TriCore™

32-bit

TriCore™ V1.6

Instruction Set
32-bit Unified Processor Core

User Manual (Volume 2)

V1.0, 2012-05

Microcontrollers

Edition 2012-05

Published by
Infineon Technologies AG
81726 Munich, Germany

TriCore™

32-bit

TriCore™ V1.6

Instruction Set
32-bit Unified Processor Core

User Manual (Volume 2)

V1.0, 2012-05

Microcontrollers

TriCore™ User Manual (Volume 2)

Revision History: V1.0 2012-05

Previous Versions:	
Page	Description
All	TC1.6 First release

Trademarks

Table of Contents

Preface

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

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

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

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

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

TriCore Versions

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

- TriCore 1.6

Please note that:

- Unless stated otherwise in the text, all descriptions are common to all TriCore versions listed in this preface.

Additional Information

For information and links to documentation for Infineon products that use TriCore, visit:

http://www.infineon.com/32-bit-microcontrollers

Text Conventions

This document uses the following text conventions:

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

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

Table 0-1 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. Read value is undefined, but must be written with 0.

1 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.*

1.1 Instruction Syntax

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

1.1.1 Operand Definitions

The Operand definitions are detailed in the following table.

Table 1-1 Operand Definitions

Operand	Definition
D[n]	Data register n
A[n]	Address register n
E[n]	Extended data register n containing a 64-bit value made from an even/odd pair of registers (D[n], D[n+1]). The format is little endian. E[n][63:32] = D[n+1][31:0]; E[n][31:0] = D[n][31:0]
dispn	Displacement value of 'n' bits, used to form the effective address in branch instructions
constn	Constant value of 'n' bits, used as instruction operand
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
width	Specifies the width of the bit field in bit field instructions

1.1.2 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 1-2 Implicit Operand Definitions

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

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

1.1.3 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 1-3 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

1.1.4 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 1-4 Data Type Modifiers

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

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

1.2 Opcode Formats

1.2.1 16-bit Opcode Formats

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

Table 1-5 16-bit Opcode Formats

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

1.2.2 32-bit Opcode Formats

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

Table 1-6 32-bit Opcode Formats

	31:30	29:28	27:26	25:24	23:22	21:20	21:20	19:18	17:16	15:14	13:12	11:10	9-8	7-6	5-0
ABS	off18 [9:6]		op2	off18 [13:10]		off18[5:0]				off18 [17:14]		s1/d		op1	
ABSB	off18 [9:6]		op2	off18 [13:10]		off18[5:0]				off18 [17:14]		b	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	o p 2	disp15								const4		s1		op1	
BRN	o p 2	disp15								n[3:0]		s1		n[4]	op1
BRR	o p 2	disp15								s2		s1		op1	
RC	d		op2				const9					s1		op1	
RCPW	d		pos		op2		width			const4		s1		op1	
RCR	d		s3		op2		const9					s1		op1	
RCRR	d		s3		op2		-			const4		s1		op1	
RCRW	d		s3		op2		width			const4		s1		op1	
RLC	d		const16									s1		op1	
RR	d		op2					-	n	s2		s1		op1	
RR1	d		op2						n	s2		s1		op1	
RR2	d		op2							s2		s1		op1	
RRPW	d		pos		op2		width			s2		s1		op1	
RRR	d		s3		op2			-	n	s2		s1		op1	
RRR1	d		s3		op2				n	s2		s1		op1	
RRR2	d		s3		op2					s2		s1		op1	
RRRR	d		s3		op2		-			s2		s1		op1	
RRRW	d		s3		op2		width			s2		s1		op1	
SYS	-		op2			-						s1/d		op1	

1.2.3 Opcode Field Definitions

Table 1-7 Opcode Field Definitions

Name	Width	Definition
s1	4	Source register(s) one
s2	4	Source register(s) number two
s3	4	Source register(s) number three
d	4	Destination register For a register pair (E), the coding follows the register number: E[0] = 0000_B, E[2] = 0010_B, E[4] = 0100_B, and so on.
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
off16	16	16-bit offset
-	-	Reserved field. Read value is undefined; should be written with zero (0). Must be set to zero (0) to allow for future compatibility.
op1		Primary Opcode
op2		Secondary Opcode

1.3 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 1-8 RTL Syntax

Syntax	Definition
bpos3	Bit position
const’n’	Constant value of ‘n’bits, used as instruction operand
disp’n’	Displacement value of ‘n’ bits, used to form the effective address in branch instructions
(expression)[p]	A single bit, wiht ordinal index ‘p’ in the bit field ‘expression’
n’bx	Constant bit string, where ‘n’ is the number of bits in the constant and ‘x’ is the constant in binary. For example; 2’b11
n’hx	Constant bit string, where ‘n’ is the number of bits in the constant and ‘x’ is the constant in hexadecimal. For example, 16’hFFFF
off’n’	Offset value of ‘n’ bits, used to form the effective address in Load and Store instructions
pos	Single bit position
signed	A value that can be positive, negative or zero
ssov	Saturation on signed overflow
suov	Saturation on unsigned overflow
unsigned	A value that can be positive or zero
{x,y}	A bit string where ‘x’ and ‘y’ are expressions representing a bit or bit field. Any number of expressions can be concatenated. For example, {x,y,z}
A[n]	Address register ‘n’
CR	Core Register
D[n]	Data register ‘n’
EA	Effective Address
E[n]	Data register containing a 64-bit value, constructed by pairing two data registers. The least-significant bit is in the even register D[n], and the most significant bit is in the odd register D[n+1]
M(EA, data_size)	Memory locations beginning at the specified byte location EA, and extending to EA + data_size - 1. data_size = byte, half-word, word, , 16-word
<mode>	And addressing mode
P[n]	Address register containing a 64-bit value, constructed by pairing two address registers. The least-significant bit is in the even register A[n], and the most significant bit is in the odd register A[n+1]
PC	The address of the instruction in memory
[x:y]	Bits y, y+1, ..., x where x>y; For example D[a][x:y], if x=y then this is a single bit range which is also denoted by [x], as in D[a][x]. For cases where x<y, this denotes an empty range.

Table 1-8 RTL Syntax

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

1.3.1 RTL Functions

Register Transfer Level functions are defined in the table which follows.

Table 1-9 RTL Functions

Function	Definition
abs(x)	abs(x) returns ((x <0) ? (0 - x) : x);
cache_address_ivld(EA)	Defined in ‘Cache RTL Functions’, which follows
cache_address_wb(EA)	Defined in ‘Cache RTL Functions’, which follows
cache_address_wi(EA)	Defined in ‘Cache RTL Functions’, which follows
cache_index_ivld	Defined in ‘Cache RTL Functions’, which follows
cache_index_wb	Defined in ‘Cache RTL Functions’, which follows
cache_index_wi	Defined in ‘Cache RTL Functions’, which follows
cache_index_wb(EA)	Defined in the ‘Cache RTL Functions’ section, which follows. (TriCore1.6)
cache_index_wi(EA)	Defined in the ‘Cache RTL Functions’ section, which follows. (TriCore1.6)

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
leading_ones(x)	Returns the number of leading ones of ‘x’
leading_signs(x)	Returns the number of leading sign bits of ‘x’
leading_zeros(x)	Returns the number of leading zeros of ‘x’
reverse16(n)	{n[0], n[1], n[2], n[3], n[4], n[5], n[6], n[7], n[8], n[9], n[10], n[11], n[12], n[13], n[14], n[15]}
round16(n)	= {(x + 32'h00008000)[31:16],16'h0000};

ssov(x,y)	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)	max_pos = (1 << y) - 1;
	return ((x > max_pos) ? max_pos : ((x < 0) ? 0 : x));

sign_ext(x)	Sign extension; high-order bit of x is left extended
trap(x)	Instruction will take trap ‘x’
zero_ext(x)	Zero extensions; high-order bits are set to 0

1.3.2 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.

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 1-10 Cache Functions

Function	Definition
cache_address_ivld(EA)	if (CACHE[EA].valid==1) then CACHE [EA].valid=0;

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

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_index_ivld	if (CACHE[index,elem].valid = = 1) then CACHE[index,elem].valid = 0;

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

Table 1-10 Cache Functions

Function	Definition
cache_index_wi	if (CACHE[index,elem].valid==1) then { if (CACHE[index,elem].modified==1) then { pa = CACHE[index,elem].physical_address; M[pa,cacheline] = CACHE[index,elem].data; } CACHE[index,elem].modified = 0; CACHE[index,elem].valid = 0; }
cache_Index_wb(location)	if ((CACHE[index,elem].valid==1) AND (CACHE[index,elem].modified==1)) then { pa = CACHE[index,elem].physical_address; M[pa,cacheline] = CACHE[index,elem].data; CACHE[index,elem].modified = 0; }
cache_index_wi(location)	if (CACHE[index,elem].valid==1) then { if (CACHE[index,elem].modified==1) then { pa = CACHE[index,elem].physical_address; M[pa,cacheline] = CACHE[index,elem].data; } CACHE[index,elem].modified = 0; CACHE[index,elem].valid = 0; }

1.3.3 Floating Point Operation Syntax

The following table defines the floating point operation syntax.

Table 1-11 Floating Point Operation Syntax

Syntax	Definition
ADD_NAN	$7FC00001_H$
MUL_NAN	$7FC00002_H$
SQRT_NAN	$7FC00004_H$
DIV_NAN	$7FC00008_H$
POS_INFINITY	$7F800000_H$
NEG_INFINITY	$FF800000_H$
is_s_nan(x)	Takes the IEEE754 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_q_nan(x)	Takes the IEEE754 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_nan(x)	Takes the IEEE754 32-bit single precision floating point format value ‘x’ and returns the boolean result of the expression: (is_s_nan(x) OR is_q_nan(x));
is_pos_inf(x)	Takes the IEEE754 32-bit single precision floating point format value ‘x’ and returns the boolean result of the expression: (x[31:0] == POS_INFINITY);
is_neg_inf(x)	Takes the IEEE754 32-bit single precision floating point format value ‘x’ and returns the boolean result of the expression: (x[31:0] == NEG_INFINITY);
is_inf(x)	Takes the IEEE754 32-bit single precision floating point format value ‘x’ and returns the boolean result of the expression: (is_neg_inf(x) OR is_pos_inf(x));
is_zero(x)	Takes the IEEE754 32-bit single precision floating point format value ‘x’ and returns the boolean result of the expression: (x[30:0] == 0);
is_denorm(x)	Takes the IEEE754 32-bit single precision floating point format value ‘x’ and returns the boolean result of the expression: (x[30:23] == 0) AND (x[22:0] != 0);

denorm_to_zero(x)	If the IEEE754 32-bit single precision floating point format value ‘x’ is a denormal value return the appropriately signed infinitely accurate real value 0. Otherwise return ‘x’ as an infinitely accurate real value; i.e.
	if((x < 0) AND (x > -2-126)) then return -0.0;
	else if((x > 0) AND (x < 2126)) then return +0.0;
	else return f_real(x);

round_to_integer(x,y)	Returns a signed integer result of infinite width by rounding the IEEE754 32-bit single precision floating point format value ‘x’ to an integer value using the IEEE754 mode specified by ‘y’.
round_to_unsigned(x,y)	Returns an unsigned integer result of infinite width by rounding the IEEE754 32-bit single precision floating point format value ‘x’ to an integer value using the IEEE754 mode specified by ‘y’.
round_to_q31(x,y)	Returns a Q format result of infinite width by rounding the real value ‘x’ to a Q format value using the IEEE754 mode specified by ‘y’.

Table 1-11 Floating Point Operation Syntax

Syntax	Definition
i_real(x)	Returns a infinitely accurate real number of equal value to the 32-bit signed integer value ‘x’.
u_real(x)	Returns a infinitely accurate real number of equal value to the 32-bit unsigned integer value ‘x’.
f_real(x)	Returns the IEEE754 32-bit single precision floating point format value ‘x’ as an infinitely accurate real value.
q_real(x)	Returns the Q31 format value ‘x’ as an infinitely accurate real value.
add(x,y)	Adds the real value ‘x’ to the real value ‘y’ and returns an infinitely accurate real result.
mul(x,y)	Multiply the real value ‘x’ by the real value ‘y’ and return an infinitely accurate real result.
divide(x,y)	Divides the real value ‘x’ by the real value ‘y’ and returns an infinitely accurate real result.
ieee754_round(x,y)	Rounds the real value ‘x’ using the type of rounding specified by ‘y’ compliant with IEEE754.
ieee754_32bit_format(x)	Returns the real value ‘x’ in the standard 32-bit single precision IEEE754 floating point format. ‘x’ is converted to the correct IEEE754 result on overflow or underflow.
ieee754_lt(x,y)	Returns TRUE if ‘x’ is less than ‘y’ according to the IEEE754 rules for 32-bit single precision floating point numbers otherwise returns FALSE.
ieee754_gt(x,y)	Returns TRUE if ‘x’ is greater than ‘y’ according to the IEEE754 rules for 32-bit single precision floating point numbers otherwise returns FALSE.
ieee754_eq(x,y)	Returns TRUE if ‘x’ is equal to ‘y’ according to the IEEE754 rules for 32-bit single precision floating point numbers otherwise returns FALSE.
fp_abs(x)	Returns the infinitely accurate absolute value of the real value ‘x’; i.e. (x < 0.0) ? (0.0 - x) : x;
approx_inv_sqrt(x)	Takes the real argument x and returns the approximate inverse square root ($x^{-0.5}$) to at least 6.75 bits of precision.

1.4 Coprocessor Instructions

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

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

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

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

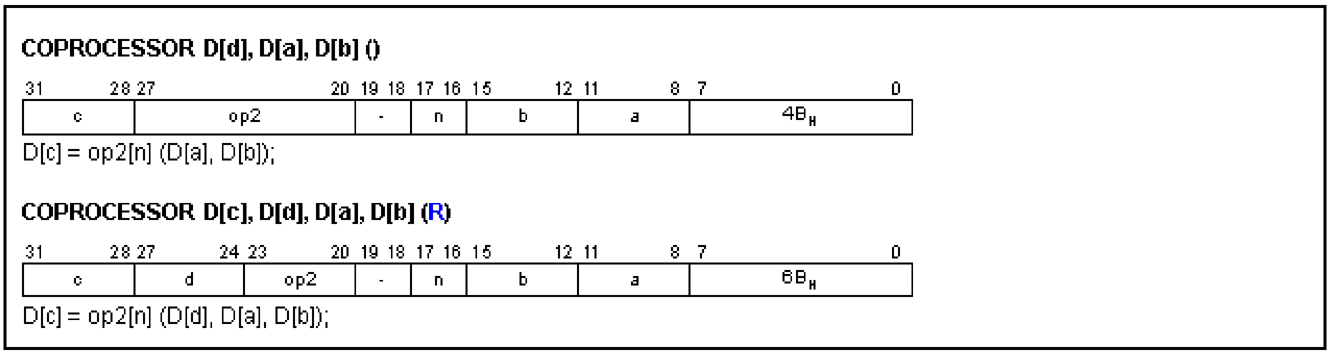

COPROCESSOR D[d], D[a], D[b] ()
31 28 27 20 19 18 17 16 15 12 11 8 7 0
c
op2
-
n
b
a
4B_H
D[c] = op2[n] (D[a], D[b]);
COPROCESSOR D[c], D[d], D[a], D[b] (R)
31 28 27 24 23 20 19 18 17 16 15 12 11 8 7 0
c
d
op2
-
n
b
a
6B_H
D[c] = op2[n] (D[d], D[a], D[b]);

Figure 1-1 Coprocessor Instructions

Table 1-12 Coprocessor Status Flags

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

1.5 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]. 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 1-13 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.

1.6 List of OS and I/O Privileged Instructions

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

Table 1-14 OS and I/O Privileged Instructions

Kernel (Supervisor)	User-1 Mode	User-0 Mode
BISR MTCR CACHEI.I CACHEA.I RFM	ENABLE DISABLE RESTORE	All others (including DEBUG)

2 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 desribed, together with a description of the selection and use of the 16-bit (short) instructions.

2.1 Integer Arithmetic

This section covers the following topics:

2.1.1 Move

The move instructions move a value in a data register or a constant value in the instruction to a destination data register, and can be used to quickly load a large constant into a data register.

A 16-bit constant is created using MOV (which sign-extends the value to 32-bits) or MOV.U (which zero-extends to 32-bits).

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

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

2.1.2 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.

2.1.3 Multiply and Multiply-Add

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

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

These translate to machine instructions producing either 32-bit or 64-bit results, depending on whether the destination operand encoded in the assembly instruction is a single data register D[n] (where n = 0, 1, …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.

2.1.4 Division

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

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.

2.1.5 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.

2.1.6 Min, Max, Saturate

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

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

2.1.7 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.

2.1.8 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).

2.1.9 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 **Figure 2-1**, there are seven zeros in the most significant portion of the input register. If the most significant bit of the input is a 1, CLZ returns 0:

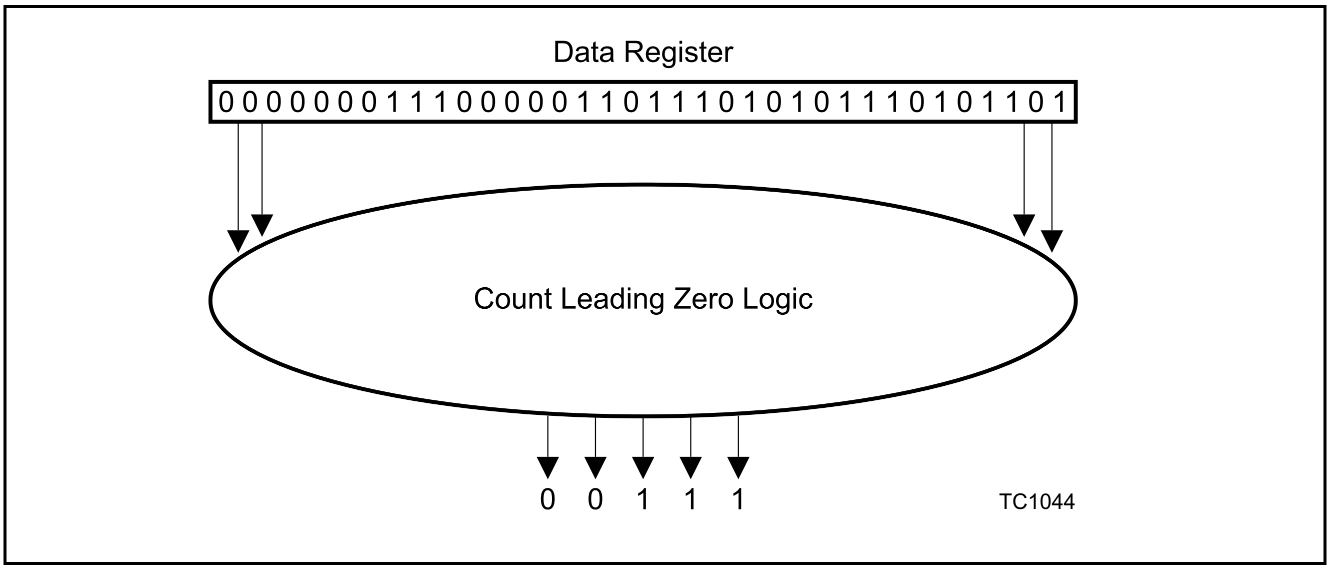


Figure 2-1 Operation of the CLZ Instruction

2.1.10 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.

2.1.11 Bit-Field Extract and Insert

The TriCore architecture supports three, bit-field extract instructions:

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

The INSERT instruction is described on **Page 2-6**.

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). See **Figure 2-2**.

EXTR.U zero-fills the most significant (32-w) bits of the result. See **Figure 2-3**.

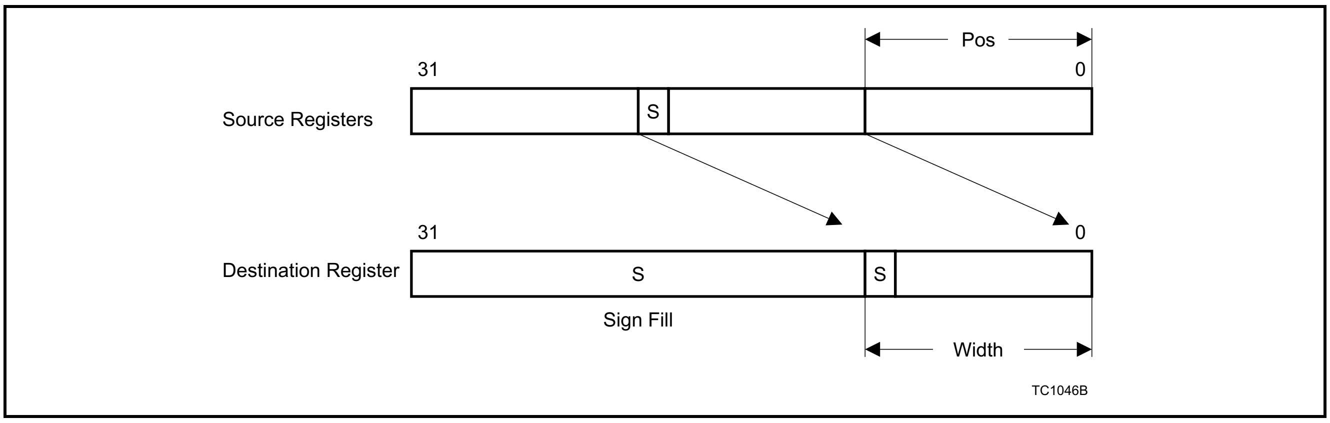


Figure 2-2 EXTR Operation

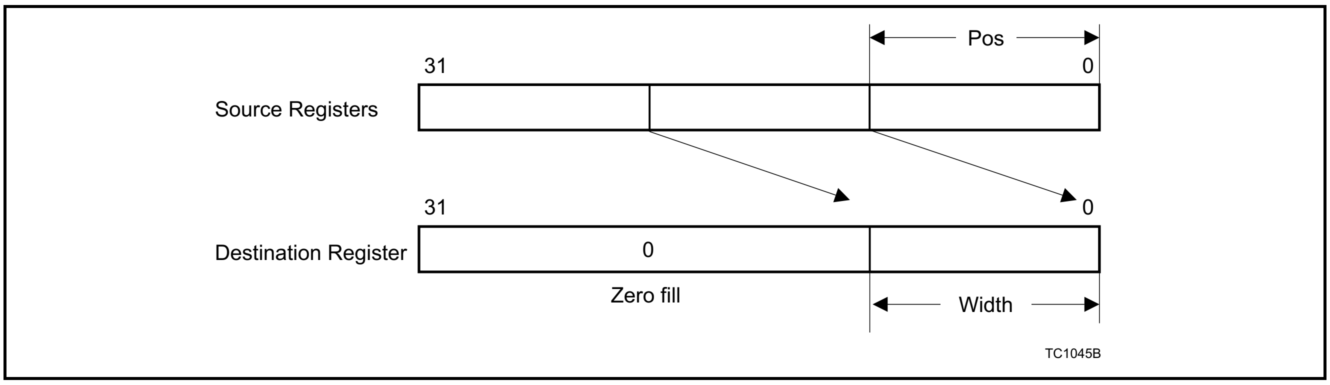


Figure 2-3 EXTR.U Operation

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).

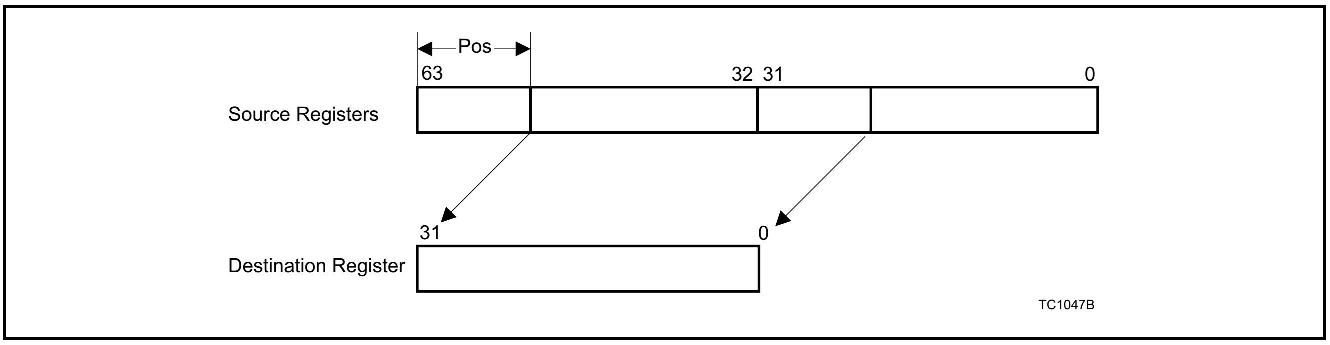


Figure 2-4 DEXTR Operation

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-w) bits of the value of the second register are passed through. The width and pos can be specified by two immediate values, by an immediate value and a data register, or by a data register pair.

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

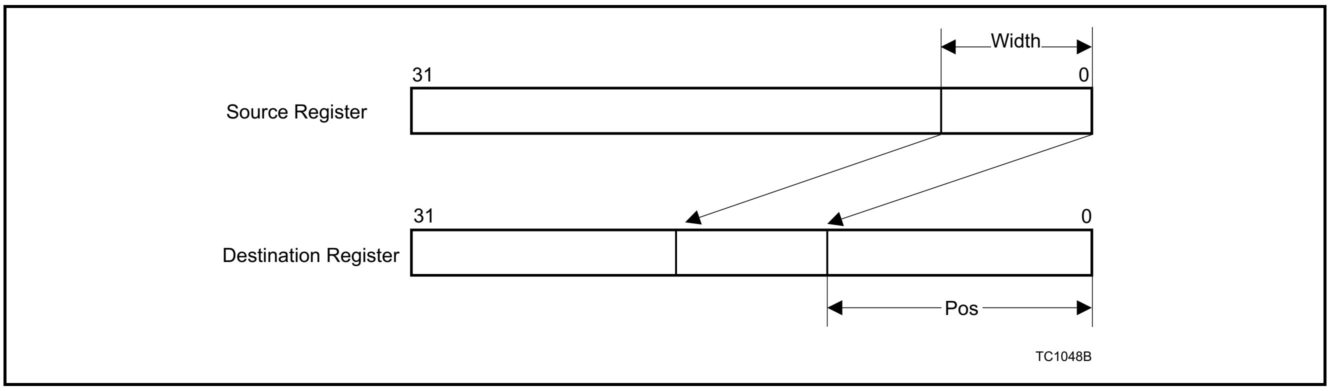


Figure 2-5 INSERT Operation

2.2 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.

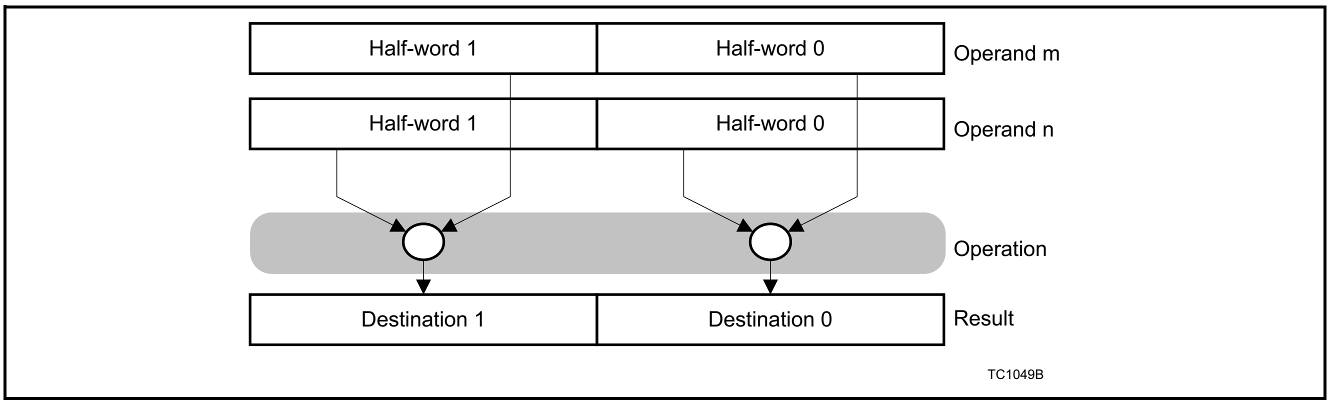


Figure 2-6 Packed Half-word Data Format

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.

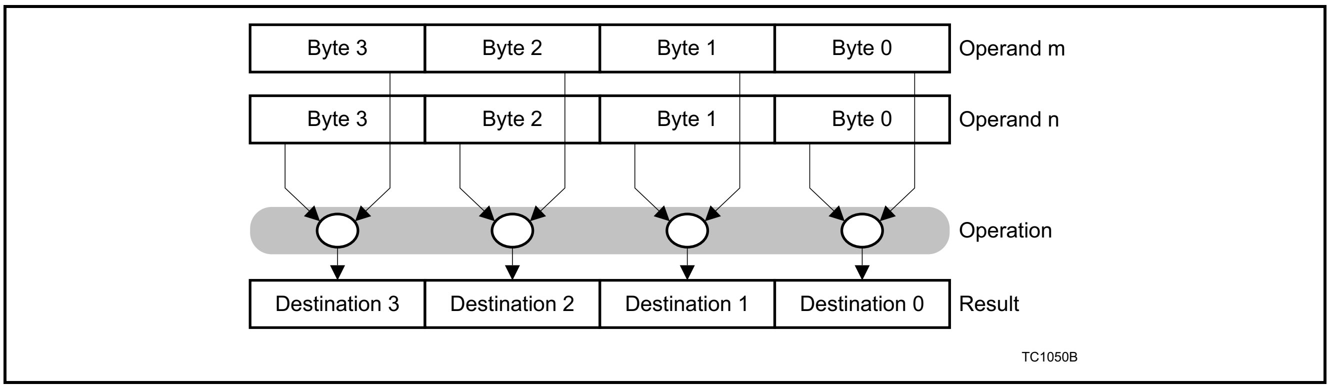


Figure 2-7 Packed Byte Data Format

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 (215-1) on individual overflow, or to the most negative 16-bit signed integer (-215) on individual underflow. Saturation for unsigned integers is also supported by the ADDS.HU instruction. Similarly, all packed addition operations have an equivalent subtraction.

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

Packed multiplication is described in the section **Packed Multiply and Packed MAC, page 2-10**.

Compare instructions are described in **Compare Instructions, page 2-11**.

2.3 PSW (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.

2.3.1 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: MTCR is only available in Supervisor mode.

The Trap on Overflow (TRAPV) and Trap on Sticky Overflow (TRAPSV) instructions can be used to cause a trap if the respective V (overflow) and SV (sticky overflow) bits are set. The overflow bits can be cleared using the Reset Overflow Bits instruction (RSTV).

Individual arithmetic operations can be checked for overflow by reading and testing V.

If it is only necessary to determine if an overflow occurred somewhere in an entire block of computation, then the SV bit is reset before the block (using the RSTV instruction) and tested after completion of the block (using MFCR).

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

2.3.2 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.

2.4 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).

2.4.1 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.

2.4.2 Special Case: -1 * -1

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

```
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.

2.4.3 Guard Bits

When accumulating sums (in filter calculations for example), guard bits are often required to prevent overflow.

The instruction set directly supports the use of one guard bit when using a 32-bit accumulator (2.30 format, where left shift by 1-bit of result is not requested).

When more guard bits are required a register pair (64-bits) can be used. In that instance the intermediate result (also in 2.30 format, where left shift by 1-bit is not performed) is left shifted by 16-bits giving effectively a 18.46 format.

2.4.4 Rounding

Rounding is used to retain the 16 most-significant bits of a 32-bit result.

Rounding is implemented by adding 1 to bit 15 of a 32-bit intermediate result.

If the operation writes a full 32-bit register (i.e. is not a component of a packed half-word operation), it then clears the lower 16-bits.

2.4.5 Overflow and Saturation

Saturation on overflow is available on all DSP instructions.

2.4.6 Sticky Advance Overflow and Block Scaling if FFT

The Sticky Advance Overflow (SAV) bit, which is set whenever an overflow ‘almost’ occurred, can be used in block scaling of intermediate results during an FFT calculation.

Before each pass of applying a butterfly operation, the SAV bit is cleared.

After the pass the SAV bit is tested. If it is set then all of the data is scaled (using an arithmetic right shift) before starting the next pass.

This procedure gives the greatest dynamic range for intermediate results without the risk of overflow.

2.4.7 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 totel of eight different cases:

- 16U * 16U $\rightarrow$ 32
- 16L * 16L $\rightarrow$ 32
- 16U * 32 $\rightarrow$ 32
- 16L * 32 $\rightarrow$ 32
- 32 * 32$\rightarrow$ 32
- 16U * 32 $\rightarrow$ 64
- 16L * 32 $\rightarrow$ 64
- 32 * 32 $\rightarrow$ 64

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

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.

2.4.8 Packed Multiply and Packed MAC

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

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

These instructions perform two 16 x 16 bit multiplications in parallel, using 16-bit source operands in the upper or lower halves of their source operand registers.

MUL.H produces two 32-bit products, stored into the upper and lower registers of an extended register pair. Its results are exact, with no need for rounding.

MULR.H produces two 16-bit Q-format products, stored into the upper and lower halves of a single 32-bit register. Its 32-bit intermediate products are rounded before discarding the low order bits, to produce the 16-bit Q-format results.

MULM.H sums the two intermediate products, producing a single result that is stored into an extended register.

For all three instruction groups there are four supported source operand combinations for the two multiplications. They are:

- 16U * 16U, 16L * 16L
- 16U * 16L, 16L * 16U
- 16U * 16L, 16L * 16L
- 16L * 16U, 16U * 16U

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

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

2.5 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.

2.5.1 Simple Compare

Figure 2-8 illustrates the operation of the LT (Less Than) compare instruction:

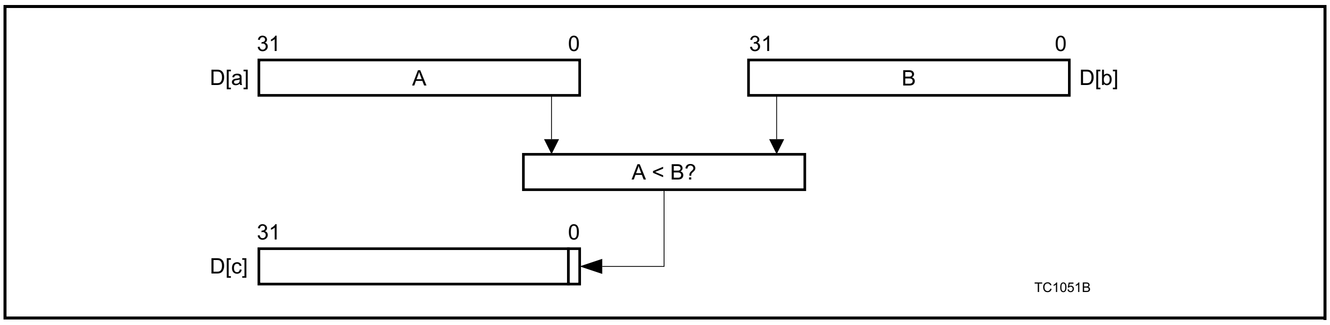


Figure 2-8 LT (Less Than) Comparison

The comparison instructions are:

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

Versions for both signed and unsigned integers are available.

Comparison conditions not explicitly provided in the instruction set can be obtained by either swapping the operands when comparing two registers, or by incrementing the constant by one when comparing a register and a constant (See **Table 2-1**).

Table 2-1 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)

2.5.2 Accumulating Compare

To accelerate the computation of complex conditional expressions, accumulating versions of the comparison instructions are supported.

These instructions, indicated in the instruction mnemonic by ‘op’ preceding the ‘.’ (for example, op.LT), combine the result of the comparison with a previous comparison result.

The combination is a logic AND, OR, or XOR; for example, AND.LT, OR.LT, and XOR.LT.

Figure 2-9 illustrates the combination of the LT instruction with a Boolean operation.

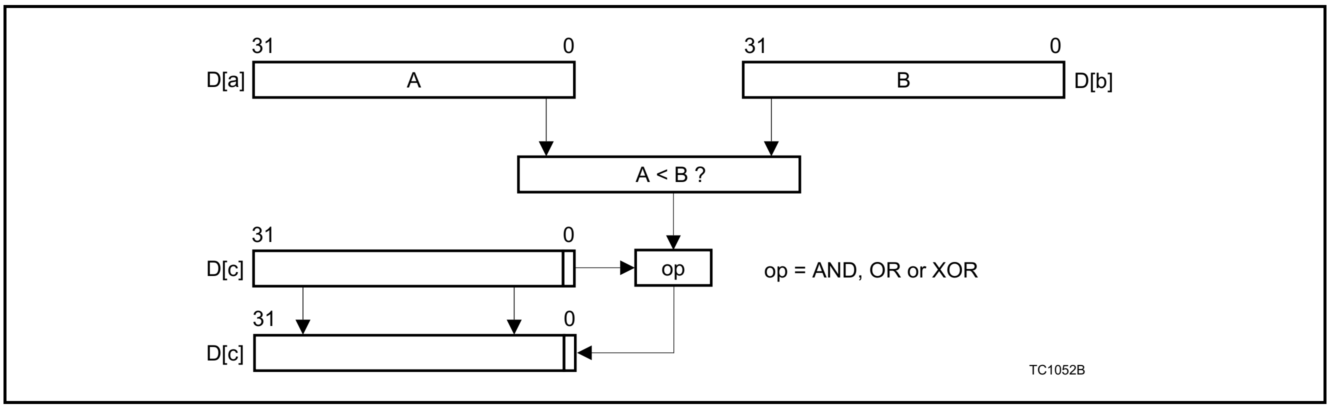


Figure 2-9 Combining LT Comparison with Boolean Operation

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

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

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

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

2.5.3 Compare with Shift

Certain control applications require that several Booleans be packed into a single register. These packed bits can be used as an index into a table of constants or a jump table, which permits complex Boolean functions and/or state machines to be evaluated efficiently.

To facilitate the packing of Boolean results into a register, compound Compare with Shift instructions (for example SH.EQ) are supported.

The result of the comparison is placed in the least-significant bit of the result after the contents of the destination register have been shifted left by one position.

Figure 2-10 illustrates the operation of the SH.LT (Shift Less Than) instruction.

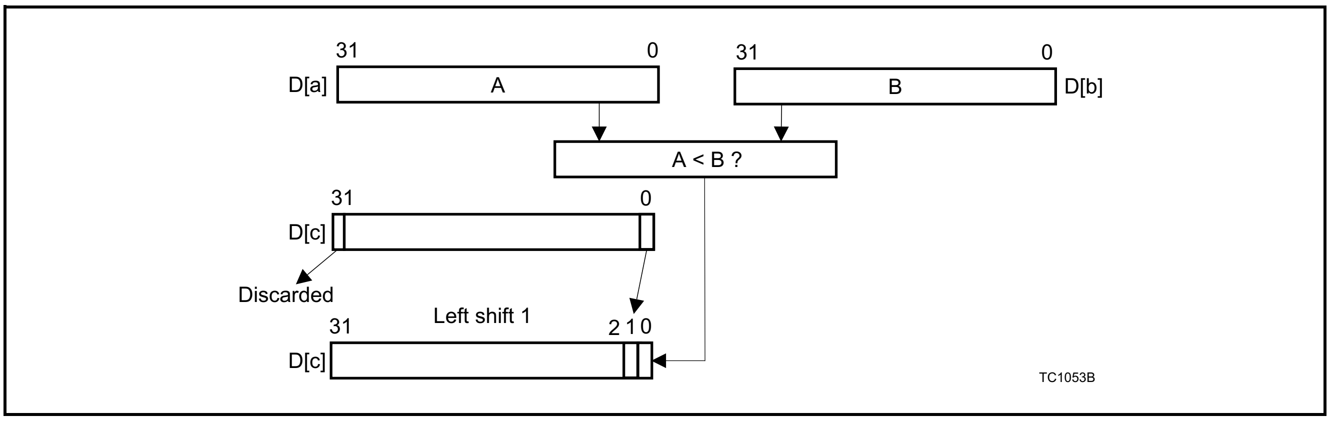


Figure 2-10 SH.LT Instruction

2.5.4 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. Figure 2-11 shows an example of the EQ.B instruction.

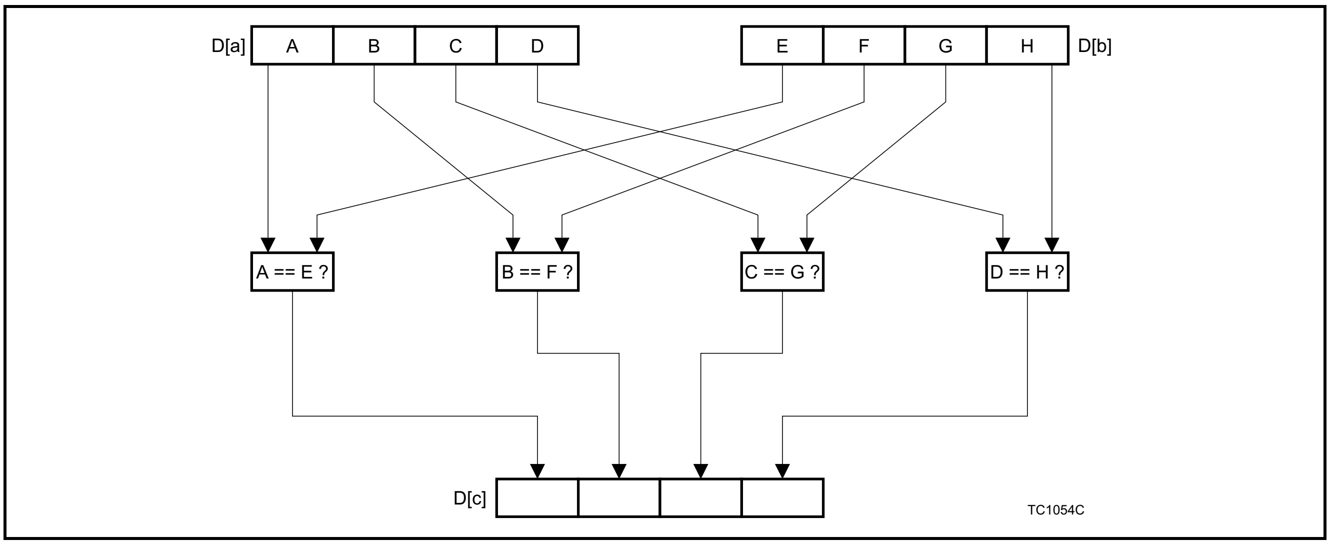


Figure 2-11 EQ.B Instruction Operation

2.6 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.

2.6.1 Simple Bit Operations

The one-bit result of a two-input function (for example, AND.T) is stored in the least significant bit of the destination data register, and the most significant 31-bits are set to zero. The source bits can be any bit of any data register. This is illustrated in Figure 2-12. The available Boolean operations are:

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

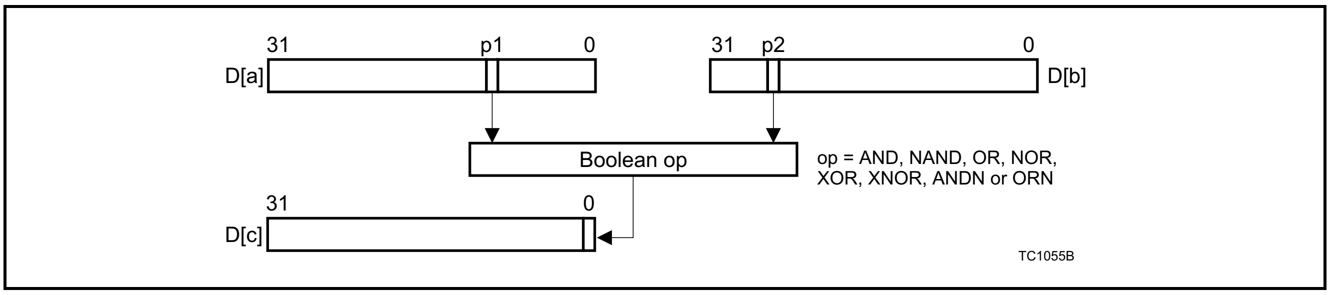


Figure 2-12 Boolean Operations

2.6.2 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 2-13**).

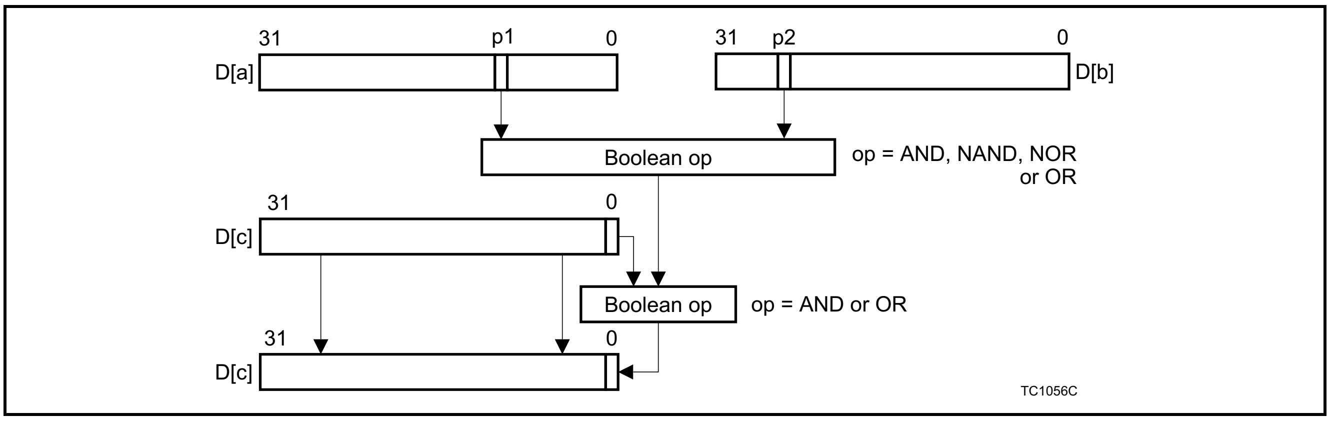


Figure 2-13 Three-Input Boolean Operation

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

2.6.3 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 2-14**, **Page 2-15**).

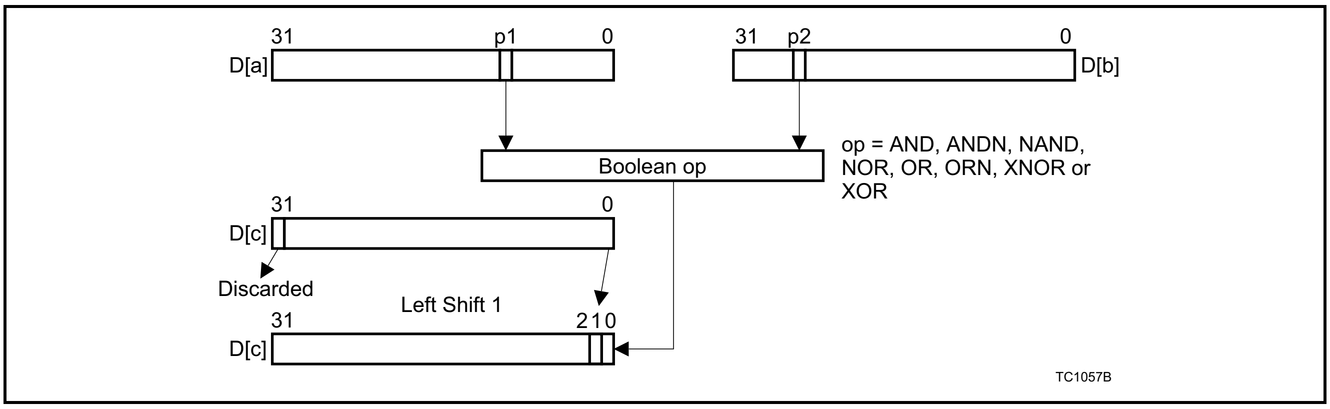


Figure 2-14 Shift Plus Boolean Operation

2.7 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.

2.8 Address Comparison

As with the comparison instructions that use the data registers (see **Compare Instructions, page 2-11**), the comparison instructions using the address registers put the result of the comparison in the least-significant bit of the destination data register and clear the remaining register bits to zeros.

An example using the Less Than (LT.A) instruction is shown in **Figure 2-15**:

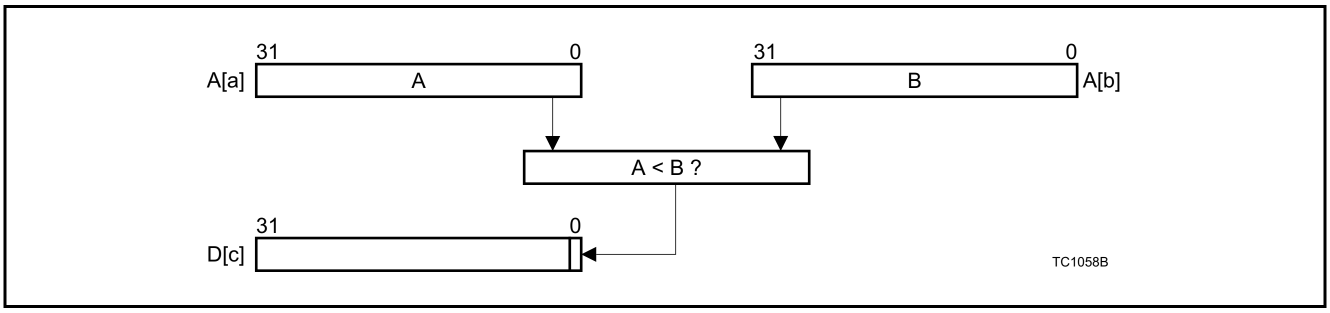


Figure 2-15 LT.A Comparison Operation

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 2-2).

Table 2-2 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.

2.9 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.

2.9.1 Unconditional Branch

There are three groups of unconditional branch instructions:

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

A Jump instruction simply loads the Program Counter with the address specified in the instruction. A Jump and Link instruction does the same, and also stores the address of the next instruction in the Return Address (RA) register A[11]. A jump and Link instruction can be used to implement a subroutine call when the called routine does not modify any of the caller's non-volatile registers.

The Call instructions differ from a Jump and Link in that the call instructions save the caller's registers upper context in a dynamically-allocated save area.

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

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

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

Both the 24-bit and 8-bit relative displacements are scaled by two before they are used, because all instructions must be aligned on an even address. The use of a 24-bit field as an absolute address is shown in **Figure 2-16**.

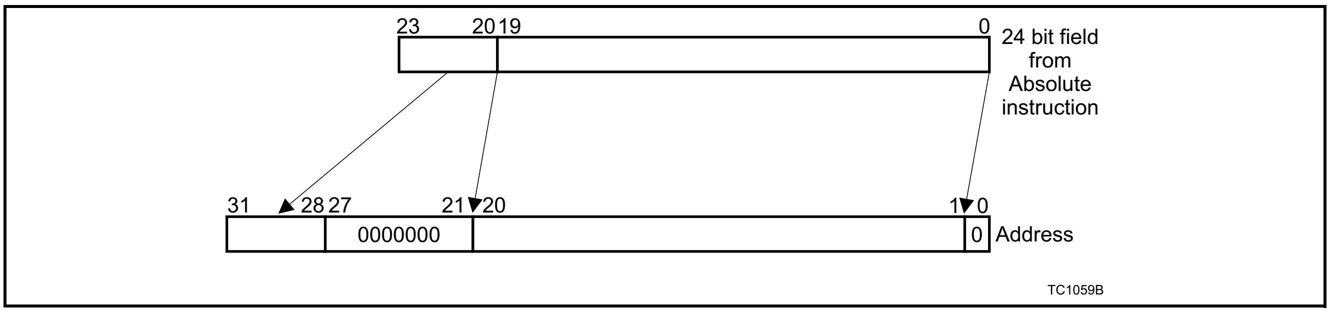


Figure 2-16 Calculation of Absolute Address

2.9.2 Conditional Branch

The conditional branch instructions use the relative addressing mode, with a displacement value encoded in 4, 8 or 15-bits.

The displacement is scaled by 2 before it is used, because all instructions must be aligned on an even address.

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

Conditional Jumps on Data Registers

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

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

The second operand to be compared may be an 8-bit sign or zero-extended constant. There are two 16-bit instructions that test whether the implicit D[15] register is equal to zero (JZ) or not equal to zero (JNZ). The displacement is 8-bit in this case.

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

2.9.3 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:

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

Loop Instructions with Reduced Execution Overhead

The LOOP instruction is a special kind of jump which utilizes the TriCore hardware that implements ‘zero overhead’ loops.

The LOOP instruction only requires execution time in the pipeline the first and last time it is executed (for a given loop). For all other iterations of the loop, the LOOP instruction has zero execution time.

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

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

This LOOP instruction (in the example above) requires execution cycles the first time it is executed, but the other 99 executions require no cycles.

Note that the LOOP instruction differs from the other conditional jump instructions in that it uses an address register, rather than a data register, for the iteration count. This allows 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.

2.10 Load and Store Instructions

The load (LD.x) and store (ST.x) instructions move data between registers and memory using seven addressing modes (**Table 2-3**).

The addressing mode determines the effective byte address for the load or store instruction and any update of the base pointer address register.

Table 2-3 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+r]	A[n]+A[n+1][15:0] (n is even)	BO

2.10.1 Load/Store Basic Data Types

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

Note that when the data loaded from memory is smaller than the destination register (i.e. 8-bit and 16-bit quantities), the data is loaded into the least-significant bits of the register (except for fractions which are loaded into the most-significant bits of a register), and the remaining register bits are sign or zero-extended to 32-bits, depending on the particular instruction.

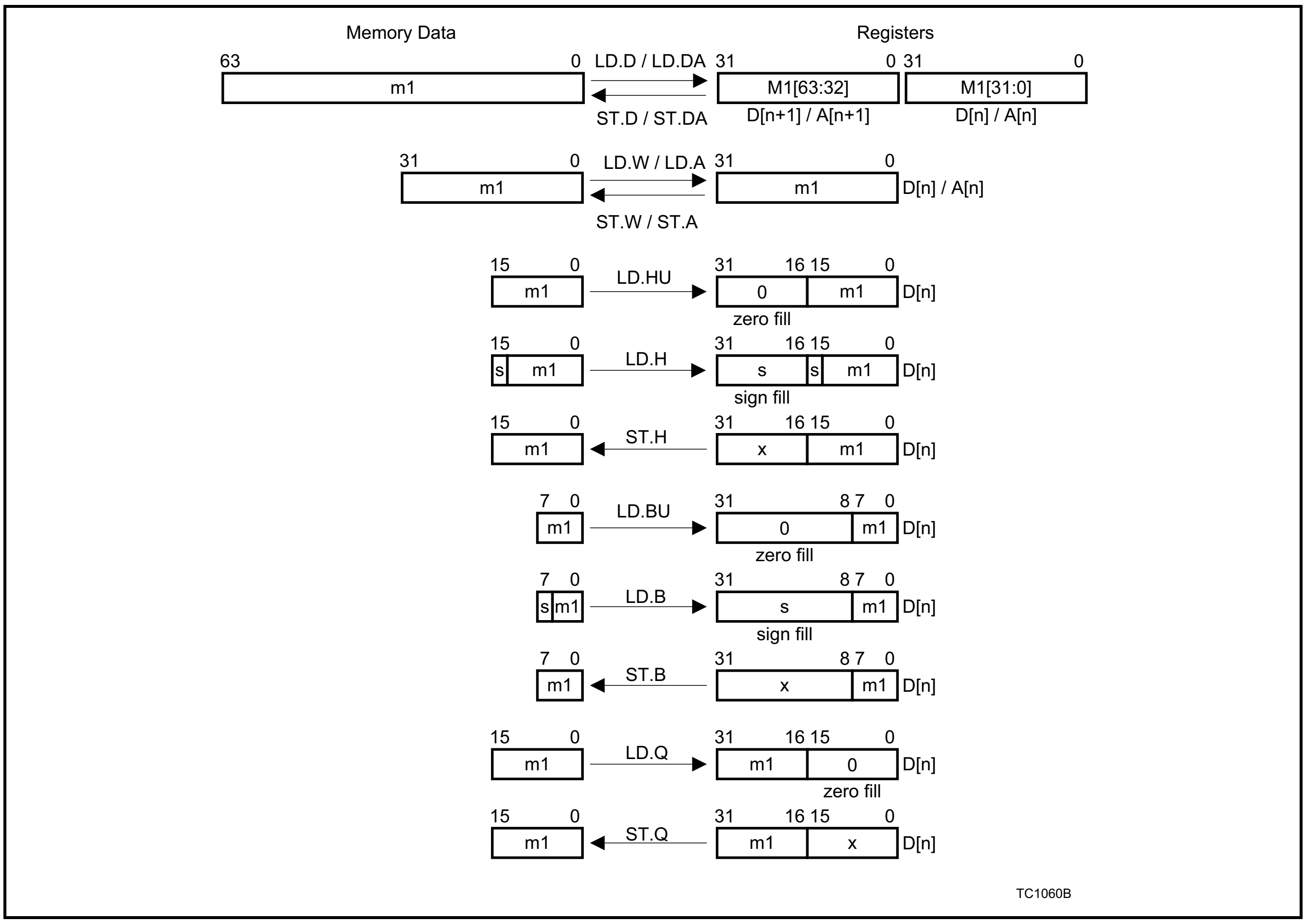


Figure 2-17 Load/Store Basic Data Type

2.10.2 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; i.e. the bits that were either shifted out or masked out above. For example:

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

2.10.3 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 1011B 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
```

2.11 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.

