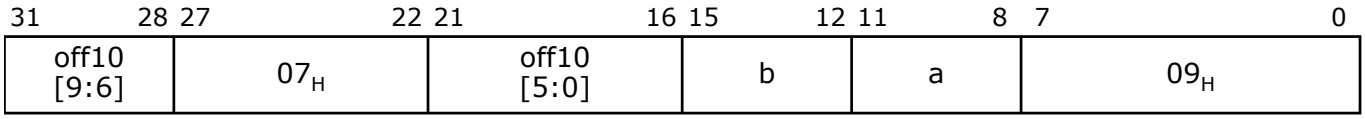
***Note**: If the target register is modified by the addressing mode, the result is undefined.*

LD.DA P[a], off18 (ABS) (Absolute Addressing Mode)



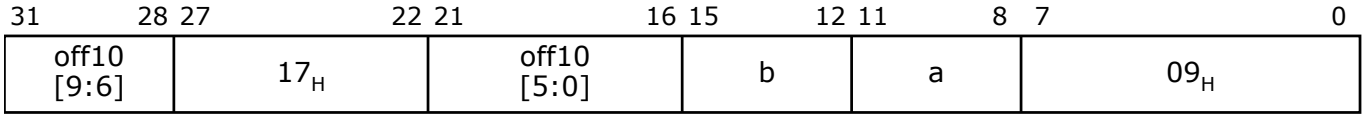

```
EA = {off18[17:14], 14b'0, off18[13:0]};
P[a] = M(EA, doubleword);
```

LD.DA P[a], A[b], off10 (BO) (Post-increment Addressing Mode)



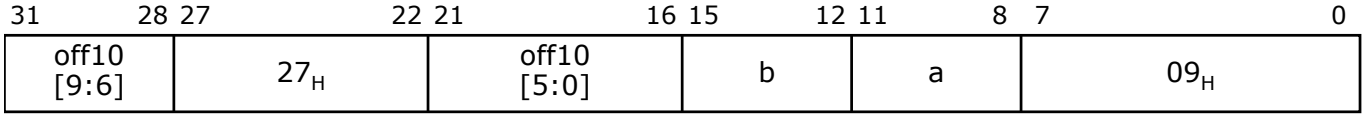

```
EA = A[b];
P[a] = M(EA, doubleword);
A[b] = EA + sign_ext(off10);
```

LD.DA P[a], A[b], off10 (BO) (Pre-increment Addressing Mode)



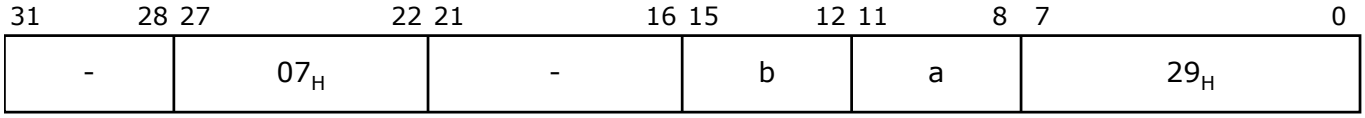

```
EA = A[b] + sign_ext(off10);
P[a] = M(EA, doubleword);
A[b] = EA;
```

LD.DA P[a], A[b], off10 (BO) (Base + Short Offset Addressing Mode)



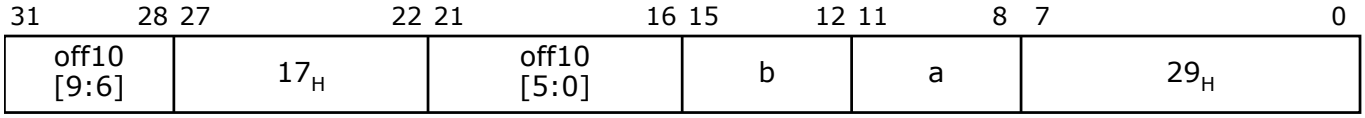

```
EA = A[b] + sign_ext(off10);
P[a] = M(EA, doubleword);
```

LD.DA P[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;
P[a] = M(EA, doubleword);
new_index = reverse16(reverse16(index) + reverse16(incr));
A[b+1] = {incr[15:0], new_index[15:0]};
```

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]};
```

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    a0/a1, _savedPointerBuffer
ld.da    a2/a3, [a0+]4
ld.da    a2/a3, [+a0]8
ld.da    a2/a3, [a0]2
ld.da    a2/a3, [a0/a1+r]
ld.da    p2, [p0+r]
ld.da    a2/a3, [a0/a1+c]
ld.da    p2, [p0+c]
```

See Also

LD.A, LD.B, LD.BU, LD.D, LD.DD, LD.H, LD.HU, LD.Q, LD.W

3.1.145 LD.DD

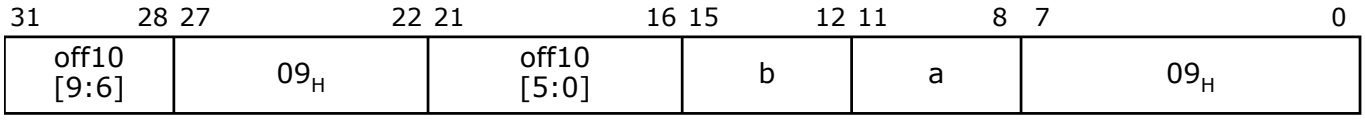
Load Double Double-word

Description

Load the quad-word (128-bit) contents of the memory location specified by the effective address, into the quad data register Q[a].

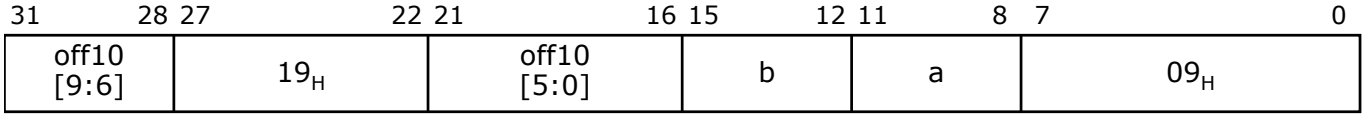
An operation with Q[a] where a != [0,4,8,12] will result in an OPD trap.

LD.DD Q[a] A[b], off10 (BO) (Post-increment Addressing Mode)



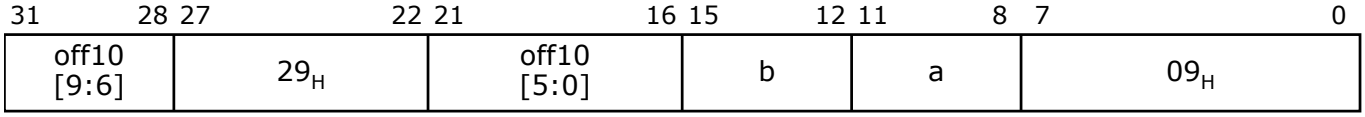

```
EA = A[b];
Q[a] = M(EA, quadword);
A[b] = EA + sign_ext(off10);
```

LD.DD Q[a], A[b], off10 (BO) (Pre-increment Addressing Mode)




```
EA = A[b] + sign_ext(off10);
Q[a] = M(EA, quadword);
A[b] = EA;
```

LD.DD Q[a], A[b], off10 (BO) (Base + Short Offset Addressing Mode)




```
EA = A[b] + sign_ext(off10);
Q[a] = M(EA, quadword);
```

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.dd    q0, [a0]; Load 128-bit
ld.dd    q4, [a0+]8; Load 128-bit
ld.dd    q8, [+a0]8; Load 128-bit
```

See Also

LD.A, LD.B, LD.BU, LD.D, LD.DA, LD.H, LD.HU, LD.Q, LD.W, ST.DD

3.1.146 LD.H

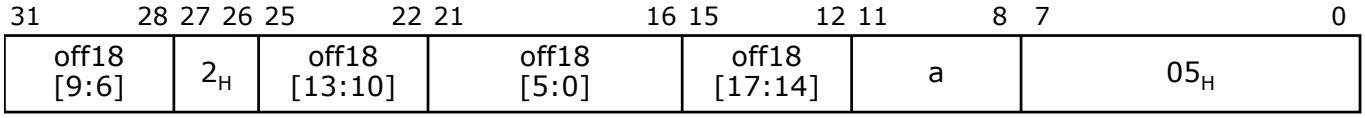
Load Half-word

Description

Load the half-word contents of the memory location specified by the effective address, sign-extended, into data register D[a].

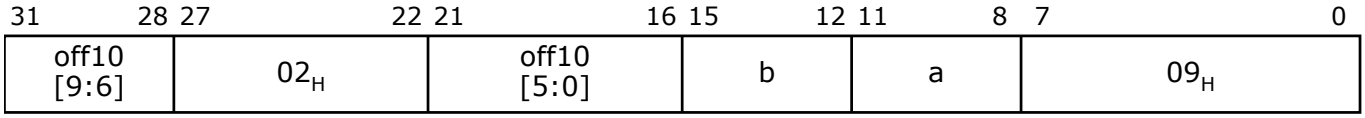
Load the half-word contents of the memory location specified by the effective address, sign-extended, into either data register D[a] or D[15].

LD.H D[a], off18 (ABS) (Absolute Addressing Mode)



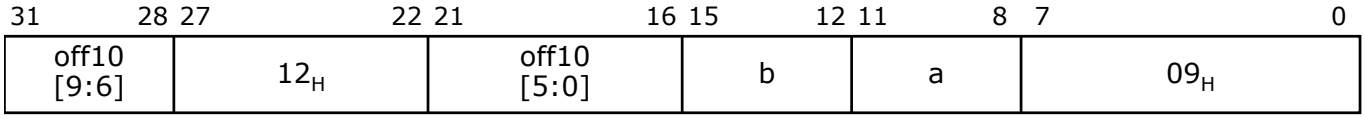

```
EA = {off18[17:14], 14b'0, off18[13:0]};
D[a] = sign_ext(M(EA, halfword));
```

LD.H D[a], A[b], off10 (BO) (Post-increment Addressing Mode)



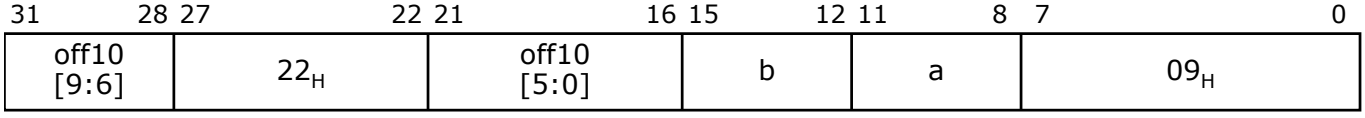

```
EA = A[b];
D[a] = sign_ext(M(EA, halfword));
A[b] = EA + sign_ext(off10);
```

LD.H D[a], A[b], off10 (BO) (Pre-increment Addressing Mode)



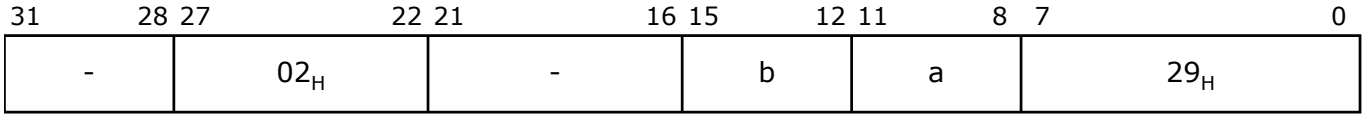

```
EA = A[b] + sign_ext(off10);
D[a] = sign_ext(M(EA, halfword));
A[b] = EA;
```

LD.H D[a], A[b], off10 (BO) (Base + Short Offset Addressing Mode)



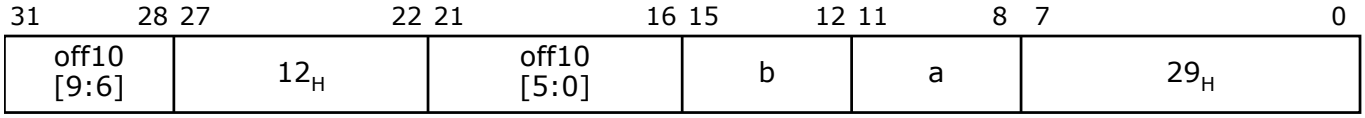

```
EA = A[b] + sign_ext(off10);
D[a] = sign_ext(M(EA, halfword));
```

LD.H D[a], P[b] (BO) (Bit-reverse Addressing Mode)



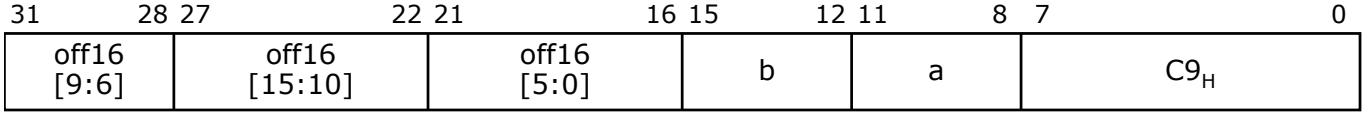

```
index = zero_ext(A[b+1][15:0]);
incr = zero_ext(A[b+1][31:16]);
EA = A[b] + index;
D[a] = sign_ext(M(EA, halfword));
new_index = reverse16(reverse16(index) + reverse16(incr));
A[b+1] = {incr[15:0], new_index[15:0]};
```

LD.H D[a], P[b], off10 (BO) (Circular Addressing Mode)



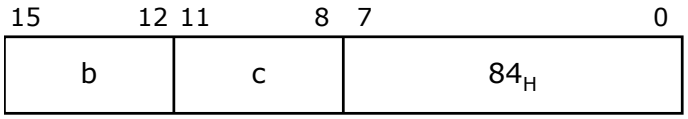

```
index = zero_ext(A[b+1][15:0]);
length = zero_ext(A[b+1][31:16]);
EA = A[b] + index;
D[a] = sign_ext(M(EA, halfword));
new_index = index + sign_ext(off10);
new_index = (new_index < 0) ? (new_index + length) : (new_index % length);
A[b+1] = {length[15:0], new_index[15:0]};
```

LD.H D[a], A[b], off16 (BOL) (Base + Long Offset Addressing Mode)



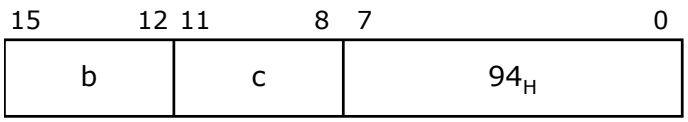

```
EA = A[b] + sign_ext(off16);
D[a] = sign_ext(M(EA, halfword));
```

LD.H D[c], A[b] (SLR) (Post-increment Addressing Mode)



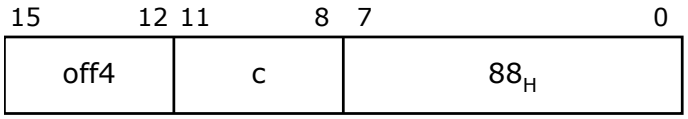

```
D[c] = sign_ext(M(A[b], half-word));
A[b] = A[b] + 2;
```

LD.H D[c], A[b] (SLR)



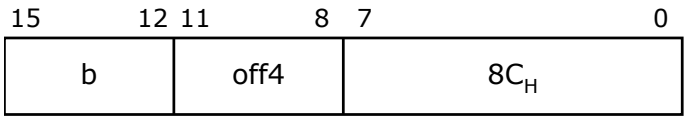
```
D[c] = sign_ext(M(A[b], halfword));
```

LD.H D[c], A[15], off4 (SLRO)




```
D[c] = sign_ext(M(A[15] + zero_ext(2 * off4), half-word));
```

LD.H D[15], A[b], off4 (SRO)




```
D[15] = sign_ext(M(A[b] + zero_ext(2 * off4), 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+]4
ld.h    d0, [+a0]8
ld.h    d0, [a0]2
ld.h    d0, [a0/a1+r]
ld.h    d0, [p0+r]
ld.h    d0, [a0/a1+c]
ld.h    d0, [p0+c]
```

```
ld.h    d0, [a0]
ld.h    d0, [a15]8
ld.h    d15, [a0]4
```

See Also

LD.A, LD.B, LD.BU, LD.D, LD.DA, LD.DD, LD.HU, LD.Q, LD.W

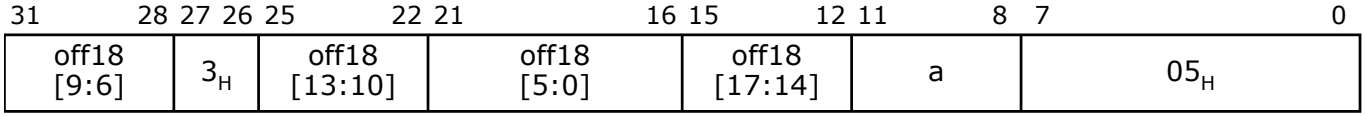
3.1.147 LD.HU

Load Half-word Unsigned

Description

Load the half-word contents of the memory location specified by the effective address, zero-extended, into data register D[a].

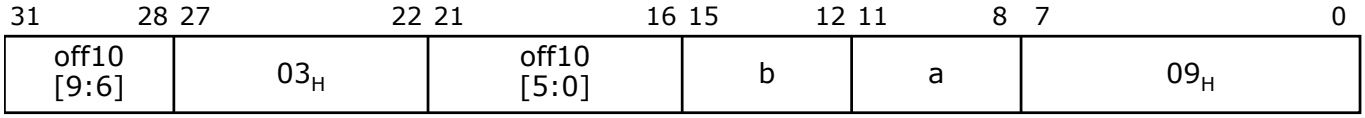
LD.HU D[a], off18 (ABS) (Absolute Addressing Mode)



31 28	27 26	25 22	21 16	15 12	11 8	7 0
off18 [9:6]	3_H	off18 [13:10]	off18 [5:0]	off18 [17:14]	a	05_H

```
EA = {off18[17:14], 14b'0, off18[13:0]};
D[a] = zero_ext(M(EA, halfword));
```

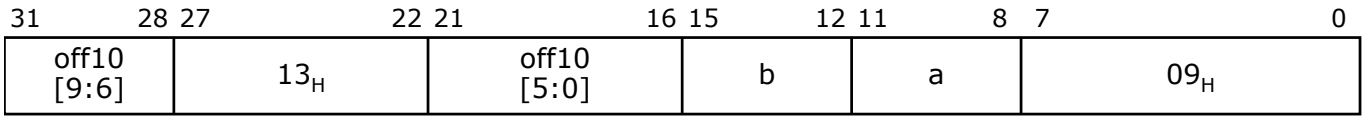
LD.HU D[a], A[b], off10 (BO) (Post-increment Addressing Mode)



31 28	27 22	21 16	15 12	11 8	7 0
off10 [9:6]	03_H	off10 [5:0]	b	a	09_H

```
EA = A[b];
D[a] = zero_ext(M(EA, halfword));
A[b] = EA + sign_ext(off10);
```

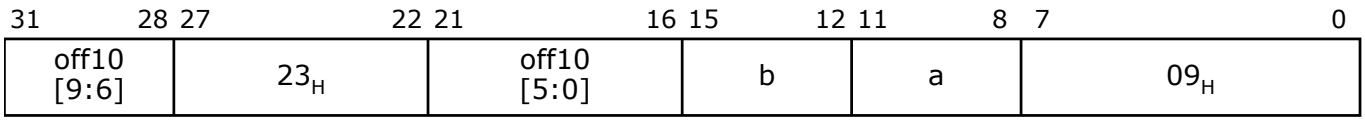
LD.HU D[a], A[b], off10 (BO) (Pre-increment Addressing Mode)



31 28	27 22	21 16	15 12	11 8	7 0
off10 [9:6]	13_H	off10 [5:0]	b	a	09_H

```
EA = A[b] + sign_ext(off10);
D[a] = zero_ext(M(EA, halfword));
A[b] = EA;
```

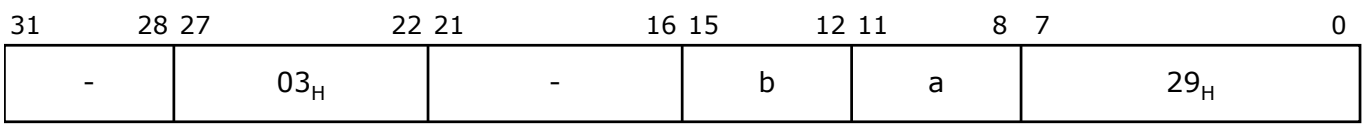
LD.HU D[a], A[b], off10 (BO) (Base + Short Offset Addressing Mode)



31 28	27 22	21 16	15 12	11 8	7 0
off10 [9:6]	23_H	off10 [5:0]	b	a	09_H

```
EA = A[b] + sign_ext(off10);
D[a] = zero_ext(M(EA, halfword));
```

LD.HU D[a], P[b] (BO) (Bit-reverse Addressing Mode)

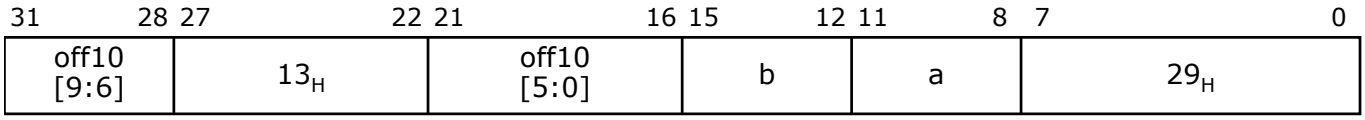


31 28	27 22	21 16	15 12	11 8	7 0
-	03_H	-	b	a	29_H

```
index = zero_ext(A[b+1][15:0]);
incr = zero_ext(A[b+1][31:16]);
```

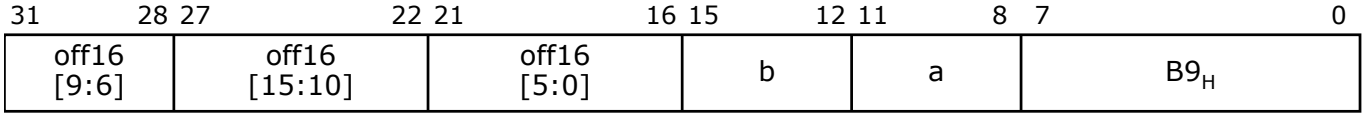
```
EA = A[b] + index;
D[a] = zero_ext(M(EA, halfword));
new_index = reverse16(reverse16(index) + reverse16(incr));
A[b+1] = {incr[15:0], new_index[15:0]};
```

LD.HU D[a], P[b], off10 (BO) (Circular Addressing Mode)




```
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, 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], off16 (BOL) (Base + Long Offset Addressing Mode)




```
EA = A[b] + sign_ext(off16);
D[a] = zero_ext(M(EA, halfword));
```

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.hu    d0, [a0+]4
ld.hu    d0, [+a0]8
ld.hu    d0, [a0]2
ld.hu    d0, [a0/a1+r]
ld.hu    d0, [p0+r]
ld.hu    d0, [a0/a1+c]
ld.hu    d0, [p0+c]
```

See Also

LD.A, LD.B, LD.BU, LD.D, LD.DA, LD.DD, LD.H, LD.Q, LD.W

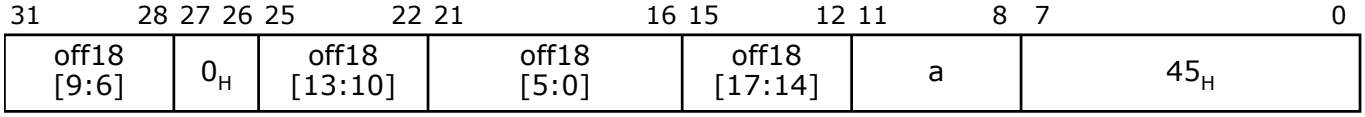
3.1.148 LD.Q

Load Half-word Signed Fraction

Description

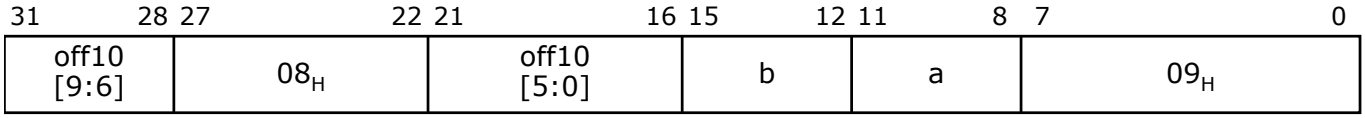
Load the half-word contents of the memory location specified by the effective address into the most-significant half-word of data register D[a], setting the 16 least-significant bits of D[a] to zero.

LD.Q D[a], off18 (ABS) (Absolute Addressing Mode)



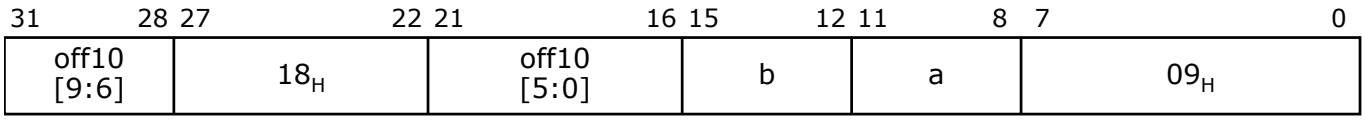

```
EA = {off18[17:14],14b'0,off18[13:0]};
D[a] = {M(EA, halfword), 16'h0000};
```

LD.Q D[a], A[b], off10 (BO) (Post-increment Addressing Mode)



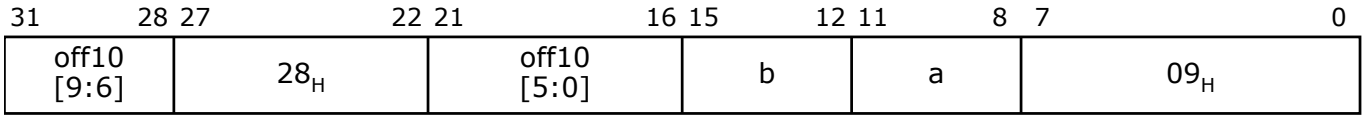

```
EA = A[b];
D[a] = {M(EA, halfword), 16'h0000};
A[b] = EA + sign_ext(off10);
```

LD.Q D[a], A[b], off10 (BO) (Pre-increment Addressing Mode)



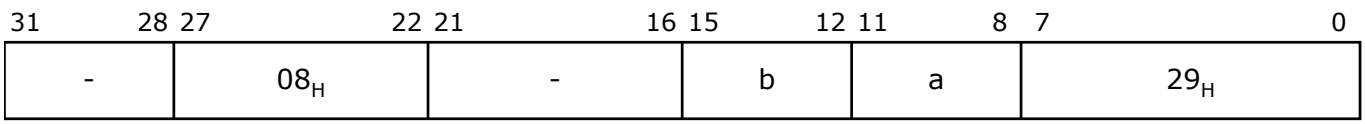

```
EA = A[b] + sign_ext(off10);
D[a] = {M(EA, halfword), 16'h0000};
A[b] = EA;
```

LD.Q D[a], A[b], off10 (BO) (Base + Short Offset Addressing Mode)




```
EA = A[b] + sign_ext(off10);
D[a] = {M(EA, halfword), 16'h0000};
```

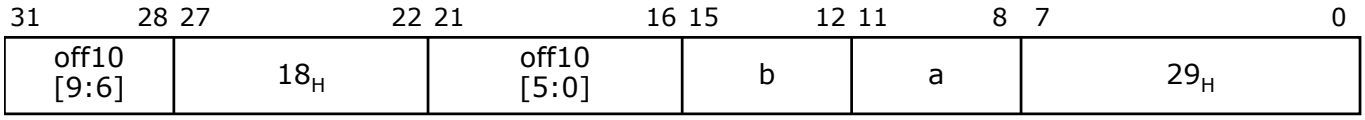
LD.Q D[a], P[b] (BO) (Bit-reverse Addressing Mode)




```
index = zero_ext(A[b+1][15:0]);
incr = zero_ext(A[b+1][31:16]);
```

```
EA = A[b] + index;
D[a] = {M(EA, 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)




```
index = zero_ext(A[b+1][15:0]);
length = zero_ext(A[b+1][31:16]);
EA = 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]};
```

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    d0, [a0+]4
ld.q    d0, [+a0]8
ld.q    d0, [a0]2
ld.q    d0, [a0/a1+r]
ld.q    d0, [p0+r]
ld.q    d0, [a0/a1+c]
ld.q    d0, [p0+c]
```

See Also

LD.A, LD.B, LD.BU, LD.D, LD.DA, LD.DD, LD.H, LD.HU, LD.W

3.1.149 LD.W

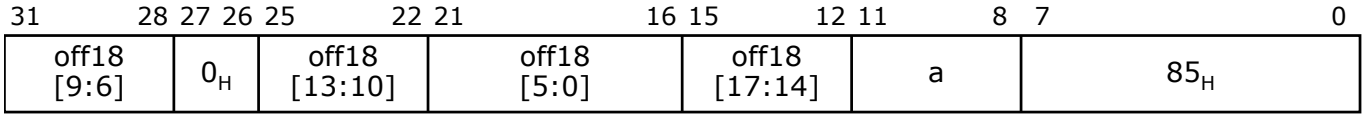
Load Word

Description

Load word contents of the memory location specified by the effective address into data register D[a].

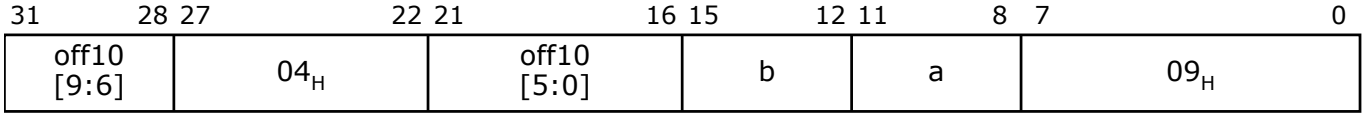
Load word contents of the memory location specified by the effective address into data register either D[a] or D[15].

LD.W D[a], off18 (ABS) (Absolute Addressing Mode)



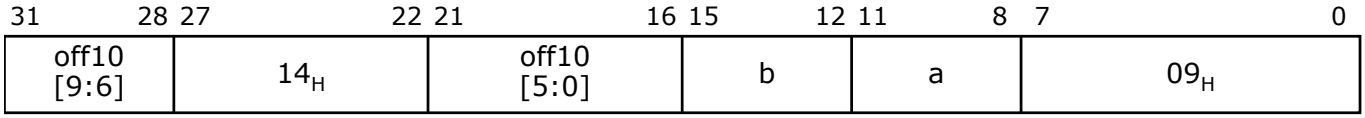

```
EA = {off18[17:14], 14b'0, off18[13:0]};
D[a] = M(EA, word);
```

LD.W D[a], A[b], off10 (BO) (Post-increment Addressing Mode)



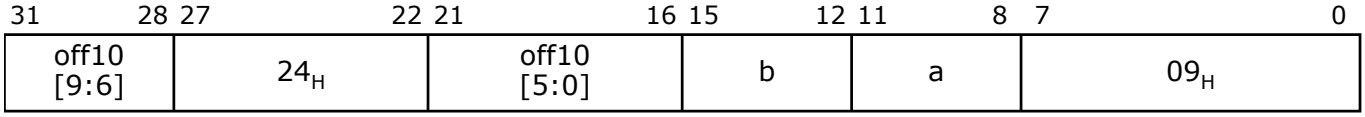

```
EA = A[b];
D[a] = M(EA, word);
A[b] = EA + sign_ext(off10);
```

LD.W D[a], A[b], off10 (BO) (Pre-increment Addressing Mode)



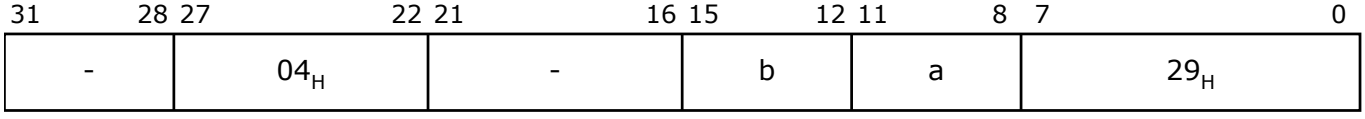

```
EA = A[b] + sign_ext(off10);
D[a] = M(EA, word);
A[b] = EA;
```

LD.W D[a], A[b], off10 (BO) (Base + Short Offset Addressing Mode)



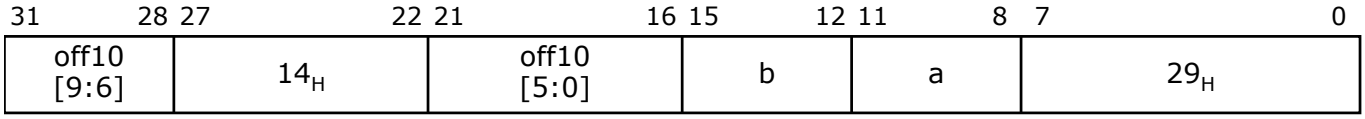

```
EA = A[b] + sign_ext(off10);
D[a] = M(EA, word);
```

LD.W D[a], P[b] (BO) (Bit-reverse Addressing Mode)



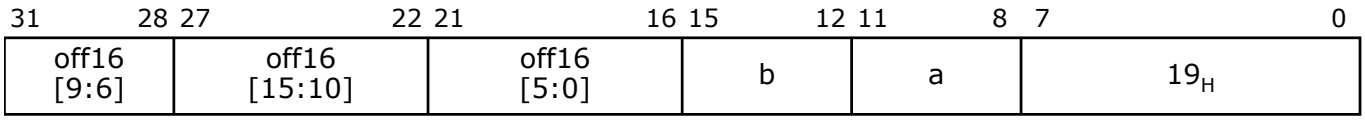
```
index = zero_ext(A[b+1][15:0]);
incr = zero_ext(A[b+1][31:16]);
EA = A[b] + index;
D[a] = M(EA, word);
new_index = reverse16(reverse16(index) + reverse16(incr));
A[b+1] = {incr[15:0], new_index[15:0]};
```

LD.W D[a], P[b], off10 (BO) (Circular Addressing Mode)



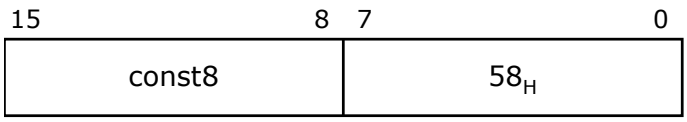

```
index = zero_ext(A[b+1][15:0]);
length = zero_ext(A[b+1][31:16]);
EA0 = A[b] + index;
EA2 = A[b] + (index + 2) % length;
D[a] = {M(EA2, halfword), M(EA0, halfword)};
new_index = index + sign_ext(off10);
new_index = (new_index < 0) ? (new_index + length) : (new_index % length);
A[b+1] = {length[15:0], new_index[15:0]};
```

LD.W D[a], A[b], off16 (BOL) (Base + Long Offset Addressing Mode)



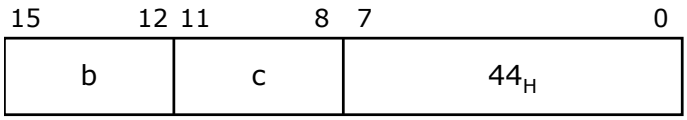

```
EA = A[b] + sign_ext(off16);
D[a] = M(EA, word);
```

LD.W D[15], A[10], const8 (SC)



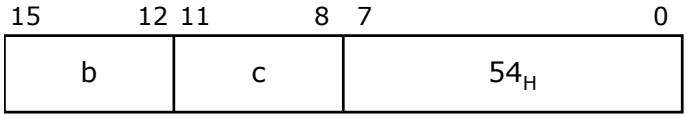

```
D[15] = M(A[10] + zero_ext(4 * const8), word);
```

LD.W D[c], A[b] (SLR) (Post-increment Addressing Mode)



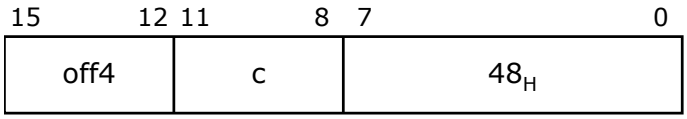

```
D[c] = M(A[b], word);
A[b] = A[b] + 4;
```

LD.W D[c], A[b] (SLR)



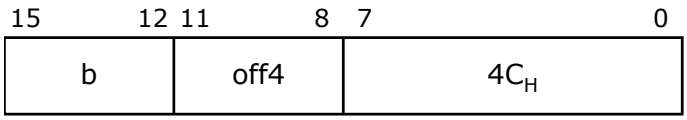

```
D[c] = M(A[b], word);
```

LD.W D[c], A[15], off4 (SLRO)




```
D[c] = M(A[15] + zero_ext(4 * off4), word);
```

LD.W D[15], A[b], off4 (SRO)




```
D[15] = M(A[b] + zero_ext(4 * off4), 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    d0, [a0+]4
ld.w    d0, [+a0]8
ld.w    d0, [a0]2
ld.w    d0, [a0/a1+r]
ld.w    d0, [p0+r]
ld.w    d0, [a0/a1+c]
ld.w    d0, [p0+c]
```

```
ld.w    d15, [a10]4
ld.w    d0, [a0+]
ld.w    d0, [a0]
ld.w    d0, [a15]8
ld.w    d15, [a0]4
```

See Also

LD.A, LD.B, LD.BU, LD.D, LD.DA, LD.DD, LD.H, LD.HU, LD.Q

3.1.150 LDLCX

Load Lower Context

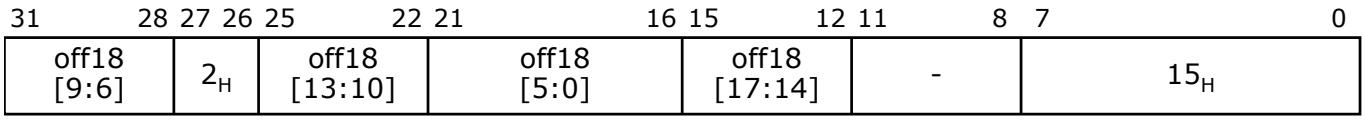
Description

Load the contents of the memory block specified by the effective address 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 (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).*

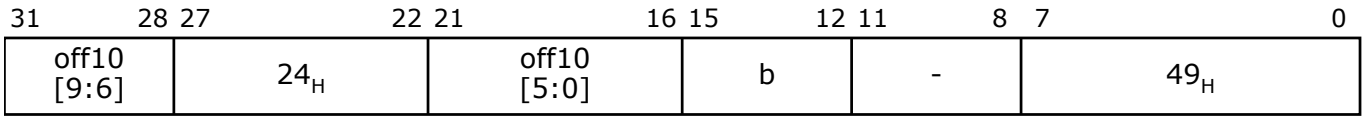
***Note**: This instruction may not be used to access peripheral space.*

LDLCX off18 (ABS) (Absolute Addressing Mode)




```
EA = {off18[17:14],14b'0,off18[13:0]};
{dummy, dummy, 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);
```

LDLCX A[b], off10 (BO) (Base + Short Offset Addressing Mode)




```
EA = A[b] + sign_ext(off10);
{dummy, dummy, 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);
```

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

```
ldlcx    [a0]32
```

See Also

LDUCX, RSLCX, STLCX, STUCX, SVLCX, BISR

3.1.151 LDMST

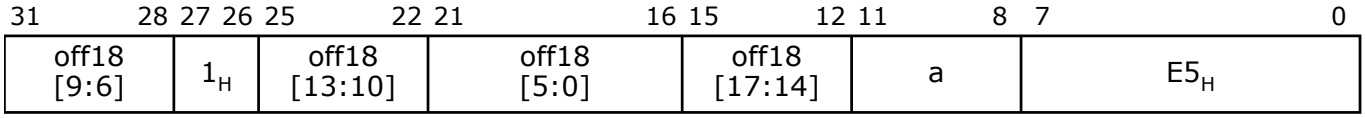
Load-Modify-Store

Description

The 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.

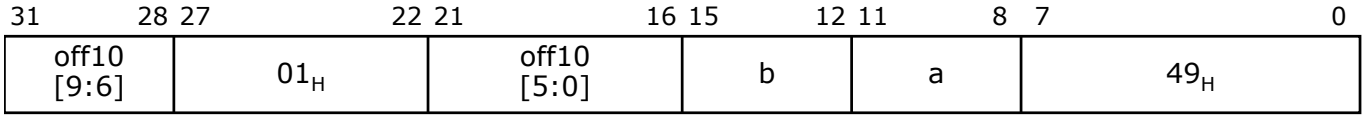
The M(EA, word) read and the M(EA, word) write are atomic.

LDMST off18, E[a] (ABS) (Absolute Addressing Mode)



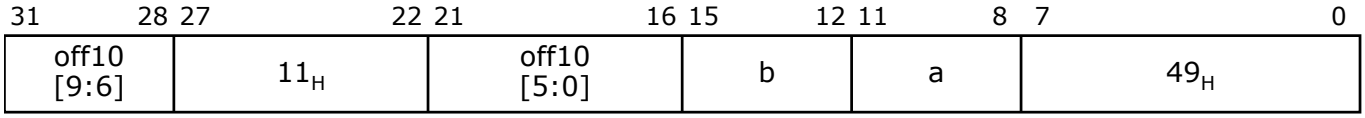

```
EA = {off18[17:14], 14b'0,off18[13:0]};
tmp = M(EA, word);
M(EA, word) = (tmp & ~D[a+1]) | (D[a] & D[a+1]);
```

LDMST A[b], off10, E[a] (BO) (Post-increment Addressing Mode)



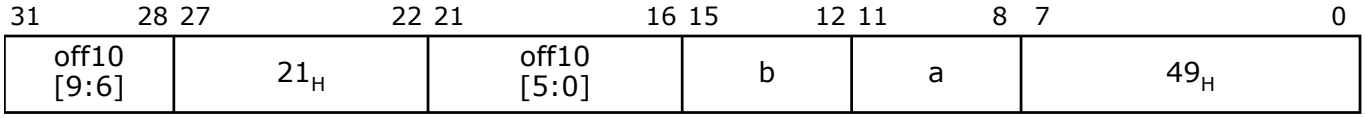

```
EA = A[b];
tmp = M(EA, word);
M(EA, word) = (tmp & ~D[a+1]) | (D[a] & D[a+1]);
A[b] = EA + sign_ext(off10);
```

LDMST A[b], off10, E[a] (BO) (Pre-increment Addressing Mode)



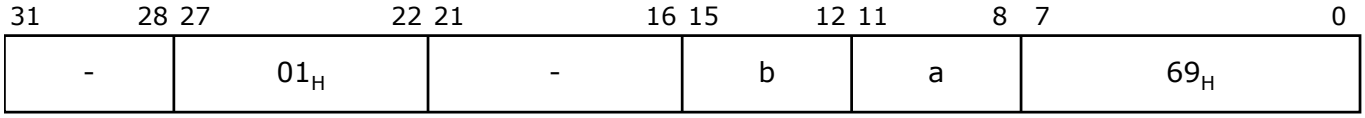

```
EA = A[b] + sign_ext(off10);
tmp = M(EA, word);
M(EA, word) = (tmp & ~D[a+1]) | (D[a] & D[a+1]);
A[b] = EA;
```

LDMST A[b], off10, E[a] (BO) (Base + Short Offset Addressing Mode)




```
EA = A[b] + sign_ext(off10);
tmp = M(EA, word);
M(EA, word) = (tmp & ~D[a+1]) | (D[a] & D[a+1]);
```

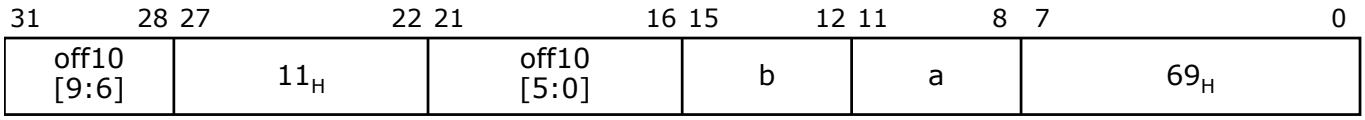
LDMST P[b], E[a] (BO) (Bit-reverse Addressing Mode)



31 – 28	27 – 22	21 – 16	15 – 12	11 – 8	7 – 0
-	01_H	-	b	a	69_H

```
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) = (tmp & ~D[a+1]) | (D[a] & D[a+1]);
new_index = reverse16(reverse16(index) + reverse16(incr));
A[b+1] = {incr[15:0], new_index[15:0]};
```

LDMST P[b], off10, E[a] (BO) (Circular Addressing Mode)



31 – 28	27 – 22	21 – 16	15 – 12	11 – 8	7 – 0
off10 [9:6]	11_H	off10 [5:0]	b	a	69_H

```
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) = (tmp & ~D[a+1]) | (D[a] & D[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]};
```

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

```
ldmst    [a0+]4, e0
ldmst    [+a0]8, e0
ldmst    [a0]4, e0
ldmst    [a0/a1+r], e0
ldmst    [p0+r], e0
ldmst    [a0/a1+c], e0
ldmst    [p0+c], e0
```

See Also

IMASK, ST.T, SWAP.W, SWAPMSK.W, CMPSWAP.W

3.1.152 LDUCX

Load Upper Context

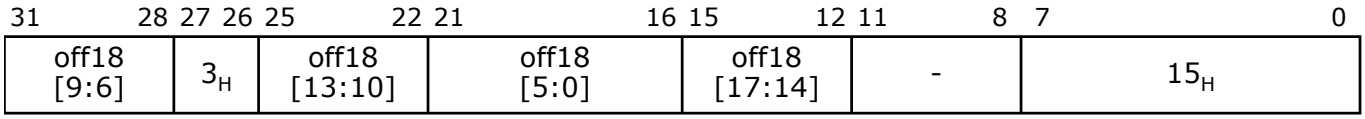
Description

Load the contents of the memory block specified by the effective address 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 address 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).*

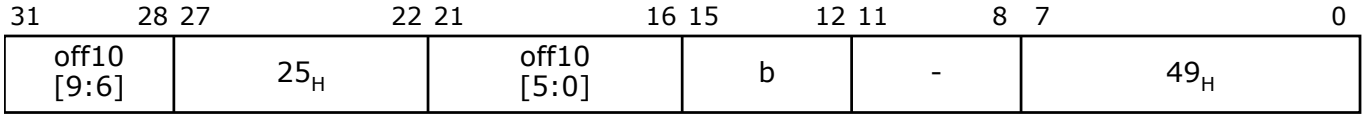
***Note**: This instruction may not be used to access peripheral space.*

LDUCX off18 (ABS) (Absolute Addressing Mode)




```
EA = {off18[17:14], 14b'0, off18[13:0]};
{dummy, dummy, 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);
```

LDUCX A[b], off10 (BO) (Base + Short Offset Addressing Mode)




```
EA = A[b][31:0] + sign_ext(off10);
{dummy, dummy, 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);
```

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

```
lducx    [a0]
```

See Also

LDLCX, RSLCX, STLCX, STUCX, SVLCX, LDUCX

3.1.153 LEA

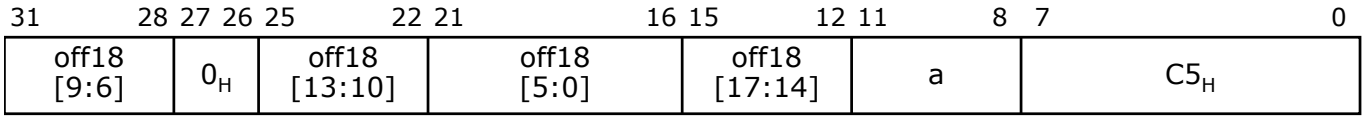
Load Effective Address

Description

Compute the absolute (effective) address defined by the addressing mode and put the result in address register A[a].

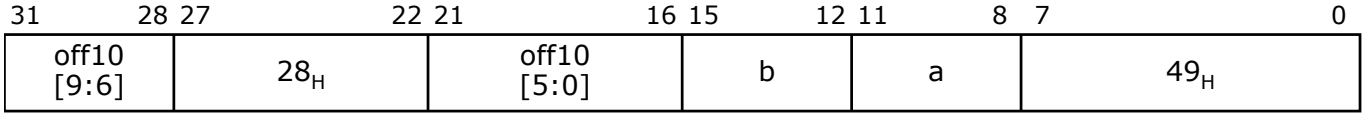
***Note**: The auto-increment addressing modes are not supported for this instruction.*

LEA A[a], off18 (ABS) (Absolute Addressing Mode)



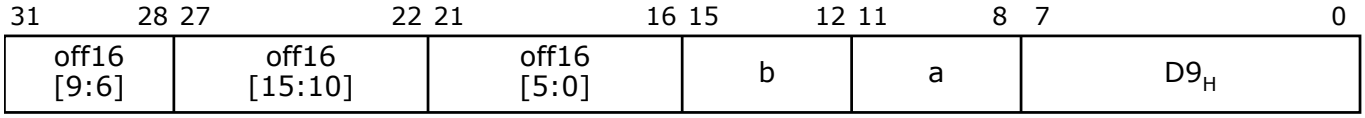

```
EA = {off18[17:14], 14b'0, off18[13:0]};
A[a] = EA[31:0];
```

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.

Examples

```
lea    a0, _absadd
lea    a7, NumberOfLoops
```

See Also

MOV.A, MOV.D, MOVH.A, LHA

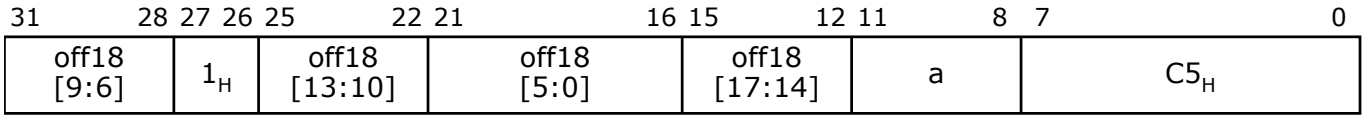
3.1.154 LHA

Load High Address

Description

Compute the effective address defined by the instruction and put the result in A[a].

LHA A[a], off18 (ABS) (Absolute Addressing Mode)




```
EA = {off18[17:0], 14b'0};
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.

Examples

```
lha    a7, NumberOfLoops
```

See Also

LEA

3.1.155 LOOP

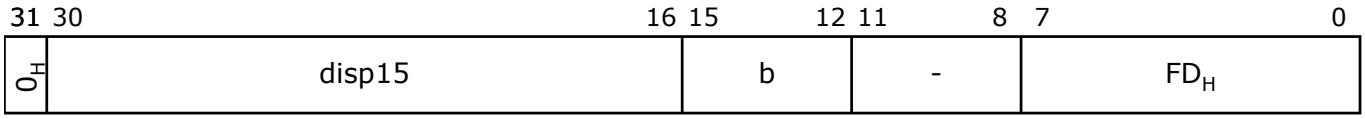
Loop

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.

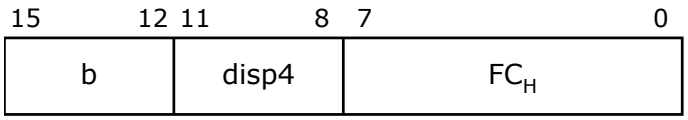
If address register A[b] is not equal to zero then add value specified by disp4, multiplied by two and one-extended to a 32-bit negative number, to the contents of PC and jump to that address. The address register is decremented unconditionally.

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'11111111111111111111111111, 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

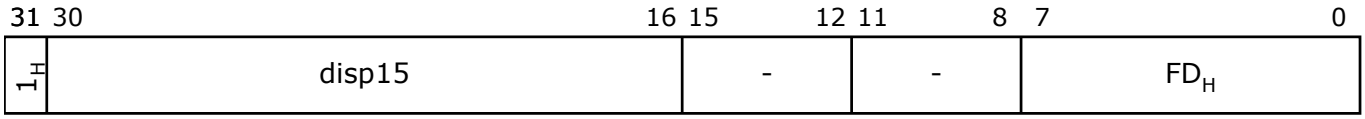
3.1.156 LOOPU

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, JRI, JNED, JNEI, LOOP

3.1.157 LSYNC

Synchronize Local Data

Description

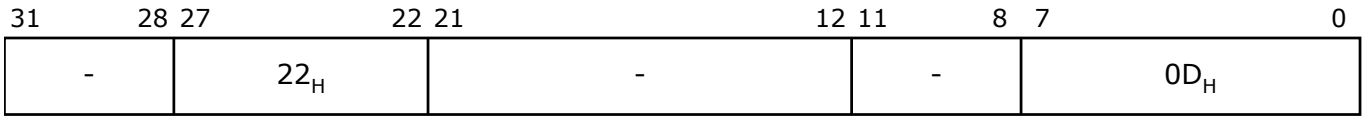
Forces all earlier data accesses to be commenced on the CPU interconnect interfaces before any data accesses associated with an instruction, semantically after the LSYNC are initiated.

***Note**: The Data Cache (DCACHE) is not invalidated by LSYNC.*

***Note**: To ensure memory consistency, a LSYNC or a DSYNC instruction must be executed prior to any access to an active CSA memory location.*

***Note**: As opposed to a DSYNC instruction, LSYNC will not wait for writes to complete at their target destination.*

LSYNC (SYS)



-

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

```
lsync
```

See Also

DSYNC, ISYNC

3.1.158 LT

Less Than

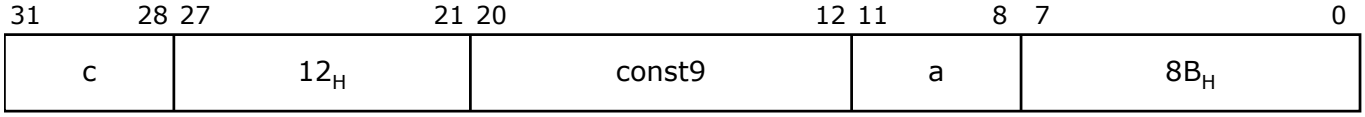
Description

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 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 32-bit signed integers. The const9 value is sign-extended.

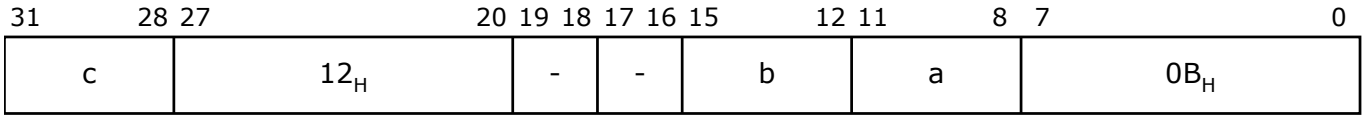
If the contents of data register D[a] are less than the contents of either data register D[b] (instruction format SRR) or const4 (instruction format SRC), set the least-significant bit of D[15] to one and clear the remaining bits to zero; otherwise clear all bits in D[15]. The operands are treated as 32-bit signed integers, and the const4 value is sign-extended.

LT D[c], D[a], const9 (RC)



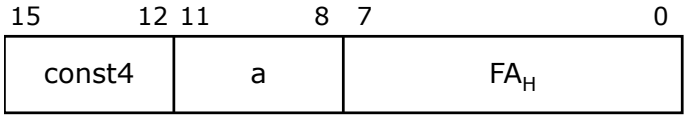

```
result = (D[a] < sign_ext(const9));
D[c] = zero_ext(result);
```

LT D[c], D[a], D[b] (RR)



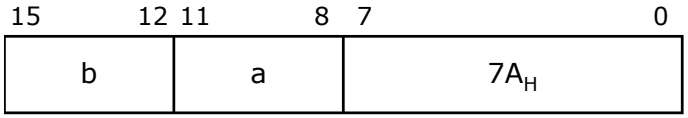

```
result = (D[a] < D[b]);
D[c] = zero_ext(result);
```

LT D[15], D[a], const4 (SRC)




```
result = (D[a] < sign_ext(const4));
D[15] = zero_ext(result);
```

LT D[15], D[a], D[b] (SRR)




```
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

```
lt    d3, d1, #126
lt    d3, d1, d2
```

```
lt    d15, d1, #6
lt    d15, d1, d2
```

See Also

EQ, GE, GE.U, LT.U, NE, EQANY.B, EQANY.H

3.1.159 LT.U

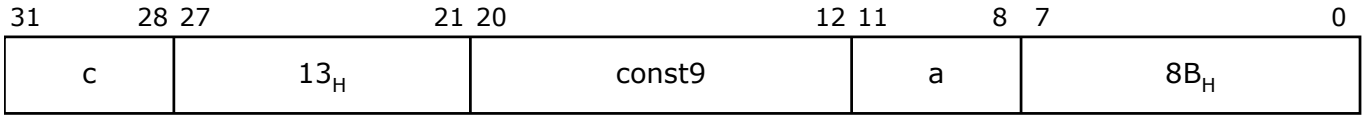
Less Than Unsigned

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 set the least-significant bit of D[c] to one and clear the remaining bits to zero; otherwise clear all bits in D[c].

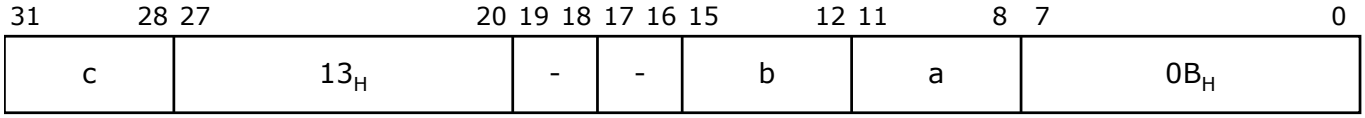
The operands are treated as 32-bit unsigned integers. The const9 value is zero-extended.

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




```
result = (D[a] < zero_ext(const9)); // unsigned
D[c] = zero_ext(result);
```

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




```
result = (D[a] < D[b]); // unsigned
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

```
lt.u    d3, d1, #253
lt.u    d3, d1, d2
```

See Also

EQ, GE, GE.U, LT, NE, EQANY.B, EQANY.H

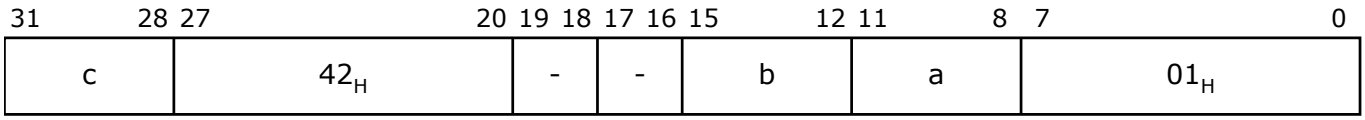
3.1.160 LT.A

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 32-bit unsigned integers.

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




```
result = (A[a] < A[b]); // unsigned
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

```
lt.a    d3, a4, a2
```

See Also

EQ.A, EQZ.A, GE.A, NE, NEZ.A

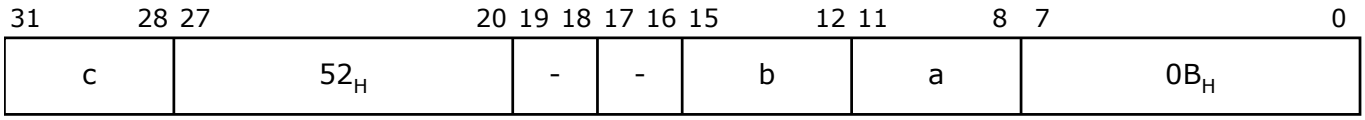
3.1.161 LT.B

Less Than Packed Byte

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 8-bit signed integers.

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




```
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;
```

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.b    d3, d1, d2
```

See Also

EQ.B, EQ.H, EQ.W, LT.BU, LT.H, LT.HU, LT.W, LT.WU

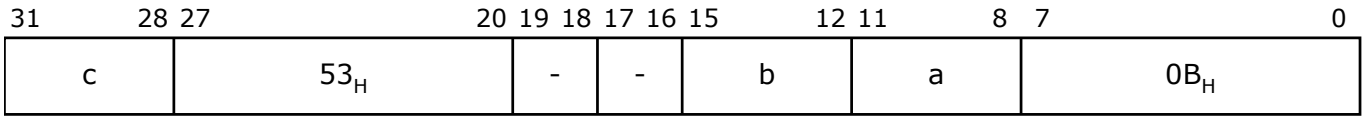
3.1.162 LT.BU

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 8-bit unsigned integers.

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




```
D[c][31:24] = (D[a][31:24] < D[b][31:24]) ? 8'hFF : 8'h00; // unsigned
D[c][23:16] = (D[a][23:16] < D[b][23:16]) ? 8'hFF : 8'h00; // unsigned
D[c][15:8] = (D[a][15:8] < D[b][15:8]) ? 8'hFF : 8'h00; // unsigned
D[c][7:0] = (D[a][7:0] < D[b][7:0]) ? 8'hFF : 8'h00; // 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.bu    d3, d1, d2
```

See Also

EQ.B, EQ.H, EQ.W, LT.B, LT.H, LT.HU, LT.W, LT.WU

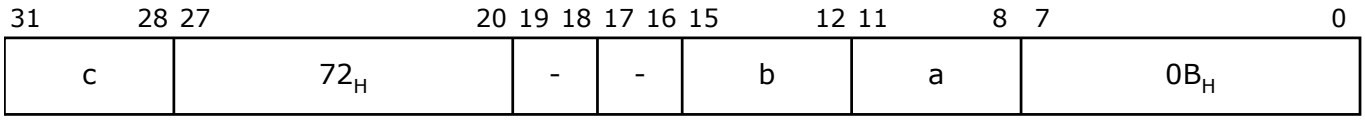
3.1.163 LT.H

Less Than Packed Half-word

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 16-bit signed 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;
```

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.h    d3, d1, d2
```

See Also

EQ.B, EQ.H, EQ.W, LT.B, LT.BU, LT.HU, LT.W, LT.WU

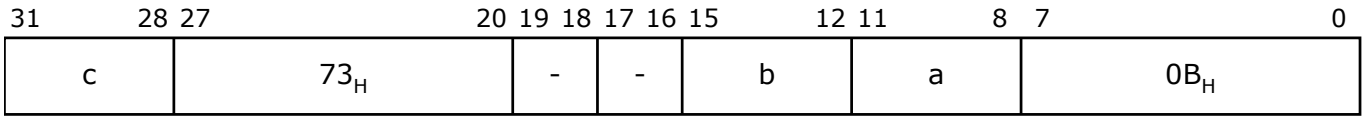
3.1.164 LT.HU

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 16-bit unsigned integers.

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; // unsigned
D[c][15:0] = (D[a][15:0] < D[b][15:0]) ? 16'hFFFF : 16'h0000; // 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.hu    d3, d1, d2
```

See Also

EQ.B, EQ.H, EQ.W, LT.B, LT.BU, LT.H, LT.W, LT.WU

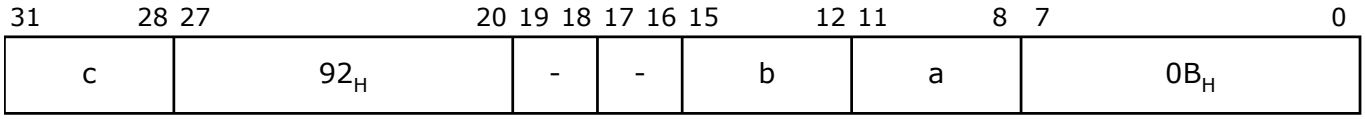
3.1.165 LT.W

Less Than Packed Word

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 32-bit signed integers.

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




```
D[c] = (D[a] < D[b]) ? 32'hFFFFFFFF : 32'h00000000;
```

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.w    d3, d1, d2
```

See Also

EQ.B, EQ.H, EQ.W, LT.B, LT.BU, LT.H, LT.HU, LT.WU

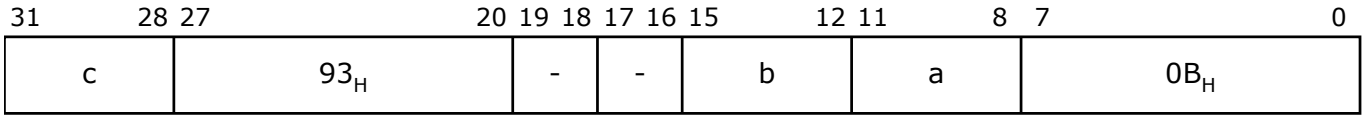
3.1.166 LT.WU

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 32-bit unsigned integers.

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 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.wu    d3, d1, d2
```

See Also

EQ.B, EQ.H, EQ.W, LT.B, LT.BU, LT.H, LT.HU, LT.W

3.1.167 MADD

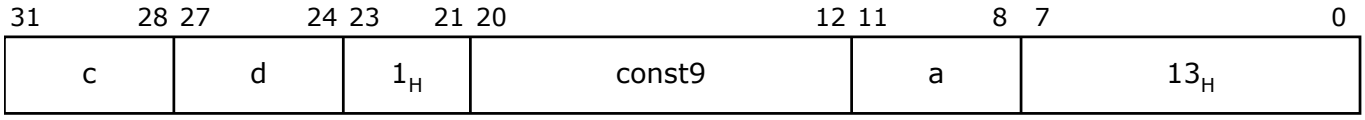
Multiply-Add

Description

Multiply two 32-bit signed integers, add the product to a 32-bit or 64-bit signed integer and put the result into a 32-bit or 64-bit register respectively. The value const9 is sign-extended before the multiplication is performed.

MADD D[c], D[d], D[a], const9 (RCR)

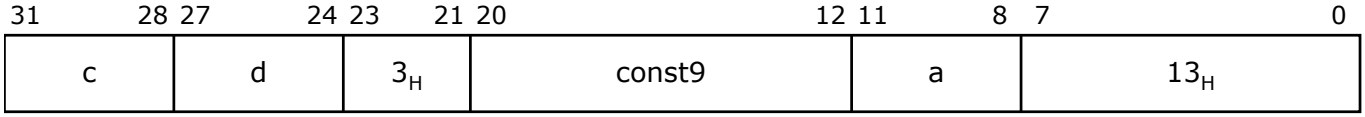
32 + (32 * K9)--> 32 signed




```
result = D[d] + (D[a] * sign_ext(const9));
D[c] = result[31:0];
```

MADD E[c], E[d], D[a], const9 (RCR)

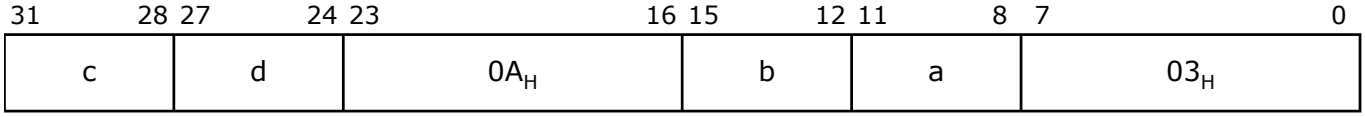
64 + (32 * K9)--> 64 signed




```
result = E[d] + (D[a] * sign_ext(const9));
E[c] = result[63:0];
```

MADD D[c], D[d], D[a], D[b] (RRR2)

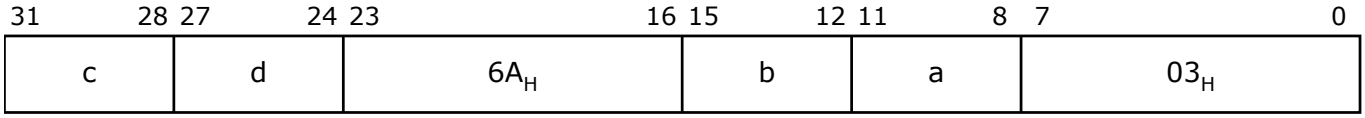
32 + (32 * 32)--> 32 signed




```
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




```
result = E[d] + (D[a] * D[b]);
E[c] = result[63:0];
```

Status Flags

C	Not set by this instruction.

V	32-bit result:
	overflow = (result > $7FFFFFFF_H$) OR (result < -80000000_H);
	if (overflow) then PSW.V = 1 else PSW.V = 0;
	64-bit result:
	overflow = (result > $7FFFFFFFFFFFFFFF_H$) OR (result < -8000000000000000_H);
	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, #7
madd    e0, e0, d3, #80
madd    d0, d1, d2, d3
madd    e0, e2, d6, d11
```

See Also

MADDS

3.1.168 MADDS

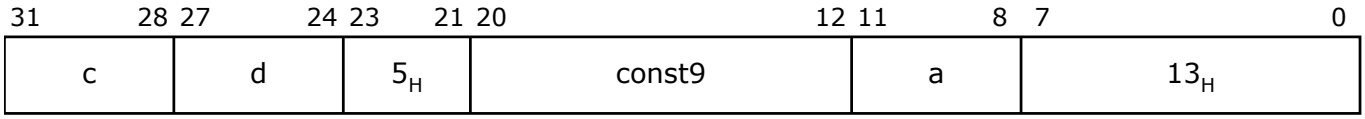
Multiply-Add, Saturated

Description

Multiply two 32-bit signed integers, add the product to a 32-bit or 64-bit signed integer and put the result into a 32-bit or 64-bit register respectively. The value const9 is sign-extended before the multiplication is performed. The result is saturated on overflow.

