Signal Processing Engine Auxiliary Processing Unit Programming Interface Manual

SPEPIM
Rev. 0
07/2004

freescale semiconductor

How to Reach Us:

USA/Europe/Locations Not Listed:
Freescale Semiconductor
Literature Distribution Center
P.O. Box 5405,
Denver, Colorado 80217
1-480-768-2130
(800) 521-6274

Japan:
Freescale Semiconductor Japan Ltd.
Technical Information Center
3-20-1, Minami-Azabu, Minato-ku
Tokyo 106-8573, Japan
81-3-3440-3569

Asia/Pacific:
Freescale Semiconductor Hong Kong Ltd.
2 Dai King Street
Tai Po Industrial Estate
Tai Po, N.T. Hong Kong
852-26668334

Home Page:
www.freescale.com

Learn More: For more information about Freescale Semiconductor products, please visit **www.freescale.com**

SPEPIM
Rev. 0
07/2004

Contents

Contents

Contents

Contents

Figures

Figures

Figures

Figures

Figures

Figures

Figures

Figures

Tables

Tables

About This Book

The primary objective of this manual is to help programmers provide software that is compatible across the family of processors that use the signal processing engine (SPE) auxiliary processing unit (APU).

Errata and Updates

Freescale recommends using the most recent version of the documentation. The information in this book is subject to change without notice, as described in the disclaimers on the title page of this book. To locate any published errata or updates for this document, refer to the website at http://www.freescale.com.

To obtain more information, contact your sales representative or visit our website at http://www.freescale.com.

Scope

The scope of this manual does not include a description of individual SPE implementations. Each PowerPC™ processor is unique in its implementation of the SPE.

Audience

This manual supports system software and application programmers who want to use the SPE APU to develop products. Users should understand the following concepts:

- Operating systems
- Microprocessor system design
- Basic principles of RISC processing
- SPE instruction set

Organization

The following list summarizes and briefly describes the major sections of this manual:

- Chapter 1, “Overview,” provides a general understanding of what the programming model defines in the SPE APU.
- Chapter 2, “High-Level Language Interface,” is useful for software engineers who need to understand how to access SPE functionality from high level languages such as C and C++.
- Chapter 3, “SPE Operations,” describes all instructions in the e500 core complex as well as Book E instructions that are defined for 32-bit implementations.
- Chapter 4, “Additional Operations,” describes data manipulation, SPE floating-point status and control register (SPEFSCR) operations, ABI extensions (malloc(), realloc(), calloc(), and new), a printf example, and additional library routines.
- Chapter 5, “Programming Interface Examples,” gives examples of valid and invalid initializations of the SPE data types.
- Appendix A, “Revision History,” lists the major differences between revisions of the *Signal Processing Engine Auxiliary Processing Unit Programming Interface Manual*.
- This manual also includes a glossary and an index.

Suggested Reading

The following sections note additional reading that can provide background for the information in this manual as well as general information about the architecture.

General Information

The following documentation, which is published by Morgan-Kaufmann Publishers, 340 Pine Street, Sixth Floor, San Francisco, CA, provides useful information about the PowerPC architecture and general computer architecture:

- *The PowerPC Architecture: A Specification for a New Family of RISC Processors*, Second Edition, by International Business Machines, Inc.
- System V Application Binary Interface, Edition 4.1.

References

The following documents may interest readers of this manual:

- *ISO/IEC 9899:1999 Programming Languages - C (ANSI C99 Specification).*
- *DWARF Debugging Information Format*, Version 3 (Revision 2.1, Draft 7), Oct. 29, 2001, available from http://www.eagercon.com/dwarf/dwarf3std.htm.

- *The PowerPC Architecture: A Specification for A New Family of RISC Processors*. International Business Machines (IBM). San Francisco: Morgan Kaufmann, 1994. (IBM provides a website at *Programming Environments Manual for 32-Bit Implementations of the PowerPC Architecture* (MPCFPE32B).
- *System V Application Binary Interface*, Edition 4.1.
- *System V Application Binary Interface Draft*, 24 April 2001.
- *System V Application Binary Interface*, PowerPC Processor Supplement, Rev. A.
- *System V Interface Definition*, Fourth Edition.

Related Documentation

Freescale documentation is available from the sources listed on the back cover of most manuals. The following list includes documentation that is related to topics in this manual:

- *AltiVec™ Technology Programming Interface Manual*
- *e500 Application Binary Interface (ABI)*
- *Freescale's Enhanced PowerPC Architecture Implementation Standards*
- *Freescale Book E Implementation Standards: APU ID Reference*
- *e500 ABI Save/Restore Routines*
- *EREF: A Reference for Freescale Book E and the e500 Core*—This book provides a higher-level view of the programming model as it is defined by Book E, the Freescale Book E implementation standards, and the e500 microprocessor.
- Reference manuals (formerly called user's manuals)—These books provide details about individual implementations.
- Addenda/errata to reference or user's manuals—Because some processors have follow-on parts, an addendum is provided that describes the additional features and functionality changes. These addenda are intended for use with the corresponding reference or user's manual.
- Application notes—These short documents address specific design issues that are useful to programmers and engineers working with Freescale processors.

Additional literature is published as new processors become available. For a current list of documentation, refer to http://www.freescale.com.

Chapter 1
Overview

This document defines a programming model to use with the signal processing engine (SPE) auxiliary processing unit (APU). This document describes three types of programming interfaces:

- A high-level language interface that is intended for use within programming languages, such as C or C++
- An application binary interface (ABI) that defines low-level coding conventions
- An assembly language interface

1.1 High-Level Language Interface

The high-level language interface enables programmers to use the SPE APU from programming languages such as C and C++, and describes fundamental data types for the SPE programming model. See Chapter 2, “High-Level Language Interface,” for details about this interface.

1.2 Application Binary Interface (ABI)

The SPE programming model extends the existing PowerPC ABIs. The extension is independent of the endian mode. The ABI reviews the data types and register usage conventions for vector register files and describes setup of the stack frame. Save and Restore functions for the vector register are included in the ABI section to advocate uniformity of method among compilers for saving and restoring vector registers.

The *AltiVec™ Technology Programming Interface Manual*, provides the valid set of argument types for specific AltiVec operations and predicates as well as specific AltiVec instructions that are generated for that set of arguments. The AltiVec operations and predicates are organized alphabetically in Chapter 4, “AltiVec Operations and Predicates.”

Chapter 2
High-Level Language Interface

2.1 Introduction

This document defines a programming model to use with the signal processing engine (SPE) auxiliary processing unit (APU) instruction set. The purpose of the programming model is to give users the ability to write code that utilizes the APU in a high-level language, such as C or C++.

Users should not be concerned with issues such as register allocation, scheduling, and conformity to the underlying ABI, which are all associated with writing code at the assembly level.

2.2 High-Level Language Interface

The high-level language interface for SPE is intended to accomplish the following goals:

- Provide an efficient and expressive mechanism to access SPE functionality from programming languages such as C and C++
- Define a minimal set of language extensions that unambiguously describe the intent of the programmer while minimizing the impact on existing PowerPC compilers and development tools
- Define a minimal set of library extensions that are needed to support SPE

2.2.1 Data Types

Table 2-1 describes a set of fundamental data types that the SPE programming model introduces.

NOTE

The type _ev64_ stands for embedded vector of data width 64 bits.

Table 2-1. Data Types

New C/C++ Type	Interpretation of Contents	Values
__ev64_u16__	4 unsigned 16-bit integers	0...65535
__ev64_s16__	4 signed 16-bit integers	-32768...32767
__ev64_u32__	2 unsigned 32-bit integers	$0...2^{32} - 1$
__ev64_s32__	2 signed 32-bit integers	$-2^{31}...2^{31} - 1$
__ev64_u64__	1 unsigned 64-bit integer	$0...2^{64} - 1$
__ev64_s64__	1 signed 64-bit integer	$-2^{63}...2^{63} - 1$
__ev64_fs__	2 floats	IEEE-754 single-precision values
__ev64_opaque__	any of the above	—

The __ev64_opaque__ data type is an opaque data type that can represent any of the specified __ev64_*__ data types. All of the __ev64_*__ data types are available to programmers.

2.2.2 Alignment

Refer to the e500 ABI for full alignment details.

2.2.2.1 Alignment of __ev64_*__ Types

A defined data item of any __ev64_*__ data type in memory is always aligned on an 8-byte boundary. A pointer to any __ev64_*__ data type always points to an 8-byte boundary. The compiler is responsible for aligning any __ev64_*__ data types on an 8-byte boundary. When __ev64_*__ data is correctly aligned, a program is incorrect if it attempts to dereference a pointer to an __ev64_*__ type if the pointer does not contain an 8-byte aligned address.

In the SPE architecture, an unaligned load/store causes an alignment exception.

2.2.2.2 Alignment of Aggregates and Unions Containing __ev64_*__ Types

Aggregates (structures and arrays) and unions containing __ev64_*__ variables must be aligned on 8-byte boundaries and their internal organization must be padded, if necessary, so that each internal __ev64_*__ variable is aligned on an 8-byte boundary.

2.2.3 Extensions of C/C++ Operators for the New Types

Most C/C++ operators do not permit any of their arguments to be one of the __ev64_*__ types. Let 'a' and 'b' be variables of any __ev64_*__ type, and 'p' be a pointer to any __ev64_*__ type. The normal C/C++ operators are extended to include the operations in the following sections.

2.2.3.1 sizeof()

The functions sizeof(a) and sizeof(*p) return 8.

2.2.3.2 Assignment

Assignment is allowed only if both the left- and right-hand sides of an expression are the same __ev64_*__ type. For example, the expression a=b is valid and represents assignment of 'b' to 'a'. The one exception to the rule occurs when 'a' or 'b' is of type __ev64_opaque__. Let 'o' be of type __ev64_opaque__ and let 'a' be of any __ev64_*__ type.

The assignments a=o and o=a are allowed and have implicit casts. Otherwise, the expression is invalid, and the compiler must signal an error.

2.2.3.3 Address Operator

The operation &a is valid if 'a' is an __ev64_*__ type. The result of the operation is a pointer to 'a'.

2.2.3.4 Pointer Arithmetic

The usual pointer arithmetic can be performed on p. In particular, p+1 is a pointer to the next __ev64_*__ element after p.

2.2.3.5 Pointer Dereferencing

If 'p' is a pointer to an __ev64_*__ type, *p implies either a 64-bit SPE load from the address, equivalent to the intrinsic __ev_ldd(p,0), or a 64-bit SPE store to that address, equivalent to the intrinsic __ev_stdd(p,0). Dereferencing a pointer to a non-__ev64_*__ type produces the standard behavior of either a load or a copy of the corresponding type.

Section 2.2.2.1, “Alignment of __ev64_*__ Types,” describes unaligned accesses.

2.2.3.6 Type Casting

Pointers to __ev64_*__ and existing types may be cast back and forth to each other. Casting a pointer to an __ev64_*__ type represents an (unchecked) assertion that the address is 8-byte aligned.

Casting from a integral type to a pointer to an __ev64_*__ type is allowed.

For example:

```
__ev64_u16__ *a = (__ev64_u16__ *) 0x48;
```

Casting between __ev64_*__ types and existing types is not allowed.

Casting between __ev64_*__ types and pointers to existing types is not allowed.

The behaviors expected from such casting are provided instead of using intrinsics.

The intrinsics provide the ability to extract existing data types out of __ev64_*__ variables as well as the ability to insert into and/or create __ev64_*__ variables from existing data types. Normal C casts provide casts from one __ev64_*__ type to another.

An implicit cast is performed when going to __ev64_opaque__ from any other __ev64_*__ type. An implicit cast occurs when going from __ev64_opaque__ to any other __ev64_*__ type. The implicit casts that occur when going between __ev64_opaque__ and any other __ev64_*__ type also apply to pointers of type __ev64_opaque__. When casting between any two __ev64_*__ types not including __ev64_opaque__, an explicit cast is required. When casting between pointers to any two __ev64_*__ types not including __ev64_opaque__, an explicit cast is required. No cast or promotion performs a conversion; the bit pattern of the result is the same as the bit pattern of the argument that is cast.

2.2.4 New Operators

New operators are introduced to construct __ev64_*__ values and allow full access to the functionality that the SPE architecture provides.

2.2.4.1 __ev64_*__ Initialization and Literals

The __ev64_opaque__ type is the only __ev64_*__ type that cannot be initialized. The remaining __ev64_*__ types can be initialized using the C99 array initialization syntax. Each type is treated as an array of the specified data contents of the appropriate size. The following code exemplifies the initialization of these types:

```
__ev64_u16__ a = { 0, 1, 2, 3 };
__ev64_s16__ b = { -1, -2, -3, 4 };
__ev64_u32__ c = { 3, 4 };
__ev64_s32__ d = { -2, 4 };
__ev64_u64__ e = { 17  } ;
__ev64_s64__ f = { 23 } ;
__ev64_fs__ g = { 2.4, -3.2 };

c = __ev_addw(a, (__ev64_s16__){2,1,5,2});
```

2.2.4.2 New Operators Representing SPE Operations

New operators are introduced to allow full access to the functionality that the SPE architecture provides. Language structures that parse like function calls represent these operators in the programming language.

The names associated with these operations are all prefixed with "__ev_". The appearance of one of these forms can indicate one of the following:

- A specific SPE operation, like __ev_addw(__ev64_opaque__ a, __ev64_opaque__ b)
- A predicate computed from a SPE operation, like __ev_all_eq(__ev64_opaque__ a, __ev64_opaque__ b)
- Creation, insertion, extraction of __ev64_opaque__ values

Each operator representing an SPE operation takes a list of arguments representing the input operands (in the order in which they are shown in the architecture specification) and returns a result that could be void. The programming model restricts the operand types that are permitted for each SPE operation. Predicate intrinsics handle comparison operations in the SPE programming model.

Each compare operation has the following predicate intrinsics associated with it:

- _any_
- _all_
- _upper_
- _lower_
- _select_

Each predicate returns an integer (0/1) with the result of the compare. The compiler allocates a CR field for use in the comparison and to optimize conditional statements.

2.2.5 Programming Interface

This document does not prohibit or require an implementation to provide any set of include files to provide access to the intrinsics. If an implementation requires that an include file be used before the use of the intrinsics described in this document, that file should be <spe.h>.

This document does require that prototypes for the additional library routines described are accessible by means of the include file <spe.h>. If an implementation should provide a __SPE__, define it with a nonzero value. That definition should not occur in the <spe.h> header file.

Chapter 3
SPE Operations

This chapter describes the following instructions:

- All instructions in the e500 core complex, including numerous instructions that Book E does not define.
- Book E instructions that are defined for 32-bit implementations, including many instructions that are not implemented on the e500 core complex.

3.1 Signal Processing Engine (SPE) APU Registers

The SPE includes the following two registers:

- The signal processing and embedded floating-point status and control register (SPEFSCR), which is described in Section 3.1.1, “Signal Processing and Embedded Floating-Point Status and Control Register (SPEFSCR).”
- A 64-bit accumulator, which is described in Section 3.1.2, “Accumulator (ACC).”

3.1.1 Signal Processing and Embedded Floating-Point Status and Control Register (SPEFSCR)

The SPEFSCR, which is shown in Figure 3-1, is used for status and control of SPE instructions.

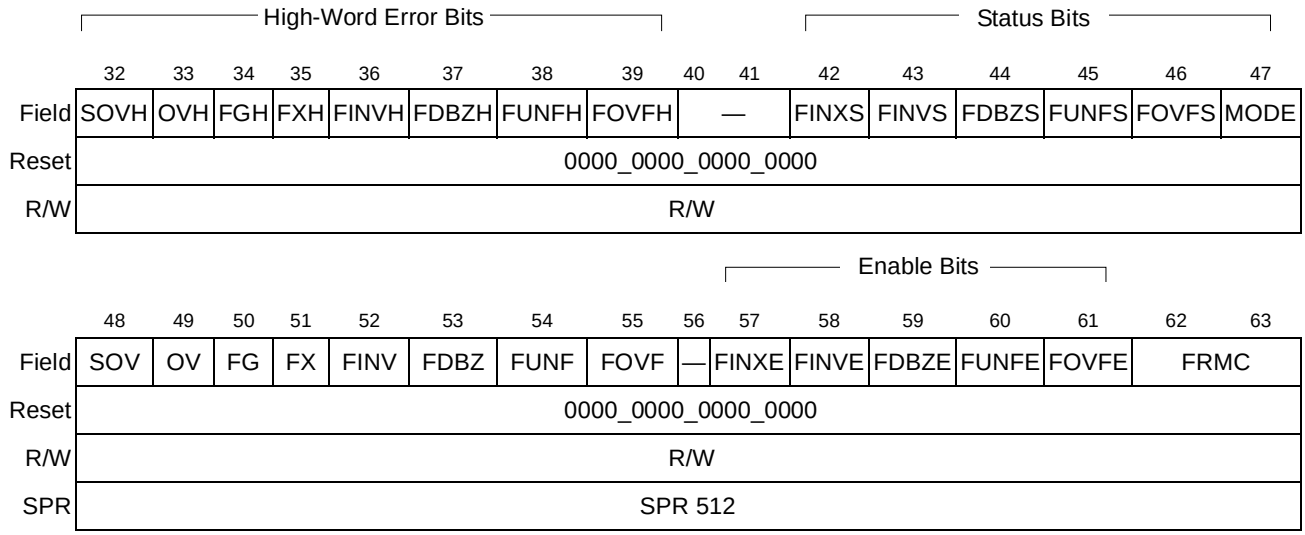

Figure 3-1. Signal Processing and Embedded Floating-Point Status and Control Register (SPEFSCR)

Table 3-1 describes SPEFSCR bits.

Table 3-1. SPEFSCR Field Descriptions

Bits	Name	Function
32	SOVH	Summary integer overflow high, which is set whenever an instruction other than **mtspr** sets OVH. SOVH remains set until a **mtspr[SPEFSCR]** clears it.
33	OVH	Integer overflow high. An overflow occurred in the upper half of the register while executing a SPE integer instruction.
34	FGH	Embedded floating-point guard bit high. Floating-point guard bit from the upper half. The value is undefined if the processor takes a floating-point exception caused by input error, floating-point overflow, or floating-point underflow.
35	FXH	Embedded floating-point sticky bit high. Floating bit from the upper half. The value is undefined if the processor takes a floating-point exception caused by input error, floating-point overflow, or floating-point underflow.
36	FINVH	Embedded floating-point invalid operation error high. Set when an input value on the high side is a NaN, Inf, or Denorm. Also set on a divide if both the dividend and divisor are zero.
37	FDBZH	Embedded floating-point divide by zero error high. Set if the dividend is non-zero and the divisor is zero.
38	FUNFH	Embedded floating-point underflow error high
39	FOVFH	Embedded floating-point overflow error high
40–41	—	Reserved and should be cleared

Table 3-1. SPEFSCR Field Descriptions (continued)

Bits	Name	Function
42	FINXS	Embedded floating-point inexact sticky. FINXS = FINXS | FGH | FXH | FG | FX
43	FINVS	Embedded floating-point invalid operation sticky. Location for software to use when implementing true IEEE floating-point.
44	FDBZS	Embedded floating-point divide by zero sticky. FDBZS = FDBZS | FDBZH | FDBZ
45	FUNFS	Embedded floating-point underflow sticky. Storage location for software to use when implementing true IEEE floating-point.
46	FOVFS	Embedded floating-point overflow sticky. Storage location for software to use when implementing true IEEE floating-point.
47	MODE	Embedded floating-point mode (read-only on e500)
48	SOV	Integer summary overflow. Set whenever an SPE instruction other than **mtspr** sets OV. SOV remains set until **mtspr[SPEFSCR]** clears it.
49	OV	Integer overflow. An overflow occurred in the lower half of the register while a SPE integer instruction was executed.
50	FG	Embedded floating-point guard bit. Floating-point guard bit from the lower half. The value is undefined if the processor takes a floating-point exception caused by input error, floating-point overflow, or floating-point underflow.
51	FX	Embedded floating-point sticky bit. Floating bit from the lower half. The value is undefined if the processor takes a floating-point exception caused by input error, floating-point overflow, or floating-point underflow.
52	FINV	Embedded floating-point invalid operation error. Set when an input value on the high side is a NaN, Inf, or Denorm. Also set on a divide if both the dividend and divisor are zero.
53	FDBZ	Embedded floating-point divide by zero error. Set if the dividend is non-zero and the divisor is zero.
54	FUNF	Embedded floating-point underflow error
55	FOVF	Embedded floating-point overflow error
56	—	Reserved and should be cleared
57	FINXE	Embedded floating-point inexact enable
58	FINVE	Embedded floating-point invalid operation/input error exception enable 0 Exception disabled 1 Exception enabled If the exception is enabled, a floating-point data exception is taken if a floating-point instruction sets FINV or FINVH.
59	FDBZE	Embedded floating-point divide-by-zero exception enable 0 Exception disabled 1 Exception enabled If the exception is enabled, a floating-point data exception is taken if a floating-point instruction sets FDBZ or FDBZH.
60	FUNFE	Embedded floating-point underflow exception enable 0 Exception disabled 1 Exception enabled If the exception is enabled, a floating-point data exception is taken if a floating-point instruction sets FUNF or FUNFH.

Table 3-1. SPEFSCR Field Descriptions (continued)

Bits	Name	Function
61	FOVFE	Embedded floating-point overflow exception enable 0 Exception disabled 1 Exception enabled If the exception is enabled, a floating-point data exception is taken if a floating-point instruction sets FOVF or FOVFH.
62–63	FRMC	Embedded floating-point rounding mode control 00 Round to nearest 01 Round toward zero 10 Round toward +infinity 11 Round toward –infinity

3.1.2 Accumulator (ACC)

The 64-bit architectural accumulator register shown in Figure 3-2 holds the results of multiply accumulate (MAC) forms of SPE integer instructions. The ACC allows back-to-back execution of dependent MAC instructions that are in inner loops of DSP code such as FIR filters. The ACC is partially visible to the programmer; its results need not be read explicitly to be used. Instead, the results are always copied into a 64-bit destination GPR specified by the instruction. The ACC, however, must be explicitly cleared when starting a new MAC loop. Depending on the instruction type, the ACC can hold either a 64-bit value or a vector of two 32-bit elements.

The Initialize Accumulator instruction (**evmra**), which is described in the Instruction Set chapter of *EREF: A Reference for Freescale Book E and the e500 Core*, initializes the ACC.

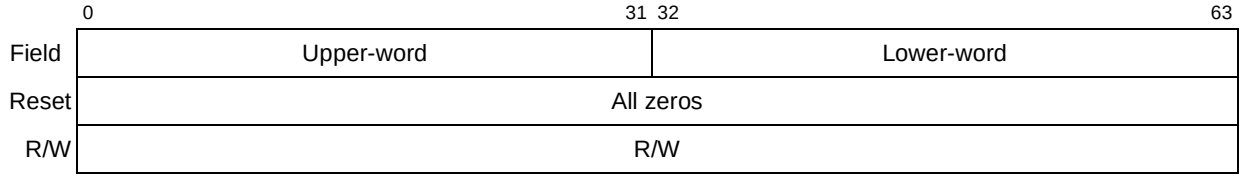

Figure 3-2. Accumulator (ACC)

Table 3-2 describes ACC fields.

Table 3-2. ACC Field Descriptions

Bits	Name	Function
0–31	Upper word	Holds the upper-word accumulate value for SPE multiply with accumulate instructions
32–63	Lower word	Holds the lower-word accumulate value for SPE multiply with accumulate instructions

3.2 Notation

Table 3-3 shows definitions and notation that appear throughout this document.

Table 3-3. Notation Conventions

Symbol	Meaning
X_p	Bit p of register/field X
$X_{p:q}$	Bits p through q of register/field X
$X_{p\ q\ ...}$	Bits p, q,... of register/field X
¬X	The ones complement of the contents of X
Field i	Bits 4×i through 4×i+3 of a register
.	As the last character of an instruction mnemonic, this character indicates that the instruction records status information in certain fields of the condition register as a side effect of execution, as described in the Register Model chapter of *EREF: A Reference for Freescale Book E and the e500 Core*.
||	Describes the concatenation of two values. For example, 010 || 111 is the same as 010111.
x^n	x raised to the n^{th} power.
^nx	Replication of x, n times (i.e., x concatenated to itself n–1 times). $^n0$ and $^n1$ are special cases: $^n0$ means a field of n bits with each bit equal to 0. Thus, ^{5}0 is equivalent to 0b0_0000. $^n1$ means a field of n bits with each bit equal to 1. Thus, ^{5}1 is equivalent to 0b1_1111.
/, //, ///,	Reserved field in an instruction or in a register. Each bit and field in instructions, in status and control registers (such as the XER), and in SPRs is defined, allocated, or reserved.

3.3 Instruction Fields

Table 3-4 describes instruction fields.

Table 3-4. Instruction Field Descriptions

Field	Description
AA (30)	Absolute address bit. 0 The immediate field represents an address relative to the current instruction address. For I-form branch instructions, the effective address of the branch target is the sum 320 || (CIA+EXTS(LI||0b00))32–63. For B-form branch instructions, the effective address of the branch target is the sum 320 || (CIA+EXTS(BD||0b00))32–63. For I-form branch extended instructions, the effective address of the branch target is the sum CIA+EXTS(LI||0b00). For B-form branch extended instructions, the effective address of the branch target is the sum CIA+EXTS(BD||0b00). 1 The immediate field represents an absolute address. For I-form branch instructions, the effective address of the branch target is the value 320 || EXTS(LI||0b00)32–63. For B-form branch instructions, the effective address of the branch target is the value 320 || EXTS(BD||0b00)32–63. For I-form branch extended instructions, the effective address of the branch target is the value EXTS(LI||0b00). For B-form branch extended instructions, the effective address of the branch target is the value EXTS(BD||0b00).
crbA (11–15)	Specifies a condition register bit to be used as a source
crbB (16–20)	Specifies a condition register bit to be used as a source
crbD (16–29)	Immediate field specifying a 14-bit signed two's complement branch displacement that is concatenated on the right with 0b00 and sign-extended to 64 bits.
crfD (6–8)	Specifies a CR field to be used as a target
crfS (11–13)	Specifies a CR field to be used as a source
BI (11–15)	Specifies a condition register bit to be used as the condition of a branch conditional instruction
BO (6–10)	Specifies options for branch conditional instructions
crbD (6–10)	Specifies a CR bit for use as a target
CT (6–10)	Cache touch instructions (**dcbt**, **dcbtst**, and **icbt**) use this field to specify the target portion of the cache facility to place the prefetched data or instructions. This field is implementation-dependent.
D (16–31)	Immediate field that specifies a 16-bit signed two's complement integer that is sign-extended to 64 bits
DE (16–27)	Immediate field that specifies a 12-bit signed two's complement integer that is sign-extended to 64 bits
DES (16–27)	Immediate field that specifies a 12-bit signed two's complement integer that is concatenated on the right with 0b00 and sign-extended to 64 bits
E (15)	Immediate field that specifies a 1-bit value that **wrteei** uses to place in MSR[EE] (external input enable bit)
CRM (12–19)	Field mask that identifies the condition register fields that the **mtcrf** instruction updates

Table 3-4. Instruction Field Descriptions (continued)

Field	Description
LI (6–29)	Immediate field that specifies a 24-bit signed two's complement integer that is concatenated on the right with 0b00 and sign-extended to 64 bits
LK (31)	Link bit that indicates whether the link register (LR) is set. 0 Do not set the LR. 1 Set the LR. The sum of the value 4 and the address of the branch instruction is placed into the LR.
MB (21–25) and ME (26–30)	Fields that M-form rotate instructions use to specify a 64-bit mask consisting of 1s from bit MB+32 through bit ME+32 inclusive and 0s elsewhere
mb (26 || 21–25)	Used in MD-form and MDS-form rotate instructions to specify the first 1-bit of a 64-bit mask
me (26 || 21–25)	Used in MD-form and MDS-form rotate instructions to specify the last 1-bit of a 64-bit mask
MO (6–10)	Specifies the subset of memory accesses that a Memory Barrier instruction (**mbar**) ordered
NB (16–20)	Specifies the number of bytes to move in an immediate Move Assist instruction
OPCD (0–5)	Primary opcode field
rA (11–15)	Specifies a GPR to be used as a source or as a target
rB (16–20)	Specifies a GPR to be used as a source
Rc (31)	Record bit. 0 Do not alter the condition register. 1 Set condition register field 0 or field 1.
RS (6–10)	Specifies a GPR to be used as a source
rD (6–10)	Specifies a GPR to be used as a target
SH (16–20)	Specifies a shift amount in Rotate Word Immediate and Shift Word Immediate instructions
sh (30 || 16–20)	Specifies a shift amount in Rotate Doubleword Immediate and Shift Doubleword Immediate instructions
SIMM (16–31)	Immediate field that specifies a 16-bit signed integer
SPRN (16–20||11–15)	Specifies an SPR for **mtspr** and **mfspr** instructions
TO (6–10)	Specifies the conditions on which to trap
UIMM (16–31)	Immediate field that specifies a 16-bit unsigned integer
WS (18–20)	Specifies a word in the TLB entry that is being accessed
XO (21–29, 21–30, 22–30, 26–30, 27–29, 27–30, 28–31)	Extended opcode field

3.4 Description of Instruction Operation

A series of statements that use a semi-formal language at the register transfer level (RTL) describes the operation of most instructions. RTL uses the general notation that is shown in Table 3-3 and Table 3-4 and conventions that are specific to RTL, shown in Table 3-5. Figure 3-3 gives an example. Some of this notation is used in the formal descriptions of instructions.

The RTL descriptions cover the normal execution of the instruction, except that the standard settings of the condition register, integer exception register, floating-point status, and control register are not always shown. (Nonstandard setting of these registers, such as the setting of the condition register field 0 by the **stwcx** instruction, is shown.) The RTL descriptions do not cover all cases in which the interrupt may be invoked, or for which the results are boundedly undefined, and may not cover all invalid forms.

RTL descriptions specify the architectural transformation that the execution of an instruction performs. They do not imply any particular implementation.

Table 3-5. RTL Notation

Notation	Meaning
$\leftarrow$	Assignment
$\leftarrow_f$	Assignment in which the data may be reformatted in the target location
$\neg$	NOT logical operator (one's complement)
+	Two's complement addition
–	Two's complement subtraction, unary minus
×	Multiplication
÷	Division (yielding quotient)
$+_{dp}$	Floating-point addition, result rounded to double-precision
$-_{dp}$	Floating-point subtraction, result rounded to double-precision
$\times_{dp}$	Floating-point multiplication, product rounded to double-precision
$\div_{dp}$	Floating-point division quotient, rounded to double-precision
$+_{sp}$	Floating-point addition, result rounded to single-precision
$-_{sp}$	Floating-point subtraction, result rounded to single-precision
$\times_{sf}$	Signed fractional multiplication
$\times_{si}$	Signed integer multiplication
$\times_{sp}$	Floating-point multiplication, result rounded to single-precision
$\div_{sp}$	Floating-point division, result rounded to single-precision
$\times_{fp}$	Floating-point multiplication to infinite precision (no rounding)
$\times_{ui}$	Unsigned integer multiplication

Table 3-5. RTL Notation (continued)

Notation	Meaning
FPSquareRoot-Double(x)	Floating-point $\sqrt{x}$, result rounded to double-precision
FPSquareRoot-Single(x)	Floating-point $\sqrt{x}$, result rounded to single-precision
FPReciprocal-Estimate(x)	Floating-point estimate of $\frac{1}{x}$
FPReciprocal-SquareRoot-Estimate(x)	Floating-point estimate of $\frac{1}{\sqrt{x}}$
Allocate-DataCache-Block(x)	If the block containing the byte addressed by x does not exist in the data cache, allocate a block in the data cache and set the contents of the block to 0.
Flush-DataCache-Block(x)	If the block containing the byte addressed by x exists in the data cache and is dirty, the block is written to main memory and is removed from the data cache.
Invalidate-DataCache-Block(x)	If the block containing the byte addressed by x exists in the data cache, the block is removed from the data cache.
Store-DataCache-Block(x)	If the block containing the byte addressed by x exists the data cache and is dirty, the block is written to main memory but may remain in the data cache.
Prefetch-DataCache-Block(x,y)	If the block containing the byte addressed by x does not exist in the portion of the data cache specified by y, the block in memory is copied into the data cache.
Prefetch-ForStore-DataCache-Block(x,y)	If the block containing the byte addressed by x does not exist in the portion of the data cache specified by y, the block in memory is copied into the data cache and made exclusive to the processor that is executing the instruction.
ZeroDataCache-Block(x)	The contents of the block containing the byte addressed by x in the data cache is cleared.
Invalidate-Instruction-CacheBlock(x)	If the block containing the byte addressed by x is in the instruction cache, the block is removed from the instruction cache.
Prefetch-Instruction-CacheBlock(x,y)	If the block containing the byte addressed by x does not exist in the portion of the instruction cache specified by y, the block in memory is copied into the instruction cache.
=, ≠	Equal to, Not Equal to relations
<, ≤, >, ≥	Signed comparison relations
$<_u$, $>_u$	Unsigned comparison relations
?	Unordered comparison relation
&, |	AND, OR logical operators
⊕, ≡	Exclusive OR, Equivalence logical operators ((a≡b) = (a⊕¬b))
ABS(x)	Absolute value of x
APID(x)	Returns an implementation-dependent information on the presence and status of the auxiliary processing extensions specified by x
CEIL(x)	Least integer ≥ x
CnvtFP32ToI32Sat(fp, signed,upper_lower,round, fractional)	Converts a 32 bit floating point number to a 32 bit integer if possible, otherwise it saturates.

Table 3-5. RTL Notation (continued)

Notation	Meaning
CnvtI32ToFP32Sat (v,signed,upper_lower, fractional)	Converts a 32 bit integer to a 32 bit floating point number if possible, otherwise it saturates.
EXTS(x)	Result of extending x on the left with signed bits
EXTZ(x)	Result of extending x on the left with zeros
GPR(x)	General purpose register x
MASK(x, y)	Mask that has ones in bit positions x through y (wrapping if x>y) and zeros elsewhere
MEM(x,1)	Contents of the byte of memory located at address x
MEM(x,y)(for y={2,4,8})	Contents of y bytes of memory starting at address x. • If big-endian memory, the byte at address x is the MSB and the byte at address x+y–1 is the LSB of the value being accessed. • If little-endian memory, the byte at address x is the LSB and the byte at address x+y– 1 is the MSB of the value being accessed.
MOD(x,y)	Modulo y of x (remainder of x divided by y)
ROTL32(x, y)	Result of rotating the value x||x left y positions, where x is 32 bits long
SINGLE(x)	Result of converting x from floating-point double format to floating-point single format
SPREG(x)	Special-purpose register x
TRAP	Invoke a trap-type program interrupt
characterization	Reference to setting status bits in a standard way that is explained in the text
undefined	Undefined value that may vary between implementations and between different executions on the same implementation
CIA	Current instruction address, which is the address of the instruction that is described in RTL. Used by relative branches to set the next instruction address (NIA) and by branch instructions with LK=1 to set the LR. CIA does not correspond to any architected register.
NIA	Next instruction address, and the address of the next instruction to be executed. For a successful branch, the next instruction address is the branch target address: in RTL, indicated by assigning a value to NIA. For other instructions that cause non-sequential instruction fetching, the RTL is similar. For instructions that do not branch, and do not otherwise cause instruction fetching to be non-sequential, the next instruction address is CIA+4. NIA does not correspond to any architected register.
if … then … else …	Conditional execution indenting shows range; else is optional.
do	Do loop, indenting shows range. ‘To’ and/or ‘by’ clauses specify incrementing an iteration variable, and a ‘while’ clause gives termination conditions.
leave	Leave innermost do loop, or do loop described in leave statement

Table 3-6 summarizes precedence rules for RTL operators. Operators that are higher in the table are applied before those that are lower in the table. Operators at the same level in the table associate from left to right, from right to left, or not at all, as shown. (For example, the – operator associates from left to right, so a–b–c = (a–b)–c.) Using parentheses can increase clarity or override the evaluation order that the table implies; parenthesized expressions are evaluated before serving as parameters.

Table 3-6. Operator Precedence

Operators	Associativity
Subscript, function evaluation	Left to right
Pre-superscript (replication), post-superscript (exponentiation)	Right to left
unary –, ¬	Right to left
×, ÷	Left to right
+, –	Left to right
||	Left to right
=, ≠, <, ≤, >, ≥, $<_u$, $>_u$, ?	Left to right
&, ⊕, ≡	Left to right
|	Left to right
: (range)	None
←	None

3.5 Intrinsics

The rest of this chapter describes individual instructions, which are listed in alphabetical order by mnemonic. Figure 3-3 shows the format for instruction description pages.

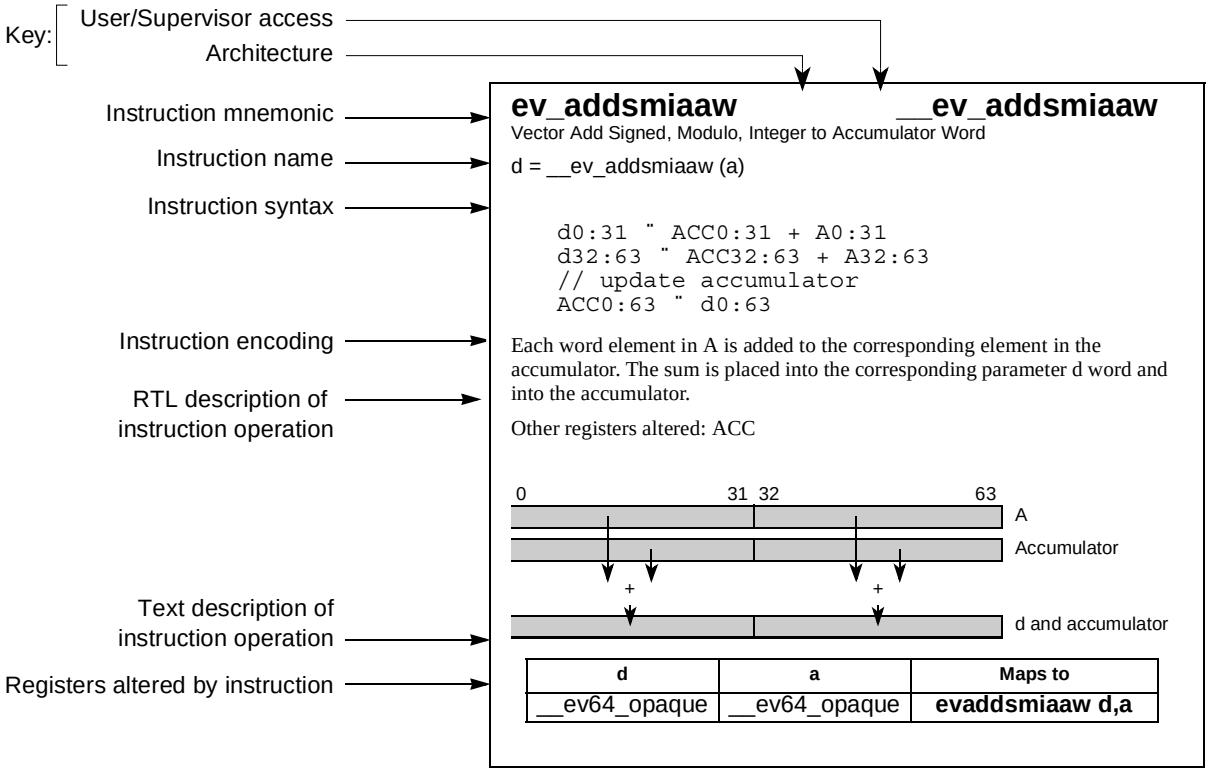

Figure 3-3. Instruction Description

3.5.1 Intrinsic Definitions

For saturation, left shifts, and bit reversal, the pseudo RTL is provided here to more accurately describe those functions that are referenced in the instruction pseudo RTL.

3.5.1.1 Saturation

```
SATURATE(overflow, carry, saturated_underflow, saturated_overflow, value)

if overflow then
    if carry then
        return saturated_underflow
    else
        return saturated_overflow
else
    return value
```

3.5.1.2 Shift

```
SL(value, cnt)

if cnt > 31 then
    return 0
else
    return (value << cnt)
```

3.5.1.3 Bit Reverse

```
BITREVERSE(value)

result ← 0
mask ← 1
shift ← 31
cnt ← 32
while cnt > 0 then do
    t ← data & mask
    if shift >= 0 then
        result ← (t << shift) | result
    else
        result ← (t >> -shift) | result
    cnt ← cnt - 1
    shift ← shift - 2
    mask ← mask << 1
return result
```

__brinc __brinc

Bit Reversed Increment

d = __brinc(a,b)

```
n ← MASKBITS                    // Imp dependent # of mask bits
mask ← b64-n:63                 // Least sig. n bits of register
temp0 ← a64-n:63
temp1 ← bitreverse(1 + bitreverse(a | (¬mask)))
d ← a0:63-n || (d & mask)
```

brinc provides a way for software to access FFT data in a bit-reversed manner. Parameter a contains the index into a buffer that contains data on which FFT is to be performed. Parameter b contains a mask that allows the index to be updated with bit-reversed addressing. Typically this instruction precedes a load with index instruction; for example,

```
brinc r2, r3, r4
lhax r8, r5, r2
```

Parameter b contains a bit-mask that is based on the number of points in an FFT. To access a buffer containing n byte sized data that is to be accessed with bit-reversed addressing, the mask has $\log_2 n$ 1s in the least significant bit positions and 0s in the remaining most significant bit positions. If, however, the data size is a multiple of a half word or a word, the mask is constructed so that the 1s are shifted left by $\log_2$ (size of the data) and 0s are placed in the least significant bit positions. Table 3-7 shows example values of masks for different data sizes and number of data.

Table 3-7. Data Samples and Sizes

Number of Data Samples	Byte	Half Word	Word	Double Word
8	000...00000111	000...00001110	000...000011100	000...0000111000
16	000...00001111	000...00011110	000...000111100	000...0001111000
32	000...00011111	000...00111110	000...001111100	000...0011111000
64	000...00111111	000...01111110	000...011111100	000...0111111000

d	a	b	Maps to
uint32_t	uint32_t	uint32_t	**brinc d,a,b**

Architecture Note: An implementation can restrict the number of bits specified in a mask. The number of bits in a mask may not exceed 32.

Architecture Note: This instruction only modifies the lower 32 bits of the destination register in 32-bit implementations. For 64-bit implementations in 32-bit mode, the contents of the upper 32 bits of the destination register are undefined.

Architecture Note: Execution of brinc does not cause SPE Unavailable exceptions, regardless of the state of MSRSPE.

__ev_abs __ev_abs

Vector Absolute Value

d = __ev_abs(a)

$d_{0:31} \leftarrow ABS(a_{0:31})$
$d_{32:63} \leftarrow ABS(a_{32:63})$

The absolute value of each element of a parameter is placed in the corresponding elements of parameter d. An absolute value of 0x8000_0000 (most negative number) returns 0x8000_0000. No overflow is detected.

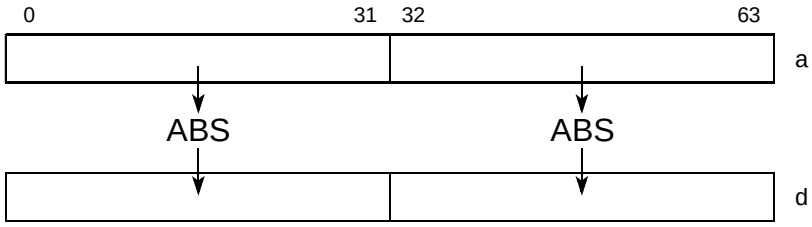

Figure 3-4. Vector Absolute Value (__ev_abs)

d	a	Maps to
__ev64_opaque	__ev64_opaque	**evabs d,a**

__ev_addiw __ev_addiw

Vector Add Immediate Word

d= __ev_addiw (a,b)

```
d0:31 ← a0:31 + EXTZ(b)// Modulo sum
d32:63 ←a32:63 + EXTZ(b)// Modulo sum
```

Parameter b is zero-extended and added to both the high and low elements of parameter a and the results are placed in the parameter d.

NOTE

The same value is added to both elements of the register.

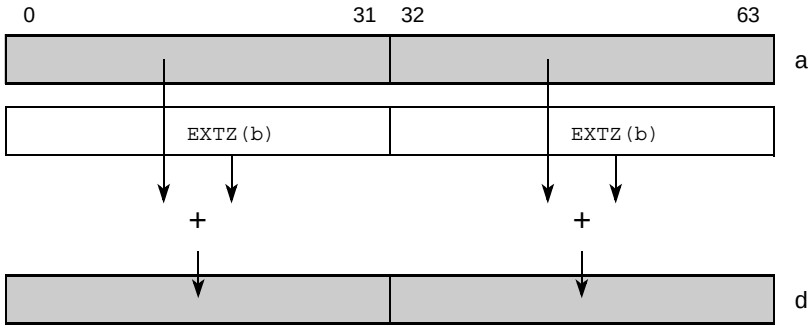

Figure 3-5. Vector Add Immediate Word (__ev_addiw)

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	5-bit unsigned literal	**evaddiw d,a,b**

__ev_addsmiaaw __ev_addsmiaaw

Vector Add Signed, Modulo, Integer to Accumulator Word

d = __ev_addsmiaaw (a)

```
// high
d0:31 ← ACC0:31 + a0:31// low
d32:63 ← ACC32:63 + a32:63
// update accumulator
ACC0:63 ← d0:63
```

Each word element in parameter a is added to the corresponding element in the accumulator and the results are placed in parameter d and into the accumulator.

Other registers altered: ACC

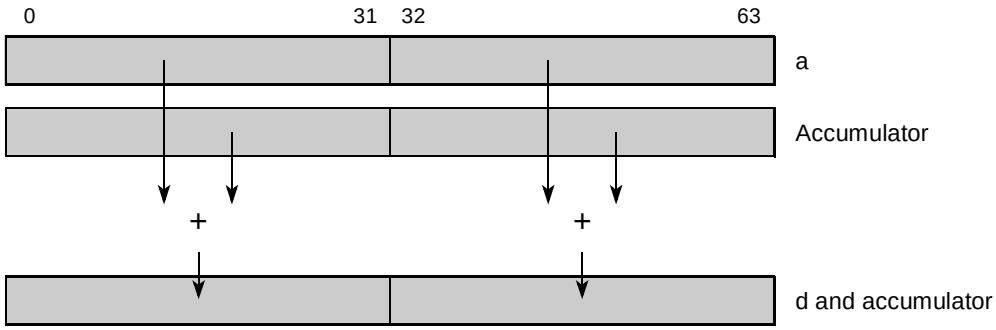

Figure 3-6. Vector Add Signed, Modulo, Integer to Accumulator Word (__ev_addsmiaaw)

d	a	Maps to
__ev64_opaque	__ev64_opaque	**evaddsmiaaw d,a**

__ev_addssiaaw __ev_addssiaaw

Vector Add Signed, Saturate, Integer to Accumulator Word

d = __ev_addssiaaw (a)

```
// high
temp0:63 ← EXTS(ACC0:31) + EXTS(a0:31)
ovh ← temp31 ⊕ temp32
d0:31 ← SATURATE(ovh, temp31, 0x80000000, 0x7fffffff, temp32:63)

// low
temp0:63 ← EXTS(ACC32:63) + EXTS(a32:63)
ovl ← temp31 ⊕ temp32
d32:63 ← SATURATE(ovl, temp31, 0x80000000, 0x7fffffff, temp32:63)

ACC0:63 ← d0:63

SPEFSCROVH ← ovh
SPEFSCROV ← ovl
SPEFSCRSOVH ← SPEFSCRSOVH | ovh
SPEFSCRSOV ← SPEFSCRSOV | ovl
```

Each signed integer word element in parameter a is sign-extended and added to the corresponding sign-extended element in the accumulator, saturating if overflow or underflow occurs, and the results are placed in parameter d and the accumulator. Any overflow or underflow is recorded in the SPEFSCR overflow and summary overflow bits.

Other registers altered: SPEFSCR ACC

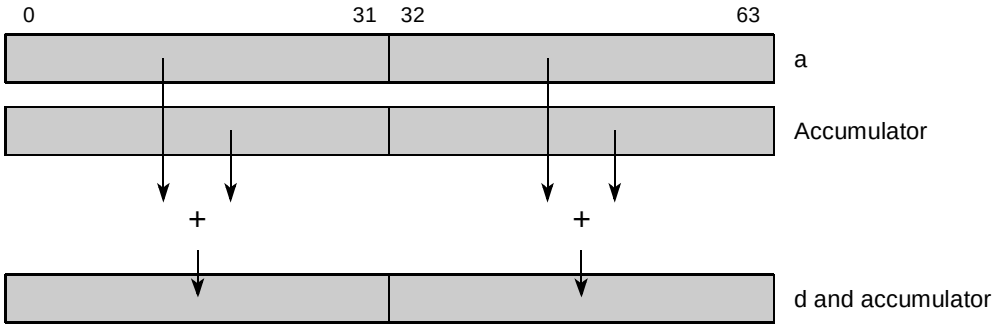

Figure 3-7. Vector Add Signed, Saturate, Integer to Accumulator Word (__ev_addssiaaw)

d	a	Maps to
__ev64_opaque	__ev64_opaque	**evaddssiaaw d,a**

__ev_addumiaaw __ev_addumiaaw

Vector Add Unsigned, Modulo, Integer to Accumulator Word

d = __ev_addumiaaw (a)

$$d_{0:31} \leftarrow ACC_{0:31} + a_{0:31}$$
$$d_{32:63} \leftarrow ACC_{32:63} + a_{32:63}$$

$$ACC_{0:63} \leftarrow d_{0:63}$$

Each unsigned integer word element in the parameter a is added to the corresponding element in the accumulator and the results are placed in the parameter d and the accumulator.

Other registers altered: ACC

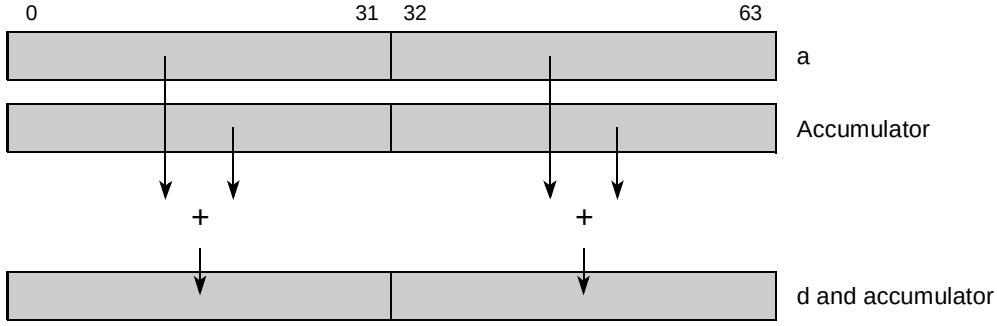

Figure 3-8. Vector Add Unsigned, Modulo, Integer to Accumulator Word (__ev_addumiaaw)

d	a	Maps to
__ev64_opaque	__ev64_opaque	**evaddumiaaw d,a**

__ev_addusiaaw __ev_addusiaaw

Vector Add Unsigned, Saturate, Integer to Accumulator Word

d = __ev_addusiaaw (a)

```
// high
temp0:63 ← EXTZ(ACC0:31) + EXTZ(a0:31)
ovh ← temp31
d0:31 ← SATURATE(ovh, temp31, 0xffffffff, 0xffffffff, temp32:63)

// low
temp0:63 ← EXTZ(ACC32:63) + EXTZ(a32:63)
ovl ← temp31
d32:63 ← SATURATE(ovl, temp31, 0xffffffff, 0xffffffff, temp32:63)

ACC0:63 ← d0:63

SPEFSCROVH ← ovh
SPEFSCROV ← ovl
SPEFSCRSOVH ← SPEFSCRSOVH | ovh
SPEFSCRSOV ← SPEFSCRSOV | ovl
```

Each unsigned integer word element in parameter a is zero-extended and added to the corresponding zero-extended element in the accumulator, saturating if overflow occurs, and the results are placed in parameter d and the accumulator. Any overflow is recorded in the SPEFSCR overflow and summary overflow bits.

Other registers altered: SPEFSCR ACC

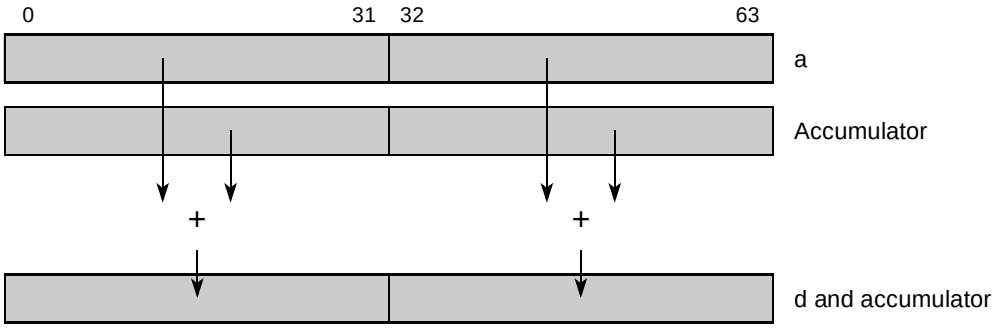

Figure 3-9. Vector Add Unsigned, Saturate, Integer to Accumulator Word (__ev_addusiaaw)

d	a	Maps to
__ev64_opaque	__ev64_opaque	**evaddusiaaw d,a**

__ev_addw __ev_addw

Vector Add Word

d = __ev_addw (a,b)

```
d0:31 ← a0:31 + b0:31   // Modulo sum
d32:63 ← a32:63 + b32:63 // Modulo sum
```

The corresponding elements of parameters a and b are added, and the results are placed in parameter d. The sum is a modulo sum.

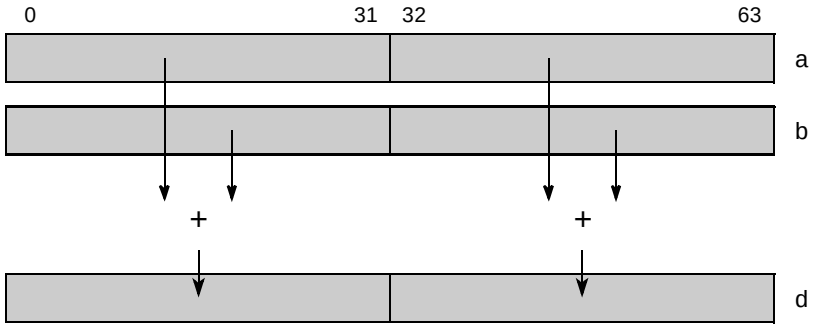

Figure 3-10. Vector Add Word (__ev_addw)

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	**evaddw d,a,b**

__ev_all_eq __ev_all_eq

Vector All Equal

d = __ev_all_eq(a,b)

```
if ( a0:31 = b0:31) & (a32:63 = b32:63)) then d ← true
else d ←false
```

This intrinsic returns true if both the upper 32 bits of parameter a are equal to the upper 32 bits of parameter b and the lower 32 bits of parameter a are equal to the lower 32 bits of parameter b.

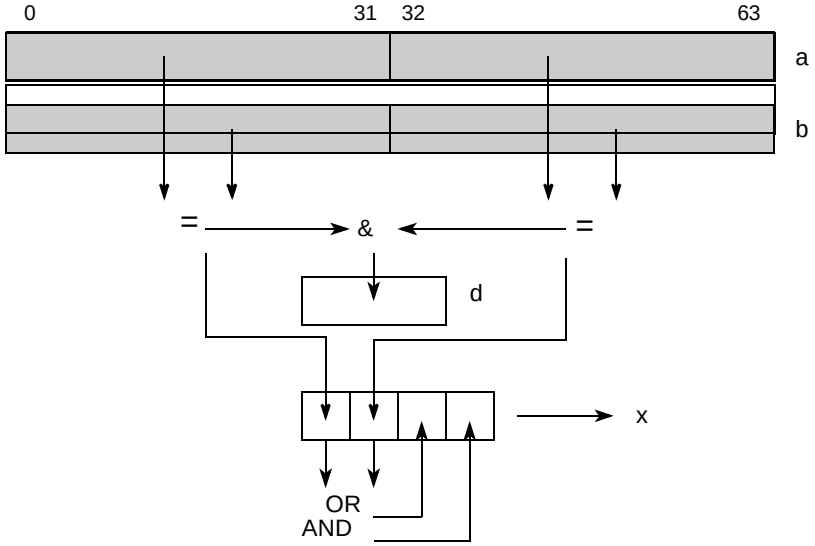

Figure 3-11. Vector All Equal (__ev_all_eq)

d	a	b	Maps to
_Bool	__ev64_opaque	__ev64_opaque	**evcmpeq x,a,b**

__ev_all_fs_eq __ev_all_fs_eq

Vector All Floating-Point Equal

d = __ev_all_fs_eq(a,b)

```
if ( (a0:31 = b0:31) & (a32:63 = b32:63)) then d ← true
else d ←false
```

This intrinsic returns true if both the upper 32 bits of parameter a are equal to the upper 32 bits of parameter b and the lower 32 bits of parameter a are equal to the lower 32 bits of parameter b.

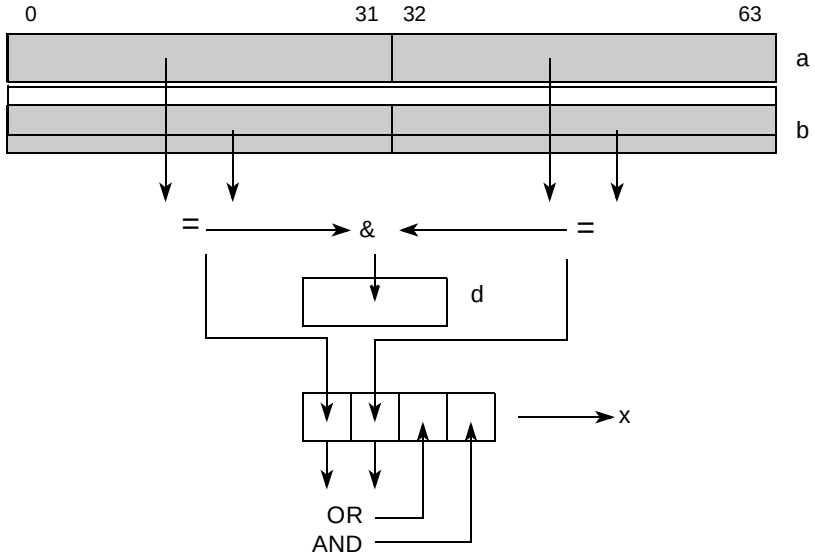

Figure 3-12. Vector All Floating-Point Equal (__ev_all_fs_eq)

d	a	b	Maps to
_Bool	__ev64_opaque	__ev64_opaque	**evfscmplt x,a,b**

__ev_all_fs_gt __ev_all_fs_gt

Vector All Floating-Point Greater Than

d = __ev_all_fs_gt(a,b)

```
if ( a0:31 > b0:31) & (a32:63 > b32:63)) then d ← true
else d ←false
```

This intrinsic returns true if both the upper 32 bits of parameter a are greater than the upper 32 bits of parameter b and the lower 32 bits of parameter a are greater than the lower 32 bits of parameter b.

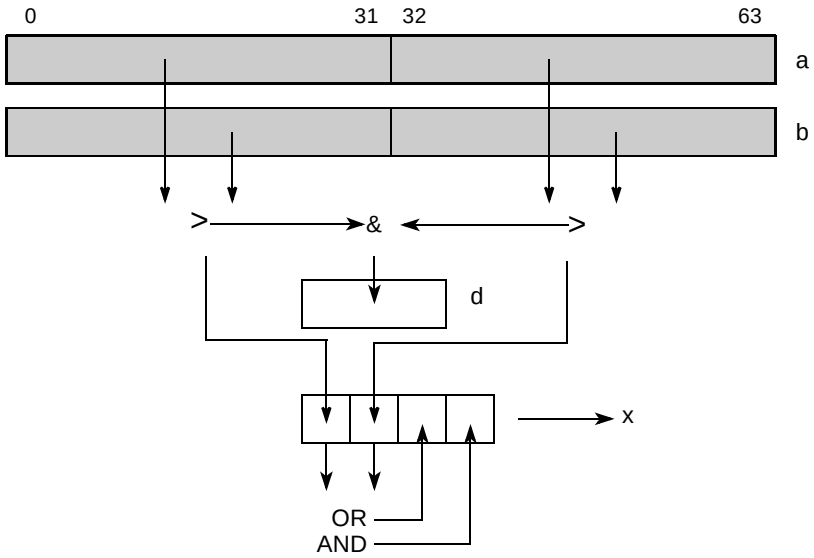

Figure 3-13. Vector All Floating-Point Greater Than (__ev_all_fs_gt)

d	a	b	Maps to
_Bool	__ev64_opaque	__ev64_opaque	**evfscmpgt x,a,b**

__ev_all_fs_lt __ev_all_fs_lt

Vector All Floating-Point Less Than

d = __ev_all_fs_lt(a,b)

```
if ( (a0:31 < b0:31) & (a32:63 < b32:63)) then d ← true
else d ←false
```

This intrinsic returns true if both the upper 32 bits of parameter a are less than the upper 32 bits of parameter b, and the lower 32 bits of parameter a are less than the lower 32 bits of parameter b.

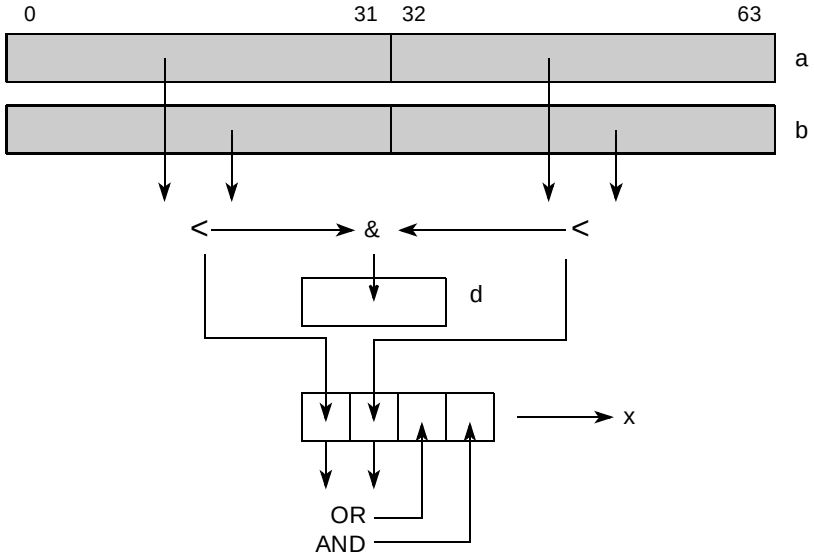

Figure 3-14. Vector All Floating-Point Less Than (__ev_all_fs_lt)

d	a	b	Maps to
_Bool	__ev64_opaque	__ev64_opaque	**evfscmplt x,a,b**

__ev_all_fs_tst_eq __ev_all_fs_tst_eq

Vector All Floating-Point Test Equal

d = __ev_all_fs_tst_eq(a,b)

```
if ( (a0:31 = unsigned b0:31) & (a32:63 = unsigned b32:63)) then d ← true
else d ←false
```

This intrinsic returns true if both the upper 32 bits of parameter a are equal to the upper 32 bits of parameter b, and the lower 32 bits of parameter a are equal to the lower 32 bits of parameter b. This intrinsic differs from __ev_all_fs_eq because no exceptions are taken during its execution. If strict IEEE 754 compliance is required, use __ev_all_fs_eq instead.

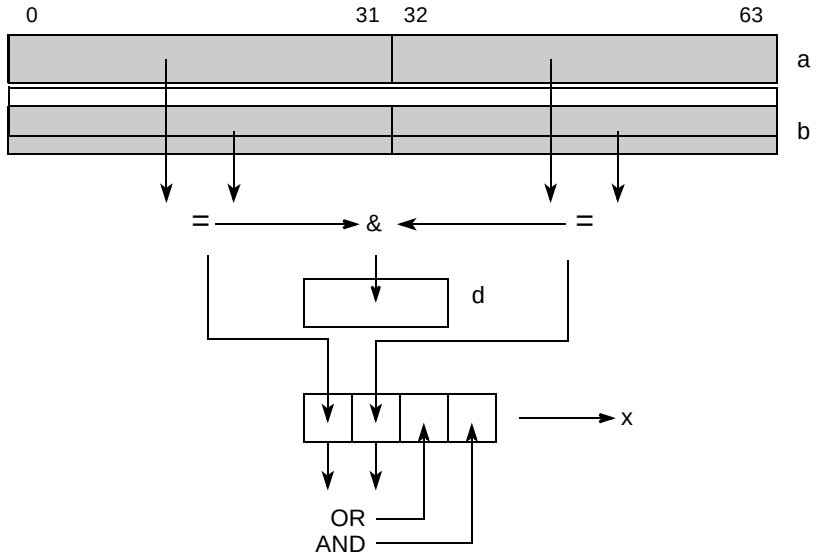

Figure 3-15. Vector All Floating-Point Test Equal (__ev_all_fs_tst_eq)

d	a	b	Maps to
_Bool	__ev64_opaque	__ev64_opaque	**evfststeq x,a,b**

__ev_all_fs_tst_gt __ev_all_fs_tst_gt

Vector All Floating-Point Test Greater Than

d = __ev_all_fs_tst_gt(a,b)

```
if ( (a0:31 > b0:31) & (a32:63 > b32:63)) then d ← true
else d ←false
```

This intrinsic returns true if both the upper 32 bits of parameter a are greater than the upper 32 bits of parameter b and the lower 32 bits of parameter a are greater than the lower 32 bits of parameter b. This intrinsic differs from __ev_all_fs_gt because no exceptions are taken during its execution. If strict IEEE 754 compliance is required, use __ev_all_fs_gt instead.

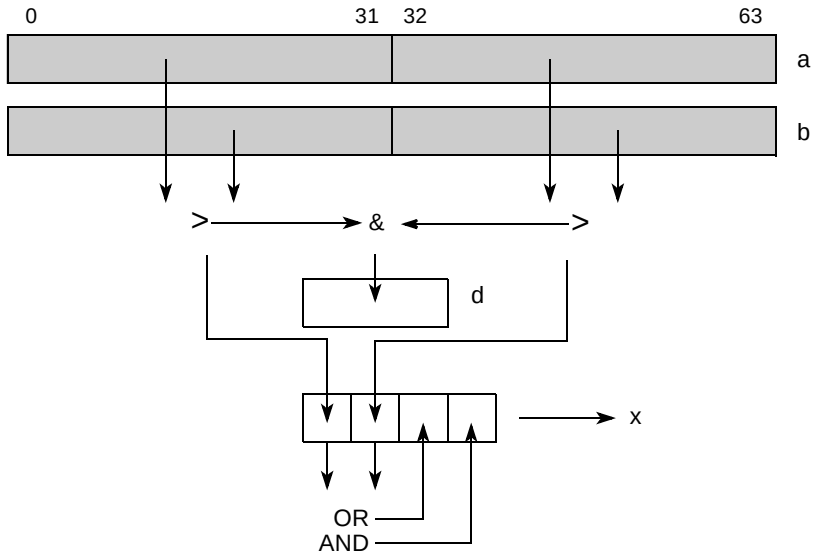

Figure 3-16. Vector All Floating-Point Test Greater Than (__ev_all_fs_tst_gt)

d	a	b	Maps to
_Bool	__ev64_opaque	__ev64_opaque	**evfststgt x,a,b**

__ev_all_fs_tst_lt __ev_all_fs_tst_lt

Vector All Floating-Point Test Less Than

d = __ev_all_fs_tst_lt(a,b)

```
if ( (a0:31 < b0:31) & (a32:63 < b32:63)) then d ← true
else d ←false
```

This intrinsic returns true if both the upper 32 bits of parameter a are less than the upper 32 bits of parameter b and the lower 32 bits of parameter a are less than the lower 32 bits of parameter b. This intrinsic differs from __ev_all_fs_lt because no exceptions are taken during its execution. If strict IEEE 754 compliance is required, use __ev_all_fs_lt instead.

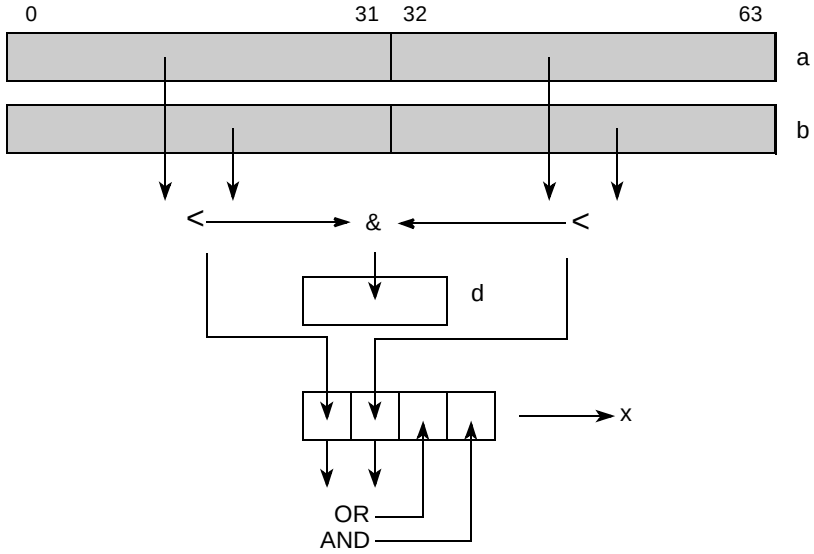

Figure 3-17. Vector All Floating-Point Test Less Than (__ev_all_fs_tst_lt)

d	a	b	Maps to
_Bool	__ev64_opaque	__ev64_opaque	**evfststlt x,a,b**

__ev_all_gts __ev_all_gts

Vector All Greater Than Signed

d = __ev_all_gts(a,b)

```
if ( (a0:31 >signed b0:31) & (a32:63 >signed b32:63)) then d ← true
else d ←false
```

This intrinsic returns true if both the upper 32 bits of parameter a are greater than the upper 32 bits of parameter b and the lower 32 bits of parameter a are greater than the lower 32 bits of parameter b.

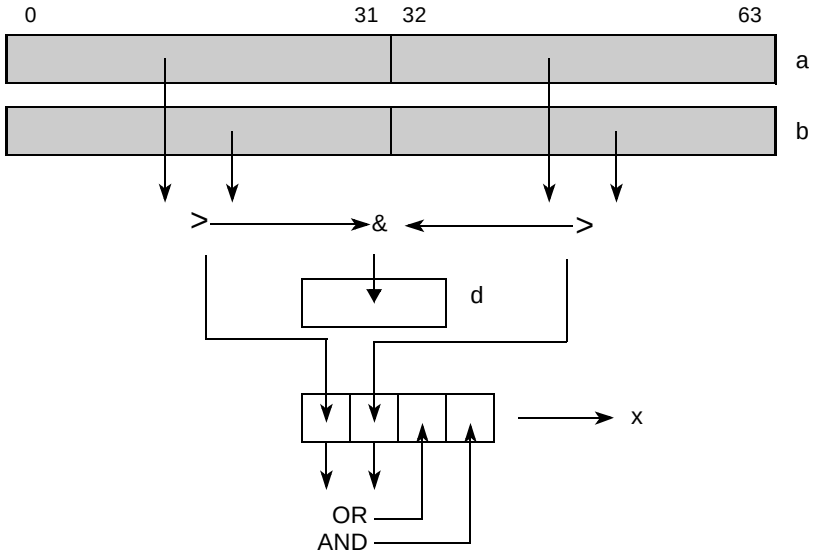

Figure 3-18. Vector All Greater Than Signed (__ev_all_gts)

d	a	b	Maps to
_Bool	__ev64_opaque	__ev64_opaque	**evcmpgts x,a,b**

__ev_all_gtu __ev_all_gtu

Vector All Elements Greater Than Unsigned

d = __ev_all_gtu(a,b)

```
if ( (a0:31 > unsigned b0:31) & (a32:63 > unsigned b32:63)) then d ← true
else a ←false
```

This intrinsic returns true if both the upper 32 bits of parameter a are greater than the upper 32 bits of parameter b and the lower 32 bits of parameter a are greater than the lower 32 bits of parameter b.

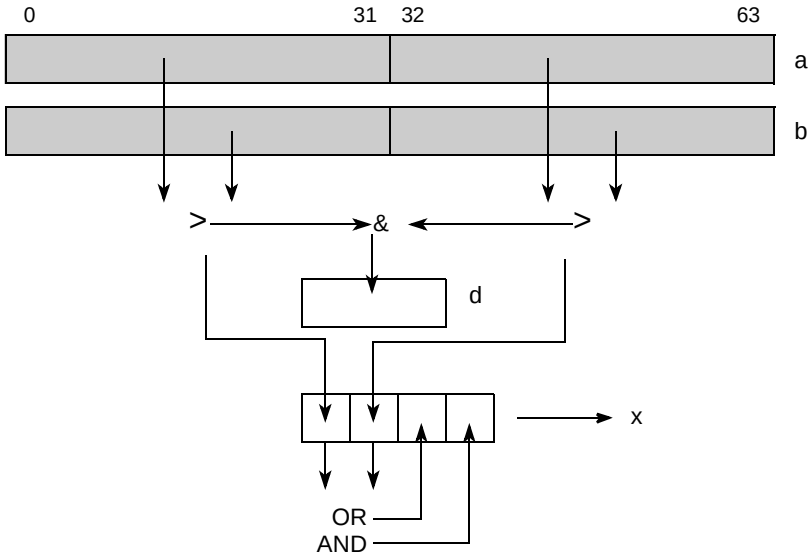

Figure 3-19. Vector All Greater Than Unsigned (__ev_all_gtu)

d	a	b	Maps to
_Bool	__ev64_opaque	__ev64_opaque	**evcmpgtu x,a,b**

__ev_all_lts __ev_all_lts

Vector All Elements Less Than Signed

d = __ev_all_lts(a,b)

```
if ( (a0:31 <signed b0:31) & (a32:63 <signed b32:63)) then d ← true
else d ←false
```

This intrinsic returns true if both the upper 32 bits of parameter a are less than the upper 32 bits of parameter b and the lower 32 bits of parameter a are less than the lower 32 bits of parameter b.

