Enhanced Signal Processing Extension and Embedded Floating-Point Version 2 Auxiliary Processing Units Programming Interface Manual

SPE2PIM
Rev. 1.0-2
08/2013

freescale™

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

Information in this document is provided solely to enable system and software implementers to use Freescale Semiconductor products. There are no express or implied copyright licenses granted hereunder to design or fabricate any integrated circuits or integrated circuits based on the information in this document.

Freescale Semiconductor reserves the right to make changes without further notice to any products herein. Freescale Semiconductor makes no warranty, representation or guarantee regarding the suitability of its products for any particular purpose, nor does Freescale Semiconductor assume any liability arising out of the application or use of any product or circuit, and specifically disclaims any and all liability, including without limitation consequential or incidental damages. “Typical” parameters which may be provided in Freescale Semiconductor data sheets and/or specifications can and do vary in different applications and actual performance may vary over time. All operating parameters, including “Typicals” must be validated for each customer application by customer's technical experts. Freescale Semiconductor does not convey any license under its patent rights nor the rights of others. Freescale Semiconductor products are not designed, intended, or authorized for use as components in systems intended for surgical implant into the body, or other applications intended to support or sustain life, or for any other application in which the failure of the Freescale Semiconductor product could create a situation where personal injury or death may occur. Should Buyer purchase or use Freescale Semiconductor products for any such unintended or unauthorized application, Buyer shall indemnify and hold Freescale Semiconductor and its officers, employees, subsidiaries, affiliates, and distributors harmless against all claims, costs, damages, and expenses, and reasonable attorney fees arising out of, directly or indirectly, any claim of personal injury or death associated with such unintended or unauthorized use, even if such claim alleges that Freescale Semiconductor was negligent regarding the design or manufacture of the part.

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

Freescale™ and the Freescale logo are trademarks of Freescale Semiconductor, Inc. The described product contains a PowerPC processor core. The PowerPC name is a trademark of IBM Corp. and used under license. All other product or service names are the property of their respective owners.

© Freescale Semiconductor, Inc., 2004, 2013. All rights reserved.

SPE2PIM
Rev. 1.0-2
08/2013

Contents

Paragraph Number	Title	Page Number

About This Book

Errata and Updates 1-i
Scope 1-i
Audience 1-i
Organization 1-ii
Suggested Reading 1-ii
General Information 1-ii
References 1-ii
Related Documentation 1-iii

Chapter 1 Overview

1.1 High-Level Language Interface 1-1
1.2 Application Binary Interface (ABI) 1-1

Chapter 2 High-Level Language Interface

2.1 Introduction 2-1
2.2 High-Level Language Interface 2-1
2.2.1 Data Types 2-1
2.2.2 Alignment 2-2
2.2.2.1 Alignment of __ev64_*__ Types 2-2
2.2.2.2 Alignment of Aggregates and Unions Containing __ev64_*__ Types 2-2
2.2.3 Extensions of C/C++ Operators for the New Types 2-2
2.2.3.1 sizeof() 2-3
2.2.3.2 Assignment 2-3
2.2.3.3 Address Operator 2-3
2.2.3.4 Pointer Arithmetic 2-3
2.2.3.5 Pointer Dereferencing 2-3
2.2.3.6 Type Casting 2-3
2.2.4 New Operators 2-4
2.2.4.1 __ev64_*__ Initialization and Literals 2-4
2.2.4.2 New Operators Representing SPE2 Operations 2-4
2.2.5 Programming Interface 2-5

Contents

Paragraph Number	Title	Page Number

Chapter 3
SPE2 Operations

3.1 Enhanced Signal Processing (SPE2) APU Extension Registers 3-1
3.1.1 Signal Processing and Embedded Floating-Point Status and Control Register (SPEFSCR) 3-1
3.1.2 Accumulator (ACC) 3-3
3.2 Load and Store Instructions 3-4
3.2.1 Addressing Modes - Non-Update forms 3-4
3.2.1.1 Base + Scaled Immediate Addressing - non-update form 3-4
3.2.1.2 Base + Index Addressing - non-update form 3-5
3.2.2 Addressing Modes - Update forms 3-5
3.2.3 Addressing Modes - Modify forms 3-5
3.2.3.1 Linear addressing update mode 3-6
3.2.3.2 Circular addressing modify mode 3-6
3.2.3.3 Bit-reversed addressing modify mode 3-7
3.3 Notation 3-7
3.4 Instruction Fields 3-8
3.5 Description of Instruction Operation 3-10
3.6 Overview of Intrinsic Definition 3-13
3.6.1 Saturation, Shift, and Bit Reverse Models 3-14
3.6.1.1 Saturation 3-14
3.6.1.2 Shift 3-14
3.6.1.3 Bit Reverse 3-15
3.6.1.4 Load and Store Indexed with-Update Calculations 3-15
3.7 Intrinsic Definitions 3-15
3.7.1 SPE2 Intrinsic Definitions 3-16
3.7.2 EFP2 Intrinsic Definitions 3-958
3.8 Basic Instruction Mapping 3-1021