MADDS D[c], D[d], D[a], const9 (RCR)

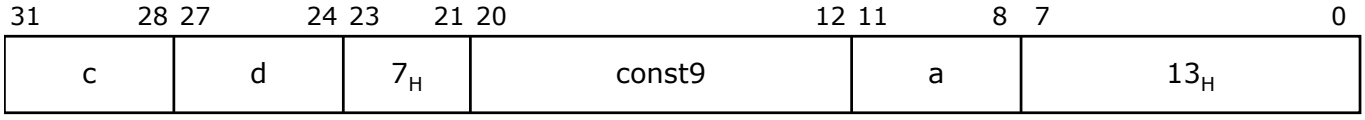
32 + (32 * K9)--> 32 signed saturated




```
result = D[d] + (D[a] * sign_ext(const9));
D[c] = ssov(result, 32);
```

MADDS E[c], E[d], D[a], const9 (RCR)

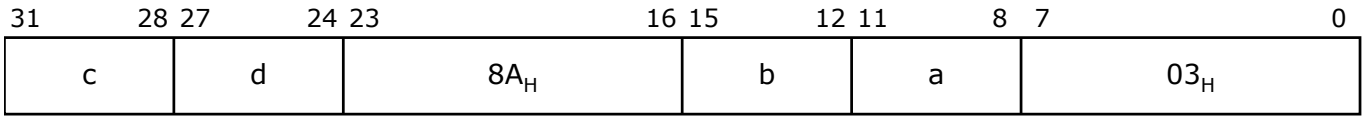
64 + (32 * K9)--> 64 signed saturated




```
result = E[d] + (D[a] * sign_ext(const9));
E[c]= ssov(result, 64);
```

MADDS D[c], D[d], D[a], D[b] (RRR2)

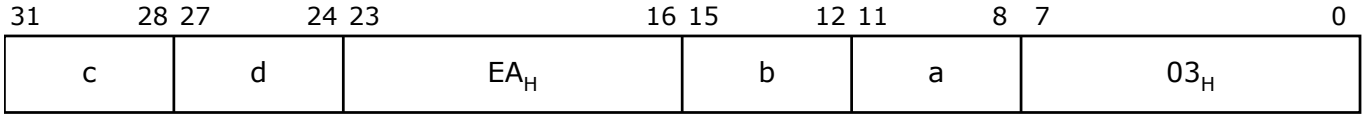
32 + (32 * 32)--> 32 signed saturated




```
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 this instruction.

V	32-bit result:
	overflow = (result > $7FFFFFFF_H$) OR (result < -80000000_H);
	if (overflow) then PSW.V = 1 else PSW.V = 0;
	64-bit result:
	overflow = (result > $7FFFFFFFFFFFFFFF_H$) OR (result < -8000000000000000_H);
	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

```
madds    d11, d1, d2, #7
madds    e8, e10, d3, #80
madds    d5, d1, d2, d2
madds    e0, e2, d6, d11
```

See Also

MADD

3.1.169 MADD.H

Packed Multiply-Add Q Format

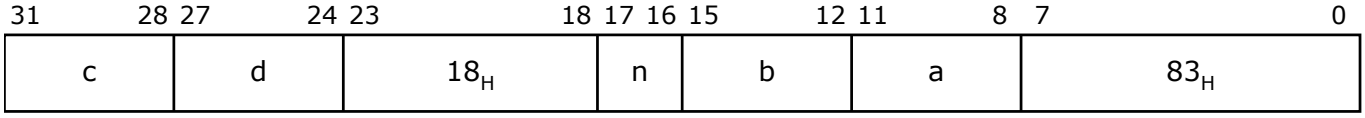
Description

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.

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 = $7FFFFFFF_H$.

MADD.H E[c], E[d], D[a], D[b] UL, n (RRR1)

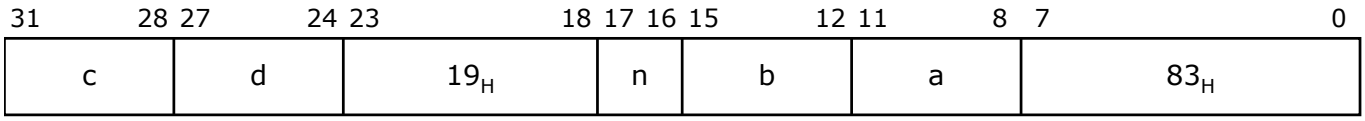
32||32 +||+ (16U * 16U || 16L * 16L)--> 32||32




```
sc1 = (D[a][31:16] == 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
```

MADD.H E[c], E[d], D[a], D[b] LU, n (RRR1)

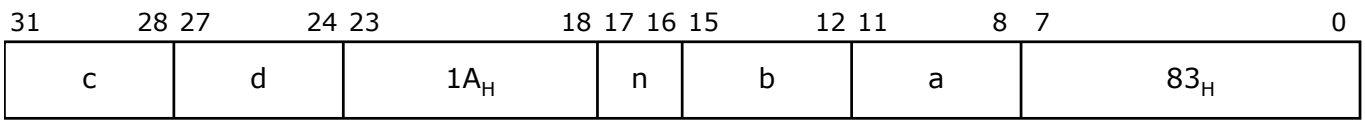
32||32 +||+ (16U * 16L || 16L * 16U)--> 32||32




```
sc1 = (D[a][31:16] == 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] = {result_word1[31:0], result_word0[31:0]}; // Packed fraction
```

MADD.H E[c], E[d], D[a], D[b] LL, n (RRR1)

32||32 +||+ (16U * 16L || 16L * 16L)--> 32||32

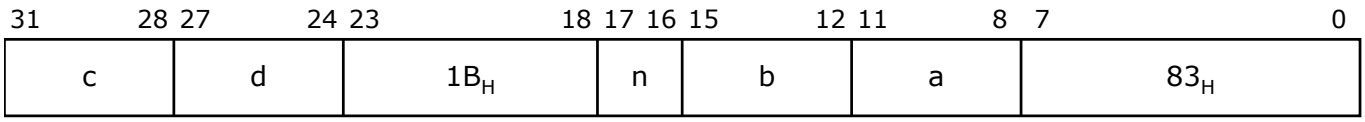



```
sc1 = (D[a][31:16] == 8000H) AND (D[b][15:0] == 8000H) AND (n == 1);
sc0 = (D[a][15:0] == 8000H) AND (D[b][15:0] == 8000H) AND (n == 1);
mul_res1 = sc1 ? 7FFFFFFFH : ((D[a][31:16] * D[b][15:0]) << n);
mul_res0 = sc0 ? 7FFFFFFFH : ((D[a][15:0] * D[b][15:0]) << n);
```

```
result_word1 = E[d][63:32] + mul_res1;
result_word0 = E[d][31:0] + mul_res0;
E[c] = {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] == 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
```

Status Flags

C	Not set by this instruction.

V	ov_word1 = (result_word1 > 7FFFFFFF$_H$) OR (result_word1 < -80000000$_H$);
	ov_word0 = (result_word0 > 7FFFFFFF$_H$) OR (result_word0 < -80000000$_H$);
	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

```
madd.h    e0, e2, d4, d5UL, #0
madd.h    e0, e2, d4, d5LU, #1
madd.h    e0, e2, d4, d5LL, #0
madd.h    e0, e2, d4, d5UU, #0
```

See Also

MADDS.H

3.1.170 MADDS.H

Packed Multiply-Add Q Format, Saturated

Description

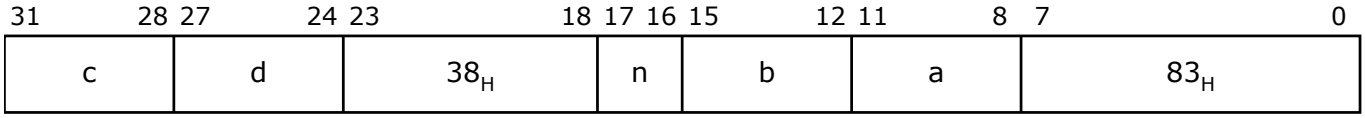
Multiply two signed 16-bit (half-word) values, add the product (left justified if n == 1) to a signed 32-bit value and put the result into a 32-bit register. There are four cases of half-word multiplication.

Each 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 = $7FFFFFFF_H$.

MADDS.H E[c], E[d], D[a], D[b] UL, n (RRR1)

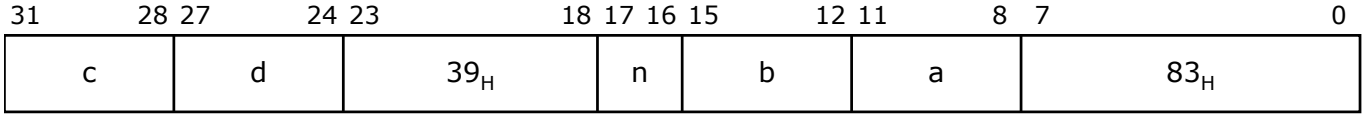
32||32 +||+ (16U * 16U || 16L * 16L)--> 32||32 saturated




```
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
```

MADDS.H E[c], E[d], D[a], D[b] LU, n (RRR1)

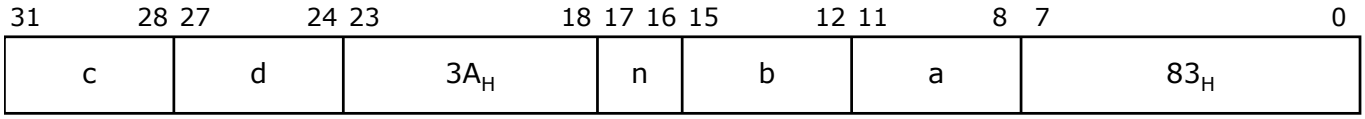
32||32 +||+ (16U * 16L || 16L * 16U)--> 32||32 saturated




```
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] LL, n (RRR1)

32||32 +||+ (16U * 16L || 16L * 16L)--> 32||32 saturated

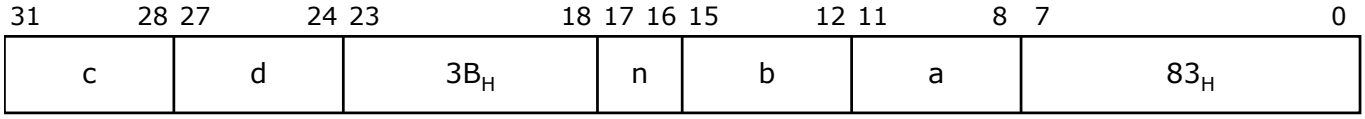



```
sc1 = (D[a][31:16] == 8000H) AND (D[b][15:0] == 8000H) AND (n == 1);
sc0 = (D[a][15:0] == 8000H) AND (D[b][15:0] == 8000H) AND (n == 1);
mul_res1 = sc1 ? 7FFFFFFFH : ((D[a][31:16] * D[b][15:0]) << n);
```

```
mul_res0 = sc0 ? 7FFFFFFF_H : ((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] == 8000_H) AND (D[b][31:16] == 8000_H) AND (n == 1);
sc0 = (D[a][31:16] == 8000_H) AND (D[b][31:16] == 8000_H) AND (n == 1);
mul_res1 = sc1 ? 7FFFFFFF_H : ((D[a][15:0] * D[b][31:16]) << n);
mul_res0 = sc0 ? 7FFFFFFF_H : ((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 this instruction.

V	ov_word1 = (result_word1 > 7FFFFFFF$_H$) OR (result_word1 < -80000000$_H$);
	ov_word0 = (result_word0 > 7FFFFFFF$_H$) OR (result_word0 < -80000000$_H$);
	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

```
madds.h    e0, e2, d4, d5UL, #0
madds.h    e0, e2, d4, d5LU, #1
madds.h    e0, e2, d4, d5LL, #0
madds.h    e0, e2, d4, d5UU, #0
```

See Also

MADD.H

3.1.171 MADD.Q

Multiply-Add Q Format

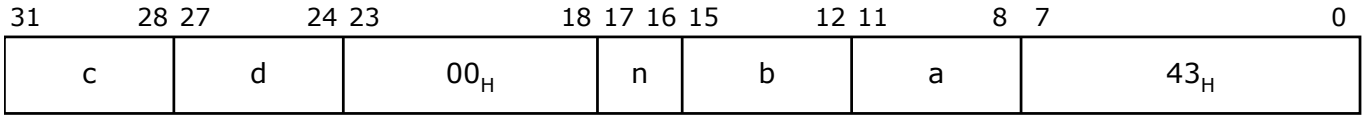
Description

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 respectively. There are four 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 16-bit * 16-bit multiplications only).

MADD.Q D[c], D[d], D[a], D[b] U, n (RRR1)

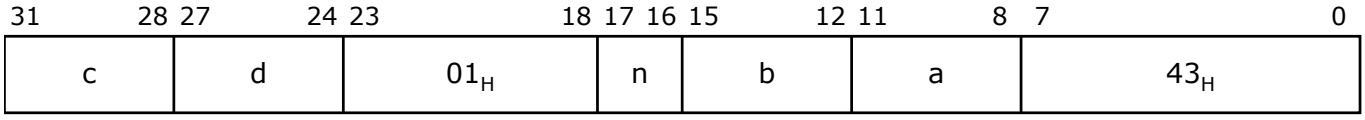
32 + (16U * 32)Up --> 32




```
result = D[d]+ (((D[a] * D[b][31:16]) << n) >> 16);
D[c] = result[31:0]; // Fraction
```

MADD.Q D[c], D[d], D[a], D[b] L, n (RRR1)

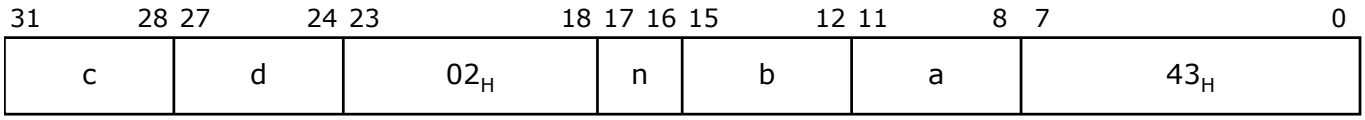
32 + (16L * 32)Up --> 32




```
result = D[d] + (((D[a] * D[b][15:0]) << n) >> 16);
D[c] = result[31:0]; // Fraction
```

MADD.Q D[c], D[d], D[a], D[b], n (RRR1)

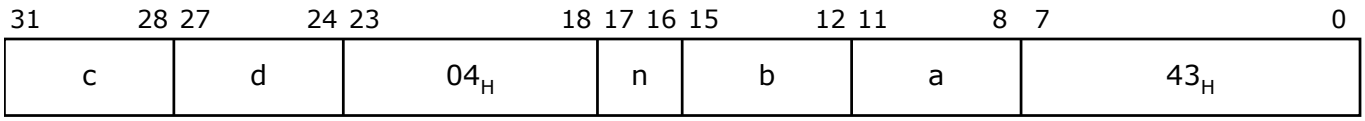
32 + (32 * 32)Up --> 32




```
result = D[d] + (((D[a] * D[b]) << n) >> 32);
D[c] = result[31:0]; // Fraction
```

MADD.Q D[c], D[d], D[a] U, D[b] U, n (RRR1)

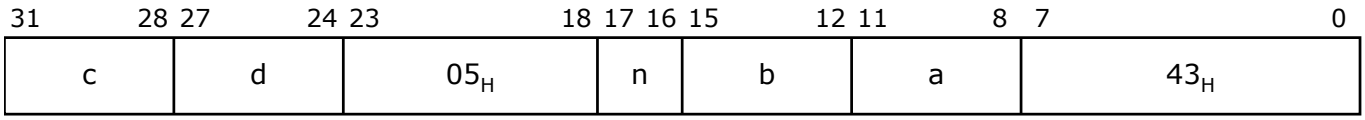
32 + (16U * 16U) --> 32




```
sc = (D[a][31:16] == 8000H) AND (D[b][31:16] == 8000H) AND (n == 1);
mul_res = sc ? 7FFFFFFFH : ((D[a][31:16] * D[b][31:16]) << n);
result = D[d] + mul_res;
D[c] = result[31:0]; // Fraction
```

MADD.Q D[c], D[d], D[a] L, D[b] L, n (RRR1)

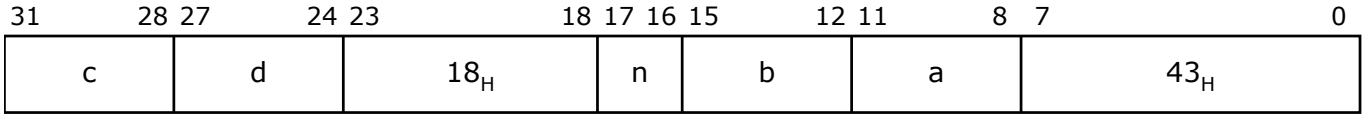
32 + (16L * 16L) --> 32




```
sc = (D[a][15:0] == 8000H) AND (D[b][15:0] == 8000H) AND (n == 1);
mul_res = sc ? 7FFFFFFFH : ((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], D[b] U, n (RRR1)

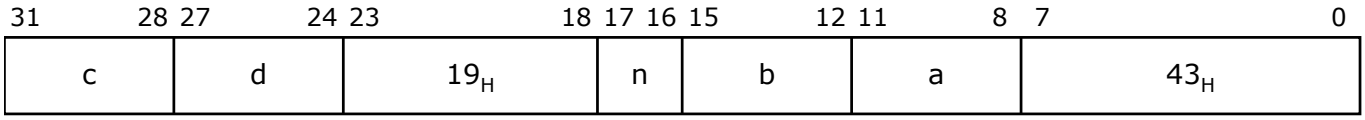
64 + (16U * 32) --> 64




```
result = E[d] + ((D[a] * D[b][31:16]) << n);
E[c] = result[63:0]; // Multi-precision accumulator
```

MADD.Q E[c], E[d], D[a], D[b] L, n (RRR1)

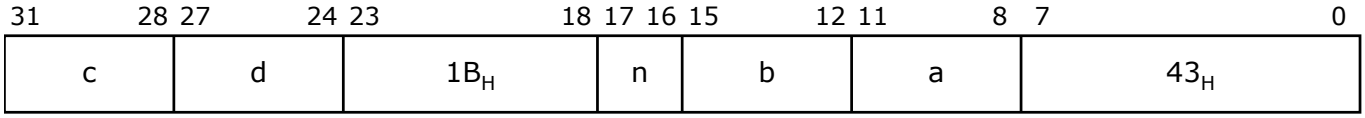
64 + (16L * 32) --> 64




```
result = E[d] + ((D[a] * D[b][15:0]) << n);
E[c] = result[63:0]; // Multi-precision accumulator
```

MADD.Q E[c], E[d], D[a], D[b], n (RRR1)

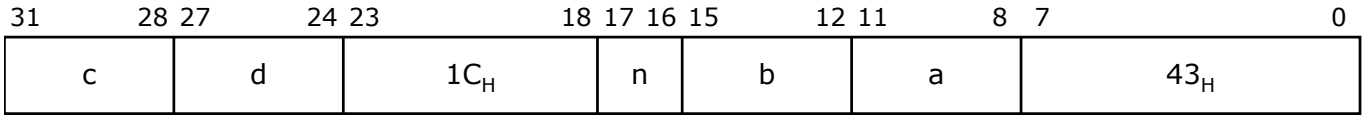
64 + (32 * 32) --> 64




```
result = E[d] + ((D[a] * D[b]) << n);
E[c] = result[63:0]; // Multi-precision fraction
```

MADD.Q E[c], E[d], D[a] U, D[b] U, n (RRR1)

64 + (16U * 16U) --> 64

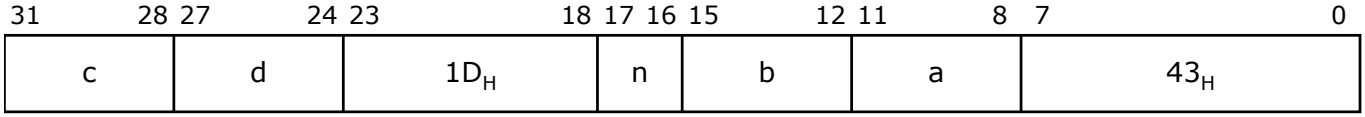



```
sc = (D[a][31:16] == 8000H) AND (D[b][31:16] == 8000H) AND (n == 1);
mul_res = sc ? 7FFFFFFFH : ((D[a][31:16] * D[b][31:16]) << n);
result = E[d] + (mul_res << 16);
```

```
E[c] = result[63:0]; // Multi-precision accumulator
```

MADD.Q E[c], E[d], D[a] L, D[b] L, n (RRR1)

64 + (16L * 16L) --> 64




```
sc = (D[a][15:0] == 8000H) AND (D[b][15:0] == 8000H) AND (n == 1);
mul_res = sc ? 7FFFFFFFH : ((D[a][15:0] * D[b][15:0]) << n);
result = E[d] + (mul_res << 16);
E[c] = result[63:0]; // Multi-precision accumulator
```

Status Flags

C	Not set by this instruction.

V	32-bit result:
	overflow = (result > 7FFFFFFF$_H$) OR (result < -80000000$_H$);
	if (overflow) then PSW.V = 1 else PSW.V = 0;
	64-bit result:
	overflow = (result > 7FFFFFFFFFFFFFFF$_H$) OR (result < -8000000000000000$_H$);
	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.q    d0, d1, d2, d6U, #1
madd.q    d0, d2, d1, d3L, #1
madd.q    d0, d1, d2, d3, #1
madd.q    d2, d0, d3U, d4U, #1
madd.q    d2, d0, d4L, d4L, #1
madd.q    e2, e2, d4, d6U, #1
madd.q    e2, e2, d5, d6L, #1
madd.q    e2, e2, d3, d7, #1
madd.q    e2, e2, d6U, d7U, #1
madd.q    e2, e2, d8L, d0L, #1
```

See Also

MADDS.Q

3.1.172 MADDS.Q

Multiply-Add Q Format, Saturated

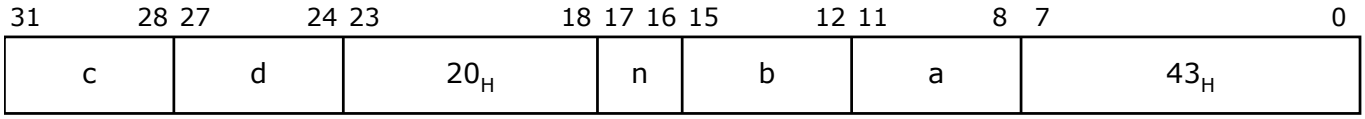
Description

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 four cases of 16*16 operations, four cases of 16*32 operations and two cases of 32*32 operations. On overflow the 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 16-bit * 16-bit multiplications only).

MADDS.Q D[c], D[d], D[a], D[b] U, n (RRR1)

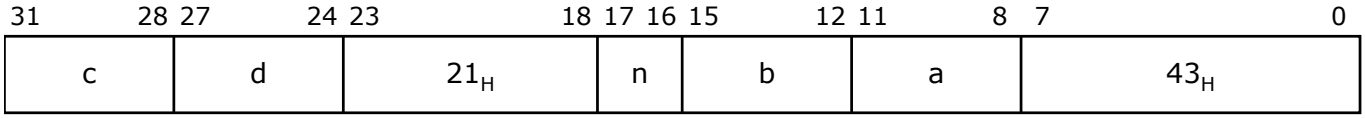
32 + (16U * 32)Up --> 32 saturated




```
result = D[d] + (((D[a] * D[b][31:16]) << n) >> 16);
D[c] = ssov(result, 32); // Fraction
```

MADDS.Q D[c], D[d], D[a], D[b] L, n (RRR1)

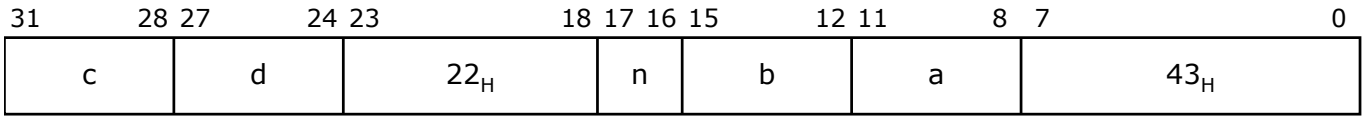
32 + (16L * 32)Up --> 32 saturated




```
result = D[d] + (((D[a] * D[b][15:0]) << n) >> 16);
D[c] = ssov(result, 32); // Fraction
```

MADDS.Q D[c], D[d], D[a], D[b], n (RRR1)

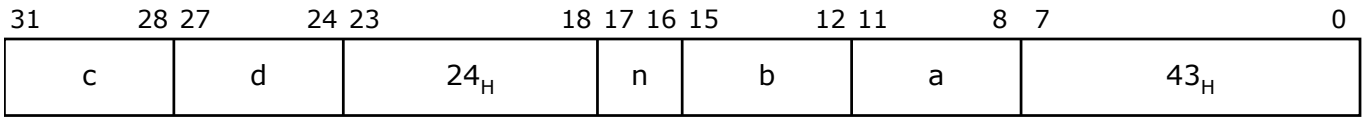
32 + (32 * 32)Up --> 32 saturated




```
result = D[d] + (((D[a] * D[b]) << n) >> 32);
D[c] = ssov(result, 32); // Fraction
```

MADDS.Q D[c], D[d], D[a] U, D[b] U, n (RRR1)

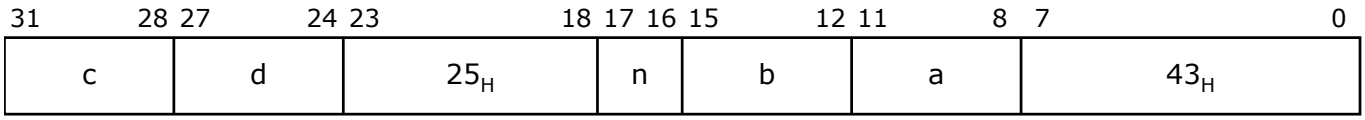
32 + (16U * 16U) --> 32 saturated




```
sc = (D[a][31:16] == 8000H) AND (D[b][31:16] == 8000H) AND (n == 1);
mul_res = sc ? 7FFFFFFFH : ((D[a][31:16] * D[b][31:16]) << n);
result = D[d] + mul_res;
D[c] = ssov(result, 32); // Fraction
```

MADDS.Q D[c], D[d], D[a] L, D[b] L, n (RRR1)

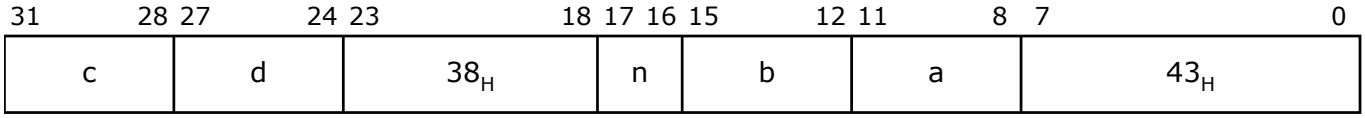
32 + (16L * 16L) --> 32 saturated




```
sc = (D[a][15:0] == 8000H) AND (D[b][15:0] == 8000H) AND (n == 1);
mul_res = sc ? 7FFFFFFFH : ((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], D[b] U, n (RRR1)

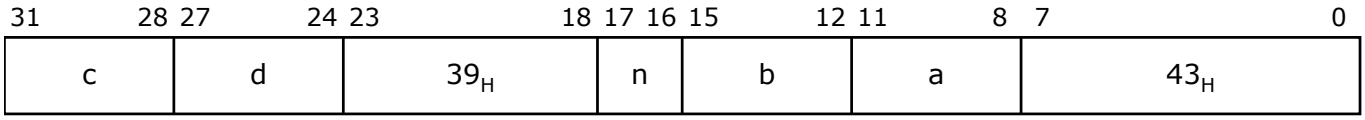
64 + (16U * 32) --> 64 saturated




```
result = E[d] + ((D[a] * D[b][31:16]) << n);
E[c] = ssov(result, 64); // Multi-precision accumulator
```

MADDS.Q E[c], E[d], D[a], D[b] L, n (RRR1)

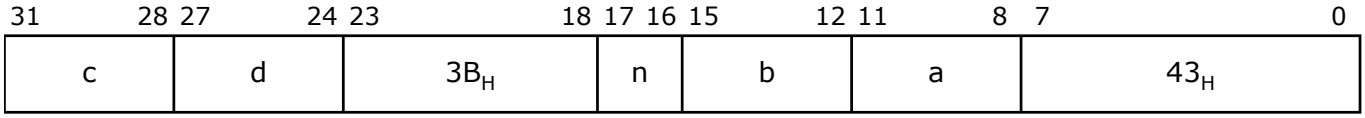
64 + (16L * 32) --> 64 saturated




```
result = E[d] + ((D[a] * D[b][15:0]) << n);
E[c] = ssov(result, 64); // Multi-precision accumulator
```

MADDS.Q E[c], E[d], D[a], D[b], n (RRR1)

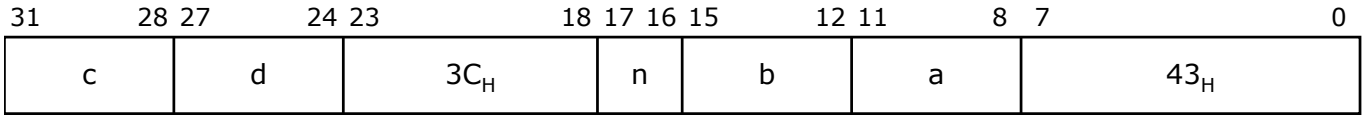
64 + (32 * 32) --> 64 saturated




```
result = E[d] + ((D[a] * D[b]) << n);
E[c] = ssov(result, 64) // Multi-precision fraction
```

MADDS.Q E[c], E[d], D[a] U, D[b] U, n (RRR1)

64 + (16U * 16U) --> 64 saturated

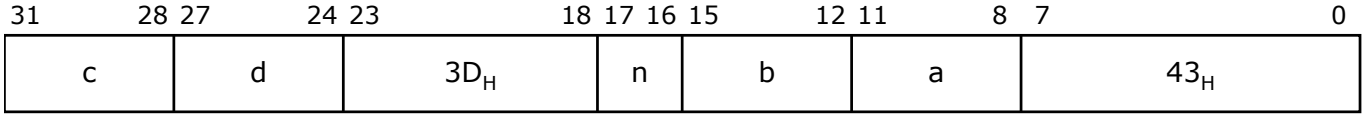



```
sc = (D[a][31:16] == 8000H) AND (D[b][31:16] == 8000H) AND (n == 1);
mul_res = sc ? 7FFFFFFFH : ((D[a][31:16] * D[b][31:16]) << n);
result = E[d] + (mul_res << 16);
```

```
E[c] = ssov(result, 64); // Multi-precision accumulator
```

MADDS.Q E[c], E[d], D[a] L, D[b] L, n (RRR1)

64 + (16L * 16L) --> 64 saturated




```
sc = (D[a][15:0] == 8000H) AND (D[b][15:0] == 8000H) AND (n == 1);
mul_res = sc ? 7FFFFFFFH : ((D[a][15:0] * D[b][15:0]) << n);
result = E[d] + (mul_res << 16);
E[c] = ssov(result, 64); // Multi-precision accumulator
```

Status Flags

C	Not set by this instruction.

V	32-bit result:
	overflow = (result > $7FFFFFFF_H$) OR (result < -80000000_H);
	if (overflow) then PSW.V = 1 else PSW.V = 0;
	64-bit result:
	overflow = (result > $7FFFFFFFFFFFFFFF_H$) OR (result < -8000000000000000_H);
	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

```
madds.q    d0, d1, d2, d6U, #1
madds.q    d0, d2, d1, d3L, #1
madds.q    d0, d1, d2, d3, #1
madds.q    d2, d0, d3U, d4U, #1
madds.q    d2, d0, d4L, d4L, #1
madds.q    e2, e2, d4, d6U, #1
madds.q    e2, e2, d5, d6L, #1
madds.q    e2, e2, d3, d7, #1
madds.q    e2, e2, d6U, d7U, #1
madds.q    e2, e0, d11L, d4L, #1
```

See Also

MADD.Q

3.1.173 MADD.U

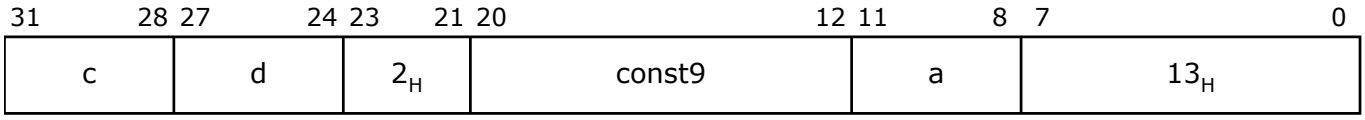
Multiply-Add Unsigned

Description

Multiply two 32-bit unsigned integers, add the product to a 64-bit unsigned integer, and put the result into a 64-bit register. The value const9 is zero-extended before the multiplication is performed.

MADD.U E[c], E[d], D[a], const9 (RCR)

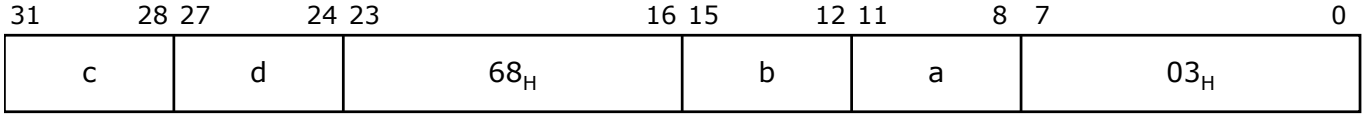
64 + (32 * K9) --> 64 unsigned




```
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)

64 + (32 * 32) --> 64 unsigned




```
result = E[d] + (D[a] * D[b]); // unsigned operators
E[c] = result[63:0];
```

Status Flags

C	Not set by this instruction.

V	32-bit result:
	overflow = (result > $FFFFFFFF_H$);
	if (overflow) then PSW.V = 1 else PSW.V = 0;
	64-bit result:
	overflow = (result > $FFFFFFFFFFFFFFFF_H$);
	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.u    e0, e0, d3, #56
madd.u    e0, e2, d6, d11
```

See Also

MADDS.U

3.1.174 MADDS.U

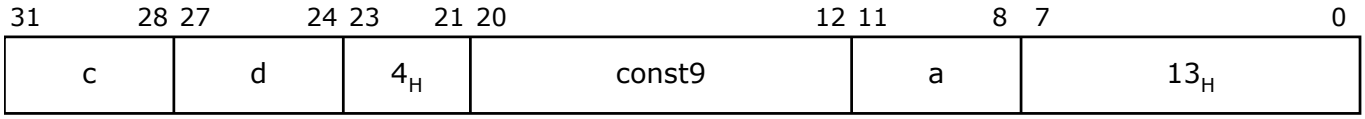
Multiply-Add Unsigned, Saturated

Description

Multiply two 32-bit unsigned integers, add the product to a 32-bit or 64-bit unsigned integer, and put the result into a 32-bit or 64-bit register respectively. The value const9 is zero-extended before the multiplication is performed. The result is saturated on overflow.

MADDS.U D[c], D[d], D[a], const9 (RCR)

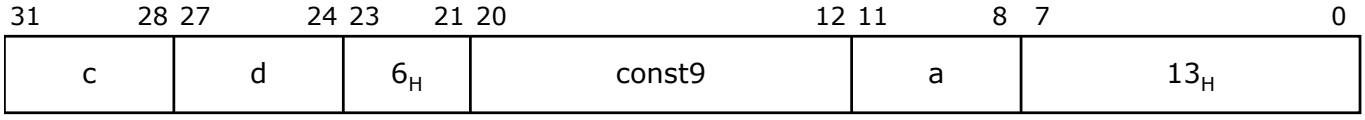
32 + (32 * K9) --> 32 unsigned saturated




```
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)

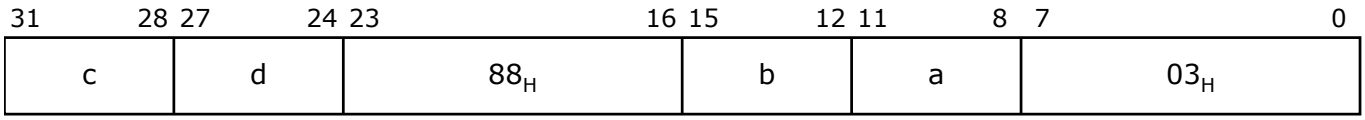
64 + (32 * K9) --> 64 unsigned saturated




```
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)

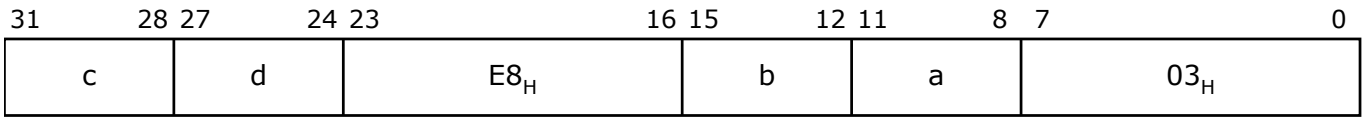
32 + (32 * 32) --> 32 unsigned saturated




```
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 this instruction.

V	32-bit result:
	overflow = (result > $FFFFFFFF_H$);
	if (overflow) then PSW.V = 1 else PSW.V = 0;
	64-bit result:
	overflow = (result > $FFFFFFFFFFFFFFFF_H$);
	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

```
madds.u    d11, d1, d2, #7
madds.u    e8, e0, d0, #80
madds.u    d5, d1, d2, d2
madds.u    e0, e2, d6, d11
```

See Also

MADD.U

3.1.175 MADDM.H

Packed Multiply-Add Q Format Multi-precision

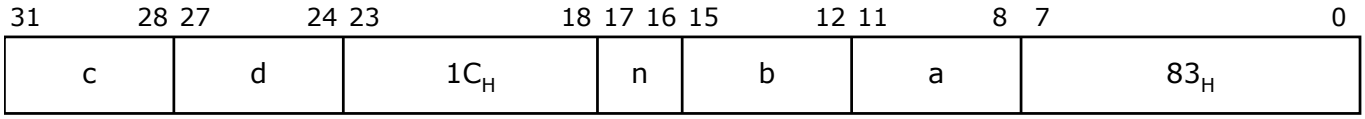
Description

Perform two multiplications of two signed 16-bit (half-word) values. Add the two products (left justified if n == 1) left-shifted by 16, to a signed 64-bit value and put the result in a 64-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= $7FFFFFFF_H$.

MADDM.H E[c], E[d], D[a], D[b] UL, n (RRR1)

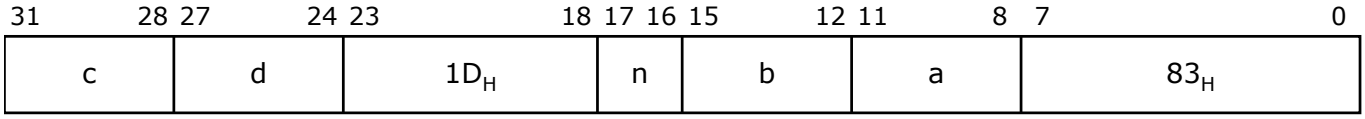
64 + (16U * 16U) + (16L * 16L) --> 64




```
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] = result[63:0]; // Multi-precision accumulator
```

MADDM.H E[c], E[d], D[a], D[b] LU, n (RRR1)

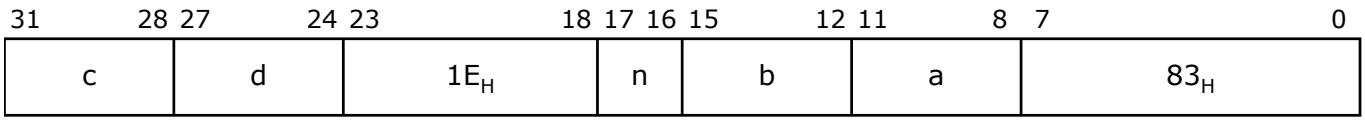
64 + (16U * 16L) + (16L * 16U) --> 64




```
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] = result[63:0]; // Multi-precision accumulator
```

MADDM.H E[c], E[d], D[a], D[b] LL, n (RRR1)

64 + (16U * 16L) + (16L * 16L) --> 64

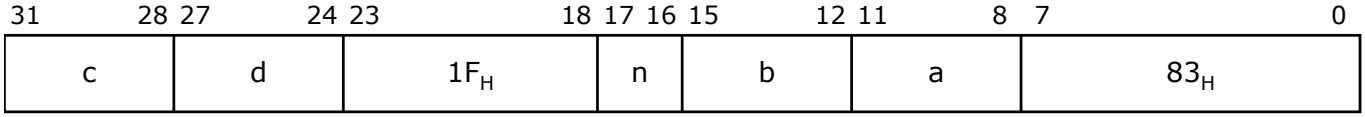



```
sc1 = (D[a][31:16] == 8000H) AND (D[b][15:0] == 8000H) AND (n == 1);
sc0 = (D[a][15:0] == 8000H) AND (D[b][15:0] == 8000H) AND (n == 1);
result_word1 = sc1 ? 7FFFFFFFH : ((D[a][31:16] * D[b][15:0]) << n);
result_word0 = sc0 ? 7FFFFFFFH : ((D[a][15:0] * D[b][15:0]) << n);
result = E[d] + ((result_word1 + result_word0) << 16);
```

```
E[c] = 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] == 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] = result[63:0]; // Multi-precision accumulator
```

Status Flags

C	Not set by this instruction.
V	overflow = (result > 7FFFFFFFFFFFFFFFH) OR (result < -8000000000000000H); 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

```
maddm.h    e0, e2, d4, d5UL, #0
maddm.h    e0, e2, d4, d5LU, #1
maddm.h    e0, e2, d4, d5LL, #0
maddm.h    e0, e2, d4, d5UU, #0
```

See Also

MADDMS.H

3.1.176 MADDMS.H

Packed Multiply-Add Q Format Multi-precision, Saturated

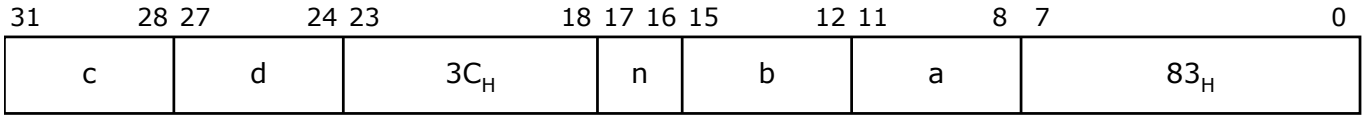
Description

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 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= $7FFFFFFF_H$.

MADDMS.H E[c], E[d], D[a], D[b] UL, n (RRR1)

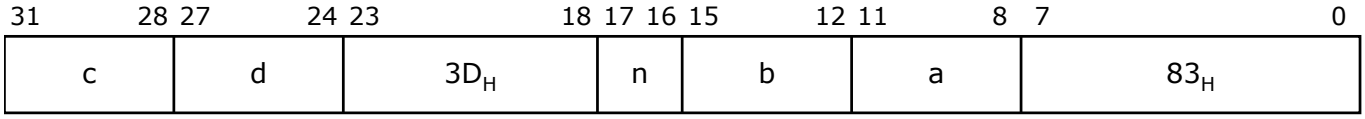
64 + (16U * 16U) + (16L * 16L) --> 64 saturated




```
sc1 = (D[a][31:16] == 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
```

MADDMS.H E[c], E[d], D[a], D[b] LU, n (RRR1)

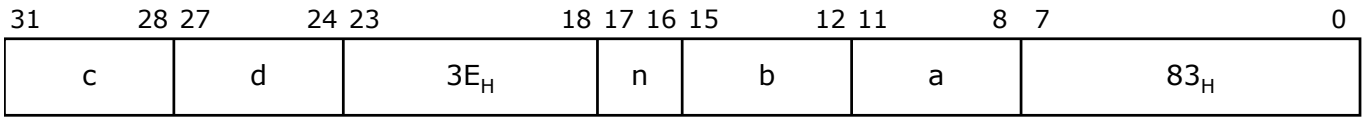
64 + (16U * 16L) + (16L * 16U) --> 64 saturated




```
sc1 = (D[a][31:16] == 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
```

MADDMS.H E[c], E[d], D[a], D[b] LL, n (RRR1)

64 + (16U * 16L) + (16L * 16L) --> 64 saturated

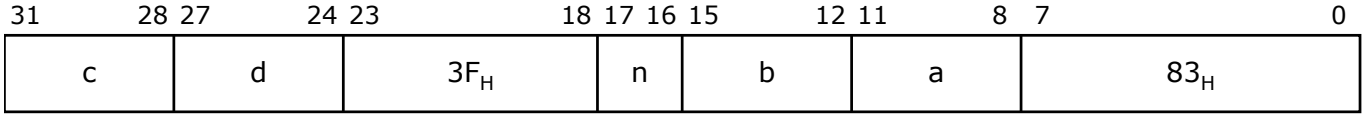



```
sc1 = (D[a][31:16] == 8000H) AND (D[b][15:0] == 8000H) AND (n == 1);
sc0 = (D[a][15:0] == 8000H) AND (D[b][15:0] == 8000H) AND (n == 1);
result_word1 = sc1 ? 7FFFFFFFH : ((D[a][31:16] * D[b][15:0]) << n);
result_word0 = sc0 ? 7FFFFFFFH : ((D[a][15:0] * D[b][15:0]) << n);
result = E[d] + ((result_word1 + result_word0) << 16);
```

```
E[c] = ssov(result, 64); // Multi-precision accumulator
```

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] == 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 this instruction.
V	overflow = (result > 7FFFFFFFFFFFFFFF$_H$) OR (result < -8000000000000000$_H$); 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

```
maddms.h    e0, e2, d4, d5UL, #0
maddms.h    e0, e2, d4, d5LU, #1
maddms.h    e0, e2, d4, d5LL, #0
maddms.h    e0, e2, d4, d5UU, #0
```

See Also

MADDM.H

3.1.177 MADDR.H

Packed Multiply-Add Q Format with Rounding

Description

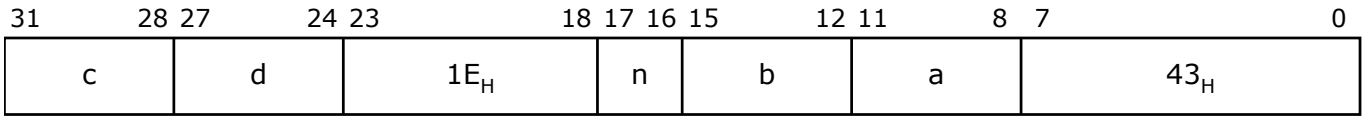
Multiply two signed 16-bit (half-word) values, add the product (left justified if n == 1) to a signed 16-bit or 32-bit value and put the rounded result into half of a 32-bit register (Note that since there are two results the two register halves are used). There are four cases of half-word multiplication:

- 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 = $7FFFFFFF_H$.

MADDR.H D[c], E[d], D[a], D[b] UL, n (RRR1)

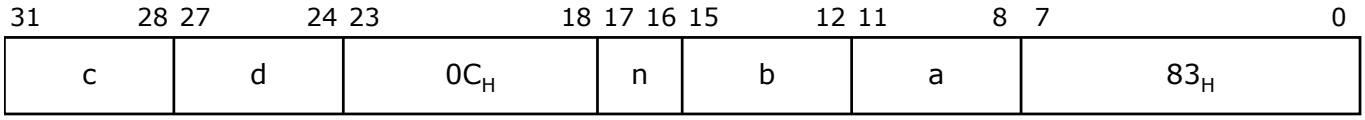
32 || 32 +||+ (16U * 16U || 16L * 16L) rounded --> 16||16




```
sc1 = (D[a][31:16] == 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] = {result_halfword1[31:16], result_halfword0[31:16]}; // Packed short fraction
```

MADDR.H D[c], D[d], D[a], D[b] UL, n (RRR1)

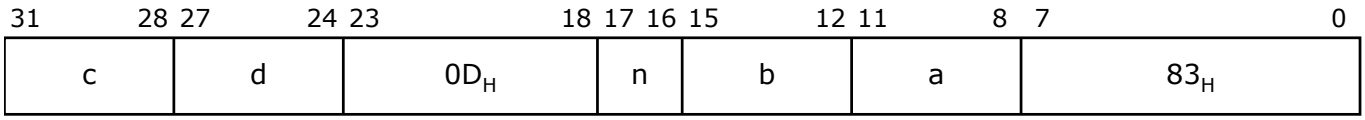
16U || 16L +||+ (16U * 16U || 16L * 16L) rounded --> 16||16




```
sc1 = (D[a][31:16] == 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
```

MADDR.H D[c], D[d], D[a], D[b] LU, n (RRR1)

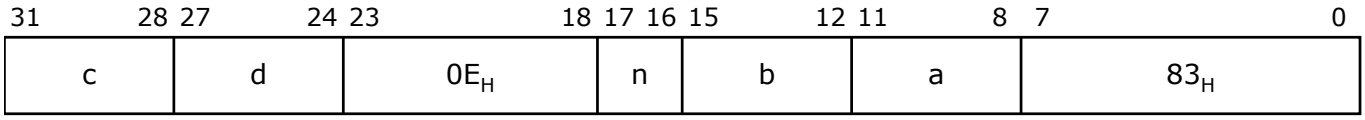
16U || 16L +||+ (16U * 16L || 16L * 16U) rounded --> 16 || 16




```
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] = {result_halfword1[31:16], result_halfword0[31:16]}; // Packed short fraction
```

MADDR.H D[c], D[d], D[a], D[b] LL, n (RRR1)

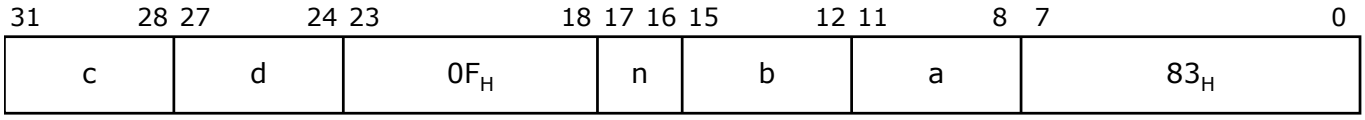
16U || 16L +||+ (16U * 16L || 16L * 16L) rounded --> 16||16




```
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] = {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




```
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
```

Status Flags

C	Not set by this instruction.

V	ov_halfword1 = (result_halfword1 > 7FFFFFFFH) OR (result_halfword1 < -80000000H);
	ov_halfword0 = (result_halfword0 > 7FFFFFFFH) OR (result_halfword0 < -80000000H);
	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

```
maddr.h    d0, e2, d4, d5UL, #0
maddr.h    d0, d2, d4, d5UL, #1
maddr.h    d0, d2, d4, d5LU, #0
maddr.h    d0, d2, d4, d5LL, #0
maddr.h    d0, d2, d4, d5UU, #0
```

See Also

MADDRS.H

3.1.178 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). The 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 = $7FFFFFFF_H$.

MADDRS.H D[c], E[d], D[a], D[b] UL, n (RRR1)

32 || 32 +||+ (16U * 16U || 16L * 16L) rounded --> 16||16 saturated

31–28	27–24	23–18	17 16	15–12	11–8	7–0
c	d	$3E_H$	n	b	a	43_H

```
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] UL, n (RRR1)

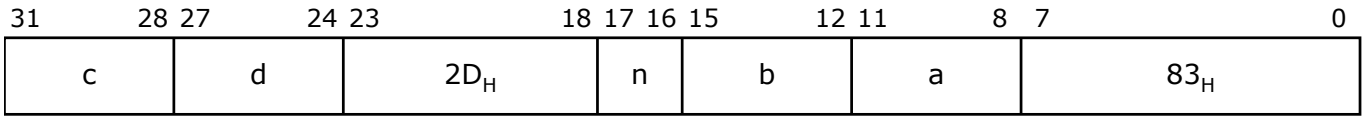
16U || 16L +||+ (16U * 16U || 16L * 16L) rounded --> 16||16 saturated

31–28	27–24	23–18	17 16	15–12	11–8	7–0
c	d	$2C_H$	n	b	a	83_H

```
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
```

MADDRS.H D[c], D[d], D[a], D[b] LU, n (RRR1)

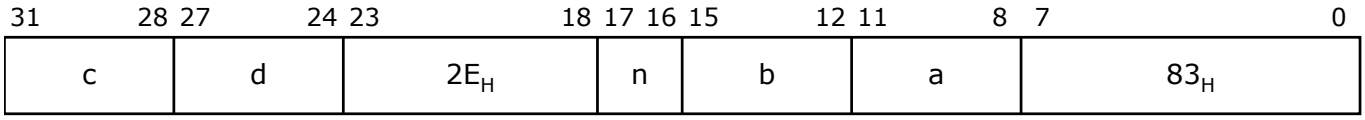
16U || 16L +||+ (16U * 16L || 16L * 16U) rounded --> 16||16 saturated




```
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
```

MADDRS.H D[c], D[d], D[a], D[b] LL, n (RRR1)

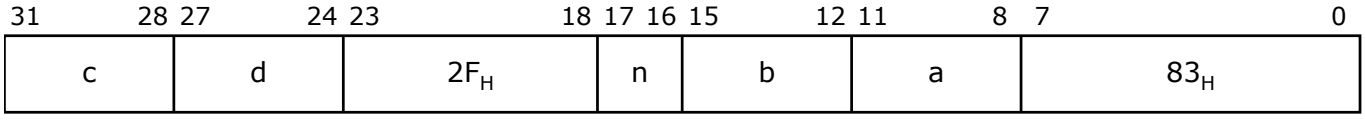
16U || 16L +||+ (16U * 16L || 16L * 16L) rounded --> 16||16 saturated




```
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
```

MADDRS.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 this instruction.

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

```
maddrs.h    d0, e2, d4, d5UL, #0
maddrs.h    d0, d2, d4, d5UL, #1
maddrs.h    d0, d2, d4, d5LU, #0
maddrs.h    d0, d2, d4, d5LL, #0
maddrs.h    d0, d2, d4, d5UU, #0
```

See Also

MADDR.H

3.1.179 MADDR.Q

Multiply-Add Q Format with Rounding

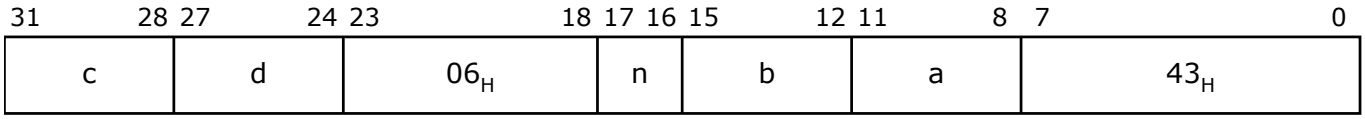
Description

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] U, D[b] U, n (RRR1)

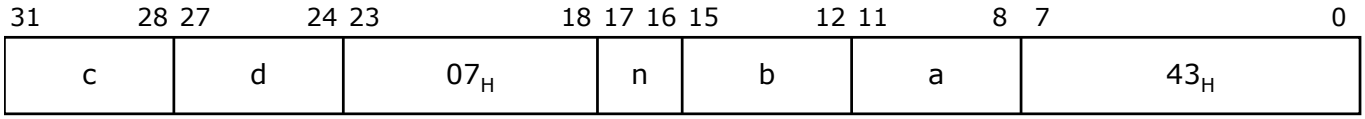
32 + (16U * 16U) rounded --> 32




```
sc = (D[a][31:16] == 8000H) AND (D[b][31:16] == 8000H) AND (n == 1);
mul_res = sc ? 7FFFFFFFH : ((D[a][31:16] * D[b][31:16]) << n);
result = D[d] + mul_res + 8000H;
D[c] = {result[31:16], 16'b0}; // Short fraction
```

MADDR.Q D[c], D[d], D[a] L, D[b] L, n (RRR1)

32 + (16L * 16L) rounded --> 32




```
sc = (D[a][15:0] == 8000H) AND (D[b][15:0] == 8000H) AND (n == 1);
mul_res = sc ? 7FFFFFFFH : ((D[a][15:0] * D[b][15:0]) << n);
result = D[d] + mul_res + 8000H;
D[c] = {result[31:16], 16'b0}; // Short fraction
```

Status Flags

C	Not set by this instruction.
V	overflow = (result > $7FFFFFFF_H$) OR (result < -80000000_H); 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

```
maddr.q    d2, d3, d0U, d1U, #0
maddr.q    d2, d3, d0L, d1L, #1
```

See Also

MADDRS.Q

3.1.180 MADDRS.Q

Multiply-Add Q Format with Rounding, Saturated

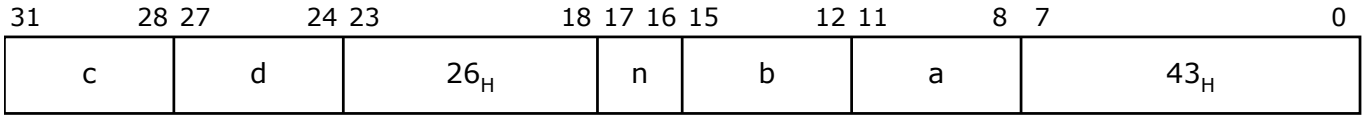
Description

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).

MADDRS.Q D[c], D[d], D[a] U, D[b] U, n (RRR1)

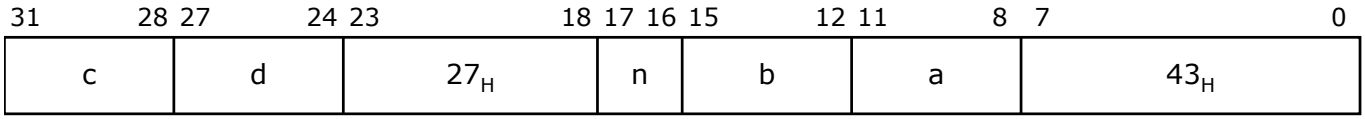
32 + (16U * 16U) rounded --> 32 saturated




```
sc = (D[a][31:16] == 8000H) AND (D[b][31:16] == 8000H) AND (n == 1);
mul_res = sc ? 7FFFFFFFH : ((D[a][31:16] * D[b][31:16]) << n);
result = D[d] + mul_res + 8000H;
D[c] = {ssov(result,32)[31:16]), 16'b0}; // Short fraction
```

MADDRS.Q D[c], D[d], D[a] L, D[b] L, n (RRR1)

32 + (16L * 16L) rounded --> 32 saturated




```
sc = (D[a][15:0] == 8000H) AND (D[b][15:0] == 8000H) AND (n == 1);
mul_res = sc ? 7FFFFFFFH : ((D[a][15:0] * D[b][15:0]) << n);
result = D[d] + mul_res + 8000H;
D[c] = {ssov(result,32)[31:16]), 16'b0}; // Short fraction
```

Status Flags

C	Not set by this instruction.
V	overflow = (result > $7FFFFFFF_H$) OR (result < -80000000_H); 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

```
maddrs.q    d2, d3, d0U, d1U, #0
maddrs.q    d2, d3, d0L, d1L, #1
```

See Also

MADDR.Q

3.1.181 MADDSU.H

Packed Multiply-Add/Subtract Q Format

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. 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] UL, n (RRR1)

32||32 +||- (16U * 16U || 16L * 16L) --> 32||32

31–28	27–24	23–18	17 16	15–12	11–8	7–0
c	d	18_H	n	b	a	$C3_H$

```
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] LU, n (RRR1)

32||32 +||- (16U * 16L || 16L * 16U) --> 32||32

31–28	27–24	23–18	17 16	15–12	11–8	7–0
c	d	19_H	n	b	a	$C3_H$

```
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] = {result_word1[31:0], result_word0[31:0]}; // Packed fraction
```

MADDSU.H E[c], E[d], D[a], D[b] LL, n (RRR1)

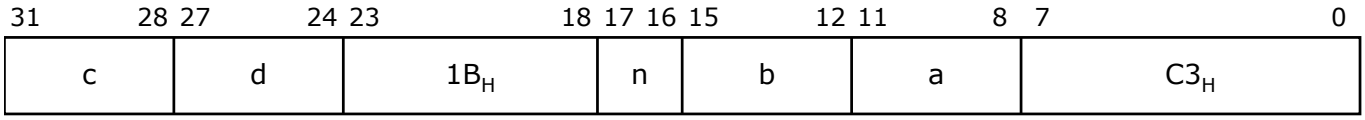
32||32 +||- (16U * 16L || 16L * 16L) --> 32||32

31–28	27–24	23–18	17 16	15–12	11–8	7–0
c	d	$1A_H$	n	b	a	$C3_H$

```
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] = {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
```

Status Flags

C	Not set by this instruction.

V	ov_word1 = (result_word1 > 7FFFFFFFH) OR (result_word1 < -80000000H);
	ov_word0 = (result_word0 > 7FFFFFFFH) OR (result_word0 < -80000000H);
	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

```
maddsu.h    e0, e2, d4, d5UL, #0
maddsu.h    e0, e2, d4, d5LU, #1
maddsu.h    e0, e2, d4, d5LL, #0
maddsu.h    e0, e2, d4, d5UU, #0
```

See Also

MADDSUS.H

3.1.182 MADDSUS.H

Packed Multiply-Add/Subtract Q Format Saturated

Description

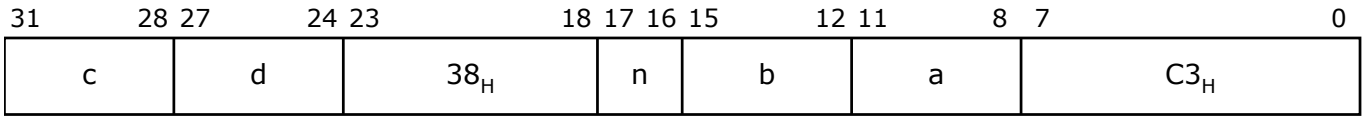
Multiply two signed 16-bit (half-word) values. Add (or subtract) the product (left justified if n == 1) to a signed 32-bit value and put the result into a 32-bit register. Each result is independently saturated on overflow. There are four cases of half-word multiplication:

- 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).

MADDSUS.H E[c], E[d], D[a], D[b] UL, n (RRR1)

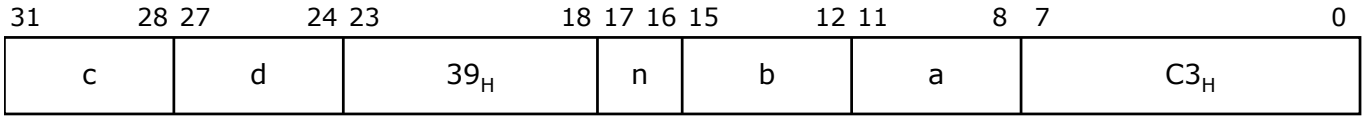
32||32 +||- (16U * 16U || 16L * 16L) --> 32||32 saturated




```
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] LU, n (RRR1)

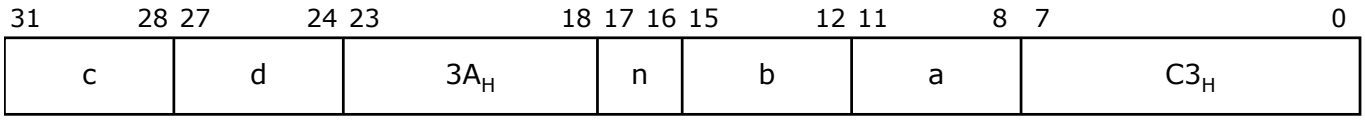
32||32 +||- (16U * 16L || 16L * 16U) --> 32||32 saturated




```
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] LL, n (RRR1)

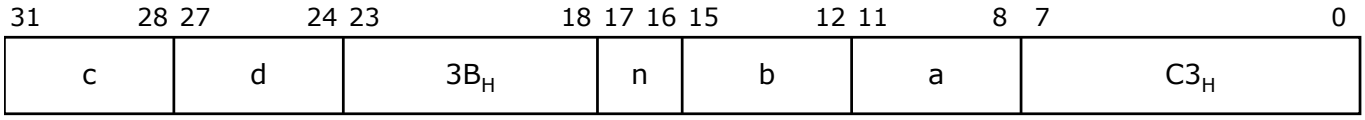
32||32 +||- (16U * 16L || 16L * 16L) --> 32||32 saturated




```
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] 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 this instruction.

V	ov_word1 = (result_word1 > 7FFFFFFFH) OR (result_word1 < -80000000H);
	ov_word0 = (result_word0 > 7FFFFFFFH) OR (result_word0 < -80000000H);
	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

```
maddsus.h    e0, e2, d4, d5UL, #0
maddsus.h    e0, e2, d4, d5LU, #1
maddsus.h    e0, e2, d4, d5LL, #0
maddsus.h    e0, e2, d4, d5UU, #0
```

See Also

MADDSU.H

3.1.183 MADDSUM.H

Packed Multiply-Add/Subtract Q Format Multi-precision

Description

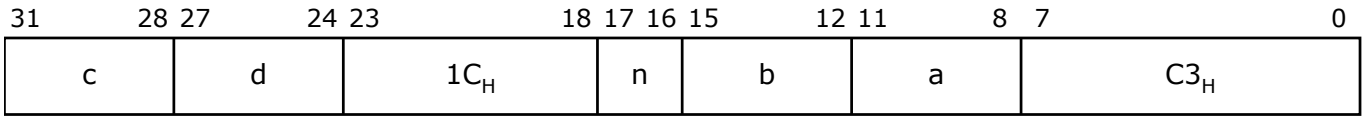
Perform two multiplications of two signed 16-bit (half-word) values. Add one product and subtract the other product (left justified if n == 1) left-shifted by 16, to/from a signed 64-bit value and put the result in a 64-bit register. There are four cases of half-word multiplication:

- 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] UL, n (RRR1)

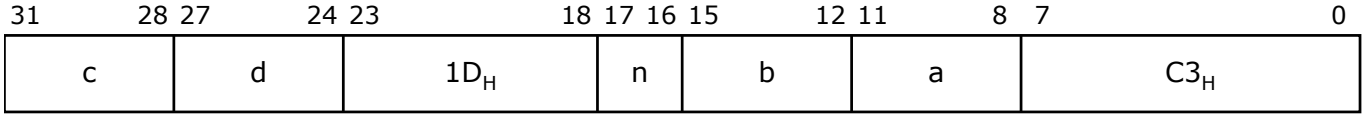
64 + (16U * 16U) - (16L * 16L) --> 64




```
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] = result[63:0]; // Multi-precision accumulator
```

MADDSUM.H E[c], E[d], D[a], D[b] LU, n (RRR1)

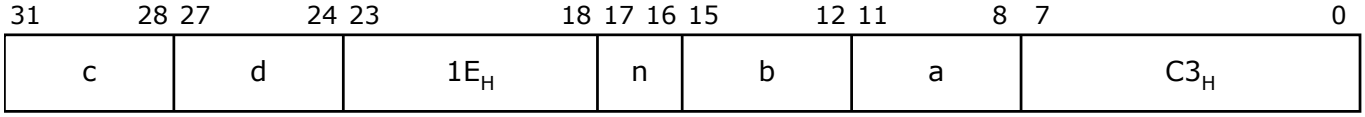
64 + (16U * 16L) - (16L * 16U) --> 64




```
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] = result[63:0]; // Multi-precision accumulator
```

MADDSUM.H E[c], E[d], D[a], D[b] LL, n (RRR1)

64 + (16U * 16L) - (16L * 16L) --> 64




```
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] = 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

31 – 28	27 – 24	23 – 18	17 16	15 – 12	11 – 8	7 – 0
c	d	$1F_H$	n	b	a	$C3_H$

```
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] = result[63:0]; // Multi-precision accumulator
```

Status Flags

C	Not set by this instruction.
V	overflow = (result > 7FFFFFFFFFFFFFFFH) OR (result < -8000000000000000H); 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

```
maddsum.h    e0, e2, d4, d5UL, #0
maddsum.h    e0, e2, d4, d5LU, #1
maddsum.h    e0, e2, d4, d5LL, #0
maddsum.h    e0, e2, d4, d5UU, #0
```

See Also

MADDSUMS.H

3.1.184 MADDSUMS.H

Packed Multiply-Add/Subtract Q Format Multi-precision Saturated

Description

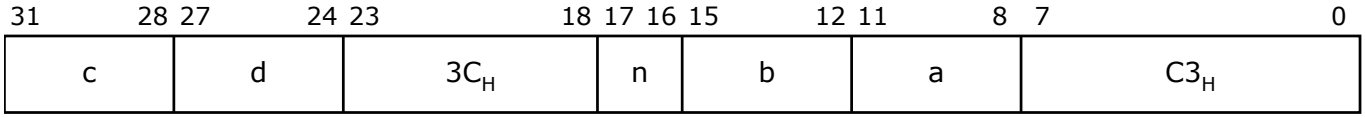
Perform two multiplications of two signed 16-bit (half-word) values. Add one product and subtract the other product (left justified if n == 1) left-shifted by 16, to/from a signed 64-bit value and put the result in a 64-bit register. The result is saturated on overflow. There are four cases of half-word 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).

MADDSUMS.H E[c], E[d], D[a], D[b] UL, n (RRR1)

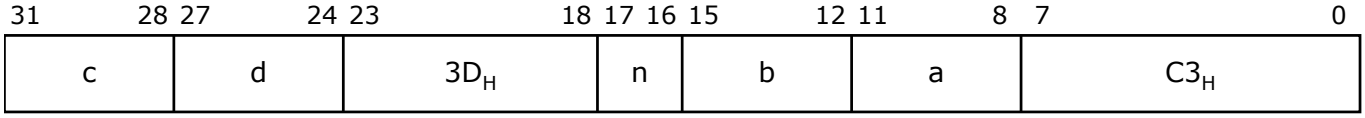
64 + (16U * 16U) - (16L * 16L) --> 64 saturated




```
sc1 = (D[a][31:16] == 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
```

MADDSUMS.H E[c], E[d], D[a], D[b] LU, n (RRR1)

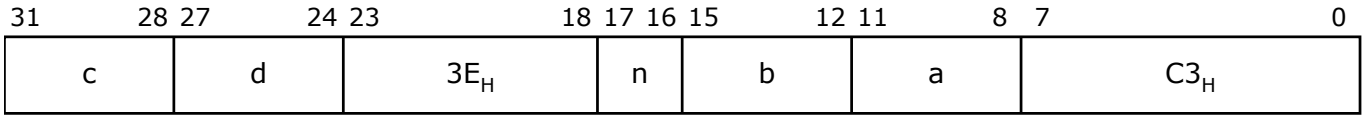
64 + (16U * 16L) - (16L * 16U) --> 64 saturated




```
sc1 = (D[a][31:16] == 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
```

MADDSUMS.H E[c], E[d], D[a], D[b] LL, n (RRR1)

64 + (16U * 16L) - (16L * 16L) --> 64 saturated

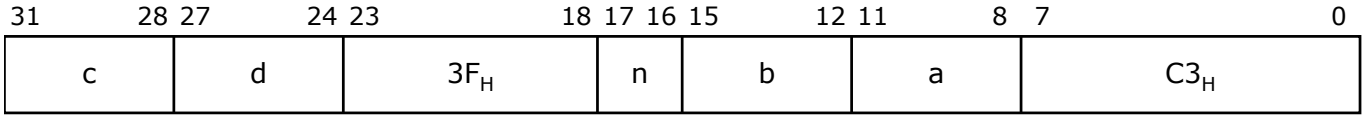



```
sc1 = (D[a][31:16] == 8000H) AND (D[b][15:0] == 8000H) AND (n == 1);
```

```
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
```

MADDSUMS.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);
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 this instruction.
V	overflow = (result > 7FFFFFFFFFFFFFFFH) OR (result < -8000000000000000H); 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

```
maddsums.h    e0, e2, d4, d5UL, #0
maddsums.h    e0, e2, d4, d5LU, #1
maddsums.h    e0, e2, d4, d5LL, #0
maddsums.h    e0, e2, d4, d5UU, #0
```

See Also

MADDSUM.H

3.1.185 MADDSUR.H

Packed Multiply-Add/Subtract Q Format with Rounding

Description

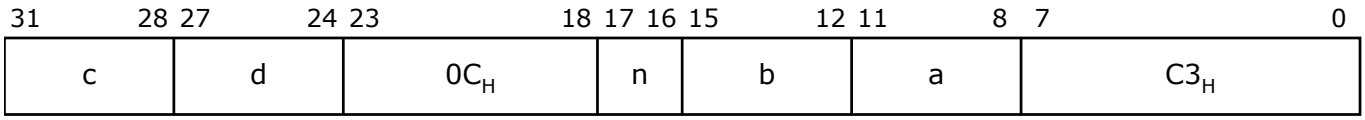
Multiply two signed 16-bit (half-word) values. Add (subtract) the product (left justified if n == 1) to (from) a signed 16-bit value and put the rounded result into half of a 32-bit register (Note that since there are two results, the two register halves are used). There are four cases of half-word multiplication:

- 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] UL, n (RRR1)

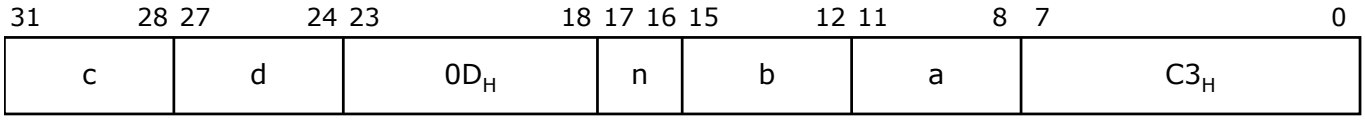
16U || 16L +||- (16U * 16U || 16L * 16L) rounded --> 16||16




```
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] LU, n (RRR1)

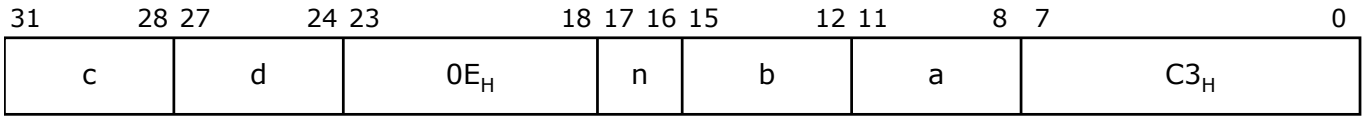
16U || 16L +||- (16U * 16L || 16L * 16U) rounded --> 16||16




```
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] = {result_halfword1[31:16], result_halfword0[31:16]}; // Packed short fraction
```

MADDSUR.H D[c], D[d], D[a], D[b] LL, n (RRR1)

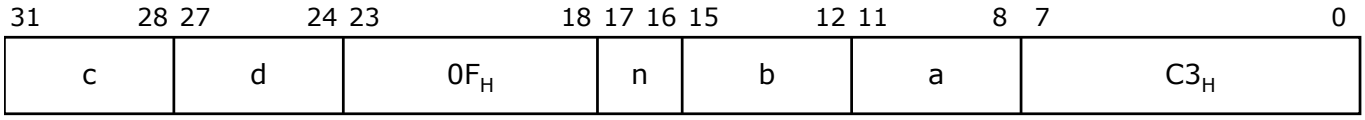
16U * 16L +||- (16U * 16L || 16L * 16L) rounded --> 16||16




```
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] = {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
```

Status Flags

C	Not set by this instruction.

V	ov_halfword1 =
	(result_halfword1 > 7FFFFFFFH) OR (result_halfword1 < -80000000H);
	ov_halfword0 =
	(result_halfword0 > 7FFFFFFFH) OR (result_halfword0 < -80000000H);
	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

```
maddsur.h    d0, d2, d4, d5UL, #0
maddsur.h    d0, d2, d4, d5LU, #1
maddsur.h    d0, d2, d4, d5LL, #0
maddsur.h    d0, d2, d4, d5UU, #0
```

See Also

MADDSURS.H

3.1.186 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).

MADDSURS.H D[c], D[d], D[a], D[b] UL, n (RRR1)

16U || 16L +||- (16U * 16U || 16L * 16L) rounded --> 16||16 saturated

31 28	27 24	23 18	17 16	15 12	11 8	7 0
c	d	$2C_H$	n	b	a	$C3_H$

```
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] LU, n (RRR1)

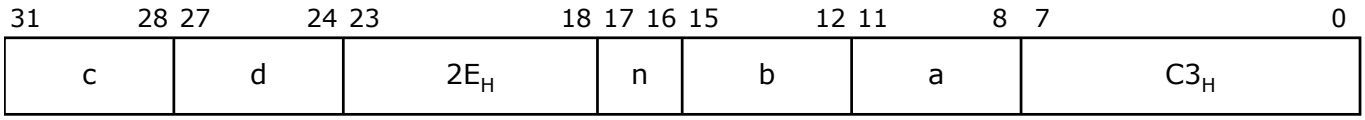
16U || 16L +||- (16U * 16L || 16L * 16U) rounded --> 16||16 saturated

31 28	27 24	23 18	17 16	15 12	11 8	7 0
c	d	$2D_H$	n	b	a	$C3_H$

```
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] LL, n (RRR1)

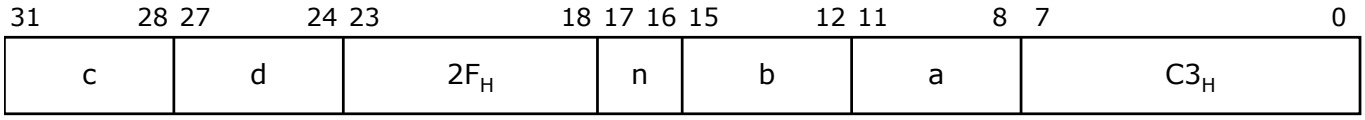
16U || 16L +||- (16U * 16L || 16L * 16L) rounded --> 16||16 saturated




```
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] 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 this instruction.

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

```
maddsurs.h    d0, d2, d4, d5UL, #0
maddsurs.h    d0, d2, d4, d5LU, #1
```

```
maddsurs.h    d0, d2, d4, d5LL, #0
maddsurs.h    d0, d2, d4, d5UU, #0
```

See Also

MADDSUR.H

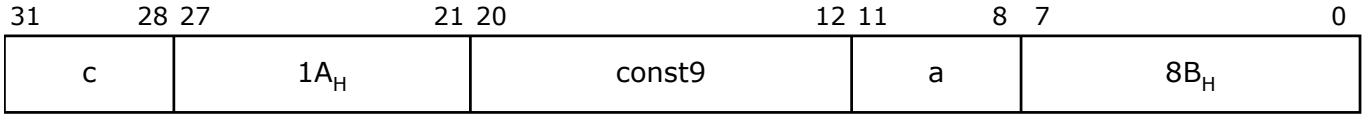
3.1.187 MAX

Maximum Value

Description

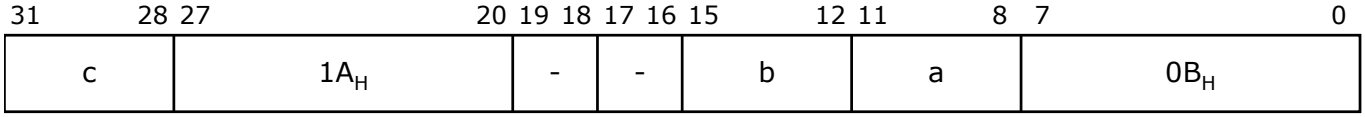
If the contents of data register D[a] are greater than the contents of either data register D[b] (instruction format RR) or const9 (instruction format RC), then put the contents of D[a] in data register D[c]; otherwise put the contents of either D[b] (format RR) or const9 (format RC) in D[c]. The operands are treated as 32-bit signed integers.

MAX D[c], D[a], const9 (RC)




```
D[c] = (D[a] > sign_ext(const9)) ? D[a] : sign_ext(const9);
```

MAX D[c], D[a], D[b] (RR)




```
D[c] = (D[a] > D[b]) ? 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

```
max    d3, d1, #126
max    d3, d1, d2
```

See Also

MAX.U, MIN, MIN.U, MOV

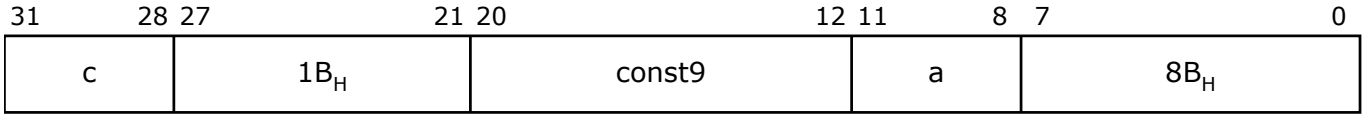
3.1.188 MAX.U

Maximum Value Unsigned

Description

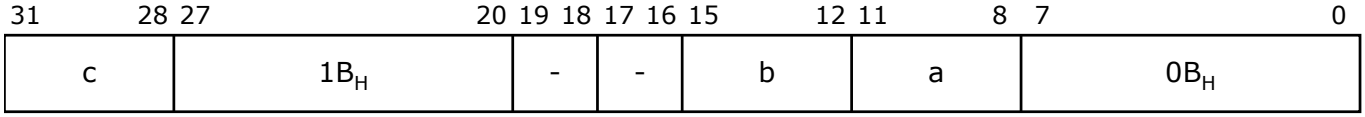
If the contents of data register D[a] are greater than the contents of either data register D[b] (instruction format RR) or const9 (instruction format RC), then put the contents of D[a] in data register D[c]; otherwise put the contents of either D[b] (format RR) or const9 (format RC) in D[c]. The operands are treated as 32-bit unsigned integers.

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




```
D[c] = (D[a] > zero_ext(const9)) ? D[a] : zero_ext(const9); // unsigned
```

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




```
D[c] = (D[a] > D[b]) ? D[a] : D[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

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

See Also

MAX, MIN, MIN.U, MOV

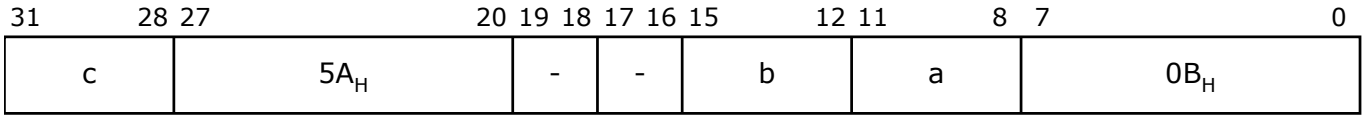
3.1.189 MAX.B

Maximum Value Packed Byte

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 8-bit signed integers.

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




```
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];
```

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

```
max.b    d3, d1, d2
```

See Also

MAX.BU, MAX.H, MAX.HU, MIN.B, MIN.BU, MIN.H, MIN.HU

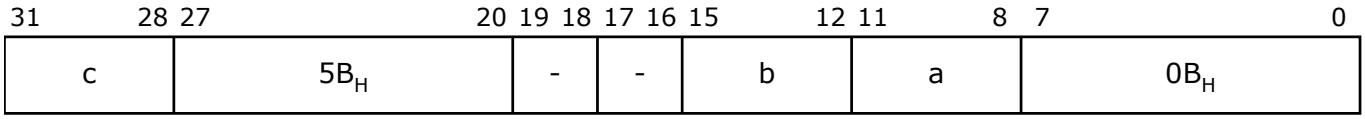
3.1.190 MAX.BU

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 8-bit unsigned integers.

MAX.BU D[c], D[a], D[b] (RR)




```
D[c][31:24] = (D[a][31:24] > D[b][31:24]) ? D[a][31:24] : D[b][31:24]; // unsigned
D[c][23:16] = (D[a][23:16] > D[b][23:16]) ? D[a][23:16] : D[b][23:16]; // unsigned
D[c][15:8] = (D[a][15:8] > D[b][15:8]) ? D[a][15:8] : D[b][15:8]; // unsigned
D[c][7:0] = (D[a][7:0] > D[b][7:0]) ? D[a][7:0] : D[b][7:0]; // 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

```
max.bu    d3, d1, d2
```

See Also

MAX.B, MAX.H, MAX.HU, MIN.B, MIN.BU, MIN.H, MIN.HU

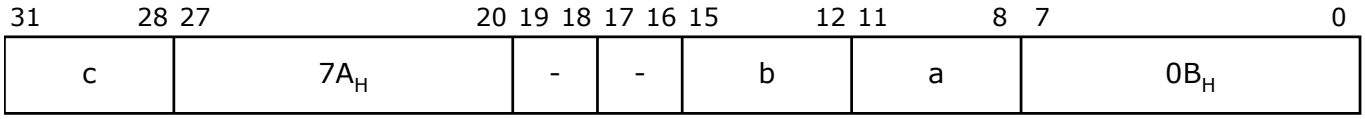
3.1.191 MAX.H

Maximum Value Packed Half-word

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 16-bit signed integers.

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




```
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];
```

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

```
max.h    d3, d1, d2
```

See Also

MAX.B, MAX.BU, MAX.HU, MIN.B, MIN.BU, MIN.H, MIN.HU

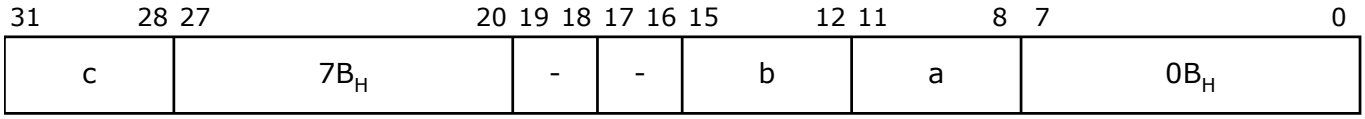
3.1.192 MAX.HU

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 16-bit unsigned integers.

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




```
D[c][31:16] = (D[a][31:16] > D[b][31:16]) ? D[a][31:16] : D[b][31:16]; // unsigned
D[c][15:0] = (D[a][15:0] > D[b][15:0]) ? D[a][15:0] : D[b][15:0]; // 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

```
max.hu    d3, d1, d2
```

See Also

MAX.B, MAX.BU, MAX.H, MIN.B, MIN.BU, MIN.H, MIN.HU

3.1.193 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. The MFCR instruction cannot be used to access the GPRs of the executing environment.