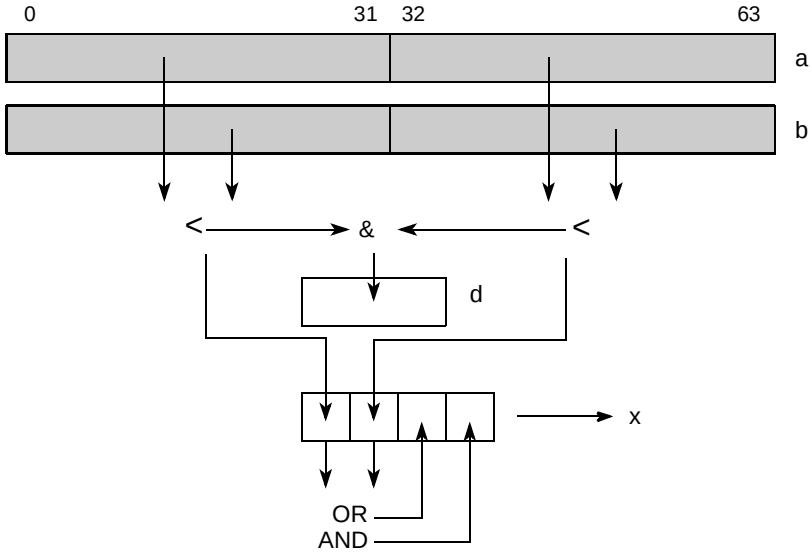

Figure 3-20. Vector All Less Than Signed (__ev_all_lts)

d	a	b	Maps to
_Bool	__ev64_opaque	__ev64_opaque	**evcmplts x,a,b**

__ev_all_ltu __ev_all_ltu

Vector All Elements Less Than Unsigned

d = __ev_all_ltu(a,b)

```
if ( (a0:31 <unsigned b0:31) & (a32:63 <unsigned b32:63)) then d ← true
else d ←false
```

This intrinsic returns true if both the upper 32 bits of parameter a are less than the upper 32 bits of parameter b and the lower 32 bits of parameter a are less than the lower 32 bits of parameter b.

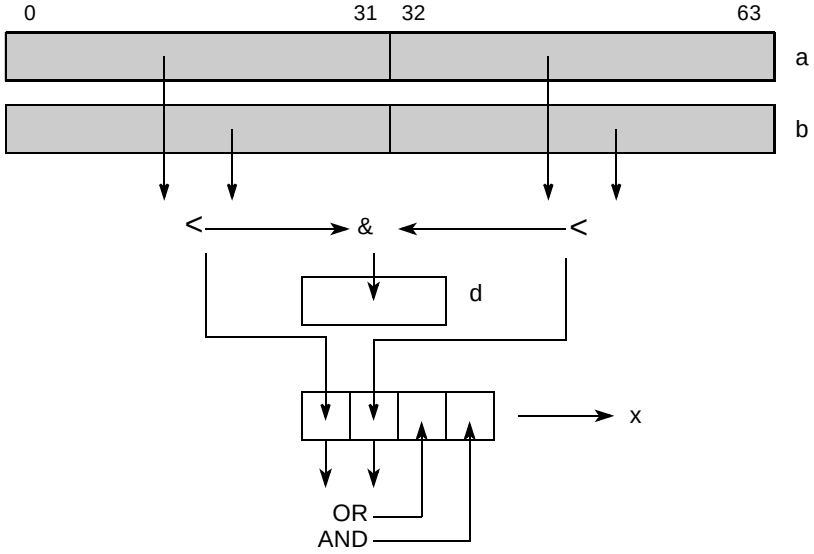

Figure 3-21. Vector All Less Than Unsigned (__ev_all_ltu)

d	a	b	Maps to
_Bool	__ev64_opaque	__ev64_opaque	**evcmpltu x,a,b**

__ev_and __ev_and

Vector AND

d = __ev_and (a,b)

```
d0:31 ← a0:31 & b0:31  // Bitwise AND
d32:63 ← a32:63 & b32:63// Bitwise AND
```

The corresponding elements of parameters a and b are ANDed bitwise, and the results are placed in the corresponding element of parameter d.

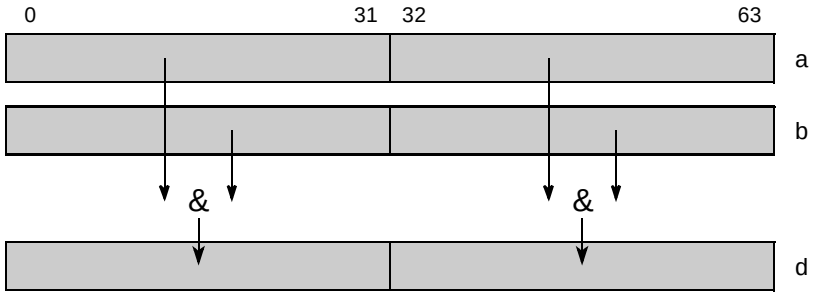

Figure 3-22. Vector AND (__ev_and)

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	**evand d,a,b**

__ev_andc __ev_andc

Vector AND with Complement

d = __ev_andc(a,b)

```
d0:31 ← a0:31 & (¬b0:31) // Bitwise ANDC
d32:63 ← a32:63 & (¬b32:63) // Bitwise ANDC
```

The word elements of parameter a and are ANDed bitwise with the complement of the corresponding elements of parameter b. The results are placed in the corresponding element of parameter d.

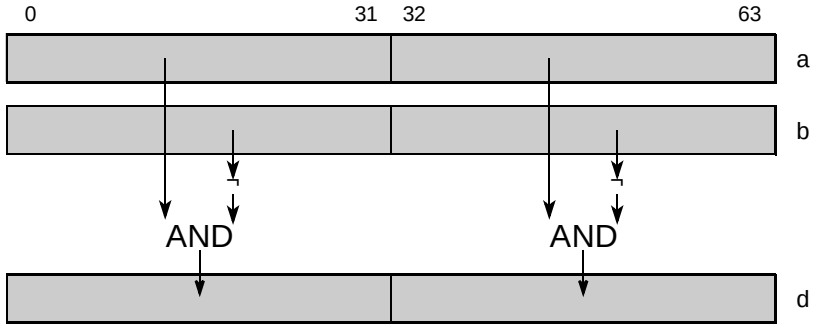

Figure 3-23. Vector AND with Complement (__ev_andc)

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	**evandc d,a,b**

__ev_any_eq __ev_any_eq

Vector Any Equal

d = __ev_any_eq(a,b)

```
if ( (a0:31 = b0:31) | (a32:63 = b32:63)) then d← true
else d ←false
```

This intrinsic returns true if either the upper 32 bits of parameter a are equal to the upper 32 bits of parameter b or the lower 32 bits of parameter a are equal to the lower 32 bits of parameter b.

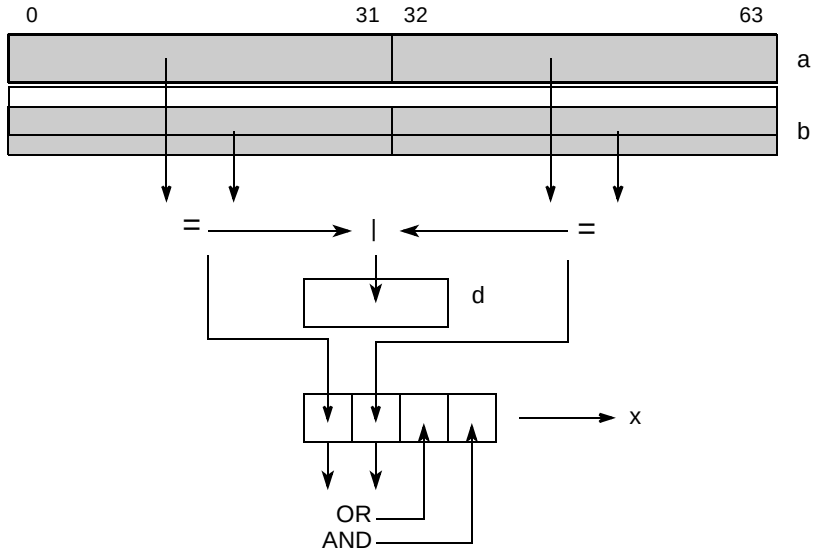

Figure 3-24. Vector Any Equal (__ev_any_eq)

d	a	b	Maps to
_Bool	__ev64_opaque	__ev64_opaque	**evcmpeq x,a,b**

__ev_any_fs_eq __ev_any_fs_eq

Vector Any Floating-Point Equal

d = __ev_any_fs_eq(a,b)

```
if ( (a0:31 = b0:31) | (a32:63 = b32:63)) then d ← true
else d ←false
```

This intrinsic returns true if either the upper 32 bits of parameter a are equal to the upper 32 bits of parameter b or the lower 32 bits of parameter a are equal to the lower 32 bits of parameter b.

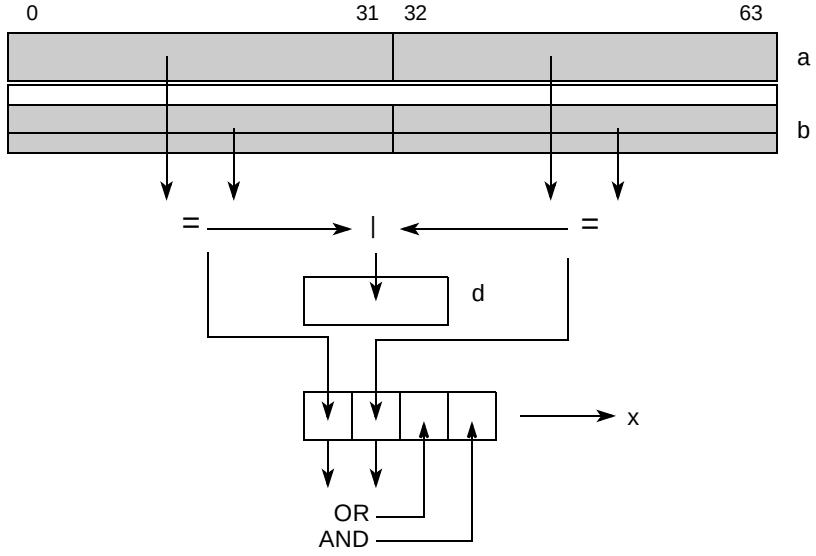

Figure 3-25. Vector Any Floating-Point Equal (__ev_any_fs_eq)

d	a	b	Maps to
_Bool	__ev64_opaque	__ev64_opaque	**evfscmpeq x,a,b**

__ev_any_fs_gt __ev_any_fs_gt

Vector Any Floating-Point Greater Than

d = __ev_any_fs_gt(a,b)

```
if ( (a0:31 > b0:31) | (a32:63 > b32:63)) then d ← true
else d ←false
```

This intrinsic returns true if either the upper 32 bits of parameter a are greater than the upper 32 bits of parameter b or the lower 32 bits of parameter a are greater than the lower 32 bits of parameter b.

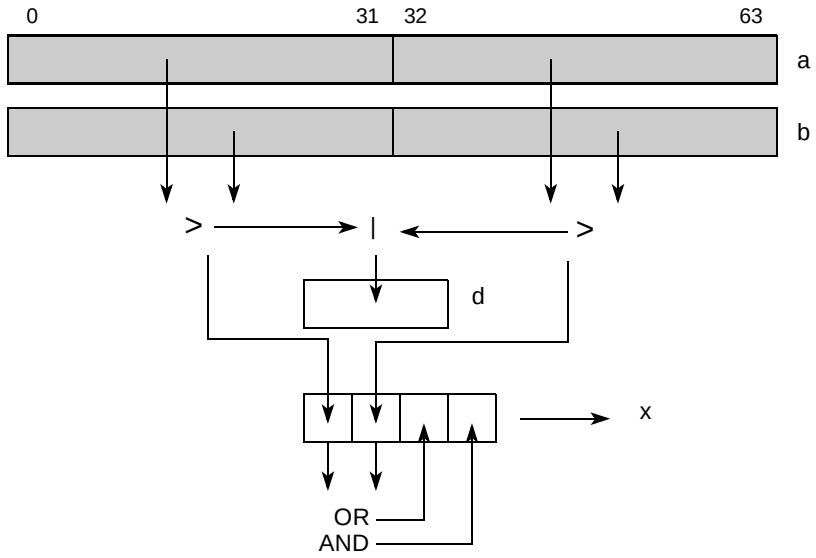

Figure 3-26. Vector Any Floating-Point Greater Than (__ev_any_fs_gt)

d	a	b	Maps to
_Bool	__ev64_opaque	__ev64_opaque	**evfscmpgt x,a,b**

__ev_any_fs_lt __ev_any_fs_lt

Vector Any Floating-Point Less Than

d = __ev_any_fs_lt(a,b)

```
if ( (a0:31 < b0:31) | (a32:63 < b32:63)) then d ← true
else d ←false
```

This intrinsic returns true if either the upper 32 bits of parameter a are less than the upper 32 bits of parameter b or the lower 32 bits of parameter a are less than the lower 32 bits of parameter b.

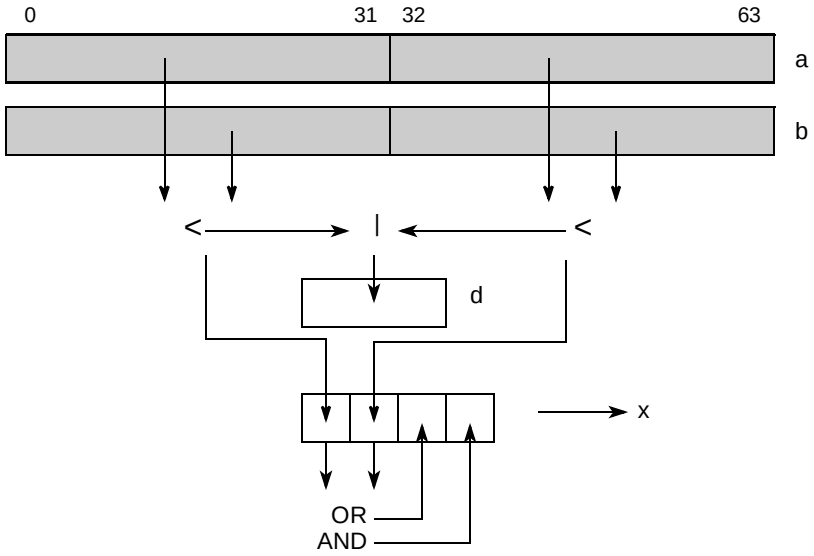

Figure 3-27. Vector Any Floating-Point Less Than (__ev_any_fs_lt)

d	a	b	Maps to
_Bool	__ev64_opaque	__ev64_opaque	**evfscmplt x,a,b**

__ev_any_fs_tst_ eq __ev_any_fs_tst_eq

Vector Any Floating-Point Test Equal

d = __ev_any_fs_tst_eq(a,b)

```
if ( (a0:31 = b0:31) | (a32:63 = b32:63)) then d ← true
else d ←false
```

This intrinsic returns true if either the upper 32 bits of parameter a are equal to the upper 32 bits of parameter b or the lower 32 bits of parameter a are equal to the lower 32 bits of parameter b. This intrinsic differs from __ev_any_fs_eq because no exceptions are taken during its execution. If strict IEEE 754 compliance is required, use __ev_any_fs_eq instead.

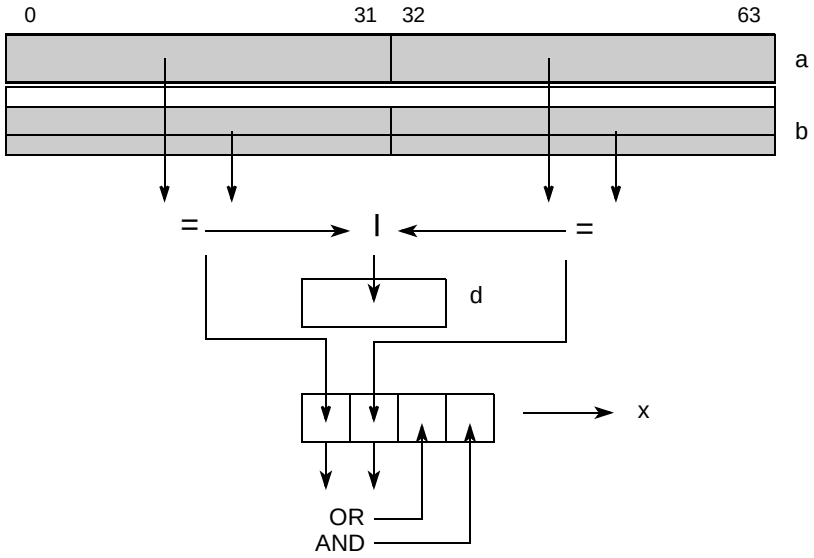

Figure 3-28. Vector Any Floating-Point Test Equal (__ev_any_fs_tst_eq)

d	a	b	Maps to
_Bool	__ev64_opaque	__ev64_opaque	**evfststeq x,a,b**

__ev_any_fs_tst_gt __ev_any_fs_tst_gt

Vector Any Floating-Point Test Greater Than

d = __ev_any_fs_tst_gt(a,b)

```
if ( (a0:31 > b0:31) | (a32:63 > b2:63)) then d← true
else d ←false
```

This intrinsic returns true if either the upper 32 bits of parameter a are greater than the upper 32 bits of parameter b or the lower 32 bits of parameter a are greater than the lower 32 bits of parameter b. This intrinsic differs from __ev_any_fs_gt because no exceptions are taken during its execution. If strict IEEE 754 compliance is required, use __ev_any_fs_gt instead.

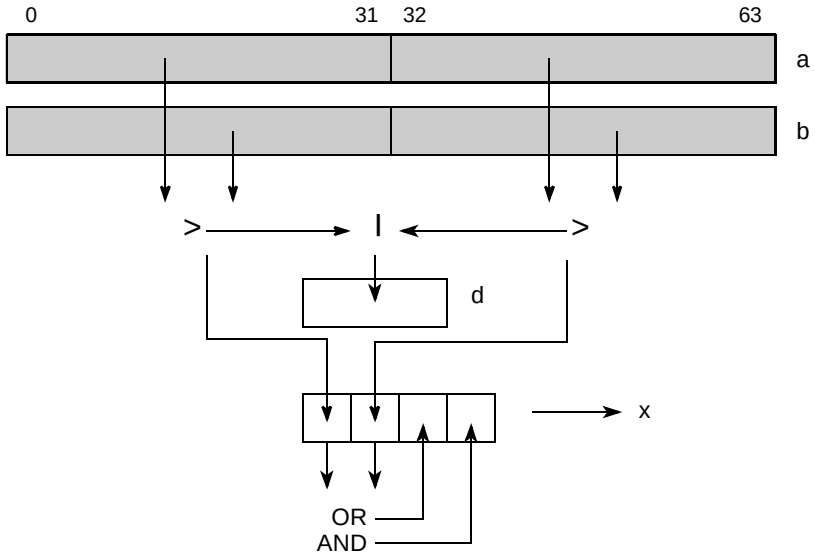

Figure 3-29. Vector Any Floating-Point Test Greater Than (__ev_any_fs_tst_gt)

d	a	b	Maps to
_Bool	__ev64_opaque	__ev64_opaque	**evfststgt x,a,b**

__ev_any_fs_tst_lt __ev_any_fs_tst_lt

Vector Any Floating-Point Test Less Than

d = __ev_any_fs_tst_lt(a,b)

```
if ( (a0:31 < b0:31) || (a32:63 < b32:63)) then d ← true
else d ←false
```

This intrinsic returns true if either the upper 32 bits of parameter a are less than the upper 32 bits of parameter b or the lower 32 bits of parameter a are less than the lower 32 bits of parameter b. This intrinsic differs from __ev_any_fs_lt because no exceptions are taken during its execution. If strict IEEE 754 compliance is required, use __ev_any_fs_lt instead.

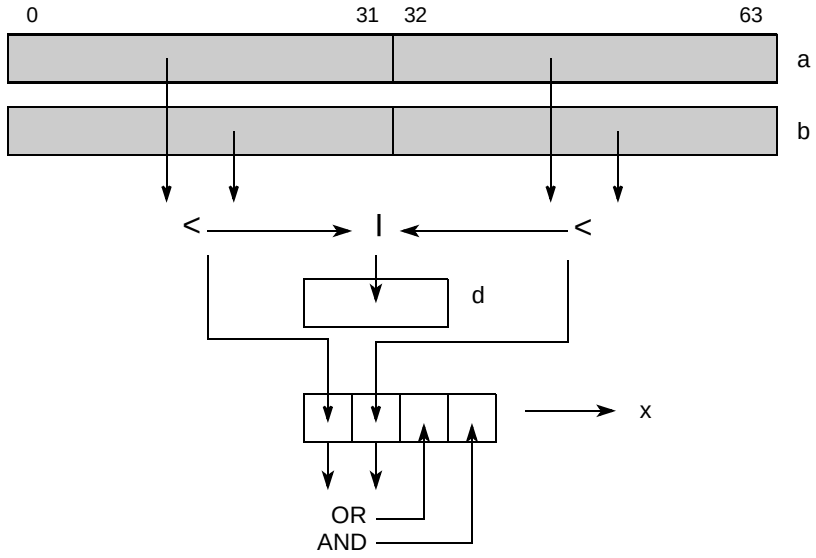

Figure 3-30. Vector Any Floating-Point Test Less Than (__ev_any_fs_tst_lt)

d	a	b	Maps to
_Bool	__ev64_opaque	__ev64_opaque	**evfststlt x,a,b**

__ev_any_gts __ev_any_gts

Vector AND with Complement

d = __ev_any_gts(a,b)

```
if ((a0:31 >signed b0:31)|(a32:63 >signed b32:63)) then d ← true
else d ← false
```

This intrinsic returns true if either the upper 32 bits of parameter a are greater than the upper 32 bits of parameter b or the lower 32 bits of parameter a are greater than the lower 32 bits of parameter b.

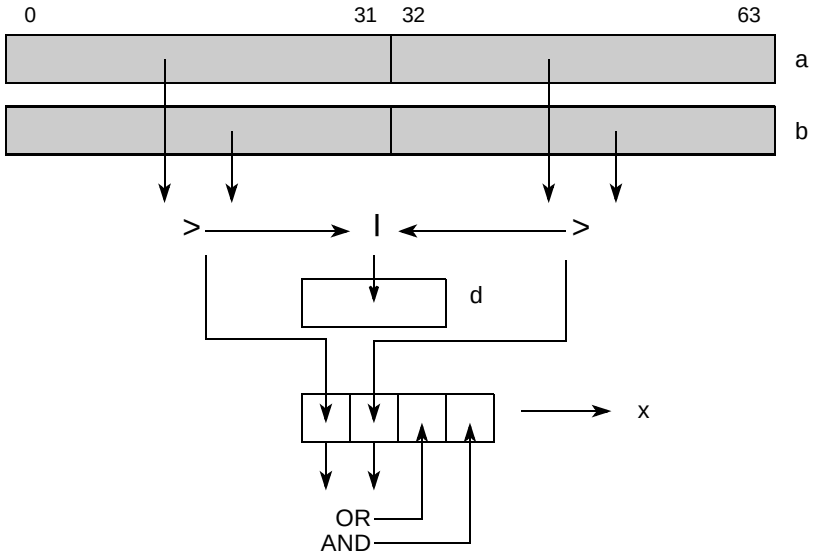

Figure 3-31. Vector Any Greater Than Signed (__ev_any_gts)

d	a	b	Maps to
_Bool	__ev64_opaque	__ev64_opaque	**evcmpgts x,a,b**

__ev_any_gtu __ev_any_gtu

Vector Any Element Greater Than Unsigned

d = __ev_any_gtu(a,b)

```
if ( (a0:31 >unsigned b0:31) | (a32:63 >unsigned b32:63)) then d ← true
else d←false
```

This intrinsic returns true if either the upper 32 bits of parameters a are greater than the upper 32 bits of parameter b or the lower 32 bits of parameter a are greater than the lower 32 bits of parameter b.

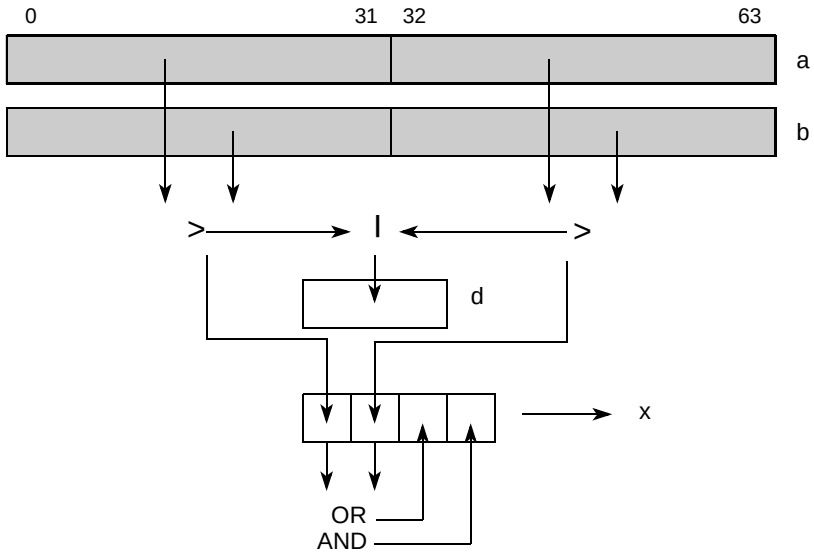

Figure 3-32. Vector Any Greater Than Unsigned (__ev_any_gtu)

d	a	b	Maps to
_Bool	__ev64_opaque	__ev64_opaque	**evcmpgtu x,a,b**

__ev_any_lts __ev_any_lts

Vector Any Element Less Than Signed

d = __ev_any_lts(a,b)

```
if ( (a0:31 <signed b0:31) | (a32:63 <signed b32:63)) then d ← true
else d ←false
```

This intrinsic returns true if either the upper 32 bits of parameter a are less than the upper 32 bits of parameter b or the lower 32 bits of parameter a are less than the lower 32 bits of parameter b.

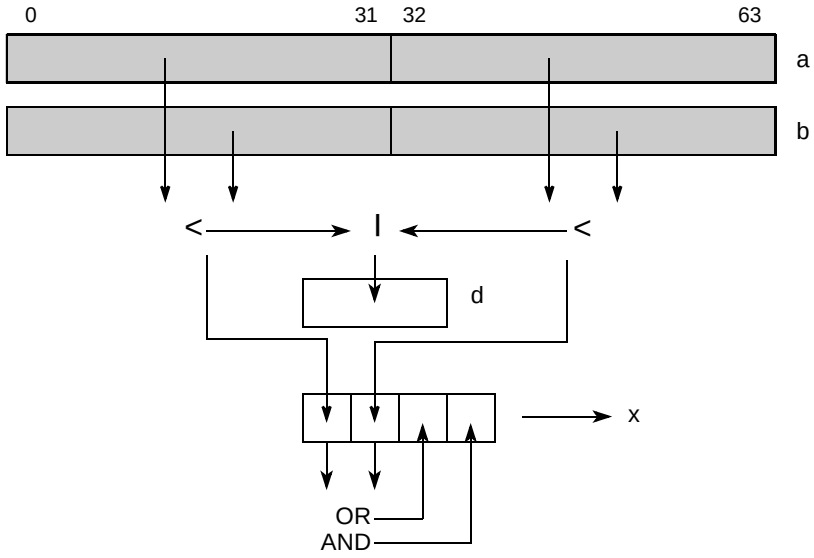

Figure 3-33. Vector Any Less Than Signed(__ev_any_lts)

d	a	b	Maps to
_Bool	__ev64_opaque	__ev64_opaque	**evcmplts x,a,b**

__ev_any_ltu __ev_any_ltu

Vector Any Element Less Than Unsigned

d = __ev_any_ltu(a,b)

```
if ( (a0:31 <unsigned b0:31) | (a32:63 <unsigned b32:63)) then d ← true
else d ←false
```

This intrinsic returns true if either the upper 32 bits of parameter a are less than the upper 32 bits of parameter b or the lower 32 bits of parameter a are less than the lower 32 bits of parameter b.

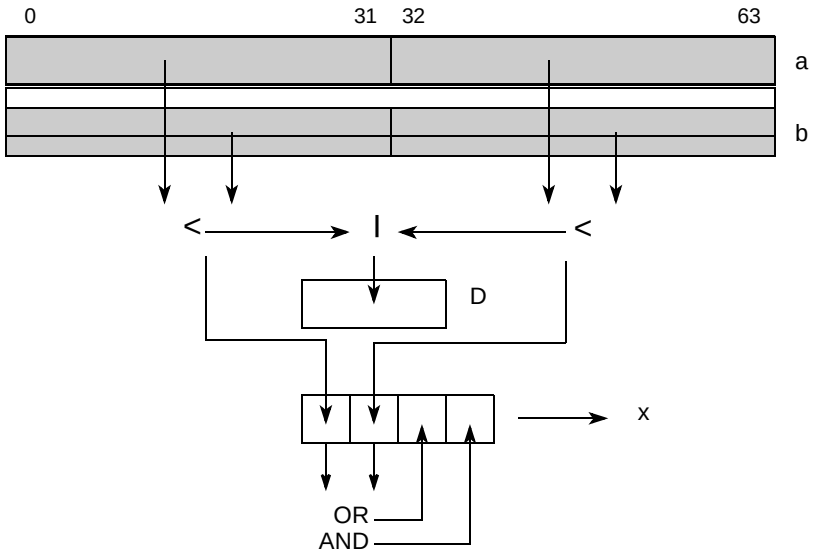

Figure 3-34. Vector Any Less Than Unsigned (__ev_any_ltu)

d	a	b	Maps to
_Bool	__ev64_opaque	__ev64_opaque	**evcmpltu x,a,b**

__ev_cntlsw __ev_cntlsw

Vector Count Leading Signed Bits Word

d = __ev_cntlsw(a)

The leading signed bits in each element of parameter a are counted, and the count is placed into each element of parameter d.

evcntlzw is used for unsigned parameters; **evcntlsw** is used for signed parameters.

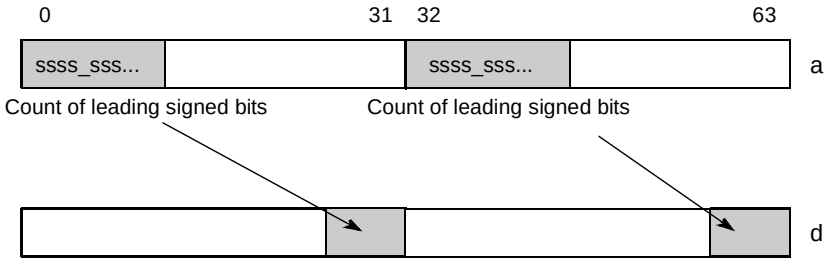

Figure 3-35. Vector Count Leading Signed Bits Word (__ev_cntlsw)

d	a	Maps to
__ev64_opaque	__ev64_opaque	**evcntlsw d,a**

__ev_cntlzw __ev_cntlzw

Vector Count Leading Zeros Word

d = __ev_cntlzw(a)

The leading zero bits in each element of parameter a are counted, and the respective count is placed into each element of parameter d.

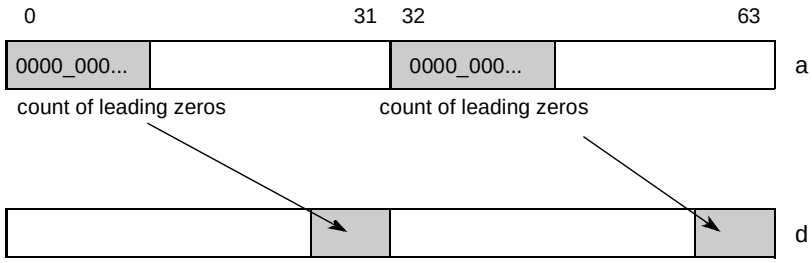

Figure 3-36. Vector Count Leading Zeros Word (__ev_cntlzw)

d	a	Maps to
__ev64_opaque	__ev64_opaque	**evcntlzw d,a**

__ev_divws __ev_divws

Vector Divide Word Signed

d = __ev_divws (a,b)

```
dividendh ← a0:31
dividendl ← a32:63
divisorh ← b0:31
divisorl ← b32:63
d0:31 ← dividendh ÷ divisorh
d32:63 ← dividendl ÷ divisorl
ovh ← 0
ovl ← 0
if ((dividendh < 0) & (divisorh = 0)) then
     d0:31 ← 0x80000000
     ovh ← 1
else if ((dividendh >= 0) & (divisorh = 0)) then
     d0:31 ← 0x7FFFFFFF
     ovh ← 1
else if ((dividendh = 0x80000000) & (divisorh = 0xFFFF_FFFF)) then
     d0:31 ← 0x7FFFFFFF
     ovh ← 1
if ((dividendl < 0) & (divisorl = 0)) then
     d32:63 ← 0x80000000
     ovl ← 1
else if ((dividendl >= 0) & (divisorl = 0)) then
     d32:63 ← 0x7FFFFFFF
     ovl ← 1
else if ((dividendl = 0x80000000) & (divisorl = 0xFFFF_FFFF)) then
     d32:63 ← 0x7FFFFFFF
     ovl ← 1
SPEFSCROVH ← ovh
SPEFSCROV ← ovl
SPEFSCRSOVH ← SPEFSCRSOVH | ovh
SPEFSCRSOV ← SPEFSCRSOV | ovl
```

The two dividends are the two elements of the contents of parameter a. The two divisors are the two elements of the contents of parameter b. The resulting two 32-bit quotients on each element are placed into parameter d. The remainders are not supplied. Parameters and quotients are interpreted as signed integers. If overflow, underflow, or divide by zero occurs, the overflow and summary overflow SPEFSCR bits are set. Note that any overflow indication is always set as a side effect of this instruction. No form is defined that disables the setting of the overflow bits. In case of overflow, a saturated value is delivered into the destination register.

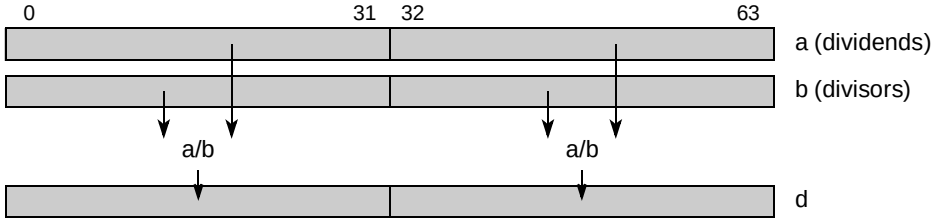

Figure 3-37. Vector Divide Word Signed (__ev_divws)

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	**evdivws d,a,b**

__ev_divwu __ev_divwu

Vector Divide Word Unsigned

d = __ev_divwu (a,b)

```
dividendh ← a0:31
dividendl ← a32:63
divisorh ← b0:31
divisorl ← b32:63
d0:31 ← dividendh ÷ divisorh
d32:63 ← dividendl ÷ divisorl
ovh ← 0
ovl ← 0
if (divisorh = 0) then
    d0:31 = 0xFFFFFFFF
    ovh ← 1
if (divisorl = 0) then
     d32:63 ← 0xFFFFFFFF
     ovl ← 1
SPEFSCROVH ← ovh
SPEFSCROV ← ovl
SPEFSCRSOVH ← SPEFSCRSOVH | ovh
SPEFSCRSOV ← SPEFSCRSOV | ovl
```

The two dividends are the two elements of the contents of parameter a. The two divisors are the two elements of the contents of parameter b. Two 32-bit quotients are formed as a result of the division on each of the high and low elements and the quotients are placed into parameter d. Remainders are not supplied. Parameters and quotients are interpreted as unsigned integers. If a divide by zero occurs, the overflow and summary overflow SPEFSCR bits are set. Note that any overflow indication is always set as a side effect of this instruction. No form is defined that disables the setting of the overflow bits. In case of overflow, a saturated value is delivered into the destination register.

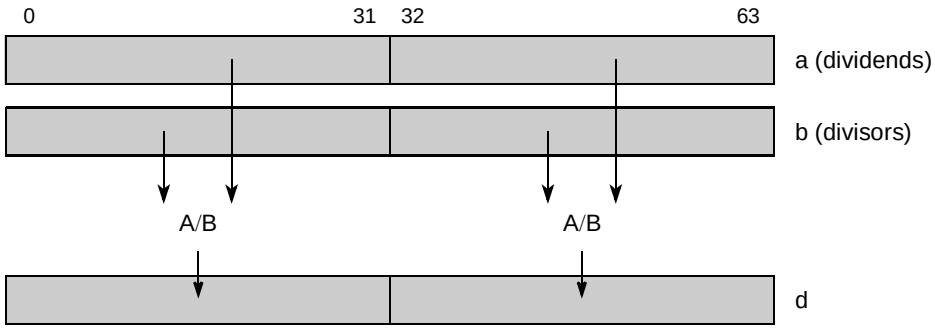

Figure 3-38. Vector Divide Word Unsigned (__ev_divwu)

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	**evdivwu d,a,b**

__ev_eqv __ev_eqv

Vector Equivalent

d = __ev_eqv (a,b)

```
d0:31 ← a0:31 ≡ b0:31 // Bitwise XNOR
d32:63 ← a32:63 ≡ b32:63 // Bitwise XNOR
```

The corresponding elements of parameters a and b are XNORed bitwise, and the results are placed in the parameter d.

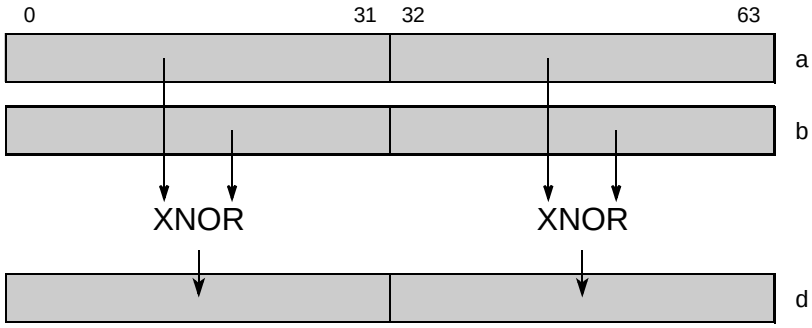

Figure 3-39. Vector Equivalent (__ev_eqv)

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	**eveqv d,a,b**

__ev_extsb

__ev_extsb

Vector Extend Sign Byte

d = __ev_extsb (a)

$d_{0:31} \leftarrow EXTS(a_{24:31})$
$d_{32:63} \leftarrow EXTS(a_{56:63})$

The signs of the byte in each of the elements in parameter a are extended, and the results are placed in the parameter d.

Figure 3-40. Vector Extend Sign Byte (__ev_extsb)

d	a	**Maps to**
__ev64_opaque	__ev64_opaque	**evextsb d,a**

__ev_extsh __ev_extsh

Vector Extend Sign Half Word

d = __ev_extsh (a)

$d_{0:31} \leftarrow EXTS(a_{16:31})$
$d_{32:63} \leftarrow EXTS(a_{48:63})$

The signs of the half words in each of the elements in parameter a are extended, and the results are placed into parameter d.

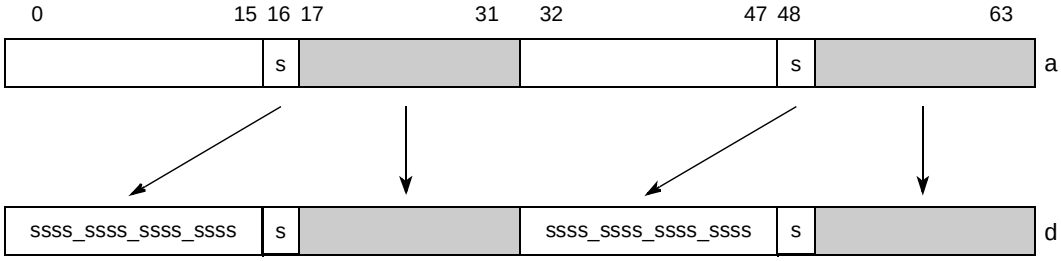

Figure 3-41. Vector Extend Sign Half Word (__ev_extsh)

d	a	Maps to
__ev64_opaque	__ev64_opaque	**evextsh d,a**

__ev_fsabs __ev_fsabs

Vector Floating-Point Absolute Value

d = __ev_fsabs (a)

$d_{0:31} \leftarrow 0b0 \ || \ a_{1:31}$
$d_{32:63} \leftarrow 0b0 \ || \ a_{33:63}$

The signed bits of each element of parameter a are cleared, and the result is placed into parameter d. No exceptions are taken during the execution of this instruction.

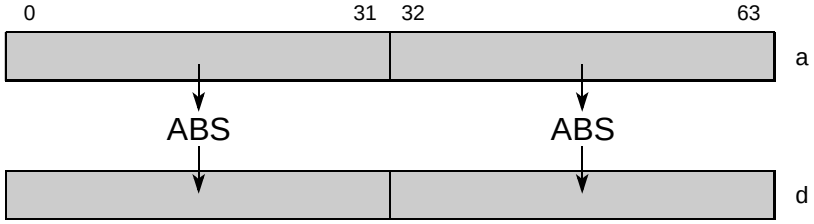

Figure 3-42. Vector Floating-Point Absolute Value (__ev_fsabs)

d	a	Maps to
__ev64_opaque	__ev64_opaque	**evtsabs d,a**

__ev_fsadd __ev_fsadd

Vector Floating-Point Add

d = __ev_fsadd (a,b)

$$d_{0:31} \leftarrow a_{0:31} +_{sp} b_{0:31}$$
$$d_{32:63} \leftarrow a_{32:63} +_{sp} b_{32:63}$$

The single-precision floating-point value of each element of parameter a is added to the corresponding element in parameter b, and the results are placed in parameter d.

If an overflow condition is detected or the contents of parameters a or b are NaN or Infinity, the result is an appropriately signed maximum floating-point value.

If an underflow condition is detected, the result is an appropriately signed floating-point 0.

The following status bits are set in the SPEFSCR:

- FINV, FINVH if the contents of **r**A or **r**B are +inf, -inf, Denorm, or NaN
- FOFV, FOFVH if an overflow occurs
- FUNF, FUNFH if an underflow occurs
- FINXS, FG, FGH, FX, FXH if the result is inexact or overflow occurred and overflow exceptions are disabled

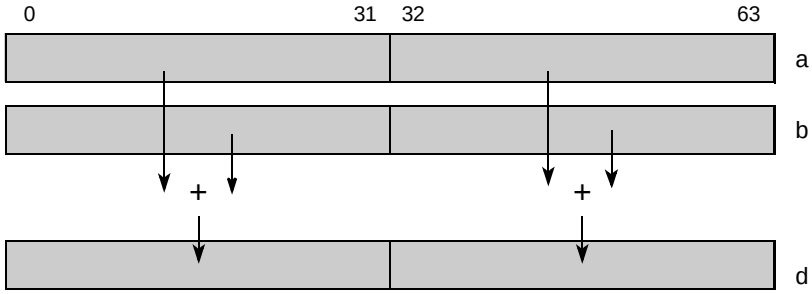

Figure 3-43. Vector Floating-Point Add (__ev_fsadd)

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	**evfsadd d,a,b**

__ev_fscfsf __ev_fscfsf

Vector Convert Floating-Point from Signed Fraction

d = __ev_fscfsf (a)

$d_{0:31} \leftarrow$ CnvtI32ToFP32Sat($a_{0:31}$, SIGN, UPPER, F)
$d_{32:63} \leftarrow$ CnvtI32ToFP32Sat($a_{32:63}$, SIGN, LOWER, F)

The signed fractional values in each element of parameter a are converted to the nearest single-precision floating-point value using the current rounding mode and placed in parameter d.

The following status bits are set in the SPEFSCR:

- FINXS, FG, FGH, FX, FXH if the result is inexact

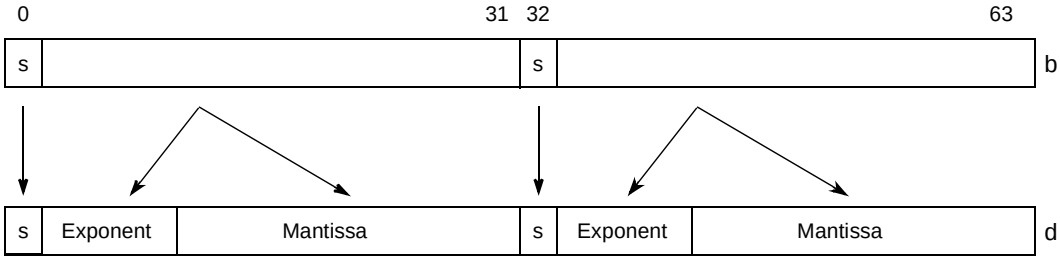

Figure 3-44. Vector Convert Floating-Point from Signed Fraction (__ev_fscfsf)

d	a	Maps to
__ev64_opaque	__ev64_opaque	**evtscfsf d,a**

__ev_fscfsi __ev_fscfsi

Vector Convert Floating-Point from Signed Integer

d = __ev_fscfsi (a)

```
d0:31 ← CnvtSI32ToFP32Sat(a0:31, SIGN, UPPER, I)
d32:63 ← CnvtSI32ToFP32Sat(a32:63, SIGN, LOWER, I)
```

The signed integer values in each element in parameter a are converted to the nearest single-precision floating-point value using the current rounding mode and placed in parameter d.

The following status bits are set in the SPEFSCR:

- FINXS, FG, FGH, FX, FXH if the result is inexact

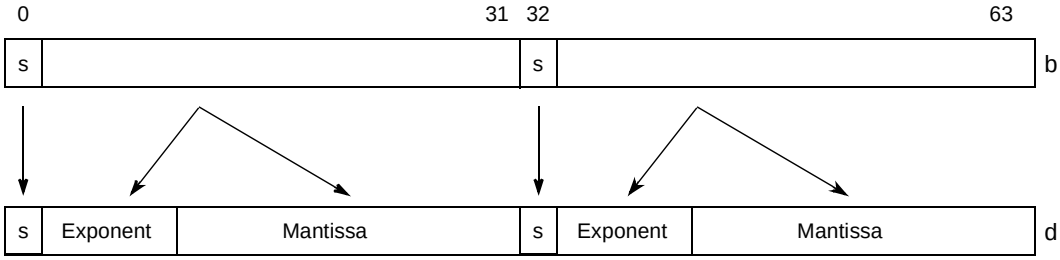

Figure 3-45. Vector Convert Floating-Point from Signed Integer (__ev_fscfsi)

d	a	Maps to
__ev64_opaque	__ev64_opaque	**evtscfsi d,a**

__ev_fscfuf __ev_fscfuf

Vector Convert Floating-Point from Unsigned Fraction

d = __ev_fscfuf (a)

$d_{0:31} \leftarrow$ CnvtI32ToFP32Sat($a_{0:31}$, UNSIGN, UPPER, F)
$d_{32:63} \leftarrow$ CnvtI32ToFP32Sat($a_{32:63}$, UNSIGN, LOWER, F)

The unsigned fractional values in each element of parameter a are converted to the nearest single-precision floating-point value using the current rounding mode and placed in parameter d.

The following status bits are set in the SPEFSCR:

- FINXS, FG, FX if the result is inexact

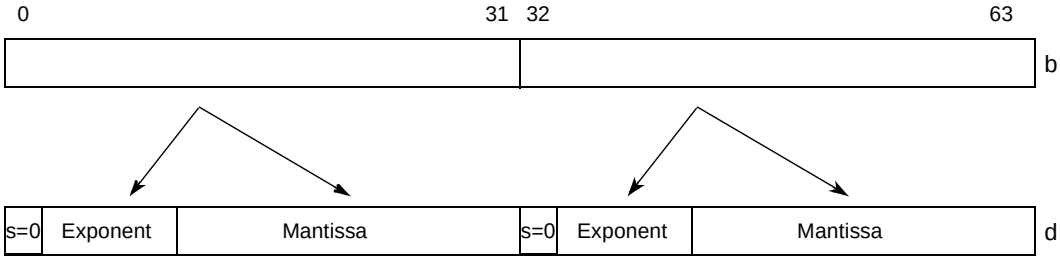

Figure 3-46. Vector Convert Floating-Point from Unsigned Fraction (__ev_fscfuf)

d	a	Maps to
__ev64_opaque	__ev64_opaque	**evtscfuf d,a**

__ev_fscfui __ev_fscfui

Vector Convert Floating-Point from Unsigned Integer

d = __ev_fscfui (a)

```
d0:31 ← CnvtI32ToFP32Sat(a031, UNSIGN, UPPER, I)
d32:63 ← CnvtI32ToFP32Sat(a32:63, UNSIGN, LOWER, I)
```

The unsigned integer value in each element of parameter a are converted to the nearest single-precision floating-point value using the current rounding mode and placed in parameter d.

The following status bits are set in the SPEFSCR:

- FINXS, FG, FGH, FX, FXH if the result is inexact

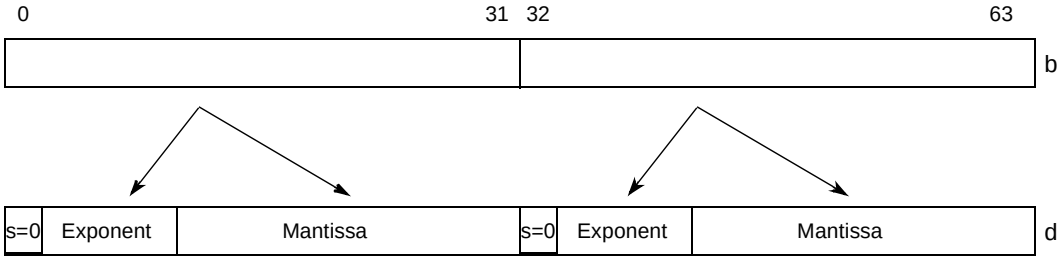

Figure 3-47. Vector Convert Floating-Point from Unsigned Integer (__ev_fscfui)

d	a	Maps to
__ev64_opaque	__ev64_opaque	**evtscfui d,a**

__ev_fsctsf __ev_fsctsf

Vector Convert Floating-Point to Signed Fraction

d = __ev_fsctsf (a)

```
d0:31 ← CnvtFP32ToISat(a0:31, SIGN, UPPER, ROUND, F)
d32:63 ← CnvtFP32ToISat(a32:63, SIGN, LOWER, ROUND, F)
```

The single-precision floating-point value in each element of parameter a is converted to a signed fraction using the current rounding mode and the results are placed in parameter d. The result saturates if it cannot be represented in a 32-bit fraction. NaNs are converted to 0.

The following status bits are set in the SPEFSCR:

- FINV, FINVH if the contents of parameter a are +inf, -inf, Denorm, or NaN or parameter a cannot be represented in the target format
- FINXS, FG, FGH, FX, FXH if the result is inexact

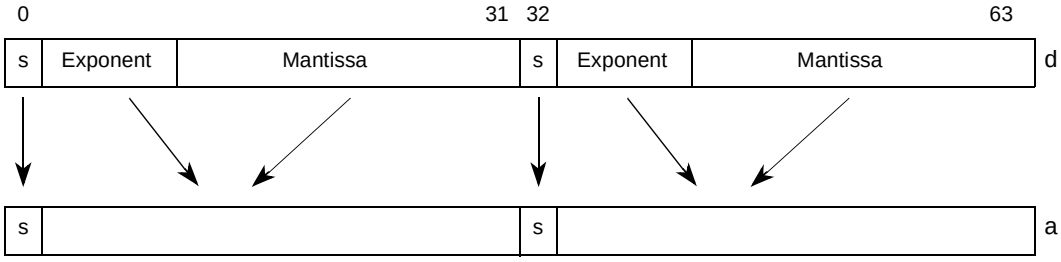

Figure 3-48. Vector Convert Floating-Point to Signed Fraction (__ev_x)

d	a	Maps to
__ev64_opaque	__ev64_opaque	**evtsctsf d,a**

__ev_fsctsi __ev_fsctsi

Vector Convert Floating-Point to Signed Integer

d = __ev_fsctsi (a)

```
d0:31 ← CnvtFP32ToISat(a0:31, SIGN, UPPER, ROUND, I)
d32:63 ← CnvtFP32ToISat(a32:63, SIGN, LOWER, ROUND, I)
```

The single-precision floating-point value in each element of parameter a is converted to a signed integer using the current rounding mode, and the results are placed in parameter d. The result saturates if it cannot be represented in a 32-bit integer. NaNs are converted to 0.

The following status bits are set in the SPEFSCR:

- FINV, FINVH if the contents of parameter a are +inf, -inf, Denorm or NaN or parameter a cannot be represented in the target format
- FINXS, FG, FGH, FX, FXH if the result is inexact

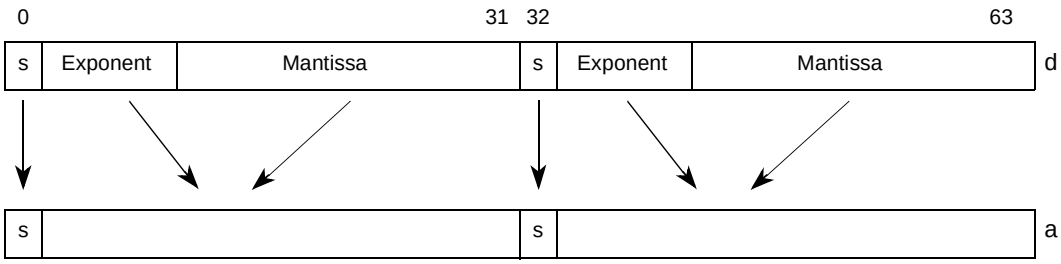

Figure 3-49. Vector Convert Floating-Point to Signed Integer (__ev_fsctsi)

d	a	Maps to
__ev64_opaque	__ev64_opaque	**evtsctsi d,a**

__ev_fsctsiz __ev_fsctsiz

Vector Convert Floating-Point to Signed Integer with Round Toward Zero

d = __ev_fsctsiz (a)

```
d0:31 ← CnvtFP32ToISat(a0:31, SIGN, UPPER, TRUNC, I)
d32:63 ← CnvtFP32ToISat(a32:63, SIGN, LOWER, TRUNC, I)
```

The single-precision floating-point value in each element of parameter a is converted to a signed integer using the rounding mode Round Towards Zero, and the results are placed in parameter d. The result saturates if it cannot be represented in a 32-bit integer. NaNs are converted to 0.

The following status bits are set in the SPEFSCR:

- FINV, FINVH if the contents of parameter a are +inf, -inf, Denorm, or NaN or if parameter a cannot be represented in the target format
- FINXS, FG, FGH, FX, FXH if the result is inexact

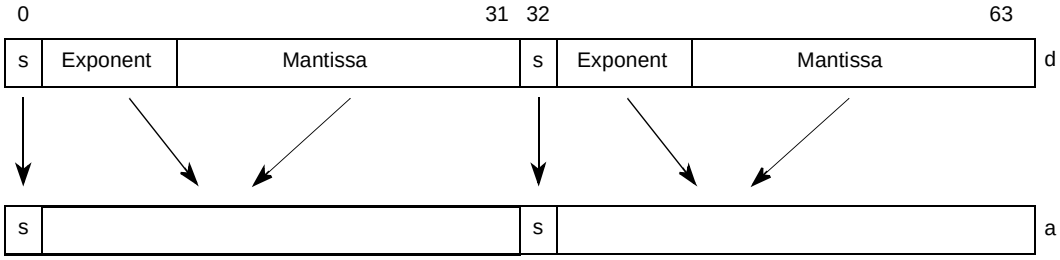

Figure 3-50. Vector Convert Floating-Point to Signed Integer with Round Toward Zero (__ev_fsctsiz)

d	a	Maps to
__ev64_opaque	__ev64_opaque	**evtsctsiz d,a**

__ev_fsctuf __ev_fsctuf

Vector Convert Floating-Point to Unsigned Fraction

d = __ev_fsctuf (a)

```
d0:31 ← CnvtFP32ToISat(a0:31, UNSIGN, UPPER, ROUND, F)
d32:63 ← CnvtFP32ToISat(a32:63, UNSIGN, LOWER, ROUND, F)
```

The single-precision floating-point value in each element of parameter a is converted to an unsigned fraction using the current rounding mode, and the results are placed in parameter d. The result saturates if it cannot be represented in a 32-bit unsigned fraction. NaNs are converted to 0.

The following status bits are set in the SPEFSCR:

- FINV, FINVH if the contents of parameter a are +inf, -inf, Denorm, or NaN or if parameter a cannot be represented in the target format
- FINXS, FG, FGH, FX, FXH if the result is inexact

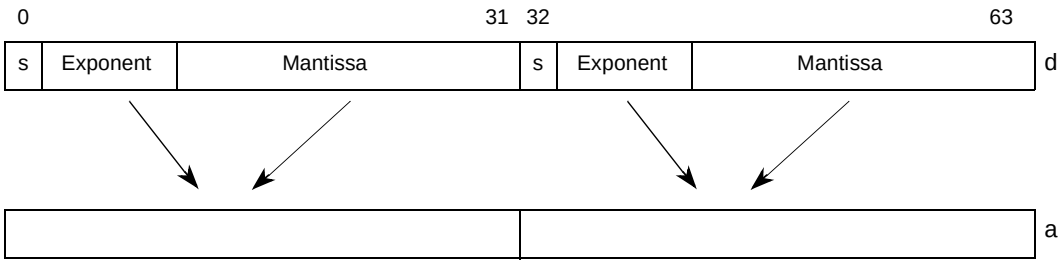

Figure 3-51. Vector Convert Floating-Point to Unsigned Fraction (__ev_fsctuf)

d	a	Maps to
__ev64_opaque	__ev64_opaque	**evtsctuf d,a**

__ev_fsctui __ev_fsctui

Vector Convert Floating-Point to Unsigned Integer

d = __ev_fsctui (a)

$d_{0:31} \leftarrow$ CnvtFP32ToISat($a_{0:31}$, UNSIGN, UPPER, ROUND, I)
$d_{32:63} \leftarrow$ CnvtFP32ToISat($a_{32:63}$, UNSIGN, LOWER, ROUND, I)

The single-precision floating-point value in each element of parameter a is converted to an unsigned integer using the current rounding mode, and the results are placed in parameter d. The result saturates if it cannot be represented in a 32-bit unsigned integer. NaNs are converted to 0.

The following status bits are set in the SPEFSCR:

- FINV, FINVH if the contents of parameter a are +inf, -inf, Denorm or NaN or parameter a cannot be represented in the target format
- FINXS, FG, FGH, FX, FXH if the result is inexact

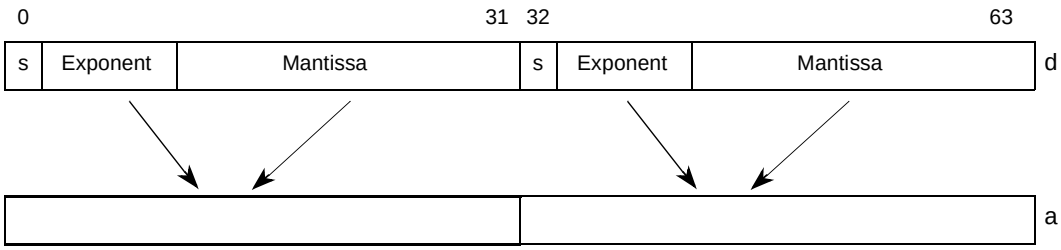

Figure 3-52. Vector Convert Floating-Point to Unsigned Integer (__ev_fsctui)

d	a	Maps to
__ev64_opaque	__ev64_opaque	**evtsctui d,a**

__ev_fsctuiz __ev_fsctuiz

Vector Convert Floating-Point to Unsigned Integer with Round toward Zero

d = __ev_fsctuiz (a)

```
d0:31 ← CnvtFP32ToISat(a0:31, UNSIGN, UPPER, TRUNC, I)
d32:63 ← CnvtFP32ToISat(a32:63, UNSIGN, LOWER, TRUNC, I)
```

The single-precision floating-point value in each element of parameter a is converted to an unsigned integer using the rounding mode Round towards Zero, and the results are placed in parameter d. The result saturates if it cannot be represented in a 32-bit unsigned integer. NaNs are converted to 0.

The following status bits are set in the SPEFSCR:

- FINV, FINVH if the contents of parameter a are +inf, -inf, Denorm, or NaN or parameter a cannot be represented in the target format
- FINXS, FG, FGH, FX, FXH if the result is inexact

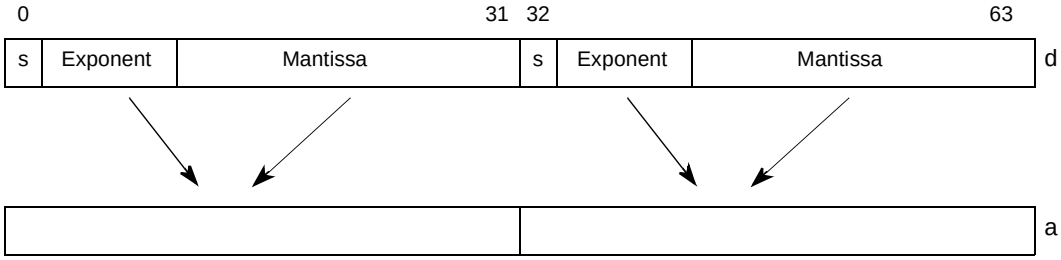

Figure 3-53. Vector Convert Floating-Point to Unsigned Integer with Round Toward Zero (__ev_fsctuiz)

d	a	Maps to
__ev64_opaque	__ev64_opaque	**evtsctuiz d,a**

__ev_fsdiv __ev_fsdiv

Vector Floating-Point Divide

d = __ev_fsdiv(a,b)

$$d_{0:31} \leftarrow a_{0:31} \div_{sp} b_{0:31}$$
$$d_{32:63} \leftarrow a_{32:63} \div_{sp} d_{32:63}$$

The single-precision floating-point value in each element of parameter a is divided by the corresponding elements in parameter b, and the results are placed in parameter d.

If an overflow is detected, parameter b is a Denorm (or 0 value), or parameter a is a NaN or Infinity and parameter b is a normalized number, the result is an appropriately signed maximum floating-point value.

If an underflow is detected or parameter b is a NaN or Infinity, the result is an appropriately signed floating-point 0.

The following status bits are set in the SPEFSCR:

- FINV, FINVH if the contents of parameter a or b are +inf, -inf, Denorm, or NaN
- FOFV, FOFVH if an overflow occurs
- FUNV, FUNVH if an underflow occurs
- FDBZS, FDBZ, FDBZH if a divide by zero occurs
- FINXS, FG, FGH, FX, FXH if the result is inexact or overflow occurred and overflow exceptions are disabled

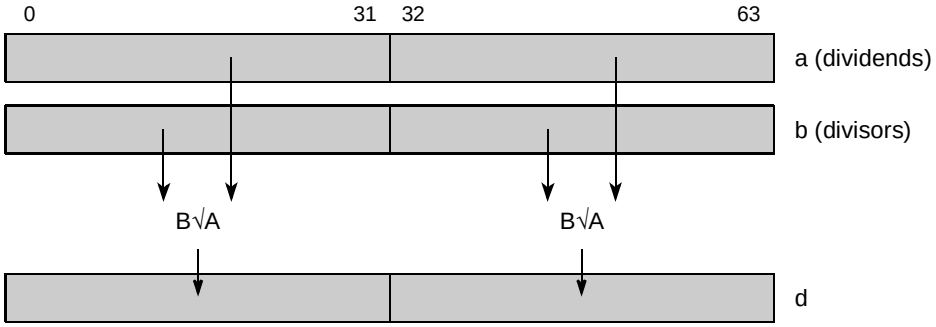

Figure 3-54. Vector Floating-Point Divide (__ev_fsdiv)

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	**evfsdiv d,a,b**

__ev_fsmul __ev_fsmul

Vector Floating-Point Multiply

d = __ev_fsmul (a,b)

$$d_{0:31} \leftarrow a_{0:31} \times_{sp} b_{0:31}$$
$$d_{32:63} \leftarrow a_{32:63} \times_{sp} b_{32:63}$$

Each single-precision floating-point element of parameter a is multiplied with the corresponding element of parameter b, and the result is stored in parameter d. If an overflow is likely, pmax or nmax is stored in parameter d. If an underflow is likely, +0, or –0 is stored in parameter d. The following condition defines when an overflow is likely and the corresponding result for each element of the vector:

```
ei = (ea - 127) + (eb - 127) + 127
if (sa = sb) then
    if (ei ≥ 127) then r = pmax
    else if (ei < -126) then r = +0
else
    if (ei ≥ 127) then r = nmax
    else if (ei < -126) then r = -0
```

- If the contents of parameter a or b are +inf, –inf, Denorm, QNaN, or SNaN, at least one of the SPEFSCR[FINVH] or SPEFSCR[FINV] bits is set.
- If an overflow occurs or is likely, at least one of the SPEFSCR[FOVFH] or SPEFSCR[FOVF] bits is set.
- If an underflow occurs or is likely, at least one of the SPEFSCR[FUNFH] or SPEFSCR[FUNF] bits is set.
- If the exception is enabled for the high or low element in which the error occurs, the exception is taken.

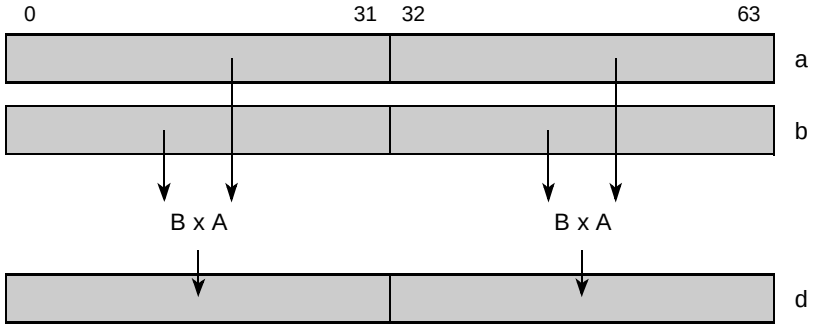

Figure 3-55. Vector Floating-Point Multiply (__ev_fsmul)

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	**evfsmul d,a,b**

__ev_fsnabs __ev_fsnabs

Vector Floating-Point Negative Absolute Value

d = __ev_fsnabs (a)

$d_{0:31} \leftarrow 0b1 \;||\; a_{1:31}$
$d_{32:63} \leftarrow 0b1 \;||\; a_{33:63}$

The signed bits of each element of parameter a are all set and the result is placed into parameter d. No exceptions are taken during the execution of this instruction.

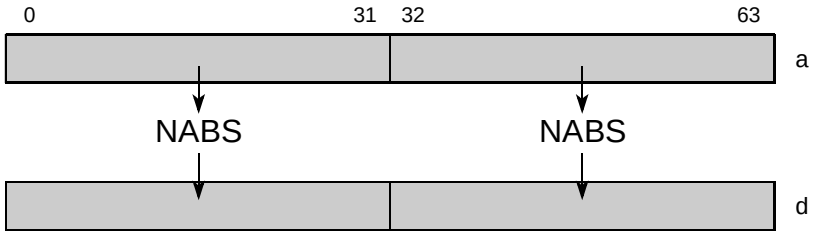

Figure 3-56. Vector Floating-Point Negative Absolute Value (__ev_fsnabs)

d	a	Maps to
__ev64_opaque	__ev64_opaque	**evtsnabs d,a**

__ev_fsneg __ev_fsneg

Vector Floating-Point Negate

d = __ev_fsneg (a)

$$d_{0:31} \leftarrow \neg a_0 \;||\; a_{1:31}$$
$$d_{32:63} \leftarrow \neg a_{32} \;||\; a_{33:63}$$

The signed bits of each element of parameter a are complemented and the result is placed into parameter d. No exceptions are taken during the execution of this instruction.

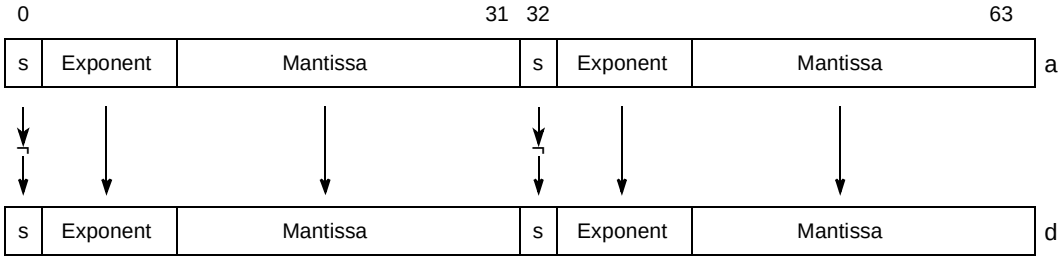

Figure 3-57. Vector Floating-Point Negate (__ev_fsneg)

d	a	Maps to
__ev64_opaque	__ev64_opaque	**evtsneg d,a**

__ev_fssub __ev_fssub

Vector Floating-Point Subtract

d = __ev_fssub (a,b)

$$d_{0:31} \leftarrow a_{0:31} -_{sp} b_{0:31}$$
$$d_{32:63} \leftarrow a_{32:63} -_{sp} b_{32:63}$$

Each single-precision floating-point element of parameter b is subtracted from the corresponding element of parameter a and the result is stored in parameter d. If an overflow is likely, pmax or nmax is stored in parameter d. If an underflow is likely, +0 or –0 is stored in parameter d. The following condition defines how boundary cases of inputs (+inf, –inf, Denorm, QNaN, SNaN) are treated, when an overflow is likely, and the corresponding result for each element of the vector:

```
if ((sa = 0) & (sb = 1)) then
    if (max(ea, eb) ≥ 127) then r = pmax
else if ((sa = 1) & (sb = 0)) then
    if (max(ea, eb) ≥ 127) then r = nmax
else if (sa = sb) then
                    // Boundary case to be defined later
```

- If the contents of parameter a or b are +inf, –inf, Denorm, QNaN, or SNaN, at least one of the SPEFSCR[FINVH] or SPEFSCR[FINV] bits is set.
- If an overflow occurs or is likely, the SPEFSCR[FOVFH] or SPEFSCR[FOVF] bits is set.
- If an underflow occurs or is likely, at least one of the SPEFSCR[FUNFH] or SPEFSCR[FUNF] bits is set.
- If the exception is enabled for the high or low element in which the error occurs, the exception is taken.

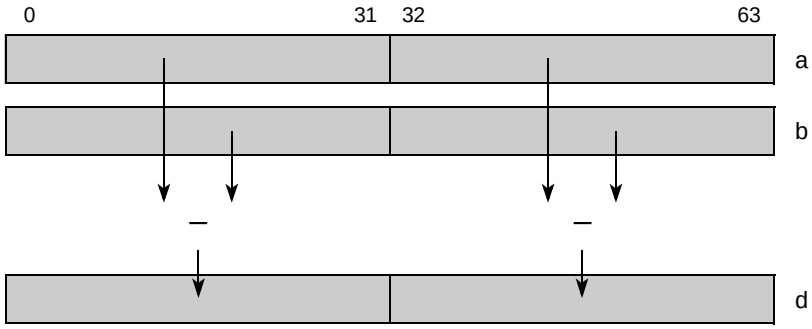

Figure 3-58. Vector Floating-Point Subtract (__ev_fssub)

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	**evfssub d,a,b**

__ev_ldd __ev_ldd

Vector Load Double Word into Double Word

d = __ev_ldd (a,b)

```
if (a = 0) then temp ← 0
else temp ← (a)
EA ← temp + EXTZ(UIMM*8)
d ← MEM(EA, 8)
```

The double word addressed by EA is loaded from memory and placed in parameter d.

Figure 3-59 shows how bytes are loaded into parameter d as determined by the endian mode.

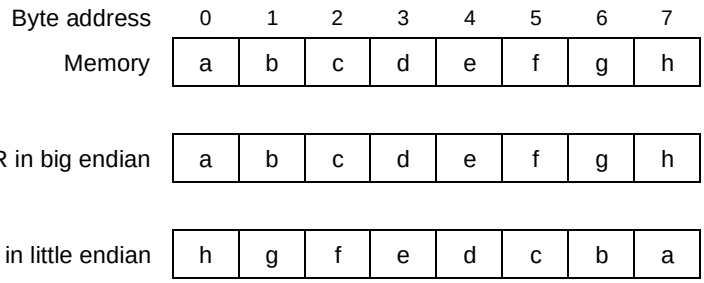

Figure 3-59. __ev_ldd Results in Big- and Little-Endian Modes

NOTE

During implementation, an alignment exception occurs if the effective address (EA) is not double-word aligned.