Chapter 4
Additional Operations

4.1 Data Manipulation 4-1
4.1.1 Creation Intrinsics 4-1
4.1.2 Convert Intrinsics 4-2
4.1.3 Get Intrinsics 4-2
4.1.3.1 Get_Upper/Lower 4-2
4.1.3.2 Get Explicit Position 4-3
4.1.4 Set Intrinsics 4-4
4.1.4.1 Set_Upper/Lower 4-4

Contents

Paragraph Number	Title	Page Number
4.1.4.2	Set Accumulator	4-4
4.1.4.3	Set Explicit Position	4-6
4.2	Signal Processing Engine (SPE) APU Registers	4-6
4.2.1	Signal Processing and Embedded Floating-Point Status and Control Register (SPEFSCR)	4-7
4.2.2	SPEFSCR Intrinsics	4-9
4.2.2.1	SPEFSCR Low-Level Accessors	4-9
4.3	Application Binary Interface (ABI) Extensions	4-10
4.3.1	malloc(), realloc(), calloc(), and new	4-10
4.3.2	printf Example	4-10
4.3.3	Additional Library Routines	4-11

Chapter 5
Programming Interface Examples

5.1	Data Type Initialization	5-1
5.1.1	__ev64_opaque__ Initialization	5-1
5.1.2	Array Initialization of SPE2 Data Types	5-2
5.2	Fixed-Point Accessors	5-3
5.2.1	__ev_create_sfix32_fs	5-3
5.2.2	__ev_create_ufix32_fs	5-4
5.2.3	__ev_set_ufix32_fs	5-4
5.2.4	__ev_set_sfix32_fs	5-5
5.2.5	__ev_get_ufix32_fs	5-5
5.2.6	__ev_get_sfix32_fs	5-5
5.3	Loads	5-6
5.3.1	__ev_lddx	5-6
5.3.2	__ev_ldd	5-6
5.3.3	__ev_lhhesplatx	5-6
5.3.4	__ev_lhhesplat	5-7

Appendix A
Revision History

Glossary of Terms and Abbreviations

Contents

Paragraph Number	Title	Page Number

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 enhanced signal processing (SPE2) and embedded floating-point version-2 (EFP2) auxiliary processing unit (APU) extension.

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 SPE2 and EFP2 implementations. Each PowerPC™ processor is unique in its implementation of the SPE2 and EFP2.

Audience

This manual supports system software and application programmers who want to use the SPE2 and EFP2 APUs to develop products. Users should understand the following concepts:

- Operating systems
- Microprocessor system design
- Basic principles of RISC processing
- SPE2 instruction set
- EFP2 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 SPE2 and EFP2 APUs.
- Chapter 2, "High-Level Language Interface," is useful for software engineers who need to understand how to access SPE2 and EFP2 functionality from high level languages such as C and C++.
- Chapter 3, "SPE2 Operations," describes all instructions in the SPE2 and EFP2 APUs.
- 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 SPE2 data types.
- Appendix A, "Revision History," lists the major differences between revisions of the *Enhanced 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 enhanced signal processing (SPE2) and embedded floating-point version 2 (EFP2) auxiliary processing units (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 SPE2 and EFP2 APUs from programming languages such as C and C++, and describes fundamental data types for the SPE2 and EFP2 programming model. See Chapter 2, “High-Level Language Interface,” for details about this interface.

1.2 Application Binary Interface (ABI)

The SPE2 and EFP2 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.

Chapter 2 High-Level Language Interface

2.1 Introduction

This document defines a programming model to use with the enhanced signal processing (SPE2) and the embedded floating-point version 2 (EFP2) auxiliary processing unit (APU) instruction sets. The purpose of the programming model is to give users the ability to write code that utilizes the APUs 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 SPE2 and EFP2 is intended to accomplish the following goals:

- Provide an efficient and expressive mechanism to access SPE2 and EFP2 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 SPE2 and EFP2

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_u8__	8 unsigned 8-bit integers	0...255
__ev64_s8__	8 signed 8-bit integers	-128...127
__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

Table 2-1. Data Types (continued)

New C/C++ Type	Interpretation of Contents	Values
__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 naturally aligned on an 8-byte boundary. A pointer to any naturally aligned __ev64_*__ data type points to an 8-byte boundary. However, some implementations of SPE2 allow relaxed alignment of vector data. The compiler is responsible for providing options to align vector data on the natural boundary and to whatever boundary an implementation may support. A program is incorrect if it attempts to dereference a pointer to an __ev64_*__ type if the pointer does not contain an address aligned to a boundary supported by the targeted implementation.

In the SPE2 architecture, a load/store to an unsupported address boundary 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 to supported boundaries and their internal organization must be padded, if necessary, so that each internal __ev64_*__ variable is aligned to a supported boundary. A supported boundary of __ev64_*__ data types always includes the natural boundary of 8-bytes and also includes whatever boundary a targeted implemtation supports.