If the virtualization extension is not present or disabled, or when executing in a Guest VM

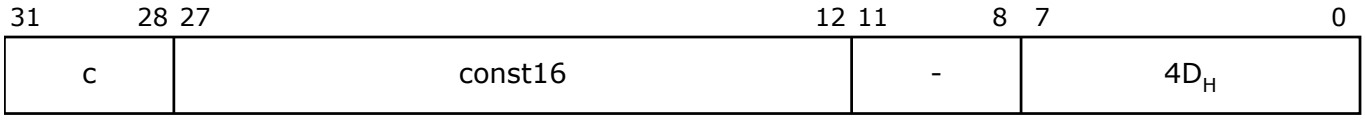
- The CSFR offset must have the least significant two bits as zero otherwise an OPD trap will result.

If the virtualization extension is present and enabled, and executing in VM0

- Using the least significant offset bits of 00_B the currently executing environment (HRHV) CSFRs are addressed.
- Using the least significant offset bits of 01_B the HRA CSFRs are addressed.
- Using the least significant offset bits of 10_B the HRB CSFRs are addressed.
- Using the least significant offset bits of 11_B will result in an OPD trap.

MFCR can be executed on any privilege level.

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, 0xfe04
```

See Also

MFDCR, MTCR, MTDCR

3.1.194 MFDCR

Move From Core Register Pair

Description

Move the contents of the pair of Core Special Function Registers (CSFR), selected by the value const16 and const16+4, to data register pair E[c]. The CSFR pair address is a const16 byte offset from the CSFR base address. The MFDCR instruction cannot be used to access the GPRs of the executing environment.

If the virtualization extension is not present or disabled, or when executing in a Guest VM

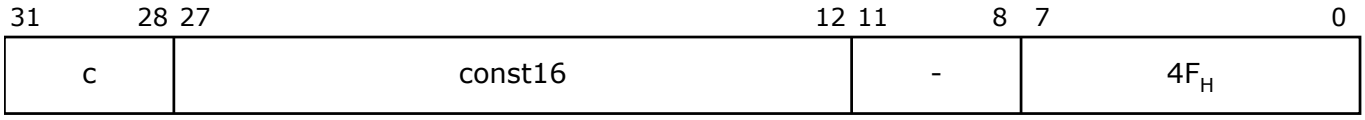
- The CSFR offset must have the least significant three bits as zero otherwise an OPD trap will result.

If the virtualization extension is present and enabled, and executing in VM0

- Using the least significant offset bits of 00_B the currently executing environment (HRHV) CSFRs are addressed.
- Using the least significant offset bits of 01_B the HRA CSFRs are addressed.
- Using the least significant offset bits of 10_B the HRB CSFRs are addressed.
- Using the least significant offset bits of 11_B will result in an OPD trap.
- Using the least significant offset bits of $1xx_B$ will result in an OPD trap.

MFDCR can be executed on any privilege level.

MFDCR E[c], const16 (RLC)




```
D[c] = CR[const16];
D[c+1] = CR[const16+4];
```

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

```
mfdcr    e2, 0xfe08
```

See Also

MFCR, MTCR, MTDCR

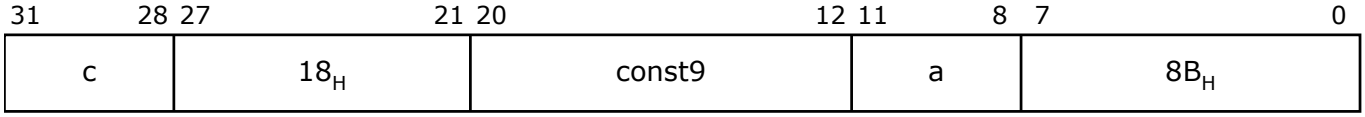
3.1.195 MIN

Minimum Value

Description

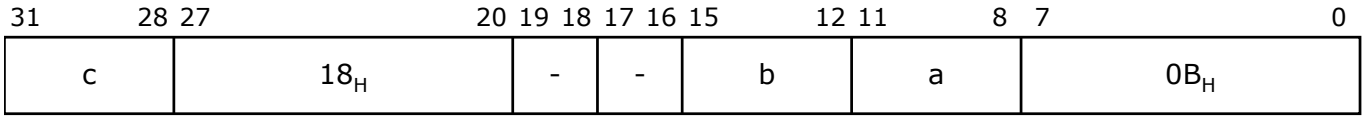
If the contents of data register D[a] are less than the contents of either data register D[b] (instruction format RR) or const9 (instruction format RC), then put the contents of D[a] in data register D[c]; otherwise put the contents of either D[b] (format RR) or const9 (format RC) in to D[c]. The operands are treated as 32-bit signed integers.

MIN D[c], D[a], const9 (RC)




```
D[c] = (D[a] < sign_ext(const9)) ? D[a] : sign_ext(const9);
```

MIN D[c], D[a], D[b] (RR)




```
D[c] = (D[a] < D[b]) ? 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

```
min    d3, d1, #126
min    d3, d1, d2
```

See Also

MAX, MAX.U, MIN.U

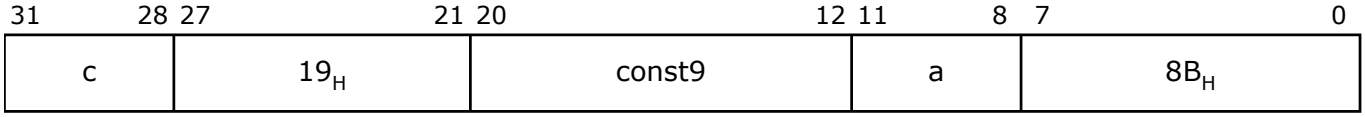
3.1.196 MIN.U

Minimum Value Unsigned

Description

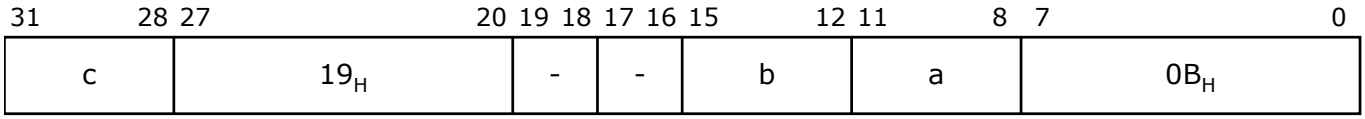
If the contents of data register D[a] are less than the contents of either data register D[b] (instruction format RR) or const9 (instruction format RC), then put the contents of D[a] in data register D[c]; otherwise put the contents of either D[b] (format RR) or const9 (format RC) in to D[c]. The operands are treated as 32-bit unsigned integers.

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




```
D[c] = (D[a] < zero_ext(const9)) ? D[a] : zero_ext(const9); // unsigned
```

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




```
D[c] = (D[a] < D[b]) ? D[a] : D[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

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

See Also

MAX, MAX.U, MIN

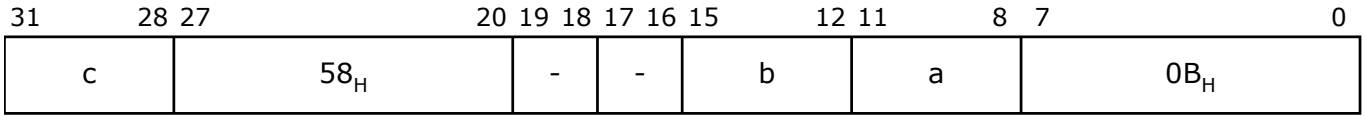
3.1.197 MIN.B

Minimum Value Packed Byte

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 8-bit signed integers.

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




```
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];
```

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

```
min.b    d3, d1, d2
```

See Also

MAX.B, MAX.BU, MAX.H, MAX.HU, MIN.BU, MIN.H, MIN.HU

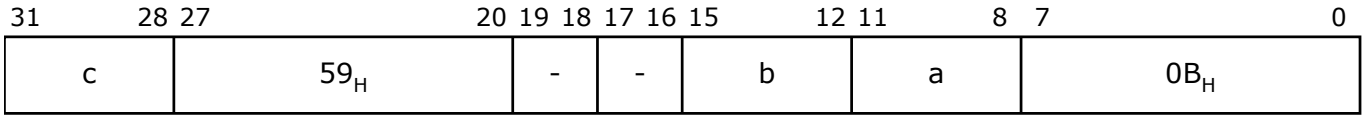
3.1.198 MIN.BU

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 8-bit unsigned integers.

MIN.BU D[c], D[a], D[b] (RR)




```
D[c][31:24] = (D[a][31:24] < D[b][31:24]) ? D[a][31:24] : D[b][31:24]; // unsigned
D[c][23:16] = (D[a][23:16] < D[b][23:16]) ? D[a][23:16] : D[b][23:16]; // unsigned
D[c][15:8] = (D[a][15:8] < D[b][15:8]) ? D[a][15:8] : D[b][15:8]; // unsigned
D[c][7:0] = (D[a][7:0] < D[b][7:0]) ? D[a][7:0] : D[b][7:0]; // 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

```
min.bu    d3, d1, d2
```

See Also

MAX.B, MAX.BU, MAX.H, MAX.HU, MIN.B, MIN.H, MIN.HU

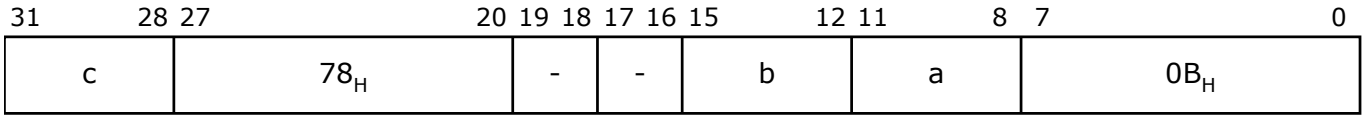
3.1.199 MIN.H

Minimum Value Packed Half-word

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 16-bit signed integers.

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




```
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];
```

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

```
min.h    d3, d1, d2
```

See Also

MAX.B, MAX.BU, MAX.H, MAX.HU, MIN.B, MIN.BU, MIN.HU

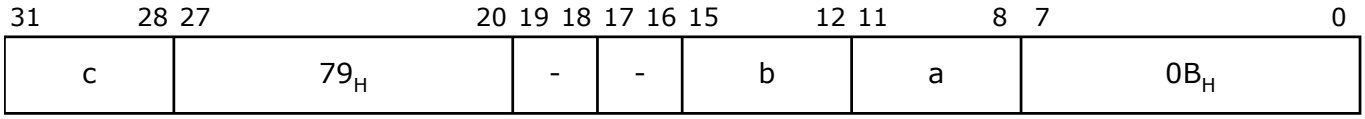
3.1.200 MIN.HU

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 16-bit unsigned integers.

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




```
D[c][31:16] = (D[a][31:16] < D[b][31:16]) ? D[a][31:16] : D[b][31:16]; // unsigned
D[c][15:0] = (D[a][15:0] < D[b][15:0]) ? D[a][15:0] : D[b][15:0]; // 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

```
min.hu    d3, d1, d2
```

See Also

MAX.B, MAX.BU, MAX.H, MAX.HU, MIN.B, MIN.BU, MIN.HU

3.1.201 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-bit before it is moved.

The syntax is also valid with a register pair as a destination. If there is a single source operand, it is sign-extended to 64-bit. If the source is a register pair the contents of data register D[a] go into E[c][63:32] and the contents of data register D[b] into E[c][31:0].

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.

The instruction also exists with a register pair as a destination. The value const4 is sign-extended to 64-bit before it is moved.

MOV D[c], const16 (RLC)

31–28	27–12	11–8	7–0
c	const16	-	$3B_H$

```
D[c] = sign_ext(const16);
```

MOV E[c], const16 (RLC)

31–28	27–12	11–8	7–0
c	const16	-	FB_H

```
E[c] = sign_ext(const16);
```

MOV D[c], D[b] (RR)

31–28	27–20	19	18	17	16–15	12–11	8–7	0
c	$1F_H$	-	-	b		-	$0B_H$	

```
D[c] = D[b];
```

MOV E[c], D[b] (RR)

31–28	27–20	19	18	17	16–15	12–11	8–7	0
c	80_H	-	-	b		-	$0B_H$	

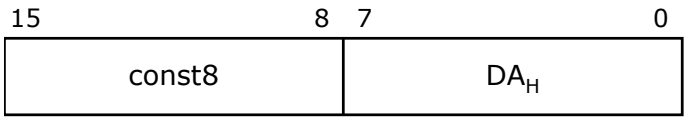
```
E[c] = sign_ext(D[b]);
```

MOV E[c], D[a], D[b] (RR)

31–28	27–20	19	18	17	16–15	12–11	8–7	0
c	81_H	-	-	b		a	$0B_H$	

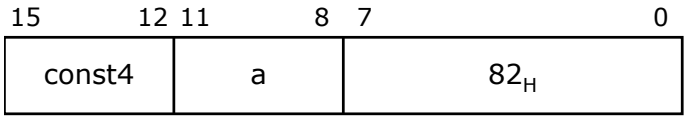
```
E[c] = {D[a], D[b]};
```

MOV D[15], const8 (SC)



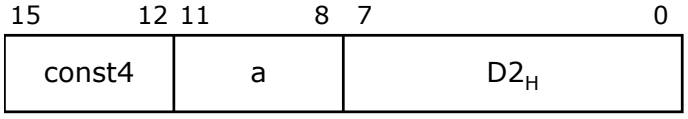

```
D[15] = zero_ext(const8);
```

MOV D[a], const4 (SRC)



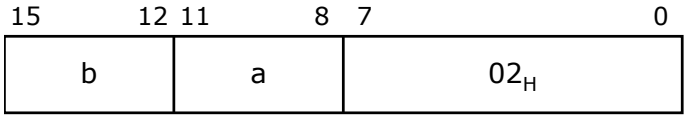

```
D[a] = sign_ext(const4);
```

MOV E[a], const4 (SRC)




```
E[a] = sign_ext(const4);
```

MOV D[a], D[b] (SRR)




```
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, #-30000
mov    e0, 0x1234; e0 = 0x0000000000001234
mov    d3, d1
mov    e0, d5; e0 = sign_ext(d5)
mov    e0, d6, d3; e0 = d6_d3 or d1=d6 d0=d3
```

```
mov    d15, #126
mov    d1, #6
mov    e0, #0xE; e0 = 0xFFFFFFFFFFFFFFFE
mov    d1, d2
```

See Also

MAX, MAX.U, MOV.U, MOVH

3.1.202 MOV.A

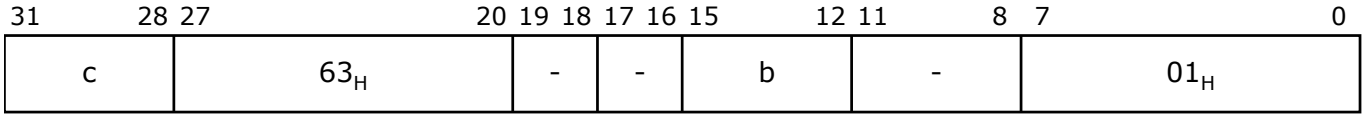
Move Value to Address Register

Description

Move the contents of data register D[b] to address register A[c].

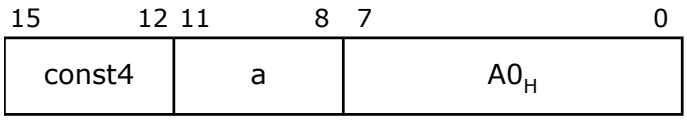
Move the contents of either data register D[b] (format SRR) or const4 (format SRC) to address register A[a]. The value const4 is zero-extended before it is moved.

MOV.A A[c], D[b] (RR)



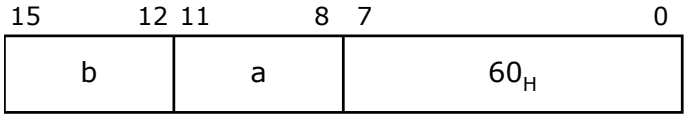

```
A[c] = D[b];
```

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, #7
mov.a    a4, d2
```

See Also

LEA, MOV.AA, MOV.D, MOVH.A

3.1.203 MOV.AA

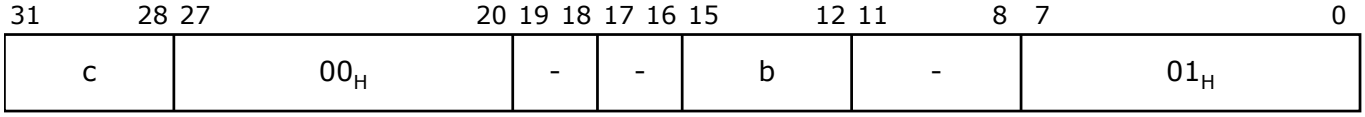
Move Address from Address Register

Description

Move the contents of address register A[b] to address register A[c].

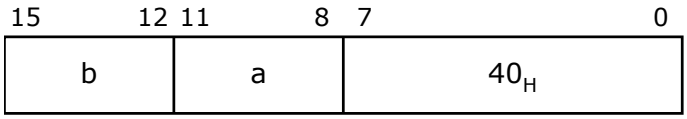
Move the contents of address register A[b] to address register A[a].

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, MOV.A, MOV.D, MOVH.A

3.1.204 MOV.D

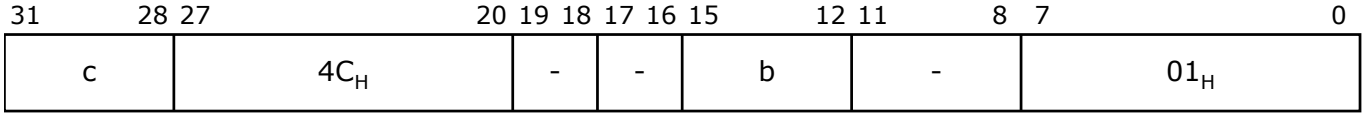
Move Address to Data Register

Description

Move the contents of address register A[b] to data register D[c].

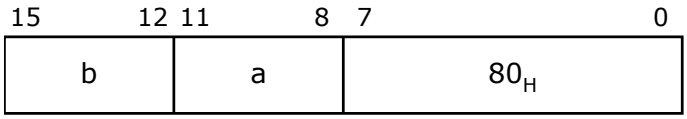
Move the contents of address register A[b] to data register D[a].

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

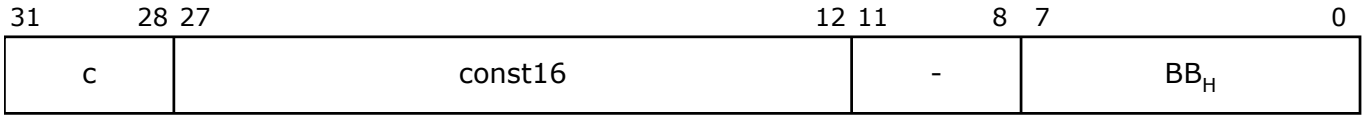
3.1.205 MOV.U

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

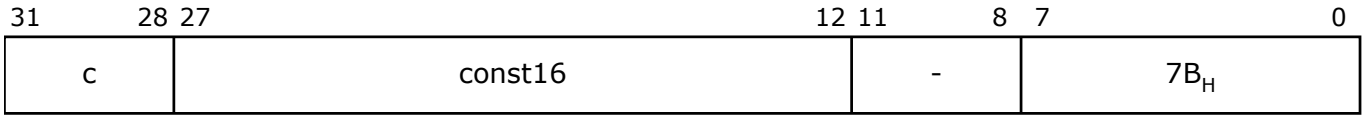
3.1.206 MOVH

Move High

Description

Move the value const16 to the most-significant half-word of data register D[c] and set the least-significant 16-bit 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

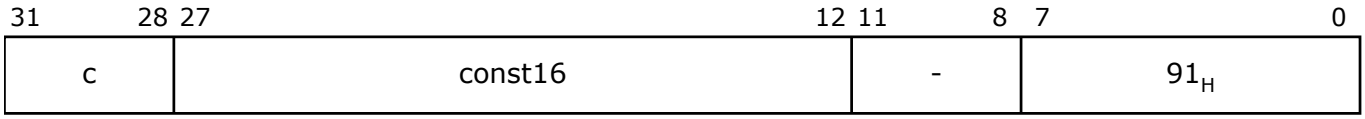
3.1.207 MOVH.A

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-bit 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

3.1.208 MSUB

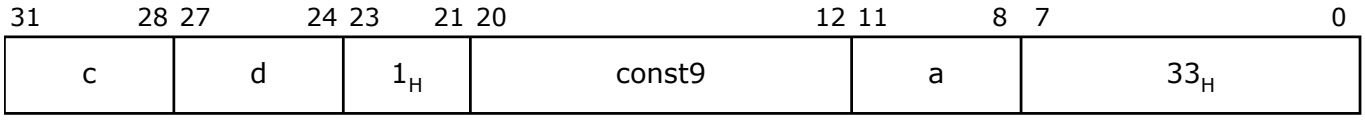
Multiply-Subtract

Description

Multiply two 32-bit signed integers. Subtract the product from a 32-bit or 64-bit signed integer and put the result into a 32-bit or 64-bit register. The value const9 is sign-extended before the multiplication is performed.

MSUB D[c], D[d], D[a], const9 (RCR)

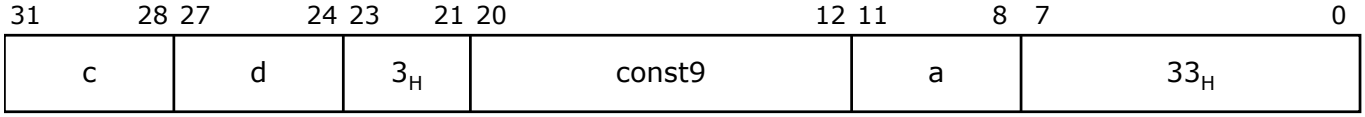
32 - (32 * K9) --> 32 signed




```
result = D[d] - (D[a] * sign_ext(const9));
D[c] = result[31:0];
```

MSUB E[c], E[d], D[a], const9 (RCR)

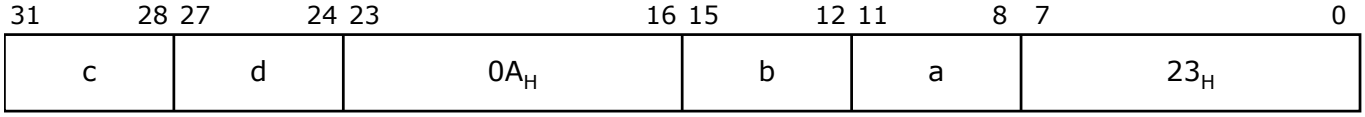
64 - (32 * K9) --> 64 signed




```
result = E[d] - (D[a] * sign_ext(const9));
E[c] = result[63:0];
```

MSUB D[c], D[d], D[a], D[b] (RRR2)

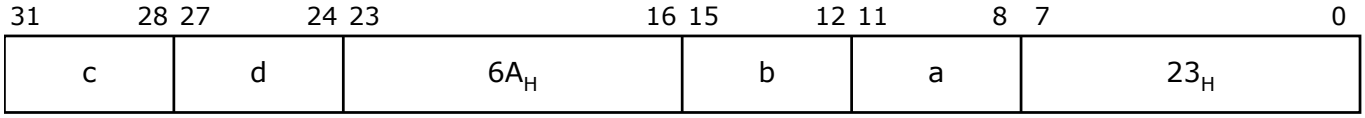
32 - (32 * 32) --> 32 signed




```
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




```
result = E[d] - (D[a] * D[b]);
E[c] = result[63:0];
```

Status Flags

C	Not set by this instruction.

V	32-bit result:
	overflow = (result > $7FFFFFFF_H$) OR (result < -80000000_H);
	if (overflow) then PSW.V = 1 else PSW.V = 0;
	64-bit result:
	overflow = (result > $7FFFFFFFFFFFFFFF_H$) OR (result < -8000000000000000_H);
	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, #7
msub    e0, e0, d3, #80
msub    d0, d1, d2, d3
msub    e0, e2, d6, d11
```

See Also

MSUBS, MUL

3.1.209 MSUBS

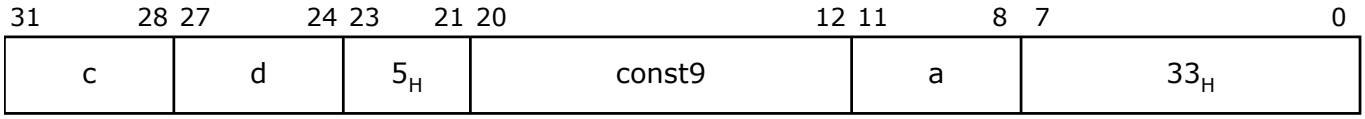
Multiply-Subtract, Saturated

Description

Multiply two 32-bit signed integers. Subtract the product from a 32-bit or 64-bit signed 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 result is saturated on overflow.

MSUBS D[c], D[d], D[a], const9 (RCR)

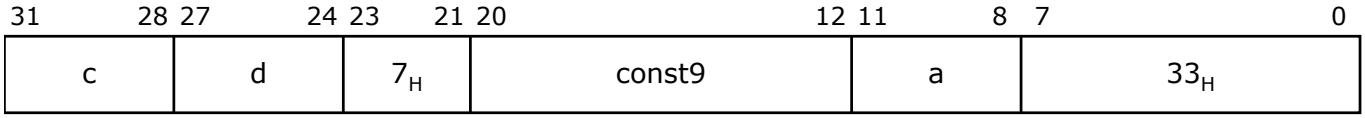
32 - (32 * K9) --> 32 signed saturated




```
result = D[d] - (D[a] * sign_ext(const9));
D[c] = ssov(result, 32);
```

MSUBS E[c], E[d], D[a], const9 (RCR)

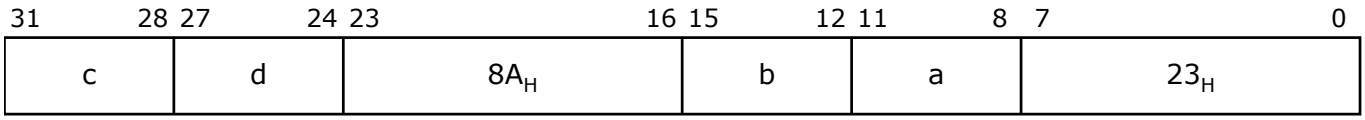
64 - (32 * K9) --> 64 signed saturated




```
result = E[d] - (D[a] * sign_ext(const9));
E[c] = ssov(result, 64);
```

MSUBS D[c], D[d], D[a], D[b] (RRR2)

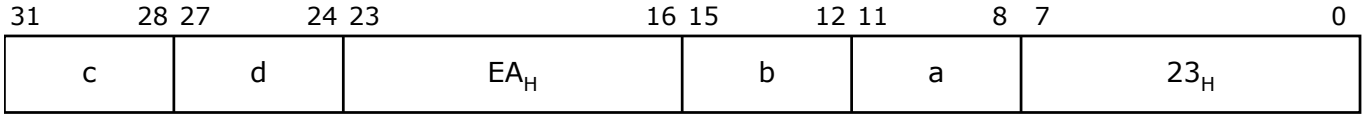
32 - (32 * 32) --> 32 signed saturated




```
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]);
E[c] = ssov(result, 64)
```

Status Flags

C	Not set by this instruction.

V	32-bit result:
	overflow = (result > $7FFFFFFF_H$) OR (result < -80000000_H);
	if (overflow) then PSW.V = 1 else PSW.V = 0;
	64-bit result:
	overflow = (result > $7FFFFFFFFFFFFFFF_H$) OR (result < -8000000000000000_H);
	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

```
msubs    d1, d1, d2, #7
msubs    e8, e4, d3, #80
msubs    d5, d1, d2, d2
msubs    e0, e2, d6, d11
```

See Also

MSUB, MUL

3.1.210 MSUB.H

Packed Multiply-Subtract Q Format

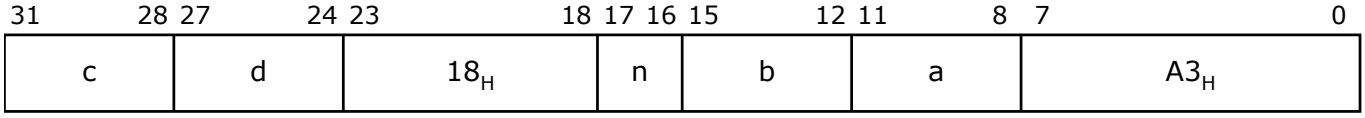
Description

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] UL, n (RRR1)

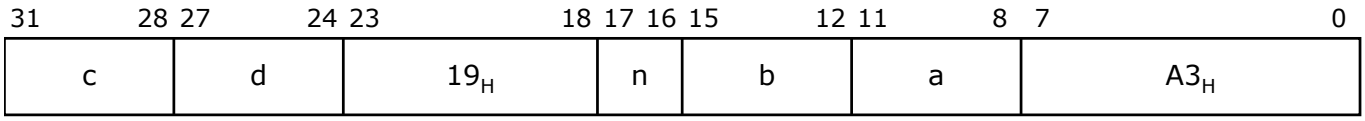
32||32 -||- (16U * 16U || 16L * 16L) --> 32||32




```
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] LU, n (RRR1)

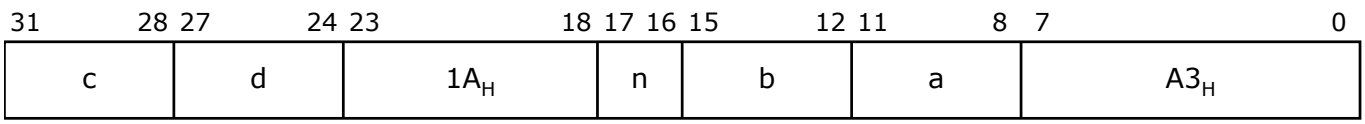
32||32 -||- (16U * 16L || 16L * 16U) --> 32||32




```
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] = {result_word1[31:0], result_word0[31:0]}; // Packed fraction
```

MSUB.H E[c], E[d], D[a], D[b] LL, n (RRR1)

32||32 -||- (16U * 16L || 16L * 16L) --> 32||32

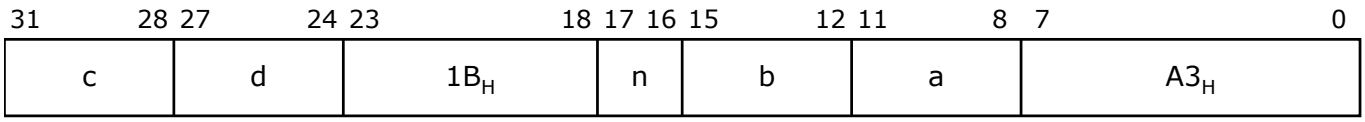



```
sc1 = (D[a][31:16] == 8000H) AND (D[b][15:0] == 8000H) AND (n == 1);
sc0 = (D[a][15:0] == 8000H) AND (D[b][15:0] == 8000H) AND (n == 1);
mul_res1 = sc1 ? 7FFFFFFFH : ((D[a][31:16] * D[b][15:0]) << n);
mul_res0 = sc0 ? 7FFFFFFFH : ((D[a][15:0] * D[b][15:0]) << n);
```

```
result_word1 = E[d][63:32] - mul_res1;
result_word0 = E[d][31:0] - mul_res0;
E[c] = {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
```

Status Flags

C	Not set by this instruction.

V	ov_word1 = (result_word1 > 7FFFFFFF$_H$) OR (result_word1 < -80000000$_H$);
	ov_word0 = (result_word0 > 7FFFFFFF$_H$) OR (result_word0 < -80000000$_H$);
	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

```
msub.h    e0, e2, d4, d5UL, #0
msub.h    e0, e2, d4, d5LU, #1
msub.h    e0, e2, d4, d5LL, #0
msub.h    e0, e2, d4, d5UU, #0
```

See Also

MSUBS.H

3.1.211 MSUBS.H

Packed Multiply-Subtract Q Format, Saturated

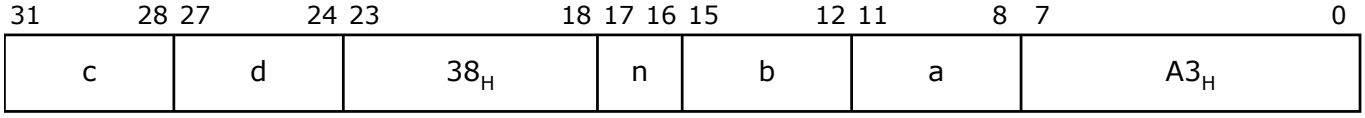
Description

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).

MSUBS.H E[c], E[d], D[a], D[b] UL, n (RRR1)

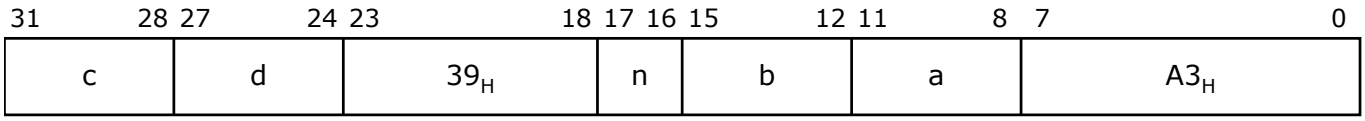
32||32 -||- (16U * 16U || 16L * 16L) --> 32||32 saturated




```
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] LU, n (RRR1)

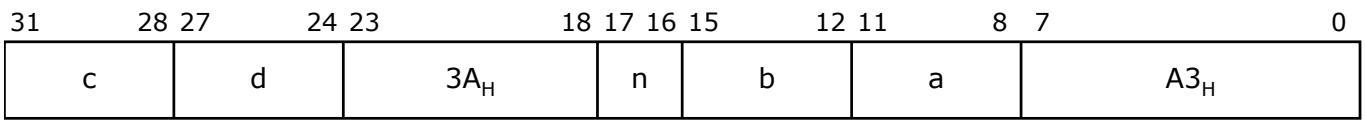
32||32 -||- (16U * 16L || 16L * 16U) --> 32||32 saturated




```
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] LL, n (RRR1)

32||32 -||- (16U * 16L || 16L * 16L) --> 32||32 saturated

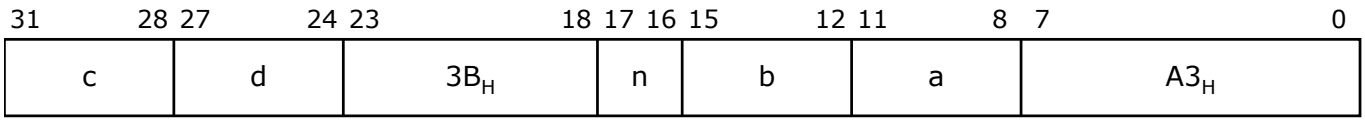



```
sc1 = (D[a][31:16] == 8000H) AND (D[b][15:0] == 8000H) AND (n == 1);
sc0 = (D[a][15:0] == 8000H) AND (D[b][15:0] == 8000H) AND (n == 1);
mul_res1 = sc1 ? 7FFFFFFFH : ((D[a][31:16] * D[b][15:0]) << n);
mul_res0 = sc0 ? 7FFFFFFFH : ((D[a][15:0] * D[b][15:0]) << n);
```

```
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] 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 this instruction.

V	ov_word1 = (result_word1 > 7FFFFFFF$_H$) OR (result_word1 < -80000000$_H$);
	ov_word0 = (result_word0 > 7FFFFFFF$_H$) OR (result_word0 < -80000000$_H$);
	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

```
msubs.h    e0, e2, d4, d5UL, #0
msubs.h    e0, e2, d4, d5LU, #1
msubs.h    e0, e2, d4, d5LL, #0
msubs.h    e0, e2, d4, d5UU, #0
```

See Also

MSUB.H

3.1.212 MSUB.Q

Multiply-Subtract Q Format

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.

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).

***Note**: In TC1.8 MSUB.Q return the same result as a MUL.Q followed by a SUB for the following cases*

- MSUB.Q D[c], D[d], D[a], D[b] U, n // 32 - (32 * 16U)Up --> 32
- MSUB.Q D[c], D[d], D[a], D[b] L, n // 32 - (32 * 16L)Up --> 32
- MSUB.Q D[c], D[d], D[a], D[b], n // 32 - (32 * 32)Up --> 32

MSUB.Q D[c], D[d], D[a], D[b] U, n (RRR1)

32 - (32 * 16U)Up --> 32

31–28	27–24	23–18	17 16	15–12	11–8	7–0
c	d	00_H	n	b	a	63_H

```
result = D[d] - (((D[a] * D[b][31:16]) << n) >> 16);
D[c] = result[31:0]; // Fraction
```

MSUB.Q D[c], D[d], D[a], D[b] L, n (RRR1)

32 - (32 * 16L)Up --> 32

31–28	27–24	23–18	17 16	15–12	11–8	7–0
c	d	01_H	n	b	a	63_H

```
result = D[d] - (((D[a] * D[b][15:0]) << n) >> 16);
D[c] = result[31:0]; // Fraction
```

MSUB.Q D[c], D[d], D[a], D[b], n (RRR1)

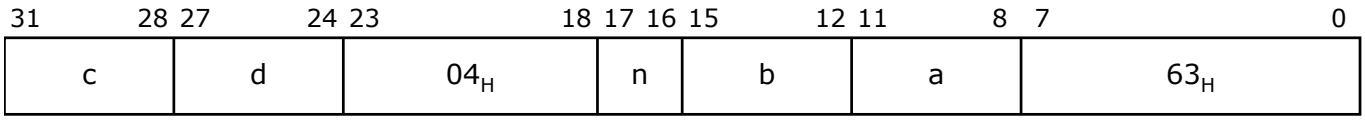
32 - (32 * 32)Up --> 32

31–28	27–24	23–18	17 16	15–12	11–8	7–0
c	d	02_H	n	b	a	63_H

```
result = D[d] - (((D[a] * D[b]) << n) >> 32);
D[c] = result[31:0]; // Fraction
```

MSUB.Q D[c], D[d], D[a] U, D[b] U, n (RRR1)

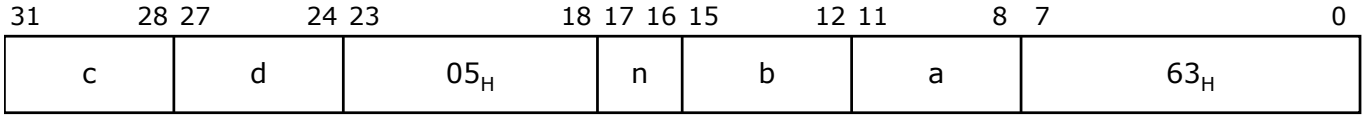
32 - (16U * 16U) --> 32




```
sc = (D[a][31:16] == 8000H) AND (D[b][31:16] == 8000H) AND (n == 1);
mul_res = sc ? 7FFFFFFFH : ((D[a][31:16] * D[b][31:16]) << n);
result = D[d] - mul_res;
D[c] = result[31:0]; // Fraction
```

MSUB.Q D[c], D[d], D[a] L, D[b] L, n (RRR1)

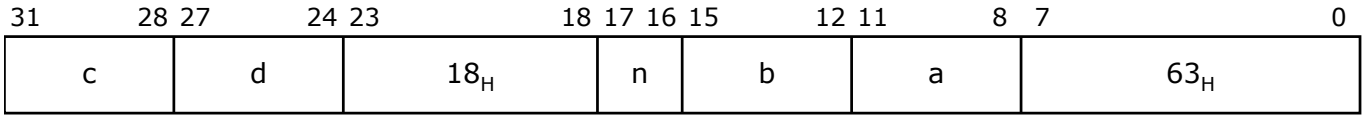
32 - (16L * 16L) --> 32




```
sc = (D[a][15:0] == 8000H) AND (D[b][15:0] == 8000H) AND (n == 1);
mul_res = sc ? 7FFFFFFFH : ((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], D[b] U, n (RRR1)

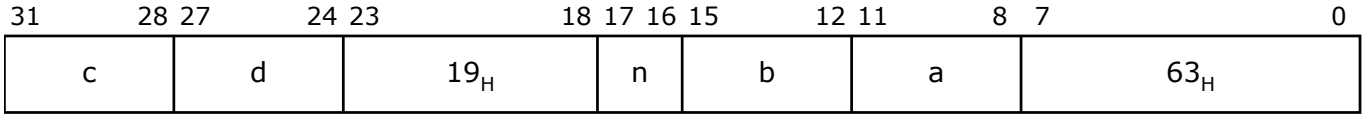
64 - (32 * 16U) --> 64




```
result = E[d] - ((D[a] * D[b][31:16]) << n);
E[c] = result[63:0]; // Multi-precision accumulator
```

MSUB.Q E[c], E[d], D[a], D[b] L, n (RRR1)

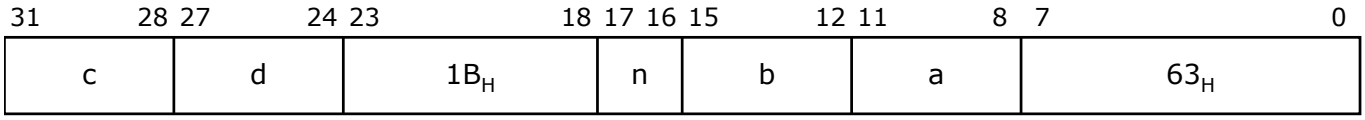
64 - (32 * 16L) --> 64




```
result = E[d] - ((D[a] * D[b][15:0]) << n);
E[c] = result[63:0]; // Multi-precision accumulator
```

MSUB.Q E[c], E[d], D[a], D[b], n (RRR1)

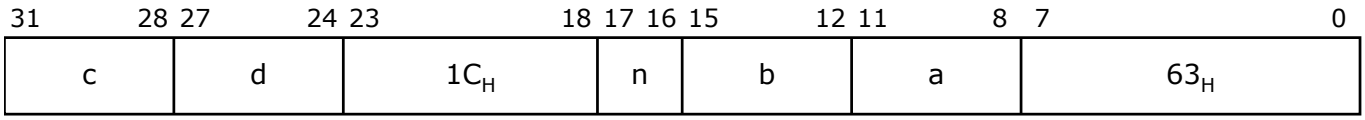
64 - (32 * 32) --> 64




```
result = E[d] - ((D[a] * D[b]) << n);
E[c] = result[63:0]; // Multi-precision fraction
```

MSUB.Q E[c], E[d], D[a] U, D[b] U, n (RRR1)

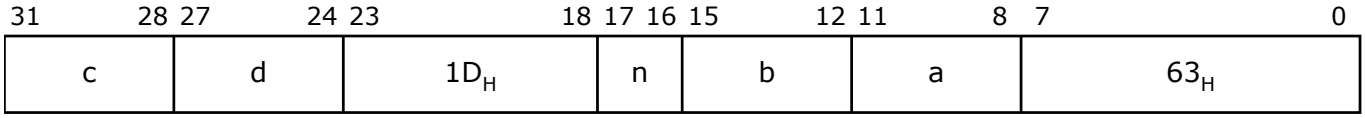
64 - (16U * 16U) --> 64




```
sc = (D[a][31:16] == 8000H) AND (D[b][31:16] == 8000H) AND (n == 1);
mul_res = sc ? 7FFFFFFFH : ((D[a][31:16] * D[b][31:16]) << n);
result = E[d] - (mul_res << 16);
E[c] = result[63:0]; // Multi-precision accumulator
```

MSUB.Q E[c], E[d], D[a] L, D[b] L, n (RRR1)

64 - (16L * 16L) --> 64




```
sc = (D[a][15:0] == 8000H) AND (D[b][15:0] == 8000H) AND (n == 1);
mul_res = sc ? 7FFFFFFFH : ((D[a][15:0] * D[b][15:0]) << n);
result = E[d] - (mul_res << 16);
E[c] = result[63:0]; // Multi-precision accumulator
```

Status Flags

C	Not set by this instruction.

V	32-bit result:
	overflow = (result > 7FFFFFFF$_H$) OR (result < -80000000$_H$)
	if (overflow) then PSW.V = 1 else PSW.V = 0;
	64-bit result:
	overflow = (result > 7FFFFFFFFFFFFFFF$_H$) OR (result < -8000000000000000$_H$);
	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, d6U, #1
msub.q    d0, d2, d1, d3L, #1
msub.q    d0, d1, d2, d3, #1
msub.q    d2, d0, d3U, d4U, #1
msub.q    d2, d0, d4L, d4L, #1
```

```
msub.q    e2, e2, d4, d6U, #1
msub.q    e2, e2, d5, d6L, #1
msub.q    e2, e2, d3, d7, #1
msub.q    e2, e2, d6U, d7U, #1
msub.q    e2, e2, d8L, d0L, #1
```

See Also

MSUBS.Q

3.1.213 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 result is saturated on overflow.

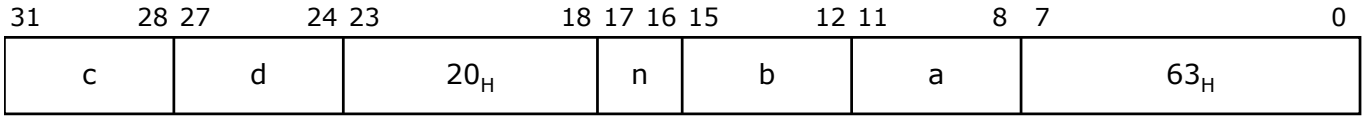
Note that n should only take the values 0 or 1, any other value returns an undefined result. If (n == 1) then 8000_H * 8000_H = $7FFFFFFF_H$ (for signed 16-bit * 16-bit multiplications only).

***Note**: In TC1.8 MSUB.Q return the same result as a MUL.Q followed by a SUBS for the following cases:*

- MSUBS.Q D[c], D[d], D[a], D[b] U, n // 32 - (32 * 16U)Up --> 32
- MSUBS.Q D[c], D[d], D[a], D[b] L, n // 32 - (32 * 16L)Up --> 32
- MSUBS.Q D[c], D[d], D[a], D[b], n // 32 - (32 * 32)Up --> 32

MSUBS.Q D[c], D[d], D[a], D[b] U, n (RRR1)

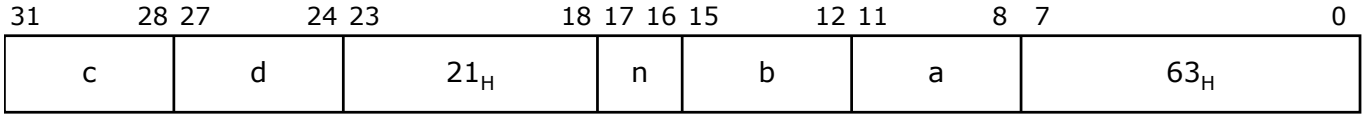
32 - (32 * 16U)Up --> 32 saturated




```
result = D[d] - (((D[a] * D[b][31:16]) << n) >> 16);
D[c] = ssov(result, 32); // Fraction
```

MSUBS.Q D[c], D[d], D[a], D[b] L, n (RRR1)

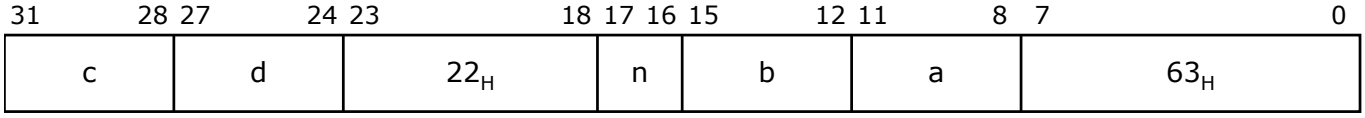
32 - (32 * 16L)Up --> 32 saturated




```
result = D[d] - (((D[a] * D[b][15:0]) << n) >> 16);
D[c] = ssov(result, 32); // Fraction
```

MSUBS.Q D[c], D[d], D[a], D[b], n (RRR1)

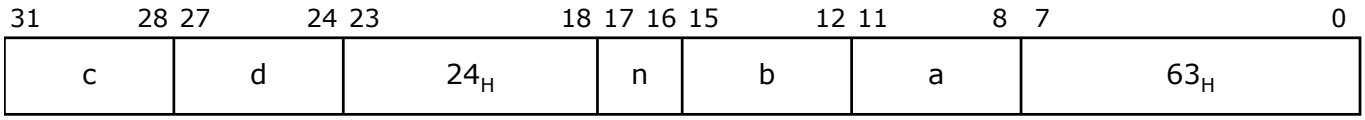
32 - (32 * 32)Up --> 32 saturated




```
result = D[d] - (((D[a] * D[b]) << n) >> 32);
D[c] = ssov(result, 32); // Fraction
```

MSUBS.Q D[c], D[d], D[a] U, D[b] U, n (RRR1)

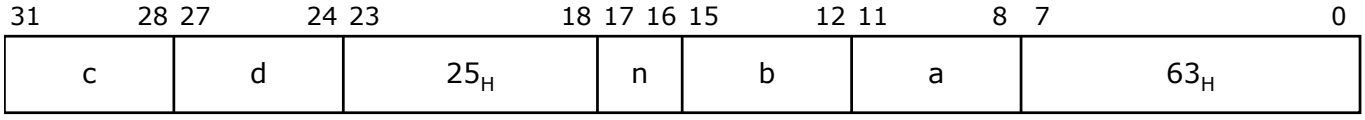
32 - (16U * 16U) --> 32 saturated




```
sc = (D[a][31:16] == 8000H) AND (D[b][31:16] == 8000H) AND (n == 1);
mul_res = sc ? 7FFFFFFFH : ((D[a][31:16] * D[b][31:16]) << n);
result = D[d] - mul_res;
D[c] = ssov(result, 32); // Fraction
```

MSUBS.Q D[c], D[d], D[a] L, D[b] L, n (RRR1)

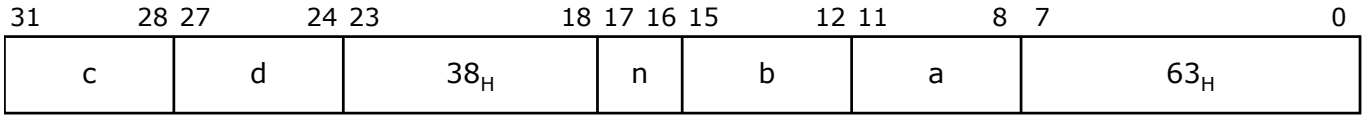
32 - (16L * 16L) --> 32 saturated




```
sc = (D[a][15:0] == 8000H) AND (D[b][15:0] == 8000H) AND (n == 1);
mul_res = sc ? 7FFFFFFFH : ((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], D[b] U, n (RRR1)

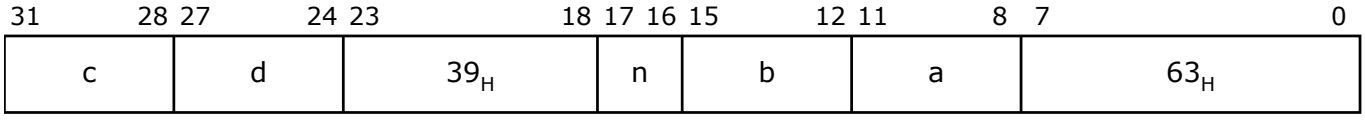
64 - (32 * 16U) --> 64 saturated




```
result = E[d] - ((D[a] * D[b][31:16]) << n);
E[c] = ssov(result, 64); // Multi-precision accumulator
```

MSUBS.Q E[c], E[d], D[a], D[b] L, n (RRR1)

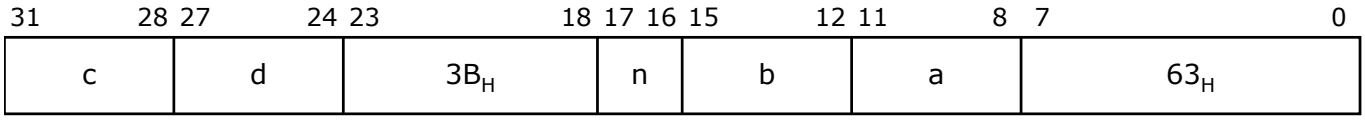
64 - (32 * 16L) --> 64 saturated




```
result = E[d] - ((D[a] * D[b][15:0]) << n);
E[c] = ssov(result, 64); // Multi-precision accumulator
```

MSUBS.Q E[c], E[d], D[a], D[b], n (RRR1)

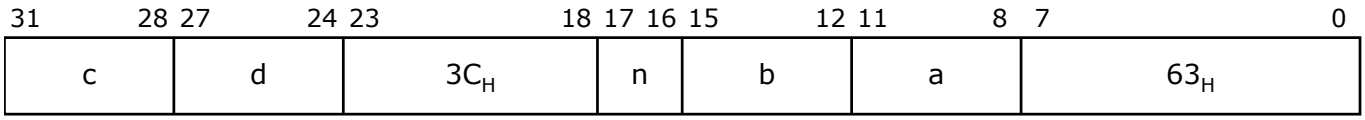
64 - (32 * 32) --> 64 saturated




```
result = E[d] - ((D[a] * D[b]) << n);
E[c] = ssov(result, 64); // Multi-precision fraction
```

MSUBS.Q E[c], E[d], D[a] U, D[b] U, n (RRR1)

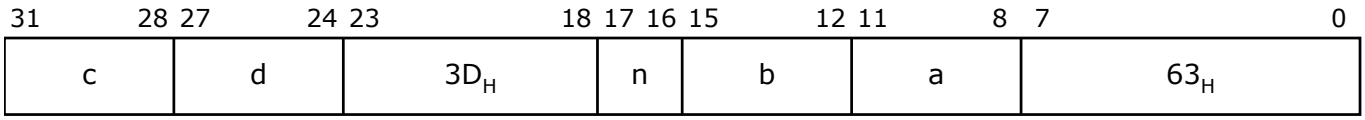
64 - (16U * 16U) --> 64 saturated




```
sc = (D[a][31:16] == 8000H) AND (D[b][31:16] == 8000H) AND (n == 1);
mul_res = sc ? 7FFFFFFFH : ((D[a][31:16] * D[b][31:16]) << n);
result = E[d] - (mul_res << 16);
E[c] = ssov(result, 64); // Multi-precision accumulator
```

MSUBS.Q E[c], E[d], D[a] L, D[b] L, n (RRR1)

64 - (16L * 16L) --> 64 saturated




```
sc = (D[a][15:0] == 8000H) AND (D[b][15:0] == 8000H) AND (n == 1);
mul_res = sc ? 7FFFFFFFH : ((D[a][15:0] * D[b][15:0]) << n);
result = E[d] - (mul_res << 16);
E[c] = ssov(result, 64); // Multi-precision accumulator
```

Status Flags

C	Not set by this instruction.

V	32-bit result:
	overflow = (result > 7FFFFFFF$_H$) OR (result < -80000000$_H$)
	if (overflow) then PSW.V = 1 else PSW.V = 0;
	64-bit result:
	overflow = (result > 7FFFFFFFFFFFFFFF$_H$) OR (result < -8000000000000000$_H$);
	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

```
msubs.q    d0, d1, d2, d6U, #1
msubs.q    d0, d2, d1, d3L, #1
msubs.q    d0, d1, d2, d3, #1
msubs.q    d2, d0, d3U, d4U, #1
msubs.q    d2, d0, d4L, d4L, #1
```

```
msubs.q    e2, e2, d4, d6U, #1
msubs.q    e2, e2, d5, d6L, #1
msubs.q    e2, e2, d3, d7, #1
msubs.q    e2, e2, d6U, d7U, #1
msubs.q    e2, e0, d11L, d4L, #1
```

See Also

MSUB.Q

3.1.214 MSUB.U

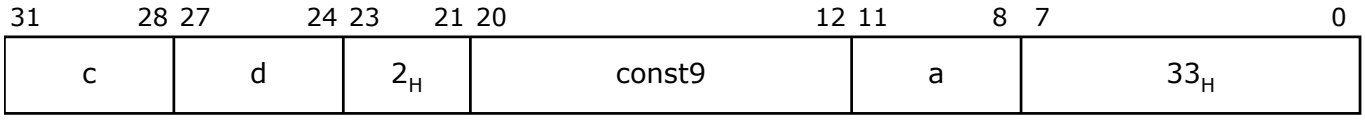
Multiply-Subtract Unsigned

Description

Multiply two 32-bit unsigned integers. Subtract the product from a 32-bit or 64-bit unsigned integer and put the result into a 32-bit or 64-bit register. The value const9 is zero-extended before the multiplication is performed.

MSUB.U E[c], E[d], D[a], const9 (RCR)

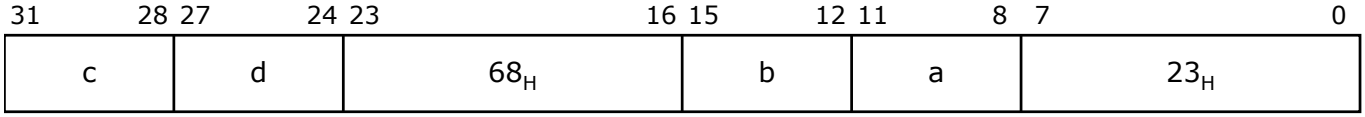
64 - (32 * K9) --> 64 unsigned




```
result = E[d] - (D[a] * zero_ext(const9)); // unsigned operators
E[c] = result[63:0];
```

MSUB.U E[c], E[d], D[a], D[b] (RRR2)

64 - (32 * 32) --> 64 unsigned




```
result = E[d] - (D[a] * D[b]); // unsigned operators
E[c] =  result[63:0];
```

Status Flags

C	Not set by this instruction.

V	32-bit result:
	overflow = (result > $FFFFFFFF_H$) OR (result < 00000000_H);
	if (overflow) then PSW.V = 1 else PSW.V = 0;
	64-bit result:
	overflow = (result > $FFFFFFFFFFFFFFFF_H$) OR (result < 0000000000000000_H);
	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.u    e0, e0, d3, #80
msub.u    e0, e2, d6, d11
```

See Also

MSUBS.U

3.1.215 MSUBS.U

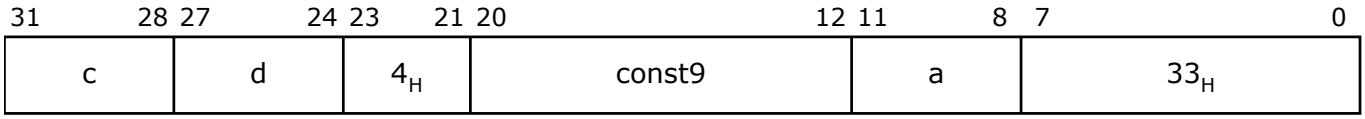
Multiply-Subtract Unsigned, Saturated

Description

Multiply two 32-bit unsigned integers. Subtract the product from a 32-bit or 64-bit unsigned 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 results are saturated on overflow.

MSUBS.U D[c], D[d], D[a], const9 (RCR)

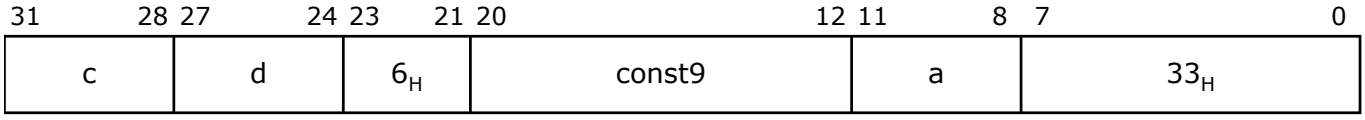
32 - (32 * K9) --> 32 unsigned saturated




```
result = D[d] - (D[a] * zero_ext(const9)); // unsigned operators
D[c] = suov(result, 32);
```

MSUBS.U E[c], E[d], D[a], const9 (RCR)

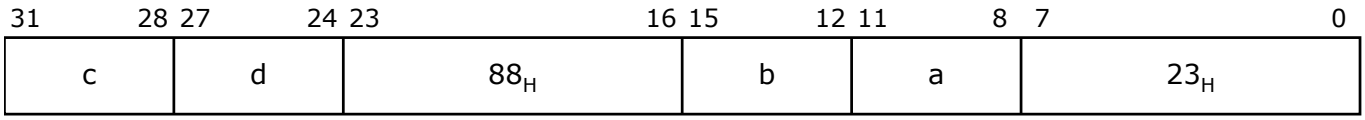
64 - (32 * K9) --> 64 unsigned saturated




```
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)

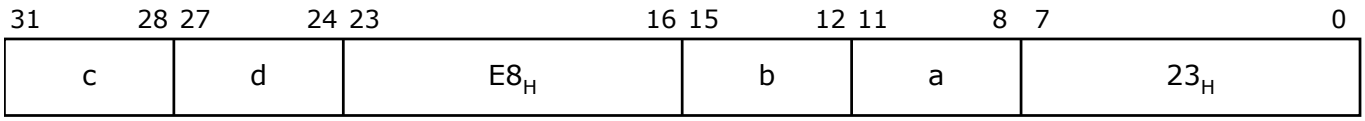
32 - (32 * 32) --> 32 unsigned saturated




```
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 this instruction.

V	32-bit result:
	overflow = (result > $FFFFFFFF_H$) OR (result < 00000000_H);
	if (overflow) then PSW.V = 1 else PSW.V = 0;
	64-bit result:
	overflow = (result > $FFFFFFFFFFFFFFFF_H$) OR (result < 0000000000000000_H);
	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

```
msubs.u    d1, d1, d2, #7
msubs.u    e8, e4, d3, #80
msubs.u    d5, d1, d2, d2
msubs.u    e0, e2, d6, d11
```

See Also

MSUB.U

3.1.216 MSUBAD.H

Packed Multiply-Subtract/Add Q Format

Description

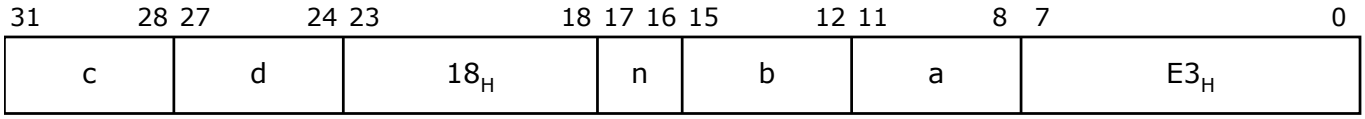
Multiply two signed 16-bit (half-word) values. Subtract (or add) the product (left justified if n == 1) from a signed 32-bit value and put the result into a 32-bit register. There are four cases of half-word multiplication:

- 16U * 16U, 16L * 16L
- 16U * 16L, 16L * 16U
- 16U * 16L, 16L * 16L
- 16L * 16U, 16U * 16U

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] UL, n (RRR1)

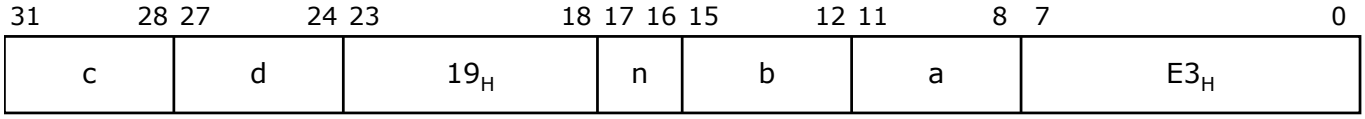
32||32 -||+ (16U * 16U || 16L * 16L) --> 32||32




```
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] LU, n (RRR1)

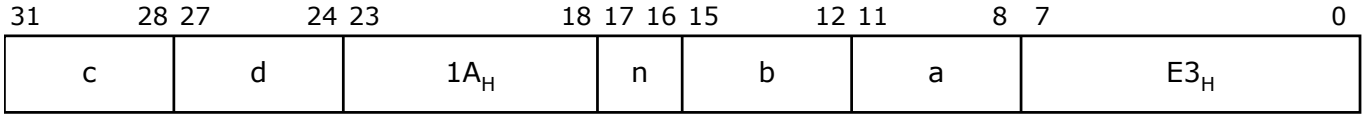
32||32 -||+ (16U * 16L || 16L * 16U) --> 32||32




```
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] = {result_word1[31:0], result_word0[31:0]}; // Packed fraction
```

MSUBAD.H E[c], E[d], D[a], D[b] LL, n (RRR1)

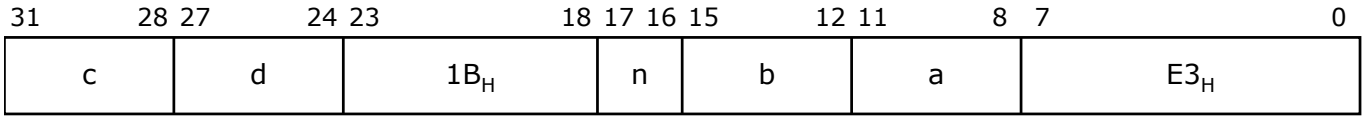
32||32 -||+ (16U * 16L || 16L * 16L) --> 32||32



```
sc1 = (D[a][31:16] == 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] = {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
```

Status Flags

C	Not set by this instruction.

V	ov_word1 = (result_word1 > 7FFFFFFFH) OR (result_word1 < -80000000H);
	ov_word0 = (result_word0 > 7FFFFFFFH) OR (result_word0 < -80000000H);
	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

```
msubad.h    e0, e2, d4, d5UL, #0
msubad.h    e0, e2, d4, d5LU, #1
msubad.h    e0, e2, d4, d5LL, #0
msubad.h    e0, e2, d4, d5UU, #0
```

See Also

MSUBADS.H

3.1.217 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

Each 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).

MSUBADS.H E[c], E[d], D[a], D[b] UL, n (RRR1)

32||32 -||+ (16U * 16U || 16L * 16L) --> 32||32 saturated

31 28	27 24	23 18	17 16	15 12	11 8	7 0
c	d	38_H	n	b	a	$E3_H$

```
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
```

MSUBADS.H E[c], E[d], D[a], D[b] LU, n (RRR1)

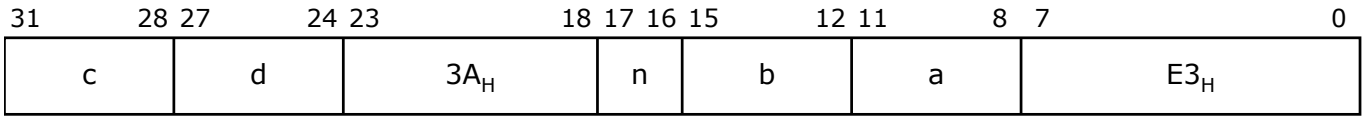
32||32 -||+ (16U * 16L || 16L * 16U) --> 32||32 saturated

31 28	27 24	23 18	17 16	15 12	11 8	7 0
c	d	39_H	n	b	a	$E3_H$

```
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
```

MSUBADS.H E[c], E[d], D[a], D[b] LL, n (RRR1)

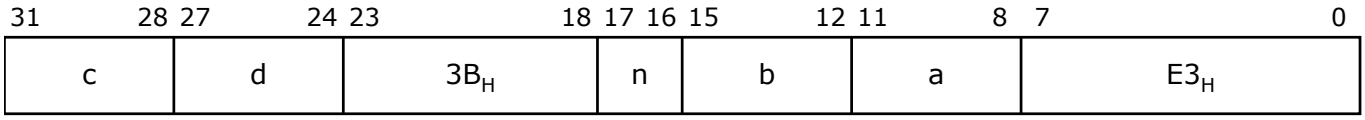
32||32 -||+ (16U * 16L || 16L * 16L) --> 32||32 saturated




```
sc1 = (D[a][31:16] == 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
```

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] == 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 this instruction.

V	ov_word1 = (result_word1 > 7FFFFFFFH) OR (result_word1 < -80000000H);
	ov_word0 = (result_word0 > 7FFFFFFFH) OR (result_word0 < -80000000H);
	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

```
msubads.h    e0, e2, d4, d5UL, #0
msubads.h    e0, e2, d4, d5LU, #1
msubads.h    e0, e2, d4, d5LL, #0
msubads.h    e0, e2, d4, d5UU, #0
```

See Also

MSUBAD.H

3.1.218 MSUBADM.H

Packed Multiply-Subtract/Add Q Format-Multi-precision

Description

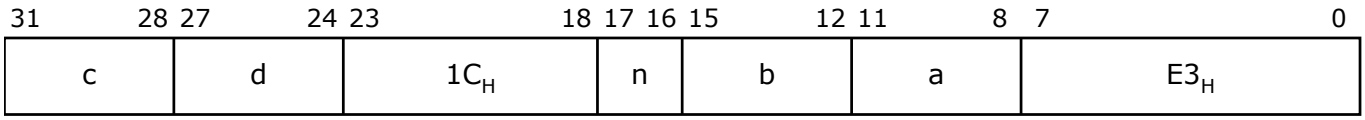
Perform two multiplications of two signed 16-bit (half-word) values. Subtract one product and add the other product (left justified if n == 1) left-shifted by 16, from/to a signed 64-bit value and put the result in a 64-bit register. There are four cases of half-word multiplication:

- 16U * 16U, 16L * 16L
- 16U * 16L, 16L * 16U
- 16U * 16L, 16L * 16L
- 16L * 16U, 16U * 16U

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] UL, n (RRR1)

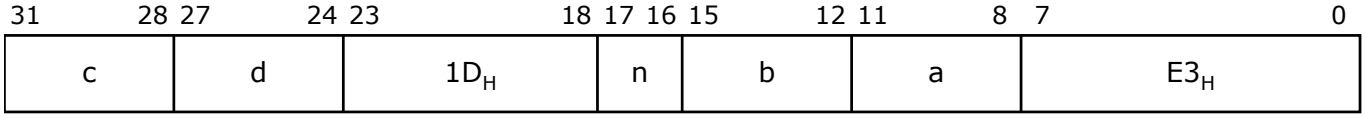
64 - (16U * 16U) + (16L * 16L) --> 64




```
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] = result[63:0]; // Multi-precision accumulator
```

MSUBADM.H E[c], E[d], D[a], D[b] LU, n (RRR1)

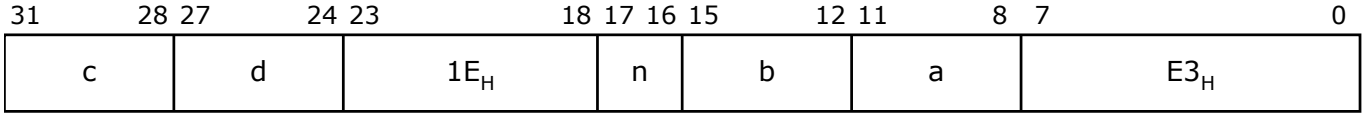
64 - (16U * 16L) + (16L * 16U) --> 64




```
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] = result[63:0]; // Multi-precision accumulator
```

MSUBADM.H E[c], E[d], D[a], D[b] LL, n (RRR1)

64 - (16U * 16L) + (16L * 16L) --> 64

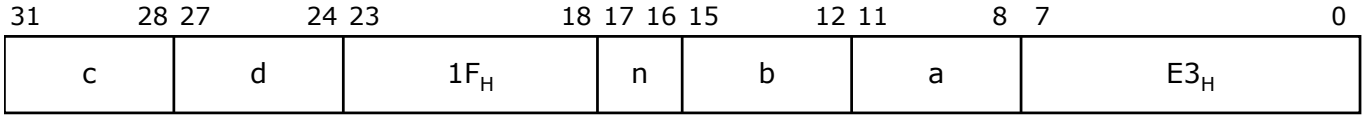



```
sc1 = (D[a][31:16] == 8000H) AND (D[b][15:0] == 8000H) AND (n == 1);
```

```
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] = 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] == 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] = result[63:0]; // Multi-precision accumulator
```

Status Flags

C	Not set by this instruction.
V	overflow = (result > 7FFFFFFFFFFFFFFFH) OR (result < -8000000000000000H); 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

```
msubadm.h    e0, e2, d4, d5UL, #0
msubadm.h    e0, e2, d4, d5LU, #1
msubadm.h    e0, e2, d4, d5LL, #0
msubadm.h    e0, e2, d4, d5UU, #0
```

See Also

MSUBADMS.H

3.1.219 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 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).

MSUBADMS.H E[c], E[d], D[a], D[b] UL, n (RRR1)

64 - (16U * 16U) + (16L * 16L) --> 64 saturated

31 28	27 24	23 18	17 16	15 12	11 8	7 0
c	d	$3C_H$	n	b	a	$E3_H$

```
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] LU, n (RRR1)

64 - (16U * 16L) + (16L * 16U) --> 64 saturated

31 28	27 24	23 18	17 16	15 12	11 8	7 0
c	d	$3D_H$	n	b	a	$E3_H$

```
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] LL, n (RRR1)

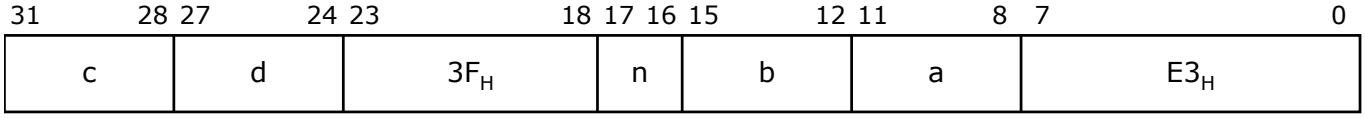
64 - (16U * 16L) + (16L * 16L) --> 64 saturated

31 28	27 24	23 18	17 16	15 12	11 8	7 0
c	d	$3E_H$	n	b	a	$E3_H$

```
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] 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);
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 this instruction.
V	overflow = (result > 7FFFFFFFFFFFFFFFH) OR (result < -8000000000000000H); 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

```
msubadms.h    e0, e2, d4, d5UL, #0
msubadms.h    e0, e2, d4, d5LU, #1
msubadms.h    e0, e2, d4, d5LL, #0
msubadms.h    e0, e2, d4, d5UU, #0
```

See Also

MSUBADM.H

3.1.220 MSUBADR.H

Packed Multiply-Subtract/Add Q Format with Rounding

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] UL, n (RRR1)

16U||16L -||+ (16U * 16U || 16L * 16L) rounded --> 16||16

31 28	27 24	23 18	17 16	15 12	11 8	7 0
c	d	$0C_H$	n	b	a	$E3_H$

```
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
```

MSUBADR.H D[c], D[d], D[a], D[b] LU, n (RRR1)

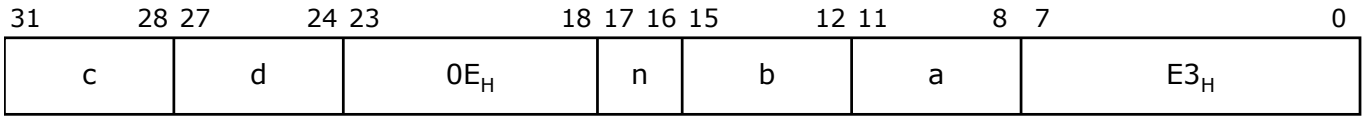
16U || 16L -||+ (16U * 16L || 16L * 16U) rounded --> 16||16

31 28	27 24	23 18	17 16	15 12	11 8	7 0
c	d	$0D_H$	n	b	a	$E3_H$

```
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] = {result_halfword1[31:16], result_halfword0[31:16]}; // Packed short fraction
```

MSUBADR.H D[c], D[d], D[a], D[b] LL, n (RRR1)

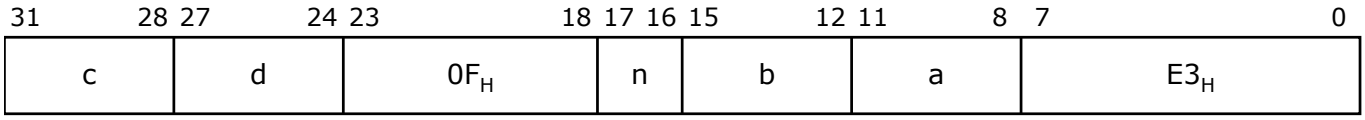
16U || 16L -||+ (16U * 16L || 16L * 16L) rounded --> 16||16




```
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] = {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] == 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
```

Status Flags

C	Not set by this instruction.

V	ov_halfword1 = (result_halfword1 > 7FFFFFFFH) OR (result_halfword1 < -80000000H);
	ov_halfword0 = (result_halfword0 > 7FFFFFFFH) OR (result_halfword0 < -80000000H);
	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

```
msubadr.h    d0, d2, d4, d5UL, #0
msubadr.h    d0, d2, d4, d5LU, #1
msubadr.h    d0, d2, d4, d5LL, #0
msubadr.h    d0, d2, d4, d5UU, #0
```

See Also

MSUBADRS.H

3.1.221 MSUBADRS.H

Packed Multiply-Subtract/Add Q Format with Rounding, Saturated

Description

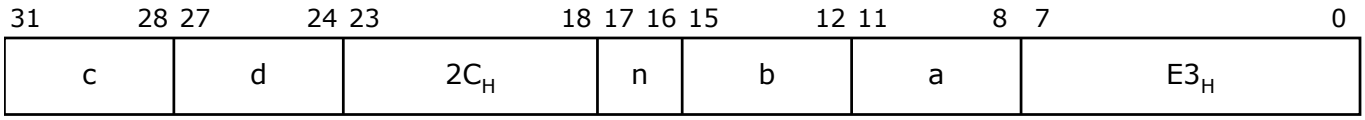
Multiply two signed 16-bit (half-word) values. Subtract (or add) the product (left justified if n == 1) from (to) a signed 16-bit value and put the rounded result into half of a 32-bit register (Note that since there are two results the two register halves are used). There are four cases of half-word 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).