2.11.1 Lower Context Saving and Restoring

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

The SVLCX instruction (Save Lower Context) saves registers A[2] through A[7] and D[0] through D[7], together with the return address (RA) in register A[11] and the PCXI. This operation is performed when using the FCX and PCX pointers to manage the CSA lists.

The RSLCX instruction (Restore Lower Context) restores the lower context. It loads registers A[2] through A[7] and D[0] through D[7] from the CSA. It also loads A[11] (Return Address) from the saved PC field. This operation is performed when using the FCX and PCX pointers to manage the CSA lists.

The BISR instruction (Begin Interrupt Service Routine) enables the interrupt system (ICR.IE = 1), allows the modification of the CPU priority number (CCPN), and saves the lower context in the same manner as the SVLCX instruction.

2.11.2 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.

2.12 System Instructions

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

2.12.1 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.

2.12.2 Synchronization Primitives (DYSNC and ISYNC)

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

DSYNC

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

The DSYNC instruction guarantees that all data accesses associated with instructions semantically prior to the DSYNC instruction are completed before any data memory accesses associated with an instruction semantically after DSYNC are initiated. This includes all accesses to the system bus and local data memory.

ISYNC

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

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

The operation of the ISYNC instruction is as follows:

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

2.12.3 Access to the Core Special Function Registers (CSFRs)

The core accesses the CSFRs through two instructions:

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

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

The CSFRs are also mapped into the memory address space. This mapping makes the complete architectural state of the core visible in the address map, which allows efficient debug and emulator support. Note that it is not permitted for the core to access the CSFRs through this mechanism. The core must use MFCR and MTCR. Figure 2-18 summarizes TriCore core behaviour when accessing CSFRs.

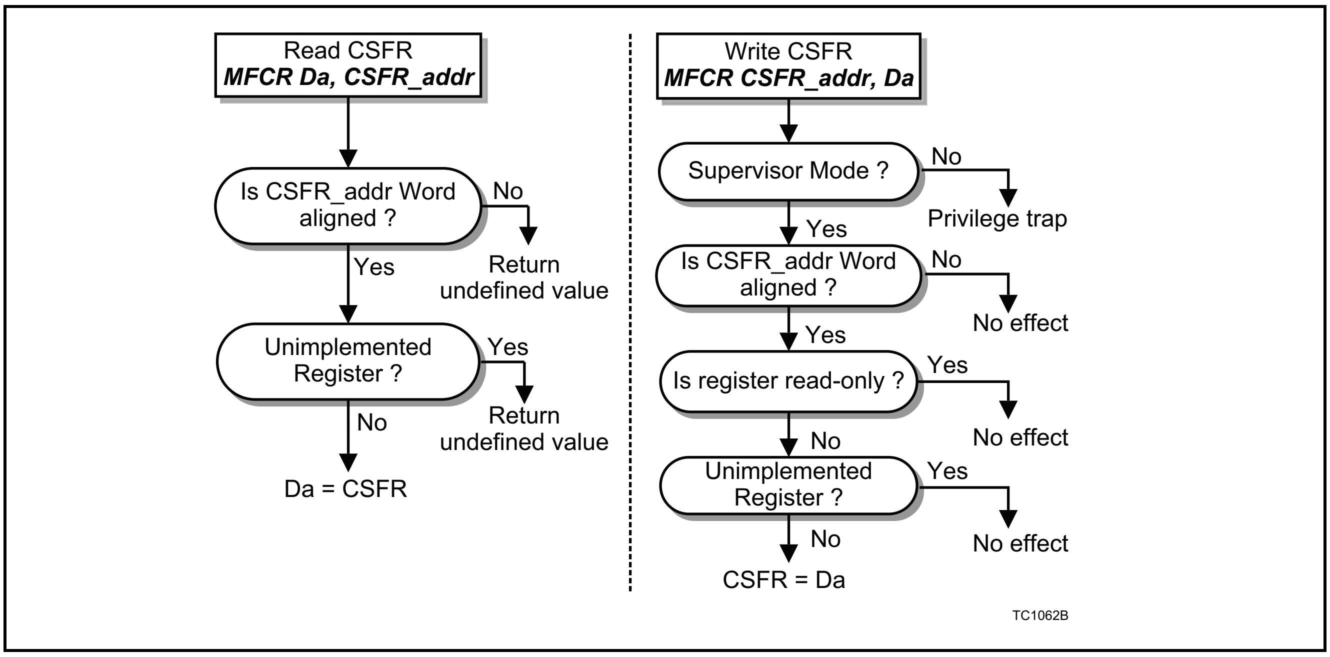


Figure 2-18 TriCore, Core Behaviour Accessing CSFRs

2.12.4 Enabling and Disabling the Interrupt System

For non-interruptible operations, the ENABLE and DISABLE instructions allow the explicit enabling and disabling of interrupts in both User and Supervisor mode. While disabled, an interrupt will not be taken by the CPU regardless of the relative priorities of the CPU and the highest interrupt pending. The only 'interrupt' that is serviced while interrupts are disabled is the NMI (Non-Maskable Interrupt), because it bypasses the normal interrupt structure.

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

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

2.12.5 Return (RET) and Return From Exception (RFE) Instructions

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

These two instructions perform very similar operations; they restore the upper context of the calling function or interrupted task and branch to the return address contained in register A[11] (prior to the context restore operation).

The two instructions differ in the error checking they perform for call depth management. Issuing an RFE instruction when the current call depth (as tracked in the PSW) is non-zero, generates a context nesting error trap. Conversely, a context call depth underflow trap is generated when an RET instruction is issued when the current call depth is zero.

2.12.6 Trap Instructions

The Trap on Overflow (TRAPV) and Trap on Sticky Overflow (TRAPSV) instructions can be used to cause a trap if the PSWs V and SV bits respectively, are set. See **PSW (Program Status Word) Status Flags and Arithmetic Instructions, page 2-8**.

2.12.7 No-Operation (NOP)

Although there are many ways to represent a no-operation (for example, adding zero to a register), an explicit NOP instruction is included so that it can be easily recognized.

2.13 Coprocessor (COP) Instructions

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

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

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

2.14 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).

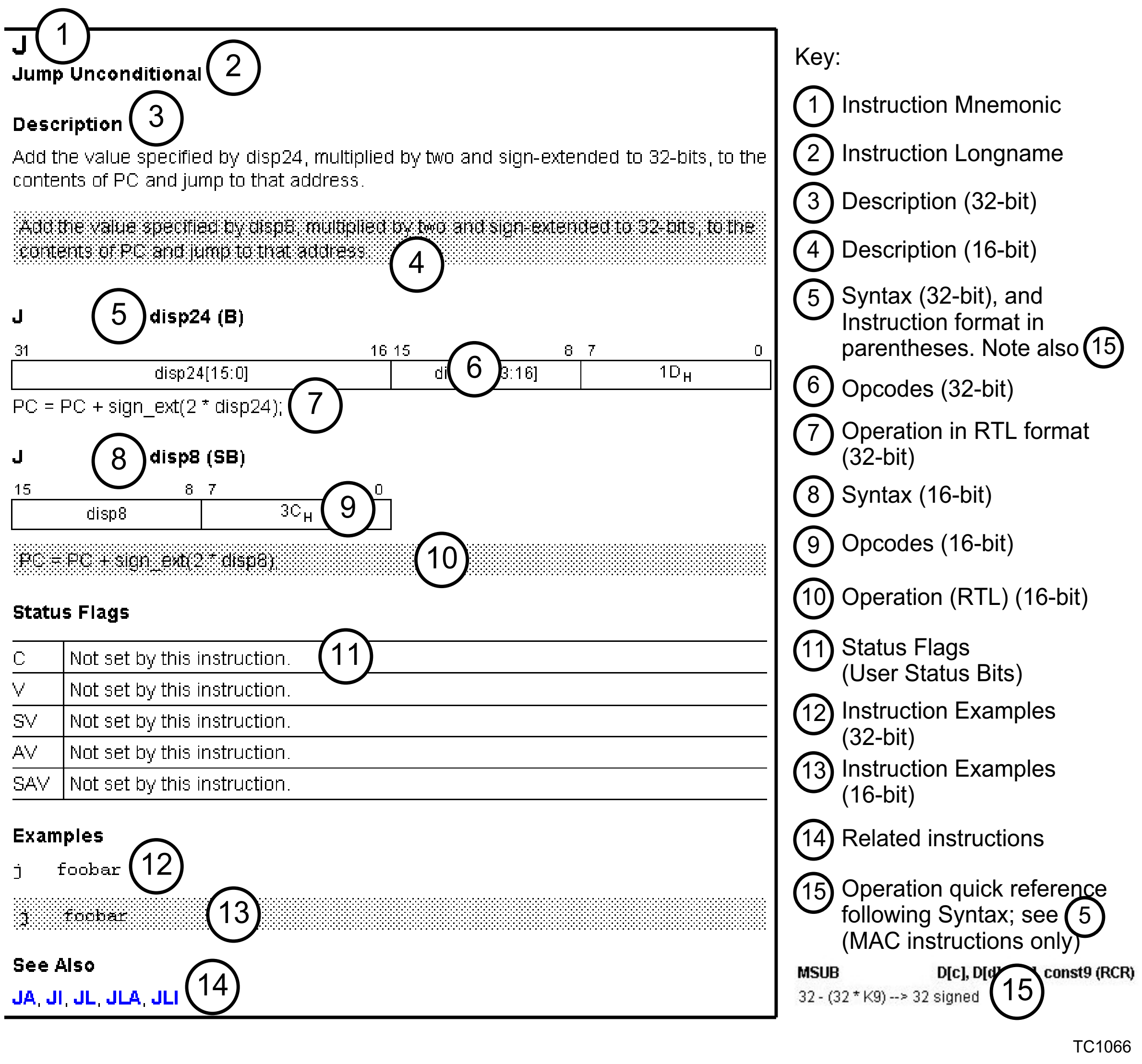
3 Instruction Set

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

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

3.1 CPU Instructions

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



J (1)

Jump Unconditional (2)

Description (3)

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

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

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

PC = PC + sign_ext(2 * disp24); (7)

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

PC = PC + sign_ext(2 * disp8); (10)

Status Flags

C	Not set by this instruction. (11)
V	Not set by this instruction.
SV	Not set by this instruction.
AV	Not set by this instruction.
SAV	Not set by this instruction.

Examples

```
j    foobar
```
(12)

```
j    foobar
```
(13)

See Also

JA, JI, JL, JLA, JLI (14)

Key:
1 Instruction Mnemonic
2 Instruction Longname
3 Description (32-bit)
4 Description (16-bit)
5 Syntax (32-bit), and Instruction format in parentheses. Note also 15
6 Opcodes (32-bit)
7 Operation in RTL format (32-bit)
8 Syntax (16-bit)
9 Opcodes (16-bit)
10 Operation (RTL) (16-bit)
11 Status Flags (User Status Bits)
12 Instruction Examples (32-bit)
13 Instruction Examples (16-bit)
14 Related instructions
15 Operation quick reference following Syntax; see 5 (MAC instructions only)
MSUB D[c], D[d] const9 (RCR)
32 - (32 * K9) --> 32 signed
15
TC1066

ABS
Absolute Value

Description

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

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

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

31 28	27 20	19 18	17 16	15 12	11 8	7 0
c	$1C_H$	-	-	b	-	$0B_H$

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 > $7FFFFFFF_H$) 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.SV = PSW.SV;

Examples

```
abs   d3, d1
```

See Also

ABSDIF, **ABSDIFS**, **ABSS**

ABS.B
Absolute Value Packed Byte
ABS.H
Absolute Value Packed Half-word

Description

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

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

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

31 28	27 20	19 18	17 16	15 12	11 8	7 0
c	$5C_H$	-	-	b	-	$0B_H$

result_byte3 = (D[b][31:24] >= 0) ? D[b][31:24] : (0 - D[b][31:24]);

result_byte2 = (D[b][23:16] >= 0) ? D[b][23:16] : (0 - D[b][23:16]);

result_byte1 = (D[b][15:8] >= 0) ? D[b][15:8] : (0 - D[b][15:8]);

result_byte0 = (D[b][7:0] >= 0) ? D[b][7:0] : (0 - D[b][7:0]);

D[c] = {result_byte3[7:0], result_byte2[7:0], result_byte1[7:0], result_byte0[7:0]};

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

31 28	27 20	19 18	17 16	15 12	11 8	7 0
c	$7C_H$	-	-	b	-	$0B_H$

result_halfword1 = (D[b][31:16] >= 0) ? D[b][31:16] : (0 - D[b][31:16]);

result_halfword0 = (D[b][15:0] >= 0) ? D[b][15:0] : (0 - D[b][15:0]);

D[c] = {result_halfword1[15:0], result_halfword0[15:0]};

Status Flags

C	Not set by these instructions.

V	ABS.B
	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;
	ABS.H
	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	ABS.B
	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;
	ABS.H
	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.b   d3, d1
abs.h   d3, d1
```

See Also

ABSS.H, **ABSDIF.B**, **ABSDIF.H**, **ABSDIFS.H**

ABSDIF
Absolute Value of Difference

Description

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

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

31 28	27 21	20 12	11 8	7 0
c	$0E_H$	const9	a	$8B_H$

result = (D[a] > sign_ext(const9)) ? D[a] - sign_ext(const9) : sign_ext(const9) - D[a];

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

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

31 28	27 20	19 18	17 16	15 12	11 8	7 0
c	$0E_H$	-	-	b	a	$0B_H$

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

See Also

ABS, **ABSS**, **ABSDIFS**

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

Description

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

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

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

31 28	27 20	19 18	17 16	15 12	11 8	7 0
c	$4E_H$	-	-	b	a	$0B_H$

```
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]};
```

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

31 28	27 20	19 18	17 16	15 12	11 8	7 0
c	$6E_H$	-	-	b	a	$0B_H$

```
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 these instructions.

V	ABSDIF.B
	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;
	ABSDIF.H
	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	ABSDIF.B
	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;
	ABSDIF.H
	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.b    d3, d1, d2
absdif.h    d3, d1, d2
```

See Also

ABS.B, **ABS.H**, **ABSS.H**, **ABSDIFS.H**

ABSDIFS
Absolute Value of Difference with Saturation

Description

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

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

31 28	27 21	20 12	11 8	7 0
c	$0F_H$	const9	a	$8B_H$

result = (D[a] > sign_ext(const9)) ? D[a] - sign_ext(const9) : sign_ext(const9) - D[a];

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

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

31 28	27 20	19 18	17 16	15 12	11 8	7 0
c	$0F_H$	-	-	b	a	$0B_H$

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

See Also

ABS, **ABSDIF**, **ABSS**

ABSDIFS.H

Absolute Value of Difference Packed Half-word with Saturation

Description

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

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

31 28	27 20	19 18	17 16	15 12	11 8	7 0
c	$6F_H$	-	-	b	a	$0B_H$

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**

ABSS
Absolute Value with Saturation

Description

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

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

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

31 28	27 20	19 18	17 16	15 12	11 8	7 0
c	$1D_H$	-	-	b	-	$0B_H$

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 > $7FFFFFFF_H$) 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

```
abss    d3, d1
```

See Also

ABS, ABSDIF, ABSDIFS

ABSS.H

Absolute Value Packed Half-word with Saturation

Description

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

ABSS.HD[c], D[b] (RR)

31 28	27 20	19	18 17	16 15 12	11 8	7 0
c	$7D_H$	-	-	b	-	$0B_H$

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 > $7FFF_H$) 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

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

See Also

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

ADD
Add

Description

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

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

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

31 28	27 21	20 12	11 8	7 0
c	00_H	const9	a	$8B_H$

result = D[a] + sign_ext(const9);

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

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

31 28	27 20	19 18	17 16	15 12	11 8	7 0
c	00_H	-	-	b	a	$0B_H$

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

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

ADDD[a], const4 (SRC)

15 12	11 8	7 0
const4	a	$C2_H$

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

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

15 12	11 8	7 0
const4	a	92_H

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

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

15 12	11 8	7 0
const4	a	$9A_H$

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

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

15 12	11 8	7 0
b	a	42_H

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

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

15 12	11 8	7 0
b	a	12_H

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

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

15 12	11 8	7 0
b	a	$1A_H$

```
result = D[a] + D[b];
D[15] = 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, d2
add   d3, d1, #126
```

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

See Also

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

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

31 28	27 20	19 18	17 16	15 12	11 8	7 0
c	01_H	-	-	b	a	01_H

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

ADD.AA[a], const4 (SRC)

15 12	11 8	7 0
const4	a	$B0_H$

A[a] = A[a] + sign_ext(const4);

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

15 12	11 8	7 0
b	a	30_H

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

Status Flags

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

Examples

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

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

See Also

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

ADD.B
Add Packed Byte
ADD.H
Add Packed Half-word

Description

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

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

31 28	27 20	19 18	17 16	15 12	11 8	7 0
c	40_H	-	-	b	a	$0B_H$

result_byte3 = D[a][31:24] + D[b][31:24];

result_byte2 = D[a][23:16] + D[b][23:16];

result_byte1 = D[a][15:8] + D[b][15:8];

result_byte0 = D[a][7:0] + D[b][7:0];

D[c] = {result_byte3[7:0], result_byte2[7:0], result_byte1[7:0], result_byte0[7:0]};

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

31 28	27 20	19 18	17 16	15 12	11 8	7 0
c	60_H	-	-	b	a	$0B_H$

result_halfword1 = D[a][31:16] + D[b][31:16];

result_halfword0 = D[a][15:0] + D[b][15:0];

D[c] = {result_halfword1[15:0], result_halfword0[15:0]};

Status Flags

C	Not set by these instructions.

V	ADD.B
	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;
	ADD.H
	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	ADD.B
	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;
	ADD.H
	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.b   d3, d1, d2
add.h   d3, d1, d2
```

See Also

ADDS.H, **ADDS.HU**

ADDC
Add with Carry

Description

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

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

31 28	27 21	20 12	11 8	7 0
c	05_H	const9	a	$8B_H$

result = D[a] + sign_ext(const9) + PSW.C;

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

carry_out = carry(D[a],sign_ext(const9),PSW.C);

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

31 28	27 20	19 18	17 16	15 12	11 8	7 0
c	05_H	-	-	b	a	$0B_H$

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, d2
addc   d3, d1, #126
```

See Also

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

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.

ADDID[c], D[a], const16 (RLC)

31 28	27 12	11 8	7 0
c	const16	a	$1B_H$

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**

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.

ADDIHD[c], D[a], const16 (RLC)

31 28	27 12	11 8	7 0
c	const16	a	$9B_H$

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**

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.AA[c], A[a], const16 (RLC)

31 28	27 12	11 8	7 0
c	const16	a	11_H

A[c] = A[a] + {const16, 16’h0000};

Status Flags

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

Examples

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

See Also

ADD.A, **ADDSC.A**, **ADDSC.AT**, **SUB.A**

ADDS
Add Signed with Saturation

Description

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

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

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

31 28	27 21	20 12	11 8	7 0
c	02_H	const9	a	$8B_H$

result = D[a] + sign_ext(const9);

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

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

31 28	27 20	19 18	17 16	15 12	11 8	7 0
c	02_H	-	-	b	a	$0B_H$

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

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

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

15 12	11 8	7 0
b	a	22_H

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

See Also

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

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

Description

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

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

31 28	27 20	19 18	17 16	15 12	11 8	7 0
c	62_H	-	-	b	a	$0B_H$

result_halfword1 = D[a][31:16] + D[b][31:16];

result_halfword0 = D[a][15:0] + D[b][15:0];

D[c] = {ssov(result_halfword1, 16), ssov(result_halfword0, 16)};

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

31 28	27 20	19 18	17 16	15 12	11 8	7 0
c	63_H	-	-	b	a	$0B_H$

result_halfword1 = D[a][31:16] + D[b][31:16]; // unsigned addition

result_halfword0 = D[a][15:0] + D[b][15:0]; // unsigned addition

D[c] = {suov(result_halfword1, 16), suov(result_halfword0, 16)};

Status Flags

C	Not set by these instructions.

V	ADDS.H
	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;
	ADDS.HU
	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

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

See Also

ADD.B, ADD.H

ADDS.U
Add Unsigned with Saturation

Description

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

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

31 28	27 21	20 12	11 8	7 0
c	03_H	const9	a	$8B_H$

result = D[a] + sign_ext(const9); // unsigned addition

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

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

31 28	27 20	19 18	17 16	15 12	11 8	7 0
c	03_H	-	-	b	a	$0B_H$

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

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

Status Flags

C	Not set by this instruction.

V	overflow = (result > $FFFFFFFF_H$) 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

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

See Also

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

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

Description

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

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

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

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

31 28	27 20	19 18	17 16	15 12	11 8	7 0
c	60_H	-	n	b	a	01_H

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

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

15 12	11 8	7 6	5 0
b	a	n	10_H

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

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

31 28	27 20	19 18	17 16	15 12	11 8	7 0
c	62_H	-	-	b	a	01_H

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

Status Flags

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

Examples

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

See Also

ADD.A, ADDIH.A, SUB.A

ADDX
Add Extended

Description

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

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

31 28	27 21	20 12	11 8	7 0
c	04_H	const9	a	$8B_H$

result = D[a] + sign_ext(const9);

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

carry_out = carry(D[a],sign_ext(const9),0);

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

31 28	27 20	19 18	17 16	15 12	11 8	7 0
c	04_H	-	-	b	a	$0B_H$

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, d2
addx   d3, d1, #126
```

See Also

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

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.

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

31 28	27 21	20 12	11 8	7 0
c	08_H	const9	a	$8F_H$

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

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

31 28	27 20	19 18	17 16	15 12	11 8	7 0
c	08_H	-	-	b	a	$0F_H$

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

ANDD[15], const8 (SC)

15 8	7 0
const8	16_H

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

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

15 12	11 8	7 0
b	a	26_H

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

Status Flags

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

Examples

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

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

See Also

ANDN, **NAND**, **NOR**, **NOT (16-bit)**, **OR**, **ORN**, **XNOR**, **XOR**

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

Description

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

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

31 28	27 23	22 21	20 16	15 12	11 8	7 0
c	pos2	00_H	pos1	b	a	47_H

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

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

31 28	27 23	22 21	20 16	15 12	11 8	7 0
c	pos2	03_H	pos1	b	a	47_H

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

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

31 28	27 23	22 21	20 16	15 12	11 8	7 0
c	pos2	02_H	pos1	b	a	47_H

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

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

31 28	27 23	22 21	20 16	15 12	11 8	7 0
c	pos2	01_H	pos1	b	a	47_H

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

Status Flags

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

SAV	Not set by these instructions.

Examples

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

See Also

OR.AND.T, **OR.ANDN.T**, **OR.NOR.T**, **OR.OR.T**, **SH.AND.T**, **SH.ANDN.T**, **SH.NAND.T**, **SH.NOR.T**, **SH.OR.T**, **SH.ORN.T**, **SH.XNOR.T**, **SH.XOR.T**

AND.EQ
Equal Accumulating

Description

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

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

31 28	27 21	20 12	11 8	7 0
c	20_H	const9	a	$8B_H$

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

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

31 28	27 20	19 18	17 16	15 12	11 8	7 0
c	20_H	-	-	b	a	$0B_H$

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

See Also

OR.EQ, XOR.EQ

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

Description

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

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

31 28	27 21	20 12	11 8	7 0
c	24_H	const9	a	$8B_H$

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

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

31 28	27 20	19 18	17 16	15 12	11 8	7 0
c	24_H	-	-	b	a	$0B_H$

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

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

31 28	27 21	20 12	11 8	7 0
c	25_H	const9	a	$8B_H$

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

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

31 28	27 20	19 18	17 16	15 12	11 8	7 0
c	25_H	-	-	b	a	$0B_H$

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

Status Flags

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

Examples

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

See Also

OR.GE, **OR.GE.U**, **XOR.GE**, **XOR.GE.U**

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

Description

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

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

31 28	27 21	20 12	11 8	7 0
c	22_H	const9	a	$8B_H$

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

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

31 28	27 20	19 18	17 16	15 12	11 8	7 0
c	22_H	-	-	b	a	$0B_H$

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

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

31 28	27 21	20 12	11 8	7 0
c	23_H	const9	a	$8B_H$

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

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

31 28	27 20	19 18	17 16	15 12	11 8	7 0
c	23_H	-	-	b	a	$0B_H$

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

Status Flags

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

Examples

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

See Also

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

AND.NE
Not Equal Accumulating

Description

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

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

31 28	27 21	20 12	11 8	7 0
c	21_H	const9	a	$8B_H$

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

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

31 28	27 20	19 18	17 16	15 12	11 8	7 0
c	21_H	-	-	b	a	$0B_H$

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

See Also

OR.NE, XOR.NE

AND.T
Bit Logical AND

Description

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

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

31 28	27 23	22 21	20 16	15 12	11 8	7 0
c	pos2	00_H	pos1	b	a	87_H

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**

ANDN
Bitwise AND-Not

Description

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

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

31 28	27 21	20 12	11 8	7 0
c	$0E_H$	const9	a	$8F_H$

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

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

31 28	27 20	19 18	17 16	15 12	11 8	7 0
c	$0E_H$	-	-	b	a	$0F_H$

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

See Also

AND, **NAND**, **NOR**, **NOT (16-bit)**, **OR**, **ORN**, **XNOR**, **XOR**

ANDN.T
Bit Logical AND-Not

Description

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

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

31 28	27 23	22 21	20 16	15 12	11 8	7 0
c	pos2	03_H	pos1	b	a	87_H

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

D[c] = zero_ext(result);

Status Flags

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

Examples

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

See Also

AND.T, **NAND.T**, **NOR.T**, **OR.T**, **ORN.T**, **XNOR.T**, **XOR.T**

BISR
Begin Interrupt Service Routine

Description

Note: BISR can only be executed in Supervisor mode.

Save the lower context by storing the contents of A[2]-A[7], D[0]-D[7], and the current A[11] (return address) to the current memory location pointed to by the FCX.

Set the current CPU priority number (ICR.CCPN) to the value of either const9[7:0] (instruction format RC) or const8 (instruction format SC), and enable interrupts (set ICR.IE to one).

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

BISRconst9 (RC)

31 28	27 21	20 12	11 8	7 0
-	00_H	const9	-	AD_H

```
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);
```

BISRconst8 (SC)

15 8	7 0
const8	$E0_H$

```
tmp_FCX = FCX;
```

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

Status Flags

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

Examples

```
bisr    #126
```

```
bisr    #126
```

See Also

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

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.

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

31 28	27 20	19 18	17 16	15 12	11 8	7 0
c	01_H	-	0_H	b	a	$4B_H$

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

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

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

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

Status Flags

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

Examples

```
bmerge   d0, d1, d2
```

See Also

BSPLIT

BSPLIT
Bit Split

Description

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

BSPLITE[c], D[a] (RR)

31 28	27 20	19 18	17 16	15 12	11 8	7 0
c	09_H	-	0_H	-	a	$4B_H$

E[c][63:48] = 0000_H;

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

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

E[c][31:16] = 0000_H;

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

CACHEA.I
Cache Address, Invalidate

Description

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

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

If the cache line containing the byte memory location specified by the addressing mode is not present in the L1 data cache, then no operation should be performed in the L1 data cache. Specifically a refill of the line containing the byte pointed to by the effective address should not be performed. Any address register updates associated with the addressing mode are always performed regardless of the cache operation. The effective address is a virtual address when operating in virtual mode.

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

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

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

CACHEA.IP[b] (BO) (Bit Reverse Addressing Mode)

31 28	27 22	21 16	15 12	11 8	7 0
-	0E$_H$	-	b	-	A9$_H$

```
index = zero_ext(A[b+1][15:0]);
incr = zero_ext(A[b+1][31:16]);
EA = A[b] + index;
cache_address_ivld(EA);
new_index = reverse16(reverse16(index) + reverse16(incr));
A[b+1] = {incr[15:0],new_index[15:0]};
```

CACHEA.IP[b], off10 (BO) (Circular Addressing Mode)

31 28	27 22	21 16	15 12	11 8	7 0
off10[9:6]	1E$_H$	off10[5:0]	b	-	A9$_H$

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

CACHEA.IA[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];

cache_address_ivld(EA);

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

CACHEA.IA[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);

cache_address_ivld(EA);

A[b] = EA;

Status Flags

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

Examples

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

See Also

CACHEA.W, **CACHEA.WI**, **CACHEI.I**, **CACHEI.W**, **CACHEI.WI**

CACHEA.W
Cache Address, Writeback

Description

If the cache line containing the byte memory location specified by the addressing mode is present in the L1 data cache, write back any modified data. The line will still be present in the L1 data cache and will be marked as unmodified.

If the cache line containing the byte memory location specified by the addressing mode is not present in the L1 data cache, then no operation should be performed in the L1 data cache. Specifically a refill of the line containing the byte pointed to by the effective address should not be performed. Any address register updates associated with the addressing mode are always performed regardless of the cache operation. The effective address is a virtual address when operating in virtual mode.

CACHEA.WA[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	-	89$_H$

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

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

31 28	27 22	21 16	15 12	11 8	7 0
-	0C$_H$	-	b	-	A9$_H$

```
index = zero_ext(A[b+1][15:0]);
incr = zero_ext(A[b+1][31:16]);
EA = A[b] + index;
cache_address_wb(EA);
new_index = reverse16(reverse16(index) + reverse16(incr));
A[b+1] = {incr[15:0], new_index[15:0]};
```

CACHEA.WP[b], off10 (BO)(Circular Addressing Mode)

31 28	27 22	21 16	15 12	11 8	7 0
off10[9:6]	1C$_H$	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;
cache_address_wb(EA);
new_index = index + sign_ext(off10);
new_index = new_index < 0 ? new_index+length : new_index % length;
A[b+1] = {length[15:0], new_index[15:0]};
```

CACHEA.WA[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];

cache_address_wb(EA);

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

CACHEA.WA[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);

cache_address_wb(EA);

A[b] = EA;

Status Flags

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

Examples

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

See Also

CACHEA.I, **CACHEA.WI**, **CACHEI.I**, **CACHEI.W**, **CACHEI.WI**

CACHEA.WI
Cache Address, Writeback and Invalidate

Description

If the cache line containing the byte memory location specified by the addressing mode is present in the L1 data cache, write back any modified data and then invalidate the line in the L1 data cache.

If the cache line containing the byte memory location specified by the addressing mode is not present in the L1 data cache then no operation should be performed in the L1 data cache. Specifically a refill of the line containing the byte pointed to by the effective address should not be performed. Any address register updates associated with the addressing mode are always performed regardless of the cache operation. The effective address is a virtual address when operating in virtual mode.

CACHEA.WIA[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);

cache_address_wi(EA);

CACHEA.WIP[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;

cache_address_wi(EA);

new_index = reverse16(reverse16(index) + reverse16(incr));

A[b+1] = {incr[15:0], new_index[15:0]};

CACHEA.WIP[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;

cache_address_wi(EA);

new_index = index + sign_ext(off10);

new_index = new_index < 0 ? new_index + length : new_index % length;

A[b+1] = {length[15:0], new_index[15:0]};

CACHEA.WIA[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];
cache_address_wi(EA);
A[b] = EA + sign_ext(off10);
```

CACHEA.WIA[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);
cache_address_wi(EA);
A[b] = EA;
```

Status Flags

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

Examples

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

See Also

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

CACHEI.W
Cache Index, Writeback

Description

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

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

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

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

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

index_way = A[b] + sign_ext(off10);

cache_index_wb(index_way);

CACHEI.WA[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

index_way = A[b];

cache_index_wb(index_way);

A[b] = index_way + sign_ext(off10);

CACHEI.WA[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

index_way = A[b] + sign_ext(off10);

cache_index_wb(index_way);

A[b] = index_way;

Status Flags

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

Examples

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

See Also

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

CACHEI.I
Cache Index, Invalidate

Description

This instruction can only be executed in Supervisor mode.

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

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

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

index_way = A[b] + sign_ext(off10);

cache_index_ivld(index_way);

CACHEI.IA[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

index_way = A[b];

cache_index_ivld(index_way);

A[b] = index_way + sign_ext(off10);

CACHEI.IA[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

index_way = A[b] + sign_ext(off10);

cache_index_ivld(index_way);

A[b] = index_way;

Status Flags

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

Examples

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

See Also

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

CACHEI.WI
Cache Index, Writeback, Invalidate

Description

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

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

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

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

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

index_way = A[b] + sign_ext(off10);

cache_index_wi(index_way);

CACHEI.WIA[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

index_way = A[b];

cache_index_wi(index_way);

A[b] = index_way + sign_ext(off10);

CACHEI.WIA[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

index_way = A[b] + sign_ext(off10);

cache_index_wi(index_way);

A[b] = index_way;

Status Flags

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

Examples

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

See Also

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

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.

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

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

condition = D[d] != 0;

result = ((condition) ? D[a] + sign_ext(const9) : D[a]);

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

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

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

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

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

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

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

15 12	11 8	7 0
const4	a	$8A_H$

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

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

See Also

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

CADDN
Conditional Add-Not

Description

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

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

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

31 28	27 24	23 21	20 12	11 8	7 0
c	d	01_H	const9	a	AB_H

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

result = ((condition) ? D[a] + sign_ext(const9) : D[a]);

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

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

31 28	27 24	23 20	19 18	17 16	15 12	11 8	7 0
c	d	01_H	-	-	b	a	$2B_H$

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

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

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

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

15 12	11 8	7 0
const4	a	CA_H

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

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

See Also

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

CALL
Call

Description

Add the value specified by disp24, multiplied by two and sign-extended, to the address of the CALL instruction and jump to the resulting address. The target address range is ±16 MBytes relative to the current PC. In parallel with the jump, save the caller's Upper Context to an available Context Save Area (CSA). Set register A[11] (return address) to the address of the next instruction beyond the call.

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

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

Add the value specified by disp8, multiplied by two and sign-extended, to the address of the CALL instruction, and jump to the resulting address. The target address range is ±256 bytes relative to the current PC.
In parallel with the jump, save the caller's Upper Context to an available Context Save Area (CSA). Set register A[11] (return address) to the address of the next instruction beyond the call.

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

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

CALLdisp24 (B)

31 … 16	15 … 8	7 … 0
disp24[15:0]	disp24[23:16]	$6D_H$

```
if (FCX == 0) trap(FCU);
if (PSW.CDE) then if (cdc_increment()) then trap(CDO);
PSW.CDE = 1;
ret_addr = PC + 4;
tmp_FCX = FCX;
EA = {FCX.FCXS, 6'b0, FCX.FCXO, 6'b0};
new_FCX = M(EA, word);
M(EA,16 * word) = {PCXI, PSW, A[10], A[11], D[8], D[9], D[10], D[11], A[12], A[13], A[14], A[15], D[12], D[13], D[14], D[15]};
PCXI.PCPN = ICR.CCPN;
PCXI.PIE = ICR.IE;
PCXI.UL = 1;
PCXI[19:0] = FCX[19:0];
FCX[19:0] = new_FCX[19:0];
PC = PC + sign_ext(2 * disp24);
A[11] = ret_addr[31:0];
if (tmp_FCX == LCX) trap(FCD);
```

CALLdisp8 (SB)

15 … 8	7 … 0
disp8	$5C_H$

```
if (FCX == 0) trap(FCU);
if (PSW.CDE) then if(cdc_increment()) then trap(CDO);
PSW.CDE = 1;
ret_addr = PC + 2 ;
tmp_FCX = FCX;
EA = {FCX.FCXS, 6'b0, FCX.FCXO, 6'b0};
new_FCX = M(EA, word);
M(EA,16 * word) = {PCXI, PSW, A[10], A[11], D[8], D[9], D[10], D[11], A[12], A[13], A[14], A[15], D[12], D[13],
D[14], D[15]};
PCXI.PCPN = ICR.CCPN;
PCXI.PIE = ICR.IE;
PCXI.UL = 1;
PCXI[19:0] = FCX[19:0];
FCX[19:0] = new_FCX[19:0];
PC = PC + sign_ext(2 * disp8);
A[11] = ret_addr[31:0];
if (tmp_FCX == LCX) trap(FCD);
```

Status Flags

C	Not changed by this instruction but read by the instruction.
V	Not changed by this instruction but read by the instruction.
SV	Not changed by this instruction but read by the instruction.
AV	Not changed by this instruction but read by the instruction.
SAV	Not changed by this instruction but read by the instruction.

Examples

```
call    foobar
```

```
call    foobar
```

See Also

CALLA, **CALLI**, **RET**, **FCALL**, **FRET**

CALLA
Call Absolute

Description

Jump to the address specified by disp24. In parallel with the jump, save the caller's Upper Context to an available Context Save Area (CSA). Set register A[11] (return address) to the address of the next instruction beyond the call.

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

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

CALLAdisp24 (B)

31 ... 16	15 ... 8	7 ... 0
disp24[15:0]	disp24[23:16]	ED_H

```
if (FCX == 0) trap(FCU);
if (PSW.CDE) then if (cdc_increment()) then trap(CDO);
PSW.CDE = 1;
ret_addr = PC + 4;
tmp_FCX = FCX;
EA = {FCX.FCXS, 6'b0, FCX.FCXO, 6'b0};
new_FCX = M(EA, word);
M(EA,16 * word) = {PCXI, PSW, A[10], A[11], D[8], D[9], D[10], D[11], A[12], A[13], A[14], A[15], D[12], D[13],
D[14], D[15]};
PCXI.PCPN = ICR.CCPN;
PCXI.PIE = ICR.IE;
PCXI.UL = 1;
PCXI[19:0] = FCX[19:0];
FCX[19:0] = new_FCX[19:0];
PC = {disp24[23:20], 7'b0, disp24[19:0], 1'b0};
A[11] = ret_addr[31:0];
if (tmp_FCX == LCX) trap(FCD);
```

Status Flags

C	Not changed by this instruction but read by the instruction.
V	Not changed by this instruction but read by the instruction.
SV	Not changed by this instruction but read by the instruction.
AV	Not changed by this instruction but read by the instruction.
SAV	Not changed by this instruction but read by the instruction.

Examples

```
calla   foobar
```

See Also

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

CALLI
Call Indirect

Description

Jump to the address specified by the contents of address register A[a]. In parallel with the jump save the caller's Upper Context to an available Context Save Area (CSA). Set register A[11] (return address) to the address of the next instruction beyond the call.

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

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

CALLIA[a] (RR)

31 28	27 20	19 18	17 16	15 12	11 8	7 0
-	00_H	-	-	-	a	$2D_H$

```
if (FCX == 0) trap(FCU);
if (PSW.CDE) then if(cdc_increment()) then trap(CDO);
PSW.CDE = 1;
ret_addr = PC + 4;
tmp_FCX = FCX;
EA = {FCX.FCXS, 6'b0, FCX.FCXO, 6'b0};
new_FCX = M(EA, word);
M(EA,16 * word) = {PCXI, PSW, A[10], A[11], D[8], D[9], D[10], D[11], A[12], A[13], A[14], A[15], D[12], D[13], D[14], D[15]};
PCXI.PCPN = ICR.CCPN;
PCXI.PIE = ICR.IE;
PCXI.UL = 1;
PCXI[19:0] = FCX[19:0];
FCX[19:0] = new_FCX[19:0];
PC = {A[a][31:1], 1'b0};
A[11] = ret_addr[31:0];
if (tmp_FCX == LCX) trap(FCD);
```

Status Flags

C	Not changed by this instruction but read by the instruction.
V	Not changed by this instruction but read by the instruction.
SV	Not changed by this instruction but read by the instruction.
AV	Not changed by this instruction but read by the instruction.
SAV	Not changed by this instruction but read by the instruction.

Examples

```
calli   a2
```

See Also

CALL, CALLA, RET, FCALLI, FRET

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].

CLOD[c], D[a] (RR)

31 28	27 20	19 18	17 16	15 12	11 8	7 0
c	$1C_H$	-	-	-	a	$0F_H$

result = leading_ones(D[a]);

D[c] = zero_ext(result);

Status Flags

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

Examples

```
clo    d3, d1
```

See Also

CLS, **CLZ**, **CLO.H**, **CLS.H**, **CLZ.H**

CLO.H
Count Leading Ones in Packed Half-words

Description

Count the number of consecutive ones in each half-word of D[a], starting with the most significant bit, and put each result in the corresponding half-word of D[c].

CLO.HD[c], D[a] (RR)

31 28	27 20	19 18	17 16	15 12	11 8	7 0
c	$7D_H$	-	-	-	a	$0F_H$

result_halfword1 = zero_ext(leading_ones(D[a][31:16]));

result_halfword0 = zero_ext(leading_ones(D[a][15:0]));

D[c] = {result_halfword1[15:0],result_halfword0[15:0]};

Status Flags

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

Examples

```
clo.h   d3, d1
```

See Also

CLO, **CLS**, **CLS.H**, **CLZ**, **CLZ.H**

CLS
Count Leading Signs

Description

Count the number of consecutive bits which have the same value as bit 31 in D[a], starting with bit 30, and put the result in D[c]. The result is the number of leading sign bits minus one, giving the number of redundant sign bits in D[a].

CLSD[c], D[a] (RR)

31 28	27 20	19 18	17 16	15 12	11 8	7 0
c	$1D_H$	-	-	-	a	$0F_H$

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**

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.HD[c], D[a] (RR)

31 28	27 20	19 18	17 16	15 12	11 8	7 0
c	$7E_H$	-	-	-	a	$0F_H$

result_halfword1 = zero_ext(leading_signs(D[a][31:16]) - 1);

result_halfword0 = zero_ext(leading_signs(D[a][15:0]) - 1);

D[c] = {result_halfword1[15:0],result_halfword0[15:0]};

Status Flags

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

Examples

```
cls.h   d3, d1
```

See Also

CLO, **CLO.H**, **CLS**, **CLZ**, **CLZ.H**

CLZ
Count Leading Zeros

Description

Count the number of consecutive zeros in D[a] starting with bit 31, and put result in D[c].

CLZD[c], D[a] (RR)

31 28	27 20	19 18	17 16	15 12	11 8	7 0
c	$1B_H$	-	-	-	a	$0F_H$

result = leading_zeros(D[a]);

D[c] = zero_ext(result);

Status Flags

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

Examples

```
clz   d3, d1
```

See Also

CLO, **CLO.H**, **CLS**, **CLS.H**, **CLZ.H**

CLZ.H
Count Leading Zeros in Packed Half-words

Description

Count the number of consecutive zeros in each half-word of D[a], starting with the most significant bit of each half-word, and put each result in the corresponding half-word of D[c].

CLZ.HD[c], D[a] (RR)

31 28	27 20	19 18	17 16	15 12	11 8	7 0
c	$7C_H$	-	-	-	a	$0F_H$

result_halfword1 = zero_ext(leading_zeros(D[a][31:16]));

result_halfword0 = zero_ext(leading_zeros(D[a][15:0]));

D[c] = {result_halfword1[15:0],result_halfword0[15:0]};

Status Flags

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

Examples

```
clz.h   d3, d1
```

See Also

CLO, **CLO.H**, **CLS**, **CLS.H**, **CLZ**

CMOV (16-bit)
Conditional Move (16-bit)

Description

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

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

15 12	11 8	7 0
const4	a	AA_H

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

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

15 12	11 8	7 0
b	a	$2A_H$

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

See Also

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

CMOVN (16-bit)
Conditional Move-Not (16-bit)

Description

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

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

15 12	11 8	7 0
const4	a	EA_H

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

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

15 12	11 8	7 0
b	a	$6A_H$

D[a] = ((D[15] == 0) ? D[b] : D[a]);

Status Flags

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

Examples

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

See Also

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

CSUB
Conditional Subtract

Description

If the contents of data register D[d] are not zero, subtract the contents of data register D[b] from the contents of data register D[a] and put the result in data register D[c]; otherwise put the contents of D[a] in D[c].

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

31 28	27 24	23 20	19 18	17 16	15 12	11 8	7 0
c	d	02_H	-	-	b	a	$2B_H$

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 (16-bit)**, **CMOVN (16-bit)**, **CSUBN**, **SEL**, **SELN**

CSUBN
Conditional Subtract-Not

Description

If the contents of data register D[d] are zero, subtract the contents of data register D[b] from the contents of data register D[a] and put the result in data register D[c]; otherwise put the contents of D[a] in D[c].

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

31 28	27 24	23 20	19 18	17 16	15 12	11 8	7 0
c	d	03_H	-	-	b	a	$2B_H$

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 (16-bit)**, **CMOVN (16-bit)**, **CSUB**, **SEL**, **SELN**

DEBUG
Debug

Description

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

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

DEBUG(SR)

15 12	11 8	7 0
$0A_H$	-	00_H

-

DEBUG(SYS)

31 28	27 22	21 12	11 8	7 0
-	04_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

```
debug
```

```
debug
```

See Also

RFM

DEXTR
Extract from Double Register

Description

Extract 32-bits from registers {D[a], D[b]}, where D[a] contains the most-significant 32-bits of the value, starting at the bit number specified by either 32 - D[d][4:0] (instruction format RRRR) or 32 - pos (instruction format RRPW). Put the result in D[c].

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

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

31 28	27 23	22 21	20 16	15 12	11 8	7 0
c	pos	00_H	-	b	a	77_H

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

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

31 28	27 24	23 21	20 16	15 12	11 8	7 0
c	d	04_H	-	b	a	17_H

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

If D[d] > 31 the result is undefined.

Status Flags

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

Examples

```
dextr   d1, d3, d5, d7
dextr   d1, d3, d5, #11
```

See Also

EXTR, **EXTR.U**, **INSERT**, **INS.T**, **INSN.T**

DISABLE
Disable Interrupts

Description

Note: DISABLE can only be executed in User-1 mode or Supervisor mode.

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

DISABLE(SYS)

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

ICR.IE = 0; // disables all interrupts

DISABLED[a] (SYS)

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

D[a][31:1] = 0H;

D[a][0] = ICR.IE;

ICR.IE = 0; // disables all interrupts

Status Flags

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

Examples

```
disable
```

See Also

ENABLE, **BISR**, **RESTORE**

DSYNC
Synchronize Data

Description

Forces all data accesses to complete before any data accesses associated with an instruction, semantically after the DSYNC is initiated.

Note: The Data Cache (DCACHE) is not invalidated by DSYNC.

Note: To ensure memory coherency, a DSYNC instruction must be executed prior to any access to an active CSA memory location.

DSYNC(SYS)

31 28	27 22	21 12	11 8	7 0
-	12_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

```
dsync
```

See Also

ISYNC

DVADJ
Divide-Adjust

Description

Divide-adjust the contents of the formatted data register E[d] using the divisor in D[b] and store the result in E[c]. E[d][63:32] contains the sign-extended final remainder from a previous DVSTEP instruction and E[d][31:0] contains the sign-extended final quotient in ones complement format. The DVADJ instruction converts the final quotient to twos complement format by adding one if the final quotient is negative, and corrects for a corner case that occurs when dividing a negative dividend that is an integer multiple of the divisor. The corner case is resolved by setting the remainder E[d][63:32] to zero and increasing the magnitude of the quotient E[d][31:0] by one. Note that the increment for converting a negative ones complement quotient to twos complement, and the decrement of a negative quotient in the corner case (described above), cancel out.

Note: This operation must not be performed at the end of an unsigned divide sequence.

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

31 28	27 24	23 20	19 18	17 16	15 12	11 8	7 0
c	d	$0D_H$	-	0_H	b	-	$6B_H$

```
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) {
    E[c] = 64'bx;
} 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

–

See Also

-

DIV
Divide
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 signed 32-bit integers for the DIV instruction and as unsigned 32-bit integers for the DIV.U instruction.

Overflow occurs if the divisor (D[b]) is zero. For DIV, Overflow also occurs if the dividend (D[a]) is the maximum negative number and the divisor is minus 1.