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	5-bit unsigned	**evldd d,a,b**

__ev_lddx __ev_lddx

Vector Load Double Word into Double Word Indexed

d = __ev_lddx **(a,b)**

```
if (a = 0) then temp ← 0
else temp ← (a)
EA ← temp + (b)
d ← MEM(EA, 8)
```

The double word addressed by EA is loaded from memory and placed in parameter d.

Figure 3-60 shows how bytes are loaded into parameter d as determined by the endian mode.

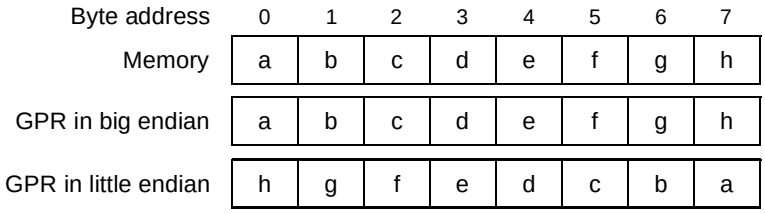

Figure 3-60. __ev_lddx Results in Big- and Little-Endian Modes

NOTE

During implementation, an alignment exception occurs if the EA is not double-word aligned.

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	int32_t	**evlddx d,a,b**

__ev_ldh __ev_ldh

Vector Load Double into Four Half Words

d = __ev_ldh (a,b)

```
if (a = 0) then temp ← 0
else temp ← (a)
EA ← temp + EXTZ(UIMM*8)
d0:15 ← MEM(EA, 2)
d16:31 ← MEM(EA+2,2)
d32:47 ← MEM(EA+4,2)
d48:63 ← MEM(EA+6,4)
```

The double word addressed by EA is loaded from memory and placed in parameter d.

Figure 3-61 shows how bytes are loaded into parameter d as determined by the endian mode.

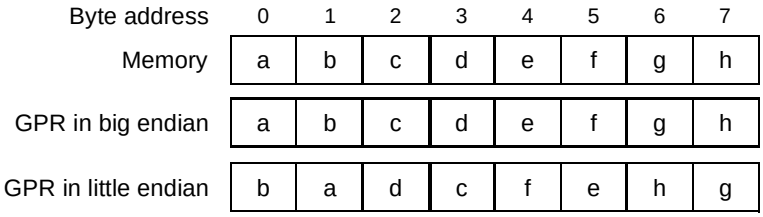

Figure 3-61. __ev_ldh Results in Big- and Little-Endian Modes

NOTE

During implementation, an alignment exception occurs if the EA is not double-word aligned.

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	5-bit unsigned	**evldh d,a,b**

__ev_ldhx __ev_ldhx

Vector Load Double into Four Half Words Indexed

d = __ev_ldhx (a,b)

```
if (a = 0) then temp ← 0
else temp ← (a)
EA ← temp + (b)
d0:15 ← MEM(EA, 2)
d16:31 ← MEM(EA+2,2)
d32:47 ← MEM(EA+4,2)
d48:63 ← MEM(EA+6,4)
```

The double word addressed by EA is loaded from memory and placed in parameter d.

Figure 3-62 shows how bytes are loaded into parameter d as determined by the endian mode.

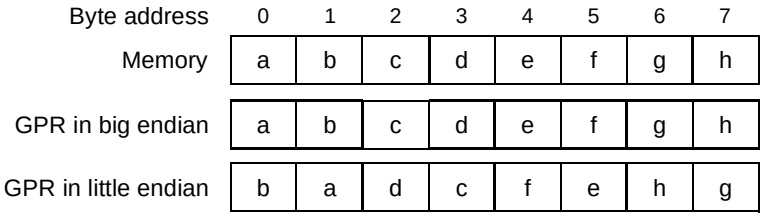

Figure 3-62. __ev_ldhx Results in Big- and Little-Endian Modes

NOTE

During implementation, an alignment exception occurs if the EA is not double-word aligned.

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	int32_t	**evldhx d,a,b**

__ev_ldw __ev_ldw

Vector Load Double into Two Words

d = __ev_ldw (a,b)

```
if (a = 0) then temp ← 0
else temp ← (a)
EA ← temp + EXTZ(UIMM*8)
d0:31 ← MEM(EA, 4)
d32:63 ← MEM(EA+4, 4)
```

The double word addressed by EA is loaded from memory and placed in parameter d.

Figure 3-63 shows how bytes are loaded into parameter d as determined by the endian mode.

Byte address	0	1	2	3	4	5	6	7
Memory	a	b	c	d	e	f	g	h
GPR in big endian	a	b	c	d	e	f	g	h
GPR in little endian	d	c	b	a	h	g	f	e

Figure 3-63. __ev_ldw Results in Big- and Little-Endian Modes

NOTE

During implementation, an alignment exception occurs if the EA is not double-word aligned.

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	5-bit unsigned	**evldw d,a,b**

__ev_ldwx __ev_ldwx

Vector Load Double into Two Words Indexed

d = __ev_ldwx (a,b)

```
if (a = 0) then temp ← 0
else temp ← (a)
EA ← temp + (b)
d0:31 ← MEM(EA, 4)
d32:63 ← MEM(EA+4, 4)
```

The double word addressed by EA is loaded from memory and placed in parameter d.

Figure 3-64 shows how bytes are loaded into parameter d as determined by the endian mode.

Byte address	0	1	2	3	4	5	6	7
Memory	a	b	c	d	e	f	g	h
GPR in big endian	a	b	c	d	e	f	g	h
GPR in little endian	d	c	b	a	h	g	f	e

Figure 3-64. __ev_ldwx Results in Big- and Little-Endian Modes

NOTE

During implementation, an alignment exception occurs if the EA is not double-word aligned.

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	int32_t	**evldwx d,a,b**

__ev_lhhesplat __ev_lhhesplat

Vector Load Half Word into Half Words Even and Splat

d = __ev_lhhesplat **(a,b)**

```
if (a = 0) then temp ← 0
else temp ← (a)
EA ← temp + EXTZ(UIMM*2)
d0:15 ← MEM(EA,2)
d16:31 ← 0x0000
d32:47 ← MEM(EA,2)
d48:63 ← 0x0000
```

The half word addressed by EA is loaded from memory and placed in the even half words of each element of parameter d.

Figure 3-65 shows how bytes are loaded into parameter d as determined by the endian mode.

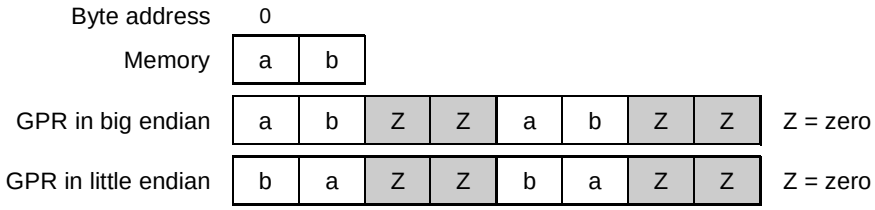

Figure 3-65. __ev_lhhesplat Results in Big- and Little-Endian Modes

NOTE

During implementation, an alignment exception occurs if the EA is not half word-aligned.

d	a	b	Maps to
__ev64_opaque	uint16_t	5-bit unsigned	**evlhhesplat d,a,b**

__ev_lhhesplatx __ev_lhhesplatx

Vector Load Half Word into Half Words Even and Splat-Indexed

d = __ev_lhhesplatx (a,b)

```
if (a = 0) then temp ← 0
else temp ← (a)
EA ← temp + (b)
d0:15 ← MEM(EA,2)
d16:31 ← 0x0000
d32:47 ← MEM(EA,2)
d48:63 ← 0x0000
```

The half word addressed by EA is loaded from memory and placed in the even half words of each element of parameter d.

Figure 3-66 shows how bytes are loaded into parameter d as determined by the endian mode.

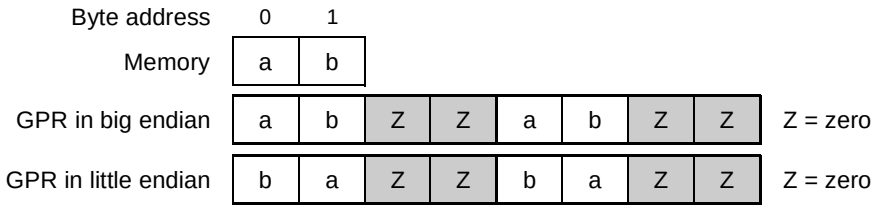

Figure 3-66. __ev_lhhesplatx Results in Big- and Little-Endian Modes

NOTE

During implementation, an alignment exception occurs if the EA is not half word-aligned.

d	a	b	Maps to
__ev64_opaque	uint16_t	int32_t	**evlhhesplatx d,a,b**

__ev_lhhossplat __ev_lhhossplat

Vector Load Half Word into Half Word Odd Signed and Splat

d = __ev_lhhossplat (a,b)

```
if (a = 0) then temp ← 0
else temp ← (a)
EA ← temp + EXTZ(UIMM*2)
d0:31 ← EXTS(MEM(EA,2))
d32:63 ← EXTS(MEM(EA,2))
```

The half word addressed by EA is loaded from memory and placed in the odd half words sign extended in each element of parameter d.

Figure 3-67 shows how bytes are loaded into parameter d as determined by the endian mode.

- In big-endian mode, the msb of parameter a is sign-extended.
- In little-endian mode, the msb of parameter b is sign-extended.

NOTE

During implementation, an alignment exception occurs if the EA is not half word-aligned.

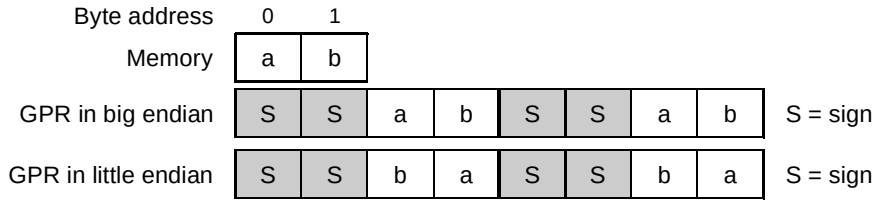

Figure 3-67. __ev_lhhossplat Results in Big- and Little-Endian Modes

d	a	b	Maps to
__ev64_opaque	uint16_t	5-bit unsigned	**evlhhosplat d,a,b**

__ev_lhhossplatx __ev_lhhossplatx

Vector Load Half Word into Half Word Odd Signed and Splat-Indexed

d = __ev_lhhossplatx (a,b)

```
if (a = 0) then temp ← 0
else temp ← (a)
EA ← temp + (b)
d0:31 ← EXTS(MEM(EA,2))
d32:63 ← EXTS(MEM(EA,2))
```

The half-word addressed by EA is loaded from memory and placed in the odd half-words sign extended in each element of parameter d.

Figure 3-68 shows how bytes are loaded into parameter d as determined by the endian mode.

- In big-endian mode, the msb of parameter a is sign-extended.
- In little-endian mode, the msb of parameter b is sign-extended.

NOTE

During implementation, an alignment exception occurs if the EA is not half word-aligned.

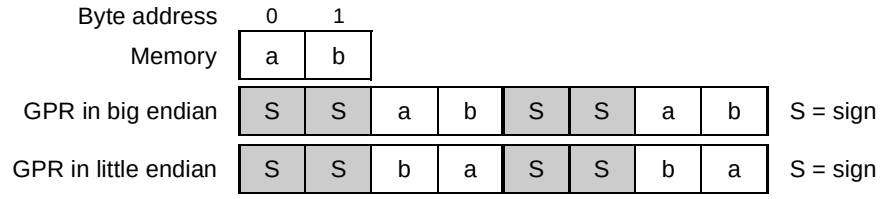

Figure 3-68. __ev_lhhossplatx Results in Big- and Little-Endian Modes

d	a	b	Maps to
__ev64_opaque	uint16_t	int32_t	**evlhhosplatx d,a,b**

__ev_lhhousplat __ev_lhhousplat

Vector Load Half Word into Half Word Odd Unsigned and Splat

d = __ev_lhhousplat (a,b)

```
if (a = 0) then temp ← 0
else temp ← (a)
EA ← temp + EXTZ(UIMM*2)
d0:15 ← 0x0000
d16:31 ← MEM(EA,2)
d32:47 ← 0x0000
d48:63 ← MEM(EA,2)
```

The half word addressed by EA is loaded from memory and placed in the odd half words zero extended in each element of parameter d.

Figure 3-69 shows how bytes are loaded into parameter d as determined by the endian mode.

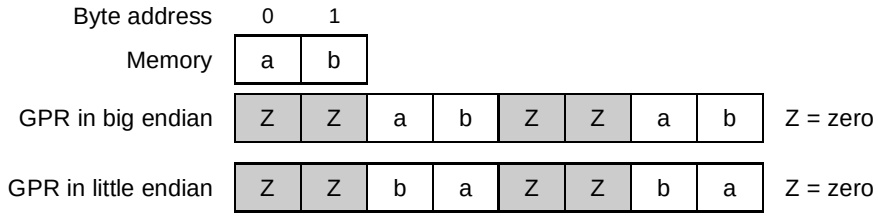

Figure 3-69. __ev_lhhousplat Results in Big- and Little-Endian Modes

NOTE

During implementation, an alignment exception occurs if the EA is not half word-aligned.

d	a	b	Maps to
__ev64_opaque	uint16_t	5-bit unsigned	**evlhhousplat d,a,b**

__ev_lhhousplatx __ev_lhhousplatx

Vector Load Half Word into Half Word Odd Unsigned and Splat-Indexed

d = __ev_lhhousplatx (a,b)

```
if (a = 0) then temp ← 0
else temp ← (a)
EA ← temp + (b)
d0:15 ← 0x0000
d16:31 ← MEM(EA,2)
d32:47 ← 0x0000
d48:63 ← MEM(EA,2)
```

The half-word addressed by EA is loaded from memory and placed in the odd half words zero extended in each element of parameter d.

Figure 3-70 shows how bytes are loaded into parameter d as determined by the endian mode.

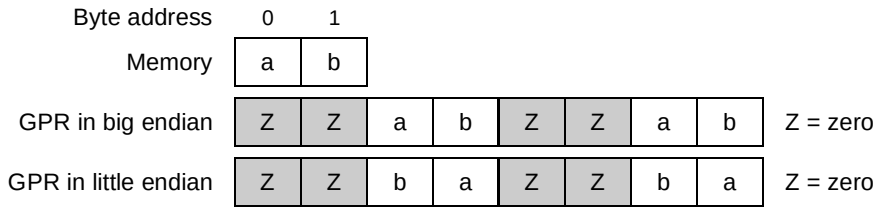

Figure 3-70. __ev_lhhousplatx Results in Big- and Little-Endian Modes

NOTE

During implementation, an alignment exception occurs if the EA is not half word-aligned.

d	a	b	Maps to
__ev64_opaque	uint16_t	int32_t	**evlhhousplatx d,a,b**

__ev_lower_eq __ev_lower_eq

Vector Lower Bits Equal

d = __ev_lower_eq(a,b)

```
if (a32:63 = b32:63) then d ← true
else d ←false
```

This intrinsic returns true if the lower 32 bits of parameter a are equal to the lower 32 bits of parameter b.

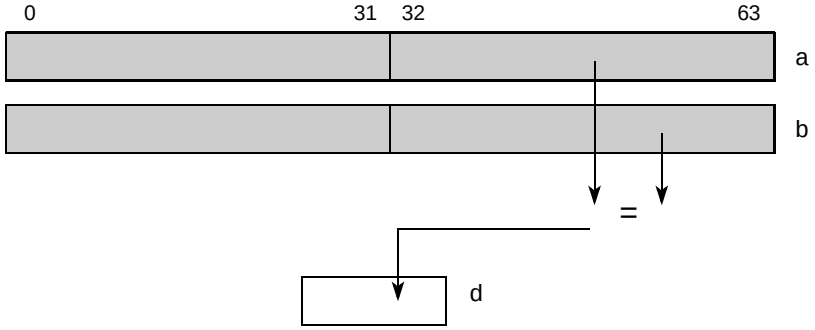

Figure 3-71. Vector Lower Equal (__ev_lower_eq)

d	a	b	Maps to
_Bool	__ev64_opaque	__ev64_opaque	**evcmpeq x,a,b**

__ev_lower_fs_eq __ev_lower_fs_eq

Vector Lower Bits Floating-Point Equal

d = __ev_lower_fs_eq(a,b)

```
if (a32:63 = b32:63) then d ← true
else d ←false
```

This intrinsic returns true if the lower 32 bits of parameter a are equal to the lower 32 bits of parameter b.

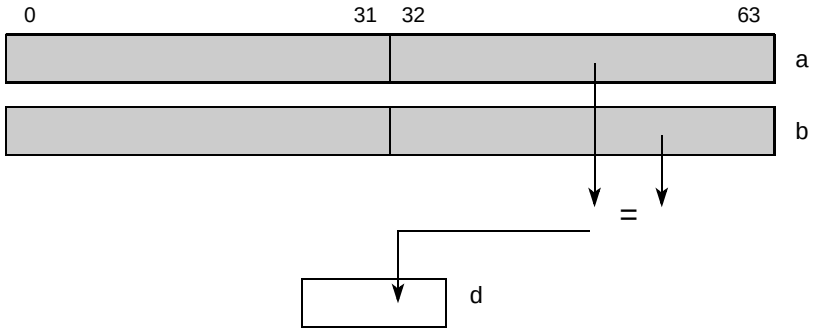

Figure 3-72. Vector Lower Floating-Point Equal (__ev_lower_fs_eq)

d	a	b	Maps to
_Bool	__ev64_opaque	__ev64_opaque	**evfscmpeq x,a,b**

__ev_lower_fs_gt __ev_lower_fs_gt

Vector Lower Bits Floating-Point Greater Than

d = __ev_lower_fs_gt(a,b)

```
if (a32:63 > b32:63) then d ← true
else d ←false
```

This intrinsic returns true if the lower 32 bits of parameter a are greater than the lower 32 bits of parameter b.

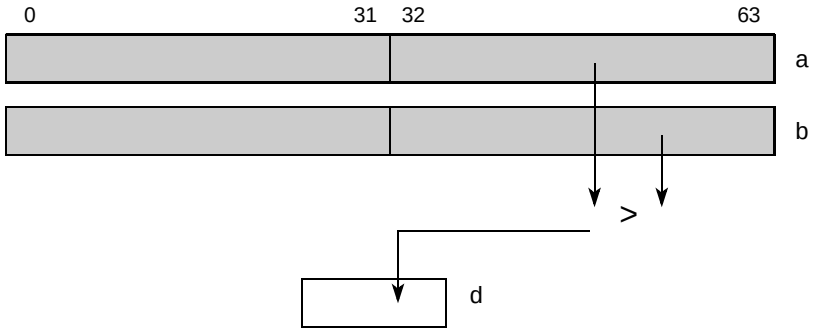

Figure 3-73. Vector Lower Floating-Point Greater Than (__ev_lower_fs_gt)

d	a	b	Maps to
_Bool	__ev64_opaque	__ev64_opaque	**evfscmpgt x,a,b**

__ev_lower_fs_lt _ev_lower_fs_lt

Vector Lower Bits Floating-Point Less Than

d = __ev_lower_fs_lt(a,b)

```
if (a32:63 < b32:63) then d← true
else d←false
```

This intrinsic returns true if the lower 32 bits of parameter a are less than the lower 32 bits of parameter b.

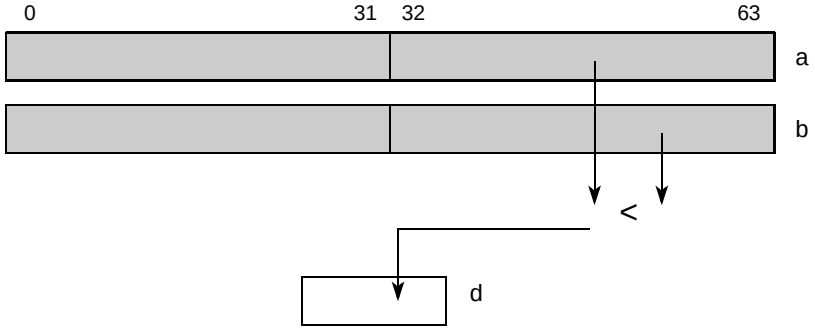

Figure 3-74. Vector Lower Floating-Point Less Than (__ev_lower_fs_lt)

d	a	b	Maps to
_Bool	__ev64_opaque	__ev64_opaque	**evfscmplt x,a,b**

__ev_lower_fs_tst_eq __ev_lower_fs_tst_eq

Vector Lower Bits Floating-Point TestEqual

d = __ev_lower_fs_tst_eq(a,b)

```
if (a32:63 = b32:63) then d ← true
else d ←false
```

This intrinsic returns true if the lower 32 bits of parameter a are equal to the lower 32 bits of parameter b. This intrinsic differs from __ev_lower_fs_eq because no exceptions are taken during its execution. If strict IEEE 754 compliance is required, use __ev_lower_fs_eq instead.

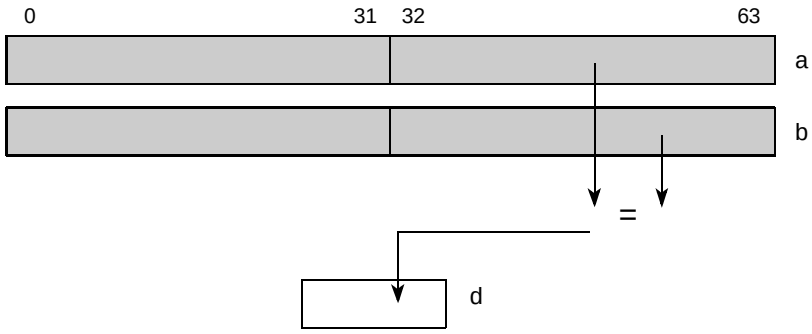

Figure 3-75. Vector Lower Floating-Point Test Equal (__ev_lower_fs_tst_eq)

d	a	b	Maps to
_Bool	__ev64_opaque	__ev64_opaque	**evfststeq x,a,b**

__ev_lower_fs_tst_gt __ev_lower_fs_tst_gt

Vector Lower Bits Floating-Point Test Greater Than

d = __ev_lower_fs_tst_gt(a,b)

```
if (a32:63 > b32:63) then d ← true
else d ←false
```

This intrinsic returns true if the lower 32 bits of parameter a are greater than the lower 32 bits of parameter b. This intrinsic differs from __ev_lower_fs_gt because no exceptions are taken during its execution. If strict IEEE 754 compliance is required, use __ev_lower_fs_gt instead.

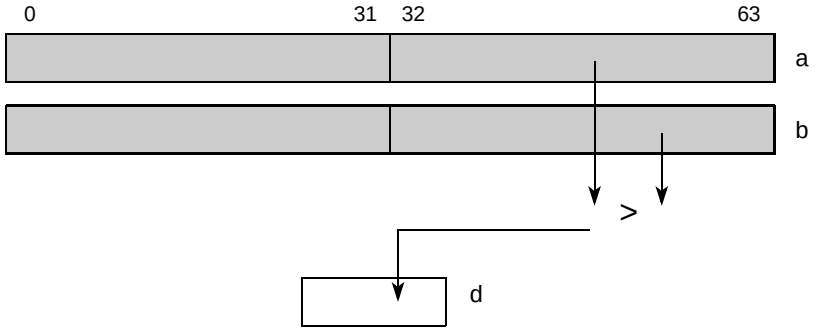

Figure 3-76. Vector Lower Floating-Point Test Greater Than (__ev_lower_fs_tst_gt)

d	a	b	Maps to
_Bool	__ev64_opaque	__ev64_opaque	**evfststgt x,a,b**

__ev_lower_fs_tst_lt __ev_lower_fs_tst_lt

Vector Lower Bits Floating-Point Test Less Than

d = __ev_lower_fs_tst_lt(a,b)

```
if (a32:63 < b32:63) then d ← true
else d ←false
```

This intrinsic returns true if the lower 32 bits of parameter a are less than the lower 32 bits of parameter b. This intrinsic differs from __ev_lower_fs_lt because no exceptions are taken during its execution. If strict IEEE 754 compliance is required, use __ev_lower_fs_lt instead.

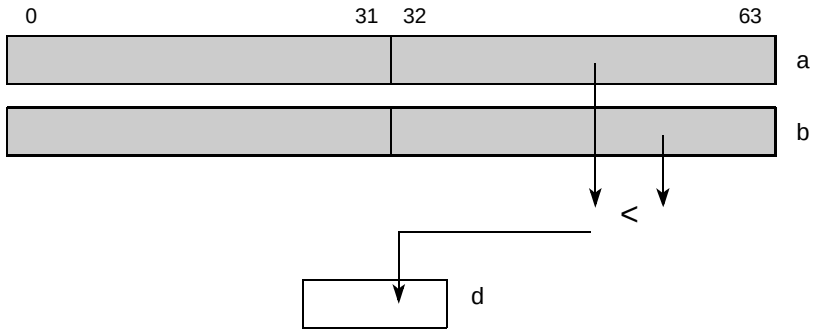

Figure 3-77. Vector Lower Floating-Point Test Less Than (__ev_lower_fs_tst_lt)

d	a	b	Maps to
_Bool	__ev64_opaque	__ev64_opaque	**evfststlt x,a,b**

__ev_lower__gts __ev_lower__gts

Vector Lower Bits Greater Than Signed

d = __ev_lower_gts(a,b)

```
if (a32:63 >signed b32:63) then d ← true
else d ←false
```

This intrinsic returns true if the lower 32 bits of parameter a are greater than the lower 32 bits of parameter b.

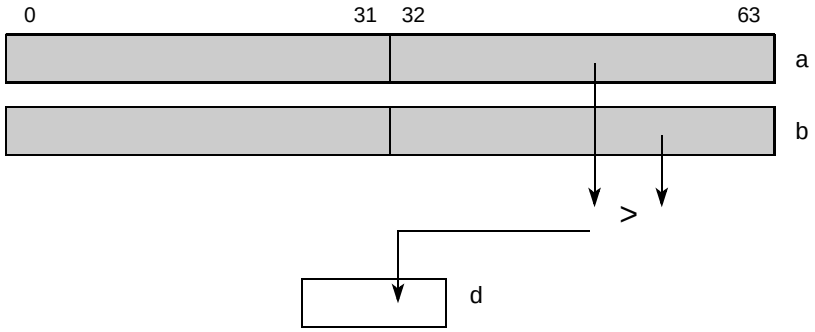

Figure 3-78. Vector Lower Greater Than Signed (__ev_lower_gts)

d	a	b	Maps to
Bool	__ev64_opaque	__ev64_opaque	**evcmpgts x,a,b**

__ev_lower_gtu __ev_lower_gtu

Vector Lower Bits Greater Than Unsigned

d = __ev_lower_gtu(a,b)

```
if (a32:63 > unsigned b32:63) then d ← true
else d ←false
```

This intrinsic returns true if the lower 32 bits of parameter a are greater than the lower 32 bits of parameter b.

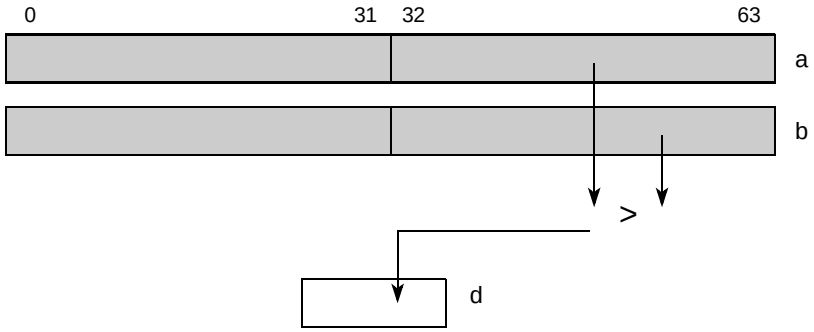

Figure 3-79. Vector Lower Greater Than Unsigned (__ev_lower_gtu)

d	a	b	Maps to
_Bool	__ev64_opaque	__ev64_opaque	**evcmpgtu x,a,b**

__ev_lower_lts __ev_lower_lts

Vector Lower Bits Less Than Signed

d = __ev_lower_lts(a,b)

```
if (a32:63 <signed b32:63) then d ← true
else d ←false
```

This intrinsic returns true if the lower 32 bits of parameter a are less than the lower 32 bits of parameter b.

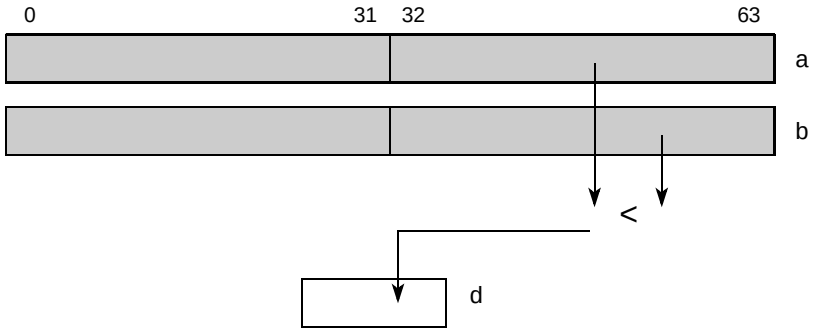

Figure 3-80. Vector Lower Less Than Signed (__ev_lower_lts)

d	a	b	Maps to
_Bool	__ev64_opaque	__ev64_opaque	**evcmplts x,a,b**

__ev_lower_ltu __ev_lower_ltu

Vector Lower Bits Less Than Unsigned

d = __ev_lower_ltu(a,b)

```
if (a32:63 <unsigned b32:63) then d ← true
else d ←false
```

This intrinsic returns true if the lower 32 bits of parameter a are less than the lower 32 bits of parameter b.

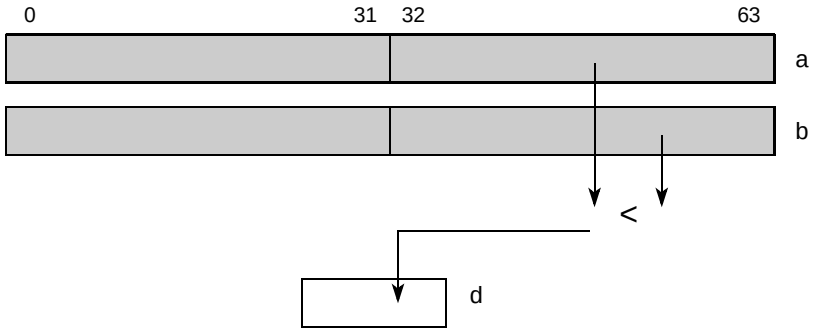

Figure 3-81. Vector Lower Less Than Unsigned (__ev_lower_ltu)

d	a	b	Maps to
_Bool	__ev64_opaque	__ev64_opaque	**evcmpltu x,a,b**

__ev_lwhe __ev_lwhe

Vector Load Word into Two Half Words Even

d = __ev_lwhe (a,b)

```
if (a = 0) then temp ← 0
else temp ← (a)
EA ← temp + EXTZ(UIMM*4)
d0:15 ← MEM(EA,2)
d16:31 ← 0x0000
d32:47 ← MEM(EA+2,2)
d48:63 ← 0x0000
```

The word addressed by EA is loaded from memory and placed in the even half words in each element of parameter d.

Figure 3-82 shows how bytes are loaded into parameter d as determined by the endian mode.

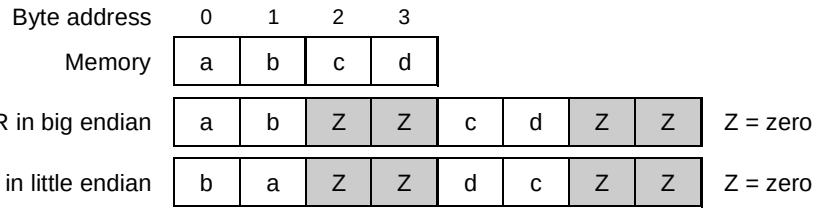

Figure 3-82. __ev_lwhe Results in Big- and Little-Endian Modes

NOTE

During implementation, an alignment exception occurs if the EA is not word-aligned.

d	a	b	Maps to
__ev64_opaque	uint32_t	5-bit unsigned	**evlwhe d,a,b**

__ev_lwhex __ev_lwhex

Vector Load Word into Two Half Words Even Indexed

d = __ev_lwhex (a,b)

```
if (a = 0) then temp ← 0
else temp ← (a)
EA ← temp + (b)
d0:15 ← MEM(EA,2)
d16:31 ← 0x0000
d32:47 ← MEM(EA+2,2)
d48:63 ← 0x0000
```

The word addressed by EA is loaded from memory and placed in the even half words in each element of parameter d.

Figure 3-83 shows how bytes are loaded into parameter d as determined by the endian mode.

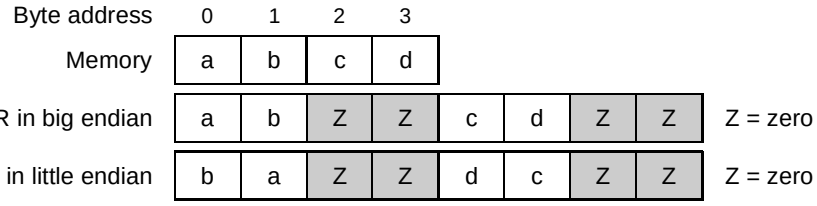

Figure 3-83. __ev_lwhex Results in Big- and Little-Endian Modes

NOTE

During implementation, an alignment exception occurs if the EA is not word-aligned.

d	a	b	Maps to
__ev64_opaque	uint32_t	int32_t	**evlwhex d,a,b**

__ev_lwhos __ev_lwhos

Vector Load Word into Two Half Words Odd Signed (with sign extension)

d = __ev_lwhos (a,b)

```
if (a = 0) then temp ← 0
else temp ← (a)
EA ← temp + EXTZ(UIMM*4)
d0:31 ← EXTS(MEM(EA,2))
d32:63 ← EXTS(MEM(EA+2,2))
```

The word addressed by EA is loaded from memory and placed in the odd half words sign extended in each element of parameter d.

Figure 3-84 shows how bytes are loaded into parameter d as determined by the endian mode.

- In big-endian memory, the msbs of parameters a and c are sign-extended.
- In little-endian memory, the msbs of parameters b and d are sign-extended.

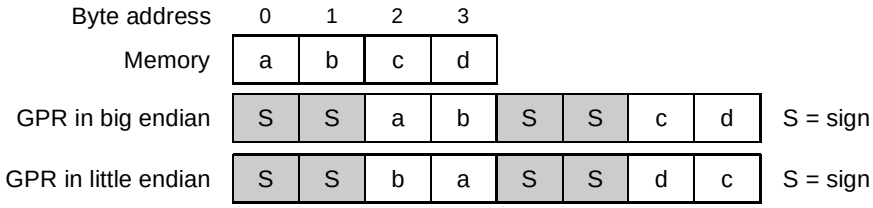

Figure 3-84. __ev_lwhos Results in Big- and Little-Endian Modes

NOTE

During implementation, an alignment exception occurs if the EA is not word-aligned.

d	a	b	Maps to
__ev64_opaque	uint32_t	5-bit unsigned	**evlwhos d,a,b**

__ev_lwhosx __ev_lwhosx

Vector Load Word into Two Half Words Odd Signed Indexed (with sign extension)

d = __ev_lwhosx (a,b)

```
if (a = 0) then temp ← 0
else temp ← (a)
EA ← temp + (b)
d0:31 ← EXTS(MEM(EA,2))
d32:63 ← EXTS(MEM(EA+2,2))
```

The word addressed by EA is loaded from memory and placed in the odd half words sign extended in each element of parameter d.

Figure 3-85 shows how bytes are loaded into parameter d as determined by the endian mode.

- In big-endian memory, the msbs of parameters a and c are sign-extended.
- In little-endian memory, the msbs of parameters b and d are sign-extended.

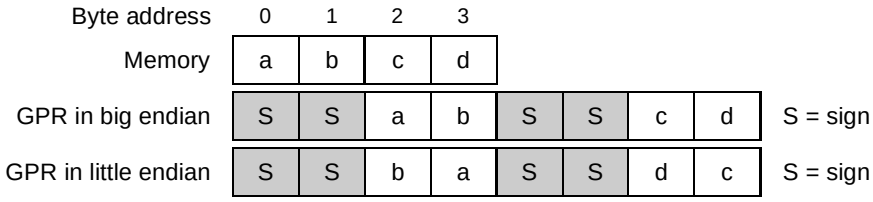

Figure 3-85. __ev_lwhosx Results in Big- and Little-Endian Modes

NOTE

During implementation, an alignment exception occurs if the EA is not word-aligned.

d	a	b	Maps to
__ev64_opaque	uint32_t	int32_t	**evlwhosx d,a,b**

__ev_lwhou __ev_lwhou

Vector Load Word into Two Half Words Odd Unsigned (zero-extended)

d = __ev_lwhou (a,b)

```
if (a = 0) then temp ← 0
else temp ← (a)
EA ← temp + EXTZ(UIMM*4)
d0:15 ← 0x0000
d16:31 ← MEM(EA,2)
d32:47 ← 0x0000
d48:63 ← MEM(EA+2,2)
```

The word addressed by EA is loaded from memory and placed in the odd half words zero extended in each element of parameter d.

Figure 3-86 shows how bytes are loaded into parameter d as determined by the endian mode.

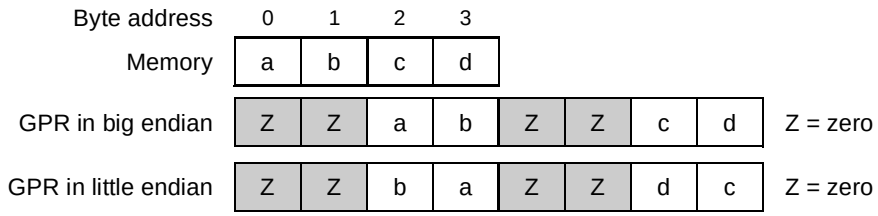

Figure 3-86. __ev_lwhou Results in Big- and Little-Endian Modes

NOTE

During implementation, an alignment exception occurs if the EA is not word-aligned.

d	a	b	Maps to
__ev64_opaque	uint32_t	5-bit unsigned	**evlwhou d,a,b**

__ev_lwhoux __ev_lwhoux

Vector Load Word into Two Half Words Odd Unsigned Indexed (zero-extended)

d = __ev_lwhoux (a,b)

```
if (a = 0) then temp ← 0
else temp ← (a)
EA ← temp + (b)
d0:15 ← 0x0000
d16:31 ← MEM(EA,2)
d32:47 ← 0x0000
d48:63 ← MEM(EA+2,2)
```

The word addressed by EA is loaded from memory and placed in the odd half words zero extended in each element of parameter d.

Figure 3-87 shows how bytes are loaded into parameter d as determined by the endian mode.

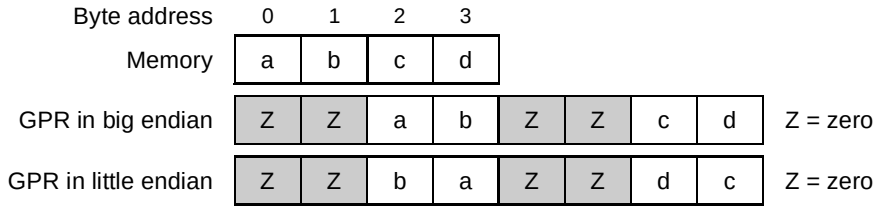

Figure 3-87. __ev_lwhoux Results in Big- and Little-Endian Modes

NOTE

During implementation, an alignment exception occurs if the EA is not word-aligned.

d	a	b	Maps to
__ev64_opaque	uint32_t	int32_t	**evlwhoux d,a,b**

__ev_lwhsplat __ev_lwhsplat

Vector Load Word into Two Half Words and Splat

d = __ev_lwhsplat (a,b)

```
if (a = 0) then temp ← 0
else temp ← (a)
EA ← temp + EXTZ(UIMM*4)
d0:15 ← MEM(EA,2)
d16:31 ← MEM(EA,2)
d32:47 ← MEM(EA+2,2)
d48:63 ← MEM(EA+2,2)
```

The word addressed by EA is loaded from memory and placed in both the even and odd half words in each element of parameter d.

Figure 3-88 shows how bytes are loaded into parameter d as determined by the endian mode.

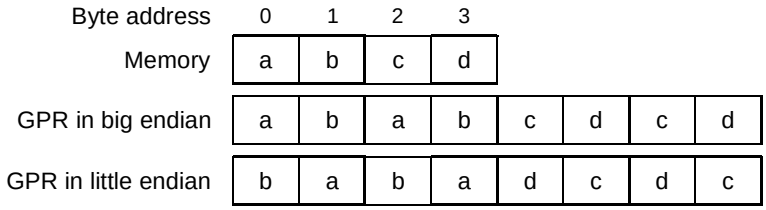

Figure 3-88. __ev_lwhsplat Results in Big- and Little-Endian Modes

NOTE

During implementation, an alignment exception occurs if the EA is not word-aligned.

d	a	b	Maps to
__ev64_opaque	uint32_t	5-bit unsigned	**evlwhsplat d,a,b**

__ev_lwhsplatx __ev_lwhsplatx

Vector Load Word into Two Half Words and Splat-Indexed

d = __ev_lwhsplatx (a,b)

```
if (a = 0) then temp ← 0
else temp ← (a)
EA ← temp + (b)
d0:15 ← MEM(EA,2)
d16:31 ← MEM(EA,2)
d32:47 ← MEM(EA+2,2)
d48:63 ← MEM(EA+2,2)
```

The word addressed by EA is loaded from memory and placed in both the even and odd half words in each element of parameter d.

Figure 3-89 shows how bytes are loaded into parameter d as determined by the endian mode.

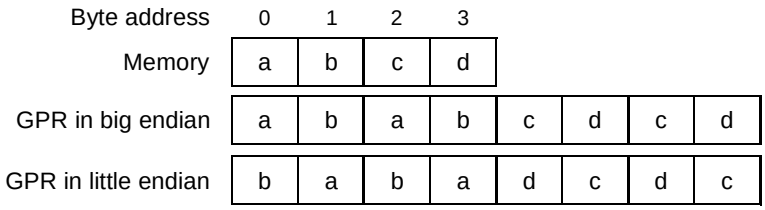

Figure 3-89. __ev_lwhsplatx Results in Big- and Little-Endian Modes

NOTE

During implementation, an alignment exception occurs if the EA is not word-aligned.

d	a	b	Maps to
__ev64_opaque	uint32_t	int32_t	**evlwhsplatx d,a,b**

__ev_lwwsplat __ev_lwwsplat

Vector Load Word into Word and Splat

d = __ev_lwwsplat (a,b)

```
if (a = 0) then temp ← 0
else temp ← (a)
EA ← temp + EXTZ(UIMM*4)
d0:31 ← MEM(EA,4)
d32:63 ← MEM(EA,4)
```

The word addressed by EA is loaded from memory and placed in both elements of parameter d.

Figure 3-90 shows how bytes are loaded into parameter d as determined by the endian mode.

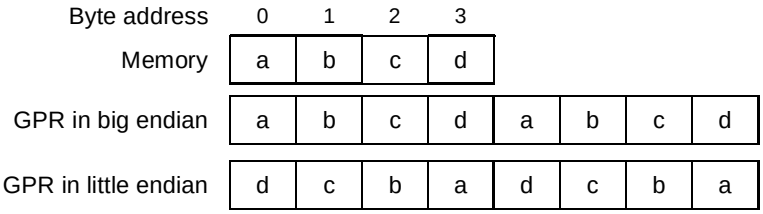

Figure 3-90. __ev_lwwsplat Results in Big- and Little-Endian Modes

NOTE

During implementation, an alignment exception occurs if the EA is not word-aligned.

d	a	b	Maps to
__ev64_opaque	uint32_t	5-bit unsigned	**evlwwsplat d,a,b**

__ev_lwwsplatx __ev_lwwsplatx

Vector Load Word into Word and Splat-Indexed

d = __ev_lwwsplatx (a,b)

```
if (a = 0) then temp ← 0
else temp ← (a)
EA ← temp + (b)
d0:31 ← MEM(EA,4)
d32:63 ← MEM(EA,4)
```

The word addressed by EA is loaded from memory and placed in both elements of parameter d.

Figure 3-91 shows how bytes are loaded into parameter d as determined by the endian mode.

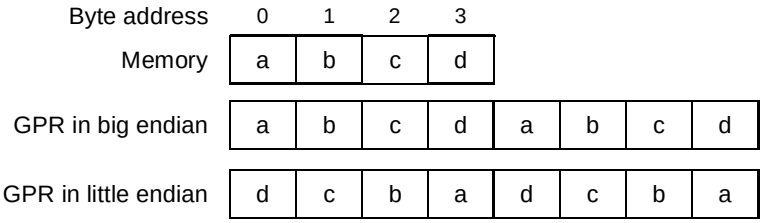

Figure 3-91. __ev_lwwsplatx Results in Big- and Little-Endian Modes

NOTE

During implementation, an alignment exception occurs if the EA is not word-aligned.

d	a	b	Maps to
__ev64_opaque	uint32_t	int32_t	**evlwwsplatx d,a,b**

__ev_mergehi __ev_mergehi

Vector Merge High

d = __ev_mergehi (a,b)

$d_{0:31} \leftarrow a_{0:31}$
$d_{32:63} \leftarrow b_{0:31}$

The high-order elements of parameters a and b are merged and placed into parameter d, as shown in Figure 3-92.

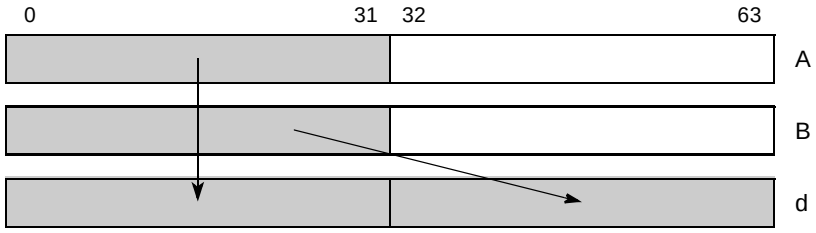

Figure 3-92. High-Order Element Merging (__ev_mergehi)

NOTE

To perform a vector splat high, specify the same register in parameters a and b.

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmergehi d,a,b**

__ev_mergehilo __ev_mergehilo

Vector Merge High/Low

d = __ev_mergehilo (a,b)

$d_{0:31} \leftarrow a_{0:31}$
$d_{32:63} \leftarrow b_{32:63}$

The high-order element of parameter a and the low-order element of parameter b are merged and placed into parameter d, as shown in Figure 3-93.

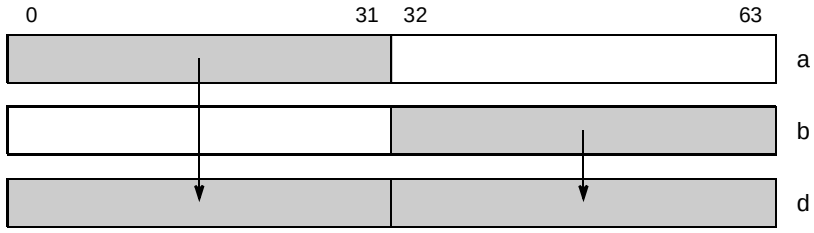

Figure 3-93. High-Order Element Merging (__ev_mergehilo)

Application note: With appropriate specification of parameter a and b, **evmergehi**, **evmergelo**, **evmergehilo**, and **evmergelohi** provide a full 32-bit permute of two source parameters.

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmergehilo d,a,b**

__ev_mergelo __ev_mergelo

Vector Merge Low

d = __ev_mergelo (a,b)

$d_{0:31} \leftarrow a_{32:63}$
$d_{32:63} \leftarrow b_{32:63}$

The low-order elements of parameters a and b are merged and placed in parameter d, as shown in Figure 3-94.

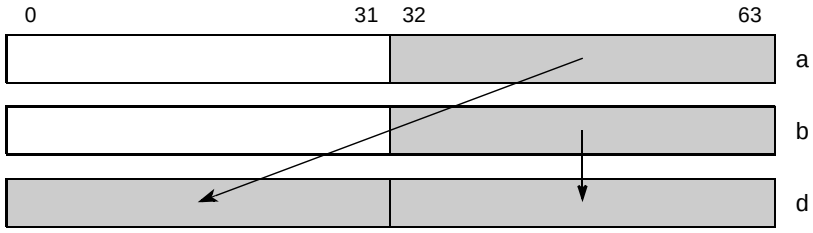

Figure 3-94. Low-Order Element Merging (__ev_mergelo)

NOTE

To perform a vector splat low, specify the same register in parameters a and b**.**

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmergelo d,a,b**

__ev_mergelohi __ev_mergelohi

Vector Merge Low/High

d = __ev_mergelohi (a,b)

$d_{0:31} \leftarrow a_{32:63}$
$d_{32:63} \leftarrow b_{0:31}$

The low-order element of parameter a and the high-order element of parameter b are merged and placed into parameter d, as shown in Figure 3-95.

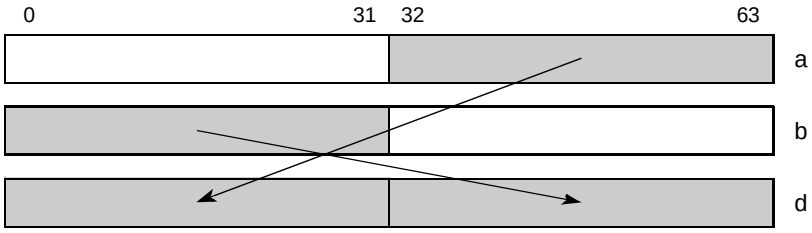

Figure 3-95. Low-Order Element Merging (__ev_mergelohi)

NOTE

To perform a vector swap, specify the same register in parameters a and b.

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmergelohi d,a,b**

__ev_mhegsmfaa __ev_mhegsmfaa

Vector Multiply Half Words, Even, Guarded, Signed, Modulo, Fractional and Accumulate

d = __ev_mhegsmfaa (a,b)

```
temp0:31 ← a32:47 ×sf b32:47
temp0:63 ← EXTS(temp0:31)
d0:63 ← ACC0:63 + temp0:63
// update accumulator
ACC0:63 ← d0:63
```

The corresponding low even-numbered, half-word signed fractional elements in parameters a and b are multiplied. The product is added to the contents of the 64-bit accumulator, and the result is placed into parameter d and the accumulator.

NOTE

This sum is a modulo sum. Neither overflow check nor saturation is performed. Any overflow of the 64-bit sum is not recorded into the SPEFSCR.

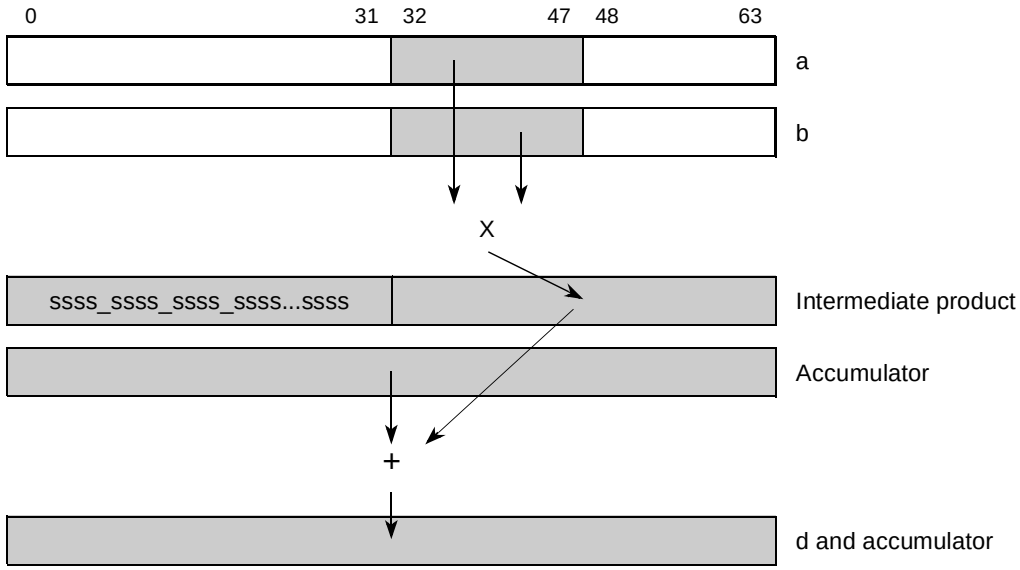

Figure 3-96. __ev_mhegsmfaa (Even Form)

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmhegsmfaa d,a,b**

__ev_mhegsmfan __ev_mhegsmfan

Vector Multiply Half Words, Even, Guarded, Signed, Modulo, Fractional and Accumulate Negative

d = __ev_mhegsmfan (a,b)

```
temp0:31 ← a32:47 ×sf b32:47
temp0:63 ← EXTS(temp0:31)
d0:63 ← ACC0:63 - temp0:63
// update accumulator
ACC0:63 ← d0:63
```

The corresponding low even-numbered, half-word signed fractional elements in parameters a and b are multiplied. The product is subtracted from the contents of the 64-bit accumulator, and the result is placed into parameter d and the accumulator.

NOTE

This difference is a modulo difference. Neither overflow check nor saturation is performed. Any overflow of the 64-bit difference is not recorded into the SPEFSCR.

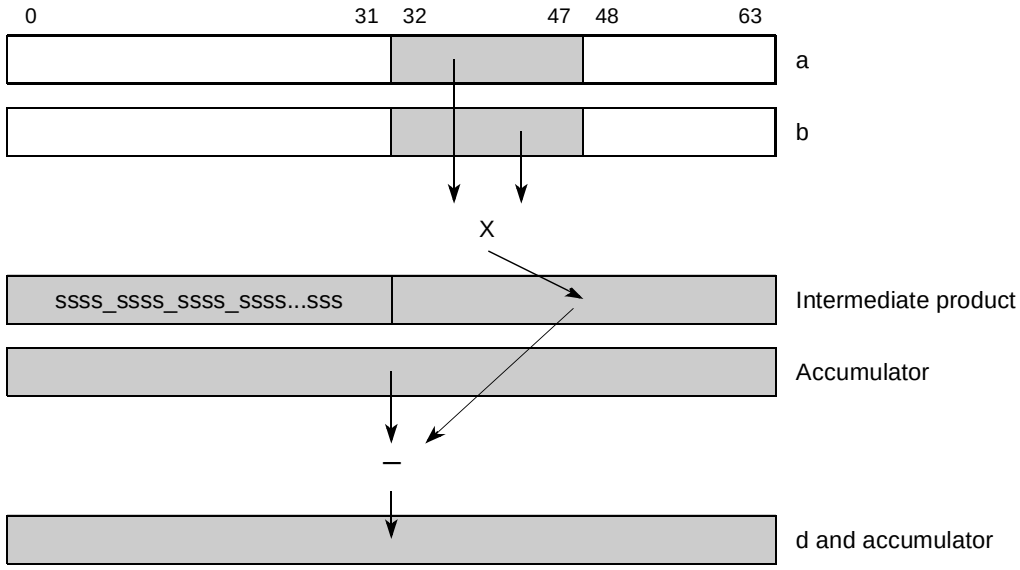

Figure 3-97. __ev_mhegsmfan (Even Form)

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmhegsmfan d,a,b**

__ev_mhegsmiaa __ev_mhegsmiaa

Vector Multiply Half Words, Even, Guarded, Signed, Modulo, Integer and Accumulate

d = __ev_mhegsmiaa (a,b)

```
temp0:31 ← a32:47 ×si b32:47
temp0:63 ← EXTS(temp0:31)
d0:63 ← ACC0:63 + temp0:63

// update accumulator
ACC0:63 ← d0:63
```

The corresponding low even-numbered half-word signed integer elements in parameters a and b are multiplied. The intermediate product is sign-extended and added to the contents of the 64-bit accumulator, and the resulting sum is placed into parameter d and the accumulator.

NOTE

This sum is a modulo sum. Neither overflow check nor saturation is performed. Any overflow of the 64-bit sum is not recorded into the SPEFSCR.

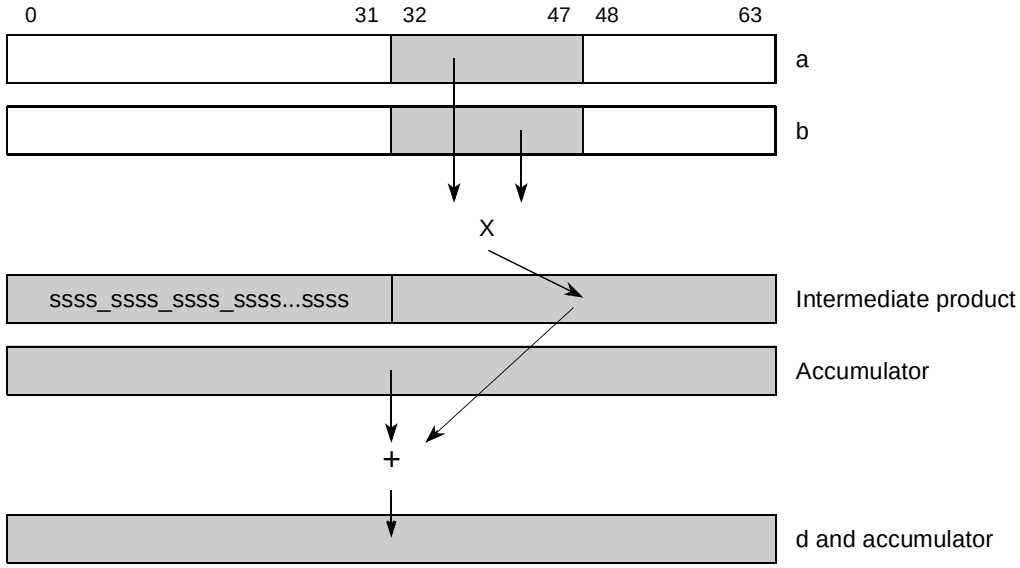

Figure 3-98. __ev_mhegsmiaa (Even Form)

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmhegsmiaa d,a,b**

__ev_mhegsmian __ev_mhegsmian

Vector Multiply Half Words, Even, Guarded, Signed, Modulo, Integer and Accumulate Negative

d = __ev_mhegsmian (a,b)

```
temp0:31 ← a32:47 ×si b32:47
temp0:63 ← EXTS(temp0:31)
d0:63 ← ACC0:63 – temp0:63
// update accumulator
ACC0:63 ← d0:63
```

The corresponding low even-numbered half-word signed integer elements in parameters a and b are multiplied. The intermediate product is sign-extended and subtracted from the contents of the 64-bit accumulator, and the result is placed into parameter d and into the accumulator.

NOTE

This difference is a modulo difference. Neither overflow check nor saturation is performed. Any overflow of the 64-bit difference is not recorded into the SPEFSCR.

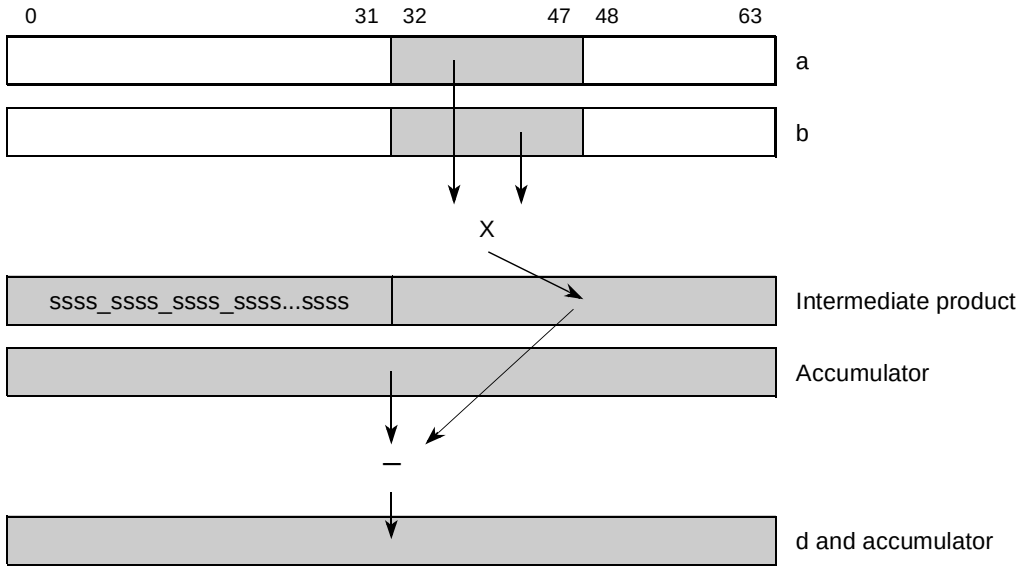

Figure 3-99. __ev_mhegsmian (Even Form)

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmhegsmian d,a,b**

__ev_mhegumfaa __ev_mhegumfaa

Vector Multiply Half Words, Even, Guarded, Unsigned, Modulo, Fractional and Accumulate

d = __ev_mhegumfaa (a,b)

```
temp0:31 ← a32:47 ×ui b32:47
temp0:63 ← EXTZ(temp0:31)
d0:63 ← ACC0:63 + temp0:63
// update accumulator
ACC0:63 ← d0:63
```

The corresponding low even-numbered elements in parameters a and b are multiplied. The intermediate product is zero-extended and added to the contents of the 64-bit accumulator. The resulting sum is placed into parameter d and into the accumulator.

NOTE

This sum is a modulo sum. Neither overflow check nor saturation is performed. Overflow of the 64-bit sum is not recorded into the SPEFSCR.

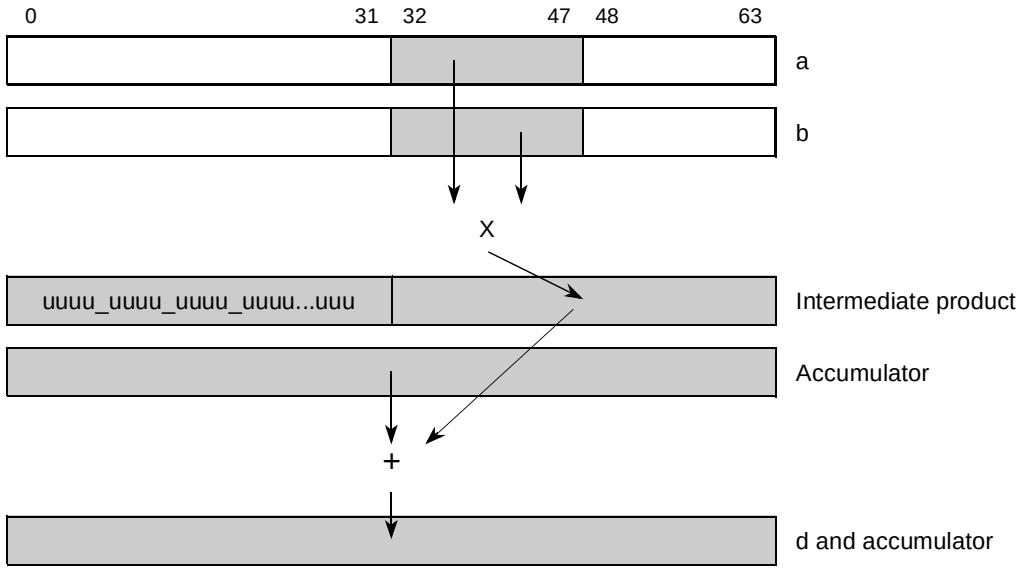

Figure 3-100. __ev_mhegumfaa (Even Form)

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmhegumiaa d,a,b**

__ev_mhegumiaa __ev_mhegumiaa

Vector Multiply Half Words, Even, Guarded, Unsigned, Modulo, Integer and Accumulate

d = __ev_mhegumiaa (a,b)

```
temp0:31 ← a32:47 ×ui b32:47
temp0:63 ← EXTZ(temp0:31)
d0:63 ← ACC0:63 + temp0:63
// update accumulator
ACC0:63 ← d0:63
```

The corresponding low even-numbered half-word unsigned integer elements in parameters a and b are multiplied. The intermediate product is zero-extended and added to the contents of the 64-bit accumulator. The resulting sum is placed into parameter d and into the accumulator.

NOTE

This sum is a modulo sum. Neither overflow check nor saturation is performed. Any overflow of the 64-bit sum is not recorded into the SPEFSCR.

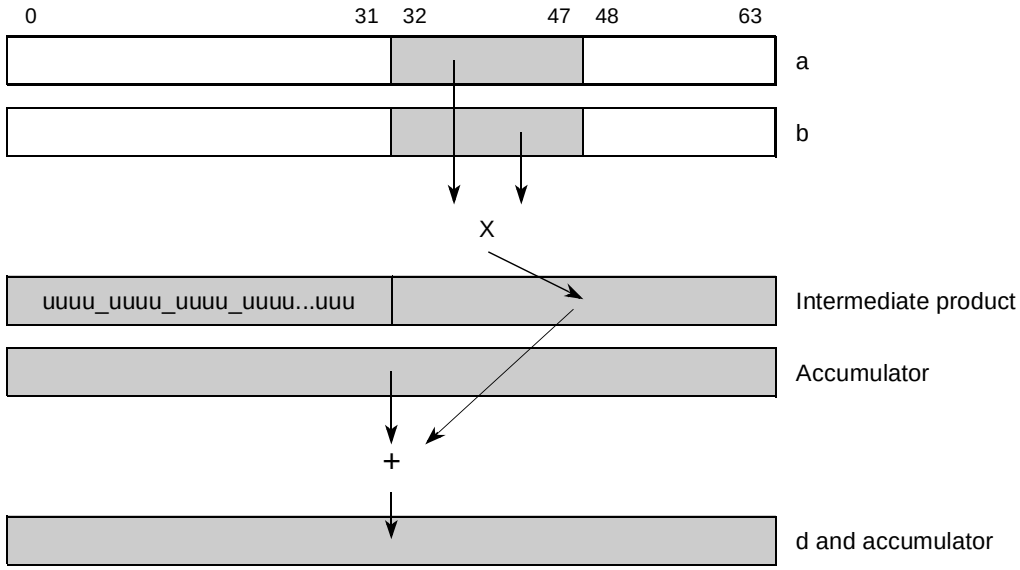

Figure 3-101. __ev_mhegumiaa (Even Form)

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmhegumiaa d,a,b**

__ev_mhegumfan __ev_mhegumfan

Vector Multiply Half Words, Even, Guarded, Unsigned, Modulo, Fractional and Accumulate Negative

d = __ev_mhegumfan (a,b)

```
temp0:31 ← a32:47 ×ui b32:47
temp0:63 ← EXTZ(temp0:31)
d0:63 ← ACC0:63 - temp0:63
// update accumulator
ACC0:63 ← d0:63
```

The corresponding low even-numbered elements in parameters a and b are multiplied. The intermediate product is zero-extended and subtracted from the contents of the 64-bit accumulator. The result is placed into parameter d and into the accumulator.

NOTE

This difference is a modulo difference. Neither overflow check nor saturation is performed. Overflow of the 64-bit difference is not recorded into the SPEFSCR.

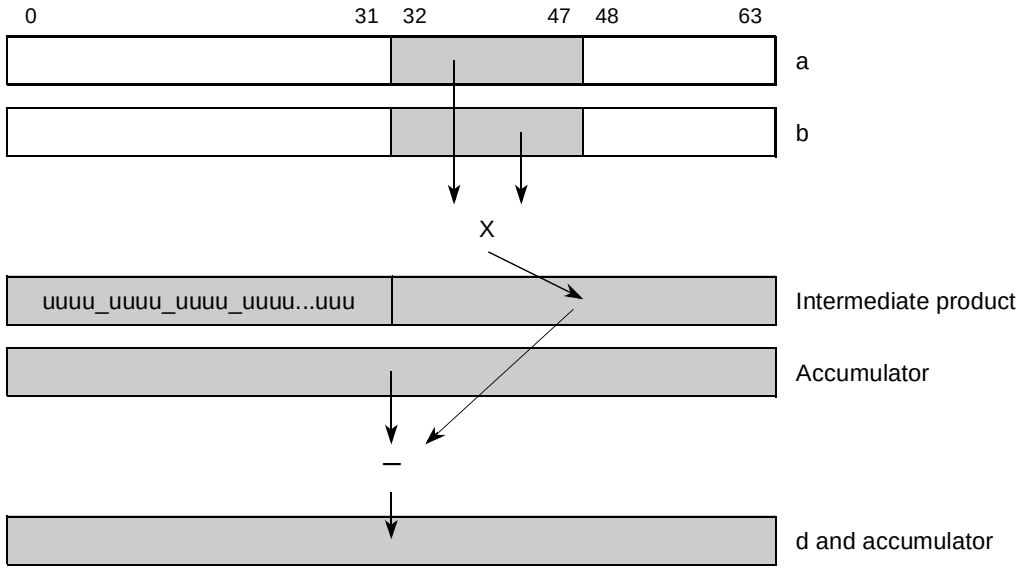

Figure 3-102. __ev_mhegumfan (Even Form)

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmhegumian d,a,b**

__ev_mhegumian __ev_mhegumian

Vector Multiply Half Words, Even, Guarded, Unsigned, Modulo, Integer and Accumulate Negative

d = __ev_mhegumian (a,b)

```
temp0:31 ← a32:47 ×ui b32:47
temp0:63 ← EXTZ(temp0:31)
d0:63 ← ACC0:63 - temp0:63
// update accumulator
ACC0:63 ← d0:63
```

The corresponding low even-numbered unsigned integer elements in parameter a and b are multiplied. The intermediate product is zero-extended and subtracted from the contents of the 64-bit accumulator. The result is placed into parameter d and into the accumulator.

NOTE

This difference is a modulo difference. Neither overflow check nor saturation is performed. Any overflow of the 64-bit difference is not recorded into the SPEFSCR.

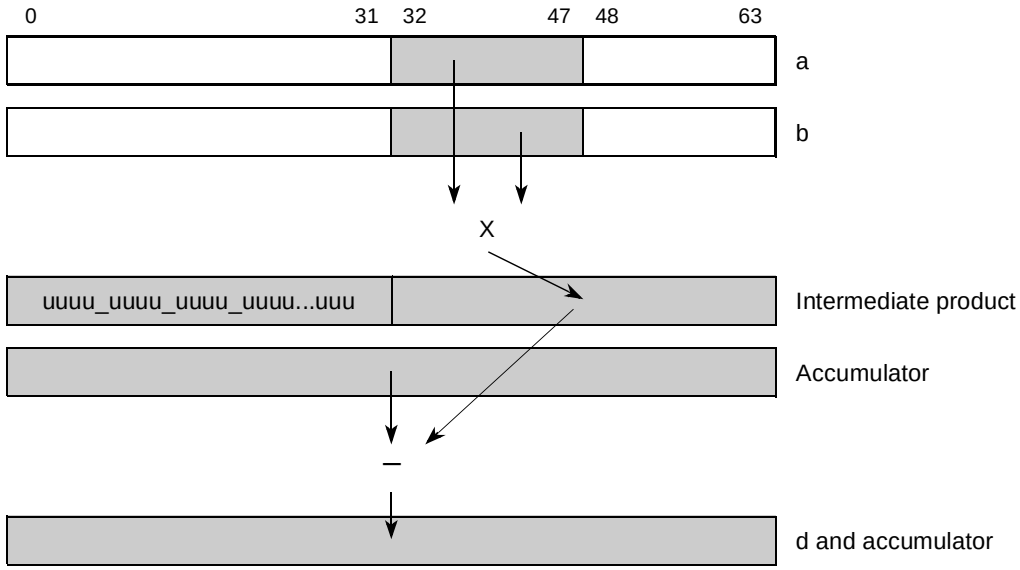

Figure 3-103. __ev_mhegumian (Even Form)

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmhegumian d,a,b**

__ev_mhesmf __ev_mhesmf

Vector Multiply Half Words, Even, Signed, Modulo, Fractional (to Accumulator)

d = __ev_mhesmf (a,b) (A = 0)
d = __ev_mhesmfa (a,b) (A = 1)

```
// high
d0:31 ←(a0:15 ×sf b0:15)

// low
d32:63←(a32:47 ×sf b32:47)

// update accumulator
if A = 1 then ACC0:63 ← d0:63
```

The corresponding even-numbered half-word signed fractional elements in parameters a and b are multiplied, and the 32 bits of each product are placed into the corresponding words of parameter d.

If A = 1, the result in parameter d is also placed into the accumulator.

Other registers altered: ACC (if A = 1)

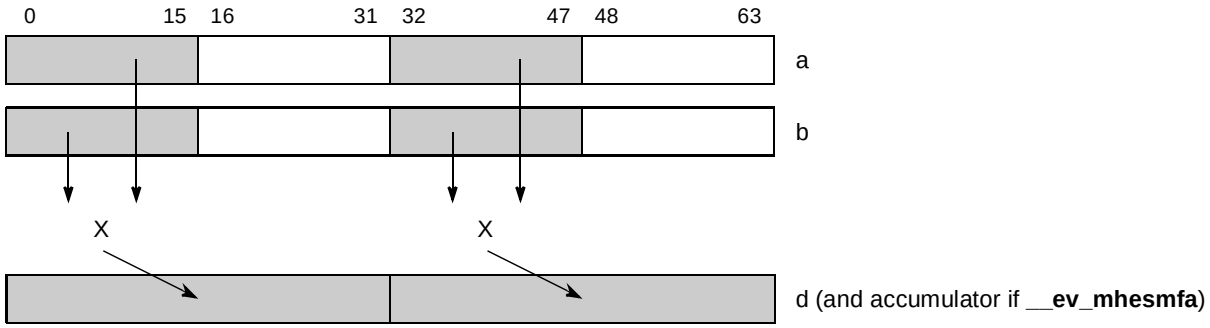

Figure 3-104. Even Multiply of Two Signed Modulo Fractional Elements (to Accumulator) (__ev_mhesmf)

A	d	a	b	Maps to
A = 0	__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmhesmf d,a,b**
A = 1	__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmhesmfa d,a,b**

__ev_mhesmfaaw __ev_mhesmfaaw

Vector Multiply Half Words, Even, Signed, Modulo, Fractional and Accumulate into Words

d = __ev_mhesmfaaw (a,b)

```
// high
temp0:31 ← (a0:15 ×sf b0:15)
d0:31 ← ACC0:31 + temp0:31

// low
temp0:31 ← (a32:47 ×sf b32:47)
d32:63 ← ACC32:63 + temp0:31

// update accumulator
ACC0:63 ← d0:63
```

For each word element in the accumulator, the corresponding even-numbered half-word signed fractional elements in parameters a and b are multiplied. The 32 bits of each intermediate product are added to the contents of the accumulator words to form intermediate sums, which are placed into the corresponding parameter d words and into the accumulator.