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 SPE2 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_u8__ a = { 0, 1, 2, 3, 4, 5, 6, 7 };
__ev64_s8__ b = { -1, -2, -3, -4, -5, -6, 0, 7 };
__ev64_u16__ c = { 0, 1, 2, 3 };
__ev64_s16__ d = { -1, -2, -3, 4 };
__ev64_u32__ e = { 3, 4 };
__ev64_s32__ f = { -2, 4 };
__ev64_u64__ g = { 17  } ;
__ev64_s64__ h = { 23 } ;
__ev64_fs__ i = { 2.4, -3.2 };

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

2.2.4.2 New Operators Representing SPE2 Operations

New operators are introduced to allow full access to the functionality that the SPE2 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 SPE2 operation, like __ev_addw(__ev64_opaque__ a, __ev64_opaque__ b)

- A predicate computed from a SPE2 operation, like __ev_all_eq(__ev64_opaque__ a, __ev64_opaque__ b)
- Creation, insertion, extraction of __ev64_opaque__ values

Each operator representing an SPE2 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 SPE2 operation. Predicate intrinsics handle comparison operations in the SPE2 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
SPE2 Operations

This chapter describes the instructions of the enhanced signal processing (SPE2) and embedded floating-point version 2 (EFP2) auxiliary processing units (APU).

3.1 Enhanced Signal Processing (SPE2) APU Extension Registers

The SPE2 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 SPE2 instructions.

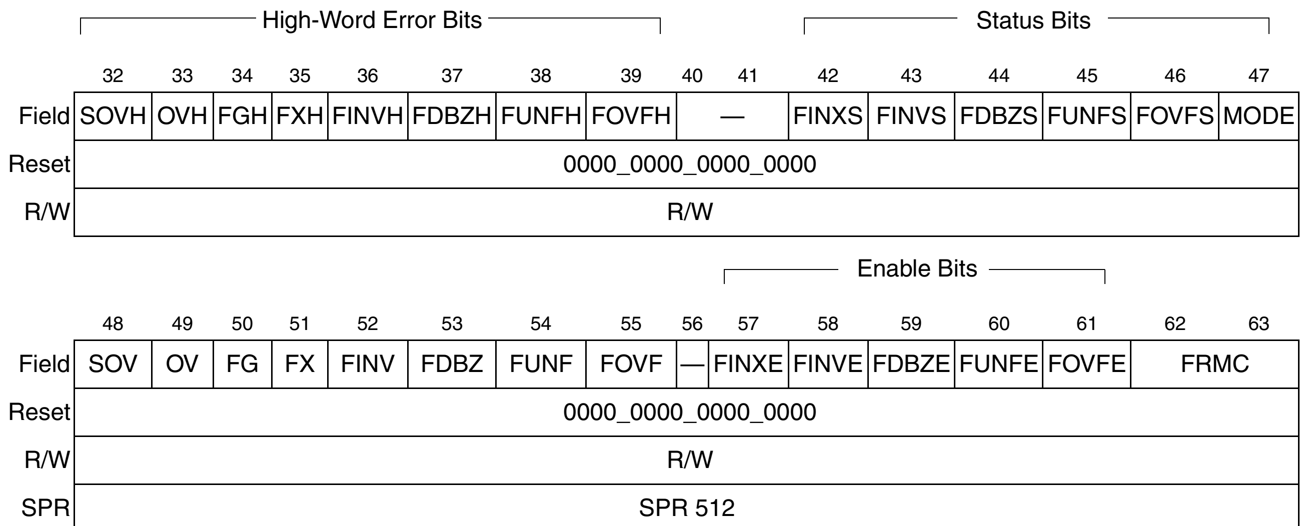


Figure 3-1. Signal Processing and Embedded Floating-Point Status and Control Register (SPEFSCR)

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
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

Table 3-1. SPEFSCR Field Descriptions (continued)

Bits	Name	Function
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.
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 SPE2 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.

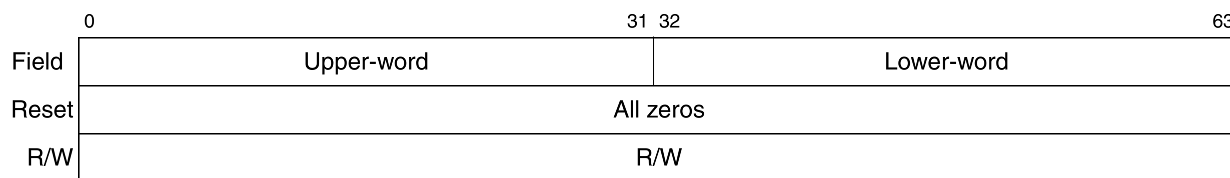


Figure 3-2. Accumulator (ACC)

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 SPE2 multiply with accumulate instructions
32–63	Lower word	Holds the lower-word accumulate value for SPE2 multiply with accumulate instructions

3.2 Load and Store Instructions

SPE provides a number of load and store instructions. These instructions provide load and store capabilities for moving data elements between the GPRs and memory. Data elements of 8, 16, 32, and 64 bits are supported. A variety of forms are provided that position data for efficient computation.