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

31 28	27 20	19 18	17 16	15 12	11 8	7 0
c	20_H	-	1_H	b	a	$4B_H$

```
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;
```

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

31 28	27 20	19 18	17 16	15 12	11 8	7 0
c	21_H	-	1_H	b	a	$4B_H$

```
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 these instructions.

V	DIV
	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;
	DIV.U
	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 these instructions.

Examples

–

See Also

DIV.F

DVINIT
Divide-Initialization Word

DVINIT.U
Divide-Initialization Word Unsigned

DVINIT.B
Divide-Initialization Byte

DVINIT.BU
Divide-Initialization Byte Unsigned

DVINIT.H
Divide-Initialization Half-word

DVINIT.HU
Divide-Initialization Half-word Unsigned

Description

The DVINIT group of instructions prepare the operands for a subsequent DVSTEP instruction (see DVSTEP) from the dividend D[a] and divisor D[b], and also check for conditions that will cause overflow of the final quotient result. After a DVINIT instruction E[c] contains the partially calculated remainder (equal to the sign-extended dividend) and partially calculated quotient (equal to + or - zero in ones complement format, depending on the signs of the dividend and divisor).

For signed operands DVINIT, DVINIT.H or DVINIT.B must be used. For unsigned operands DVINIT.U, DVINIT.HU and DVINIT.BU are used.

The size of the remainder and quotient bit fields in E[c] depend upon the variant of DVINIT used, which in turn depends upon the number of subsequent DVSTEP instructions required to calculate the final remainder and quotient results.

If the final quotient result is guaranteed to fit into 8 bits then a DVINIT.B(U) can be used, but must be followed by only one DVSTEP instruction.

If the quotient result is guaranteed to fit into 16 bits then a DVINIT.H(U) can be used but must be followed by two DVSTEP instructions.

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

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

- DVINIT(.U) E[c][63:0] = partially calculated remainder.
- DVINIT.H(U) E[c][63:16] = partially calculated remainder, E[c][15:0] = partially calculated quotient.
- DVINIT.B(.U) E[c][63:24] = partially calculated remainder, E[c][23:0] = partially calculated quotient.

The .B(U) and .H(U) suffixes of the DVINIT group of instructions indicate an 8-bit and 16-bit quotient result, not 8-bit and16-bit operands as in other instructions. The operands supplied to a DVINIT, DVINIT.H or DVINIT.B instruction are required to be 32-bit sign-extended values. The operands supplied to the DVINIT.U, DVINIT.HU and DVINIT.BU instructions are 32-bit zero-extended values.

Overflow occurs if the divisor is zero, or if the dividend is the maximum negative value for the instruction variant and the divisor is minus one. No check is performed to ensure that the expected quotient can be represented in 32, 16, 8 bits, depending on the DVINIT variant used.

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

31 28	27 20	19 18	17 16	15 12	11 8	7 0
c	$5A_H$	-	0_H	b	a	$4B_H$

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;

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

31 28	27 20	19 18	17 16	15 12	11 8	7 0
c	$4A_H$	-	0_H	b	a	$4B_H$

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

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

31 28	27 20	19 18	17 16	15 12	11 8	7 0
c	$3A_H$	-	0_H	b	a	$4B_H$

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;

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

31 28	27 20	19 18	17 16	15 12	11 8	7 0
c	$2A_H$	-	0_H	b	a	$4B_H$

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

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

31 28	27 20	19 18	17 16	15 12	11 8	7 0
c	$1A_H$	-	0_H	b	a	$4B_H$

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

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

31 28	27 20	19 18	17 16	15 12	11 8	7 0
c	$0A_H$	-	0_H	b	a	$4B_H$

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

Status Flags

C	Not set by these instructions.

V	DVINIT
	if ((D[b] == 0) OR ((D[b] == 32’hFFFFFFFF) AND (D[a] == 32’h80000000))) then overflow = 1 else overflow = 0;
	DVINIT.U
	if (D[b] == 0) then overflow = 1 else overflow = 0;
	DVINIT.B
	if ((D[b] == 0) OR ((D[b] == 32’hFFFFFFFF AND (D[a] == 32’hFFFFFF80)) then overflow = 1 else overflow = 0;
	DVINIT.BU
	if (D[b]==0) then overflow = 1 else overflow = 0;
	DVINIT.H
	if ((D[b] == 0) OR ((D[b] == 32’hFFFFFFFF AND (D[a] == 32’hFFFF8000))) then overflow = 1 else overflow=0;
	DVINIT.HU
	if (D[b] == 0) then overflow = 1 else overflow = 0;
	For all the DVINIT variations:
	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 these instructions.

Examples

–

See Also

DVADJ, **DVSTEP**, **DVSTEP.U**

DVSTEP
Divide-Step
DVSTEP.U
Divide-Step Unsigned

Description

The DVSTEP(.U) instruction divides the contents of the formatted data register E[d] by the divisor in D[b], producing 8-bits of quotient at a time. E[d] contains a partially calculated remainder and partially calculated quotient (in ones complement format) in bit fields that depend on the number of DVSTEP instructions required to produce a final result (see DVSTEP).

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

Resultant bit field sizes of E[c]:

- If E[d][63:0] = partially calculated remainder then E[c][63:8] = partially calculated remainder and E[c][7:0] = partially calculated quotient.
- If E[d][63:8] = partially calculated remainder then E[c][63:16] = partially calculated remainder and E[c][15:0] = partially calculated quotient.
- If E[d][63:16] = partially calculated remainder then E[c][63:24] = partially calculated remainder and E[c][23:0] = partially calculated quotient.
- If E[d][63:24] = partially calculated remainder then E[c][63:32] = final remainder and E[c][31:0] = final quotient.

The DVSTEP and DVSTEP.U operate on signed and unsigned operands respectively. A DVSTEP instruction that yields the final remainder and final quotient should be followed by a DVADJ instruction (see DVADJ).

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

31 28	27 24	23 20	19 18	17 16	15 12	11 8	7 0
c	d	$0F_H$	-	0_H	b	-	$6B_H$

```
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]};
```

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

31 28	27 24	23 20	19 18	17 16	15 12	11 8	7 0
c	d	$0E_H$	-	0_H	b	-	$6B_H$

```
divisor = D[b];
dividend_quotient = E[d][31:0];
remainder = E[d][63:32];
for i = 0 to 7 {
    remainder = (remainder << 1) | dividend_quotient[31];
    dividend_quotient <<= 1;
    temp = remainder - divisor;
    remainder = (temp < 0) ? remainder : temp;
    dividend_quotient = dividend_quotient | !(temp < 0);
}
E[c] = {remainder[31:0], dividend_quotient[31:0]};
```

Status Flags

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

Examples

–

See Also

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

ENABLE
Enable Interrupts

Description

Note: ENABLE can only be executed in User-1 or Supervisor mode.

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

ENABLE(SYS)

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

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**

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.

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

31 28	27 21	20 12	11 8	7 0
c	10_H	const9	a	$8B_H$

result = (D[a] == sign_ext(const9));

D[c] = zero_ext(result);

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

31 28	27 20	19 18	17 16	15 12	11 8	7 0
c	10_H	-	-	b	a	$0B_H$

result = (D[a] == D[b]);

D[c] = zero_ext(result);

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

15 12	11 8	7 0
const4	a	BA_H

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

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

15 12	11 8	7 0
b	a	$3A_H$

result = (D[a] == D[b]);
D[15] = zero_ext(result);

Status Flags

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

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

Examples

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

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

See Also

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

EQ.A
Equal to Address

Description

If the contents of address registers A[a] and A[b] are equal, set the least-significant bit of D[c] to one and clear the remaining bits to zero; otherwise clear all bits in D[c].

EQ.AD[c], A[a], A[b] (RR)

31 28	27 20	19 18	17 16	15 12	11 8	7 0
c	40_H	-	-	b	a	01_H

D[c] = (A[a] == A[b]);

Status Flags

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

Examples

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

See Also

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

EQ.B
Equal Packed Byte
EQ.H
Equal Packed Half-word
EQ.W
Equal Packed Word

Description

Compare each byte (EQ.B), half-word (EQ.H) or word (EQ.W) of D[a] with the corresponding byte, half-word or word of D[b].

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

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

31 28	27 20	19 18	17 16	15 12	11 8	7 0
c	50_H	-	-	b	a	$0B_H$

D[c][31:24] = (D[a][31:24] == D[b][31:24]) ? 8’hFF : 8’h00;

D[c][23:16] = (D[a][23:16] == D[b][23:16]) ? 8’hFF : 8’h00;

D[c][15:8] = (D[a][15:8] == D[b][15:8]) ? 8’hFF : 8’h00;

D[c][7:0] = (D[a][7:0] == D[b][7:0]) ? 8’hFF : 8’h00;

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

31 28	27 20	19 18	17 16	15 12	11 8	7 0
c	70_H	-	-	b	a	$0B_H$

D[c][31:16] = (D[a][31:16] == D[b][31:16]) ? 16’hFFFF : 16’h0000;

D[c][15:0] = (D[a][15:0] == D[b][15:0]) ? 16’hFFFF : 16’h0000;

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

31 28	27 20	19 18	17 16	15 12	11 8	7 0
c	90_H	-	-	b	a	$0B_H$

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

Status Flags

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

Examples

```
eq.b   d3, d1, d2
eq.h   d3, d1, d2
eq.w   d3, d1,  d2
```

See Also

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

EQANY.B
Equal Any Byte
EQANY.H
Equal Any Half-word

Description

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

EQANY.BD[c], D[a], const9 (RC)

31 28	27 21	20 12	11 8	7 0
c	56_H	const9	a	$8B_H$

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.BD[c], D[a], D[b] (RR)

31 28	27 20	19 18	17 16	15 12	11 8	7 0
c	56_H	-	-	b	a	$0B_H$

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);

EQANY.HD[c], D[a], const9 (RC)

31 28	27 21	20 12	11 8	7 0
c	76_H	const9	a	$8B_H$

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.HD[c], D[a], D[b] (RR)

31 28	27 20	19 18	17 16	15 12	11 8	7 0
c	76$_H$	-	-	b	a	0B$_H$

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

Examples

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

See Also

EQ, **GE**, **GE.U**, **LT**, **LT.U**, **NE**

EQZ.A
Equal Zero Address

Description

If the contents of address register A[a] are equal to zero, set the least significant bit of D[c] to one and clear the remaining bits to zero; otherwise clear all bits in D[c].

EQZ.AD[c], A[a] (RR)

31 28	27 20	19 18	17 16	15 12	11 8	7 0
c	48_H	-	-	-	a	01_H

D[c] = (A[a] == 0);

Status Flags

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

Examples

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

See Also

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

EXTR
Extract Bit Field
EXTR.U
Extract Bit Field Unsigned

Description

Extract the number of consecutive bits specified by either E[d][36:32] (instruction format RRRR) or width (instruction formats RRRW and RRPW) from D[a], starting at the bit number specified by either E[d][4:0] (instruction format RRRR), D[d][4:0] (instruction format RRRW) or pos (instruction format RRPW). Put the result in D[c], sign-extended (EXTR) or zero-extended (EXTR.U).

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

31 28	27 23	22 21	20 16	15 12	11 8	7 0
c	pos	02_H	width	-	a	37_H

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

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

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

31 28	27 24	23 21	20 16	15 12	11 8	7 0
c	d	02_H	-	-	a	17_H

width = E[d][36:32];

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

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

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

31 28	27 24	23 21	20 16	15 12	11 8	7 0
c	d	02_H	width	-	a	57_H

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

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

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

31 28	27 23	22 21	20 16	15 12	11 8	7 0
c	pos	03_H	width	-	a	37_H

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

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

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

31 28	27 24	23 21	20 16	15 12	11 8	7 0
c	d	03_H	-	-	a	17_H

width = E[d][36:32];

D[c] = zero_ext((D[a] >> E[d][4:0])[width-1:0]);
If E[d][4:0] + width > 32 or if width = 0, then the results are undefined.

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

31 28	27 24	23 21	20 16	15 12	11 8	7 0
c	d	03_H	width	-	a	57_H

D[c] = zero_ext((D[a] >> D[d][4:0])[width-1:0]);
If D[d][4:0] + width > 32 or if width = 0, then the results are undefined.

Status Flags

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

Examples

```
extr      d3, d1, e2
extr      d3, d1, d2, #4
extr      d3, d1, #2, #4
extr.u    d3, d1, e2
extr.u    d3, d1, d2, #4
extr.u    d3, d1, #2, #4
```

See Also

DEXTR, **INSERT**, **INS.T**, **INSN.T**

FCALL
Fast Call

Description

Add the value specified by disp24, multiplied by two and sign-extended to 32-bits, 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].

FCALLdisp24 (B)

31 ... 16	15 ... 8	7 ... 0
disp24[15:0]	disp24[23:16]	61_H

```
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**

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].

FCALLAdisp24 (B)

31 16	15 8	7 0
disp24[15:0]	disp24[23:16]	E1$_H$

```
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**

FCALLI
Fast Call Indirect

Description

Jump to the address specified by the contents of address register A[a]. 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].

FCALLIA[a] (RR)

31 28	27 20	19 18	17 16	15 12	11 8	7 0
-	01_H	-	-	-	a	$2D_H$

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**

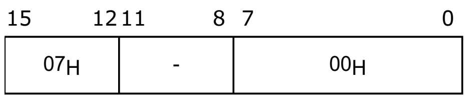
FRET
Return from Fast Call

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.

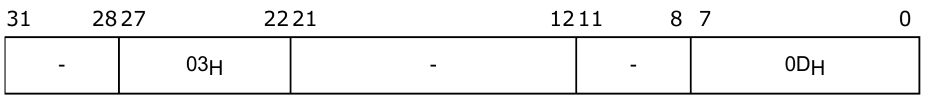
Return from a function that was invoked with an FCALL instruction. Jump to the address specified by A[11]. Load A[11] from the address specified by A[10] then increment A[10] by 4.

FRET (SR)



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

FRET (SYS)



```
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**

GE

Greater Than or Equal

GE.U

Greater Than or Equal Unsigned

Description

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

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

31 28	27 21	20 12	11 8	7 0
c	14_H	const9	a	$8B_H$

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

D[c] = zero_ext(result);

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

31 28	27 20	19 18	17 16	15 12	11 8	7 0
c	14_H	-	-	b	a	$0B_H$

result = (D[a] >= D[b]);

D[c] = zero_ext(result);

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

31 28	27 21	20 12	11 8	7 0
c	15_H	const9	a	$8B_H$

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

D[c] = zero_ext(result);

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

31 28	27 20	19 18	17 16	15 12	11 8	7 0
c	15_H	-	-	b	a	$0B_H$

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

D[c] = zero_ext(result);

Status Flags

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

SAV	Not set by these instructions.

Examples

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

See Also

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

GE.A
Greater Than or Equal Address

Description

If the contents of address register A[a] are greater than or equal to the contents of address register A[b], set the least-significant bit of D[c] to one and clear the remaining bits to zero; otherwise clear all bits in D[c]. Operands are treated as unsigned 32-bit integers.

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

31 28	27 20	19 18	17 16	15 12	11 8	7 0
c	43_H	-	-	b	a	01_H

D[c] = (A[a] >= A[b]); // unsigned

Status Flags

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

Examples

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

See Also

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

IMASK
Insert Mask

Description

Create a mask containing the number of bits specified by width, starting at the bit number specified by either D[d][4:0] (instruction formats RRRW and RCRW) or pos (instruction formats RRPW and RCPW), and put the mask in data register E[c][63:32].

Left-shift the value in either D[b] (formats RRRW and RRPW) or const4 (formats RCRW and RCPW) by the amount specified by either D[d][4:0] (formats RRRW and RCRW) or pos (formats RRPW and RCPW) and put the result value in data register E[c][31:0].

The value const4 is zero-extended. This mask and value can be used by the Load-Modify-Store (LDMST) instruction to write a specified bit field to a location in memory.

IMASKE[c], const4, pos, width (RCPW)

31 28	27 23	22 21	20 16	15 12	11 8	7 0
c	pos	01_H	width	const4	-	$B7_H$

E[c][63:32] = $((2^{width} -1) << pos)$;

E[c][31:0] = (zero_ext(const4) << pos);

If pos + width > 32 the result is undefined.

IMASKE[c], const4, D[d], width (RCRW)

31 28	27 24	23 21	20 16	15 12	11 8	7 0
c	d	01_H	width	const4	-	$D7_H$

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.

IMASKE[c], D[b], pos, width (RRPW)

31 28	27 23	22 21	20 16	15 12	11 8	7 0
c	pos	01_H	width	b	-	37_H

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.

IMASKE[c], D[b], D[d], width (RRRW)

31 28	27 24	23 21	20 16	15 12	11 8	7 0
c	d	01_H	width	b	-	57_H

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, d1, d2, #11
imask           e2, d1, #5, #11
imask           e2, #6, d2, #11
imask           e2, #6, #5, #11
```

See Also

LDMST, **ST.T**

INS.T
Insert Bit
INSN.T
Insert Bit-Not

Description

Move the value of D[a] to D[c] with either:

- For INS.T, bit pos1 of this value replaced with bit pos2 of register D[b].
- For INSN.T, bit pos1 of this value replaced with the inverse of bit pos2 of register D[b].

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

31 28	27 23	22 21	20 16	15 12	11 8	7 0
c	pos2	00_H	pos1	b	a	67_H

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

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

31 28	27 23	22 21	20 16	15 12	11 8	7 0
c	pos2	01_H	pos1	b	a	67_H

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

Status Flags

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

Examples

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

See Also

DEXTR, **EXTR**, **EXTR.U**, **INSERT**

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].

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

31 28	27 23	22 21	20 16	15 12	11 8	7 0
c	pos	00_H	width	const4	a	$B7_H$

mask = (2^{width} -1) << pos;

D[c] = (D[a] & ~mask) | ((zero_ext(const4) << pos) & mask);

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

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

31 28	27 24	23 21	20 16	15 12	11 8	7 0
c	d	00_H	-	const4	a	97_H

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);

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

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

31 28	27 24	23 21	20 16	15 12	11 8	7 0
c	d	00_H	width	const4	a	$D7_H$

mask = (2^{width} -1) << D[d][4:0];

D[c] = (D[a] & ~mask) | ((zero_ext(const4) << D[d][4:0]) & mask);

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

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

31 28	27 23	22 21	20 16	15 12	11 8	7 0
c	pos	00_H	width	b	a	37_H

mask = (2^{width} -1) << pos;

D[c] = (D[a] & ~mask) | ((D[b] << pos) & mask);

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

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

31 28	27 24	23 21	20 16	15 12	11 8	7 0
c	d	00_H	-	b	a	17_H

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);

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

INSERTD[c], D[a], D[b], D[d], width (RRRW)

31 28	27 24	23 21	20 16	15 12	11 8	7 0
c	d	00_H	width	b	a	57_H

mask = (2^{width} -1) << D[d][4:0];

D[c] = (D[a] & ~mask) | ((D[b] << D[d][4:0]) & mask);

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, d2, e4
insert   d3, d1, d2, d4, #8
insert   d3, d1, d2, #16,#8
insert   d3, d1, 0, e4
insert   d3, d1, 0, d4, #8
insert   d3, d1, 0, #16, #8
```

See Also

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

ISYNC
Synchronize Instructions

Description

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

Note: I-cache is not invalidated by ISYNC.

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

ISYNC(SYS)

31 28	27 22	21 12	11 8	7 0
-	13_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

```
isync
```

See Also

DSYNC

IXMAX
Find Maximum Index

IXMAX.U
Find Maximum Index (unsigned)

Description

Enables a search of maximum value and its related index in a vector of 16-bit signed (IXMAX) or unsigned (IXMAX.U) values.

The IXMAX and IXMAX.U instructions are not available in the TriCore 1.2 Architecture.

For all operations:

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

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

31 28	27 24	23 20	19 18	17 16	15 12	11 8	7 0
c	d	$0A_H$	-	0_H	b	-	$6B_H$

```
E[c][15:0] = E[d][15:0] + 2;
E[c][63:48] = $00_H$;
if (D[b][15:0] >= D[b][31:16]) AND (D[b][15:0] > E[d][47:32]) then {
    E[c][47:32] = D[b][15:0];
    E[c][31:16] = E[d][15:0];
} else if (D[b][31:16] > D[b][15:0]) AND (D[b][31:16] > E[d][47:32]) then {
    E[c][47:32] = D[b][31:16];
    E[c][31:16] = E[d][15:0]+1;
} else {
    E[c][47:32] = E[d][47:32];
    E[c][31:16] = E[d][31:16];
}
```

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

31 28	27 24	23 20	19 18	17 16	15 12	11 8	7 0
c	d	$0B_H$	-	0_H	b	-	$6B_H$

For IXMAX.U, the comparison is on unsigned numbers.

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

```
if (D[b][15:0] >= D[b][31:16]) AND (D[b][15:0] > E[d][47:32]) then {
E[c][47:32] = D[b][15:0];
E[c][31:16] = E[d][15:0];
} else if (D[b][31:16] > D[b][15:0]) AND (D[b][31:16] > E[d][47:32]) then {
E[c][47:32] = D[b][31:16];
E[c][31:16] = E[d][15:0]+1;
} else {
E[c][47:32] = E[d][47:32];
E[c][31:16] = E[d][31:16];
}
```

For all index additions, on overflow: wrapping, no trap.

If the 1st compare value and 2nd compare value and current maximum value are the same, the priority select is: current maximum is the highest priority then 1st compare value and 2nd compare value is in the lowest priority.

Status Flags

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

Examples

```
ixmax     e2, e8, d6
ixmax.u   e2, e0, d4
```

See Also

IXMIN

IXMIN
Find Minimum Index
IXMIN.U
Find Minimum Index (unsigned)

Description

Enables search of minimum value and its related index in a vector of 16-bit signed (IXMIN) or unsigned (IXMIN.U) values.

The IXMIN and IXMIN.U instructions are not available in the TriCore 1.2 Architecture.

For all operations:

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

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

31 28	27 24	23 20	19 18	17 16	15 12	11 8	7 0
c	d	08_H	-	0_H	b	-	$6B_H$

```
E[c][15:0] = E[d][15:0] + 2;
E[c][63:48] = $00_H$;
if (D[b][15:0] <= D[b][31:16]) AND (D[b][15:0] < E[d][47:32]) then {
E[c][47:32] = D[b][15:0];
E[c][31:16] = E[d][15:0];
} else if (D[b][31:16] < D[b][15:0]) AND (D[b][31:16] < E[d][47:32]) then {
E[c][47:32] = D[b][31:16];
E[c][31:16] = E[d][15:0]+1;
} else {
E[c][47:32] = E[d][47:32];
E[c][31:16] = E[d][31:16];
}
```

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

31 28	27 24	23 20	19 18	17 16	15 12	11 8	7 0
c	d	09_H	-	0_H	b	-	$6B_H$

For IXMIN.U, the comparison is on unsigned numbers.

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

```
if (D[b][15:0] <= D[b][31:16]) AND (D[b][15:0] < E[d][47:32]) then {
E[c][47:32] = D[b][15:0];
E[c][31:16] = E[d][15:0];
} else if (D[b][31:16] < D[b][15:0]) AND (D[b][31:16] < E[d][47:32]) then {
E[c][47:32] = D[b][31:16];
E[c][31:16] = E[d][15:0]+1;
} else {
E[c][47:32] = E[d][47:32];
E[c][31:16] = E[d][31:16];
}
```

For all index additions, on overflow: wrapping, no trap

If the 1st compare value and 2nd compare value and current minimum value are the same, the priority select is: current minimum is the highest priority then 1st compare value and 2nd compare value is in the lowest priority.

Status Flags

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

Examples

```
ixmin     e10, e2, d0
ixmin.u   e14, e2, d7
```

See Also

IXMAX

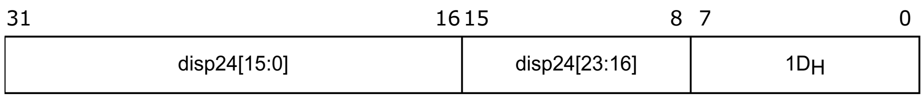
J
Jump Unconditional

Description

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

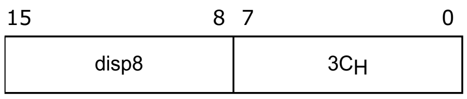
Add the value specified by disp8, sign-extended and multiplied by 2, to the contents of PC and jump to that address.

Jdisp24 (B)



PC = PC + sign_ext(disp24) * 2;

Jdisp8 (SB)



PC = PC + sign_ext(disp8) * 2;

Status Flags

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

Examples

```
j   foobar
```

```
j   foobar
```

See Also

JA, **JI**, **JL**, **JLA**, **JLI**, **LOOPU**

JA
Jump Unconditional Absolute

Description

Load the value specified by disp24 into PC and jump to that address.

The value disp24 is used to form the Effective Address (EA).

JAdisp24 (B)

31 … 16	15 … 8	7 … 0
disp24[15:0]	disp24[23:16]	$9D_H$

PC = {disp24[23:20], 7’b0000000, disp24[19:0], 1’b0};

Status Flags

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

Examples

```
ja          foobar
```

See Also

J, **JI**, **JL**, **JLA**, **JLI**, **LOOPU**

JEQ
Jump if Equal

Description

If the contents of D[a] are equal to the contents of either D[b] or const4, then add the value specified by disp15, sign-extended and multiplied by 2, to the contents of PC and jump to that address. The const4 value is sign-extended.

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

JEQD[a], const4, disp15 (BRC)

31	30 16	15 12	11 8	7 0
00H	disp15	const4	a	DFH

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

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

31	30 16	15 12	11 8	7 0
00H	disp15	b	a	5FH

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

JEQD[15], const4, disp4 (SBC)

15 12	11 8	7 0
const4	disp4	1EH

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

JEQD[15], const4, disp4 (SBC)

15 12	11 8	7 0
const4	disp4	9EH

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

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

15 12	11 8	7 0
b	disp4	3EH

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

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

15 12	11 8	7 0
b	disp4	BE_H

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

Status Flags

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

Examples

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

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

See Also

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

JEQ.A
Jump if Equal Address

Description

If the contents of A[a] are equal to the contents of A[b], then add the value specified by disp15, sign-extended and multiplied by 2, to the contents of PC and jump to that address.

JEQ.AA[a], A[b], disp15 (BRR)

31	30 16	15 12	11 8	7 0
00_H	disp15	b	a	$7D_H$

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

Status Flags

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

Examples

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

See Also

JNE.A

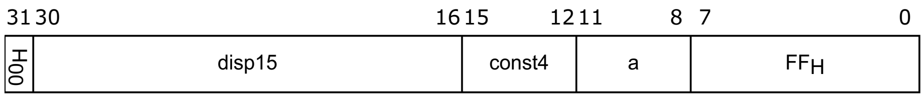
JGE
Jump if Greater Than or Equal
JGE.U
Jump if Greater Than or Equal Unsigned

Description

If the contents of D[a] are greater than or equal to the contents of either D[b] (instruction format BRR) or const4 (instruction format BRC), then add the value specified by disp15, sign-extended and multiplied by 2, to the contents of PC and jump to that address.

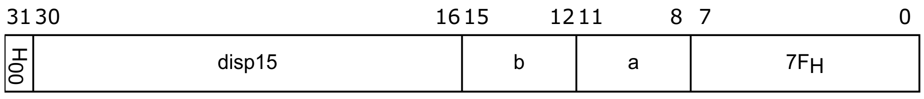
Operands are treated as signed (JGE) or unsigned (JGE.U), 32-bit integers. The const4 value is sign-extended (JGE) or zero-extended (JGE.U).

JGED[a], const4, disp15 (BRC)



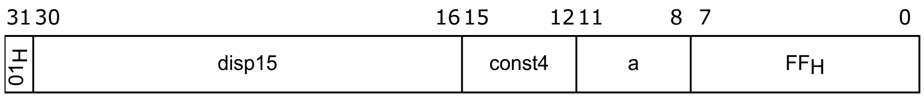
if (D[a] >= sign_ext(const4)) then PC = PC + sign_ext(disp15) * 2;

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



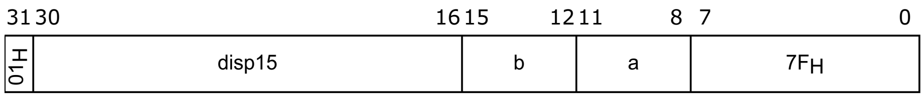
if (D[a] >= D[b]) then PC = PC + sign_ext(disp15) * 2;

JGE.UD[a], const4, disp15 (BRC)



```
if (D[a] >= zero_ext(const4)) then { // unsigned comparison
 PC = PC + sign_ext(disp15) * 2;
}
```

JGE.UD[a], D[b], disp15 (BRR)



if (D[a] >= D[b]) then PC = PC + sign_ext(disp15) * 2; // unsigned comparison

Status Flags

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

Examples

```
jge     d1, d2, foobar
jge     d1, #6, foobar
jge.u   d1, d2, foobar
jge.u   d1, #6, foobar
```

See Also

JEQ, **JLT**, **JLT.U**, **JNE**

JGEZ (16-bit)

Jump if Greater Than or Equal to Zero (16-bit)

Description

If the contents of D[b] are greater than or equal to zero, then add the value specified by disp4, zero-extended and multiplied by 2, to the contents of PC and jump to that address.

JGEZD[b], disp4 (SBR)

15 12	11 8	7 0
b	disp4	CE_H

```
if (D[b] >= 0) then PC = PC + zero_ext(disp4) * 2;
```

Status Flags

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

Examples

```
jgez   d2, foobar
```

See Also

JGTZ (16-bit), **JLEZ (16-bit)**, **JLTZ (16-bit)**, **JNZ (16-bit)**, **JZ (16-bit)**

JGTZ (16-bit)
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, zero-extended and multiplied by 2, to the contents of PC and jump to that address.

JGTZD[b], disp4 (SBR)

15 12	11 8	7 0
b	disp4	$4E_H$

```
if (D[b] > 0) then PC = PC + zero_ext(disp4) * 2;
```

Status Flags

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

Examples

```
jgtz    d2, foobar
```

See Also

JGEZ (16-bit), JLEZ (16-bit), JLTZ (16-bit), JNZ (16-bit), JZ (16-bit)

JI
Jump Indirect

Description

Load the contents of address register A[a] into PC and jump to that address. The least-significant bit is always set to 0.

Load the contents of address register A[a] into PC and jump to that address. The least-significant bit is always set to 0.

JIA[a] (RR)

31 28	27 20	19 18	17 16	15 12	11 8	7 0
-	03_H	-	-	-	a	$2D_H$

PC = {A[a][31:1], 1’b0};

JIA[a] (SR)

15 12	11 8	7 0
00_H	a	DC_H

PC = {A[a][31:1], 1’b0};

Status Flags

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

Examples

```
ji    a2
```

```
ji    a2
```

See Also

J, **JA**, **JL**, **JLA**, **JLI**, **LOOPU**

JL
Jump and Link

Description

Store the address of the next instruction in A[11] (return address). Add the value specified by disp24, sign-extended and multiplied by 2, to the contents of PC and jump to that address.

JLdisp24 (B)

31 … 16	15 … 8	7 … 0
disp24[15:0]	disp24[23:16]	$5D_H$

A[11] = PC + 4;

PC = PC + sign_ext(disp24) * 2;

Status Flags

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

Examples

```
jl    foobar
```

See Also

J, **JI**, **JA**, **JLA**, **JLI**, **CALLA**, **FCALL**, **FCALLA**, **LOOPU**

JLA
Jump and Link Absolute

Description

Store the address of the next instruction in A[11] (return address). Load the value specified by disp24 into PC and jump to that address. The value disp24 is used to form the effective address (EA).

JLAdisp24 (B)

31 16	15 8	7 0
disp24[15:0]	disp24[23:16]	DD_H

A[11] = PC + 4;

PC = {disp24[23:20], 7’b0000000, disp24[19:0], 1’b0};

Status Flags

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

Examples

```
jla   foobar
```

See Also

J, **JI**, **JA**, **JL**, **JLI**, **CALLA**, **FCALL**, **FCALLA**, **LOOPU**

JLEZ (16-bit)

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

JLEZD[b], disp4 (SBR)

15 12	11 8	7 0
b	disp4	$8E_H$

If (D[b] <= 0) then PC = PC + zero_ext(disp4) * 2;

Status Flags

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

Examples

```
jlez   d2, foobar
```

See Also

JGEZ (16-bit), JGTZ (16-bit), JLTZ (16-bit), JNZ (16-bit), JZ (16-bit)

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.

JLIA[a] (RR)

31 28	27 20	19 18	17 16	15 12	11 8	7 0
-	02_H	-	-	-	a	$2D_H$

A[11] = PC + 4;

PC = {A[a][31:1], 1’b0};

Status Flags

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

Examples

```
jli    a2
```

See Also

J, **JI**, **JA**, **JL**, **JLA**, **LOOPU**

JLT
Jump if Less Than
JLT.U
Jump if Less Than Unsigned

Description

If the contents of D[a] are less than the contents of either D[b] (instruction format BRR) or const4 (instruction format BRC), then add the value specified by disp15, sign-extended and multiplied by 2, to the contents of PC and jump to that address. The operands are treated as signed (JLT) or unsigned (JLT.U), 32-bit integers. The const4 value is sign-extended (JLT) or zero-extended (JLT.U).

JLTD[a], const4, disp15 (BRC)

31	30 … 16	15 … 12	11 … 8	7 … 0
00_H	disp15	const4	a	BF_H

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

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

31	30 … 16	15 … 12	11 … 8	7 … 0
00_H	disp15	b	a	$3F_H$

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

JLT.UD[a], const4, disp15 (BRC)

31	30 … 16	15 … 12	11 … 8	7 … 0
01_H	disp15	const4	a	BF_H

```
if (D[a] < zero_ext(const4)) then { // unsigned comparison
PC = PC + sign_ext(disp15) * 2;
}
```

JLT.UD[a], D[b], disp15 (BRR)

31	30 … 16	15 … 12	11 … 8	7 … 0
01_H	disp15	b	a	$3F_H$

if (D[a] < D[b]) then PC = PC + sign_ext(disp15) * 2; // unsigned comparison

Status Flags

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

Examples

```
jlt     d1, d2, foobar
jlt     d1, #6, foobar
jlt.u   d1, d2, foobar
jlt.u   d1, #6, foobar
```

See Also

JEQ, **JGE**, **JGE.U**, **JNE**

JLTZ (16-bit)

Jump if Less Than Zero (16-bit)

Description

If the contents of D[b] are less than zero then add the value specified by disp4, zero-extended and multiplied by 2, to the contents of PC and jump to that address.

JLTZD[b], disp4 (SBR)

15 12	11 8	7 0
b	disp4	$0E_H$

if (D[b] < 0) then PC = PC + zero_ext(disp4) * 2;

Status Flags

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

Examples

```
jltz   d2, foobar
```

See Also

JGEZ (16-bit), JGTZ (16-bit), JLEZ (16-bit), JNZ (16-bit), JZ (16-bit)

JNE
Jump if Not Equal

Description

If the contents of D[a] are not equal to the contents of either D[b] (instruction format BRR) or const4 (instruction format BRC), then add the value specified by disp15, sign-extended and multiplied by 2, to the contents of PC and jump to that address. The const4 value is sign-extended.

If the contents of D[15] are not equal to the contents of either D[b] (instruction format SBR) or const4 (instruction format SBC), then add the value specified by either disp4 or disp4 + 16 , zero-extended and multiplied by 2, to the contents of PC and jump to that address. The const4 value is sign-extended.

JNED[a], const4, disp15 (BRC)

31	30 16	15 12	11 8	7 0
01_H	disp15	const4	a	DF_H

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

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

31	30 16	15 12	11 8	7 0
01_H	disp15	b	a	$5F_H$

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

JNED[15], const4, disp4 (SBC)

15 12	11 8	7 0
const4	disp4	$5E_H$

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

JNED[15], const4, disp4 (SBC)

15 12	11 8	7 0
const4	disp4	DE_H

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

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

15 12	11 8	7 0
b	disp4	$7E_H$

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

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

15 12	11 8	7 0
b	disp4	FE_H

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

Status Flags

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

Examples

```
jne    d1, d2, foobar
jne    d1, #6, foobar
```

```
jne    d15, d2, foobar
jne    d15, #6, foobar
```

See Also

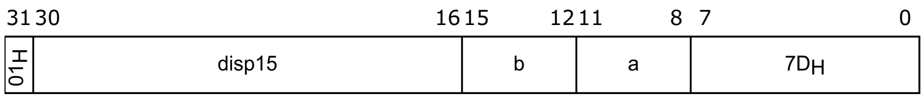
JEQ, **JGE**, **JGE.U**, **JLT**, **JLT.U**

JNE.A
Jump if Not Equal Address

Description

If the contents of A[a] are not equal to the contents of A[b] then add the value specified by disp15, sign-extended and multiplied by 2, to the contents of PC and jump to that address.

JNE.AA[a], A[b], disp15 (BRR)



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

Status Flags

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

Examples

```
jne.a   a4, a2, foobar
```

See Also

JEQ.A

JNED
Jump if Not Equal and Decrement

Description

If the contents of D[a] are not equal to the contents of either D[b] (instruction format BRR) or const4 (instruction format BRC), then add the value specified by disp15, sign-extended and multiplied by 2, to the contents of PC and jump to that address. Decrement the value in D[a] by one. The const4 value is sign-extended.

JNEDD[a], const4, disp15 (BRC)

31	30 16	15 12	11 8	7 0
00_H	disp15	const4	a	$9F_H$

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

D[a] = D[a] - 1;

The decrement is unconditional.

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

31	30 16	15 12	11 8	7 0
01_H	disp15	b	a	$1F_H$

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

D[a] = D[a] - 1;

The decrement is unconditional.

Status Flags

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

Examples

```
jned   d1, d2, foobar
jned   d1, #6, foobar
```

See Also

JNEI, **LOOP**, **LOOPU**

JNEI
Jump if Not Equal and Increment

Description

If the contents of D[a] are not equal to the contents of either D[b] (instruction format BRR) or const4 (instruction format BRC), then add the value specified by disp15, sign-extended and multiplied by 2, to the contents of PC and jump to that address. Increment the value in D[a] by one. The const4 value is sign-extended.

JNEID[a], const4, disp15 (BRC)

31	30 16	15 12	11 8	7 0
00_H	disp15	const4	a	$9F_H$

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

D[a] = D[a] + 1;

The increment is unconditional.

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

31	30 16	15 12	11 8	7 0
00_H	disp15	b	a	$1F_H$

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

D[a] = D[a] + 1;

The increment is unconditional.

Status Flags

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

Examples

```
jnei    d1, d2, foobar
jnei    d1, #6, foobar
```

See Also

JNED, **LOOP**, **LOOPU**

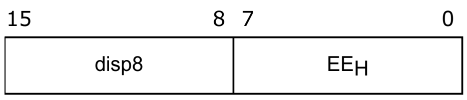
JNZ (16-bit)

Jump if Not Equal to Zero (16-bit)

Description

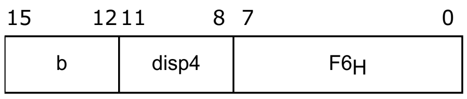
If contents of either D[b] (instruction format SBR) or D[15] (instruction format SB) are not equal to zero, then add value specified by either disp4 (format SBR) or disp8 (format SB), zero-extended (disp4) or sign-extended (disp8) and multiplied by 2, to the contents of PC and jump to that address.

JNZD[15], disp8 (SB)



if (D[15] != 0) then PC = PC + sign_ext(disp8) * 2;

JNZD[b], disp4 (SBR)



if (D[b] != 0) then PC = PC + zero_ext(disp4) * 2;

Status Flags

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

Examples

```
jnz    d2,   foobar
jnz    d15,  foobar
```

See Also

JGEZ (16-bit), JGTZ (16-bit), JLEZ (16-bit), JLTZ (16-bit), JZ (16-bit)

JNZ.A
Jump if Not Equal to Zero Address

Description

If the contents of A[a] are not equal to zero, then add the value specified by disp15, sign-extended and multiplied by 2, to the contents of PC and jump to that address.

If the contents of A[b] are not equal to zero, then add the value specified by disp4, zero-extended and multiplied by 2, to the contents of PC and jump to that address.

JNZ.AA[a], disp15 (BRR)

31	30 … 16	15 … 12	11 … 8	7 … 0
01_H	disp15	-	a	BD_H

if (A[a] != 0) then PC = PC + sign_ext(disp15) * 2;

JNZ.AA[b], disp4 (SBR)

15 … 12	11 … 8	7 … 0
b	disp4	$7C_H$

if (A[b] != 0) then PC = PC + zero_ext(disp4) * 2;

Status Flags

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

Examples

```
jnz.a   a4, foobar
```

```
jnz.a   a4, foobar
```

See Also

JZ.A

JNZ.T
Jump if Not Equal to Zero Bit

Description

If bit n of register D[a] is not equal to zero, then add the value specified by disp15, sign-extended and multiplied by 2, to the contents of PC and jump to that address.

if bit n of register D[15] is not equal to zero, then add the value specified by disp4, zero-extended and multiplied by 2, to the contents of PC and jump to that address.

JNZ.TD[a], n, disp15 (BRN)

31	30 16	15 12	11 8	7	6 0
01H	disp15	n[3:0]	a	n[4]	6FH

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

JNZ.TD[15], n, disp4 (SBRN)

15 12	11 8	7 0
n	disp4	AEH

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

Status Flags

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

Examples

```
jnz.t          d1, 1, foobar
```

```
jnz.t          d15, 1, foobar
```

See Also

JZ.T

JZ (16-bit)
Jump if Zero (16-bit)

Description

If the contents of either D[15] (instruction format SB) or D[b] (instruction format SBR) are equal to zero, then add the value specified by either disp8 (format SB) or disp4 (format SBR), sign-extended (disp8) or zero-extended (disp4) and multiplied by 2, to the contents of PC, and jump to that address.

JZD[15], disp8 (SB)

15 8	7 0
disp8	$6E_H$

if (D[15] == 0) then PC = PC + sign_ext(disp8) * 2;

JZD[b], disp4 (SBR)

15 12	11 8	7 0
b	disp4	76_H

if (D[b] == 0) then PC = PC + zero_ext(disp4) * 2;

Status Flags

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

Examples

```
jz   d2, foobar
jz   d15, foobar
```

See Also

JGEZ (16-bit), JGTZ (16-bit), JLEZ (16-bit), JLTZ (16-bit), JNZ (16-bit)

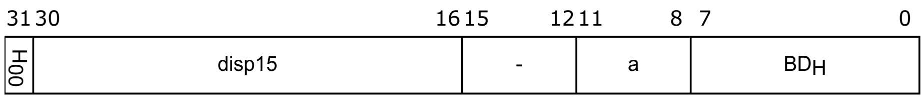
JZ.A
Jump if Zero Address

Description

If the contents of A[a] are equal to zero then add the value specified by disp15, sign-extended and multiplied by 2, to the contents of PC and jump to that address.

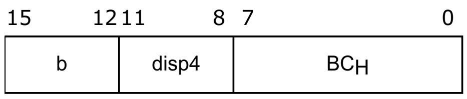
If the contents of A[b] are equal to zero then add the value specified by disp4, zero-extended and multiplied by 2, to the contents of PC and jump to that address.

JZ.AA[a], disp15 (BRR)



if (A[a] == 0) then PC = PC + sign_ext(disp15) * 2;

JZ.AA[b], disp4 (SBR)



if (A[b] == 0) then PC = PC + zero_ext(disp4) * 2;

Status Flags

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

Examples

```
jz.a    a4, foobar
```

```
jz.a    a2, foobar
```

See Also

JNZ.A

JZ.T
Jump if Zero Bit

Description

If bit n of register D[a] is equal to zero then add the value specified by disp15, sign-extended and multiplied by 2, to the contents of PC and jump to that address.

If bit n of register D[15] is equal to zero then add the value specified by disp4, zero-extended and multiplied by 2, to the contents of PC and jump to that address.

JZ.TD[a], n, disp15 (BRN)

31	30 16	15 12	11 8	7	6 0
00_H	disp15	n[3:0]	a	n[4]	$6F_H$

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

JZ.TD[15], n, disp4 (SBRN)

15 12	11 8	7 0
n	disp4	$2E_H$

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

Status Flags

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

Examples

```
jz.t   d1, 1, foobar
```

```
jz.t   d15, 1, foobar
```

See Also

JNZ.T

LD.A
Load Word to Address Register

Description

Load the word contents of the memory location specified by the addressing mode into address register A[a].

Note: If the target register is modified by the addressing mode, the result is undefined.

Load the word contents of the memory location specified by the addressing mode into either address register A[a] or A[15].

Note: If the target register is modified by the addressing mode, the result is undefined.

LD.AA[a], off18 (ABS)(Absolute Addressing Mode)

31 28	27 26	25 22	21 16	15 12	11 8	7 0
off18[9:6]	02_H	off18[13:10]	off18[5:0]	off18[17:14]	a	85_H

EA = {off18[17:14], 14b'0, off18[13:0]};

A[a] = M(EA, word);

LD.AA[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]	26_H	off10[5:0]	b	a	09_H

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

A[a] = M(EA, word);

LD.AA[a], P[b] (BO)(Bit-reverse Addressing Mode)

31 28	27 22	21 16	15 12	11 8	7 0
-	06_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;

A[a] = M(EA, word);

new_index = reverse16(reverse16(index) + reverse16(incr));

A[b+1] = {incr[15:0], new_index[15:0]};

LD.AA[a], P[b], off10 (BO)(Circular Addressing Mode)

31 28	27 22	21 16	15 12	11 8	7 0
off10[9:6]	16_H	off10[5:0]	b	a	29_H

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.AA[a], A[b], off10 (BO)(Post-increment Addressing Mode)

31 28	27 22	21 16	15 12	11 8	7 0
off10[9:6]	06_H	off10[5:0]	b	a	09_H

EA = A[b];
A[a] = M(EA, word);
A[b] = EA + sign_ext(off10);

LD.AA[a], A[b], off10 (BO)(Pre-increment Addressing Mode)

31 28	27 22	21 16	15 12	11 8	7 0
off10[9:6]	16_H	off10[5:0]	b	a	09_H

EA = A[b] + sign_ext(off10);
A[a] = M(EA, word);
A[b] = EA;

LD.AA[a], A[b], off16 (BOL)(Base + Long Offset Addressing Mode)

31 28	27 22	21 16	15 12	11 8	7 0
off16[9:6]	off16[15:10]	off16[5:0]	b	a	99_H

EA = A[b] + sign_ext(off16);
A[a] = M(EA, word);

LD.AA[15], A[10], const8 (SC)

15 8	7 0
const8	$D8_H$

A[15] = M(A[10] + zero_ext(4 * const8), word);

LD.AA[c], A[b] (SLR)

15 12	11 8	7 0
b	c	$D4_H$

A[c] = M(A[b], word);

LD.AA[c], A[b] (SLR)(Post-increment Addressing Mode)

15 12	11 8	7 0
b	c	$C4_H$

A[c] = M(A[b], word);
A[b] = A[b] + 4;

LD.AA[c], A[15], off4 (SLRO)

15 12	11 8	7 0
off4	c	C8$_H$

A[c] = M(A[15] + zero_ext(4 * off4), word);

LD.AA[15], A[b], off4 (SRO)

15 12	11 8	7 0
b	off4	CC$_H$

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    a0, [a0]
ld.a    a5, [a0+]4
```

See Also

LD.B, **LD.BU**, **LD.D**, **LD.DA**, **LD.H**, **LD.HU**, **LD.Q**, **LD.W**

LD.B
Load Byte
LD.BU
Load Byte Unsigned

Description

Load the byte contents of the memory location specified by the addressing mode, sign-extended or zero-extended, into data register D[a].

Load the byte contents of the memory location specified by the addressing mode, zero-extended, into either data register D[a] or D[15].

LD.BD[a], off18 (ABS)(Absolute Addressing Mode)

31 28	27 26	25 22	21 16	15 12	11 8	7 0
off18[9:6]	00_H	off18[13:10]	off18[5:0]	off18[17:14]	a	05_H

EA = {off18[17:14], 14b'0, off18[13:0]};

D[a] = sign_ext(M(EA, byte));

LD.BD[a], A[b], off10 (BO)(Base + Short Offset Addressing Mode)

31 28	27 22	21 16	15 12	11 8	7 0
off10[9:6]	20_H	off10[5:0]	b	a	09_H

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

D[a] = sign_ext(M(EA, byte));

LD.BD[a], P[b] (BO)(Bit-reverse Addressing Mode)

31 28	27 22	21 16	15 12	11 8	7 0
-	00_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] = sign_ext(M(EA, byte));

new_index = reverse16(reverse16(index) + reverse16(incr));

A[b+1] = {incr[15:0], new_index[15:0]};

LD.BD[a], P[b], off10 (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	29_H

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.BD[a], A[b], off10 (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	09_H

EA = A[b];
D[a] = sign_ext(M(EA, byte));
A[b] = EA + sign_ext(off10);

LD.BD[a], A[b], off10 (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	09_H

EA = A[b] + sign_ext(off10);
D[a] = sign_ext(M(EA, byte));
A[b] = EA;

LD.BD[a], A[b], off16 (BOL)(Base + Long Offset Addressing Mode)

31 28	27 22	21 16	15 12	11 8	7 0
off16[9:6]	off16[15:10]	off16[5:0]	b	a	79_H

EA = A[b] + sign_ext(off16);
D[a] = sign_ext(M(EA, byte));

LD.BUD[a], off18 (ABS)(Absolute Addressing Mode)

31 28	27 26	25 22	21 16	15 12	11 8	7 0
off18[9:6]	01_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, byte));

LD.BUD[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]	21_H	off10[5:0]	b	a	09_H

EA = A[b] + sign_ext(off10);
D[a] = zero_ext(M(EA, byte));

LD.BUD[a], P[b] (BO)(Bit-reverse Addressing Mode)

31 28	27 22	21 16	15 12	11 8	7 0
-	01_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, byte));
new_index = reverse16(reverse16(index) + reverse16(incr));
A[b+1] = {incr[15:0], new_index[15:0]};
```

LD.BUD[a], P[b], off10 (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	29_H

```
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.BUD[a], A[b], off10 (BO)(Post-increment Addressing Mode)

31 28	27 22	21 16	15 12	11 8	7 0
off10[9:6]	01_H	off10[5:0]	b	a	09_H

```
EA = A[b];
D[a] = zero_ext(M(EA, byte));
A[b] = EA+sign_ext(off10);
```

LD.BUD[a], A[b], off10 (BO)(Pre-increment Addressing Mode)

31 28	27 22	21 16	15 12	11 8	7 0
off10[9:6]	11_H	off10[5:0]	b	a	09_H

```
EA = A[b] + sign_ext(off10);
D[a] = zero_ext(M(EA, byte));
A[b] = EA;
```

LD.BUD[a], A[b], off16 (BOL)(Base + Long Offset Addressing Mode)

31 28	27 22	21 16	15 12	11 8	7 0
off16[9:6]	off16[15:10]	off16[5:0]	b	a	39_H

```
EA = A[b] + sign_ext(off16);
D[a] = zero_ext(M(EA, byte));
```

LD.BUD[c], A[b] (SLR)

15 12	11 8	7 0
b	c	14_H

D[c] = zero_ext(M(A[b], byte));

LD.BUD[c], A[b] (SLR)(Post-increment Addressing Mode)

15 12	11 8	7 0
b	c	04_H

D[c] = zero_ext(M(A[b], byte));
A[b] = A[b] + 1;

LD.BUD[c], A[15], off4 (SLRO)

15 12	11 8	7 0
off4	c	08_H

D[c] = zero_ext(M(A[15] + zero_ext(off4), byte));

LD.BUD[15], A[b], off4 (SRO)

15 12	11 8	7 0
b	off4	$0C_H$

D[15] = zero_ext(M(A[b] + zero_ext(off4), byte));

Status Flags

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

Examples

```
ld.b    d0, [a0]
ld.bu    d5, [a0+]4
```

See Also

LD.A, LD.D, LD.DA, LD.H, LD.HU, LD.Q, LD.W

LD.D
Load Double-word

Description

Load the double-word contents of the memory location specified by the addressing mode into the extended data register E[a]. The least-significant word of the double-word value is loaded into the even register (D[n]) and the most-significant word is loaded into the odd register (D[n+1]).

LD.DE[a], off18 (ABS)(Absolute Addressing Mode)

31 28	27 26	25 22	21 16	15 12	11 8	7 0
off18[9:6]	01_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.DE[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.DE[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.DE[a], P[b], off10 (BO)(Circular Addressing Mode)

31 28	27 22	21 16	15 12	11 8	7 0
off10[9:6]	15_H	off10[5:0]	b	a	29_H

index = zero_ext(A[b+1][15:0]);

length = zero_ext(A[b+1][31:16]);

EA0 = A[b] + index;

EA2 = A[b] + (index + 2) % length;

EA4 = A[b] + (index + 4) % length;

EA6 = A[b] + (index + 6) % length;

EA = {M(EA6, halfword), M(EA4, halfword), M(EA2, halfword), M(EA0, halfword)};

new_index = index + sign_ext(off10);

new_index = new_index < 0 ? new_index+length : new_index % length;

A[b+1] = {length[15:0], new_index[15:0]};

LD.DE[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.DE[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;

Status Flags

C	Not set by this instruction.
V	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, [a0]
ld.d    d0/d1, [a0]
ld.d    e4, [a10+]4
```

See Also

LD.A, **LD.B**, **LD.BU**, **LD.DA**, **LD.H**, **LD.HU**, **LD.Q**, **LD.W**

LD.DA
Load Double-word to Address Register

Description

Load the double-word contents of the memory location specified by the addressing mode into an address register pair A[a]. The least-significant word of the double-word value is loaded into the even register (A[a]) and the most-significant word is loaded into the odd register (A[a+1]).

Note: If the target register is modified by the addressing mode, the result is undefined.

LD.DAP[a], off18 (ABS)(Absolute Addressing Mode)

31 28	27 26	25 22	21 16	15 12	11 8	7 0
off18[9:6]	03_H	off18[13:10]	off18[5:0]	off18[17:14]	a	85_H

```
EA = {off18[17:14], 14b'0, off18[13:0]};
P[a] = M(EA, doubleword);
```

LD.DA P[a], A[b], off10 (BO)(Base + Short Offset Addressing Mode)

31 28	27 22	21 16	15 12	11 8	7 0
off10[9:6]	27_H	off10[5:0]	b	a	09_H

```
EA = A[b] + sign_ext(off10);
P[a] = M(EA, doubleword);
```

LD.DAP[a], P[b] (BO)(Bit-reverse Addressing Mode)

31 28	27 22	21 16	15 12	11 8	7 0
-	07_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;
P[a] = M(EA, doubleword);
new_index = reverse16(reverse16(index) + reverse16(incr));
A[b+1] = {incr[15:0], new_index[15:0]};
```

LD.DAP[a], P[b], off10 (BO)(Circular Addressing Mode)

31 28	27 22	21 16	15 12	11 8	7 0
off10[9:6]	17_H	off10[5:0]	b	a	29_H

```
index = zero_ext(A[b+1][15:0]);
length = zero_ext(A[b+1][31:16]);
EA0 = A[b] + index;
EA4 = A[b] + (index + 4) % length;
P[a] = {M(EA4, word), M(EA0, word)};
new_index = index + sign_ext(off10);
new_index = new_index < 0 ? new_index + length : new_index % length;
```

A[b+1] = {length[15:0], new_index[15:0]};

LD.DA P[a], A[b], off10 (BO)(Post-increment Addressing Mode)

31 28	27 22	21 16	15 12	11 8	7 0
off10[9:6]	07_H	off10[5:0]	b	a	09_H

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)

31 28	27 22	21 16	15 12	11 8	7 0
off10[9:6]	17_H	off10[5:0]	b	a	09_H

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

P[a] = M(EA, doubleword);

A[b] = EA;

Status Flags

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

Examples

```
ld.da   a4/a5, [a6]+8
ld.da   a0/a1, _savedPointerBuffer
```

See Also

LD.A, **LD.B**, **LD.BU**, **LD.D**, **LD.H**, **LD.HU**, **LD.Q**, **LD.W**

LD.H
Load Half-word
LD.HU
Load Half-word Unsigned

Description

Load the half-word contents of the memory location specified by the addressing mode, sign-extended, into data register D[a].

Load the half-word contents of the memory location specified by the addressing mode, sign-extended, into either data register D[a] or D[15].

LD.HD[a], off18 (ABS)(Absolute Addressing Mode)

31 28	27 26	25 22	21 16	15 12	11 8	7 0
off18[9:6]	02_H	off18[13:10]	off18[5:0]	off18[17:14]	a	05_H

EA = {off18[17:14], 14b'0, off18[13:0]};

D[a] = sign_ext(M(EA, halfword));

LD.HD[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]	22_H	off10[5:0]	b	a	09_H

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

D[a] = sign_ext(M(EA, halfword));

LD.HD[a], P[b] (BO)(Bit-reverse Addressing Mode)

31 28	27 22	21 16	15 12	11 8	7 0
-	02_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] = sign_ext(M(EA, halfword));

new_index = reverse16(reverse16(index) + reverse16(incr));

A[b+1] = {incr[15:0], new_index[15:0]};

LD.HD[a], P[b], off10 (BO)(Circular Addressing Mode)

31 28	27 22	21 16	15 12	11 8	7 0
off10[9:6]	12_H	off10[5:0]	b	a	29_H

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.HD[a], A[b], off10 (BO)(Post-increment Addressing Mode)

31 28	27 22	21 16	15 12	11 8	7 0
off10[9:6]	02_H	off10[5:0]	b	a	09_H

EA = A[b];

D[a] = sign_ext(M(EA, halfword));

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

LD.HD[a], A[b], off10 (BO)(Pre-increment Addressing Mode)

31 28	27 22	21 16	15 12	11 8	7 0
off10[9:6]	12_H	off10[5:0]	b	a	09_H

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

D[a] = sign_ext(M(EA, halfword));

A[b] = EA;

LD.HD[a], A[b], off16 (BOL)(Base + Long Offset Addressing Mode)

31 28	27 22	21 16	15 12	11 8	7 0
off16[9:6]	off16[15:10]	off16[5:0]	b	a	$C9_H$

EA = A[b] + sign_ext(off16);

D[a] = sign_ext(M(EA, halfword));

LD.HD[c], A[b] (SLR)

15 12	11 8	7 0
b	c	94_H

D[c] = sign_ext(M(A[b], halfword));

LD.HD[c], A[b] (SLR)(Post-increment Addressing Mode)

15 12	11 8	7 0
b	c	84_H

D[c] = sign_ext(M(A[b], half-word));
A[b] = A[b] + 2;

LD.HD[c], A[15], off4 (SLRO)

15 12	11 8	7 0
off4	c	88_H

D[c] = sign_ext(M(A[15] + zero_ext(2 * off4), half-word));

LD.HD[15], A[b], off4 (SRO)

15 12	11 8	7 0
b	off4	$8C_H$

D[15] = sign_ext(M(A[b] + zero_ext(2 * off4), half-word));

LD.HUD[a], off18 (ABS)(Absolute Addressing Mode)

31 28	27 26	25 22	21 16	15 12	11 8	7 0
off18[9:6]	03_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.HUD[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.HUD[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.HUD[a], P[b], off10 (BO)(Circular Addressing Mode)

31 28	27 22	21 16	15 12	11 8	7 0
off10[9:6]	13_H	off10[5:0]	b	a	29_H

index = zero_ext(A[b+1][15:0]);

length = zero_ext(A[b+1][31:16]);

EA0 = A[b] + index;

D[a] = zero_ext(EA, halfword);

new_index = index + sign_ext(off10);

new_index = new_index < 0 ? new_index + length : new_index % length;

A[b+1] = {length[15:0], new_index[15:0]};

LD.HUD[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.HUD[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.HUD[a], A[b], off16 (BOL)(Base + Long Offset Addressing Mode)

31 28	27 22	21 16	15 12	11 8	7 0
off16[9:6]	off16[15:10]	off16[5:0]	b	a	$B9_H$

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.h   d0, [a0]
ld.hu   d1, [a0]
```

See Also

LD.A, LD.B, LD.BU, LD.D, LD.DA, LD.Q, LD.W

LD.Q
Load Half-word Signed Fraction

Description

Load the half-word contents of the memory location specified by the addressing mode into the most-significant half-word of data register D[a], setting the 16 least-significant bits of D[a] to zero.