Other registers altered: ACC

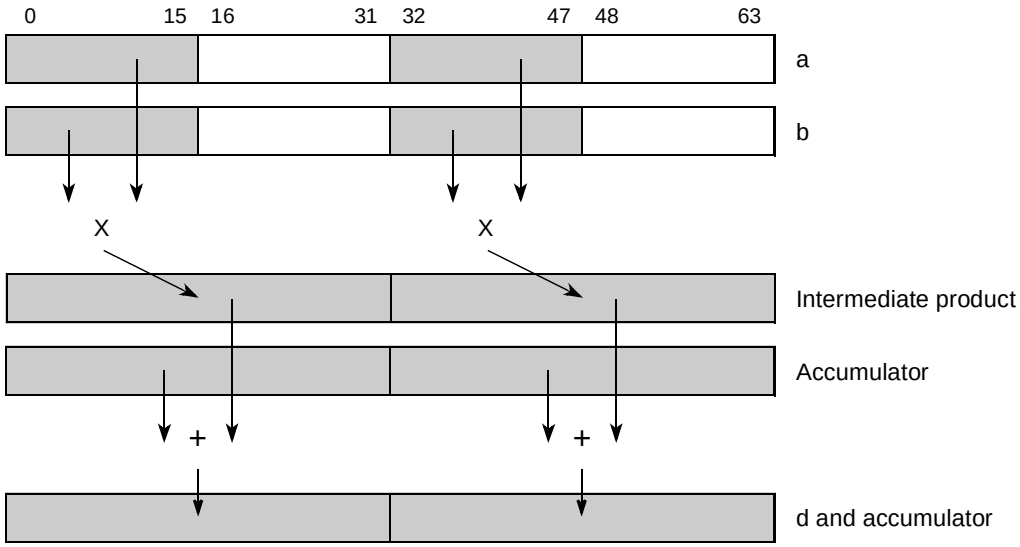

Figure 3-105. Even Form of Vector Half-Word Multiply (__ev_mhesmfaaw)

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmhesmfaaw d,a,b**

__ev_mhesmfanw __ev_mhesmfanw

Vector Multiply Half Words, Even, Signed, Modulo, Fractional and Accumulate Negative into Words

d = __ev_mhesmfanw (a,b)

```
// high
temp0:31 ← a0:15 ×sf b0:15
d0:31 ← ACC0:31 - temp0:31

// low
temp0:31 ← a32:47 ×sf b32:47
d32:63← ACC32:63 - temp0:31

// update accumulator
ACC0:63 ← d0:63
```

For each word element in the accumulator, the corresponding even-numbered half-word signed fractional elements in parameters a and b are multiplied. The 32-bit intermediate products are subtracted from the contents of the accumulator words to form intermediate differences, which are placed into the corresponding parameter d words and into the accumulator.

Other registers altered: ACC

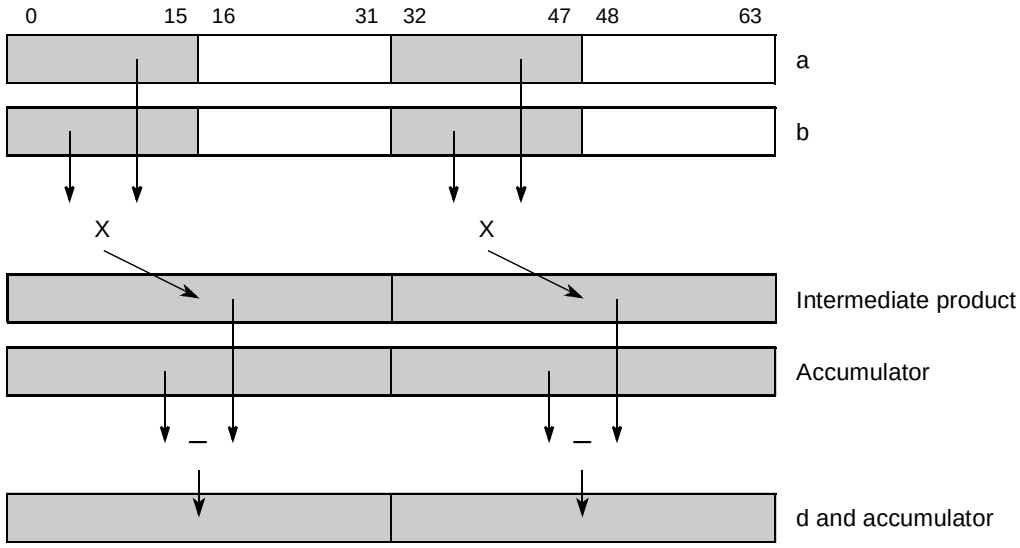

Figure 3-106. Even Form of Vector Half-Word Multiply (__ev_mhesmfanw)

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmhesmfanw d,a,b**

__ev_mhesmi __ev_mhesmi

Vector Multiply Half Words, Even, Signed, Modulo, Integer (to Accumulator)

d = __ev_mhesmi (a,b) (A = 0)
d = __ev_mhesmia (a,b) (A = 1)

```
// high
d0:31 ← a0:15 ×si b0:15

// low
d32:63 ← a32:47 ×si b32:47

// update accumulator
if A = 1, then ACC0:63 ← d0:63
```

The corresponding even-numbered half-word signed integer elements in parameters a and b are multiplied. The two 32-bit products are placed into the corresponding words of parameter d.

If A = 1, the result in parameter d is also placed into the accumulator.

Other registers altered: ACC (if A = 1)

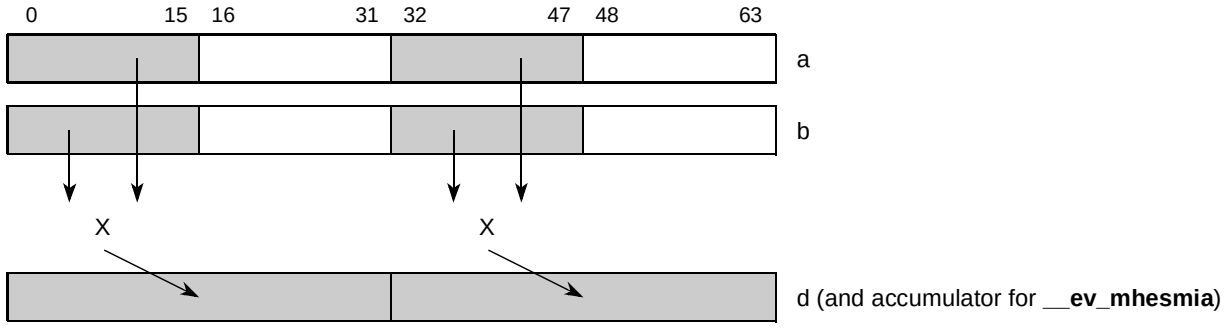

Figure 3-107. Even Form for Vector Multiply (to Accumulator) (__ev_mhesmi)

A	d	a	b	Maps to
A = 0	__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmhesmi d,a,b**
A = 1	__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmhesmia d,a,b**

__ev_mhesmiaaw __ev_mhesmiaaw

Vector Multiply Half Words, Even, Signed, Modulo, Integer and Accumulate into Words

d = __ev_mhesmiaaw (a,b)

```
// high
temp0:31 ← a0:15 ×si b0:15
d0:31 ← ACC0:31 + temp0:31

// low
temp0:31 ← a32:47 ×si b32:47
d32:63 ← ACC32:63 + temp0:31

// update accumulator
ACC0:63 ← d0:63
```

For each word element in the accumulator, the corresponding even-numbered half-word signed integer elements in parameters a and b are multiplied. Each intermediate 32-bit product is added to the contents of the accumulator words to form intermediate sums, which are placed into the corresponding parameter d words and into the accumulator.

Other registers altered: ACC

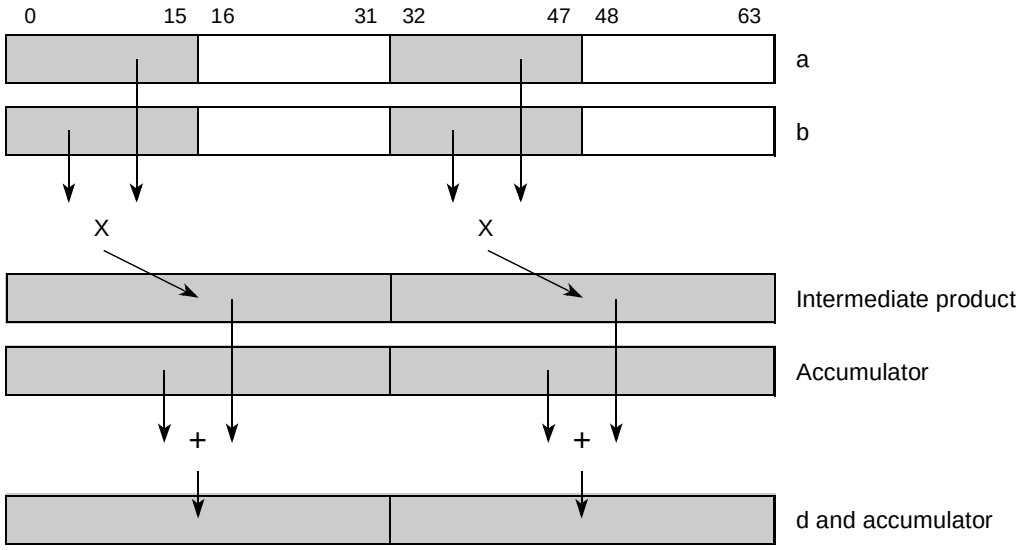

Figure 3-108. Even Form of Vector Half-Word Multiply (__ev_mhesmiaaw)

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmhesmiaaw d,a,b**

__ev_mhesmianw __ev_mhesmianw

Vector Multiply Half Words, Even, Signed, Modulo, Integer and Accumulate Negative into Words

d = __ev_mhesmianw (a,b)

```
// high
temp0[0:31] ←a[0:15] ×si b[0:15]
d[0:31] ← ACC[0:31] - temp0[0:31]
// low
temp1[0:31] ← a[32:47] ×si b[32:47]
d[32:63] ← ACC[32:63] - temp1[0:31]
// update accumulator
ACC[0:63] ← d[0:63]
```

For each word element in the accumulator, the corresponding even-numbered half-word signed integer elements in parameters a and b are multiplied. Each intermediate 32-bit product is subtracted from the contents of the accumulator words to form intermediate differences, which are placed into the corresponding parameter d words and into the accumulator.

Other registers altered: ACC

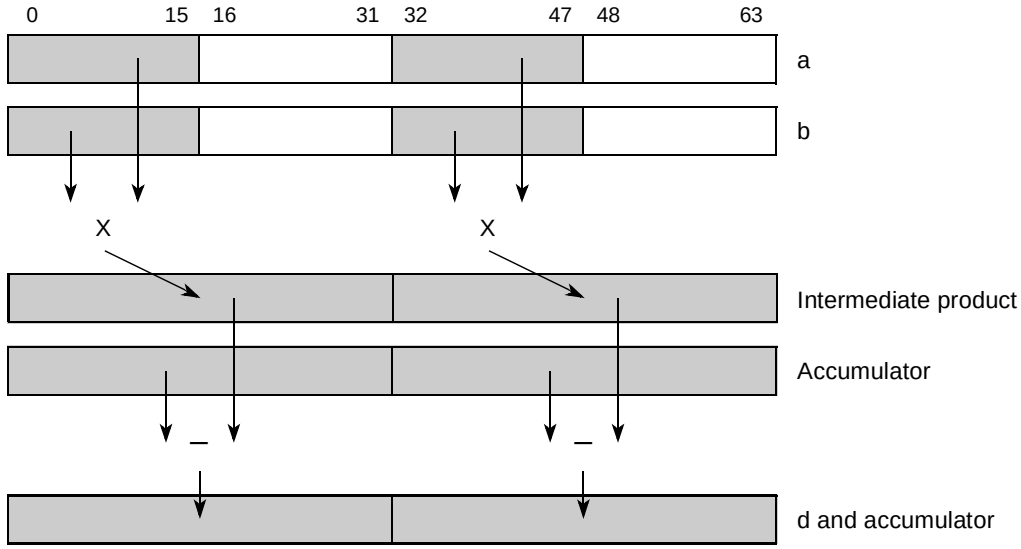

Figure 3-109. Even Form of Vector Half-Word Multiply (__ev_mhesmianw)

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmhesmianw d,a,b**

__ev_mhessf __ev_mhessf

Vector Multiply Half Words, Even, Signed, Saturate, Fractional (to Accumulator)

d = __ev_mhessf (a,b) (A = 0)
d = __ev_mhessfa (a,b) (A = 1)

```
// high
temp0:31 ← a0:15 ×sf b0:15
if (a0:15 = 0x8000) & (b0:15 = 0x8000) then
    d0:31 ← 0x7FFF_FFFF //saturate
    movh ← 1
else
    d0:31 ← temp0:31
    movh ← 0

// low
temp0:31 ← a32:47 ×sf b32:47
if (a32:47 = 0x8000) & (b32:47 = 0x8000) then
    d32:63 ← 0x7FFF_FFFF //saturate
    movl ← 1
else
    d32:63 ← temp0:31
    movl ← 0

// update accumulator
if A = 1 then ACC0:63 ← d0:63

// update SPEFSCR
SPEFSCROVH ← movh
SPEFSCROV ← movl
SPEFSCRSOVH ← SPEFSCRSOVH | movh
SPEFSCRSOV ← SPEFSCRSOV | movl
```

The corresponding even-numbered half-word signed fractional elements in parameters a and b are multiplied. The 32 bits of each product are placed into the corresponding words of parameter d. If both inputs are -1.0, the result saturates to the largest positive signed fraction and the overflow and summary overflow bits are recorded in the SPEFSCR.

If A = 1, the result in parameter d is also placed into the accumulator.

Other registers altered: SPEFSCR
ACC (if A = 1)

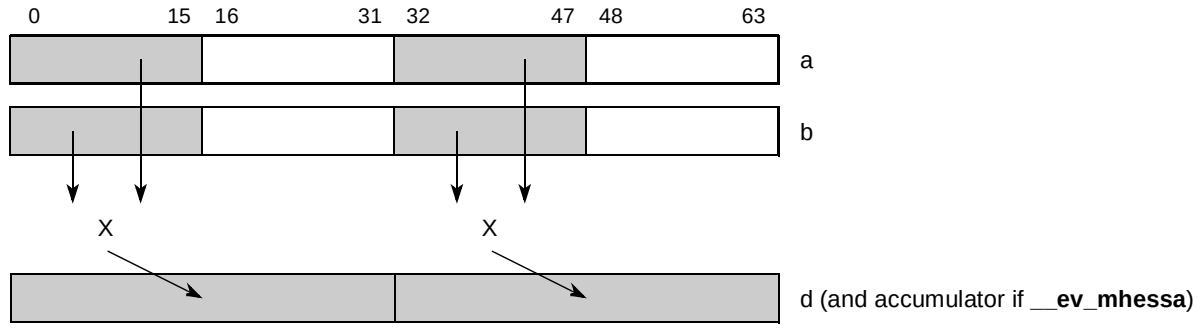

Figure 3-110. Even Multiply of Two Signed Saturate Fractional Elements (to Accumulator) (__ev_mhessf)

A	d	a	b	Maps to
A = 0	__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmhessf d,a,b**
A = 1	__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmhessfa d,a,b**

__ev_mhessfaaw __ev_mhessfaaw

Vector Multiply Half Words, Even, Signed, Saturate, Fractional and Accumulate into Words

d = __ev_mhessfaaw (a,b)

```
// high
temp0:31 ← a0:15 ×sf b0:15
if (a0:15 = 0x8000) & (b0:15 = 0x8000) then
    temp0:31 ← 0x7FFF_FFFF //saturate
    movh ← 1
else
    movh ← 0
temp0:63 ← EXTS(ACC0:31) + EXTS(temp0:31)
ovh ← (temp31 ⊕ temp32)
d0:31 ← SATURATE(ovh, temp31, 0x8000_0000, 0x7FFF_FFFF, temp32:63)

// low
temp0:31 ← a32:47 ×sf b32:47
if (a32:47 = 0x8000) & (b32:47 = 0x8000) then
    temp0:31 ← 0x7FFF_FFFF //saturate
    movl ← 1
else
    movl ← 0
temp0:63 ← EXTS(ACC32:63) + EXTS(temp0:31)
ovl ← (temp31 ⊕ temp32)
d32:63 ← SATURATE(ovl, temp31, 0x8000_0000, 0x7FFF_FFFF, temp32:63)

// update accumulator
ACC0:63 ← d0:63

// update SPEFSCR
SPEFSCROVH ← movh
SPEFSCROV ← movl
SPEFSCRSOVH ← SPEFSCRSOVH | ovh | movh
SPEFSCRSOV ← SPEFSCRSOV | ovl| movl
```

The corresponding even-numbered half-word signed fractional elements in parameters a and b are multiplied, producing a 32-bit product. If both inputs are –1.0, the result saturates to 0x7FFF_FFFF. Each 32-bit product is then added to the corresponding word in the accumulator, saturating if overflow or underflow occurs, and the result is placed in parameter d and the accumulator.

If there is an overflow or underflow from either the multiply or the addition, the overflow and summary overflow bits are recorded in the SPEFSCR.

Other registers altered: SPEFSCR ACC

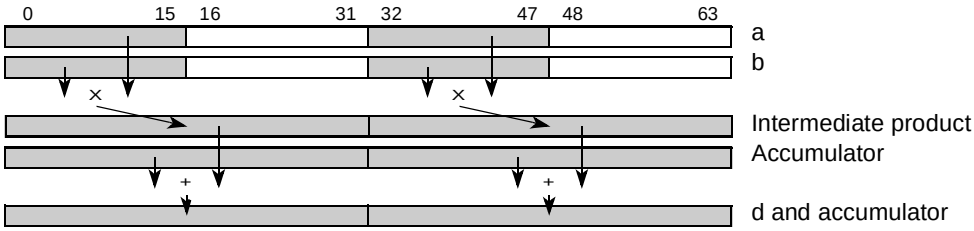

Figure 3-111. Even Form of Vector Half-Word Multiply (__ev_mhessfaaw)

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmhessfaaw d,a,b**

__ev_mhessfanw __ev_mhessfanw

Vector Multiply Half Words, Even, Signed, Saturate, Fractional and Accumulate Negative into Words

d = __ev_mhessfanw (a,b)

```
// high
temp0:31 ← a0:15 ×sf b0:15
if (a0:15 = 0x8000) & (b0:15 = 0x8000) then
    temp0:31 ← 0x7FFF_FFFF //saturate
    movh ← 1
else
    movh ← 0
temp0:63 ← EXTS(ACC0:31) - EXTS(temp0:31)
ovh ← (temp31 ⊕ temp32)
d0:31 ← SATURATE(ovh, temp31, 0x8000_0000, 0x7FFF_FFFF, temp32:63)

// low
temp0:31 ← a32:47 ×sf b32:47
if (a32:47 = 0x8000) & (b32:47 = 0x8000) then
    temp0:31 ← 0x7FFF_FFFF //saturate
    movl ← 1
else
    movl ← 0
temp0:63 ← EXTS(ACC32:63) - EXTS(temp0:31)
ovl ← (temp31 ⊕ temp32)
d32:63 ← SATURATE(ovl, temp31, 0x8000_0000, 0x7FFF_FFFF, temp32:63)

// update accumulator
ACC0:63 ← d0:63

// update SPEFSCR
SPEFSCROVH ← movh
SPEFSCROV ← movl
SPEFSCRSOVH ← SPEFSCRSOVH | ovh | movh
SPEFSCRSOV ← SPEFSCRSOV | ovl| movl
```

The corresponding even-numbered half-word signed fractional elements in parameters a and b are multiplied, producing a 32-bit product. If both inputs are –1.0, the result saturates to 0x7FFF_FFFF. Each 32-bit product is then subtracted from the corresponding word in the accumulator, saturating if overflow or underflow occurs, and the result is placed in parameter d and the accumulator.

If there is an overflow or underflow from either the multiply or the addition, the overflow and summary overflow bits are recorded in the SPEFSCR.

Other registers altered: SPEFSCR ACC

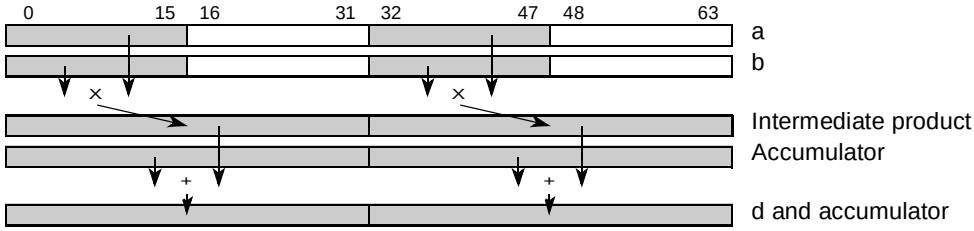

Figure 3-112. Even Form of Vector Half-Word Multiply (__ev_mhessfanw)

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmhessfanw d,a,b**

__ev_mhessiaaw __ev_mhessiaaw

Vector Multiply Half Words, Even, Signed, Saturate, Integer and Accumulate into Words

d = __ev_mhessiaaw (a,b)

```
// high
temp0:31 ← a0:15 ×si b0:15
temp0:63 ← EXTS(ACC0:31) + EXTS(temp0:31)
ovh ← (temp31 ⊕ temp32)
d0:31 ← SATURATE(ovh, temp31, 0x8000_0000, 0x7FFF_FFFF, temp32:63)

// low
temp0:31 ← a32:47 ×si b32:47
temp0:63 ← EXTS(ACC32:63) + EXTS(temp0:31)
ovl ← (temp31 ⊕ temp32)
d32:63 ← SATURATE(ovl, temp31, 0x8000_0000, 0x7FFF_FFFF, temp32:63)

// update accumulator
ACC0:63 ← d0:63

// update SPEFSCR
SPEFSCROVH ← ovh
SPEFSCROV ← ovl
SPEFSCRSOVH ← SPEFSCRSOVH | ovh
SPEFSCRSOV ← SPEFSCRSOV | ovl
```

The corresponding even-numbered half-word signed integer elements in parameters a and b are multiplied, producing a 32-bit product. Each 32-bit product is then added to the corresponding word in the accumulator, saturating if overflow occurs, and the result is placed in parameter d and the accumulator.

If there is an overflow or underflow from either the multiply or the addition, the overflow and summary overflow bits are recorded in the SPEFSCR.

Other registers altered: SPEFSCR ACC

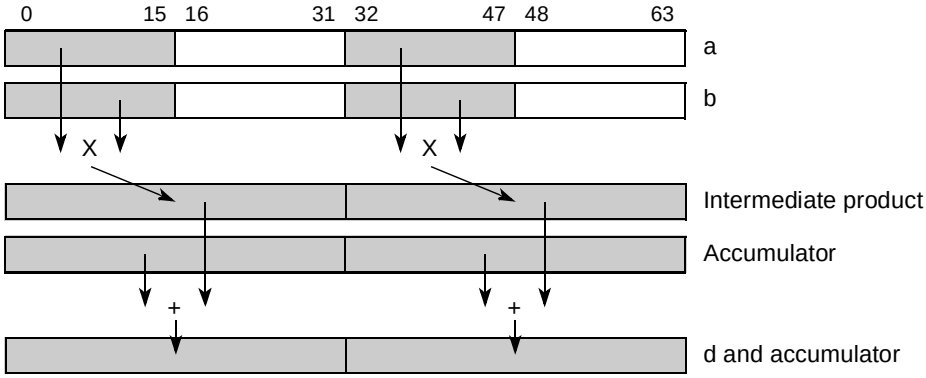

Figure 3-113. Even Form of Vector Half-Word Multiply (__ev_mhessiaaw)

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmhessiaaw d,a,b**

__ev_mhessianw __ev_mhessianw

Vector Multiply Half Words, Even, Signed, Saturate, Integer and Accumulate Negative into Words

d = __ev_mhessianw (a,b)

```
// high
temp0:31 ← a0:15 ×si b0:15
temp0:63 ← EXTS(ACC0:31) - EXTS(temp0:31)
ovh ← (temp31 ⊕ temp32)
d0:31 ← SATURATE(ovh, temp31, 0x8000_0000, 0x7FFF_FFFF, temp32:63)

// low
temp0:31 ← a32:47 ×si b32:47
temp0:63 ← EXTS(ACC32:63) - EXTS(temp0:31)
ovl ← (temp31 ⊕ temp32)
d32:63 ← SATURATE(ovl, temp31, 0x8000_0000, 0x7FFF_FFFF, temp32:63)

// update accumulator
ACC0:63 ← d0:63

// update SPEFSCR
SPEFSCROVH ← ovh
SPEFSCROV ← ovl
SPEFSCRSOVH ← SPEFSCRSOVH | ovh
SPEFSCRSOV ← SPEFSCRSOV | ovl
```

For each word element in the accumulator, the corresponding even-numbered half-word signed integer elements in parameters a and b are multiplied, producing a 32-bit product. Each 32-bit product is then subtracted from the corresponding word in the accumulator, saturating if overflow occurs, and the result is placed in parameter d and the accumulator.

If there is an overflow or underflow from either the multiply or the addition, the overflow and summary overflow bits are recorded in the SPEFSCR.

Other registers altered: SPEFSCR ACC

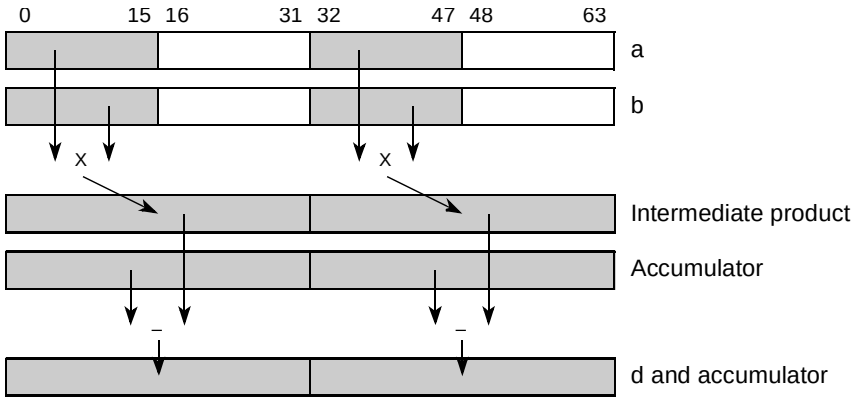

Figure 3-114. Even Form of Vector Half-Word Multiply (__ev_mhessianw)

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmhessianw d,a,b**

__ev_mheumf __ev_mheumf

Vector Multiply Half Words, Even, Unsigned, Modulo, Fractional (to Accumulator)

d = __ev_mheumf (a,b) (A = 0)
d = __ev_mheumfa (a,b) (A = 1)

```
// high
d0:31 ← a0:15 ×ui b0:15
// low
d32:63 ← a32:47 ×ui b32:47
// update accumulator
if A = 1, ACC0:63 ← d0:63
```

The corresponding even-numbered half word elements in parameters a and b are multiplied. The two 32-bit products are placed into the corresponding words of parameter d.

If A = 1, the result in parameter d is also placed into the accumulator.

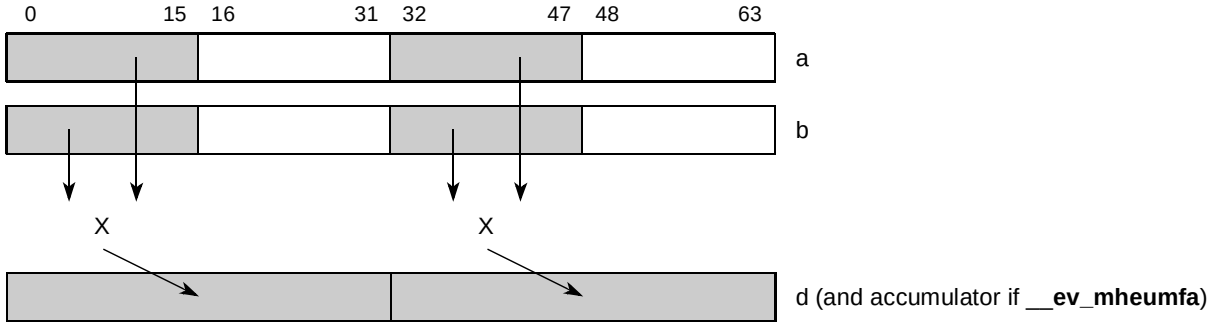

Figure 3-115. Vector Multiply Half Words, Even, Unsigned, Modulo, Fractional (to Accumulator) (__ev_mheumf)

A	d	a	b	Maps to
A = 0	__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmheumi d,a,b**
A = 1	__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmheumia d,a,b**

__ev_mheumi __ev_mheumi

Vector Multiply Half Words, Even, Unsigned, Modulo, Integer (to Accumulator)

d = __ev_mheumi (a,b) (A = 0)
d = __ev_mheumia (a,b) (A = 1)

```
// high
d0:31 ← a0:15 ×ui b0:15

// low
d32:63 ← a32:47 ×ui b32:47

// update accumulator
if A = 1 then ACC0:63 ← d0:63
```

The corresponding even-numbered half-word unsigned integer elements in parameters a and b are multiplied. The two 32-bit products are placed into the corresponding words of parameter d.

If A = 1, the result in parameter d is also placed into the accumulator.

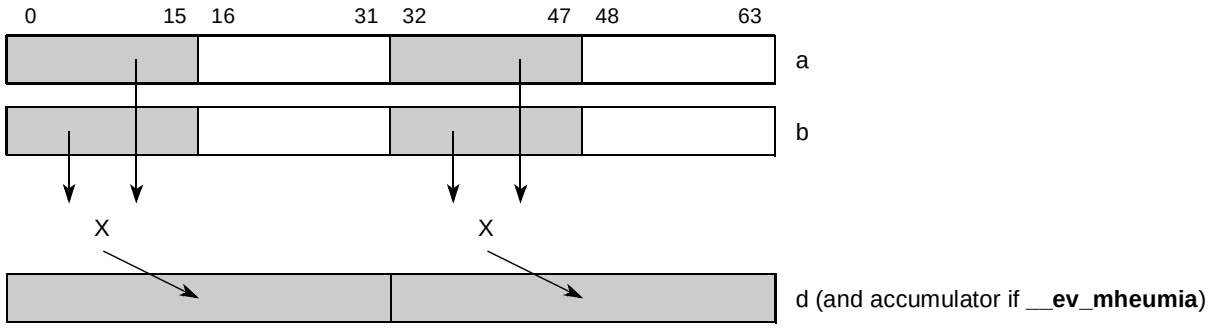

Figure 3-116. Vector Multiply Half Words, Even, Unsigned, Modulo, Integer (to Accumulator) (__ev_mheumi)

A	d	a	b	Maps to
A = 0	__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmheumi d,a,b**
A = 1	__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmheumia d,a,b**

__ev_mheumfaaw __ev_mheumfaaw

Vector Multiply Half Words, Even, Unsigned, Modulo, Fractional and Accumulate into Words

d = __ev_mheumfaaw (a,b)

```
// high
temp0[0:31] ← a[0:15] ×ui b[0:15]
d[0:31] ← ACC[0:31] + temp0[0:31]
// low
temp1[0:31] ← a[32:47] ×ui b[32:47]
d[32:63] ← ACC[32:63] + temp1[0:31]
// update accumulator
ACC[0:63] ← d[0:63]
```

For each word element in the accumulator, the corresponding even-numbered half word elements in parameters a and b are multiplied. Each intermediate product is added to the contents of the corresponding accumulator words, and the sums are placed into the corresponding parameter d and accumulator words.

Other registers altered: ACC

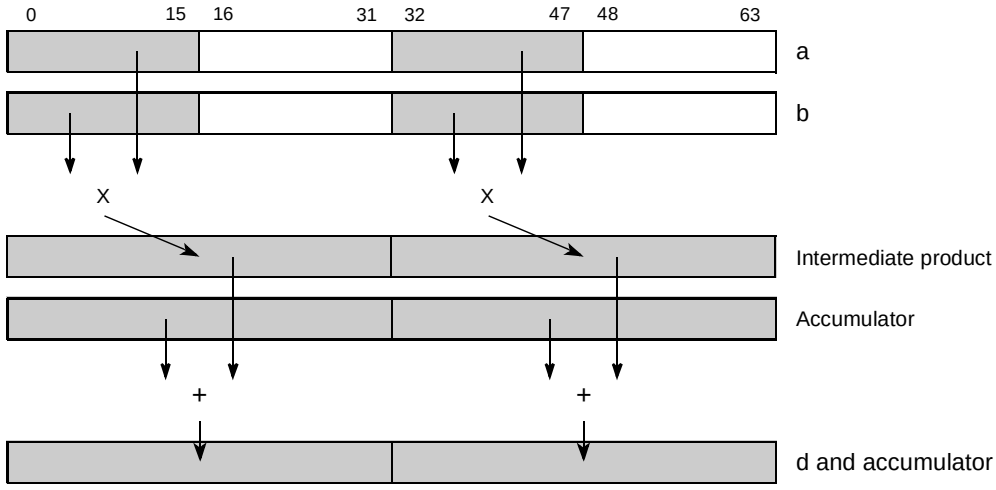

Figure 3-117. Even Form of Vector Half-Word Multiply (__ev_mheumfaaw)

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmheumiaaw d,a,b**

__ev_mheumiaaw __ev_mheumiaaw

Vector Multiply Half Words, Even, Unsigned, Modulo, Integer and Accumulate into Words

d = __ev_mheumiaaw (a,b)

```
// high
temp0:31 ← a0:15 ×ui b0:15
d0:31 ← ACC0:31 + temp0:31

// low
temp0:31 ← a32:47 ×ui b32:47
d32:63 ← ACC32:63 + temp0:31

// update accumulator
ACC0:63 ← d0:63
```

For each word element in the accumulator, the corresponding even-numbered half-word unsigned integer elements in parameters a and b are multiplied. Each intermediate product is added to the contents of the corresponding accumulator words, and the sums are placed into the corresponding parameter d and accumulator words.

Other registers altered: ACC

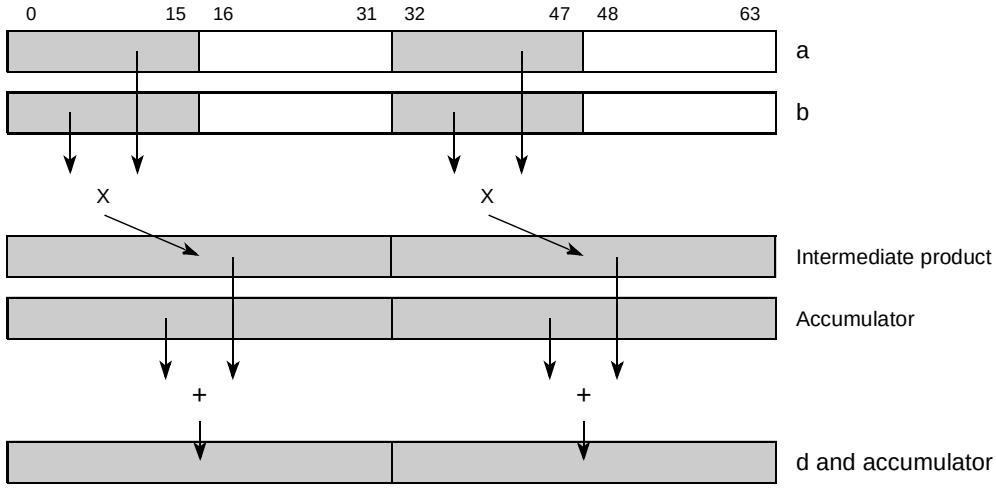

Figure 3-118. Even Form of Vector Half-Word Multiply (__ev_mheumiaaw)

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmheumiaaw d,a,b**

__ev_mheumfanw __ev_mheumfanw

Vector Multiply Half Words, Even, Unsigned, Modulo, Fractional and Accumulate Negative into Words

d = __ev_mheumfanw (a,b)

```
// high
temp0[0:31] ← a[0:15] ×ui b[0:15]
d[0:31] ← ACC[0:31] - temp0[0:31]
// low
temp1[0:31] ← a[32:47] ×ui b[32:47]
d[32:63] ← ACC[32:63] - temp1[0:31]
// update accumulator
ACC[0:63] ← d[0:63]
```

For each word element in the accumulator, the corresponding even-numbered half word elements in parameters a and b are multiplied. Each intermediate product is subtracted from the contents of the corresponding accumulator words. The differences are placed into the corresponding parameter d and accumulator words.

Other registers altered: ACC

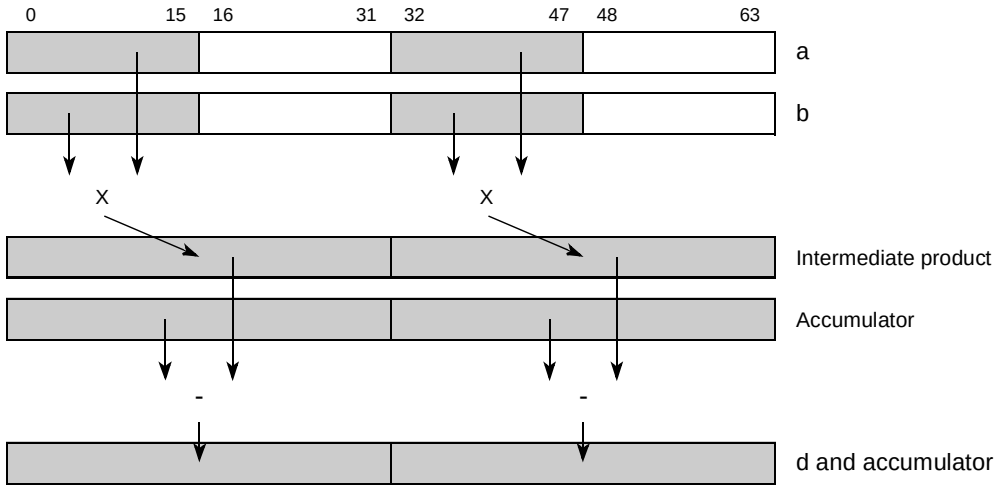

Figure 3-119. Even Form of Vector Half-Word Multiply (__ev_mheumfanw)

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmheumianw d,a,b**

__ev_mheumianw __ev_mheumianw

Vector Multiply Half Words, Even, Unsigned, Modulo, Integer and Accumulate Negative into Words

d = __ev_mheumianw (a,b)

```
// high
temp0:31 ← a0:15 ×ui b0:15
d0:31 ← ACC0:31 - temp0:31

// low
temp0:31 ← a32:47 ×ui b32:47
d32:63 ← ACC32:63 - temp0:31

// update accumulator
ACC0:63 ← d0:63
```

For each word element in the accumulator, the corresponding even-numbered half-word unsigned integer elements in parameters a and b are multiplied. Each intermediate product is subtracted from the contents of the corresponding accumulator words. The differences are placed into the corresponding parameter d and accumulator words.

Other registers altered: ACC

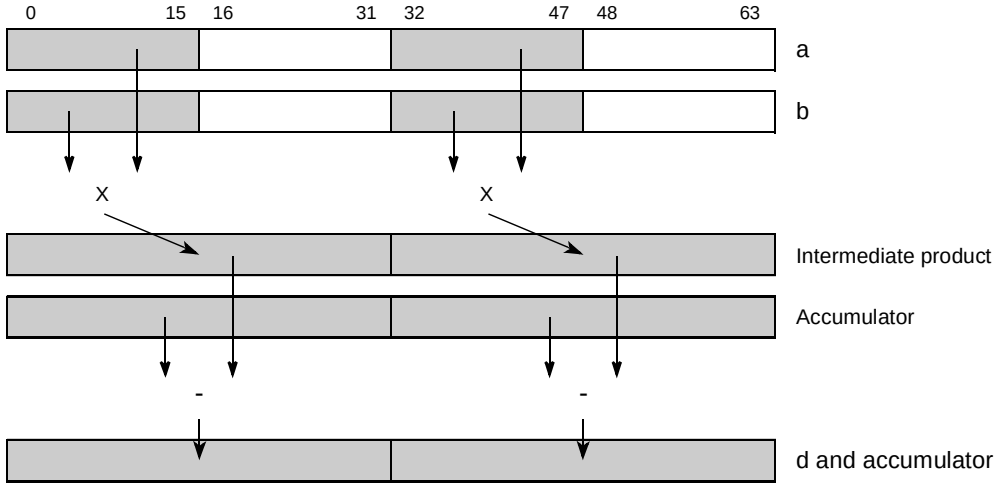

Figure 3-120. Even Form of Vector Half-Word Multiply (__ev_mheumianw)

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmheumianw d,a,b**

__ev_mheusfaaw __ev_mheusfaaw

Vector Multiply Half Words, Even, Unsigned, Saturate, Fractional and Accumulate into Words

d = __ev_mheusfaaw (a,b)

```
// high
temp0[0:31] ← a[0:15] ×ui b[0:15]
temp0[0:63] ← EXTZ(ACC[0:31]) + EXTZ(temp0[0:31])
if temp0[31] = 1
    d[0:31] ← 0xFFFF_FFFF //overflow
    ovh ← 1
else
    d[0:31] ← temp0[32:63]
    ovh ← 0
//low
temp1[0:31] ← a[32:47] ×ui b[32:47]
temp1[0:63] ← EXTZ(ACC[32:63]) + EXTZ(temp1[0:31])
if temp1[31] = 1
    d[32:63] ← 0xFFFF_FFFF //overflow
    ovl ← 1
else
    d[32:63] ← temp1[32:63]
    ovl ← 0
// update accumulator
ACC[0:63] ← d[0:63]
// update SPEFSCR
SPEFSCR[OVH] ← ovh
SPEFSCR[OV] ← ovl
SPEFSCR[SOVH] ← SPEFSCR[SOVH] | ovh
SPEFSCR[SOV] ← SPEFSCR[SOV] | ovl
```

For each word element in the accumulator, corresponding even-numbered half word elements in parameters a and b are multiplied. Each product is added to the contents of the corresponding accumulator words. If a sum overflows, 0xFFFF_FFFF is placed into the corresponding parameter d and accumulator words. Otherwise, the intermediate sums are placed there.

Overflow information is recorded in SPEFSCR overflow and summary overflow bits.

Other registers altered: SPEFSCR ACC

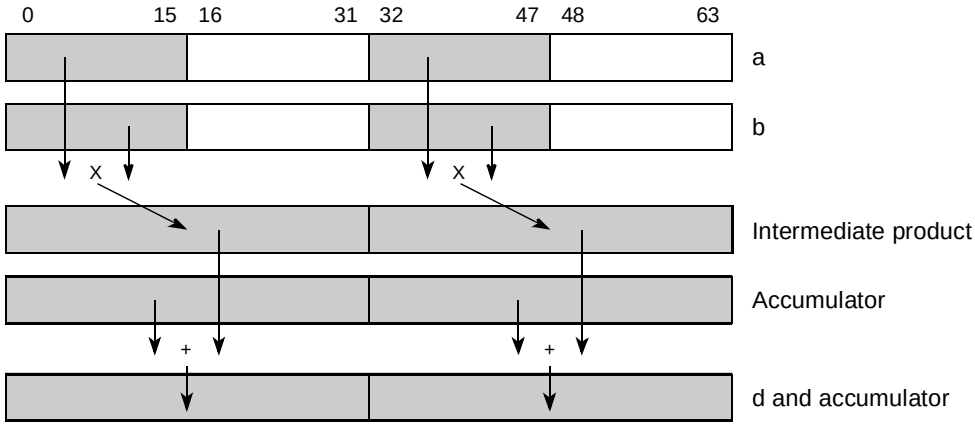

Figure 3-121. Even Form of Vector Half-Word Multiply (__ev_mheusfaaw)

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmheusiaaw d,a,b**

__ev_mheusiaaw __ev_mheusiaaw

Vector Multiply Half Words, Even, Unsigned, Saturate, Integer and Accumulate into Words

d = __ev_mheusiaaw (a,b)

```
// high
temp0:31 ← a0:15 ×ui b0:15
temp0:63 ← EXTZ(ACC0:31) + EXTZ(temp0:31)
ovh ← temp31
d0:31 ← SATURATE(ovh, 0, 0xFFFF_FFFF, 0xFFFF_FFFF, temp32:63)

//low
temp0:31 ← a32:47 ×ui b32:47
temp0:63 ← EXTZ(ACC32:63) + EXTZ(temp0:31)
ovl ← temp31
d32:63 ← SATURATE(ovl, 0, 0xFFFF_FFFF, 0xFFFF_FFFF, temp32:63)

// update accumulator
ACC0:63 ← d0:63

// update SPEFSCR
SPEFSCROVH ← ovh
SPEFSCROV ← ovl
SPEFSCRSOVH ← SPEFSCRSOVH | ovh
SPEFSCRSOV ← SPEFSCRSOV | ovl
```

For each word element in the accumulator, corresponding even-numbered half-word unsigned integer elements in parameters a and b are multiplied, producing a 32-bit product. Each 32-bit product is then added to the corresponding word in the accumulator, saturating if overflow occurs, and the result is placed in parameter d and the accumulator.

If there is an overflow from the addition, the overflow and summary overflow bits are recorded in the SPEFSCR.

Other registers altered: SPEFSCR ACC

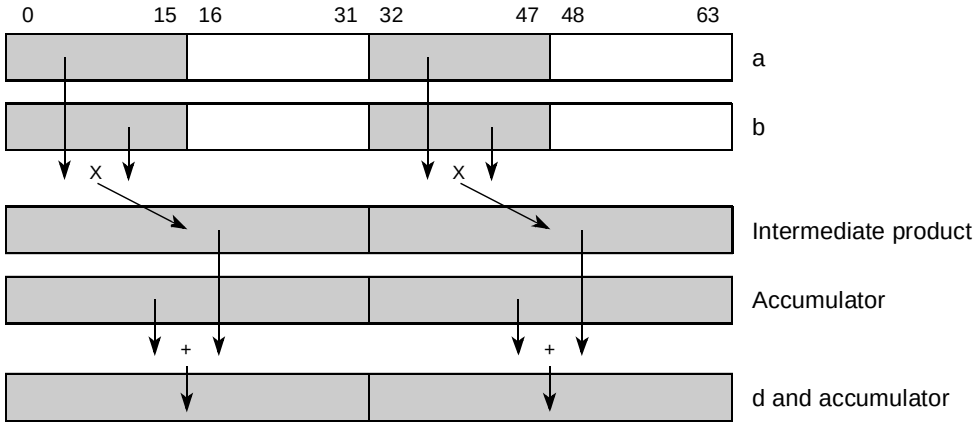

Figure 3-122. Even Form of Vector Half-Word Multiply (__ev_mheusiaaw)

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmheusiaaw d,a,b**

__ev_mheusfanw __ev_mheusfanw

Vector Multiply Half Words, Even, Unsigned, Saturate, Fractional and Accumulate Negative into Words

d = __ev_mheusfanw (a,b)

```
// high
temp0[0:31] ← a[0:15] ×ui b[0:15]
temp0[0:63] ← EXTZ(ACC[0:31]) - EXTZ(temp0[0:31])
if temp0[31] = 1
    d[0:31] ← 0xFFFF_FFFF //overflow
    ovh ← 1
else
    d[0:31] ← temp0[32:63]
    ovh ← 0
//low
temp1[0:31] ← a[32:47] ×ui b[32:47]
temp1[0:63] ← EXTZ(ACC[32:63]) - EXTZ(temp1[0:31])
if temp1[31] = 1
    d[32:63] ← 0xFFFF_FFFF //overflow
    ovl ← 1
else
    d[32:63] ← temp1[32:63]
    ovl ← 0
// update accumulator
ACC[0:63] ← d[0:63]
// update SPEFSCR
SPEFSCR[OVH] ← ovh
SPEFSCR[OV] ← ovl
SPEFSCR[SOVH] ← SPEFSCR[SOVH] | ovh
SPEFSCR[SOV] ← SPEFSCR[SOV] | ovl
```

For each word element in the accumulator, corresponding even-numbered half word elements in parameters a and b are multiplied. Each product is subtracted from the contents of the corresponding accumulator words. If a result overflows, 0xFFFF_FFFF is placed into the corresponding parameter d and accumulator words. Otherwise, the intermediate results are placed there.

Overflow information is recorded in SPEFSCR overflow and summary overflow bits.

Other registers altered: SPEFSCR ACC

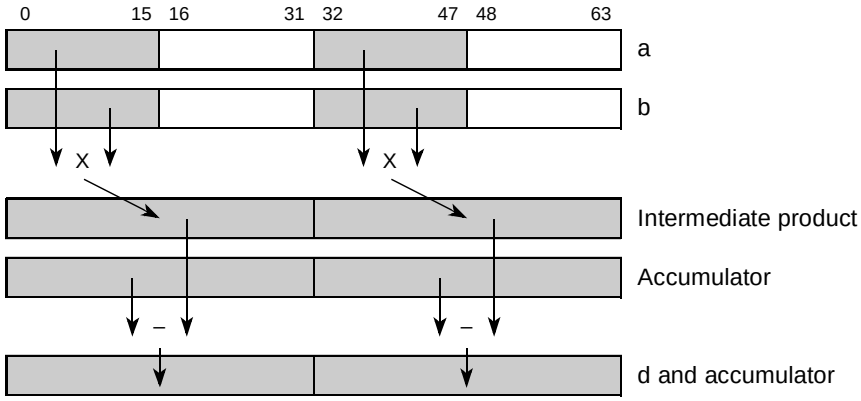

Figure 3-123. Even Form of Vector Half-Word Multiply (__ev_mheusfanw)

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmheusianw d,a,b**

__ev_mheusianw __ev_mheusianw

Vector Multiply Half Words, Even, Unsigned, Saturate, Integer and Accumulate Negative into Words

d = __ev_mheusianw (a,b)

```
// high
temp0:31 ← a0:15 ×ui b0:15
temp0:63 ← EXTZ(ACC0:31) - EXTZ(temp0:31)
ovh ← temp31
d0:31 ← SATURATE(ovh, 0, 0x0000_0000, 0x0000_0000, temp32:63)

//low
temp0:31 ← a32:47 ×ui b32:47
temp0:63 ← EXTZ(ACC32:63) - EXTZ(temp0:31)
ovl ← temp31
d32:63 ← SATURATE(ovl, 0, 0x0000_0000, 0x0000_0000, temp32:63)

// update accumulator
ACC0:63 ← d0:63

// update SPEFSCR
SPEFSCROVH ← ovh
SPEFSCROV ← ovl
SPEFSCRSOVH ← SPEFSCRSOVH | ovh
SPEFSCRSOV ← SPEFSCRSOV | ovl
```

For each word element in the accumulator, corresponding even-numbered half-word unsigned integer elements in parameters a and b are multiplied, producing a 32-bit product. Each 32-bit product is then subtracted from the corresponding word in the accumulator, saturating if underflow occurs, and the result is placed in parameter d and the accumulator.

If there is an underflow from the subtraction, the overflow and summary overflow bits are recorded in the SPEFSCR.

Other registers altered: SPEFSCR ACC

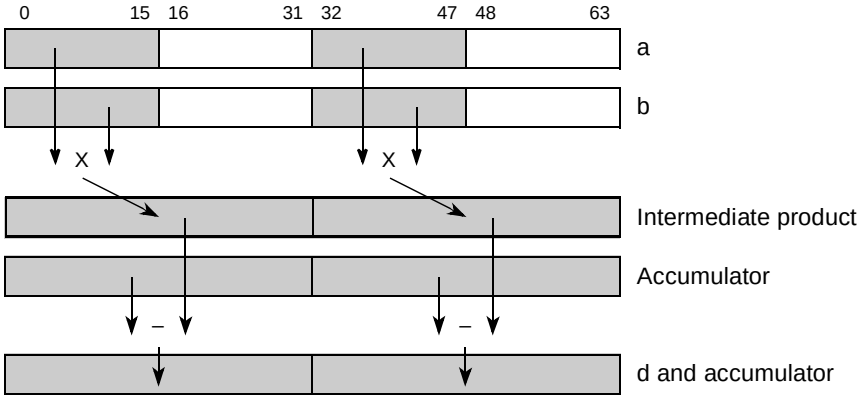

Figure 3-124. Even Form of Vector Half-Word Multiply (__ev_mheusianw)

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmheusianw d,a,b**

__ev_mhogsmfaa __ev_mhogsmfaa

Vector Multiply Half Words, Odd, Guarded, Signed, Modulo, Fractional and Accumulate

d = __ev_mhogsmfaa (a,b)

```
temp0:31 ← a48:63 ×sf b48:63
temp0:63 ← EXTS(temp0:31)
d0:63 ← ACC0:63 + temp0:63
// update accumulator
ACC0:63 ← d0:63
```

The corresponding low odd-numbered half-word signed fractional elements in parameters a and b are multiplied. The intermediate product is sign-extended to 64 bits and added to the contents of the 64-bit accumulator. This result is placed into parameter d and into the accumulator.

NOTE

This sum is a modulo sum. Neither overflow check nor saturation is performed. If an overflow from the 64-bit sum occurs, it is not recorded into the SPEFSCR.

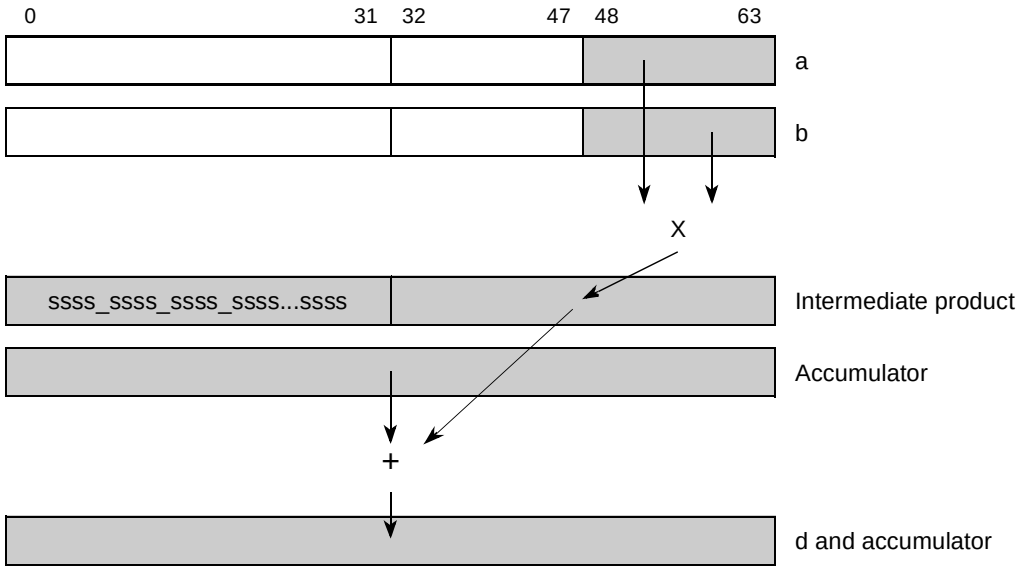

Figure 3-125. __ev_mhogsmfaa (Odd Form)

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmhogsmfaa d,a,b**

__ev_mhogsmfan __ev_mhogsmfan

Vector Multiply Half Words, Odd, Guarded, Signed, Modulo, Fractional and Accumulate Negative

d = __ev_mhogsmfan (a,b)

```
temp0:31 ← a48:63 ×sf b48:63
temp0:63 ← EXTS(temp0:31)
d0:63 ← ACC0:63 - temp0:63
// update accumulator
ACC0:63 ← d0:63
```

The corresponding low odd-numbered half-word signed fractional elements in parameters a and b are multiplied. The intermediate product is sign-extended to 64 bits and subtracted from the contents of the 64-bit accumulator. This result is placed into parameter d and into the accumulator.

NOTE

This difference is a modulo difference. Neither overflow check nor saturation is performed. Any overflow of the 64-bit difference is not recorded into the SPEFSCR.

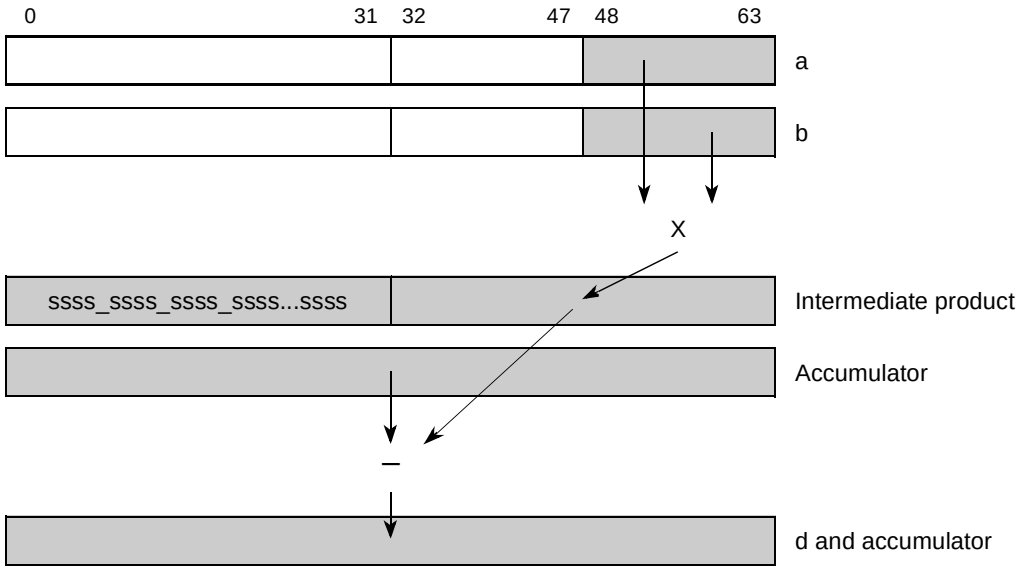

Figure 3-126. __ev_mhogsmfan (Odd Form)

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmhogsmfan d,a,b**

__ev_mhogsmiaa __ev_mhogsmiaa

Vector Multiply Half Words, Odd, Guarded, Signed, Modulo, Intege and Accumulate

d = __ev_mhogsmiaa (a,b)

```
temp0:31 ← a48:63 ×si b48:63
temp0:63 ← EXTS(temp0:31)
d0:63 ← ACC0:63 + temp0:63
// update accumulator
ACC0:63 ← d0:63
```

The corresponding low odd-numbered half-word signed integer elements in parameters a and b are multiplied. The intermediate product is sign-extended to 64 bits and added to the contents of the 64-bit accumulator. This sum is placed into parameter d and into the accumulator.

NOTE

This sum is a modulo sum. Neither overflow check nor saturation is performed. An overflow from the 64-bit sum, if one occurs, is not recorded into the SPEFSCR.

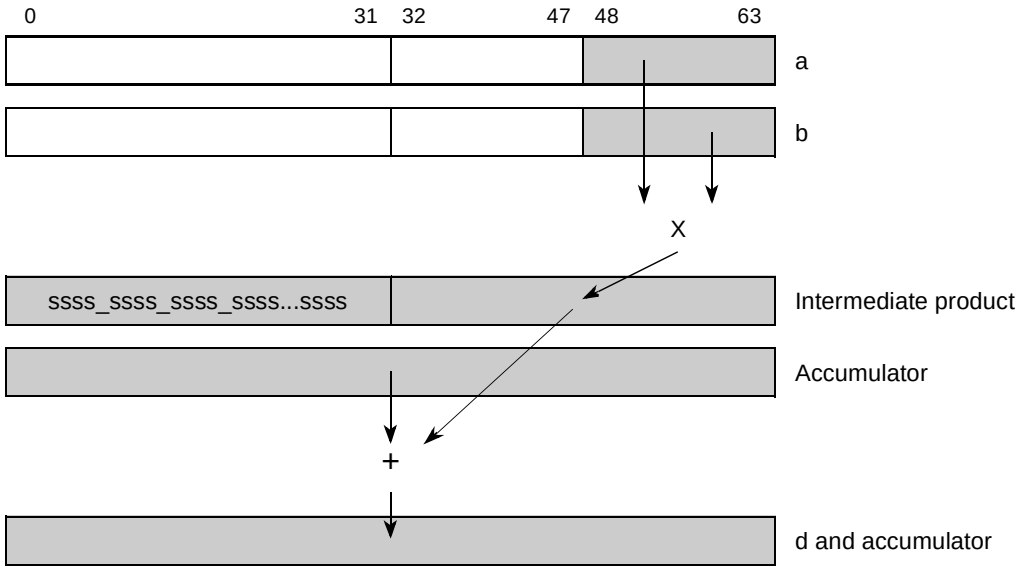

Figure 3-127. __ev_mhogsmiaa (Odd Form)

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmhogsmiaa d,a,b**

__ev_mhogsmian __ev_mhogsmian

Vector Multiply Half Words, Odd, Guarded, Signed, Modulo, Integer and Accumulate Negative

d = __ev_mhogsmian (a,b)

```
temp0:31 ← a48:63 ×si b48:63
temp0:63 ← EXTS(temp0:31)
d0:63 ← ACC0:63 - temp0:63
// update accumulator
ACC0:63 ← d0:63
```

The corresponding low odd-numbered half-word signed integer elements in parameters a and b are multiplied. The intermediate product is sign-extended to 64 bits and subtracted from the contents of the 64-bit accumulator. This result is placed into parameter d and into the accumulator.

NOTE

This difference is a modulo difference. Neither overflow check nor saturation is performed. Any overflow of the 64-bit difference is not recorded into the SPEFSCR.

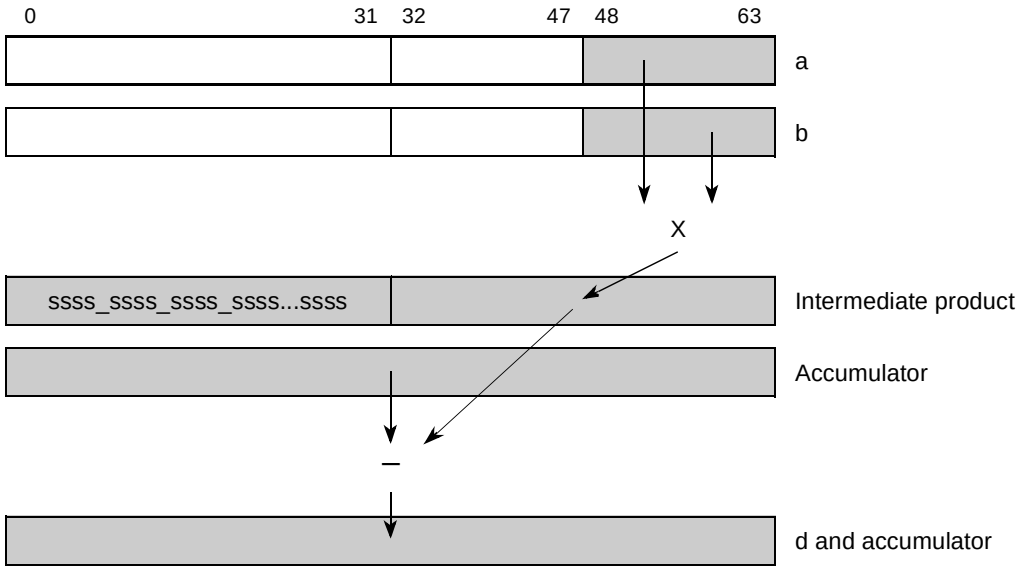

Figure 3-128. __ev_mhogsmian (Odd Form)

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmhogsmian d,a,b**

__ev_mhogumfaa __ev_mhogumfaa

Vector Multiply Half Words, Odd, Guarded, Unsigned, Modulo, Fractional and Accumulate

d = __ev_mhogumfaa (a,b)

```
temp0:31 ← a48:63 ×ui b48:63
temp0:63 ← EXTZ(temp0:31)
d0:63 ← ACC0:63 + temp0:63
// update accumulator
ACC0:63 ← d0:63
```

The corresponding low odd-numbered half word elements in parameters a and b are multiplied. The intermediate product is zero-extended to 64 bits and added to the contents of the 64-bit accumulator. This sum is placed into parameter d and into the accumulator.

NOTE

This sum is a modulo sum. Neither overflow check nor saturation is performed. An overflow from the 64-bit sum, if one occurs, is not recorded into the SPEFSCR.

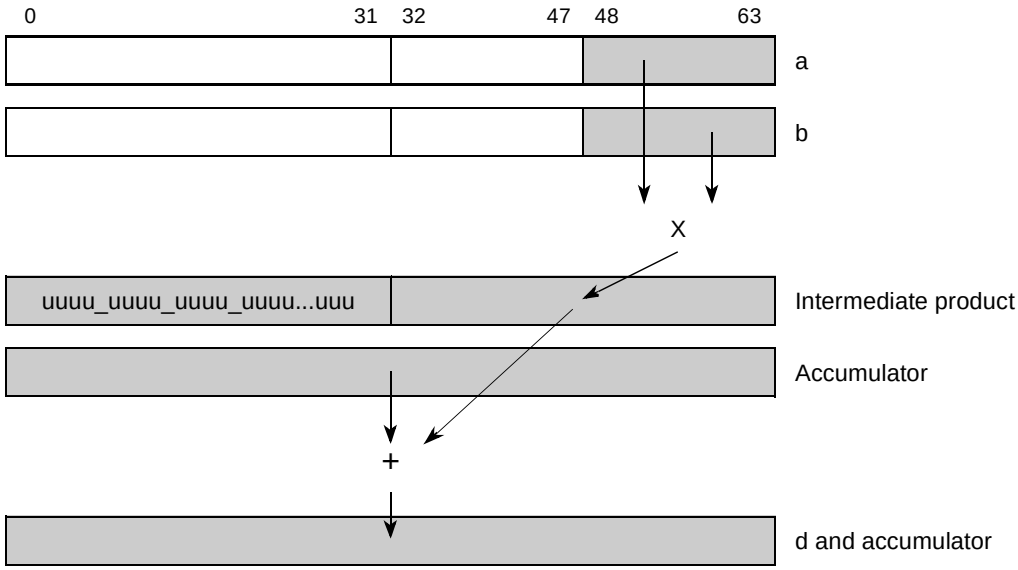

Figure 3-129. __ev_mhogumfaa (Odd Form)

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmhogumiaa d,a,b**

__ev_mhogumiaa __ev_mhogumiaa

Vector Multiply Half Words, Odd, Guarded, Unsigned, Modulo, Integer and Accumulate

d = __ev_mhogumiaa (a,b)

```
temp0:31 ← a48:63 ×ui b48:63
temp0:63 ← EXTZ(temp0:31)
d0:63 ← ACC0:63 + temp0:63
// update accumulator
ACC0:63 ← d0:63
```

The corresponding low odd-numbered half-word unsigned integer elements in parameters a and b are multiplied. The intermediate product is zero-extended to 64 bits and added to the contents of the 64-bit accumulator. This sum is placed into parameter d and into the accumulator.

NOTE

This sum is a modulo sum. Neither overflow check nor saturation is performed. An overflow from the 64-bit sum, if one occurs, is not recorded into the SPEFSCR.

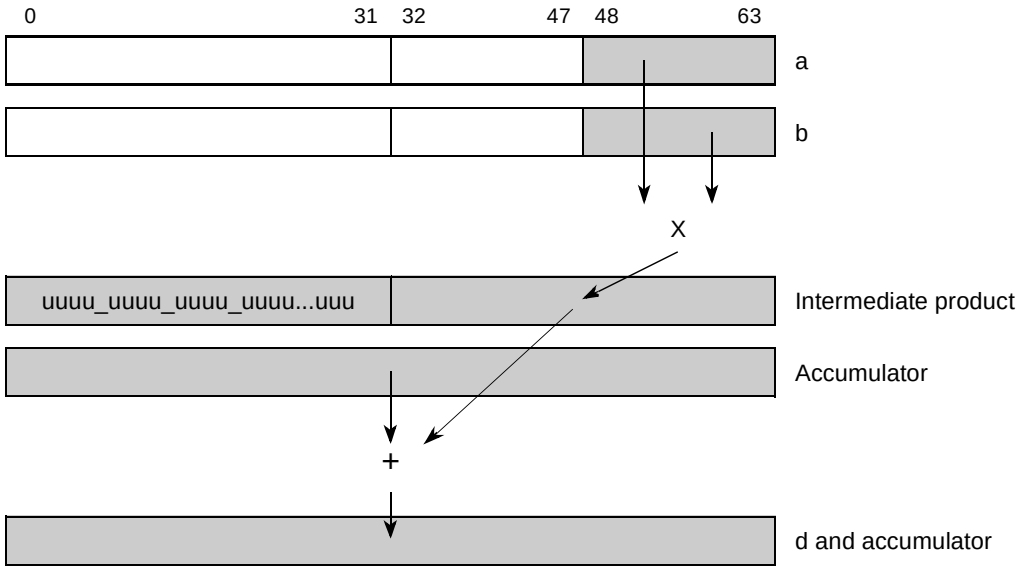

Figure 3-130. __ev_mhogumiaa (Odd Form)

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmhogumiaa d,a,b**

__ev_mhogumfan __ev_mhogumfan

Vector Multiply Half Words, Odd, Guarded, Unsigned, Modulo, Fractional and Accumulate Negative

d = __ev_mhogumfan (a,b)

```
temp0:31 ← a48:63 ×ui b48:63
temp0:63 ← EXTZ(temp0:31)
d0:63 ← ACC0:63 - temp0:63
// update accumulator
ACC0:63 ← d0:63
```

The corresponding low odd-numbered half word elements in parameters a and b are multiplied. The intermediate product is zero-extended to 64 bits and subtracted from the contents of the 64-bit accumulator. This result is placed into parameter d and into the accumulator.

NOTE

This difference is a modulo difference. Neither overflow check nor saturation is performed. Overflow of the 64-bit difference is not recorded into the SPEFSCR.

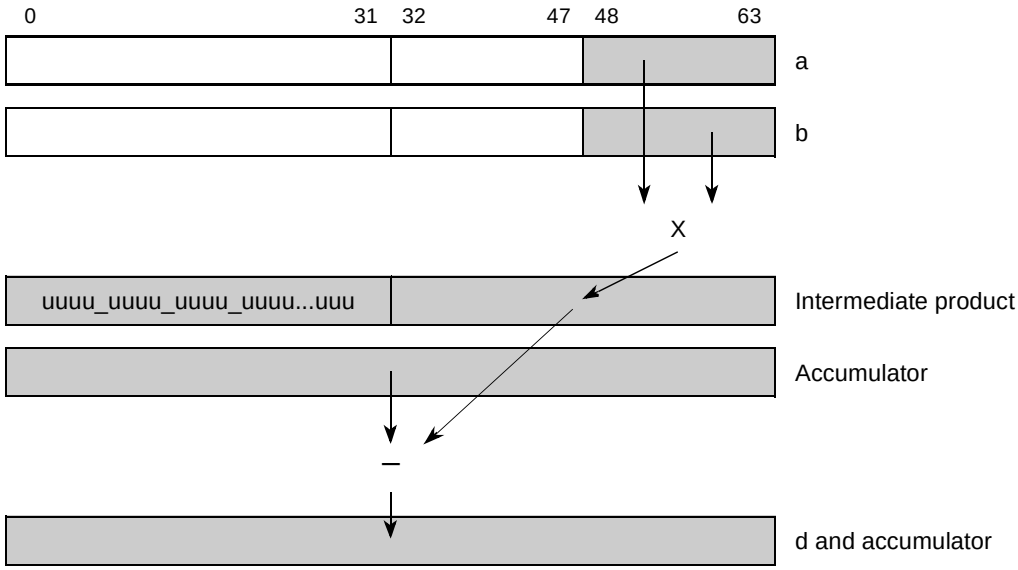

Figure 3-131. __ev_mhogumfan (Odd Form)

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmhogumian d,a,b**

__ev_mhogumian __ev_mhogumian

Vector Multiply Half Words, Odd, Guarded, Unsigned, Modulo, Integer and Accumulate Negative

d = __ev_mhogumian (a,b)

```
temp0:31 ← a48:63 ×ui b48:63
temp0:63 ← EXTZ(temp0:31)
d0:63 ← ACC0:63 - temp0:63
// update accumulator
ACC0:63 ← d0:63
```

The corresponding low odd-numbered half-word unsigned integer elements in parameters a and b are multiplied. The intermediate product is zero-extended to 64 bits and subtracted from the contents of the 64-bit accumulator. This result is placed into parameter d and into the accumulator.

NOTE

This difference is a modulo difference. Neither overflow check nor saturation is performed. Any overflow of the 64-bit difference is not recorded into the SPEFSCR.