MSUBADRS.H D[c], D[d], D[a], D[b] UL, n (RRR1)

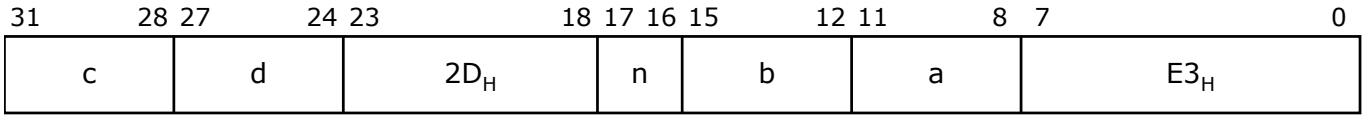
16U || 16L -||+ (16U * 16U || 16L * 16L) rounded --> 16||16 saturated




```
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] LU, n (RRR1)

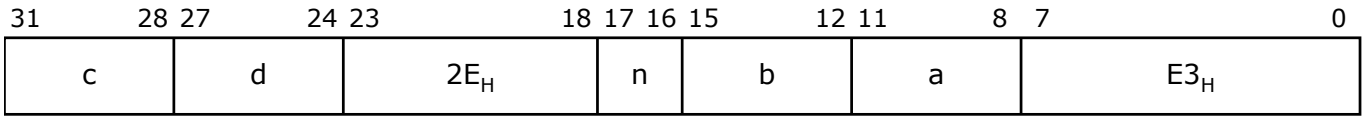
16U || 16L -||+ (16U * 16L || 16L * 16U) rounded --> 16||16 saturated




```
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] LL, n (RRR1)

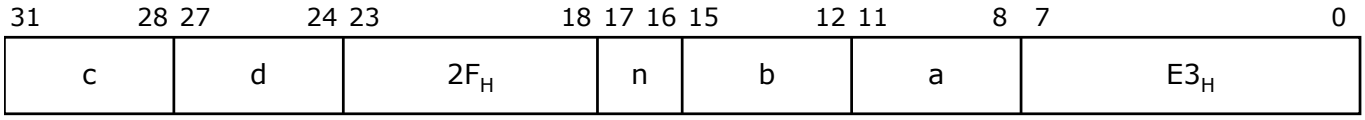
16U || 16L -||+ (16U * 16L || 16L * 16L) rounded --> 16||16 saturated




```
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] 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 this instruction.

V	ov_halfword1 = (result_halfword1 > 7FFFFFFFH) OR (result_halfword1 < -80000000H);
	ov_halfword0 = (result_halfword0 > 7FFFFFFFH) OR (result_halfword0 < -80000000H);
	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

```
msubadrs.h    d0, d2, d4, d5UL, #0
msubadrs.h    d0, d2, d4, d5LU, #1
msubadrs.h    d0, d2, d4, d5LL, #0
msubadrs.h    d0, d2, d4, d5UU, #0
```

See Also

MSUBADR.H

3.1.222 MSUBM.H

Packed Multiply-Subtract Q Format-Multi-precision

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] UL, n (RRR1)

64 - (16U * 16U) - (16L * 16L) --> 64

31 28	27 24	23 18	17 16	15 12	11 8	7 0
c	d	$1C_H$	n	b	a	$A3_H$

```
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] = result[63:0]; // Multi-precision accumulator
```

MSUBM.H E[c], E[d], D[a], D[b] LU, n (RRR1)

64 - (16U * 16L) - (16L * 16U) --> 64

31 28	27 24	23 18	17 16	15 12	11 8	7 0
c	d	$1D_H$	n	b	a	$A3_H$

```
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] = result[63:0]; // Multi-precision accumulator
```

MSUBM.H E[c], E[d], D[a], D[b] LL, n (RRR1)

64 - (16U * 16L) - (16L * 16L) --> 64

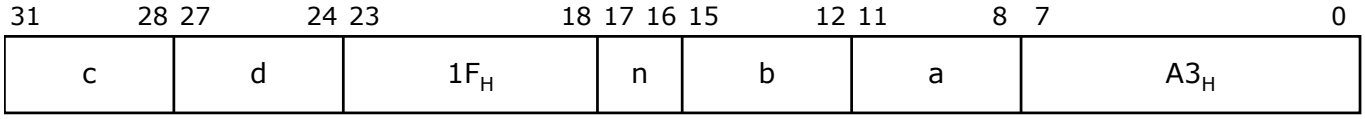
31 28	27 24	23 18	17 16	15 12	11 8	7 0
c	d	$1E_H$	n	b	a	$A3_H$

```
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] = 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] == 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] = result[63:0]; // Multi-precision accumulator
```

Status Flags

C	Not set by this instruction.
V	overflow = (result > 7FFFFFFFFFFFFFFFH) OR (result < -8000000000000000H); 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

```
msubm.h    e0, e2, d4, d5UL, #0
msubm.h    e0, e2, d4, d5LU, #1
msubm.h    e0, e2, d4, d5LL, #0
msubm.h    e0, e2, d4, d5UU, #0
```

See Also

MSUBMS.H

3.1.223 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).

MSUBMS.H E[c], E[d], D[a], D[b] UL, n (RRR1)

64 - (16U * 16U) - (16L * 16L) > 64 saturated

31 28	27 24	23 18	17 16	15 12	11 8	7 0
c	d	$3C_H$	n	b	a	$A3_H$

```
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] LU, n (RRR1)

64 - (16U * 16L) - (16L * 16U) --> saturated

31 28	27 24	23 18	17 16	15 12	11 8	7 0
c	d	$3D_H$	n	b	a	$A3_H$

```
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] LL, n (RRR1)

64 - (16U * 16L) - (16L * 16L) --> 64 saturated

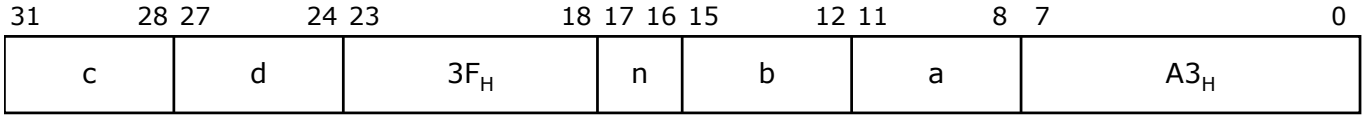
31 28	27 24	23 18	17 16	15 12	11 8	7 0
c	d	$3E_H$	n	b	a	$A3_H$

```
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
```

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);
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 this instruction.
V	overflow = (result > 7FFFFFFFFFFFFFFFH) OR (result < -8000000000000000H); 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

```
msubms.h    e0, e2, d4, d5UL, #0
msubms.h    e0, e2, d4, d5LU, #1
msubms.h    e0, e2, d4, d5LL, #0
msubms.h    e0, e2, d4, d5UU, #0
```

See Also

MSUBM.H

3.1.224 MSUBR.H

Packed Multiply-Subtract Q Format with Rounding

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], E[d], D[a], D[b] UL, n (RRR1)

32 || 32 -||- (16U * 16U || 16L * 16L) rounded > 16 || 16

31 28	27 24	23 18	17 16	15 12	11 8	7 0
c	d	$1E_H$	n	b	a	63_H

```
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] = {result_halfword1[31:16], result_halfword0[31:16]}; // Packed short fraction
```

MSUBR.H D[c], D[d], D[a], D[b] UL, n (RRR1)

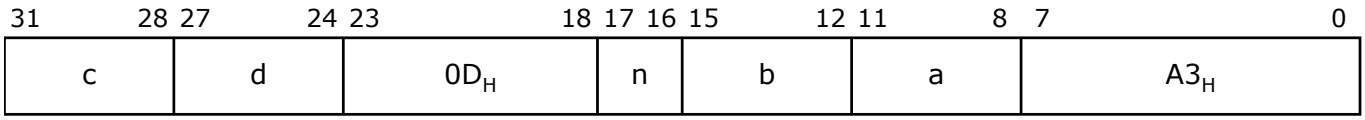
16U || 16L -||- (16U * 16U || 16L * 16L) rounded --> 16||16

31 28	27 24	23 18	17 16	15 12	11 8	7 0
c	d	$0C_H$	n	b	a	$A3_H$

```
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
```

MSUBR.H D[c], D[d], D[a], D[b] LU, n (RRR1)

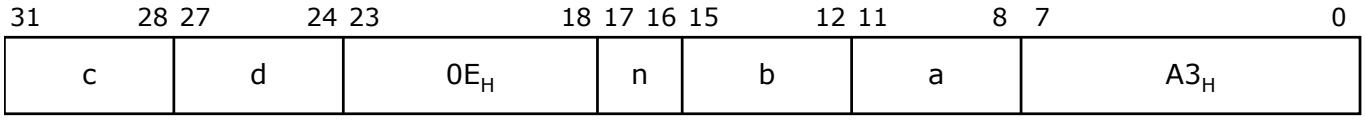
16U || 16L -||- (16U * 16L || 16L * 16U) rounded --> 16||16




```
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] = {result_halfword1[31:16], result_halfword0[31:16]}; // Packed short fraction
```

MSUBR.H D[c], D[d], D[a], D[b] LL, n (RRR1)

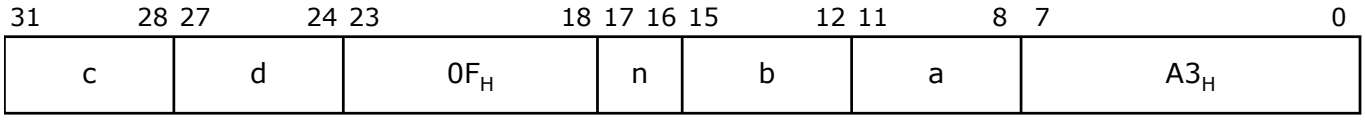
16U || 16L -||- (16U * 16L || 16L * 16L) rounded --> 16||16




```
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] = {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




```
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
```

Status Flags

C	Not set by this instruction.

V	ov_halfword1 = (result_halfword1 > 7FFFFFFFH) OR (result_halfword1 < -80000000H);
	ov_halfword0 = (result_halfword0 > 7FFFFFFFH) OR (result_halfword0 < -80000000H);
	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_overflow1 = result_halfword1[31] ^ result_halfword1[30];
	aov_overflow0 = result_halfword0[31] ^ result_halfword0[30];
	advanced_overflow = aov_overflow1 OR aov_overflow0;
	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

```
msubr.h    d0, e2, d4, d5UL, #0
msubr.h    d0, d2, d4, d5UL, #1
msubr.h    d0, d2, d4, d5LU, #0
msubr.h    d0, d2, d4, d5LL, #0
msubr.h    d0, d2, d4, d5UU, #0
```

See Also

MSUBRS.H

3.1.225 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).

MSUBRS.H D[c], E[d], D[a], D[b] UL, n (RRR1)

32||32 -||- (16U * 16U || 16L * 16L) rounded --> 16||16 saturated

31 28	27 24	23 18	17 16	15 12	11 8	7 0
c	d	$3E_H$	n	b	a	63_H

```
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] UL, n (RRR1)

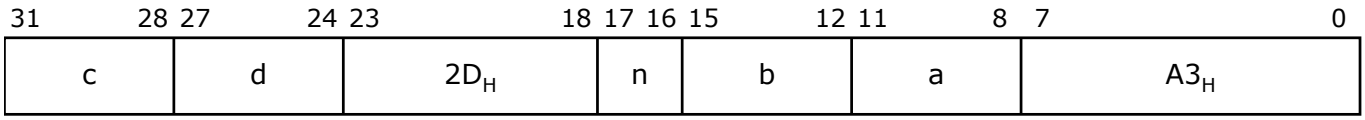
16U || 16L -||- (16U * 16U || 16L * 16L) rounded --> 16||16 saturated

31 28	27 24	23 18	17 16	15 12	11 8	7 0
c	d	$2C_H$	n	b	a	$A3_H$

```
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], D[d], D[a], D[b] LU, n (RRR1)

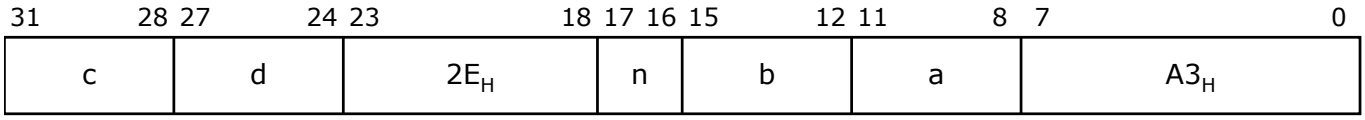
16U || 16L -||- (16U * 16L || 16L * 16U) rounded --> 16||16 saturated




```
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
```

MSUBRS.H D[c], D[d], D[a], D[b] LL, n (RRR1)

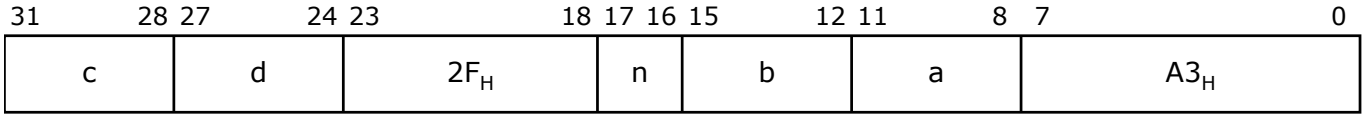
16U || 16L -||- (16U * 16L || 16L * 16L) rounded --> 16||16 saturated




```
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
```

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 this instruction.

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_overflow1 = result_halfword1[31] ^ result_halfword1[30];
	aov_overflow0 = result_halfword0[31] ^ result_halfword0[30];
	advanced_overflow = aov_overflow1 OR aov_overflow0;
	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

```
msubrs.h    d0, e2, d4, d5UL, #0
msubrs.h    d0, d2, d4, d5UL, #1
msubrs.h    d0, d2, d4, d5LU, #0
msubrs.h    d0, d2, d4, d5LL, #0
msubrs.h    d0, d2, d4, d5UU, #0
```

See Also

MSUBR.H

3.1.226 MSUBR.Q

Multiply-Subtract Q Format with Rounding

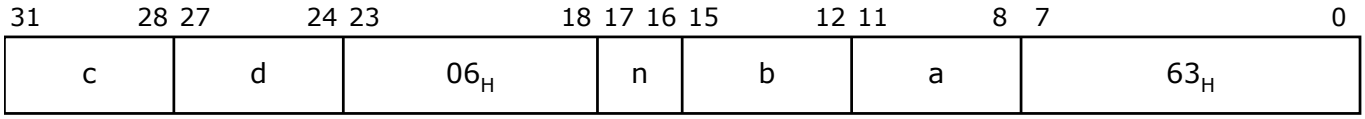
Description

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] U, D[b] U, n (RRR1)

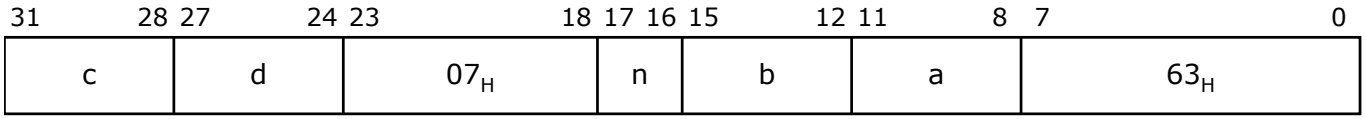
32 - (16U * 16U) rounded --> 32




```
sc = (D[a][31:16] == 8000H) AND (D[b][31:16] == 8000H) AND (n == 1);
mul_res = sc ? 7FFFFFFFH : ((D[a][31:16] * D[b][31:16]) << n);
result = D[d] - mul_res + 8000H;
D[c] = {result[31:16], 16'b0}; // Short fraction
```

MSUBR.Q D[c], D[d], D[a] L, D[b] L, n (RRR1)

32 - (16L * 16L) rounded --> 32




```
sc = (D[a][15:0] == 8000H) AND (D[b][15:0] == 8000H) AND (n == 1);
mul_res = sc ? 7FFFFFFFH : ((D[a][15:0] * D[b][15:0]) << n);
result = D[d] - mul_res + 8000H;
D[c] = {result[31:16], 16'b0}; // Short fraction
```

Status Flags

C	Not set by this instruction.
V	overflow = (result > $7FFFFFFF_H$) OR (result < -80000000_H); 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

```
msubr.q    d2, d3, d0U, d1U, #0
msubr.q    d2, d3, d0L, d1L, #1
```

See Also

MSUBRS.Q

3.1.227 MSUBRS.Q

Multiply-Subtract Q Format with Rounding, Saturated

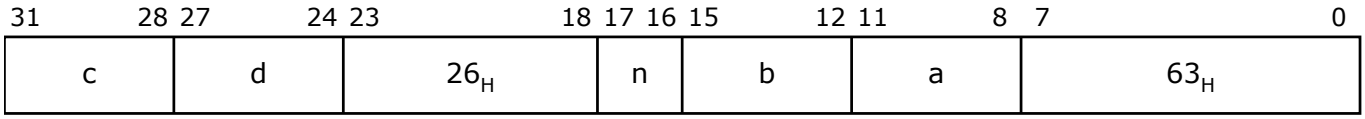
Description

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).

MSUBRS.Q D[c], D[d], D[a] U, D[b] U, n (RRR1)

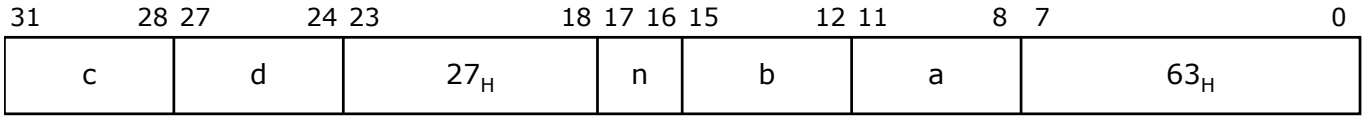
32 - (16U * 16U) rounded --> 32 saturated




```
sc = (D[a][31:16] == 8000H) AND (D[b][31:16] == 8000H) AND (n == 1);
mul_res = sc ? 7FFFFFFFH : ((D[a][31:16] * D[b][31:16]) << n);
result = D[d] - mul_res + 8000H;
D[c] = {ssov(result,32)[31:16]), 16'b0}; // Short fraction
```

MSUBRS.Q D[c], D[d], D[a] L, D[b] L, n (RRR1)

32 - (16L * 16L) rounded -->32 saturated




```
sc = (D[a][15:0] == 8000H) AND (D[b][15:0] == 8000H) AND (n == 1);
mul_res = sc ? 7FFFFFFFH : ((D[a][15:0] * D[b][15:0]) << n);
result = D[d] - mul_res + 8000H;
D[c] = {ssov(result,32)[31:16]), 16'b0}; // Short fraction
```

Status Flags

C	Not set by this instruction.
V	overflow = (result > $7FFFFFFF_H$) OR (result < -80000000_H); 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

```
msubrs.q    d2, d3, d0U, d1U, #0
msubrs.q    d2, d3, d0L, d1L, #1
```

See Also

MSUBR.Q

3.1.228 MTCR

Move To Core Register

Description

***Note**: Execution Mode - This instruction can only be executed in the following mode: Supervisor.*

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. The MTCR instruction cannot be used to access the GPRs of the executing environment.

If the virtualization extension is not present or disabled, or when executing in a Guest VM

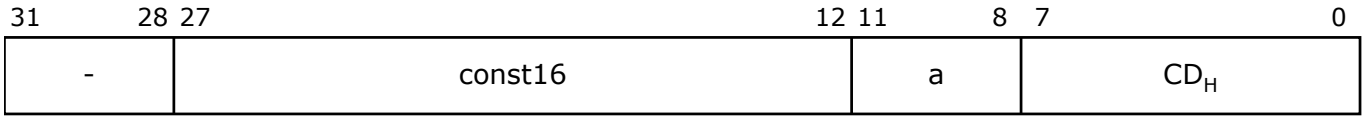
- The CSFR offset must have the least significant two bits as zero otherwise an OPD trap will result.

If the virtualization extension is present and enabled, and executing in VM0

- Using the least significant offset bits of 00_B the currently executing environment (HRHV) CSFRs are addressed.
- Using the least significant offset bits of 01_B the HRA CSFRs are addressed.
- Using the least significant offset bits of 10_B the HRB CSFRs are addressed.
- Using the least significant offset bits of 11_B will result in an OPD trap.

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)




```
CR[const16] = D[a];
```

Status Flags

C	if (const16 == $FE04_H$) then PSW.C = D[a][31];
V	if (const16 == $FE04_H$) then PSW.V = D[a][30];
SV	if (const16 == $FE04_H$) then PSW.SV = D[a][29];
AV	if (const16 == $FE04_H$) then PSW.AV = D[a][28];
SAV	if (const16 == $FE04_H$) then PSW.SAV = D[a][27];

Examples

```
mtcr    0xFE04, d1
```

See Also

MFCR, MFDCR, MTDCR, RSTV

3.1.229 MTDCR

Move to Core Register Pair

Description

***Note**: Execution Mode - This instruction can only be executed in the following mode: Supervisor.*

Move the value in data register pair E[a] to the pair of Core Special Function Registers (CSFR) selected by the value const16 and const16+4. The CSFR pair address is a const16 byte offset from the CSFR base address. The MTDCR instruction cannot be used to access the GPRs of the executing environment.

If the virtualization extension is not present or disabled, or when executing in a Guest VM

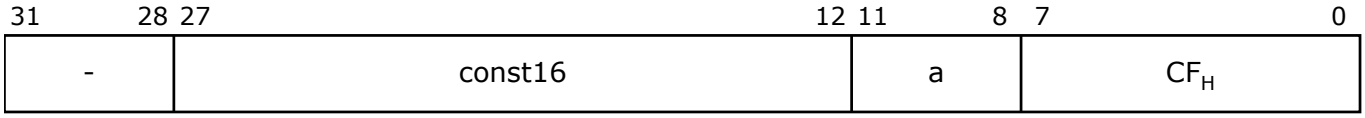
- The CSFR offset must have the least significant three bits as zero otherwise an OPD trap will result.

If the virtualization extension is present and enabled, and executing in VM0

- Using the least significant offset bits of 00_B the currently executing environment (HRHV) CSFRs are addressed.
- Using the least significant offset bits of 01_B the HRA CSFRs are addressed.
- Using the least significant offset bits of 10_B the HRB CSFRs are addressed.
- Using the least significant offset bits of 11_B will result in an OPD trap.
- Using the least significant offset bits of $1xx_B$ will result in an OPD trap.

An MTDCR instruction should be followed by an ISYNC instruction. This ensures that all instructions following the MTDCR see the effects of the CSFR update.

MTDCR const16, E[a] (RLC)




```
CR[const16] = D[a];
CR[const16+4] = D[a+1];
```

Status Flags

C	if ((const16+4) == $FE04_H$) then PSW.C = D[a+1][31];
V	if ((const16+4) == $FE04_H$) then PSW.V = D[a+1][30];
SV	if ((const16+4) == $FE04_H$) then PSW.SV = D[a+1][29];
AV	if ((const16+4) == $FE04_H$) then PSW.AV = D[a+1][28];
SAV	if ((const16+4) == $FE04_H$) then PSW.SAV = D[a+1][27];

Examples

```
mtdcr    0xC000, E0
```

See Also

MTCR, MFCR, MFDCR

3.1.230 MUL

Multiply

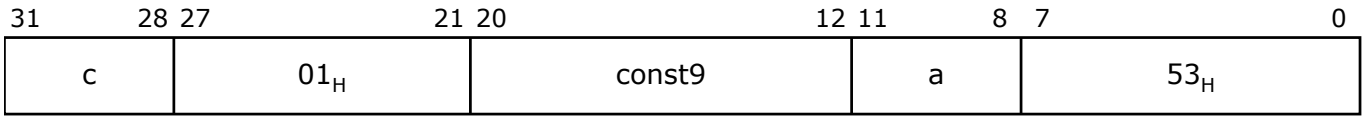
Description

Multiply two 32-bit signed integers and put the product into a 32-bit or 64-bit register. The value const9 is sign-extended before the multiplication is performed.

Multiply D[a] by D[b] (two 32-bit signed integers) and put the product into D[a].

MUL D[c], D[a], const9 (RC)

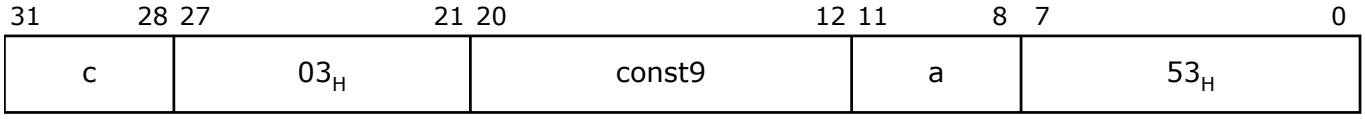
(32 * K9) --> 32 signed




```
result = D[a] * sign_ext(const9);
D[c] = result[31:0];
```

MUL E[c], D[a], const9 (RC)

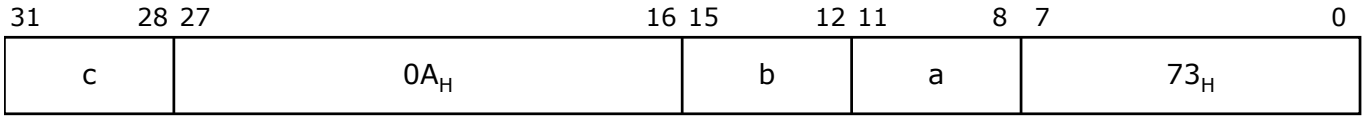
(32 * K9) --> 64 signed




```
result = D[a] * sign_ext(const9);
E[c] = result[63:0];
```

MUL D[c], D[a], D[b] (RR2)

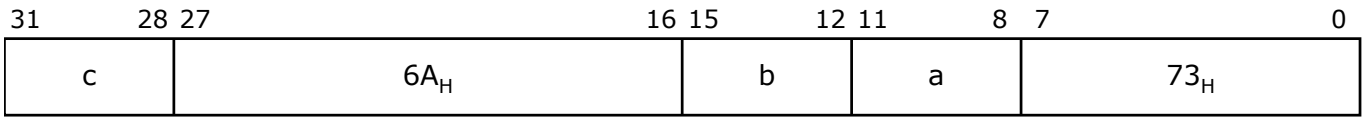
(32 * 32) --> 32 signed




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

MUL E[c], D[a], D[b] (RR2)

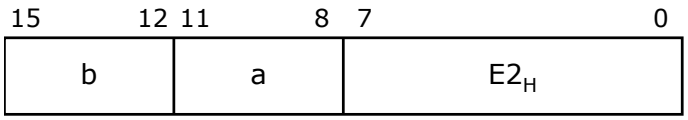
(32 * 32) --> 64 signed




```
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];
```

Status Flags

C	Not set by this instruction.

V	32-bit result:
	overflow = (result > 7FFFFFFF$_H$) OR (result < -80000000$_H$);
	if (overflow) then PSW.V = 1 else PSW.V = 0;
	64-bit result:
	overflow = 0;
	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    d2, d4, #0x21
mul    e2, d4, #0x12
mul    d3, d1, d2
mul    e2, d5, d1
mul    d3, d11
```

See Also

MUL.U, MULS, MULS.U, MADD, MSUB

3.1.231 MULS

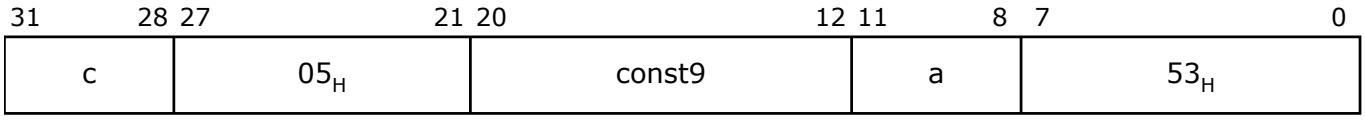
Multiply, Saturated

Description

Multiply two 32-bit signed integers and put the product into a 32-bit register. The value const9 is sign-extended before the multiplication is performed. The result is saturated on overflow.

MULS D[c], D[a], const9 (RC)

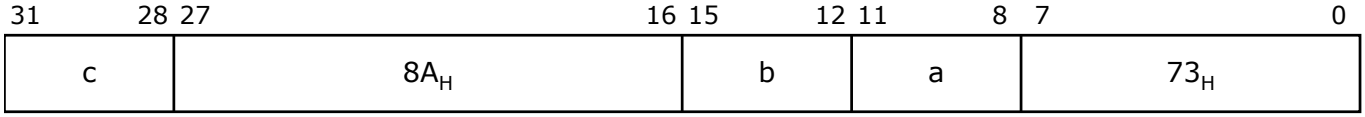
(32 * K9) --> 32 signed saturated




```
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 this instruction.
V	overflow = (result > 7FFFFFFF$_H$) OR (result < -80000000$_H$); 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

```
muls    d2, d0, d0
```

See Also

MUL, MUL.U, MULS.U, MADD, MSUB

3.1.232 MUL.H

Packed Multiply Q Format

Description

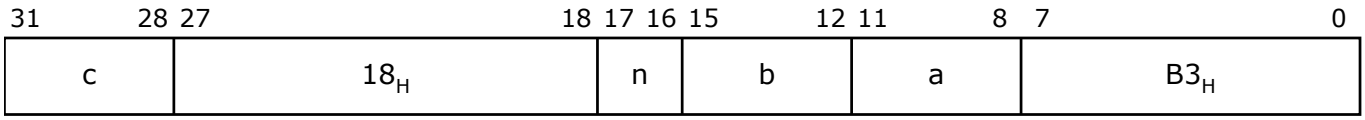
Multiply two signed 16-bit (half-word) values and put the product (left justified if n == 1) into a 32-bit register. Note that since there are two results both halves of an extended data register are used. There are four cases of half-word multiplication:

- 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 = $7FFFFFFF_H$.

MUL.H E[c], D[a], D[b] UL, n (RR1)

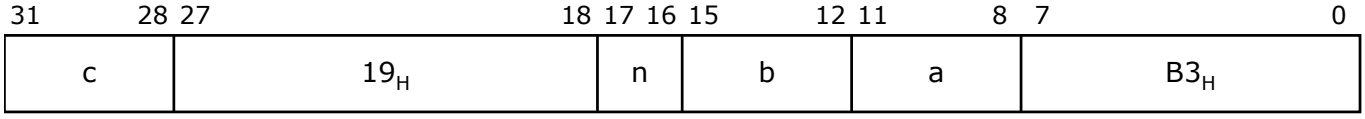
(16U * 16U || 16L * 16L) --> 32||32




```
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);
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);
E[c] = {result_word1[31:0], result_word0[31:0]}; // Packed fraction
```

MUL.H E[c], D[a], D[b] LU, n (RR1)

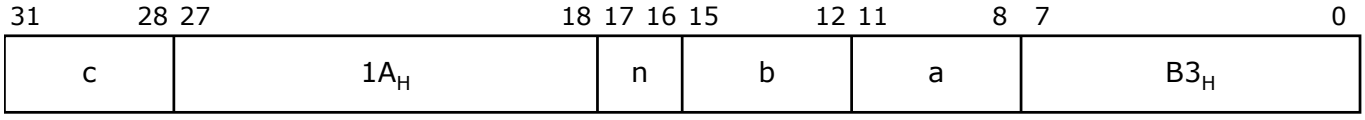
(16U * 16L || 16L * 16U) --> 32||32




```
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);
E[c] = {result_word1[31:0], result_word0[31:0]}; // Packed fraction
```

MUL.H E[c], D[a], D[b] LL, n (RR1)

(16U * 16L || 16L * 16L) --> 32||32

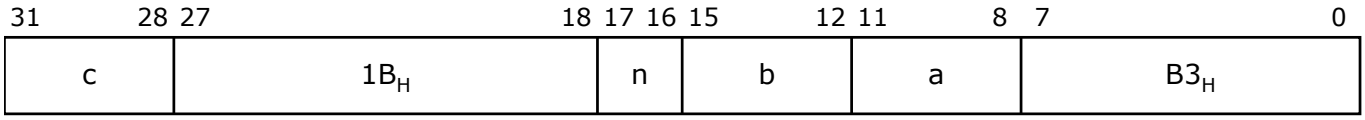



```
sc1 = (D[a][31:16] == 8000H) AND (D[b][15:0] == 8000H) AND (n == 1);
sc0 = (D[a][15:0] == 8000H) AND (D[b][15:0] == 8000H) AND (n == 1);
result_word1 = sc1 ? 7FFFFFFFH : ((D[a][31:16] * D[b][15:0]) << n);
```

```
result_word0 = sc0 ? 7FFFFFFF_H : ((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_H) AND (D[b][31:16] == 8000_H) AND (n == 1);
sc0 = (D[a][31:16] == 8000_H) AND (D[b][31:16] == 8000_H) AND (n == 1);
result_word1 = sc1 ? 7FFFFFFF_H : ((D[a][15:0] * D[b][31:16]) << n);
result_word0 = sc0 ? 7FFFFFFF_H : ((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

```
mul.h    e0, d4, d5UL, #0
mul.h    e0, d4, d5LU, #1
mul.h    e0, d4, d5LL, #0
mul.h    e0, d4, d5UU, #0
```

See Also

-

3.1.233 MUL.Q

Multiply Q Format

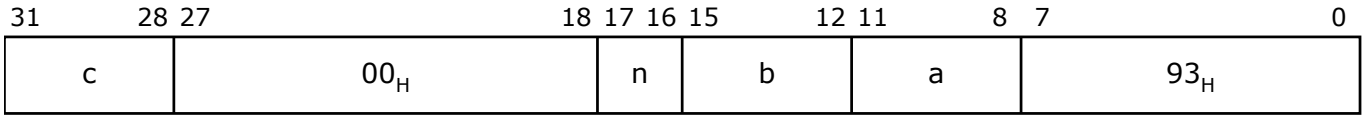
Description

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] U, n (RR1)

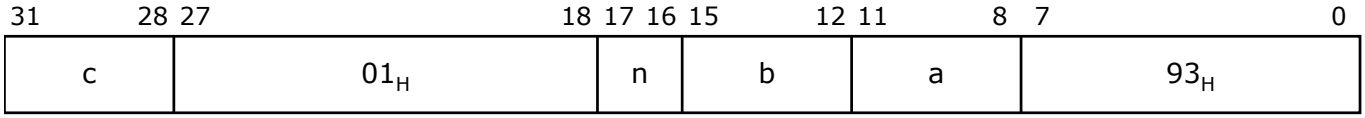
(32 * 16U)Up --> 32




```
result = ((D[a] * D[b][31:16]) << n) >> 16;
D[c] = result[31:0];// Fraction
```

MUL.Q D[c], D[a], D[b] L, n (RR1)

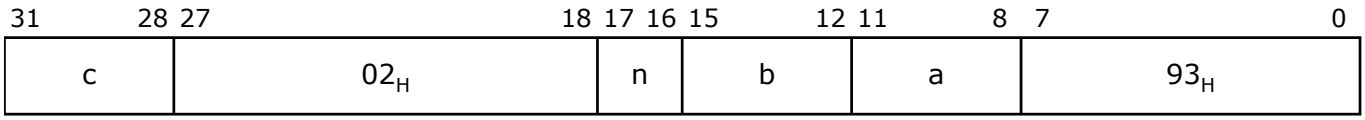
(32 * 16L)Up --> 32




```
result = ((D[a] * D[b][15:0]) << n) >> 16;
D[c] = result[31:0]; // Fraction
```

MUL.Q D[c], D[a], D[b], n (RR1)

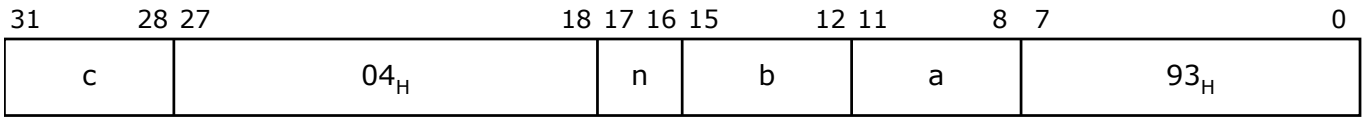
(32 * 32)Up --> 32




```
result = ((D[a] * D[b]) << n) >> 32;
D[c] = result[31:0]; // Fraction
```

MUL.Q D[c], D[a] U, D[b] U, n (RR1)

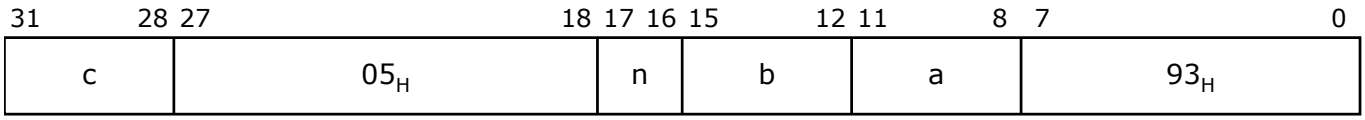
(16U * 16U) --> 32




```
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
```

MUL.Q D[c], D[a] L, D[b] L, n (RR1)

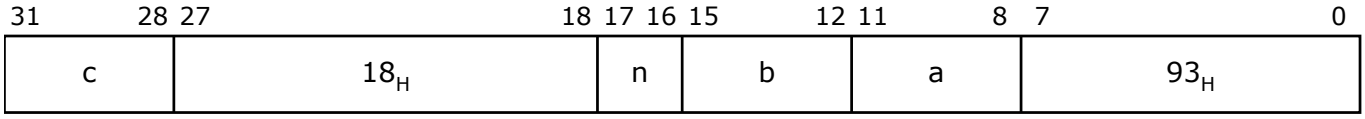
(16L * 16L) --> 32




```
sc = (D[a][15:0] == 8000H) AND (D[b][15:0] == 8000H) AND (n == 1);
result = sc ? 7FFFFFFFH : ((D[a][15:0] * D[b][15:0]) << n);
D[c] = result[31:0]; // Fraction
```

MUL.Q E[c], D[a], D[b] U, n (RR1)

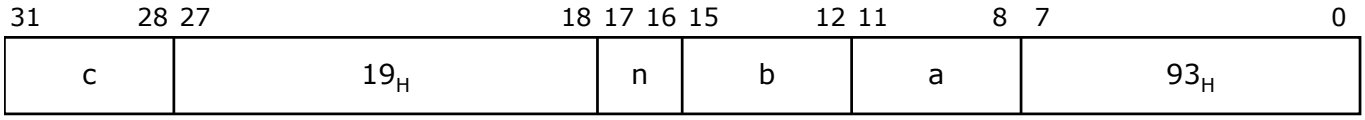
(32 * 16U) --> 64




```
result = (D[a] * D[b][31:16]) << n;
E[c] = result[63:0]; // Multi-precision accumulator
```

MUL.Q E[c], D[a], D[b] L, n (RR1)

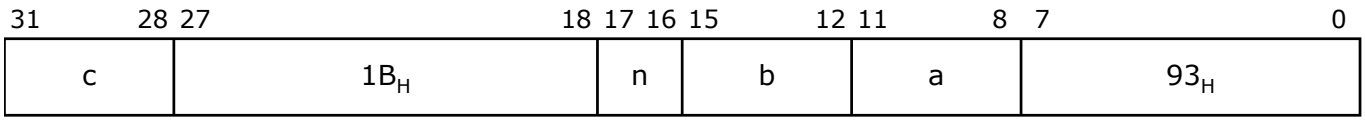
(32 * 16L) --> 64




```
result = (D[a] * D[b][15:0]) << n;
E[c] = result[63:0]; // Multi-precision accumulator
```

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
```

Status Flags

C	Not set by this instruction.

V	32-bit result:
	overflow = (result > 7FFFFFFFH) OR (result < -80000000H);
	if (overflow) then PSW.V = 1 else PSW.V = 0;
	64-bit result:
	overflow = (result > 7FFFFFFFFFFFFFFFH) OR (result < -8000000000000000H);
	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, d1, d2U, #0
mul.q    d3, d1, d2L, #1
mul.q    d2, d1, d2, #1
mul.q    d3, d1U, d2U, #1
mul.q    d3, d1L, d2L, #1
mul.q    e2, d1, d0U, #1
mul.q    e2, d1, d0L, #1
mul.q    e2, d1, d7, #1
```

See Also

-

3.1.234 MUL.U

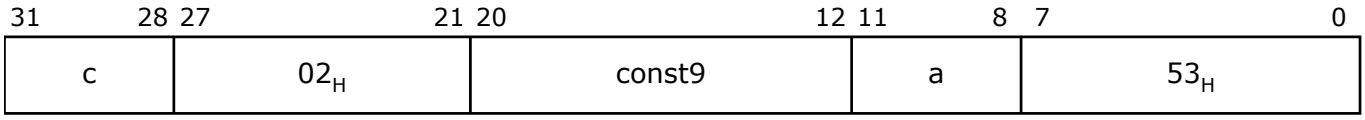
Multiply Unsigned

Description

Multiply two 32-bit unsigned integers and put the product into a 64-bit register. The value const9 (instruction format RC) is zero-extended before the multiplication is performed.

MUL.U E[c], D[a], const9 (RC)

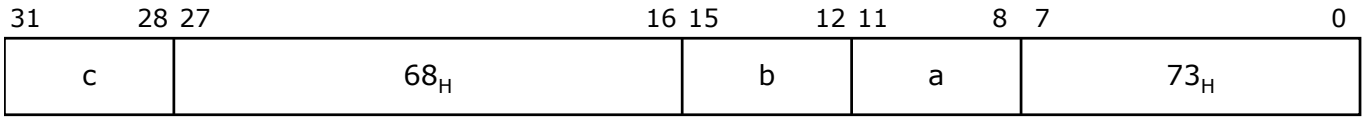
(32 * K9) --> 64 unsigned




```
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];
```

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[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, #7
mul.u    e0, d2, d3
```

See Also

MUL, MULS, MULS.U

3.1.235 MULS.U

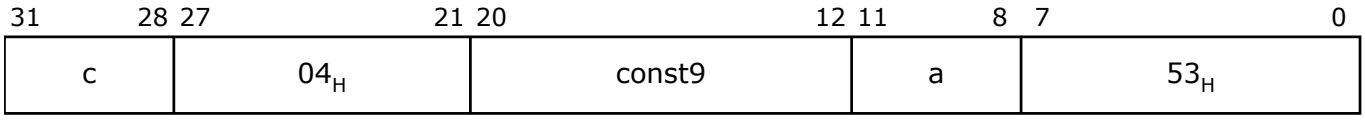
Multiply Unsigned, Saturated

Description

Multiply two 32-bit unsigned integers and put the product into a 32-bit register. The value const9 (instruction format RC) is zero-extended before the multiplication is performed. The result is saturated on overflow.

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

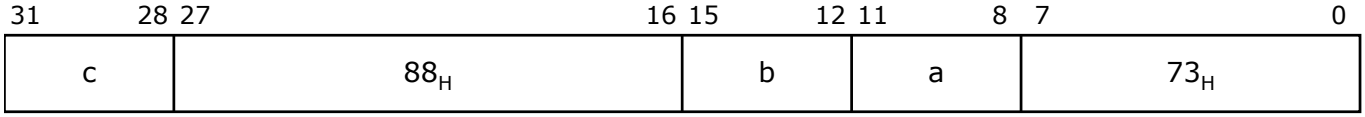
(32 * K9) --> 32 unsigned saturated




```
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




```
result = D[a] * D[b]; // unsigned
D[c] = suov(result, 32);
```

Status Flags

C	Not set by this instruction.
V	overflow = (result > FFFFFFFF$_H$) OR (result < 00000000$_H$); 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

```
muls.u    d3, d2, #7
muls.u    d3, d5, d9
```

See Also

MUL, MULS, MUL.U

3.1.236 MULM.H

Packed Multiply Q Format-Multi-precision

Description

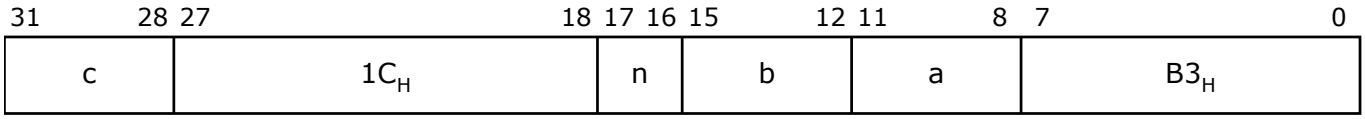
Perform two multiplications of two signed 16-bit (half-word) values. Add the two products (left justified if n == 1) left-shifted by 16, in a 64-bit register. There are four cases of half-word multiplication:

- 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 = $7FFFFFFF_H$.

MULM.H E[c], D[a], D[b] UL, n (RR1)

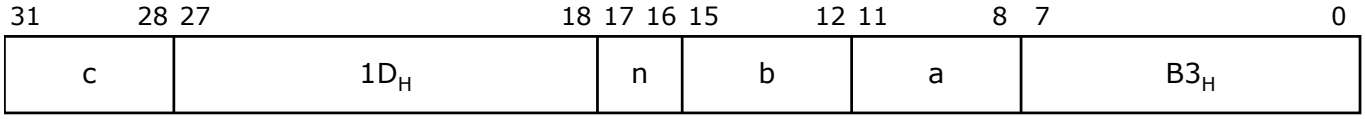
16U * 16U + 16L * 16L --> 64




```
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 = (result_word1 + result_word0) << 16;
E[c] = result[63:0]; // Multi-precision accumulator
```

MULM.H E[c], D[a], D[b] LU, n (RR1)

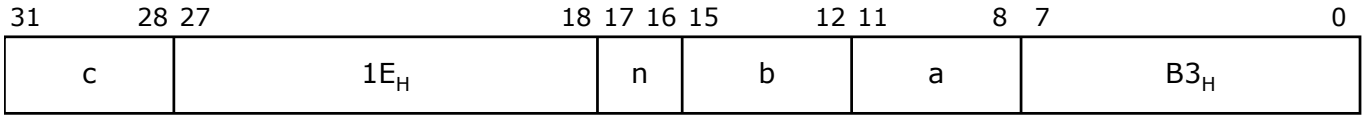
16U * 16L + 16L * 16U --> 64




```
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 = (result_word1 + result_word0) << 16;
E[c] = result[63:0]; // Multi-precision accumulator
```

MULM.H E[c], D[a], D[b] LL, n (RR1)

16U * 16L + 16L * 16L --> 64




```
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 ? 7FFFFFFF_H : ((D[a][31:16] * D[b][15:0]) << n);
result_word0 = sc0 ? 7FFFFFFF_H : ((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

31 28	27 18	17 16	15 12	11 8	7 0
c	$1F_H$	n	b	a	$B3_H$

```
sc1 = (D[a][15:0] == 8000_H) AND (D[b][31:16] == 8000_H) AND (n == 1);
sc0 = (D[a][31:16] == 8000_H) AND (D[b][31:16] == 8000_H) AND (n == 1);
result_word1 = sc1 ? 7FFFFFFF_H : ((D[a][15:0] * D[b][31:16]) << n);
result_word0 = sc0 ? 7FFFFFFF_H : ((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

```
mulm.h    e0, d4, d5UL, #0
mulm.h    e0, d4, d5LU, #1
mulm.h    e0, d4, d5LL, #0
mulm.h    e0, d4, d5UU, #0
```

See Also

-

3.1.237 MULP.B

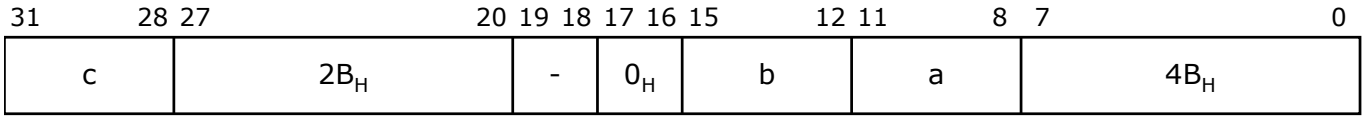
Packed carry-less multiplication

Description

Performs byte wide carry-less multiplication on four byte lanes in parallel. The four input byte are treated as unsigned operands. The four 16-bit results are packed in to a 64-bit result registers E[c].

***Note**: Carry-less multiplication returns the result of the multiplication of operands 'a' and 'b' without generating or propagating carry.*

MULP.B E[c], D[a], D[b] (RR)




```
res0 = D[a][7:0] ⊕ D[b][7:0];
res1 = D[a][15:8] ⊕ D[b][15:8];
res2 = D[a][23:16] ⊕ D[b][23:16];
res3 = D[a][31:24] ⊕ D[b][31:24];
E[c] = {res3[15:0], res2[15:0], res1[15:0], res0[15:0]}; // Packed carry-less multiplication
```

***Note**: Where ⊕ denotes carry-less mulitply.*

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

```
mulp.b    e2, d0, d1
```

See Also

CRCN

3.1.238 MULR.H

Packed Multiply Q Format with Rounding

Description

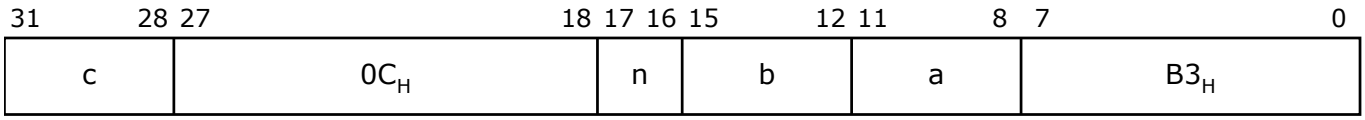
Multiply two signed 16-bit (half-word) values. Add the product (left justified if n == 1) to a signed 16-bit value and put the rounded result into half of a 32-bit register. Note that since there are two results the two register halves are used). There are four cases of half-word multiplication:

- 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 = $7FFFFFFF_H$.

MULR.H D[c], D[a], D[b] UL, n (RR1)

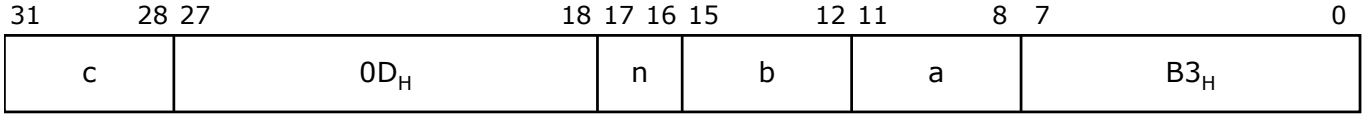
(16U * 16U || 16L * 16L) rounded --> 16||16




```
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);
result_halfword1 = sc1 ? 7FFFFFFFH : (((D[a][31:16] * D[b][31:16]) << n) + 8000H);
result_halfword0 = sc0 ? 7FFFFFFFH : (((D[a][15:0] * D[b][15:0]) << n) + 8000H);
D[c] = {result_halfword1[31:16], result_halfword0[31:16]}; // Packed short fraction
```

MULR.H D[c], D[a], D[b] LU, n (RR1)

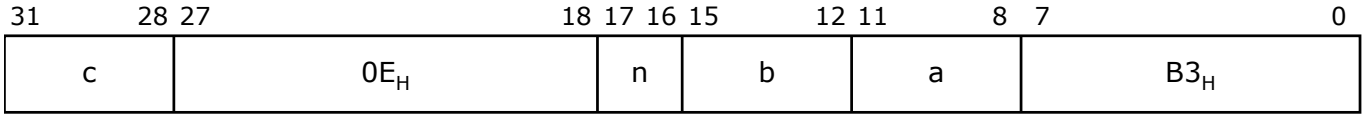
(16U * 16L || 16L * 16U) rounded --> 16||16




```
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_halfword1 = sc1 ? 7FFFFFFFH : (((D[a][31:16] * D[b][15:0]) << n) + 8000H);
result_halfword0 = sc0 ? 7FFFFFFFH : (((D[a][15:0] * D[b][31:16]) << n) + 8000H);
D[c] = {result_halfword1[31:16], result_halfword0[31:16]}; // Packed short fraction
```

MULR.H D[c], D[a], D[b] LL, n (RR1)

(16U * 16L || 16L * 16L) rounded --> 16||16

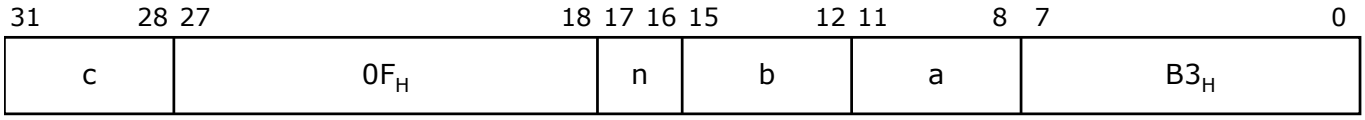



```
sc1 = (D[a][31:16] == 8000H) AND (D[b][15:0] == 8000H) AND (n == 1);
sc0 = (D[a][15:0] == 8000H) AND (D[b][15:0] == 8000H) AND (n == 1);
result_halfword1 = sc1 ? 7FFFFFFFH : (((D[a][31:16] * D[b][15:0]) << n) + 8000H);
```

```
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]}; // 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_H) AND (D[b][31:16] == 8000_H) AND (n == 1);
sc0 = (D[a][31:16] == 8000_H) AND (D[b][31:16] == 8000_H) AND (n == 1);
result_halfword1 = sc1 ? 7FFFFFFF_H : (((D[a][15:0] * D[b][31:16]) << n) + 8000_H);
result_halfword0 = sc0 ? 7FFFFFFF_H : (((D[a][31:16] * D[b][31:16]) << n) + 8000_H);
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

```
mulr.h    d0, d2, d4, d5UL, #1
mulr.h    d0, d2, d4, d5LU, #0
mulr.h    d0, d2, d4, d5LL, #0
mulr.h    d0, d2, d4, d5UU, #0
```

See Also

-

3.1.239 MULR.Q

Multiply Q Format with Rounding

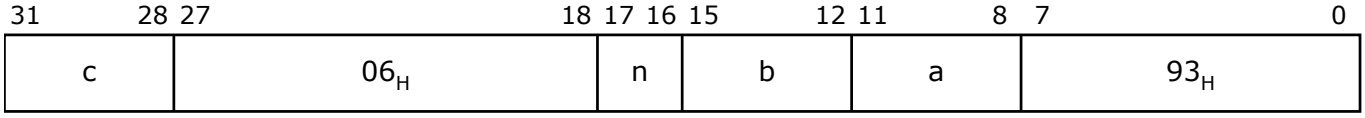
Description

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 = $7FFFFFFF_H$.

MULR.Q D[c], D[a] U, D[b] U, n (RR1)

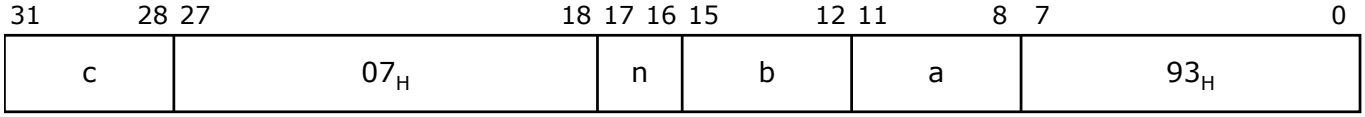
(16U * 16U) rounded --> 32




```
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) + 8000H);
D[c] = {result[31:16], 16'b0}; // Short fraction
```

MULR.Q D[c], D[a] L, D[b] L, n (RR1)

(16L * 16L) rounded --> 32




```
sc = (D[a][15:0] == 8000H) AND (D[b][15:0] == 8000H) AND (n == 1);
result = sc ? 7FFFFFFFH : (((D[a][15:0] * D[b][15:0]) << n) + 8000H);
D[c] = {result[31:16], 16'b0}; // 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] ^ 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

```
mulr.q    d0, d2U, d3U, #0
mulr.q    d0, d2L, d3L, #1
```

See Also

-

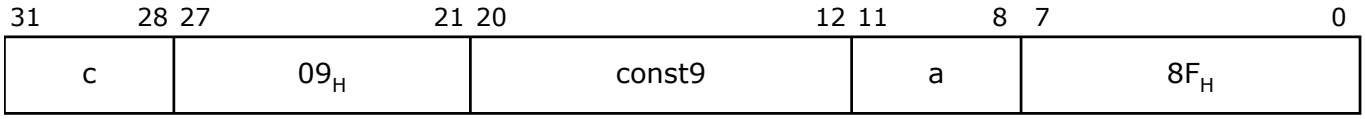
3.1.240 NAND

Bitwise NAND

Description

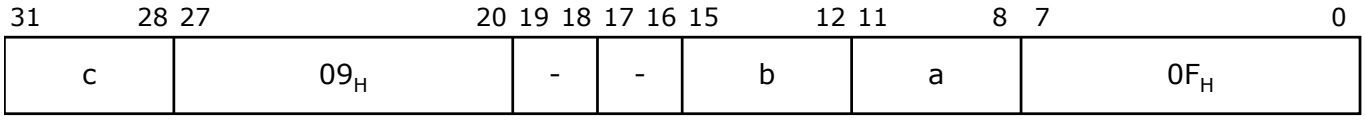
Compute the bitwise NAND of the contents of data register D[a] and either data register D[b] (instruction format RR) or const9 (instruction format RC). Put the result in data register D[c]. The const9 value is zero-extended.

NAND D[c], D[a], const9 (RC)




```
D[c] = ~(D[a] & zero_ext(const9));
```

NAND D[c], D[a], D[b] (RR)




```
D[c] = ~(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, #126
nand    d3, d1, d2
```

See Also

AND, ANDN, NOR, NOT, OR, ORN, XNOR, XOR

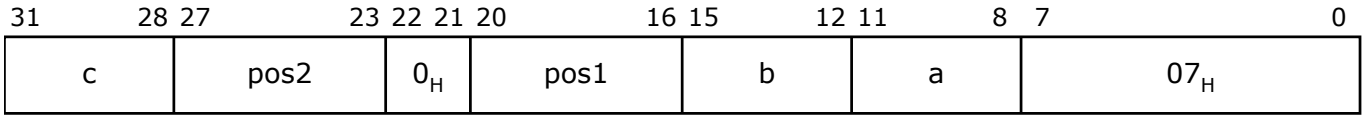
3.1.241 NAND.T

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)




```
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

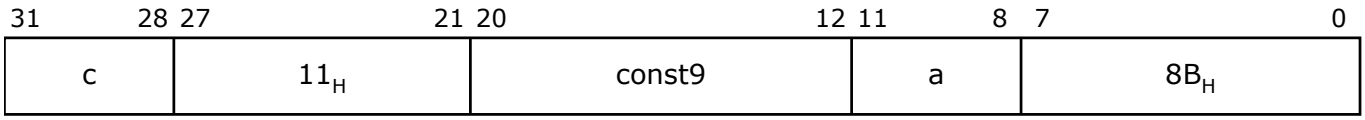
3.1.242 NE

Not Equal

Description

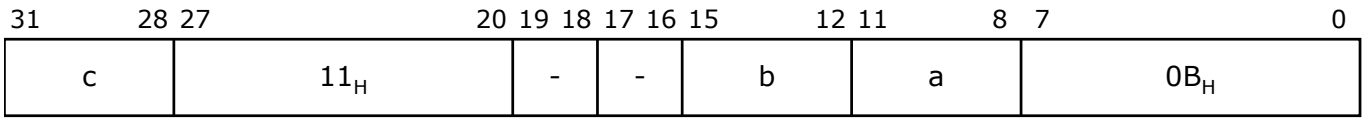
If the contents of data register D[a] are not equal to the contents of either data register D[b] (instruction format RR) or const9 (instruction format RC), set the least-significant bit of D[c] to one and clear the remaining bits to zero; otherwise clear all bits in D[c]. The const9 value is sign-extended.

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

See Also

EQ, GE, GE.U, LT, LT.U, EQANY.B, EQANY.H, NEZ.A

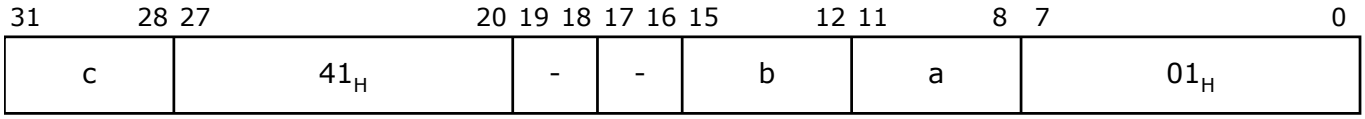
3.1.243 NE.A

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)




```
result = (A[a] != A[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.a    d3, a4, a2
```

See Also

EQ.A, EQZ.A, GE.A, LT.A, NEZ.A

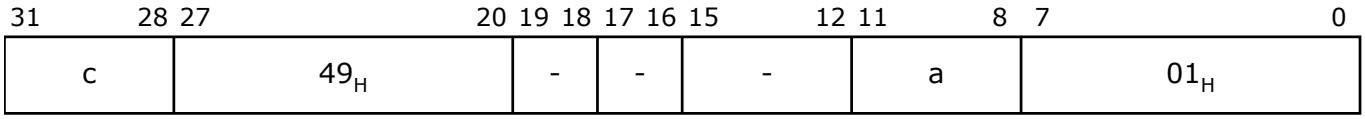
3.1.244 NEZ.A

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)




```
result = (A[a] != 0);
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

```
nez.a    d3, a4
```

See Also

EQ.A, EQZ.A, GE.A, LT.A, NE

3.1.245 NOP

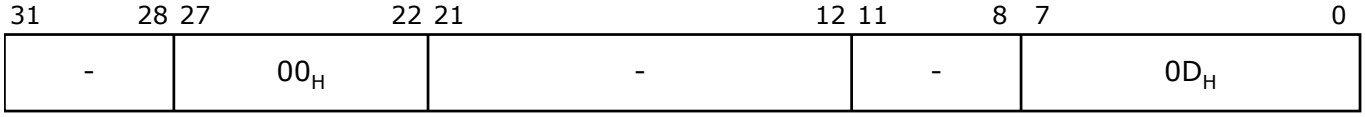
No Operation

Description

Used to implement efficient low-power, non-operational instructions.

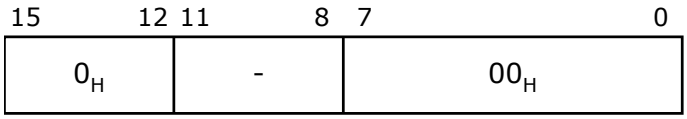
Used to implement efficient low-power, non-operational instructions.

NOP (SYS)




```
No operation.
```

NOP (SR)




```
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

-

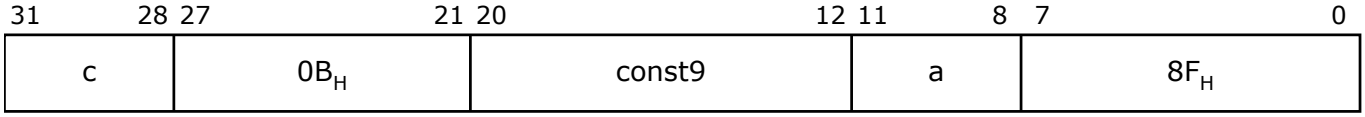
3.1.246 NOR

Bitwise NOR

Description

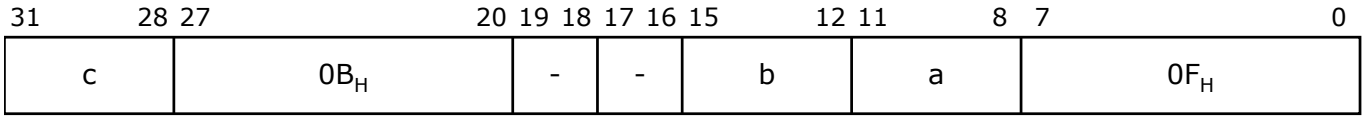
Compute the bitwise NOR of the contents of data register D[a] and the contents of either data register D[b] (instruction format RR) or const9 (instruction format RC) and put the result in data register D[c]. The const9 value is zero-extended.

NOR D[c], D[a], const9 (RC)




```
D[c] = ~(D[a] | zero_ext(const9));
```

NOR D[c], D[a], D[b] (RR)




```
D[c] = ~(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

```
nor    d3, d1, #126
nor    d3, d1, d2
```

See Also

AND, ANDN, NAND, NOT, OR, ORN, XNOR, XOR

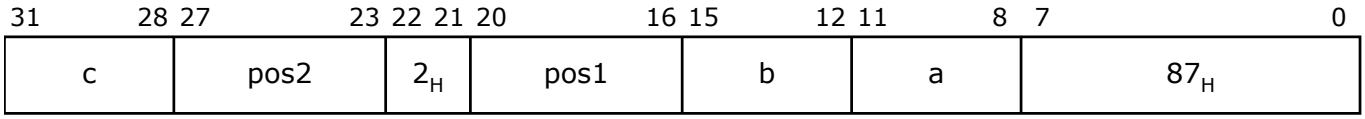
3.1.247 NOR.T

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

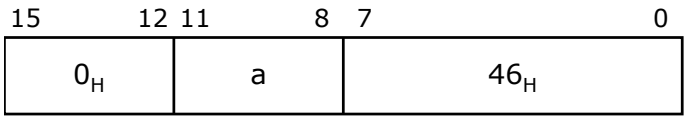
3.1.248 NOT

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].

NOT D[a] (SR)




```
D[a] = ~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

3.1.249 OR

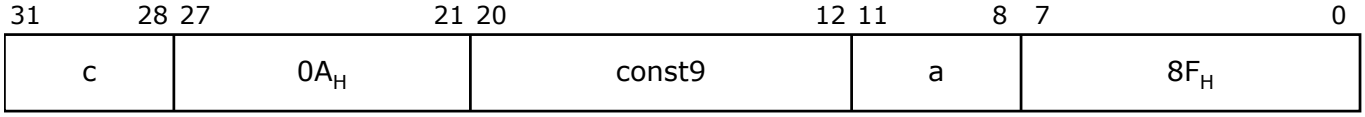
Bitwise OR

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.

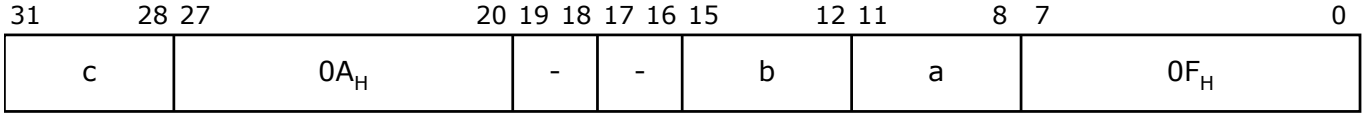
Compute the bitwise OR of the contents of either data register D[a] (instruction format SRR) or D[15] (instruction format SC) and the contents of either data register D[b] (format SRR) or const8 (format SC). Put the result in either data register D[a] (format SRR) or D[15] (format SC). The const8 value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

OR D[c], D[a], const9 (RC)



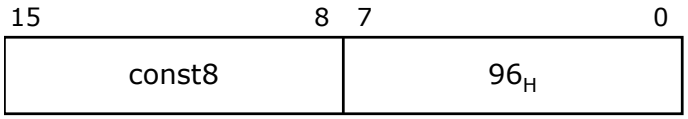

```
D[c] = D[a] | zero_ext(const9);
```

OR D[c], D[a], D[b] (RR)



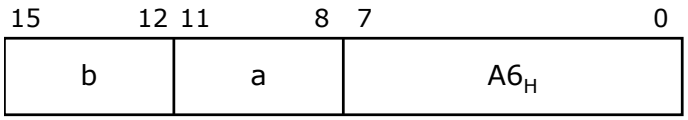

```
D[c] = D[a] | D[b];
```

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, #126
or    d3, d1, d2
```

```
or    d15, #126
or    d1, d2
```

See Also

AND, ANDN, NAND, NOR, NOT, ORN, XNOR, XOR

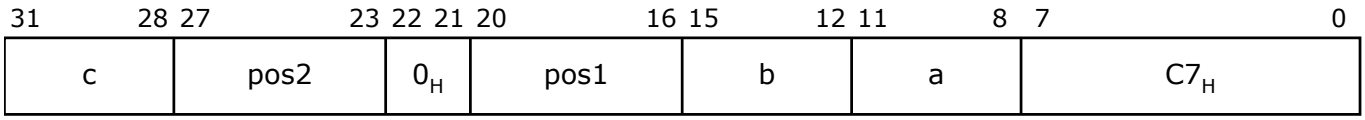
3.1.250 OR.AND.T

Accumulating Bit Logical OR-AND

Description

Compute the logical operation AND 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] OR (D[a][pos1] AND D[b][pos2])};
```

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.and.t    d3, d1, #3, d2, #5
```

See Also

AND.AND.T, AND.ANDN.T, AND.NOR.T, AND.OR.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

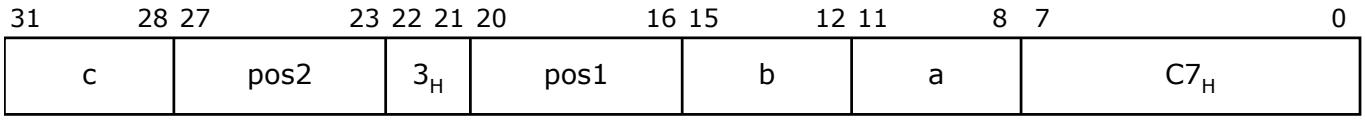
3.1.251 OR.ANDN.T

Accumulating Bit Logical OR-AND-Not

Description

Compute the logical operation ANDN 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.ANDN.T D[c], D[a], pos1, D[b], pos2 (BIT)




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

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.andn.t    d3, d1, #3, d2, #5
```

See Also

AND.AND.T, AND.ANDN.T, AND.NOR.T, AND.OR.T, OR.AND.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

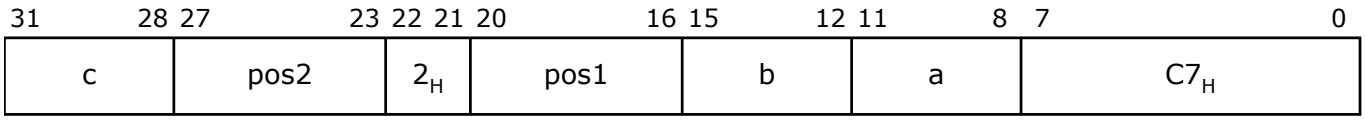
3.1.252 OR.NOR.T

Accumulating Bit Logical OR-NOR

Description

Compute the logical operation NOR 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.NOR.T D[c], D[a], pos1, D[b], pos2 (BIT)




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

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.nor.t    d3, d1, #3, d2, #5
```

See Also

AND.AND.T, AND.ANDN.T, AND.NOR.T, AND.OR.T, OR.AND.T, OR.ANDN.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

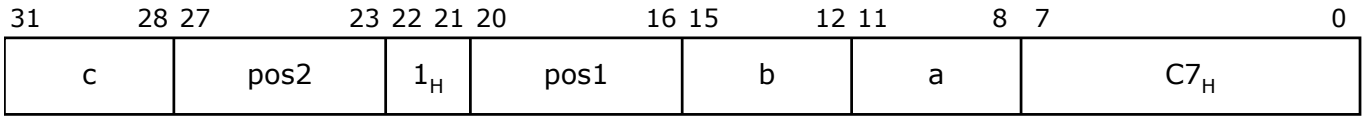
3.1.253 OR.OR.T

Accumulating Bit Logical OR-OR

Description

Compute the logical operation OR 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.OR.T D[c], D[a], pos1, D[b], pos2 (BIT)




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

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.or.t     d3, d1, 3, d2, 5
```

See Also

AND.AND.T, AND.ANDN.T, AND.NOR.T, AND.OR.T, OR.AND.T, OR.ANDN.T, OR.NOR.T, SH.AND.T, SH.ANDN.T, SH.NAND.T, SH.NOR.T, SH.OR.T, SH.ORN.T, SH.XNOR.T

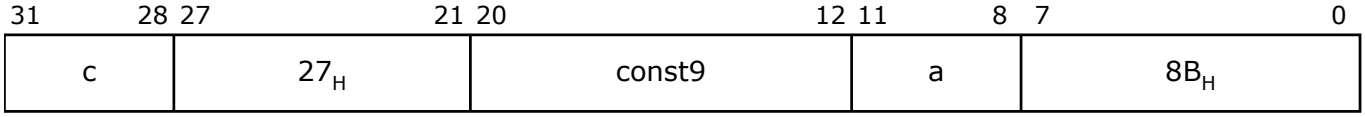
3.1.254 OR.EQ

Equal, OR Accumulating

Description

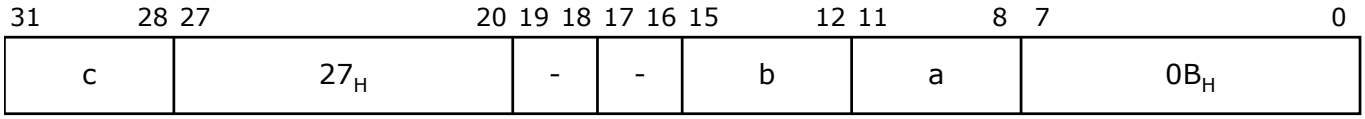
Compute the logical OR of D[c][0] and the Boolean result of the EQ operation on the contents of data register D[a] and either data register D[b] (instruction format RR) or const9 (instruction format RC). Put the result in D[c][0]. All other bits in D[c] are unchanged. The const9 value is sign-extended.