LD.QD[a], off18 (ABS)(Absolute Addressing Mode)

31 28	27 26	25 22	21 16	15 12	11 8	7 0
off18[9:6]	00_H	off18[13:10]	off18[5:0]	off18[17:14]	a	45_H

```
EA = {off18[17:14],14b'0,off18[13:0]};
D[a] = {M(EA, halfword), 16'h0000};
```

LD.QD[a], A[b], off10 (BO)(Base + Short Offset Addressing Mode)

31 28	27 22	21 16	15 12	11 8	7 0
off10[9:6]	28_H	off10[5:0]	b	a	09_H

```
EA = A[b] + sign_ext(off10);
D[a] = {M(EA, halfword), 16'h0000};
```

LD.QD[a], P[b] (BO)(Bit-reverse Addressing Mode)

31 28	27 22	21 16	15 12	11 8	7 0
-	08_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] = {M(EA, halfword), 16'h0000};
new_index = reverse16(reverse16(index) + reverse16(incr));
A[b+1] = {incr[15:0], new_index[15:0]};
```

LD.QD[a], P[b], off10 (BO)(Circular Addressing Mode)

31 28	27 22	21 16	15 12	11 8	7 0
off10[9:6]	18_H	off10[5:0]	b	a	29_H

```
index = zero_ext(A[b+1][15:0]);
length = zero_ext(A[b+1][31:16]);
EA0 = A[b] + index;
D[a] = {M(EA, halfword), 16'h0000};
new_index = index + sign_ext(off10);
new_index = new_index < 0 ? new_index + length : new_index % length;
A[b+1] = {length[15:0], new_index[15:0]};
```

LD.QD[a], A[b], off10 (BO)(Post-increment Addressing Mode)

31 28	27 22	21 16	15 12	11 8	7 0
off10[9:6]	08_H	off10[5:0]	b	a	09_H

EA = A[b];

D[a] = {M(EA, halfword), 16'h0000};

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

LD.QD[a], A[b], off10 (BO)(Pre-increment Addressing Mode)

31 28	27 22	21 16	15 12	11 8	7 0
off10[9:6]	18_H	off10[5:0]	b	a	09_H

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

D[a] = {M(EA, halfword), 16'h0000};

A[b] = EA;

Status Flags

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

Examples

```
ld.q    d4, [a0+]2
ld.q    d2, [a2+]22
```

See Also

LD.A, **LD.B**, **LD.BU**, **LD.D**, **LD.DA**, **LD.H**, **LD.HU**, **LD.W**

LD.W
Load Word

Description

Load word contents of the memory location specified by the addressing mode into data register D[a].

Load word contents of the memory location specified by the addressing mode into data register either D[a] or D[15].

LD.WD[a], off18 (ABS)(Absolute Addressing Mode)

31 28	27 26	25 22	21 16	15 12	11 8	7 0
off18[9:6]	00_H	off18[13:10]	off18[5:0]	off18[17:14]	a	85_H

EA = {off18[17:14], 14b'0, off18[13:0]};

D[a] = M(EA, word);

LD.WD[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]	24_H	off10[5:0]	b	a	09_H

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

D[a] = M(EA, word);

LD.WD[a], P[b] (BO)(Bit-reverse Addressing Mode)

31 28	27 22	21 16	15 12	11 8	7 0
-	04_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] = M(EA, word);

new_index = reverse16(reverse16(index) + reverse16(incr));

A[b+1] = {incr[15:0], new_index[15:0]};

LD.WD[a], P[b], off10 (BO)(Circular Addressing Mode)

31 28	27 22	21 16	15 12	11 8	7 0
off10[9:6]	14_H	off10[5:0]	b	a	29_H

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.WD[a], A[b], off10 (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	09_H

EA = A[b];

D[a] = M(EA, word);

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

LD.WD[a], A[b], off10 (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	09_H

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

D[a] = M(EA, word);

A[b] = EA;

LD.WD[a], A[b], off16 (BOL)(Base + Long Offset Addressing Mode)

31 28	27 22	21 16	15 12	11 8	7 0
off16[9:6]	off16[15:10]	off16[5:0]	b	a	19_H

EA = A[b] + sign_ext(off16);

D[a] = M(EA, word);

LD.WD[15], A[10], const8 (SC)

15 8	7 0
const8	58_H

D[15] = M(A[10] + zero_ext(4 * const8), word);

LD.WD[c], A[b] (SLR)

15 12	11 8	7 0
b	c	54_H

D[c] = M(A[b], word);

LD.WD[c], A[b] (SLR)(Post-increment Addressing Mode)

15 12	11 8	7 0
b	c	44_H

D[c] = M(A[b], word);
A[b] = A[b] + 4;

LD.WD[c], A[15], off4 (SLRO)

15 12	11 8	7 0
off4	c	48_H

D[c] = M(A[15] + zero_ext(4 * off4), word);

LD.WD[15], A[b], off4 (SRO)

15 12	11 8	7 0
b	off4	$4C_H$

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    d4, [a0+]2
ld.w    d2, [a2+]22
```

See Also

LD.A, **LD.B**, **LD.BU**, **LD.D**, **LD.DA**, **LD.H**, **LD.HU**, **LD.Q**

LDLCX
Load Lower Context

Description

Load the contents of the memory block specified by the addressing mode into registers A[2]-A[7] and D[0]-D[7]. This operation is normally used to restore GPR values that were saved previously by an STLCX instruction.

Note: The effective address specified by the addressing mode must be aligned on a 16-word boundary. For this instruction the addressing mode is restricted to absolute (ABS) or base plus short offset (BO).

LDLCXoff18 (ABS)(Absolute Addressing Mode)

31 28	27 26	25 22	21 16	15 12	11 8	7 0
off18[9:6]	02_H	off18[13:10]	off18[5:0]	off18[17:14]	-	15_H

EA = {off18[17:14],14b'0,off18[13:0]};

{dummy, dummy, A[2:3], D[0:3], A[4:7], D[4:7]} = M(EA, 16-word);

LDLCXA[b], off10 (BO) (Base + Short Index Addressing Mode)

31 28	27 22	21 16	15 12	11 8	7 0
off10[9:6]	24_H	off10[5:0]	b	-	49_H

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

{dummy, dummy, A[2:3], D[0:3], A[4:7], D[4:7]} = M(EA, 16-word);

Status Flags

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

Examples

–

See Also

LDUCX, **RSLCX**, **STLCX**, **STUCX**, **SVLCX**, **BISR**

LDMST
Load-Modify-Store

Description

The atomic Load-Modify-Store implements a store under a mask of a value to the memory word, whose address is specified by the addressing mode. Only those bits of the value E[a][31:0] where the corresponding bits in the mask E[a][63:32] are set, are stored into memory. The value and mask may be generated using the **IMASK** instruction.

LDMSToff18, E[a] (ABS)(Absolute Addressing Mode)

31 28	27 26	25 22	21 16	15 12	11 8	7 0
off18[9:6]	01_H	off18[13:10]	off18[5:0]	off18[17:14]	a	$E5_H$

EA = {off18[17:14], 14b'0,off18[13:0]};

M(EA, word) = (M(EA, word) & ~E[a][63:32]) | (E[a][31:0] & E[a][63:32]);

LDMSTA[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]	21_H	off10[5:0]	b	a	49_H

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

M(EA, word) = (M(EA, word) & ~E[a][63:32]) | (E[a][31:0] & E[a][63:32]);

LDMSTP[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;
M(EA, word) = (M(EA, word) & ~E[a][63:32]) | (E[a][31:0] & E[a][63:32]);
new_index = reverse16(reverse16(index) + reverse16(incr));
A[b+1] = {incr[15:0], new_index[15:0]};
```

LDMSTP[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;
M(EA, word) = (M(EA, word) & ~E[a][63:32]) | (E[a][31:0] & E[a][63:32]);
new_index = index + sign_ext(off10);
new_index = new_index < 0 ? new_index + length : new_index % length;
A[b+1] = {length[15:0], new_index[15:0]};
```

LDMST A[b], off10, E[a] (BO)(Post-increment Addressing Mode)

31 28	27 22	21 16	15 12	11 8	7 0
off10[9:6]	01_H	off10[5:0]	b	a	49_H

```
EA = A[b];
M(EA, word) = (M(EA, word) & ~E[a][63:32]) | (E[a][31:0] & E[a][63:32]);
A[b] = EA + sign_ext(off10);
```

LDMST A[b], off10, E[a] (BO)(Pre-increment Addressing Mode)

31 28	27 22	21 16	15 12	11 8	7 0
off10[9:6]	11_H	off10[5:0]	b	a	49_H

```
EA = A[b] + sign_ext(off10);
M(EA, word) = (M(EA, word) & ~E[a][63:32]) | (E[a][31:0] & E[a][63:32]);
A[b] = EA;
```

Status Flags

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

Examples

–

See Also

IMASK, **ST.T**, **SWAP.W**

LDUCX
Load Upper Context

Description

Load the contents of the memory block specified by the addressing mode into registers A[10] to A[15] and D[8] to D[15]. This operation is used normally to restore GPR values that were saved previously by a **STUCX** instruction.

Note: The effective address (EA) specified by the addressing mode must be aligned on a 16-word boundary. For this instruction the addressing mode is restricted to absolute (ABS) or base plus short offset (BO).

LDUCXoff18 (ABS)(Absolute Addressing Mode)

31 28	27 26	25 22	21 16	15 12	11 8	7 0
off18[9:6]	03_H	off18[13:10]	off18[5:0]	off18[17:14]	-	15_H

EA = {off18[17:14], 14b'0, off18[13:0]};

{dummy, dummy, A[10:11], D[8:11], A[12:15], D[12:15]} = M(EA, 16-word);

LDUCXA[b], off10 (BO)(Base + Short Index Addressing Mode)

31 28	27 22	21 16	15 12	11 8	7 0
off10[9:6]	25_H	off10[5:0]	b	-	49_H

EA = A[b][31:0] + sign_ext(off10);

{dummy, dummy, A[10:11], D[8:11], A[12:15], D[12:15]} = M(EA, 16-word);

Status Flags

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

Examples

–

See Also

LDLCX, **RSLCX**, **STLCX**, **STUCX**, **SVLCX**, **LDUCX**

LEA
Load Effective Address

Description

Compute the absolute (effective) address defined by the addressing mode and put the result in address register A[a].

Note: The auto-increment addressing modes are not supported for this instruction.

LEAA[a], off18 (ABS)(Absolute Addressing Mode)

31 28	27 26	25 22	21 16	15 12	11 8	7 0
off18[9:6]	00_H	off18[13:10]	off18[5:0]	off18[17:14]	a	$C5_H$

EA = {off18[17:14], 14b'0, off18[13:0]};

A[a] = EA[31:0];

LEAA[a], A[b], off10 (BO)(Base + Short Offset Addressing Mode)

31 28	27 22	21 16	15 12	11 8	7 0
off10[9:6]	28_H	off10[5:0]	b	a	49_H

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

A[a] = EA[31:0];

LEAA[a], A[b], off16 (BOL)(Base + Long Offset Addressing Mode)

31 28	27 22	21 16	15 12	11 8	7 0
off16[9:6]	off16[15:10]	off16[5:0]	b	a	$D9_H$

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**

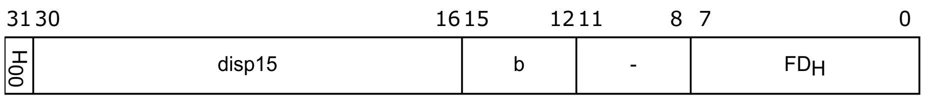
LOOP
Loop

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.

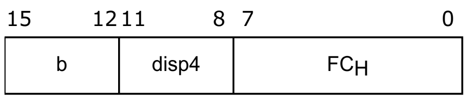
LOOPA[b], disp15 (BRR)



if (A[b] != 0) then PC = PC + sign_ext(2 * disp15);

A[b] = A[b] - 1;

LOOPA[b], disp4 (SBR)



if (A[b] != 0) then PC = PC + {27b'111111111111111111111111111, 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

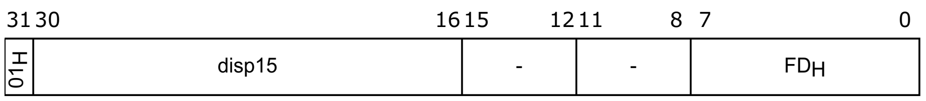
JNED, JNEI, LOOPU

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.

LOOPUdisp15 (BRR)



PC = PC + sign_ext(2 * disp15);

Status Flags

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

Examples

```
loopu   iloop
```

See Also

J, **JA**, **JI**, **JL**, **JLA**, **JLI**, **JNED**, **JNEI**, **LOOP**

LT
Less Than
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 signed (LT) or unsigned (LT.U) integers. The const9 value is sign-extended (LT) or zero-extended (LT.U).

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 signed 32-bit integers, and the const4 value is sign-extended.

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

31 28	27 21	20 12	11 8	7 0
c	12_H	const9	a	$8B_H$

result = (D[a] < sign_ext(const9));

D[c] = zero_ext(result);

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

31 28	27 20	19 18	17 16	15 12	11 8	7 0
c	12_H	-	-	b	a	$0B_H$

result = (D[a] < D[b]);

D[c] = zero_ext(result);

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

15 12	11 8	7 0
const4	a	FA_H

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

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

15 12	11 8	7 0
b	a	$7A_H$

result = (D[a] < D[b]);
D[15] = zero_ext(result);

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

31 28	27 21	20 12	11 8	7 0
c	13_H	const9	a	$8B_H$

result = (D[a] < zero_ext(const9)); // unsigned

D[c] = zero_ext(result);

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

31 28	27 20	19 18	17 16	15 12	11 8	7 0
c	13_H	-	-	b	a	$0B_H$

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

D[c] = zero_ext(result);

Status Flags

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

Examples

```
lt    d3, d1, d2
lt    d3, d1, #126
lt.u    d3, d1, d2
lt.u    d3, d1, #253
```

```
lt    d15, d1, d2
lt    d15, d1, #6
```

See Also

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

LT.A
Less Than Address

Description

If the contents of address register A[a] are less than the contents of address register A[b], set the least-significant bit of D[c] to one and clear the remaining bits to zero; otherwise clear all bits in D[c]. The operands are treated as unsigned 32-bit integers.

LT.AD[c], A[a], A[b] (RR)

31 28	27 20	19 18	17 16	15 12	11 8	7 0
c	42_H	-	-	b	a	01_H

D[c] = (A[a] < A[b]); // unsigned

Status Flags

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

Examples

```
lt.a    d3, a4, a2
```

See Also

EQ.A, **EQZ.A**, **GE.A**, **NE**, **NEZ.A**

LT.B
Less Than Packed Byte
LT.BU
Less Than Packed Byte Unsigned

Description

Compare each byte of data register D[a] with the corresponding byte of D[b]. In each case, if the value of the byte in D[a] is less than the value of the byte in D[b], set all bits in the corresponding byte of D[c] to one; otherwise clear all the bits. The operands are treated as signed (LT.B) or unsigned (LT.BU) 8-bit integers.

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

31 28	27 20	19 18	17 16	15 12	11 8	7 0
c	52_H	-	-	b	a	$0B_H$

D[c][31:24] = (D[a][31:24] < D[b][31:24]) ? 8’hFF : 8’h00;

D[c][23:16] = (D[a][23:16] < D[b][23:16]) ? 8’hFF : 8’h00;

D[c][15:8] = (D[a][15:8] < D[b][15:8]) ? 8’hFF : 8’h00;

D[c][7:0] = (D[a][7:0] < D[b][7:0]) ? 8’hFF : 8’h00;

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

31 28	27 20	19 18	17 16	15 12	11 8	7 0
c	53_H	-	-	b	a	$0B_H$

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

Examples

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

See Also

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

LT.H
Less Than Packed Half-word
LT.HU
Less Than Packed Half-word Unsigned

Description

Compare each half-word of data register D[a] with the corresponding half-word of D[b]. In each case, if the value of the half-word in D[a] is less than the value of the corresponding half-word in D[b], set all bits of the corresponding half-word of D[c] to one; otherwise clear all the bits. Operands are treated as signed (LT.H) or unsigned (LT.HU) 16-bit integers.

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

31 28	27 20	19 18	17 16	15 12	11 8	7 0
c	72_H	-	-	b	a	$0B_H$

D[c][31:16] = (D[a][31:16] < D[b][31:16]) ? 16’hFFFF : 16’h0000;

D[c][15:0] = (D[a][15:0] < D[b][15:0]) ? 16’hFFFF : 16’h0000;

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

31 28	27 20	19 18	17 16	15 12	11 8	7 0
c	73_H	-	-	b	a	$0B_H$

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

Examples

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

See Also

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

LT.W
Less Than Packed Word
LT.WU
Less Than Packed Word Unsigned

Description

If the contents of data register D[a] are less than the contents of data register D[b], set all bits in D[c] to one; otherwise clear all bits in D[c]. D[a] and D[b] are treated as either signed (LT.W) or unsigned (LT.WU) 32-bit integers.

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

31 28	27 20	19 18	17 16	15 12	11 8	7 0
c	92_H	-	-	b	a	$0B_H$

D[c] = (D[a] < D[b]) ? 32’hFFFFFFFF : 32’h00000000;

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

31 28	27 20	19 18	17 16	15 12	11 8	7 0
c	93_H	-	-	b	a	$0B_H$

D[c] = (D[a] < D[b]) ? 32’hFFFFFFFF : 32’h00000000; // unsigned

Status Flags

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

Examples

```
lt.w    d3, d1, d2
lt.wu   d3, d1, d2
```

See Also

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

MADD
Multiply-Add
MADDS
Multiply-Add, Saturated

Description

Multiply two signed 32-bit integers, add the product to a signed 32-bit or 64-bit integer and put the result into a 32-bit or 64-bit register. The value const9 is sign-extended before the multiplication is performed. The MADDS result is saturated on overflow.

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

32 + (32 * K9)--> 32 signed

31 28	27 24	23 21	20 12	11 8	7 0
c	d	01_H	const9	a	13_H

result = D[d] + (D[a] * sign_ext(const9));

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

MADDE[c], E[d], D[a], const9 (RCR)

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

31 28	27 24	23 21	20 12	11 8	7 0
c	d	03_H	const9	a	13_H

result = E[d] + (D[a] * sign_ext(const9));

E[c] = result[63:0];

MADDD[c], D[d], D[a], D[b] (RRR2)

32 + (32 * 32)--> 32 signed

31 28	27 24	23 16	15 12	11 8	7 0
c	d	$0A_H$	b	a	03_H

result = D[d] + (D[a] * D[b]);

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

MADDE[c], E[d], D[a], D[b] (RRR2)

64 + (32 * 32)--> 64 signed

31 28	27 24	23 16	15 12	11 8	7 0
c	d	$6A_H$	b	a	03_H

result = E[d] + (D[a] * D[b]);

E[c] = result[63:0];

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

32 + (32 * K9)--> 32 signed saturated

31 28	27 24	23 21	20 12	11 8	7 0
c	d	05_H	const9	a	13_H

result = D[d] + (D[a] * sign_ext(const9));

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

MADDSE[c], E[d], D[a], const9 (RCR)

64 + (32 * K9)--> 64 signed saturated

31 28	27 24	23 21	20 12	11 8	7 0
c	d	07_H	const9	a	13_H

result = E[d] + (D[a] * sign_ext(const9));

E[c]= ssov(result, 64);

MADDSD[c], D[d], D[a], D[b] (RRR2)

32 + (32 * 32)--> 32 signed saturated

31 28	27 24	23 16	15 12	11 8	7 0
c	d	$8A_H$	b	a	03_H

result = D[d] + (D[a] * D[b]);

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

MADDSE[c], E[d], D[a], D[b] (RRR2)

64 + (32 * 32)--> 64 signed saturated

31 28	27 24	23 16	15 12	11 8	7 0
c	d	EA_H	b	a	03_H

result = E[d] + (D[a] * D[b]);

E[c] = ssov(result, 64);

Status Flags

C	Not set by these instructions.
V	32-bit result:
	overflow = (result > $7FFFFFFF_H$) 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, d3
madd    d0, d1, d2, #7
madd    e0, e2, d6, d11
madd    e0, e0, d3, #80
madds   d5, d1, d2, d2
madds   d11, d1, d2, #7
madds   e0, e2, d6, d11
madds   e8, e10, d3, #80
```

See Also

-

MADD.H

Packed Multiply-Add Q Format

MADDS.H

Packed Multiply-Add Q Format, Saturated

Description

Multiply two signed 16-bit (half-word) values, add the product (left justified if n == 1) to a signed 32-bit value and put the result into a 32-bit register. There are four cases of half-word multiplication.

Each MADDS.H result is independently saturated on overflow.

Note that n should only take the values 0 or 1, any other value returns an undefined result. If (n == 1) then 8000_H * 8000_H = $7FFFFFFF_H$ (for signed 16-bit * 16-bit multiplications only).

MADD.HE[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	83_H

sc1 = (D[a][31:16] == 8000_H) AND (D[b][15:0] == 8000_H) AND (n == 1);

sc0 = (D[a][15:0] == 8000_H) AND (D[b][15:0] == 8000_H) AND (n == 1);

mul_res1 = sc1 ? $7FFFFFFF_H$: ((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] = {result_word1[31:0], result_word0[31:0]}; // Packed fraction

MADD.HE[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	83_H

sc1 = (D[a][31:16] == 8000_H) AND (D[b][15:0] == 8000_H) AND (n == 1);

sc0 = (D[a][15:0] == 8000_H) AND (D[b][31:16] == 8000_H) AND (n == 1);

mul_res1 = sc1 ? $7FFFFFFF_H$: ((D[a][31:16] * D[b][15:0]) << n);

mul_res0 = sc0 ? $7FFFFFFF_H$: ((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.HE[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	83_H

sc1 = (D[a][31:16] == 8000_H) AND (D[b][31:16] == 8000_H) AND (n == 1);

sc0 = (D[a][15:0] == 8000_H) AND (D[b][15:0] == 8000_H) AND (n == 1);

mul_res1 = sc1 ? $7FFFFFFF_H$: ((D[a][31:16] * D[b][31:16]) << 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] = {result_word1[31:0], result_word0[31:0]}; // Packed fraction

MADD.HE[c], E[d], D[a], D[b] UU, n (RRR1)

32||32 +||+ (16L * 16U || 16U * 16U)--> 32||32

31 28	27 24	23 18	17 16	15 12	11 8	7 0
c	d	$1B_H$	n	b	a	83_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);

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] = {result_word1[31:0], result_word0[31:0]}; // Packed fraction

MADDS.HE[c], E[d], D[a], D[b] LL, n (RRR1)

32||32 +||+ (16U * 16L || 16L * 16L)--> 32||32 saturated

31 28	27 24	23 18	17 16	15 12	11 8	7 0
c	d	$3A_H$	n	b	a	83_H

sc1 = (D[a][31:16] == 8000_H) AND (D[b][15:0] == 8000_H) AND (n == 1);

sc0 = (D[a][15:0] == 8000_H) AND (D[b][15:0] == 8000_H) AND (n == 1);

mul_res1 = sc1 ? $7FFFFFFF_H$: ((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.HE[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	83_H

sc1 = (D[a][31:16] == 8000_H) AND (D[b][15:0] == 8000_H) AND (n == 1);

sc0 = (D[a][15:0] == 8000_H) AND (D[b][31:16] == 8000_H) AND (n == 1);

mul_res1 = sc1 ? $7FFFFFFF_H$: ((D[a][31:16] * D[b][15:0]) << n);

mul_res0 = sc0 ? $7FFFFFFF_H$: ((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.HE[c], E[d], D[a], D[b] UL, n (RRR1)

32||32 +||+ (16U * 16U || 16L * 16L)--> saturated

31 28	27 24	23 18	17 16	15 12	11 8	7 0
c	d	38_H	n	b	a	83_H

```
sc1 = (D[a][31:16] == $8000_H$) AND (D[b][31:16] == $8000_H$) AND (n == 1);
sc0 = (D[a][15:0] == $8000_H$) AND (D[b][15:0] == $8000_H$) AND (n == 1);
mul_res1 = sc1 ? $7FFFFFFF_H$ : ((D[a][31:16] * D[b][31:16]) << 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.HE[c], E[d], D[a], D[b] UU, n (RRR1)

32||32 +||+ (16L * 16U || 16U * 16U)--> 32||32 saturated

31 28	27 24	23 18	17 16	15 12	11 8	7 0
c	d	$3B_H$	n	b	a	83_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);
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 these instructions.

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

–

See Also

-

MADD.Q
Multiply-Add Q Format
MADDS.Q
Multiply-Add Q Format, Saturated

Description

Multiply two signed 16-bit or 32-bit values, add the product (left justified if n == 1) to a signed 32-bit or 64-bit value and put the result into a 32-bit or 64-bit register. There are eight cases of 16*16 operations, eight cases of 16*32 operations and four cases of 32*32 operations. On overflow the MADDS.Q result is saturated.

Note that n should only take the values 0 or 1, any other value returns an undefined result. If (n == 1) then 8000_H * 8000_H = $7FFFFFFF_H$ (for signed 16-bit * 16-bit multiplications only).

MADD.QD[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	43_H

result = D[d] + (((D[a] * D[b]) << n) >> 32);

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

MADD.QE[c], E[d], D[a], D[b], n (RRR1)

64 + (32 * 32) --> 64

31 28	27 24	23 18	17 16	15 12	11 8	7 0
c	d	$1B_H$	n	b	a	43_H

result = E[d] + ((D[a] * D[b]) << n);

E[c] = result[63:0]; // Multi-precision fraction

MADD.QD[c], D[d], D[a], D[b] L, n (RRR1)

32 + (16L * 32)Up --> 32

31 28	27 24	23 18	17 16	15 12	11 8	7 0
c	d	01_H	n	b	a	43_H

result = D[d] + (((D[a] * D[b][15:0]) << n) >> 16);

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

MADD.QE[c], E[d], D[a], D[b] L, n (RRR1)

64 + (16L * 32) --> 64

31 28	27 24	23 18	17 16	15 12	11 8	7 0
c	d	19_H	n	b	a	43_H

result = E[d] + ((D[a] * D[b][15:0]) << n);

E[c] = result[63:0]; // Multi-precision accumulator

MADD.QD[c], D[d], D[a], D[b] U, n (RRR1)

32 + (16U * 32)Up --> 32

31 28	27 24	23 18	17 16	15 12	11 8	7 0
c	d	00_H	n	b	a	43_H

result = D[d]+ (((D[a] * D[b][31:16]) << n) >> 16);

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

MADD.QE[c], E[d], D[a], D[b] U, n (RRR1)

64 + (16U * 32) --> 64

31 28	27 24	23 18	17 16	15 12	11 8	7 0
c	d	18_H	n	b	a	43_H

result = E[d] + ((D[a] * D[b][31:16]) << n);

E[c] = result[63:0]; // Multi-precision accumulator

MADD.QD[c], D[d], D[a] L, D[b] L, n (RRR1)

32 + (16L * 16L) --> 32

31 28	27 24	23 18	17 16	15 12	11 8	7 0
c	d	05_H	n	b	a	43_H

sc = (D[a][15:0] == 8000_H) AND (D[b][15:0] == 8000_H) AND (n == 1);

mul_res = sc ? $7FFFFFFF_H$: ((D[a][15:0] * D[b][15:0]) << n);

result = D[d] + mul_res;

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

MADD.QE[c], E[d], D[a] L, D[b] L, n (RRR1)

64 + (16L * 16L) --> 64

31 28	27 24	23 18	17 16	15 12	11 8	7 0
c	d	$1D_H$	n	b	a	43_H

sc = (D[a][15:0] == 8000_H) AND (D[b][15:0] == 8000_H) AND (n == 1);

mul_res = sc ? $7FFFFFFF_H$: ((D[a][15:0] * D[b][15:0]) << n);

result = E[d] + (mul_res << 16);

E[c] = result[63:0]; // Multi-precision accumulator

MADD.QD[c], D[d], D[a] U, D[b] U, n (RRR1)

32 + (16U * 16U) --> 32

31 28	27 24	23 18	17 16	15 12	11 8	7 0
c	d	04_H	n	b	a	43_H

sc = (D[a][31:16] == 8000_H) AND (D[b][31:16] == 8000_H) AND (n == 1);

mul_res = sc ? $7FFFFFFF_H$: ((D[a][31:16] * D[b][31:16]) << n);

result = D[d] + mul_res;

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

MADD.QE[c], E[d], D[a] U, D[b] U, n (RRR1)

64 + (16U * 16U) --> 64

31 28	27 24	23 18	17 16	15 12	11 8	7 0
c	d	$1C_H$	n	b	a	43_H

sc = (D[a][31:16] == 8000_H) AND (D[b][31:16] == 8000_H) AND (n == 1);

mul_res = sc ? $7FFFFFFF_H$: ((D[a][31:16] * D[b][31:16]) << n);

result = E[d] + (mul_res << 16);

E[c] = result[63:0]; // Multi-precision accumulator

MADDS.QD[c], D[d], D[a], D[b], n (RRR1)

32 + (32 * 32)Up --> 32 saturated

31 28	27 24	23 18	17 16	15 12	11 8	7 0
c	d	22_H	n	b	a	43_H

result = D[d] + (((D[a] * D[b]) << n) >> 32);

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

MADDS.QE[c], E[d], D[a], D[b], n (RRR1)

64 + (32 * 32) --> 64 saturated

31 28	27 24	23 18	17 16	15 12	11 8	7 0
c	d	$3B_H$	n	b	a	43_H

result = E[d] + ((D[a] * D[b]) << n);

E[c] = ssov(result, 64) // Multi-precision fraction

MADDS.QD[c], D[d], D[a], D[b] L, n (RRR1)

32 + (16L * 32)Up --> 32 saturated

31 28	27 24	23 18	17 16	15 12	11 8	7 0
c	d	21_H	n	b	a	43_H

result = D[d] + (((D[a] * D[b][15:0]) << n) >> 16);

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

MADDS.QE[c], E[d], D[a], D[b] L, n (RRR1)

64 + (16L * 32) --> 64 saturated

31 28	27 24	23 18	17 16	15 12	11 8	7 0
c	d	39_H	n	b	a	43_H

result = E[d] + ((D[a] * D[b][15:0]) << n);

E[c] = ssov(result, 64); // Multi-precision accumulator

MADDS.QD[c], D[d], D[a], D[b] U, n (RRR1)

32 + (16U * 32)Up --> 32 saturated

31 28	27 24	23 18	17 16	15 12	11 8	7 0
c	d	20_H	n	b	a	43_H

result = D[d] + (((D[a] * D[b][31:16]) << n) >> 16);

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

MADDS.QE[c], E[d], D[a], D[b] U, n (RRR1)
64 + (16U * 32) --> 64 saturated

31 28	27 24	23 18	17 16	15 12	11 8	7 0
c	d	38_H	n	b	a	43_H

result = E[d] + ((D[a] * D[b][31:16]) << n);

E[c] = ssov(result, 64); // Multi-precision accumulator

MADDS.QD[c], D[d], D[a] L, D[b] L, n (RRR1)
32 + (16L * 16L) --> 32 saturated

31 28	27 24	23 18	17 16	15 12	11 8	7 0
c	d	25_H	n	b	a	43_H

sc = (D[a][15:0] == 8000_H) AND (D[b][15:0] == 8000_H) AND (n == 1);

mul_res = sc ? $7FFFFFFF_H$: ((D[a][15:0] * D[b][15:0]) << n);

result = D[d] + mul_res;

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

MADDS.QE[c], E[d], D[a] L, D[b] L, n (RRR1)
64 + (16L * 16L) --> 64 saturated

31 28	27 24	23 18	17 16	15 12	11 8	7 0
c	d	$3D_H$	n	b	a	43_H

sc = (D[a][15:0] == 8000_H) AND (D[b][15:0] == 8000_H) AND (n == 1);

mul_res = sc ? $7FFFFFFF_H$: ((D[a][15:0] * D[b][15:0]) << n);

result = E[d] + (mul_res << 16);

E[c] = ssov(result, 64); // Multi-precision accumulator

MADDS.QD[c], D[d], D[a] U, D[b] U, n (RRR1)
32 + (16U * 16U) --> 32 saturated

31 28	27 24	23 18	17 16	15 12	11 8	7 0
c	d	24_H	n	b	a	43_H

sc = (D[a][31:16] == 8000_H) AND (D[b][31:16] == 8000_H) AND (n == 1);

mul_res = sc ? $7FFFFFFF_H$: ((D[a][31:16] * D[b][31:16]) << n);

result = D[d] + mul_res;

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

MADDS.QE[c], E[d], D[a] U, D[b] U, n (RRR1)

64 + (16U * 16U) --> 64 saturated

31 28	27 24	23 18	17 16	15 12	11 8	7 0
c	d	$3C_H$	n	b	a	43_H

sc = (D[a][31:16] == 8000_H) AND (D[b][31:16] == 8000_H) AND (n == 1);

mul_res = sc ? $7FFFFFFF_H$: ((D[a][31:16] * D[b][31:16]) << n);

result = E[d] + (mul_res << 16);

E[c] = ssov(result, 64); // Multi-precision accumulator

Status Flags

C	Not set by these instructions.

V	32-bit result:
	overflow = (result > $7FFFFFFF_H$) 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, d3, #1
madd.q   d0, d1, d2, d6U, #1
madd.q   d0, d2, d1, d3L, #1
madd.q   d2, d0, d3U, d4U, #1
madd.q   d2, d0, d4L, d4L, #1
madd.q   e2, e2, d3, d7, #1
madd.q   e2, e2, d4, d6U, #1
madd.q   e2, e2, d5, d6L, #1
madd.q   e2, e2, d6U, d7U, #1
madd.q   e2, e2, d8L, d0L, #1
madds.q   d0, d1, d2, d3, #1
madds.q   d0, d1, d2, d6U, #1
madds.q   d0, d2, d1, d3L, #1
madds.q   d2, d0, d3U, d4U, #1
madds.q   d2, d0, d4L, d4L, #1
madds.q   e2, e2, d3, d7, #1
madds.q   e2, e2, d4, d6U, #1
madds.q   e2, e2, d5, d6L, #1
madds.q   e2, e2, d6U, d7U, #1
madds.q   e2, e0, d11L, d4L, #1
```

See Also

-

MADD.U
Multiply-Add Unsigned
MADDS.U
Multiply-Add Unsigned, Saturated

Description

Multiply two unsigned 32-bit integers, add the product to an unsigned 32-bit or 64-bit integer, and put the result into a 32-bit or 64-bit register. The value const9 is zero-extended before the multiplication is performed. The MADDS.U result is saturated on overflow.

MADD.UE[c], E[d], D[a], const9 (RCR)

64 + (32 * K9) --> 64 unsigned

31 28	27 24	23 21	20 12	11 8	7 0
c	d	02_H	const9	a	13_H

result = E[d] + (D[a] * zero_ext(const9)); // unsigned operators

E[c] = result[63:0];

MADD.UE[c], E[d], D[a], D[b] (RRR2)

32 + (32 * 32) --> 32 unsigned

31 28	27 24	23 16	15 12	11 8	7 0
c	d	68_H	b	a	03_H

result = E[d] + (D[a] * D[b]); // unsigned operators

E[c] = result[63:0];

MADDS.UD[c], D[d], D[a], const9 (RCR)

32 + (32 * K9) --> 32 unsigned saturated

31 28	27 24	23 21	20 12	11 8	7 0
c	d	04_H	const9	a	13_H

result = D[d] + (D[a] * zero_ext(const9)); // unsigned operators

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

MADDS.UE[c], E[d], D[a], const9 (RCR)

64 + (32 * K9) --> 64 unsigned saturated

31 28	27 24	23 21	20 12	11 8	7 0
c	d	06_H	const9	a	13_H

result = E[d] + (D[a] * zero_ext(const9)); // unsigned operators

E[c] = suov(result, 64);

MADDS.UD[c], D[d], D[a], D[b] (RRR2)

32 + (32 * 32) --> unsigned saturated

31 28	27 24	23 16	15 12	11 8	7 0
c	d	88_H	b	a	03_H

result= D[d] + (D[a] * D[b]); // unsigned operators

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

MADDS.UE[c], E[d], D[a], D[b] (RRR2)

64 + (32 * 32) --> 64 unsigned saturated

31 28	27 24	23 16	15 12	11 8	7 0
c	d	$E8_H$	b	a	03_H

result = E[d] + (D[a] * D[b]); // unsigned operators

E[c] = suov(result, 64);

Status Flags

C	Not set by these instructions.

V	32-bit result:
	overflow = (result > $FFFFFFFF_H$);
	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, e2, d6, d11
madd.u    e0, e0, d3, #56
madds.u   d5, d1, d2, d2
madds.u   d11, d1, d2, #7
madds.u   e0, e2, d6, d11
madds.u   e8, e0, d0, #80
```

See Also

-

MADDM.H

Packed Multiply-Add Q Format Multi-precision

MADDMS.H

Packed Multiply-Add Q Format Multi-precision, Saturated

Description

Perform two multiplications of two signed 16-bit (half-word) values. Add the two products (left justified if n == 1) left-shifted by 16, to a signed 64-bit value and put the result in a 64-bit register. The MADDMS.H result is saturated on overflow. There are four cases of half-word multiplication.

Note that n should only take the values 0 or 1, any other value returns an undefined result. If (n == 1) then 8000_H * 8000_H = $7FFFFFFF_H$ (for signed 16-bit * 16-bit multiplications only).

MADDM.HE[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	83_H

sc1 = (D[a][31:16] == 8000_H) AND (D[b][15:0] == 8000_H) AND (n == 1);

sc0 = (D[a][15:0] == 8000_H) AND (D[b][15:0] == 8000_H) 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 = E[d] + ((result_word1 + result_word0) << 16);

E[c] = result[63:0]; // Multi-precision accumulator

MADDM.HE[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	83_H

sc1 = (D[a][31:16] == 8000_H) AND (D[b][15:0] == 8000_H) AND (n == 1);

sc0 = (D[a][15:0] == 8000_H) AND (D[b][31:16] == 8000_H) 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][31:16]) << n);

result = E[d] + ((result_word1 + result_word0) << 16);

E[c] = result[63:0]; // Multi-precision accumulator

MADDM.HE[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	83_H

sc1 = (D[a][31:16] == 8000_H) AND (D[b][31:16] == 8000_H) AND (n == 1);

sc0 = (D[a][15:0] == 8000_H) AND (D[b][15:0] == 8000_H) AND (n == 1);

result_word1 = sc1 ? $7FFFFFFF_H$: ((D[a][31:16] * D[b][31:16]) << n);

result_word0 = sc0 ? $7FFFFFFF_H$: ((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.HE[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	83_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 = E[d] + ((result_word1 + result_word0) << 16);

E[c] = result[63:0]; // Multi-precision accumulator

MADDMS.HE[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	83_H

sc1 = (D[a][31:16] == 8000_H) AND (D[b][15:0] == 8000_H) AND (n == 1);

sc0 = (D[a][15:0] == 8000_H) AND (D[b][15:0] == 8000_H) 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 = E[d] + ((result_word1 + result_word0) << 16);

E[c] = ssov(result, 64); // Multi-precision accumulator

MADDMS.HE[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	83_H

sc1 = (D[a][31:16] == 8000_H) AND (D[b][15:0] == 8000_H) AND (n == 1);

sc0 = (D[a][15:0] == 8000_H) AND (D[b][31:16] == 8000_H) 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][31:16]) << n);

result = E[d] + ((result_word1 + result_word0) << 16);

E[c] = ssov(result, 64); // Multi-precision accumulator

MADDMS.HE[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	83_H

sc1 = (D[a][31:16] == 8000_H) AND (D[b][31:16] == 8000_H) AND (n == 1);

sc0 = (D[a][15:0] == 8000_H) AND (D[b][15:0] == 8000_H) AND (n == 1);

result_word1 = sc1 ? $7FFFFFFF_H$: ((D[a][31:16] * D[b][31:16]) << n);

result_word0 = sc0 ? $7FFFFFFF_H$: ((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.HE[c], E[d], D[a], D[b] UU, n (RRR1)

64 + (16L * 16U) + (16U * 16U) --> 64 saturated

31 28	27 24	23 18	17 16	15 12	11 8	7 0
c	d	$3F_H$	n	b	a	83_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 = E[d] + ((result_word1 + result_word0) << 16);

E[c] = ssov(result, 64); // Multi-precision accumulator

Status Flags

C	Not set by these instructions.

V	overflow = (result > $7FFFFFFFFFFFFFFF_H$) 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

–

See Also

-

MADDR.H
Packed Multiply-Add Q Format with Rounding
MADDRS.H
Packed Multiply-Add Q Format with Rounding, Saturated

Description

Multiply two signed 16-bit (half-word) values, add the product (left justified if n == 1) to a signed 16-bit or 32-bit value and put the rounded result into half of a 32-bit register (Note that since there are two results the two register halves are used). There are four cases of half-word multiplication:

- 16U * 16U, 16L * 16L
- 16U * 16L, 16L * 16U
- 16U * 16L, 16L * 16L
- 16L * 16U, 16U * 16U

Note that n should only take the values 0 or 1, any other value returns an undefined result. If (n == 1) then 8000_H * 8000_H = $7FFFFFFF_H$ (for signed 16-bit * 16-bit multiplications only).

MADDR.HD[c], D[d], D[a], D[b] LL, n (RRR1)

16U || 16L +||+ (16U * 16L || 16L * 16L) rounded --> 16||16

31 28	27 24	23 18	17 16	15 12	11 8	7 0
c	d	$0E_H$	n	b	a	83_H

sc1 = (D[a][31:16] == 8000_H) AND (D[b][15:0] == 8000_H) AND (n == 1);

sc0 = (D[a][15:0] == 8000_H) AND (D[b][15:0] == 8000_H) AND (n == 1);

mul_res1 = sc1 ? $7FFFFFFF_H$: ((D[a][31:16] * D[b][15:0]) << n);

mul_res0 = sc0 ? $7FFFFFFF_H$: ((D[a][15:0] * D[b][15:0]) << n);

result_halfword1 = {D[d][31:16], 16'b0} + mul_res1 + 8000_H;

result_halfword0 = {D[d][15:0], 16'b0} + mul_res0 + 8000_H;

D[c] = {result_halfword1[31:16], result_halfword0[31:16]}; // Packed short fraction

MADDR.HD[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	83_H

sc1 = (D[a][31:16] == 8000_H) AND (D[b][15:0] == 8000_H) AND (n == 1);

sc0 = (D[a][15:0] == 8000_H) AND (D[b][31:16] == 8000_H) AND (n == 1);

mul_res1 = sc1 ? $7FFFFFFF_H$: ((D[a][31:16] * D[b][15:0]) << n);

mul_res0 = sc0 ? $7FFFFFFF_H$: ((D[a][15:0] * D[b][31:16]) << n);

result_halfword1 = {D[d][31:16], 16'b0} + mul_res1 + 8000_H;

result_halfword0 = {D[d][15:0], 16'b0} + mul_res0 + 8000_H;

D[c] = {result_halfword1[31:16], result_halfword0[31:16]}; // Packed short fraction

MADDR.HD[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	83_H

sc1 = (D[a][31:16] == 8000_H) AND (D[b][31:16] == 8000_H) AND (n == 1);

sc0 = (D[a][15:0] == 8000_H) AND (D[b][15:0] == 8000_H) AND (n == 1);

mul_res1 = sc1 ? $7FFFFFFF_H$: ((D[a][31:16] * D[b][31:16]) << n);

mul_res0 = sc0 ? $7FFFFFFF_H$: ((D[a][15:0] * D[b][15:0]) << n);

result_halfword1 = {D[d][31:16], 16’b0} + mul_res1 + 8000_H;

result_halfword0 = {D[d][15:0], 16’b0} + mul_res0 + 8000_H;

D[c] = {result_halfword1[31:16], result_halfword0[31:16]}; // Packed short fraction

MADDR.HD[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	43_H

sc1 = (D[a][31:16] == 8000_H) AND (D[b][31:16] == 8000_H) AND (n == 1);

sc0 = (D[a][15:0] == 8000_H) AND (D[b][15:0] == 8000_H) AND (n == 1);

mul_res1 = sc1 ? $7FFFFFFF_H$: ((D[a][31:16] * D[b][31:16]) << n);

mul_res0 = sc0 ? $7FFFFFFF_H$: ((D[a][15:0] * D[b][15:0]) << n);

result_halfword1 = E[d][63:32] + mul_res1 + 8000_H;

result_halfword0 = E[d][31:0] + mul_res0 + 8000_H;

D[c] = {result_halfword1[31:16], result_halfword0[31:16]}; // Packed short fraction

MADDR.HD[c], D[d], D[a], D[b] UU, n (RRR1)

16U || 16L +||+ (16L * 16U || 16U * 16U) rounded --> 16||16

31 28	27 24	23 18	17 16	15 12	11 8	7 0
c	d	$0F_H$	n	b	a	83_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);

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_halfword1 = {D[d][31:16], 16’b0} + mul_res1 + 8000_H;

result_halfword0 = {D[d][15:0], 16’b0} + mul_res0 + 8000_H;

D[c] = {result_halfword1[31:16], result_halfword0[31:16]}; // Packed short fraction

MADDRS.HD[c], D[d], D[a], D[b] LL, n (RRR1)

16U || 16L +||+ (16U * 16L || 16L * 16L) rounded --> 16||16 saturated

31 28	27 24	23 18	17 16	15 12	11 8	7 0
c	d	$2E_H$	n	b	a	83_H

sc1 = (D[a][31:16] == 8000_H) AND (D[b][15:0] == 8000_H) AND (n == 1);

sc0 = (D[a][15:0] == 8000_H) AND (D[b][15:0] == 8000_H) AND (n == 1);

mul_res1 = sc1 ? $7FFFFFFF_H$: ((D[a][31:16] * D[b][15:0]) << n);

mul_res0 = sc0 ? $7FFFFFFF_H$: ((D[a][15:0] * D[b][15:0]) << n);

result_halfword1 = {D[d][31:16], 16’b0} + mul_res1 + 8000_H;

result_halfword0 = {D[d][15:0], 16’b0} + mul_res0 + 8000_H;

D[c] = {ssov(result_halfword1, 32)[31:16], ssov(result_halfword0, 32)[31:16]};

// Packed short fraction result

MADDRS.HD[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	83_H

sc1 = (D[a][31:16] == 8000_H) AND (D[b][15:0] == 8000_H) AND (n == 1);

sc0 = (D[a][15:0] == 8000_H) AND (D[b][31:16] == 8000_H) AND (n == 1);

mul_res1 = sc1 ? $7FFFFFFF_H$: ((D[a][31:16] * D[b][15:0]) << n);

mul_res0 = sc0 ? $7FFFFFFF_H$: ((D[a][15:0] * D[b][31:16]) << n);

result_halfword1 = {D[d][31:16], 16’b0} + mul_res1 + 8000_H;

result_halfword0 = {D[d][15:0], 16’b0} + mul_res0 + 8000_H;

D[c] = {ssov(result_halfword1, 32)[31:16], ssov(result_halfword0, 32)[31:16]};

// Packed short fraction result

MADDRS.HD[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] == 8000_H) AND (D[b][31:16] == 8000_H) AND (n == 1);

sc0 = (D[a][15:0] == 8000_H) AND (D[b][15:0] == 8000_H) AND (n == 1);

mul_res1 = sc1 ? $7FFFFFFF_H$: ((D[a][31:16] * D[b][31:16]) << n);

mul_res0 = sc0 ? $7FFFFFFF_H$: ((D[a][15:0] * D[b][15:0]) << n);

result_halfword1 = {D[d][31:16], 16’b0} + mul_res1 + 8000_H;

result_halfword0 = {D[d][15:0], 16’b0} + mul_res0 + 8000_H;

D[c] = {ssov(result_halfword1, 32)[31:16], ssov(result_halfword0, 32)[31:16]};

// Packed short fraction result

// Packed short fraction result

MADDRS.HD[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] == 8000_H) AND (D[b][31:16] == 8000_H) AND (n == 1);

sc0 = (D[a][15:0] == 8000_H) AND (D[b][15:0] == 8000_H) AND (n == 1);

mul_res1 = sc1 ? $7FFFFFFF_H$: ((D[a][31:16] * D[b][31:16]) << n);

mul_res0 = sc0 ? $7FFFFFFF_H$: ((D[a][15:0] * D[b][15:0]) << n);

result_halfword1 = E[d][63:32] + mul_res1 + 8000_H;

result_halfword0 = E[d][31:0] + mul_res0 + 8000_H;

D[c] = {ssov(result_halfword1, 32)[31:16], ssov(result_halfword0, 32)[31:16]};

MADDRS.HD[c], D[d], D[a], D[b] UU, n (RRR1)

16U || 16L +||+ (16L * 16U || 16U * 16U) rounded --> 16||16 saturated

31 28	27 24	23 18	17 16	15 12	11 8	7 0
c	d	$2F_H$	n	b	a	83_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);

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_halfword1 = {D[d][31:16], 16’b0} + mul_res1 + 8000_H;

result_halfword0 = {D[d][15:0], 16’b0} + mul_res0 + 8000_H;

D[c] = {ssov(result_halfword1, 32)[31:16], ssov(result_halfword0, 32)[31:16]};

// Packed short fraction result

Status Flags

C	Not set by these instructions.

V	ov_halfword1 = (result_halfword1 > $7FFFFFFF_H$) OR (result_halfword1 < -80000000_H);
	ov_halfword0 = (result_halfword0 > $7FFFFFFF_H$) OR (result_halfword0 < -80000000_H);
	overflow = ov_halfword1 OR ov_halfword0;
	if (overflow) then PSW.V = 1 else PSW.V = 0;

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

AV	aov_halfword1 = result_halfword1[31] ^ result_halfword1[30];
	aov_halfword0 = result_halfword0[31] ^ result_halfword0[30];
	advanced_overflow = aov_halfword1 OR aov_halfword0;
	if (advanced_overflow) then PSW.AV = 1 else PSW.AV = 0;

SAV	if (advanced_overflow) then PSW.SAV = 1 else PSW.SAV = PSW.SAV;

Examples

–

See Also

-

MADDR.Q

Multiply-Add Q Format with Rounding

MADDRS.Q

Multiply-Add Q Format with Rounding, Saturated

Description

Multiply two signed 16-bit (half-word) values, add the product (left justified if n == 1) to a 32-bit signed value, and put the rounded result in a 32-bit register. The lower half-word is cleared. Overflow and advanced overflow are calculated on the final results.

Note that n should only take the values 0 or 1, any other value returns an undefined result. If (n == 1) then 8000_H * 8000_H = $7FFFFFFF_H$ (for signed 16-bit * 16-bit multiplications only).