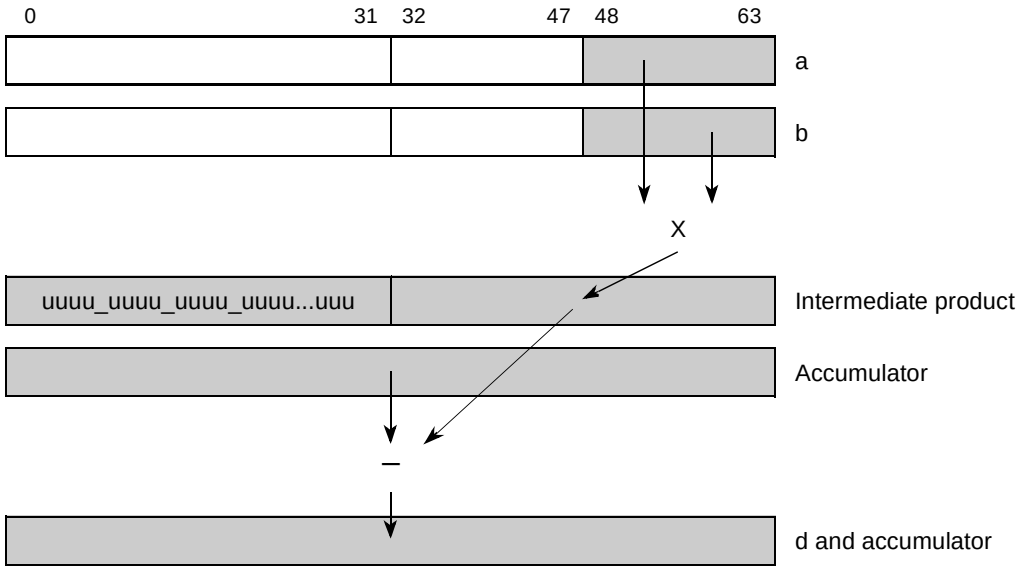

Figure 3-132. __ev_mhogumian (Odd Form)

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmhogumian d,a,b**

__ev_mhosmf __ev_mhosmf

Vector Multiply Half Words, Odd, Signed, Modulo, Fractional (to Accumulator)

d = __ev_mhosmf (a,b) (A = 0)
d = __ev_mhosmfa (a,b) (A = 1)

```
// high
d0:31 ← a16:31 ×sf b16:31
// low
d32:63 ← a48:63 ×sf b48:63
// update accumulator
if A = 1, then ACC0:63 ← d0:63
```

The corresponding odd-numbered, half-word signed fractional elements in parameters a and b are multiplied. Each product is placed into the corresponding words of parameter d.

If A = 1, the result in parameter d is also placed into the accumulator.

Other registers altered: ACC (if A = 1)

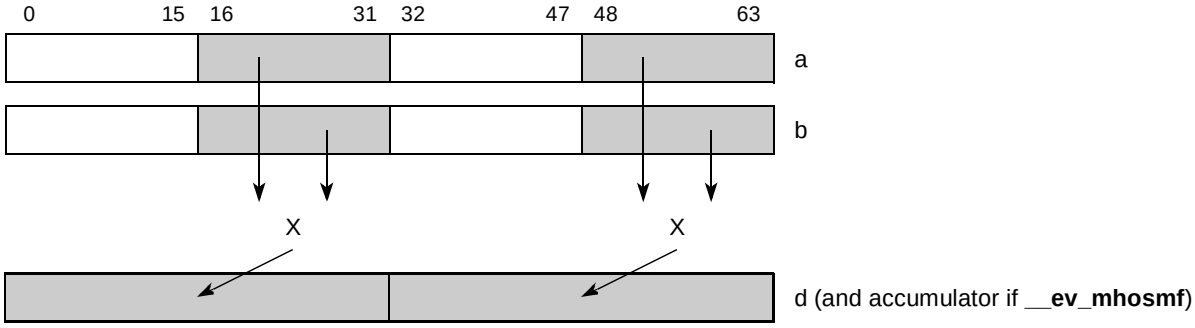

Figure 3-133. Vector Multiply Half Words, Odd, Signed, Modulo, Fractional (to Accumulator) (__ev_mhosmf)

A	d	a	b	Maps to
A = 0	__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmhosmf d,a,b**
A = 1	__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmhosmfa d,a,b**

__ev_mhosmfaaw __ev_mhosmfaaw

Vector Multiply Half Words, Odd, Signed, Modulo, Fractional and Accumulate into Words

d = __ev_mhosmfaaw (a,b)

```
// high
temp0:31 ← a16:31 ×sf b16:31
d0:31 ← ACC0:31 + temp0:31

// low
temp0:31 ← a48:63 ×sf b48:63
d32:63 ← ACC32:63 + temp0:31

// update accumulator
ACC0:63 ← d0:63
```

For each word element in the accumulator, the corresponding odd-numbered half-word signed fractional elements in parameters a and b are multiplied. The 32 bits of each intermediate product is added to the contents of the corresponding accumulator word, and the results are placed into the corresponding parameter d words and into the accumulator

Other registers altered: ACC

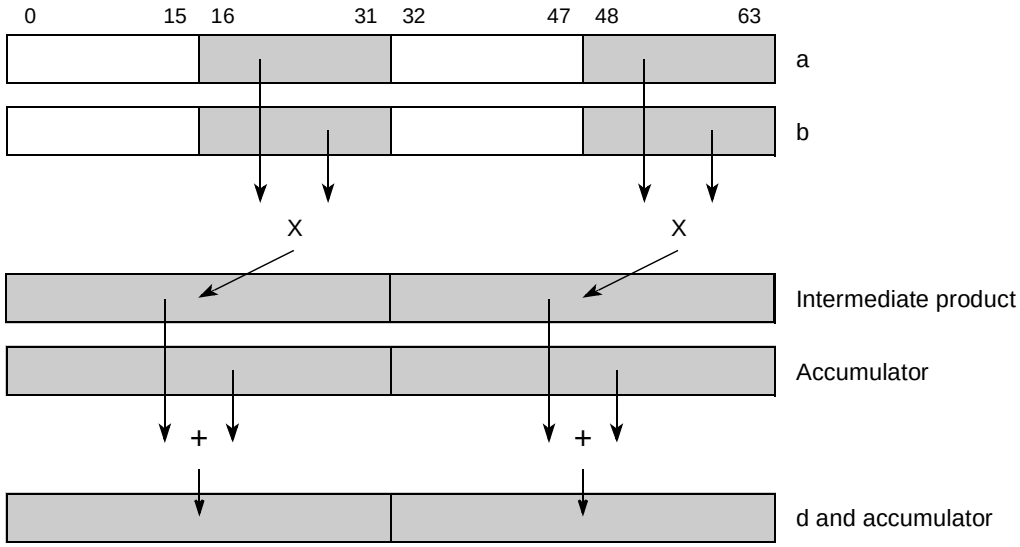

Figure 3-134. Odd Form of Vector Half-Word Multiply (__ev_mhosmfaaw)

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmhosmfaaw d,a,b**

__ev_mhosmfanw __ev_mhosmfanw

Vector Multiply Half Words, Odd, Signed, Modulo, Fractional and Accumulate Negative into Words

d = __ev_mhosmfanw (a,b)

```
// high
temp0:31 ← a16:31 ×sf b16:31
d0:31 ← ACC0:31 - temp0:31

// low
temp0:31 ← a48:63 ×sf b48:63
d32:63 ← ACC32:63 - temp0:31

// update accumulator
ACC0:63 ← d0:63
```

For each word element in the accumulator, the corresponding odd-numbered half-word signed fractional elements in parameters a and b are multiplied. The 32 bits of each intermediate product is subtracted from the contents of the corresponding accumulator word. The word and the results are placed into the corresponding parameter d word and into the accumulator.

Other registers altered: ACC

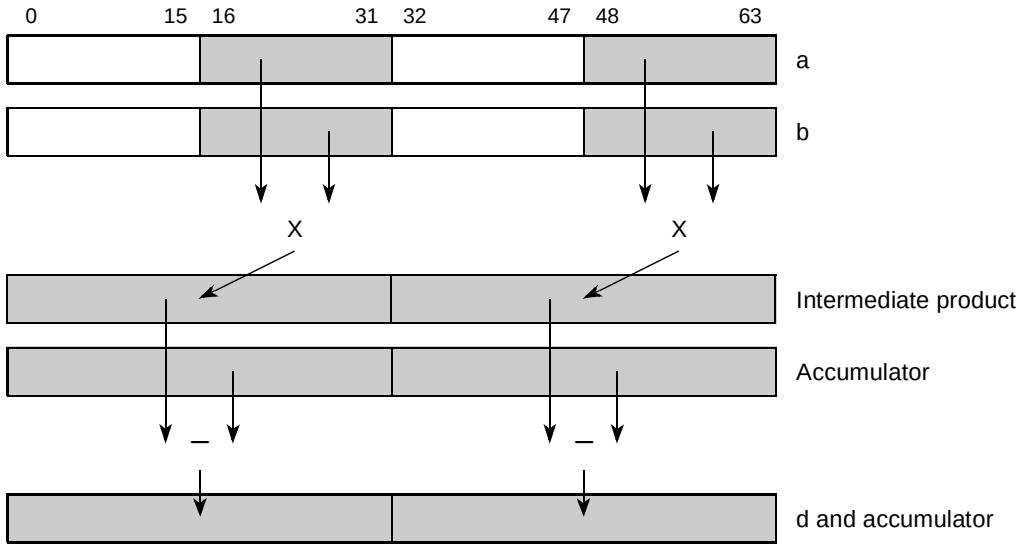

Figure 3-135. Odd Form of Vector Half-Word Multiply (__ev_mhosmfanw)

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmhosmfanw d,a,b**

__ev_mhosmi __ev_mhosmi

Vector Multiply Half Words, Odd, Signed, Modulo, Integer (to Accumulator)

d = __ev_mhosmi (a,b) (A = 0)
d = __ev_mhosmia (a,b) (A = 1)

```
// high
d0:31 ← a16:31 ×si b16:31
// low
d32:63 ← a48:63 ×si b48:63
// update accumulator
if A = 1, then ACC0:63 ← d0:63
```

The corresponding odd-numbered half-word signed integer elements in parameters a and b are multiplied. The two 32-bit products are placed into the corresponding words of parameter d.

If A = 1, the result in parameter d is also placed into the accumulator.

Other registers altered: ACC (if A = 1)

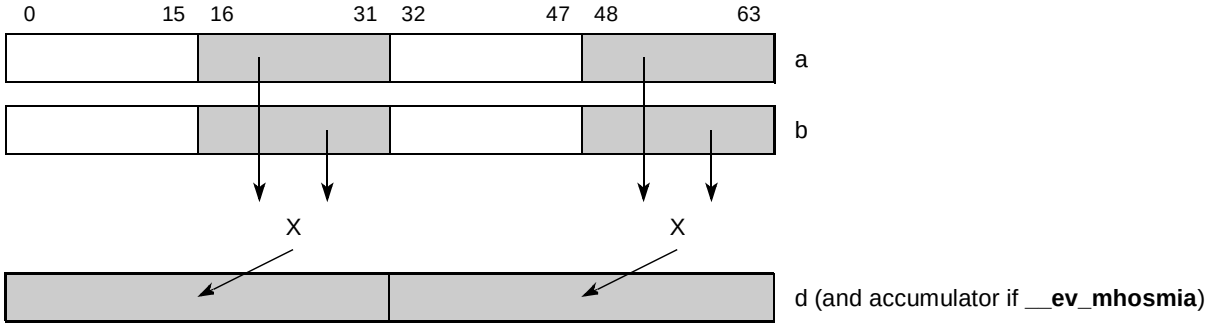

Figure 3-136. Vector Multiply Half Words, Odd, Signed, Modulo, Integer (to Accumulator) (__ev_mhosmi)

A	d	a	b	Maps to
A = 0	__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmhosmi d,a,b**
A = 1	__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmhosmia d,a,b**

__ev_mhosmiaaw __ev_mhosmiaaw

Vector Multiply Half Words, Odd, Signed, Modulo, Integer and Accumulate into Words

d = __ev_mhosmiaaw (a,b)

```
// high
temp0:31 ← a16:31 ×si b16:31
d0:31 ← ACC0:31 + temp0:31

// low
temp0:31 ← a48:63 ×si b48:63
d32:63 ← ACC32:63 + temp0:31

// update accumulator
ACC0:63 ← d0:63
```

For each word element in the accumulator, the corresponding odd-numbered half-word signed integer elements in parameters a and b are multiplied. Each intermediate 32-bit product is added to the contents of the corresponding accumulator word and the results are placed into the corresponding parameter d words and into the accumulator.

Other registers altered: ACC

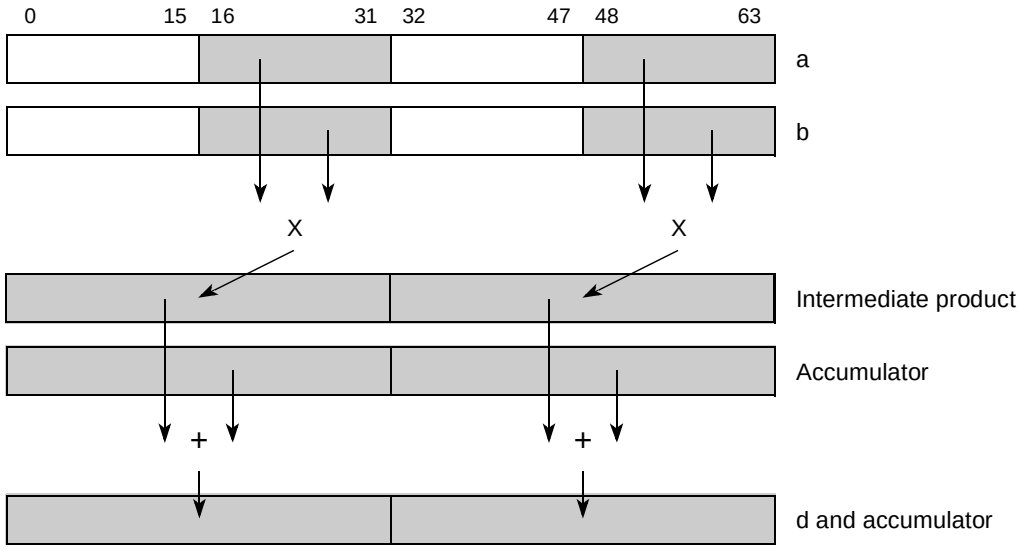

Figure 3-137. Odd Form of Vector Half-Word Multiply (__ev_mhosmiaaw)

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmhosmiaaw d,a,b**

__ev_mhosmianw __ev_mhosmianw

Vector Multiply Half Words, Odd, Signed, Modulo, Integer and Accumulate Negative into Words

d = __ev_mhosmianw (a,b)

```
// high
temp0:31 ←a16:31 ×si b16:31
d0:31 ← ACC0:31 - temp0:31

// low
temp0:31 ← a48:63 ×si b48:63
d32:63 ← ACC32:63 - temp0:31

// update accumulator
ACC0:63 ← d0:63
```

For each word element in the accumulator, the corresponding odd-numbered half-word signed integer elements in parameters a and b are multiplied. Each intermediate 32-bit product is subtracted from the contents of the corresponding accumulator word and the results are placed into the corresponding parameter d words and into the accumulator.

Other registers altered: ACC

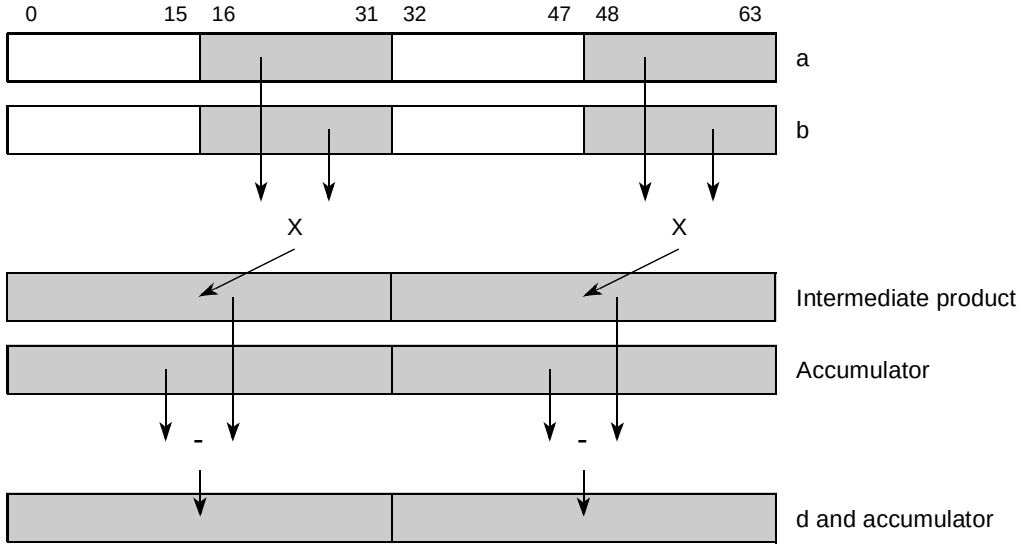

Figure 3-138. Odd Form of Vector Half-Word Multiply (__ev_mhosmianw)

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmhosmianw d,a,b**

__ev_mhossf __ev_mhossf

Vector Multiply Half Words, Odd, Signed, Saturate, Fractional (to Accumulator)

d = __ev_mhossf (a,b) (A = 0)
d = __ev_mhossfa (a,b) (A = 1)

```
// high
temp0:31 ← a16:31 ×sf b16:31
if (a16:31 = 0x8000) & (b16:31 = 0x8000) then
    d0:31 ← 0x7FFF_FFFF //saturate
    movh ← 1
else
    d0:31 ← temp0:31
    movh ← 0

// low
temp0:31 ← a48:63 ×sf b48:63
if (a48:63 = 0x8000) & (b48:63 = 0x8000) then
    d32:63 ← 0x7FFF_FFFF //saturate
    movl ← 1
else
    d32:63 ← temp0:31
    movl ← 0

// update accumulator
if A = 1 then ACC0:63 ← d0:63

// update SPEFSCR
SPEFSCROVH ← movh
SPEFSCROV ← movl
SPEFSCRSOVH ← SPEFSCRSOVH | movh
SPEFSCRSOV ← SPEFSCRSOV | movl
```

The corresponding odd-numbered half-word signed fractional elements in parameters a and b are multiplied. The 32 bits of each product are placed into the corresponding words of parameter d. If both inputs are -1.0, the result saturates to the largest positive signed fraction and the overflow and summary overflow bits are recorded in the SPEFSCR.

If A = 1, the result in parameter d is also placed into the accumulator.

Other registers altered: SPEFSCR
ACC (if A = 1)

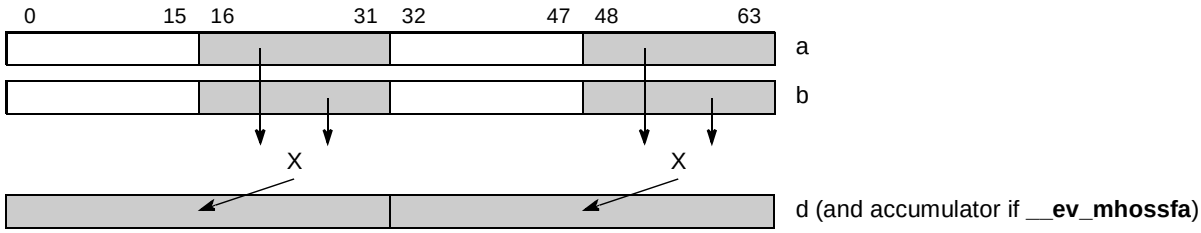

Figure 3-139. Vector Multiply Half Words, Odd, Signed, Saturate, Fractional (to Accumulator) (__ev_mhossf)

A	d	a	b	Maps to
A = 0	__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmhossf d,a,b**
A = 1	__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmhossfa d,a,b**

__ev_mhossfaaw __ev_mhossfaaw

Vector Multiply Half Words, Odd, Signed, Saturate, Fractional and Accumulate into Words

d = __ev_mhossfaaw (a,b)

```
// high
temp0:31 ← a16:31 ×sf b16:31
if (a16:31 = 0x8000) & (b16:31 = 0x8000) then
    temp0:31 ← 0x7FFF_FFFF //saturate
    movh ← 1
else
    movh ← 0
temp0:63 ← EXTS(ACC0:31) + EXTS(temp0:31)
ovh ← (temp31 ⊕ temp32)
d0:31 ← SATURATE(ovh, temp31, 0x8000_0000, 0x7FFF_FFFF, temp32:63)

// low
temp0:31 ← a48:63 ×sf b48:63
if (a48:63 = 0x8000) & (b48:63 = 0x8000) then
    temp0:31 ← 0x7FFF_FFFF //saturate
    movl ← 1
else
    movl ← 0
temp0:63 ← EXTS(ACC32:63) + EXTS(temp0:31)
ovl ← (temp31 ⊕ temp32)
d32:63 ← SATURATE(ovl, temp31, 0x8000_0000, 0x7FFF_FFFF, temp32:63)

// update accumulator
ACC0:63 ← d0:63

// update SPEFSCR
SPEFSCROVH ← movh
SPEFSCROV ← movl
SPEFSCRSOVH ← SPEFSCRSOVH | ovh | movh
SPEFSCRSOV ← SPEFSCRSOV | ovl| movl
```

The corresponding odd-numbered half-word signed fractional elements in parameters a and b are multiplied, producing a 32-bit product. If both inputs are -1.0, the result saturates to 0x7FFF_FFFF. Each 32-bit product is then added to the corresponding word in the accumulator, saturating if overflow or underflow occurs, and the result is placed in parameter d and the accumulator.

If there is an overflow or underflow from either the multiply or the addition, the overflow and summary overflow bits are recorded in the SPEFSCR.

Other registers altered: SPEFSCR ACC

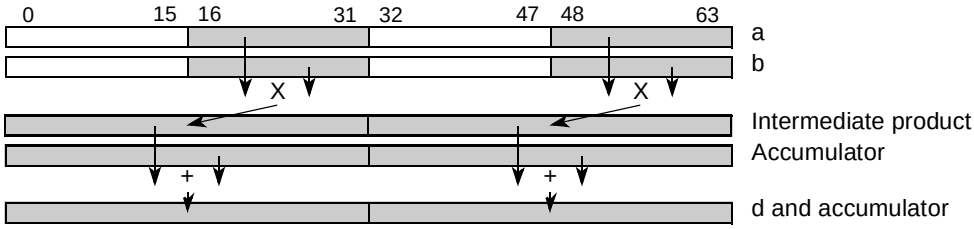

Figure 3-140. Odd Form of Vector Half-Word Multiply (__ev_mhossfaaw)

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmhossfaaw d,a,b**

__ev_mhossfanw __ev_mhossfanw

Vector Multiply Half Words, Odd, Signed, Saturate, Fractional and Accumulate Negative into Words

d = __ev_mhossfanw (a,b)

```
// high
temp0:31 ← a16:31 ×sf b16:31
if (a16:31 = 0x8000) & (b16:31 = 0x8000) then
    temp0:31 ← 0x7FFF_FFFF //saturate
    movh ← 1
else
    movh ← 0
temp0:63 ← EXTS(ACC0:31) - EXTS(temp0:31)
ovh ← (temp31 ⊕ temp32)
d0:31 ← SATURATE(ovh, temp31, 0x8000_0000, 0x7FFF_FFFF, temp32:63)

// low
temp0:31 ← a48:63 ×sf b48:63
if (a48:63 = 0x8000) & (b48:63 = 0x8000) then
    temp0:31 ← 0x7FFF_FFFF //saturate
    movl ← 1
else
    movl ← 0
temp0:63 ← EXTS(ACC32:63) - EXTS(temp0:31)
ovl ← (temp31 ⊕ temp32)
d32:63 ← SATURATE(ovl, temp31, 0x8000_0000, 0x7FFF_FFFF, temp32:63)

// update accumulator
ACC0:63 ← d0:63

// update SPEFSCR
SPEFSCROVH ← movh
SPEFSCROV ← movl
SPEFSCRSOVH ← SPEFSCRSOVH | ovh | movh
SPEFSCRSOV ← SPEFSCRSOV | ovl| movl
```

The corresponding odd-numbered half-word signed fractional elements in parameters a and b are multiplied, producing a 32-bit product. If both inputs are -1.0, the result saturates to 0x7FFF_FFFF. Each 32-bit product is then subtracted from the corresponding word in the accumulator, saturating if overflow or underflow occurs, and the result is placed in parameter d and the accumulator.

If there is an overflow or underflow from either the multiply or the subtraction, the overflow and summary overflow bits are recorded in the SPEFSCR.

Other registers altered: SPEFSCR ACC

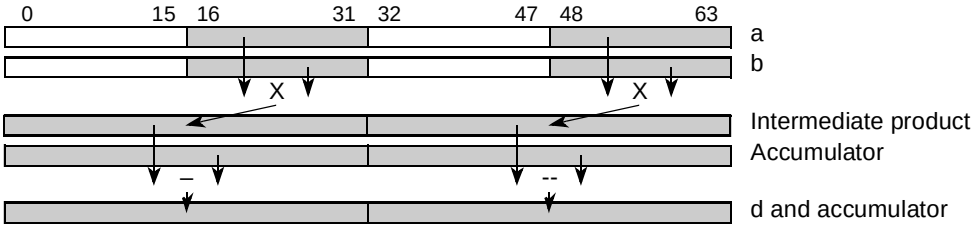

Figure 3-141. Odd Form of Vector Half-Word Multiply (__ev_mhossfanw)

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmhossfanw d,a,b**

__ev_mhossiaaw __ev_mhossiaaw

Vector Multiply Half Words, Odd, Signed, Saturate, Integer and Accumulate into Words

d = __ev_mhossiaaw (a,b)

```
// high
temp0:31 ← a16:31 ×si b16:31
temp0:63 ← EXTS(ACC0:31) + EXTS(temp0:31)
ovh ← (temp31 ⊕ temp32)
d0:31 ← SATURATE(ovh, temp31, 0x8000_0000, 0x7FFF_FFFF, temp32:63)

// low
temp0:31 ← a48:63 ×si b48:63
temp0:63 ← EXTS(ACC32:63) + EXTS(temp0:31)
ovl ← (temp31 ⊕ temp32)
d32:63 ← SATURATE(ovl, temp31, 0x8000_0000, 0x7FFF_FFFF, temp32:63)

// update accumulator
ACC0:63 ← d0:63

// update SPEFSCR
SPEFSCROVH ← ovh
SPEFSCROV ← ovl
SPEFSCRSOVH ← SPEFSCRSOVH | ovh
SPEFSCRSOV ← SPEFSCRSOV | ovl
```

The corresponding odd-numbered half-word signed integer elements in parameters a and b are multiplied, producing a 32-bit product. Each 32-bit product is then added to the corresponding word in the accumulator, saturating if overflow occurs, and the result is placed in parameter d and the accumulator.

If there is an overflow or underflow from the addition, the overflow and summary overflow bits are recorded in the SPEFSCR.

Other registers altered: SPEFSCR ACC

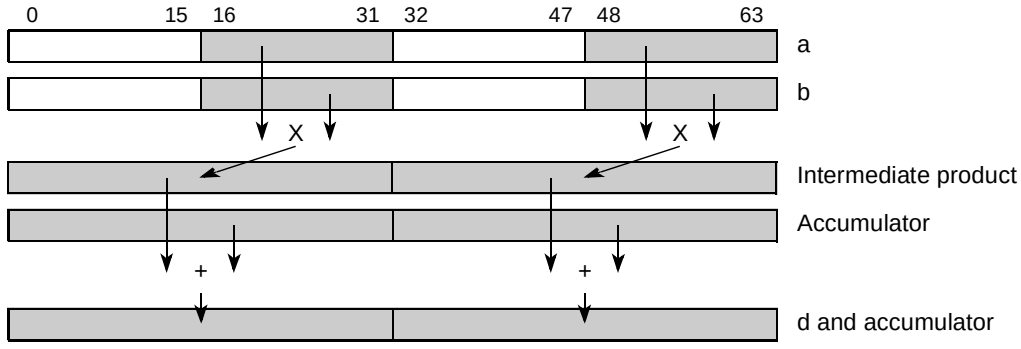

Figure 3-142. Odd Form of Vector Half-Word Multiply (__ev_mhossiaaw)

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmhossiaaw d,a,b**

__ev_mhossianw __ev_mhossianw

Vector Multiply Half Words, Odd, Signed, Saturate, Integer and Accumulate Negative into Words

d = __ev_mhossianw (a,b)

```
// high
temp0:31 ← a16:31 ×si b16:31
temp0:63 ← EXTS(ACC0:31) - EXTS(temp0:31)
ovh ← (temp31 ⊕ temp32)
d0:31 ← SATURATE(ovh, temp31, 0x8000_0000, 0x7FFF_FFFF, temp32:63)

// low
temp0:31 ← a48:63 ×si b48:63
temp0:63 ← EXTS(ACC32:63) - EXTS(temp0:31)
ovl ← (temp31 ⊕ temp32)
d32:63 ← SATURATE(ovl, temp31, 0x8000_0000, 0x7FFF_FFFF, temp32:63)

// update accumulator
ACC0:63 ← d0:63

// update SPEFSCR
SPEFSCROVH ← ovh
SPEFSCROV ← ovl
SPEFSCRSOVH ← SPEFSCRSOVH | ovh
SPEFSCRSOV ← SPEFSCRSOV | ovl
```

The corresponding odd-numbered half-word signed integer elements in parameter a and b are multiplied, producing a 32-bit product. Each 32-bit product is then subtracted from the corresponding word in the accumulator, saturating if overflow occurs, and the result is placed in parameter d and the accumulator.

If there is an overflow or underflow from the subtraction, the overflow and summary overflow bits are recorded in the SPEFSCR.

Other registers altered: SPEFSCR ACC

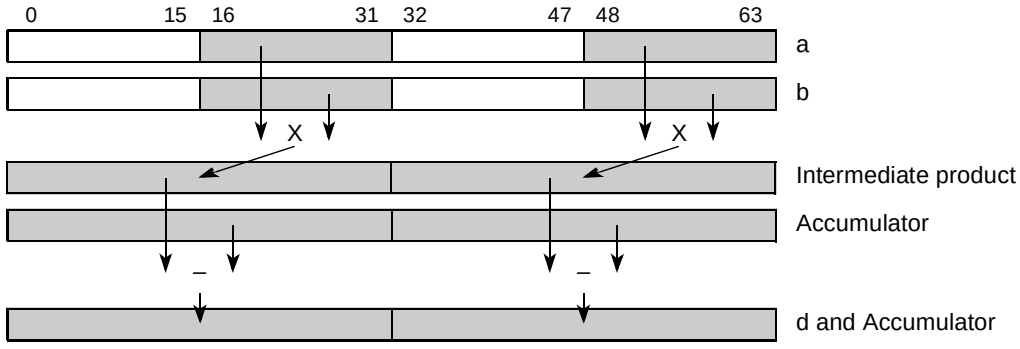

Figure 3-143. Odd Form of Vector Half-Word Multiply (__ev_mhossianw)

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmhossianw d,a,b**

__ev_mhoumf __ev_mhoumf

Vector Multiply Half Words, Odd, Unsigned, Modulo, Fractional (to Accumulator)

d = __ev_mhoumf (a,b) (A = 0)
d = __ev_mhoumfa (a,b) (A = 1)

```
// high
d0:31 ← a16:31 ×ui b16:31
// low
d32:63 ← a48:63 ×ui b48:63
// update accumulator
if A = 1, ACC0:63 ← d0:63
```

The corresponding odd-numbered half-word elements in parameters a and b are multiplied. The two 32-bit products are placed into the corresponding words of parameter d.

If A = 1, the result in parameter d is also placed into the accumulator.

Other registers altered: ACC (if A = 1)

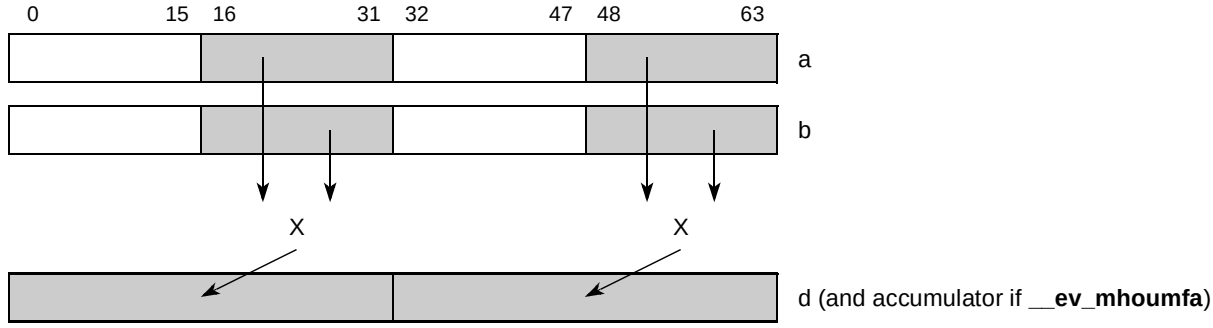

Figure 3-144. Vector Multiply Half Words, Odd, Unsigned, Modulo, Fractional (to Accumulator) (__ev_mhoumf)

A	d	a	b	Maps to
A = 0	__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmhoumi d,a,b**
A = 1	__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmhoumia d,a,b**

__ev_mhoumi __ev_mhoumi

Vector Multiply Half Words, Odd, Unsigned, Modulo, Integer (to Accumulator)

d = __ev_mhoumi (a,b) (A = 0)
d = __ev_mhoumia (a,b) (A = 1)

```
// high
d0:31 ← a16:31 ×ui b16:31
// low
d32:63 ← a48:63 ×ui b48:63
// update accumulator
if A = 1, then ACC0:63 ← d0:63
```

The corresponding odd-numbered half-word unsigned integer elements in parameters a and b are multiplied. The two 32-bit products are placed into the corresponding words of parameter d.

If A = 1, the result in parameter d is also placed into the accumulator.

Other registers altered: ACC (if A = 1)

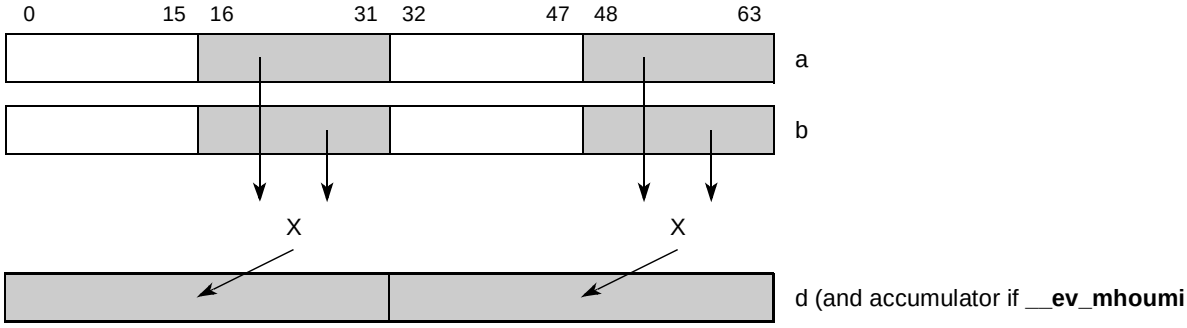

Figure 3-145. Vector Multiply Half Words, Odd, Unsigned, Modulo, Integer (to Accumulator) (__ev_mhoumi)

A	d	a	b	Maps to
A = 0	__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmhoumi d,a,b**
A = 1	__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmhoumia d,a,b**

__ev_mhoumfaaw __ev_mhoumfaaw

Vector Multiply Half Words, Odd, Unsigned, Modulo, Fractional and Accumulate into Words

d = __ev_mhoumfaaw (a,b)

```
// high
temp0[0:31] ← a[16:31] ×ui b[16:31]
d[0:31] ← ACC[0:31] + temp0[0:31]
// low
temp1[0:31] ← a[48:63] ×ui b[48:63]
d[32:63] ← ACC[32:63] + temp1[0:31]
// update accumulator
ACC[0:63] ← d[0:63]
```

For each word element in the accumulator, the corresponding odd-numbered half-word elements in parameters a and b are multiplied. Each intermediate product is added to the contents of the corresponding accumulator word. The sums are placed into the corresponding parameter d and accumulator words.

Other registers altered: ACC

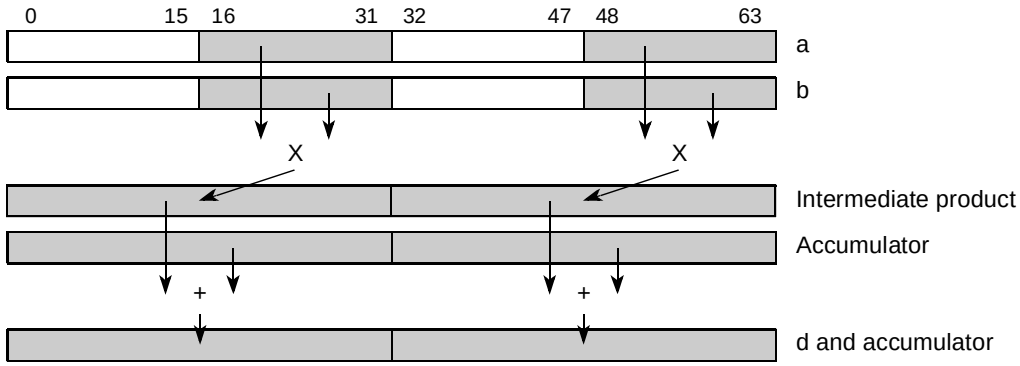

Figure 3-146. Odd Form of Vector Half-Word Multiply (__ev_mhoumfaaw)

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmhoumiaaw d,a,b**

__ev_mhoumiaaw __ev_mhoumiaaw

Vector Multiply Half Words, Odd, Unsigned, Modulo, Integer and Accumulate into Words

d = __ev_mhoumiaaw (a,b)

```
// high
temp0:31 ← a16:31 ×ui b16:31
d0:31 ← ACC0:31 + temp0:31

// low
temp0:31 ← a48:63 ×ui b48:63
d32:63 ← ACC32:63 + temp0:31

// update accumulator
ACC0:63 ← d0:63
```

For each word element in the accumulator, the corresponding odd-numbered half-word unsigned integer elements in parameters a and b are multiplied. Each intermediate product is added to the contents of the corresponding accumulator word. The sums are placed into the corresponding parameter d and accumulator words.

Other registers altered: ACC

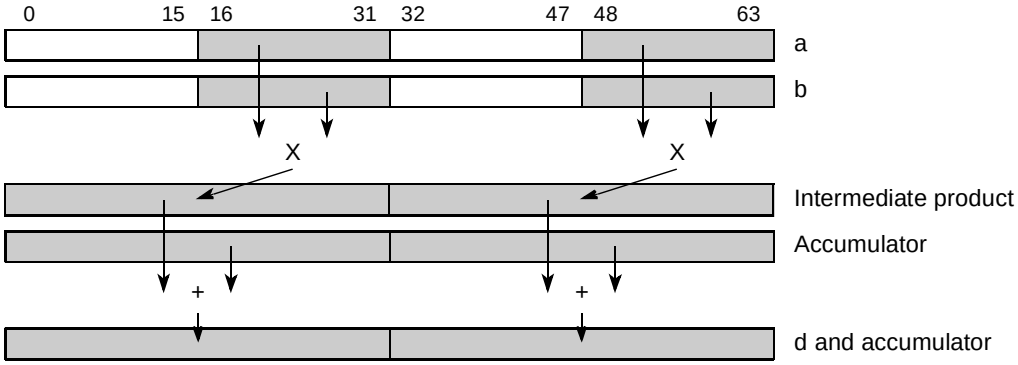

Figure 3-147. Odd Form of Vector Half-Word Multiply (__ev_mhoumiaaw)

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmhoumiaaw d,a,b**

__ev_mhoumfanw __ev_mhoumfanw

Vector Multiply Half Words, Odd, Unsigned, Modulo, Fractional and Accumulate Negative into Words

d = __ev_mhoumfanw (a,b)

```
// high
temp0[0:31] ← a[0:15] ×ui b[0:15]
d[0:31] ← ACC[0:31] - temp0[0:31]
// low
temp1[0:31] ← a[32:47] ×ui b[32:47]
d[32:63] ← ACC[32:63] - temp1[0:31]
// update accumulator
ACC[0:63] ← d[0:63]
```

For each word element in the accumulator, the corresponding odd-numbered half word elements in parameters a and b are multiplied. Each intermediate product is subtracted from the contents of the corresponding accumulator word. The results are placed into the corresponding parameter d and accumulator words.

Other registers altered: ACC

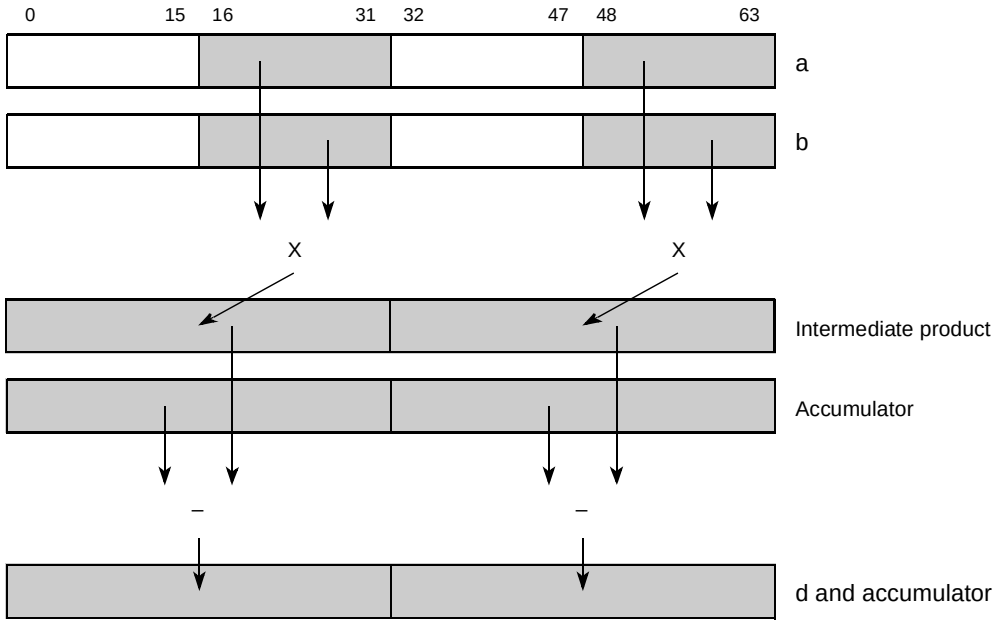

Figure 3-148. Odd Form of Vector Half-Word Multiply (__ev_mhoumfanw)

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmhoumianw d,a,b**

__ev_mhoumianw __ev_mhoumianw

Vector Multiply Half Words, Odd, Unsigned, Modulo, Integer and Accumulate Negative into Words

d = __ev_mhoumianw (a,b)

```
// high
temp0:31 ← a0:15 ×ui b0:15
d0:31 ← ACC0:31 - temp0:31
/
/ low
temp0:31 ← a32:47 ×ui b32:47
d32:63 ← ACC32:63 - temp0:31

// update accumulator
ACC0:63 ← d0:63
```

For each word element in the accumulator, the corresponding odd-numbered half-word unsigned integer elements in parameters a and b are multiplied. Each intermediate product is subtracted from the contents of the corresponding accumulator word. The results are placed into the corresponding parameter d and accumulator words.

Other registers altered: ACC

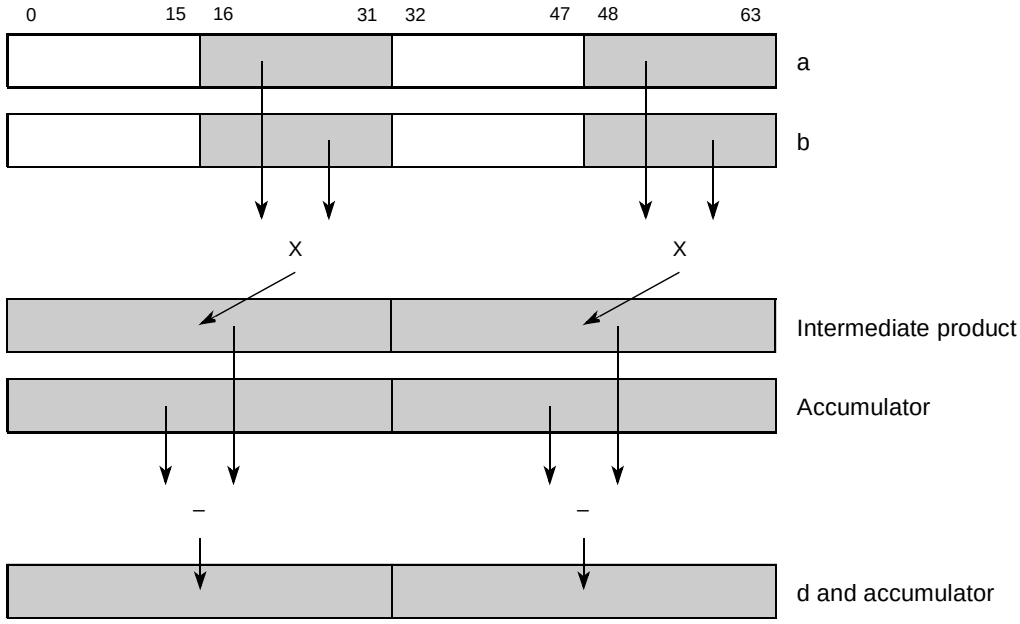

Figure 3-149. Odd Form of Vector Half-Word Multiply (__ev_mhoumianw)

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmhoumianw d,a,b**

__ev_mhousfaaw __ev_mhousfaaw

Vector Multiply Half Words, Odd, Unsigned, Saturate, Fractional and Accumulate into Words

d = __ev_mhousfaaw (a,b)

```
// high
temp0[0:31] ← a[16:31] ×ui b[16:31]
temp0[0:63] ← EXTZ(ACC[0:31]) + EXTZ(temp0[0:31])
if temp0[31] = 1
    d[0:31] ← 0xFFFF_FFFF //overflow
    ovh ← 1
else
    d[0:31] ← temp0[32:63]
    ovh ← 0
//low
temp1[0:31] ← a[48:63] ×ui b[48:63]
temp1[0:63] ← EXTZ(ACC[32:63]) + EXTZ(temp1[0:31])
if temp1[31] = 1
    d[32:63] ← 0xFFFF_FFFF //overflow
    ovl ← 1
else
    d[32:63] ← temp1[32:63]
    ovl ← 0
// update accumulator
ACC[0:63] ← d[0:63]
// update SPEFSCR
SPEFSCR[OVH] ← ovh
SPEFSCR[OV] ← ovl
SPEFSCR[SOVH] ← SPEFSCR[SOVH] | ovh
SPEFSCR[SOV] ← SPEFSCR[SOV] | ovl
```

For each word element in the accumulator, the corresponding odd-numbered half word elements in parameters a and b are multiplied. Each product is added to the corresponding accumulator word contents. If a sum overflows, the appropriate saturation value is placed into the corresponding parameter d and accumulator words. Otherwise, the sums are placed there. The SPEFSCR records overflow or summary overflow information.

Other registers altered: SPEFSCR ACC

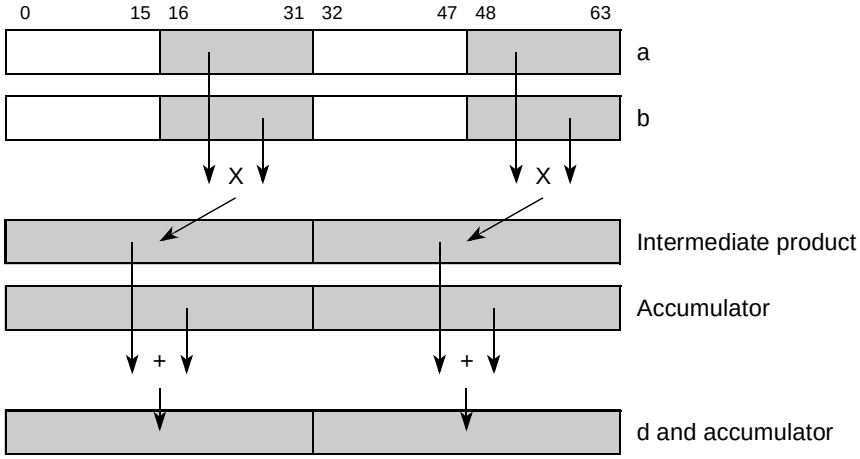

Figure 3-150. Odd Form of Vector Half Word Multiply (__ev_mhousfaaw)

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmhousiaaw d,a,b**

__ev_mhousiaaw __ev_mhousiaaw

Vector Multiply Half Words, Odd, Unsigned, Saturate, Integer and Accumulate into Words

d = __ev_mhousiaaw (a,b)

```
// high
temp0:31 ← a16:31 ×ui b16:31
temp0:63 ← EXTZ(ACC0:31) + EXTZ(temp0:31)
ovh ← temp31
d0:31 ← SATURATE(ovh, 0, 0xFFFF_FFFF, 0xFFFF_FFFF, temp32:63)

//low
temp0:31 ← a48:63 ×ui b48:63
temp0:63 ← EXTZ(ACC32:63) + EXTZ(temp0:31)
ovl ← temp31
d32:63 ← SATURATE(ovl, 0, 0xFFFF_FFFF, 0xFFFF_FFFF, temp32:63)

// update accumulator
ACC0:63 ← d0:63

// update SPEFSCR
SPEFSCROVH ← ovh
SPEFSCROV ← ovl
SPEFSCRSOVH ← SPEFSCRSOVH | ovh
SPEFSCRSOV ← SPEFSCRSOV | ovl
```

For each word element in the accumulator, corresponding odd-numbered half-word unsigned integer elements in parameters a and b are multiplied, producing a 32-bit product. Each 32-bit product is then added to the corresponding word in the accumulator, saturating if overflow occurs, and the result is placed in parameter d and the accumulator.

If there is an overflow from the addition, the overflow and summary overflow bits are recorded in the SPEFSCR.

Other registers altered: SPEFSCR ACC

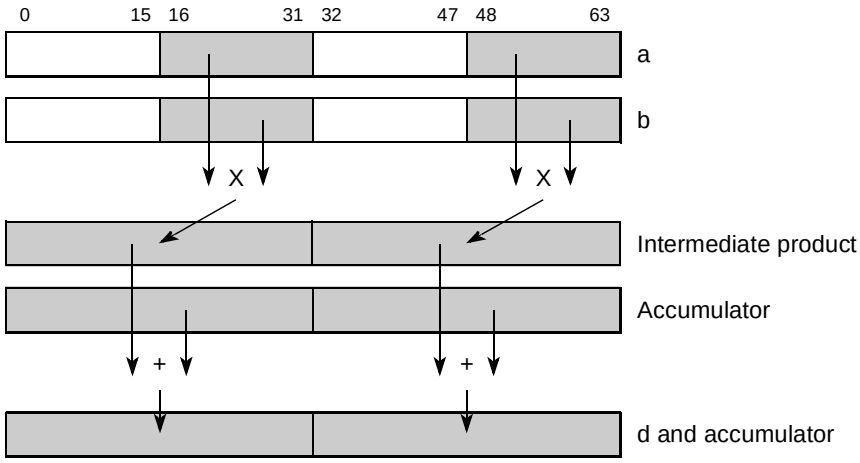

Figure 3-151. Odd Form of Vector Half Word Multiply (__ev_mhousiaaw)

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmhousiaaw d,a,b**

__ev_mhousfanw __ev_mhousfanw

Vector Multiply Half Words, Odd, Unsigned, Saturate, Fractional and Accumulate Negative into Words

d = __ev_mhousfanw (a,b)

```
// high
temp0[0:31] ← a[16:31] ×ui b[16:31]
temp0[0:63] ← EXTZ(ACC[0:31]) - EXTZ(temp0[0:31])
if temp0[31] = 1
    d[0:31] ← 0xFFFF_FFFF //overflow
    ovh ← 1
else
    d[0:31] ← temp0[32:63]
    ovh ← 0
//low
temp1[0:31] ← a[48:63] ×ui b[48:63]
temp1[0:63] ← EXTZ(ACC[32:63]) - EXTZ(temp1[0:31])
if temp1[31] = 1
    d[32:63] ← 0xFFFF_FFFF //overflow
    ovl ← 1
else
    d[32:63] ← temp1[32:63]
    ovl ← 0
// update accumulator
ACC[0:63] ← d[0:63]
// update SPEFSCR
SPEFSCR[OVH] ← ovh
SPEFSCR[OV] ← ovl
SPEFSCR[SOVH] ← SPEFSCR[SOVH] | ovh
SPEFSCR[SOV] ← SPEFSCR[SOV] | ovl
```

For each word element in the accumulator, the corresponding odd-numbered half word elements in parameters a and b are multiplied. Each product is subtracted from the accumulator word contents. If a result overflows, the appropriate saturation value is placed into the corresponding parameter d and accumulator words. Otherwise, the sums are placed there. The SPEFSCR records overflow or summary overflow information.

Other registers altered: SPEFSCR ACC

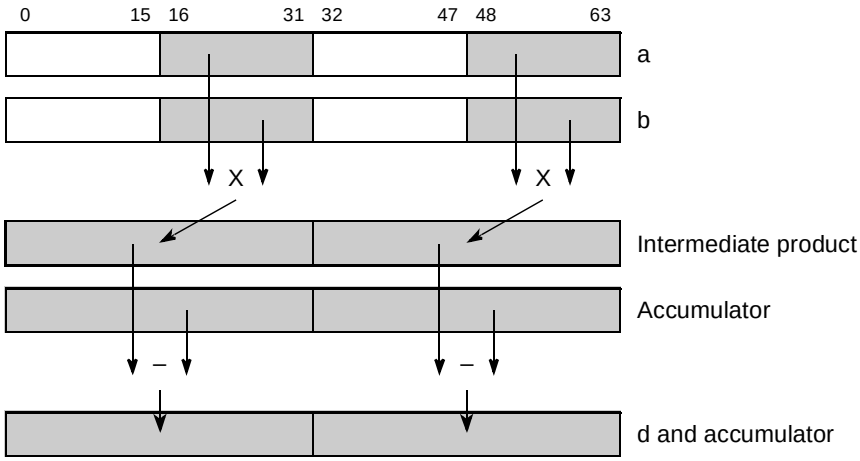

Figure 3-152. Odd Form of Vector Half Word Multiply (__ev_mhousfanw)

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmhousianw d,a,b**

__ev_mhousianw __ev_mhousianw

Vector Multiply Half Words, Odd, Unsigned, Saturate, Integer and Accumulate Negative into Words

d = __ev_mhousianw (a,b)

```
// high
temp0:31 ← a16:31 ×ui b16:31
temp0:63 ← EXTZ(ACC0:31) - EXTZ(temp0:31)
ovh ← temp31
d0:31 ← SATURATE(ovh, 0, 0, 0, temp32:63)

//low
temp0:31 ← a48:63 ×ui b48:63
temp0:63 ← EXTZ(ACC32:63) - EXTZ(temp0:31)
ovl ← temp31
d32:63 ← SATURATE(ovl, 0, 0, 0, temp32:63)

// update accumulator
ACC0:63 ← d0:63

// update SPEFSCR
SPEFSCROVH ← ovh
SPEFSCROV ← ovl
SPEFSCRSOVH ← SPEFSCRSOVH | ovh
SPEFSCRSOV ← SPEFSCRSOV | ovl
```

For each word element in the accumulator, corresponding odd-numbered half-word unsigned integer elements in parameters a and b are multiplied, producing a 32-bit product. Each 32-bit product is then subtracted from the corresponding word in the accumulator, saturating if overflow occurs, and the result is placed in parameter d and the accumulator.

If there is an overflow from the subtraction, the overflow and summary overflow bits are recorded in the SPEFSCR.

Other registers altered: SPEFSCR ACC

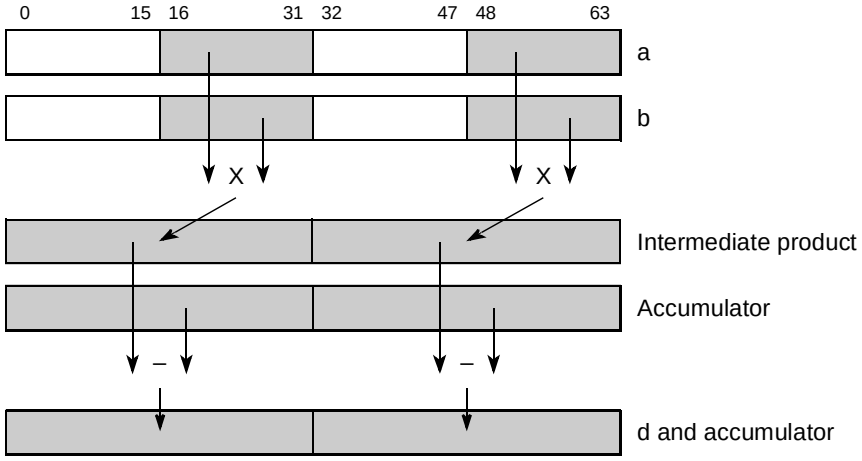

Figure 3-153. Odd Form of Vector Half Word Multiply (__ev_mhousianw)

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmhousianw d,a,b**

__ev_mra __ev_mra

Initialize Accumulator

d = __ev_mra (a)

$ACC_{0:63} \leftarrow a_{0:63}$
$d_{0:63} \leftarrow a_{0:63}$

The contents of parameter a are written into the accumulator and copied into parameter d. This is the method for initializing the accumulator.

Other registers altered: ACC

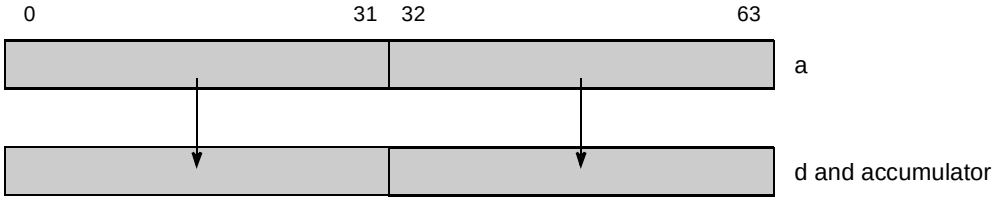

Figure 3-154. Initialize Accumulator (__ev_mra)

d	a	Maps to
__ev64_opaque	__ev64_opaque	**evmra d,a**

__ev_mwhsmf __ev_mwhsmf

Vector Multiply Word High Signed, Modulo, Fractional (to Accumulator)

d = __ev_mwhsmf (a,b) (A = 0)
d = __ev_mwhsmfa (a,b) (A = 1)

```
// high
temp0:63 ← a0:31 ×sf b0:31
d0:31 ← temp0:31

// low
temp0:63 ← a32:63 ×sf b32:63
d32:63 ← temp0:31

// update accumulator
if A = 1 then ACC0:63 ← d0:63
```

The corresponding word signed fractional elements in parameters a and b are multiplied, and bits 0–31 of the two products are placed into the two corresponding words of parameter d.

If A = 1, the result in parameter d is also placed into the accumulator.

Other registers altered: ACC (if A =1)

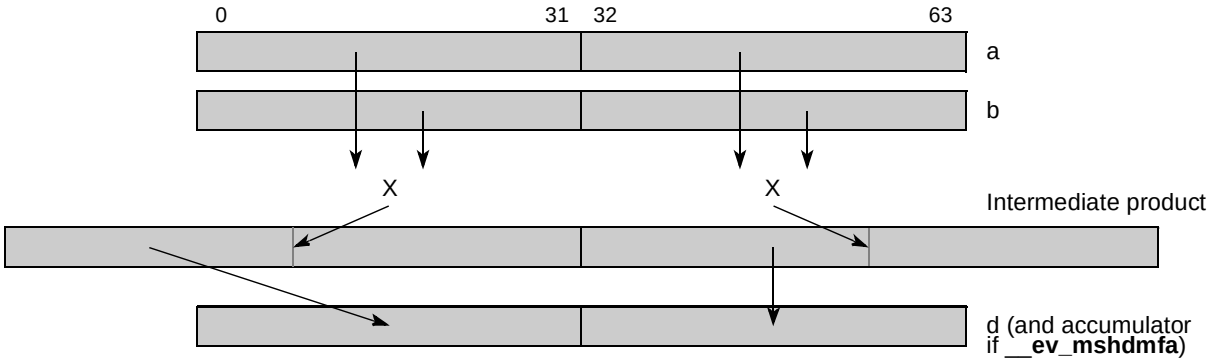

Figure 3-155. Vector Multiply Word High Signed, Modulo, Fractional (to Accumulator) (__ev_mwhsmf)

A	d	a	b	Maps to
A = 0	__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmwhsmf d,a,b**
A = 1	__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmwhsmfa d,a,b**

__ev_mwhsmi __ev_mwhsmi

Vector Multiply Word High Signed, Modulo, Integer (to Accumulator)

d = __ev_mwhsmi (a,b) (A = 0)
d = __ev_mwhsmia (a,b) (A = 1)

```
// high
temp0:63 ← a0:31 ×si b0:31
d0:31 ← temp0:31

// low
temp0:63 ← a32:63 ×si b32:63
d32:63 ← temp0:31

// update accumulator
if A = 1 then ACC0:63 ← d0:63
```

The corresponding word signed integer elements in parameters a and b are multiplied. Bits 0–31 of the two 64-bit products are placed into the two corresponding words of parameter d.

If A = 1,The result in parameter d is also placed into the accumulator.

Other registers altered: ACC (if A = 1)

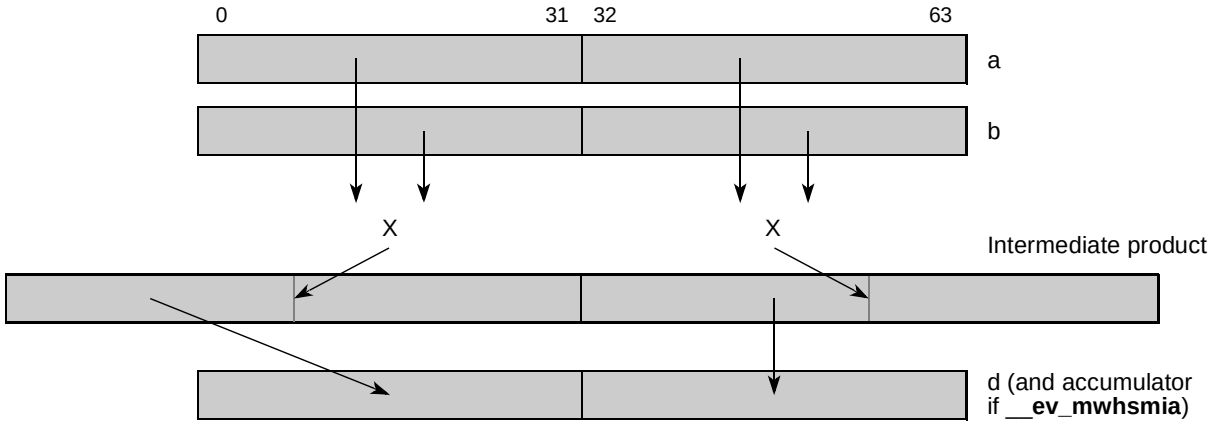

Figure 3-156. Vector Multiply Word High Signed, Modulo, Integer (to Accumulator) (__ev_mwhsmi)

A	d	a	b	Maps to
A = 0	__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmwhsmi d,a,b**
A = 1	__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmwhsmia d,a,b**

__ev_mwhssf __ev_mwhssf

Vector Multiply Word High Signed, Saturate, Fractional (to Accumulator)

d = __ev_mwhssf (a,b) (A = 0)
d = __ev_mwhssfa (a,b) (A = 1)

```
// high
temp0:63 ← a0:31 ×sf b0:31
if (a0:31 = 0x8000_0000) & (b0:31 = 0x8000_0000) then
    d0:31 ← 0x7FFF_FFFF //saturate
    movh ← 1
else
    d0:31 ← temp0:31
    movh ← 0

// low
temp0:63 ← a32:63 ×sf b32:63
if (a32:63 = 0x8000_0000) & (b32:63 = 0x8000_0000) then
    d32:63 ← 0x7FFF_FFFF //saturate
    movl ← 1
else
    d32:63 ← temp0:31
    movl ← 0

// update accumulator
if A = 1 then ACC0:63 ← d0:63

// update SPEFSCR
SPEFSCROVH ← movh
SPEFSCROV ← movl
SPEFSCRSOVH ← SPEFSCRSOVH | movh
SPEFSCRSOV ← SPEFSCRSOV | movl
```

The corresponding word signed fractional elements in parameters a and b are multiplied. Bits 0–31 of each product are placed into the corresponding words of parameter d. If both inputs are -1.0, the result saturates to the largest positive signed fraction and the overflow and summary overflow bits are recorded in the SPEFSCR.

Other registers altered: SPEFSCR ACC (if A = 1)

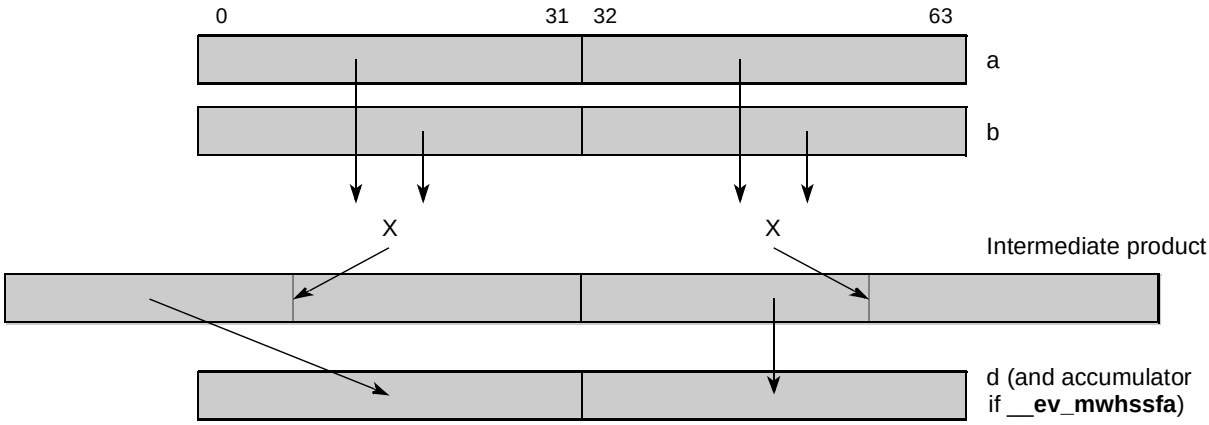

Figure 3-157. Vector Multiply Word High Signed, Saturate, Fractional (to Accumulator) (__ev_mwhssf)

A	d	a	b	Maps to
A = 0	__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmwhssf d,a,b**
A = 1	__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmwhssfa d,a,b**

__ev_mwhumf __ev_mwhumf

Vector Multiply Word High Unsigned, Modulo, Fractional (to Accumulator)

d = __ev_mwhumf (a,b) (A = 0)
d = __ev_mwhumfa (a,b) (A = 1)

```
// high
temp0 0:63 ← a 0:31 ×ui b 0:31
d 0:31 ← temp0 0:31
// low
temp1 0:63 ← a 32:63 ×ui b 32:63
d 32:63 ← temp1 0:31
// update accumulator
if A = 1, ACC 0:63 ← d 0:63
```

The corresponding word unsigned integer elements in parameters a and b are multiplied. Bits 0–31 of the two products are placed into the two corresponding words of parameter d.

If A = 1, the result in parameter d is also placed into the accumulator.

Other registers altered: ACC (if A = 1)

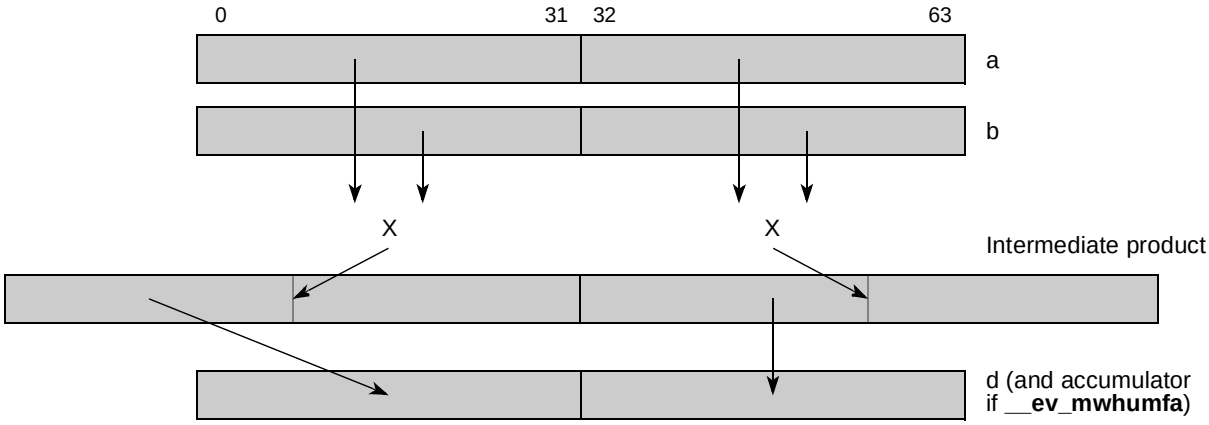

Figure 3-158. Vector Multiply Word High Unsigned, Modulo, Integer (to Accumulator) (__ev_mwhumf)

A	d	a	b	Maps to
A = 0	__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmwhumi d,a,b**
A = 1	__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmwhumia d,a,b**

__ev_mwhumi __ev_mwhumi

Vector Multiply Word High Unsigned, Modulo, Integer (to Accumulator)

d = __ev_mwhumi (a,b) (A = 0)
d = __ev_mwhumia (a,b) (A = 1)

```
// high
temp0:63 ← a0:31 ×ui b0:31
d0:31 ← temp0:31

// low
temp0:63 ← a32:63 ×ui b32:63
d32:63 ← temp0:31

// update accumulator
if A = 1, ACC0:63 ← d0:63
```

The corresponding word unsigned integer elements in parameters a and b are multiplied. Bits 0–31 of the two products are placed into the two corresponding words of parameter d.

If A = 1, the result in parameter d is also placed into the accumulator.

Other registers altered: ACC (if A = 1)

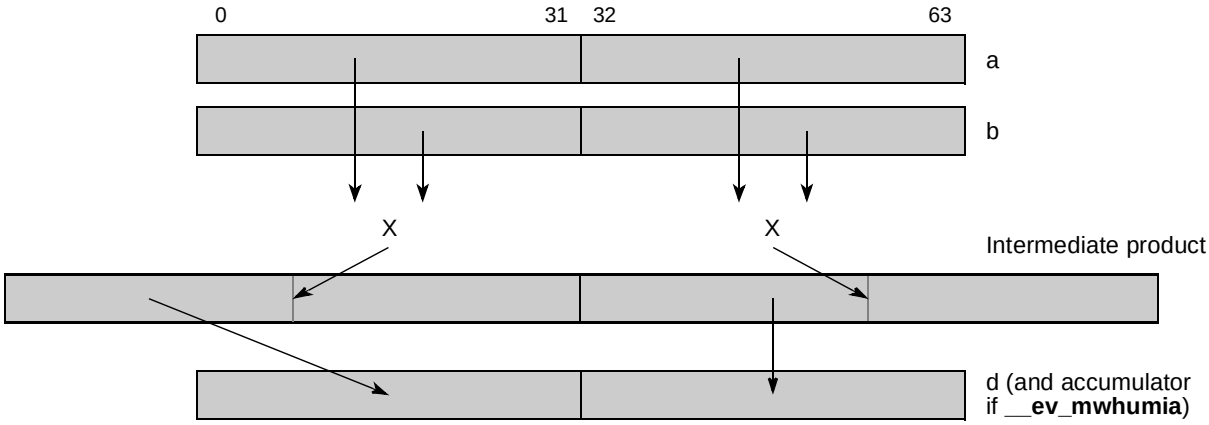

Figure 3-159. Vector Multiply Word High Unsigned, Modulo, Integer (to Accumulator) (__ev_mwhumi)

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmwhumi d,a,b**

__ev_mwlsmiaaw __ev_mwlsmiaaw

Vector Multiply Word Low Signed, Modulo, Integer and Accumulate in Words

d = __ev_mwlsmiaaw (a,b)

```
// high
temp0:63 ← a0:31 ×si b0:31
d0:31 ← ACC0:31 + temp32:63

// low
temp0:63 ← a32:63 ×si b32:63
d32:63 ← ACC32:63 + temp32:63

// update accumulator
ACC0:63 ← d0:63
```

For each word element in the accumulator, the corresponding word signed integer elements in parameters a and b are multiplied. The least significant 32 bits of each intermediate product is added to the contents of the corresponding accumulator words, and the result is placed into parameter d and the accumulator.