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




```
D[c] = {D[c][31:1], D[c][0] OR (D[a] == sign_ext(const9))};
```

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




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

See Also

AND.EQ, XOR.EQ

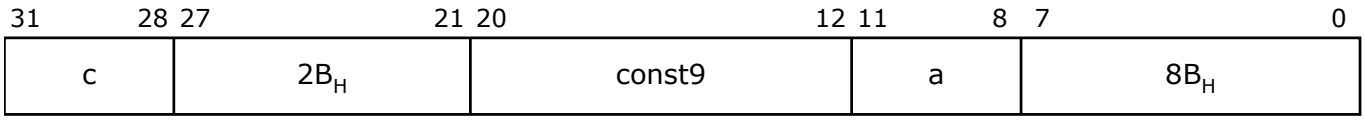
3.1.255 OR.GE

Greater Than or Equal, OR Accumulating

Description

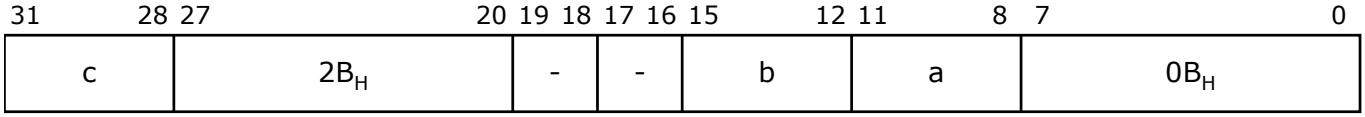
Calculate the logical OR of D[c][0] and the Boolean result of the GE operation on the contents of data register D[a] and either data register D[b] (instruction format RR) or const9 (instruction format RC). Put the result in D[c][0]. All other bits in D[c] are unchanged. D[a] and D[b] are treated as 32-bit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

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




```
D[c] = {D[c][31:1], D[c][0] OR (D[a] >= sign_ext(const9))};
```

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




```
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.ge    d3, d1, #126
or.ge    d3, d1, d2
```

See Also

AND.GE, AND.GE.U, OR.GE.U, XOR.GE, XOR.GE.U

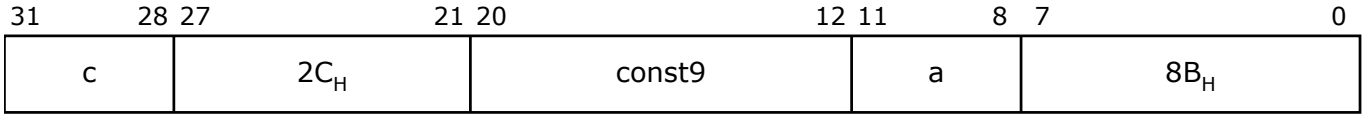
3.1.256 OR.GE.U

Greater Than or Equal, OR Accumulating Unsigned

Description

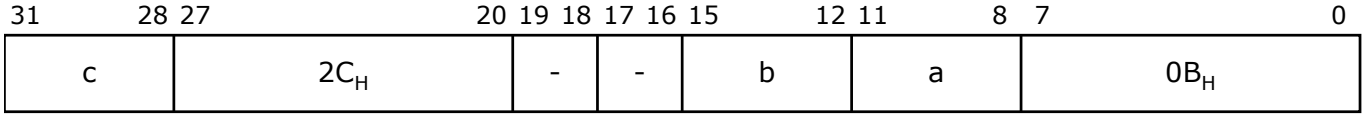
Calculate the logical OR of D[c][0] and the Boolean result of the GE.U operation on the contents of data register D[a] and either data register D[b] (instruction format RR) or const9 (instruction format RC). Put the result in D[c][0]. All other bits in D[c] are unchanged. D[a] and D[b] are treated as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

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




```
D[c] = {D[c][31:1], D[c][0] OR (D[a] >= zero_ext(const9))}; // unsigned
```

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




```
D[c] = {D[c][31:1], D[c][0] OR (D[a] >= D[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

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

See Also

AND.GE, AND.GE.U, OR.GE, XOR.GE, XOR.GE.U

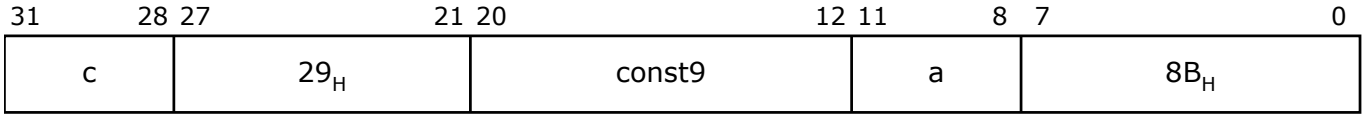
3.1.257 OR.LT

Less Than, OR Accumulating

Description

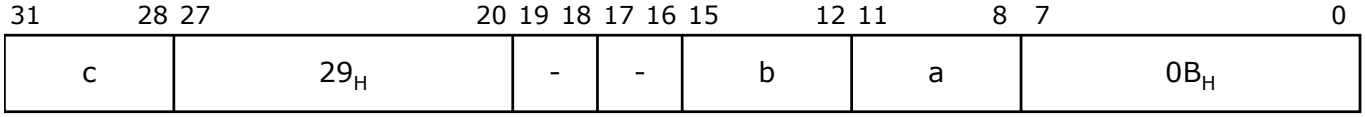
Calculate the logical OR of D[c][0] and the Boolean result of the LT operation on the contents of data register D[a] and either data register D[b] (instruction format RR) or const9 (instruction format RC). Put the result in D[c][0]. All other bits in D[c] are unchanged. D[a] and D[b] are treated as 32-bit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

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




```
D[c] = {D[c][31:1], D[c][0] OR (D[a] < sign_ext(const9))};
```

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




```
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.lt    d3, d1, #126
or.lt    d3, d1, d2
```

See Also

AND.LT, AND.LT.U, OR.LT.U, XOR.LT, XOR.LT.U

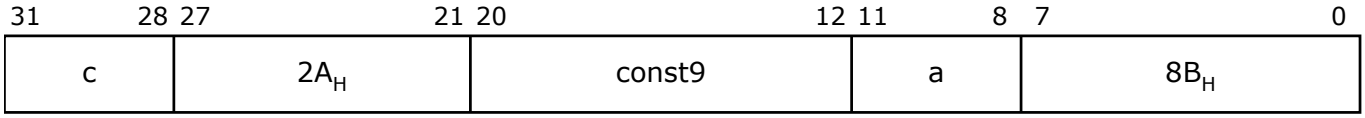
3.1.258 OR.LT.U

Less Than, OR Accumulating Unsigned

Description

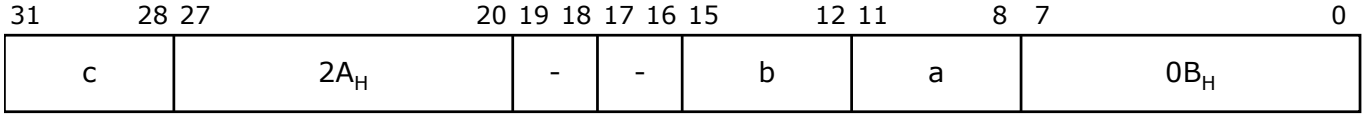
Calculate the logical OR of D[c][0] and the Boolean result of the LT.U operation on the contents of data register D[a] and either data register D[b] (instruction format RR) or const9 (instruction format RC). Put the result in D[c][0]. All other bits in D[c] are unchanged. D[a] and D[b] are treated as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

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




```
D[c] = {D[c][31:1], D[c][0] OR (D[a] < zero_ext(const9))}; // unsigned
```

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




```
D[c] = {D[c][31:1], D[c][0] OR (D[a] < D[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

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

See Also

AND.LT, AND.LT.U, OR.LT, XOR.LT, XOR.LT.U

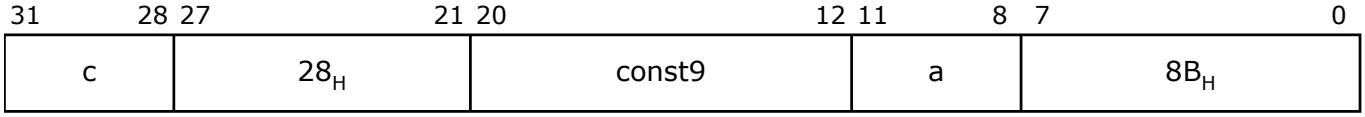
3.1.259 OR.NE

Not Equal, OR Accumulating

Description

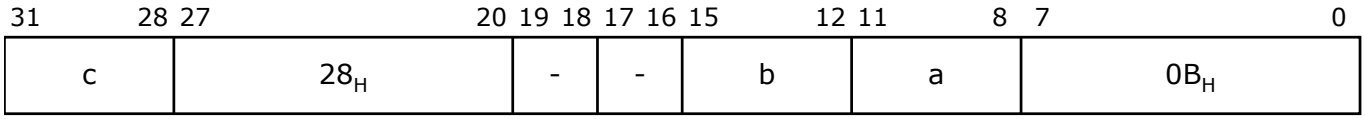
Calculate the logical OR of D[c][0] and the Boolean result of the NE operation on the contents of data register D[a] and either data register D[b] (instruction format RR) or const9 (instruction format RC). Put the result in D[c][0]. All other bits in D[c] are unchanged.

OR.NE D[c], D[a], const9 (RC)




```
D[c] = {D[c][31:1], D[c][0] OR (D[a] != sign_ext(const9))};
```

OR.NE D[c], D[a], D[b] (RR)




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

See Also

AND.NE, XOR.NE

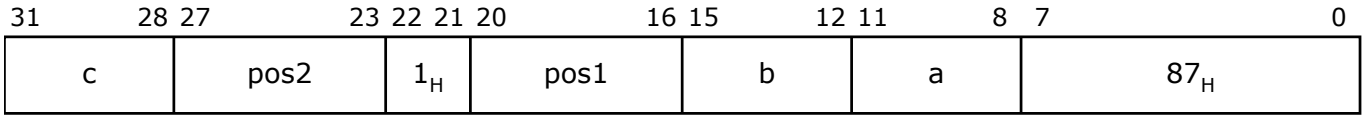
3.1.260 OR.T

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

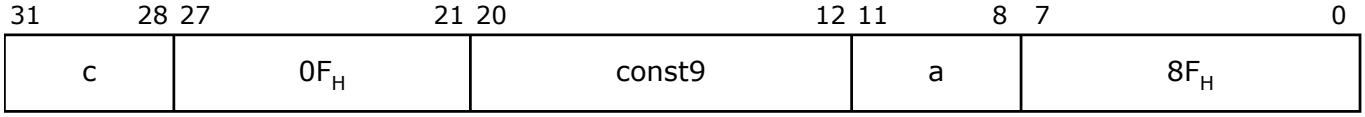
3.1.261 ORN

Bitwise OR-Not

Description

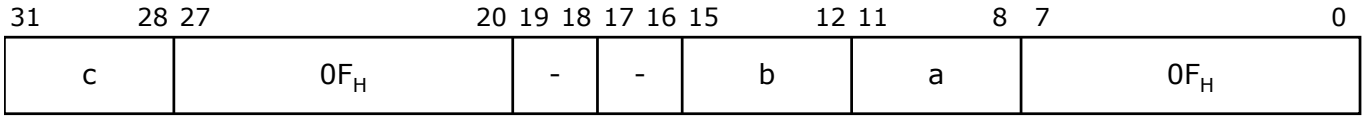
Compute the bitwise OR of the contents of data register D[a] and the ones' complement of the contents of either data register D[b] (instruction format RR) or const9 (instruction format RC). Put the result in data register D[c]. The const9 value is zero-extended to 32-bit.

ORN D[c], D[a], const9 (RC)




```
D[c] = D[a] | ~zero_ext(const9);
```

ORN D[c], D[a], D[b] (RR)




```
D[c] = 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

```
orn    d3, d1, #126
orn    d3, d1, d2
```

See Also

AND, ANDN, NAND, NOR, NOT, OR, XNOR, XOR

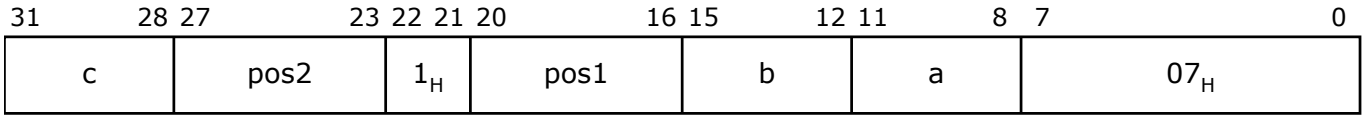
3.1.262 ORN.T

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

3.1.263 PACK

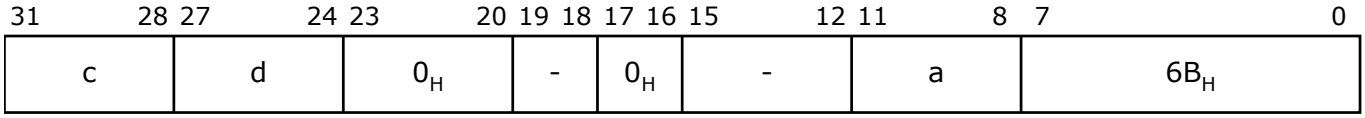
Pack

Description

Take the data register pair E[d] and bit 31 of data register D[a] and pack them into a single-precision IEEE 754-2019 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 subnormal 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 subnormal 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 754-2019 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)




```
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 {
        // subnormal
        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

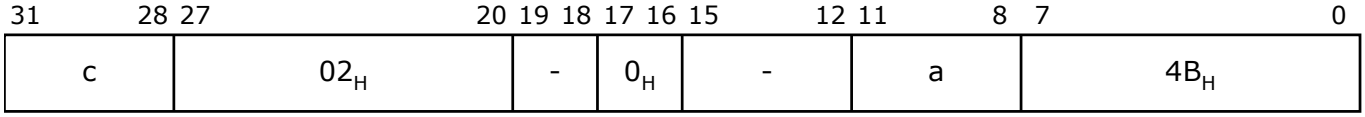
3.1.264 PARITY

Parity

Description

Compute the four byte parity bits of data register D[a]. Put each byte parity value into the LSB of each respective byte of data register D[c] and 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)




```
D[c][31:24] = {7'b0, D[a][31] ^ D[a][30] ^ D[a][29] ^ D[a][28] ^ D[a][27] ^ D[a][26] ^ D[a][25] ^ D[a]
[24]};
D[c][23:16] = {7'b0, D[a][23] ^ D[a][22] ^ D[a][21] ^ D[a][20] ^ D[a][19] ^ D[a][18] ^ D[a][17] ^ D[a]
[16]};
D[c][15:8] = {7'b0, D[a][15] ^ D[a][14] ^ D[a][13] ^ D[a][12] ^ D[a][11] ^ D[a][10] ^ D[a][9] ^ D[a]
[8]};
D[c][7:0] = {7'b0, D[a][7] ^ D[a][6] ^ D[a][5] ^ D[a][4] ^ D[a][3] ^ D[a][2] ^ D[a][1] ^ 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

CRCN

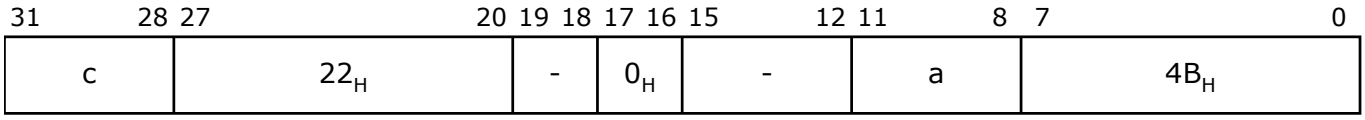
3.1.265 POPCNT.W

Population Count Word

Description

Count the total number of ones in register D[a] and put the result in register D[c].

POPCNT.W D[c], D[a] (RR)




```
D[c][5:0] = population_count( D[a] );
D[c][31:6] = 0000000H;
```

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

```
popcnt.w    d0, d1
```

See Also

CLO, CLS, CLZ, CLO.H, CLS.H, CLZ.H

3.1.266 REM64

Remainder 64-bit Long

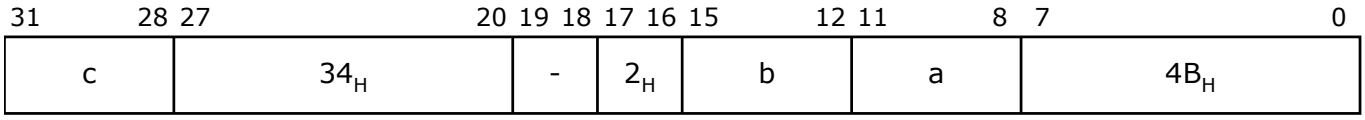
Description

Divide the contents of register E[a] by the contents of register E[b]. Put the resulting remainder in E[c][63:0]. This instruction does not provide a quotient.

The operands and results are treated as 64-bit signed integers.

Overflow occurs if the divisor (E[b]) is zero or if the dividend (E[a]) is the minimum negative number and the divisor is minus 1.

REM64 E[c], E[a], E[b] (RR)




```
dividend = E[a];
divisor = E[b];
if (divisor == 0) then {
    if (dividend >= 0) then {
        remainder = 0x00000000_00000000;
    } else {
        remainder = 0x00000000_00000000;
    }
} else if ((divisor == 0xffffffff_ffffffff) AND (dividend == 0x80000000_00000000)) then {
    remainder = 0x00000000_00000000;
} else {
    remainder = dividend % divisor
}
E[c][63:0] = remainder;
```

Status Flags

C	Not set by this instruction.
V	if ((E[b] == 0) OR ((E[b] == 64'hFFFFFFFFFFFFFFFF) AND (E[a] == 64'h8000000000000000))) then overflow = 1 else overflow = 0; if overflow then PSW.V = 1 else PSW.V = 0;
SV	if (overflow) then PSW.SV = 1 else PSW.SV = PSW.SV;
AV	PSW.AV = 0;
SAV	Not set by this instruction.

Examples

```
rem64    e0, e2, e4
```

See Also

DIV, DIV.U, DIV64, DIV64.U, REM64.U

3.1.267 REM64.U

Remainder 64-bit Unsigned Long

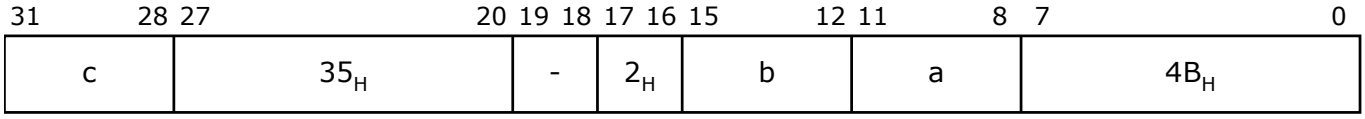
Description

Divide the contents of register E[a] by the contents of register E[b]. Put the resulting remainder in E[c][63:0]. This instruction does not provide a quotient.

The operands and results are treated as 64-bit unsigned long integers.

Overflow occurs if the divisor (E[b]) is zero.

REM64.U E[c], E[a], E[b] (RR)




```
dividend = E[a];
divisor = E[b];
if (divisor == 0) then {
    remainder = 0x00000000_00000000;
} else {
    remainder = dividend % divisor
}
E[c][63:0] = remainder;
```

Status Flags

C	Not set by this instruction.
V	if (E[b] == 0) then overflow = 1 else overflow = 0; if overflow then PSW.V = 1 else PSW.V = 0;
SV	if (overflow) then PSW.SV = 1 else PSW.SV = PSW.SV;
AV	PSW.AV = 0;
SAV	Not set by this instruction.

Examples

```
rem64.u    e0, e2, e4
```

See Also

DIV, DIV.U, DIV64, DIV64.U, REM64

3.1.268 RESTORE

Restore

Description

***Note**: Execution Mode - This instruction can only be executed in the following modes: User-1, Supervisor.*

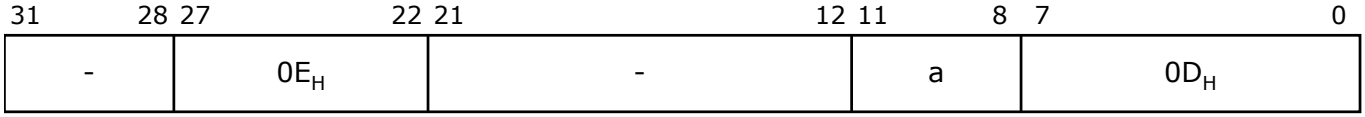
If the virtualization extension is not present or disabled

- Restore the Interrupt Enable bit (ICR.IE) to the value in D[a][0].

If the virtualization extension is present and enabled

- Restore the Interrupt Enable bit (VMn_ICR.IE) for the current virutal machine (VMn) to the value in D[a][0] (ICR and VMn_ICR are aliases).

RESTORE D[a] (SYS)




```
if (TCCON.VIRT==1 AND VCON0.EN==1) then {
    // Restore interrupt enable for virtual machine n
    // ICR is aliased to VMn_ICR
    VMn_ICR.IE = D[a][0];
} else {
    ICR.IE = 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

```
restore    d7
```

See Also

DISABLE, ENABLE

3.1.269 RET

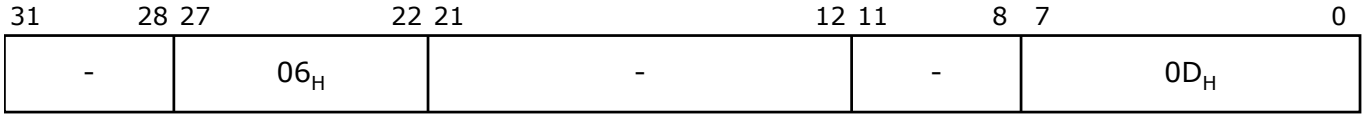
Return from Call

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.

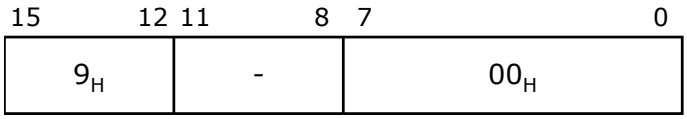
Return from a function that was invoked with a CALL instruction. The return address is in register A[11] (return address). The caller's upper context register values are restored as part of the return operation.

RET (SYS)




```
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 = {A[11] [31:1], 1'b0};
EA = {PCXI.PCXS, 6'b0, PCXI.PCXO, 6'b0};
{new_PCXI, csa_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  = {csa_PSW[31:26], PSW[25:24], csa_PSW[23:16], PPRS[2], csa_PSW[14], PPRS[1:0], csa_PSW[11:0]};
PPRS = {csa_PSW.PRS[2], csa_PSW.PRS[1:0]};
```

RET (SR)




```
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 = {A[11] [31:1], 1'b0};
EA = {PCXI.PCXS, 6'b0, PCXI.PCXO, 6'b0};
{new_PCXI, csa_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  = {csa_PSW[31:16], PSW[25:24], csa_PSW[23:16], PPRS[2], csa_PSW[14], PPRS[1:0], csa_PSW[11:0]};
PPRS = {csa_PSW.PRS[2], csa_PSW.PRS[1:0]};
```

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, FCALL, FCALLA, FCALLI

3.1.270 RFE

Return From Exception

Description

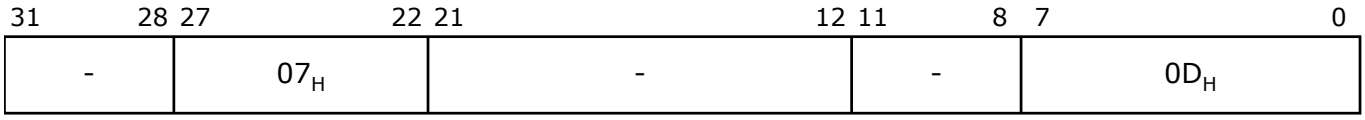
Return from an interrupt service routine or trap handler to the task whose saved upper context is specified by the contents of the Previous Context Information register (PCXI). The contents are normally the context of the task that was interrupted or that took a trap. However, in some cases Task Management software may have altered the contents of the PCXI register to cause another task to be dispatched.

The return address is taken from register A[11] (return address) in the current context. 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.

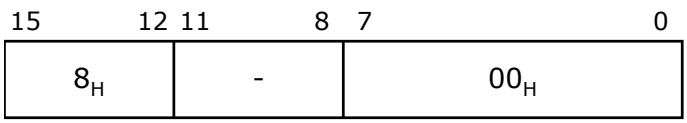
The return address is taken from register A[11] (return address) in the current context. The upper context registers and PSW in the saved context are restored.

RFE (SYS)




```
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 = {A[11] [31:1], 1'b0};
ICR.IE = PCXI.PIE;
ICR.CCPN = PCXI.PCPN;
EA = {PCXI.PCXS, 6'b0, PCXI.PCXO, 6'b0};
{new_PCXI, csa_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 = {csa_PSW[31:16], PPRS[2], csa_PSW[14], PPRS[1:0], csa_PSW[11:0]};
PPRS = {csa_PSW.PRS[2], csa_PSW.PRS[1:0]};
```

RFE (SR)



```
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 = {A[11] [31:1], 1'b0};
ICR.IE = PCXI.PIE;
ICR.CCPN = PCXI.PCPN;
EA = {PCXI.PCXS, 6'b0, PCXI.PCXO, 6'b0};
{new_PCXI, csa_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 = {csa_PSW[31:16], PPRS[2], csa_PSW[14], PPRS[1:0], csa_PSW[11:0]};
PPRS = {csa_PSW.PRS[2], csa_PSW.PRS[1:0]};
```

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

```
rfe
```

```
rfe
```

See Also

CALL, CALLA, CALLI, RET, SYSCALL, BISR, RFH, RFM

3.1.271 RFM

Return From Monitor

Description

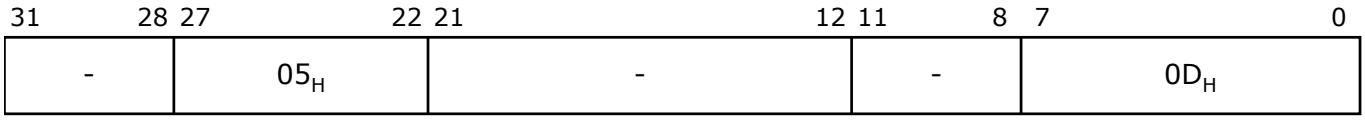
***Note**: Execution Mode - This instruction can only be executed in the following mode: Supervisor.*

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]. 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.

The Debug Trap active bit (DBGTCR.DTA) is cleared.

RFM (SYS)



***Note**: The Debug Context Pointer (DCX) value is implementation dependent.*

```
if (DBGSR.DE) then {
    PC = {A[11] [31:1], 1'b0};
    ICR.IE = PCXI.IE;
    ICR.CCPN = PCXI.PCPN;
    EA = DCX;
    {PCXI, csa_PSW, A[10], A[11]} = M(EA, 4 * word);
    PSW = {csa_PSW[31:16], PPRS[2], csa_PSW[14], PPRS[1:0], csa_PSW[11:0]};
    PPRS = {csa_PSW.PRS[2], csa_PSW.PRS[1:0]};
    DBGTCR.DTA = 0;
} 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.
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

```
rfm
```

See Also

DEBUG, RFE

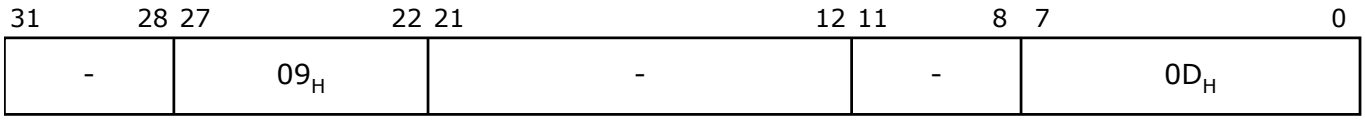
3.1.272 RSLCX

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)




```
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

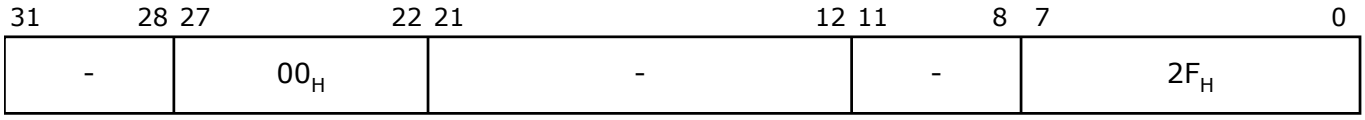
3.1.273 RSTV

Reset Overflow Bits

Description

Reset overflow status flags in the Program Status Word (PSW).

RSTV (SYS)




```
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, MTDCR, TRAPV, TRAPSV

3.1.274 RSUB

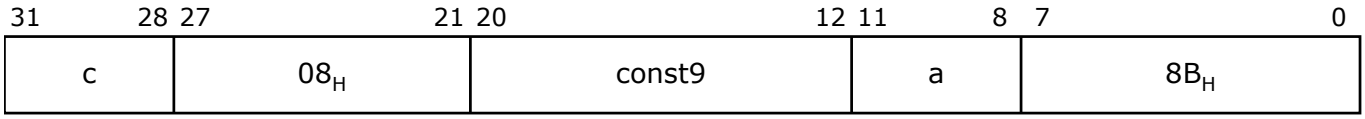
Reverse-Subtract

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 signed integers. The value const9 is sign-extended before the subtraction is performed.

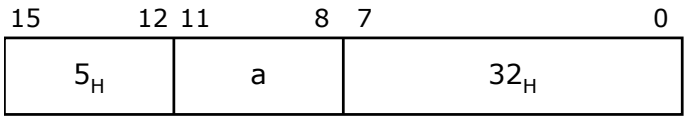
Subtract the contents of data register D[a] from zero and put the result in data register D[a]. The operand is treated as a 32-bit signed integer.

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 > $7FFFFFFF_H$) OR (result < -80000000_H); 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

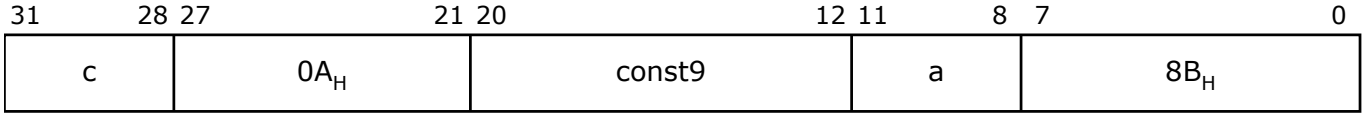
3.1.275 RSUBS

Reverse-Subtract 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 32-bit signed integers, with saturation on signed overflow. The value const9 is sign-extended before the operation is performed.

RSUBS D[c], D[a], const9 (RC)




```
result = sign_ext(const9) - D[a];
D[c] = ssov(result, 32);
```

Status Flags

C	Not set by this instruction.

V	signed:
	overflow = (result > 7FFFFFFF$_H$) OR (result < -80000000$_H$);
	if (overflow) then PSW.V = 1 else PSW.V = 0;
	unsigned:
	overflow = (result > FFFFFFFF$_H$) OR (result < 00000000$_H$);
	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
```

See Also

RSUB, RSUBS.U

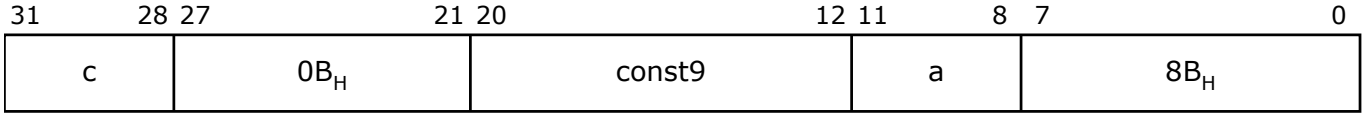
3.1.276 RSUBS.U

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 32-bit unsigned integers, with saturation on unsigned overflow. The value const9 is sign-extended before the operation is performed.

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




```
result = sign_ext(const9) - D[a]; // unsigned
D[c] = suov(result, 32);
```

Status Flags

C	Not set by this instruction.

V	signed:
	overflow = (result > 7FFFFFFF$_H$) OR (result < -80000000$_H$);
	if (overflow) then PSW.V = 1 else PSW.V = 0;
	unsigned:
	overflow = (result > FFFFFFFF$_H$) OR (result < 00000000$_H$);
	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.u    d3, d1, #126
```

See Also

RSUB, RSUBS

3.1.277 SAT.B

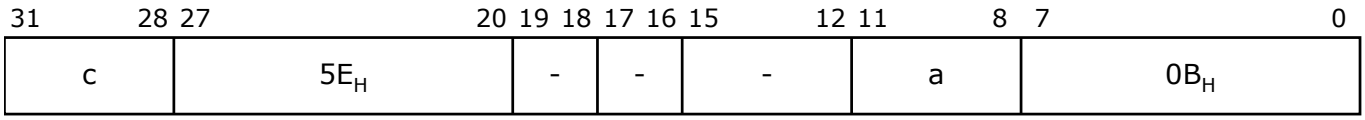
Saturate Byte

Description

If the signed 32-bit value in D[a] is less than -128, then store the value -128 in D[c]. If D[a] is greater than 127, then store the value 127 in D[c]. Otherwise, copy D[a] to D[c].

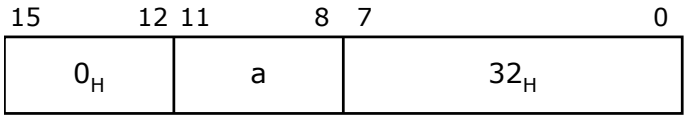
If the signed 32-bit value in D[a] is less than -128, then store the value -128 in D[a].
If D[a] is greater than 127, then store the value 127 in D[a]. Otherwise, leave the contents of D[a] unchanged.

SAT.B D[c], D[a] (RR)




```
sat_neg = (D[a] < -80H) ? -80H : D[a];
D[c] = (sat_neg > 7FH) ? 7FH : sat_neg;
```

SAT.B D[a] (SR)




```
sat_neg = (D[a] < -80H) ? -80H : D[a];
D[a] = (sat_neg > 7FH) ? 7FH : sat_neg;
```

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.b    d3, d1
```

```
sat.b    d1
```

See Also

SAT.BU, SAT.H, SAT.HU

3.1.278 SAT.BU

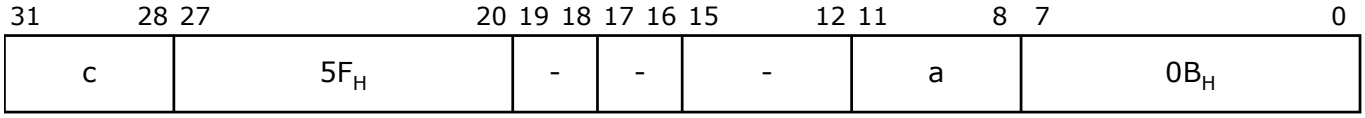
Saturate Byte Unsigned

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].

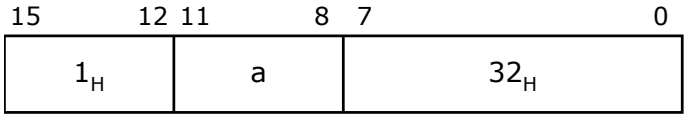
If the unsigned 32-bit value in D[a] is greater than 255, then store the value 255 in D[a]. Otherwise leave the contents of D[a] unchanged.

SAT.BU D[c], D[a] (RR)




```
D[c] = (D[a] > FF_H) ? FF_H : D[a]; // unsigned comparison
```

SAT.BU D[a] (SR)




```
D[a] = (D[a] > FF_H) ? FF_H : 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

3.1.279 SAT.H

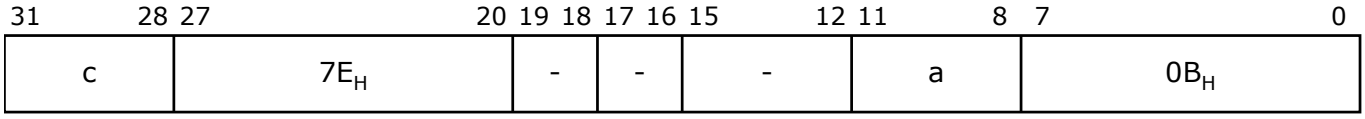
Saturate Half-word

Description

If the signed 32-bit value in D[a] is less than -32,768, then store the value -32,768 in D[c]. If D[a] is greater than 32,767, then store the value 32,767 in D[c]. Otherwise copy D[a] to D[c].

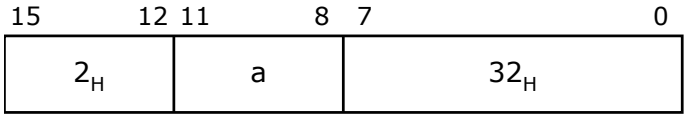
If the signed 32-bit value in D[a] is less than -32,768, then store the value -32,768 in D[a]. If D[a] is greater than 32,767, then store the value 32,767 in D[a]. Otherwise leave the contents of D[a] unchanged.

SAT.H D[c], D[a] (RR)




```
sat_neg = (D[a] < -8000H) ? -8000H : D[a];
D[c] = (sat_neg > 7FFFH) ? 7FFFH : sat_neg;
```

SAT.H D[a] (SR)




```
sat_neg = (D[a] < -8000H) ? -8000H : D[a];
D[a] = (sat_neg > 7FFFH) ? 7FFFH : sat_neg;
```

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.h    d3, d1
```

```
sat.h    d1
```

See Also

SAT.B, SAT.BU, SAT.HU

3.1.280 SAT.HU

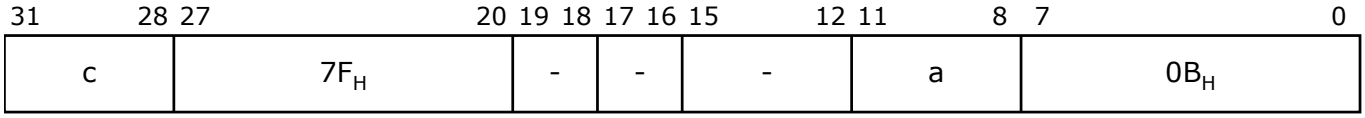
Saturate Half-word Unsigned

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].

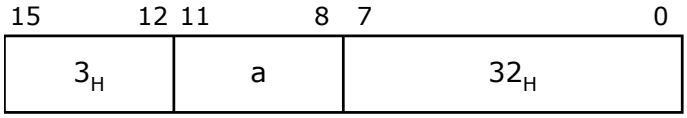
If the unsigned 32-bit value in D[a] is greater than 65,535, then store the value 65,535 in D[a]; otherwise leave the contents of D[a] unchanged.

SAT.HU D[c], D[a] (RR)




```
D[c] = (D[a] > FFFF_H) ? FFFF_H : D[a]; // unsigned comparison
```

SAT.HU D[a] (SR)




```
D[a] = (D[a] > FFFF_H) ? FFFF_H : 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

3.1.281 SEL

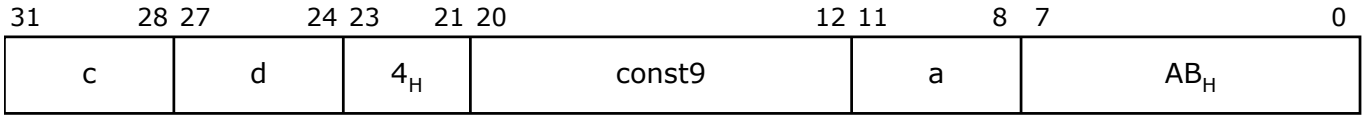
Select

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].

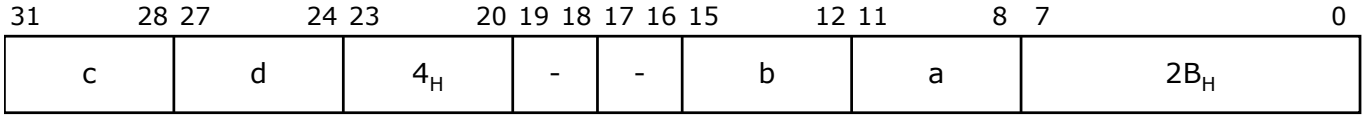
The value const9 (instruction format RCR) is sign-extended.

SEL D[c], D[d], D[a], const9 (RCR)




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

SEL D[c], D[d], D[a], D[b] (RRR)




```
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

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

See Also

CADD, CADDN, CMOV, CMOVN, CSUB, CSUBN, SELN

3.1.282 SELN

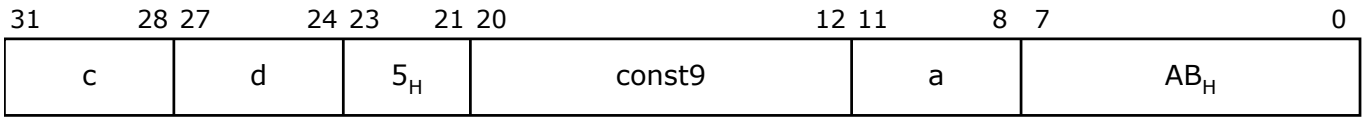
Select-Not

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].

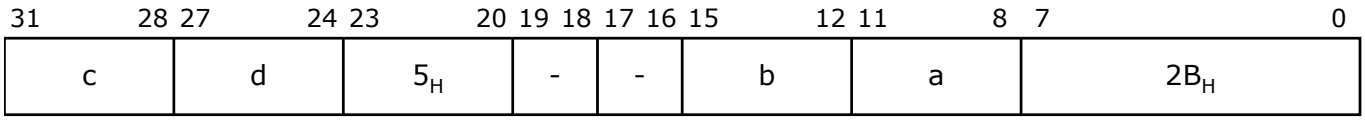
The value const9 (instruction format RCR) is sign-extended.

SELN D[c], D[d], D[a], const9 (RCR)




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

SELN D[c], D[d], D[a], D[b] (RRR)




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

See Also

CADD, CADDN, CMOV, CMOVN, CSUB, CSUBN, SEL

3.1.283 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.

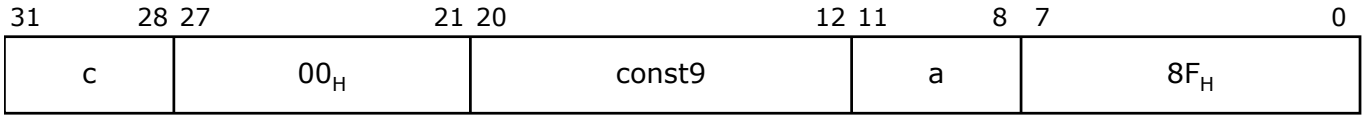
The shift count is a 6-bit signed number, derived from either D[b][5:0] or const9[5:0]. The range for the shift count is therefore -32 to +31, allowing a shift left up to 31 bit positions and to shift right up to 32 bit positions (Note that a shift right by 32 bits leaves zeros in the result).

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.

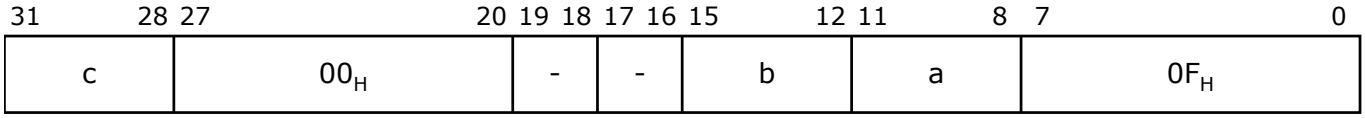
The shift count is a 4-bit signed number, derived from the sign-extension of const4[3:0]. The resulting range for the shift count therefore is -8 to +7, allowing a shift left up to 7-bit positions and to shift right up to 8-bit positions.

SH D[c], D[a], const9 (RC)



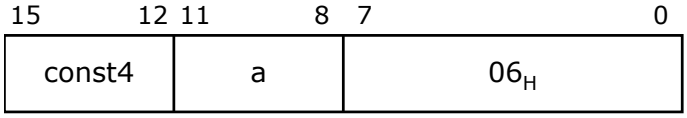

```
shift_count = sign_ext(const9[5:0]);
D[c] = (shift_count >= 0) ? (D[a] << shift_count) : (D[a] >> (0 - shift_count));
```

SH D[c], D[a], D[b] (RR)




```
shift_count = sign_ext(D[b][5:0]);
D[c] = (shift_count >= 0) ? (D[a] << shift_count) : (D[a] >> (0 - shift_count));
```

SH D[a], const4 (SRC)




```
shift_count = sign_ext(const4[3:0]);
D[a] = (shift_count >= 0) ? (D[a] << shift_count) : (D[a] >> (0 - 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, #26
sh    d3, d1, d2
sh    d1, #6
```

See Also

SH.H, SHA, SHA.H, SHAS

3.1.284 SH.EQ

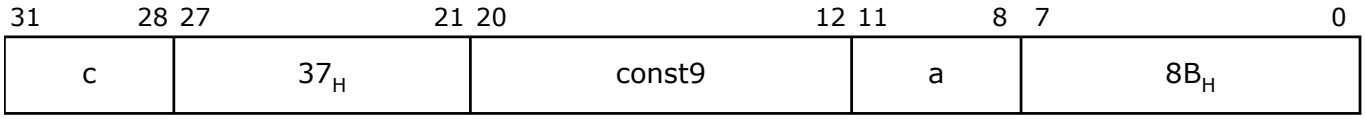
Shift Equal

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.

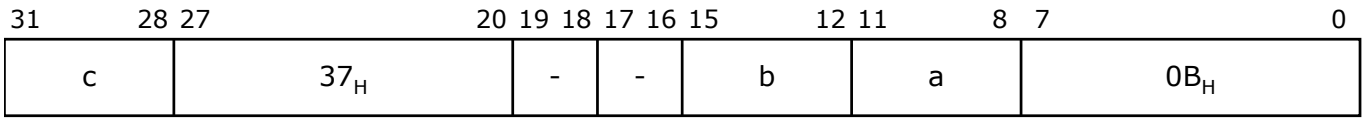
The value const9 (format RC) is sign-extended.

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




```
D[c] = {D[c][30:0], (D[a] == sign_ext(const9))};
```

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




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

See Also

SH.GE, SH.GE.U, SH.LT, SH.LT.U, SH.NE

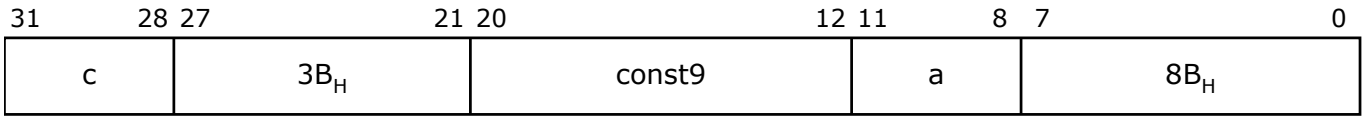
3.1.285 SH.GE

Shift Greater Than or Equal

Description

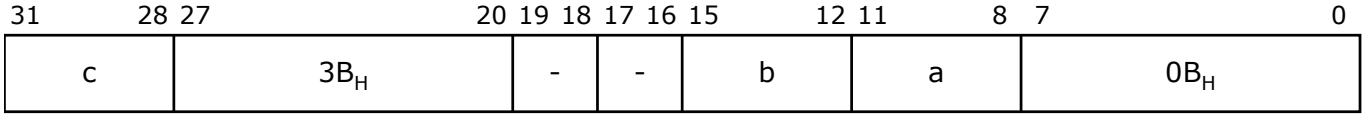
Left shift D[c] by one. If the contents of data register D[a] are greater than or equal to the contents of either data register D[b] (instruction format RR) or const9 (instruction format RC), set the least-significant bit of D[c] to one; otherwise set the least-significant bit of D[c] to 0. D[a] and D[b] are treated as signed integers. The value const9 is sign-extended.

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




```
D[c] = {D[c][30:0], (D[a] >= sign_ext(const9))};
```

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




```
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.ge    d3, d1, #126
sh.ge    d3, d1, d2
```

See Also

SH.EQ, SH.GE.U, SH.LT, SH.LT.U, SH.NE

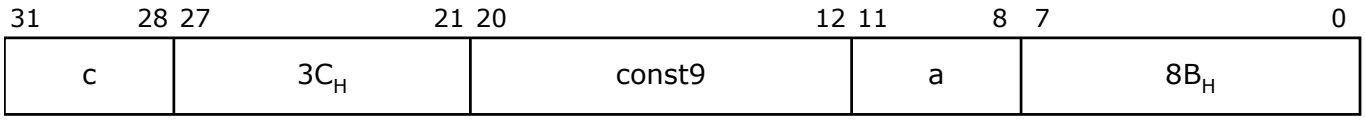
3.1.286 SH.GE.U

Shift Greater Than or Equal Unsigned

Description

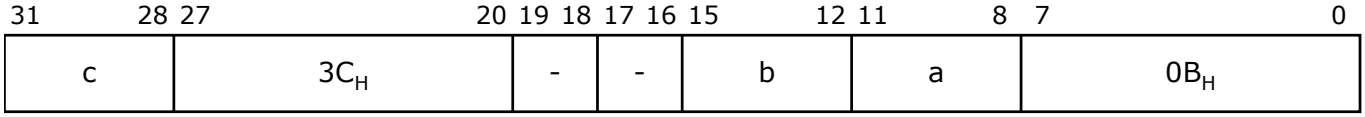
Left shift D[c] by one. If the contents of data register D[a] are greater than or equal to the contents of either data register D[b] (instruction format RR) or const9 (instruction format RC), set the least-significant bit of D[c] to one; otherwise set the least-significant bit of D[c] to 0. D[a] and D[b] are treated as unsigned integers. The value const9 is zero-extended.

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




```
D[c] = {D[c][30:0], (D[a] >= zero_ext(const9))}; // unsigned
```

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




```
D[c] = {D[c][30:0], (D[a] >= D[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

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

See Also

SH.EQ, SH.GE, SH.LT, SH.LT.U, SH.NE

3.1.287 SH.H

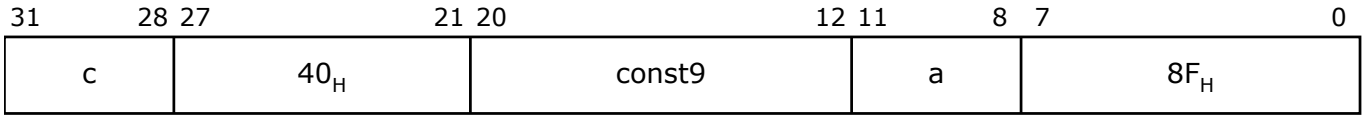
Shift Packed Half-words

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.

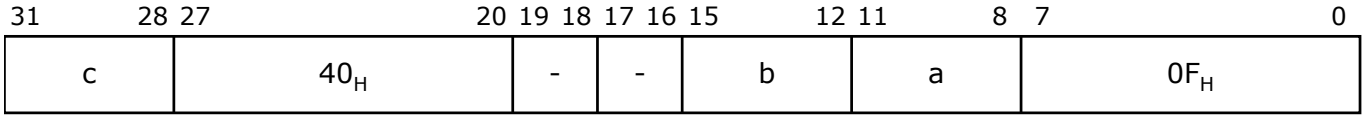
The shift count is a signed number, derived from the sign-extension of either D[b][4:0] (instruction format RR) or const9[4:0] (instruction format RC). The range for the shift count is therefore -16 to +15. The result for a shift count of -16 for half-words is zero.

SH.H D[c], D[a], const9 (RC)




```
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)




```
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, #12
sh.h    d3, d1, d2
```

See Also

SH, SHA, SHA.H, SHAS

3.1.288 SH.LT

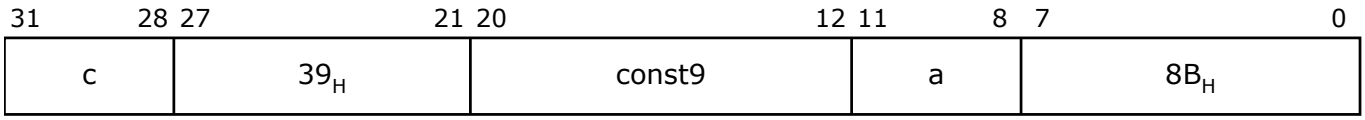
Shift Less Than

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.

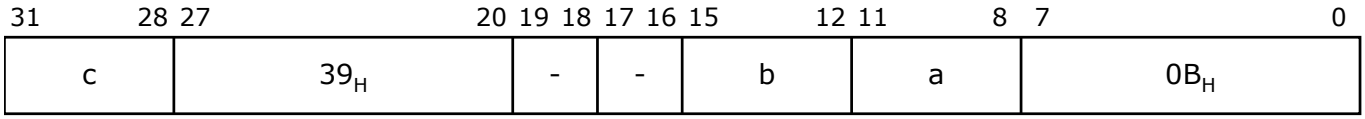
D[a] and either D[b] (format RR) or const9 (format RC) are treated as signed integers. The value const9 is sign-extended.

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




```
D[c] = {D[c][30:0], (D[a] < sign_ext(const9))};
```

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




```
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.lt    d3, d1, #126
sh.lt    d3, d1, d2
```

See Also

SH.EQ, SH.GE, SH.GE.U, SH.LT.U, SH.NE

3.1.289 SH.LT.U

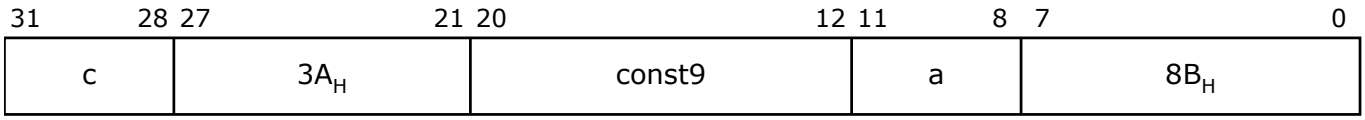
Shift Less Than Unsigned

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.

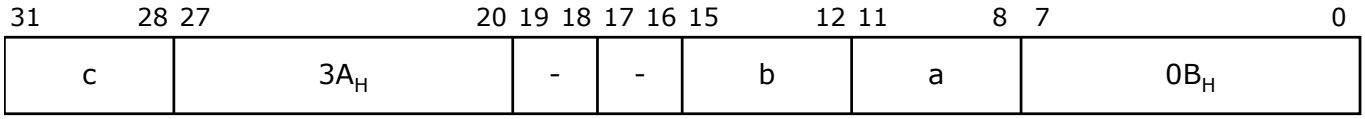
D[a] and either D[b] (format RR) or const9 (format RC) are treated as unsigned integers. The value const9 is zero-extended.

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




```
D[c] = {D[c][30:0], (D[a] < zero_ext(const9))}; // unsigned
```

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




```
D[c] = {D[c][30:0], (D[a] < D[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

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

See Also

SH.EQ, SH.GE, SH.GE.U, SH.LT, SH.NE

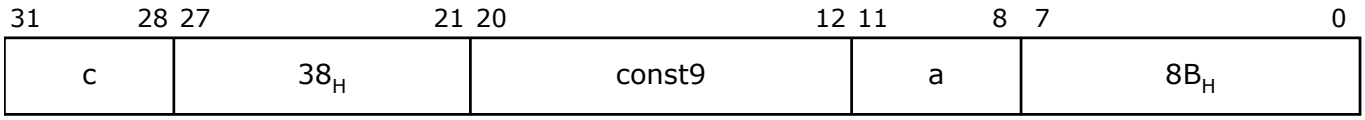
3.1.290 SH.NE

Shift Not Equal

Description

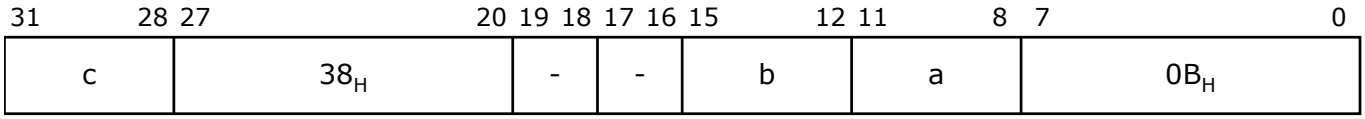
Left shift D[c] by one. If the contents of data register D[a] are not equal to the contents of either data register D[b] (instruction format RR) or const9 (instruction format RC), set the least-significant bit of D[c] to one; otherwise set the least-significant bit of D[c] to zero. The value const9 is sign-extended.

SH.NE D[c], D[a], const9 (RC)




```
D[c] = {D[c][30:0], (D[a] != sign_ext(const9))};
```

SH.NE D[c], D[a], D[b] (RR)




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

See Also

SH.EQ, SH.GE, SH.GE.U, SH.LT, SH.LT.U

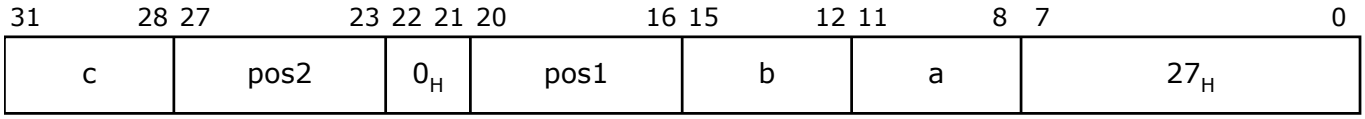
3.1.291 SH.AND.T

Accumulating Shift-AND

Description

Left shift D[c] by one. The bit shifted out is discarded. Compute the logical AND 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] AND D[b][pos2])};
```

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.and.t    d3, d1, #4, d2, #7
```

See Also

AND.AND.T, AND.ANDN.T, AND.NOR.T, AND.OR.T, SH.ANDN.T, SH.NAND.T, SH.NOR.T, SH.OR.T, SH.ORN.T, SH.XNOR.T, SH.XOR.T, OR.AND.T, OR.ANDN.T, OR.NOR.T, OR.OR.T

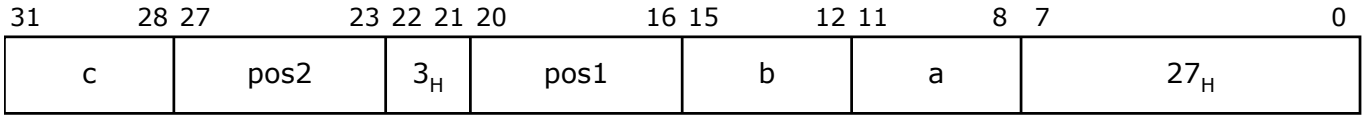
3.1.292 SH.ANDN.T

Accumulating Shift-AND-Not

Description

Left shift D[c] by one. The bit shifted out is discarded. Compute the logical ANDN 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.ANDN.T D[c], D[a], pos1, D[b], pos2 (BIT)




```
D[c] = {D[c][30:0], (D[a][pos1] AND !(D[b][pos2]))};
```

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.andn.t    d3, d1, #4, d2, #7
```

See Also

AND.AND.T, AND.ANDN.T, AND.NOR.T, AND.OR.T, SH.AND.T, SH.NAND.T, SH.NOR.T, SH.OR.T, SH.ORN.T, SH.XNOR.T, SH.XOR.T, OR.AND.T, OR.ANDN.T, OR.NOR.T, OR.OR.T

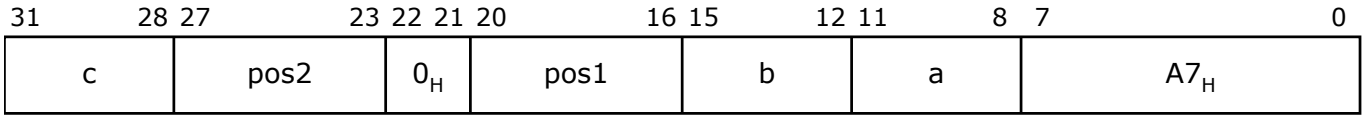
3.1.293 SH.NAND.T

Accumulating Shift-NAND

Description

Left shift D[c] by one. The bit shifted out is discarded. Compute the logical NAND 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.NAND.T D[c], D[a], pos1, D[b], pos2 (BIT)




```
D[c] = {D[c][30:0], !(D[a][pos1] AND D[b][pos2])};
```

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.nand.t    d3, d1, #4, d2, #7
```

See Also

AND.AND.T, AND.ANDN.T, AND.NOR.T, AND.OR.T, SH.AND.T, SH.ANDN.T, SH.NOR.T, SH.OR.T, SH.ORN.T, SH.XNOR.T, SH.XOR.T, OR.AND.T, OR.ANDN.T, OR.NOR.T, OR.OR.T

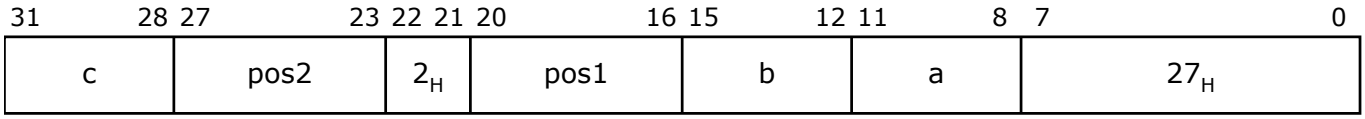
3.1.294 SH.NOR.T

Accumulating Shift-NOR

Description

Left shift D[c] by one. The bit shifted out is discarded. Compute the logical NOR 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.NOR.T D[c], D[a], pos1, D[b], pos2 (BIT)




```
D[c] = {D[c][30:0], !(D[a][pos1] OR D[b][pos2])};
```

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.nor.t    d3, d1, #4, d2, #7
```

See Also

AND.AND.T, AND.ANDN.T, AND.NOR.T, AND.OR.T, SH.AND.T, SH.ANDN.T, SH.NAND.T, SH.OR.T, SH.ORN.T, SH.XNOR.T, SH.XOR.T, OR.AND.T, OR.ANDN.T, OR.NOR.T, OR.OR.T

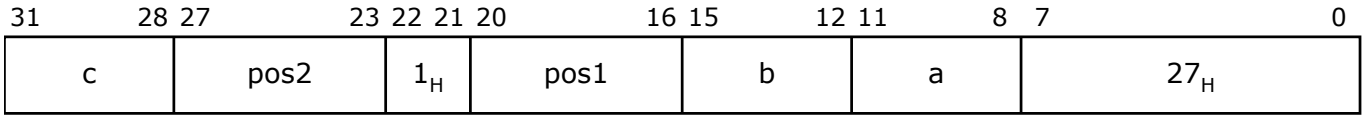
3.1.295 SH.OR.T

Accumulating Shift-OR

Description

Left shift D[c] by one. The bit shifted out is discarded. Compute the logical OR 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.OR.T D[c], D[a], pos1, D[b], pos2 (BIT)




```
D[c] = {D[c][30:0], (D[a][pos1] OR D[b][pos2])};
```

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.or.t    d3, d1, #4, d2, #7
```

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.ORN.T, SH.XNOR.T, SH.XOR.T, OR.AND.T, OR.ANDN.T, OR.NOR.T, OR.OR.T

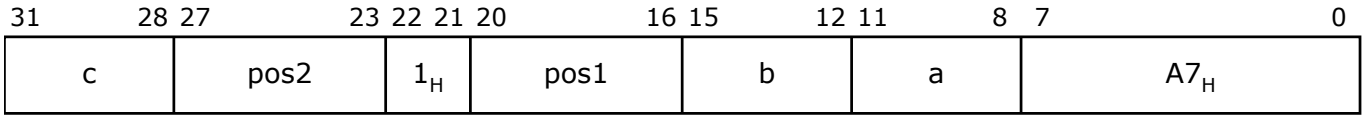
3.1.296 SH.ORN.T

Accumulating Shift-OR-Not

Description

Left shift D[c] by one. The bit shifted out is discarded. Compute the logical operation ORN 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.ORN.T D[c], D[a], pos1, D[b], pos2 (BIT)




```
D[c] = {D[c][30:0], (D[a][pos1] OR !(D[b][pos2]))};
```

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.orn.t    d3, d1, #4, d2, #7
```

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.XNOR.T, SH.XOR.T, OR.AND.T, OR.ANDN.T, OR.NOR.T, OR.OR.T

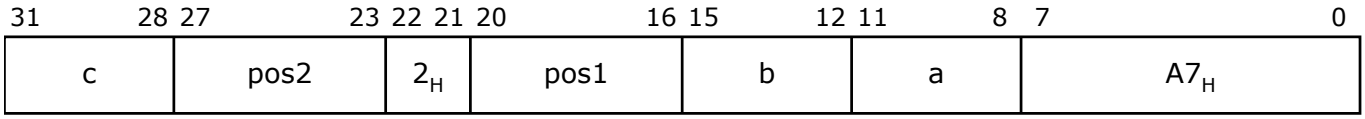
3.1.297 SH.XNOR.T

Accumulating Shift-XNOR

Description

Left shift D[c] by one. The bit shifted out is discarded. Compute the logical XNOR 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.XNOR.T D[c], D[a], pos1, D[b], pos2 (BIT)




```
D[c] = {D[c][30:0], !(D[a][pos1] XOR D[b][pos2])};
```

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.xnor.t    d3, d1, #4, d2, #7
```

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.XOR.T, OR.AND.T, OR.ANDN.T, OR.NOR.T, OR.OR.T

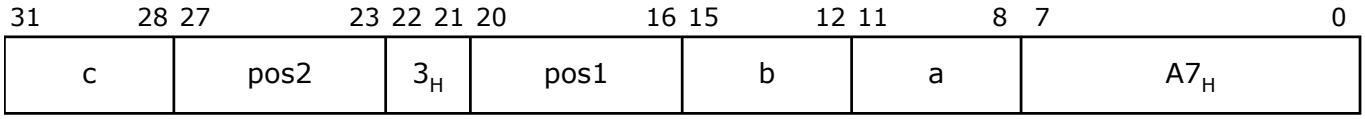
3.1.298 SH.XOR.T

Accumulating Shift-XOR

Description

Left shift D[c] by one. The bit shifted out is discarded. Compute the logical 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.XOR.T D[c], D[a], pos1, D[b], pos2 (BIT)




```
D[c] = {D[c][30:0], (D[a][pos1] XOR D[b][pos2])};
```

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.xor.t    d3, d1, #4, d2, #7
```

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.AND.T, OR.ANDN.T, OR.NOR.T, OR.OR.T

3.1.299 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 PSW.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 4-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 PSW.C is cleared.

SHA D[c], D[a], const9 (RC)

31 28	27 21	20 12	11 8	7 0
c	01_H	const9	a	$8F_H$

```
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];
```

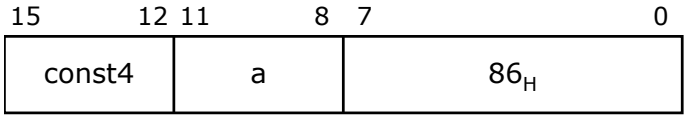
SHA D[c], D[a], D[b] (RR)

31 28	27 20	19	18	17 16	15 12	11 8	7 0
c	01_H	-		-	b	a	$0F_H$

```
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)




```
if (const4[3:0] >= 0) then {
    carry_out = const4[3:0] ? (D[a][31:32 - const4[3:0]] != 0) : 0;
    result = D[a] << const4[3:0];
} else {
    shift_count = 0 - const4[3:0];
    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 > 7FFFFFFF$_H$) OR (result < -80000000$_H$); 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 = 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, #26
sha     d3, d1, d2
```

```
sha     d1, #6
```

See Also

SH, SH.H, SHAS, SHA.H

3.1.300 SHA.H

Arithmetic Shift Packed Half-words

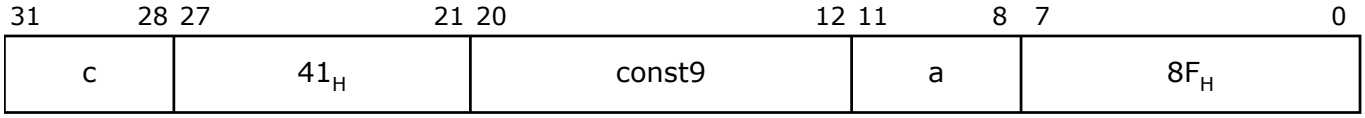
Description

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).

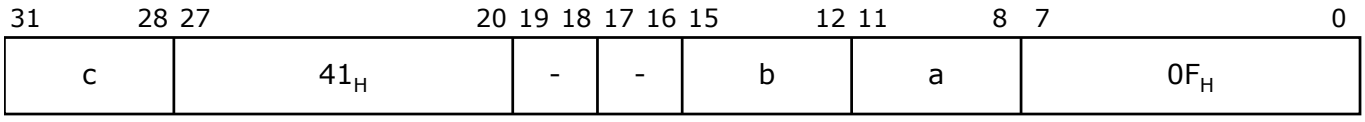
The range for the shift count is -16 to +15. The result for each half-word for a shift count of -16 is either all zeros or all ones, depending on the sign-bit of the respective half-word.

SHA.H D[c], D[a], const9 (RC)




```
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, #12
sha.h    d3, d1, d2
```

See Also

SH, SHA, SHAS, SH.H

3.1.301 SHAS

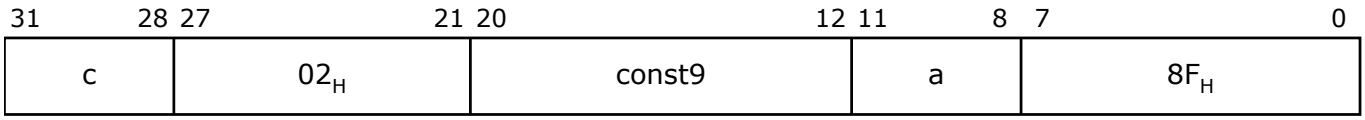
Arithmetic Shift with Saturation

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).

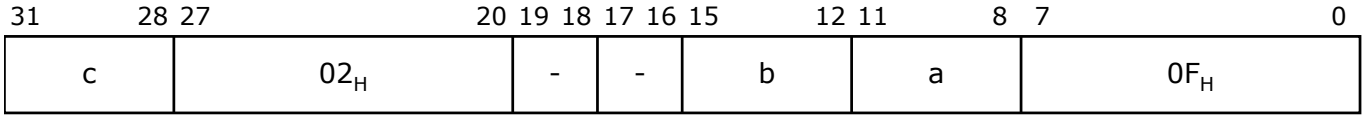
The range for the shift count is -32 to +31, allowing shift left up to 31 bit positions and to shift right up to 32 bit positions. Note that a shift right by 32 bits leaves all zeros or all ones in the result, depending on the sign-bit.

SHAS D[c], D[a], const9 (RC)




```
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)




```
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$_H$) OR (result < -80000000$_H$); 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, #26
shas    d3, d1, d2
```

See Also

SH, SH.H, SHA, SHA.H

3.1.302 SHUFFLE

Byte Shuffle

Description

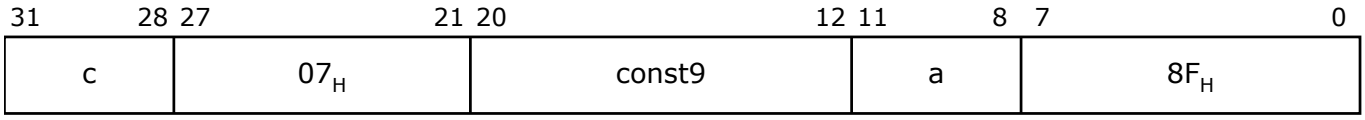
Shuffle the order of the bytes in register D[a] according to const9, and put the result in register D[c].

The value const9 contains four 2-bit fields which specify which bytes from D[a] are chosen for each byte in D[c]. The value const9[1:0] controls byte D[c][7:0], const9[3:2] controls byte D[c][15:8], const9[5:4] controls byte D[c][23:16], and const9[7:6] controls byte D[c][31:24].

The value of each 2-bit byte select field specifies the index of the source byte from D[a], where 11_B and 00_B are the most and the least significant bytes, respectively.

If const9[8] is set, each byte from D[a] is also bit reflected.

SHUFFLE D[c], D[a], const9 (RC)




```
A = byte_select( D[a], const9[1:0] );
B = byte_select( D[a], const9[3:2] );
C = byte_select( D[a], const9[5:4] );
D = byte_select( D[a], const9[7:6] );

D[c][7:0]   = (const9[8]) ? reverse(A, 8) : A;
D[c][15:8]  = (const9[8]) ? reverse(B, 8) : B;
D[c][23:16] = (const9[8]) ? reverse(C, 8) : C;
D[c][31:24] = (const9[8]) ? reverse(D, 8) : D;
```

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

```
; Change the byte endianness
shuffle    d0, d1, #0x01B

; Bit reflect the entire word
shuffle    d0, d1, #0x11B

; Copy the least significant input byte into all four byte positions
shuffle    d0, d1, #0x000
```

See Also

EXTR, INSERT

3.1.303 ST.A

Store Word from Address Register

Description

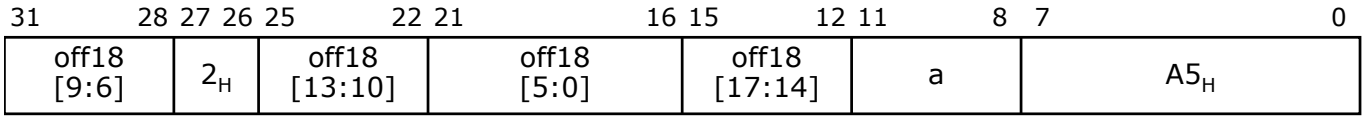
Store the value in address register A[a] to the memory location specified by the effective address.

***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 effective address.

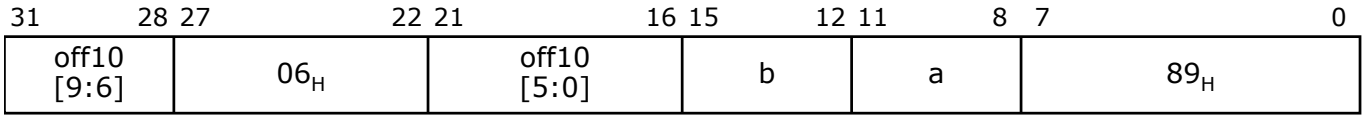
***Note**: If the source register is modified by the addressing mode, the value stored to memory is undefined.*

ST.A off18, A[a] (ABS) (Absolute Addressing Mode)



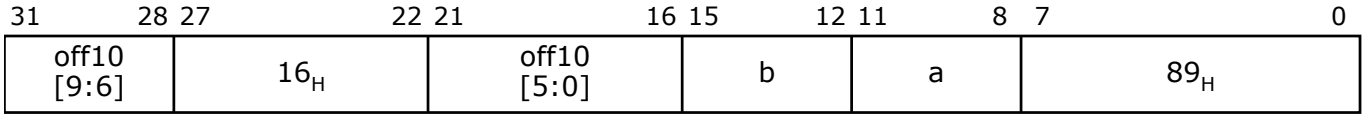

```
EA = {off18[17:14], 14b'0, off18[13:0]};
M(EA, word) = A[a];
```

ST.A A[b], off10, A[a] (BO) (Post-increment Addressing Mode)



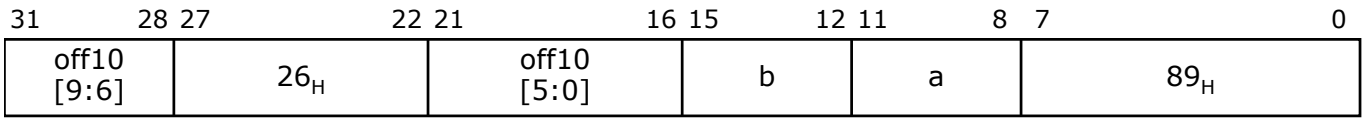

```
EA = A[b];
M(EA, word) = A[a];
A[b] = EA + sign_ext(off10);
```

ST.A A[b], off10, A[a] (BO) (Pre-increment Addressing Mode)



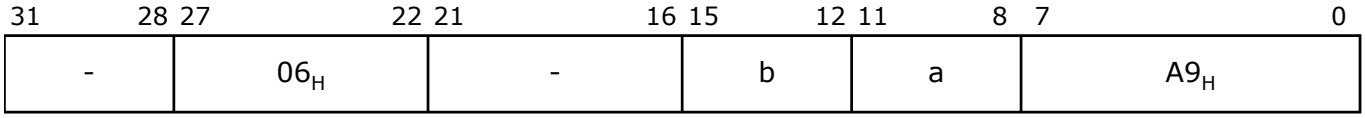

```
EA = A[b] + sign_ext(off10);
M(EA, word) = A[a];
A[b] = EA;
```

ST.A A[b], off10, A[a] (BO) (Base + Short Offset Addressing Mode)



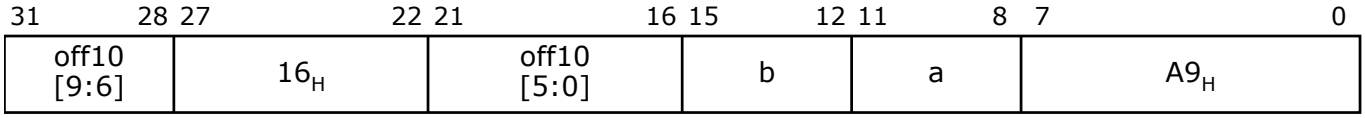

```
EA = A[b] + sign_ext(off10);
M(EA, word) = A[a];
```

ST.A P[b], A[a] (BO) (Bit-reverse Addressing Mode)



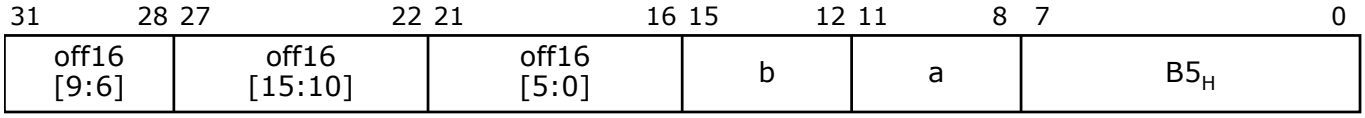

```
index = zero_ext(A[b+1][15:0]);
incr = zero_ext(A[b+1][31:16]);
EA = A[b] + index;
M(EA, word) = A[a];
new_index = reverse16(reverse16(index) + reverse16(incr));
A[b+1] = {incr[15:0], new_index[15:0]};
```

ST.A P[b], off10, A[a] (BO) (Circular Addressing Mode)



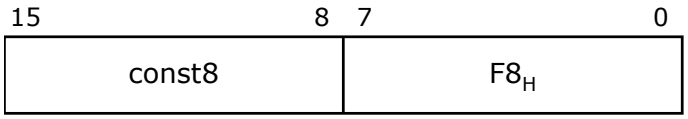

```
index = zero_ext(A[b+1][15:0]);
length = zero_ext(A[b+1][31:16]);
EA = A[b] + index;
M(EA, word) = A[a];
new_index = index + sign_ext(off10);
new_index = (new_index < 0) ? (new_index + length) : (new_index % length);
A[b+1] = {length[15:0], new_index[15:0]};
```

ST.A A[b], off16, A[a] (BOL) (Base + Long Offset Addressing Mode)



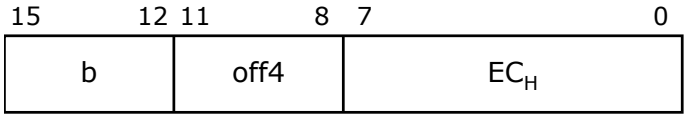

```
EA = A[b] + sign_ext(off16);
M(EA, word) = A[a];
```

ST.A A[10], const8, A[15] (SC)



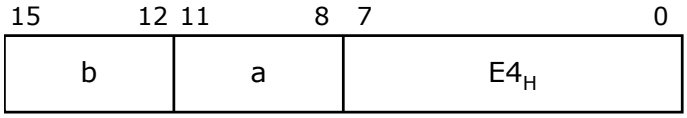

```
M(A[10] + zero_ext(4 * const8), word) = A[15];
```

ST.A A[b], off4, A[15] (SRO)



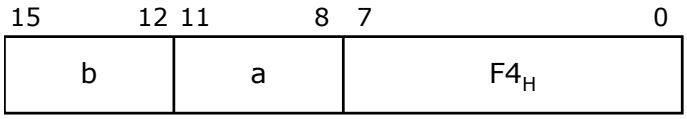

```
M(A[b] + zero_ext(4 * off4), word) = A[15];
```

ST.A A[b], A[a] (SSR) (Post-increment Addressing Mode)



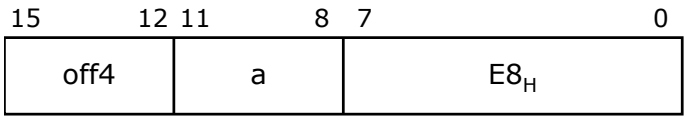

```
M(A[b], word) = A[a];
A[b] = A[b] + 4;
```

ST.A A[b], A[a] (SSR)




```
M(A[b], word) = A[a];
```

ST.A A[15], off4, A[a] (SSRO)




```
M(A[15] + zero_ext(4 * off4), 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+]4, a5
st.a    [+a0]8, a2
st.a    [a1], a0
st.a    [a6/a7+r], a3
st.a    [p6+r], a3
st.a    [a8/a9+c], a1
st.a    [p8+c], a1
```

```
st.a    [a10]2, a15
st.a    [a2]4, a15
st.a    [a2+], a5
st.a    [a2], a5
st.a    [a15]4, a2
```

See Also

ST.B, ST.D, ST.DA, ST.DD, ST.H, ST.Q, ST.W

3.1.304 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 effective address.