3.2.1 Addressing Modes - Non-Update forms

Base + Index and Base + Scaled Immediate addressing modes are provided. Base parameters hold 64-bit pointer values (32-bit pointers in a 32-bit implementation of the architecture), while parameters used as index values provide 64-bit index values (32-bit index values in 32-bit mode, or in a 32-bit implementation of the architecture). Scaled immediate values are unsigned and are scaled by the size of the access.

3.2.1.1 Base + Scaled Immediate Addressing - non-update form

In the Base + Scaled Immediate addressing mode, parameter **a** holds a 64-bit pointer value (32-bit pointer in 32-bit mode, or in a 32-bit implementation of the architecture), or a value of zero (if parameter **a** =r0), and a 5-bit unsigned immediate value provided by parameter **b** which is zero-extended and scaled (shifted left) by 1, 2, or 3 depending on the size (halfword, word, or doubleword) of the access. The sum of the parameter **a** value and the zero-extended scaled immediate (parameter **b**) form the effective address:

```
if (a = r0) then temp ← 640
else temp ← (a0:63)
SCL ←{1,2,3} // halfword, word, or doubleword
EA ←temp + EXTZ(b*SCL)
```

3.2.1.2 Base + Index Addressing - non-update form

In the Base + Index addressing mode, parameter **a** holds a 64-bit pointer value (32-bit pointer in 32-bit mode, or in a 32-bit implementation of the architecture) or a value of zero (if **a**=r0), while parameter **b** provides a 64-bit index (32-bit index in 32-bit mode, or in a 32-bit implementation of the architecture). The sum forms the effective address:

```
if (a = r0) then temp ← 64 0
else temp ← (a0:63)
EA ← temp + (b)
```

3.2.2 Addressing Modes - Update forms

The Base + Scaled Immediate addressing mode is also provided with an update form. As in the non-update form, base register parameter **a** holds 64-bit pointer values (32-bit pointers in a 32-bit implementation of the architecture). For the update form of the Base+Scaled Immediate addressing mode, the same effective address calculation is used as defined in Section 3.2.1.1, "Base + Scaled Immediate Addressing - non-update form, and the calculated effective address is placed into parameter **a** by the instruction.

For the Base + Scaled Immediate with update addressing mode, scaled immediate values of 0 are reserved for future definition and are treated as illegal. Instruction encodings with parameter **a**=0 are also reserved for future definition and are treated as illegal instructions.

3.2.3 Addressing Modes - Modify forms

NOTE: definitions for how parameter a is used in computing 64-bit addresses for 64-bit implementations operating in 64-bit mode is TBD.

The Base + Index addressing mode is also provided with a set of Modify forms. In the modify forms, parameter **b** holds 64-bit pointer values (32-bit pointers in a 32-bit implementation of the architecture), while parameter **a** is used to provide an index value, as well as to provide specialized control information for performing a post-modification to the lower 32 bits of parameter **a**.

Modify forms are provided to allow for parallel address computations to occur, which are useful for sequential accessing of arrays, lists, circular buffers, and other complex data structures. Modify forms of load and store instructions cause a calculated update value to be placed in the lower portion of parameter **a**. Support for specialized addressing modes are available when using Base + Index modify forms.

For the Base+Index modify forms, the modify calculation mode selection is based on a **`mode`** field in parameter **a** ($\mathbf{a}_{0:3}$). Modify forms modify the original value in parameter **a** based on an addressing calculation performed in parallel with the load or store instruction, which may or may not be the value of the effective address of the load or store instruction, depending on the actual calculation mode. This is in contrast to normal "update forms" of the Power Arch load and store

instructions since the new value placed into parameter **a** need not correspond to the effective address of the load or store.

Three modify calculation modes are currently defined, and are selected by the value in parameter $\mathbf{a}_{0:3}$:

- Linear addressing: mode = 0000
- Circular addressing: mode = 1000
- Bit-reversed addressing: mode = 1010

All other mode encodings are reserved, and either result in an unimplemented instruction exception, or a boundedly undefined result depending on the implementation.

Instruction encodings with **a**=0 are reserved for future definition and are treated as illegal instructions.

3.2.3.1 Linear addressing update mode

Linear addressing update calculation mode causes the sum of $\mathbf{a}_{32:63}$ and $\mathbf{b}_{32:63}$ to be placed into $\mathbf{a}_{32:63}$:

```
if(mode=0000) then
        a32:63 ← a32:63 + b32:63
```

3.2.3.2 Circular addressing modify mode

Circular addressing modify mode is provided to support addressing of circular buffers. Circular addressing mode causes a circular increment to be performed on a portion of parameter $\mathbf{a}_{32:63}$ (the circular buffer index portion of parameter **a**) after the EA calculation, using the Offset and Length specifiers in parameter **a** and the result is placed into parameter $\mathbf{a}_{32:63}$. Parameter $\mathbf{a}_{0:31}$ is left unchanged. Parameter $\mathbf{a}_{32:63}$ must be $>=_{si}$ 0 and $<=_{ui}$ Length, and the magnitude of Offset must be <= Length+1, or the result is boundedly undefined. Parameter **a** must point to a doubleword boundary in memory, and Length+1 must be a multiple of eight bytes or an alignment error will be generated.