MADDR.QD[c], D[d], D[a] L, D[b] L, n (RRR1)

32 + (16L * 16L) rounded --> 32

31 28	27 24	23 18	17 16	15 12	11 8	7 0
c	d	07_H	n	b	a	43_H

sc = (D[a][15:0] == 8000_H) AND (D[b][15:0] == 8000_H) AND (n == 1);

mul_res = sc ? $7FFFFFFF_H$: ((D[a][15:0] * D[b][15:0]) << n);

result = D[d] + mul_res + 8000_H;

D[c] = {result[31:16], 16’b0}; // Short fraction

MADDR.QD[c], D[d], D[a] U, D[b] U, n (RRR1)

32 + (16U * 16U) rounded --> 32

31 28	27 24	23 18	17 16	15 12	11 8	7 0
c	d	06_H	n	b	a	43_H

sc = (D[a][31:16] == 8000_H) AND (D[b][31:16] == 8000_H) AND (n == 1);

mul_res = sc ? $7FFFFFFF_H$: ((D[a][31:16] * D[b][31:16]) << n);

result = D[d] + mul_res + 8000_H;

D[c] = {result[31:16], 16’b0}; // Short fraction

MADDRS.QD[c], D[d], D[a] L, D[b] L, n (RRR1)

32 + (16L * 16L) rounded --> 32 saturated

31 28	27 24	23 18	17 16	15 12	11 8	7 0
c	d	27_H	n	b	a	43_H

sc = (D[a][15:0] == 8000_H) AND (D[b][15:0] == 8000_H) AND (n == 1);

mul_res = sc ? $7FFFFFFF_H$: ((D[a][15:0] * D[b][15:0]) << n);

result = D[d] + mul_res + 8000_H;

D[c] = {ssov(result,32)[31:16]), 16’b0}; // Short fraction

MADDRS.QD[c], D[d], D[a] U, D[b] U, n (RRR1)

32 + (16U * 16U) rounded --> 32 saturated

31 28	27 24	23 18	17 16	15 12	11 8	7 0
c	d	26_H	n	b	a	43_H

```
sc = (D[a][31:16] == $8000_H$) AND (D[b][31:16] == $8000_H$) AND (n == 1);
mul_res = sc ? $7FFFFFFF_H$ : ((D[a][31:16] * D[b][31:16]) << n);
result = D[d] + mul_res + $8000_H$;
D[c] = {ssov(result,32)[31:16]), 16'b0}; // Short fraction
```

Status Flags

C	Not set by these instructions.

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

–

See Also

-

MADDSU.H
Packed Multiply-Add/Subtract Q Format
MADDSUS.H
Packed Multiply-Add/Subtract Q Format Saturated

Description

Multiply two signed 16-bit (half-word) values. Add (or subtract) the product (left justified if n == 1) to a signed 32-bit value and put the result into a 32-bit register. Each MADDSUS.H result is independently saturated on overflow. There are four cases of half-word multiplication:

- 16U * 16U, 16L * 16L
- 16U * 16L, 16L * 16U
- 16U * 16L, 16L * 16L
- 16L * 16U, 16U * 16U

Note that n should only take the values 0 or 1, any other value returns an undefined result. If (n == 1) then 8000_H * 8000_H = $7FFFFFFF_H$ (for signed 16-bit * 16-bit multiplications only).

MADDSU.HE[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] == 8000_H) AND (D[b][15:0] == 8000_H) AND (n == 1);

sc0 = (D[a][15:0] == 8000_H) AND (D[b][15:0] == 8000_H) AND (n == 1);

mul_res1 = sc1 ? $7FFFFFFF_H$: ((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] = {result_word1[31:0], result_word0[31:0]}; // Packed fraction

MADDSU.HE[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] == 8000_H) AND (D[b][15:0] == 8000_H) AND (n == 1);

sc0 = (D[a][15:0] == 8000_H) AND (D[b][31:16] == 8000_H) AND (n == 1);

mul_res1 = sc1 ? $7FFFFFFF_H$: ((D[a][31:16] * D[b][15:0]) << n);

mul_res0 = sc0 ? $7FFFFFFF_H$: ((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.HE[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] == 8000$_H$) AND (D[b][31:16] == 8000$_H$) AND (n == 1);

sc0 = (D[a][15:0] == 8000$_H$) AND (D[b][15:0] == 8000$_H$) AND (n == 1);

mul_res1 = sc1 ? 7FFFFFFF$_H$: ((D[a][31:16] * D[b][31:16]) << 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] = {result_word1[31:0], result_word0[31:0]}; // Packed fraction

MADDSU.HE[c], E[d], D[a], D[b] UU, n (RRR1)

32||32 +||- (16L * 16U || 16U * 16U) --> 32||32

31 28	27 24	23 18	17 16	15 12	11 8	7 0
c	d	$1B_H$	n	b	a	$C3_H$

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);

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] = {result_word1[31:0], result_word0[31:0]}; // Packed fraction

MADDSUS.HE[c], E[d], D[a], D[b] LL, n (RRR1)

32||32 +||- (16U * 16L || 16L * 16L) --> 32||32 saturated

31 28	27 24	23 18	17 16	15 12	11 8	7 0
c	d	$3A_H$	n	b	a	$C3_H$

sc1 = (D[a][31:16] == 8000$_H$) AND (D[b][15:0] == 8000$_H$) AND (n == 1);

sc0 = (D[a][15:0] == 8000$_H$) AND (D[b][15:0] == 8000$_H$) AND (n == 1);

mul_res1 = sc1 ? 7FFFFFFF$_H$: ((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

MADDSUS.HE[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	$C3_H$

sc1 = (D[a][31:16] == 8000$_H$) AND (D[b][15:0] == 8000$_H$) AND (n == 1);

sc0 = (D[a][15:0] == 8000$_H$) AND (D[b][31:16] == 8000$_H$) AND (n == 1);

mul_res1 = sc1 ? 7FFFFFFF$_H$: ((D[a][31:16] * D[b][15:0]) << n);
mul_res0 = sc0 ? 7FFFFFFF$_H$: ((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.HE[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	C3$_H$

sc1 = (D[a][31:16] == 8000$_H$) AND (D[b][31:16] == 8000$_H$) AND (n == 1);
sc0 = (D[a][15:0] == 8000$_H$) AND (D[b][15:0] == 8000$_H$) AND (n == 1);
mul_res1 = sc1 ? 7FFFFFFF$_H$: ((D[a][31:16] * D[b][31:16]) << 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

MADDSUS.HE[c], E[d], D[a], D[b] UU, n (RRR1)
32||32 +||- (16L * 16U || 16U * 16U) --> 32||32 saturated

31 28	27 24	23 18	17 16	15 12	11 8	7 0
c	d	3B$_H$	n	b	a	C3$_H$

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);
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 these instructions.

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

–

See Also

-

MADDSUM.H

Packed Multiply-Add/Subtract Q Format Multi-precision

MADDSUMS.H

Packed Multiply-Add/Subtract Q Format Multi-precision Saturated

Description

Perform two multiplications of two signed 16-bit (half-word) values. Add one product and subtract the other product (left justified if n == 1) left-shifted by 16, to/from a signed 64-bit value and put the result in a 64-bit register. The MADDSUMS.H result is saturated on overflow. There are four cases of half-word multiplication:

- 16U * 16U, 16L * 16L
- 16U * 16L, 16L * 16U
- 16U * 16L, 16L * 16L
- 16L * 16U, 16U * 16U

Note that n should only take the values 0 or 1, any other value returns an undefined result. If (n == 1) then 8000_H * 8000_H = $7FFFFFFF_H$ (for signed 16-bit * 16-bit multiplications only).

MADDSUM.HE[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	$C3_H$

sc1 = (D[a][31:16] == 8000_H) AND (D[b][15:0] == 8000_H) AND (n == 1);

sc0 = (D[a][15:0] == 8000_H) AND (D[b][15:0] == 8000_H) 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 = E[d] + ((result_word1 - result_word0) << 16);

E[c] = result[63:0]; // Multi-precision accumulator

MADDSUM.HE[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	$C3_H$

sc1 = (D[a][31:16] == 8000_H) AND (D[b][15:0] == 8000_H) AND (n == 1);

sc0 = (D[a][15:0] == 8000_H) AND (D[b][31:16] == 8000_H) 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][31:16]) << n);

result = E[d] + ((result_word1 - result_word0) << 16);

E[c] = result[63:0]; // Multi-precision accumulator

MADDSUM.HE[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	$C3_H$

sc1 = (D[a][31:16] == 8000_H) AND (D[b][31:16] == 8000_H) AND (n == 1);

sc0 = (D[a][15:0] == 8000_H) AND (D[b][15:0] == 8000_H) AND (n == 1);

result_word1 = sc1 ? $7FFFFFFF_H$: ((D[a][31:16] * D[b][31:16]) << n);

result_word0 = sc0 ? $7FFFFFFF_H$: ((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.HE[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] == 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 = E[d] + ((result_word1 - result_word0) << 16);

E[c] = result[63:0]; // Multi-precision accumulator

MADDSUMS.H E[c], E[d], D[a], D[b] LL, n (RRR1)

64 + (16U * 16L) - (16L * 16L) --> 64 saturated

31 28	27 24	23 18	17 16	15 12	11 8	7 0
c	d	$3E_H$	n	b	a	$C3_H$

sc1 = (D[a][31:16] == 8000_H) AND (D[b][15:0] == 8000_H) AND (n == 1);

sc0 = (D[a][15:0] == 8000_H) AND (D[b][15:0] == 8000_H) 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 = 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

31 28	27 24	23 18	17 16	15 12	11 8	7 0
c	d	$3D_H$	n	b	a	$C3_H$

sc1 = (D[a][31:16] == 8000_H) AND (D[b][15:0] == 8000_H) AND (n == 1);

sc0 = (D[a][15:0] == 8000_H) AND (D[b][31:16] == 8000_H) 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][31:16]) << n);

result = E[d] + ((result_word1 - result_word0) << 16);

E[c] = ssov(result, 64); // Multi-precision accumulator

MADDSUMS.H E[c], E[d], D[a], D[b] UL, n (RRR1)
64 + (16U * 16U) - (16L * 16L) --> 64 saturated

31 28	27 24	23 18	17 16	15 12	11 8	7 0
c	d	$3C_H$	n	b	a	$C3_H$

sc1 = (D[a][31:16] == 8000_H) AND (D[b][31:16] == 8000_H) AND (n == 1);
sc0 = (D[a][15:0] == 8000_H) AND (D[b][15:0] == 8000_H) AND (n == 1);
result_word1 = sc1 ? $7FFFFFFF_H$: ((D[a][31:16] * D[b][31:16]) << n);
result_word0 = sc0 ? $7FFFFFFF_H$: ((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

31 28	27 24	23 18	17 16	15 12	11 8	7 0
c	d	$3F_H$	n	b	a	$C3_H$

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 = E[d] + ((result_word1 - result_word0) << 16);
E[c] = ssov(result, 64); // Multi-precision accumulator

Status Flags

C	Not set by these instructions.

V	overflow = (result > $7FFFFFFFFFFFFFFF_H$) 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

–

See Also

-

MADDSUR.H
Packed Multiply-Add/Subtract Q Format with Rounding
MADDSURS.H
Packed Multiply-Add/Subtract Q Format with Rounding Saturated

Description

Multiply two signed 16-bit (half-word) values. Add (subtract) the product (left justified if n == 1) to (from) a signed 16-bit value and put the rounded result into half of a 32-bit register (Note that since there are two results, the two register halves are used). There are four cases of half-word multiplication:

- 16U * 16U, 16L * 16L
- 16U * 16L, 16L * 16U
- 16U * 16L, 16L * 16L
- 16L * 16U, 16U * 16U

Note that n should only take the values 0 or 1, any other value returns an undefined result. If (n == 1) then 8000_H * 8000_H = $7FFFFFFF_H$ (for signed 16-bit * 16-bit multiplications only).

MADDSUR.HD[c], D[d], D[a], D[b] LL, n (RRR1)

16U * 16L +||- (16U * 16L || 16L * 16L) rounded --> 16||16

31 28	27 24	23 18	17 16	15 12	11 8	7 0
c	d	$0E_H$	n	b	a	$C3_H$

sc1 = (D[a][31:16] == 8000_H) AND (D[b][15:0] == 8000_H) AND (n == 1);
sc0 = (D[a][15:0] == 8000_H) AND (D[b][15:0]] == 8000_H) AND (n == 1);
mul_res1 = sc1 ? $7FFFFFFF_H$: ((D[a][31:16] * D[b][15:0]) << n);
mul_res0 = sc0 ? $7FFFFFFF_H$: ((D[a][15:0] * D[b][15:0]) << n);
result_halfword1 = {D[d][31:16], 16'b0} + mul_res1 + 8000_H;
result_halfword0 = {D[d][15:0], 16'b0} - mul_res0 + 8000_H;
D[c] = {result_halfword1[31:16], result_halfword0[31:16]}; // Packed short fraction

MADDSUR.HD[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	$C3_H$

sc1 = (D[a][31:16] == 8000_H) AND (D[b][15:0] == 8000_H) AND (n == 1);
sc0 = (D[a][15:0] == 8000_H) AND (D[b][31:16]] == 8000_H) AND (n == 1);
mul_res1 = sc1 ? $7FFFFFFF_H$: ((D[a][31:16] * D[b][15:0]) << n);
mul_res0 = sc0 ? $7FFFFFFF_H$: ((D[a][15:0] * D[b][31:16]) << n);
result_halfword1 = {D[d][31:16], 16'b0} + mul_res1 + 8000_H;
result_halfword0 = {D[d][15:0], 16'b0} - mul_res0 + 8000_H;
D[c] = {result_halfword1[31:16], result_halfword0[31:16]}; // Packed short fraction

MADDSUR.HD[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	$C3_H$

sc1 = (D[a][31:16] == 8000_H) AND (D[b][31:16] == 8000_H) AND (n == 1);

sc0 = (D[a][15:0] == 8000_H) AND (D[b][15:0] == 8000_H) AND (n == 1);

mul_res1 = sc1 ? $7FFFFFFF_H$: ((D[a][31:16] * D[b][31:16]) << n);

mul_res0 = sc0 ? $7FFFFFFF_H$: ((D[a][15:0] * D[b][15:0]) << n);

result_halfword1 = {D[d][31:16], 16’b0} + mul_res1 + 8000_H;

result_halfword0 = {D[d][15:0], 16’b0} - mul_res0 + 8000_H;

D[c] = {result_halfword1[31:16], result_halfword0[31:16]}; // Packed short fraction

MADDSUR.HD[c], D[d], D[a], D[b] UU, n (RRR1)

16U || 16L +||- (16L * 16U || 16U * 16U) rounded --> 16||16

31 28	27 24	23 18	17 16	15 12	11 8	7 0
c	d	$0F_H$	n	b	a	$C3_H$

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);

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_halfword1 = {D[d][31:16], 16’b0} + mul_res1 + 8000_H;

result_halfword0 = {D[d][15:0], 16’b0} - mul_res0 + 8000_H;

D[c] = {result_halfword1[31:16], result_halfword0[31:16]}; // Packed short fraction

MADDSURS.HD[c], D[d], D[a], D[b] LL, n (RRR1)

16U || 16L +||- (16U * 16L || 16L * 16L) rounded --> 16||16 saturated

31 28	27 24	23 18	17 16	15 12	11 8	7 0
c	d	$2E_H$	n	b	a	$C3_H$

sc1 = (D[a][31:16] == 8000_H) AND (D[b][15:0] == 8000_H) AND (n == 1);

sc0 = (D[a][15:0] == 8000_H) AND (D[b][15:0]] == 8000_H) AND (n == 1);

mul_res1 = sc1 ? $7FFFFFFF_H$: ((D[a][31:16] * D[b][15:0]) << n);

mul_res0 = sc0 ? $7FFFFFFF_H$: ((D[a][15:0] * D[b][15:0]) << n);

result_halfword1 = {D[d][31:16], 16’b0} + mul_res1 + 8000_H;

result_halfword0 = {D[d][15:0], 16’b0} - mul_res0 + 8000_H;

D[c] = {ssov(result_halfword1, 32)[31:16], ssov(result_halfword0, 32)[31:16]};

// Packed short fraction result

MADDSURS.HD[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] == 8000_H) AND (D[b][15:0] == 8000_H) AND (n == 1);

sc0 = (D[a][15:0] == 8000$_H$) AND (D[b][31:16]] == 8000$_H$) AND (n == 1);
mul_res1 = sc1 ? 7FFFFFFF$_H$: ((D[a][31:16] * D[b][15:0]) << n);
mul_res0 = sc0 ? 7FFFFFFF$_H$: ((D[a][15:0] * D[b][31:16]) << n);
result_halfword1 = {D[d][31:16], 16’b0} + mul_res1 + 8000$_H$;
result_halfword0 = {D[d][15:0], 16’b0} - mul_res0 + 8000$_H$;
D[c] = {ssov(result_halfword1, 32)[31:16], ssov(result_halfword0, 32)[31:16]};
// Packed short fraction result

MADDSURS.HD[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] == 8000$_H$) AND (D[b][31:16] == 8000$_H$) AND (n == 1);
sc0 = (D[a][15:0] == 8000$_H$) AND (D[b][15:0] == 8000$_H$) AND (n == 1);
mul_res1 = sc1 ? 7FFFFFFF$_H$: ((D[a][31:16] * D[b][31:16]) << n);
mul_res0 = sc0 ? 7FFFFFFF$_H$: ((D[a][15:0] * D[b][15:0]) << n);
result_halfword1 = {D[d][31:16], 16’b0} + mul_res1 + 8000$_H$;
result_halfword0 = {D[d][15:0], 16’b0} - mul_res0 + 8000$_H$;
D[c] = {ssov(result_halfword1, 32)[31:16], ssov(result_halfword0, 32)[31:16]};
// Packed short fraction result

MADDSURS.HD[c], D[d], D[a], D[b] UU, n (RRR1)
16U || 16L +||- (16L * 16U || 16U * 16U) rounded --> 16||16 saturated

31 28	27 24	23 18	17 16	15 12	11 8	7 0
c	d	2F$_H$	n	b	a	C3$_H$

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);
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_halfword1 = {D[d][31:16], 16’b0} + mul_res1 + 8000$_H$;
result_halfword0 = {D[d][15:0], 16’b0} - mul_res0 + 8000$_H$;
D[c] = {ssov(result_halfword1, 32)[31:16], ssov(result_halfword0, 32)[31:16]};
// Packed short fraction result

Status Flags

C	Not set by these instructions.

V	ov_halfword1 =
	(result_halfword1 > 7FFFFFFF$_H$) OR (result_halfword1 < -80000000$_H$);
	ov_halfword0 =
	(result_halfword0 > 7FFFFFFF$_H$) OR (result_halfword0 < -80000000$_H$);
	overflow = ov_halfword1 OR ov_halfword0;
	if (overflow) then PSW.V = 1 else PSW.V = 0;

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

AV	aov_halfword1 = result_halfword1[31] ^ result_halfword1[30];
	aov_halfword0 = result_halfword0[31] ^ result_halfword0[30];
	advanced_overflow = aov_halfword1 OR aov_halfword0;
	if (advanced_overflow) then PSW.AV = 1 else PSW.AV = 0;

SAV	if (advanced_overflow) then PSW.SAV = 1 else PSW.SAV = PSW.SAV;

Examples

–

See Also

-

MAX
Maximum Value
MAX.U
Maximum Value Unsigned

Description

If the contents of data register D[a] are greater than the contents of either data register D[b] (instruction format RR) or const9 (instruction format RC), then put the contents of D[a] in data register D[c]; otherwise put the contents of either D[b] (format RR) or const9 (format RC) in D[c]. The operands are treated as either signed (MAX) or unsigned (MAX.U) 32-bit integers.

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

31 28	27 21	20 12	11 8	7 0
c	$1A_H$	const9	a	$8B_H$

D[c] = (D[a] > sign_ext(const9)) ? D[a] : sign_ext(const9);

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

31 28	27 20	19 18	17 16	15 12	11 8	7 0
c	$1A_H$	-	-	b	a	$0B_H$

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

MAX.UD[c], D[a], const9 (RC)

31 28	27 21	20 12	11 8	7 0
c	$1B_H$	const9	a	$8B_H$

D[c] = (D[a] > zero_ext(const9)) ? D[a] : zero_ext(const9); // unsigned

MAX.UD[c], D[a], D[b] (RR)

31 28	27 20	19 18	17 16	15 12	11 8	7 0
c	$1B_H$	-	-	b	a	$0B_H$

D[c] = (D[a] > D[b]) ? D[a] : D[b]; // unsigned

Status Flags

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

Examples

```
max    d3, d1, d2
max    d3, d1, #126
```

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

See Also

MIN, **MOV**

MAX.B
Maximum Value Packed Byte
MAX.BU
Maximum Value Packed Byte Unsigned

Description

Compute the maximum value of the corresponding bytes in D[a] and D[b] and put each result in the corresponding byte of D[c]. The operands are treated as either signed (MAX.B) or unsigned (MAX.BU), 8-bit integers.

MAX.BD[c], D[a], D[b] (RR)

31 28	27 20	19 18	17 16	15 12	11 8	7 0
c	$5A_H$	-	-	b	a	$0B_H$

D[c][31:24] = (D[a][31:24] > D[b][31:24]) ? D[a][31:24] : D[b][31:24];

D[c][23:16] = (D[a][23:16] > D[b][23:16]) ? D[a][23:16] : D[b][23:16];

D[c][15:8] = (D[a][15:8] > D[b][15:8]) ? D[a][15:8] : D[b][15:8];

D[c][7:0] = (D[a][7:0] > D[b][7:0]) ? D[a][7:0] : D[b][7:0];

MAX.BUD[c], D[a], D[b] (RR)

31 28	27 20	19 18	17 16	15 12	11 8	7 0
c	$5B_H$	-	-	b	a	$0B_H$

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

Examples

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

See Also

MAX.H, **MAX.HU**, **MIN.B**, **MIN.BU**, **MIN.H**, **MIN.HU**

MAX.H
Maximum Value Packed Half-word
MAX.HU
Maximum Value Packed Half-word Unsigned

Description

Compute the maximum value of the corresponding half-words in D[a] and D[b] and put each result in the corresponding half-word of D[c]. The operands are treated as either signed (MAX.H) or unsigned (MAX.HU), 16-bit integers.

MAX.HD[c], D[a], D[b] (RR)

31 28	27 20	19 18	17 16	15 12	11 8	7 0
c	$7A_H$	-	-	b	a	$0B_H$

D[c][31:16] = (D[a][31:16] > D[b][31:16]) ? D[a][31:16] : D[b][31:16];

D[c][15:0] = (D[a][15:0] > D[b][15:0]) ? D[a][15:0] : D[b][15:0];

MAX.HUD[c], D[a], D[b] (RR)

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

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

Examples

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

See Also

MAX.B, **MAX.BU**, **MIN.B**, **MIN.BU**, **MIN.H**, **MIN.HU**

MFCR
Move From Core Register

Description

Move the contents of the Core Special Function Register (CSFR), selected by the value const16, to data register D[c]. The CSFR address is a const16 byte offset from the CSFR base address. It must be word-aligned (the least-significant two bits equal zero). Nonaligned addresses have an undefined effect.

MFCR can be executed on any privilege level. This instruction may not be used to access GPRs. Attempting to access a GPR with this instruction will return an undefined value.

MFCRD[c], const16 (RLC)

31 28	27 12	11 8	7 0
c	const16	-	$4D_H$

D[c] = CR[const16];

Status Flags

C	Read by the instruction but not changed.
V	Read by the instruction but not changed.
SV	Read by the instruction but not changed.
AV	Read by the instruction but not changed.
SAV	Read by the instruction but not changed.

Examples

```
mfcr    d3, fe04H
```

See Also

MTCR

MIN
Minimum Value
MIN.U
Minimum Value Unsigned

Description

If the contents of data register D[a] are less than the contents of either data register D[b] (instruction format RR) or const9 (instruction format RC), then put the contents of D[a] in data register D[c]; otherwise put the contents of either D[b] (format RR) or const9 (format RC) in to D[c]. The operands are treated as either signed (MIN) or unsigned (MIN.U) 32-bit integers.

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

31 28	27 21	20 12	11 8	7 0
c	18_H	const9	a	$8B_H$

D[c] = (D[a] < sign_ext(const9)) ? D[a] : sign_ext(const9);

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

31 28	27 20	19 18	17 16	15 12	11 8	7 0
c	18_H	-	-	b	a	$0B_H$

D[c] = (D[a] < D[b]) ? D[a] : D[b];

MIN.UD[c], D[a], const9 (RC)

31 28	27 21	20 12	11 8	7 0
c	19_H	const9	a	$8B_H$

D[c] = (D[a] < zero_ext(const9)) ? D[a] : zero_ext(const9); // unsigned

MIN.UD[c], D[a], D[b] (RR)

31 28	27 20	19 18	17 16	15 12	11 8	7 0
c	19_H	-	-	b	a	$0B_H$

D[c] = (D[a] < D[b]) ? D[a] : D[b]; // unsigned

Status Flags

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

Examples

```
min    d3, d1, d2
min    d3, d1, #126
```

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

See Also

MAX, **MAX.U**

MIN.B
Minimum Value Packed Byte
MIN.BU
Minimum Value Packed Byte Unsigned

Description

Compute the minimum value of the corresponding bytes in D[a] and D[b] and put each result in the corresponding byte of D[c]. The operands are treated as either signed (MIN.B) or unsigned (MIN.BU), 8-bit integers.

MIN.BD[c], D[a], D[b] (RR)

31 28	27 20	19 18	17 16	15 12	11 8	7 0
c	58_H	-	-	b	a	$0B_H$

D[c][31:24] = (D[a][31:24] < D[b][31:24]) ? D[a][31:24] : D[b][31:24];

D[c][23:16] = (D[a][23:16] < D[b][23:16]) ? D[a][23:16] : D[b][23:16];

D[c][15:8] = (D[a][15:8] < D[b][15:8]) ? D[a][15:8] : D[b][15:8];

D[c][7:0] = (D[a][7:0] < D[b][7:0]) ? D[a][7:0] : D[b][7:0];

MIN.BUD[c], D[a], D[b] (RR)

31 28	27 20	19 18	17 16	15 12	11 8	7 0
c	59_H	-	-	b	a	$0B_H$

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

Examples

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

See Also

MAX.B, **MAX.BU**, **MAX.H**, **MAX.HU**, **MIN.H**, **MIN.HU**

MIN.H
Minimum Value Packed Half-word
MIN.HU
Minimum Value Packed Half-word Unsigned

Description

Compute the minimum value of the corresponding half-words in D[a] and D[b] and put each result in the corresponding half-word of D[c]. The operands are treated as either signed (MIN.H) or unsigned (MIN.HU), 16-bit integers.

MIN.HD[c], D[a], D[b] (RR)

31 28	27 20	19 18	17 16	15 12	11 8	7 0
c	78_H	-	-	b	a	$0B_H$

D[c][31:16] = (D[a][31:16] < D[b][31:16]) ? D[a][31:16] : D[b][31:16];

D[c][15:0] = (D[a][15:0] < D[b][15:0]) ? D[a][15:0] : D[b][15:0];

MIN.HUD[c], D[a], D[b] (RR)

31 28	27 20	19 18	17 16	15 12	11 8	7 0
c	79_H	-	-	b	a	$0B_H$

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

Examples

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

See Also

MAX.B, **MAX.BU**, **MAX.H**, **MAX.HU**, **MIN.B**, **MIN.BU**

MOV
Move

Description

Move the contents of either data register D[b] (instruction format RR) or const16 (instruction format RLC), to data register D[c]. The value const16 is sign-extended to 32-bits before it is moved.

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-bits. 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.

MOVD[c], const16 (RLC)

31 28	27 12	11 8	7 0
c	const16	-	$3B_H$

D[c] = sign_ext(const16);

MOVE[c], const16 (RLC)

31 28	27 12	11 8	7 0
c	const16	-	FB_H

E[c] = sign_ext(const16);

MOVD[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];

MOVE[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]);

MOVE[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]};

MOVD[15], const8 (SC)

15 8	7 0
const8	DA_H

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

MOVD[a], const4 (SRC)

15 12	11 8	7 0
const4	a	82_H

```
D[a] = sign_ext(const4);
```

MOVE[a], const4 (SRC)

15 12	11 8	7 0
const4	a	$D2_H$

```
E[a] = sign_ext(const4);
```

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

15 12	11 8	7 0
b	a	02_H

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

Status Flags

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

Examples

```
mov   d3, d1
mov   d3, #-30000
mov   e0, d5; e0 = sign_ext(d5)
mov   e0, d6, d3; e0 = d6_d3 or d1=d6 d0=d3
mov   e0, 1234H; e0 = 0000000000001234H
```

```
mov   d1, d2
mov   d1, #6
mov   d15, #126
mov   e0, #EH; e0 = FFFFFFFFFFFFFFFEH
```

See Also

MAX, **MAX.U**, **MOV.U**, **MOVH**

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.AA[c], D[b] (RR)

31 28	27 20	19 18	17 16	15 12	11 8	7 0
c	63_H	-	-	b	-	01_H

A[c] = D[b];

MOV.AA[a], const4 (SRC)

15 12	11 8	7 0
const4	a	$A0_H$

A[a] = zero_ext(const4);

MOV.AA[a], D[b] (SRR)

15 12	11 8	7 0
b	a	60_H

A[a] = D[b];

Status Flags

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

Examples

```
mov.a    a3, d1
```

```
mov.a    a4, d2
mov.a    a4, 7
```

See Also

LEA, MOV.AA, MOV.D, MOVH.A

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.AAA[c], A[b] (RR)

31 28	27 20	19 18	17 16	15 12	11 8	7 0
c	00_H	-	-	b	-	01_H

A[c] = A[b];

MOV.AAA[a], A[b] (SRR)

15 12	11 8	7 0
b	a	40_H

A[a] = A[b];

Status Flags

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

Examples

```
mov.aa    a3, a4
```

```
mov.aa    a4, a2
```

See Also

LEA, **MOVH.A**, **MOV.D**, **MOVH.A**

MOV.D
Move Address to Data Register

Description

Move the contents of address register A[b] to data register D[c].

Move the contents of address register A[b] to data register D[a].

MOV.DD[c], A[b] (RR)

31 28	27 20	19	18 17	16 15 12	11 8	7 0
c	$4C_H$	-	-	b	-	01_H

D[c] = A[b];

MOV.DD[a], A[b] (SRR)

15 12	11 8	7 0
b	a	80_H

D[a] = A[b];

Status Flags

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

Examples

```
mov.d   d3, a4
```

```
mov.d   d1, a2
```

See Also

LEA, **MOV.A**, **MOV.AA**, **MOVH.A**

MOV.U
Move Unsigned

Description

Move the zero-extended value const16 to data register D[c].

MOV.UD[c], const16 (RLC)

31 28	27 12	11 8	7 0
c	const16	-	BB$_H$

D[c] = zero_ext(const16);

Status Flags

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

Examples

```
mov.u   d3, #526
```

See Also

MOV, **MOVH**

MOVH
Move High

Description

Move the value const16 to the most-significant half-word of data register D[c] and set the least-significant 16-bits to zero.

MOVHD[c], const16 (RLC)

31 28	27 12	11 8	7 0
c	const16	-	$7B_H$

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**

MOVH.A
Move High to Address

Description

Move the value const16 to the most-significant half-word of address register A[c] and set the least-significant 16-bits to zero.

MOVH.AA[c], const16 (RLC)

31 28	27 12	11 8	7 0
c	const16	-	91_H

A[c] = {const16, 16’h0000};

Status Flags

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

Examples

```
movh.a    a3, #526
```

See Also

LEA, **MOV.A**, **MOV.AA**, **MOV.D**

MSUB
Multiply-Subtract
MSUBS
Multiply-Subtract, Saturated

Description

Multiply two signed 32-bit integers. Subtract the product from a signed 32-bit or 64-bit integer and put the result into a 32-bit or 64-bit register. The value const9 is sign-extended before the multiplication is performed. The MSUBS result is saturated on overflow.

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

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

31 28	27 24	23 21	20 12	11 8	7 0
c	d	01_H	const9	a	33_H

result = D[d] - (D[a] * sign_ext(const9));

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

MSUBE[c], E[d], D[a], const9 (RCR)

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

31 28	27 24	23 21	20 12	11 8	7 0
c	d	03_H	const9	a	33_H

result = E[d] - (D[a] * sign_ext(const9));

E[c] = result[63:0];

MSUBD[c], D[d], D[a], D[b] (RRR2)

32 - (32 * 32) --> 32 signed

31 28	27 24	23 16	15 12	11 8	7 0
c	d	$0A_H$	b	a	23_H

result = D[d] - (D[a] * D[b]);

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

MSUBE[c], E[d], D[a], D[b] (RRR2)

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

31 28	27 24	23 16	15 12	11 8	7 0
c	d	$6A_H$	b	a	23_H

result = E[d] - (D[a] * D[b]);

E[c] = result[63:0];

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

32 - (32 * K9) --> 32 signed saturated

31 28	27 24	23 21	20 12	11 8	7 0
c	d	05_H	const9	a	33_H

result = D[d] - (D[a] * sign_ext(const9));

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

MSUBSE[c], E[d], D[a], const9 (RCR)

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

31 28	27 24	23 21	20 12	11 8	7 0
c	d	07_H	const9	a	33_H

result = E[d] - (D[a] * sign_ext(const9));

E[c] = ssov(result, 64);

MSUBSD[c], D[d], D[a], D[b] (RRR2)

32 - (32 * 32) --> 32 signed saturated

31 28	27 24	23 16	15 12	11 8	7 0
c	d	$8A_H$	b	a	23_H

result = D[d] - (D[a] * D[b]);

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

MSUBSE[c], E[d], D[a], D[b] (RRR2)

64 - (32 * 32) --> 64 signed saturated

31 28	27 24	23 16	15 12	11 8	7 0
c	d	EA_H	b	a	23_H

result = E[d] - (D[a] * D[b]);

E[c] = ssov(result, 64)

Status Flags

C	Not set by these instructions.

V	32-bit result:
	overflow = (result > $7FFFFFFF_H$) 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, d3
msub   d0, d1, d2, #7
msub   e0, e2, d6, d11
msub   e0, e0, d3, #80
msubs   d5, d1, d2, d2
msubs   d1, d1, d2, #7
msubs   e0, e2, d6, d11
msubs   e8, e4, d3, #80
```

See Also

MUL

MSUB.H

Packed Multiply-Subtract Q Format

MSUBS.H

Packed Multiply-Subtract Q Format, Saturated

Description

Multiply two signed 16-bit (half-word) values. Subtract the product (left justified if n == 1) from a signed 32-bit value and put the result into a 32-bit register. There are four cases of half-word multiplication.

Note that n should only take the values 0 or 1, any other value returns an undefined result. If (n == 1) then 8000_H * 8000_H = $7FFFFFFF_H$ (for signed 16-bit * 16-bit multiplications only).

MSUB.HE[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	$A3_H$

sc1 = (D[a][31:16] == 8000_H) AND (D[b][15:0] == 8000_H) AND (n == 1);

sc0 = (D[a][15:0] == 8000_H) AND (D[b][15:0] == 8000_H) AND (n == 1);

mul_res1 = sc1 ? $7FFFFFFF_H$: ((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] = {result_word1[31:0], result_word0[31:0]}; // Packed fraction

MSUB.HE[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	$A3_H$

sc1 = (D[a][31:16] == 8000_H) AND (D[b][15:0] == 8000_H) AND (n == 1);

sc0 = (D[a][15:0] == 8000_H) AND (D[b][31:16] == 8000_H) AND (n == 1);

mul_res1 = sc1 ? $7FFFFFFF_H$: ((D[a][31:16] * D[b][15:0]) << n);

mul_res0 = sc0 ? $7FFFFFFF_H$: ((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.HE[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	$A3_H$

sc1 = (D[a][31:16] == 8000_H) AND (D[b][31:16] == 8000_H) AND (n == 1);

sc0 = (D[a][15:0] == 8000_H) AND (D[b][15:0] == 8000_H) AND (n == 1);

mul_res1 = sc1 ? $7FFFFFFF_H$: ((D[a][31:16] * D[b][31:16]) << 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] = {result_word1[31:0], result_word0[31:0]}; // Packed fraction

MSUB.HE[c], E[d], D[a], D[b] UU, n (RRR1)

32||32 -||- (16L * 16U || 16U * 16U) --> 32||32

31 28	27 24	23 18	17 16	15 12	11 8	7 0
c	d	$1B_H$	n	b	a	$A3_H$

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);

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] = {result_word1[31:0], result_word0[31:0]}; // Packed fraction

MSUBS.HE[c], E[d], D[a], D[b] LL, n (RRR1)

32||32 -||- (16U * 16L || 16L * 16L) --> 32||32 saturated

31 28	27 24	23 18	17 16	15 12	11 8	7 0
c	d	$3A_H$	n	b	a	$A3_H$

sc1 = (D[a][31:16] == 8000_H) AND (D[b][15:0] == 8000_H) AND (n == 1);

sc0 = (D[a][15:0] == 8000_H) AND (D[b][15:0] == 8000_H) AND (n == 1);

mul_res1 = sc1 ? $7FFFFFFF_H$: ((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

MSUBS.HE[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	$A3_H$

sc1 = (D[a][31:16] == 8000_H) AND (D[b][15:0] == 8000_H) AND (n == 1);

sc0 = (D[a][15:0] == 8000_H) AND (D[b][31:16] == 8000_H) AND (n == 1);

mul_res1 = sc1 ? $7FFFFFFF_H$: ((D[a][31:16] * D[b][15:0]) << n);

mul_res0 = sc0 ? $7FFFFFFF_H$: ((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.HE[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	$A3_H$

sc1 = (D[a][31:16] == 8000_H) AND (D[b][31:16] == 8000_H) AND (n == 1);

sc0 = (D[a][15:0] == 8000_H) AND (D[b][15:0] == 8000_H) AND (n == 1);

mul_res1 = sc1 ? $7FFFFFFF_H$: ((D[a][31:16] * D[b][31:16]) << 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

MSUBS.HE[c], E[d], D[a], D[b] UU, n (RRR1)

32||32 -||- (16L * 16U || 16U * 16U) --> 32||32 saturated

31 28	27 24	23 18	17 16	15 12	11 8	7 0
c	d	$3B_H$	n	b	a	$A3_H$

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);

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 these instructions.

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

–

See Also

-

MSUB.Q
Multiply-Subtract Q Format
MSUBS.Q
Multiply-Subtract Q Format, Saturated

Description

Multiply two signed 16-bit or 32-bit values, subtract the product (left justified if n == 1) from a signed 32-bit or 64-bit value and put the result into a 32-bit or 64-bit register.

There are eight cases of 16*16 operations, eight cases of 16*32 operations and four cases of 32*32 operations.

The MSUBS.Q result is saturated on overflow.

Note that n should only take the values 0 or 1, any other value returns an undefined result. If (n == 1) then 8000_H * 8000_H = $7FFFFFFF_H$ (for signed 16-bit * 16-bit multiplications only).

MSUB.QD[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.QE[c], E[d], D[a], D[b], n (RRR1)

64 - (32 * 32) --> 64

31 28	27 24	23 18	17 16	15 12	11 8	7 0
c	d	$1B_H$	n	b	a	63_H

result = E[d] - ((D[a] * D[b]) << n);

E[c] = result[63:0]; // Multi-precision fraction

MSUB.QD[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.QE[c], E[d], D[a], D[b] L, n (RRR1)

64 - (32 * 16L) --> 64

31 28	27 24	23 18	17 16	15 12	11 8	7 0
c	d	19_H	n	b	a	63_H

result = E[d] - ((D[a] * D[b][15:0]) << n);

E[c] = result[63:0]; // Multi-precision accumulator

MSUB.QD[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.QE[c], E[d], D[a], D[b] U, n (RRR1)

64 - (32 * 16U) --> 64

31 28	27 24	23 18	17 16	15 12	11 8	7 0
c	d	18_H	n	b	a	63_H

result = E[d] - ((D[a] * D[b][31:16]) << n);

E[c] = result[63:0]; // Multi-precision accumulator

MSUB.QD[c], D[d], D[a] L, D[b] L, n (RRR1)

32 - (16L * 16L) --> 32

31 28	27 24	23 18	17 16	15 12	11 8	7 0
c	d	05_H	n	b	a	63_H

sc = (D[a][15:0] == 8000_H) AND (D[b][15:0] == 8000_H) AND (n == 1);

mul_res = sc ? $7FFFFFFF_H$: ((D[a][15:0] * D[b][15:0]) << n);

result = D[d] - mul_res;

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

MSUB.QE[c], E[d], D[a] L, D[b] L, n (RRR1)

64 - (16L * 16L) --> 64

31 28	27 24	23 18	17 16	15 12	11 8	7 0
c	d	$1D_H$	n	b	a	63_H

sc = (D[a][15:0] == 8000_H) AND (D[b][15:0] == 8000_H) AND (n == 1);

mul_res = sc ? $7FFFFFFF_H$: ((D[a][15:0] * D[b][15:0]) << n);

result = E[d] - (mul_res << 16);

E[c] = result[63:0]; // Multi-precision accumulator

MSUB.QD[c], D[d], D[a] U, D[b] U, n (RRR1)

32 - (16U * 16U) --> 32

31 28	27 24	23 18	17 16	15 12	11 8	7 0
c	d	04_H	n	b	a	63_H

sc = (D[a][31:16] == 8000_H) AND (D[b][31:16] == 8000_H) AND (n == 1);

mul_res = sc ? $7FFFFFFF_H$: ((D[a][31:16] * D[b][31:16]) << n);

result = D[d] - mul_res;

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

MSUB.QE[c], E[d], D[a] U, D[b] U, n (RRR1)

64 - (16U * 16U) --> 64

31 28	27 24	23 18	17 16	15 12	11 8	7 0
c	d	$1C_H$	n	b	a	63_H

sc = (D[a][31:16] == 8000_H) AND (D[b][31:16] == 8000_H) AND (n == 1);

mul_res = sc ? $7FFFFFFF_H$: ((D[a][31:16] * D[b][31:16]) << n);

result = E[d] - (mul_res << 16);

E[c] = result[63:0]; // Multi-precision accumulator

MSUBS.QD[c], D[d], D[a], D[b], n (RRR1)

32 - (32 * 32)Up --> 32 saturated

31 28	27 24	23 18	17 16	15 12	11 8	7 0
c	d	22_H	n	b	a	63_H

result = D[d] - (((D[a] * D[b]) << n) >> 32);

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

MSUBS.QE[c], E[d], D[a], D[b], n (RRR1)

64 - (32 * 32) --> 64 saturated

31 28	27 24	23 18	17 16	15 12	11 8	7 0
c	d	$3B_H$	n	b	a	63_H

result = E[d] - ((D[a] * D[b]) << n);

E[c] = ssov(result, 64); // Multi-precision fraction

MSUBS.QD[c], D[d], D[a], D[b] L, n (RRR1)

32 - (32 * 16L)Up --> 32 saturated

31 28	27 24	23 18	17 16	15 12	11 8	7 0
c	d	21_H	n	b	a	63_H

result = D[d] - (((D[a] * D[b][15:0]) << n) >> 16);

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

MSUBS.QE[c], E[d], D[a], D[b] L, n (RRR1)

64 - (32 * 16L) --> 64 saturated

31 28	27 24	23 18	17 16	15 12	11 8	7 0
c	d	39_H	n	b	a	63_H

result = E[d] - ((D[a] * D[b][15:0]) << n);

E[c] = ssov(result, 64); // Multi-precision accumulator

MSUBS.QD[c], D[d], D[a], D[b] U, n (RRR1)

32 - (32 * 16U)Up --> 32 saturated

31 28	27 24	23 18	17 16	15 12	11 8	7 0
c	d	20_H	n	b	a	63_H

result = D[d] - (((D[a] * D[b][31:16]) << n) >> 16);

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

MSUBS.QE[c], E[d], D[a], D[b] U, n (RRR1)

64 - (32 * 16U) --> 64 saturated

31 28	27 24	23 18	17 16	15 12	11 8	7 0
c	d	38_H	n	b	a	63_H

result = E[d] - ((D[a] * D[b][31:16]) << n);

E[c] = ssov(result, 64); // Multi-precision accumulator

MSUBS.QD[c], D[d], D[a] L, D[b] L, n (RRR1)

32 - (16L * 16L) --> 32 saturated

31 28	27 24	23 18	17 16	15 12	11 8	7 0
c	d	25_H	n	b	a	63_H

sc = (D[a][15:0] == 8000_H) AND (D[b][15:0] == 8000_H) AND (n == 1);

mul_res = sc ? $7FFFFFFF_H$: ((D[a][15:0] * D[b][15:0]) << n);

result = D[d] - mul_res;

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

MSUBS.QE[c], E[d], D[a] L, D[b] L, n (RRR1)

64 - (16L * 16L) --> 64 saturated

31 28	27 24	23 18	17 16	15 12	11 8	7 0
c	d	$3D_H$	n	b	a	63_H

sc = (D[a][15:0] == 8000_H) AND (D[b][15:0] == 8000_H) AND (n == 1);

mul_res = sc ? $7FFFFFFF_H$: ((D[a][15:0] * D[b][15:0]) << n);

result = E[d] - (mul_res << 16);

E[c] = ssov(result, 64); // Multi-precision accumulator

MSUBS.QD[c], D[d], D[a] U, D[b] U, n (RRR1)

32 - (16U * 16U) --> 32 saturated

31 28	27 24	23 18	17 16	15 12	11 8	7 0
c	d	24_H	n	b	a	63_H

sc = (D[a][31:16] == 8000_H) AND (D[b][31:16] == 8000_H) AND (n == 1);

mul_res = sc ? $7FFFFFFF_H$: ((D[a][31:16] * D[b][31:16]) << n);

result = D[d] - mul_res;

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

MSUBS.QE[c], E[d], D[a] U, D[b] U, n (RRR1)

64 - (16U * 16U) --> 64 saturated

31 28	27 24	23 18	17 16	15 12	11 8	7 0
c	d	$3C_H$	n	b	a	63_H

sc = (D[a][31:16] == 8000_H) AND (D[b][31:16] == 8000_H) AND (n == 1);

mul_res = sc ? $7FFFFFFF_H$: ((D[a][31:16] * D[b][31:16]) << n);

result = E[d] - (mul_res << 16);

E[c] = ssov(result, 64); // Multi-precision accumulator

Status Flags

C	Not set by these instructions.

V	32-bit result:
	overflow = (result > $7FFFFFFF_H$) 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, d3, #1
msub.q   d0, d1, d2, d6U, #1
msub.q   d0, d2, d1, d3L, #1
msub.q   d2, d0, d3U, d4U, #1
msub.q   d2, d0, d4L, d4L, #1
msub.q   e2, e2, d3, d7, #1
msub.q   e2, e2, d4, d6U, #1
msub.q   e2, e2, d5, d6L, #1
msub.q   e2, e2, d6U, d7U, #1
msub.q   e2, e2, d8L, d0L, #1
msubs.q   d0, d1, d2, d3, #1
msubs.q   d0, d1, d2, d6U, #1
msubs.q   d0, d2, d1, d3L, #1
msubs.q   d2, d0, d3U, d4U, #1
msubs.q   d2, d0, d4L, d4L, #1
msubs.q   e2, e2, d3, d7, #1
msubs.q   e2, e2, d4, d6U, #1
msubs.q   e2, e2, d5, d6L, #1
msubs.q   e2, e2, d6U, d7U, #1
msubs.q   e2, e0, d11L, d4L, #1
```

See Also

-

MSUB.U

Multiply-Subtract Unsigned

MSUBS.U

Multiply-Subtract Unsigned, Saturated

Description

Multiply two unsigned 32-bit integers. Subtract the product from an unsigned 32-bit or 64-bit integer and put the result into a 32-bit or 64-bit register. The value const9 is zero-extended before the multiplication is performed. The MSUBS.U results are saturated on overflow.

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

64 - (32 * K9) --> 64 unsigned

31 28	27 24	23 21	20 12	11 8	7 0
c	d	02_H	const9	a	33_H

result = E[d] - (D[a] * zero_ext(const9)); // unsigned operators

E[c] = result[63:0];

MSUB.UE[c], E[d], D[a], D[b] (RRR2)

64 - (32 * 32) --> 64 unsigned

31 28	27 24	23 16	15 12	11 8	7 0
c	d	68_H	b	a	23_H

result = E[d] - (D[a] * D[b]); // unsigned operators

E[c] = result[63:0];

MSUBS.UD[c], D[d], D[a], const9 (RCR)

32 - (32 * K9) --> 32 unsigned saturated

31 28	27 24	23 21	20 12	11 8	7 0
c	d	04_H	const9	a	33_H

result = D[d] - (D[a] * zero_ext(const9)); // unsigned operators

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

MSUBS.UE[c], E[d], D[a], const9 (RCR)

64 - (32 * K9) --> 64 unsigned saturated

31 28	27 24	23 21	20 12	11 8	7 0
c	d	06_H	const9	a	33_H

result = E[d] - (D[a] * zero_ext(const9)); // unsigned operators

E[c] = suov(result, 64);

MSUBS.UD[c], D[d], D[a], D[b] (RRR2)

32 - (32 * 32) --> 32 unsigned saturated

31 28	27 24	23 16	15 12	11 8	7 0
c	d	88_H	b	a	23_H

result = D[d] - (D[a] * D[b]); // unsigned operators

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

MSUBS.UE[c], E[d], D[a], D[b] (RRR2)

64 - (32 * 32) --> 64 unsigned saturated

31 28	27 24	23 16	15 12	11 8	7 0
c	d	$E8_H$	b	a	23_H

result = E[d] - (D[a] * D[b]); // unsigned operators

E[c] = suov(result, 64);

Status Flags

C	Not set by these instructions.