Other registers altered: ACC

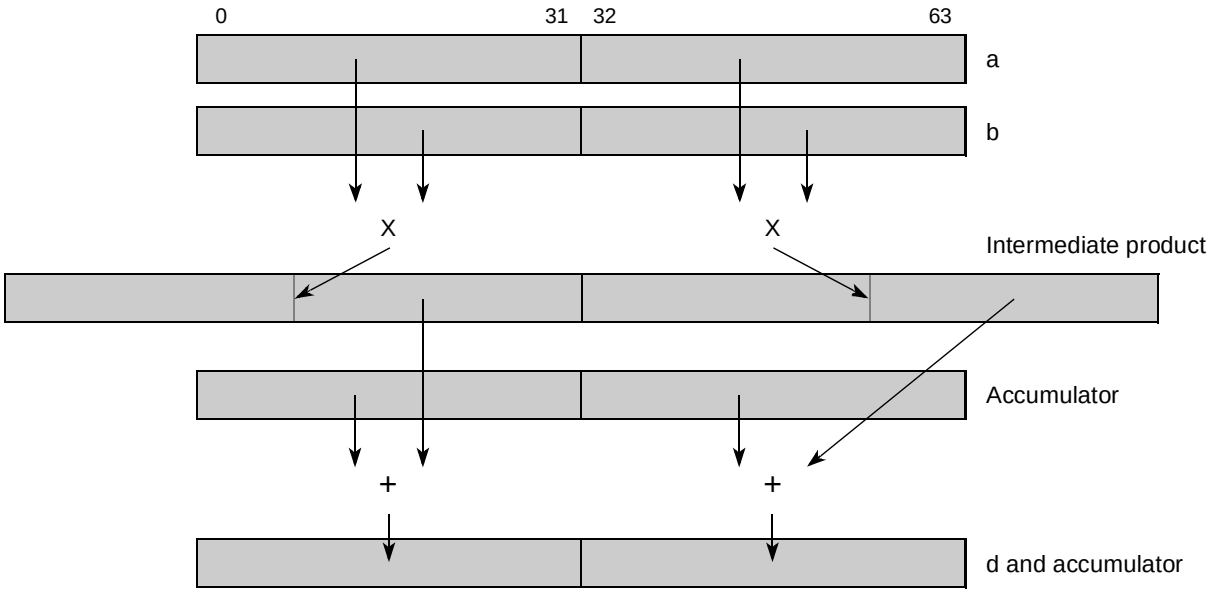

Figure 3-160. Vector Multiply Word Low Signed, Modulo, Integer and Accumulate in Words (__ev_mwlsmiaaw)

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmwlsmiaaw d,a,b**

__ev_mwlsmianw __ev_mwlsmianw

Vector Multiply Word Low Signed, Modulo, Integer and Accumulate Negative in Words

d = __ev_mwlsmianw (a,b)

```
// high
temp0:63 ← a0:31 ×si b0:31
d0:31 ← ACC0:31 - temp32:63

// low
temp0:63 ← a32:63 ×si b32:63
d32:63 ← ACC32:63 - temp32:63

// update accumulator
ACC0:63 ← d0:63
```

For each word element in the accumulator, the corresponding word elements in parameters a and b are multiplied. The least significant 32 bits of each intermediate product is subtracted from the contents of the corresponding accumulator words, and the result is placed in parameter d and the accumulator.

Other registers altered: ACC

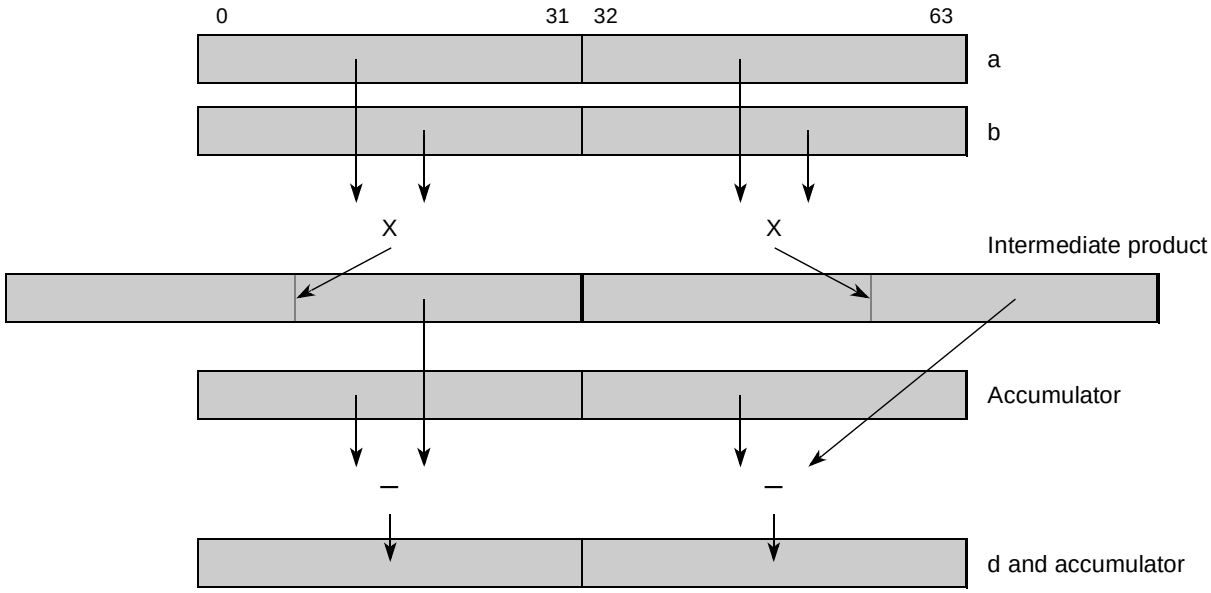

Figure 3-161. Vector Multiply Word Low Signed, Modulo, Integer and Accumulate Negative in Words (__ev_mwlsmianw)

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmwlsmfanw d,a,b**

__ev_mwlssiaaw __ev_mwlssiaaw

Vector Multiply Word Low Signed, Saturate, Integer and Accumulate in Words

d = __ev_mwlssiaaw (a,b)

```
// high
temp0:63 ← a0:31 ×si b0:31
temp0:63 ← EXTS(ACC0:31) + EXTS(temp32:63)
ovh ← (temp31 ⊕ temp32)
d0:31 ← SATURATE(ovh, temp31, 0x8000_0000, 0x7FFF_FFFF, temp32:63)

// low
temp0:63 ← a32:63 ×si b32:63
temp0:63 ← EXTS(ACC32:63) + EXTS(temp32:63)
ovl ← (temp31 ⊕ temp32)
d32:63 ← SATURATE(ovl, temp31, 0x8000_0000, 0x7FFF_FFFF, temp32:63)

// update accumulator
ACC0:63 ← d0:63

// update SPEFSCR
SPEFSCROVH ← ovh
SPEFSCROV ← ovl
SPEFSCRSOVH ← SPEFSCRSOVH | ovh
SPEFSCRSOV ← SPEFSCRSOV | ovl
```

The corresponding word signed integer elements in parameters a and b are multiplied, producing a 64-bit product. The least significant 32 bits of each product are then added to the corresponding word in the accumulator, saturating if overflow or underflow occurs, and the result is placed in parameter d and the accumulator.

If there is an overflow or underflow from the addition, the overflow and summary overflow bits are recorded in the SPEFSCR.

Other registers altered: SPEFSCR ACC

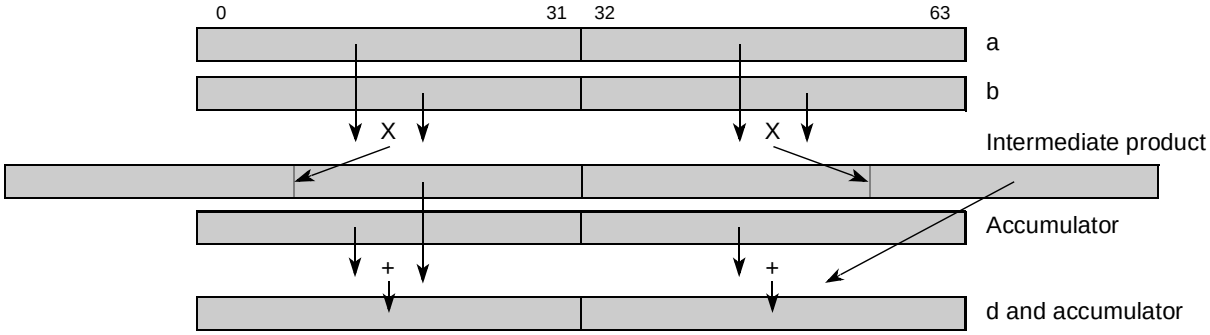

Figure 3-162. Vector Multiply Word Low Signed, Saturate, Integer and Accumulate in Words (__ev_mwlssiaaw)

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmwlssiaaw d,a,b**

__ev_mwlssianw __ev_mwlssianw

Vector Multiply Word Low Signed, Saturate, Integer and Accumulate Negative in Words

d = __ev_mwlssianw (a,b)

```
// high
temp0:63 ← a0:31 ×si b0:31
temp0:63 ← EXTS(ACC0:31) - EXTS(temp32:63)
ovh ← (temp31 ⊕ temp32)
d0:31 ← SATURATE(ovh, temp31, 0x8000_0000, 0x7FFF_FFFF, temp32:63)

// low
temp0:63 ← a32:63 ×si b32:63
temp0:63 ← EXTS(ACC32:63) - EXTS(temp32:63)
ovl ← (temp31 ⊕ temp32)
d32:63 ← SATURATE(ovl, temp31, 0x8000_0000, 0x7FFF_FFFF, temp32:63)

// update accumulator
ACC0:63 ← d0:63

// update SPEFSCR
SPEFSCROVH ← ovh
SPEFSCROV ← ovl
SPEFSCRSOVH ← SPEFSCRSOVH | ovh
SPEFSCRSOV ← SPEFSCRSOV | ovl
```

The corresponding word signed integer elements in parameters a and b are multiplied, producing a 64-bit product. The least significant 32 bits of each product are then subtracted from the corresponding word in the accumulator, saturating if overflow or underflow occurs, and the result is placed in parameter d and the accumulator.

If there is an overflow or underflow from the addition, the overflow and summary overflow bits are recorded in the SPEFSCR.

Other registers altered: SPEFSCR ACC

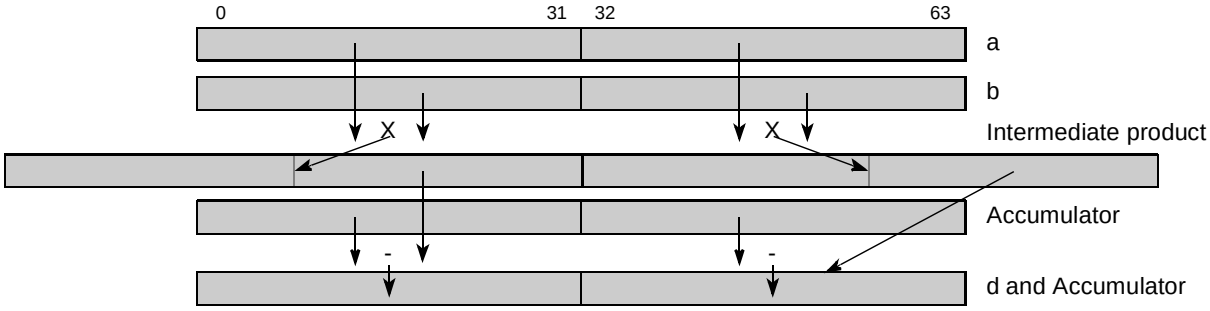

Figure 3-163. Vector Multiply Word Low Signed, Saturate, Integer and Accumulate Negative in Words (__ev_mwlssianw)

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmwlssianw d,a,b**

__ev_mwlumi __ev_mwlumi

Vector Multiply Word Low Unsigned, Modulo, Integer

d = __ev_mwlumi (a,b)

d = __ev_mwlumia (a,b)

```
// high
temp0:63 ← a0:31 ×ui b0:31
d0:31 ← temp32:63

// low
temp0:63 ← a32:63 ×ui b32:63
d32:63 ← temp32:63

// update accumulator
If A = 1 then ACC0:63 ← d0:63
```

The corresponding word unsigned integer elements in parameters a and b are multiplied. The least significant 32 bits of each product are placed into the two corresponding words of parameter d.

NOTE

The least significant 32 bits of the product are independent of whether the word elements in parameters a and b are treated as signed or unsigned 32-bit integers.

If A = 1, the result in parameter d is also placed into the accumulator.

Other registers altered: ACC (if A = 1)

Note that **evmwlumi** and **evmwlumia** can be used for signed or unsigned integers.

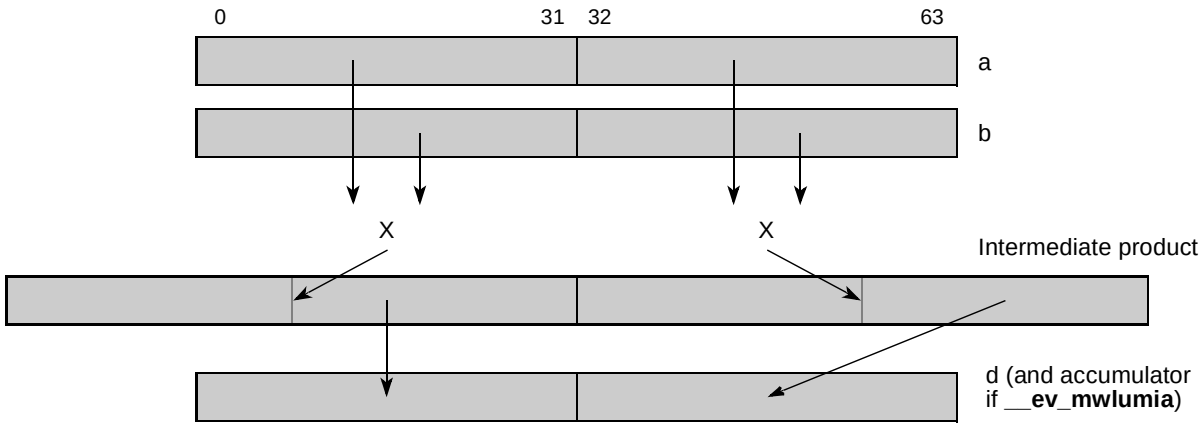

Figure 3-164. Vector Multiply Word Low Unsigned, Modulo, Integer (__ev_mwlumi)

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmwlumi d,a,b**
__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmwlumia d,a,b**

__ev_mwlumiaaw __ev_mwlumiaaw

Vector Multiply Word Low Unsigned, Modulo, Integer and Accumulate in Words

d = __ev_mwlumiaaw (a,b)

```
// high
temp0:63 ← a0:31 ×ui b0:31
d0:31 ← ACC0:31 + temp32:63

// low
temp0:63 ← a32:63 ×ui b32:63
d32:63 ← ACC32:63 + temp32:63

// update accumulator
ACC0:63 ← d0:63
```

For each word element in the accumulator, the corresponding word unsigned integer elements in parameters a and b are multiplied. The least significant 32 bits of each product are added to the contents of the corresponding accumulator word, and the result is placed into the corresponding parameter d and accumulator word.

Other registers altered: ACC

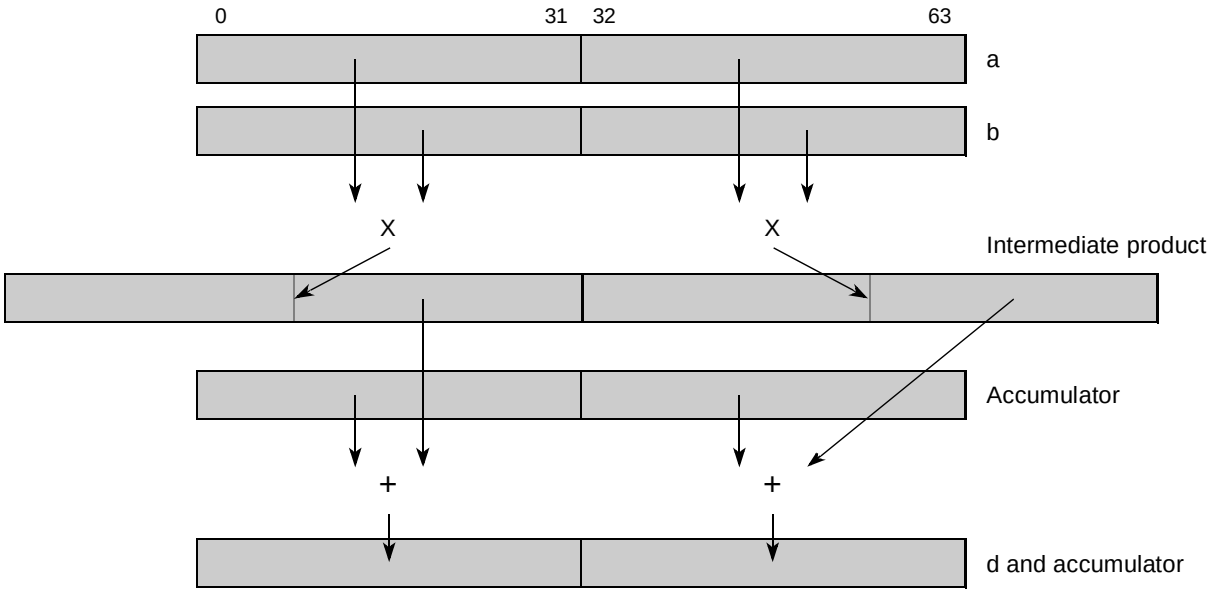

Figure 3-165. Vector Multiply Word Low Unsigned, Modulo, Integer and Accumulate in Words (__ev_mwlumiaaw)

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmwlumiaaw d,a,b**

__ev_mwlumianw __ev_mwlumianw

Vector Multiply Word Low Unsigned, Modulo, Integer and Accumulate Negative in Words

d = __ev_mwlumianw (a,b)

```
// high
temp0:63 ← a0:31 ×ui b0:31
d0:31 ← ACC0:31 - temp32:63

// low
temp0:63 ← a32:63 ×ui b32:63
d32:63 ← ACC32:63 - temp32:63

// update accumulator
ACC0:63 ← d0:63
```

For each word element in the accumulator, the corresponding word unsigned integer elements in parameters a and b are multiplied. The least significant 32 bits of each product are subtracted from the contents of the corresponding accumulator word, and the result is placed into parameter d and the accumulator.

Other registers altered: ACC

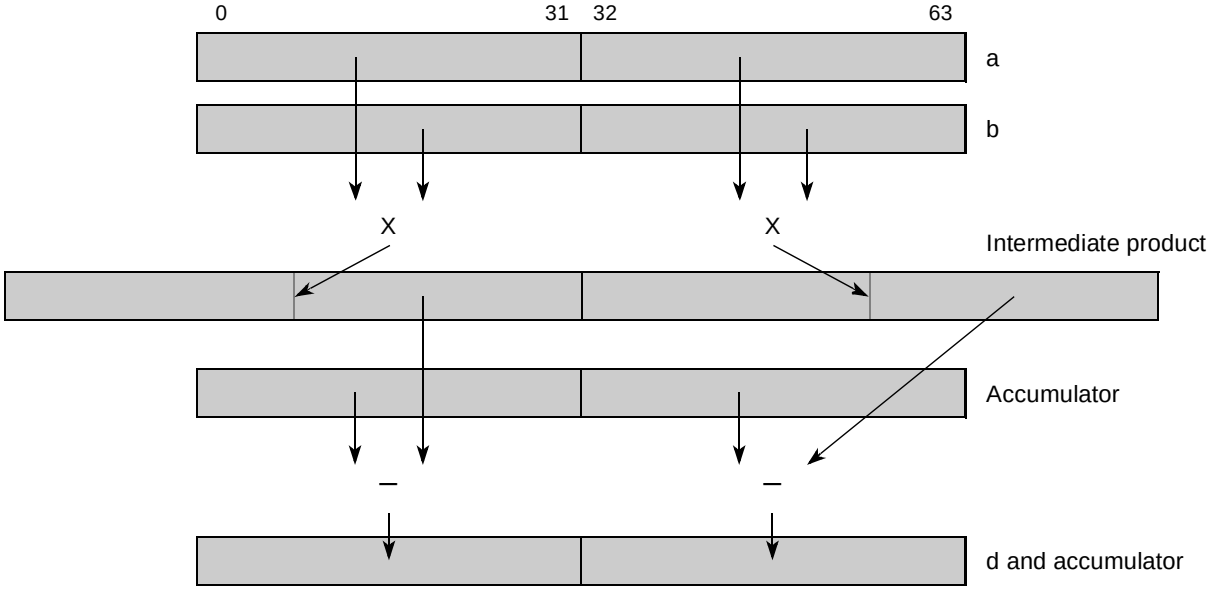

Figure 3-166. Vector Multiply Word Low Unsigned, Modulo, Integer and Accumulate Negative in Words (__ev_mwlumianw)

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmwlumianw d,a,b**

__ev_mwlusiaaw __ev_mwlusiaaw

Vector Multiply Word Low Unsigned, Saturate, Integer and Accumulate in Words

d = __ev_mwlusiaaw (a,b)

```
// high
temp0:63 ← a0:31 ×ui b0:31
temp0:63 ← EXTZ(ACC0:31) + EXTZ(temp32:63)
ovh ← temp31
d0:31 ← SATURATE(ovh, 0, 0xFFFF_FFFF, 0xFFFF_FFFF, temp32:63)

//low
temp0:63 ← a32:63 ×ui b32:63
temp0:63 ← EXTZ(ACC32:63) + EXTZ(temp32:63)
ovl ← temp31
d32:63 ← SATURATE(ovl, 0, 0xFFFF_FFFF, 0xFFFF_FFFF, temp32:63)

// update accumulator
ACC0:63 ← d0:63

// update SPEFSCR
SPEFSCROVH ← ovh
SPEFSCROV ← ovl
SPEFSCRSOVH ← SPEFSCRSOVH | ovh
SPEFSCRSOV ← SPEFSCRSOV | ovl
```

For each word element in the accumulator, corresponding word unsigned integer elements in parameters a and b are multiplied, producing a 64-bit product. The least significant 32 bits of each product are then added to the corresponding word in the accumulator, saturating if overflow occurs, and the result is placed in parameter d and the accumulator.

If there is an overflow from the addition, the overflow and summary overflow bits are recorded in the SPEFSCR.

Other registers altered: SPEFSCR ACC

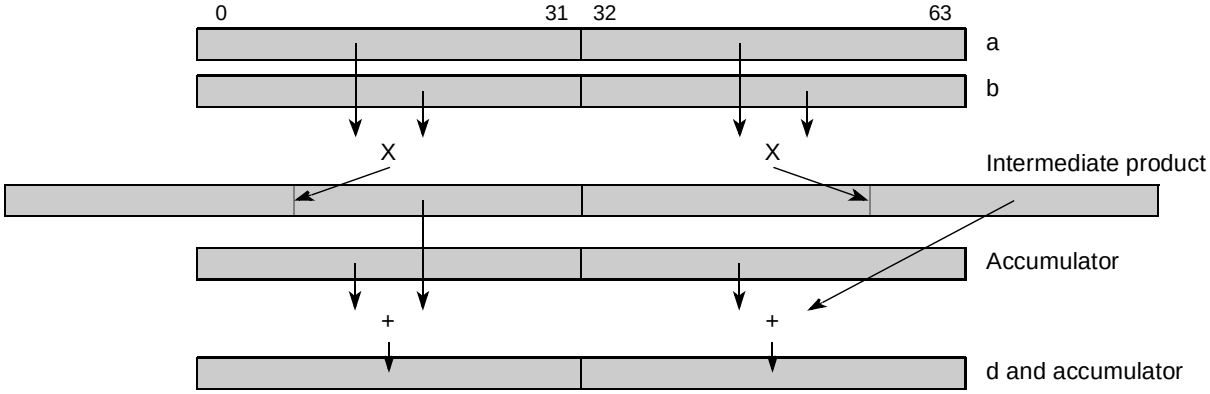

Figure 3-167. Vector Multiply Word Low Unsigned, Saturate, Integer and Accumulate in Words (__ev_mwlusiaaw)

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmwlusiaaw d,a,b**

__ev_mwlusianw __ev_mwlusianw

Vector Multiply Word Low Unsigned, Saturate, Integer and Accumulate Negative in Words

d = __ev_mwlusianw (a,b)

```
// high
temp0:63 ← a0:31 ×ui b0:31
temp0:63 ← EXTZ(ACC0:31) - EXTZ(temp32:63)
ovh ← temp31
d0:31 ← SATURATE(ovh, 0, 0x0000_0000, 0x0000_0000, temp32:63)

//low
temp0:63 ← a32:63 ×ui b32:63
temp0:63 ← EXTZ(ACC32:63) - EXTZ(temp32:63)
ovl ← temp31
d32:63 ← SATURATE(ovl, 0, 0x0000_0000, 0x0000_0000, temp32:63)

// update accumulator
ACC0:63 ← d0:63

// update SPEFSCR
SPEFSCROVH ← ovh
SPEFSCROV ← ovl
SPEFSCRSOVH ← SPEFSCRSOVH | ovh
SPEFSCRSOV ← SPEFSCRSOV | ovl
```

For each word element in the accumulator, corresponding word unsigned integer elements in parameters a and b are multiplied, producing a 64-bit product. The least significant 32 bits of each product are then subtracted from the corresponding word in the accumulator, saturating if underflow occurs, and the result is placed in parameter d and the accumulator.

If there is an underflow from the subtraction, the overflow and summary overflow bits are recorded in the SPEFSCR.

Other registers altered: SPEFSCR ACC

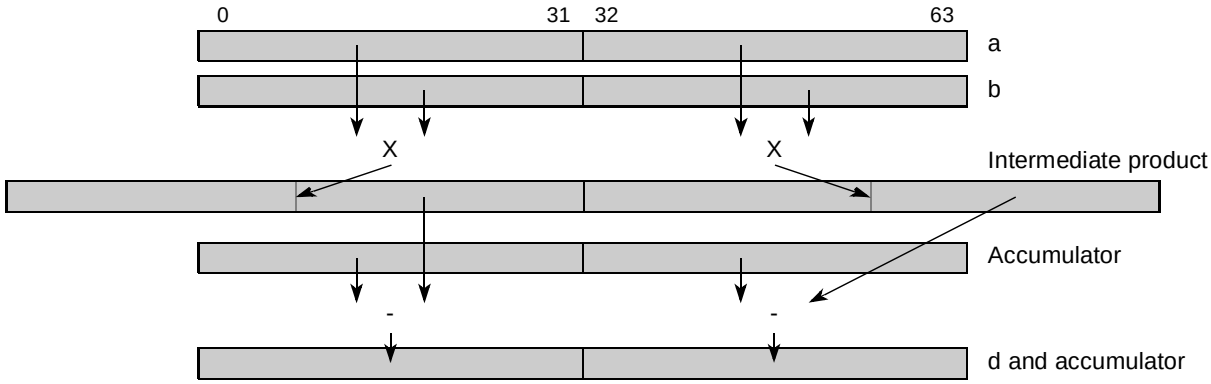

Figure 3-168. Vector Multiply Word Low Unsigned, Saturate, Integer and Accumulate Negative in Words (__ev_mwlusianw)

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmwlusianw d,a,b**

__ev_mwsmf __ev_mwsmf

Vector Multiply Word Signed, Modulo, Fractional (to Accumulator)

d = __ev_mwsmf (a,b) (A = 0)

d = __ev_mwsmfa (a,b) (A = 1)

```
d0:63 ← a32:63 ×sf b32:63

// update accumulator
if A = 1 then ACC0:63 ← d0:63
```

The corresponding low word signed fractional elements in parameters a and b are multiplied. The product is placed into parameter d.

If A = 1, the result in parameter d is also placed into the accumulator.

Other registers altered: ACC (if A = 1)

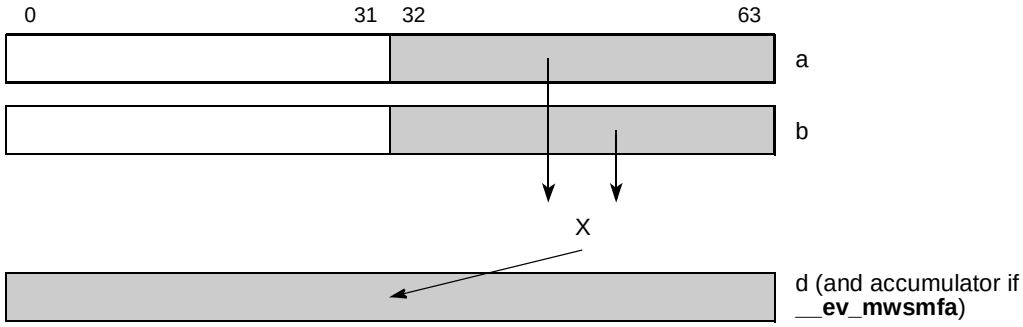

Figure 3-169. Vector Multiply Word Signed, Modulo, Fractional (to Accumulator) (__ev_mwsmf)

A	d	a	b	Maps to
A = 0	__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmwsmf d,a,b**
A = 1	__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmwsmfa d,a,b**

__ev_mwsmfaa __ev_mwsmfaa

Vector Multiply Word Signed, Modulo, Fractional and Accumulate

d = __ev_mwsmfaa (a,b)

```
temp0:63 ← a32:63 ×sf b32:63
d0:63 ← ACC0:63 + temp0:63
// update accumulator
ACC0:63 ← d0:63
```

The corresponding low word signed fractional elements in parameters a and b are multiplied. The intermediate product is added to the contents of the 64-bit accumulator and the result is placed in parameter d and the accumulator.

Other registers altered: ACC

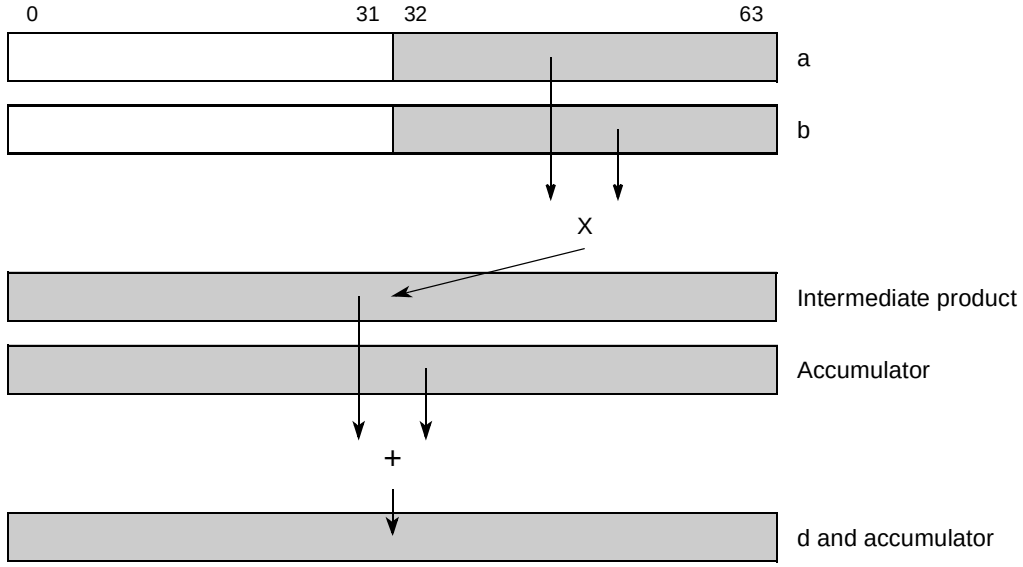

Figure 3-170. Vector Multiply Word Signed, Modulo, Fractional and Accumulate (__ev_mwsmfaa)

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmwsmfaa d,a,b**

__ev_mwsmfan __ev_mwsmfan

Vector Multiply Word Signed, Modulo, Fractional and Accumulate Negative

d = __ev_mwsmfan (a,b)

```
temp0:63 ← a32:63 ×sf b32:63
d0:63 ← ACC0:63 - temp0:63

// update accumulator
ACC0:63 ← d0:63
```

The corresponding low word signed fractional elements in parameters a and b are multiplied. The intermediate product is subtracted from the contents of the accumulator, and the result is placed in parameter d and the accumulator.

Other registers altered: ACC

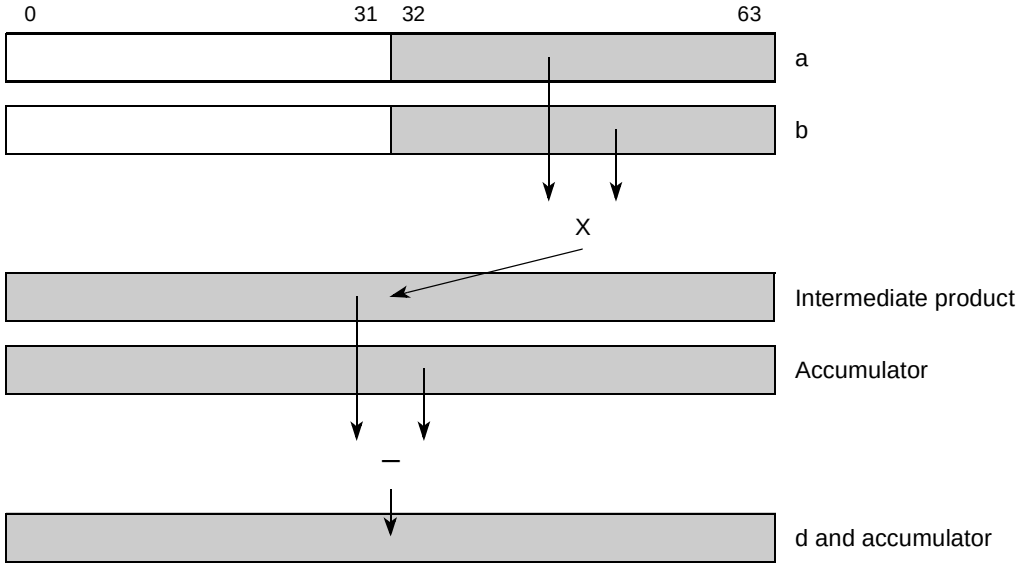

Figure 3-171. Vector Multiply Word Signed, Modulo, Fractional, and Accumulate Negative (__ev_mwsmfan)

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmwsmfan d,a,b**

__ev_mwsmi __ev_mwsmi

Vector Multiply Word Signed, Modulo, Integer (to Accumulator)

d = __ev_mwsmi (a,b) (A = 0)

d = __ev_mwsmia (a,b) (A = 1)

```
d0:63 ← a32:63 ×si b32:63

// update accumulator
if A = 1 then ACC0:63 ← d0:63
```

The low word signed integer elements in parameters a and b are multiplied. The product is placed into the parameter d.

If A = 1, the result in parameter d is also placed into the accumulator.

Other registers altered: ACC (if A = 1)

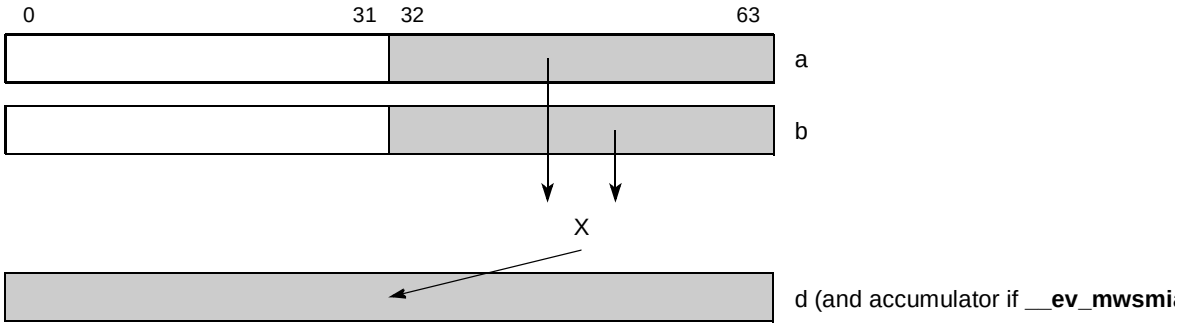

Figure 3-172. Vector Multiply Word Signed, Modulo, Integer (to Accumulator) (__ev_mwsmi)

A	d	a	b	Maps to
A = 0	__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmwsmi d,a,b**
A = 1	__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmwsmia d,a,b**

__ev_mwsmiaa __ev_mwsmiaa

Vector Multiply Word Signed, Modulo, Integer and Accumulate

d = __ev_mwsmiaa (a,b)

```
temp0:63 ← a32:63 ×si b32:63
d0:63 ← ACC0:63 + temp0:63
// update accumulator
ACC0:63 ← d0:63
```

The low word signed integer elements in parameters a and b are multiplied. The intermediate product is added to the contents of the 64-bit accumulator, and the result is placed into parameter d and the accumulator.

Other registers altered: ACC

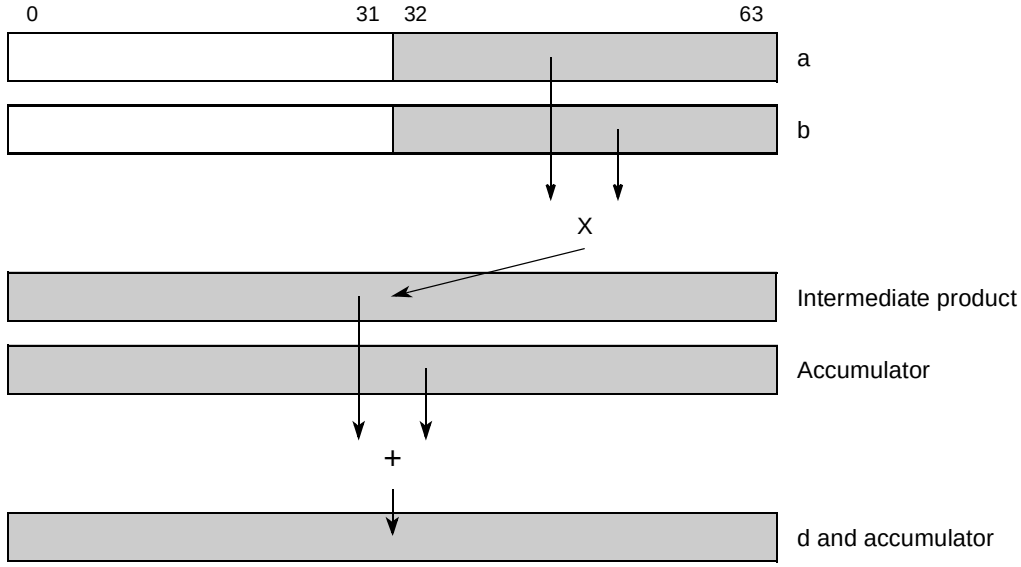

Figure 3-173. Vector Multiply Word Signed, Modulo, Integer and Accumulate (__ev_mwsmiaa)

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmwsmiaa d,a,b**

__ev_mwsmian __ev_mwsmian

Vector Multiply Word Signed, Modulo, Integer and Accumulate Negative

d = __ev_mwsmian (a,b)

```
temp0:63 ← a32:63 ×si b32:63
d0:63 ← ACC0:63 – temp0:63
// update accumulator
ACC0:63 ← d0:63
```

The corresponding low word signed integer elements in parameters a and b are multiplied. The intermediate product is subtracted from the contents of the 64-bit accumulator and the result is placed into parameter d and the accumulator.

Other registers altered: ACC

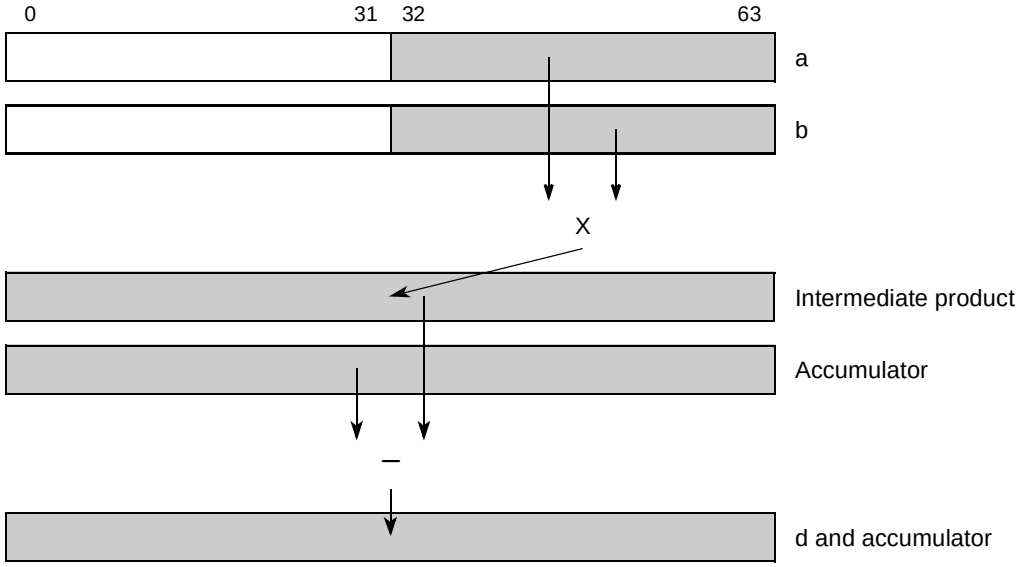

Figure 3-174. Vector Multiply Word Signed, Modulo, Integer and Accumulate Negative (__ev_mwsmian)

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmwsmian d,a,b**

__ev_mwssf __ev_mwssf

Vector Multiply Word Signed, Saturate, Fractional (to Accumulator)

d = __ev_mwssf (a,b) (A = 0)

d = __ev_mwssfa (a,b) (A = 1)

```
temp0:63 ← a32:63 ×sf b32:63
if (a32:63 = 0x8000_0000) & (b32:63 = 0x8000_0000) then
    d0:63 ← 0x7FFF_FFFF_FFFF_FFFF //saturate
    mov ← 1
else
    d0:63 ← temp0:63
    mov ← 0

// update accumulator
if A = 1 then ACC0:63 ← d0:63

// update SPEFSCR
SPEFSCROVH ← 0
SPEFSCROV ← mov
SPEFSCRSOV ← SPEFSCRSOV | mov
```

The low word signed fractional elements in parameters a and b are multiplied. The 64-bit product is placed into parameter d. If both inputs are -1.0, the result saturates to the largest positive signed fraction, and the overflow and summary overflow bits are recorded in the SPEFSCR.

If A = 1, the result in parameter d is also placed into the accumulator.

Other registers altered: SPEFSCR ACC (if A = 1)

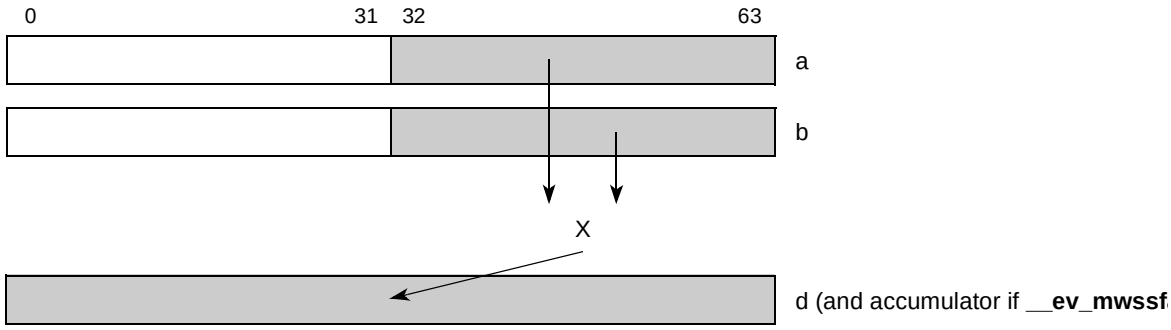

Figure 3-175. Vector Multiply Word Signed, Saturate, Fractional (to Accumulator) (__ev_mwssf)

A	d	a	b	Maps to
A = 0	__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmwssf d,a,b**
A = 1	__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmwssfa d,a,b**

__ev_mwssfaa __ev_mwssfaa

Vector Multiply Word Signed, Saturate, Fractional and Accumulate

d = __ev_mwssfaa (a,b)

```
temp0:63 ← a32:63 ×sf b32:63
if (a32:63 = 0x8000_0000) & (b32:63 = 0x8000_0000) then
    temp0:63 ← 0x7FFF_FFFF_FFFF_FFFF //saturate
    mov ← 1
else
    mov ← 0
temp0:64 ← EXTS(ACC0:63) + EXTS(temp0:63)
ov ← (temp0 ⊕ temp1)
d0:63 ← temp1:64)
// update accumulator
ACC0:63 ← d0:63

// update SPEFSCR
SPEFSCROVH ← 0
SPEFSCROV ← mov
SPEFSCRSOV ← SPEFSCRSOV | ov | mov
```

The low word signed fractional elements in parameters a and b are multiplied, producing a 64-bit product. If both inputs are -1.0, the product saturates to the largest positive signed fraction. The 64-bit product is added to the accumulator, and the result is placed in parameter d and in the accumulator.

If there is an overflow from the multiply, the overflow and summary overflow bits are recorded in the SPEFSCR.

Note: There is no saturation on the addition with the accumulator.

Other registers altered: SPEFSCR ACC

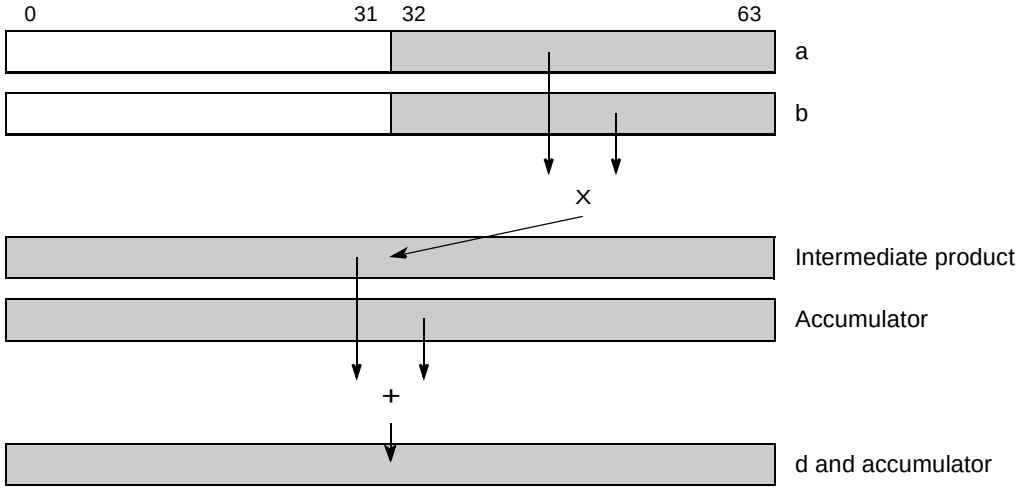

Figure 3-176. Vector Multiply Word Signed, Saturate, Fractional and Accumulate (__ev_mwssfaa)

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmwssfaa d,a,b**

__ev_mwssfan __ev_mwssfan

Vector Multiply Word Signed, Saturate, Fractional and Accumulate Negative

d = __ev_mwssfan (a,b)

```
temp0:63 ← a32:63 ×sf b32:63
if (a32:63 = 0x8000_0000) & (b32:63 = 0x8000_0000) then
    temp0:63 ← 0x7FFF_FFFF_FFFF_FFFF //saturate
    mov ← 1
else
    mov ← 0
temp0:64 ← EXTS(ACC0:63) - EXTS(temp 0:63)
ov ← (temp0 ⊕ temp1)
d0:63 ← temp1:64


// update accumulator
ACC0:63 ← d0:63

// update SPEFSCR
SPEFSCROVH ← 0
SPEFSCROV ← mov
SPEFSCRSOV ← SPEFSCRSOV | ov | mov
```

The low word signed fractional elements in parameters a and b are multiplied producing a 64-bit product. If both inputs are -1.0, the product saturates to the largest positive signed fraction. The 64-bit product is then subtracted from the accumulator and the result is placed in parameter d and the accumulator.

If there is an overflow from the multiply, the overflow and summary overflow bits are recorded in the SPEFSCR.

NOTE

There is no saturation on the subtraction with the accumulator.

Other registers altered: SPEFSCR ACC

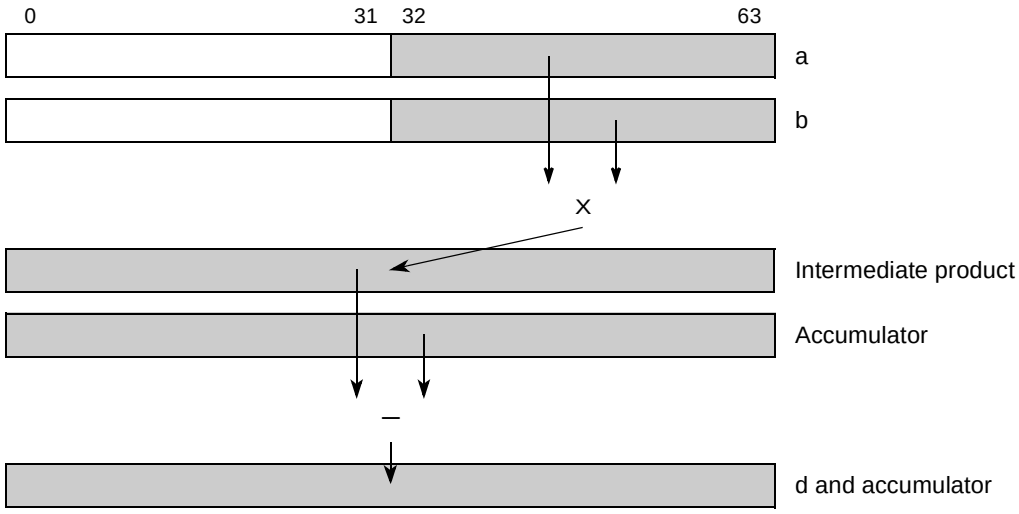

Figure 3-177. Vector Multiply Word Signed, Saturate, Fractional and Accumulate Negative (__ev_mwssfan)

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmwssfan d,a,b**

__ev_mwumi __ev_mwumi

Vector Multiply Word Unsigned, Modulo, Integer (to Accumulator)

d = __ev_mwumi (a,b) (A = 0)
d = __ev_mwumia (a,b) (A = 1)

```
d0:63 ← a32:63 ×ui b32:63

// update accumulator
if A = 1 then ACC0:63 ← d0:63
```

The low word unsigned integer elements in parameters a and b are multiplied to form a 64-bit product that is placed into parameter d.

If A = 1, the result in parameter d is also placed into the accumulator.

Other registers altered: ACC (if A = 1)

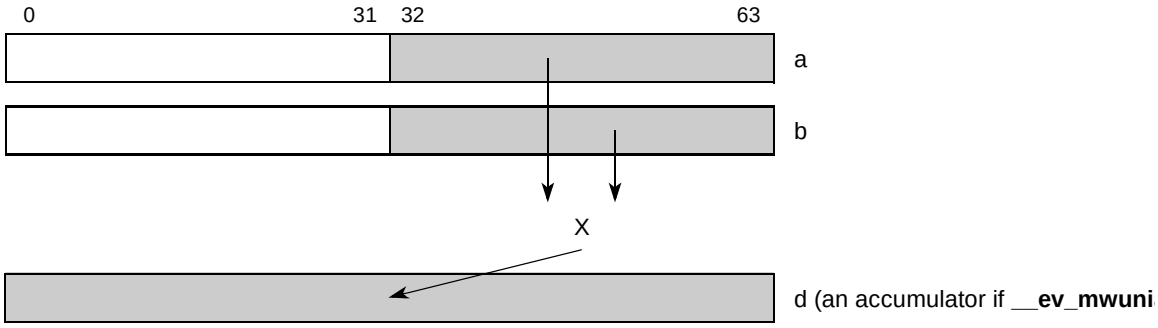

Figure 3-178. Vector Multiply Word Unsigned, Modulo, Integer (to Accumulator) (__ev_mwumi)

A	d	a	b	Maps to
A = 0	__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmwumi d,a,b**
A = 1	__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmwumia d,a,b**

__ev_mwumiaa __ev_mwumiaa

Vector Multiply Word Unsigned, Modulo, Integer and Accumulate

d = __ev_mwumiaa (a,b)

```
temp0:63 ← a32:63 ×ui b32:63
d0:63 ← ACC0:63 + temp0:63
// update accumulator
ACC0:63 ← d0:63
```

The low word unsigned integer elements in parameters a and b are multiplied. The intermediate product is added to the contents of the 64-bit accumulator, and the resulting value is placed into the accumulator and into parameter d.

Other registers altered: ACC

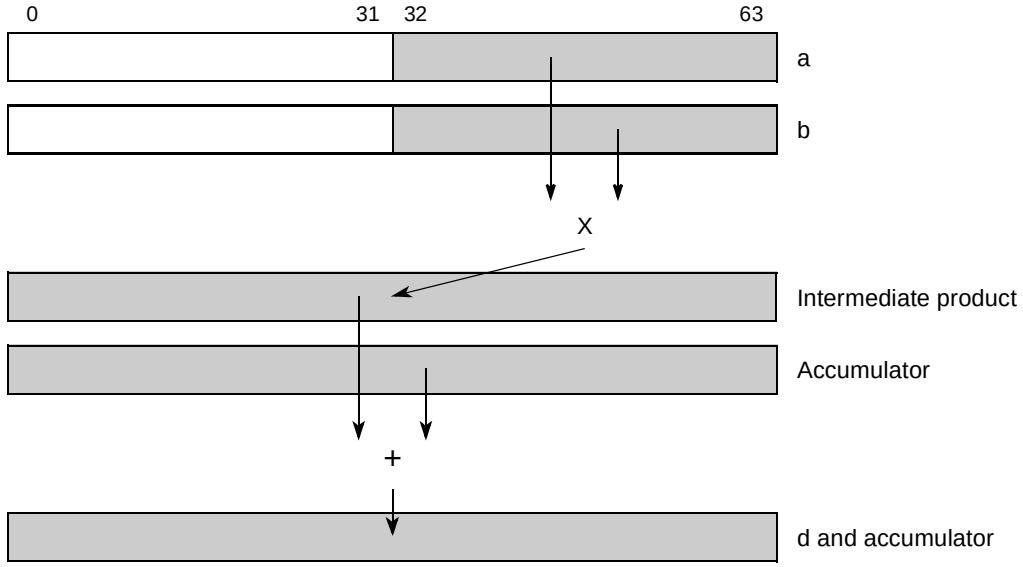

Figure 3-179. Vector Multiply Word Unsigned, Modulo, Integer and Accumulate (__ev_mwumiaa)

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmwumiaa d,a,b**

__ev_mwumian __ev_mwumian

Vector Multiply Word Unsigned, Modulo, Integer and Accumulate Negative

d = __ev_mwumian (a,b)

```
temp0:63 ← a32:63 ×ui b32:63
d0:63 ← ACC0:63 – temp0:63
// update accumulator
ACC0:63 ← d0:63
```

The low word unsigned integer elements in parameters a and b are multiplied. The intermediate product is subtracted from the contents of the 64-bit accumulator, and the resulting value is placed into the accumulator and into parameter d.

Other registers altered: ACC

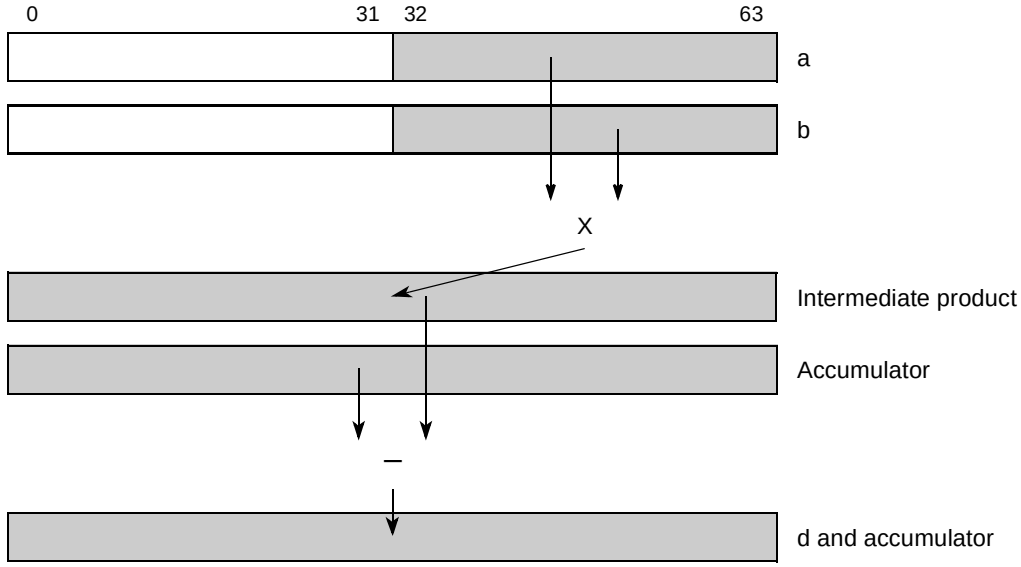

Figure 3-180. Vector Multiply Word Unsigned, Modulo, Integer and Accumulate Negative (__ev_mwumian)

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	**evmwumian d,a,b**

__ev_nand __ev_nand

Vector NAND

d = __ev_nand (a,b)

```
d0:31 ← ¬(a0:31 & b0:31)// Bitwise NAND
d32:63 ← ¬(a32:63 & b32:63) // Bitwise NAND
```

Each element of parameters a and b are bitwise NANDed. The result is placed in the corresponding element of parameter d.

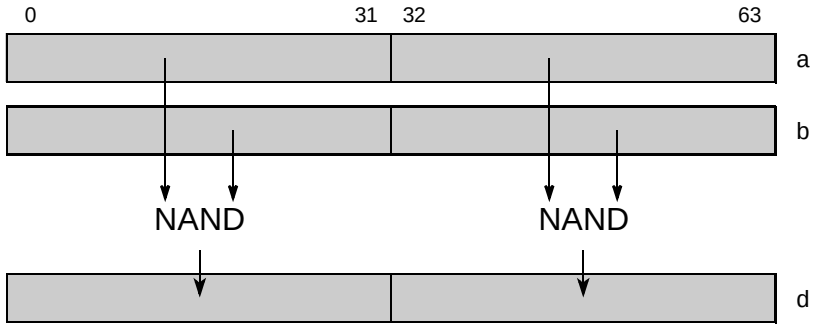

Figure 3-181. Vector NAND (__ev_nand)

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	**evnand d,a,b**

__ev_neg __ev_neg

Vector Negate

d = __ev_neg(a)

$d_{0:31} \leftarrow NEG(a_{0:31})$
$d_{32:63} \leftarrow NEG(a_{32:63})$

The negative of each element of parameter a is placed in parameter d. The negative of 0x8000_0000 (most negative number) returns 0x8000_0000. No overflow is detected.

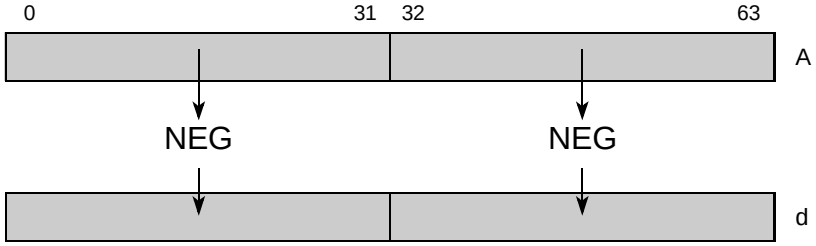

Figure 3-182. Vector Negate (__ev_neg)

d	a	Maps to
__ev64_opaque	__ev64_opaque	**evneg d,a,b**

__ev_nor __ev_nor

Vector NOR

d = __ev_nor (a,b)

```
d0:31 ← ¬(a0:31 | b0:31) // Bitwise NOR
d32:63 ← ¬(a32:63 | b32:63) // Bitwise NOR
```

Each element of parameters a and b is bitwise NORed. The result is placed in the corresponding element of parameter d.

NOTE

Use **evnand** or **evnor** for **evnot**.

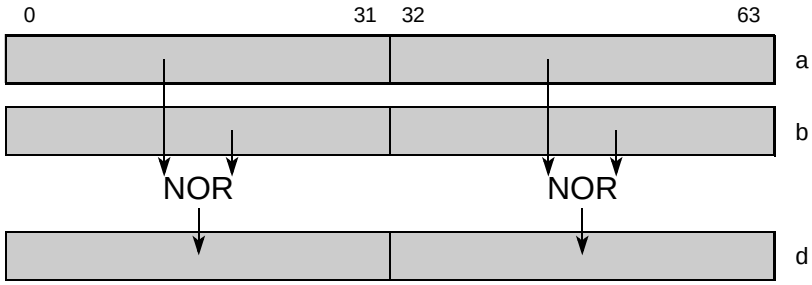

Figure 3-183. Vector NOR (__ev_nor)

Simplified mnemonic: **evnot** d,a performs a complement register.

evnot d,a equivalent to **evnor** d,a,a

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	**evnor d,a,b**

__ev_or __ev_or

Vector OR

d = __ev_or (a,b)

```
d0:31 ← a0:31 | b0:31 //Bitwise OR
d32:63 ← a32:63 | b32:63// Bitwise OR
```

Each element of parameters a and b is bitwise ORed. The result is placed in the corresponding element of parameter d.

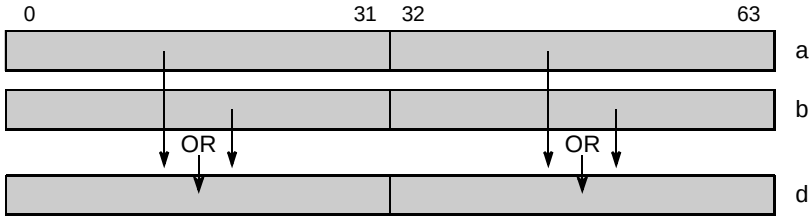

Figure 3-184. Vector OR (__ev_or)

Simplified mnemonic: **evmr** d,a handles moving of the full 64-bit SPE register.

evmr d,a equivalent to **evor** d,a,a

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	**evor d,a,b**

__ev_orc __ev_orc

Vector OR with Complement

d = __ev_orc (a,b)

```
d0:31 ← a0:31 | (¬b0:31) // Bitwise ORC
d32:63 ← a32:63 | (¬b32:63) // Bitwise ORC
```

Each element of parameter a is bitwise ORed with the complement of parameter b. The result is placed in the corresponding element of parameter d.

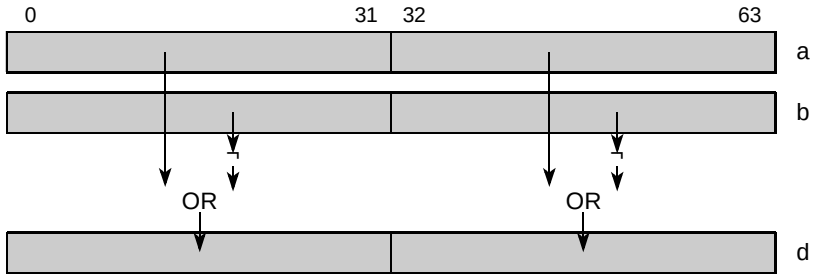

Figure 3-185. Vector OR with Complement (__ev_orc)

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	**evorc d,a,b**

__ev_rlw __ev_rlw

Vector Rotate Left Word

d = __ev_rlw(a,b)

```
nh ← b27:31
nl ← b59:63
d0:31 ← ROTL(a0:31, nh)
d32:63 ← ROTL(a32:63, nl)
```

Each of the high and low elements of parameter a is rotated left by an amount specified in parameter b. The result is placed into parameter d. Rotate values for each element of parameter a are found in bit positions b[27–31] and b[59–63].

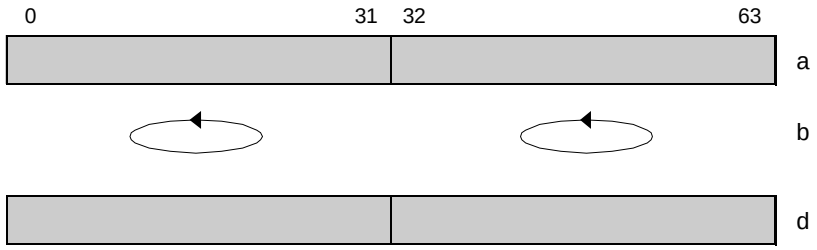

Figure 3-186. Vector Rotate Left Word (__ev_rlw)

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	**evrlw d,a,b**

__ev_rlwi __ev_rlwi

Vector Rotate Left Word Immediate

d = __ev_rlwi (a,b)

```
n ← UIMM
d0:31 ← ROTL(a0:31, n)
d32:63 ← ROTL(a32:63, n)
```

Both the high and low elements of parameter a are rotated left by an amount specified by a 5-bit immediate value.

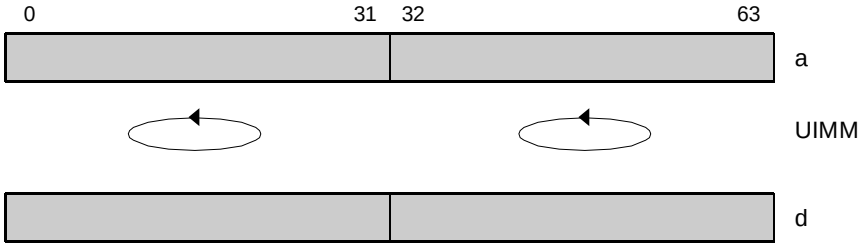

Figure 3-187. Vector Rotate Left Word Immediate (__ev_rlwi)

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	5-bit unsigned	**evrlwi d,a,b**

__ev_rndw __ev_rndw

Vector Round Word

d = __ev_rndw(a)

```
d0:31 ← (a0:31+0x00008000) & 0xFFFF0000 // Modulo sum
d32:63 ← (a32:63+0x00008000) & 0xFFFF0000 // Modulo sum
```

The 32-bit elements of parameter a are rounded into 16 bits. The result is placed into parameter d. The resulting 16 bits are placed in the most significant 16 bits of each element of parameter d, zeroing out the low order 16 bits of each element.

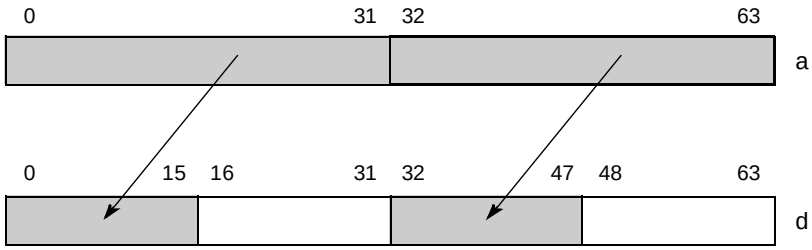

Figure 3-188. Vector Round Word (__ev_rndw)

d	a	Maps to
__ev64_opaque	__ev64_opaque	**evrndw d,a**

__ev_select_eq __ev_select_eq

Vector Select Equal

e = __ev_select_eq(a,b,c,d)

```
if  (a0:31 = b0:31) then e0:31 ← c0:31
else e0:31 ← d0:31

if (a32:63 = b32:63) then e32:63 ← c32:63
else e32:63 ← d32:63
```

This intrinsic returns a concatenated value of the upper and lower bits of parameters c or d based on the sizes of the upper and lower bits of parameters a and b. The __ev_select_* functions work like the ? : operator in C. For example, the aforementioned intrinsic maps to the following logical expression: a = b? c : d.

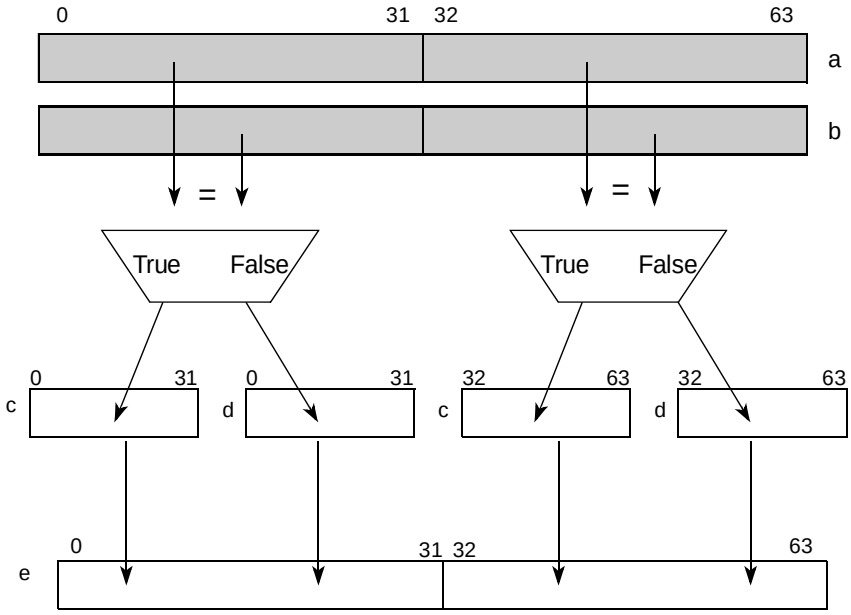

Figure 3-189. Vector Select Equal (__ev_select_eq)

e	a	b	c	d	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	__ev64_opaque	__ev64_opaque	**evcmpeq x,a,b** **evsel e,c,d,x**

__ev_select_fs_eq __ev_select_fs_eq

Vector Select Floating-Point Equal

e = __ev_select_fs_eq(a,b,c,d)

```
if  (a0:31 = b0:31) then e0:31 ← c0:31
else e0:31 ← d0:31

if (a32:63 = b32:63) then e32:63 ← c32:63
else e32:63 ← d32:63
```

This intrinsic returns a concatenated value of the upper and lower bits of parameter c or d based on the sizes of the upper and lower bits of parameters a and b. The __ev_select_* functions work like the ? : operator in the C programming language. For example, the aforementioned intrinsic maps to the following logical expression: a = b? c : d.

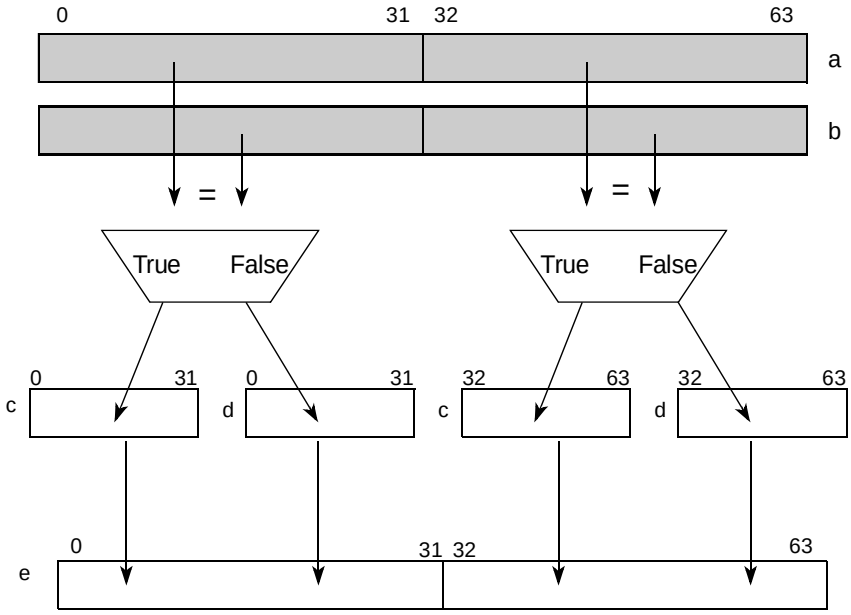

Figure 3-190. Vector Select Floating-Point Equal (__ev_select_fs_eq)

e	a	b	c	d	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	__ev64_opaque	__ev64_opaque	**evfscmpeq x,a,b** **evsel e,c,d,x**

__ev_select_fs_gt __ev_select_fs_gt

Vector Select Floating-Point Greater Than

e = ___ev_select_fs_gt(a,b,c,d)

```
if  (a0:31 > b0:31) then e0:31 ← c0:31
else e0:31 ← d0:31

if (a32:63 > b32:63) then e32:63 ← c32:63
else e32:63 ← d32:63
```

This intrinsic returns a concatenated value of the upper and lower bits of parameter c or d based on the sizes of the upper and lower bits of parameters a and b. The __ev_select_* functions work like the ? : operator in C. For example, the aforementioned intrinsic maps to the following logical expression: a > b ? c : d.

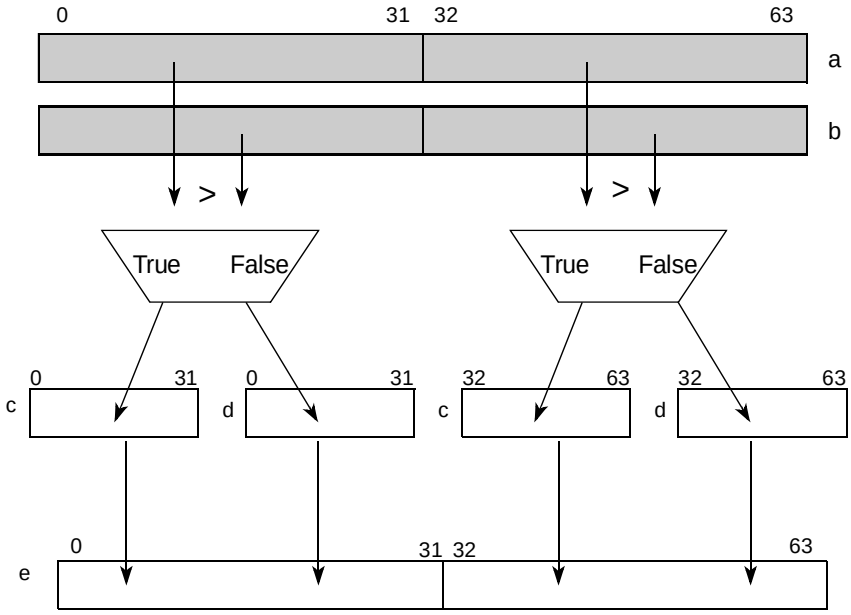

Figure 3-191. Vector Select Floating-Point Greater Than (__ev_select_fs_gt)

e	a	b	c	d	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	__ev64_opaque	__ev64_opaque	**evfscmpgt x,a,b** **evsel e,c,d,x**

__ev_select_fs_lt __ev_select_fs_lt

Vector Select Floating-Point Less Than

e = __ev_select_fs_lt(a,b,c,d)

```
if  (a0:31 < b0:31) then e0:31 ← c0:31
else e0:31 ← d0:31

if (a32:63 < b32:63) then e32:63 ← c32:63
else e32:63 ← d32:63
```

This intrinsic returns a concatenated value of the upper and lower bits of parameter c or d based on the sizes of the upper and lower bits of parameters a and b. The __ev_select_* functions work like the ? : operator in C. For example, the aforementioned intrinsic maps to the following logical expression: a < b? c : d.