The following shows how parameter **a** is used in forming the update value for mode 1000 (circinc).

0....3	4 5	6....13	14 15	16 31	32 ... 63	
Mode (1000)	-	**Offset** (signed)	-	**Length** (unsigned)	**Index** (must always be positive and $<=_{ui}$ **Length**)	**a**

```
Offset0:7 ←a8:15; // signed byte offset, must be <= Length+1
Length0:15 ←a16:31; // unsigned buffer length-1 in bytes. Length is byte index of
                     // last byte in buffer.
                     // buffer must be aligned on a doubleword boundary, and be a
                     // multiple of 8 bytes, i.e. Length13:15 =3'b111.
Index0:31 ←a32:63;  // index into buffer, must be <=ui Length0:15, (so always >=si 0).
```

```
if ((Offset0 = 0) & ((EXTS32(Offset0:7) + Index0:31) >ui EXTZ32(Length0:15)) then
    a32:63 ←Index0:31 + EXTS32(Offset0:7) - EXTZ32(Length0:15) - 1; // wrap at end

elseif (Offset0 = 1) & ((EXTS32(Offset0:7) + Index0:31) <si 0)) then
    a32:63 ←Index0:31 + EXTS32(Offset0:7) + EXTZ32(Length0:15) + 1; // wrap at start

else a32:63 ←Index0:31 + EXTS32(Offset0:7);
```

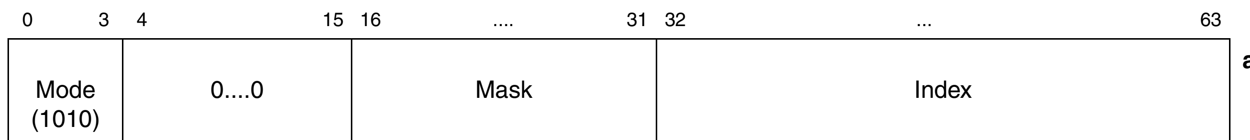
Note that **misalignment** may cause the operand fetched to span the “virtual boundary” between the last byte of the buffer at byte Buffer[Length] and the first byte of the Buffer at byte Buffer[0].

3.2.3.3 Bit-reversed addressing modify mode

Bit-reversed addressing modify calculation mode is provided to support addressing of buffers and arrays used in FFT calculations.

When using bit-reversed addressing modify mode, a bit-reversed increment is performed on $\mathbf{a}_{32:63}$ after the EA calculation, using a Mask specifier in parameter **a**. The Mask specifier is also used to indicate the bits of $\mathbf{a}_{32:63}$ which are updated.

The following figure shows how parameter **a** is used in forming the update value for bit-reversed addressing update mode. Note that the computation is similar to the **brinc** instruction computation, but the mask is applied to updating only those bits of parameter **a** indicated by a ‘1’ in the mask value, unlike in the **brinc** instruction, in which all low order bits of parameter **d** corresponding to the maximum mask size are updated.




```
Mask ←a16:31

// mask value is log2(#points)1, zero extended, then left shifted log2(element size
// in bytes). e.g., a 16 point FFT on halfwords has a mask of 16‘b0000000000011110

tempA ←a48:63 // up to 64Kbytes in a single FFT
tempD ←bitreverse(1 + bitreverse(tempA | ~Mask)))
a32:63 ←a32:47 || ((a48:63 & ~Mask) |(tempD & Mask)) // different than brinc. allows main
pointer sharing to multiple buffers less than 64Kbyte in size.
```

3.3 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 nth 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, ^{50} is equivalent to 0b0_0000. $^n1$ means a field of n bits with each bit equal to 1. Thus, ^{51} 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.

3.4 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

Table 3-4. Instruction Field Descriptions (continued)

Field	Description
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
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

Table 3-4. Instruction Field Descriptions (continued)

Field	Description
SH (16–20)	Specifies a shift amount in Rotate Word Immediate and Shift Word Immediate instructions
sh (30 ǁ 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 (16–20ǁ11–15)	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

3.5 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
$\times$	Multiplication
$\div$	Division (yielding quotient)

Table 3-5. RTL Notation (continued)

Notation	Meaning
$+_{dp}$	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
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

Table 3-5. RTL Notation (continued)

Notation	Meaning
<, ≨>, ≥	Signed comparison relations
$<_u$, $>_u$	Unsigned comparison relations
?	Unordered comparison relation
&, $\vert$	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.
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 $x \Vert x$ 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.

Table 3-5. RTL Notation (continued)

Notation	Meaning
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
calc_update	Load and Store Indexed with-Update parameter a Calculation

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
$\Vert$	Left to right
=, ≠, <, ≤, >, ≥, $<_u$, $>_u$, ?	Left to right
&, ⊕, ≡	Left to right
$\vert$	Left to right
: (range)	None
←	None

3.6 Overview of Intrinsic Definition

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.