V	32-bit result:
	overflow = (result > $FFFFFFFF_H$) 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, e2, d6, d11
msub.u   e0, e0, d3, #80
msubs.u   d5, d1, d2, d2
msubs.u   d1, d1, d2, #7
msubs.u   e0, e2, d6, d11
msubs.u   e8, e4, d3, #80
```

See Also

-

MSUBAD.H

Packed Multiply-Subtract/Add Q Format

MSUBADS.H

Packed Multiply-Subtract/Add Q Format, Saturated

Description

Multiply two signed 16-bit (half-word) values. Subtract (or add) the product (left justified if n == 1) from a signed 32-bit value and put the result into a 32-bit register. There are four cases of half-word multiplication:

- 16U * 16U, 16L * 16L
- 16U * 16L, 16L * 16U
- 16U * 16L, 16L * 16L
- 16L * 16U, 16U * 16U

On overflow each MSUBADS.H result is independently saturated.

Note that n should only take the values 0 or 1, any other value returns an undefined result. If (n == 1) then 8000_H * 8000_H = $7FFFFFFF_H$ (for signed 16-bit * 16-bit multiplications only).

MSUBAD.HE[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	$E3_H$

sc1 = (D[a][31:16] == 8000_H) AND (D[b][15:0] == 8000_H) AND (n == 1);

sc0 = (D[a][15:0] == 8000_H) AND (D[b][15:0] == 8000_H) AND (n == 1);

mul_res1 = sc1 ? $7FFFFFFF_H$: ((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] = {result_word1[31:0], result_word0[31:0]}; // Packed fraction

MSUBAD.HE[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	$E3_H$

sc1 = (D[a][31:16] == 8000_H) AND (D[b][15:0] == 8000_H) AND (n == 1);

sc0 = (D[a][15:0] == 8000_H) AND (D[b][31:16] == 8000_H) AND (n == 1);

mul_res1 = sc1 ? $7FFFFFFF_H$: ((D[a][31:16] * D[b][15:0]) << n);

mul_res0 = sc0 ? $7FFFFFFF_H$: ((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.HE[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	$E3_H$

sc1 = (D[a][31:16] == 8000_H) AND (D[b][31:16] == 8000_H) AND (n == 1);

sc0 = (D[a][15:0] == 8000_H) AND (D[b][15:0] == 8000_H) AND (n == 1);

mul_res1 = sc1 ? $7FFFFFFF_H$: ((D[a][31:16] * D[b][31:16]) << 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] = {result_word1[31:0], result_word0[31:0]}; // Packed fraction

MSUBAD.HE[c], E[d], D[a], D[b] UU, n (RRR1)

32||32 -||+ (16L * 16U || 16U * 16U) --> 32||32

31 28	27 24	23 18	17 16	15 12	11 8	7 0
c	d	$1B_H$	n	b	a	$E3_H$

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);

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] = {result_word1[31:0], result_word0[31:0]}; // Packed fraction

MSUBADS.HE[c], E[d], D[a], D[b] LL, n (RRR1)

32||32 -||+ (16U * 16L || 16L * 16L) --> 32||32 saturated

31 28	27 24	23 18	17 16	15 12	11 8	7 0
c	d	$3A_H$	n	b	a	$E3_H$

sc1 = (D[a][31:16] == 8000_H) AND (D[b][15:0] == 8000_H) AND (n == 1);

sc0 = (D[a][15:0] == 8000_H) AND (D[b][15:0] == 8000_H) AND (n == 1);

mul_res1 = sc1 ? $7FFFFFFF_H$: ((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

MSUBADS.HE[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] == 8000_H) AND (D[b][15:0] == 8000_H) AND (n == 1);

sc0 = (D[a][15:0] == 8000_H) AND (D[b][31:16] == 8000_H) AND (n == 1);

mul_res1 = sc1 ? $7FFFFFFF_H$: ((D[a][31:16] * D[b][15:0]) << n);

mul_res0 = sc0 ? $7FFFFFFF_H$: ((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.HE[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] == 8000_H) AND (D[b][31:16] == 8000_H) AND (n == 1);

sc0 = (D[a][15:0] == 8000_H) AND (D[b][15:0] == 8000_H) AND (n == 1);

mul_res1 = sc1 ? $7FFFFFFF_H$: ((D[a][31:16] * D[b][31:16]) << 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

MSUBADS.HE[c], E[d], D[a], D[b] UU, n (RRR1)

32||32 -||+ (16L * 16U || 16U * 16U) --> 32||32 saturated

31 28	27 24	23 18	17 16	15 12	11 8	7 0
c	d	$3B_H$	n	b	a	$E3_H$

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);

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 these instructions.

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

–

See Also

-

MSUBADM.H
Packed Multiply-Subtract/Add Q Format-Multi-precision
MSUBADMS.H
Packed Multiply-Subtract/Add Q Format-Multi-precision, Saturated

Description

Perform two multiplications of two signed 16-bit (half-word) values. Subtract one product and add the other product (left justified if n == 1) left-shifted by 16, from/to a signed 64-bit value and put the result in a 64-bit register. There are four cases of half-word multiplication:

- 16U * 16U, 16L * 16L
- 16U * 16L, 16L * 16U
- 16U * 16L, 16L * 16L
- 16L * 16U, 16U * 16U

On overflow the MSUBADMS.H result is saturated.

Note that n should only take the values 0 or 1, any other value returns an undefined result. If (n == 1) then 8000_H * 8000_H = $7FFFFFFF_H$ (for signed 16-bit * 16-bit multiplications only).

MSUBADM.HE[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	$E3_H$

sc1 = (D[a][31:16] == 8000_H) AND (D[b][15:0] == 8000_H) AND (n == 1);
sc0 = (D[a][15:0] == 8000_H) AND (D[b][15:0] == 8000_H) 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 = E[d] - ((result_word1 - result_word0) << 16);
E[c] = result[63:0]; // Multi-precision accumulator

MSUBADM.HE[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	$E3_H$

sc1 = (D[a][31:16] == 8000_H) AND (D[b][15:0] == 8000_H) AND (n == 1);
sc0 = (D[a][15:0] == 8000_H) AND (D[b][31:16] == 8000_H) 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][31:16]) << n);
result = E[d] - ((result_word1 - result_word0) << 16);
E[c] = result[63:0]; // Multi-precision accumulator

MSUBADM.HE[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	$E3_H$

sc1 = (D[a][31:16] == 8000_H) AND (D[b][31:16] == 8000_H) AND (n == 1);

sc0 = (D[a][15:0] == 8000_H) AND (D[b][15:0] == 8000_H) AND (n == 1);

result_word1 = sc1 ? $7FFFFFFF_H$: ((D[a][31:16] * D[b][31:16]) << n);

result_word0 = sc0 ? $7FFFFFFF_H$: ((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.HE[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	$E3_H$

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 = E[d] - ((result_word1 - result_word0) << 16);

E[c] = result[63:0]; // Multi-precision accumulator

MSUBADMS.HE[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] == 8000_H) AND (D[b][15:0] == 8000_H) AND (n == 1);

sc0 = (D[a][15:0] == 8000_H) AND (D[b][15:0] == 8000_H) 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 = E[d] - ((result_word1 - result_word0) << 16);

E[c] = ssov(result, 64); // Multi-precision accumulator

MSUBADMS.HE[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] == 8000_H) AND (D[b][15:0] == 8000_H) AND (n == 1);

sc0 = (D[a][15:0] == 8000_H) AND (D[b][31:16] == 8000_H) 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][31:16]) << n);

result = E[d] - ((result_word1 - result_word0) << 16);

E[c] = ssov(result, 64); // Multi-precision accumulator

MSUBADMS.H E[c], E[d], D[a], D[b] UL, n (RRR1)
64 - (16U * 16U) + (16L * 16L) --> 64 saturated

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] == 8000_H) AND (D[b][31:16] == 8000_H) AND (n == 1);
sc0 = (D[a][15:0] == 8000_H) AND (D[b][15:0] == 8000_H) AND (n == 1);
result_word1 = sc1 ? $7FFFFFFF_H$: ((D[a][31:16] * D[b][31:16]) << n);
result_word0 = sc0 ? $7FFFFFFF_H$: ((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.HE[c], E[d], D[a], D[b] UU, n (RRR1)
64 - (16L * 16U) + (16U * 16U) --> 64 saturated

31 28	27 24	23 18	17 16	15 12	11 8	7 0
c	d	$3F_H$	n	b	a	$E3_H$

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 = E[d] - ((result_word1 - result_word0) << 16);
E[c] = ssov(result, 64); // Multi-precision accumulator

Status Flags

C	Not set by these instructions.

V	overflow = (result > $7FFFFFFFFFFFFFFF_H$) 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

–

See Also

-

MSUBADR.H

Packed Multiply-Subtract/Add Q Format with Rounding

MSUBADRS.H

Packed Multiply-Subtract/Add Q Format with Rounding, Saturated

Description

Multiply two signed 16-bit (half-word) values. Subtract (or add) the product (left justified if n == 1) from (to) a signed 16-bit value and put the rounded result into half of a 32-bit register (Note that since there are two results the two register halves are used). There are four cases of half-word multiplication:

- 16U * 16U, 16L * 16L
- 16U * 16L, 16L * 16U
- 16U * 16L, 16L * 16L
- 16L * 16U, 16U * 16U

Note that n should only take the values 0 or 1, any other value returns an undefined result. If (n == 1) then 8000_H * 8000_H = $7FFFFFFF_H$ (for signed 16-bit * 16-bit multiplications only).

MSUBADR.HD[c], D[d], D[a], D[b] LL, n (RRR1)

16U || 16L -||+ (16U * 16L || 16L * 16L) rounded --> 16||16

31 28	27 24	23 18	17 16	15 12	11 8	7 0
c	d	$0E_H$	n	b	a	$E3_H$

sc1 = (D[a][31:16] == 8000_H) AND (D[b][15:0] == 8000_H) AND (n == 1);

sc0 = (D[a][15:0] == 8000_H) AND (D[b][15:0]] == 8000_H) AND (n == 1);

mul_res1 = sc1 ? $7FFFFFFF_H$: ((D[a][31:16] * D[b][15:0]) << n);

mul_res0 = sc0 ? $7FFFFFFF_H$: ((D[a][15:0] * D[b][15:0]) << n);

result_halfword1 = {D[d][31:16], 16'b0} - mul_res1 + 8000_H;

result_halfword0 = {D[d][15:0], 16'b0} + mul_res0 + 8000_H;

D[c] = {result_halfword1[31:16], result_halfword0[31:16]}; // Packed short fraction

MSUBADR.HD[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] == 8000_H) AND (D[b][15:0] == 8000_H) AND (n == 1);

sc0 = (D[a][15:0] == 8000_H) AND (D[b][31:16]] == 8000_H) AND (n == 1);

mul_res1 = sc1 ? $7FFFFFFF_H$: ((D[a][31:16] * D[b][15:0]) << n);

mul_res0 = sc0 ? $7FFFFFFF_H$: ((D[a][15:0] * D[b][31:16]) << n);

result_halfword1 = {D[d][31:16], 16'b0} - mul_res1 + 8000_H;

result_halfword0 = {D[d][15:0], 16'b0} + mul_res0 + 8000_H;

D[c] = {result_halfword1[31:16], result_halfword0[31:16]}; // Packed short fraction

MSUBADR.HD[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] == 8000_H) AND (D[b][31:16] == 8000_H) AND (n == 1);
sc0 = (D[a][15:0] == 8000_H) AND (D[b][15:0] == 8000_H) AND (n == 1);
mul_res1 = sc1 ? $7FFFFFFF_H$: ((D[a][31:16] * D[b][31:16]) << n);
mul_res0 = sc0 ? $7FFFFFFF_H$: ((D[a][15:0] * D[b][15:0]) << n);
result_halfword1 = {D[d][31:16], 16'b0} - mul_res1 + 8000_H;
result_halfword0 = {D[d][15:0], 16'b0} + mul_res0 + 8000_H;
D[c] = {result_halfword1[31:16], result_halfword0[31:16]}; // Packed short fraction

MSUBADR.HD[c], D[d], D[a], D[b] UU, n (RRR1)
16U || 16L -||+ (16L * 16U || 16U * 16U) rounded --> 16||16

31 28	27 24	23 18	17 16	15 12	11 8	7 0
c	d	$0F_H$	n	b	a	$E3_H$

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);
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_halfword1 = {D[d][31:16], 16'b0} - mul_res1 + 8000_H;
result_halfword0 = {D[d][15:0], 16'b0} + mul_res0 + 8000_H;
D[c] = {result_halfword1[31:16], result_halfword0[31:16]}; // Packed short fraction

MSUBADRS.HD[c], D[d], D[a], D[b] LL, n (RRR1)
16U || 16L -||+ (16U * 16L || 16L * 16L) rounded --> 16||16 saturated

31 28	27 24	23 18	17 16	15 12	11 8	7 0
c	d	$2E_H$	n	b	a	$E3_H$

sc1 = (D[a][31:16] == 8000_H) AND (D[b][15:0] == 8000_H) AND (n == 1);
sc0 = (D[a][15:0] == 8000_H) AND (D[b][15:0]] == 8000_H) AND (n == 1);
mul_res1 = sc1 ? $7FFFFFFF_H$: ((D[a][31:16] * D[b][15:0]) << n);
mul_res0 = sc0 ? $7FFFFFFF_H$: ((D[a][15:0] * D[b][15:0]) << n);
result_halfword1 = {D[d][31:16], 16'b0} - mul_res1 + 8000_H;
result_halfword0 = {D[d][15:0], 16'b0} + mul_res0 + 8000_H;
D[c] = {ssov(result_halfword1, 32)[31:16], ssov(result_halfword0, 32)[31:16]};
// Packed short fraction result

MSUBADRS.HD[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	$E3_H$

sc1 = (D[a][31:16] == 8000_H) AND (D[b][15:0] == 8000_H) AND (n == 1);

sc0 = (D[a][15:0] == 8000_H) AND (D[b][31:16]] == 8000_H) AND (n == 1);
mul_res1 = sc1 ? $7FFFFFFF_H$: ((D[a][31:16] * D[b][15:0]) << n);
mul_res0 = sc0 ? $7FFFFFFF_H$: ((D[a][15:0] * D[b][31:16]) << n);
result_halfword1 = {D[d][31:16], 16'b0} - mul_res1 + 8000_H;
result_halfword0 = {D[d][15:0], 16'b0} + mul_res0 + 8000_H;
D[c] = {ssov(result_halfword1, 32)[31:16], ssov(result_halfword0, 32)[31:16]};
// Packed short fraction result

MSUBADRS.HD[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	$E3_H$

sc1 = (D[a][31:16] == 8000_H) AND (D[b][31:16] == 8000_H) AND (n == 1);
sc0 = (D[a][15:0] == 8000_H) AND (D[b][15:0] == 8000_H) AND (n == 1);
mul_res1 = sc1 ? $7FFFFFFF_H$: ((D[a][31:16] * D[b][31:16]) << n);
mul_res0 = sc0 ? $7FFFFFFF_H$: ((D[a][15:0] * D[b][15:0]) << n);
result_halfword1 = {D[d][31:16], 16'b0} - mul_res1 + 8000_H;
result_halfword0 = {D[d][15:0], 16'b0} + mul_res0 + 8000_H;
D[c] = {ssov(result_halfword1, 32)[31:16], ssov(result_halfword0, 32)[31:16]};
// Packed short fraction result

MSUBADRS.HD[c], D[d], D[a], D[b] UU, n (RRR1)
16U || 16L -||+ (16L * 16U || 16U * 16U) rounded > 16||16 saturated

31 28	27 24	23 18	17 16	15 12	11 8	7 0
c	d	$2F_H$	n	b	a	$E3_H$

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);
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_halfword1 = {D[d][31:16], 16'b0} - mul_res1 + 8000_H;
result_halfword0 = {D[d][15:0], 16'b0} + mul_res0 + 8000_H;
D[c] = {ssov(result_halfword1, 32)[31:16], ssov(result_halfword0, 32)[31:16]};
// Packed short fraction result

Status Flags

C	Not set by these instructions.

V	ov_halfword1 = (result_halfword1 > $7FFFFFFF_H$) OR (result_halfword1 < -80000000_H);
	ov_halfword0 = (result_halfword0 > $7FFFFFFF_H$) OR (result_halfword0 < -80000000_H);
	overflow = ov_halfword1 OR ov_halfword0;
	if (overflow) then PSW.V = 1 else PSW.V = 0;

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

AV	aov_halfword1 = result_halfword1[31] ^ result_halfword1[30];
	aov_halfword0 = result_halfword0[31] ^ result_halfword0[30];
	advanced_overflow = aov_halfword1 OR aov_halfword0;
	if (advanced_overflow) then PSW.AV = 1 else PSW.AV = 0;

SAV	if (advanced_overflow) then PSW.SAV = 1; else PSW.SAV = PSW.SAV;

Examples

–

See Also

-

MSUBM.H

Packed Multiply-Subtract Q Format-Multi-precision

MSUBMS.H

Packed Multiply-Subtract Q Format-Multi-precision, Saturated

Description

Perform two multiplications of two signed 16-bit (half-word) values. Subtract the two products (left justified if n == 1) left-shifted by 16, from a signed 64-bit value and put the result in a 64-bit register. There are four cases of half-word multiplication:

- 16U * 16U, 16L * 16L
- 16U * 16L, 16L * 16U
- 16U * 16L, 16L * 16L
- 16L * 16U, 16U * 16U

Note that n should only take the values 0 or 1, any other value returns an undefined result. If (n == 1) then 8000_H * 8000_H = $7FFFFFFF_H$ (for signed 16-bit * 16-bit multiplications only).

MSUBM.HE[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] == 8000_H) AND (D[b][15:0] == 8000_H) AND (n == 1);

sc0 = (D[a][15:0] == 8000_H) AND (D[b][15:0] == 8000_H) 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 = E[d] - ((result_word1 + result_word0) << 16);

E[c] = result[63:0]; // Multi-precision accumulator

MSUBM.HE[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] == 8000_H) AND (D[b][15:0] == 8000_H) AND (n == 1);

sc0 = (D[a][15:0] == 8000_H) AND (D[b][31:16] == 8000_H) 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][31:16]) << n);

result = E[d] - ((result_word1 + result_word0) << 16);

E[c] = result[63:0]; // Multi-precision accumulator

MSUBM.HE[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] == 8000_H) AND (D[b][31:16] == 8000_H) AND (n == 1);

sc0 = (D[a][15:0] == 8000_H) AND (D[b][15:0] == 8000_H) AND (n == 1);

result_word1 = sc1 ? $7FFFFFFF_H$: ((D[a][31:16] * D[b][31:16]) << n);

result_word0 = sc0 ? $7FFFFFFF_H$: ((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.HE[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	$A3_H$

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 = E[d] - ((result_word1 + result_word0) << 16);

E[c] = result[63:0]; // Multi-precision accumulator

MSUBMS.HE[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] == 8000_H) AND (D[b][15:0] == 8000_H) AND (n == 1);

sc0 = (D[a][15:0] == 8000_H) AND (D[b][15:0] == 8000_H) 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 = E[d] - ((result_word1 + result_word0) << 16);

E[c] = ssov(result, 64); // Multi-precision accumulator

MSUBMS.HE[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] == 8000_H) AND (D[b][15:0] == 8000_H) AND (n == 1);

sc0 = (D[a][15:0] == 8000_H) AND (D[b][31:16] == 8000_H) 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][31:16]) << n);

result = E[d] - ((result_word1 + result_word0) << 16);

E[c] = ssov(result, 64); // Multi-precision accumulator

MSUBMS.HE[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] == 8000_H) AND (D[b][31:16] == 8000_H) AND (n == 1);
sc0 = (D[a][15:0] == 8000_H) AND (D[b][15:0] == 8000_H) AND (n == 1);
result_word1 = sc1 ? $7FFFFFFF_H$: ((D[a][31:16] * D[b][31:16]) << n);
result_word0 = sc0 ? $7FFFFFFF_H$: ((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.HE[c], E[d], D[a], D[b] UU, n (RRR1)
64 - (16L * 16U) - (16U * 16U) --> 64 saturated

31 28	27 24	23 18	17 16	15 12	11 8	7 0
c	d	$3F_H$	n	b	a	$A3_H$

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 = E[d] - ((result_word1 + result_word0) << 16);
E[c] = ssov(result, 64); // Multi-precision accumulator

Status Flags

C	Not set by these instructions.

V	overflow = (result > $7FFFFFFFFFFFFFFF_H$) 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

–

See Also

-

MSUBR.H

Packed Multiply-Subtract Q Format with Rounding

MSUBRS.H

Packed Multiply-Subtract Q Format with Rounding, Saturated

Description

Multiply two signed 16-bit (half-word) values. Subtract the product (left justified if n == 1) from a signed 16-bit or 32-bit value and put the rounded result into half of a 32-bit register. Note that since there are two results the two register halves are used. There are four cases of half-word multiplication:

- 16U * 16U, 16L * 16L
- 16U * 16L, 16L * 16U
- 16U * 16L, 16L * 16L
- 16L * 16U, 16U * 16U

Note that n should only take the values 0 or 1. Any other value returns an undefined result. If (n == 1) then 8000_H * 8000_H = $7FFFFFFF_H$ (for signed 16-bit * 16-bit multiplications only).

MSUBR.HD[c], D[d], D[a], D[b] LL, n (RRR1)

16U || 16L -||- (16U * 16L || 16L * 16L) rounded --> 16||16

31 28	27 24	23 18	17 16	15 12	11 8	7 0
c	d	$0E_H$	n	b	a	$A3_H$

sc1 = (D[a][31:16] == 8000_H) AND (D[b][15:0] == 8000_H) AND (n == 1);

sc0 = (D[a][15:0] == 8000_H) AND (D[b][15:0] == 8000_H) AND (n == 1);

mul_res1 = sc1 ? $7FFFFFFF_H$: ((D[a][31:16] * D[b][15:0]) << n);

mul_res0 = sc0 ? $7FFFFFFF_H$: ((D[a][15:0] * D[b][15:0]) << n);

result_halfword1 = {D[d][31:16], 16’b0} - mul_res1 + 8000_H;

result_halfword0 = {D[d][15:0], 16’b0} - mul_res0 + 8000_H;

D[c] = {result_halfword1[31:16], result_halfword0[31:16]}; // Packed short fraction

MSUBR.HD[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	$A3_H$

sc1 = (D[a][31:16] == 8000_H) AND (D[b][15:0] == 8000_H) AND (n == 1);

sc0 = (D[a][15:0] == 8000_H) AND (D[b][31:16] == 8000_H) AND (n == 1);

mul_res1 = sc1 ? $7FFFFFFF_H$: ((D[a][31:16] * D[b][15:0]) << n);

mul_res0 = sc0 ? $7FFFFFFF_H$: ((D[a][15:0] * D[b][31:16]) << n);

result_halfword1 = {D[d][31:16], 16’b0} - mul_res1 + 8000_H;

result_halfword0 = {D[d][15:0], 16’b0} - mul_res0 + 8000_H;

D[c] = {result_halfword1[31:16], result_halfword0[31:16]}; // Packed short fraction

MSUBR.HD[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] == 8000_H) AND (D[b][31:16] == 8000_H) AND (n == 1);
sc0 = (D[a][15:0] == 8000_H) AND (D[b][15:0] == 8000_H) AND (n == 1);
mul_res1 = sc1 ? $7FFFFFFF_H$: ((D[a][31:16] * D[b][31:16]) << n);
mul_res0 = sc0 ? $7FFFFFFF_H$: ((D[a][15:0] * D[b][15:0]) << n);
result_halfword1 = {D[d][31:16], 16’b0} - mul_res1 + 8000_H;
result_halfword0 = {D[d][15:0], 16’b0} - mul_res0 + 8000_H;
D[c] = {result_halfword1[31:16], result_halfword0[31:16]}; // Packed short fraction

MSUBR.HD[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] == 8000_H) AND (D[b][31:16] == 8000_H) AND (n == 1);
sc0 = (D[a][15:0] == 8000_H) AND (D[b][15:0] == 8000_H) AND (n == 1);
mul_res1 = sc1 ? $7FFFFFFF_H$: ((D[a][31:16] * D[b][31:16]) << n);
mul_res0 = sc0 ? $7FFFFFFF_H$: ((D[a][15:0] * D[b][15:0]) << n);
result_halfword1 = E[d][63:32] - mul_res1 + 8000_H;
result_halfword0 = E[d][31:0] - mul_res0 + 8000_H;
D[c] = {result_halfword1[31:16], result_halfword0[31:16]}; // Packed short fraction

MSUBR.HD[c], D[d], D[a], D[b] UU, n (RRR1)
16U || 16L -||- (16L * 16U || 16U * 16U) rounded --> 16||16

31 28	27 24	23 18	17 16	15 12	11 8	7 0
c	d	$0F_H$	n	b	a	$A3_H$

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);
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_halfword1 = {D[d][31:16], 16’b0} - mul_res1 + 8000_H;
result_halfword0 = {D[d][15:0], 16’b0} - mul_res0 + 8000_H;
D[c] = {result_halfword1[31:16], result_halfword0[31:16]}; // Packed short fraction

MSUBRS.HD[c], D[d], D[a], D[b] LL, n (RRR1)
16U || 16L -||- (16U * 16L || 16L * 16L) rounded --> 16||16 saturated

31 28	27 24	23 18	17 16	15 12	11 8	7 0
c	d	$2E_H$	n	b	a	$A3_H$

sc1 = (D[a][31:16] == 8000_H) AND (D[b][15:0] == 8000_H) AND (n == 1);
sc0 = (D[a][15:0] == 8000_H) AND (D[b][15:0] == 8000_H) AND (n == 1);

mul_res1 = sc1 ? 7FFFFFFF$_H$: ((D[a][31:16] * D[b][15:0]) << n);

mul_res0 = sc0 ? 7FFFFFFF$_H$: ((D[a][15:0] * D[b][15:0]) << n);

result_halfword1 = {D[d][31:16], 16’b0} - mul_res1 + 8000$_H$;

result_halfword0 = {D[d][15:0], 16’b0} - mul_res0 + 8000$_H$;

D[c] = {ssov(result_halfword1, 32)[31:16], ssov(result_halfword0, 32)[31:16]};

// Packed short fraction result

MSUBRS.HD[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	A3$_H$

sc1 = (D[a][31:16] == 8000$_H$) AND (D[b][15:0] == 8000$_H$) AND (n == 1);

sc0 = (D[a][15:0] == 8000$_H$) AND (D[b][31:16] == 8000$_H$) AND (n == 1);

mul_res1 = sc1 ? 7FFFFFFF$_H$: ((D[a][31:16] * D[b][15:0]) << n);

mul_res0 = sc0 ? 7FFFFFFF$_H$: ((D[a][15:0] * D[b][31:16]) << n);

result_halfword1 = {D[d][31:16], 16’b0} - mul_res1 + 8000$_H$;

result_halfword0 = {D[d][15:0], 16’b0} - mul_res0 + 8000$_H$;

D[c] = {ssov(result_halfword1, 32)[31:16], ssov(result_halfword0, 32)[31:16]};

// Packed short fraction result

MSUBRS.HD[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] == 8000$_H$) AND (D[b][31:16] == 8000$_H$) AND (n == 1);

sc0 = (D[a][15:0] == 8000$_H$) AND (D[b][15:0] == 8000$_H$) AND (n == 1);

mul_res1 = sc1 ? 7FFFFFFF$_H$: ((D[a][31:16] * D[b][31:16]) << n);

mul_res0 = sc0 ? 7FFFFFFF$_H$: ((D[a][15:0] * D[b][15:0]) << n);

result_halfword1 = {D[d][31:16], 16’b0} - mul_res1 + 8000$_H$;

result_halfword0 = {D[d][15:0], 16’b0} - mul_res0 + 8000$_H$;

D[c] = {ssov(result_halfword1, 32)[31:16], ssov(result_halfword0, 32)[31:16]};

// Packed short fraction result

MSUBRS.HD[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] == 8000$_H$) AND (D[b][31:16] == 8000$_H$) AND (n == 1);

sc0 = (D[a][15:0] == 8000$_H$) AND (D[b][15:0] == 8000$_H$) AND (n == 1);

mul_res1 = sc1 ? 7FFFFFFF$_H$: ((D[a][31:16] * D[b][31:16]) << n);

mul_res0 = sc0 ? $7FFFFFFF_H$: ((D[a][15:0] * D[b][15:0]) << n);

result_halfword1 = E[d][63:32] - mul_res1 + 8000_H;

result_halfword0 = E[d][31:0] - mul_res0 + 8000_H;

D[c] = {ssov(result_halfword1, 32)[31:16], ssov(result_halfword0, 32)[31:16]};

// Packed short fraction result

MSUBRS.HD[c], D[d], D[a], D[b] UU, n (RRR1)

16U || 16L -||- (16L * 16U || 16U * 16U) rounded --> 16||16 saturated

31 28	27 24	23 18	17 16	15 12	11 8	7 0
c	d	$2F_H$	n	b	a	$A3_H$

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);

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_halfword1 = {D[d][31:16], 16'b0} - mul_res1 + 8000_H;

result_halfword0 = {D[d][15:0], 16'b0} - mul_res0 + 8000_H;

D[c] = {ssov(result_halfword1, 32)[31:16], ssov(result_halfword0, 32)[31:16]};

// Packed short fraction result

Status Flags

C	Not set by these instructions.

V	ov_halfword1 = (result_halfword1 > $7FFFFFFF_H$) OR (result_halfword1 < -80000000_H);
	ov_halfword0 = (result_halfword0 > $7FFFFFFF_H$) OR (result_halfword0 < -80000000_H);
	overflow = ov_halfword1 OR ov_halfword0;
	if (overflow) then PSW.V = 1 else PSW.V = 0;

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

AV	aov_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

–

See Also

-

MSUBR.Q
Multiply-Subtract Q Format with Rounding
MSUBRS.Q
Multiply-Subtract Q Format with Rounding, Saturated

Description

Multiply two signed 16-bit (half-word) values. Subtract the product (left justified if n == 1) from a 32-bit signed value, and put the rounded result in a 32-bit register. The lower half-word is cleared. Overflow and advanced overflow are calculated on the final results.

Note that n should only take the values 0 or 1, any other value returns an undefined result. If (n == 1) then 8000_H * 8000_H = $7FFFFFFF_H$ (for signed 16-bit * 16-bit multiplications only).

MSUBR.QD[c], D[d], D[a] L, D[b] L, n (RRR1)

32 - (16L * 16L) rounded --> 32

31 28	27 24	23 18	17 16	15 12	11 8	7 0
c	d	07_H	n	b	a	63_H

sc = (D[a][15:0] == 8000_H) AND (D[b][15:0] == 8000_H) AND (n == 1);

mul_res = sc ? $7FFFFFFF_H$: ((D[a][15:0] * D[b][15:0]) << n);

result = D[d] - mul_res + 8000_H;

D[c] = {result[31:16], 16’b0}; // Short fraction

MSUBR.QD[c], D[d], D[a] U, D[b] U, n (RRR1)

32 - (16U * 16U) rounded --> 32

31 28	27 24	23 18	17 16	15 12	11 8	7 0
c	d	06_H	n	b	a	63_H

sc = (D[a][31:16] == 8000_H) AND (D[b][31:16] == 8000_H) AND (n == 1);

mul_res = sc ? $7FFFFFFF_H$: ((D[a][31:16] * D[b][31:16]) << n);

result = D[d] - mul_res + 8000_H;

D[c] = {result[31:16], 16’b0}; // Short fraction

MSUBRS.QD[c], D[d], D[a] L, D[b] L, n (RRR1)

32 - (16L * 16L) rounded -->32 saturated

31 28	27 24	23 18	17 16	15 12	11 8	7 0
c	d	27_H	n	b	a	63_H

sc = (D[a][15:0] == 8000_H) AND (D[b][15:0] == 8000_H) AND (n == 1);

mul_res = sc ? $7FFFFFFF_H$: ((D[a][15:0] * D[b][15:0]) << n);

result = D[d] - mul_res + 8000_H;

D[c] = {ssov(result,32)[31:16]), 16’b0}; // Short fraction

MSUBRS.QD[c], D[d], D[a] U, D[b] U, n (RRR1)

32 - (16U * 16U) rounded --> 32 saturated

31 28	27 24	23 18	17 16	15 12	11 8	7 0
c	d	26_H	n	b	a	63_H

sc = (D[a][31:16] == 8000_H) AND (D[b][31:16] == 8000_H) AND (n == 1);

mul_res = sc ? $7FFFFFFF_H$: ((D[a][31:16] * D[b][31:16]) << n);

result = D[d] - mul_res + 8000_H;

D[c] = {ssov(result,32)[31:16]), 16’b0}; // Short fraction

Status Flags

C	Not set by these instructions.

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

–

See Also

-

MTCR
Move To Core Register

Description

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

Move the value in data register D[a] to the Core Special Function Register (CSFR) selected by the value const16. The CSFR address is a const16 byte offset from the CSFR base address. It must be word-aligned (the least-significant two bits are zero). Non-aligned address have an undefined effect.

The MTCR instruction can not be used to access GPRs. Attempting to update a GPR with this instruction will have no effect.

An MTCR instruction should be followed by an ISYNC instruction. This ensures that all instructions following the MTCR see the effects of the CSFR update.

MTCRconst16, D[a] (RLC)

31 28	27 12	11 8	7 0
-	const16	a	CD_H

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    4, d1
```

See Also

MFCR, RSTV

MUL
Multiply
MULS
Multiply, Saturated

Description

Multiply two signed 32-bit integers and put the product into a 32-bit or 64-bit register. The value const9 is sign-extended before the multiplication is performed. The MULS result is saturated on overflow.

Multiply D[a] by D[b] (two signed 32-bit integers) and put the product into D[a].

MULD[c], D[a], const9 (RC)
(32 * K9) --> 32 signed

31 28	27 21	20 12	11 8	7 0
c	01_H	const9	a	53_H

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

MULE[c], D[a], const9 (RC)
(32 * K9) --> 64 signed

31 28	27 21	20 12	11 8	7 0
c	03_H	const9	a	53_H

result = D[a] * sign_ext(const9);
E[c] = result[63:0];

MULD[c], D[a], D[b] (RR2)
(32 * 32) --> 32 signed

31 28	27 16	15 12	11 8	7 0
c	$0A_H$	b	a	73_H

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

MULE[c], D[a], D[b] (RR2)
(32 * 32) --> 64 signed

31 28	27 16	15 12	11 8	7 0
c	$6A_H$	b	a	73_H

result = D[a] * D[b];
E[c] = result[63:0];

MULD[a], D[b] (SRR)
(32 * 32) --> 32 signed

15 12	11 8	7 0
b	a	$E2_H$

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

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

(32 * K9) --> 32 signed saturated

31 28	27 21	20 12	11 8	7 0
c	05_H	const9	a	53_H

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

MULSD[c], D[a], D[b] (RR2)

(32 * 32) --> 32 signed saturated

31 28	27 16	15 12	11 8	7 0
c	$8A_H$	b	a	73_H

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

Status Flags

C	Not set by these instructions.

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:
	It is mathematically impossible to generate an overflow when multiplying two 32-bit numbers and storing the result in a 64-bit register.

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

AV	32-bit result:
	advanced_overflow = result[31] ^ result[30];
	if (advanced_overflow) then PSW.AV = 1 else PSW.AV = 0;
	64-bit result:
	advanced_overflow = result[63] ^ result[62];
	if (advanced_overflow) then PSW.AV = 1 else PSW.AV = 0;

SAV	if (advanced_overflow) then PSW.SAV = 1 else PSW.SAV = PSW.SAV;

Examples

```
mul    d3, d1, d2
mul    d2, d4, #21H
mul    e2, d5, d1
muls    d2, d0, d0
```

```
mul   d3, d11
```

See Also

MUL.U, **MADD**, **MSUB**

MUL.H
Packed Multiply Q Format

Description

Multiply two signed 16-bit (half-word) values and put the product (left justified if n == 1) into a 32-bit register. Note that since there are two results both halves of an extended data register are used. There are four cases of half-word multiplication:

- 16U * 16U, 16L * 16L
- 16U * 16L, 16L * 16U
- 16U * 16L, 16L * 16L
- 16L * 16U, 16U * 16U

Note that n should only take the values 0 or 1, any other value returns an undefined result. If (n == 1) then 8000_H * 8000_H = $7FFFFFFF_H$ (for signed 16-bit * 16-bit multiplications only).

MUL.HE[c], D[a], D[b] LL, n (RR1)

(16U * 16L || 16L * 16L) --> 32||32

31 28	27 18	17 16	15 12	11 8	7 0
c	$1A_H$	n	b	a	$B3_H$

sc1 = (D[a][31:16] == 8000_H) AND (D[b][15:0] == 8000_H) AND (n == 1);

sc0 = (D[a][15:0] == 8000_H) AND (D[b][15:0] == 8000_H) 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);

E[c] = {result_word1[31:0], result_word0[31:0]}; // Packed fraction

MUL.HE[c], D[a], D[b] LU, n (RR1)

(16U * 16L || 16L * 16U) --> 32||32

31 28	27 18	17 16	15 12	11 8	7 0
c	19_H	n	b	a	$B3_H$

sc1 = (D[a][31:16] == 8000_H) AND (D[b][15:0] == 8000_H) AND (n == 1);

sc0 = (D[a][15:0] == 8000_H) AND (D[b][31:16] == 8000_H) 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][31:16]) << n);

E[c] = {result_word1[31:0], result_word0[31:0]}; // Packed fraction

MUL.HE[c], D[a], D[b] UL, n (RR1)

(16U * 16U || 16L * 16L) --> 32||32

31 28	27 18	17 16	15 12	11 8	7 0
c	18_H	n	b	a	$B3_H$

sc1 = (D[a][31:16] == 8000_H) AND (D[b][31:16] == 8000_H) AND (n == 1);

sc0 = (D[a][15:0] == 8000_H) AND (D[b][15:0] == 8000_H) AND (n == 1);

result_word1 = sc1 ? $7FFFFFFF_H$: ((D[a][31:16] * D[b][31:16]) << 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.HE[c], D[a], D[b] UU, n (RR1)
(16L * 16U || 16U * 16U) --> 32||32

31 28	27 18	17 16	15 12	11 8	7 0
c	$1B_H$	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);
E[c] = {result_word1[31:0], result_word0[31:0]}; // Packed fraction

Status Flags

C	Not set by this instruction.
V	The PSW.V status bit is cleared.
SV	Not set by this instruction.

AV	aov_word1 = result_word1[31] ^ result_word1[30];
	aov_word0 = result_word0[31] ^ result_word0[30];
	advanced_overflow = aov_word1 OR aov_word0;
	if (advanced_overflow) then PSW.AV = 1 else PSW.AV = 0;

SAV	if (advanced_overflow) then PSW.SV = 1 else PSW.SV = PSW.SV;

Examples

–

See Also

-

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.QD[c], D[a], D[b], n (RR1)

(32 * 32)Up --> 32

31 28	27 18	17 16	15 12	11 8	7 0
c	02_H	n	b	a	93_H

result = ((D[a] * D[b]) << n) >> 32;

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

MUL.QE[c], D[a], D[b], n (RR1)

(32 * 32) --> 64

31 28	27 18	17 16	15 12	11 8	7 0
c	$1B_H$	n	b	a	93_H

result = (D[a] * D[b]) << n;

E[c] = result[63:0]; // Multi-precision fraction

MUL.QD[c], D[a], D[b] L, n (RR1)

(32 * 16L)Up --> 32

31 28	27 18	17 16	15 12	11 8	7 0
c	01_H	n	b	a	93_H

result = ((D[a] * D[b][15:0]) << n) >> 16;

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

MUL.QE[c], D[a], D[b] L, n (RR1)

(32 * 16L) --> 64

31 28	27 18	17 16	15 12	11 8	7 0
c	19_H	n	b	a	93_H

result = (D[a] * D[b][15:0]) << n;

E[c] = result[63:0]; // Multi-precision accumulator

MUL.QD[c], D[a], D[b] U, n (RR1)

(32 * 16U)Up --> 32

31 28	27 18	17 16	15 12	11 8	7 0
c	00_H	n	b	a	93_H

result = ((D[a] * D[b][31:16]) << n) >> 16;

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

MUL.QE[c], D[a], D[b] U, n (RR1)

(32 * 16U) --> 64

31 28	27 18	17 16	15 12	11 8	7 0
c	18_H	n	b	a	93_H

result = (D[a] * D[b][31:16]) << n;

E[c] = result[63:0]; // Multi-precision accumulator

MUL.QD[c], D[a] L, D[b] L, n (RR1)

(16L * 16L) --> 32

31 28	27 18	17 16	15 12	11 8	7 0
c	05_H	n	b	a	93_H

sc = (D[a][15:0] == 8000_H) AND (D[b][15:0] == 8000_H) AND (n == 1);

result = sc ? $7FFFFFFF_H$: ((D[a][15:0] * D[b][15:0]) << n);

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

MUL.QD[c], D[a] U, D[b] U, n (RR1)

(16U * 16U) --> 32

31 28	27 18	17 16	15 12	11 8	7 0
c	04_H	n	b	a	93_H

sc = (D[a][31:16] == 8000_H) AND (D[b][31:16] == 8000_H) AND (n == 1);

result = sc ? $7FFFFFFF_H$: ((D[a][31:16] * D[b][31:16]) << n);

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

Status Flags

C	Not set by this instruction.

V	32-bit result:
	overflow = (result > $7FFFFFFF_H$) 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

```
mul.q   d3, d1U, d2U, #1
mul.q   d3, d1L, d2L, #1
mul.q   d3, d1, d2U, #0
mul.q   d3, d1, d2L, #1
mul.q   d2, d1, d2, #1
mul.q   e2, d1, d0U, #1
mul.q   e2, d1, d0L, #1
mul.q   e2, d1, d7, #1
```

See Also

-

MUL.U
Multiply Unsigned
MULS.U
Multiply Unsigned, Saturated

Description

Multiply two unsigned 32-bit integers and put the product into a 32-bit or 64-bit register. The value const9 (instruction format RC) is zero-extended before the multiplication is performed. The MULS.U result is saturated on overflow.

MUL.UE[c], D[a], const9 (RC)

(32 * K9) --> 64 unsigned

31 28	27 21	20 12	11 8	7 0
c	02_H	const9	a	53_H

result = D[a] * zero_ext(const9); // unsigned

E[c] = result[63:0];

MUL.UE[c], D[a], D[b] (RR2)

(32 * 32) --> 64 unsigned

31 28	27 16	15 12	11 8	7 0
c	68_H	b	a	73_H

result = D[a] * D[b]; // unsigned

E[c] = result[63:0];

MULS.UD[c], D[a], const9 (RC)

(32 * K9) --> 32 unsigned saturated

31 28	27 21	20 12	11 8	7 0
c	04_H	const9	a	53_H

result = D[a] * zero_ext(const9); // unsigned

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

MULS.UD[c], D[a], D[b] (RR2)

(32 * 32) --> 32 unsigned saturated

31 28	27 16	15 12	11 8	7 0
c	88_H	b	a	73_H

result = D[a] * D[b]; // unsigned

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

Status Flags

C	Not set by this instruction.

V	32-bit result:
	overflow = (result > $FFFFFFFF_H$) OR (result < 00000000_H);
	if (overflow) then PSW.V = 1 else PSW.V = 0;
	64-bit result:
	It is mathematically impossible to generate an overflow when multiplying two 32-bit numbers and storing the result in a 64-bit register.

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

AV	32-bit result:
	advanced_overflow = result[31] ^ result[30];
	if (advanced_overflow) then PSW.AV = 1 else PSW.AV = 0;
	64-bit result:
	advanced_overflow = result[63] ^ result[62];
	if (advanced_overflow) then PSW.AV = 1 else PSW.AV = 0;

SAV	if (advanced_overflow) then PSW.SAV = 1 else PSW.SAV = PSW.SAV;

Examples

```
mul.u           e0, d2, d3
muls.u          d3, d5, d9
```

See Also

MUL

MULM.H

Packed Multiply Q Format-Multi-precision

Description

Perform two multiplications of two signed 16-bit (half-word) values. Add the two products (left justified if n == 1) left-shifted by 16, in a 64-bit register. There are four cases of half-word multiplication:

- 16U * 16U, 16L * 16L
- 16U * 16L, 16L * 16U
- 16U * 16L, 16L * 16L
- 16L * 16U, 16U * 16U

Note that n should only take the values 0 or 1, any other value returns an undefined result. If (n == 1) then 8000_H * 8000_H = $7FFFFFFF_H$ (for signed 16-bit * 16-bit multiplications only).

MULM.HE[c], D[a], D[b] LL, n (RR1)

16U * 16L + 16L * 16L --> 64

31 28	27 18	17 16	15 12	11 8	7 0
c	$1E_H$	n	b	a	$B3_H$

sc1 = (D[a][31:16] == 8000_H) AND (D[b][15:0] == 8000_H) AND (n == 1);

sc0 = (D[a][15:0] == 8000_H) AND (D[b][15:0] == 8000_H) 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.HE[c], D[a], D[b] LU, n (RR1)

16U * 16L + 16L * 16U --> 64

31 28	27 18	17 16	15 12	11 8	7 0
c	$1D_H$	n	b	a	$B3_H$

sc1 = (D[a][31:16] == 8000_H) AND (D[b][15:0] == 8000_H) AND (n == 1);

sc0 = (D[a][15:0] == 8000_H) AND (D[b][31:16] == 8000_H) 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][31:16]) << n);

result = (result_word1 + result_word0) << 16;

E[c] = result[63:0]; // Multi-precision accumulator

MULM.HE[c], D[a], D[b] UL, n (RR1)

16U * 16U + 16L * 16L --> 64

31 28	27 18	17 16	15 12	11 8	7 0
c	$1C_H$	n	b	a	$B3_H$

sc1 = (D[a][31:16] == 8000_H) AND (D[b][31:16] == 8000_H) AND (n == 1);

sc0 = (D[a][15:0] == 8000_H) AND (D[b][15:0] == 8000_H) AND (n == 1);

result_word1 = sc1 ? $7FFFFFFF_H$: ((D[a][31:16] * D[b][31:16]) << 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.HE[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

–

See Also

-

MULMS.H

MULMS.H (DEPRECATED) (Note: Deprecated in TC2!)

Description

-

MULMS.H E[c], D[a], D[b] LL, n (RR1)

31 28	27 18	17 16	15 12	11 8	7 0
c	$3E_H$	n	b	a	$B3_H$

-

MULMS.H E[c], D[a], D[b] LU, n (RR1)

31 28	27 18	17 16	15 12	11 8	7 0
c	$3D_H$	n	b	a	$B3_H$

-

MULMS.H E[c], D[a], D[b] UL, n (RR1)

31 28	27 18	17 16	15 12	11 8	7 0
c	$3C_H$	n	b	a	$B3_H$

-

MULMS.H E[c], D[a], D[b] UU, n (RR1

31 28	27 18	17 16	15 12	11 8	7 0
c	$3F_H$	n	b	a	$B3_H$

-

Status Flags

C	C Status flag is not set.
V	V Status flag is not set.
SV	SV Status flag is not set.
AV	AV Status flag is not set.
SAV	SAV Status flag is not set.

Examples

–

See Also

-

MULR.H

Packed Multiply Q Format with Rounding

Description

Multiply two signed 16-bit (half-word) values. Add the product (left justified if n == 1) to a signed 16-bit value and put the rounded result into half of a 32-bit register. Note that since there are two results the two register halves are used). There are four cases of half-word multiplication:

- 16U * 16U, 16L * 16L
- 16U * 16L, 16L * 16U
- 16U * 16L, 16L * 16L
- 16L * 16U, 16U * 16U

Note that n should only take the values 0 or 1, any other value returns an undefined result. If (n == 1) then 8000_H * 8000_H = $7FFFFFFF_H$ (for signed 16-bit * 16-bit multiplications only).

MULR.HD[c], D[a], D[b] LL, n (RR1)

(16U * 16L || 16L * 16L) rounded --> 16||16

31 28	27 18	17 16	15 12	11 8	7 0
c	$0E_H$	n	b	a	$B3_H$

sc1 = (D[a][31:16] == 8000_H) AND (D[b][15:0] == 8000_H) AND (n == 1);

sc0 = (D[a][15:0] == 8000_H) AND (D[b][15:0] == 8000_H) AND (n == 1);

result_halfword1 = sc1 ? $7FFFFFFF_H$: (((D[a][31:16] * D[b][15:0]) << n) + 8000_H);

result_halfword0 = sc0 ? $7FFFFFFF_H$: (((D[a][15:0] * D[b][15:0]) << n) + 8000_H);

D[c] = {result_halfword1[31:16], result_halfword0[31:16]}; // Packed short fraction

MULR.HD[c], D[a], D[b] LU, n (RR1)

(16U * 16L || 16L * 16U) rounded --> 16||16

31 28	27 18	17 16	15 12	11 8	7 0
c	$0D_H$	n	b	a	$B3_H$

sc1 = (D[a][31:16] == 8000_H) AND (D[b][15:0] == 8000_H) AND (n == 1);

sc0 = (D[a][15:0] == 8000_H) AND (D[b][31:16] == 8000_H) AND (n == 1);

result_halfword1 = sc1 ? $7FFFFFFF_H$: (((D[a][31:16] * D[b][15:0]) << n) + 8000_H);

result_halfword0 = sc0 ? $7FFFFFFF_H$: (((D[a][15:0] * D[b][31:16]) << n) + 8000_H);

D[c] = {result_halfword1[31:16], result_halfword0[31:16]}; // Packed short fraction

MULR.HD[c], D[a], D[b] UL, n (RR1)

(16U * 16U || 16L * 16L) rounded --> 16||16

31 28	27 18	17 16	15 12	11 8	7 0
c	$0C_H$	n	b	a	$B3_H$

sc1 = (D[a][31:16] == 8000_H) AND (D[b][31:16] == 8000_H) AND (n == 1);

sc0 = (D[a][15:0] == 8000_H) AND (D[b][15:0] == 8000_H) AND (n == 1);

result_halfword1 = sc1 ? $7FFFFFFF_H$: (((D[a][31:16] * D[b][31:16]) << n) + 8000_H);

result_halfword0 = sc0 ? $7FFFFFFF_H$: (((D[a][15:0] * D[b][15:0]) << n) + 8000_H);

D[c] = {result_halfword1[31:16], result_halfword0[31:16]}; // Packed short fraction

MULR.HD[c], D[a], D[b] UU, n (RR1)
(16L * 16U || 16U * 16U) rounded --> 16||16

31 28	27 18	17 16	15 12	11 8	7 0
c	$0F_H$	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_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

–

See Also

-

MULR.Q
Multiply Q Format with Rounding

Description

Multiply two signed 16-bit (half-word) values and put the rounded result (left justified if n == 1) into a 32-bit register. The lower half-word is cleared.

Note that n should only take the values 0 or 1, any other value returns an undefined result. If (n == 1) then 8000_H * 8000_H = $7FFFFFFF_H$ (for signed 16-bit * 16-bit multiplications only).

MULR.QD[c], D[a] L, D[b] L, n (RR1)

(16L * 16L) rounded --> 32

31 28	27 18	17 16	15 12	11 8	7 0
c	07_H	n	b	a	93_H

sc = (D[a][15:0] == 8000_H) AND (D[b][15:0] == 8000_H) AND (n == 1);

result = sc ? $7FFFFFFF_H$: (((D[a][15:0] * D[b][15:0]) << n) + 8000_H);

D[c] = {result[31:16], 16’b0}; // Short fraction

MULR.QD[c], D[a] U, D[b] U, n (RR1)

(16U * 16U) rounded --> 32

31 28	27 18	17 16	15 12	11 8	7 0
c	06_H	n	b	a	93_H

sc = (D[a][31:16] == 8000_H) AND (D[b][31:16] == 8000_H) AND (n == 1);

result = sc ? $7FFFFFFF_H$: (((D[a][31:16] * D[b][31:16]) << n) + 8000_H);

D[c] = {result[31:16], 16’b0}; // 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

–

See Also

-

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.

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

31 28	27 21	20 12	11 8	7 0
c	09_H	const9	a	$8F_H$

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

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

31 28	27 20	19 18	17 16	15 12	11 8	7 0
c	09_H	-	-	b	a	$0F_H$

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, d2
nand   d3, d1, #126
```

See Also

AND, **ANDN**, **NOR**, **NOT (16-bit)**, **OR**, **ORN**, **XNOR**, **XOR**

NAND.T
Bit Logical NAND

Description

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

NAND.TD[c], D[a], pos1, D[b], pos2 (BIT)

31 28	27 23	22 21	20 16	15 12	11 8	7 0
c	pos2	00_H	pos1	b	a	07_H

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

D[c] = zero_ext(result);

Status Flags

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

Examples

```
nand.t   d3, d1, 2, d2, #4
```

See Also

AND.T, **ANDN.T**, **OR.T**, **ORN.T**, **XNOR.T**, **XOR.T**

NE
Not Equal

Description

If the contents of data register D[a] are not equal to the contents of either data register D[b] (instruction format RR) or const9 (instruction format RC), set the least-significant bit of D[c] to one and clear the remaining bits to zero; otherwise clear all bits in D[c]. The const9 value is sign-extended.

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

31 28	27 21	20 12	11 8	7 0
c	11_H	const9	a	$8B_H$

result = (D[a] != sign_ext(const9));

D[c] = zero_ext(result);

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

31 28	27 20	19 18	17 16	15 12	11 8	7 0
c	11_H	-	-	b	a	$0B_H$

result = (D[a] != D[b]);

D[c] = zero_ext(result);

Status Flags

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

Examples

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

See Also

EQ, **GE**, **GE.U**, **LT**, **LT.U**, **EQANY.B**, **EQANY.H**, **NEZ.A**

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

31 28	27 20	19 18	17 16	15 12	11 8	7 0
c	41_H	-	-	b	a	01_H

D[c] = (A[a] != A[b]);

Status Flags

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

Examples

```
ne.a    d3, a4, a2
```

See Also

EQ.A, **EQZ.A**, **GE.A**, **LT.A**, **NEZ.A**

NEZ.A
Not Equal Zero Address

Description

If the contents of address register A[a] are not equal to zero, set the least significant bit of D[c] to one and clear the remaining bits to zero; otherwise clear all bits in D[c].

NEZ.AD[c], A[a] (RR)

31 28	27 20	19	18 17	16 15 12	11 8	7 0
c	49_H	-	-	-	a	01_H

D[c] = (A[a] != 0);

Status Flags

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

Examples

```
nez.a   d3, a4
```

See Also

EQ.A, **EQZ.A**, **GE.A**, **LT.A**, **NE**

NOP
No Operation

Description

Used to implement efficient low-power, non-operational instructions.

Used to implement efficient low-power, non-operational instructions.

NOP(SR)

15–12	11–8	7–0
00_H	-	00_H

No operation.

NOP(SYS)

31–28	27–22	21–12	11–8	7–0
-	00_H	-	-	$0D_H$

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

-

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.

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

31 28	27 21	20 12	11 8	7 0
c	$0B_H$	const9	a	$8F_H$

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

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

31 28	27 20	19 18	17 16	15 12	11 8	7 0
c	$0B_H$	-	-	b	a	$0F_H$

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, d2
nor   d3, d1, #126
```

See Also

AND, **ANDN**, **NAND**, **NOT (16-bit)**, **OR**, **ORN**, **XNOR**, **XOR**

NOR.T
Bit Logical NOR

Description

Compute the logical NOR of bit pos1 of data register D[a] and bit pos2 of data register D[b]. Put the result in the least-significant bit of data register D[c] and clear the remaining bits of D[c] to zero.

NOR.TD[c], D[a], pos1, D[b], pos2 (BIT)

31 28	27 23	22 21	20 16	15 12	11 8	7 0
c	pos2	02_H	pos1	b	a	87_H

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**

NOT (16-bit)
Bitwise Complement NOT (16-bit)

Description

Compute the bitwise NOT of the contents of register D[a] and put the result in data register D[a].

NOTD[a] (SR)

15 12	11 8	7 0
00_H	a	46_H

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**

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.

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

31 28	27 21	20 12	11 8	7 0
c	$0A_H$	const9	a	$8F_H$

D[c] = D[a] | zero_ext(const9);

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

31 28	27 20	19 18	17 16	15 12	11 8	7 0
c	$0A_H$	-	-	b	a	$0F_H$

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

ORD[15], const8 (SC)

15 8	7 0
const8	96_H

D[15] = D[15] | zero_ext(const8);

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

15 12	11 8	7 0
b	a	$A6_H$

D[a] = D[a] | D[b];

Status Flags

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

Examples

```
or   d3, d1, d2
or   d3, d1, #126
```

```
or   d1, d2
or   d15, #126
```

See Also

AND, **ANDN**, **NAND**, **NOR**, **NOT (16-bit)**, **ORN**, **XNOR**, **XOR**

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

Description

Compute the logical operation (AND, ANDN, NOR or OR as appropriate) of the value of bit pos1 of data register D[a], and bit pos2 of D[b]. Compute the logical OR of that result and bit [0] of D[c]. Put the result back in bit [0] of D[c]. All other bits in D[c] are unchanged.

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

31 28	27 23	22 21	20 16	15 12	11 8	7 0
c	pos2	00_H	pos1	b	a	$C7_H$

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

OR.ANDN.TD[c], D[a], pos1, D[b], pos2 (BIT)

31 28	27 23	22 21	20 16	15 12	11 8	7 0
c	pos2	03_H	pos1	b	a	$C7_H$

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

OR.NOR.TD[c], D[a], pos1, D[b], pos2 (BIT)

31 28	27 23	22 21	20 16	15 12	11 8	7 0
c	pos2	02_H	pos1	b	a	$C7_H$

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

OR.OR.TD[c], D[a], pos1, D[b], pos2 (BIT)

31 28	27 23	22 21	20 16	15 12	11 8	7 0
c	pos2	01_H	pos1	b	a	$C7_H$

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

Status Flags

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

SAV	Not set by these instructions.

Examples

```
or.and.t    d3, d1, 3, d2, 5
or.andn.t   d3, d1, 3, d2, 5
or.nor.t    d3, d1, 3, d2, 5
or.or.t     d3, d1, 3, d2, 5
```

See Also

AND.AND.T, **AND.ANDN.T**, **AND.NOR.T**, **AND.OR.T**, **SH.AND.T**, **SH.ANDN.T**, **SH.NAND.T**, **SH.NOR.T**, **SH.OR.T**, **SH.ORN.T**, **SH.XNOR.T**

OR.EQ
Equal Accumulating

Description

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

OR.EQD[c], D[a], const9 (RC)

31 28	27 21	20 12	11 8	7 0
c	27_H	const9	a	$8B_H$

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

OR.EQD[c], D[a], D[b] (RR)

31 28	27 20	19 18	17 16	15 12	11 8	7 0
c	27_H	-	-	b	a	$0B_H$

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

See Also

AND.EQ, **XOR.EQ**

OR.GE
Greater Than or Equal Accumulating
OR.GE.U
Greater Than or Equal Accumulating Unsigned

Description

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

OR.GED[c], D[a], const9 (RC)

31 28	27 21	20 12	11 8	7 0
c	$2B_H$	const9	a	$8B_H$

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

OR.GED[c], D[a], D[b] (RR)

31 28	27 20	19 18	17 16	15 12	11 8	7 0
c	$2B_H$	-	-	b	a	$0B_H$

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

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

31 28	27 21	20 12	11 8	7 0
c	$2C_H$	const9	a	$8B_H$

D[c] = {D[c][31:1], D[c][0] OR (D[a] >= zero_ext(const9))}; // unsigned

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

31 28	27 20	19 18	17 16	15 12	11 8	7 0
c	$2C_H$	-	-	b	a	$0B_H$

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

Status Flags

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

Examples

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

See Also

AND.GE, **AND.GE.U**, **XOR.GE**, **XOR.GE.U**

OR.LT
Less Than Accumulating
OR.LT.U
Less Than Accumulating Unsigned

Description

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

OR.LTD[c], D[a], const9 (RC)

31 28	27 21	20 12	11 8	7 0
c	29_H	const9	a	$8B_H$

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

OR.LTD[c], D[a], D[b] (RR)

31 28	27 20	19 18	17 16	15 12	11 8	7 0
c	29_H	-	-	b	a	$0B_H$

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

OR.LT.UD[c], D[a], const9 (RC)

31 28	27 21	20 12	11 8	7 0
c	$2A_H$	const9	a	$8B_H$

D[c] = {D[c][31:1], D[c][0] OR (D[a] < zero_ext(const9))}; // unsigned

OR.LT.UD[c], D[a], D[b] (RR)

31 28	27 20	19 18	17 16	15 12	11 8	7 0
c	$2A_H$	-	-	b	a	$0B_H$

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

Status Flags

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

Examples

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

See Also

AND.LT, **AND.LT.U**, **XOR.LT**, **XOR.LT.U**

OR.NE
Not Equal Accumulating

Description

Calculate the logical OR of D[c][0] and the Boolean result of the NE operation on the contents of data register D[a] and either data register D[b] (instruction format RR) or const9 (instruction format RC). Put the result in D[c][0]. All other bits in D[c] are unchanged.