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 effective address.

ST.B off18, D[a] (ABS) (Absolute Addressing Mode)

31 28	27 26	25 22	21 16	15 12	11 8	7 0
off18 [9:6]	0_H	off18 [13:10]	off18 [5:0]	off18 [17:14]	a	25_H

```
EA = {off18[17:14], 14b'0, off18[13:0]};
M(EA, byte) = D[a][7:0];
```

ST.B A[b], off10, D[a] (BO) (Post-increment Addressing Mode)

31 28	27 22	21 16	15 12	11 8	7 0
off10 [9:6]	00_H	off10 [5:0]	b	a	89_H

```
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)

31 28	27 22	21 16	15 12	11 8	7 0
off10 [9:6]	10_H	off10 [5:0]	b	a	89_H

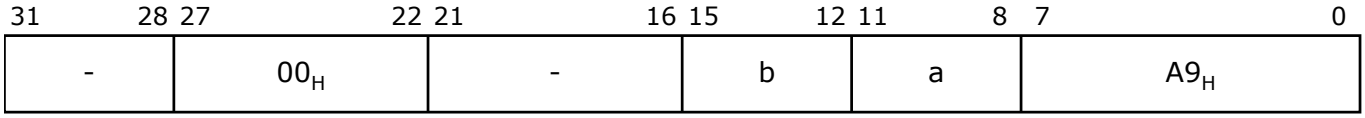
```
EA = A[b] + sign_ext(off10);
M(EA, byte) = D[a][7:0];
A[b] = EA;
```

ST.B 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]	20_H	off10 [5:0]	b	a	89_H

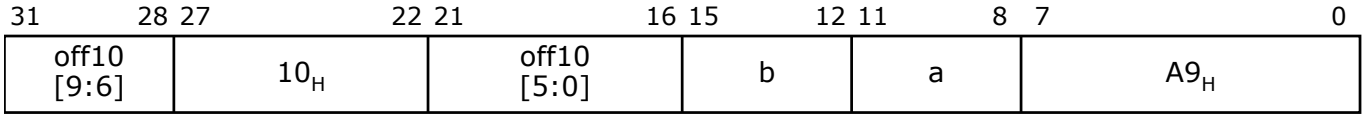
```
EA = A[b] + sign_ext(off10);
M(EA, byte) = D[a][7:0];
```

ST.B P[b], D[a] (BO) (Bit-reverse Addressing Mode)



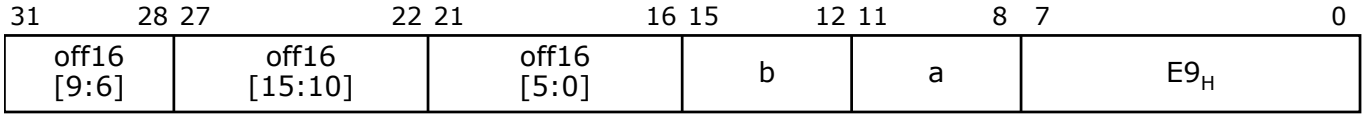

```
index = zero_ext(A[b+1][15:0]);
incr = zero_ext(A[b+1][31:16]);
EA = A[b] + index;
M(EA, byte) = D[a][7:0];
new_index = reverse16(reverse16(index) + reverse16(incr));
A[b+1] = {incr[15:0], new_index[15:0]};
```

ST.B P[b], off10, D[a] (BO) (Circular Addressing Mode)



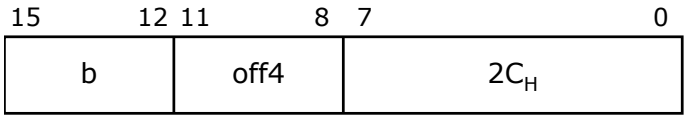

```
index = zero_ext(A[b+1][15:0]);
length = zero_ext(A[b+1][31:16]);
EA = A[b] + index;
M(EA, byte) = D[a][7:0];
new_index = index + sign_ext(off10);
new_index = (new_index < 0) ? (new_index + length) : (new_index % length);
A[b+1] = {length[15:0], new_index[15:0]};
```

ST.B A[b], off16, D[a] (BOL) (Base + Long Offset Addressing Mode)



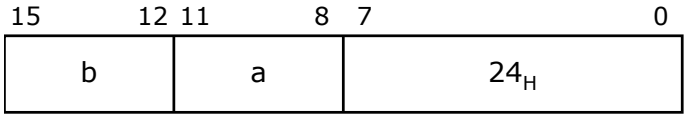

```
EA = A[b] + sign_ext(off16);
M(EA, byte) = D[a][7:0];
```

ST.B A[b], off4, D[15] (SRO)



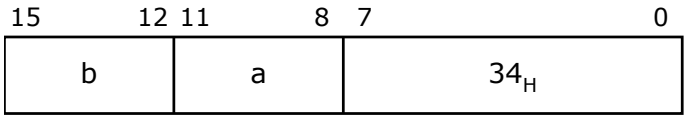

```
M(A[b] + zero_ext(off4), byte) = D[15][7:0];
```

ST.B A[b], D[a] (SSR) (Post-increment Addressing Mode)



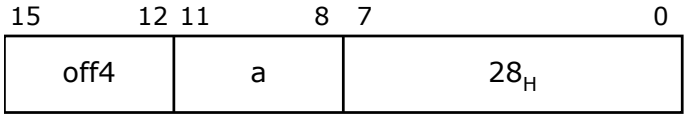
```
M(A[b], byte) = D[a][7:0];
A[b] = A[b] + 1;
```

ST.B A[b], D[a] (SSR)




```
M(A[b], byte) = D[a][7:0];
```

ST.B A[15], off4, D[a] (SSRO)




```
M(A[15] + zero_ext(off4), 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    [+a0]2, d0
st.b    [a3]+24, d2
st.b    [a0/a1+r], d0
st.b    [p0+r], d0
st.b    [a0/a1+c], d0
st.b    [p0+c], d0
```

```
st.b    [a0]4, d15
st.b    [a0+], d2
st.b    [a0], d2
st.b    [a15]4, d0
```

See Also

ST.A, ST.D, ST.DA, ST.DD, ST.H, ST.Q, ST.W

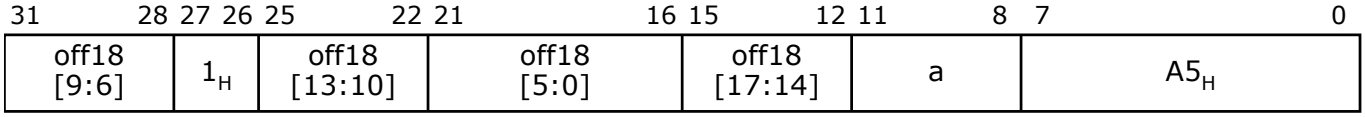
3.1.305 ST.D

Store Double-word

Description

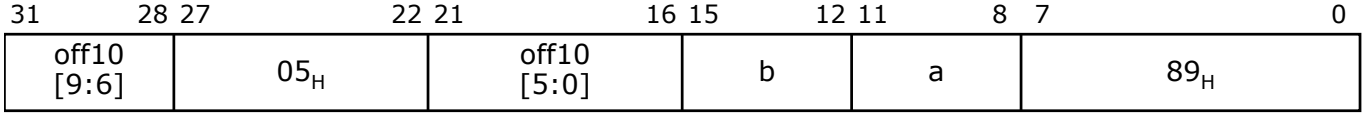
Store the value in the extended data register pair E[a] to the memory location specified by the effective address. The value in the even register D[a] is stored in the least-significant memory word, and the value in the odd register (D[a+1]) is stored in the most-significant memory word.

ST.D off18, E[a] (ABS) (Absolute Addressing Mode)



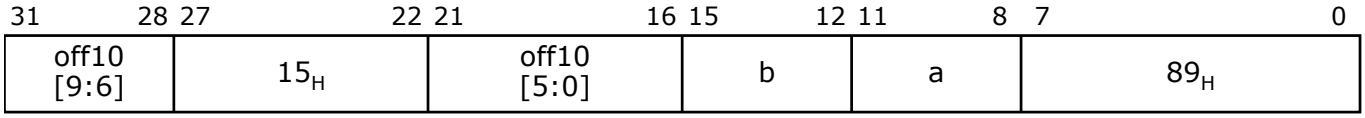

```
EA = {off18[17:14], 14b'0, off18[13:0]};
M(EA, doubleword) = E[a];
```

ST.D A[b], off10, E[a] (BO) (Post-increment Addressing Mode)



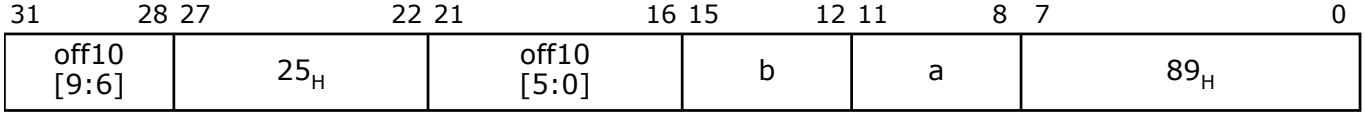

```
EA = A[b];
M(EA, doubleword) = E[a];
A[b] = EA + sign_ext(off10);
```

ST.D A[b], off10, E[a] (BO) (Pre-increment Addressing Mode)



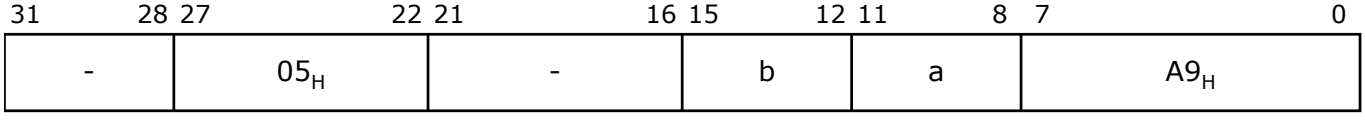

```
EA = A[b] + sign_ext(off10);
M(EA, doubleword) = E[a];
A[b] = EA;
```

ST.D A[b], off10, E[a] (BO) (Base + Short Offset Addressing Mode)




```
EA = A[b] + sign_ext(off10);
M(EA, doubleword) = E[a];
```

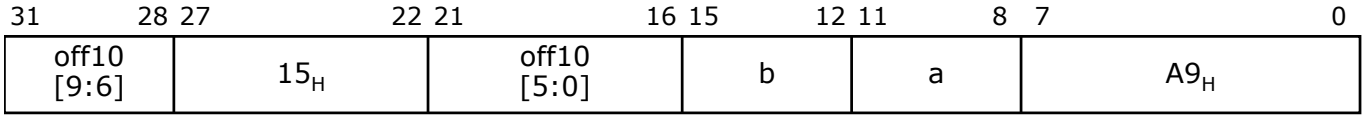
ST.D P[b], E[a] (BO) (Bit-reverse Addressing Mode)




```
index = zero_ext(A[b+1][15:0]);
```

```
incr = zero_ext(A[b+1][31:16]);
EA = A[b] + index;
M(EA, 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)




```
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]};
```

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.d    [a0+]8, e0
st.d    [+a0]8, e0
st.d    [a3]+24, e2
st.d    [a0/a1+r], e0
st.d    [p0+r], e0
st.d    [a0/a1+c], e0
st.d    [p0+c], e0
```

See Also

ST.A, ST.B, ST.DA, ST.DD, ST.H, ST.Q, ST.W

3.1.306 ST.DA

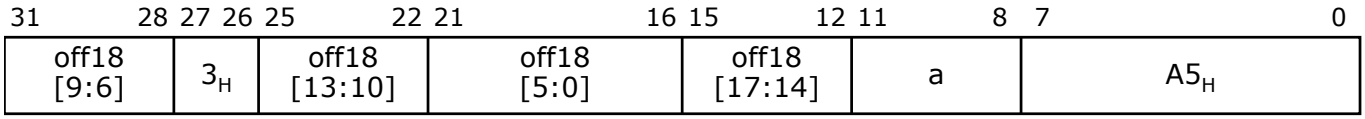
Store Double-word from Address Registers

Description

Store the value in the address register pair P[a] to the memory location specified by the effective address. 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.

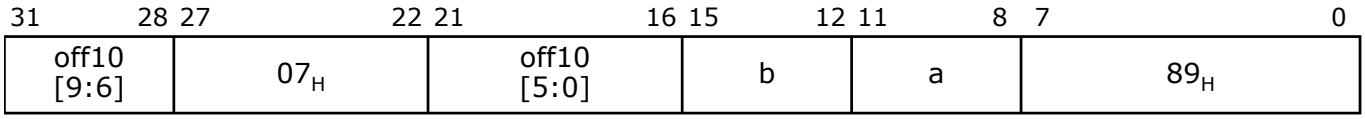
***Note**: If the source register is modified by the addressing mode, the value stored to memory is undefined.*

ST.DA off18, P[a] (ABS) (Absolute Addressing Mode)



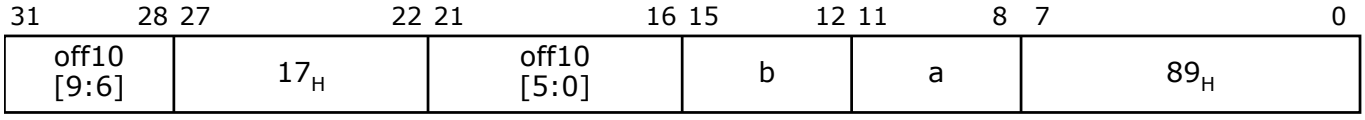

```
EA = {off18[17:14], 14b'0, off18[13:0]};
M(EA, doubleword) = P[a];
```

ST.DA A[b], off10, P[a] (BO) (Post-increment Addressing Mode)



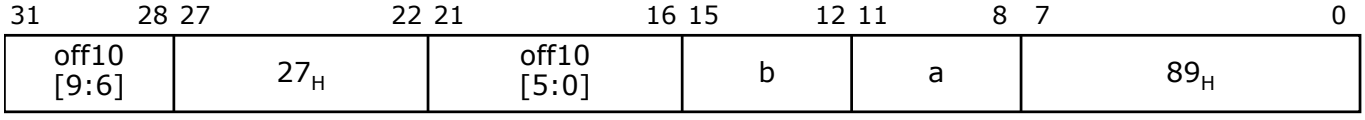

```
EA = A[b];
M(EA, doubleword) = P[a];
A[b] = EA + sign_ext(off10);
```

ST.DA A[b], off10, P[a] (BO) (Pre-increment Addressing Mode)



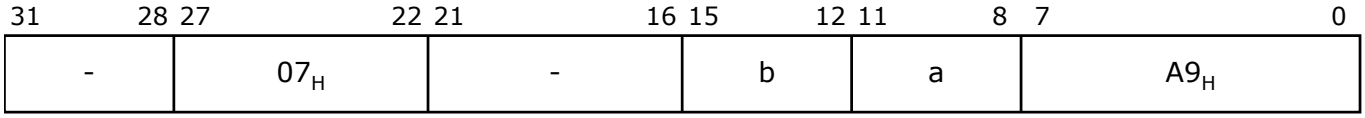

```
EA = A[b] + sign_ext(off10);
M(EA, doubleword) = P[a];
A[b] = EA;
```

ST.DA A[b], off10, P[a] (BO) (Base + Short Offset Addressing Mode)



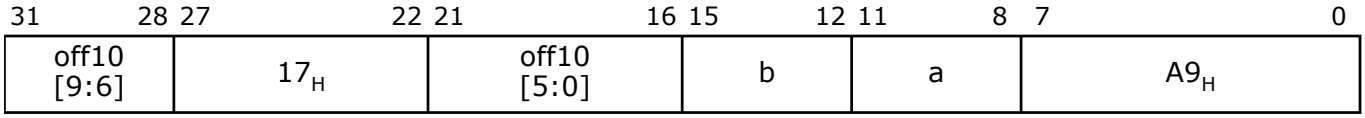

```
EA = A[b] + sign_ext(off10);
M(EA, doubleword) = P[a];
```

ST.DA P[b], P[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, 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)




```
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]};
```

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    _savedPointerBuffer, a0/a1
st.da    _savedPointerBuffer, p0
st.da    [a0+]8, a0/a1
st.da    [a0+]8, p0
st.da    [+a0]8, a2/a3
st.da    [+a0]8, p2
st.da    [a6]+24, a2/a3
st.da    [a6]+24, p2
st.da    [a0/a1+r], a2/a3
st.da    [p0+r], p2
```

```
st.da    [a0/a1+c], a2/a3
st.da    [p0+c], p2
```

See Also

ST.A, ST.B, ST.D, ST.DD, ST.H, ST.Q, ST.W

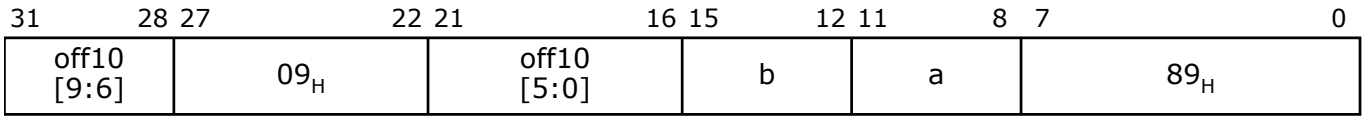
3.1.307 ST.DD

Store Double Double

Description

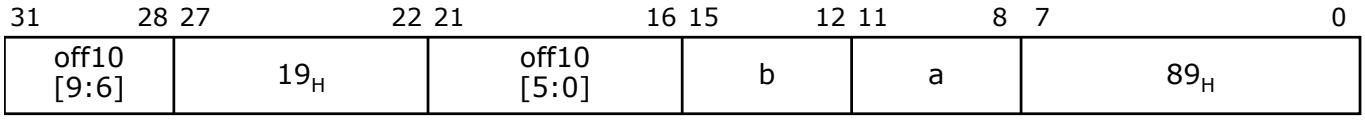
Store the value in the quad data register Q[a] to the memory location specified by the effective address.
An operation with Q[a] where a != [0,4,8,12] will cause and OPD trap

ST.DD A[b], off10, Q[a] (BO) (Post-increment Addressing Mode)



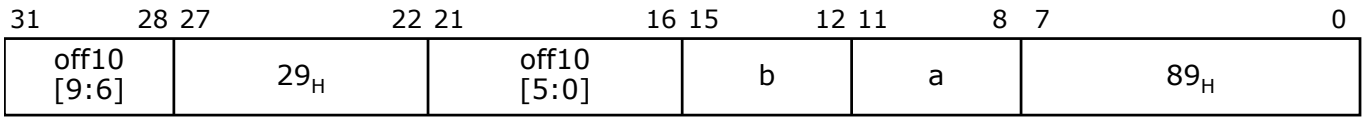

```
EA = A[b];
M(EA, quadword) = Q[a];
A[b] = EA + sign_ext(off10);
```

ST.DD A[b], off10, Q[a] (BO) (Pre-increment Addressing Mode)




```
EA = A[b] + sign_ext(off10);
M(EA, quadword) = Q[a];
A[b] = EA;
```

ST.DD A[b], off10, Q[a] (BO) (Base + Short Offset Addressing Mode)




```
EA = A[b] + sign_ext(off10);
M(EA, quadword) = Q[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.dd    [a0+]8, q8; Store 128-bit
```

See Also

ST.A, ST.B, ST.D, ST.DA, ST.Q, ST.W, LD.DD

3.1.308 ST.H

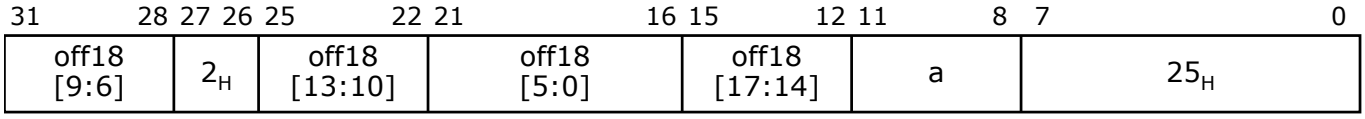
Store Half-word

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 effective address.

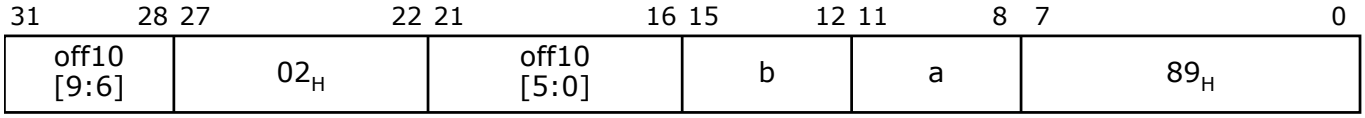
Store the half-word value in the 16 least-significant bits of either data register D[a] (instruction format or D[15] to the half-word memory location specified by the effective address.

ST.H off18, D[a] (ABS) (Absolute Addressing Mode)



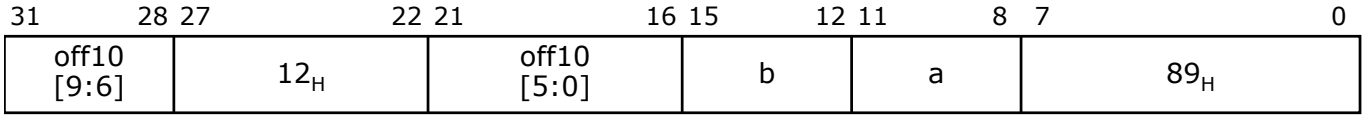

```
EA = {off18[17:14], 14b'0, off18[13:0]};
M(EA, halfword) = D[a][15:0];
```

ST.H A[b], off10, D[a] (BO) (Post-increment Addressing Mode)



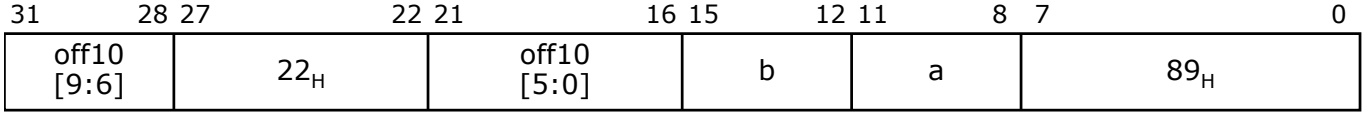

```
EA = A[b];
M(EA, halfword) = D[a][15:0];
A[b] = EA + sign_ext(off10);
```

ST.H A[b], off10, D[a] (BO) (Pre-increment Addressing Mode)



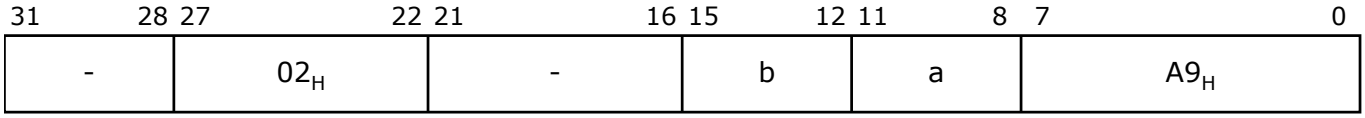

```
EA = A[b] + sign_ext(off10);
M(EA, halfword) = D[a][15:0];
A[b] = EA;
```

ST.H A[b], off10, D[a] (BO) (Base + Short Offset Addressing Mode)



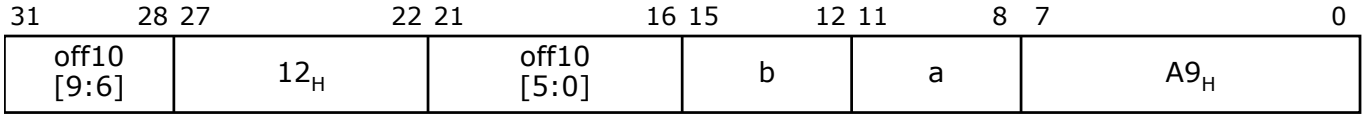

```
EA = A[b] + sign_ext(off10);
M(EA, halfword) = D[a][15:0];
```

ST.H P[b], D[a] (BO) (Bit-reverse Addressing Mode)



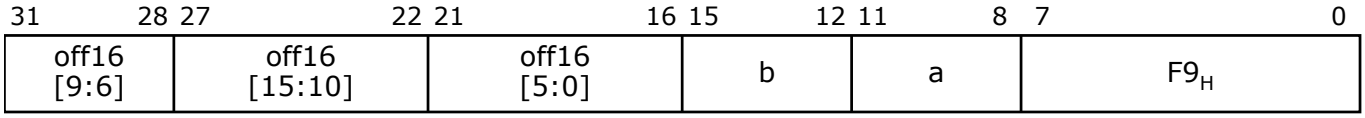

```
index = zero_ext(A[b+1][15:0]);
incr = zero_ext(A[b+1][31:16]);
EA = A[b] + index;
M(EA, halfword) = D[a][15:0];
new_index = reverse16(reverse16(index) + reverse16(incr));
A[b+1] = {incr[15:0], new_index[15:0]};
```

ST.H P[b], off10, D[a] (BO) (Circular Addressing Mode)



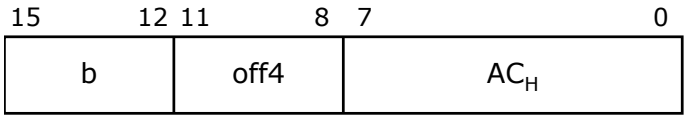

```
index = zero_ext(A[b+1][15:0]);
length = zero_ext(A[b+1][31:16]);
EA = A[b] + index;
M(EA, halfword) = D[a][15:0];
new_index = index + sign_ext(off10);
new_index = (new_index < 0) ? (new_index + length) : (new_index % length);
A[b+1] = {length[15:0], new_index[15:0]};
```

ST.H A[b], off16, D[a] (BOL) (Base + Long Offset Addressing Mode)



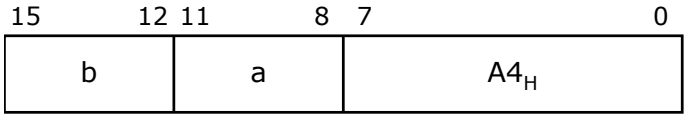

```
EA = A[b] + sign_ext(off16);
M(EA, halfword) = D[a][15:0];
```

ST.H A[b], off4, D[15] (SRO)



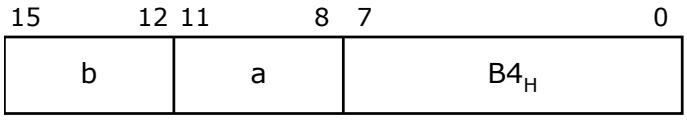

```
M(A[b] + zero_ext(2 * off4), half-word) = D[15][15:0];
```

ST.H A[b], D[a] (SSR) (Post-increment Addressing Mode)



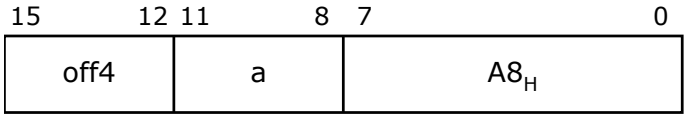
```
M(A[b], half-word) = D[a][15:0];
A[b] = A[b] + 2;
```

ST.H A[b], D[a] (SSR)




```
M(A[b], half-word) = D[a][15:0];
```

ST.H A[15], off4, D[a] (SSRO)




```
M(A[15] + zero_ext(2 * off4), 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+]2, d0
st.h    [+a0]2, d0
st.h    [a3]+24, d2
st.h    [a0/a1+r], d0
st.h    [p0+r], d0
st.h    [a0/a1+c], d0
st.h    [p0+c], d0
```

```
st.h    [a0]4, d15
st.h    [a0+], d2
st.h    [a0], d2
st.h    [a15]4, d0
```

See Also

ST.A, ST.B, ST.D, ST.DA, ST.Q, ST.W

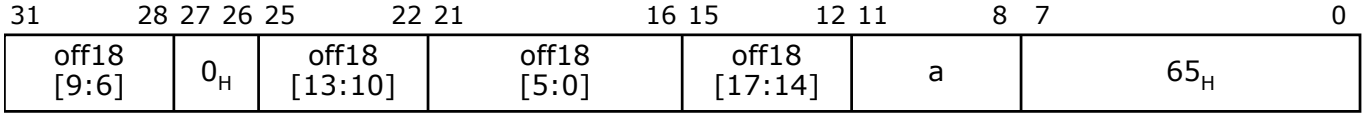
3.1.309 ST.Q

Store Half-word Signed Fraction

Description

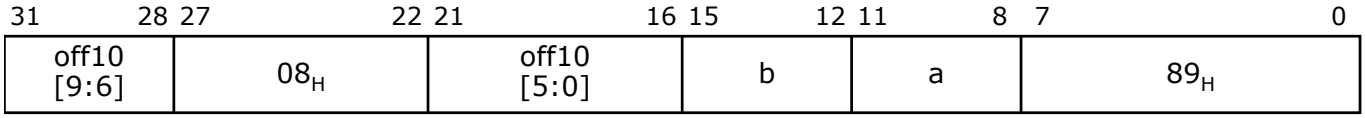
Store the value in the most-significant half-word of data register D[a] to the memory location specified by the effective address.

ST.Q off18, D[a] (ABS) (Absolute Addressing Mode)



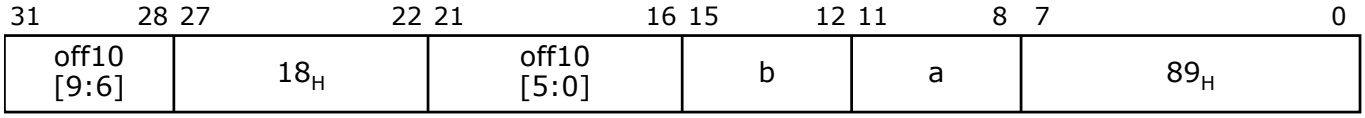

```
EA = {off18[17:14],14b'0,off18[13:0]};
M(EA, halfword) = D[a][31:16];
```

ST.Q A[b], off10, D[a] (BO) (Post-increment Addressing Mode)



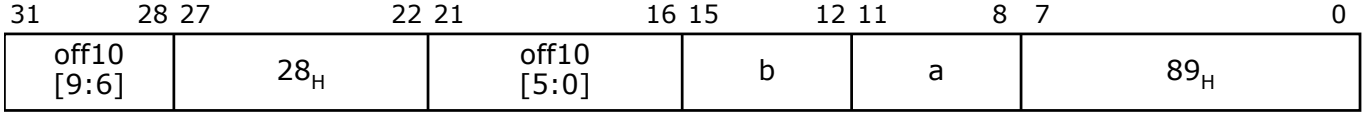

```
EA = A[b];
M(EA, halfword) = D[a][31:16];
A[b] = EA + sign_ext(off10);
```

ST.Q A[b], off10, D[a] (BO) (Pre-increment Addressing Mode)



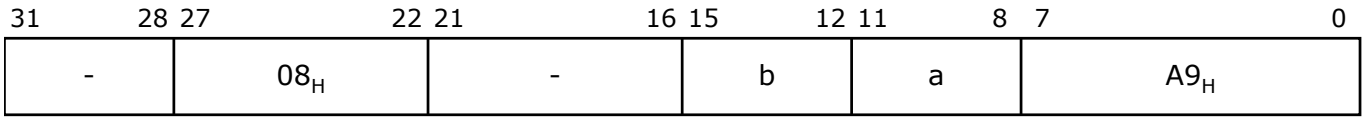

```
EA = A[b] + sign_ext(off10);
M(EA, halfword) = D[a][31:16];
A[b] = EA;
```

ST.Q A[b], off10, D[a] (BO) (Base + Short Offset Addressing Mode)




```
EA = A[b] + sign_ext(off10);
M(EA, halfword) = D[a][31:16];
```

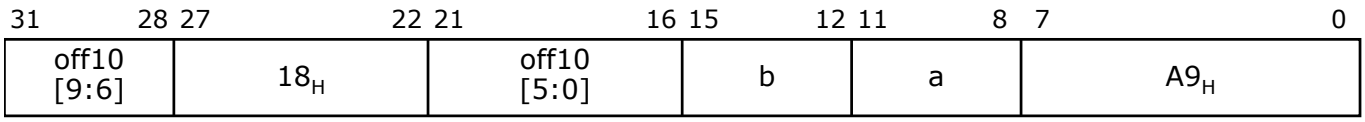
ST.Q P[b], D[a] (BO) (Bit-reverse Addressing Mode)




```
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][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)




```
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]};
```

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]2, d0
st.q    [a3]+24, d2
st.q    [a0/a1+r], d0
st.q    [p0+r], d0
st.q    [a0/a1+c], d0
st.q    [p0+c], d0
```

See Also

ST.A, ST.B, ST.D, ST.DA, ST.DD, ST.H, ST.W

3.1.310 ST.T

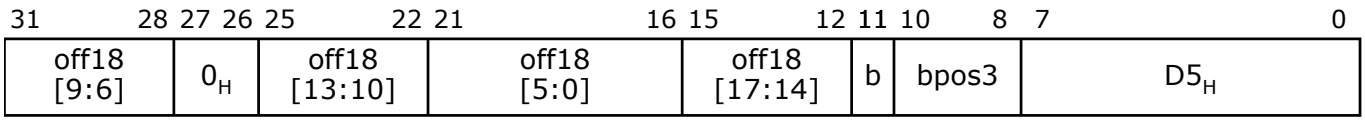
Store Bit

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 semaphores.

The M(EA, byte) read and the M(EA, byte) write are atomic.

ST.T off18, bpos3, b (ABSB)




```
EA = {off18[17:14], 14'b0, off18[13:0]};
tmp = M(EA, byte);
M(EA, byte) = (tmp 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, SWAPMSK.W, CMPSWAP.W

3.1.311 ST.W

Store Word

Description

Store the word value in data register D[a] to the memory location specified by the effective address.

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 effective address.

ST.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]	0_H	off18 [13:10]	off18 [5:0]	off18 [17:14]	a	$A5_H$

```
EA = {off18[17:14], 14b'0, off18[13:0]};
M(EA, word) = D[a];
```

ST.W A[b], off10, D[a] (BO) (Post-increment Addressing Mode)

31 28	27 22	21 16	15 12	11 8	7 0
off10 [9:6]	04_H	off10 [5:0]	b	a	89_H

```
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)

31 28	27 22	21 16	15 12	11 8	7 0
off10 [9:6]	14_H	off10 [5:0]	b	a	89_H

```
EA = A[b] + sign_ext(off10);
M(EA, word) = D[a];
A[b] = EA;
```

ST.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]	24_H	off10 [5:0]	b	a	89_H

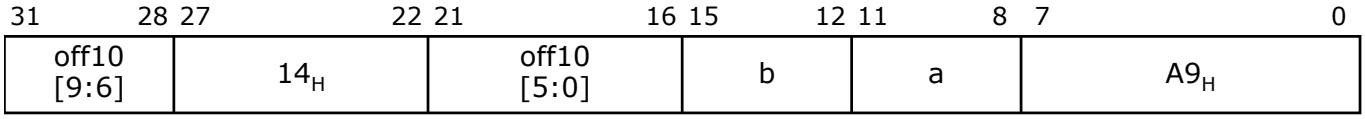
```
EA = A[b] + sign_ext(off10);
M(EA, word) = D[a];
```

ST.W P[b], D[a] (BO) (Bit-reverse Addressing Mode)

31 28	27 22	21 16	15 12	11 8	7 0
-	04_H	-	b	a	$A9_H$

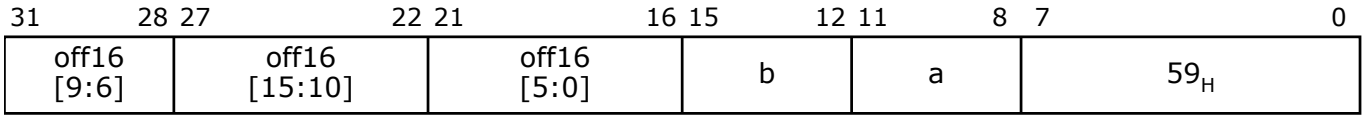
```
index = zero_ext(A[b+1][15:0]);
incr = zero_ext(A[b+1][31:16]);
EA = A[b] + index;
M(EA, word) = D[a];
new_index = reverse16(reverse16(index) + reverse16(incr));
A[b+1] = {incr[15:0], new_index[15:0]};
```

ST.W P[b], off10, D[a] (BO) (Circular Addressing Mode)



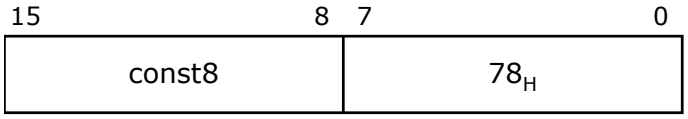

```
index = zero_ext(A[b+1][15:0]);
length = zero_ext(A[b+1][31:16]);
EA0 = A[b] + index;
EA2 = A[b] + (index + 2) % length;
M(EA0, halfword) = D[a][15:0];
M(EA2, halfword) = D[a][31:16];
new_index = index + sign_ext(off10);
new_index = (new_index < 0) ? (new_index + length) : (new_index % length);
A[b+1] = {length[15:0], new_index[15:0]};
```

ST.W A[b], off16, D[a] (BOL) (Base + Long Offset Addressing Mode)



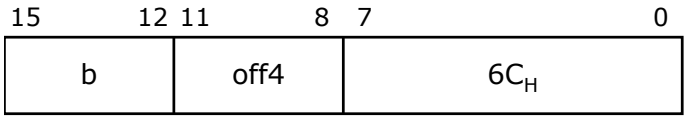

```
EA = A[b] + sign_ext(off16);
M(EA, word) = D[a];
```

ST.W A[10], const8, D[15] (SC)



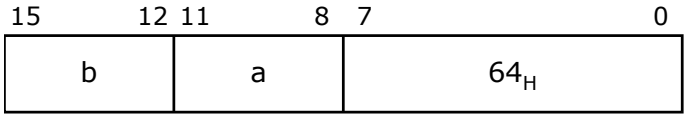

```
M(A[10] + zero_ext(4 * const8), word) = D[15];
```

ST.W A[b], off4, D[15] (SRO)



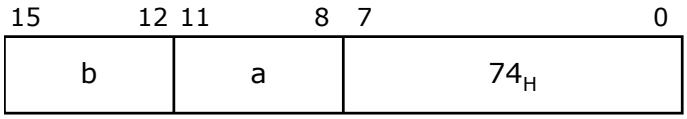

```
M(A[b] + zero_ext(4 * off4), word) = D[15];
```

ST.W A[b], D[a] (SSR) (Post-increment Addressing Mode)



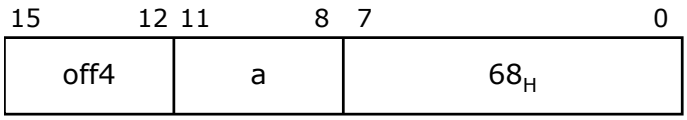

```
M(A[b], word) = D[a];
A[b] = A[b] + 4;
```

ST.W A[b], D[a] (SSR)




```
M(A[b], word) = D[a];
```

ST.W A[15], off4, D[a] (SSRO)




```
M(A[15] + zero_ext(4 * off4), 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+]4, d0
st.w    [+a0]4, d0
st.w    [a3]+24, d2
st.w    [a0/a1+r], d0
st.w    [p0+r], d0
st.w    [a0/a1+c], d0
st.w    [p0+c], d0
```

```
st.w    [a10]4, d15
st.w    [a0]4, d15
st.w    [a0+], d2
st.w    [a0], d2
st.w    [a15]4, d0
```

See Also

ST.A, ST.B, ST.D, ST.DA, ST.DD, ST.H, ST.Q

3.1.312 STLCX

Store Lower Context

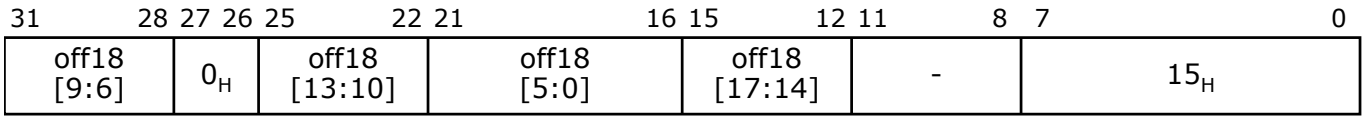
Description

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 effective address. 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 effective address must be aligned on a 16-word boundary.*

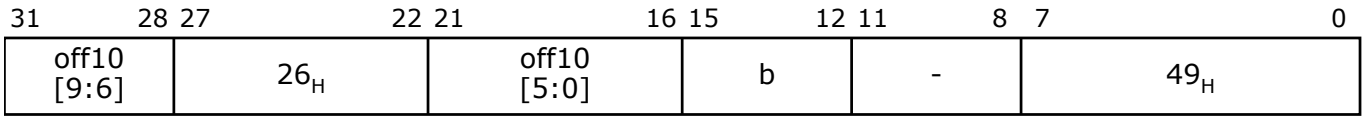
***Note**: This instruction may not be used to access peripheral space.*

STLCX off18 (ABS) (Absolute Addressing Mode)




```
EA = {off18[17:14], 14b'0, off18[13:0]};
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]};
```

STLCX A[b], off10 (BO) (Base + Short Offset Addressing Mode)




```
EA = A[b] + sign_ext(off10)[9:0]};
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]};
```

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

```
stlcx    a0
```

See Also

LDLCX, LDUCX, RSLCX, STUCX, SVLCX, BISR

3.1.313 STUCX

Store Upper Context

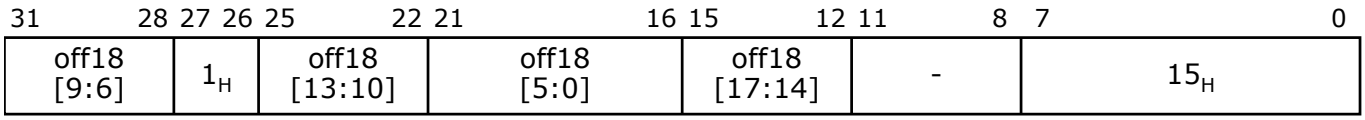
Description

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 effective address. 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 effective address must be aligned on a 16-word boundary.*

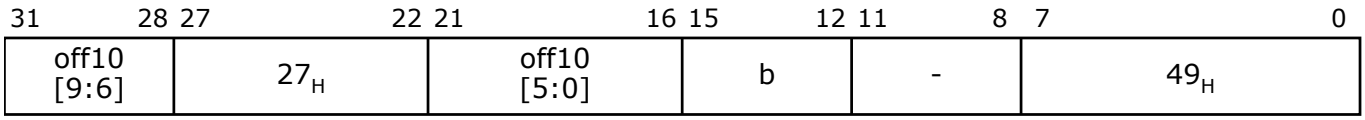
***Note**: This instruction may not be used to access peripheral space.*

STUCX off18 (ABS) (Absolute Addressing Mode)




```
EA = {off18[17:14], 14b'0, off18[13:0]};
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]};
```

STUCX A[b], off10 (BO) (Base + Short Offset Addressing Mode)




```
EA = A[b] + sign_ext(off10)[9:0]};
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]};
```

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

```
stucx    a0
```

See Also

LDLCX, LDUCX, RSLCX, STLCX, SVLCX, STUCX

3.1.314 SUB

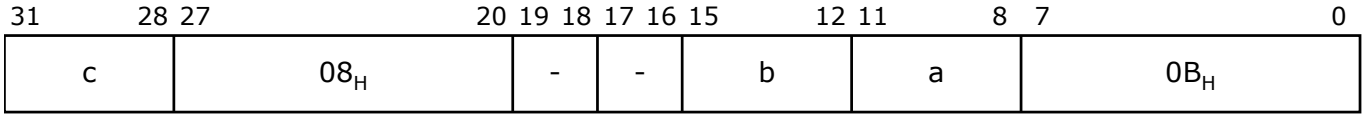
Subtract

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 signed integers.

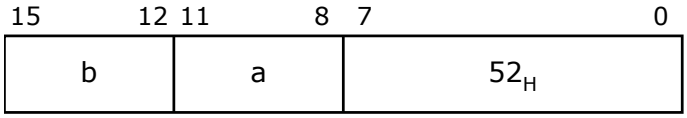
Subtract the contents of data register D[b] from the contents of either data register D[a] or D[15] and put the result in either data register D[a] or D[15]. The operands are treated as 32-bit signed integers.

SUB D[c], D[a], D[b] (RR)



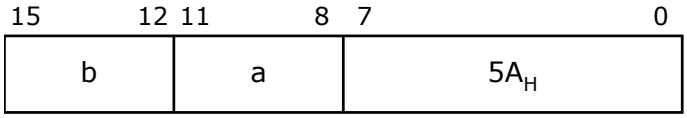

```
result = D[a] - D[b];
D[c] = result[31:0];
```

SUB D[a], D[15], D[b] (SRR)



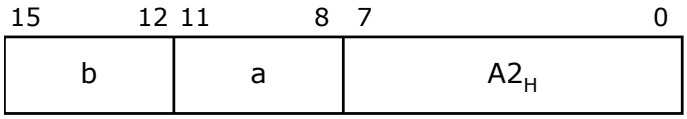

```
result = D[15] - D[b];
D[a] = result[31:0];
```

SUB D[15], D[a], D[b] (SRR)




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

SUB D[a], D[b] (SRR)




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

Status Flags

C	Not set by this instruction.
V	overflow = (result > 7FFFFFFF$_H$) OR (result < -80000000$_H$); 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, d15, d2
sub    d15, d1, d2
sub    d1, d2
```

See Also

SUBS, SUBS.U, SUBX, SUBC

3.1.315 SUB.A

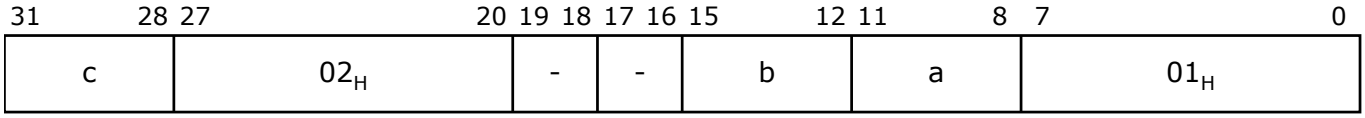
Subtract Address

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].

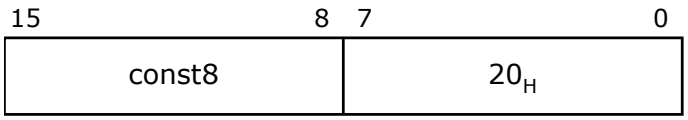
Decrement the Stack Pointer (A[10]) by the zero-extended value of const8 (a range of 0 through to 255).

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

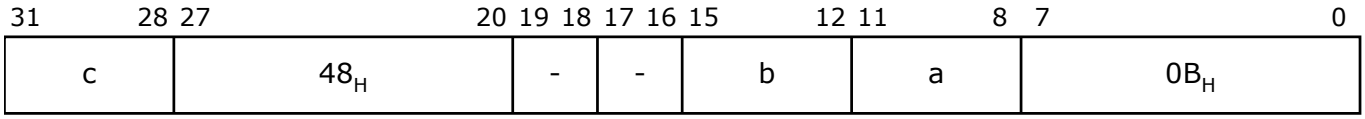
3.1.316 SUB.B

Subtract Packed Byte

Description

Subtract the contents of each byte of data register D[b] from the contents of data register D[a]. Put the result in each corresponding byte of data register D[c].

SUB.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]};
```

Status Flags

C	Not set by this instruction.

V	ov_byte3 = (result_byte3 > $7F_H$) OR (result_byte3 < -80_H);
	ov_byte2 = (result_byte2 > $7F_H$) OR (result_byte2 < -80_H);
	ov_byte1 = (result_byte1 > $7F_H$) OR (result_byte1 < -80_H);
	ov_byte0 = (result_byte0 > $7F_H$) OR (result_byte0 < -80_H);
	overflow = ov_byte3 OR ov_byte2 OR ov_byte1 OR ov_byte0;
	if (overflow) then PSW.V = 1 else PSW.V = 0;

SV	if (overflow) then PSW.SV = 1 else PSW.SV = PSW.SV;

AV	aov_byte3 = result_byte3[7] ^ result_byte3[6];
	aov_byte2 = result_byte2[7] ^ result_byte2[6];
	aov_byte1 = result_byte1[7] ^ result_byte1[6];
	aov_byte0 = result_byte0[7] ^ result_byte0[6];
	advanced_overflow = aov_byte3 OR aov_byte2 OR aov_byte1 OR aov_byte0;
	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.b    d3, d1, d2
```

See Also

SUB.H, SUBS.H, SUBS.HU

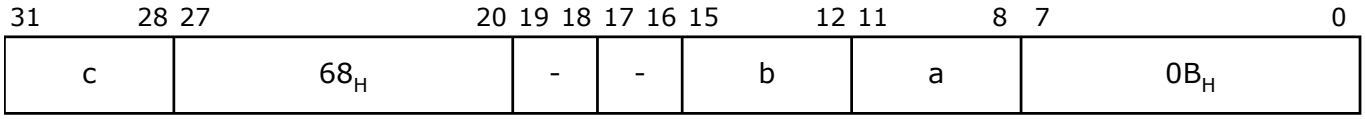
3.1.317 SUB.H

Subtract Packed Half-word

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].

SUB.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 this instruction.

V	ov_halfword1 = (result_halfword1 > $7FFF_H$) OR (result_halfword1 < -8000_H);
	ov_halfword0 = (result_halfword0 > $7FFF_H$) OR (result_halfword0 < -8000_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[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

```
sub.h    d3, d1, d2
```

See Also

SUB.B, SUBS.H, SUBS.HU

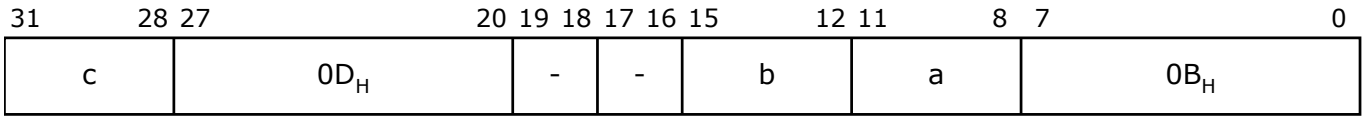
3.1.318 SUBC

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 signed 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$_H$) OR (result < -80000000$_H$); 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

ADDC, SUB, SUBS, SUBS.U, SUBX

3.1.319 SUBS

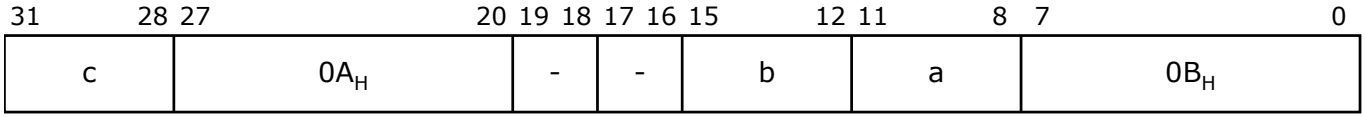
Subtract with Saturation

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 32-bit signed integers, with saturation on signed overflow.

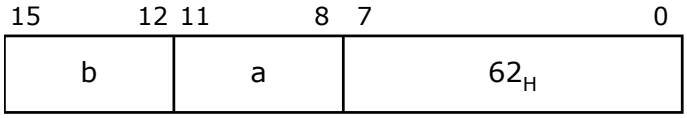
Subtract the contents of data register D[b] from the contents of data register D[a] and put the result in data register D[a]. The operands are treated as 32-bit signed integers, with saturation on signed overflow.

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);
```

Status Flags

C	Not set by this instruction.
V	overflow = (result > 7FFFFFFF$_H$) OR (result < -80000000$_H$); 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    d3, d1
```

See Also

SUB, SUBS.U, SUBX, SUBC

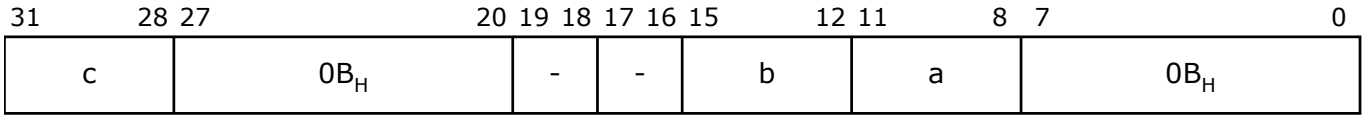
3.1.320 SUBS.U

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 32-bit unsigned integers, with saturation on unsigned overflow.

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 this instruction.
V	overflow = (result > $FFFFFFFF_H$) OR (result < 00000000_H); 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.u    d3, d1, d2
```

See Also

SUB, SUBS, SUBX, SUBC

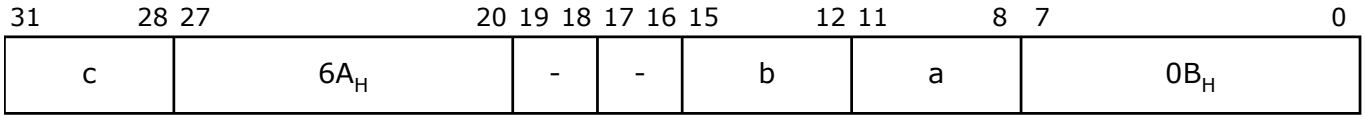
3.1.321 SUBS.H

Subtract Packed Half-word 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 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)};
```

Status Flags

C	Not set by this instruction.

V	ov_halfword1 = (result_halfword1 > $7FFF_H$) OR (result_halfword1 < -8000_H);
	ov_halfword0 = (result_halfword0 > $7FFF_H$) OR (result_halfword0 < -8000_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[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
```

See Also

SUB.B, SUB.H, SUBS.HU

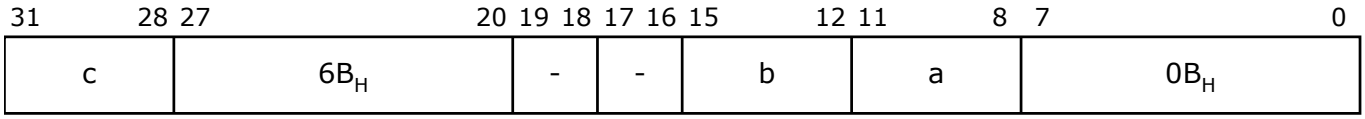
3.1.322 SUBS.HU

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 unsigned overflow.

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 this instruction.

V	ov_halfword1 = (result_halfword1 > $FFFF_H$) OR (result_halfword1 < 0000_H);
	ov_halfword0 = (result_halfword0 > $FFFF_H$) OR (result_halfword0 < 0000_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[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.hu    d3, d1, d2
```

See Also

SUB.B, SUB.H, SUBS.H

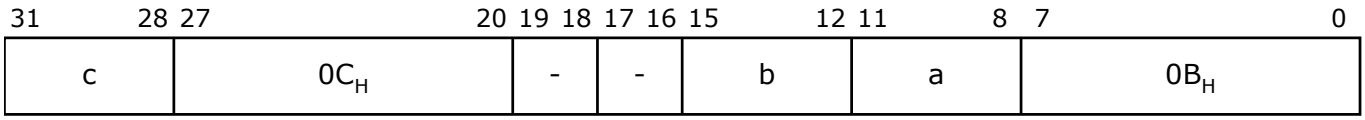
3.1.323 SUBX

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 signed 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$_H$) OR (result < -80000000$_H$); 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

ADDX, SUB, SUBC, SUBS, SUBS.U

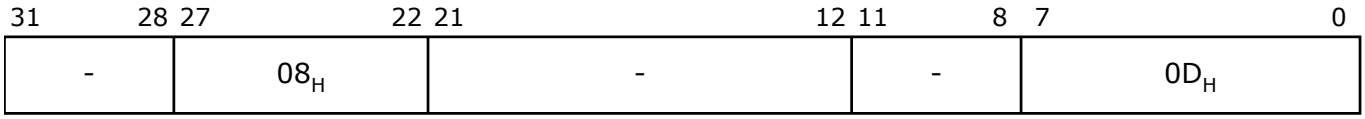
3.1.324 SVLCX

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)




```
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

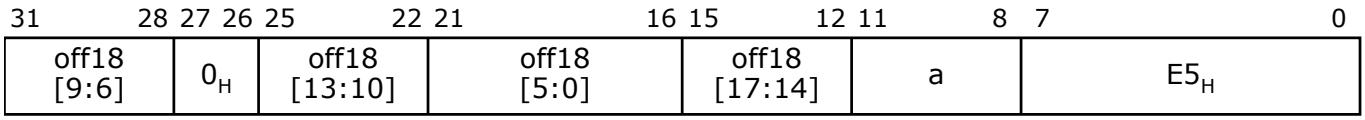
3.1.325 SWAP.W

Swap with Data Register

Description

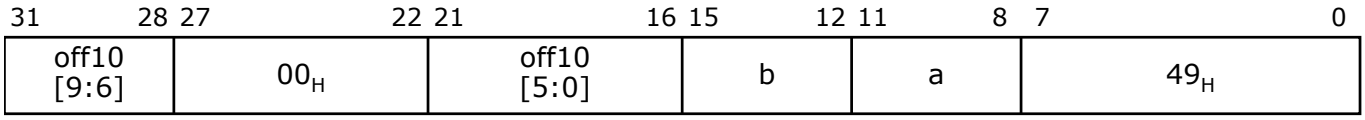
Swap the contents of data register D[a] and the memory word specified by the effective address.
The M(EA, word) read and the M(EA, word) write are atomic.

SWAP.W off18, D[a] (ABS) (Absolute Addressing Mode)



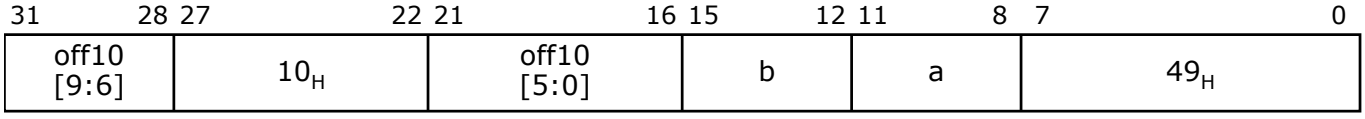

```
EA = {off18[17:14], 14b'0, off18[13:0]};
tmp = M(EA, word);
M(EA, word) = D[a];
D[a] = tmp[31:0];
```

SWAP.W A[b], off10, D[a] (BO) (Post-increment Addressing Mode)



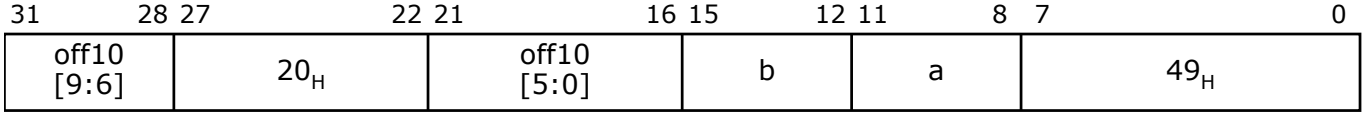

```
EA = A[b];
tmp = M(EA, word);
M(EA, word) = D[a];
D[a] = tmp[31:0];
A[b] = EA + sign_ext(off10);
```

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;
```

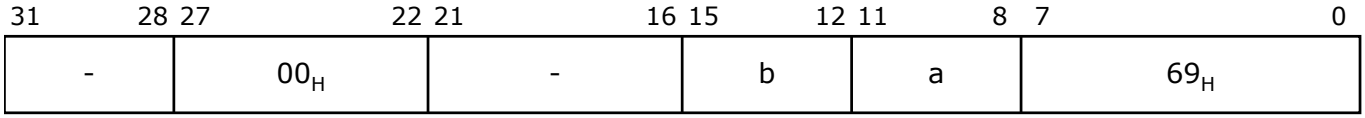
SWAP.W A[b], off10, D[a] (BO) (Base + Short Offset Addressing Mode)




```
EA = A[b] + sign_ext(off10);
tmp = M(EA, word);
M(EA, word) = D[a];
```

```
D[a] = tmp[31:0];
```

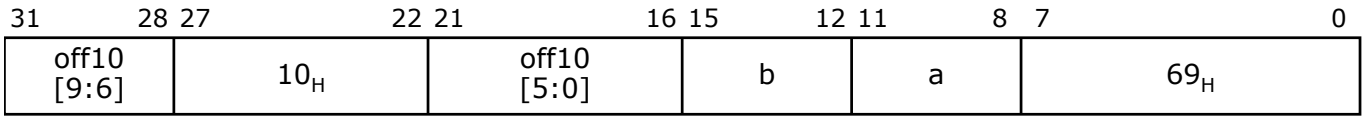
SWAP.W P[b], D[a] (BO) (Bit-reverse Addressing Mode)



31 – 28	27 – 22	21 – 16	15 – 12	11 – 8	7 – 0
-	00_H	-	b	a	69_H

```
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)



31 – 28	27 – 22	21 – 16	15 – 12	11 – 8	7 – 0
off10 [9:6]	10_H	off10 [5:0]	b	a	69_H

```
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]};
```

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

```
swap.w    [a0+]2, d0
swap.w    [+a0]2, d0
swap.w    [a3]+24, d2
swap.w    [a0/a1+r], d0
swap.w    [p0+r], d0
swap.w    [a0/a1+c], d0
swap.w    [p0+c], d0
```

See Also

ST.T, LDMST, SWAPMSK.W, CMPSWAP.W

3.1.326 SWAPMSK.W

Swap under Mask

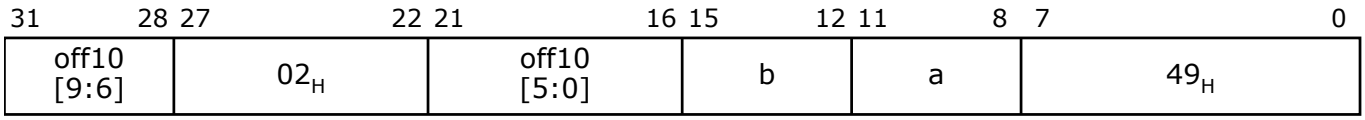
Description

The SWAPMSK.W instruction enables individual bits or bytes to be used as semaphore. Up to 32 semaphores (bits) can be tested in parallel.

SWAPMSK.W uses an Extended register to provide both mask and swap data. It swaps through a mask the contents of E[a][31:0] with the contents of the memory word specified by the effective address. Only those bits are swapped where the corresponding bits in the mask E[a][63:32] are set. Register D[a] is unconditionally updated with the contents of the memory word specified by the effective address.

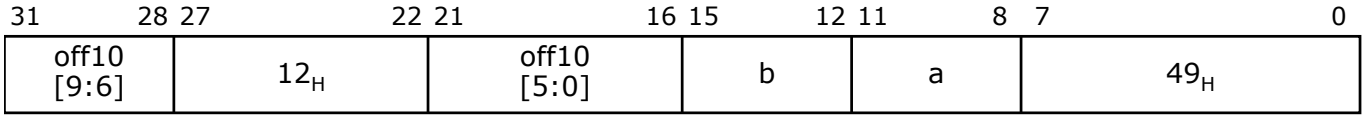
The M(EA, word) read and the M(EA, word) write are atomic.

SWAPMSK.W A[b], off10, E[a] (BO) (Post-increment Addressing Mode)



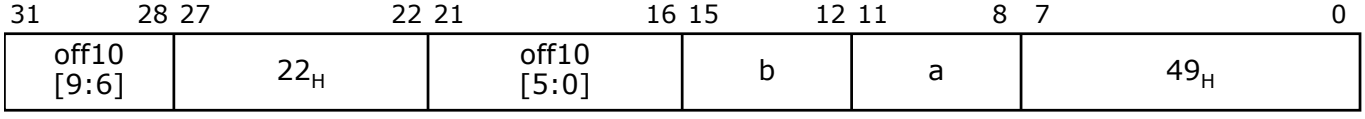

```
EA = A[b];
tmp = M(EA, word);
M(EA, word) = (tmp & ~D[a+1]) | (D[a] & D[a+1]);
D[a] = tmp;
A[b] = EA + sign_ext(off10);
```

SWAPMSK.W A[b], off10, E[a] (BO) (Pre-increment Addressing Mode)



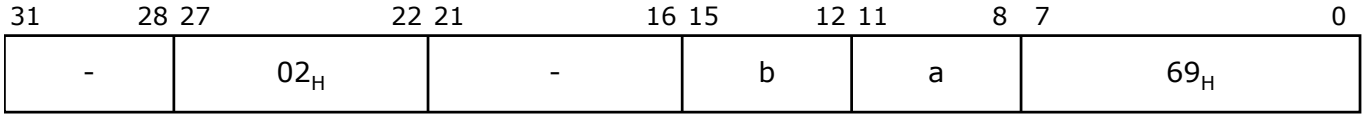

```
EA = A[b] + sign_ext(off10);
tmp = M(EA, word);
M(EA, word) = (tmp & ~D[a+1]) | (D[a] & D[a+1]);
D[a] = tmp;
A[b] = EA;
```

SWAPMSK.W A[b], off10, E[a] (BO) (Base + Short Offset Addressing Mode)



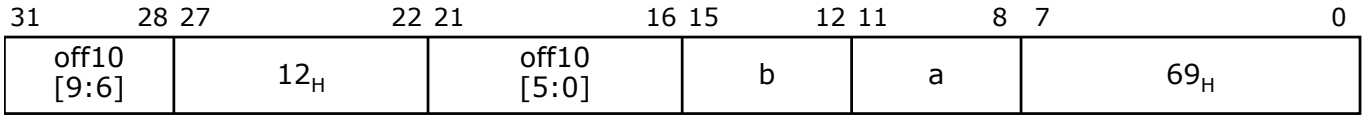

```
EA = A[b] + sign_ext(off10);
tmp = M(EA, word);
M(EA, word) = (tmp & ~D[a+1]) | (D[a] & D[a+1]);
D[a] = tmp;
```

SWAPMSK.W 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;
tmp = M(EA, word);
M(EA, word) = (tmp & ~D[a+1]) | (D[a] & D[a+1]);
D[a] = tmp;
new_index = reverse16(reverse16(index) + reverse16(incr));
A[b+1] = {incr[15:0], new_index[15:0]};
```

SWAPMSK.W 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;
tmp = M(EA, word);
M(EA, word) = (tmp & ~D[a+1]) | (D[a] & D[a+1]);
D[a] = tmp;
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]};
```

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

```
swap.w    [a0+]2, e0
swap.w    [+a0]2, e0
swap.w    [a3]+24, e2
swap.w    [a0/a1+r], e0
swap.w    [p0+r], e0
swap.w    [a0/a1+c], e0
swap.w    [p0+c], e0
```

See Also

LDMST, ST.T, SWAP.W, CMPSWAP.W

3.1.327 SYSCALL

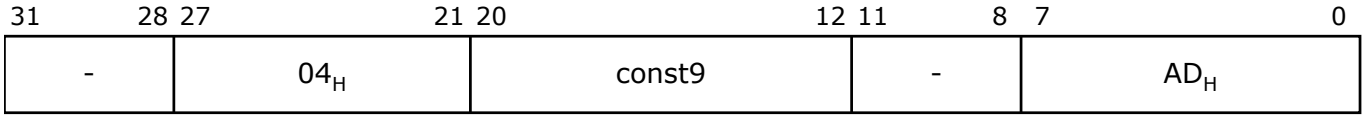
System Call

Description

Cause a system call trap, using Trap Identification Number (TIN) specified by const9[7:0].

***Note**: The trap return address 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

HVCALL, RET, RFE, TRAPV, TRAPSV, UNPACK

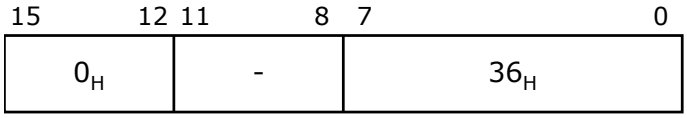
3.1.328 TRAPINV

Trap Invalid Opcode

Description

Generate an IOPC (invalid Opcode) trap if execution is attempted. Any program memory filled with this instruction will generated an IOPC (Invalid Opcode) trap if execution is attempted.

TRAPINV (SR)




```
trap(IOPC);
```

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

```
trapinv
```

See Also

TRAPSV, TRAPV

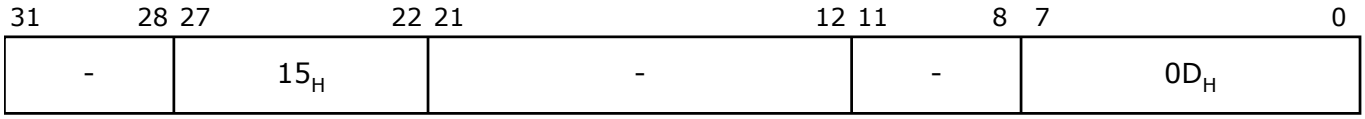
3.1.329 TRAPSV

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 (SOVF-trap).

TRAPSV (SYS)




```
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, TRAPINV, TRAPV

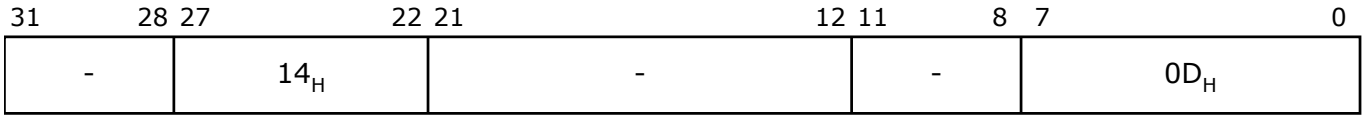
3.1.330 TRAPV

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)




```
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, TRAPINV, TRAPSV, UNPACK

3.1.331 UNPACK

Unpack Floating Point

Description

Take a single-precision IEEE 754-2019 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.

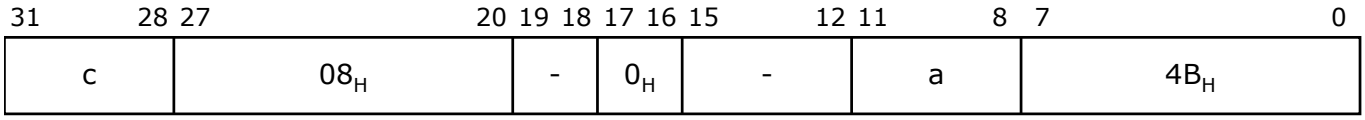
The odd register E[c][63:32] receives the unbiased exponent. The even register E[c][31:0] receives the mantissa. Note that the sign-bit of the floating point number is available in bit 31 of data register D[a].