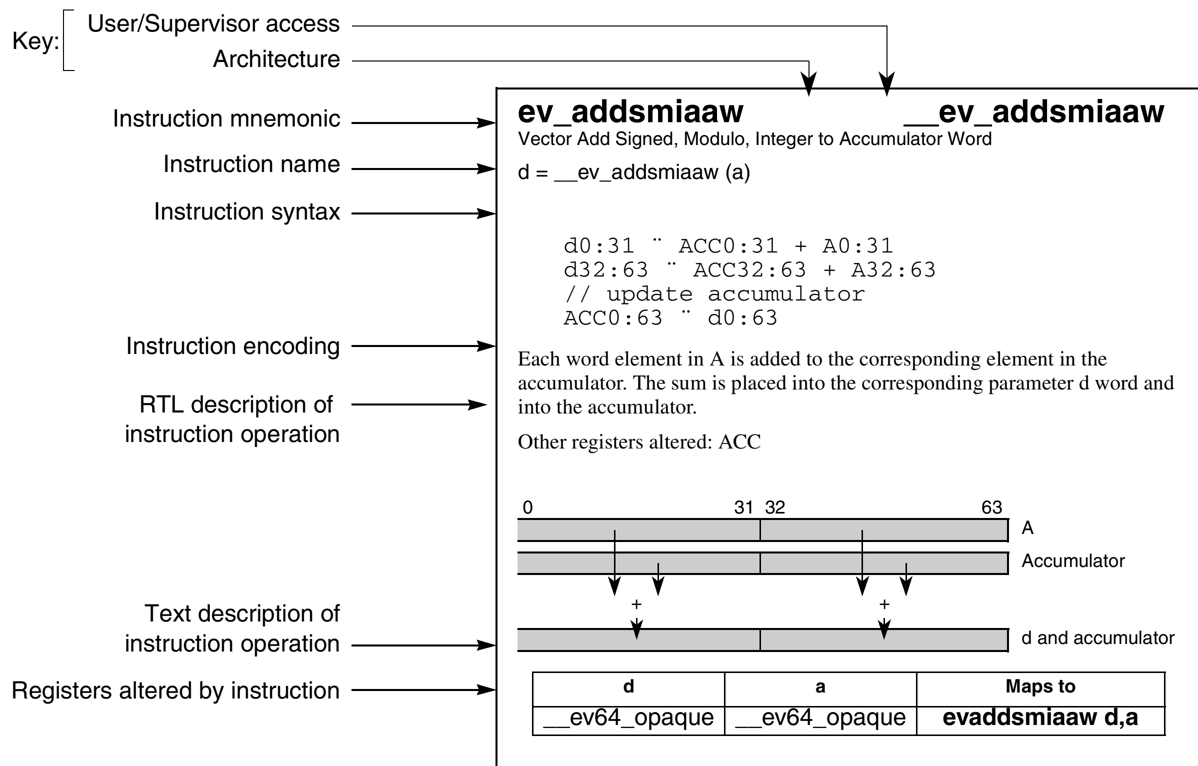


Figure 3-3. Instruction Description

3.6.1 Saturation, Shift, and Bit Reverse Models

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

3.6.1.1 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
```

3.6.1.2 Shift

```
SL(value, cnt)

if cnt > 31 then
    return 0
else
```

```
return (value << cnt)
```

3.6.1.3 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
```

3.6.1.4 Load and Store Indexed with-Update Calculations

NOTE: THIS IS STILL IN DEFINITION and is NOT FINALIZED

```
Calc_update(a,b)

mode ← b0:2

if(mode=000) then
   if 64-bit mode then a0:63 ← a0:63 + EXTS(b3:63)
   else a32:63 ← a32:63 + b32:63
elseif (mode=001) then
   Offset ← b8:15;// signed byte offset
   Length ← b16:31;// unsigned buffer length-1 in bytes. Length is byte index of last
                   // byte in buffer. buffer must be aligned on a power of 2 byte boundary.
                   // Length must be >7.
   Mod ← CEIL(log2(Length+1)); // power of 2 byte boundary of 1st buffer element.
                               // Mod must be >3.
   a(64-Mod):63 ← (a48:63 + exts(Offset))(64-Mod):63;
elseif (mode=010) then
   Mask ← b16:31
   // mask value is log2(#points)1, zero extended, then left shifted log2(element size
   // in bytes). e.g., a 16 point FFT on halfwords has a mask of 16'b0000000000011110
   tmpa ← a48:63 // up to 64Kbytes in a single FFT
   tmpd ← bitreverse(1 + bitreverse(tmpa | ~Mask)))
   a32:63 ← (a32:63 & ~Mask) || (tmpd & Mask)
else "program illegal exception"
```

3.7 Intrinsic Definitions

The intrinsic definitions are split into the integer operations defined by the Enhanced Signal Processing Engine (SPE2) and the Embedded Floating-Point Version 2 (EFP2) APU.

3.7.1 SPE2 Intrinsic Definitions

__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 || (temp1 & 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 SPE2 Unavailable exceptions, regardless of the state of MSRSPE.