OR.NED[c], D[a], const9 (RC)

31 28	27 21	20 12	11 8	7 0
c	28_H	const9	a	$8B_H$

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

OR.NED[c], D[a], D[b] (RR)

31 28	27 20	19 18	17 16	15 12	11 8	7 0
c	28_H	-	-	b	a	$0B_H$

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

Status Flags

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

Examples

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

See Also

AND.NE, XOR.NE

OR.T
Bit Logical OR

Description

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

OR.TD[c], D[a], pos1, D[b], pos2 (BIT)

31 28	27 23	22 21	20 16	15 12	11 8	7 0
c	pos2	01_H	pos1	b	a	87_H

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**

ORN
Bitwise OR-Not

Description

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

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

31 28	27 21	20 12	11 8	7 0
c	$0F_H$	const9	a	$8F_H$

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

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

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

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, d2
orn   d3, d1, #126
```

See Also

AND, **ANDN**, **NAND**, **NOR**, **NOT (16-bit)**, **OR**, **XNOR**, **XOR**

ORN.T
Bit Logical OR-Not

Description

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

ORN.TD[c], D[a], pos1, D[b], pos2 (BIT)

31 28	27 23	22 21	20 16	15 12	11 8	7 0
c	pos2	01_H	pos1	b	a	07_H

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**

PACK
Pack

Description

Take the data register pair E[d] and bit 31 of data register D[a] and pack them into an IEEE-754 single precision floating point format number, in data register D[c]. The odd register E[d][63:32], holds the unbiased exponent. The even register E[d][31:0], holds the normalised mantissa in a fractional 1.31 format. Bit 31 of data register D[a] holds the sign bit.

To compute the floating point format number, the input number is first checked for special cases: Infinity, NAN, Overflow, Underflow and Zero. If the input number is not one of these special cases, it is either a normal or denormal number. In both cases, rounding of the input number is performed. First an intermediate biased exponent is calculated, by adding 128 to the unpacked exponent for normal numbers and set to zero for denormal numbers, and inserted into bits [30:23] of the intermediate result. Bits [30:8] of E[d] are inserted into bits [22:0] of the intermediate result. A round flag is calculated from bits [8:0] of E[d] using the IEEE-754 Round-to-Nearest rounding definition, with the PSW.C field acting as an additional sticky bit. If the round flag is set, the intermediate result is incremented by one. Bits [30:0] of the intermediate result are then inserted into bits [30:0] of D[c]. In all cases, bit 31 from D[a] is copied into bit 31 of D[c]. The special cases are handled as described below.

PACKD[c], E[d], D[a] (RRR)

31 28	27 24	23 20	19 18	17 16	15 12	11 8	7 0
c	d	00_H	-	0_H	-	a	$6B_H$

```
int_exp = E[d][63:32];
int_mant = E[d][31:0];
flag_rnd = int_mant[7] AND (int_mant[8] OR int_mant[6:0] OR PSW.C);
if ((int_mant[31] == 0) AND (int_exp == +255)) then {
// Infinity or NaN
fp_exp = +255;
 fp_frac = int_mant[30:8];
} else if ((int_mant[31] == 1) AND (int_exp >= +127)) then {
// Overflow ? Infinity.
fp_exp = +255;
fp_frac = 0;
} else if ((int_mant[31] == 1) AND (int_exp <= -128)) then {
// Underflow ? Zero
fp_exp = 0;
 fp_frac = 0;
} else if (int_mant == 0) then {
// Zero
 fp_exp = 0;
 fp_frac = 0;
} else {
if (int_mant[31] == 0) then {
// Denormal
   temp_exp = 0;
 } else {
// Normal
    temp_exp = int_exp + 128;
 }
fp_exp_frac[30:0] = {tmp_exp[7:0], int_mant[30:8]} + flag_rnd;
fp_exp = fp_exp_frac[30:23];
fp_frac = fp_exp_frac[22:0];
}
D[c][31] = D[a][31];
D[c][30:23] = fp_exp;
D[c][22:0] = fp_frac;
```

Status Flags

C	PSW.C is read by the instruction but not set by this instruction.
V	Not set by this instruction.
SV	Not set by this instruction.
AV	Not set by this instruction.
SAV	Not set by this instruction.

Examples

```
pack   d8, e2, d10
```

See Also

UNPACK

PARITY
Parity

Description

Compute the four byte parity bits of data register D[a]. Put each byte parity bit into every 8th bit of the data register D[c] and then clear the remaining bits of D[c]. A byte parity bit is set to one if the number of ones in a byte is an odd number.

PARITYD[c], D[a] (RR)

31 28	27 20	19 18	17 16	15 12	11 8	7 0
c	02_H	-	0_H	-	a	$4B_H$

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

-

RESTORE
Restore

Description

Restore the Interrupt Enable bit (ICR.IE) to the value saved in D[a][0].

RESTORE can only be executed in User-1 or Supervisor mode.

RESTORED[a] (SYS)

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

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**

RET
Return from Call

Description

Return from a function that was invoked with a CALL instruction. The return address is in register A[11] (return address). The caller's upper context register values are restored as part of the return operation.

Return from a function that was invoked with a CALL instruction. The return address is in register A[11] (return address). The caller's upper context register values are restored as part of the return operation.

RET(SR)

15 12	11 8	7 0
09_H	-	00_H

```
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, new_PSW, A[10], A[11], D[8], D[9], D[10], D[11], A[12], A[13], A[14], A[15], D[12], D[13], D[14],
D[15]} = M(EA, 16 * word);
M(EA, word) = FCX;
FCX[19:0] = PCXI[19:0];
PCXI = new_PCXI;
PSW = {new_PSW[31:26], PSW[25:24], new_PSW[23:0]};
```

RET(SYS)

31 28	27 22	21 12	11 8	7 0
-	06_H	-	-	$0D_H$

```
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, new_PSW, A[10], A[11], D[8], D[9], D[10], D[11], A[12], A[13], A[14], A[15], D[12], D[13], D[14], D[15]}
= M(EA, 16 * word);
M(EA, word) = FCX;
FCX[19:0] = PCXI[19:0];
PCXI = new_PCXI;
PSW = {new_PSW[31:26], PSW[25:24], new_PSW[23:0]};
```

Status Flags

C	PSW.C is overwritten with the value restored from the Context Save Area (CSA).

V	PSW.V is overwritten with the value restored from the CSA.
SV	PSW.SV is overwritten with the value restored from the CSA.
AV	PSW.AV is overwritten with the value restored from the CSA.
SAV	PSW.SAV is overwritten with the value restored from the CSA.

Examples

```
ret
```

```
ret
```

See Also

CALL, **CALLA**, **CALLI**, **RFE**, **SYSCALL**, **BISR**, **FCALL**, **FCALLA**, **FCALLI**

RFE
Return From Exception

Description

Return from an interrupt service routine or trap handler to the task whose saved upper context is specified by the contents of the Previous Context Information register (PCXI). The contents are normally the context of the task that was interrupted or that took a trap. However in some cases Task Management software may have altered the contents of the PCXI register to cause another task to be dispatched.

The return PC value is taken from register A[11] (register address) in the current context. In parallel with the jump to the return PC address, the upper context registers and PSW in the saved context are restored.

Return from an interrupt service routine or trap handler to the task whose saved upper context is specified by the contents of the Previous Context Information register (PCXI). The contents are normally the context of the task that was interrupted or that took a trap. However in some cases Task Management software may have altered the contents of the PXCI register to cause another task to be dispatched.
The return PC value is taken from register A[11] (register address) in the current context. In parallel with the jump to the return PC address, the upper context registers and PSW in the saved context are restored.

RFE(SR)

15 12	11 8	7 0
08_H	-	00_H

```
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, 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;
```

RFE(SYS)

31 28	27 22	21 12	11 8	7 0
-	07_H	-	-	$0D_H$

```
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, PSW, A[10], A[11], D[8], D[9], D[10], D[11], A[12], A[13], A[14], A[15], D[12], D[13], D[14], D[15]} = M(EA, 16 * word);
M(EA, word) = FCX;
FCX[19:0] = PCXI[19:0];
PCXI = new_PCXI;
```

Status Flags

C	PSW.C is overwritten with the value restored from the Context Save Area (CSA).
V	PSW.V is overwritten with the value restored from the CSA.
SV	PSW.SV is overwritten with the value restored from the CSA.
AV	PSW.AV is overwritten with the value restored from the CSA.
SAV	PSW.SAV is overwritten with the value restored from the CSA.

Examples

```
rfe
```

```
rfe
```

See Also

CALL, **CALLA**, **CALLI**, **RET**, **SYSCALL**, **BISR**, **RFM**

RFM
Return From Monitor

Description

Note: The RFM instruction can only be executed in Supervisor mode.

If the Debug mode is disabled (DBGSR.DE==0) execute as a NOP; otherwise return from a breakpoint monitor to the task whose saved debug context area is located at DCX (Debug Context Pointer).

The Debug Context Area is a four word subset of the context of the task that took a Debug trap, which is saved on entry to the monitor routine. The return PC value is taken from register A[11]. In parallel with the jump to the return PC address, the PCXI and PSW, together with the saved A[10] and A[11] values in the Debug Context Area, are restored to the original task.

The Debug Trap active bit (DBGTCR.DTA) is cleared.

RFM(SYS)

31 28	27 22	21 12	11 8	7 0
-	05_H	-	-	$0D_H$

The Debug Context Pointer (DCX) value is implementation dependent.

```
if (PSW.IO != 2'b10) then trap (PRIV);
if (DBGSR.DE) then{
    PC = {A[11] [31:1], 1'b0};
    ICR.IE = PCXI.IE;
    ICR.CCPN = PCXI.PCPN;
    EA = DCX;
    {PCXI, PSW, A[10], A[11]} = M(EA, 4 * word);
    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**

RSLCX
Restore Lower Context

Description

Load the contents of the memory block pointed to by the PCX field in PCXI into registers A[2] to A[7], D[0] to D[7], A[11] (return address), and PCXI. This operation restores the register contents of a previously saved lower context.

RSLCX(SYS)

31–28	27–22	21–12	11–8	7–0
-	09_H	-	-	$0D_H$

```
if(PCXI[19:0] == 0) then trap(CSU);
if(PCXI.UL == 1) then trap(CTYP);
EA = {PCXI.PCXS, 6'b0, PCXI.PCXO, 6'b0};
{new_PCXI, A[11], A[2], A[3], D[0], D[1], D[2], D[3], A[4], A[5], A[6], A[7], D[4], D[5], D[6], D[7]} = M(EA, 16*word);
M(EA, word) = FCX;
FCX[19:0] = PCXI[19:0];
PCXI = new_PCXI;
```

Status Flags

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

Examples

```
rslcx
```

See Also

LDLCX, **LDUCX**, **STLCX**, **STUCX**, **SVLCX**, **BISR**

RSTV
Reset Overflow Bits

Description

Reset overflow status flags in the Program Status Word (PSW).

RSTV(SYS)

31 28	27 22	21 12	11 8	7 0
-	00_H	-	-	$2F_H$

PSW.{V, SV, AV, SAV} = {0, 0, 0, 0};

Status Flags

C	Not set by this instruction.
V	The PSW.V status bit is cleared.
SV	The PSW.SV status bit is cleared.
AV	The PSW.AV status bit is cleared.
SAV	The PSW.SAV status bit is cleared.

Examples

```
rstv
```

See Also

BISR, **DISABLE**, **ENABLE**, **MTCR**, **TRAPV**, **TRAPSV**

RSUB
Reverse-Subtract

Description

Subtract the contents of data register D[a] from the value const9 and put the result in data register D[c]. The operands are treated as 32-bit integers. The value const9 is sign-extended before the subtraction is performed.

Subtract the contents of data register D[a] from zero and put the result in data register D[a]. The operand is treated as a 32-bit integer.

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

31 28	27 21	20 12	11 8	7 0
c	08_H	const9	a	$8B_H$

result = sign_ext(const9) - D[a];

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

RSUBD[a] (SR)

15 12	11 8	7 0
05_H	a	32_H

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**

RSUBS
Reverse-Subtract with Saturation
RSUBS.U
Reverse-Subtract Unsigned with Saturation

Description

Subtract the contents of data register D[a] from the value const9 and put the result in data register D[c]. The operands are treated as signed (RSUBS) or unsigned (RSUBS.U) 32-bit integers, with saturation on signed (RSUBS) or unsigned (RSUBS.U) overflow. The value const9 is sign-extended before the operation is performed.

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

31 28	27 21	20 12	11 8	7 0
c	$0A_H$	const9	a	$8B_H$

result = sign_ext(const9) - D[a];

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

RSUBS.UD[c], D[a], const9 (RC)

31 28	27 21	20 12	11 8	7 0
c	$0B_H$	const9	a	$8B_H$

result = sign_ext(const9) - D[a]; // unsigned

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

Status Flags

C	Not set by these instructions.

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
rsubs.u   d3, d1, #126
```

See Also

RSUB

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.BD[c], D[a] (RR)

31 28	27 20	19 18	17 16	15 12	11 8	7 0
c	$5E_H$	-	-	-	a	$0B_H$

sat_neg = (D[a] < -80_H) ? -80_H : D[a];

D[c] = (sat_neg > $7F_H$) ? $7F_H$: sat_neg;

SAT.BD[a] (SR)

15 12	11 8	7 0
00_H	a	32_H

sat_neg = (D[a] < -80_H) ? -80_H : D[a];
D[a] = (sat_neg > $7F_H$) ? $7F_H$: 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**

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.BUD[c], D[a] (RR)

31 28	27 20	19	18 17	16 15 12	11 8	7 0
c	5F$_H$	-	-	-	a	0B$_H$

D[c] = (D[a] > FF$_H$) ? FF$_H$: D[a]; // unsigned comparison

SAT.BUD[a] (SR)

15 12	11 8	7 0
01$_H$	a	32$_H$

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

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.HD[c], D[a] (RR)

31 - 28	27 - 20	19 - 18	17 - 16	15 - 12	11 - 8	7 - 0
c	$7E_H$	-	-	-	a	$0B_H$

sat_neg = (D[a] < -8000_H) ? -8000_H : D[a];

D[c] = (sat_neg > $7FFF_H$) ? $7FFF_H$: sat_neg;

SAT.HD[a] (SR)

15 - 12	11 - 8	7 - 0
02_H	a	32_H

sat_neg = (D[a] < -8000_H) ? -8000_H : D[a];
D[a] = (sat_neg > $7FFF_H$) ? $7FFF_H$: 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**

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.HUD[c], D[a] (RR)

31 28	27 20	19 18	17 16	15 12	11 8	7 0
c	$7F_H$	-	-	-	a	$0B_H$

D[c] = (D[a] > $FFFF_H$) ? $FFFF_H$: D[a]; // unsigned comparison

SAT.HUD[a] (SR)

15 12	11 8	7 0
03_H	a	32_H

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

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.

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

31 28	27 24	23 21	20 12	11 8	7 0
c	d	04_H	const9	a	AB_H

D[c] = ((D[d] != 0) ? D[a] : sign_ext(const9));

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

31 28	27 24	23 20	19 18	17 16	15 12	11 8	7 0
c	d	04_H	-	-	b	a	$2B_H$

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

See Also

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

SELN
Select-Not

Description

If the contents of data register D[d] are zero, copy the contents of data register D[a] to data register D[c]; otherwise copy the contents of either D[b] or const9 to D[c].

The value const9 (instruction format RCR) is sign-extended.

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

31 28	27 24	23 21	20 12	11 8	7 0
c	d	05_H	const9	a	AB_H

D[c] = ((D[d] == 0) ? D[a] : sign_ext(const9));

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

31 28	27 24	23 20	19 18	17 16	15 12	11 8	7 0
c	d	05_H	-	-	b	a	$2B_H$

D[c] = ((D[d] == 0) ? D[a] : D[b]);

Status Flags

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

Examples

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

See Also

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

SH
Shift

Description

Shift the value in D[a] by the amount specified by shift count. If the shift count specified through the contents of either D[b] (instruction format RR) or const9 (instruction format RC) is greater than or equal to zero, then left-shift. Otherwise right-shift by the absolute value of the shift count. Put the result in D[c]. In both cases the vacated bits are filled with zeros and the bits shifted out are discarded.

The shift count is a 6-bit signed number, derived from either D[b][5:0] or const9[5:0]. The range for the shift count is therefore -32 to +31, allowing a shift left up to 31 bit positions and to shift right up to 32 bit positions (Note that a shift right by 32 bits leaves zeros in the result).

If the shift count specified through the value const4 is greater than or equal to zero, then left-shift the value in D[a] by the amount specified by the shift count. Otherwise right-shift the value in D[a] by the absolute value of the shift count. Put the result in D[a].
In both cases, the vacated bits are filled with zeros and bits shifted out are discarded.
The shift count is a 4-bit signed number, derived from the sign-extension of const4[3:0]. The resulting range for the shift count therefore is -8 to +7, allowing a shift left up to 7-bit positions and to shift right up to 8-bit positions.

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

31 28	27 21	20 12	11 8	7 0
c	00_H	const9	a	$8F_H$

D[c] = (const9[5:0] >= 0) ? D[a] << const9[5:0] : D[a] >> (-const9[5:0]);

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

31 28	27 20	19 18	17 16	15 12	11 8	7 0
c	00_H	-	-	b	a	$0F_H$

D[c] = (D[b][5:0] >= 0) ? D[a] << D[b][5:0] : D[a] >> (-D[b][5:0]);

SHD[a], const4 (SRC)

15 12	11 8	7 0
const4	a	06_H

shift_count = sign_ext(const4[3:0]);
D[a] = (shift_count >= 0) ? D[a] << shift_count : D[a] >> (-shift_count);

Status Flags

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

Examples

```
sh    d3, d1, d2
sh    d3, d1, #26
```

```
sh    d1, #6
```

See Also

SH.H, SHA, SHA.H, SHAS

SH.EQ
Shift Equal

Description

Left shift D[c] by one. If the contents of data register D[a] are equal to the contents of either data register D[b] (instruction format RR) or const9 (instruction format RC), set the least-significant bit of D[c] to one; otherwise set the least-significant bit of D[c] to 0.

The value const9 (format RC) is sign-extended.

SH.EQD[c], D[a], const9 (RC)

31 28	27 21	20 12	11 8	7 0
c	37_H	const9	a	$8B_H$

D[c] = {D[c][30:0], (D[a] == sign_ext(const9))};

SH.EQD[c], D[a], D[b] (RR)

31 28	27 20	19 18	17 16	15 12	11 8	7 0
c	37_H	-	-	b	a	$0B_H$

D[c] = {D[c][30:0], (D[a] == D[b])};

Status Flags

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

Examples

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

See Also

SH.GE, SH.GE.U, SH.LT, SH.LT.U, SH.NE

SH.GE
Shift Greater Than or Equal
SH.GE.U
Shift Greater Than or Equal Unsigned

Description

Left shift D[c] by one. If the contents of data register D[a] are greater than or equal to the contents of either data register D[b] (instruction format RR) or const9 (instruction format RC), set the least-significant bit of D[c] to one; otherwise set the least-significant bit of D[c] to 0. D[a] and D[b] are treated as signed (SH.GE) or unsigned (SH.GE.U) integers. The value const9 is sign-extended (SH.GE) or zero-extended (SH.GE.U).

SH.GED[c], D[a], const9 (RC)

31 28	27 21	20 12	11 8	7 0
c	$3B_H$	const9	a	$8B_H$

D[c] = {D[c][30:0], (D[a] >= sign_ext(const9))};

SH.GED[c], D[a], D[b] (RR)

31 28	27 20	19 18	17 16	15 12	11 8	7 0
c	$3B_H$	-	-	b	a	$0B_H$

D[c] = {D[c][30:0], (D[a] >= D[b])};

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

31 28	27 21	20 12	11 8	7 0
c	$3C_H$	const9	a	$8B_H$

D[c] = {D[c][30:0], (D[a] >= zero_ext(const9))}; // unsigned

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

31 28	27 20	19 18	17 16	15 12	11 8	7 0
c	$3C_H$	-	-	b	a	$0B_H$

D[c] = {D[c][30:0], (D[a] >= D[b])}; // unsigned

Status Flags

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

Examples

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

See Also

SH.EQ, **SH.LT**, **SH.LT.U**, **SH.NE**

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.HD[c], D[a], const9 (RC)

31 28	27 21	20 12	11 8	7 0
c	40_H	const9	a	$8F_H$

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.HD[c], D[a], D[b] (RR)

31 28	27 20	19 18	17 16	15 12	11 8	7 0
c	40_H	-	-	b	a	$0F_H$

shift_count = sign_ext(D[b][4:0]);

result_halfword1 = (shift_count >= 0) ? D[a][31:16] << shift_count : D[a][31:16] >> (0 - shift_count);

result_halfword0 = (shift_count >= 0) ? D[a][15:0] << shift_count : D[a][15:0] >> (0 - shift_count);

D[c] = {result_halfword1[15:0], result_halfword0[15:0]};

Status Flags

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

Examples

```
sh.h    d3, d1, d2
sh.h    d3, d1, #12
```

See Also

SH, **SHA**, **SHA.H**, **SHAS**

SH.LT
Shift Less Than
SH.LT.U
Shift Less Than Unsigned

Description

Left shift D[c] by one. If the contents of data register D[a] are less than the contents of either data register D[b] (instruction format RR) or const9 (instruction format RC), set the least-significant bit of D[c] to one; otherwise set the least-significant bit of D[c] to zero.

D[a] and either D[b] (format RR) or const9 (format RC) are treated as signed (SH.LT) or unsigned (SH.LT.U) integers. The value const9 is sign-extended (SH.LT) or zero-extended (SH.LT.U).

SH.LTD[c], D[a], const9 (RC)

31 28	27 21	20 12	11 8	7 0
c	39_H	const9	a	$8B_H$

D[c] = {D[c][30:0], (D[a] < sign_ext(const9))};

SH.LTD[c], D[a], D[b] (RR)

31 28	27 20	19 18	17 16	15 12	11 8	7 0
c	39_H	-	-	b	a	$0B_H$

D[c] = {D[c][30:0], (D[a] < D[b])};

SH.LT.UD[c], D[a], const9 (RC)

31 28	27 21	20 12	11 8	7 0
c	$3A_H$	const9	a	$8B_H$

D[c] = {D[c][30:0], (D[a] < zero_ext(const9))}; // unsigned

SH.LT.UD[c], D[a], D[b] (RR)

31 28	27 20	19 18	17 16	15 12	11 8	7 0
c	$3A_H$	-	-	b	a	$0B_H$

D[c] = {D[c][30:0], (D[a] < D[b])}; // unsigned

Status Flags

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

Examples

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

See Also

SH.EQ, **SH.GE**, **SH.GE.U**, **SH.NE**

SH.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.NED[c], D[a], const9 (RC)

31 28	27 21	20 12	11 8	7 0
c	38_H	const9	a	$8B_H$

D[c] = {D[c][30:0], (D[a] != sign_ext(const9))};

SH.NED[c], D[a], D[b] (RR)

31 28	27 20	19 18	17 16	15 12	11 8	7 0
c	38_H	-	-	b	a	$0B_H$

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

See Also

SH.EQ, **SH.GE**, **SH.GE.U**, **SH.LT**, **SH.LT.U**

SH.AND.T
Accumulating Shift-AND
SH.ANDN.T
Accumulating Shift-AND-Not
SH.NAND.T
Accumulating Shift-NAND
SH.NOR.T
Accumulating Shift-NOR
SH.OR.T
Accumulating Shift-OR
SH.ORN.T
Accumulating Shift-OR-Not
SH.XNOR.T
Accumulating Shift-XNOR
SH.XOR.T
Accumulating Shift-XOR

Description

Left shift D[c] by one. The bit shifted out is discarded. Compute the logical operation (AND, ANDN, NAND, NOR, OR, ORN, XNOR or XOR) of the value of bit pos1 of data register D[a], and bit pos2 of D[b]. Put the result in D[c][0].

SH.AND.TD[c], D[a], pos1, D[b], pos2 (BIT)

31 28	27 23	22 21	20 16	15 12	11 8	7 0
c	pos2	00_H	pos1	b	a	27_H

D[c] = {D[c][30:0], (D[a][pos1] AND D[b][pos2])};

SH.ANDN.TD[c], D[a], pos1, D[b], pos2 (BIT)

31 28	27 23	22 21	20 16	15 12	11 8	7 0
c	pos2	03_H	pos1	b	a	27_H

D[c] = {D[c][30:0], (D[a][pos1] AND !(D[b][pos2]))};

SH.NAND.TD[c], D[a], pos1, D[b], pos2 (BIT)

31 28	27 23	22 21	20 16	15 12	11 8	7 0
c	pos2	00_H	pos1	b	a	$A7_H$

D[c] = {D[c][30:0], !(D[a][pos1] AND D[b][pos2])};

SH.NOR.TD[c], D[a], pos1, D[b], pos2 (BIT)

31 28	27 23	22 21	20 16	15 12	11 8	7 0
c	pos2	02_H	pos1	b	a	27_H

D[c] = {D[c][30:0], !(D[a][pos1] OR D[b][pos2])};

SH.OR.TD[c], D[a], pos1, D[b], pos2 (BIT)

31 28	27 23	22 21	20 16	15 12	11 8	7 0
c	pos2	01_H	pos1	b	a	27_H

D[c] = {D[c][30:0], (D[a][pos1] OR D[b][pos2])};

SH.ORN.TD[c], D[a], pos1, D[b], pos2 (BIT)

31 28	27 23	22 21	20 16	15 12	11 8	7 0
c	pos2	01_H	pos1	b	a	$A7_H$

D[c] = {D[c][30:0], (D[a][pos1] OR !(D[b][pos2]))};

SH.XNOR.TD[c], D[a], pos1, D[b], pos2 (BIT)

31 28	27 23	22 21	20 16	15 12	11 8	7 0
c	pos2	02_H	pos1	b	a	$A7_H$

D[c] = {D[c][30:0], !(D[a][pos1] XOR D[b][pos2])};

SH.XOR.TD[c], D[a], pos1, D[b], pos2 (BIT)

31 28	27 23	22 21	20 16	15 12	11 8	7 0
c	pos2	03_H	pos1	b	a	$A7_H$

D[c] = {D[c][30:0], (D[a][pos1] XOR D[b][pos2])};

Status Flags

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

Examples

```
sh.and.t    d3, d1, 4, d2, 7
sh.andn.t   d3, d1, 4, d2, 7
sh.nand.t   d3, d1, 4, d2, 7
sh.nor.t    d3, d1, 4, d2, 7
sh.or.t     d3, d1, 4, d2, 7
sh.orn.t    d3, d1, 4, d2, 7
sh.xnor.t   d3, d1, 4, d2, 7
sh.xor.t    d3, d1, 4, d2, 7
```

See Also

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

SHA
Arithmetic Shift

Description

If shift count specified through contents of either D[b] (instruction format RR) or const9 (instruction format RC) is greater than or equal to zero, then left-shift the value in D[a] by the amount specified by shift count. The vacated bits are filled with zeros and bits shifted out are discarded. If the shift count is less than zero, right-shift the value in D[a] by the absolute value of the shift count. The vacated bits are filled with the sign-bit (the most significant bit) and bits shifted out are discarded. Put the result in D[c].

The shift count is a 6-bit signed number, derived from either D[b][5:0] or const9[5:0]. The range for shift count is therefore -32 to +31, allowing a shift left up to 31 bit positions and a shift right up to 32 bit positions (a shift right by 32 bits leaves all zeros or all ones in the result, depending on the sign bit). On all 1-bit or greater shifts (left or right), PSW.C is set to the logical-OR of the shifted out bits. On zero-bit shifts C is cleared.

If shift count specified through the value const4 is greater than or equal to zero, then left-shift the value in D[a] by the amount specified by the shift count. The vacated bits are filled with zeros and bits shifted out are discarded. If the shift count is less than zero, right-shift the value in D[a] by the absolute value of the shift count. The vacated bits are filled with the sign-bit (the most significant bit) and bits shifted out are discarded. Put the result in D[a]. The shift count is a 6-bit signed number, derived from the sign-extension of const4[3:0]. The resulting range for the shift count is therefore -8 to +7, allowing a shift left up to 7 bit positions, and a shift right up to 8 bit positions. On all shifts of 1-bit or greater (left or right), PSW.C is set to the logical-OR of the shifted out bits. On zero-bit shifts C is cleared.

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

31 28	27 21	20 12	11 8	7 0
c	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];
```

SHAD[a], const4 (SRC)

15 12	11 8	7 0
const4	a	86_H

```
if (const4[0:3] >= 0) then {
carry_out = const4[0:3] ? (D[a][31:32 - const4[0:3]] != 0) : 0;
result = D[a] << const4[0:3];
} else {
shift_count = 0 - const4[0:3];
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, d2
sha    d3, d1, #26
```

```
sha    d1, #6
```

See Also

SH, SH.H, SHAS, SHA.H

SHA.H
Arithmetic Shift Packed Half-words

Description

If the shift count specified through the contents of either D[b] (instruction format RR) or const9 (instruction format RC) is greater than or equal to zero, then left-shift each half-word in D[a] by the amount specified by shift count. The vacated bits are filled with zeros and bits shifted out are discarded. If the shift count is less than zero, right-shift each half-word in D[a] by the absolute value of the shift count. The vacated bits are filled with the sign-bit (the most significant bit) of the respective half-word, and bits shifted out are discarded. Put the result in D[c]. Note that for the shifts, each half-word is treated individually, and bits shifted out of a half-word are not shifted into the next half-word.

The shift count is a signed number, derived from the sign-extension of either D[b][4:0] (format RR) or const9[4:0] (format RC).

The range for the shift count is -16 to +15. The result for each half-word for a shift count of -16 is either all zeros or all ones, depending on the sign-bit of the respective half-word.

SHA.HD[c], D[a], const9 (RC)

31 28	27 21	20 12	11 8	7 0
c	41_H	const9	a	$8F_H$

```
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.HD[c], D[a], D[b] (RR)

31 28	27 20	19 18	17 16	15 12	11 8	7 0
c	41_H	-	-	b	a	$0F_H$

```
if (D[b][4:0] >= 0) then {
result_halfword0 = D[a][15:0] << D[b][4:0];
result_halfword1 = D[a][31:16] << D[b][4:0];
} else {
shift_count = 0 - D[b][4:0];
msk = D[a][31] ? (((1 << shift_count) - 1) << (16 - shift_count)) : 0;
result_halfword0 = msk | (D[a][15:0] >> shift_count);
result_halfword1 = msk | (D[a][31:16] >> shift_count);
}
D[c][15:0] = result_halfword0[15:0];
D[c][31:16] = result_halfword1[15:0];
```

Status Flags

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

Examples

```
sha.h   d3, d1, d2
sha.h   d3, d1, #12
```

See Also

SH, **SHA**, **SHAS**, **SH.H**

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.

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

31 28	27 21	20 12	11 8	7 0
c	02_H	const9	a	$8F_H$

```
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);
```

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

31 28	27 20	19 18	17 16	15 12	11 8	7 0
c	02_H	-	-	b	a	$0F_H$

```
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, d2
shas    d3, d1, #26
```

See Also

SH, **SH.H**, **SHA**, **SHA.H**

ST.A
Store Word from Address Register

Description

Store the value in address register A[a] to the memory location specified by the addressing mode.

Note: If the source register is modified by the addressing mode, the value stored to memory is undefinded.

Store the value in address register A[a] (instruction format BO, SSR, SSRO or SSR) or A[15] (instruction format SRO or SC) to the memory location specified by the addressing mode.

Note: If the source register is modified by the addressing mode, the value stored to memory is undefined.

ST.Aoff18, A[a] (ABS)(Absolute Addressing Mode)

31 28	27 26	25 22	21 16	15 12	11 8	7 0
off18[9:6]	02_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) = A[a];

ST.AA[b], off10, A[a] (BO)(Base + Short Offset Addressing Mode)

31 28	27 22	21 16	15 12	11 8	7 0
off10[9:6]	26_H	off10[5:0]	b	a	89_H

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

M(EA, word) = A[a];

ST.AP[b], A[a] (BO)(Bit-reverse Addressing Mode)

31 28	27 22	21 16	15 12	11 8	7 0
-	06_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) = A[a];

new_index = reverse16(reverse16(index) + reverse16(incr));

A[b+1] = {incr[15:0], new_index[15:0]};

ST.AP[b], off10, A[a] (BO)(Circular Addressing Mode)

31 28	27 22	21 16	15 12	11 8	7 0
off10[9:6]	16_H	off10[5:0]	b	a	$A9_H$

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.AA[b], off10, A[a] (BO)(Post-increment Addressing Mode)

31 28	27 22	21 16	15 12	11 8	7 0
off10[9:6]	06_H	off10[5:0]	b	a	89_H

EA = A[b];
M(EA, word) = A[a];
A[b] = EA + sign_ext(off10);

ST.AA[b], off10, A[a] (BO)(Pre-increment Addressing Mode)

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

EA = A[b] + sign_ext(off10);
M(EA, word) = A[a];
A[b] = EA;

ST.AA[b], off16, A[a] (BOL)(Base + Long Offset Addressing Mode)

31 28	27 22	21 16	15 12	11 8	7 0
off16[9:6]	off16[15:10]	off16[5:0]	b	a	$B5_H$

EA = A[b] + sign_ext(off16);
M(EA, word) = A[a];

ST.AA[10], const8, A[15] (SC)

15 8	7 0
const8	$F8_H$

M(A[10] + zero_ext(4 * const8), word) = A[15];

ST.AA[b], off4, A[15] (SRO)

15 12	11 8	7 0
b	off4	EC_H

M(A[b] + zero_ext(4 * off4), word) = A[15];

ST.AA[b], A[a] (SSR)

15 12	11 8	7 0
b	a	$F4_H$

M(A[b], word) = A[a];

ST.AA[b], A[a] (SSR)(Post-increment Addressing Mode)

15 12	11 8	7 0
b	a	$E4_H$

```
M(A[b], word) = A[a];
A[b] = A[b] + 4;
```

ST.AA[15], off4, A[a] (SSRO)

15 12	11 8	7 0
off4	a	$E8_H$

```
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], a0
st.a    [a15]+4, a2
```

See Also

ST.B, **ST.D**, **ST.DA**, **ST.H**, **ST.Q**, **ST.W**

ST.B
Store Byte

Description

Store the byte value in the eight least-significant bits of data register D[a] to the byte memory location specified by the addressing mode.

Store the byte value in the eight least-significant bits of either data register D[a] (instruction format SSR, SSR0 or BO) or D[15] (instruction format SRO) to the byte memory location specified by the addressing mode.

ST.Boff18, D[a] (ABS)(Absolute Addressing Mode)

31 28	27 26	25 22	21 16	15 12	11 8	7 0
off18[9:6]	00_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.BA[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)

31 28	27 22	21 16	15 12	11 8	7 0
-	00_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, byte) = D[a][7:0];
new_index = reverse16(reverse16(index) + reverse16(incr));
A[b+1] = {incr[15:0], new_index[15:0]};
```

ST.BP[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	$A9_H$

```
index = zero_ext(A[b+1][15:0]);
length = zero_ext(A[b+1][31:16]);
EA0 = A[b] + index;
M(EA, byte) = D[a][7:0];
new_index = index + sign_ext(off10);
new_index = new_index < 0 ? new_index + length : new_index % length;
```

A[b+1] = {length[15:0], new_index[15:0]};

ST.BA[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.BA[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.BA[b], off16, D[a] (BOL)(Base + Long Offset Addressing Mode)

31 28	27 22	21 16	15 12	11 8	7 0
off16[9:6]	off16[15:10]	off16[5:0]	b	a	$E9_H$

EA = A[b] + sign_ext(off16);

M(EA, byte) = D[a][7:0];

ST.BA[b], off4, D[15] (SRO)

15 12	11 8	7 0
b	off4	$2C_H$

M(A[b] + zero_ext(off4), byte) = D[15][7:0];

ST.BA[b], D[a] (SSR)

15 12	11 8	7 0
b	a	34_H

M(A[b], byte) = D[a][7:0];

ST.BA[b], D[a] (SSR)(Post-increment Addressing Mode)

15 12	11 8	7 0
b	a	24_H

M(A[b], byte) = D[a][7:0];
A[b] = A[b] + 1;

ST.BA[15], off4, D[a] (SSRO)

15 12	11 8	7 0
off4	a	28_H

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    [a3]+24, d2
```

```
st.b    [a0], d0
st.b    [a15]+14, d2
```

See Also

ST.A, **ST.D**, **ST.DA**, **ST.H**, **ST.Q**, **ST.W**

ST.D
Store Double-word

Description

Store the value in the extended data register pair E[a] to the memory location specified by the addressing mode. The value in the even register D[n] is stored in the least-significant memory word, and the value in the odd register (D[n+1]) is stored in the most-significant memory word.

ST.Doff18, E[a] (ABS)(Absolute Addressing Mode)

31 28	27 26	25 22	21 16	15 12	11 8	7 0
off18[9:6]	01_H	off18[13:10]	off18[5:0]	off18[17:14]	a	$A5_H$

EA = {off18[17:14], 14b'0, off18[13:0]};

M(EA, doubleword) = E[a];

ST.DA[b], off10, E[a] (BO)(Base + Short Offset Addressing Mode)

31 28	27 22	21 16	15 12	11 8	7 0
off10[9:6]	25_H	off10[5:0]	b	a	89_H

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

M(EA, doubleword) = E[a];

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

31 28	27 22	21 16	15 12	11 8	7 0
-	05_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, doubleword) = E[a];

new_index = reverse16(reverse16(index) + reverse16(incr));

A[b+1] = {incr[15:0], new_index[15:0]};

ST.DP[b], off10, E[a] (BO)(Circular Addressing Mode)

31 28	27 22	21 16	15 12	11 8	7 0
off10[9:6]	15_H	off10[5:0]	b	a	$A9_H$

index = zero_ext(A[b+1][15:0]);

length = zero_ext(A[b+1][31:16]);

EA0 = A[b] + index;

EA2 = A[b] + (index + 2) % length;

EA4 = A[b] + (index + 4) % length;

EA6 = A[b] + (index + 6) % length;

M(EA0, halfword) = D[a][15:0];

M(EA2, halfword) = D[a][31:16];

M(EA4, halfword) = D[a+1][15:0];
M(EA6, halfword) = D[a+1][31:16];
new_index = index + sign_ext(off10);
new_index = new_index < 0 ? new_index + length : new_index % length;
A[b+1] = {length[15:0], new_index[15:0]};

ST.DA[b], off10, E[a] (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	89_H

EA = A[b];
M(EA, doubleword) = E[a];
A[b] = EA + sign_ext(off10);

ST.DA[b], off10, E[a] (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	89_H

EA = A[b] + sign_ext(off10);
M(EA, doubleword) = E[a];
A[b] = EA;

Status Flags

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

Examples

```
st.d    [a0], e0
st.d    [a0], d0/d1
st.d    [a15+]8, e12
st.d    [a15+]8, d12/d13
```

See Also

ST.A, **ST.B**, **ST.DA**, **ST.H**, **ST.Q**, **ST.W**

ST.DA
Store Double-word from Address Registers

Description

Store the value in the address register pair A[a]/A[a+1] to the memory location specified by the addressing mode. The value in the even register A[a] is stored in the least-significant memory word, and the value in the odd register (A[a+1]) is stored in the most-significant memory word.

ST.DAoff18, P[a] (ABS)(Absolute Addressing Mode)

31 28	27 26	25 22	21 16	15 12	11 8	7 0
off18[9:6]	03_H	off18[13:10]	off18[5:0]	off18[17:14]	a	$A5_H$

EA = {off18[17:14], 14b'0, off18[13:0]};

M(EA, doubleword) = P[a];

ST.DAA[b], off10, P[a] (BO)(Base + Short Offset Addressing Mode)

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

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

M(EA, doubleword) = P[a];

ST.DAP[b], P[a] (BO)(Bit-reverse Addressing Mode)

31 28	27 22	21 16	15 12	11 8	7 0
-	07_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, doubleword) = P[a];

new_index = reverse16(reverse16(index) + reverse16(incr));

A[b+1] = {incr[15:0], new_index[15:0]};

ST.DAP[b], off10, P[a] (BO)(Circular Addressing Mode)

31 28	27 22	21 16	15 12	11 8	7 0
off10[9:6]	17_H	off10[5:0]	b	a	$A9_H$

index = zero_ext(A[b+1][15:0]);

length = zero_ext(A[b+1][31:16]);

EA0 = A[b] + index;

EA4 = A[b] + (index + 4) % length;

(M(EA0, word) = A[a];

(M(EA4, word) = A[a+1];

new_index = index + sign_ext(off10);

new_index = new_index < 0 ? new_index + length : new_index % length;

A[b+1] = {length[15:0], new_index[15:0]};

ST.DAA[b], off10, P[a] (BO)(Post-increment Addressing Mode)

31 28	27 22	21 16	15 12	11 8	7 0
off10[9:6]	07_H	off10[5:0]	b	a	89_H

EA = A[b];

M(EA, doubleword) = P[a];

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

ST.DAA[b], off10, P[a] (BO)(Pre-increment Addressing Mode)

31 28	27 22	21 16	15 12	11 8	7 0
off10[9:6]	17_H	off10[5:0]	b	a	89_H

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

M(EA, doubleword) = P[a];

A[b] = EA;

Status Flags

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

Examples

```
st.da   [a6]+8, a4/a5
st.da   _savedPointerBuffer, a0/a1
```

See Also

ST.A, **ST.B**, **ST.D**, **ST.H**, **ST.Q**, **ST.W**

ST.H
Store Half-word

Description

Store the half-word value in the 16 least-significant bits of data register D[a] to the half-word memory location specified by the addressing mode.

Store the half-word value in the 16 least-significant bits of either data register D[a] (instruction format or D[15] to the half-word memory location specified by the addressing mode.

ST.Hoff18, D[a] (ABS)(Absolute Addressing Mode)

31 28	27 26	25 22	21 16	15 12	11 8	7 0
off18[9:6]	02_H	off18[13:10]	off18[5:0]	off18[17:14]	a	25_H

EA = {off18[17:14], 14b'0, off18[13:0]};

M(EA, halfword) = D[a][15:0];

ST.HA[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]	22_H	off10[5:0]	b	a	89_H

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

M(EA, halfword) = D[a][15:0];

ST.H P[b], D[a] (BO)(Bit-reverse Addressing Mode)

31 28	27 22	21 16	15 12	11 8	7 0
-	02_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, 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)

31 28	27 22	21 16	15 12	11 8	7 0
off10[9:6]	12_H	off10[5:0]	b	a	$A9_H$

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.HA[b], off10, D[a] (BO)(Post-increment Addressing Mode)

31 28	27 22	21 16	15 12	11 8	7 0
off10[9:6]	02_H	off10[5:0]	b	a	89_H

EA = A[b];

M(EA, halfword) = D[a][15:0];

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

ST.HA[b], off10, D[a] (BO)(Pre-increment Addressing Mode)

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

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

M(EA, halfword) = D[a][15:0];

A[b] = EA;

ST.HA[b], off16, D[a] (BOL)(Base + Long Offset Addressing Mode)

31 28	27 22	21 16	15 12	11 8	7 0
off16[9:6]	off16[15:10]	off16[5:0]	b	a	$F9_H$

EA = A[b] + sign_ext(off16);

M(EA, halfword) = D[a][15:0];

ST.HA[b], off4, D[15] (SRO)

15 12	11 8	7 0
b	off4	AC_H

M(A[b] + zero_ext(2 * off4), half-word) = D[15][15:0];

ST.HA[b], D[a] (SSR)

15 12	11 8	7 0
b	a	$B4_H$

M(A[b], half-word) = D[a][15:0];

ST.HA[b], D[a] (SSR)(Post-increment Addressing Mode)

15 12	11 8	7 0
b	a	$A4_H$

M(A[b], half-word) = D[a][15:0];
A[b] = A[b] + 2;

ST.HA[15], off4, D[a] (SSRO)

15 12	11 8	7 0
off4	a	A8$_H$

```
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+]8, d0
st.h    [a0+]24, d2
```

```
st.h    [a0], d0
```

See Also

ST.A, **ST.B**, **ST.D**, **ST.DA**, **ST.Q**, **ST.W**

ST.Q
Store Half-word Signed Fraction

Description

Store the value in the most-significant half-word of data register D[a] to the memory location specified by the addressing mode.

ST.Qoff18, D[a] (ABS)(Absolute Addressing Mode)

31 28	27 26	25 22	21 16	15 12	11 8	7 0
off18[9:6]	00_H	off18[13:10]	off18[5:0]	off18[17:14]	a	65_H

EA = {off18[17:14],14b'0,off18[13:0]};

M(EA, halfword) = D[a][31:16];

ST.QA[b], off10, D[a] (BO)(Base + Short Offset Addressing Mode)

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

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

M(EA, halfword) = D[a][31:16];

ST.Q P[b], D[a] (BO)(Bit-reverse Addressing Mode)

31 28	27 22	21 16	15 12	11 8	7 0
-	08_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, halfword) = D[a][31:16];

new_index = reverse16(reverse16(index) + reverse16(incr));

A[b+1] = {incr[15:0], new_index[15:0]};

ST.Q P[b], off10, D[a] (BO)(Circular Addressing Mode)

31 28	27 22	21 16	15 12	11 8	7 0
off10[9:6]	18_H	off10[5:0]	b	a	$A9_H$

index = zero_ext(A[b+1][15:0]);

length = zero_ext(A[b+1][31:16]);

EA = A[b] + index;

M(EA, halfword) = D[a][31:16];

new_index = index + sign_ext(off10);

new_index = new_index < 0 ? new_index + length : new_index % length;

A[b+1] = {length[15:0], new_index[15:0]};

ST.QA[b], off10, D[a] (BO)(Post-increment Addressing Mode)

31 28	27 22	21 16	15 12	11 8	7 0
off10[9:6]	08_H	off10[5:0]	b	a	89_H

```
EA = A[b];
M(EA, halfword) = D[a][31:16];
A[b] = EA + sign_ext(off10);
```

ST.QA[b], off10, D[a] (BO)(Pre-increment Addressing Mode)

31 28	27 22	21 16	15 12	11 8	7 0
off10[9:6]	18_H	off10[5:0]	b	a	89_H

```
EA = A[b] + sign_ext(off10);
M(EA, halfword) = D[a][31:16];
A[b] = EA;
```

Status Flags

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

Examples

```
st.q    [a0+]2, d0
st.q    [a0+]22, d0
```

See Also

ST.A, ST.B, ST.D, ST.DA, ST.H, ST.W

ST.T
Store Bit

Description

Store the bit value b to the byte at the memory address specified by off18, in the bit position specified by bpos3. The other bits of the byte are unchanged. Individual bits can be used as semaphore.

ST.Toff18, bpos3, b (ABSB)

31 28	27 26	25 22	21 16	15 12	11	10 8	7 0
off18[9:6]	00_H	off18[13:10]	off18[5:0]	off18[17:14]	b	bpos3	$D5_H$

EA = {off18[17:14], 14’b0, off18[13:0]};

M(EA, byte) = (M(EA, byte) AND ~(1 << bpos3)) | (b << bpos3);

Status Flags

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

Examples

st.t 90000000_H, $\#7_H$, $\#1_H$

See Also

IMASK, **LDMST**, **SWAP.W**

ST.W
Store Word

Description

Store the word value in data register D[a] to the memory location specified by the addressing mode.

Store the word value in either data register D[a] (instruction format SSR, SSRO) or D[15] (instruction format SRO, SC) to the memory location specified by the addressing mode.

ST.Woff18, D[a] (ABS)(Absolute Addressing Mode)

31 28	27 26	25 22	21 16	15 12	11 8	7 0
off18[9:6]	00_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.WA[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.WP[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.WP[b], off10, D[a] (BO)(Circular Addressing Mode)

31 28	27 22	21 16	15 12	11 8	7 0
off10[9:6]	14_H	off10[5:0]	b	a	$A9_H$

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.WA[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.WA[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.WA[b], off16, D[a] (BOL)(Base + Long Offset Addressing Mode)

31 28	27 22	21 16	15 12	11 8	7 0
off16[9:6]	off16[15:10]	off16[5:0]	b	a	59_H

EA = A[b] + sign_ext(off16);

M(EA, word) = D[a];

ST.WA[10], const8, D[15] (SC)

15 8	7 0
const8	78_H

M(A[10] + zero_ext(4 * const8), word) = D[15];

ST.WA[b], off4, D[15] (SRO)

15 12	11 8	7 0
b	off4	$6C_H$

M(A[b] + zero_ext(4 * off4), word) = D[15];

ST.WA[b], D[a] (SSR)

15 12	11 8	7 0
b	a	74_H

M(A[b], word) = D[a];

ST.WA[b], D[a] (SSR)(Post-increment Addressing Mode)

15 12	11 8	7 0
b	a	64_H

```
M(A[b], word) = D[a];
A[b] = A[b] + 4;
```

ST.WA[15], off4, D[a] (SSRO)

15 12	11 8	7 0
off4	a	68_H

```
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+]2, d0
st.w    [a0+]22, d0
```

See Also

ST.A, **ST.B**, **ST.D**, **ST.DA**, **ST.H**, **ST.Q**

STLCX
Store Lower Context

Description

Store the contents of registers A[2] to A[7], D[0] to D]7], A[11] (return address) and PCXI, to the memory block specified by the addressing mode. For this instruction, the addressing mode is limited to absolute (ABS) or base plus short offset (BO).

Note: The effective address (EA) specified by the addressing mode must be aligned on a 16-word boundary.

STLCXoff18 (ABS)(Absolute Addressing Mode)

31 28	27 26	25 22	21 16	15 12	11 8	7 0
off18[9:6]	00_H	off18[13:10]	off18[5:0]	off18[17:14]	-	15_H

EA = {off18[17:14], 14b'0, off18[13:0]};

M(EA,16-word) = {PCXI, A[11], A[2:3], D[0:3], A[4:7], D[4:7]};

STLCXA[b], off10 (BO)(Base + Short Index Addressing Mode)

31 28	27 22	21 16	15 12	11 8	7 0
off10[9:6]	26_H	off10[5:0]	b	-	49_H

EA = A[b] + sign_ext(off10)[9:0]};

M(EA,16-word) = {PCXI, A[11], A[2:3], D[0:3], A[4:7], D[4:7]};

Status Flags

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

Examples

–

See Also

LDLCX, **LDUCX**, **RSLCX**, **STUCX**, **SVLCX**, **BISR**

STUCX
Store Upper Context

Description

Store the contents of registers A[10] to A[15], D[8] to D[15], and the current PSW (the registers which comprise a task’s upper context) to the memory block specified by the addressing mode. For this instruction, the addressing mode is limited to absolute (ABS) or base plus short offset (BO).

Note: The effective address (EA) specified by the addressing mode must be aligned on a 16-word boundary.

STUCXoff18 (ABS)(Absolute Addressing Mode)

31 28	27 26	25 22	21 16	15 12	11 8	7 0
off18[9:6]	01_H	off18[13:10]	off18[5:0]	off18[17:14]	-	15_H

EA = {off18[17:14], 14b'0, off18[13:0]};

M(EA,16-word) = {PCXI, PSW, A[10:11], D[8:11], A[12:15], D[12:15]};

STUCXA[b], off10 (BO)(Base + Short Index Addressing Mode)

31 28	27 22	21 16	15 12	11 8	7 0
off10[9:6]	27_H	off10[5:0]	b	-	49_H

EA = A[b] + sign_ext(off10)[9:0]};

M(EA,16-word) = {PCXI, PSW, A[10:11], D[8:11], A[12:15], D[12:15]};

Status Flags

C	PSW.C is read by the instruction but not changed.
V	PSW.V is read by the instruction but not changed.
SV	PSW.SV is read by the instruction but not changed.
AV	PSW.AV is read by the instruction but not changed.
SAV	PSW.SAV is read by the instruction but not changed.

Examples

–

See Also

LDLCX, **LDUCX**, **RSLCX**, **STLCX**, **SVLCX**, **STUCX**

SUB
Subtract

Description

Subtract the contents of data register D[b] from the contents of data register D[a] and put the result in data register D[c]. The operands are treated as 32-bit integers.

Subtract the contents of data register D[b] from the contents of either data register D[a] or D[15] and put the result in either data register D[a] or D[15]. The operands are treated as 32-bit integers.

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

31 28	27 20	19 18	17 16	15 12	11 8	7 0
c	08_H	-	-	b	a	$0B_H$

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

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

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

15 12	11 8	7 0
b	a	$A2_H$

result = D[a] - D[b];
D[a] = result[31:0];

SUBD[a], D[15], D[b] (SRR)

15 12	11 8	7 0
b	a	52_H

result = D[15] - D[b];
D[a] = result[31:0];

SUBD[15], D[a], D[b] (SRR)

15 12	11 8	7 0
b	a	$5A_H$

result = D[a] - D[b];
D[15] = 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, d2
sub   d15, d1, d2
sub   d1, d15, d2
```

See Also

SUBS, **SUBS.U**, **SUBX**, **SUBC**

SUB.A
Subtract Address

Description

Subtract the contents of address register A[b] from the contents of address register A[a] and put the result in address register A[c].

Decrement the Stack Pointer (A[10]) by the zero-extended value of const8 (a range of 0 through to 255).