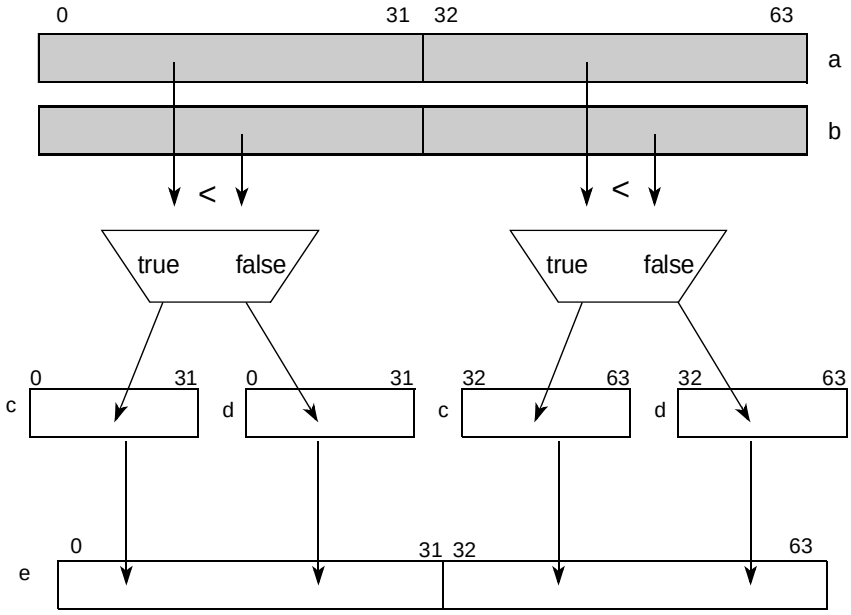

Figure 3-192. Vector Select Floating-Point Less Than (__ev_select_fs_lt)

e	a	b	c	d	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	__ev64_opaque	__ev64_opaque	**evfscmplt x,a,b** **evsel e,c,d,x**

__ev_select_fs_tst_eq __ev_select_fs_tst_eq

Vector Select Floating-Point Test Equal

e = __ev_select_fs_tst_eq(a,b,c,d)

```
if  (a0:31 = b0:31) then e0:31 ← c0:31
else e0:31 ← d0:31

if (a32:63 = b32:63) then e32:63 ← c32:63
else e32:63 ← d32:63
```

This intrinsic returns a concatenated value of the upper and lower bits of parameter c or d based on the sizes of the upper and lower bits of parameters a and b. The __ev_select_* functions work like the ? : operator in C. For example, the aforementioned intrinsic maps to the following logical expression: a =b? c : d. This intrinsic differs from __ev_select_fs_eq because no exceptions are taken during its execution. If strict IEEE 754 compliance is required, use __ev_select_fs_eq instead.

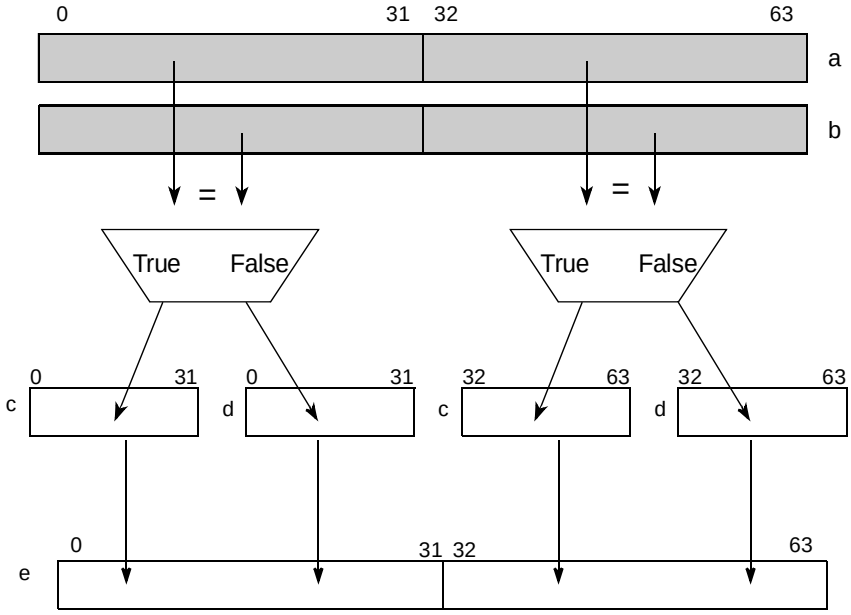

Figure 3-193. Vector Select Floating-Point Test Equal (__ev_select_fs_tst_eq)

e	a	b	c	d	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	__ev64_opaque	__ev64_opaque	**evfststeq x,a,b** **evsel e,c,d,x**

__ev_select_fs_tst_gt __ev_select_fs_tst_gt

Vector Select Floating-Point Test Greater Than

e = ___ev_select_fs_tst_gt(a,b,c,d)

```
if  (a0:31 > b0:31) then e0:31 ← c0:31
else e0:31 ← d0:31

if (a32:63 > b32:63) then e32:63 ← c32:63
else e32:63 ← d32:63
```

This intrinsic returns a concatenated value of the upper and lower bits of parameter c or d based on the sizes of the upper and lower bits of parameters a and b. The __ev_select_* functions work like the ? : operator in C. For example, the aforementioned intrinsic maps to the following logical expression: a > b ? c : d. This intrinsic differs from __ev_select_fs_gt because no exceptions are taken during its execution. If strict IEEE 754 compliance is required, use __ev_select_fs_gt instead.

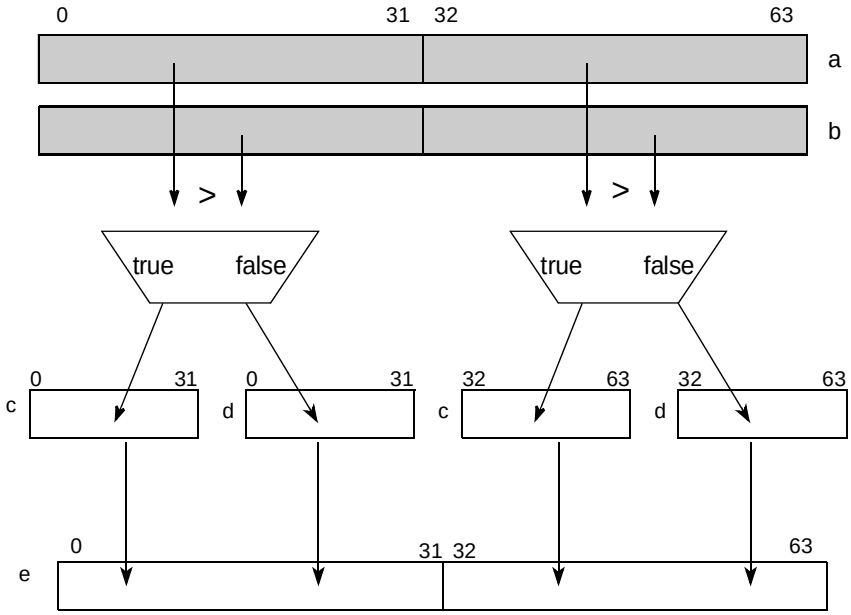

Figure 3-194. Vector Select Floating-Point Test Greater Than (__ev_select_fs_tst_gt)

e	a	b	c	d	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	__ev64_opaque	__ev64_opaque	**evfststgt x,a,b** **evsel e,c,d,x**

__ev_select_fs_tst_lt __ev_select_fs_tst_lt

Vector Select Floating-Point Test Less Than

e = __ev_select_fs_tst_lt(a,b,c,d)

```
if  (a0:31 < b0:31) then e0:31 ← c0:31
else e0:31 ← d0:31

if (a32:63 < b32:63) then e32:63 ← c32:63
else e32:63 ← d32:63
```

This intrinsic returns a concatenated value of the upper and lower bits of parameter c or d based on the sizes of the upper and lower bits of parameters a and b. The __ev_select_* functions work like the ? : operator in C. For example, the aforementioned intrinsic maps to the following logical expression: a < b? c : d. This intrinsic differs from __ev_select_fs_lt because no exceptions are taken during its execution. If strict IEEE 754 compliance is required, use __ev_select_fs_lt instead.

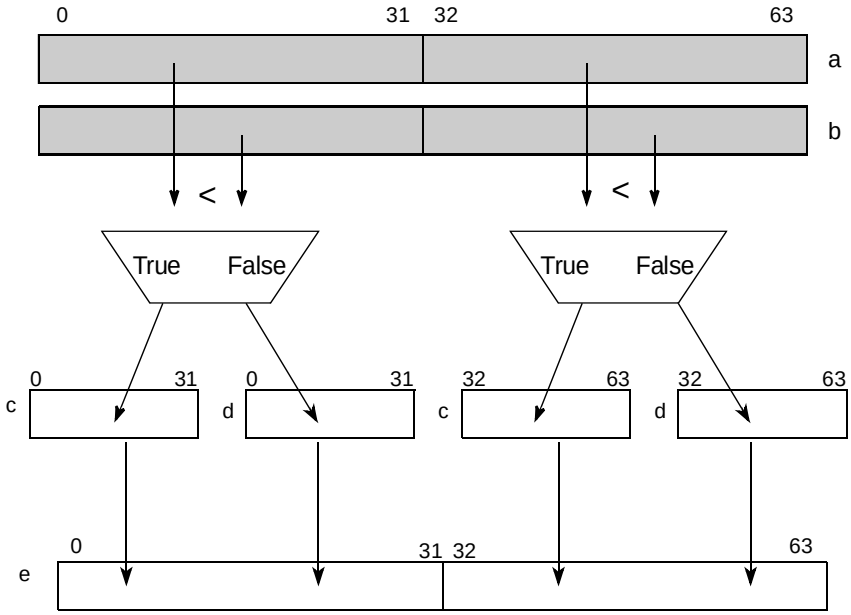

Figure 3-195. Vector Select Floating-Point Test Less Than (__ev_select_fs_tst_lt)

e	a	b	c	d	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	__ev64_opaque	__ev64_opaque	**evfststlt x,a,b** **evsel e,c,d,x**

__ev_select_gts __ev_select_gts

Vector Select Greater Than Signed

e = __ev_select_gts(a,b,c,d)

```
if  (a0:31 >signed b0:31) then e0:31 ← c0:31
else e0:31 ← d0:31

if (a32:63 >signed b32:63) then e32:63← c32:63
else e32:63 ← d32:63
```

This intrinsic returns a concatenated value of the upper and lower bits of parameter c or d based on the sizes of the upper and lower bits of parameters a and b. The __ev_select_* functions work like the ? : operator in C. For example, the aforementioned intrinsic maps to the following logical expression: a > b ? c : d.

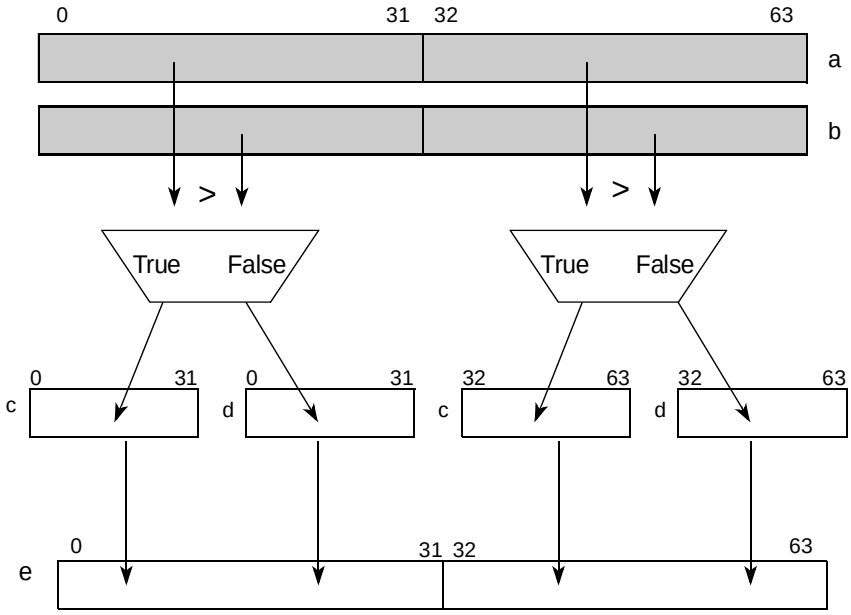

Figure 3-196. Vector Select Greater Than Signed (__ev_select_gts)

e	a	b	c	d	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	__ev64_opaque	__ev64_opaque	**evcmpgts x,a,b** **evsel e,c,d,x**

__ev_select_gtu __ev_select_gtu

Vector Select Greater Than Unsigned

e = __ev_select_gtu(a,b,c,d)

```
if  (a0:31 > unsigned c0:31) then e0:31 ← c0:31
else e0:31 ← d0:31

if (a32:63 > unsigned b32:63) then e32:63 ← c32:63
else e32:63 ← d32:63
```

This intrinsic returns a concatenated value of the upper and lower bits of parameter c or d based on the sizes of the upper and lower bits of parameters a and b. The __ev_select_* functions work like the ? : operator in C. For example, the aforementioned intrinsic maps to the following logical expression: a > b? c : d.

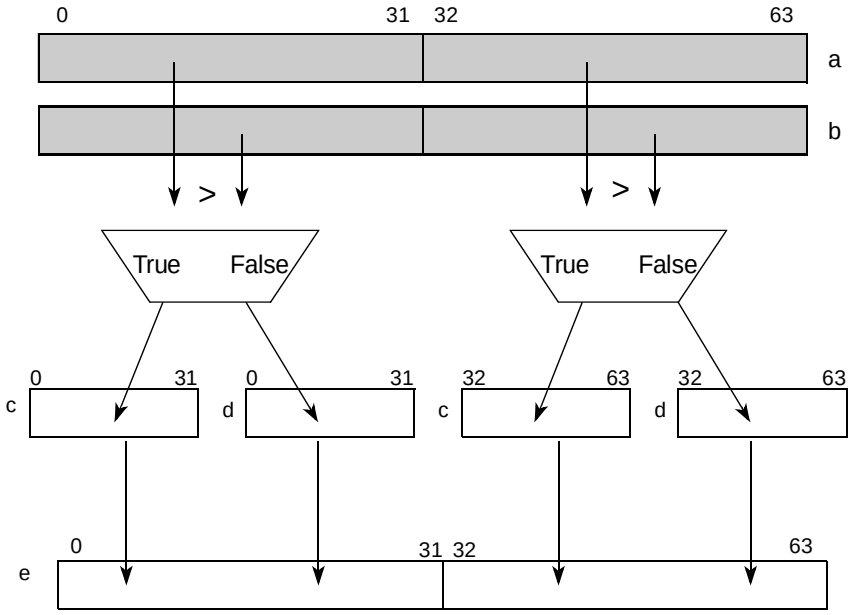

Figure 3-197. Vector Select Greater Than Unsigned (__ev_select_gtu)

e	A	B	C	D	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	__ev64_opaque	__ev64_opaque	**evcmpgtu x,a,b** **evsel e,c,d,x**

__ev_select_lts __ev_select_lts

Vector Select Less Than Signed

e = __ev_select_lts(a,b,c,d)

```
if  (a0:31 <signed b0:31) then e0:31 ← c0:31
else e0:31 ← d0:31

if (a32:63 <signed b32:63) then e ← c32:63
else e ← d32:63
```

This intrinsic returns a concatenated value of the upper and lower bits of parameter c or d based on the sizes of the upper and lower bits of parameters a and b. The __ev_select_* functions work like the ? : operator in C. For example, the aforementioned intrinsic maps to the following logical expression: a < b? c : d.

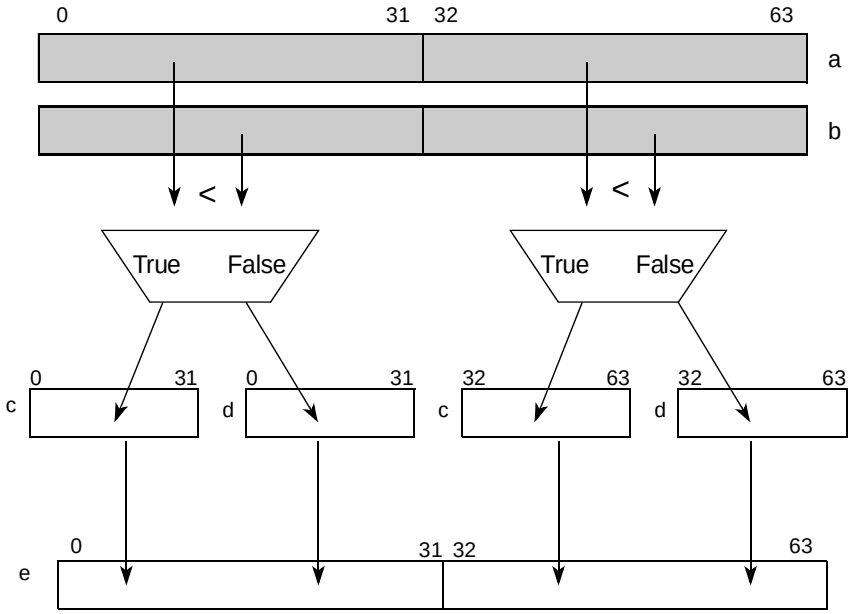

Figure 3-198. Vector Select Less Than Signed (__ev_select_lts)

e	a	b	c	d	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	__ev64_opaque	__ev64_opaque	**evcmplts x,a,b** **evsel e,c,d,x**

__ev_select_ltu __ev_select_ltu

Vector Select Less Than Unsigned

e = __ev_select_ltu(a,b,c,d)

```
if  (a0:31 <<unsigned b0:31) then e0:31 ← c0:31
else e0:31 ← d0:31

if (a32:63 <unsigned b32:63) then e32:63 ← c32:63
else e32:63 ← d32:63
```

This intrinsic returns a concatenated value of the upper and lower bits of parameter c or d based on the sizes of the upper and lower bits of parameters a and b. The __ev_select_* functions work like the ? : operator in C. For example, the aforementioned intrinsic maps to the following logical expression: a < b? c : d.

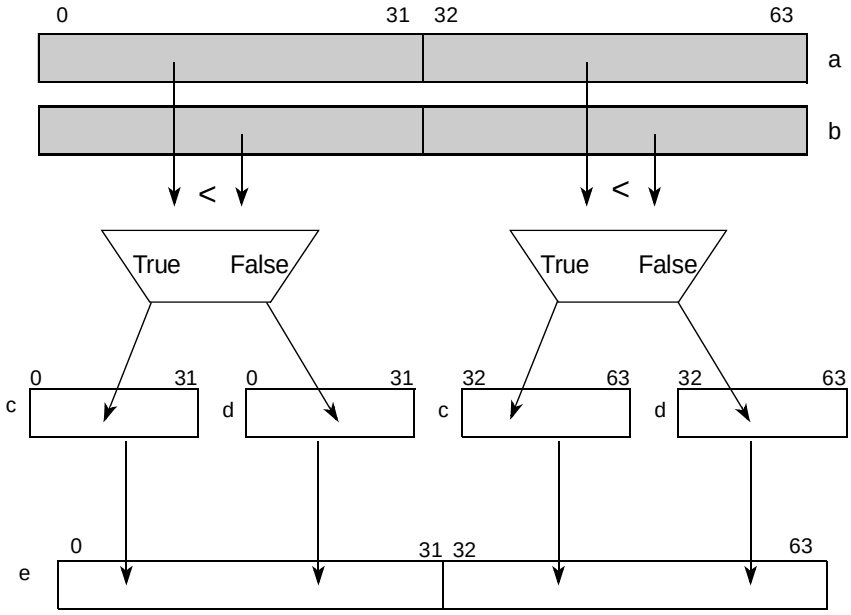

Figure 3-199. Vector Select Less Than Unsigned (__ev_select_ltu)

e	a	b	c	d	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	__ev64_opaque	__ev64_opaque	**evcmpltu x,a,b** **evsel e,c,d,x**

__ev_slw __ev_slw

Vector Shift Left Word

d = __ev_slw (a,b)

```
nh ← b26:31
nl ← b58:63
d0:31 ← SL(a0:31, nh)
d32:63 ← SL(a32:63, nl)
```

Each of the high and low elements of parameter a are shifted left by an amount specified in parameter b. The result is placed into parameter d. The separate shift amounts for each element are specified by 6 bits in parameter b that lie in bit positions 26–31 and 58–63.

Shift amounts from 32 to 63 give a zero result.

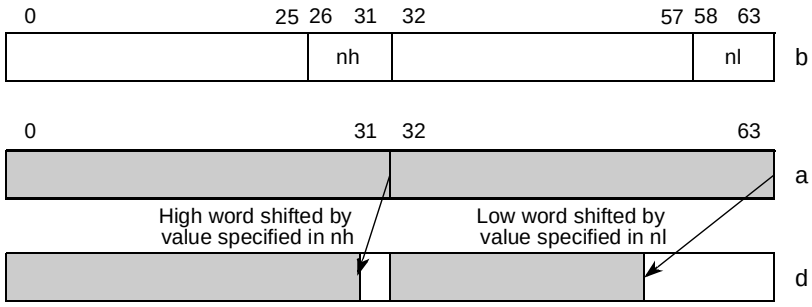

Figure 3-200. Vector Shift Left Word (__ev_slw)

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	**evslw d,a,b**

__ev_slwi __ev_slwi

Vector Shift Left Word Immediate

d = __ev_slwi (a,b)

```
n ← UIMM
d0:31 ← SL(a0:31, n)
d32:63 ← SL(a32:63, n)
```

Both high and low elements of parameter a are shifted left by the 5-bit UIMM value, and the results are placed in parameter d.

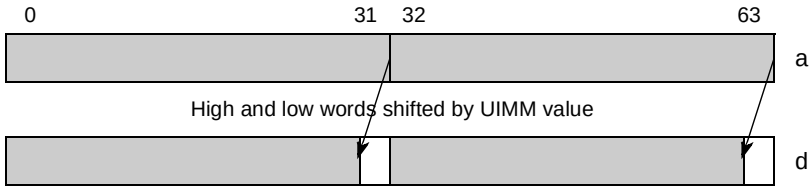

Figure 3-201. Vector Shift Left Word Immediate (__ev_slwi)

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	5-bit unsigned	**evslwi d,a,b**

__ev_splatfi __ev_splatfi

Vector Splat Fractional Immediate

d = __ev_splatfi(a)

$d_{0:31} \leftarrow SIMM \;||\; {}^{27}0$
$d_{32:63} \leftarrow SIMM \;||\; {}^{27}0$

The 5-bit immediate value is padded with trailing zeros and placed in both elements of parameter d, as shown in Figure 3-202. The SIMM ends up in bit positions d[0–4] and d[32–36].

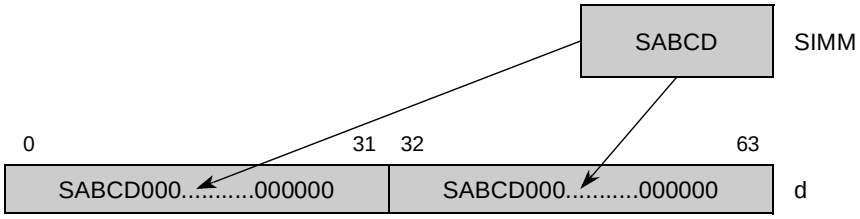

Figure 3-202. Vector Splat Fractional Immediate (__ev_splatfi)

d	a	Maps to
__ev64_opaque	5-bit signed	**evsplatfi d,a**

__ev_splati __ev_splati

Vector Splat Immediate

d = __ev_splati (a)

```
d0:31 ← EXTS(SIMM)
d32:63 ← EXTS(SIMM)
```

The 5-bit immediate value is sign-extended and placed in both elements of parameter d, as shown in Figure 3-203.

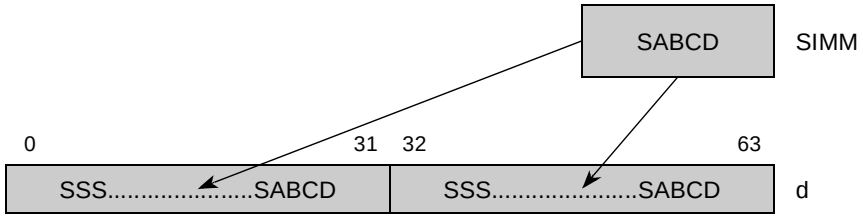

Figure 3-203. __ev_splati Sign Extend

d	a	Maps to
__ev64_opaque	5-bit signed	**evsplati d,a**

__ev_srwis __ev_srwis

Vector Shift Right Word Immediate Signed

d = __ev_srwis(a,b)

```
n ← UIMM
d0:31 ← EXTS(a0:31-n)
d32:63 ← EXTS(b32:63-n)
```

Both high and low elements of parameter a are shifted right by the 5-bit UIMM value. Bits in the most significant positions vacated by the shift are filled with a copy of the sign bit.

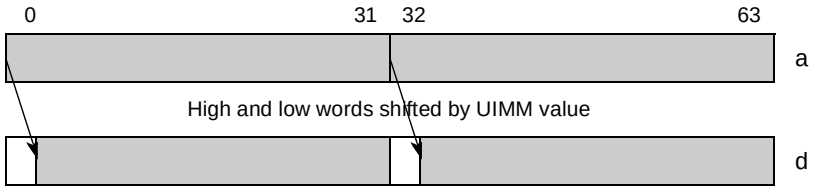

Figure 3-204. Vector Shift Right Word Immediate Signed (__ev_srwis)

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	5-bit unsigned	**evsrwis d,a,b**

__ev_srwiu __ev_srwiu

Vector Shift Right Word Immediate Unsigned

d = __ev_srwiu(a,b)

```
n ← UIMM
d0:31 ← EXTZ(a0:31-n)
d32:63 ← EXTZ(a32:63-n)
```

Both high and low elements of parameter a are shifted right by the 5-bit UIMM value; 0 bits are shifted in to the most significant position. Bits in the most significant positions vacated by the shift are filled with a zero bit.

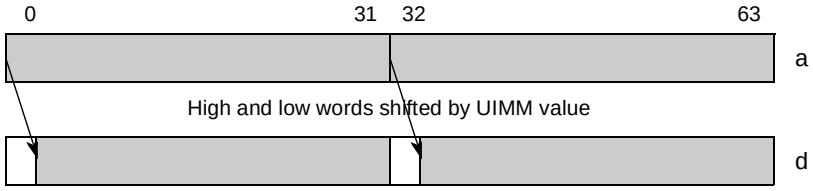

Figure 3-205. Vector Shift Right Word Immediate Unsigned (__ev_srwiu)

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	5-bit unsigned	**evsrwiu d,a,b**

__ev_srws __ev_srws

Vector Shift Right Word Signed

d = __ev_srws (a,b)

```
nh ← b26:31
nl ← b58:63
d0:31 ← EXTS(a0:31-nh)
d32:63 ← EXTS(a32:63-nl)
```

Both the high and low elements of parameter a are shifted right by an amount specified in parameter b. The result is placed into parameter d. The separate shift amounts for each element are specified by 6 bits in parameter b that lie in bit positions 26–31 and 58–63. The sign bits are shifted in to the most significant position.

Shift amounts from 32 to 63 give a result of 32 sign bits.

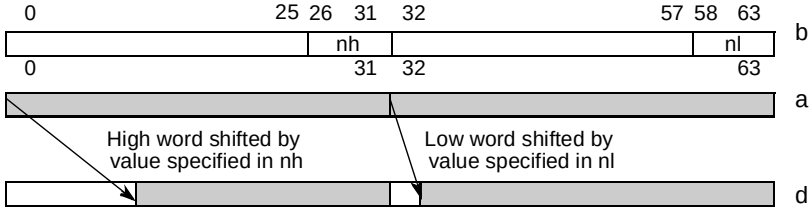

Figure 3-206. Vector Shift Right Word Signed (__ev_srws)

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	**evsrws d,a,b**

__ev_srwu __ev_srwu

Vector Shift Right Word Unsigned

d = __ev_srwu (a,b)

```
nh ← b26:31
nl ← b58:63
d0:31 ← EXTZ(a0:31-nh)
d32:63 ← EXTZ(a32:63-nl)
```

Both the high and low elements of parameter a are shifted right by an amount specified in parameter b. The result is placed into parameter d. The separate shift amounts for each element are specified by 6 bits in parameter b that lie in bit positions 26–31 and 58–63. Zero bits are shifted in to the most significant position.

Shift amounts from 32 to 63 give a zero result.

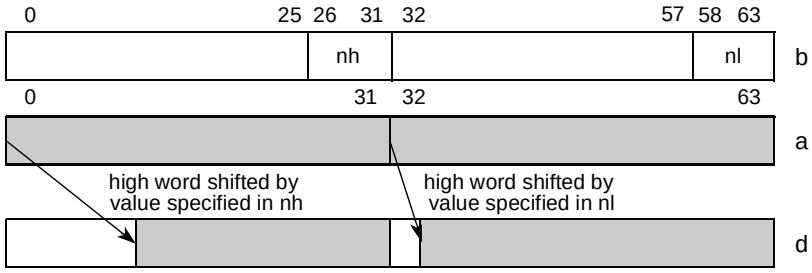

Figure 3-207. Vector Shift Right Word Unsigned (__ev_srwu)

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	**evsrwu d,a,b**

__ev_stdd __ev_stdd

Vector Store Double of Double

d = __ev_stdd (a,b,c)

```
if (a = 0) then temp ← 0
else temp ← (a)
EA ← temp + EXTZ(UIMM*8)
MEM(EA,8) ← RS0:63
```

The contents of **r**S are stored as a double word in storage addressed by EA.

Figure 3-208 shows how bytes are stored in memory as determined by the endian mode.

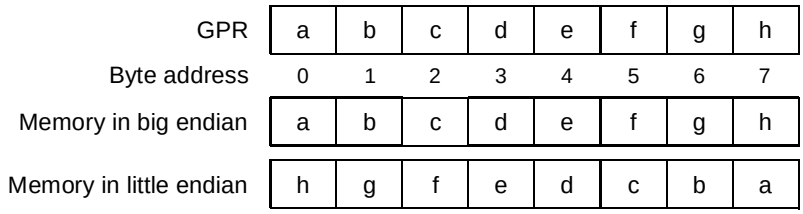

Figure 3-208. __ev_stdd Results in Big- and Little-Endian Modes

NOTE

During implementation, an alignment exception occurs if the EA is not double-word aligned.

d	a	b	c	Maps to
void	__ev64_opaque	__ev64_opaque	5-bit unsigned	**evstdd d,a,b,c**

__ev_stddx __ev_stddx

Vector Store Double of Double Indexed

d = __ev_stddx (a,b,c)

```
if (a = 0) then temp ← 0
else temp ← (a)
EA ← temp + (b)
MEM(EA,8) ← RS0:63
```

The contents of **r**S are stored as a double word in storage addressed by EA.

Figure 3-209 shows how bytes are stored in memory as determined by the endian mode.

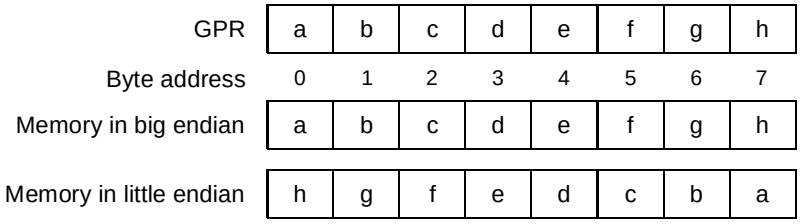

Figure 3-209. __ev_stdd[x] Results in Big- and Little-Endian Modes

NOTE

During implementation, an alignment exception occurs if the EA is not double-word aligned.

d	a	b	c	Maps to
void	__ev64_opaque	__ev64_opaque	int32_t	**evstddx d,a,b,c**

__ev_stdh __ev_stdh

Vector Store Double of Four Half Words

d = __ev_stdh (a,b,c)

```
if (a = 0) then temp ← 0
else temp ← a
EA ← temp + EXTZ(C*8)
MEM(EA,2) ← RS0:15
MEM(EA+2,2) ← RS16:31
MEM(EA+4,2) ← RS32:47
MEM(EA+6,2) ← RS48:63
```

The contents of **r**S are stored as four half words in storage addressed by EA.

Figure 3-210 shows how bytes are stored in memory as determined by the endian mode.

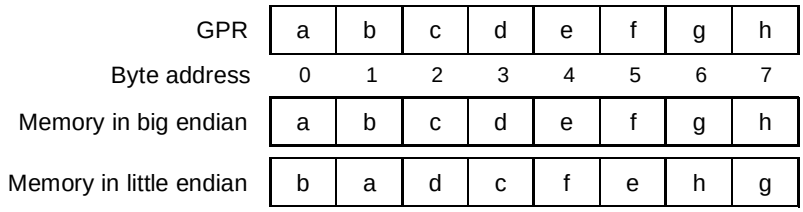

Figure 3-210. __ev_stdh Results in Big- and Little-Endian Modes

NOTE

During implementation, an alignment exception occurs if the EA is not double-word aligned.

d	a	b	c	Maps to
void	__ev64_opaque	__ev64_opaque	5-bit unsigned	**evstdh d,a,b,c**

__ev_stdhx __ev_stdhx

Vector Store Double of Four Half Words Indexed

d = __ev_stdhx (a,b,c)

```
if (a = 0) then temp ← 0
else temp ←(a)
EA ←temp + (b)
MEM(EA,2) ← RS0:15
MEM(EA+2,2) ← RS16:31
MEM(EA+4,2) ← RS32:47
MEM(EA+6,2) ← RS48:63
```

The contents of **r**S are stored as four half words in storage addressed by EA.

Figure 3-211 shows how bytes are stored in memory as determined by the endian mode.

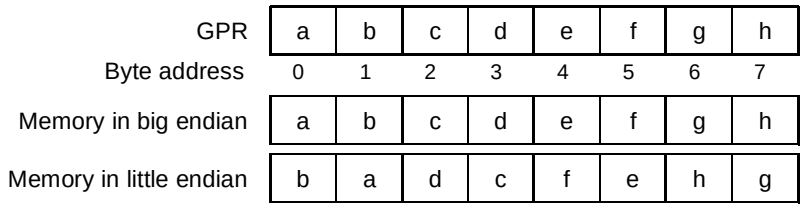

Figure 3-211. __ev_stdhx Results in Big- and Little-Endian Modes

NOTE

During implementation, an alignment exception occurs if the EA is not double-word aligned.

d	a	b	c	Maps to
void	__ev64_opaque	__ev64_opaque	int32_t	**evstdhx d,a,b,c**

__ev_stdw __ev_stdw

Vector Store Double of Two Words

d = __ev_stdw (a,b,c)

```
if (a = 0) then temp ← 0
else temp ←(a)
EA ←temp + EXTZ(UIMM*8)
MEM(EA,4) ← RS0:31
MEM(EA+4,4) ← RS32:63
```

The contents of **r**S are stored as two words in storage addressed by EA.

Figure 3-212 shows how bytes are stored in memory as determined by the endian mode.

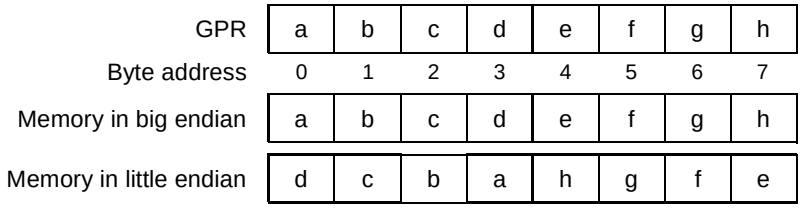

Figure 3-212. __ev_stdw Results in Big- and Little-Endian Modes

NOTE

During implementation, an alignment exception occurs if the EA is not double-word aligned.

d	a	b	c	Maps to
void	__ev64_opaque	__ev64_opaque	5-bit unsigned	**evstdw d,a,b,c**

__ev_stdwx __ev_stdwx

Vector Store Double of Two Words Indexed

d = __ev_stdwx (a,b,c)

```
if (a = 0) then temp ← 0
else temp ←(a)
EA ←temp + (b)
MEM(EA,4) ← RS0:31
MEM(EA+4,4) ← RS32:63
```

The contents of **r**S are stored as two words in storage addressed by EA.

Figure 3-213 shows how bytes are stored in memory as determined by the endian mode.

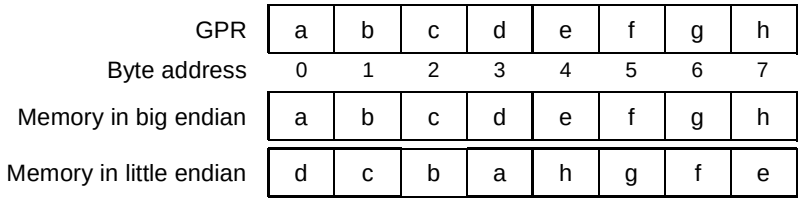

Figure 3-213. __ev_stdwx Results in Big- and Little-Endian Modes

NOTE

During implementation, an alignment exception occurs if the EA is not double-word aligned.

d	a	b	c	Maps to
void	__ev64_opaque	__ev64_opaque	int32_t	**evstdwx d,a,b,c**

__ev_stwhe __ev_stwhe

Vector Store Word of Two Half Words from Even

d = __ev_stwhe (a,b,c)

```
if (a = 0) then temp ← 0
else temp ←(a)
EA ←temp + EXTZ(UIMM*4)
MEM(EA,2) ← RS0:15
MEM(EA+2,2) ← RS32:47
```

The even half words from each element of **r**S are stored as two half words in storage addressed by EA.

Figure 3-214 shows how bytes are stored in memory as determined by the endian mode.

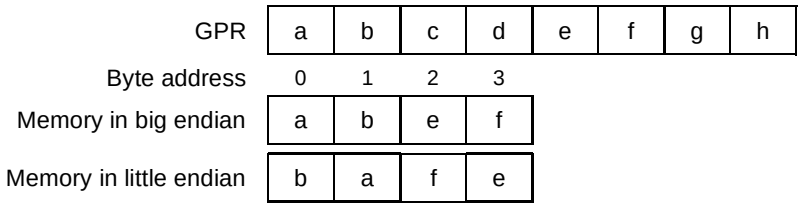

Figure 3-214. __ev_stwhe Results in Big- and Little-Endian Modes

NOTE

During implementation, an alignment exception occurs if the EA is not word-aligned.

d	a	b	c	Maps to
void	__ev64_opaque	uint32_t	5-bit unsigned	**evstdwhe d,a,b**

__ev_stwhex __ev_stwhex

Vector Store Word of Two Half Words from Even Indexed

d = __ev_stwhex (a,b,c)

```
if (a = 0) then temp← 0
else temp ←(a)
EA ←temp + (b)
MEM(EA,2) ← RS0:15
MEM(EA+2,2) ← RS32:47
```

The even half words from each element of **r**S are stored as two half words in storage addressed by EA.

Figure 3-215 shows how bytes are stored in memory as determined by the endian mode.

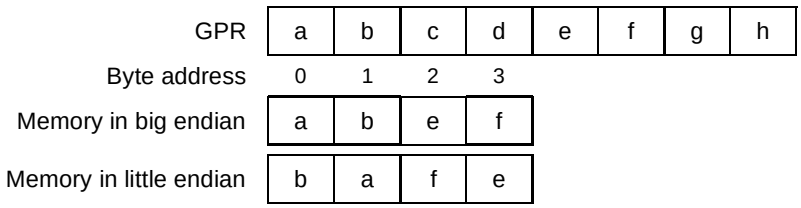

Figure 3-215. __ev_stwhex Results in Big- and Little-Endian Modes

NOTE

During implementation, an alignment exception occurs if the EA is not word-aligned.

d	a	b	c	Maps to
void	__ev64_opaque	uint32_t	int32_t	**evstwhex d,a,b,c**

__ev_stwho __ev_stwho

Vector Store Word of Two Half Words from Odd

d = __ev_stwho (a,b,c)

```
if (a = 0) then temp ← 0
else temp ←(a)
EA ←temp + EXTZ(UIMM*4)
MEM(EA,2) ← RS16:31
MEM(EA+2,2) ← RS48:63
```

The odd half words from each element of **r**S are stored as two half words in storage addressed by EA.

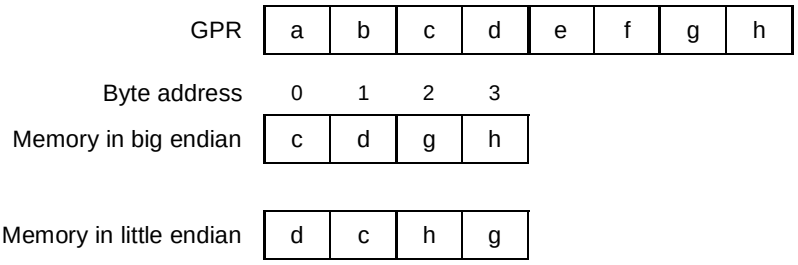

Figure 3-216. __ev_stwho Results in Big- and Little-Endian Modes

d	a	b	c	Maps to
void	__ev64_opaque	uint32_t	5-bit unsigned	**evstwho d,a,b,c**

__ev_stwhox __ev_stwhox

Vector Store Word of Two Half Words from Odd Indexed

d = __ev_stwhox (a,b,c)

```
if (a = 0) then temp ← 0
else temp ←(a)
EA ←temp + (b)
MEM(EA,2) ← RS16:31
MEM(EA+2,2) ← RS48:63
```

The odd half words from each element of **r**S are stored as two half words in storage addressed by EA.

Figure 3-217 shows how bytes are stored in memory as determined by the endian mode.

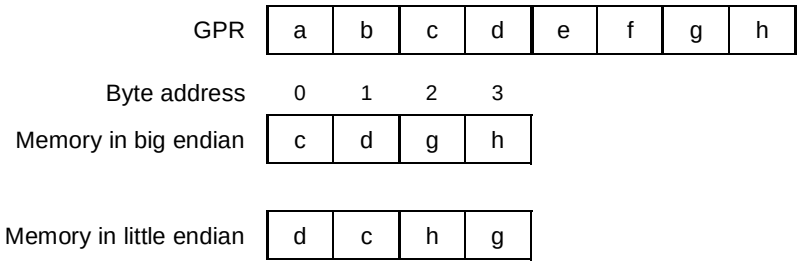

Figure 3-217. __ev_stwhox Results in Big- and Little-Endian Modes

NOTE

During implementation, an alignment exception occurs if the EA is not word-aligned.

d	a	b	c	Maps to
void	__ev64_opaque	uint32_t	int32_t	**evstwhox d,a,b,c**

__ev_stwwe __ev_stwwe

Vector Store Word of Word from Even

d = __ev_stwwe (a,b,c)

```
if (a = 0) then temp ← 0
else temp ←(a)
EA ←temp + EXTZ(UIMM*4)
MEM(EA,4) ← RS0:31
```

The even word of rS is stored in storage addressed by EA.

Figure 3-218 shows how bytes are stored in memory as determined by the endian mode.

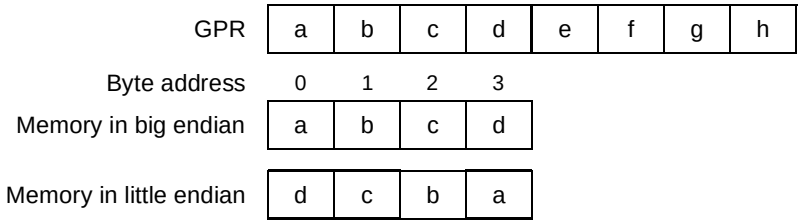

Figure 3-218. __ev_stwwe Results in Big- and Little-Endian Modes

NOTE

During implementation, an alignment exception occurs if the EA is not word-aligned.

d	a	b	c	Maps to
void	__ev64_opaque	uint32_t	5-bit unsigned	**evstwwe d,a,b,c**

__ev_stwwex __ev_stwwex

Vector Store Word of Word from Even Indexed

d = __ev_stwwex (a,b,c)

```
if (a = 0) then temp ← 0
else temp ←(a)
EA ←temp + (b)
MEM(EA,4) ← RS0:31
```

The even word of **r**S is stored in storage addressed by EA.

Figure 3-219 shows how bytes are stored in memory as determined by the endian mode.

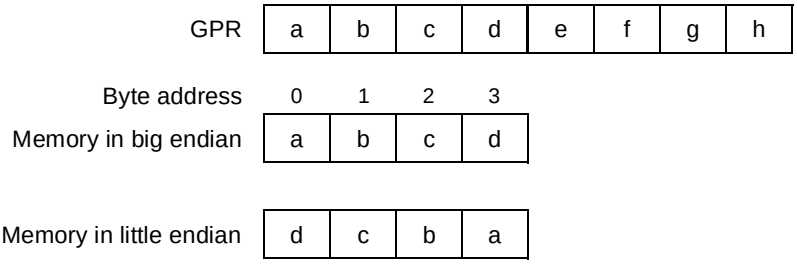

Figure 3-219. __ev_stwwex Results in Big- and Little-Endian Modes

NOTE

During implementation, an alignment exception occurs if the EA is not word-aligned.

d	a	b	c	Maps to
void	__ev64_opaque	uint32_t	int32_t	**evstwwex d,a,b,c**

__ev_stwwo __ev_stwwo

Vector Store Word of Word from Odd

d =__ev_stwwo (a,b,c)

```
if (a = 0) then temp ← 0
else temp ←(a)
EA ←temp + EXTZ(UIMM*4)
MEM(EA,4) ← rS32:63
```

The odd word of **r**S is stored in storage addressed by EA.

Figure 3-220 shows how bytes are stored in memory as determined by the endian mode.

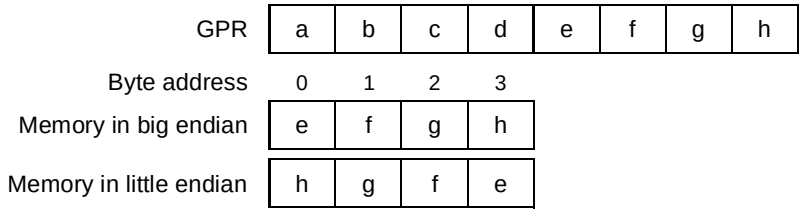

Figure 3-220. __ev_stwwo Results in Big- and Little-Endian Modes

NOTE

During implementation, an alignment exception occurs if the EA is not word-aligned.

d	a	b	c	Maps to
void	__ev64_opaque	uint32_t	5-bit unsigned	**evstwwo d,a,b,c**

__ev_stwwox __ev_stwwox

Vector Store Word of Word from Odd Indexed

d = __ev_stwwox(a,b,c)

```
if (a = 0) then temp ← 0
else temp ←(a)
EA ←temp + (b)
MEM(EA,4) ← rS32:63
```

The odd word of **r**S is stored in storage addressed by EA.

Figure 3-221 shows how bytes are stored in memory as determined by the endian mode.

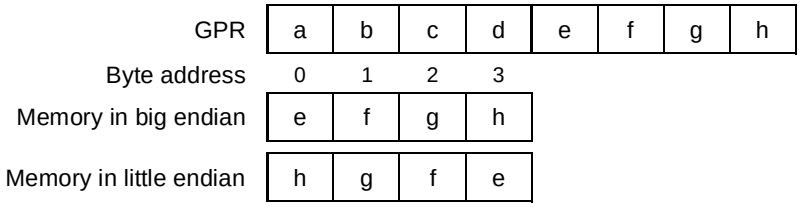

Figure 3-221. __ev_stwwox Results in Big- and Little-Endian Modes

NOTE

During implementation, an alignment exception occurs if the EA is not word-aligned.

d	a	b	c	Maps to
void	__ev64_opaque	uint32_t	int32_t	**evstwwox d,a,b,c**

__ev_subfsmiaaw __ev_subfsmiaaw

Vector Subtract Signed, Modulo, Integer to Accumulator Word

d = __ev_subfsmiaaw(a)

```
// high
d0:31 ← ACC0:31 - a0:31
// low
d32:63 ← ACC32:63 - a32:63
// update accumulator
ACC0:63 ← d0:63
```

Each word element in parameter a is subtracted from the corresponding element in the accumulator and the difference is placed into the corresponding parameter d word and into the accumulator.

Other registers altered: ACC

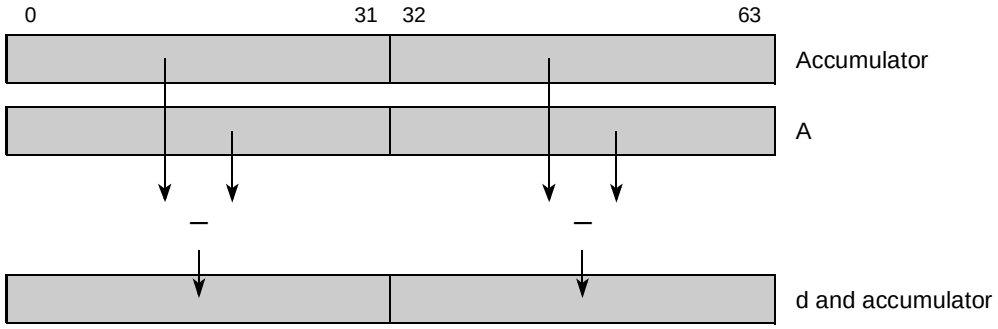

Figure 3-222. Vector Subtract Signed, Modulo, Integer to Accumulator Word (__ev_subfsmiaaw)

d	a	Maps to
__ev64_opaque	__ev64_opaque	**evsubfsmiaaw d,a**

__ev_subfssiaaw __ev_subfssiaaw

Vector Subtract Signed, Saturate, Integer to Accumulator Word

d = __ev_subfssiaaw(a)

```
// high
temp0:63 ← EXTS(ACC0:31) - EXTS(a0:31)
ovh ← temp31 ⊕ temp32
d0:31 ← SATURATE(ovh, temp31, 0x80000000, 0x7fffffff, temp32:63)

// low
temp0:63 ← EXTS(ACC32:63) - EXTS(a32:63)
ovl ← temp31 ⊕ temp32
d32:63 ← SATURATE(ovl, temp31, 0x80000000, 0x7fffffff, temp32:63)

// update accumulator
ACC0:63 ← d0:63

SPEFSCROVH ← ovh
SPEFSCROV ← ovl
SPEFSCRSOVH ← SPEFSCRSOVH | ovh
SPEFSCRSOV ← SPEFSCRSOV | ovl
```

Each signed integer word element in parameter a is sign-extended and subtracted from the corresponding sign-extended element in the accumulator, saturating if overflow occurs, and the results are placed in parameter d and the accumulator. Any overflow is recorded in the SPEFSCR overflow and summary overflow bits.

Other registers altered: SPEFSCR ACC

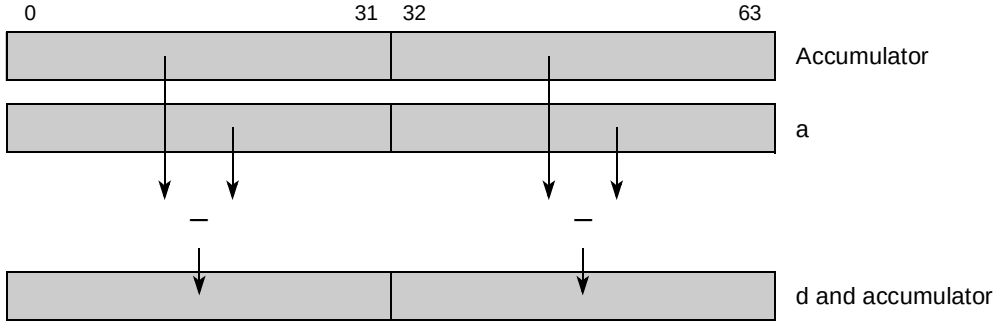

Figure 3-223. Vector Subtract Signed, Saturate, Integer to Accumulator Word (__ev_subfssiaaw)

d	a	Maps to
__ev64_opaque	__ev64_opaque	**evsubfssiaaw d,a**

__ev_subfumiaaw __ev_subfumiaaw

Vector Subtract Unsigned, Modulo, Integer to Accumulator Word

d = __ev_subfumiaaw(a)

```
// high
d0:31 ← ACC0:31 - a0:31
// low
d32:63 ← ACC32:63 - a32:63
// update accumulator
ACC0:63 ← d0:63
```

Each unsigned integer word element in parameter a is subtracted from the corresponding element in the accumulator, and the results are placed in the corresponding parameter d and into the accumulator.

Other registers altered: ACC

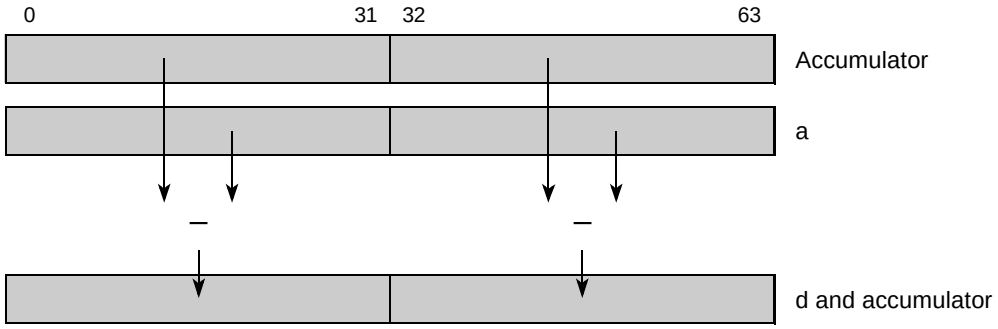

Figure 3-224. Vector Subtract Unsigned, Modulo, Integer to Accumulator Word (__ev_subfumiaaw)

d	a	Maps to
__ev64_opaque	__ev64_opaque	**evsubfumiaaw d,a**

__ev_subfusiaaw __ev_subfusiaaw

Vector Subtract Unsigned, Saturate, Integer to Accumulator Word

d = __ev_subfusiaaw(a)

```
// high
temp0:63 ← EXTZ(ACC0:31) - EXTZ(a0:31)
ovh ← temp31
d0:31 ← SATURATE(ovh, temp31, 0x00000000, 0x00000000, temp32:63)

// low
temp0:63 ← EXTS(ACC32:63) - EXTS(a32:63)
ovl ← temp31
d32:63 ← SATURATE(ovl, temp31, 0x00000000, 0x00000000, temp32:63)

// update accumulator
ACC0:63 ← d0:63

SPEFSCROVH ← ovh
SPEFSCROV ← ovl
SPEFSCRSOVH ← SPEFSCRSOVH | ovh
SPEFSCRSOV ← SPEFSCRSOV | ovl
```

Each unsigned integer word element in parameter a is zero-extended and subtracted from the corresponding zero-extended element in the accumulator, saturating if underflow occurs, and the results are placed in parameter d and the accumulator. Any underflow is recorded in the SPEFSCR overflow and summary overflow bits.

Other registers altered: SPEFSCR ACC

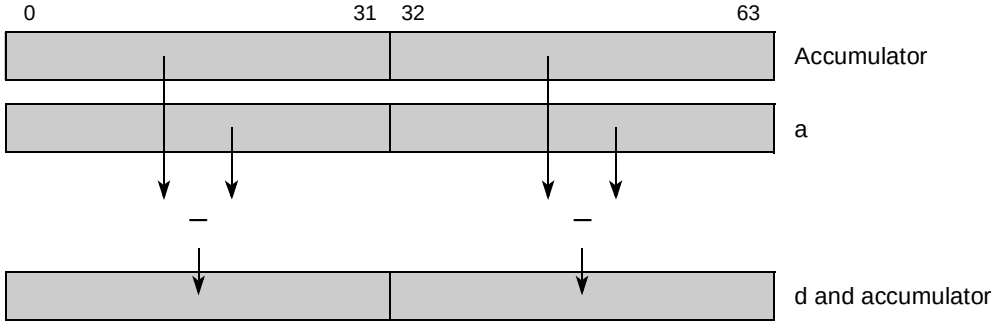

Figure 3-225. Vector Subtract Unsigned, Saturate, Integer to Accumulator Word (__ev_subfusiaaw)

d	a	Maps to
__ev64_opaque	__ev64_opaque	**evsubfusiaaw d,a**

__ev_subfw __ev_subfw

Vector Subtract from Word

d = __ev_subfw(a,b)

```
d0:31 ← b0:31 - a0:31                    // Modulo difference
d32:63 ← b32:63 - a32:63                 // Modulo difference
```

Each signed integer element of parameter a is subtracted from the corresponding element of parameter b, and the results are placed into parameter d.

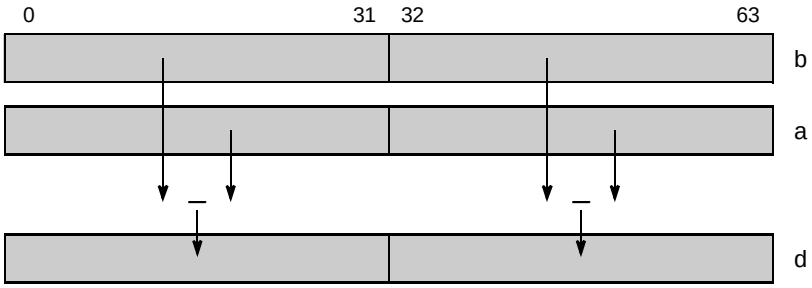

Figure 3-226. Vector Subtract from Word (__ev_subfw)

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	**evsubfw d,a,b**

__ev_subifw __ev_subifw

Vector Subtract Immediate from Word

d = __ev_subifw(a,b)

```
d0:31 ← b0:31 - EXTZ(UIMM) // Modulo difference
d32:63 ← b32:63 - EXTZ(UIMM)// Modulo difference
```

UIMM is zero-extended and subtracted from both the high and low elements of parameter b. Note that the same value is subtracted from both elements of the register. UIMM is 5 bits.

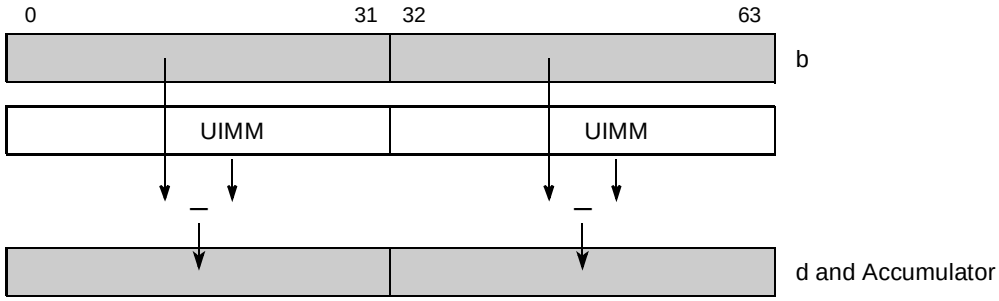

Figure 3-227. Vector Subtract Immediate from Word (__ev_subifw)

d	a	b	Maps to
__ev64_opaque	5-bit unsigned	__ev64_opaque	**evsubifw d,a,b**

__ev_upper_eq __ev_upper_eq

Vector Upper Bits Equal

d = __ev_upper_eq(a,b)

```
if (a0:31 = b0:31) then d ← true
else d ←false
```

This intrinsic returns true if the upper 32 bits of parameter a are equal to the upper 32 bits of parameter b.

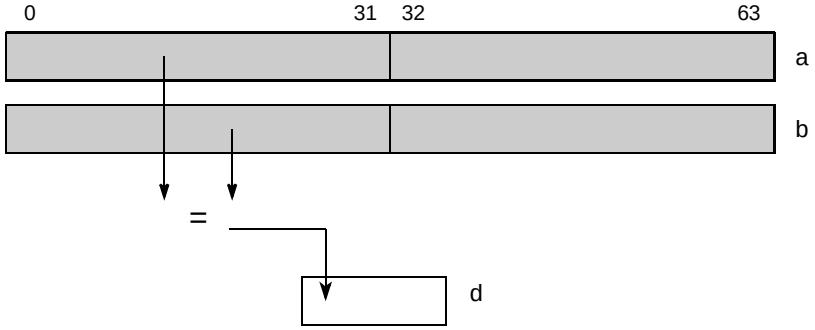

Figure 3-228. Vector Upper Equal(__ev_upper_eq)

d	a	b	Maps to
_Bool	__ev64_opaque	__ev64_opaque	**evcmpeq x,a,b**

__ev_upper_fs_eq __ev_upper_fs_eq

Vector Upper Bits Floating-Point Equal

d = __ev_upper_fs_eq(a,b)

```
if (a0:31 = b0:31) then d ← true
else d ←false
```

This intrinsic returns true if the upper 32 bits of parameter a are equal to the upper 32 bits of parameter b.

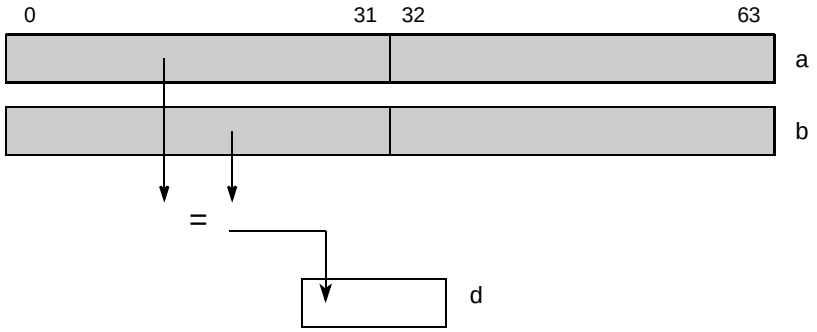

Figure 3-229. Vector Upper Floating-Point Equal(__ev_upper_fs_eq)

d	a	b	Maps to
_Bool	__ev64_opaque	__ev64_opaque	**evfscmpeq x,a,b**

__ev_upper_fs_gt __ev_upper_fs_gt

Vector Upper Bits Floating-Point Greater Than

d = __ev_upper_fs_gt(a,b)

```
if (a0:31 > b0:31) then d ← true
else d ←false
```

This intrinsic returns true if the upper 32 bits of parameter a are greater than the upper 32 bits of parameter b.

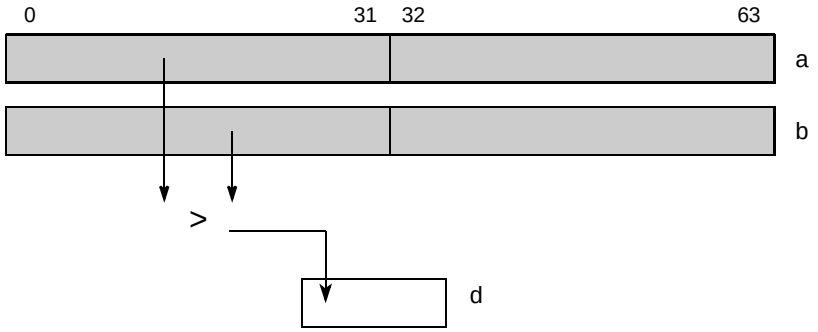

Figure 3-230. Vector Upper Floating-Point Greater Than (__ev_upper_fs_gt)

d	a	b	Maps to
_Bool	__ev64_opaque	__ev64_opaque	**evfscmpgt x,a,b**

__ev_upper_fs_lt __ev_upper_fs_lt

Vector Upper Bits Floating-Point Less Than

d = __ev_upper_fs_lt(a,b)

```
if (a0:31 < b0:31) then d← true
else d ←false
```

This intrinsic returns true if the upper 32 bits of parameter a are less than the upper 32 bits of parameter b.

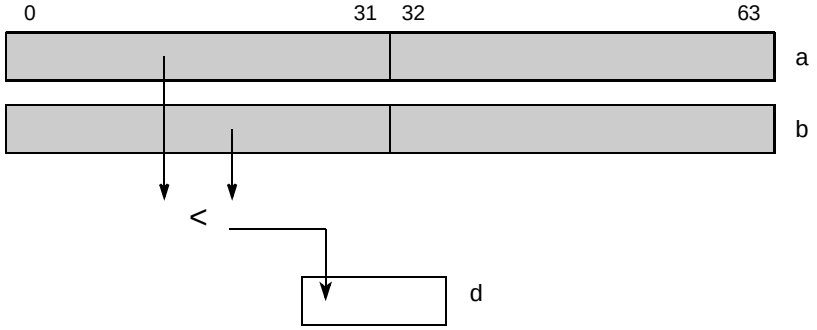

Figure 3-231. Vector Upper Floating-Point Less Than (__ev_upper_fs_lt)

d	a	b	Maps to
_Bool	__ev64_opaque	__ev64_opaque	**evfscmplt x,a,b**

__ev_upper_fs_tst_eq __ev_upper_fs_tst_eq

Vector Upper Bits Floating-Point Test Equal

d = __ev_upper_fs_tst_eq(a,b)

```
if (a0:31 = b0:31) then d ← true
else d ←false
```

This intrinsic returns true if the upper 32 bits of parameter a are equal to the upper 32 bits of parameter b. This intrinsic differs from __ev_upper_fs_eq because no exceptions are taken during its execution. If strict IEEE 754 compliance is required, use __ev_upper_fs_eq instead.

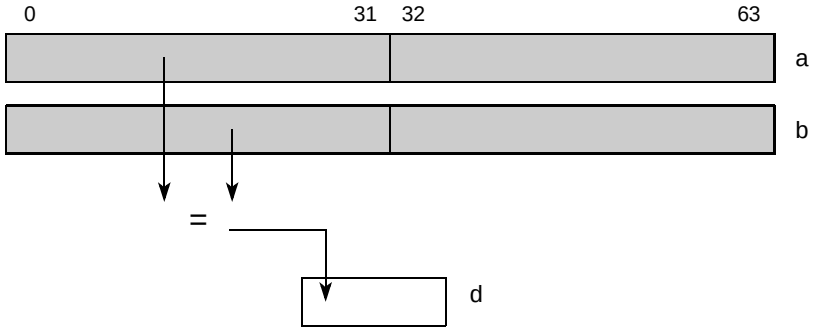

Figure 3-232. Vector Upper Floating-Point Test Equal (__ev_upper_fs_tst_eq)

d	a	b	Maps to
_Bool	__ev64_opaque	__ev64_opaque	**evfststeq x,a,b**

__ev_upper_fs_tst_gt __ev_upper_fs_tst_gt

Vector Upper Bits Floating-Point Test Greater Than

d = __ev_upper_fs_tst_gt(a,b)

```
if (a0:31 > b0:31) then d ← true
else d ←false
```

This intrinsic returns true if the upper 32 bits of parameter a are greater than the upper 32 bits of parameter b. This intrinsic differs from __ev_upper_fs_gt because no exceptions are taken during its execution. If strict IEEE 754 compliance is required, use __ev_upper_fs_gt instead.

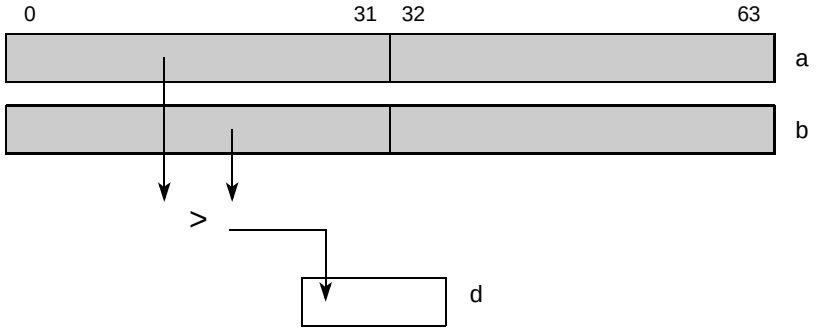

Figure 3-233. Vector Upper Floating-Point Test Greater Than (__ev_upper_fs_tst_gt)

d	a	b	Maps to
_Bool	__ev64_opaque	__ev64_opaque	**evfststgt x,a,b**

__ev_upper_fs_tst_lt __ev_upper_fs_tst_lt

Vector Upper Bits Floating-Point TestLess Than

d = __ev_upper_fs_tst_lt(a,b)

```
if (a0:31 < b0:31) then d ← true
else d ←false
```

This intrinsic returns true if the upper 32 bits of parameter a are less than the upper 32 bits of parameter b. This intrinsic differs from __ev_upper_fs_lt because no exceptions are taken during its execution. If strict IEEE 754 compliance is required, use __ev_upper_fs_lt instead.

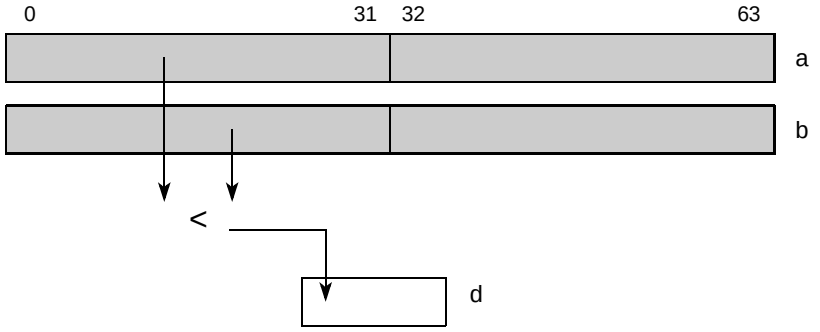

Figure 3-234. Vector Upper Floating-Point Test Less Than (__ev_upper_fs_tst_lt)

d	a	b	Maps to
_Bool	__ev64_opaque	__ev64_opaque	**evfststlt x,a,b**

__ev_upper_gts __ev_upper_gts

Vector Upper Bits Greater Than Signed

d = __ev_upper_gts(a,b)

```
if (a0:31 >signed b0:31) then d ← true
else d ←false
```

This intrinsic returns true if the upper 32 bits of parameter a are greater than the upper 32 bits of parameter b.

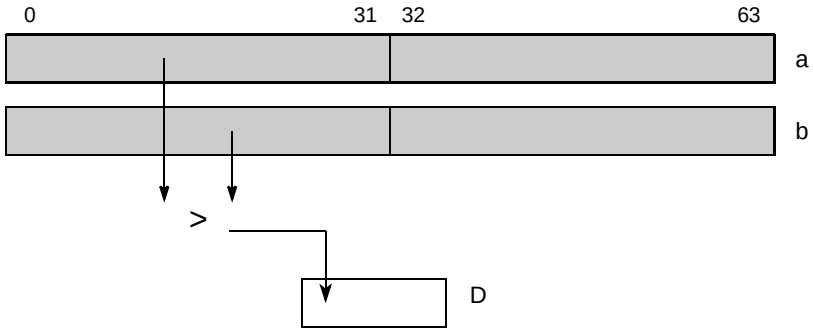

Figure 3-235. Vector Upper Greater Than Signed (__ev_upper_gts)

d	a	b	Maps to
_Bool	__ev64_opaque	__ev64_opaque	**evcmpgts x,a,b**

__ev_upper_gtu __ev_upper_gtu

Vector Upper Bits Greater Than Unsigned

d = __ev_upper_gtu(a,b)

```
if (a0:31 > unsigned b0:31) then d ← true
else d ←false
```

This intrinsic returns true if the upper 32 bits of parameter a are greater than the upper 32 bits of parameter b.

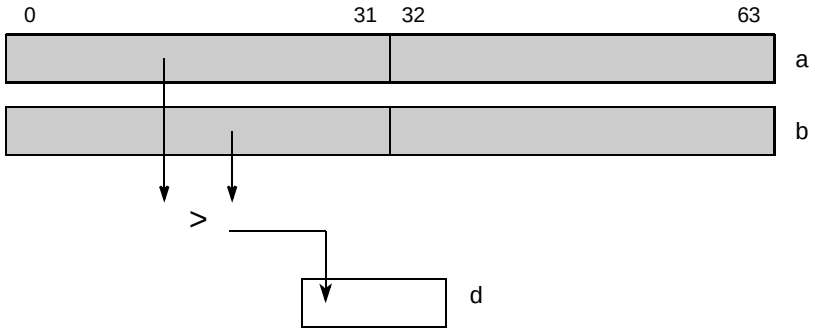

Figure 3-236. Vector Upper Greater Than Unsigned (__ev_upper_gtu)

d	a	b	Maps to
_Bool	__ev64_opaque	__ev64_opaque	**evcmpgtu x,a,b**

__ev_upper_lts __ev_upper_lts

Vector Upper Bits Less Than Signed

d = __ev_upper_lts(a,b)

```
if (a0:31 <signed b0:31) then d ← true
else d ←false
```

This intrinsic returns true if the upper 32 bits of parameter a are less than the upper 32 bits of parameter b.