__ev_circinc

__ev_circinc

Circular Increment

d = __ev_circinc (a,b)

```
Offset_{0:15} ← b_{48:63}; // signed byte offset in b
Length_{0:15} ← a_{16:31}; // unsigned buffer length-1 in bytes. Length is index of
                           // last byte in buffer.
Index_{0:31} ← a_{32:63};  // index into buffer, must be positive.

if ((Offset_0 = 0) & ((EXTS_32(Offset_{0:15}) + Index_{0:31}) >_ui EXTZ_32(Length_{0:15}))) then
    d_{32:63} ← Index_{0:31} + EXTS_32(Offset_{0:15}) - EXTZ_32(Length_{0:15}) - 1; // wrap at end

elseif (Offset_0 = 1) & ((EXTS_32(Offset_{0:15}) + Index_{0:31}) <_si 0)) then
    d_{32:63} ← Index_{0:31} + EXTS_32(Offset_{0:15}) + EXTZ_32(Length_{0:15}) + 1; // wrap at start

else d_{32:63} ← Index_{0:31} + EXTS_32(Offset_{0:15});
```

Note: Length+1 must be a multiple of 8 bytes, Index must be within buffer, and abs(Offset) must be less or equal to Length+1 for the calculation to be used properly in a future buffer access in all implementations.

circinc provides a way for software to modify a circular buffer index value. Parameter **a** contains the index value into a circular buffer that contains data accessed by a load or store instruction using circular addressing. Parameter **b** contains an offset value that allows the index to be updated with circular addressing. Typically this instruction is used to update a new buffer pointer value following a sequence of buffer accesses, to allow the pointer into the buffer to be moved to the next starting element for a future calculation. For forward compatibility, the Offset value in parameter b should be a 32-bit value with a magnitude <= 16-bits.

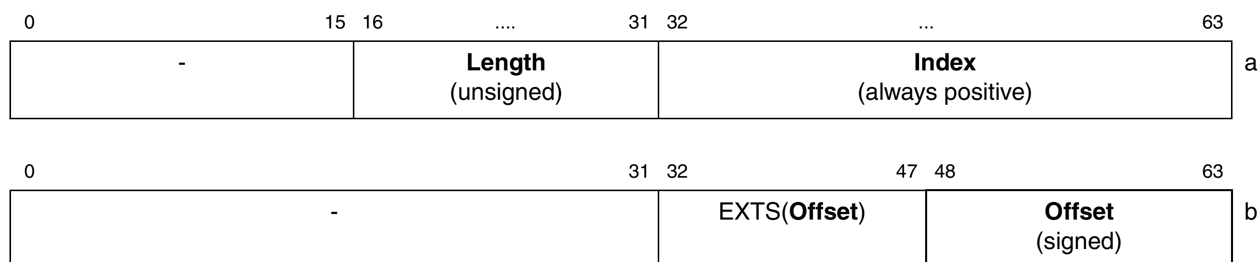


Figure 3-4.

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**circinc d,a,b**

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

Architecture Note: An implementation can restrict the allowable values of the Length, Offset, and Index. Unallowed values may generate a boundedly undefined result. Implementations must support a set of values that allow for calculations supporting circular buffers that are a multiple of 8 bytes, where the index points to an element of the buffer, and the magnitude of the offset is <= the size of the buffer.

__ev_abs __ev_abs

Vector Absolute Value

d = __ev_abs (a)

```
d0:31 ←ABS(a0:31)
d32:63 ←ABS(a32:63)
```

The absolute value of each element of parameter **a** 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.

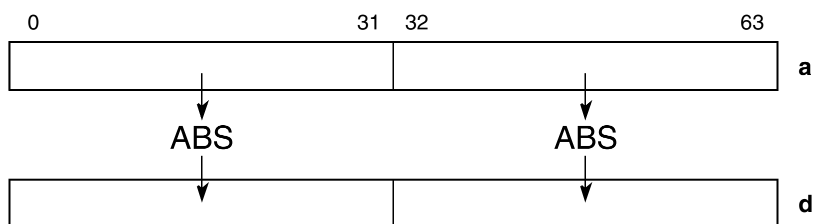


Figure 3-5. Vector Absolute Value (__ev_abs)

d	a	Maps to
__ev64_opaque__	__ev64_opaque__	**evabs d,a**

__ev_absb __ev_absb

Vector Absolute Value Byte

d = __ev_absb (a)

```
d0:7 ←ABS(a0:7)
d8:15 ←ABS(a8:15)
d16:23 ←ABS(a16:23)
d24:31 ←ABS(a24:31)
d32:39 ←ABS(a32:39)
d40:47 ←ABS(a40:47)
d48:55 ←ABS(a48:55)
d56:63 ←ABS(a56:63)
```

The absolute value of each byte element of parameter **a** is placed in the corresponding elements of parameter **d**. An absolute value of 0x80 (most negative number) returns 0x80. No overflow is detected.