SUB.AA[c], A[a], A[b] (RR)

31 28	27 20	19 18	17 16	15 12	11 8	7 0
c	02_H	-	-	b	a	01_H

A[c] = A[a] - A[b];

SUB.AA[10], const8 (SC)

15 8	7 0
const8	20_H

A[10] = A[10] - zero_ext(const8);

Status Flags

C	Not set by this instruction.
V	Not set by this instruction.
SV	Not set by this instruction.
AV	Not set by this instruction.
SAV	Not set by this instruction.

Examples

```
sub.a   a3, a4, a2
```

```
sub.a   sp, #126
```

See Also

ADD.A, **ADDIH.A**, **ADDSC.A**, **ADDSC.AT**

SUB.B
Subtract Packed Byte
SUB.H
Subtract Packed Half-word

Description

Subtract the contents of each byte or half-word of data register D[b] from the contents of data register D[a]. Put the result in each corresponding byte or half-word of data register D[c].

SUB.BD[c], D[a], D[b] (RR)

31 28	27 20	19 18	17 16	15 12	11 8	7 0
c	48_H	-	-	b	a	$0B_H$

result_byte3 = D[a][31:24] - D[b][31:24];

result_byte2 = D[a][23:16] - D[b][23:16];

result_byte1 = D[a][15:8] - D[b][15:8];

result_byte0 = D[a][7:0] - D[b][7:0];

D[c] = {result_byte3[7:0], result_byte2[7:0], result_byte1[7:0], result_byte0[7:0]};

SUB.HD[c], D[a], D[b] (RR)

31 28	27 20	19 18	17 16	15 12	11 8	7 0
c	68_H	-	-	b	a	$0B_H$

result_halfword1 = D[a][31:16] - D[b][31:16];

result_halfword0 = D[a][15:0] - D[b][15:0];

D[c] = {result_halfword1[15:0], result_halfword0[15:0]};

Status Flags

C	Not set by these instructions.

V	SUB.B
	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;
	SUB.H
	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	SUB.B
	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;
	SUB.H
	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.b   d3, d1, d2
sub.h   d3, d1, d2
```

See Also

SUBS.H, **SUBS.HU**

SUBC
Subtract With Carry

Description

Subtract the contents of data register D[b] from contents of data register D[a] plus the carry bit minus one. Put the result in data register D[c]. The operands are treated as 32-bit integers. The PSW carry bit is set to the value of the ALU carry out.

SUBCD[c], D[a], D[b] (RR)

31 28	27 20	19 18	17 16	15 12	11 8	7 0
c	$0D_H$	-	-	b	a	$0B_H$

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

SUB, **SUBS**, **SUBS.U**, **SUBX**

SUBS
Subtract Signed with Saturation
SUBS.U
Subtract Unsigned with Saturation

Description

Subtract the contents of data register D[b] from the contents of data register D[a] and put the result in data register D[c]. The operands are treated as signed (SUBS) or unsigned (SUBS.U) 32-bit integers, with saturation on signed (SUBS) or (SUBS.U) unsigned overflow.

Subtract the contents of data register D[b] from the contents of data register D[a] and put the result in data register D[a].The operands are treated as signed 32-bit integers, with saturation on signed overflow.

SUBSD[c], D[a], D[b] (RR)

31 28	27 20	19 18	17 16	15 12	11 8	7 0
c	$0A_H$	-	-	b	a	$0B_H$

result = D[a] - D[b];

D[c] = ssov(result, 32);

SUBSD[a], D[b] (SRR)

15 12	11 8	7 0
b	a	62_H

result = D[a] - D[b];
D[a] = ssov(result, 32);

SUBS.UD[c], D[a], D[b] (RR)

31 28	27 20	19 18	17 16	15 12	11 8	7 0
c	$0B_H$	-	-	b	a	$0B_H$

result = D[a] - D[b];

D[c] = suov(result, 32);

Status Flags

C	Not set by these instructions.

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

```
subs    d3, d1, d2
subs.u  d3, d1, d2
```

```
subs    d3, d1
```

See Also

SUB, **SUBX**, **SUBC**

SUBS.H
Subtract Packed Half-word with Saturation
SUBS.HU
Subtract Packed Half-word Unsigned with Saturation

Description

Subtract the contents of each half-word of data register D[b] from the contents of data register D[a]. Put the result in each corresponding half-word of data register D[c], with saturation on signed (SUBS.H) or unsigned (SUBS.HU) overflow.

SUBS.HD[c], D[a], D[b] (RR)

31 28	27 20	19 18	17 16	15 12	11 8	7 0
c	$6A_H$	-	-	b	a	$0B_H$

result_halfword1 = D[a][31:16] - D[b][31:16];

result_halfword0 = D[a][15:0] - D[b][15:0];

D[c] = {ssov(result_halfword1, 16), ssov(result_halfword0, 16)};

SUBS.HUD[c], D[a], D[b] (RR)

31 28	27 20	19 18	17 16	15 12	11 8	7 0
c	$6B_H$	-	-	b	a	$0B_H$

result_halfword1 = D[a][31:16] - D[b][31:16];

result_halfword0 = D[a][15:0] - D[b][15:0];

D[c] = {suov(result_halfword1, 16), suov(result_halfword0, 16)};

Status Flags

C	Not set by these instructions.

V	signed:
	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;
	unsigned:
	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.h   d3, d1, d2
subs.hu   d3, d1, d2
```

See Also

SUB.B, **SUB.H**

SUBX
Subtract Extended

Description

Subtract the contents of data register D[b] from the contents of data register D[a] and put the result in data register D[c]. The operands are treated as 32-bit integers. The PSW carry bit is set to the value of the ALU carry out.

SUBXD[c], D[a], D[b] (RR)

31 28	27 20	19 18	17 16	15 12	11 8	7 0
c	$0C_H$	-	-	b	a	$0B_H$

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

SUB, **SUBC**, **SUBS**, **SUBS.U**

SVLCX
Save Lower Context

Description

Store the contents of registers A[2] to A[7], D[0] to D[7], A[11] (return address) and PCXI, to the memory location pointed to by the FCX register. This operation saves the lower context of the currently executing task.

SVLCX(SYS)

31 28	27 22	21 12	11 8	7 0
-	08_H	-	-	$0D_H$

```
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**

SWAP.W
Swap with Data Register

Description

Swap atomically the contents of data register D[a] and the memory word specified by the addressing mode.

SWAP.Woff18, D[a] (ABS)(Absolute Addressing Mode)

31 28	27 26	25 22	21 16	15 12	11 8	7 0
off18[9:6]	00_H	off18[13:10]	off18[5:0]	off18[17:14]	a	$E5_H$

```
EA = {off18[17:14], 14b'0, off18[13:0]};
tmp = M(EA, word);
M(EA, word) = D[a];
D[a] = tmp[31:0];
```

SWAP.WA[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	49_H

```
EA = A[b] + sign_ext(off10);
tmp = M(EA, word);
M(EA, word) = D[a];
D[a] = tmp[31:0];
```

SWAP.WP[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.WP[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]};
```

SWAP.WA[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	49_H

```
EA = A[b];
tmp = M(EA, word);
M(EA, word) = D[a];
D[a] = tmp[31:0];
A[b] = EA + sign_ext(off10);
```

SWAP.WA[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	49_H

```
EA = A[b] + sign_ext(off10);
tmp = M(EA, word);
M(EA, word) = D[a];
D[a] = tmp[31:0];
A[b] = EA;
```

Status Flags

C	Not set by this instruction.
V	Not set by this instruction.
SV	Not set by this instruction.
AV	Not set by this instruction.
SAV	Not set by this instruction.

Examples

–

See Also

ST.T, **LDMST**

SYSCALL
System Call

Description

Cause a system call trap, using Trap Identification Number (TIN) specified by const9.

Note: The trap return PC will be the instruction following the SYSCALL instruction.

SYSCALLconst9 (RC)

31 28	27 21	20 12	11 8	7 0
-	04_H	const9	-	AD_H

trap(SYS, const9[7:0]);

Status Flags

C	PSW.C is read, but not set by the instruction.
V	PSW.V is read, but not set by the instruction.
SV	PSW.SV is read, but not set by the instruction.
AV	PSW.AV is read, but not set by the instruction.
SAV	PSW.SAV is read, but not set by the instruction.

Examples

```
syscall   4
```

See Also

RET, **RFE**, **TRAPV**, **TRAPSV**, **UNPACK**

TRAPSV
Trap on Sticky Overflow

Description

If the PSW sticky overflow status flag (PSW.SV) is set, generate a trap to the vector entry for the sticky overflow trap handler (SOV-trap).

TRAPSV(SYS)

31 28	27 22	21 12	11 8	7 0
-	15_H	-	-	$0D_H$

if PSW.SV == 1 then trap(SOVF);

Status Flags

C	Not set by this instruction.
V	Not set by this instruction.
SV	PSW.SV is read, but not set by this instruction.
AV	Not set by this instruction.
SAV	Not set by this instruction.

Examples

```
trapsv
```

See Also

RSTV, **SYSCALL**, **TRAPV**

TRAPV
Trap on Overflow

Description

If the PSW overflow status flag (PSW.V) is set, generate a trap to the vector entry for the overflow trap handler (OVF trap).

TRAPV(SYS)

31 28	27 22	21 12	11 8	7 0
-	14_H	-	-	$0D_H$

if PSW.V then trap(OVF);

Status Flags

C	Not set by this instruction.
V	PSW.V is read, but not set by this instruction.
SV	Not set by this instruction.
AV	Not set by this instruction.
SAV	Not set by this instruction.

Examples

```
trapv
```

See Also

RSTV, **SYSCALL**, **TRAPSV**, **UNPACK**

UNPACK
Unpack Floating Point

Description

Take an IEEE 754 single precision floating point number in data register D[a] and unpack it as exponent and mantissa into data register pair E[c], such that it can be more easily processed through regular instructions.

The odd register E[c][63:32] receives the unbiased exponent. The even register E[c][31:0] receives the mantissa. Note that the sign-bit of the floating point number is available in bit 31 of data register D[a].

To compute the mantissa and the exponent, the input number is first checked for special cases: Infinity, NAN, Zero & Denormalised. If the input number is not one of these special cases it is a normalised number. Bits [22:0] of D[a] are then copied to bits [29:7] of E[c], with bits [6:0] of E[c] cleared to 0. Bit 30 is set to one, as the implicit high order bit for a 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 denormalised input numbers the output mantissa is in a fractional 2.30 format.

The special cases are handled as shown in the operation, described below.

UNPACKE[c], D[a] (RR)

31 28	27 20	19 18	17 16	15 12	11 8	7 0
c	08_H	-	0_H	-	a	$4B_H$

```
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 {
    // Denormalised
    int_exp = -126;
    int_mant = {2'b00, fp_frac[22:0], 7'b0000000};
} else {
    // Normalized
    int_exp = fp_exp - 127;
    int_mant = {2'b01, fp_frac[22:0], 7'b0000000};
}
E[c][63:32] = int_exp;
E[c][31:0] = int_mant;
```

Status Flags

C	Not set by this instruction.
V	Not set by this instruction.
SV	Not set by this instruction.
AV	Not set by this instruction.
SAV	Not set by this instruction.

Examples

```
unpack   e2, d5
```

See Also

PACK, **SYSCALL**, **TRAPV**

XNOR
Bitwise XNOR

Description

Compute the bitwise exclusive NOR of the contents of data register D[a] and the contents of either data register D[b] (instruction format RR) or const9 (instruction format RC). Put the result in to data register D[c]. The value const9 is zero-extended.

XNORD[c], D[a], const9 (RC)

31 28	27 21	20 12	11 8	7 0
c	$0D_H$	const9	a	$8F_H$

D[c] = ~(D[a] ^ zero_ext(const9));

XNORD[c], D[a], D[b] (RR)

31 28	27 20	19 18	17 16	15 12	11 8	7 0
c	$0D_H$	-	-	b	a	$0F_H$

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, d2
xnor   d3, d1, #126
```

See Also

AND, **ANDN**, **NAND**, **NOR**, **NOT (16-bit)**, **OR**, **ORN**, **XOR**

XNOR.T
Bit Logical XNOR

Description

Compute the logical exclusive NOR of bit pos1 of data register D[a] and bit pos2 of data register D[b]. Put the result in the least-significant bit of data register D[c] and clear the remaining bits of D[c] to zero.

XNOR.TD[c], D[a], pos1, D[b], pos2 (BIT)

31 28	27 23	22 21	20 16	15 12	11 8	7 0
c	pos2	02_H	pos1	b	a	07_H

result = !(D[a][pos1] XOR D[b][pos2]);

D[c] = zero_ext(result);

Status Flags

C	Not set by this instruction.
V	Not set by this instruction.
SV	Not set by this instruction.
AV	Not set by this instruction.
SAV	Not set by this instruction.

Examples

```
xnor.t   d3, d1, 3, d2, 5
```

See Also

AND.T, **ANDN.T**, **NAND.T**, **NOR.T**, **OR.T**, **ORN.T**, **XOR.T**

XOR
Bitwise XOR

Description

Compute the bitwise exclusive OR of the contents of data register D[a] and the contents of either data register D[b] (instruction format RR) or const9 (instruction format RC). Put the result in data register D[c]. The value const9 is zero-extended to 32-bits.

Compute the bitwise exclusive OR of the contents of data register D[a] and the contents of data register D[b]. Put the result in data register D[a].

XORD[c], D[a], const9 (RC)

31 28	27 21	20 12	11 8	7 0
c	$0C_H$	const9	a	$8F_H$

D[c] = D[a] ^ zero_ext(const9);

XORD[c], D[a], D[b] (RR)

31 28	27 20	19 18	17 16	15 12	11 8	7 0
c	$0C_H$	-	-	b	a	$0F_H$

D[c] = D[a] ^ D[b];

XORD[a], D[b] (SRR)

15 12	11 8	7 0
b	a	$C6_H$

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, d2
xor   d3, d1, #126
```

```
xor   d3, d2
```

See Also

AND, **ANDN**, **NAND**, **NOR**, **NOT (16-bit)**, **OR**, **ORN**, **XNOR**

XOR.EQ
Equal Accumulating

Description

Compute the logical XOR of D[c][0] and the Boolean result of the EQ operation on the contents of data register D[a] and either data register D[b] (instruction format RR) or const9 (instruction format RC). Put the result in D[c][0]. All other bits in D[c] are unchanged. The value const9 is sign-extended.

XOR.EQD[c], D[a], const9 (RC)

31 28	27 21	20 12	11 8	7 0
c	$2F_H$	const9	a	$8B_H$

D[c] = {D[c][31:1], D[c][0] XOR (D[a] == sign_ext(const9))};

XOR.EQD[c], D[a], D[b] (RR)

31 28	27 20	19 18	17 16	15 12	11 8	7 0
c	$2F_H$	-	-	b	a	$0B_H$

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, d2
xor.eq   d3, d1, #126
```

See Also

AND.EQ, **OR.EQ**

XOR.GE
Greater Than or Equal Accumulating
XOR.GE.U
Greater Than or Equal Accumulating Unsigned

Description

Calculate the logical XOR of D[c][0] and the Boolean result of the GE or GE.U operation on the contents of data register D[a] and either data register D[b] (instruction format RR) or const9 (instruction format RC). Put the result in D[c][0]. All other bits in D[c] are unchanged. D[a] and D[b] are treated as 32-bit signed (XOR.GE) or unsigned (XOR.GE.U) integers. The value const9 is sign-extended (XOR.GE) or zero-extended (XOR.GE.U).

XOR.GED[c], D[a], const9 (RC)

31 28	27 21	20 12	11 8	7 0
c	33_H	const9	a	$8B_H$

D[c] = {D[c][31:1], D[c][0] XOR (D[a] >= sign_ext(const9))};

XOR.GED[c], D[a], D[b] (RR)

31 28	27 20	19 18	17 16	15 12	11 8	7 0
c	33_H	-	-	b	a	$0B_H$

D[c] = {D[c][31:1], D[c][0] XOR (D[a] >= D[b])};

XOR.GE.UD[c], D[a], const9 (RC)

31 28	27 21	20 12	11 8	7 0
c	34_H	const9	a	$8B_H$

D[c] = {D[c][31:1], D[c][0] XOR (D[a] >= zero_ext(const9))}; // unsigned

XOR.GE.UD[c], D[a], D[b] (RR)

31 28	27 20	19 18	17 16	15 12	11 8	7 0
c	34_H	-	-	b	a	$0B_H$

D[c] = {D[c][31:1], D[c][0] XOR (D[a] >= D[b])}; // unsigned

Status Flags

C	Not set by these instructions.
V	Not set by these instructions.
SV	Not set by these instructions.
AV	Not set by these instructions.
SAV	Not set by these instructions.

Examples

```
xor.ge    d3, d1, d2
xor.ge    d3, d1, #126
xor.ge.u    d3, d1, d2
xor.ge.u    d3, d1, #126
```

See Also

AND.GE, **AND.GE.U**, **OR.GE**, **OR.GE.U**

XOR.LT
Less Than Accumulating
XOR.LT.U
Less Than Accumulating Unsigned

Description

Calculate the logical XOR of D[c][0] and the Boolean result of the LT or LT.U operation on the contents of data register D[a] and either data register D[b] (instruction format RR) or const9 (instruction format RC). Put the result in D[c][0]. All other bits in D[c] are unchanged. D[a] and D[b] are treated as 32-bit signed (XOR.LT) or unsigned (XOR.LT.U) integers. The value const9 is sign-extended (XOR.LT) or zero-extended (XOR.LT.U).

XOR.LTD[c], D[a], const9 (RC)

31 28	27 21	20 12	11 8	7 0
c	31_H	const9	a	$8B_H$

D[c] = {D[c][31:1], D[c][0] XOR (D[a] < sign_ext(const9))};

XOR.LTD[c], D[a], D[b] (RR)

31 28	27 20	19 18	17 16	15 12	11 8	7 0
c	31_H	-	-	b	a	$0B_H$

D[c] = {D[c][31:1], D[c][0] XOR (D[a] < D[b])};

XOR.LT.UD[c], D[a], const9 (RC)

31 28	27 21	20 12	11 8	7 0
c	32_H	const9	a	$8B_H$

D[c] = {D[c][31:1], D[c][0] XOR (D[a] < zero_ext(const9))}; // unsigned

XOR.LT.UD[c], D[a], D[b] (RR)

31 28	27 20	19 18	17 16	15 12	11 8	7 0
c	32_H	-	-	b	a	$0B_H$

D[c] = {D[c][31:1], D[c][0] XOR (D[a] < D[b])}; // unsigned

Status Flags

C	Not set by these instructions.
V	Not set by these instructions.
SV	Not set by these instructions.
AV	Not set by these instructions.
SAV	Not set by these instructions.

Examples

```
xor.lt   d3, d1, d2
xor.lt   d3, d1, #126
xor.lt.u   d3, d1, d2
xor.lt.u   d3, d1, #126
```

See Also

AND.LT, **AND.LT.U**, **OR.LT**, **OR.LT.U**

XOR.NE
Not Equal Accumulating

Description

Calculate the logical XOR of D[c][0] and the Boolean result of the NE operation on the contents of data register D[a] and either data register D[b] (instruction format RR) or const9. (instruction format RC). Put the result in D[c][0]. All other bits in D[c] are unchanged. The value const9 is sign-extended.

XOR.NED[c], D[a], const9 (RC)

31 28	27 21	20 12	11 8	7 0
c	30_H	const9	a	$8B_H$

D[c] = {D[c][31:1], D[c][0] XOR (D[a] != sign_ext(const9))};

XOR.NED[c], D[a], D[b] (RR)

31 28	27 20	19 18	17 16	15 12	11 8	7 0
c	30_H	-	-	b	a	$0B_H$

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, d2
xor.ne    d3, d1, #126
```

See Also

AND.NE, **OR.NE**

XOR.T
Bit Logical XOR

Description

Compute the logical XOR of bit pos1 of data register D[a] and bit pos2 of data register D[b]. Put the result in the least-significant bit of data register D[c] and clear the remaining bits of D[c] to zero.

XOR.TD[c], D[a], pos1, D[b], pos2 (BIT)

31 28	27 23	22 21	20 16	15 12	11 8	7 0
c	pos2	03_H	pos1	b	a	07_H

result = D[a][pos1] XOR D[b][pos2];

D[c] = zero_ext(result);

Status Flags

C	Not set by this instruction.
V	Not set by this instruction.
SV	Not set by this instruction.
AV	Not set by this instruction.
SAV	Not set by this instruction.

Examples

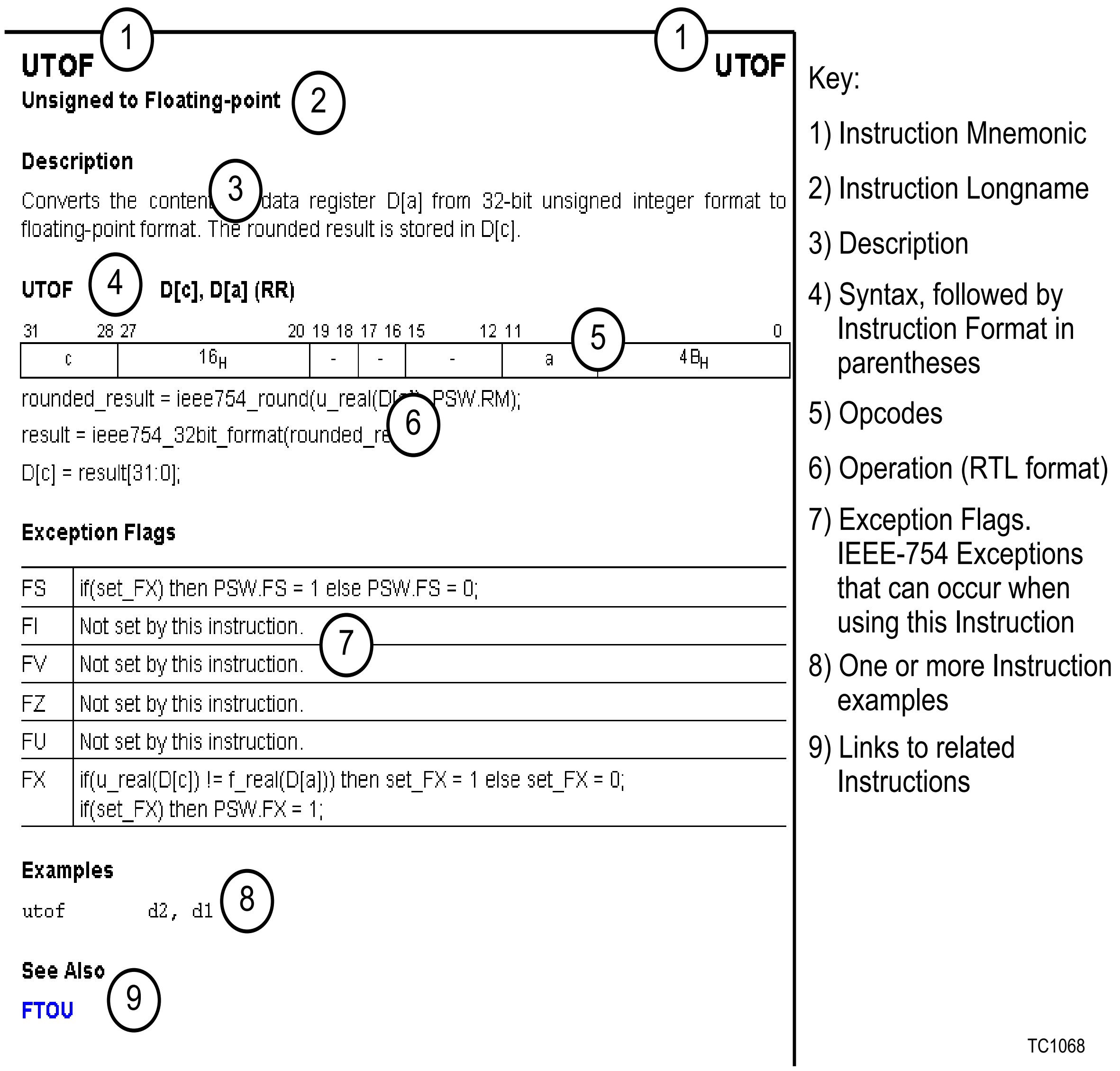
```
xor.t   d3, d1, 3, d2, #7
```

See Also

AND.T, **ANDN.T**, **NAND.T**, **NOR.T**, **OR.T**, **ORN.T**, **XNOR.T**

3.2 FPU Instructions

Each page for this group of instructions is laid out as follows:


1
1
2
3
4
5
6
7
8
9
TC1068

UTOF **UTOF**

Unsigned to Floating-point

Description

Converts the conten data register D[a] from 32-bit unsigned integer format to floating-point format. The rounded result is stored in D[c].

UTOF **D[c], D[a] (RR)**

31 28	27 20	19 18	17 16	15 12	11	0
c	16_H	-	-	-	a	$4B_H$

```
rounded_result = ieee754_round(u_real(D[ PSW.RM);
result = ieee754_32bit_format(rounded_re
D[c] = result[31:0];
```

Exception Flags

FS	if(set_FX) then PSW.FS = 1 else PSW.FS = 0;
FI	Not set by this instruction.
FV	Not set by this instruction.
FZ	Not set by this instruction.
FU	Not set by this instruction.

FX	if(u_real(D[c]) != f_real(D[a])) then set_FX = 1 else set_FX = 0;
	if(set_FX) then PSW.FX = 1;

Examples

```
utof        d2, d1
```

See Also

FTOU

Key:

1) Instruction Mnemonic
2) Instruction Longname
3) Description
4) Syntax, followed by Instruction Format in parentheses
5) Opcodes
6) Operation (RTL format)
7) Exception Flags. IEEE-754 Exceptions that can occur when using this Instruction
8) One or more Instruction examples
9) Links to related Instructions

ADD.F
Add Float

Description

Add the contents of data register D[a] to the contents of data register D[d]. Put the result in data register D[c]. The operands and result are single precision IEEE-754 floating-point numbers. If either operand is a NaN (quiet or signalling), then the return result will be the quiet NaN 7FC00000$_H$.

ADD.FD[c], D[d], D[a] (RRR)

31 28	27 24	23 20	19 18	17 16	15 12	11 8	7 0
c	d	02$_H$	-	1$_H$	-	a	6B$_H$

```
arg_a = denorm_to_zero(f_real(D[a]);
arg_b = denorm_to_zero(f_real(D[d]);
if(is_nan(D[a]) OR is_nan(D[d])) then result = QUIET_NAN;
else if(is_pos_inf(D[a]) AND is_neg_inf(D[d])) then result = ADD_NAN;
else if(is_neg_inf(D[a]) AND is_pos_inf(D[d])) then result = ADD_NAN;
else {
precise_result = add(arg_a,arg_b);
normal_result = denorm_to_zero(precise_result);
rounded_result = ieee754_round(normal_result, PSW.RM);
result = ieee754_32bit_format(rounded_result);
}
D[c] = result[31:0];
```

Exception Flags

FS	if(set_FI OR set_FV OR set_FU OR set_FX) then PSW.FS = 1 else PSW.FS = 0;

FI	if(is_s_nan(D[a]) OR is_s_nan(D[d])) then set_FI = 1 else set_FI = 0;
	if(set_FI) then PSW.FI = 1;

FV	if(rounded_result >= 2^{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 != f_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

CMP.F
Compare Float

Description

This instruction compares the IEEE-754 single-precision floating-point operands and asserts bits in the result if their associated condition is true:

bit [0] D[a] < D[b]

bit [1] D[a] == D[b]

bit [2] D[a] > D[b]

bit [3] Unordered

bit [4] D[a] is denormal

bit [5] D[b] is denormal

bits[31:06] are cleared.

The ‘unordered’ bit is asserted if either operand is a NaN.

Note: CMP.F is the only FPU instruction that does not substitute denormal operands for zero before computation.

CMP.FD[c], D[a], D[b] (RR)

31 28	27 20	19 18	17 16	15 12	11 8	7 0
c	00_H	-	1_H	b	a	$4B_H$

D[c][0] = ieee754_lt(D[a], D[b]);

D[c][1] = ieee754_eq(D[a], D[b]);

D[c][2] = ieee754_gt(D[a], D[b]);

D[c][3] = (is_nan(D[a]) OR is_nan(D[b]));

D[c][4] = is_denorm(D[a]);

D[c][5] = is_denorm(D[b]);

Exception Flags

FS	if(set_FI) then PSW.FS = 1 else PSW.FS = 0;

FI	if(is_s_nan(D[a]) OR is_s_nan(D[b])) then set_FI = 1;
	if(set_FI) then PSW.FI = 1;

FV	Not set by this instruction.
FZ	Not set by this instruction.
FU	Not set by this instruction.
FX	Not set by this instruction.

Examples

```
cmp.f     d3, d1, d2
```

See Also

-

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 floating-point numbers. If either operand is a NaN (quiet or signalling), then the return result will be the quiet NaN 7FC00000$_H$.

DIV.FD[c], D[a], D[b] (RR)

31 28	27 20	19 18	17 16	15 12	11 8	7 0
c	05$_H$	-	1$_H$	b	a	4B$_H$

```
arg_a = denorm_to_zero(f_real(D[a]);
arg_b = denorm_to_zero(f_real(D[b]);
if(is_nan(D[a]) OR is_nan(D[b])) then result = QUIET_NAN;
else if(is_inf(D[a]) AND is_inf(D[b])) then result = DIV_NAN;
else if(is_zero(D[a]) AND is_zero(D[b])) then result = DIV_NAN;
else {
precise_result = divide(arg_a,arg_b);
normal_result = denorm_to_zero(precise_result);
rounded_result = ieee754_round(normal_result, PSW.RM);
result = ieee754_32bit_format(rounded_result);
}
D[c] = result[31:0];
```

Exception Flags

FS	if(set_FI OR set_FV OR set_FZ OR set_FU OR set_FX) then PSW.FS = 1 else PSW.FS = 0;

FI	if(is_s_nan(D[a]) OR is_s_nan(D[b]) OR (D[c] == DIV_NAN)) then set_FI = 1 else set_FI = 0;
	if(set_FI) then PSW.FI = 1;

FV	if(rounded_result >= 2^{128}) then set_FV = 1 else set_FV = 0;
	if(set_FV) then PSW.FV = 1;

FZ	if(is_zero(D[b]) AND !(is_inf(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 != f_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

-

FTOI
Float to Integer

Description

Converts the contents of data register D[a] from floating-point format to a 32-bit two’s complement signed integer format. The rounded result is put in data register D[c]. The rounding mode used for the conversion is defined by the PSW.RM field.

FTOID[c], D[a] (RR)

31 28	27 20	19 18	17 16	15 12	11 8	7 0
c	10_H	-	1_H	-	a	$4B_H$

```
if(is_nan(D[a])) then result = 0;
else if(f_real(D[a]) > $2^{31}$-1) then result = $7FFFFFFF_H$;
else if(f_real(D[a]) < -$2^{31}$) then result = $80000000_H$;
else result = round_to_integer(D[a], PSW.RM);
D[c] = result[31:0];
```

Exception Flags

FS	if(set_FI OR set_FX) then PSW.FS = 1 else PSW.FS = 0;

FI	if((f_real(D[a]) > 2^{31}-1) OR (f_real(D[a]) < -2^{31}) OR is_nan(D[a])) then set_FI = 1 else set_FI = 0;
	if(set_FI) then PSW.FI = 1;

FV	Not set by this instruction.
FZ	Not set by this instruction.
FU	Not set by this instruction.

FX	if(f_real(D[a]) != i_real(result)) then set_FX = 1 else set_FX = 0;
	if(set_FX AND !set_FI) then PSW.FX = 1;

Examples

```
ftoi    d2, d1
```

See Also

ITOF, FTOIZ

FTOIZ

Float to Integer, Round towards Zero

Description

Converts the contents of data register D[a] from floating-point format to a 32-bit two’s complement signed integer format. The result is rounded towards zero and put in data register D[c].

FTOIZD[c], D[a] (RR)

31 – 28	27 – 20	19 – 18	17 – 16	15 – 12	11 – 8	7 – 0
c	13_H	-	1_H	-	a	$4B_H$

```
if(is_nan(D[a])) then result = 0;
else if(f_real(D[a]) > $2^{31}$-1) then result = $7FFFFFFF_H$;
else if(f_real(D[a]) < $-2^{31}$) then result = $80000000_H$;
else result = 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((f_real(D[a]) > 2^{31}-1) OR (f_real(D[a]) < -2^{31}) OR is_nan(D[a])) then set_FI = 1 else set_FI = 0;
	if(set_FI) then PSW.FI = 1;

FV	Not set by this instruction.
FZ	Not set by this instruction.
FU	Not set by this instruction.

FX	if(f_real(D[a]) != i_real(result)) then set_FX = 1 else set_FX = 0;
	if(set_FX) then PSW.FX = 1;

Examples

```
ftoiz   d2, d1
```

See Also

ITOF, FTOI

FTOQ31
Float to Fraction

Description

Subtracts D[b] from the exponent of the floating-point input value D[a] and converts the result to the Q31fraction format. The result is stored in D[c]. The rounding mode used for the conversion is defined by the PSW.RM field.

The exponent adjustment is a 9-bit two's complement number taken from D[b][8:0], with a value of [-256, 255]. D[b][31:9] is ignored.

Q31 fraction format is a 32-bit two's complement format which represents a value in the range [-1,1).

- Bit 31 represents -1
- Bit 30 represents +1/2
- Bit 29 represents +1/4
- Bit 28 represents +1/8
- etc.

FTOQ31D[c], D[a], D[b] (RR)

31 28	27 20	19 18	17 16	15 12	11 8	7 0
c	11_H	-	1_H	b	a	$4B_H$

```
arg_a = denorm_to_zero(f_real(D[a]);
if(is_nan(D[a])) then result = 0;
else precise_result = mul(arg_a, $2^{-D[b][8:0]}$);
if(precise_result > q_real(7FFFFFFF$_H$)) then result = 7FFFFFFF$_H$;
else if(precise_result < -1.0) then result = 80000000$_H$;
else result = round_to_q31(precise_result);
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_nan(D[a])) then set_FI = 1 else set_FI = 0;
	if(set_FI) then PSW.FI = 1;

FV	Not set by this instruction.
FZ	Not set by this instruction.
FU	Not set by this instruction.

FX	if(f_real(D[a]) != q_real(result)) then set_FX = 1 else set_FX = 0;
	if(set_FX AND !set_FI) then PSW.FX = 1;

Examples

```
ftoq31          d3, d1, d2
```

See Also

Q31TOF, **FTOQ31Z**

FTOQ31Z
Float to Fraction, Round towards Zero

Description

Subtracts D[b] from the exponent of the floating-point input value D[a] and converts the result to the Q31fraction format. The result is rounded towards zero and stored in D[c].

The exponent adjustment is a 9-bit two's complement number taken from D[b][8:0], with a value of [-256, 255]. D[b][31:9] is ignored.

Q31 fraction format is a 32-bit two's complement format which represents a value in the range [-1,1).

Bit 31 represents -1

Bit 30 represents +1/2

Bit 29 represents +1/4

Bit 28 represents +1/8

etc.

FTOQ31ZD[c], D[a], D[b] (RR)

31 28	27 20	19 18	17 16	15 12	11 8	7 0
c	18_H	-	1_H	b	a	$4B_H$

```
arg_a = denorm_to_zero(f_real(D[a]);
if(is_nan(D[a])) then result = 0;
else precise_result = mul(arg_a, $2^{-D[b][8:0]}$);
if(precise_result > q_real($7FFFFFFF_H$)) then result = $7FFFFFFF_H$;
else if(precise_result < -1.0) then result = $80000000_H$;
else result = round_to_q31(precise_result, $11_B$);
D[c] = result[31:0];
```

Exception Flags

FS	if(set_FI 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_nan(D[a])) then set_FI = 1 else set_FI = 0;
	if(set_FI) then PSW.FI = 1;

FV	Not set by this instruction.
FZ	Not set by this instruction.
FU	Not set by this instruction.

FX	if(f_real(D[a]) != q_real(result)) then set_FX = 1 else set_FX = 0;
	if(set_FX) then PSW.FX = 1;

Examples

```
ftoq31z         d3, d1, d2
```

See Also

Q31TOF, **FTOQ31**

FTOU
Float to Unsigned

Description

Converts the contents of data register D[a] from floating-point format to a 32-bit unsigned integer format. The rounded result is put in data register D[c].

FTOUD[c], D[a] (RR)

31 28	27 20	19 18	17 16	15 12	11 8	7 0
c	12_H	-	1_H	-	a	$4B_H$

if(is_nan(D[a])) then result = 0;

else if(f_real(D[a]) > 2^{32}-1) then result = $FFFFFFFF_H$;

else if(f_real(D[a]) < 0.0) then result = 0;

else result = round_to_unsigned(D[a], PSW.RM);

D[c] = result[31:0];

Exception Flags

FS	if(set_FI OR set_FX) then PSW.FS = 1 else PSW.FS = 0;

FI	if((f_real(D[a]) > 2^{32}-1) OR (f_real(D[a]) < 0.0) OR is_nan(D[a])) then set_FI = 1 else set_FI = 0;
	if(set_FI) then PSW.FI = 1;

FV	Not set by this instruction.
FZ	Not set by this instruction.
FU	Not set by this instruction.

FX	if(f_real(D[a]) != u_real(result)) then set_FX = 1 else set_FX = 0;
	if(set_FX AND !set_FI) then PSW.FX = 1;

Examples

```
ftou          d2, d1
```

See Also

UTOF, FTOUZ

FTOUZ
Float to Unsigned, Round towards Zero

Description

Converts the contents of data register D[a] from floating-point format to a 32-bit unsigned integer format. The result is rounded towards zero and put in data register D[c].

FTOUZD[c], D[a] (RR)

31 28	27 20	19 18	17 16	15 12	11 8	7 0
c	17_H	-	1_H	-	a	$4B_H$

```
if(is_nan(D[a])) then result = 0;
else if(f_real(D[a]) > $2^{32}$-1) then result = $FFFFFFFF_H$;
else if(f_real(D[a]) < 0.0) then result = 0;
else result = 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((f_real(D[a]) > 2^{32}-1) OR (f_real(D[a]) < 0.0) OR is_nan(D[a])) then set_FI = 1 else set_FI = 0;
	if(set_FI) then PSW.FI = 1;

FV	Not set by this instruction.
FZ	Not set by this instruction.
FU	Not set by this instruction.

FX	if(f_real(D[a]) != u_real(result)) then set_FX = 1 else set_FX = 0;
	if(set_FX) then PSW.FX = 1;

Examples

```
ftouz          d2, d1
```

See Also

UTOF, **FTOU**

ITOF
Integer to Float

Description

Converts the contents of data register D[a] from 32-bit two's complement signed integer format to floating-point format. The rounded result is put in data register D[c].

ITOFD[c], D[a] (RR)

31 28	27 20	19 18	17 16	15 12	11 8	7 0
c	14_H	-	1_H	-	a	$4B_H$

rounded_result = ieee754_round(i_real(D[a]), PSW.RM);

result = ieee754_32bit_format(rounded_result);

D[c] = result[31:0];

Exception Flags

FS	if(set_FX) then PSW.FS = 1 else PSW.FS = 0;
FI	Not set by this instruction.
FV	Not set by this instruction.
FZ	Not set by this instruction.
FU	Not set by this instruction.

FX	if(f_real(result) != i_real(D[a])) then set_FX = 1 else set_FX = 0;
	if(set_FX) then PSW.FX = 1;

Examples

```
itof       d2, d1
```

See Also

FTOI, **FTOIZ**

MADD.F
Multiply Add Float

Description

Multiplies D[a] and D[b] and adds the product to D[d]. The result is put in D[c]. The operands and result are floating-point numbers. If an operand is a NaN (quiet or signalling), then the return result will be the quiet NaN $7FC00000_H$.

MADD.FD[c], D[d], D[a], D[b] (RRR)

31 28	27 24	23 20	19 18	17 16	15 12	11 8	7 0
c	d	06_H	-	1_H	b	a	$6B_H$

```
arg_a = denorm_to_zero(f_real(D[a]);
arg_b = denorm_to_zero(f_real(D[b]);
arg_c = denorm_to_zero(f_real(D[d]);
if(is_nan(D[a]) OR is_nan(D[b]) OR is_nan(D[d])) then result = QUIET_NAN;
else if(is_inf(D[a]) AND is_zero(D[b])) then result = MUL_NAN;
else if(is_zero(D[a]) AND is_inf(D[b])) then result = MUL_NAN;
else if(((is_neg_inf(D[a]) AND is_neg_inf(D[b])) OR
((is_pos_inf(D[a]) AND is_pos_inf(D[b]))) AND
is_neg_inf(D[d])) then result = ADD_NAN;
    else if(((is_neg_inf(D[a]) AND is_pos_inf(D[b])) OR
((is_pos_inf(D[a]) AND is_neg_inf(D[b]))) AND
is_pos_inf(D[b])) then result = ADD_NAN;
else {
precise_mul_result = mul(arg_a, arg_b);
precise_result = add(precise_mul_result, arg_c);
normal_result = denorm_to_zero(precise_result);
rounded_result = ieee754_round(normal_result, PSW.RM);
result = ieee754_32bit_format(rounded_result);
}
D[c] = result[31:0];
```

Exception Flags

FS	if(set_FI OR set_FV OR set_FU or set_FX) then PSW.FS = 1 else PSW.FS = 0;

FI	if(is_s_nan(D[a]) OR is_s_nan(D[b]) OR is_s_nan(D[d]) OR (result == ADD_NAN) OR (result == MUL_NAN)) then set_FI = 1 else set_FI = 0;
	if(set_FI) then PSW.FI = 1;

FV	if(rounded_result >= 2^{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 != f_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**

MSUB.F
Multiply Subtract Float

Description

Multiplies D[a] and D[b] and subtracts the product from D[d], putting the result in D[c]. The operands and result are floating-point numbers. If any operand is a NaN (quiet or signalling), then the return result will be the quiet NaN 7FC0 0000_H.

MSUB.FD[c], D[d], D[a], D[b] (RRR)

31 … 28	27 … 24	23 … 20	19 … 18	17 … 16	15 … 12	11 … 8	7 … 0
c	d	07_H	-	1_H	b	a	$6B_H$

```
arg_a = denorm_to_zero(f_real(D[a]);
arg_b = denorm_to_zero(f_real(D[b]);
arg_c = denorm_to_zero(f_real(D[d]);
if(is_nan(D[a]) OR is_nan(D[b]) OR is_nan(D[d])) then result = QUIET_NAN;
else if(is_inf(D[a]) AND is_zero(D[b])) then result = MUL_NAN;
  else if(is_zero(D[a]) AND is_inf(D[b])) then result = MUL_NAN;
        else if(((is_neg_inf(D[a]) AND is_neg_inf(D[b])) OR
((is_pos_inf(D[a]) AND is_pos_inf(D[b]))) AND
is_pos_inf(D[d])) then result = ADD_NAN;
else if(((is_neg_inf(D[a]) AND is_pos_inf(D[b])) OR
((is_pos_inf(D[a]) AND is_neg_inf(D[b]))) AND
is_neg_inf(D[b])) then result = ADD_NAN;
                            else {
precise_mul_result = ieee754_mul(arg_a, arg_b);
           precise_result = ieee754_add(-precise_mul_result, arg_c);
           normal_result = denorm_to_zero(precise_result);
           rounded_result = ieee754_round(normal_result, PSW.RM);
           result = ieee754_32bit_format(rounded_result);
}
D[c] = result[31:0];
```

Exception Flags

FS	if(set_FI OR set_FV OR set_FU OR set_FX) then PSW.FS = 1 else PSW.FS = 0;

FI	if(is_s_nan(D[a]) OR is_s_nan(D[b]) OR is_s_nan(D[d]) OR (result == ADD_NAN) OR (result == MUL_NAN)) then set_FI = 1 else set_FI = 0;
	if(set_FI) then PSW.FI = 1;

FV	if(rounded_result >= 2^{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 != f_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

MUL.F
Multiply Float

Description

Multiplies D[a] and D[b] and stores the result in D[c]. The operands and result are floating-point numbers. If an operand is a NaN (quiet or signalling), then the return result will be the quiet NaN 7FC00000$_H$.

MUL.FD[c], D[a], D[b] (RR)

31 28	27 20	19 18	17 16	15 12	11 8	7 0
c	04$_H$	-	1$_H$	b	a	4B$_H$

```
arg_a = denorm_to_zero(f_real(D[a]);
arg_b = denorm_to_zero(f_real(D[b]);
if(is_nan(D[a]) OR is_nan(D[b])) then result = QUIET_NAN;
else if(is_inf(D[a]) AND is_zero(D[b])) then result = MUL_NAN;
else if(is_inf(D[b]) AND is_zero(D[a])) then result = MUL_NAN;
else {
precise_result = mul(arg_a, arg_b);
normal_result = denorm_to_zero(precise_result);
rounded_result = ieee754_round(normal_result, PSW.RM);
result = ieee754_32bit_format(rounded_result);
}
D[c] = result[31:0];
```

Exception Flags

FS	if(set_FI OR set_FV OR set_FU OR set_FX) then PSW.FS = 1 else PSW.FS = 0;

FI	if(is_s_nan(D[a]) OR is_s_nan(D[b]) OR (result == MUL_NAN)) then set_FI = 1 else set_FI = 0;
	if(set_FI) then PSW.FI = 1;

FV	if(rounded_result >= 2^{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 != f_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

-

Q31TOF
Fraction to Floating-point

Description

Converts the D[a] from Q31 fraction format to floating-point format, then adds D[b] to the exponent and stores the resulting value in D[c]. The exponent adjustment is a 9-bit two's complement number taken from D[b][8:0], with a value in the range [-256, 255]. D[b][31:9] is ignored. Q31 fraction format is a 32-bit two's complement format which represents a value in the range [-1,1).

- Bit 31 represents -1
- Bit 30 represents +1/2
- Bit 29 represents +1/4
- etc.

Q31TOFD[c], D[a], D[b] (RR)

31 28	27 20	19 18	17 16	15 12	11 8	7 0
c	15_H	-	1_H	b	a	$4B_H$

precise_result = mul(q_real(D[a]),$2^{D[b][8:0]}$);

rounded_result = ieee754_round(precise_result, PSW.RM);

result = ieee754_32bit_format(rounded_result);

D[c] = result[31:0];

Exception Flags

FS	if(set_FX) then PSW.FS = 1 else PSW.FS = 0;
FI	Not set by this instruction.
FV	Not set by this instruction.
FZ	Not set by this instruction.

FU	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 != f_real(result)) then set_FX = 1 else set_FX = 0;
	if(set_FX) then PSW.FX = 1;

Examples

```
q31tof          d3, d1, d2
```

See Also

FTOQ31, FTOQ31Z

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 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.FD[c], D[a] (RR)

31 28	27 20	19 18	17 16	15 12	11 8	7 0
c	19$_H$	-	1$_H$	-	a	4B$_H$

```
arg_a = denorm_to_zero(f_real(D[a]);
if(is_nan(D[a])) then result = QUIET_NAN;
else if(arg_a == +0.0) then result = POS_INFINITY;
else if(arg_a == -0.0) then result = NEG_INFINITY;
else if(arg_a < 0.0) then result = SQRT_NAN;
else {
normal_result = approx_inv_sqrt(arg_a);
result = ieee754_32bit_format(nomral_result);
}
D[c] = result[31:0];
```

Exception Flags

FS	if(set_FI) then PSW.FS = 1 else PSW.FS = 0;

FI	if(is_s_nan(D[a]) OR (D[c] == SQRT_NAN)) then set_FI = 1 else set_FI = 0;
	if(set_FI) then PSW.FI = 1;

FV	Not set by this instruction.
FZ	Not set by this instruction.
FU	Not set by this instruction.
FX	Not set by this instruction.

Examples

```
qseed.f        d2, d1
```

See Also

-

SUB.F
Subtract Float

Description

Subtracts D[a] from D[d] and stores the result in D[c]. The operands and result are floating-point numbers. If any operand is a NaN (quiet or signalling), then the return result will be the quiet NaN 7FC00000$_H$.

SUB.FD[c], D[d], D[a] (RRR)

31 28	27 24	23 20	19 18	17 16	15 12	11 8	7 0
c	d	03_H	-	1_H	-	a	$6B_H$

```
arg_a = denorm_to_zero(f_real(D[a]);
arg_b = denorm_to_zero(f_real(D[d]);
if(is_nan(D[a]) OR is_nan(D[b])) then result = QUIET_NAN;
else if(is_pos_inf(D[a]) AND is_pos_inf(D[b])) then result = ADD_NAN;
else if(is_neg_inf(D[a]) AND is_nef_inf(D[b])) then result = ADD_NAN;
else {
precise_result = add(-arg_a, arg_b);
normal_result = denorm_to_zero(precise_result);
rounded_result = ieee754_round(normal_result, PSW.RM);
result = ieee754_32bit_format(rounded_result);
}
D[c] = result[31:0];
```

Exception Flags

FS	if(set_FI OR set_FV OR set_FU OR set_FX) then PSW.FS = 1 else PSW.FS = 0;

FI	if(is_s_nan(D[a]) OR is_s_nan(D[b])) then set_FI = 1 else set_FI = 0;
	if(set_FI) then PSW.FI = 1;

FV	if(rounded_result >= 2^{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 != f_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

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].

UPDFLD[a] (RR)

31 28	27 20	19 18	17 16	15 12	11 8	7 0
a	$0C_H$	-	1_H	-	a	$4B_H$

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

-

UTOF
Unsigned to Floating-point

Description

Converts the contents of data register D[a] from 32-bit unsigned integer format to floating-point format. The rounded result is stored in D[c].

UTOFD[c], D[a] (RR)

31 28	27 20	19 18	17 16	15 12	11 8	7 0
c	16_H	-	1_H	-	a	$4B_H$

rounded_result = ieee754_round(u_real(D[a]), PSW.RM);

result = ieee754_32bit_format(rounded_result);

D[c] = result[31:0];

Exception Flags

FS	if(set_FX) then PSW.FS = 1 else PSW.FS = 0;
FI	Not set by this instruction.
FV	Not set by this instruction.
FZ	Not set by this instruction.
FU	Not set by this instruction.

FX	if(u_real(D[c]) != f_real(D[a])) then set_FX = 1 else set_FX = 0;
	if(set_FX) then PSW.FX = 1;

Examples

```
utof          d2, d1
```

See Also

FTOU, FTOUZ

3.3 LS and IP Instruction Summary Lists

This section contains two lists; one of the LS instructions and one of the IP instructions.

3.3.1 List of 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.W** - Cache Index, Writeback
- **CACHEI.I** - Cache Index, Invalidate
- **CACHEI.WI** - Cache Index, Writeback, Invalidate
- **CALL** - Call
- **CALLA** - Call Absolute
- **CALLI** - Call Indirect
- **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
- **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.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
- **LOOP** - Loop
- **LOOPU** - Loop Unconditional
- **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.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
- **SYSCALL** - System Call
- **TRAPSV** - Trap on Sticky Overflow
- **TRAPV** - Trap on Overflow

3.3.2 List of 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 Signed with Saturation
- **ADDS.H** - Add Signed 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 Accumulating
- **AND.GE** - Greater Than or Equal Accumulating
- **AND.GE.U** - Greater Than or Equal Accumulating Unsigned
- **AND.LT** - Less Than Accumulating
- **AND.LT.U** - Less Than Accumulating Unsigned
- **AND.NE** - Not Equal 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 (16-bit)** - Conditional Move (16-bit)
- **CMOVN (16-bit)** - Conditional Move-Not (16-bit)
- **CSUB** - Conditional Subtract
- **CSUBN** - Conditional Subtract-Not
- **DEXTR** - Extract from Double Register
- **DVADJ** - Divide-Adjust
- **DIV** - Divide
- **DIV.U** - Divide Unsigned
- **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 (16-bit)** - Jump if Greater Than or Equal to Zero (16-bit)
- **JGTZ (16-bit)** - Jump if Greater Than Zero (16-bit)
- **JLEZ (16-bit)** - Jump if Less Than or Equal to Zero (16-bit)
- **JLT** - Jump if Less Than
- **JLT.U** - Jump if Less Than Unsigned
- **JLTZ (16-bit)** - 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 (16-bit)** - Jump if Not Equal to Zero (16-bit)
- **JNZ.T** - Jump if Not Equal to Zero Bit
- **JZ (16-bit)** - 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
- **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
- **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
- **MULMS.H** - MULMS.H (DEPRECATED)
- **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 (16-bit)** - 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 Accumulating
- **OR.GE** - Greater Than or Equal Accumulating
- **OR.GE.U** - Greater Than or Equal Accumulating Unsigned
- **OR.LT** - Less Than Accumulating
- **OR.LT.U** - Less Than Accumulating Unsigned
- **OR.NE** - Not Equal Accumulating
- **OR.T** - Bit Logical OR
- **ORN** - Bitwise OR-Not
- **ORN.T** - Bit Logical OR-Not
- **PACK** - Pack
- **PARITY** - Parity
- **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
- **SUB** - Subtract
- **SUB.B** - Subtract Packed Byte
- **SUB.H** - Subtract Packed Half-word
- **SUBC** - Subtract With Carry
- **SUBS** - Subtract Signed 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
- **UNPACK** - Unpack Floating Point
- **XNOR** - Bitwise XNOR
- **XNOR.T** - Bit Logical XNOR
- **XOR** - Bitwise XOR
- **XOR.EQ** - Equal Accumulating
- **XOR.GE** - Greater Than or Equal Accumulating
- **XOR.GE.U** - Greater Than or Equal Accumulating Unsigned
- **XOR.LT** - Less Than Accumulating
- **XOR.LT.U** - Less Than Accumulating Unsigned
- **XOR.NE** - Not Equal Accumulating
- **XOR.T** - Bit Logical XOR
- **ADD.F** - Add Float
- **CMP.F** - Compare Float
- **DIV.F** - Divide Float
- **FTOI** - Float to Integer
- **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
- **ITOF** - Integer to Float
- **MADD.F** - Multiply Add Float
- **MSUB.F** - Multiply Subtract Float
- **MUL.F** - Multiply Float
- **Q31TOF** - Fraction to Floating-point
- **QSEED.F** - Inverse Square Root Seed
- **SUB.F** - Subtract Float
- **UPDFL** - Update Flags
- **UTOF** - Unsigned to Floating-point

List of Instructions by Shortname

List of Instructions by Longname

Keyword Index