To compute the mantissa and the exponent, the input number is first checked for special cases: Infinity, NAN, Zero & subnormalised. 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 normalized 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 subnormalised 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)




```
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'b0000000};
} 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 {
    // subnormalised
    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

3.1.332 WAIT

Wait

Description

The WAIT instruction causes the processor to suspend execution until the next enabled interrupt or asynchronous event is detected.

WAIT (SYS)

31 28	27 22	21 12	11 8	7 0
-	16_H	-	-	$0D_H$

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

```
wait
```

See Also

-

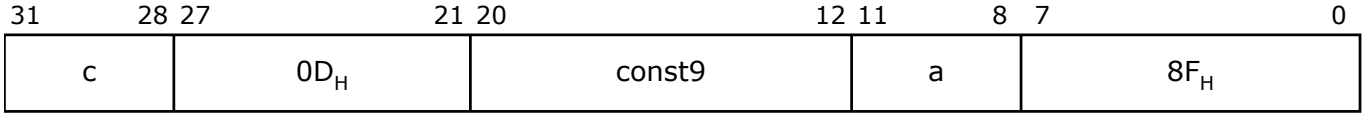
3.1.333 XNOR

Bitwise XNOR

Description

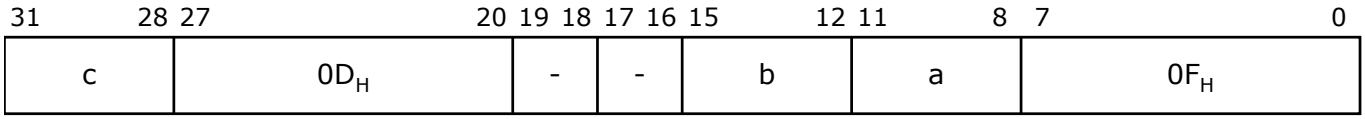
Compute the bitwise exclusive NOR of the contents of data register D[a] and the contents of either data register D[b] (instruction format RR) or const9 (instruction format RC). Put the result in to data register D[c]. The value const9 is zero-extended.

XNOR D[c], D[a], const9 (RC)




```
D[c] = ~(D[a] ^ zero_ext(const9));
```

XNOR D[c], D[a], D[b] (RR)




```
D[c] = ~(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

```
xnor    d3, d1, #126
xnor    d3, d1, d2
```

See Also

AND, ANDN, NAND, NOR, NOT, OR, ORN, XOR

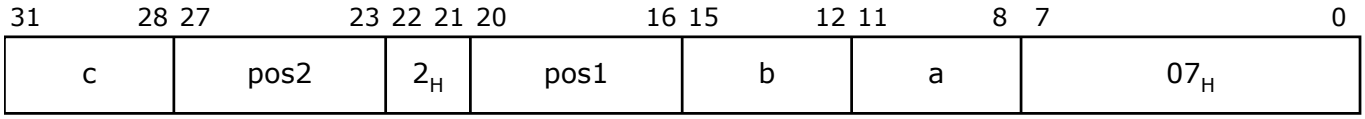
3.1.334 XNOR.T

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)




```
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

3.1.335 XOR

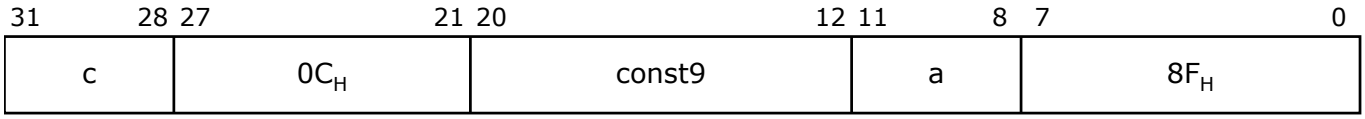
Bitwise XOR

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-bit.

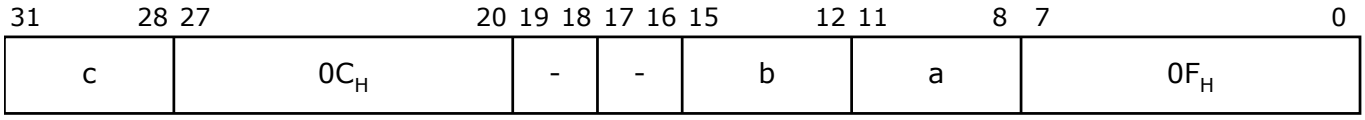
Compute the bitwise exclusive OR of the contents of data register D[a] and the contents of data register D[b]. Put the result in data register D[a].

XOR D[c], D[a], const9 (RC)



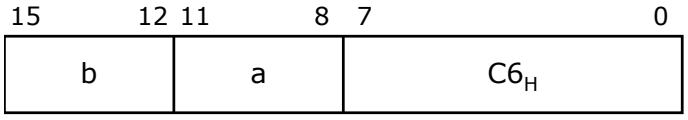

```
D[c] = D[a] ^ zero_ext(const9);
```

XOR D[c], D[a], D[b] (RR)




```
D[c] = D[a] ^ D[b];
```

XOR 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

```
xor    d3, d1, #126
xor    d3, d1, d2
```

```
xor    d3, d2
```

See Also

AND, ANDN, NAND, NOR, NOT, OR, ORN, XNOR

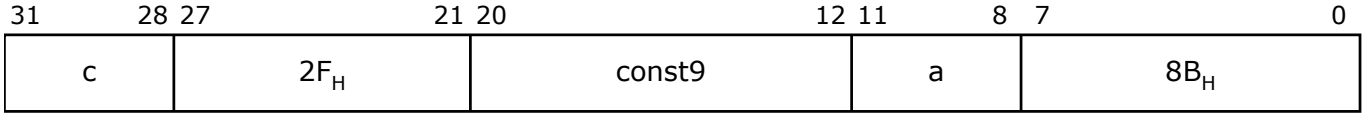
3.1.336 XOR.EQ

Equal, XOR Accumulating

Description

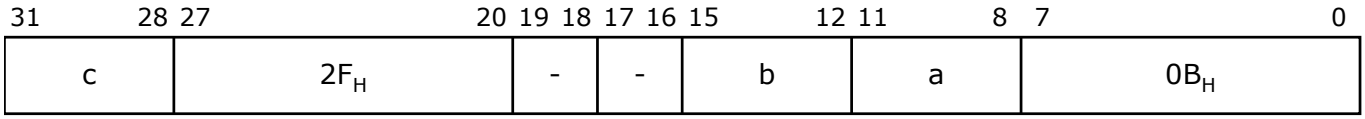
Compute the logical XOR of D[c][0] and the Boolean result of the EQ operation on the contents of data register D[a] and either data register D[b] (instruction format RR) or const9 (instruction format RC). Put the result in D[c][0]. All other bits in D[c] are unchanged. The value const9 is sign-extended.

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




```
D[c] = {D[c][31:1], D[c][0] XOR (D[a] == sign_ext(const9))};
```

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




```
D[c] = {D[c][31:1], D[c][0] 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, #126
xor.eq    d3, d1, d2
```

See Also

AND.EQ, OR.EQ

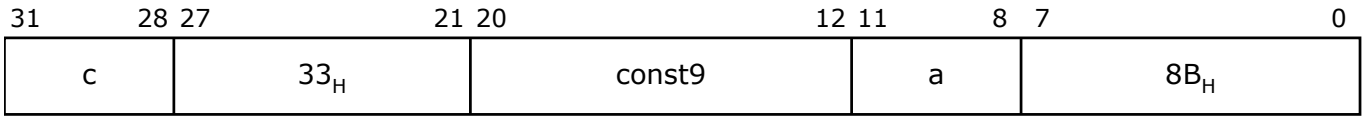
3.1.337 XOR.GE

Greater Than or Equal, XOR Accumulating

Description

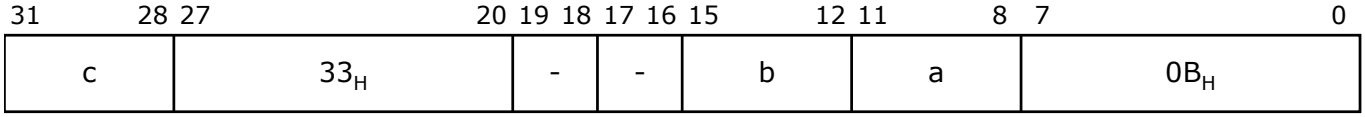
Calculate the logical XOR of D[c][0] and the Boolean result of the GE operation on the contents of data register D[a] and either data register D[b] (instruction format RR) or const9 (instruction format RC). Put the result in D[c][0]. All other bits in D[c] are unchanged. D[a] and D[b] are treated as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

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])};
```

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.ge    d3, d1, #126
xor.ge    d3, d1, d2
```

See Also

AND.GE, AND.GE.U, OR.GE, OR.GE.U, XOR.GE.U

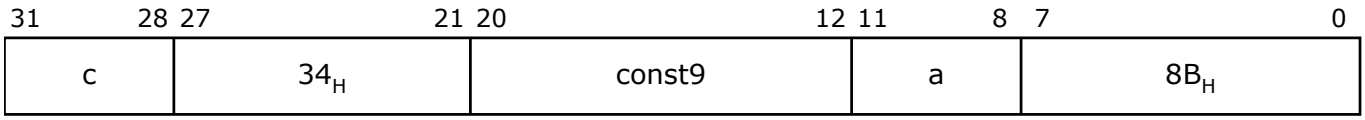
3.1.338 XOR.GE.U

Greater Than or Equal, XOR Accumulating Unsigned

Description

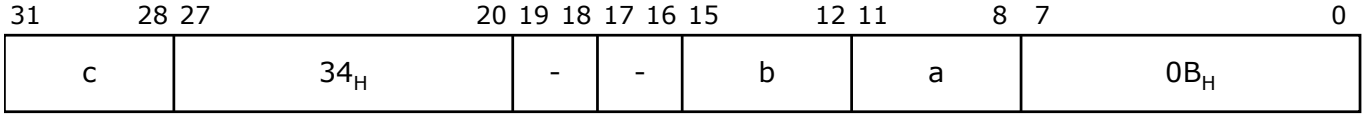
Calculate the logical XOR of D[c][0] and the Boolean result of the GE.U operation on the contents of data register D[a] and either data register D[b] (instruction format RR) or const9 (instruction format RC). Put the result in D[c][0]. All other bits in D[c] are unchanged. D[a] and D[b] are treated as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

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
```

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

See Also

AND.GE, AND.GE.U, OR.GE, OR.GE.U, XOR.GE

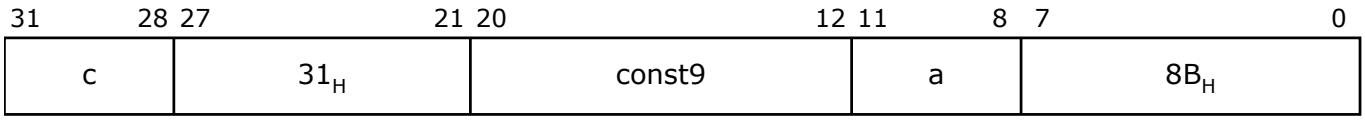
3.1.339 XOR.LT

Less Than, XOR Accumulating

Description

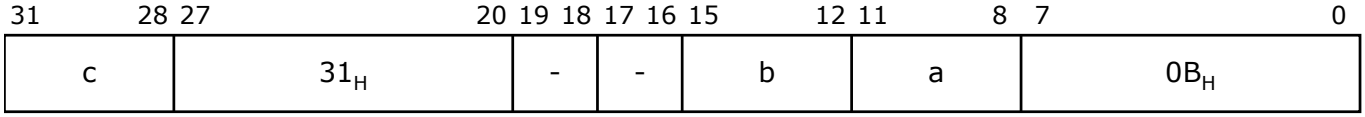
Calculate the logical XOR of D[c][0] and the Boolean result of the LT operation on the contents of data register D[a] and either data register D[b] (instruction format RR) or const9 (instruction format RC). Put the result in D[c][0]. All other bits in D[c] are unchanged. D[a] and D[b] are treated as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

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




```
D[c] = {D[c][31:1], D[c][0] XOR (D[a] < sign_ext(const9))};
```

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




```
D[c] = {D[c][31:1], D[c][0] 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.lt    d3, d1, #126
xor.lt    d3, d1, d2
```

See Also

AND.LT, AND.LT.U, OR.LT, OR.LT.U, XOR.LT.U

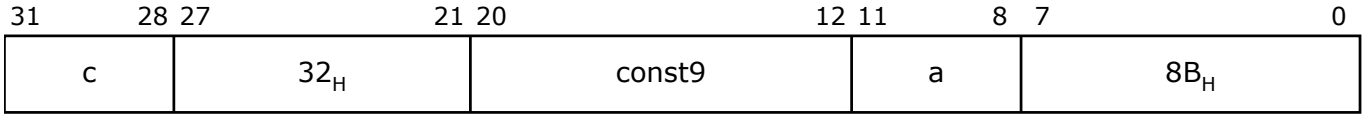
3.1.340 XOR.LT.U

Less Than, XOR Accumulating Unsigned

Description

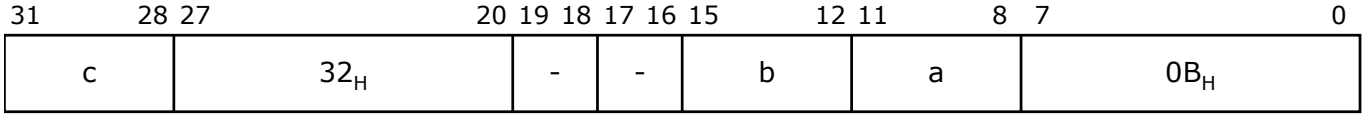
Calculate the logical XOR of D[c][0] and the Boolean result of the LT.U operation on the contents of data register D[a] and either data register D[b] (instruction format RR) or const9 (instruction format RC). Put the result in D[c][0]. All other bits in D[c] are unchanged. D[a] and D[b] are treated as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

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




```
D[c] = {D[c][31:1], D[c][0] XOR (D[a] < zero_ext(const9))}; // unsigned
```

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




```
D[c] = {D[c][31:1], D[c][0] XOR (D[a] < D[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

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

See Also

AND.LT, AND.LT.U, OR.LT, OR.LT.U, XOR.LT

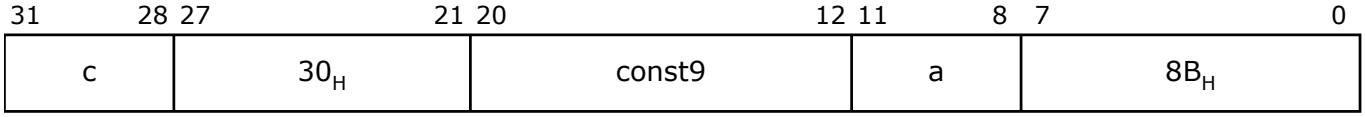
3.1.341 XOR.NE

Not Equal, XOR Accumulating

Description

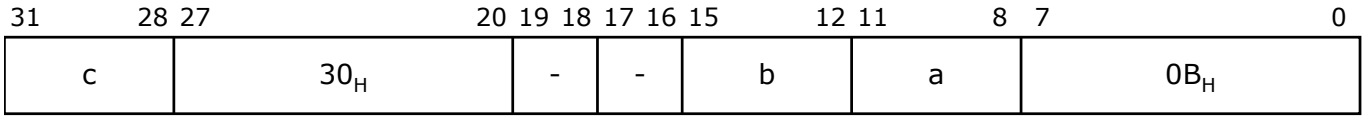
Calculate the logical XOR of D[c][0] and the Boolean result of the NE operation on the contents of data register D[a] and either data register D[b] (instruction format RR) or const9. (instruction format RC). Put the result in D[c][0]. All other bits in D[c] are unchanged. The value const9 is sign-extended.

XOR.NE D[c], D[a], const9 (RC)




```
D[c] = {D[c][31:1], D[c][0] XOR (D[a] != sign_ext(const9))};
```

XOR.NE D[c], D[a], D[b] (RR)




```
D[c] = {D[c][31:1], D[c][0] 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.ne    d3, d1, #126
xor.ne    d3, d1, d2
```

See Also

AND.NE, OR.NE

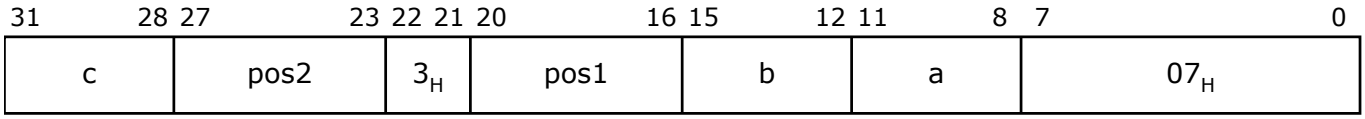
3.1.342 XOR.T

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 Single-precision FPU instructions

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

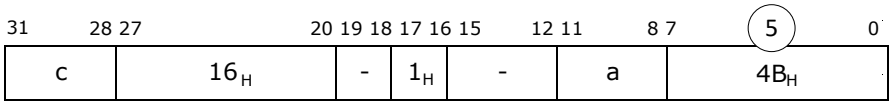
3.x.xxx UTOF (1)

Unsigned to Floating-point (2)

Description (3)

Converts the contents of data register D[a] from 32-bit unsigned integer format to single-precisionIEEE-754-2019 floating-point format. The rounded result is stored in D[c]. The rounding mode used for the conversion is defined by the PSW.RM field.

UTOF D[c], D[a] (RR) (4)




```
rounded_result = sp_round(u_real(D[a]), PSW.RM);
result = ieee754_sp_format(rounded_result);
D[c] = result[31:0];
```
(6)

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. (7)
FZ	Not set by this instruction.
FU	Not set by this instruction.
FX	if(sp_real(result) != u_real(D[a])) then set_FX = 1 else set_FX = 0; if(set_FX) then PSW.FX = 1;

Examples (8)

```
utof        d2, d1
```

See also (9)

FTOU, FTOUZ

Key:

1. Instruction Mnemonic
2. Instruction Longname
3. Description
4. Syntax, followed by Instruction Format in parentheses
5. Opcodes
6. Operation in RTL format
7. Exception Flags. IEEE-754 Exceptions that can occur when using this instruction
8. Instruction examples
9. Related instructions

3.2.1 ABS.F

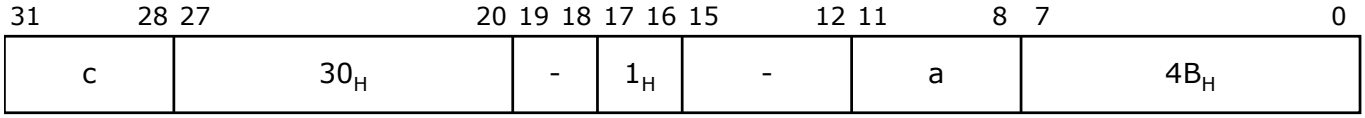
Absolute Value Float

Description

Copies the absolute value of data register D[a] to data register D[c] setting the sign bit to 0 (positive).
The operand and result are treated as single-precision IEEE 754-2019 floating-point numbers.

***Note**: This instruction is a quiet-computational operation, which treats floating-point numbers and NaNs alike and does not signal exceptions.*

ABS.F D[c], D[a] (RR)




```
D[c][30:0] = D[a][30:0];
D[c][31] = 0;
```

Exception Flags

FS	Not set by this instruction.
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	Not set by this instruction.

Examples

```
abs.f    d4, d1
```

See Also

NEG.F

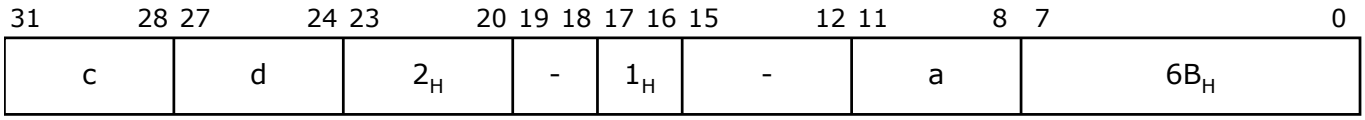
3.2.2 ADD.F

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 754-2019 floating-point numbers. If either operand is a NaN (quiet or signalling), then the return result will be the quiet NaN $7FC00000_H$.

ADD.F D[c], D[d], D[a] (RRR)




```
arg_a = sp_subnorm_to_zero(f_real(D[a]);
arg_b = sp_subnorm_to_zero(f_real(D[d]);
if(is_sp_nan(D[a]) OR is_sp_nan(D[d])) then
    result = SP_QUIET_NAN;
else if(is_sp_pos_inf(D[a]) AND is_sp_neg_inf(D[d])) then
    result = SP_ADD_NAN;
else if(is_sp_neg_inf(D[a]) AND is_sp_pos_inf(D[d])) then
    result = SP_ADD_NAN;
else {
    precise_result = add(arg_a,arg_b);
    normal_result  = sp_subnorm_to_zero(precise_result);
    rounded_result = sp_round(normal_result, PSW.RM);
    result         = ieee754_sp_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_sp_s_nan(D[a]) OR is_sp_s_nan(D[d]) OR (result == SP_ADD_NAN)) then set_FI = 1 else set_FI = 0; if(set_FI) then PSW.FI = 1;
FV	if(fp_abs(rounded_result) >= 2^{128}) 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^{-126}) then set_FU = 1 else set_FU = 0; if(set_FU) then PSW.FU = 1;
FX	if(precise_result != sp_real(result)) then set_FX = 1 else set_FX = 0; if(set_FX AND !set_FI) then PSW.FX = 1;

Examples

```
add.f    d3, d1, d2
```

See Also

SUB.F

3.2.3 CMP.F

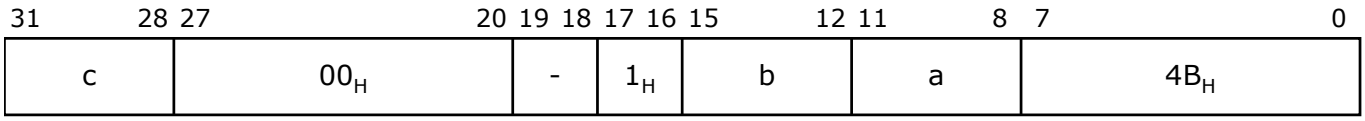
Compare Float

Description

This instruction compares the single-precision IEEE 754-2019 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 subnormal
bit [5] D[b] is subnormal
bits[31:6] are cleared.
The 'unordered' bit is asserted if either operand is a NaN.

***Note**: CMP.F does not substitute subnormal operands for zero before computation.*

CMP.F D[c], D[a], D[b] (RR)




```
D[c][0] = sp_lt(D[a], D[b]);
D[c][1] = sp_eq(D[a], D[b]);
D[c][2] = sp_gt(D[a], D[b]);
D[c][3] = (is_sp_nan(D[a]) OR is_sp_nan(D[b]));
D[c][4] = sp_is_subnorm(D[a]);
D[c][5] = sp_is_subnorm(D[b]);
D[c][31:6] = 0;
```

Exception Flags

FS	if(set_FI) then PSW.FS = 1 else PSW.FS = 0;
FI	if(is_sp_s_nan(D[a]) OR is_sp_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

-

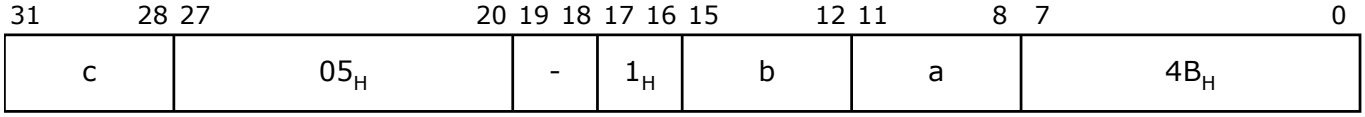
3.2.4 DIV.F

Divide Float

Description

Divides the contents of data register D[a] by the contents of data register D[b] and put the result in data register D[c]. The operands and result are single-precision IEEE 754-2019 floating-point numbers. If either operand is a NaN (quiet or signalling), then the return result will be the quiet NaN $7FC00000_H$.

DIV.F D[c], D[a], D[b] (RR)




```
arg_a = sp_subnorm_to_zero(f_real(D[a]);
arg_b = sp_subnorm_to_zero(f_real(D[b]);
if(is_sp_nan(D[a]) OR is_sp_nan(D[b])) then
    result = SP_QUIET_NAN;
else if(is_sp_inf(D[a]) AND is_sp_inf(D[b])) then
    result = SP_DIV_NAN;
else if((arg_a == 0.0) AND (arg_b == 0.0)) then
    result = SP_DIV_NAN;
else {
    precise_result = divide(arg_a, arg_b);
    normal_result  = sp_subnorm_to_zero(precise_result);
    rounded_result = sp_round(normal_result, PSW.RM);
    result         = ieee754_sp_format(rounded_result);
}
D[c] = result[31:0];
```

Exception Flags

FS	if(set_FI OR set_FV OR set_FZ OR set_FU OR set_FX) then PSW.FS = 1 else PSW.FS = 0;
FI	if(is_sp_s_nan(D[a]) OR is_sp_s_nan(D[b]) OR (result == SP_DIV_NAN)) then set_FI = 1 else set_FI = 0; if(set_FI) then PSW.FI = 1;
FV	if(fp_abs(rounded_result) >= 2^{128}) then set_FV = 1 else set_FV = 0; if(set_FV) then PSW.FV = 1;
FZ	if(is_sp_zero(D[b]) AND !is_sp_inf(D[a]) AND !is_sp_zero(D[a]) AND !is_sp_nan(D[a])) then set_FZ = 1 else set_FZ = 0; if(set_FZ) then PSW.FZ = 1;
FU	if(fp_abs(precise_result) < 2^{-126}) then set_FU = 1 else set_FU = 0; if(set_FU) then PSW.FU = 1;
FX	if(precise_result != sp_real(result)) then set_FX = 1 else set_FX = 0; if(set_FX AND !set_FI AND !set_FZ) then PSW.FX = 1;

Examples

```
div.f    d3, d1, d2
```

See Also

-

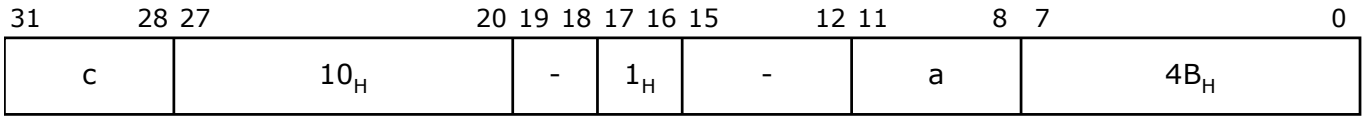
3.2.5 FTOI

Float to Integer

Description

Converts the contents of data register D[a] from single-precision IEEE 754-2019 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_sp_nan(D[a])) then
    result = 0;
else if(sp_real(D[a]) > 2^31-1) then
    result = 7FFFFFFFH;
else if(sp_real(D[a]) < -2^31) then
    result = 80000000H;
else {
    result = sp_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((sp_real(D[a]) > 2^{31}-1) OR (sp_real(D[a]) < -2^{31}) OR is_sp_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(sp_real(D[a]) != i_real(result)) then set_FX = 1 else set_FX = 0; if(set_FX AND !set_FI) then PSW.FX = 1;

Examples

```
ftoi    d2, d1
```

See Also

ITOF, FTOIN, FTOIZ

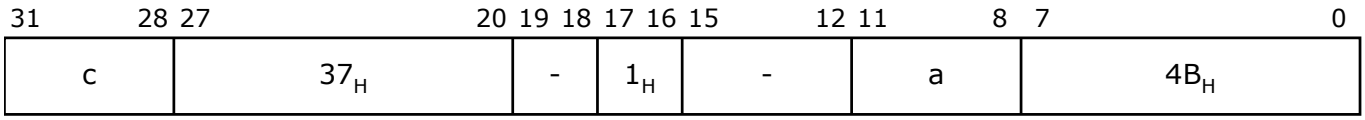
3.2.6 FTOIN

Float to Integer, Round to Nearest

Description

Converts the contents of data register D[a] from single-precision IEEE 754-2019 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 round to nearest.

FTOIN D[c], D[a] (RR)




```
if(is_sp_nan(D[a])) then
    result = 0;
else if(sp_real(D[a]) > 2^31-1) then
    result = 7FFFFFFF_H;
else if(sp_real(D[a]) < -2^31) then
    result = 80000000_H;
else {
    result = sp_round_to_integer(D[a], 00_B);
}
D[c] = result[31:0];
```

Exception Flags

FS	if(set_FI OR set_FX) then PSW.FS = 1 else PSW.FS = 0;
FI	if((sp_real(D[a]) > 2^{31}-1) OR (sp_real(D[a]) < -2^{31}) OR is_sp_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(sp_real(D[a]) != i_real(result)) then set_FX = 1 else set_FX = 0; if(set_FX AND !set_FI) then PSW.FX = 1;

Examples

```
ftoin    d2, d1
```

See Also

ITOF, FTOI, FTOIZ

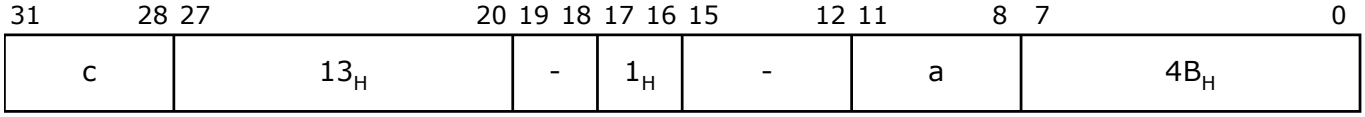
3.2.7 FTOIZ

Float to Integer, Round towards Zero

Description

Converts the contents of data register D[a] from single-precision IEEE 754-2019 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 round towards zero.

FTOIZ D[c], D[a] (RR)




```
if(is_sp_nan(D[a])) then
    result = 0;
else if(sp_real(D[a]) > 2^31-1) then
    result = 7FFFFFFF_H;
else if(sp_real(D[a]) < -2^31) then
    result = 80000000_H;
else {
    result = sp_round_to_integer(D[a], 11_B);
}
D[c] = result[31:0];
```

Exception Flags

FS	if(set_FI OR set_FX) then PSW.FS = 1 else PSW.FS = 0;
FI	if((sp_real(D[a]) > 2^{31}-1) OR (sp_real(D[a]) < -2^{31}) OR is_sp_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(sp_real(D[a]) != i_real(result)) then set_FX = 1 else set_FX = 0; if(set_FX AND !set_FI) then PSW.FX = 1;

Examples

```
ftoiz    d2, d1
```

See Also

ITOF, FTOI, FTOIN

3.2.8 FTOQ31

Float to Fraction

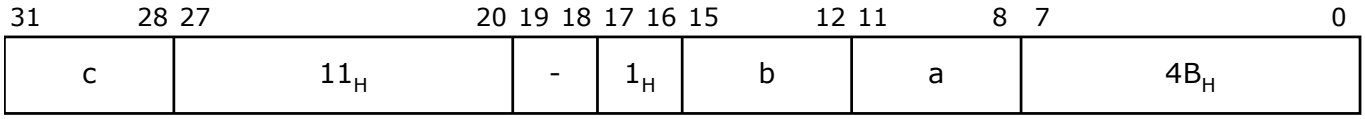
Description

Subtracts D[b] from the exponent of the single-precision IEEE 754-2019 floating-point input value D[a] and converts the result to the Q31 fraction 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).

FTOQ31 D[c], D[a], D[b] (RR)




```
arg_a = sp_subnorm_to_zero(f_real(D[a]);
if(is_sp_nan(D[a])) then
    result = 0;
else {
    precise_result = mul(arg_a, 2^-D[b][8:0]);
    if(precise_result > q_real(7FFFFFFFH)) then
        result = 7FFFFFFFH;
    else if(precise_result < -1.0)
        result = 80000000H;
    else {
        result = sp_round_to_q31(precise_result, 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((precise_result > q_real(7FFFFFFF$_H$)) OR (precise_result < -1.0) OR is_sp_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(precise_result != q_real(result)) then set_FX = 1 else set_FX = 0; if(set_FX AND !set_FI) then PSW.FX = 1;

Examples

```
ftoq31    d3, d1, d2
```

See Also

Q31TOF, FTOQ31Z

3.2.9 FTOQ31Z

Float to Fraction, Round towards Zero

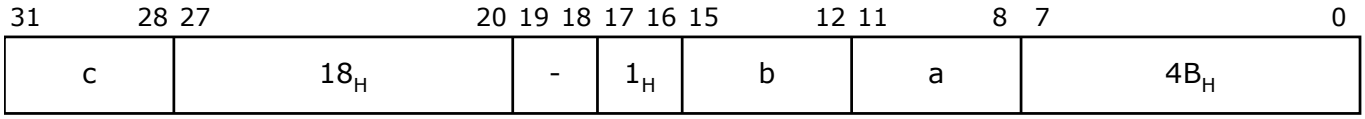
Description

Subtracts D[b] from the exponent of the single-precision IEEE 754-2019 floating-point input value D[a] and converts the result to the Q31 fraction format. The rounded result is put in data register D[c]. The rounding mode used for the conversion is round towards zero.

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).

FTOQ31Z D[c], D[a], D[b] (RR)




```
arg_a = sp_subnorm_to_zero(f_real(D[a]);
if(is_sp_nan(D[a])) then
    result = 0;
else {
    precise_result = mul(arg_a, 2^-D[b][8:0]);
    if(precise_result > q_real(7FFFFFFFH)) then
        result = 7FFFFFFFH;
    else if(precise_result < -1.0) then
        result = 80000000H;
    else {
        result = sp_round_to_q31(precise_result, 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((precise_result > q_real(7FFFFFFF$_H$)) OR (precise_result < -1.0) OR is_sp_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(precise_result != q_real(result)) then set_FX = 1 else set_FX = 0; if(set_FX AND !set_FI) then PSW.FX = 1;

Examples

```
ftoq31z    d3, d1, d2
```

See Also

Q31TOF, FTOQ31

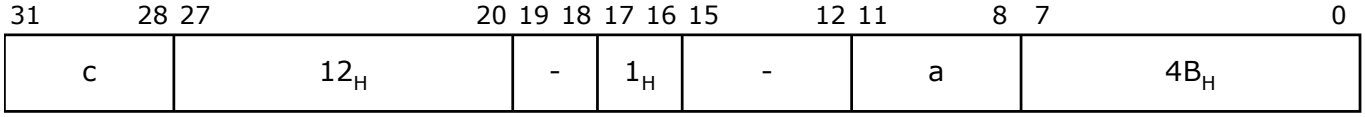
3.2.10 FTOU

Float to Unsigned

Description

Converts the contents of data register D[a] from single-precision IEEE 754-2019 floating-point format to a 32-bit unsigned 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.

FTOU D[c], D[a] (RR)




```
if(is_sp_nan(D[a])) then
    result = 0;
else if(sp_real(D[a]) > 2^32-1) then
    result = FFFFFFFFH;
else if(sp_real(D[a]) < 0.0) then
    result = 0;
else {
    result = sp_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((sp_real(D[a]) > 2^{32}-1) OR (sp_real(D[a]) < 0.0) OR is_sp_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(sp_real(D[a]) != u_real(result)) then set_FX = 1 else set_FX = 0; if(set_FX AND !set_FI) then PSW.FX = 1;

Examples

```
ftou    d2, d1
```

See Also

UTOF, FTOUZ

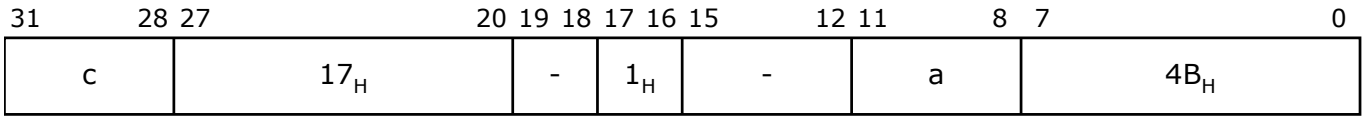
3.2.11 FTOUZ

Float to Unsigned, Round towards Zero

Description

Converts the contents of data register D[a] from single-precision IEEE 754-2019 floating-point format to a 32-bit unsigned integer format. The rounded resut is put in data register D[c]. The rounding mode used for the conversion is round towards zero.

FTOUZ D[c], D[a] (RR)




```
if(is_sp_nan(D[a])) then
    result = 0;
else if(sp_real(D[a]) > 2^32-1) then
    result = FFFFFFFF_H;
else if(sp_real(D[a]) < 0.0) then
    result = 0;
else {
    result = sp_round_to_unsigned(D[a], 11_B);
}
D[c] = result[31:0];
```

Exception Flags

FS	if(set_FI OR set_FX) then PSW.FS = 1 else PSW.FS = 0;
FI	if((sp_real(D[a]) > 2^{32}-1) OR (sp_real(D[a]) < 0.0) OR is_sp_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(sp_real(D[a]) != u_real(result)) then set_FX = 1 else set_FX = 0; if(set_FX AND !set_FI) then PSW.FX = 1;

Examples

```
ftouz    d2, d1
```

See Also

UTOF, FTOU

3.2.12 FTOHP

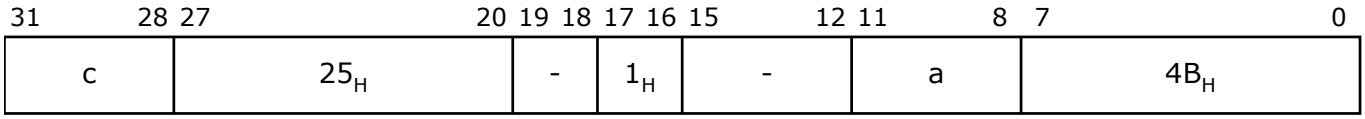
Single Precision to Half Precision

Description

Convert the contents of data register D[a] from single-precision IEEE 754-2019 to 16-bit half-precision (data interchange) IEEE 754-2019 floating point format. The rounded result is put in data register D[c][15:0]. D[c][31:16] is set to zero. The rounding mode used for the conversion is defined by the PSW.RM field

If D[a] contains a NaN value, the result in D[c] will also contain a NaN. In order to maintain both the distinction between signalling and quiet NaNs, and the quiet NaN invalid operation code, the top two output mantissa bits are set to the top two input mantissa bits and the remaining 8 output mantissa bits are set to the bottom 8 input mantissa bits, whenever D[a] contains a NaN value. In the case when no output mantissa bits would be set on an input NaN, D[c][8] is set to maintain the NaN value.

FTOHP D[c], D[a] (RR)




```
if (is_sp_inf(D[a])) then {
    if(sp_sign_bit(D[a])) then {
        result[15:0] = HP_NEG_INFINITY;
    } else {
        result[15:0] = HP_POS_INFINITY;
    }
} else if (is_sp_nan(D[a])) then {
    result[15] = sp_sign_bit(D[a]);
    result[14:10] = 1FH;
    result[9:8] = D[a][22:21];
    result[7:0] = D[a][7:0];
    if (result[9:0] == 0) then {
        result[8] = 1B;
    }
} else {
    precise_result = sp_subnorm_to_zero(D[a]);
    rounded_result = hp_round(precise_result, PSW.RM);
    result[15:0] = ieee754_hp_format(rounded_result);
}
D[c] = 0 | result[15:0];
```

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_sp_s_nan(D[a]) then set_FI = 1 else set_FI = 0; if(set_FI) then PSW.FI = 1;
FV	if(fp_abs(rounded_result) >= 2^{15}) 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(rounded_result) < 2^{-14})) then set_FU = 1 else set_FU = 0; if(set_FU) then PSW.FU = 1;
FX	if(precise_result != hp_real(result[15:0])) then set_FX = 1 else set_FX = 0; if(set_FX AND !set_FI) then PSW.FX = 1;

Examples

```
ftohp    d1, d2
```

See Also

HPTOF

3.2.13 HPTOF

Half Precision to Single Precision

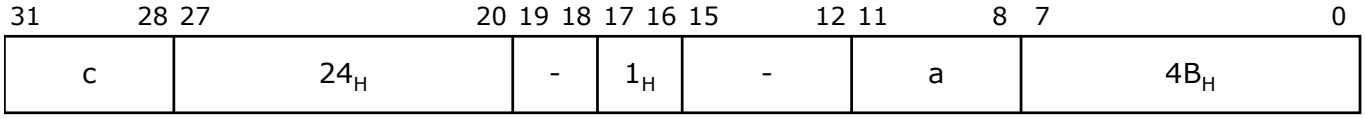
Description

Convert the contents of data register D[a][15:0] from 16-bit half-precision (data interchange) IEEE 754-2019 floating point to single-precision IEEE 754-2019 floating-point format and put the result in data register D[c].

If D[a][15:0] contains a NaN value, the result in D[c] will also contain a NaN. In order to maintain both the distinction between signalling and quiet NaNs, and the quiet NaN invalid operation code, D[c][22:21] is set to the top two input mantissa bits and D[c][7:0] is set to the remaining input mantissa bits, whenever D[a] contains a NaN value. All other output mantissa bits are cleared.

***Note**: This instruction does not substitute subnormal operands for zero before conversion.*

HPTOF D[c], D[a] (RR)




```
hp = D[a][15:0];
if (is_hp_inf(hp)) then {
    if(hp_sign_bit(hp)) then {
        result = NEG_INFINITY;
    } else {
        result = POS_INFINITY;
    }
} else if (is_hp_nan(hp)) then {
    result[31] = hp_sign_bit(hp);
    result[30:23] = FFH;
    result[22:21] = D[a][9:8];
    result[20:8] = 0;
    result[7:0] = D[a][7:0];
} else {
    precise_result = hp_real(hp);
    result = ieee754_sp_format(precise_result);
}
D[c] = result;
```

Exception Flags

FS	if(set_FI) then PSW.FS = 1 else PSW.FS = 0;
FI	if(is_hp_s_nan(D[a][15:0])) 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

```
hptof    d1, d2
```

See Also

FTOHP

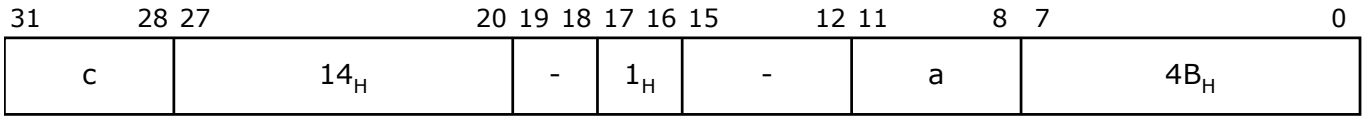
3.2.14 ITOF

Integer to Float

Description

Converts the contents of data register D[a] from 32-bit two's complement signed integer format to single-precision IEEE 754-2019 floating-point 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.

ITOF D[c], D[a] (RR)




```
rounded_result = sp_round(i_real(D[a]), PSW.RM);
result = ieee754_sp_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(sp_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, FTOIN, FTOIZ

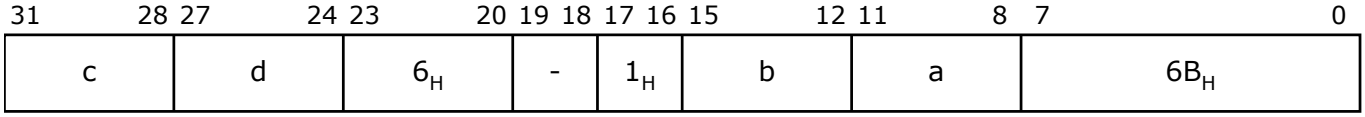
3.2.15 MADD.F

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 single-precision IEEE 754-2019 floating-point numbers. If an operand is a NaN (quiet or signalling), then the return result will be the quiet NaN $7FC00000_H$.

MADD.F D[c], D[d], D[a], D[b] (RRR)




```
arg_a = sp_subnorm_to_zero(sp_real(D[a]);
arg_b = sp_subnorm_to_zero(sp_real(D[b]);
arg_c = sp_subnorm_to_zero(sp_real(D[d]);
if(is_sp_nan(D[a]) OR is_sp_nan(D[b]) OR is_sp_nan(D[d])) then
    result = SP_QUIET_NAN;
else if(is_sp_inf(D[a]) AND is_sp_zero(D[b])) then
    result = SP_MUL_NAN;
else if(is_sp_zero(D[a]) AND is_sp_inf(D[b])) then
    result = SP_MUL_NAN;
else if((   (is_sp_pos_inf(D[a]) AND !sp_sign_bit(D[b]))
         OR (is_sp_neg_inf(D[a]) AND  sp_sign_bit(D[b]))
         OR (!sp_sign_bit(D[a])  AND is_sp_pos_inf(D[b]))
         OR ( sp_sign_bit(D[a])  AND is_sp_neg_inf(D[b]))
        ) AND is_sp_neg_inf(D[d]) ) then
    result = SP_ADD_NAN;
else if((   (is_sp_pos_inf(D[a]) AND  sp_sign_bit(D[b]))
         OR (is_sp_neg_inf(D[a]) AND !sp_sign_bit(D[b]))
         OR ( sp_sign_bit(D[a])  AND is_sp_pos_inf(D[b]))
         OR (!sp_sign_bit(D[a])  AND is_sp_neg_inf(D[b]))
        ) AND is_sp_pos_inf(D[d])) then
    result = SP_ADD_NAN;
else {
    precise_mul_result = mul(arg_a, arg_b);
    precise_result = add(precise_mul_result, arg_c);
    normal_result  = sp_subnorm_to_zero(precise_result);
    rounded_result = sp_round(normal_result, PSW.RM);
    result = ieee754_sp_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_sp_s_nan(D[a]) OR is_sp_s_nan(D[b]) OR is_sp_s_nan(D[d]) OR (result == SP_ADD_NAN) OR (result == SP_MUL_NAN)) then set_FI = 1 else set_FI = 0; if(set_FI) then PSW.FI = 1;
FV	if(fp_abs(rounded_result) >= 2^{128}) 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^{-126}) then set_FU = 1 else set_FU = 0; if(set_FU) then PSW.FU = 1;
FX	if(precise_result != sp_real(result)) then set_FX = 1 else set_FX = 0; if(set_FX AND !set_FI) then PSW.FX = 1;

Examples

```
madd.f    d4, d3, d1, d2
```

See Also

MSUB.F, MUL.F

3.2.16 MAX.F

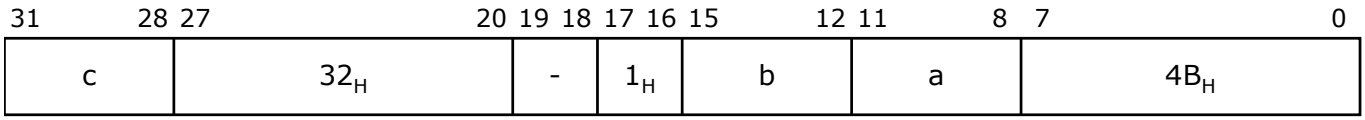
Maximum Value Float

Description

If the content of data register D[a] is greater than or equal to the content of data register D[b], then put the content of D[a] in data register D[c]; otherwise put the content of D[b] in D[c]. The operands and result are treated as single-precision IEEE 754-2019 floating-point. For this operation +0 compares greater than -0. If only one argument is a NaN the number will be returned in D[c]. If both operands are NaN (quiet or signaling), then the return result will be the quiet NaN $7FC00000_H$.

***Note**: MAX.F does not substitute subnormal operands for zero before computation.*

MAX.F D[c], D[a], D[b] (RR)




```
if (is_sp_nan(D[a]) AND is_sp_nan(D[b])) then
    result = SP_QUIET_NAN;
else if (is_sp_nan(D[a])) then
    result = D[b];
else if (is_sp_nan(D[b])) then
    result = D[a];
else if (D[a] >= D[b]) then
    result = D[a];
else {
    result = D[b];
}
D[c][31:0] = result;
```

Exception Flags

FS	if(set_FI) then PSW.FS = 1 else PSW.FS = 0;
FI	if(is_sp_s_nan(D[a]) OR is_sp_s_nan(D[b])) 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

```
max.f    d0, d1, d2
```

See Also

MIN.F

3.2.17 MIN.F

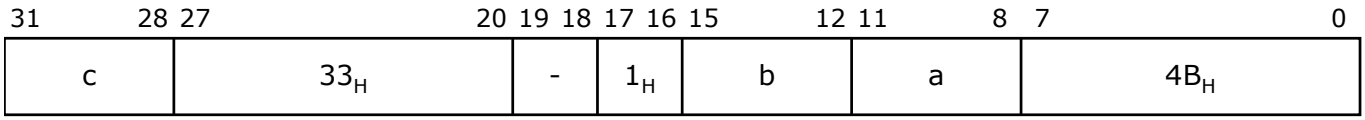
Minumum Value Float

Description

If the content of data register D[a] is lesser than or equal to the content of data register D[b], then put the content of D[a] in data register D[c]; otherwise put the content of D[b] in D[c]. The operands and result are treated as single-precision IEEE 754-2019 floating-point. For this operation +0 compares greater than -0. If only one argument is a NaN the number will be returned in D[c]. If both operands are NaN (quiet or signaling), then the return result will be the quiet NaN $7FC00000_H$.

***Note**: MIN.F does not substitute subnormal operands for zero before computation.*

MIN.F D[c], D[a], D[b] (RR)




```
if (is_sp_nan(D[a]) AND is_sp_nan(D[b])) then
    result = SP_QUIET_NAN;
else if (is_sp_nan(D[a])) then
    result = D[b];
else if (is_sp_nan(D[b])) then
    result = D[a];
else if (D[a] <= D[b]) then
    result = D[a];
else {
    result = D[b];
}
D[c][31:0] = result;
```

Exception Flags

FS	if(set_FI) then PSW.FS = 1 else PSW.FS = 0;
FI	if(is_sp_s_nan(D[a]) OR is_sp_s_nan(D[b])) 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

```
min.f    d4, d3, d5
```

See Also

MAX.F

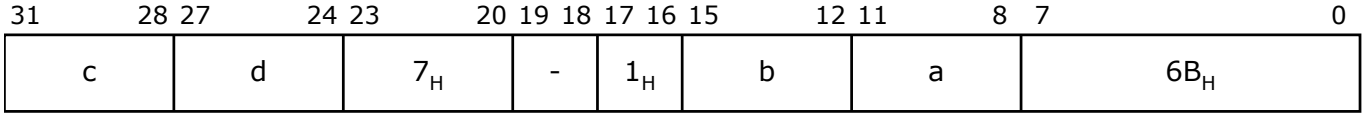
3.2.18 MSUB.F

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 single-precision IEEE 754-2019 floating-point numbers. If any operand is a NaN (quiet or signalling), then the return result will be the quiet NaN $7FC00000_H$.

MSUB.F D[c], D[d], D[a], D[b] (RRR)




```
arg_a = sp_subnorm_to_zero(sp_real(D[a]);
arg_b = sp_subnorm_to_zero(sp_real(D[b]);
arg_c = sp_subnorm_to_zero(sp_real(D[d]);
if(is_sp_nan(D[a]) OR is_sp_nan(D[b]) OR is_sp_nan(D[d])) then
    result = SP_QUIET_NAN;
else if(is_sp_inf(D[a]) AND is_sp_zero(D[b])) then
    result = SP_MUL_NAN;
else if(is_sp_zero(D[a]) AND is_sp_inf(D[b])) then
    result = SP_MUL_NAN;
else if((   (is_sp_pos_inf(D[a]) AND !sp_sign_bit(D[b]))
         OR (is_sp_neg_inf(D[a]) AND  sp_sign_bit(D[b]))
         OR (!sp_sign_bit(D[a]) AND is_sp_pos_inf(D[b]))
         OR ( sp_sign_bit(D[a]) AND is_sp_neg_inf(D[b]))
        ) AND is_sp_pos_inf(D[d]) ) then
    result = SP_ADD_NAN;
else if((   (is_sp_pos_inf(D[a]) AND  sp_sign_bit(D[b]))
         OR (is_sp_neg_inf(D[a]) AND !sp_sign_bit(D[b]))
         OR ( sp_sign_bit(D[a]) AND is_sp_pos_inf(D[b]))
         OR (!sp_sign_bit(D[a]) AND is_sp_neg_inf(D[b]))
        ) AND is_sp_neg_inf(D[d])) then
    result = SP_ADD_NAN;
else {
    precise_mul_result = mul(arg_a, arg_b);
    precise_result = add(-precise_mul_result, arg_c);
    normal_result  = sp_subnorm_to_zero(precise_result);
    rounded_result = sp_round(normal_result, PSW.RM);
    result = ieee754_sp_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_sp_s_nan(D[a]) OR is_sp_s_nan(D[b]) OR is_sp_s_nan(D[d]) OR (result == SP_ADD_NAN) OR (result == SP_MUL_NAN)) then set_FI = 1 else set_FI = 0; if(set_FI) then PSW.FI = 1;
FV	if(fp_abs(rounded_result) >= 2^{128}) 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^{-126}) then set_FU = 1 else set_FU = 0; if(set_FU) then PSW.FU = 1;
FX	if(precise_result != sp_real(result)) then set_FX = 1 else set_FX = 0; if(set_FX AND !set_FI) then PSW.FX = 1;

Examples

```
msub.f    d4, d3, d1, d2
```

See Also

MADD.F

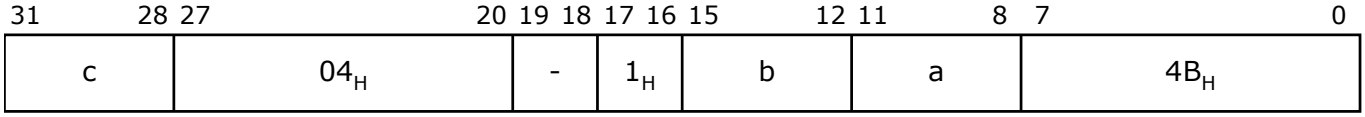
3.2.19 MUL.F

Multiply Float

Description

Multiplies D[a] and D[b] and stores the result in D[c]. The operands and result are single-precision IEEE 754-2019 floating-point numbers. If an operand is a NaN (quiet or signalling), then the return result will be the quiet NaN $7FC00000_H$.

MUL.F D[c], D[a], D[b] (RR)




```
arg_a = sp_subnorm_to_zero(sp_real(D[a]);
arg_b = sp_subnorm_to_zero(sp_real(D[b]);
if(is_sp_nan(D[a]) OR is_sp_nan(D[b])) then
    result = SP_QUIET_NAN;
else if(is_sp_inf(D[a]) AND is_sp_zero(D[b])) then
    result = SP_MUL_NAN;
else if(is_sp_inf(D[b]) AND is_sp_zero(D[a])) then
    result = SP_MUL_NAN;
else {
    precise_result = mul(arg_a, arg_b);
    normal_result  = sp_subnorm_to_zero(precise_result);
    rounded_result = sp_round(normal_result, PSW.RM);
    result = ieee754_sp_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_sp_s_nan(D[a]) OR is_sp_s_nan(D[b]) OR (result == SP_MUL_NAN)) then set_FI = 1 else set_FI = 0; if(set_FI) then PSW.FI = 1;
FV	if(fp_abs(rounded_result) >= 2^{128}) 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^{-126}) then set_FU = 1 else set_FU = 0; if(set_FU) then PSW.FU = 1;
FX	if(precise_result != sp_real(result)) then set_FX = 1 else set_FX = 0; if(set_FX AND !set_FI) then PSW.FX = 1;

Examples

```
mul.f    d3, d1, d2
```

See Also

-

3.2.20 NEG.F

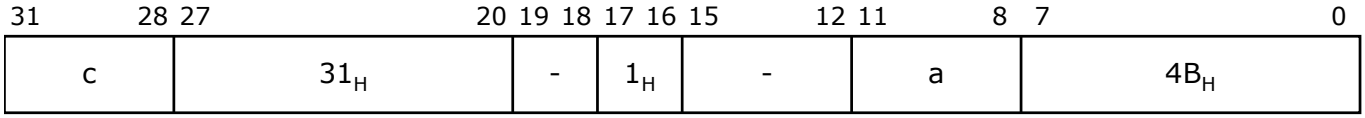
Negate Value Float

Description

Copies the value of data register D[a] to data register D[c] reversing the sign bit.
The operand and result are treated as single-precision IEEE 754-2019 floating-point numbers.

***Note**: This instruction is a quiet-computational operation, which treats floating-point numbers and NaNs alike and does not signal exceptions.*

NEG.F D[c], D[a] (RR)




```
D[c][30:0] = D[a][30:0]
if (D[a][31] == 1) then
    D[c][31] = 0;
else
    D[c][31] = 1;
```

Exception Flags

FS	Not set by this instruction.
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	Not set by this instruction.

Examples

```
neg.f    d1, d0
```

See Also

ABS.F

3.2.21 Q31TOF

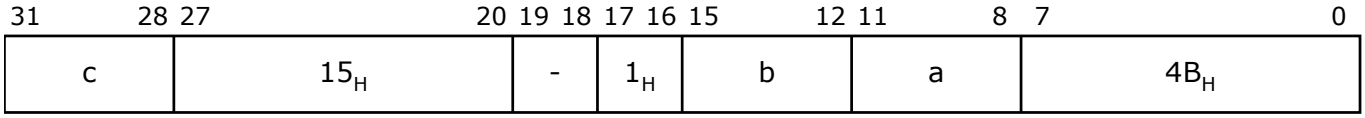
Fraction to Floating-point

Description

Converts the D[a] from Q31 fraction format to single-precision IEEE 754-2019 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]), 2^D[b][8:0]);
rounded_result = sp_round(precise_result, PSW.RM);
result = ieee754_sp_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	if(fp_abs(precise_result) < 2^{-126}) then set_FU = 1 else set_FU = 0; if(set_FU) then PSW.FU = 1;
FX	if(precise_result != sp_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

3.2.22 QSEED.F

Inverse Square Root Seed

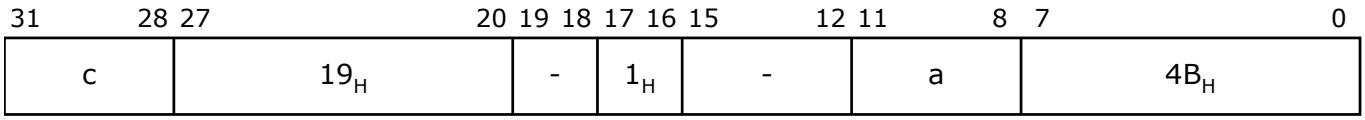
Description

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 ±1% of the accurate result.

The operand and result are single-precision IEEE 754-2019 floating-point numbers. If the operand is ±0 then the result will be the appropriately signed ∞. If the operand is a NaN (quiet or signalling), then the return result will be the quiet NaN $7FC00000_H$.

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 = sp_subnorm_to_zero(sp_real(D[a]));
if(is_sp_nan(D[a])) then
    result = SP_QUIET_NAN;
else if(arg_a == +0.0) then
    result = SP_POS_INFINITY;
else if(arg_a == -0.0) then
    result = SP_NEG_INFINITY;
else if(arg_a < 0.0) then
    result = SP_SQRT_NAN;
else {
    normal_result = approx_inv_sqrt(arg_a);
    result = ieee754_sp_format(normal_result);
}
D[c] = result[31:0];
```

Exception Flags

FS	if(set_FI) then PSW.FS = 1 else PSW.FS = 0;
FI	if(is_sp_s_nan(D[a]) OR (result == SP_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

-

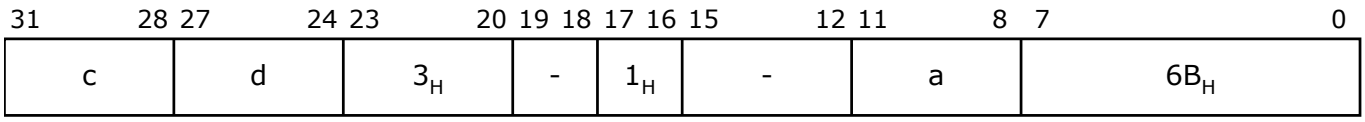
3.2.23 SUB.F

Subtract Float

Description

Subtracts D[a] from D[d] and stores the result in D[c]. The operands and result are single-precision IEEE 754-2019 floating-point numbers.
If any operand is a NaN (quiet or signalling), then the return result will be the quiet NaN $7FC00000_H$.

SUB.F D[c], D[d], D[a] (RRR)




```
arg_a = sp_subnorm_to_zero(sp_real(D[a]);
arg_b = sp_subnorm_to_zero(sp_real(D[d]);
if(is_sp_nan(D[a]) OR is_sp_nan(D[b])) then
    result = SP_QUIET_NAN;
else if(is_sp_pos_inf(D[a]) AND is_sp_pos_inf(D[b])) then
    result = SP_ADD_NAN;
else if(is_sp_neg_inf(D[a]) AND if_sp_neg_inf(D[b])) then
    result = SP_ADD_NAN;
else {
    precise_result = add(-arg_a, arg_b);
    normal_result  = sp_subnorm_to_zero(precise_result);
    rounded_result = sp_round(normal_result, PSW.RM);
    result         = ieee754_sp_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_sp_s_nan(D[a]) OR is_sp_s_nan(D[b]) OR (result == SP_ADD_NAN)) then set_FI = 1 else set_FI = 0; if(set_FI) then PSW.FI = 1;
FV	if(fp_abs(rounded_result) >= 2^{128}) 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^{-126}) then set_FU = 1 else set_FU = 0; if(set_FU) then PSW.FU = 1;
FX	if(precise_result != sp_real(result)) then set_FX = 1 else set_FX = 0; if(set_FX AND !set_FI) then PSW.FX = 1;

Examples

```
sub.f    d3, d1, d2
```

See Also

ADD.F

3.2.24 UPDFL

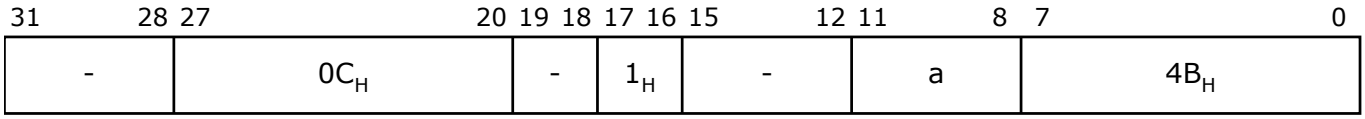
Update Flags

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 +∞, without modifying any of the current exception flag settings, can be accomplished by loading the literal value 0301_H into register D[0], and issuing the instruction, UPDFL D[0].

UPDFL D[a] (RR)




```
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

```
updfl    d1
```

See Also

-

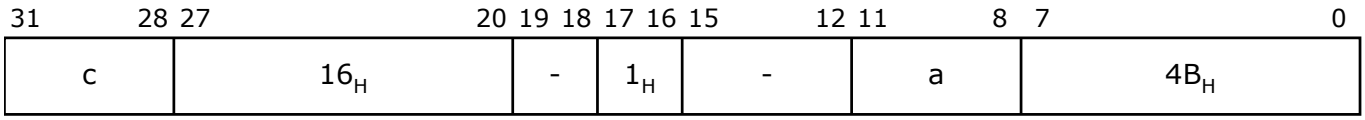
3.2.25 UTOF

Unsigned to Floating-point

Description

Converts the contents of data register D[a] from 32-bit unsigned integer format to single-precision IEEE 754-2019 floating-point format. The rounded result is stored in D[c]. The rounding mode used for the conversion is defined by the PSW.RM field.

UTOF D[c], D[a] (RR)




```
rounded_result = sp_round(u_real(D[a]), PSW.RM);
result = ieee754_sp_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(sp_real(result) != u_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

3.3 Double-precision FPU instructions

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

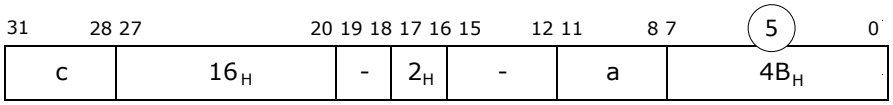
3.x.xxx UTODF (1)

Unsigned to Floating-point (2)

Description (3)

Converts the contents of data register D[a] from 32-bit unsigned integer format to double-precisionIEEE-754-2019 floating-point format. The rounded result is stored in E[c]. The rounding mode used for the conversion is defined by the PSW.RM field.

UTODF E[c], D[a] (RR) (4)




```
rounded_result = dp_round(u_real(D[a]), PSW.RM);
result = ieee754_dp_format(rounded_result);
E[c] = result[63:0];
```
(6)

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. (7)
FZ	Not set by this instruction.
FU	Not set by this instruction.
FX	if(dp_real(result) != u_real(D[a])) then set_FX = 1 else set_FX = 0; if(set_FX) then PSW.FX = 1;

Examples

```
utodf        e2, d1
```
(8)

See also

DFTOU, DFTOUZ (9)

Key:

1. Instruction Mnemonic
2. Instruction Longname
3. Description
4. Syntax, followed by Instruction Format in parentheses
5. Opcodes
6. Operation in RTL format
7. Exception Flags. IEEE-754 Exceptions that can occur when using this instruction
8. Instruction examples
9. Related instructions

3.3.1 ABS.DF

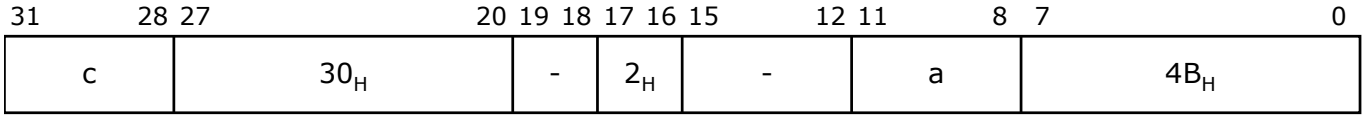
Absolute Value Double

Description

Copies the absolute value of data register E[a] to data register E[c] setting the sign bit to 0 (positive).
The operand and result are treated as double-precision IEEE 754-2019 floating-point numbers.

***Note**: This instruction is a quiet-computational operation, which treats floating-point numbers and NaNs alike and does not signal exceptions.*

ABS.DF E[c], E[a] (RR)




```
E[c][62:0] = E[a][62:0];
E[c][63] = 0;
```

Exception Flags

FS	Not set by this instruction.
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	Not set by this instruction.

Examples

```
abs.df    e4, e0
```

See Also

NEG.DF

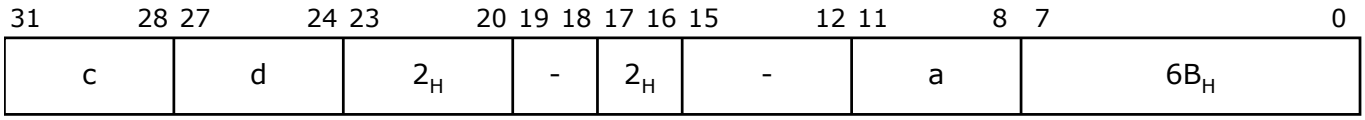
3.3.2 ADD.DF

Add Double

Description

Add the contents of data register E[a] to the contents of data register E[d]. Put the result in data register E[c]. The operands and result are double-precision IEEE 754-2019 floating-point numbers. If either operand is a NaN (quiet or signalling), then the return result will be the quiet NaN $7FF8000000000000_H$.

ADD.DF E[c], E[d], E[a] (RRR)




```
arg_a = dp_real(E[a]);
arg_b = dp_real(E[d]);
if(is_dp_nan(E[a]) OR is_dp_nan(E[d])) then
    result = DP_QUIET_NAN;
else if(is_dp_pos_inf(E[a]) AND is_dp_neg_inf(E[d])) then
    result = DP_ADD_NAN;
else if(is_dp_neg_inf(E[a]) AND is_dp_pos_inf(E[d])) then
    result = DP_ADD_NAN;
else {
    precise_result = add(arg_a,arg_b);
    rounded_result = dp_round(precise_result, PSW.RM);
    result = ieee754_dp_format(rounded_result);
}
E[c] = result[63:0];
```

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_dp_s_nan(E[a]) OR is_dp_s_nan(E[d]) OR (result == DP_ADD_NAN) then set_FI = 1 else set_FI = 0; if(set_FI) then PSW.FI = 1;
FV	if(fp_abs(rounded_result) >= 2^{1024}) 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^{-1022}) then set_FU = 1 else set_FU = 0; if(set_FU) then PSW.FU = 1;
FX	if(precise_result != dp_real(result)) then set_FX = 1 else set_FX = 0; if(set_FX AND !set_FI) then PSW.FX = 1;

Examples

```
add.df    e4, e0, e2
```

See Also

SUB.DF

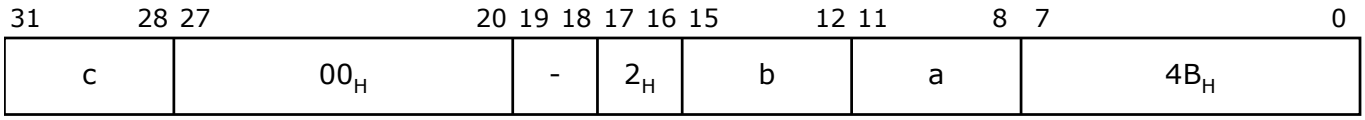
3.3.3 CMP.DF

Compare Double

Description

This instruction compares the double-precision IEEE 754-2019 floating-point operands and asserts bits in the result if their associated condition is true:
bit [0] E[a] < E[b]
bit [1] E[a] == E[b]
bit [2] E[a] > E[b]
bit [3] Unordered
bit [4] E[a] is subnormal
bit [5] E[b] is subnormal
bits[31:6] are cleared.
The 'unordered' bit is asserted if either operand is a NaN.

CMP.DF D[c], E[a], E[b] (RR)




```
D[c][0] = dp_lt(E[a], E[b]);
D[c][1] = dp_eq(E[a], E[b]);
D[c][2] = dp_gt(E[a], E[b]);
D[c][3] = (is_dp_nan(E[a]) OR is_dp_nan(E[b]));
D[c][4] = is_dp_subnorm(E[a]);
D[c][5] = is_dp_subnorm(E[b]);
D[c][31:6] = 0;
```

Exception Flags

FS	if(set_FI) then PSW.FS = 1 else PSW.FS = 0;
FI	if(is_dp_s_nan(E[a]) OR is_dp_s_nan(E[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.df    d4, e0, e2
```

See Also

-

3.3.4 DIV.DF

Divide Double

Description

Divides the contents of data register E[a] by the contents of data register E[b] and put the result in data register E[c]. The operands and result are double-precision IEEE 754-2019 floating-point numbers. If either operand is a NaN (quiet or signalling), then the return result will be the quiet NaN $7FF8000000000000_H$.

DIV.DF E[c], E[a], E[b] (RR)

31 – 28	27 – 20	19 – 18	17 – 16	15 – 12	11 – 8	7 – 0
c	05_H	-	2_H	b	a	$4B_H$

```
arg_a = dp_real(E[a]);
arg_b = dp_real(E[b]);
if(is_dp_nan(E[a]) OR is_dp_nan(E[b])) then
    result = DP_QUIET_NAN;
else if(is_dp_inf(E[a]) AND is_dp_inf(E[b])) then
    result = DP_DIV_NAN;
else if((arg_a == 0.0) AND (arg_b == 0.0)) then
    result = DP_DIV_NAN;
else {
    precise_result = divide(arg_a, arg_b);
    rounded_result = dp_round(precise_result, PSW.RM);
    result = ieee754_dp_format(rounded_result);
}
E[c] = result[63:0];
```

Exception Flags

FS	if(set_FI OR set_FV OR set_FZ OR set_FU OR set_FX) then PSW.FS = 1 else PSW.FS = 0;
FI	if(is_dp_s_nan(E[a]) OR is_dp_s_nan(E[b]) OR (result == DP_DIV_NAN)) then set_FI = 1 else set_FI = 0; if(set_FI) then PSW.FI = 1;
FV	if(fp_abs(rounded_result) >= 2^{1024}) then set_FV = 1 else set_FV = 0; if(set_FV) then PSW.FV = 1;
FZ	if(is_dp_zero(E[b]) AND !is_dp_inf(E[a]) AND !is_dp_zero(E[a]) AND !is_dp_nan(E[a])) then set_FZ = 1 else set_FZ = 0; if(set_FZ) then PSW.FZ = 1;
FU	if(fp_abs(precise_result) < 2^{-1022}) then set_FU = 1 else set_FU = 0; if(set_FU) then PSW.FU = 1;
FX	if(precise_result != dp_real(result)) then set_FX = 1 else set_FX = 0; if(set_FX AND !set_FI AND !set_FZ) then PSW.FX = 1;

Examples

```
div.df    e4, e0, e2
```

See Also

-

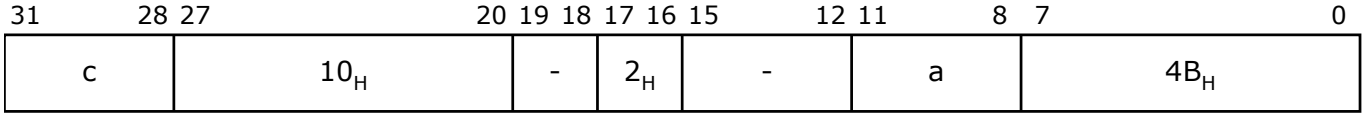
3.3.5 DFTOI

Double to Integer

Description

Converts the contents of data register E[a] from double-precision IEEE 754-2019 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.

DFTOI D[c], E[a] (RR)




```
if(is_dp_nan(E[a])) then
    result = 0;
else if(dp_real(E[a]) > 2^31-1) then
    result = 7FFFFFFF_H;
else if(dp_real(E[a]) < -2^31) then
    result = 80000000_H;
else {
    result = dp_round_to_integer(E[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((dp_real(E[a]) > 2^{31}-1) OR (dp_real(E[a]) < -2^{31}) OR is_dp_nan(E[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(dp_real(E[a]) != i_real(result)) then set_FX = 1 else set_FX = 0; if(set_FX AND !set_FI) then PSW.FX = 1;

Examples

```
dftoi    d2, e10
```

See Also

ITODF, DFTOIN, DFTOIZ

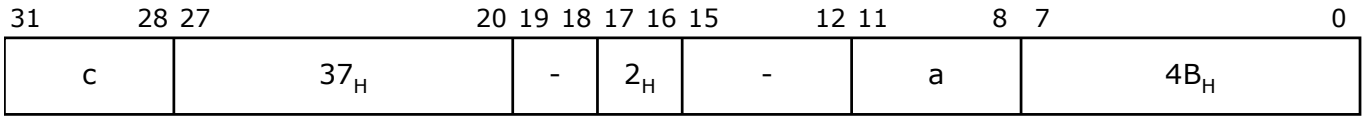
3.3.6 DFTOIN

Double to Integer, Round to Nearest

Description

Converts the contents of data register E[a] from double-precision IEEE 754-2019 floating-point format to a 32-bit two's complement signed integer format. The rounded resut is put in data register D[c]. The rounding mode used for the conversion is round to nearest.

DFTOIN D[c], E[a] (RR)




```
if(is_dp_nan(E[a])) then
    result = 0;
else if(dp_real(E[a]) > 2^31-1) then
    result = 7FFFFFFF_H;
else if(dp_real(D[a]) < -2^31) then
    result = 80000000_H;
else {
    result = dp_round_to_integer(E[a], 00_B);
}
D[c] = result[31:0];
```

Exception Flags

FS	if(set_FI OR set_FX) then PSW.FS = 1 else PSW.FS = 0;
FI	if((dp_real(D[a]) > 2^{31}-1) OR (dp_real(D[a]) < -2^{31}) OR is_dp_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(dp_real(E[a]) != i_real(result)) then set_FX = 1 else set_FX = 0; if(set_FX AND !set_FI) then PSW.FX = 1;

Examples

```
dftoin    d2, e0
```

See Also

ITODF, DFTOI, DFTOIZ

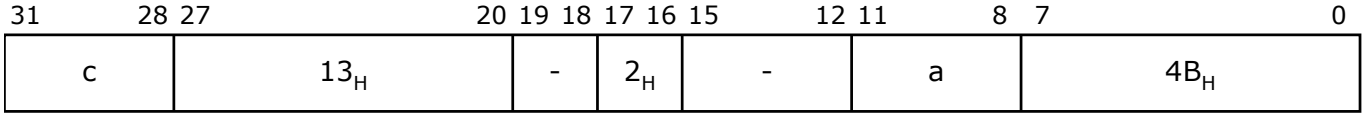
3.3.7 DFTOIZ

Double to Integer, Round towards Zero

Description

Converts the contents of data register E[a] from double-precision IEEE 754-2019 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 round towards zero.

DFTOIZ D[c], E[a] (RR)




```
if(is_dp_nan(E[a])) then
    result = 0;
else if(dp_real(E[a]) > 2^31-1) then
    result = 7FFFFFFF_H;
else if(dp_real(D[a]) < -2^31) then
    result = 80000000_H;
else {
    result = dp_round_to_integer(E[a], 11_B);
}
D[c] = result[31:0];
```

Exception Flags

FS	if(set_FI OR set_FX) then PSW.FS = 1 else PSW.FS = 0;
FI	if((dp_real(E[a]) > 2^{31}-1) OR (dp_real(E[a]) < -2^{31}) OR is_dp_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(dp_real(E[a]) != i_real(result)) then set_FX = 1 else set_FX = 0; if(set_FX AND !set_FI) then PSW.FX = 1;

Examples

```
dftoiz    d2, e10
```

See Also

ITODF, DFTOI, DFTOIN

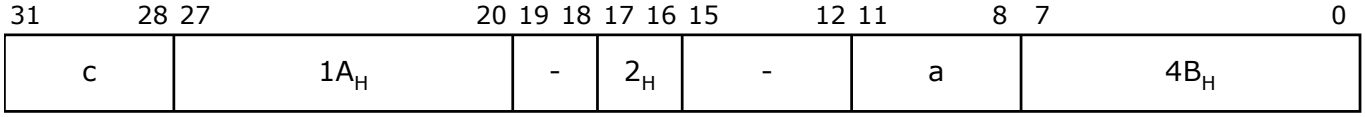
3.3.8 DFTOL

Double to Long Integer

Description

Converts the contents of data register E[a] from double-precision IEEE 754-2019 floating-point format to a 64-bit two's complement signed long integer format. The rounded result is put in data register E[c]. The rounding mode used for the conversion is defined by the PSW.RM field.

DFTOL E[c], E[a] (RR)




```
if(is_dp_nan(E[a])) then
    result = 0;
else if(dp_real(E[a]) > 2^63-1) then
    result = 7FFFFFFFFFFFFFFF_H;
else if(dp_real(E[a]) < -2^63) then
    result = 8000000000000000_H;
else {
    result = dp_round_to_long_integer(E[a], PSW.RM);
}
E[c] = result[33:0];
```

Exception Flags

FS	if(set_FI OR set_FX) then PSW.FS = 1 else PSW.FS = 0;
FI	if((dp_real(E[a]) > 2^{31}-1) OR (dp_real(E[a]) < -2^{31}) OR is_dp_nan(E[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(dp_real(E[a]) != l_real(result)) then set_FX = 1 else set_FX = 0; if(set_FX AND !set_FI) then PSW.FX = 1;

Examples

```
dftol    e2, e0
```

See Also

DFTOLZ, LTODF

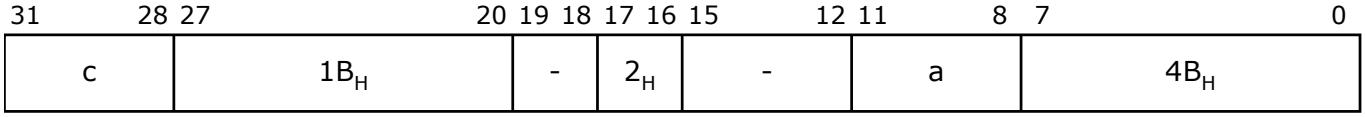
3.3.9 DFTOLZ

Double to Long Integer, Round towards Zero

Description

Converts the contents of data register E[a] from double-precision IEEE 754-2019 floating-point format to a 64-bit two's complement signed integer format. The rounded result is put in data register E[c]. The rounding mode used for the conversion is round towards zero.

DFTOLZ E[c], E[a] (RR)




```
if(is_dp_nan(E[a])) then
    result = 0;
else if(dp_real(E[a]) > 2^63-1) then
    result = 7FFFFFFFFFFFFFFF_H;
else if(dp_real(E[a]) < -2^63) then
    result = 8000000000000000_H;
else {
    result = dp_round_to_long_integer(E[a], 11_B);
}
E[c] = result[63:0];
```

Exception Flags

FS	if(set_FI OR set_FX) then PSW.FS = 1 else PSW.FS = 0;
FI	if((dp_real(E[a]) > 2^{31}-1) OR (dp_real(E[a]) < -2^{31}) OR is_dp_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(dp_real(E[a]) != l_real(result)) then set_FX = 1 else set_FX = 0; if(set_FX AND !set_FI) then PSW.FX = 1;

Examples

```
dftolz    e2, e10
```

See Also

DFTOL, LTODF

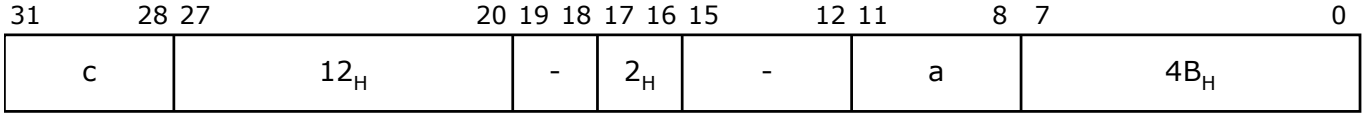
3.3.10 DFTOU

Double to Unsigned

Description

Converts the contents of data register E[a] from double-precision IEEE 754-2019 floating-point format to a 32-bit unsigned 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.

DFTOU D[c], E[a] (RR)




```
if(dp_nan(E[a])) then
    result = 0;
else if(dp_real(E[a]) > 2^32-1) then
    result = FFFFFFFFH;
else if(dp_real(E[a]) < 0.0) then
    result = 0;
else {
    result = dp_round_to_unsigned(E[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((dp_real(E[a]) > 2^{31}-1) OR (dp_real(E[a]) < 0.0) OR is_dp_nan(E[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(dp_real(E[a]) != u_real(result)) then set_FX = 1 else set_FX = 0; if(set_FX AND !set_FI) then PSW.FX = 1;

Examples

```
dftou    d2, e0
```

See Also

UTODF, DFTOUZ

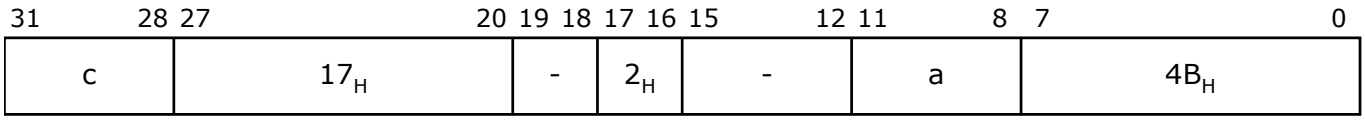
3.3.11 DFTOUZ

Double to Unsigned, Round towards Zero

Description

Converts the contents of data register E[a] from double-precision IEEE 754-2019 floating-point format to a 32-bit unsigned integer format. The rounded result is put in data register D[c]. The rounding mode used for the conversion is round towards zero.