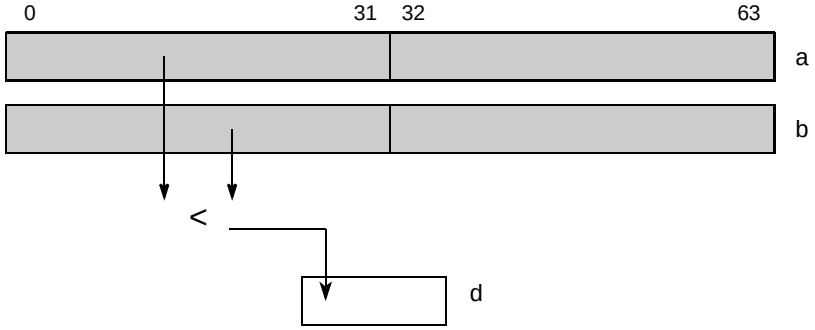

Figure 3-237. Vector Upper Less Than Signed (__ev_upper_lts)

d	a	b	Maps to
_Bool	__ev64_opaque	__ev64_opaque	**evcmplts x,a,b**

__ev_upper_ltu __ev_upper_ltu

Vector Upper Bits Less Than Unsigned

```
d = __ev_upper_ltu(a,b)

if (a0:31 <unsigned b0:31) then d ← true
else d ←false
```

This intrinsic returns true if the upper 32 bits of parameter a are less than the upper 32 bits of parameter b.

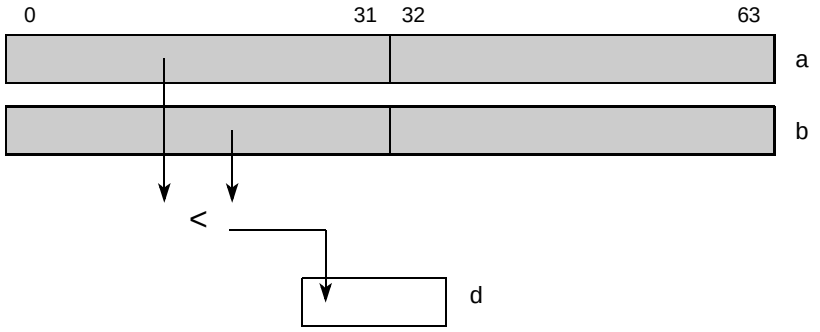

Figure 3-238. Vector Upper Less Than Unsigned (__ev_upper_ltu)

d	a	b	Maps to
_Bool	__ev64_opaque	__ev64_opaque	**evcmpltu x,a,b**

__ev_xor __ev_xor

Vector XOR

d = __ev_xor (a,b)

```
d0:31 ← a0:31 ⊕ b0:31 // Bitwise XOR
d32:63 ← a32:63 ⊕ b32:63// Bitwise XOR
```

Each element of parameters a and b is exclusive-ORed. The results are placed in parameter d.

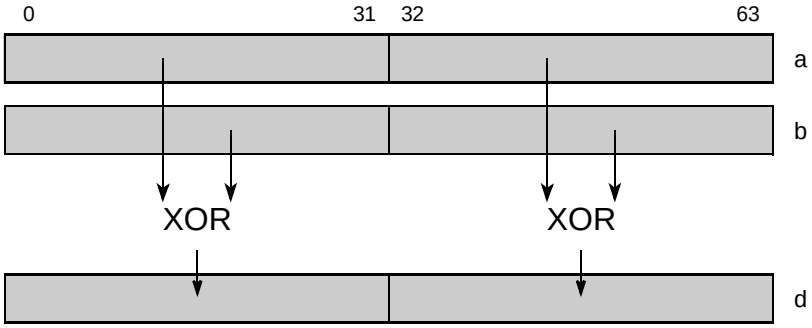

Figure 3-239. Vector XOR (__ev_xor)

d	a	b	Maps to
__ev64_opaque	__ev64_opaque	__ev64_opaque	**evxor d,a,b**

3.6 Basic Instruction Mapping

```
__ev64_opaque__ __ev_addw( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_addiw( __ev64_opaque__ a, 5-bit unsigned literal );

// returns ( B - A )
__ev64_opaque__ __ev_subfw( __ev64_opaque__ a, __ev64_opaque__ b );

// returns ( B - UIMM )
__ev64_opaque__ __ev_subifw( 5-bit unsigned literal, __ev64_opaque__ b );

// returns ( A - B )
__ev64_opaque__ __ev_subw( __ev64_opaque__ a, __ev64_opaque__ b );

// returns ( A - UIMM )
__ev64_opaque__ __ev_subiw( __ev64_opaque__ a, 5-bit unsigned literal );
__ev64_opaque__ __ev_abs( __ev64_opaque__ a );
__ev64_opaque__ __ev_neg( __ev64_opaque__ a );
__ev64_opaque__ __ev_extsb( __ev64_opaque__ a );
__ev64_opaque__ __ev_extsh( __ev64_opaque__ a );
__ev64_opaque__ __ev_and( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_or( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_xor( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_nand( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_nor( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_eqv( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_andc( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_orc( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_rlw( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_rlwi( __ev64_opaque__ a, 5-bit unsigned literal );
__ev64_opaque__ __ev_slw( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_slwi( __ev64_opaque__ a, 5-bit unsigned literal );
__ev64_opaque__ __ev_srws( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_srwu( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_srwis( __ev64_opaque__ a, 5-bit unsigned literal );
__ev64_opaque__ __ev_srwiu( __ev64_opaque__ a, 5-bit unsigned literal );
__ev64_opaque__ __ev_cntlzw( __ev64_opaque__ a );
__ev64_opaque__ __ev_cntlsw( __ev64_opaque__ a );
__ev64_opaque__ __ev_rndw( __ev64_opaque__ a );
__ev64_opaque__ __ev_mergehi( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mergelo( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mergelohi( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mergehilo( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_splati( 5-bit signed literal );
__ev64_opaque__ __ev_splatfi( 5-bit signed literal );
__ev64_opaque__ __ev_divws( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_divwu( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mra( __ev64_opaque__ a ); uint
32_t __brinc( uint32_t a, uint32_t b );

# COMPARE PREDICATES
```

NOTE

The __ev_select_* operations work much like the ? : operator does in C. For example:

```
__ev_select_gts(a,b,c,d) maps to the logical expression a > b ? c : d.
```

The following code shows an example of the assembly code:

```
    evcmpgts crfD, A, B
    evsel ret, C, D, crfD

_Bool __ev_any_gts( __ev64_opaque__ a, __ev64_opaque__ b);
_Bool __ev_all_gts( __ev64_opaque__ a, __ev64_opaque__ b);
_Bool __ev_upper_gts( __ev64_opaque__ a, __ev64_opaque__ b);
_Bool __ev_lower_gts( __ev64_opaque__ a, __ev64_opaque__ b);
__ev64_opaque__ __ev_select_gts( __ev64_opaque__ a, __ev64_opaque__ b,
                      __ev64_opaque__ c, __ev64_opaque__ d);

_Bool __ev_any_gtu(__ev64_opaque__ a, __ev64_opaque__ b);
_Bool __ev_all_gtu( __ev64_opaque__ a, __ev64_opaque__ b);
_Bool __ev_upper_gtu( __ev64_opaque__ a, __ev64_opaque__ b);
_Bool __ev_lower_gtu( __ev64_opaque__ a, __ev64_opaque__ b);
__ev64_opaque__ __ev_select_gtu( __ev64_opaque__ a, __ev64_opaque__ b,
                      __ev64_opaque__ c, __ev64_opaque__ d);
_Bool __ev_any_lts( __ev64_opaque__ a, __ev64_opaque__ b);
_Bool __ev_all_lts( __ev64_opaque__ a, __ev64_opaque__ b);
_Bool __ev_upper_lts( __ev64_opaque__ a, __ev64_opaque__ b);
_Bool __ev_lower_lts( __ev64_opaque__ a, __ev64_opaque__ b);
__ev64_opaque__ __ev_select_lts( __ev64_opaque__ a, __ev64_opaque__ b,
                      __ev64_opaque__ c, __ev64_opaque__ d);

_Bool __ev_any_ltu( __ev64_opaque__ a, __ev64_opaque__ b);
_Bool __ev_all_ltu( __ev64_opaque__ a, __ev64_opaque__ b);
_Bool __ev_upper_ltu( __ev64_opaque__ a, __ev64_opaque__ b);
_Bool __ev_lower_ltu( __ev64_opaque__ a, __ev64_opaque__ b);
__ev64_opaque__ __ev_select_ltu( __ev64_opaque__ a, __ev64_opaque__ b,
                      __ev64_opaque__ c, __ev64_opaque__ d);

_Bool __ev_any_eq( __ev64_opaque__ a, __ev64_opaque__ b);
_Bool __ev_all_eq( __ev64_opaque__ a, __ev64_opaque__ b);
_Bool __ev_upper_eq( __ev64_opaque__ a, __ev64_opaque__ b);
_Bool __ev_lower_eq( __ev64_opaque__ a, __ev64_opaque__ b);
__ev64_opaque__ __ev_select_eq( __ev64_opaque__ a, __ev64_opaque__ b,
                    __ev64_opaque__ c, __ev64_opaque__ d);

_Bool __ev_any_fs_gt( __ev64_opaque__ a, __ev64_opaque__ b);
_Bool __ev_all_fs_gt( __ev64_opaque__ a, __ev64_opaque__ b);
_Bool __ev_upper_fs_gt( __ev64_opaque__ a, __ev64_opaque__ b);
_Bool __ev_lower_fs_gt( __ev64_opaque__ a, __ev64_opaque__ b);
__ev64_opaque__ __ev_select_fs_gt( __ev64_opaque__ a, __ev64_opaque__ b,
                        __ev64_opaque__ c, __ev64_opaque__ d);

_Bool __ev_any_fs_lt( __ev64_opaque__ a, __ev64_opaque__ b);
_Bool __ev_all_fs_lt( __ev64_opaque__ a, __ev64_opaque__ b);
_Bool __ev_upper_fs_lt( __ev64_opaque__ a, __ev64_opaque__ b);
_Bool __ev_lower_fs_lt( __ev64_opaque__ a, __ev64_opaque__ b);
```

```
__ev64_opaque__ __ev_select_fs_lt( __ev64_opaque__ a, __ev64_opaque__ b,
                          __ev64_opaque__ c, __ev64_opaque__ d);

_Bool __ev_any_fs_eq( __ev64_opaque__ a, __ev64_opaque__ b);
_Bool __ev_all_fs_eq( __ev64_opaque__ a, __ev64_opaque__ b);
_Bool __ev_upper_fs_eq( __ev64_opaque__ a, __ev64_opaque__ b);
_Bool __ev_lower_fs_eq( __ev64_opaque__ a, __ev64_opaque__ b);
__ev64_opaque__ __ev_select_fs_eq( __ev64_opaque__ a, __ev64_opaque__ b,
                          __ev64_opaque__ c, __ev64_opaque__ d);

_Bool __ev_any_fs_tst_gt( __ev64_opaque__ a, __ev64_opaque__ b);
_Bool __ev_all_fs_tst_gt( __ev64_opaque__ a, __ev64_opaque__ b);
_Bool __ev_upper_fs_tst_gt( __ev64_opaque__ a, __ev64_opaque__ b);
_Bool __ev_lower_fs_tst_gt( __ev64_opaque__ a, __ev64_opaque__ b);
__ev64_opaque__ __ev_select_fs_tst_gt( __ev64_opaque__ a, __ev64_opaque__ b,
                             __ev64_opaque__ c, __ev64_opaque__ d);

_Bool __ev_any_fs_tst_lt( __ev64_opaque__ a, __ev64_opaque__ b);
_Bool __ev_all_fs_tst_lt( __ev64_opaque__ a, __ev64_opaque__ b);
_Bool __ev_upper_fs_tst_lt( __ev64_opaque__ a, __ev64_opaque__ b);
_Bool __ev_lower_fs_tst_lt( __ev64_opaque__ a, __ev64_opaque__ b);
__ev64_opaque__ __ev_select_fs_tst_lt( __ev64_opaque__ a, __ev64_opaque__ b,
                             __ev64_opaque__ c, __ev64_opaque__ d);

_Bool __ev_any_fs_tst_eq( __ev64_opaque__ a, __ev64_opaque__ b);
_Bool __ev_all_fs_tst_eq( __ev64_opaque__ a, __ev64_opaque__ b);
_Bool __ev_upper_fs_tst_eq( __ev64_opaque__ a, __ev64_opaque__ b);
_Bool __ev_lower_fs_tst_eq( __ev64_opaque__ a, __ev64_opaque__ b);
__ev64_opaque__ __ev_select_fs_tst_eq( __ev64_opaque__ a, __ev64_opaque__ b,
                             __ev64_opaque__ c, __ev64_opaque__ d);
# LOAD/STORE
```

NOTE

The 5-bit unsigned literal in the immediate form is scaled by the size of the load or store to determine how many bytes the pointer 'p' is offset by. The size of the load is determined by the first letter after the 'l': 'd'—double-word (8 bytes), 'w'—word (4 bytes), 'h'—half word (2 bytes). For details, see Chapter 5, “Programming Interface Examples”.

```
__ev64_opaque__ __ev_lddx( __ev64_opaque__ * p, int32_t offset );
__ev64_opaque__ __ev_lddx( __ev64_opaque__ * p, int32_t offset );
__ev64_opaque__ __ev_ldwx( __ev64_opaque__ * p, int32_t offset );
__ev64_opaque__ __ev_ldhx( __ev64_opaque__ * p, int32_t offset );
__ev64_opaque__ __ev_lwhex( uint32_t * p, int32_t offset );
__ev64_opaque__ __ev_lwhoux( uint32_t * p, int32_t offset );
__ev64_opaque__ __ev_lwhosx( uint32_t * p, int32_t offset );
__ev64_opaque__ __ev_lwwsplatx( uint32_t * p, int32_t offset );
__ev64_opaque__ __ev_lwhsplatx( uint32_t * p, int32_t offset );
__ev64_opaque__ __ev_lhhesplatx( uint16_t * p, int32_t offset );
__ev64_opaque__ __ev_lhhousplatx( uint16_t * p, int32_t offset );
__ev64_opaque__ __ev_lhhossplatx( uint16_t * p, int32_t offset );

__ev64_opaque__ __ev_ldd( __ev64_opaque__ * p, 5-bit unsigned literal );
__ev64_opaque__ __ev_ldw( __ev64_opaque__ * p, 5-bit unsigned literal );
```

```
__ev64_opaque__ __ev_ldh( __ev64_opaque__ * p, 5-bit unsigned literal );
__ev64_opaque__ __ev_lwhe( uint32_t * p, 5-bit unsigned literal );
__ev64_opaque__ __ev_lwhou( uint32_t * p, 5-bit unsigned literal );
__ev64_opaque__ __ev_lwhos( uint32_t * p, 5-bit unsigned literal );
__ev64_opaque__ __ev_lwwsplat( uint32_t * p, 5-bit unsigned literal );
__ev64_opaque__ __ev_lwhsplat( uint32_t * p, 5-bit unsigned literal );
__ev64_opaque__ __ev_lhhesplat( uint16_t * p, 5-bit unsigned literal );
__ev64_opaque__ __ev_lhhousplat( uint16_t * p, 5-bit unsigned literal );
__ev64_opaque__ __ev_lhhossplat( uint16_t * p, 5-bit unsigned literal );

void __ev_stddx( __ev64_opaque__ a, __ev64_opaque__ * p, int32_t offset );
void __ev_stdwx( __ev64_opaque__ a, __ev64_opaque__ * p, int32_t offset );
void __ev_stdhx( __ev64_opaque__ a, __ev64_opaque__ * p, int32_t offset );
void __ev_stwwex( __ev64_opaque__ a, uint32_t * p, int32_t offset );
void __ev_stwwox( __ev64_opaque__ a, uint32_t * p, int32_t offset );
void __ev_stwhex( __ev64_opaque__ a, uint32_t * p, int32_t offset );
void __ev_stwhox( __ev64_opaque__ a, uint32_t * p, int32_t offset );

void __ev_stdd( __ev64_opaque__ a, __ev64_opaque__ * p, 5-bit unsigned literal );
void __ev_stdw( __ev64_opaque__ a, __ev64_opaque__ * p, 5-bit unsigned literal );
void __ev_stdh( __ev64_opaque__ a, __ev64_opaque__ * p, 5-bit unsigned literal );
void __ev_stwwe( __ev64_opaque__ a, uint32_t * p, 5-bit unsigned literal );
void __ev_stwwo( __ev64_opaque__ a, uint32_t * p, 5-bit unsigned literal );
void __ev_stwhe( __ev64_opaque__ a, uint32_t * p, 5-bit unsigned literal );
void __ev_stwho( __ev64_opaque__ a, uint32_t * p, 5-bit unsigned literal );


*** FIXED-POINT COMPLEX ***

__ev64_opaque__ __ev_mhossf( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mhosmf( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mhosmi( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mhoumi( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mhessf( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mhesmf( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mhesmi( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mheumi( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mhossfa( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mhosmfa( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mhosmia( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mhoumia( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mhessfa( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mhesmfa( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mhesmia( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mheumia( __ev64_opaque__ a, __ev64_opaque__ b );

 // maps to __ev_mhoumi
 __ev64_opaque__ __ev_mhoumf( __ev64_opaque__ a, __ev64_opaque__ b );

// maps to __ev_mheumi
 __ev64_opaque__ __ev_mheumf( __ev64_opaque__ a, __ev64_opaque__ b );
 // maps to __ev_mhoumia
 __ev64_opaque__ __ev_mhoumfa( __ev64_opaque__ a, __ev64_opaque__ b );
 // maps to __ev_mheumia
 __ev64_opaque__ __ev_mheumfa( __ev64_opaque__ a, __ev64_opaque__ b );
```

```
__ev64_opaque__ __ev_mhossfaaw( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mhossiaaw( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mhosmfaaw( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mhosmiaaw( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mhousiaaw( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mhoumiaaw( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mhessfaaw( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mhessiaaw( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mhesmfaaw( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mhesmiaaw( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mheusiaaw( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mheumiaaw( __ev64_opaque__ a, __ev64_opaque__ b );


// maps to __ev_mhousiaaw
 __ev64_opaque__ __ev_mhousfaaw( __ev64_opaque__ a, __ev64_opaque__ b );


// maps to __ev_mhoumiaaw
 __ev64_opaque__ __ev_mhoumfaaw( __ev64_opaque__ a, __ev64_opaque__ b );


// maps to __ev_mheusiaaw
 __ev64_opaque__ __ev_mheusfaaw( __ev64_opaque__ a, __ev64_opaque__ b );


// maps to __ev_mheumiaaw
 __ev64_opaque__ __ev_mheumfaaw( __ev64_opaque__ a, __ev64_opaque__ b );


__ev64_opaque__ __ev_mhossfanw( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mhossianw( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mhosmfanw( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mhosmianw( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mhousianw( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mhoumianw( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mhessfanw( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mhessianw( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mhesmfanw( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mhesmianw( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mheusianw( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mheumianw( __ev64_opaque__ a, __ev64_opaque__ b );


// maps to __ev_mhousianw
 __ev64_opaque__ __ev_mhousfanw( __ev64_opaque__ a, __ev64_opaque__ b );


// maps to __ev_mhoumianw
 __ev64_opaque__ __ev_mhoumfanw( __ev64_opaque__ a, __ev64_opaque__ b );


// maps to __ev_mheusianw
 __ev64_opaque__ __ev_mheusfanw( __ev64_opaque__ a, __ev64_opaque__ b );


// maps to __ev_mheumianw
 __ev64_opaque__ __ev_mheumfanw( __ev64_opaque__ a, __ev64_opaque__ b );
```

```
__ev64_opaque__ __ev_mhogsmfaa( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mhogsmiaa( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mhogumiaa( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mhegsmfaa( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mhegsmiaa( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mhegumiaa( __ev64_opaque__ a, __ev64_opaque__ b );

// maps to __ev_mhogumiaa
 __ev64_opaque__ __ev_mhogumfaa( __ev64_opaque__ a, __ev64_opaque__ b );

// maps to __ev_mhegumiaa
 __ev64_opaque__ __ev_mhegumfaa( __ev64_opaque__ a, __ev64_opaque__ b );


__ev64_opaque__ __ev_mhogsmfan( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mhogsmian( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mhogumian( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mhegsmfan( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mhegsmian( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mhegumian( __ev64_opaque__ a, __ev64_opaque__ b );

// maps to __ev_mhogumian
 __ev64_opaque__ __ev_mhogumfan( __ev64_opaque__ a, __ev64_opaque__ b );

// maps to __ev_mhegumian
 __ev64_opaque__ __ev_mhegumfan( __ev64_opaque__ a, __ev64_opaque__ b );

__ev64_opaque__ __ev_mwhssf( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mwhsmf( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mwhsmi( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mwhumi( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mwhssfa( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mwhsmfa( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mwhsmia( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mwhumia( __ev64_opaque__ a, __ev64_opaque__ b );

// maps to __ev_mwhumi
 __ev64_opaque__ __ev_mwhumf( __ev64_opaque__ a, __ev64_opaque__ b );

// maps to __ev_mwhumia
 __ev64_opaque__ __ev_mwhumfa( __ev64_opaque__ a, __ev64_opaque__ b );

__ev64_opaque__ __ev_mwlumi( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mwlumia( __ev64_opaque__ a, __ev64_opaque__ b );

__ev64_opaque__ __ev_mwlssiaaw( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mwlsmiaaw( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mwlusiaaw( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mwlumiaaw( __ev64_opaque__ a, __ev64_opaque__ b );

__ev64_opaque__ __ev_mwlssianw( __ev64_opaque__ a, __ev64_opaque__ b );
```

```
__ev64_opaque__ __ev_mwlsmianw( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mwlusianw( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mwlumianw( __ev64_opaque__ a, __ev64_opaque__ b );

__ev64_opaque__ __ev_mwhssfaaw( __ev64_opaque__ a, __ev64_opaque__ b );
 __ev64_opaque__ temp = __ev_mwhssf(a,b);
 __ev_addssiaaw(temp);

__ev64_opaque__ __ev_mwhssiaaw( __ev64_opaque__ a, __ev64_opaque__ b );
 __ev64_opaque__ temp = __ev_mwhsmi(a,b);
 __ev_addssiaaw(temp);

__ev64_opaque__ __ev_mwhsmfaaw( __ev64_opaque__ a, __ev64_opaque__ b );
 __ev64_opaque__ temp = __ev_mwhsmf(a,b);
 __ev_addsmiaaw(temp);

__ev64_opaque__ __ev_mwhsmiaaw( __ev64_opaque__ a, __ev64_opaque__ b );
 __ev64_opaque__ temp = __ev_mwhsmi(a,b);
 __ev_addsmiaaw(temp);

__ev64_opaque__ __ev_mwhusiaaw( __ev64_opaque__ a, __ev64_opaque__ b );
 __ev64_opaque__ temp = __ev_mwhumi(a,b);
 __ev_addusiaaw(temp);

__ev64_opaque__ __ev_mwhumiaaw( __ev64_opaque__ a, __ev64_opaque__ b );
 __ev64_opaque__ temp = __ev_mwhumi(a,b);
 __ev_addumiaaw(temp);


// maps to __ev_mwhusiaaw
 __ev64_opaque__ __ev_mwhusfaaw( __ev64_opaque__ a, __ev64_opaque__ b );


// maps to __ev_mwhumiaaw
 __ev64_opaque__ __ev_mwhumfaaw( __ev64_opaque__ a, __ev64_opaque__ b );


__ev64_opaque__ __ev_mwhssfanw( __ev64_opaque__ a, __ev64_opaque__ b );
 __ev64_opaque__ temp = __ev_mwhssf(a,b);
 __ev_subfssiaaw(temp);

__ev64_opaque__ __ev_mwhssianw( __ev64_opaque__ a, __ev64_opaque__ b );
 __ev64_opaque__ temp = __ev_mwhsmi(a,b);
 __ev_subfssiaaw(temp);

__ev64_opaque__ __ev_mwhsmfanw( __ev64_opaque__ a, __ev64_opaque__ b );
 __ev64_opaque__ temp = __ev_mwhsmf(a,b);
 __ev_subfsmiaaw(temp);

__ev64_opaque__ __ev_mwhsmianw( __ev64_opaque__ a, __ev64_opaque__ b );
 __ev64_opaque__ temp = __ev_mwhsmi(a,b);
 __ev_subfsmiaaw(temp);

__ev64_opaque__ __ev_mwhusianw( __ev64_opaque__ a, __ev64_opaque__ b );
 __ev64_opaque__ temp = __ev_mwhumi(a,b);
 __ev_subfusiaaw(temp);
```

```
__ev64_opaque__ __ev_mwhumianw( __ev64_opaque__ a, __ev64_opaque__ b );
 __ev64_opaque__ temp = __ev_mwhumi(a,b);
 __ev_subfumiaaw(temp);


**
__ev64_opaque__ __ev_mwhgssfaa( __ev64_opaque__ a, __ev64_opaque__ b );
 __ev64_opaque__ temp = __ev_mwhssf(a, b);
 // Note: the upper 32 bits of the immediate is a do not care.  Therefore
 // we spec {1, 1} because it can easily be generated by a __ev_splati(1)
 __ev_mwsmiaa(temp, (__ev64_u32__){1, 1});

__ev64_opaque__ __ev_mwhgsmfaa( __ev64_opaque__ a, __ev64_opaque__ b );
 __ev64_opaque__ temp = __ev_mwhsmf(a, b);
 // Note: the upper 32 bits of the immediate is a do not care.  Therefore
 // we spec {1, 1} because it can easily be generated by a __ev_splati(1)
 __ev_mwsmiaa(temp, (__ev64_u32__){1, 1});

__ev64_opaque__ __ev_mwhgsmiaa( __ev64_opaque__ a, __ev64_opaque__ b );
 __ev64_opaque__ temp = __ev_mwhsmi(a, b);
 // Note: the upper 32 bits of the immediate is a do not care.  Therefore
 // we spec {1, 1} because it can easily be generated by a __ev_splati(1)
 __ev_mwsmiaa(temp, (__ev64_u32__){1, 1});

__ev64_opaque__ __ev_mwhgumiaa( __ev64_opaque__ a, __ev64_opaque__ b );
 __ev64_opaque__ temp = __ev_mwhumi(a, b);
 // Note: the upper 32 bits of the immediate is a do not care.  Therefore
 // we spec {1, 1} because it can easily be generated by a __ev_splati(1)
 __ev_mwumiaa(temp, (__ev64_u32__){1, 1});

 // maps to __ev_mwhgumiaa
 __ev64_opaque__ __ev_mwhgumfaa( __ev64_opaque__ a, __ev64_opaque__ b );

__ev64_opaque__ __ev_mwhgssfan( __ev64_opaque__ a, __ev64_opaque__ b );
 __ev64_opaque__ temp = __ev_mwhssf(a, b);
 // Note: the upper 32 bits of the immediate is a do not care.  Therefore
 // we spec {1, 1} because it can easily be generated by a __ev_splati(1)
 __ev_mwsmian(temp, (__ev64_u32__){1, 1});

__ev64_opaque__ __ev_mwhgsmfan( __ev64_opaque__ a, __ev64_opaque__ b );
 __ev64_opaque__ temp = __ev_mwhsmf(a, b);
 // Note: the upper 32 bits of the immediate is a do not care.  Therefore
 // we spec {1, 1} because it can easily be generated by a __ev_splati(1)
 __ev_mwsmian(temp, (__ev64_u32__){1, 1});

__ev64_opaque__ __ev_mwhgsmian( __ev64_opaque__ a, __ev64_opaque__ b );
 __ev64_opaque__ temp = __ev_mwhsmi(a, b);
 // Note: the upper 32 bits of the immediate is a do not care.  Therefore
 // we spec {1, 1} because it can easily be generated by a __ev_splati(1)
 __ev_mwsmian(temp, (__ev64_u32__){1, 1});

__ev64_opaque__ __ev_mwhgumian( __ev64_opaque__ a, __ev64_opaque__ b );
 __ev64_opaque__ temp = __ev_mwhumi(a, b);
 // Note: the upper 32 bits of the immediate is a do not care.  Therefore
 // we spec {1, 1} because it can easily be generated by a __ev_splati(1)
 __ev_mwumian(temp, (__ev64_u32__){1, 1});
```

```
 // maps to __ev_mwhgumian
 __ev64_opaque__ __ev_mwhgumfan( __ev64_opaque__ a, __ev64_opaque__ b );

NOTE:

An optimizing compiler should be able to improve performance by scheduling
the instructions implementing an intrinsic, that is, __ev_mwhgumfan.

** END OF NOT SUPPORTED **

__ev64_opaque__ __ev_mwssf( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mwsmf( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mwsmi( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mwumi( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mwssfa( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mwsmfa( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mwsmia( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mwumia( __ev64_opaque__ a, __ev64_opaque__ b );
 // maps to __ev_mwumi
 __ev64_opaque__ __ev_mwumf( __ev64_opaque__ a, __ev64_opaque__ b );
 // maps to __ev_mwumia
 __ev64_opaque__ __ev_mwumfa( __ev64_opaque__ a, __ev64_opaque__ b );

__ev64_opaque__ __ev_mwssfaa( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mwsmfaa( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mwsmiaa( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mwumiaa( __ev64_opaque__ a, __ev64_opaque__ b );
 // maps to __ev_mwumiaa
 __ev64_opaque__ __ev_mwumfaa( __ev64_opaque__ a, __ev64_opaque__ b );

__ev64_opaque__ __ev_mwssfan( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mwsmfan( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mwsmian( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mwumian( __ev64_opaque__ a, __ev64_opaque__ b );
 // maps to __ev_mwumian
 __ev64_opaque__ __ev_mwumfan( __ev64_opaque__ a, __ev64_opaque__ b );

__ev64_opaque__ __ev_addssiaaw( __ev64_opaque__ a );
__ev64_opaque__ __ev_addsmiaaw( __ev64_opaque__ a );
__ev64_opaque__ __ev_addusiaaw( __ev64_opaque__ a );
__ev64_opaque__ __ev_addumiaaw( __ev64_opaque__ a );
 // maps to __ev_addusiaaw
 __ev64_opaque__ __ev_addusfaaw( __ev64_opaque__ a );
 // maps to __ev_addumiaaw
 __ev64_opaque__ __ev_addumfaaw( __ev64_opaque__ a );
 // maps to __ev_addsmiaaw
 __ev64_opaque__ __ev_addsmfaaw( __ev64_opaque__ a );
 // maps to __ev_addssiaaw
 __ev64_opaque__ __ev_addssfaaw( __ev64_opaque__ a );

__ev64_opaque__ __ev_subfssiaaw( __ev64_opaque__ a );
__ev64_opaque__ __ev_subfsmiaaw( __ev64_opaque__ a );
__ev64_opaque__ __ev_subfusiaaw( __ev64_opaque__ a );
__ev64_opaque__ __ev_subfumiaaw( __ev64_opaque__ a );
 // maps to __ev_subfusiaaw
 __ev64_opaque__ __ev_subfusfaaw( __ev64_opaque__ a );
 // maps to __ev_subfumiaaw
```

```
__ev64_opaque__ __ev_subfumfaaw( __ev64_opaque__ a );
// maps to __ev_subfsmiaaw
__ev64_opaque__ __ev_subfsmfaaw( __ev64_opaque__ a );
// maps to __ev_subfssiaaw
__ev64_opaque__ __ev_subfssfaaw( __ev64_opaque__ a );


# Floating-Point SIMD Instructions

__ev64_opaque__ __ev_fsabs( __ev64_opaque__ a);
__ev64_opaque__ __ev_fsnabs( __ev64_opaque__ a)
__ev64_opaque__ __ev_fsneg( __ev64_opaque__ a);
__ev64_opaque__ __ev_fsadd( __ev64_opaque__ a, __ev64_opaque__ b);
__ev64_opaque__ __ev_fssub( __ev64_opaque__ a, __ev64_opaque__ b);
__ev64_opaque__ __ev_fsmul( __ev64_opaque__ a, __ev64_opaque__ b);
__ev64_opaque__ __ev_fsdiv( __ev64_opaque__ a, __ev64_opaque__ b);
__ev64_opaque__ __ev_fscfui( __ev64_opaque__ b);
__ev64_opaque__ __ev_fscfsi( __ev64_opaque__ b);
__ev64_opaque__ __ev_fscfuf( __ev64_opaque__ b);
__ev64_opaque__ __ev_fscfsf( __ev64_opaque__ b);
__ev64_opaque__ __ev_fsctui( __ev64_opaque__ b);
__ev64_opaque__ __ev_fsctsi( __ev64_opaque__ b);
__ev64_opaque__ __ev_fsctuf( __ev64_opaque__ b);
__ev64_opaque__ __ev_fsctsf( __ev64_opaque__ b);
__ev64_opaque__ __ev_fsctuiz( __ev64_opaque__ b);
__ev64_opaque__ __ev_fsctsiz( __ev64_opaque__ b);

# creation/insertion/extraction
```

Chapter 4
Additional Operations

4.1 Data Manipulation

The intrinsics in section one act like functions with parameters that are passed by value. Figure 4-1 and Figure 4-2 show the layout of a __ev64_opaque__ variable in the register with reference to creation, insertion, and extraction routines (regardless of endianess).

Figure 4-2 shows byte, half-word, and word ordering.

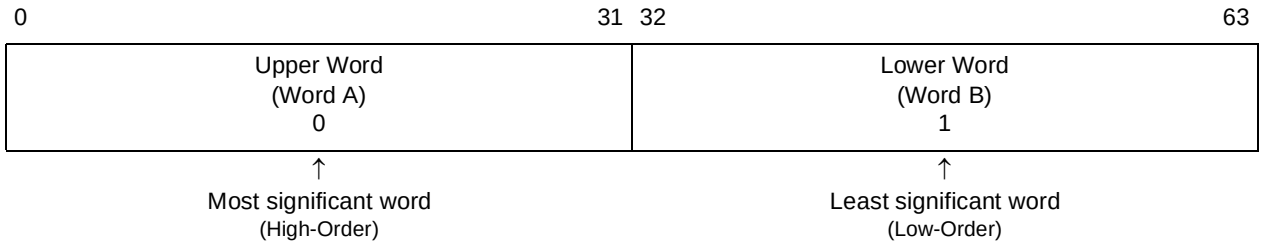

Figure 4-1. Big-Endian Word Ordering

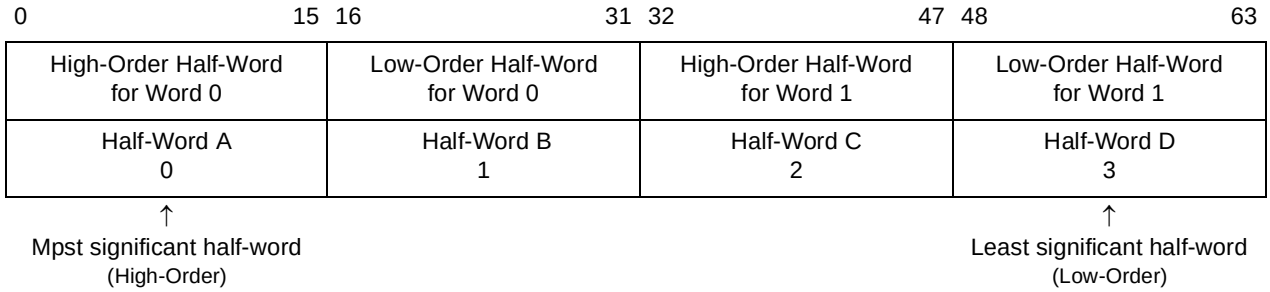

Figure 4-2. Big-Endian Half-Word Ordering

4.1.1 Creation Intrinsics

These intrinsics create new generic 64-bit opaque data types from the given inputs passed by value. More specifically, they are created from the following inputs: 1 signed or unsigned 64-bit integer, 2 single-precision floats, 2 signed or unsigned 32-bit integers, or 4 signed or unsigned 16-bit integers.

```
__ev64_opaque__ __ev_create_u64( uint64_t a );
__ev64_opaque__ __ev_create_s64( int64_t a );
__ev64_opaque__ __ev_create_fs( float a, float b );
```

```
__ev64_opaque__ __ev_create_u32( uint32_t a, uint32_t b );
__ev64_opaque__ __ev_create_s32( int32_t a, int32_t b );
__ev64_opaque__ __ev_create_u16( uint16_t a, uint16_t b, uint16_t c, uint16_t d );
__ev64_opaque__ __ev_create_s16( int16_t a, int16_t b, int16_t c, int16_t d );
__ev64_opaque__ __ev_create_sfix32_fs( float a, float b );
__ev64_opaque__ __ev_create_ufix32_fs( float a, float b );


//maps to__ev_create_u32
__ev64_opaque__ __ev_create_ufix32_u32( uint32_t a, uint32_t b );

// maps to __ev_create_s32
__ev64_opaque__ __ev_create_sfix32_s32( int32_t a, int32_t b );
```

4.1.2 Convert Intrinsics

These intrinsics convert a generic 64-bit opaque data type to a specific signed or unsigned integral form.

```
uint64_t __ev_convert_u64( __ev64_opaque__ a );
int64_t __ev_convert_s64( __ev64_opaque__ a );
```

4.1.3 Get Intrinsics

These intrinsics allow the user to access data from within a specified location of the generic 64-bit opaque data type.

4.1.3.1 Get_Upper/Lower

These intrinsics specify whether the upper 32-bits or lower 32-bits of the 64-bit opaque data type are returned. Only signed/unsigned 32-bit integers or single-precision floats are returned.

```
uint32_t __ev_get_upper_u32( __ev64_opaque__ a );
uint32_t __ev_get_lower_u32( __ev64_opaque__ a );
int32_t __ev_get_upper_s32( __ev64_opaque__ a );
int32_t __ev_get_lower_s32( __ev64_opaque__ a );
float __ev_get_upper_fs( __ev64_opaque__ a );
float __ev_get_lower_fs( __ev64_opaque__ a );

// maps to __ev_get_upper_u32
uint32_t __ev_get_upper_ufix32_u32( __ev64_opaque__ a );


// maps to __ev_get_lower_u32
uint32_t __ev_get_lower_ufix32_u32( __ev64_opaque__ a );
```

```
// maps to __ev_get_upper_s32
int32_t __ev_get_upper_sfix32_s32( __ev64_opaque__ a );

// maps to __ev_get_lower_s32
int32_t __ev_get_lower_sfix32_s32( __ev64_opaque__ a );

// equivalent to __ev_get_sfix32_fs(a, 0);
float __ev_get_upper_sfix32_fs( __ev64_opaque__ a );

// equivalent to __ev_get_sfix32_fs(a, 1);
float __ev_get_lower_sfix32_fs( __ev64_opaque__ a );

// equivalent to __ev_get_ufix32_fs(a, 0);
float __ev_get_upper_ufix32_fs( __ev64_opaque__ a );

// equivalent to __ev_get_ufix32_fs(a, 1);
float __ev_get_lower_ufix32_fs( __ev64_opaque__ a );
```

4.1.3.2 Get Explicit Position

These intrinsics allow the user to specify the position (pos) in the 64-bit opaque data type where the data is accessed and returned. The position is 0 or 1 for words and either 0, 1, 2, or 3 for half-words.

```
uint32_t __ev_get_u32( __ev64_opaque__ a, uint32_t pos );
int32_t  __ev_get_s32( __ev64_opaque__ a, uint32_t pos );
float    __ev_get_fs( __ev64_opaque__ a, uint32_t pos );
uint16_t __ev_get_u16( __ev64_opaque__ a, uint32_t pos );
int16_t  __ev_get_s16( __ev64_opaque__ a, uint32_t pos );

// maps to __ev_get_u32
uint32_t __ev_get_ufix32_u32( __ev64_opaque__ a, uint32_t pos );

// maps to __ev_get_s32
int32_t  __ev_get_sfix32_s32( __ev64_opaque__ a, uint32_t pos );

float    __ev_get_ufix32_fs( __ev64_opaque__ a, uint32_t pos );
float    __ev_get_sfix32_fs( __ev64_opaque__ a, uint32_t pos );
```

4.1.4 Set Intrinsics

These intrinsics provide the capability of setting values in a 64-bit opaque data type that the intrinsic or the user specifies.

4.1.4.1 Set_Upper/Lower

These intrinsics specify which word (either upper or lower 32-bits) of the 64-bit opaque data type is set to input value b.

```
__ev64_opaque__ __ev_set_upper_u32( __ev64_opaque__ a, uint32_t b );
__ev64_opaque__ __ev_set_lower_u32( __ev64_opaque__ a, uint32_t b );
__ev64_opaque__ __ev_set_upper_s32( __ev64_opaque__ a, int32_t b );
__ev64_opaque__ __ev_set_lower_s32( __ev64_opaque__ a, int32_t b );
__ev64_opaque__ __ev_set_upper_fs( __ev64_opaque__ a, float b );
__ev64_opaque__ __ev_set_lower_fs( __ev64_opaque__ a, float b );

// maps to __ev_set_upper_u32
__ev64_opaque__ __ev_set_upper_ufix32_u32( __ev64_opaque__ a, uint32_t b );

// maps to __ev_set_lower_u32
__ev64_opaque__ __ev_set_lower_ufix32_u32( __ev64_opaque__ a, uint32_t b );

// maps to __ev_set_upper_s32
__ev64_opaque__ __ev_set_upper_sfix32_s32( __ev64_opaque__ a, int32_t b );

// maps to __ev_set_lower_s32
__ev64_opaque__ __ev_set_lower_sfix32_s32( __ev64_opaque__ a, int32_t b );

// equivalent to __ev_set_sfix32_fs(a, b, 0);
__ev64_opaque__ __ev_set_upper_sfix32_fs( __ev64_opaque__ a, float b );

// equivalent to __ev_set_sfix32_fs(a, b, 1);
__ev64_opaque__ __ev_set_lower_sfix32_fs( __ev64_opaque__ a, float b );

// equivalent to __ev_set_ufix32_fs(a, b, 0);
__ev64_opaque__ __ev_set_upper_ufix32_fs( __ev64_opaque__ a, float b );

// equivalent to __ev_set_ufix32_fs(a, b, 1);
__ev64_opaque__ __ev_set_lower_ufix32_fs( __ev64_opaque__ a, float b );
```

4.1.4.2 Set Accumulator

These intrinsics initialize the accumulator to the input value a.

```
__ev64_opaque__ __ev_set_acc_u64( uint64_t a );
__ev64_opaque__ __ev_set_acc_s64( int64_t a );
__ev64_opaque__ __ev_set_acc_vec64( __ev64_opaque__ a );
```

4.1.4.3 Set Explicit Position

These intrinsics set the 64-bit opaque input value a to the value in b based on the position given in pos. Unlike the intrinsics in 4.1.4.1, the positional value is specified by the user to be either 0 or 1 for words or 0, 1, 2, or 3 for half-words.

```
__ev64_opaque__ __ev_set_u32( __ev64_opaque__ a, uint32_t b, uint32_t pos );
__ev64_opaque__ __ev_set_s32( __ev64_opaque__ a, int32_t b, uint32_t pos );
__ev64_opaque__ __ev_set_fs( __ev64_opaque__ a, float b, uint32_t pos );
__ev64_opaque__ __ev_set_u16( __ev64_opaque__ a, uint16_t b, uint32_t pos );
__ev64_opaque__ __ev_set_s16( __ev64_opaque__ a, int16_t b, uint32_t pos );

// maps to __ev_set_u32
__ev64_opaque__ __ev_set_ufix32_u32( __ev64_opaque__ a, uint32_t b, uint32_t pos);

// maps to __ev_set_s32
__ev64_opaque__ __ev_set_sfix32_s32( __ev64_opaque__ a, int32_t b, uint32_t pos);

__ev64_opaque__ __ev_set_ufix32_fs( __ev64_opaque__ a, float b, uint32_t pos );
__ev64_opaque__ __ev_set_sfix32_fs( __ev64_opaque__ a, float b, uint32_t pos );
```

4.2 Signal Processing Engine (SPE) APU Registers

The SPE includes the following two registers:

- The signal processing and embedded floating-point status and control register (SPEFSCR), described in Section 4.2.1, “Signal Processing and Embedded Floating-Point Status and Control Register (SPEFSCR).”
- A 64-bit accumulator, described in Section 3.1.2, “Accumulator (ACC).”

4.2.1 Signal Processing and Embedded Floating-Point Status and Control Register (SPEFSCR)

The SPEFSCR, which is shown in Figure 4-3, is used for status and control of SPE instructions.

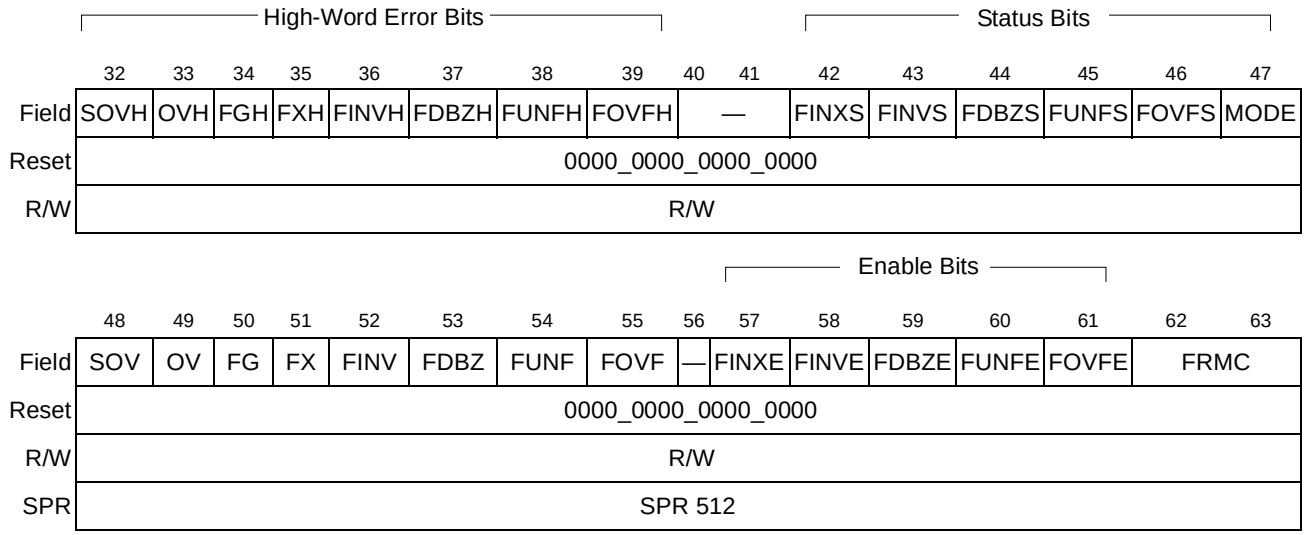

Figure 4-3. Signal Processing and Embedded Floating-Point Status and Control Register (SPEFSCR)

Table 4-1 describes SPEFSCR bits.

Table 4-1. SPEFSCR Field Descriptions

Bits	Name	Function
32	SOVH	Summary integer overflow high. Set whenever an instruction (except **mtspr**) sets OVH. SOVH remains set until it is cleared by an **mtspr[SPEFSCR]**.
33	OVH	Integer overflow high. An overflow occurred in the upper half of the register while executing a SPE integer instruction.
34	FGH	Embedded floating-point guard bit high. Floating-point guard bit from the upper half. The value is undefined if the processor takes a floating-point exception due to input error, floating-point overflow, or floating-point underflow.
35	FXH	Embedded floating-point sticky bit high. Floating bit from the upper half. The value is undefined if the processor takes a floating-point exception due to input error, floating-point overflow, or floating-point underflow.
36	FINVH	Embedded floating-point invalid operation error high. Set when an input value on the high side is a NaN, Inf, or Denorm. Also set on a divide if both the dividend and divisor are zero.
37	FDBZH	Embedded floating-point divide by zero error high. Set if the dividend is non-zero and the divisor is zero.
38	FUNFH	Embedded floating-point underflow error high
39	FOVFH	Embedded floating-point overflow error high
40–41	—	Reserved, and should be cleared

Table 4-1. SPEFSCR Field Descriptions (continued)

Bits	Name	Function
42	FINXS	Embedded floating-point inexact sticky. FINXS = FINXS | FGH | FXH | FG | FX.
43	FINVS	Embedded floating-point invalid operation sticky. Location for software to use when implementing true IEEE floating point.
44	FDBZS	Embedded floating-point divide by zero sticky. FDBZS = FDBZS | FDBZH | FDBZ.
45	FUNFS	Embedded floating-point underflow sticky. Storage location for software to use when implementing true IEEE floating point.
46	FOVFS	Embedded floating-point overflow sticky. Storage location for software to use when implementing true IEEE floating point.
47	MODE	Embedded floating-point mode (read-only on e500)
48	SOV	Integer summary overflow. Set whenever an SPE instruction (except **mtspr**) sets OV. SOV remains set until it is cleared by **mtspr[SPEFSCR]**.
49	OV	Integer overflow. An overflow occurred in the lower half of the register while a SPE integer instruction was executed.
50	FG	Embedded floating-point guard bit. Floating-point guard bit from the lower half. The value is undefined if the processor takes a floating-point exception due to input error, floating-point overflow, or floating-point underflow.
51	FX	Embedded floating-point sticky bit. Floating bit from the lower half. The value is undefined if the processor takes a floating-point exception due to input error, floating-point overflow, or floating-point underflow.
52	FINV	Embedded floating-point invalid operation error. Set when an input value on the high side is a NaN, Inf, or Denorm. Also set on a divide if both the dividend and divisor are zero.
53	FDBZ	Embedded floating-point divide by zero error. Set of the dividend is non-zero and the divisor is zero.
54	FUNF	Embedded floating-point underflow error
55	FOVF	Embedded floating-point overflow error
56	—	Reserved, and should be cleared
57	FINXE	Embedded floating-point inexact enable
58	FINVE	Embedded floating-point invalid operation/input error exception enable 0 Exception disabled 1 Exception enabled If the exception is enabled, a floating-point data exception is taken if FINV or FINVH is set by a floating-point instruction.
59	FDBZE	Embedded floating-point divide-by-zero exception enable 0 Exception disabled 1 Exception enabled If the exception is enabled, a floating-point data exception is taken if FDBZ or FDBZH is set by a floating-point instruction.
60	FUNFE	Embedded floating-point underflow exception enable 0 Exception disabled 1 Exception enabled If the exception is enabled, a floating-point data exception is taken if FUNF or FUNFH is set by a floating-point instruction.

Table 4-1. SPEFSCR Field Descriptions (continued)

Bits	Name	Function
61	FOVFE	Embedded floating-point overflow exception enable 0 Exception disabled 1 Exception enabled If the exception is enabled, a floating-point data exception is taken if FOVF or FOVFH is set by a floating-point instruction.
62–63	FRMC	Embedded floating-point rounding mode control 00 Round to nearest 01 Round toward zero 10 Round toward +infinity 11 Round toward -infinity

4.2.2 SPEFSCR Intrinsics

The following sections discuss SPEFSCR low-level accessors and SPEFSCR Clear and Set functions.

4.2.2.1 SPEFSCR Low-Level Accessors

These intrinsics allow the user to access specific bits in the status and control registers.

```
uint32_t __ev_get_spefscr_sovh( );
uint32_t __ev_get_spefscr_ovh( );
uint32_t __ev_get_spefscr_fgh( );
uint32_t __ev_get_spefscr_fxh( );
uint32_t __ev_get_spefscr_finvh( );
uint32_t __ev_get_spefscr_fdbzh( );
uint32_t __ev_get_spefscr_funfh( );
uint32_t __ev_get_spefscr_fovfh( );

uint32_t __ev_get_spefscr_finxs( );
uint32_t __ev_get_spefscr_finvs( );
uint32_t __ev_get_spefscr_fdbzs( );
uint32_t __ev_get_spefscr_funfs( );
uint32_t __ev_get_spefscr_fovfs( );

uint32_t __ev_get_spefscr_mode( );

uint32_t __ev_get_spefscr_sov( );
uint32_t __ev_get_spefscr_ov( );
uint32_t __ev_get_spefscr_fg( );
uint32_t __ev_get_spefscr_fx( );
uint32_t __ev_get_spefscr_finv( );
uint32_t __ev_get_spefscr_fdbz( );
uint32_t __ev_get_spefscr_funf( );
uint32_t __ev_get_spefscr_fovf( );

uint32_t __ev_get_spefscr_finxe( );
```

```
uint32_t __ev_get_spefscr_finve( );
uint32_t __ev_get_spefscr_fdbze( );
uint32_t __ev_get_spefscr_funfe( );
uint32_t __ev_get_spefscr_fovfe( );

uint32_t __ev_get_spefscr_frmc( );

SPEFSCR Clear and Set Functions
```

These intrinsics allow the user to clear and set specific bits in the status and control register. Note that the user can set only the rounding mode bits.

```
void __ev_clr_spefscr_sovh( );
void __ev_clr_spefscr_sov( );

void __ev_clr_spefscr_finxs( );
void __ev_clr_spefscr_finvs( );
void __ev_clr_spefscr_fdbzs( );
void __ev_clr_spefscr_funfs( );
void __ev_clr_spefscr_fovfs( );

void __ev_set_spefscr_frmc( uint32_t rnd );
                                // rnd = 0 (nearest), rnd = 1 (zero),
                                // rnd = 2 (+inf), rnd = 3 (-inf)
```

4.3 Application Binary Interface (ABI) Extensions

The following sections discuss ABI extensions.

4.3.1 malloc(), realloc(), calloc(), and new

The malloc(), realloc(), and calloc() functions are required to return a pointer with the proper alignment for the object in question. Therefore, to conform to the ABI, these functions must return pointers to memory locations that are at least 8-byte aligned. In the case of the C++ operator new, the implementation of new is required to use the appropriate set of functions based on the alignment requirements of the type.

4.3.2 printf Example

The programming model specifies several new conversion format tokens. The programming model expects a combination of existing format tokens, new format tokens, and __ev_get_* intrinsics. Table 4-2 lists new tokens specified to handle fixed-point data types.

Table 4-2. New Tokens for Fixed-Point Data Types

Token	Data Representation
%hr	Signed 16-bit fixed point
%r	Signed 32-bit fixed point
%lr	Signed 64-bit fixed point
%hR	Unsigned 16-bit fixed point
%R	Unsigned 32-bit fixed point
%lR	Unsigned 64-bit fixed point

Example:

```
__ev64_opaque__ a ;

a = __ev_create_s32 ( 2, -3 );

printf ( " %d %d \n", __ev_get_upper_s32(a), __ev_get_lower_s32(a) );

// output:
// 2 -3
```

The default precision for the new tokens is 6 digits. The tokens should be treated like the %f token with respect to floating-point values. The same field width and precision options should be respected for the new tokens, as the following example shows:

```
printf ("%lr", 0x4000);==> "0.500000"
printf ("%r", 0x40000000); ==> "0.500000"
printf ("%hr", 0x4000000000000000ull);==> "0.500000"
printf ("%09.5r",0x40000000);==> "000.50000"
printf ("%09.5f",0.5);==> "000.50000"
```

4.3.3 Additional Library Routines

The functions atosfix16, atosfix32, atosfix64, atoufix16, atoufix32, and atoufix64 need not affect the value of the integer expression errno on an error. If the value of the result cannot be represented, the behavior is undefined.

```
#include <spe.h>
int16_t atosfix16(const char *str);
int32_t atosfix32(const char *str);
int64_t atosfix64(const char *str);

uint16_t atoufix16(const char *str);
uint32_t atoufix32(const char *str);
uint64_t atoufix64(const char *str);
```

The atosfix16, atosfix32, atosfix64, atoufix16, atoufix32, atoufix64 functions convert the initial portion of the string to which str points to the following numbers:

- 16-bit signed fixed-point number
- 32-bit signed fixed-point number
- 64-bit signed fixed-point number
- 16-bit unsigned fixed-point number
- 32-bit unsigned fixed-point number
- 64-bit unsigned fixed-point number

These numbers are represented as int16_t, int32_t, int64_t, uint16_t, uint32_t, and uint64_t, respectively.

Except for the behavior on error, they are equivalent to the following:

```
atosfix16: strtosfix16(str, (char **)NULL)
atosfix32: strtosfix32(str, (char **)NULL)
atosfix64: strtosfix64(str, (char **)NULL)
atoufix16: strtoufix16(str, (char **)NULL)
atoufix32: strtoufix32(str, (char **)NULL)
atoufix64: strtoufix64(str, (char **)NULL)
```

```
#include <spe.h>
int16_t strtosfix16(const char *str, char **endptr);
int32_t strtosfix32(const char *str, char **endptr);
int64_t strtosfix64(const char *str, char **endptr);

uint16_t strtoufix16(const char *str, char **endptr);
uint32_t strtoufix32(const char *str, char **endptr);
uint64_t strtoufix64(const char *str, char **endptr);
```

The strtosfix16, strtosfix32, strtosfix64, strtoufix16, strtoufix32, strtoufix64 functions convert the initial portion of the string to which str points to the following numbers:

- 16-bit signed fixed-point number
- 32-bit signed fixed-point number
- 64-bit signed fixed-point number
- 16-bit unsigned fixed-point number
- 32-bit unsigned fixed-point number
- 64-bit unsigned fixed-point number

These numbers are represented as int16_t, int32_t, int64_t, uint16_t, uint32_t, and uint64_t, respectively.

The functions support the same string representations for fixed-point numbers that the strtod, strtof, strtold functions support, with the exclusion of NAN and INFINITY support.

For the signed functions, if the input value is greater than or equal to 1.0, positive saturation should occur and errno should be set to ERANGE. If the input value is less than -1.0, negative saturation should occur, and errno should be set to ERANGE.

For the unsigned functions, if the input value is greater than or equal to 1.0, saturation should occur to the upper bound, and errno should be set to ERANGE. If the input value is less than 0.0, saturation should occur to the lower bound and errno should be set to ERANGE.

Chapter 5
Programming Interface Examples

5.1 Data Type Initialization

The following examples show valid and invalid initializations of the SPE data types.

5.1.1 __ev64_opaque__ Initialization

The following examples show valid and invalid initializations of __ev64_opaque__:

- Example 1 (Invalid)

```
__ev64_opaque__ x1 = { 0, 1 };
```

  This example is invalid because it lacks qualification for interpreting the array initialization. The compiler is unable to interpret whether the array consists of two unsigned integers, two signed integers, four unsigned integers, four signed integers, or two floats.

- Example 2 (Invalid)

```
__ev64_opaque__ x2 = (__ev64_opaque__) { 0, 1 };
```

  This example is invalid because the qualification provides no additional information for interpreting the array initialization.

- Example 3 (Valid)

```
__ev64_opaque__ x3 = (__ev64_u32__) { 0, 1 };
```

  This example is valid because the array initialization is qualified so that it provides the compiler with a unique interpretation. The array initialization is interpreted as an __ev64_u32__ with an implicit cast from the __ev64_u32__ to __ev64_opaque__.

- Example 4 (Valid)

```
__ev64_opaque__ x4 = (__ev64_opaque__)(__ev64_u32__) { 0, 1 };
```

  Although this example is the same as Example 3, it includes an explicit cast, rather than depending on the implicit casting to __ev64_opaque__ on assignment.

- Example 5 (Valid)

```
__ev64_opaque__ x5 = (__ev64_u16__) (__ev64_opaque__) (__ev64_u32__) { 0, 1 };
```

  This example shows a series of casts; at the end, the result in x5 is no different from what it would be in Example 3. The example depends on the implicit cast from __ev64_u16__ to __ev64_opaque__.

- Example 6 (Valid)

```
__ev64_opaque__ x6 = (__ev64_opaque__) (__ev64_u16__) (__ev64_u32__) { 0, 1 };
```

  This example shows a series of casts; at the end, the result in x6 is no different from what it would be in Example 3. The example explicitly casts to __ev64_opaque__ rather than depending on the implicit cast.

- Example 7 (Valid)

```
__ev64_opaque__ x7 = (__ev64_u16__) (__ev64_u32__) { 0, 1 };
```

  This example shows a series of casts; at the end, the result in x6 is no different from what it would be in Example 3. The example depends on the implicit cast from __ev64_u16__ to __ev64_opaque__.

- Example 8 (Valid)

```
__ev64_opaque__ x8 = (__ev64_u16__) { 0, 1, 2, 3 };
```

  This example is similar to Example 3. It shows that any SPE data types except __ev64_opaque__ can be used to qualify the array initialization.

5.1.2 Array Initialization of SPE Data Types

The following examples show array initialization of SPE data types:

- Example 1 shows how to initialize an array of four __ev64_u32__.

```
__ev64_u32__ x1[4] = {
    { 0, 1 },
    { 2, 3 },
    { 4, 5 },
    { 6, 7 }
};
```

- Example 2 shows how to initialize an array of four __ev64_u16__.

```
__ev64_u16 x2[4] = {
    { 0, 1, 2, 3 },
    { 4, 5, 6, 7 },
    { 8, 9, 10, 11 },
    { 12, 13, 14, 15 },
};
```

- Example 3 shows how to initialize an array of four __ev64_fs__.

```
__ev64_fs__ x3[4] = {
    { 1.1f, 2.2f },
    { -3.3f, 4.4f },
    { 5.5f, 6.6f },
    { 7.7f, -8.8f }
};
```

- Example 4 shows explicit casting, and is the same as Example 1:

```
__ev64_u32__ x4[4] = {
 (__ev64_u32__){0, 1},
 (__ev64_u32__){2, 3},
 (__ev64_u32__){4, 5},
 (__ev64_u32__){6, 7}
};
```

- Example 5 shows mixed explicit casting. x5[1] is equal to (__ev64_u32__){131075, 262149}.

```
__ev64_u32__ x5[4] = {
 (__ev64_u32__){0, 1},
 (__ev64_u16__){2, 3, 4, 5},
 (__ev64_u32__){6, 7},
 (__ev64_u32__){8, 9}
};
```

5.2 Fixed-Point Accessors

The following sections discuss fixed-point accessors.

5.2.1 __ev_create_sfix32_fs

The following examples show use of __ev_create_sfix32_fs:

- Example 1

```
__ev64_s32__ x1 = __ev_create_sfix32_fs (0.5, -0.125);
// x1 = {0x40000000, 0xF0000000}
```

  The floating-point numbers 0.5 and -0.125 are converted to their fixed-point representations and stored in x1.

- Example 2

```
__ev64_s32__ x2 = __ev_create_sfix32_fs (-1.1, 1.0);
// x2 = {0x80000000, 0x7fffffff}
```

  The floating-point numbers are –1.1 and 1.0. Both values are outside of the range that signed fixed-point [–1, 1) supports. Therefore, the results of the conversion are saturated to the most negative number, 0x80000000, and the most positive number, 0x7FFFFFFF.

5.2.2 __ev_create_ufix32_fs

The following examples show use of __ev_create_ufix32_fs:

- Example 1

```
__ev64_u32__ x1 = __ev_create_ufix32_fs(0.5, 0.125);
// x1 = {0x80000000, 0x20000000}
```

  The floating-point numbers 0.5 and 0.125 are converted to their unsigned fixed-point representations and stored in x1.

- Example 2

```
__ev64_u32__ x2 = __ev_create_ufix32_fs(-1.1, 1.0);
// x2 = {0x00000000, 0xFFFFFFFF}
```

  Both floating-point values, –1.1 and 1.0, are outside of the range that unsigned fixed-point [0, 1) supports. Therefore, the results of the conversion are saturated to the lower bound, 0x00000000, and the upper bound, 0xFFFFFFFF.

5.2.3 __ev_set_ufix32_fs

The following examples show use of __ev_set_ufix32_fs:

- Example 1

```
__ev64_u32__ x1a = { 0x00000000 0xffffffff };
__ev64_u32__ x1b = __ev_set_ufix32_fs (x1a, 0.5, 0);
// x1b = {0x80000000, 0xffffffff}
```

  This example shows modification of an element in an SPE variable. The intrinsics work like the create routine in that the floating-point number 0.5 is converted to its unsigned fixed-point representation and placed into element 0.

- Example 2

```
__ev64_u32__ x2a = { 0x00000000 0xffffffff };
__ev64_u32__ x2b = __ev_set_ufix32_fs (x2a, 1.5, 0);
// x2b = {0xffffffff, 0xffffffff}
```

  This example shows modification of an element in an SPE variable. The intrinsics work like the create routine in that the floating-point number 1.5 is saturated to the upper bound for unsigned fixed-point representation and placed into element 0.

5.2.4 __ev_set_sfix32_fs

The following examples show use of __ev_set_sfix32_fs:

- Example 1

```
__ev64_u32__ x1a = { 0x00000000 0xffffffff };
__ev64_u32__ x1b = __ev_set_sfix32_fs (x1a, 0.5, 0);
// x1b = {0x40000000, 0xffffffff}
```

  This example shows modification of an element in an SPE variable. The intrinsics work like the create routine in that the floating-point number 0.5 is converted to its signed fixed-point representation and placed into element 0.

- Example 2

```
__ev64_s32__ x2a = { 0x00000000 0xffffffff };
__ev64_s32__ x2b = __ev_set_sfix32_fs (x2a, 1.5, 0);
// x2b = {0x7fffffff, 0xffffffff}
```

  This example shows modification of an element in an SPE variable. The intrinsics work like the create routine in that the floating-point number 1.5 is saturated to the upper bound for signed fixed-point representation and placed into element 0.

5.2.5 __ev_get_ufix32_fs

This example shows extraction of a floating-point number from an SPE variable interpreted as an unsigned fixed-point number. The intrinsic extracts element 1 of the variable and converts it from an unsigned fixed-point number to the closest floating-point representation.

```
__ev64_u32__ x1 = { 0x80000000, 0xffffffff };
float f1 = __ev_get_ufix32_fs (x1, 1);
// f1 = 1.0
```

5.2.6 __ev_get_sfix32_fs

This example shows extraction of a floating-point number from an SPE variable interpreted as a signed fixed-point number. The intrinsic extracts element 0 of the variable and converts it from a signed fixed-point number to the closest floating-point value.

```
__ev64_s32__ x1 = { 0xf0000000, 0xffffffff };
float f1 = __ev_get_ufix32_fs (x1, 0);
// f1 = -0.125
```

5.3 Loads

These examples apply to load and store intrinsics. All of the examples reference the same 'ev_table':

```
__ev64_u32__ ev_table[] = {
  (__ev64_u32__){0x01020304, 0x05060708},
  (__ev64_u32__){0x090a0b0c, 0x0d0e0f10},
  (__ev64_u32__){0x11121314, 0x15161718},
  (__ev64_u32__){0x191a1b1c, 0x1d1e1f20},

  (__ev64_u32__){0x797a7b7c, 0x7d7e7f80},
  (__ev64_u32__){0x81828384, 0x85868788},
  (__ev64_u32__){0x898a8b8c, 0x8d8e8f90},
  (__ev64_u32__){0x91929394, 0x95969798}
};
```

5.3.1 __ev_lddx

This example shows indexing of double-word load. The base pointer is set to the address of ev_table. The intrinsic offsets the base pointer by 2 double-words (16 bytes). This load is equivalent to ev_table[2].

```
__ev64_u32__ x1 = __ev_lddx((__ev64_opaque__ *)(&ev_table[0]), 16);

// x1 = {0x11121314, 0x15161718};
```

5.3.2 __ev_ldd

This example shows an immediate double-word load. The base pointer is set to the address of ev_table. The intrinsic offsets the base pointer by 2 double-words. This load is equivalent to ev_table[2]. The offset in the immediate pointer is scaled by the double-word load size.

```
__ev64_u32__ x1 = __ev_ldd((__ev64_opaque__ *)(&ev_table[0]), 2);

// x1 = {0x11121314, 0x15161718};
```

5.3.3 __ev_lhhesplatx

This example shows an index half-word even splat load. The base pointer is set to the address of ev_table. The intrinsic offsets the base pointer by 4 bytes.

```
__ev64_u32__ x1 = __ev_lhhesplatx((__ev64_opaque__ *)(&ev_table[0]), 4);

// x1 = {0x05060000, 0x05060000}
```

5.3.4 __ev_lhhesplat

This example shows an immediate half-word even splat load. The base pointer is set to the address of ev_table. The intrinsic offsets the base pointer by 4 half-words (8 bytes). Note that the load size, a half-word in this case, scales the offset in the immediate pointer.

```
__ev64_u32__ x1 = __ev_lhhesplat((__ev64_opaque__ *)(&ev_table[0]), 4);
// x1 = {0x090a0000, 0x090a0000}
```

Appendix A
Revision History

This appendix provides a list of the major differences between revisions of the *Signal Processing Engine Auxiliary Processing Unit Programming Interface Manual*. This is the initial version of the manual so there currrently are no differences.

Glossary of Terms and Abbreviations

The glossary contains an alphabetical list of terms, phrases, and abbreviations used in this book. Some of the terms and definitions included in the glossary are reprinted from *IEEE Std. 754-1985, IEEE Standard for Binary Floating-Point Arithmetic*, copyright ©1985 by the Institute of Electrical and Electronics Engineers, Inc. with the permission of the IEEE.

Note that some terms are defined in the context of their usage in this book.

A

Application binary interface (ABI). A standardized interface that defines calling conventions and stack usage between applications and the operating system.

Architecture. A detailed specification of requirements for a processor or computer system. It does not specify details for implementing the processor or computer system; instead it provides a template for a family of compatible *implementations*.

B

Biased exponent. An *exponent* whose range of values is shifted by a constant (bias). Typically a bias is provided to allow a range of positive values to express a range that includes both positive and negative values.

Big-endian. A byte-ordering method in memory where the address *n* of a word corresponds to the *most-significant byte*. In an addressed memory word, the bytes are ordered (left to right) 0, 1, 2, 3, with 0 as the most-significant byte. *See* Little-endian.

C

Cast. A cast expression consists of a left parenthesis, a type name, a right parenthesis, and an operand expression. The cast causes the operand value to be converted to the type name within the parentheses.

D

Denormalized number. A nonzero floating-point number whose *exponent* has a reserved value, usually the format's minimum, and whose explicit or implicit leading significand bit is zero.

E

Effective address (EA). The 32- or 64-bit address specified for a load, store, or an instruction fetch. This address is then submitted to the MMU for translation to either a *physical memory* address or an I/O address.

Exponent. In the binary representation of a floating-point number, the exponent is the component that normally signifies the integer power to which the value two is raised in determining the value of the represented number. *See also* Biased exponent.

F

Fixed-point. (see *Fractional*)

Fractional. SPE supports 16- and 32-bit signed fractional two's complement data formats. For these two N-bit fractional data types, data is represented using the 1. [N-1] bit format. The MSB is the sign bit (-2^0) and the remaining N-1 bits are fractional bits (2^-1 2^-2 ... 2^-(N-1)).

G

General-purpose register (GPR). Any of the 32 registers in the general-purpose register file. These registers provide the source operands and destination results for all integer data manipulation instructions. Integer load instructions move data from memory to GPRs and store instructions move data from GPRs to memory.

I

IEEE 754. A standard written by the Institute of Electrical and Electronics Engineers that defines operations and representations of binary floating-point arithmetic.

Inexact. Loss of accuracy in an arithmetic operation when the rounded result differs from the infinitely precise value with unbounded range.

L

Least-significant bit (lsb). The bit of least value in an address, register, data element, or instruction encoding.

Little-endian. A byte-ordering method in memory where the address *n* of a word corresponds to the *least-significant byte*. In an addressed memory word, the bytes are ordered (left to right) 3, 2, 1, 0, with 3 as the *most-significant byte*. *See* Big-endian.

M

Mnemonic. The abbreviated name of an instruction used for coding.

Modulo. A value *v* that lies outside the range of numbers that an n-bit wide destination type can represent is replaced by the low-order n bits of the two's complement representation of *v.*

Most-significant bit (msb). The highest-order bit in an address, registers, data element, or instruction encoding.

N

NaN. An abbreviation for ‘Not a Number’; a symbolic entity encoded in floating-point format. The two types of NaNs are *signaling* NaNs (SNaNs) and *quiet* NaNs (QNaNs).

Normalization. A process by which a floating-point value is manipulated such that it can be represented in the format for the appropriate precision (single- or double-precision). For a floating-point value to be representable in the single- or double-precision format, the leading implied bit must be a 1.

O

Overflow. An error condition that occurs during arithmetic operations when the result cannot be stored accurately in the destination register(s). For example, if two 32-bit numbers are multiplied, the result may not be representable in 32 bits.

R

Reserved field. In a register, a reserved field is one that is not assigned a function. A reserved field may be a single bit. The handling of reserved bits is *implementation-dependent*. Software can write any value to such a bit. A subsequent reading of the bit returns 0 if the value last written to the bit was 0 and returns an undefined value (0 or 1) otherwise.

S

Saturate. A value *v* that lies outside the range of numbers representable by a destination type is replaced by the representable number closest to *v.*

Significand. The component of a binary floating-point number that consists of an explicit or implicit leading bit to the left of its implied binary point and a fraction field to the right.

SIMD (Single-instruction, multiple-data). An instruction set architecture that performs operations on multiple, parallel values within a single operand.

Splat. To replicate a value in multiple elements of an SIMD target operand.

Sticky bit. A bit that when *set* must be cleared explicitly.

Supervisor mode. The privileged operation state of a processor. In supervisor mode, software, typically the operating system, can access all control registers and the supervisor memory space, among other privileged operations.

U

Underflow. An error condition that occurs during arithmetic operations when the result cannot be represented accurately in the destination register. For

example, underflow can happen if two floating-point fractions are multiplied and the result requires a smaller *exponent* and/or mantissa than the single-precision format can provide. In other words, the result is too small for accurate representation.

User mode. The unprivileged operating state of a processor used typically by application software. In user mode, software can only access certain control registers and can access only user memory space. No privileged operations can be performed. Also known as *problem state*.

V

Vector literal. A constant expression with a value that is taken as a vector type.

W

Word. A 32-bit data element.

Index

F

G

H

I

L

M

N

O

P

R

S

T

U

V

W