DFTOUZ D[c], E[a] (RR)




```
if(is_dp_nan(E[a])) then
    result = 0;
else if(dp_real(E[a]) > 2^64-1) then
    result = FFFFFFFF_H;
else if(dp_real(E[a]) < 0.0) then
    result = 0;
else {
    result = dp_round_to_unsigned(E[a], 11_B);
}
D[c] = result[31:0];
```

Exception Flags

FS	if(set_FI OR set_FX) then PSW.FS = 1 else PSW.FS = 0;
FI	if((dp_real(E[a]) > 2^{32}-1) OR (dp_real(E[a]) < 0.0) OR is_dp_nan(E[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(dp_real(E[a]) != u_real(result)) then set_FX = 1 else set_FX = 0; if(set_FX AND !set_FI) then PSW.FX = 1;

Examples

```
dftouz    d2, e0
```

See Also

UTODF, DFTOU

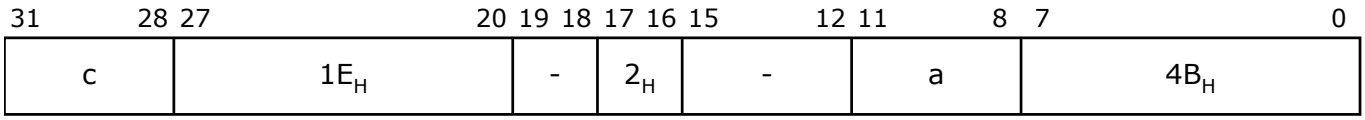
3.3.12 DFTOUL

Double to Unsigned Long

Description

Converts the contents of data register E[a] from double-precision IEEE 754-2019 floating-point format to a 64-bit unsigned integer format. The rounded result is put in data register E[c]. The rounding mode used for the conversion is defined by the PSW.RM field.

DFTOUL E[c], E[a] (RR)




```
if(is_dp_nan(E[a])) then
    result = 0;
else if(dp_real(E[a]) > 2^64-1) then
    result = FFFFFFFFFFFFFFFF_H;
else if(dp_real(E[a]) < 0.0) then
    result = 0;
else {
    result = dp_round_to_unsigned_long(E[a], PSW.RM);
}
E[c] = result[63:0];
```

Exception Flags

FS	if(set_FI OR set_FX) then PSW.FS = 1 else PSW.FS = 0;
FI	if((dp_real(E[a]) > 2^{31}-1) OR (dp_real(E[a]) < 0.0) OR is_dp_nan(E[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(dp_real(E[a]) != ul_real(result)) then set_FX = 1 else set_FX = 0; if(set_FX AND !set_FI) then PSW.FX = 1;

Examples

```
dftoul    e0, e2
```

See Also

DFTOULZ, ULTODF

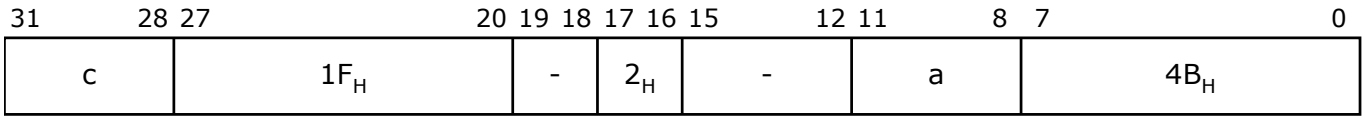
3.3.13 DFTOULZ

Double to Unsigned Long Integer, Round towards Zero

Description

Converts the contents of data register E[a] from double-precision IEEE 754-2019 floating-point format to a 64-bit unsigned integer format. The rounded result is put in data register E[c]. The rounding mode used for the conversion is round towards zero.

DFTOULZ E[c], E[a] (RR)




```
if(is_dp_nan(E[a])) then
    result = 0;
else if(dp_real(E[a]) > 2^64-1) then
    result = FFFFFFFFFFFFFFFF_H;
else if(dp_real(E[a]) < 0.0) then
    result = 0;
else
    result = dp_round_to_unsigned_long(E[a], 11_B);
}
E[c] = result[63:0];
```

Exception Flags

FS	if(set_FI OR set_FX) then PSW.FS = 1 else PSW.FS = 0;
FI	if((dp_real(E[a]) > 2^{32}-1) OR (dp_real(E[a]) < 0.0) OR is_dp_nan(E[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(dp_real(E[a]) != ul_real(result)) then set_FX = 1 else set_FX = 0; if(set_FX AND !set_FI) then PSW.FX = 1;

Examples

```
dftoulz    e6, e8
```

See Also

DFTOUL, ULTODF

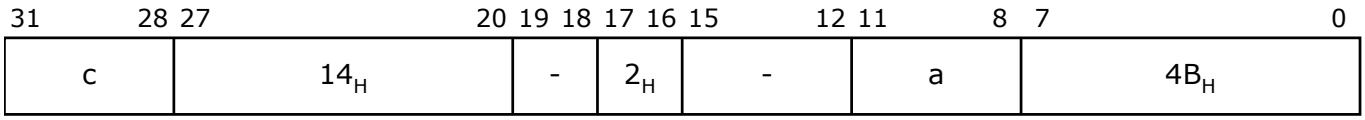
3.3.14 ITODF

Integer to Double

Description

Converts the contents of data register D[a] from 32-bit two's complement signed integer format to double-precision IEEE 754-2019 floating-point format. The rounded result is put in data register E[c]. The rounding mode used for the conversion is defined by the PSW.RM field.

ITODF E[c], D[a] (RR)




```
rounded_result = dp_round(i_real(D[a]), PSW.RM);
result = ieee754_dp_format(rounded_result);
E[c] = result[63:0];
```

Exception Flags

FS	Not set by this instruction.
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	Not set by this instruction.

Examples

```
itodf    e2, d0
```

See Also

DFTOI, DFTOIN, DFTOIZ

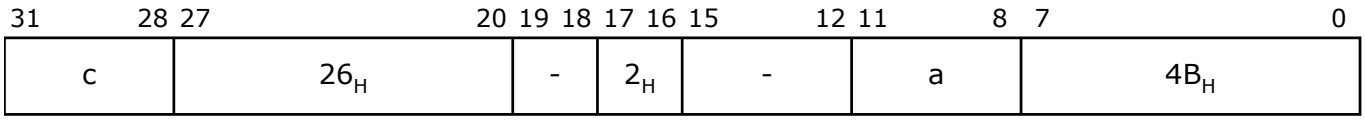
3.3.15 LTODF

Long Integer to Double

Description

Converts the contents of data register E[a] from 64-bit two's complement signed long integer format to double-precision IEEE 754-2019 floating-point format. The rounded result is put in data register E[c]. The rounding mode used for the conversion is defined by the PSW.RM field.

LTODF E[c], E[a] (RR)




```
rounded_result = dp_round(l_real(E[a]), PSW.RM);
result = ieee754_dp_format(rounded_result);
E[c] = result[63:0];
```

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(dp_real(result) != l_real(E[a])) then set_FX = 1 else set_FX = 0; if(set_FX) then PSW.FX = 1;

Examples

```
ltodf    e4, e10
```

See Also

DFTOL, DFTOLZ

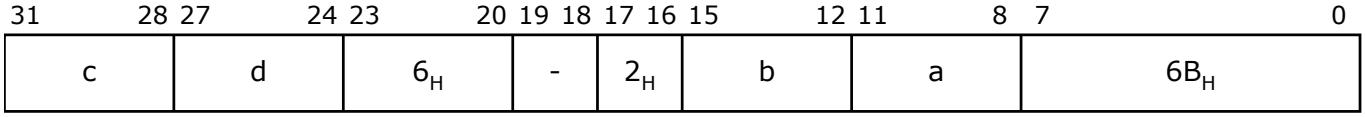
3.3.16 MADD.DF

Multiply Add Double

Description

Multiplies E[a] and E[b] and adds the product to E[d]. The result is put in E[c]. The operands and result are double-precision IEEE 754-2019 floating-point numbers. If an operand is a NaN (quiet or signalling), then the return result will be the quiet NaN $7FF8000000000000_H$.

MADD.DF E[c], E[d], E[a], E[b] (RRR)




```
arg_a = dp_real(E[a]);
arg_b = dp_real(E[b]);
arg_c = dp_real(E[d]);
if(is_dp_nan(E[a]) OR is_dp_nan(E[b]) OR is_dp_nan(E[d])) then
    result = DP_QUIET_NAN;
else if(is_dp_inf(E[a]) AND is_dp_zero(E[b])) then
    result = DP_MUL_NAN;
else if(is_dp_zero(E[a]) AND is_dp_inf(E[b])) then
    result = DP_MUL_NAN;
else if((   (is_dp_pos_inf(E[a]) AND !dp_sign_bit(E[b]))
         OR (is_dp_neg_inf(E[a]) AND  dp_sign_bit(E[b]))
         OR (!dp_sign_bit(E[a])  AND is_dp_pos_inf(E[b]))
         OR ( dp_sign_bit(E[a])  AND is_dp_neg_inf(E[b]))
        ) AND is_dp_neg_inf(E[d]) ) then
    result = DP_ADD_NAN;
else if((   (is_dp_pos_inf(E[a]) AND  dp_sign_bit(E[b]))
         OR (is_dp_neg_inf(E[a]) AND !dp_sign_bit(E[b]))
         OR ( dp_sign_bit(E[a])  AND is_dp_pos_inf(E[b]))
         OR (!dp_sign_bit(E[a])  AND is_dp_neg_inf(E[b]))
        ) AND is_dp_pos_inf(E[d])) then
    result = DP_ADD_NAN;
else {
    precise_mul_result = mul(arg_a, arg_b);
    precise_result = add(precise_mul_result, arg_c);
    rounded_result = dp_round(precise_result, PSW.RM);
    result = ieee754_dp_format(rounded_result);
}
E[c] = result[63:0];
```

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_dp_s_nan(E[a]) OR is_dp_s_nan(E[b]) OR is_dp_s_nan(E[d]) OR (result == DP_ADD_NAN) OR (result == DP_MUL_NAN)) then set_FI = 1 else set_FI = 0; if(set_FI) then PSW.FI = 1;
FV	if(fp_abs(rounded_result) >= 2^{2048}) 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^{-2046}) then set_FU = 1 else set_FU = 0; if(set_FU) then PSW.FU = 1;
FX	if(precise_result != dp_real(result)) then set_FX = 1 else set_FX = 0; if(set_FX AND !set_FI) then PSW.FX = 1;

Examples

```
madd.df    e6, e4, e0, e2
```

See Also

MSUB.DF, MUL.F

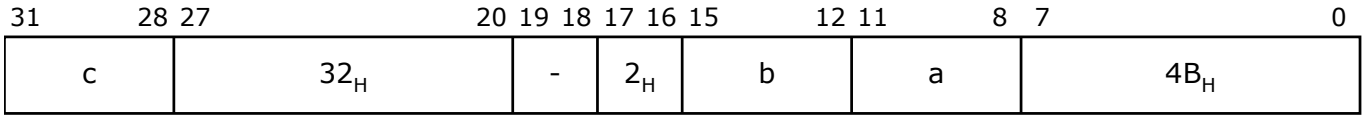
3.3.17 MAX.DF

Maximum Value Double

Description

If the content of data register E[a] is greater than or equal to the content of data register E[b], then put the content of E[a] in data register E[c]; otherwise put the content of E[b] in E[c]. The operands and result are treated as double-precision IEEE 754-2019 floating-point. For this operation +0 compares greater than -0. If only one argument is a NaN the number will be returned in E[c]. If both operands are NaN (quiet or signaling), then the return result will be the quiet NaN $7FF8000000000000_H$.

MAX.DF E[c], E[a], E[b] (RR)




```
if (is_dp_nan(E[a]) AND is_dp_nan(E[b])) then
    result = DP_QUIET_NAN;
else if (is_dp_nan(E[a])) then
    result = E[b];
else if (is_dp_nan(E[b])) then
    result = E[a];
else if (E[a] >= E[b]) then
    result = E[a];
else {
    result = E[b];
}
E[c][63:0] = result;
```

Exception Flags

FS	if(set_FI) then PSW.FS = 1 else PSW.FS = 0;
FI	if(is_dp_s_nan(D[a]) OR is_dp_s_nan(D[b])) 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

```
max.df    e4, e0, e2
```

See Also

MIN.DF

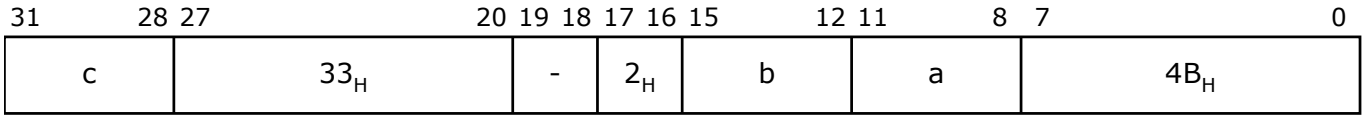
3.3.18 MIN.DF

Minumum Value Double

Description

If the content of data register E[a] is lesser than or equal to the content of data register E[b], then put the content of E[a] in data register E[c]; otherwise put the content of E[b] in E[c]. The operands and result are treated as dboule-precision IEEE 754-2019 floating-point. For this operation -0 compares lesser than +0. If only one argument is a NaN the number will be returned in E[c]. If both operands are NaN (quiet or signaling), then the return result will be the quiet NaN $7FF8000000000000_H$.

MIN.DF E[c], E[a], E[b] (RR)




```
if (is_dp_nan(E[a]) AND is_dp_nan(E[b])) then
    result = DP_QUIET_NAN;
else if (is_dp_nan(E[a])) then
    result = E[b];
else if (is_dp_nan(E[b])) then
    result = E[a];
else if (E[a] <= E[b]) then
    result = E[a];
else {
    result = E[b];
}
E[c][63:0] = result;
```

Exception Flags

FS	if(set_FI) then PSW.FS = 1 else PSW.FS = 0;
FI	if(is_dp_s_nan(D[a]) OR is_dp_s_nan(D[b])) 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

```
min.df    e4, e2, e6
```

See Also

MAX.DF

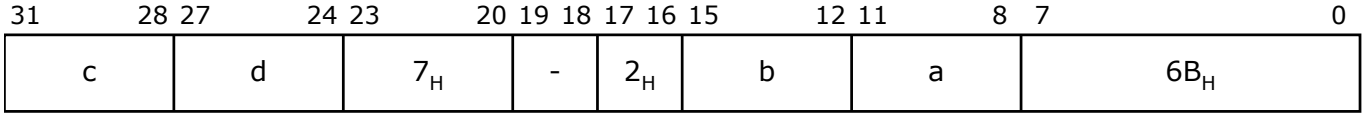
3.3.19 MSUB.DF

Multiply Subtract Double

Description

Multiplies E[a] and E[b] and subtracts the product from E[d], putting the result in E[c]. The operands and result are double-precision IEEE 754-2019 floating-point numbers. If any operand is a NaN (quiet or signalling), then the return result will be the quiet NaN $7FF8000000000000_H$.

MSUB.DF E[c], E[d], E[a], E[b] (RRR)




```
arg_a = dp_real(E[a]);
arg_b = dp_real(E[b]);
arg_c = dp_real(E[d]);
if(is_dp_nan(E[a]) OR is_dp_nan(E[b]) OR is_dp_nan(E[d])) then
    result = DP_QUIET_NAN;
else if(is_dp_inf(E[a]) AND is_dp_zero(E[b])) then
    result = DP_MUL_NAN;
else if(is_dp_zero(E[a]) AND is_dp_inf(E[b])) then
    result = DP_MUL_NAN;
else if((   (is_dp_pos_inf(E[a]) AND !dp_sign_bit(E[b]))
         OR (is_dp_neg_inf(E[a]) AND  dp_sign_bit(E[b]))
         OR (!dp_sign_bit(E[a])  AND is_dp_pos_inf(E[b]))
         OR ( dp_sign_bit(E[a])  AND is_dp_neg_inf(E[b]))
        ) AND is_dp_pos_inf(E[d]) ) then
    result = DP_ADD_NAN;
else if((   (is_dp_pos_inf(E[a]) AND  dp_sign_bit(E[b]))
         OR (is_dp_neg_inf(E[a]) AND !dp_sign_bit(E[b]))
         OR ( dp_sign_bit(E[a])  AND is_dp_pos_inf(E[b]))
         OR (!dp_sign_bit(E[a])  AND is_dp_neg_inf(E[b]))
        ) AND is_dp_neg_inf(E[d])) then
    result = DP_ADD_NAN;
else {
    precise_mul_result = mul(arg_a, arg_b);
    precise_result = add(-precise_mul_result, arg_c);
    rounded_result = dp_round(precise_result, PSW.RM);
    result = ieee754_dp_format(rounded_result);
}
E[c] = result[63:0];
```

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_dp_s_nan(E[a]) OR is_dp_s_nan(E[b]) OR is_dp_s_nan(E[d]) OR (result == DP_ADD_NAN) OR (result == DP_MUL_NAN)) then set_FI = 1 else set_FI = 0; if(set_FI) then PSW.FI = 1;
FV	if(fp_abs(rounded_result) >= 2^{1024}) 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^{-1022}) then set_FU = 1 else set_FU = 0; if(set_FU) then PSW.FU = 1;
FX	if(precise_result != dp_real(result)) then set_FX = 1 else set_FX = 0; if(set_FX AND !set_FI) then PSW.FX = 1;

Examples

```
msub.df    e6, e4, e0, e2
```

See Also

MADD.DF

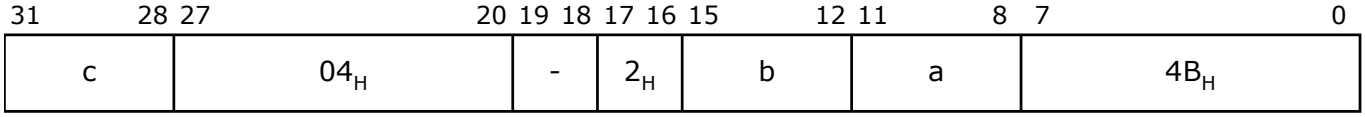
3.3.20 MUL.DF

Multiply Double

Description

Multiplies E[a] and E[b] and stores the result in E[c]. The operands and result are double-precision IEEE 754-2019 floating-point numbers. If an operand is a NaN (quiet or signalling), then the return result will be the quiet NaN $7FF8000000000000_H$.

MUL.DF E[c], E[a], E[b] (RR)




```
arg_a = dp_real(E[a]);
arg_b = dp_real(E[b]);
if(is_dp_nan(E[a]) OR is_dp_nan(E[b])) then;
    result = DP_QUIET_NAN;
else if(is_dp_inf(E[a]) AND is_dp_zero(E[b])) then
    result = DP_MUL_NAN;
else if(is_dp_inf(E[b]) AND is_dp_zero(E[a])) then
    result = DP_MUL_NAN;
else {
    precise_result = mul(arg_a, arg_b);
    rounded_result = dp_round(precise_result, PSW.RM);
    result = ieee754_dp_format(rounded_result);
}
E[c] = result[63:0];
```

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_dp_s_nan(E[a]) OR is_dp_s_nan(E[b]) OR (result == DP_MUL_NAN)) then set_FI = 1 else set_FI = 0; if(set_FI) then PSW.FI = 1;
FV	if(fp_abs(rounded_result) >= 2^{1024}) 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^{-1022}) then set_FU = 1 else set_FU = 0; if(set_FU) then PSW.FU = 1;
FX	if(precise_result != dp_real(result)) then set_FX = 1 else set_FX = 0; if(set_FX AND !set_FI) then PSW.FX = 1;

Examples

```
mul.df    e4, e0, e2
```

See Also

-

3.3.21 NEG.DF

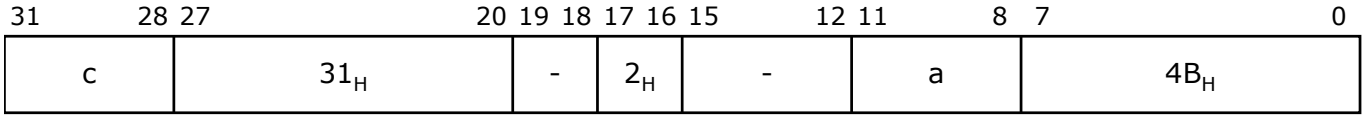
Negate Value Double

Description

Copies the value of data register E[a] to data register E[c] reversing the sign bit.
The operand and result are treated as double-precision IEEE 754-2019 floating-point numbers.

***Note**: This instruction is a quiet-computational operation, which treats floating-point numbers and NaNs alike and does not signal exceptions.*

NEG.DF E[c], E[a] (RR)




```
E[c][62:0] = D[a][62:0]
if (E[a][63] == 1) then
    E[c][63] = 0;
else
    E[c][63] = 1;
```

Exception Flags

FS	Not set by this instruction.
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	Not set by this instruction.

Examples

```
neg.df    e2, e0
```

See Also

ABS.DF

3.3.22 QSEED.DF

Inverse Square Root Seed

Description

An approximation of the reciprocal of the square root of E[a] is stored in E[c]. The accuracy of the result is no less than 6.75 bits, and therefore always within ±1% of the accurate result.

The operand and result are double-precision IEEE 754-2019 floating-point numbers. If the operand is ±0 then the result will be the appropriately signed ∞. If the operand is a NaN (quiet or signalling), then the return result will be the quiet NaN $7FF8000000000000_H$.

This instruction can be used to implement a double-precision floating-point square root function in software using the Newton-Raphson iterative method.

QSEED.DF E[c], E[a] (RR)

31 – 28	27 – 20	19	18	17 – 16	15 – 12	11 – 8	7 – 0
c	19_H	-		2_H	-	a	$4B_H$

```
arg_a = dp_real(E[a]);
if(is_dp_nan(E[a])) then
    result = DP_QUIET_NAN;
else if(arg_a == +0.0) then
    result = DP_POS_INFINITY;
else if(arg_a == -0.0) then
    result = DP_NEG_INFINITY;
else if(arg_a < 0.0) then
    result = DP_SQRT_NAN;
else {
    normal_result = approx_inv_sqrt(arg_a);
    result = ieee754_dp_format(normal_result);
}
E[c] = result[63:0];
```

Exception Flags

FS	if(set_FI) then PSW.FS = 1 else PSW.FS = 0;
FI	if(is_dp_s_nan(E[a]) OR (result == DP_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.df    e2, e0
```

See Also

-

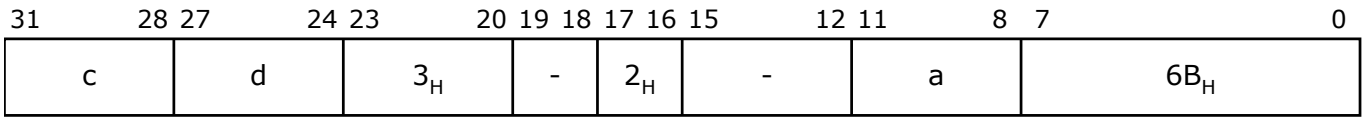
3.3.23 SUB.DF

Subtract Double

Description

Subtracts E[a] from E[d] and stores the result in E[c]. The operands and result are double-precision IEEE 754-2019 floating-point numbers.
If any operand is a NaN (quiet or signalling), then the return result will be the quiet NaN $7FF8000000000000_H$.

SUB.DF E[c], E[d], E[a] (RRR)




```
arg_a = dp_real(E[a]);
arg_b = dp_real(E[d]);
if(is_dp_nan(E[a]) OR is_dp_nan(E[b])) then
    result = DP_QUIET_NAN;
else if(is_dp_pos_inf(E[a]) AND is_dp_pos_inf(E[b])) then
    result = DP_ADD_NAN;
else if(is_dp_neg_inf(E[a]) AND is_dp_neg_inf(E[b])) then
    result = DP_ADD_NAN;
else {
    precise_result = add(-arg_a, arg_b);
    rounded_result = dp_round(precise_result, PSW.RM);
    result = ieee754_dp_format(rounded_result);
}
E[c] = result[63:0];
```

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_dp_s_nan(E[a]) OR is_dp_s_nan(E[b]) OR (result == DP_ADD_NAN)) then set_FI = 1 else set_FI = 0; if(set_FI) then PSW.FI = 1;
FV	if(fp_abs(rounded_result) >= 2^{1024}) 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^{-1022}) then set_FU = 1 else set_FU = 0; if(set_FU) then PSW.FU = 1;
FX	if(precise_result != dp_real(result)) then set_FX = 1 else set_FX = 0; if(set_FX AND !set_FI) then PSW.FX = 1;

Examples

```
sub.df    e4, e0, e2
```

See Also

ADD.DF

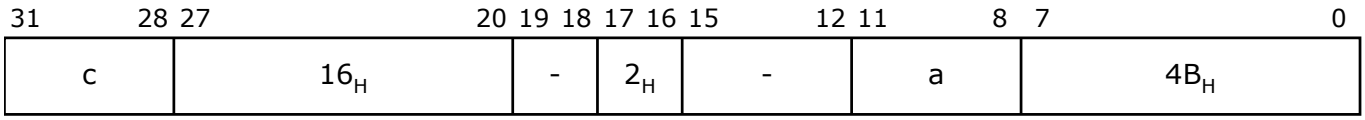
3.3.24 UTODF

Unsigned to Double

Description

Converts the contents of data register D[a] from 32-bit unsigned integer format to double-precision IEEE 754-2019 floating-point format. The rounded result is stored in E[c]. The rounding mode used for the conversion is defined by the PSW.RM field.

UTODF E[c], D[a] (RR)




```
rounded_result = dp_round(u_real(D[a]), PSW.RM);
result = ieee754_dp_format(rounded_result);
E[c] = result[63:0];
```

Exception Flags

FS	Not set by this instruction.
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	Not set by this instruction.

Examples

```
utodf    E2, d10
```

See Also

DFTOU, DFTOUZ

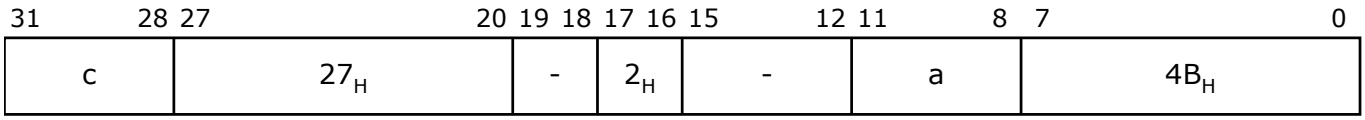
3.3.25 ULTODF

Unsigned Long Integer to Double

Description

Converts the contents of data register E[a] from 64-bit unsigned long integer format to double-precision IEEE 754-2019 floating-point format. The rounded result is stored in E[c]. The rounding mode used for the conversion is defined by the PSW.RM field.

ULTODF E[c], E[a] (RR)




```
rounded_result = dp_round(ul_real_64bit(E[a]), PSW.RM);
result = ieee754_dp_format(rounded_result);
E[c] = result[63:0];
```

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(dp_real(result) != ul_real(E[a])) then set_FX = 1 else set_FX = 0; if(set_FX) then PSW.FX = 1;

Examples

```
ultodf    e10, e12
```

See Also

DFTOUL, DFTOULZ

3.3.26 DFTOF

Double Precision to Single Precision

Description

Convert the contents of data register E[a] from double-precision IEEE 754-2019 floating-point to single-precision IEEE 754-2019 floating-point 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.

If E[a] contains a NaN value, the result in D[c] will also contain a NaN. In order to maintain both the distinction between signalling and quiet NaNs, and the quiet NaN invalid operation code, the top two output mantissa bits are set to the top two input mantissa bits and the remaining 21 output mantissa bits are set to the bottom 21 input mantissa bits, whenever E[a] contains a NaN value. In the case when no output mantissa bits would be set on an input NaN, D[c][21] is set to maintain the NaN value.

DFTOF D[c], E[a] (RR)

31 – 28	27 – 20	19 – 18	17 – 16	15 – 12	11 – 8	7 – 0
c	28_H	-	2_H	-	a	$4B_H$

```
if (is_dp_inf(E[a])) then {
    if(dp_sign_bit(E[a])) then {
        result = NEG_INFINITY;
    } else {
        result = POS_INFINITY;
    }
} else if (is_dp_nan_dp(E[a])) then {
    result[31]    = dp_sign_bit(E[a]);
    result[30:23] = FF_H;
    result[22:21] = E[a][51:50];
    result[20:0]  = E[a][20:0];
    if ((result[22:0] == 0)) then {
        result[21] = 1_B;
    }
} else {
    precise_result = dp_real(E[a]);
    rounded_result = sp_round(precise_result, PSW.RM);
    result = ieee754_sp_format(rounded_result);
}
D[c] = result;
```

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_dp_s_nan(E[a]) then set_FI = 1 else set_FI = 0; if(set_FI) then PSW.FI = 1;
FV	if(fp_abs(rounded_result) >= 2^{128}) 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(rounded_result) < 2^{-126})) then set_FU = 1 else set_FU = 0; if(set_FU) then PSW.FU = 1;
FX	if(precise_result != sp_real(result)) then set_FX = 1 else set_FX = 0; if(set_FX AND !set_FI) then PSW.FX = 1;

Examples

```
dftof    d1, e2
```

See Also

FTODF

3.3.27 FTODF

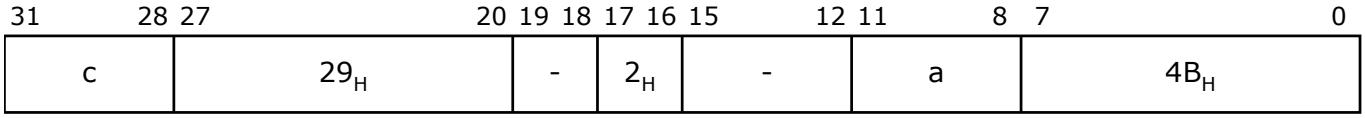
Single Precision to Double Precision

Description

Convert the contents of data register D[a] from single-precision IEEE 754-2019 floating-point to double-precision IEEE 754-2019 floating-point format and put the result in data register E[c].

If D[a] contains a NaN value, the result in E[c] will also contain a NaN. In order to maintain both the distinction between signalling and quiet NaNs, and the quiet NaN invalid operation code, E[c][51:50] is set to the top two input mantissa bits and E[c][20:0] is set to the remaining input mantissa bits, whenever D[a] contains a NaN value. All other output mantissa bits are cleared.

FTODF E[c], D[a] (RR)




```
if (is_sp_inf(D[a])) then {
    if(sp_sign_bit(D[a])) then {
        result = DP_NEG_INFINITY;
    } else {
        result = DP_POS_INFINITY;
    }
} else if (is_sp_nan(D[a])) then {
    result[63]    = sp_sign_bit(D[a]);
    result[62:52] = 7FF_H;
    result[51:50] = D[a][22:21];
    result[49:21] = 0;
    result[20:0]  = D[a][20:0];
} else {
    precise_result = sp_real(D[a]);
    result = ieee754_dp_format(precise_result);
}
E[c] = result;
```

Exception Flags

FS	if(set_FI) then PSW.FS = 1 else PSW.FS = 0;
FI	if(is_sp_s_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	Not set by this instruction.

Examples

```
ftodf    e0, d2
```

See Also

DFTOF

3.4 Virtualization instructions

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

3.x.xxx HVCALL (1)

Hypervisor Call (2)

Description

Note: *Execution Mode - This instruction can only be executed in the following mode: Supervisor* (3)

The HVCALL instruction allows a virtual machine to call the hypervisor. The instruction causes a hypervisor call trap, using hypervisor Trap Identification Number (HVTIN) specified by const9[7:0].

If Virtualization extension is not present

- An attempt to execute this instruction will result in UOPC trap. (4)

If Virtualization extension is present

- An attempt to execute this instruction when virtualization is not enabled will result in a PRIV trap in the current context.
- An attempt to execute this instruction when already running as hypervisor will result in a PRIV trap in the hypervisor.
- An attempt to execute this instruction when not in Supervisor mode will result in a PRIV trap in the current VM.

HVCALL const9 (RC) (5)

31 – 28	27 – 22	21 – 12	11 – 8	7 – 0
-	02_H	const9	-	AD_H

```
if (VCON0.VEN==0) then trap(PRIV)
if (VCON1.CVMN == 0) then trap(PRIV)
if (PSW.IO != 10B) then trap(PRIV)
hvtrap(HVCALL, const9[7:0])
```
(7)

Status flags

C	Read by the instruction but not changed. (8)
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 (9)

```
hvcall 4
```

See also (10)

SYSCALL, RFH

Key:

1. Instruction Mnemonic
2. Instruction Longname
3. Example of execution mode restrictions for instructions requiring elevated privileges.
4. Description (32-bit) – Including potential execution mode restrictions
5. Syntax (32-bit) and instruction format in parentheses.
6. Opcodes (32-bit)
7. Operation in RTL format (32-bit)
8. Status flags (User Status Bits)
9. Instruction examples (32-bit)
10. Related instructions

3.4.1 CACHEA.I.VM

Cache Address, Invalidate Virtual Machine entry

Description

***Note**: Execution Mode - This instruction can only be executed in the following mode: Supervisor (only when VCON1.CVMN == 0).*

If the virtualization extension is not present

- An attempt to execute this instruction will result in UOPC trap.

If the virtualization extension is present

- An attempt to execute this instruction when the virtualization extension is not enabled will result in a PRIV trap in the current context.
- An attempt to execute this instruction when not running in Hypervisor will result in a PRIV trap in the current VM.
- An attempt to execute this instruction when not in Supervisor mode will result in a PRIV trap in the current VM.

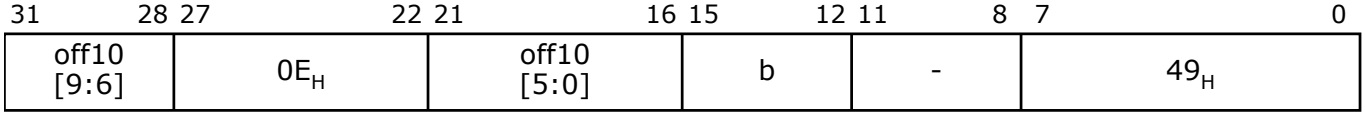
If the cache line containing the byte memory location specified by the effective address is present in the L1 data cache, invalidate the line, regardless of the virtual machine it belongs to. Note that there is no writeback of any modified data in the cache line prior to the invalidation.

The effective address undergoes standard protection checks and requires write permission

If the cache line containing the byte memory location specified by the effective address 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.

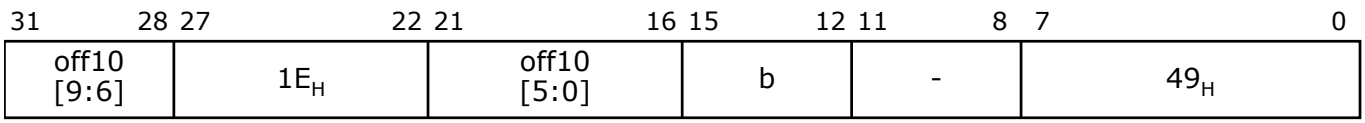
***Note**: This instruction is only valid If the virtualization extension is present, enabled and VCON1.VMN == 0.*

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



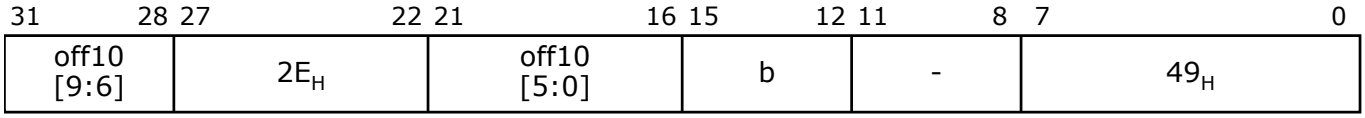

```
If (VCON0.EN == 0) then trap(UOPC);
If (VCON1.CVMN != 0) then trap(PRIV);
EA = A[b];
cache_address_ivld(EA);
A[b] = EA + sign_ext(off10);
```

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




```
If (VCON0.EN == 0) then trap(UOPC);
If (VCON1.CVMN != 0) then trap(PRIV);
EA = A[b] + sign_ext(off10);
cache_address_ivld(EA);
A[b] = EA;
```

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




```
If (VCON0.EN == 0) then trap(UOPC);
If (VCON1.CVMN != 0) then trap(PRIV);
EA = A[b] + sign_ext(off10);
cache_address_ivld(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.vm    [a3]4
cachea.i.vm    [+a3]4
cachea.i.vm    [a3+]4
```

See Also

CACHEA.I, CACHEA.W, CACHEA.WI, CACHEI.I, CACHEI.W, CACHEI.WI, CACHEA.W.VM, CACHEA.WI.VM, CACHEI.I.VM, CACHEI.W.VM, CACHEI.WI.VM

3.4.2 CACHEA.W.VM

Cache Address, Writeback Virtual Machine entry

Description

***Note**: Execution Mode - This instruction can only be executed when VCON1.CVMN == 0.*

If the virtualization extension is not present

- An attempt to execute this instruction will result in UOPC trap.

If the virtualization extension is present

- An attempt to execute this instruction when the virtualization extension is not enabled will result in a PRIV trap in the current context.
- An attempt to execute this instruction when not running in Hypervisor will result in a PRIV trap in the current VM.

If the cache line containing the byte memory location specified by the effective address is present in the L1 data cache, write back any modified data, regardless of the virtual machines it belongs to. 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 effective address 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.

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

31 28	27 22	21 16	15 12	11 8	7 0
off10 [9:6]	$0C_H$	off10 [5:0]	b	-	49_H

```
If (VCON0.EN == 0) then trap(UOPC);
If (VCON1.CVMN != 0) then trap(PRIV);
EA = A[b];
cache_address_wb(EA);
A[b] = EA + sign_ext(off10);
```

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

31 28	27 22	21 16	15 12	11 8	7 0
off10 [9:6]	$1C_H$	off10 [5:0]	b	-	49_H

```
If (VCON0.EN == 0) then trap(UOPC);
If (VCON1.CVMN != 0) then trap(PRIV);
EA = A[b] + sign_ext(off10);
cache_address_wb(EA);
A[b] = EA;
```

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

31 28	27 22	21 16	15 12	11 8	7 0
off10 [9:6]	$2C_H$	off10 [5:0]	b	-	49_H

```
If (VCON0.EN == 0) then trap(UOPC);
```

```
If (VCON1.CVMN != 0) then trap(PRIV);
EA = A[b] + sign_ext(off10);
cache_address_wb(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.vm    [a3]4
cachea.w.vm    [+a3]4
cachea.w.vm    [a3+]4
```

See Also

CACHEA.I, CACHEA.W, CACHEA.WI, CACHEI.I, CACHEI.W, CACHEI.WI, CACHEA.I.VM, CACHEA.WI.VM, CACHEI.I.VM, CACHEI.W.VM, CACHEI.WI.VM

3.4.3 CACHEA.WI.VM

Cache Address, Writeback and Invalidate Virtual Machine entry

Description

***Note**: Execution Mode - This instruction can only be executed when VCON1.CVMN == 0.*

If the virtualization extension is not present

- An attempt to execute this instruction will result in UOPC trap.

If the virtualization extension is present

- An attempt to execute this instruction when the virtualization extension is not enabled will result in a PRIV trap in the current context.
- An attempt to execute this instruction when not running in Hypervisor will result in a PRIV trap in the current VM.

If the cache line containing the byte memory location specified by the effective address is present in the L1 data cache, write back any modified data and then invalidate the line in the L1 data cache, regardless of the virtual machine it belongs to.

If the cache line containing the byte memory location specified by the effective address 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.

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

31 28	27 22	21 16	15 12	11 8	7 0
off10 [9:6]	$0D_H$	off10 [5:0]	b	-	49_H

```
If (VCON0.EN == 0) then trap(UOPC);
If (VCON1.CVMN != 0) then trap(PRIV);
EA = A[b];
cache_address_wi(EA);
A[b] = EA + sign_ext(off10);
```

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

31 28	27 22	21 16	15 12	11 8	7 0
off10 [9:6]	$1D_H$	off10 [5:0]	b	-	49_H

```
If (VCON0.EN == 0) then trap(UOPC);
If (VCON1.CVMN != 0) then trap(PRIV);
EA = A[b] + sign_ext(off10);
cache_address_wi(EA);
A[b] = EA;
```

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

31 28	27 22	21 16	15 12	11 8	7 0
off10 [9:6]	$2D_H$	off10 [5:0]	b	-	49_H

```
If (VCON0.EN == 0) then trap(UOPC);
```

```
If (VCON1.CVMN != 0) then trap(PRIV);
EA = A[b] + sign_ext(off10);
cache_address_wi(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.vm    [a3]4
cachea.wi.vm    [+a3]4
cachea.wi.vm    [a3+]4
```

See Also

CACHEA.I, CACHEA.W, CACHEA.WI, CACHEI.I, CACHEI.W, CACHEI.WI, CACHEA.I.VM, CACHEA.W.VM, CACHEI.I.VM, CACHEI.W.VM, CACHEI.WI.VM

3.4.4 CACHEI.I.VM

Cache Index, Invalidate Virtual Machine entry

Description

***Note**: Execution Mode - This instruction can only be executed in the following mode: Supervisor (only when VCON1.CVMN == 0).*

If the virtualization extension is not present

- An attempt to execute this instruction will result in UOPC trap.

If the virtualization extension is present

- An attempt to execute this instruction when the virtualization extension is not enabled will result in a PRIV trap in the current context.
- An attempt to execute this instruction when not running in Hypervisor will result in a PRIV trap in the current VM.
- An attempt to execute this instruction when not in Supervisor mode will result in a PRIV trap in the current VM.

If the cache line at the index and way specified by the address register A[b] is valid in the L1 data cache, then invalidate the line, regardless of the virtual machine it belongs to. Note that there is no writeback of any modified data in the cache line prior to invalidation.

The effective address undergoes standard protection checks and requires write permission. Address register updates associated with the addressing mode are performed regardless of the cache operation.

***Note**: The location of the cache index and cache way fields within the effective address is implementation dependent.*

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

31 28	27 22	21 16	15 12	11 8	7 0
off10 [9:6]	$0A_H$	off10 [5:0]	b	-	49_H

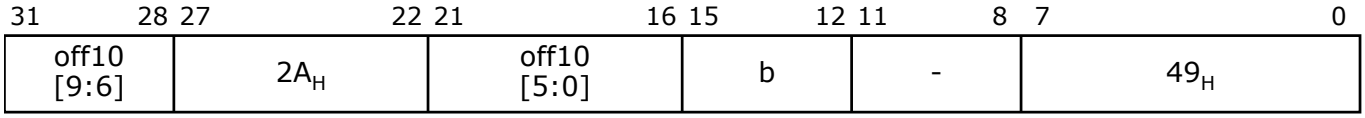
```
If (VCON0.EN == 0) then trap(UOPC);
If (VCON1.CVMN != 0) then trap(PRIV);
EA = A[b];
index_way = EA;
cache_index_ivld(index_way);
A[b] = EA + sign_ext(off10);
```

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

31 28	27 22	21 16	15 12	11 8	7 0
off10 [9:6]	$1A_H$	off10 [5:0]	b	-	49_H

```
If (VCON0.EN == 0) then trap(UOPC);
If (VCON1.CVMN != 0) then trap(PRIV);
EA = A[b] + sign_ext(off10);
index_way = EA;
cache_index_ivld(index_way);
A[b] = EA;
```

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




```
If (VCON0.EN == 0) then trap(UOPC);
If (VCON1.CVMN != 0) then trap(PRIV);
EA = A[b] + sign_ext(off10);
index_way = EA;
cache_index_ivld(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.i.vm    [a3]4
cachei.i.vm    [+a3]4
cachei.i.vm    [a3+]4
```

See Also

CACHEA.I, CACHEA.W, CACHEA.WI, CACHEI.I, CACHEI.W, CACHEI.WI, CACHEA.I.VM, CACHEA.W.VM, CACHEA.WI.VM, CACHEI.W.VM, CACHEI.WI.VM

3.4.5 CACHEI.W.VM

Cache Index, Writeback Virtual Machine entry

Description

***Note**: Execution Mode - This instruction can only be executed when VCON1.CVMN == 0.*

If the virtualization extension is not present

- An attempt to execute this instruction will result in UOPC trap.

If the virtualization extension is present

- An attempt to execute this instruction when the virtualization extension is not enabled will result in a PRIV trap in the current context.
- An attempt to execute this instruction when not running in Hypervisor will result in a PRIV trap in the current VM.

If any modified cache line at the memory index and way specified by the address register A[b] is valid in the L1 data cache, writeback the modified data, regardless of the virtual machine it belongs to. The line will still be present within the L1 data cache but will be marked as unmodified.

The effective address undergoes standard protection checks. Address register updates associated with the addressing mode are performed regardless of the cache operation.

***Note**: The location of the cache index and cache way fields within the effective address is implementation dependent.*

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

31 28	27 22	21 16	15 12	11 8	7 0
off10 [9:6]	$0B_H$	off10 [5:0]	b	-	49_H

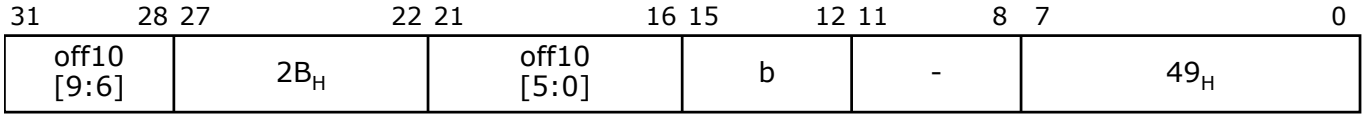
```
If (VCON0.EN == 0) then trap(UOPC);
If (VCON1.CVMN != 0) then trap(PRIV);
EA = A[b];
index_way = EA;
cache_index_wb(index_way);
A[b] = EA + sign_ext(off10);
```

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

31 28	27 22	21 16	15 12	11 8	7 0
off10 [9:6]	$1B_H$	off10 [5:0]	b	-	49_H

```
If (VCON0.EN == 0) then trap(UOPC);
If (VCON1.CVMN != 0) then trap(PRIV);
EA = A[b] + sign_ext(off10);
index_way = EA;
cache_index_wb(index_way);
A[b] = EA;
```

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




```
If (VCON0.EN == 0) then trap(UOPC);
If (VCON1.CVMN != 0) then trap(PRIV);
EA = A[b] + sign_ext(off10);
index_way = EA;
cache_index_wb(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.w.vm    [a3]4
cachei.w.vm    [+a3]4
cachei.w.vm    [a3+]4
```

See Also

CACHEA.I, CACHEA.W, CACHEA.WI, CACHEI.I, CACHEI.W, CACHEI.WI, CACHEA.I.VM, CACHEA.W.VM, CACHEA.WI.VM, CACHEI.I.VM, CACHEI.WI.VM

3.4.6 CACHEI.WI.VM

Cache Index, Writeback, Invalidate Virtual Machine entry

Description

***Note**: Execution Mode - This instruction can only be executed when VCON1.CVMN == 0.*

If the virtualization extension is not present

- An attempt to execute this instruction will result in UOPC trap.

If the virtualization extension is present

- An attempt to execute this instruction when the virtualization extension is not enabled will result in a PRIV trap in the current context.
- An attempt to execute this instruction when not running in Hypervisor will result in a PRIV trap in the current VM.

If the cache line at the memory index and way specified by the address register A[b] is valid in the L1 data cache, write back the modified data and then invalidate the line in the L1 data cache, regardless of the virtual machine it belongs to.

The effective address undergoes standard protection checks. Address register updates associated with the addressing mode are performed regardless of the cache operation.

***Note**: The location of the cache index and cache way fields within the effective address is implementation dependent.*

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

31 28	27 22	21 16	15 12	11 8	7 0
off10 [9:6]	$0F_H$	off10 [5:0]	b	-	49_H

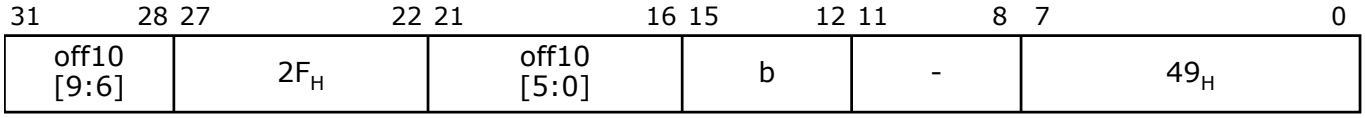
```
If (VCON0.EN == 0) then trap(UOPC);
If (VCON1.CVMN != 0) then trap(PRIV);
EA = A[b];
index_way = EA;
cache_index_wi(index_way);
A[b] = EA + sign_ext(off10);
```

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

31 28	27 22	21 16	15 12	11 8	7 0
off10 [9:6]	$1F_H$	off10 [5:0]	b	-	49_H

```
If (VCON0.EN == 0) then trap(UOPC);
If (VCON1.CVMN != 0) then trap(PRIV);
EA = A[b] + sign_ext(off10);
index_way = EA;
cache_index_wi(index_way);
A[b] = EA;
```

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




```
If (VCON0.EN == 0) then trap(UOPC);
If (VCON1.CVMN != 0) then trap(PRIV);
EA = A[b] + sign_ext(off10);
index_way = EA;
cache_index_wi(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.vm    [a3]4
cachei.wi.vm    [+a3]4
cachei.wi.vm    [a3+]4
```

See Also

CACHEA.I, CACHEA.W, CACHEA.WI, CACHEI.I, CACHEI.W, CACHEI.WI, CACHEA.I.VM, CACHEA.W.VM, CACHEA.WI.VM, CACHEI.I.VM, CACHEI.WI.VM

3.4.7 HVCALL

Hypervisor Call

Description

***Note**: Execution Mode - This instruction can only be executed in the following mode: Supervisor.*

The HVCALL instruction allows a virtual machine to call the hypervisor. The instruction causes a hypervisor call trap, using hypervisor Trap Identification Number (HVTIN) specified by const9[7:0].

If the virtualization extension is not present

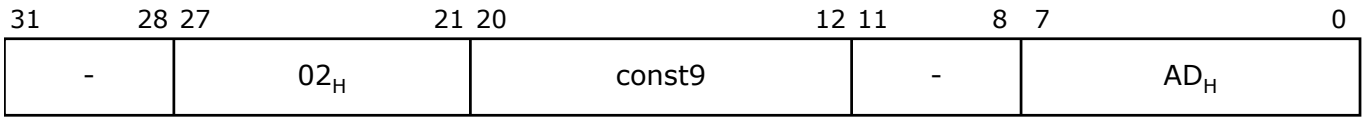
- An attempt to execute this instruction will result in UOPC trap.

If the virtualization extension is present

- An attempt to execute this instruction when the virtualization extension is not enabled will result in a PRIV trap in the current context.
- An attempt to execute this instruction when already running as hypervisor will result in a PRIV trap in the hypervisor.
- An attempt to execute this instruction when not in Supervisor mode will result in a PRIV trap in the current VM.

***Note**: The hypervisor trap return address will be the instruction following the HVCALL instruction.*

HVCALL const9 (RC)




```
if (VCON0.VEN == 0) then trap(PRIV)
if (VCON1.CVMN == 0) then trap(PRIV)
if (PSW.IO != 10B) then trap(PRIV)
hvtrap(HVCALL, const9[7:0])
```

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

```
hvcall    4
```

See Also

SYSCALL, RFH

3.4.8 RFH

Return From Hypervisor

Description

***Note**: Execution Mode - This instruction can only be executed when VCON1.CVMN == 0.*

Return from the Hypervisor to a virtual machine task whose saved upper context is specified by the contents of the Previous Context Information register (PCXI). The VCON2 register is used to define the machine number to return to and the HRHV PRS set to be used for the level 2 MPU. The contents are normally the context of the VM that was interrupted or that took a hypervisor trap. However, in some cases Task Management software may have altered the contents of the PCXI register to cause another virtual machine to be dispatched.

The return address is taken from register A[11] (return address) in the current context. The upper context registers and PSW in the saved context are restored.

If the virtualization extension is not present

- An attempt to execute an RFH will result in an UOPC trap.

If the virtualization extension is present

- An attempt to execute a RFH when the virtualization extension is not enabled will result in a PRIV trap in the current context.
- An attempt to execute a RFH when not running in Hypervisor will result in a PRIV trap in the current context.
- An attempt to execute a RFH in hypervisor with VCON2.VMN == 0 will result in a OPD trap in the hypervisor.

RFH (SYS)

31 28	27 22	21 12	11 8	7 0
-	17_H	-	-	$0D_H$

```
if (VCON1.CVMN != 0) then trap(PRIV);
if (VCON2.VMN == 0) then trap(OPD)
if (PCXI[19:0] == 0) then trap(CSU);
if (PCXI.UL == 0) then trap(CTYP);
if (!cdc_zero() AND PSW.CDE) then trap(NEST);
new_PC = {A[11] [31:1], 1'b0};
HRHV.ICR.IE = PCXI.PIE;
HRHV.ICR.CCPN = PCXI.PCPN;
EA = {PCXI.PCXS, 6'b0, PCXI.PCXO, 6'b0};
{new_PCXI, csa_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) = HRHV.FCX;
HRHV.FCX[19:0] = PCXI[19:0];
PCXI = new_PCXI;
PSW  = {csa_PSW[31:16], PPRS[2], csa_PSW[14], PPRS[1:0], csa_PSW[11:0]};
PPRS = {csa_PSW.PRS[2], csa_PSW.PRS[1:0]};
VCON1.CVMN = VCON2.VMN
// HRHV MPU now acting as Level 2 MPU using VCON2.L2_PRS
if (VCON1.CVMN == 1)
    Current hardware resource = HRA
else
```

```
    Current hardware resource = HRB
PC = new_PC
```

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

```
rfh
```

See Also

HVCALL, SYSCALL

3.5 List of Load Store (LS) instructions

ADD.A - Add Address
ADDIH.A - Add Immediate High to Address
ADDSC.A - Add Scaled Index to Address
ADDSC.AT - Add Bit-Scaled Index to Address
BISR - Begin Interrupt Service Routine
CACHEA.I - Cache Address, Invalidate
CACHEA.W - Cache Address, Writeback
CACHEA.WI - Cache Address, Writeback and Invalidate
CACHEI.I - Cache Index, Invalidate
CACHEI.W - Cache Index, Writeback
CACHEI.WI - Cache Index, Writeback, Invalidate
CALL - Call
CALLA - Call Absolute
CALLI - Call Indirect
CMPSWAP.W - Compare and Swap
DEBUG - Debug
DISABLE - Disable Interrupts
DSYNC - Synchronize Data
ENABLE - Enable Interrupts
EQ.A - Equal to Address
EQZ.A - Equal Zero Address
FCALL - Fast Call
FCALLA - Fast Call Absolute
FCALLI - Fast Call Indirect
FRET - Return from Fast Call
GE.A - Greater Than or Equal Address
ISYNC - Synchronize Instructions
J - Jump Unconditional
JA - Jump Unconditional Absolute
JEQ.A - Jump if Equal Address
JI - Jump Indirect
JL - Jump and Link
JLA - Jump and Link Absolute
JLI - Jump and Link Indirect
JNE.A - Jump if Not Equal Address
JNZ.A - Jump if Not Equal to Zero Address
JRI - Jump Relative Indirect
JZ.A - Jump if Zero Address
LD.A - Load Word to Address Register
LD.B - Load Byte
LD.BU - Load Byte Unsigned
LD.D - Load Double-word

LD.DA - Load Double-word to Address Register
LD.DD - Load Double Double-word
LD.H - Load Half-word
LD.HU - Load Half-word Unsigned
LD.Q - Load Half-word Signed Fraction
LD.W - Load Word
LDLCX - Load Lower Context
LDMST - Load-Modify-Store
LDUCX - Load Upper Context
LEA - Load Effective Address
LHA - Load High Address
LOOP - Loop
LOOPU - Loop Unconditional
LSYNC - Synchronize Local Data
LT.A - Less Than Address
MFCR - Move From Core Register
MOV.A - Move Value to Address Register
MOV.AA - Move Address from Address Register
MOV.D - Move Address to Data Register
MOVH.A - Move High to Address
MTCR - Move To Core Register
NE.A - Not Equal Address
NEZ.A - Not Equal Zero Address
NOP - No Operation
RESTORE - Restore
RET - Return from Call
RFE - Return From Exception
RFM - Return From Monitor
RSLCX - Restore Lower Context
ST.A - Store Word from Address Register
ST.B - Store Byte
ST.D - Store Double-word
ST.DA - Store Double-word from Address Registers
ST.DD - Store Double Double
ST.H - Store Half-word
ST.Q - Store Half-word Signed Fraction
ST.T - Store Bit
ST.W - Store Word
STLCX - Store Lower Context
STUCX - Store Upper Context
SUB.A - Subtract Address
SVLCX - Save Lower Context
SWAP.W - Swap with Data Register

SWAPMSK.W - Swap under Mask
SYSCALL - System Call
TRAPSV - Trap on Sticky Overflow
TRAPV - Trap on Overflow
WAIT - Wait
CACHEA.I.VM - Cache Address, Invalidate Virtual Machine entry
CACHEA.W.VM - Cache Address, Writeback Virtual Machine entry
CACHEA.WI.VM - Cache Address, Writeback and Invalidate Virtual Machine entry
CACHEI.I.VM - Cache Index, Invalidate Virtual Machine entry
CACHEI.W.VM - Cache Index, Writeback Virtual Machine entry
CACHEI.WI.VM - Cache Index, Writeback, Invalidate Virtual Machine entry
HVCALL - Hypervisor Call
RFH - Return From Hypervisor

3.6 List of Integer Pipeline (IP) and Floating Point (FPU) instructions

IP instructions

ABS - Absolute Value
ABS.B - Absolute Value Packed Byte
ABS.H - Absolute Value Packed Half-word
ABSDIF - Absolute Value of Difference
ABSDIF.B - Absolute Value of Difference Packed Byte
ABSDIF.H - Absolute Value of Difference Packed Half-word
ABSDIFS - Absolute Value of Difference with Saturation
ABSDIFS.H - Absolute Value of Difference Packed Half-word with Saturation
ABSS - Absolute Value with Saturation
ABSS.H - Absolute Value Packed Half-word with Saturation
ADD - Add
ADD.B - Add Packed Byte
ADD.H - Add Packed Half-word
ADDC - Add with Carry
ADDI - Add Immediate
ADDIH - Add Immediate High
ADDS - Add with Saturation
ADDS.H - Add Packed Half-word with Saturation
ADDS.HU - Add Unsigned Packed Half-word with Saturation
ADDS.U - Add Unsigned with Saturation
ADDX - Add Extended
AND - Bitwise AND
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
AND.EQ - Equal, AND Accumulating
AND.GE - Greater Than or Equal, AND Accumulating
AND.GE.U - Greater Than or Equal, AND Accumulating Unsigned
AND.LT - Less Than, AND Accumulating
AND.LT.U - Less Than, AND Accumulating Unsigned
AND.NE - Not Equal, AND Accumulating
AND.T - Bit Logical AND
ANDN - Bitwise AND-Not
ANDN.T - Bit Logical AND-Not
BMERGE - Bit Merge
BSPLIT - Bit Split
CADD - Conditional Add
CADDN - Conditional Add-Not
CLO - Count Leading Ones

CLO.H - Count Leading Ones in Packed Half-words
CLS - Count Leading Signs
CLS.H - Count Leading Signs in Packed Half-words
CLZ - Count Leading Zeros
CLZ.H - Count Leading Zeros in Packed Half-words
CMOV - Conditional Move (16-bit)
CMOVN - Conditional Move-Not (16-bit)
CRC32.B - CRC32 Byte
CRC32B.W - CRC32 Word Big-Endian
CRC32L.W - CRC32 Word Little-Endian
CRCN - User-Defined CRC
CSUB - Conditional Subtract
CSUBN - Conditional Subtract-Not
DEXTR - Extract from Double Register
DVADJ - Divide-Adjust
DIV - Divide
DIV64 - Divide 64-bit Long
DIV.U - Divide Unsigned
DIV64.U - Divide 64-bit Unsigned Long
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
DVSTEP - Divide-Step
DVSTEP.U - Divide-Step Unsigned
EQ - Equal
EQ.B - Equal Packed Byte
EQ.H - Equal Packed Half-word
EQ.W - Equal Packed Word
EQANY.B - Equal Any Byte
EQANY.H - Equal Any Half-word
EXTR - Extract Bit Field
EXTR.U - Extract Bit Field Unsigned
GE - Greater Than or Equal
GE.U - Greater Than or Equal Unsigned
IMASK - Insert Mask
INS.T - Insert Bit
INSN.T - Insert Bit-Not
INSERT - Insert Bit Field
IXMAX - Find Maximum Index
IXMAX.U - Find Maximum Index Unsigned

IXMIN - Find Minimum Index
IXMIN.U - Find Minimum Index Unsigned
JEQ - Jump if Equal
JGE - Jump if Greater Than or Equal
JGE.U - Jump if Greater Than or Equal Unsigned
JGEZ - Jump if Greater Than or Equal to Zero (16-bit)
JGTZ - Jump if Greater Than Zero (16-bit)
JLEZ - Jump if Less Than or Equal to Zero (16-bit)
JLT - Jump if Less Than
JLT.U - Jump if Less Than Unsigned
JLTZ - Jump if Less Than Zero (16-bit)
JNE - Jump if Not Equal
JNED - Jump if Not Equal and Decrement
JNEI - Jump if Not Equal and Increment
JNZ - Jump if Not Equal to Zero (16-bit)
JNZ.T - Jump if Not Equal to Zero Bit
JZ - Jump if Zero (16-bit)
JZ.T - Jump if Zero Bit
LT - Less Than
LT.U - Less Than Unsigned
LT.B - Less Than Packed Byte
LT.BU - Less Than Packed Byte Unsigned
LT.H - Less Than Packed Half-word
LT.HU - Less Than Packed Half-word Unsigned
LT.W - Less Than Packed Word
LT.WU - Less Than Packed Word Unsigned
MADD - Multiply-Add
MADDS - Multiply-Add, Saturated
MADD.H - Packed Multiply-Add Q Format
MADDS.H - Packed Multiply-Add Q Format, Saturated
MADD.Q - Multiply-Add Q Format
MADDS.Q - Multiply-Add Q Format, Saturated
MADD.U - Multiply-Add Unsigned
MADDS.U - Multiply-Add Unsigned, Saturated
MADDM.H - Packed Multiply-Add Q Format Multi-precision
MADDMS.H - Packed Multiply-Add Q Format Multi-precision, Saturated
MADDR.H - Packed Multiply-Add Q Format with Rounding
MADDRS.H - Packed Multiply-Add Q Format with Rounding, Saturated
MADDR.Q - Multiply-Add Q Format with Rounding
MADDRS.Q - Multiply-Add Q Format with Rounding, Saturated
MADDSU.H - Packed Multiply-Add/Subtract Q Format
MADDSUS.H - Packed Multiply-Add/Subtract Q Format Saturated
MADDSUM.H - Packed Multiply-Add/Subtract Q Format Multi-precision

MADDSUMS.H - Packed Multiply-Add/Subtract Q Format Multi-precision Saturated
MADDSUR.H - Packed Multiply-Add/Subtract Q Format with Rounding
MADDSURS.H - Packed Multiply-Add/Subtract Q Format with Rounding Saturated
MAX - Maximum Value
MAX.U - Maximum Value Unsigned
MAX.B - Maximum Value Packed Byte
MAX.BU - Maximum Value Packed Byte Unsigned
MAX.H - Maximum Value Packed Half-word
MAX.HU - Maximum Value Packed Half-word Unsigned
MFDCR - Move From Core Register Pair
MIN - Minimum Value
MIN.U - Minimum Value Unsigned
MIN.B - Minimum Value Packed Byte
MIN.BU - Minimum Value Packed Byte Unsigned
MIN.H - Minimum Value Packed Half-word
MIN.HU - Minimum Value Packed Half-word Unsigned
MOV - Move
MOV.U - Move Unsigned
MOVH - Move High
MSUB - Multiply-Subtract
MSUBS - Multiply-Subtract, Saturated
MSUB.H - Packed Multiply-Subtract Q Format
MSUBS.H - Packed Multiply-Subtract Q Format, Saturated
MSUB.Q - Multiply-Subtract Q Format
MSUBS.Q - Multiply-Subtract Q Format, Saturated
MSUB.U - Multiply-Subtract Unsigned
MSUBS.U - Multiply-Subtract Unsigned, Saturated
MSUBAD.H - Packed Multiply-Subtract/Add Q Format
MSUBADS.H - Packed Multiply-Subtract/Add Q Format, Saturated
MSUBADM.H - Packed Multiply-Subtract/Add Q Format-Multi-precision
MSUBADMS.H - Packed Multiply-Subtract/Add Q Format-Multi-precision, Saturated
MSUBADR.H - Packed Multiply-Subtract/Add Q Format with Rounding
MSUBADRS.H - Packed Multiply-Subtract/Add Q Format with Rounding, Saturated
MSUBM.H - Packed Multiply-Subtract Q Format-Multi-precision
MSUBMS.H - Packed Multiply-Subtract Q Format-Multi-precision, Saturated
MSUBR.H - Packed Multiply-Subtract Q Format with Rounding
MSUBRS.H - Packed Multiply-Subtract Q Format with Rounding, Saturated
MSUBR.Q - Multiply-Subtract Q Format with Rounding
MSUBRS.Q - Multiply-Subtract Q Format with Rounding, Saturated
MTDCR - Move to Core Register Pair
MUL - Multiply
MULS - Multiply, Saturated
MUL.H - Packed Multiply Q Format

MUL.Q - Multiply Q Format
MUL.U - Multiply Unsigned
MULS.U - Multiply Unsigned, Saturated
MULM.H - Packed Multiply Q Format-Multi-precision
MULP.B - Packed carry-less multiplication
MULR.H - Packed Multiply Q Format with Rounding
MULR.Q - Multiply Q Format with Rounding
NAND - Bitwise NAND
NAND.T - Bit Logical NAND
NE - Not Equal
NOR - Bitwise NOR
NOR.T - Bit Logical NOR
NOT - Bitwise Complement NOT (16-bit)
OR - Bitwise OR
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
OR.EQ - Equal, OR Accumulating
OR.GE - Greater Than or Equal, OR Accumulating
OR.GE.U - Greater Than or Equal, OR Accumulating Unsigned
OR.LT - Less Than, OR Accumulating
OR.LT.U - Less Than, OR Accumulating Unsigned
OR.NE - Not Equal, OR Accumulating
OR.T - Bit Logical OR
ORN - Bitwise OR-Not
ORN.T - Bit Logical OR-Not
PACK - Pack
PARITY - Parity
POPCNT.W - Population Count Word
REM64 - Remainder 64-bit Long
REM64.U - Remainder 64-bit Unsigned Long
RSTV - Reset Overflow Bits
RSUB - Reverse-Subtract
RSUBS - Reverse-Subtract with Saturation
RSUBS.U - Reverse-Subtract Unsigned with Saturation
SAT.B - Saturate Byte
SAT.BU - Saturate Byte Unsigned
SAT.H - Saturate Half-word
SAT.HU - Saturate Half-word Unsigned
SEL - Select
SELN - Select-Not
SH - Shift

SH.EQ - Shift Equal
SH.GE - Shift Greater Than or Equal
SH.GE.U - Shift Greater Than or Equal Unsigned
SH.H - Shift Packed Half-words
SH.LT - Shift Less Than
SH.LT.U - Shift Less Than Unsigned
SH.NE - Shift Not Equal
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
SHA - Arithmetic Shift
SHA.H - Arithmetic Shift Packed Half-words
SHAS - Arithmetic Shift with Saturation
SHUFFLE - Byte Shuffle
SUB - Subtract
SUB.B - Subtract Packed Byte
SUB.H - Subtract Packed Half-word
SUBC - Subtract With Carry
SUBS - Subtract with Saturation
SUBS.U - Subtract Unsigned with Saturation
SUBS.H - Subtract Packed Half-word with Saturation
SUBS.HU - Subtract Packed Half-word Unsigned with Saturation
SUBX - Subtract Extended
TRAPINV - Trap Invalid Opcode
UNPACK - Unpack Floating Point
XNOR - Bitwise XNOR
XNOR.T - Bit Logical XNOR
XOR - Bitwise XOR
XOR.EQ - Equal, XOR Accumulating
XOR.GE - Greater Than or Equal, XOR Accumulating
XOR.GE.U - Greater Than or Equal, XOR Accumulating Unsigned
XOR.LT - Less Than, XOR Accumulating
XOR.LT.U - Less Than, XOR Accumulating Unsigned
XOR.NE - Not Equal, XOR Accumulating
XOR.T - Bit Logical XOR

FPU and DP FPU instructions

ABS.F - Absolute Value Float

ADD.F - Add Float
CMP.F - Compare Float
DIV.F - Divide Float
FTOI - Float to Integer
FTOIN - Float to Integer, Round to Nearest
FTOIZ - Float to Integer, Round towards Zero
FTOQ31 - Float to Fraction
FTOQ31Z - Float to Fraction, Round towards Zero
FTOU - Float to Unsigned
FTOUZ - Float to Unsigned, Round towards Zero
FTOHP - Single Precision to Half Precision
HPTOF - Half Precision to Single Precision
ITOF - Integer to Float
MADD.F - Multiply Add Float
MAX.F - Maximum Value Float
MIN.F - Minumum Value Float
MSUB.F - Multiply Subtract Float
MUL.F - Multiply Float
NEG.F - Negate Value Float
Q31TOF - Fraction to Floating-point
QSEED.F - Inverse Square Root Seed
SUB.F - Subtract Float
UPDFL - Update Flags
UTOF - Unsigned to Floating-point
ABS.DF - Absolute Value Double
ADD.DF - Add Double
CMP.DF - Compare Double
DIV.DF - Divide Double
DFTOI - Double to Integer
DFTOIN - Double to Integer, Round to Nearest
DFTOIZ - Double to Integer, Round towards Zero
DFTOL - Double to Long Integer
DFTOLZ - Double to Long Integer, Round towards Zero
DFTOU - Double to Unsigned
DFTOUZ - Double to Unsigned, Round towards Zero
DFTOUL - Double to Unsigned Long
DFTOULZ - Double to Unsigned Long Integer, Round towards Zero
ITODF - Integer to Double
LTODF - Long Integer to Double
MADD.DF - Multiply Add Double
MAX.DF - Maximum Value Double
MIN.DF - Minumum Value Double
MSUB.DF - Multiply Subtract Double

MUL.DF - Multiply Double
NEG.DF - Negate Value Double
QSEED.DF - Inverse Square Root Seed
SUB.DF - Subtract Double
UTODF - Unsigned to Double
ULTODF - Unsigned Long Integer to Double
DFTOF - Double Precision to Single Precision
FTODF - Single Precision to Double Precision

3.7 List of instructions only available or modified when Virtualization is present and enabled

Instructions only available when Virtualization is enabled

CACHEA.I.VM - Cache Address, Invalidate Virtual Machine entry
CACHEA.W.VM - Cache Address, Writeback Virtual Machine entry
CACHEA.WI.VM - Cache Address, Writeback and Invalidate Virtual Machine entry
CACHEI.I.VM - Cache Index, Invalidate Virtual Machine entry
CACHEI.W.VM - Cache Index, Writeback Virtual Machine entry
CACHEI.WI.VM - Cache Index, Writeback, Invalidate Virtual Machine entry
HVCALL - Hypervisor Call
RFH - Return From Hypervisor

Modified instructions

CACHEA.I - Cache Address, Invalidate
CACHEA.W - Cache Address, Writeback
CACHEA.WI - Cache Address, Writeback and Invalidate
CACHEI.I - Cache Index, Invalidate
CACHEI.W - Cache Index, Writeback
CACHEI.WI - Cache Index, Writeback, Invalidate
DISABLE - Disable Interrupts
ENABLE - Enable Interrupts
MFCR - Move From Core Register
MFDCR - Move From Core Register Pair
MTCR - Move To Core Register
MTDCR - Move to Core Register Pair
RESTORE - Restore

Revision history

Reference	Description of change(s)	Comment
V0.7		
	First public release	
V0.71D		
	Added DFLUSH instruction Updated DSYNC description	
V0.72D		
	Added DIV64, DIV64.U, REM64 and REM64.U instructions to support 64-bit integer divide and remainder functions.	
V0.73D		
	Added TRAPINV instruction.	
V0.74D		
	Renamed DFLUSH into LSYNC.	
V0.9, 2022-11-18		
RTL functions	Minor update to crc_div description	
Trap instructions	Added mention of TRAPINV instruction	
CACHEI.I, CACHEI.W, CACHEI.WI	Now using the correct cache function (cache_index_xxx instead of cache_address_xxx)	
CMPSWAP.W, SWAPMSK.W	Updated argument ordering in example to match the instruction description.	
CRCN	Fixed operation description. The size of the output crc is not dependent on the size of the data	
DIV/DIV64/REM64	Fixed wording of DIV/DIV64/REM64 regarding overflow. The intention of the description was to refer to the minimum negative number (-2^{31}) rather than the maximum negative number (-1)	
DIV64/DIV64.U/ REM64/REM64.U	Updated instruction longname to make it clear that these instructions operate on 64-bit numbers	
DSYNC, LSYNC	Updated descriptions	
MFCR, MFDCR, MTCR, MTDCR	Simplified description in line with updates in the virtualization section in the architecture manual volume 1	
MOV.AA	Replaced duplicated see also link to MOVH.A with missing link to MOV.A	
RESTORE	Removed duplicated statement	
V0.9.1, 2023-02-14		
About this document	Section now consistent with volume 1	
Usage	Improved description to include MFDCR and MTDCR cases	
Conditional branch	Added mention of 5-bit displacements and availability of 16-bit JNE/JEQ to compare against data registers	
Load and store basic data types	Added mention of quad-word type	

Revision history

Reference	Description of change(s)	Comment
Access to the core special function registers (CSFRs)	Fixed "Write CSFR" case to correctly mention MTCR Added mention of MTCR or MTDCR sequences Updated diagram	
32-bit opcode formats	Opcode tables tidy-up	
V0.9.2, 2023-12-05		
All instructions	Updated examples. Re-ordered to match instruction order. Added missing instruction examples, including use of double address register using Px notation	
Synchronization primitives (DSYNC, LSYNC and ISYNC)	Added mention of LSYNC	
CALLI	Pseudo code for the 16-bit variant updated to show correct operation "ret_addr = PC + 2" and not "PC+4	
V1.0.0, 2024-02-14		
Overall document	Minor formatting and type consistency updates.	
Load bit	Changed example capitalization	
Store bit and bit-field	Changed example capitalization	
Lower context saving and restoring	Explicitly mentioned CSA	
Enabling and disabling the interrupt system	Clarified impact of disabling interrupts in both guest VM and hypervisor. Removed mention that VM0 can globally disable interrupts as this is not the case	
Trap instructions	Explicit mention of PSW.V and PSW.SV	
Coprocessor instructions	Added note stating that the coprocessor interface is implementation specific	
List of OS and I/O privileged instructions	Removed mention of kernel	

Reference	Description of change(s)	Comment
ABSDIF, ABSDIFS, CACHEA.I, CACHEA.W, CACHE.WI, CADD, CADDN, CMOV, CMON, CMPSWAP.W, CSUB, CSUBN, LD.A, LD.B, LD.BU, LD.D, LD.DA, LD.DD, LD.DH, LD.HU, LD.Q, LD.W, LDMST, LD.W, SEL, SELN, SH, SH.H, ST.A, ST.B, ST.D, ST.DA, ST.DD, ST.H, ST.Q, ST.W, SWAP.W, SWAPMSK.W	Added brackets in instruction operation to highlight precedence	
ADDSC.A	Removed spurious brackets in pseudo code	
AND.ANDN.T	Fixed misplaced coma in short_name	
AND.AND.T, AND.ANDN.T, AND.NOR.T, AND.OR.T, AND.T, ANDN.T, CADD, NAND.T, NOR.T, OR.AND.T, OR.ANDN.T, OR.NOT.T, OR.T, ORN.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, XNOT.T, XOR.T	Added missing # characters in several instruction examples	
CACHEA.I, LD.HU, LD.Q, ST.B	Removed typo in operation section (wrong EA0 instead of EA)	
DVSTEP	Removed double ::	
OR.NOR.T	Example was incorrectly referring to OR.ANDN.T	
TRAPINV	Small wording change	

Trademarks

All referenced product or service names and trademarks are the property of their respective owners.

Edition 2024-02-14

Published by

Infineon Technologies AG

81726 Munich, Germany

Do you have a question about any aspect of this document?

Email: erratum@infineon.com

Document reference
IFX-lhe1591790853267

Important notice

The information given in this document shall in no event be regarded as a guarantee of conditions or characteristics ("Beschaffenheitsgarantie").

With respect to any examples, hints or any typical values stated herein and/or any information regarding the application of the product, 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.

In addition, any information given in this document is subject to customer's compliance with its obligations stated in this document and any applicable legal requirements, norms and standards concerning customer's products and any use of the product of Infineon Technologies in customer's applications.

The data contained in this document is exclusively intended for technically trained staff. It is the responsibility of customer's technical departments to evaluate the suitability of the product for the intended application and the completeness of the product information given in this document with respect to such application.

Warnings

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

Except as otherwise explicitly approved by Infineon Technologies in a written document signed by authorized representatives of Infineon Technologies, Infineon Technologies' products may not be used in any applications where a failure of the product or any consequences of the use thereof can reasonably be expected to result in personal injury.