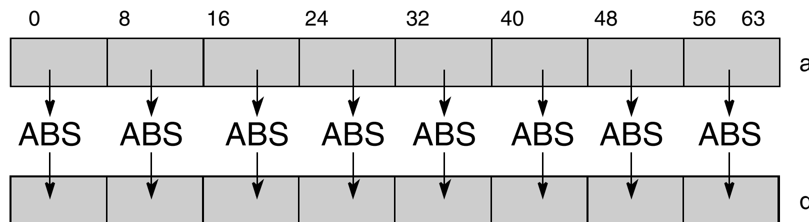


Figure 3-6. Vector Absolute Value Byte (__ev_absb)

d	a	Maps to
__ev64_opaque__	__ev64_opaque__	**evabsb d,a**

__ev_absbs __ev_absbs

Vector Absolute Value Byte and Saturate

d = __ev_absbs (a)

```
if (a0:7 = 0x80) then
    d0:7 ← 0x7F
    ovh ← 1
else
    d0:7 ← ABS(a0:7)
    ovh ← 0
if (a8:15 = 0x80) then
    d8:15 ← 0x7F
    ovh ← 1
else d8:15 ← ABS(a8:15)
if (a16:23 = 0x80) then
    d16:23 ← 0x7F
    ovh ← 1
else d16:23 ← ABS(a16:23)
if (a24:31 = 0x80) then
    d24:31 ← 0x7F
    ovh ← 1
else d24:31 ← ABS(a24:31)
if (a32:39 = 0x80) then
    d32:39 ← 0x7F
    ovl ← 1
else
    d32:39 ← ABS(a32:39)
    ovl ← 0
if (a40:47 = 0x80) then
    d40:47 ← 0x7F
    ovl ← 1
else d40:47 ← ABS(a40:47)
if (a48:55 = 0x80) then
    d48:55 ← 0x7F
    ovl ← 1
else d48:55 ← ABS(a48:55)
if (a56:63 = 0x80) then
    d56:63 ← 0x7F
    ovl ← 1
else d56:63 ← ABS(a56:63)

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

The absolute value of each signed byte element of parameter **a** is placed into parameter **d**. The absolute value of 0x80 (most negative number) returns 0x7F. Any overflow is reported in the SPEFSCR overflow and summary overflow bits.

Other registers altered: SPEFSCR

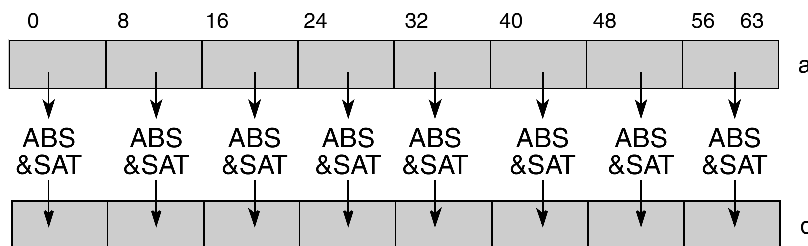


Figure 3-7. Vector Absolute Value Byte and Saturate (__ev_absbs)

d	a	Maps to
__ev64_opaque__	__ev64_opaque__	**evabsbs d,a**

__ev_absd

__ev_absd

Vector Absolute Value Doubleword

d = __ev_absd (a)

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

The absolute value of the doubleword in parameter **a** is placed into parameter **d**. The absolute value of 0x8000_0000_0000_0000 (most negative number) returns 0x8000_0000_0000_0000. No overflow is detected.

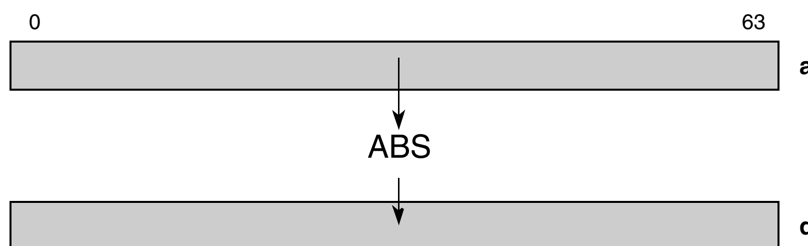


Figure 3-8. Vector Absolute Value Doubleword (__ev_absd)

d	a	Maps to
__ev64_opaque__	__ev64_opaque__	**evabsd d,a**

__ev_absdifsb __ev_absdifsb

Vector Absolute Difference of Signed Bytes

d = __ev_absdifsb (a,b)

$d_{0:7} \leftarrow ABS(b_{0:7} - a_{0:7})$
$d_{8:15} \leftarrow ABS(b_{8:15} - a_{8:15})$
$d_{16:23} \leftarrow ABS(b_{16:23} - a_{16:23})$
$d_{24:31} \leftarrow ABS(b_{24:31} - a_{24:31})$
$d_{32:39} \leftarrow ABS(b_{32:39} - a_{32:39})$
$d_{40:47} \leftarrow ABS(b_{40:47} - a_{40:47})$
$d_{48:55} \leftarrow ABS(b_{48:55} - a_{48:55})$
$d_{56:63} \leftarrow ABS(b_{56:63} - a_{56:63})$

The eight signed byte elements of parameter **a** are subtracted from the corresponding elements of parameter **b** and the absolute values of the differences are placed into parameter **d**.

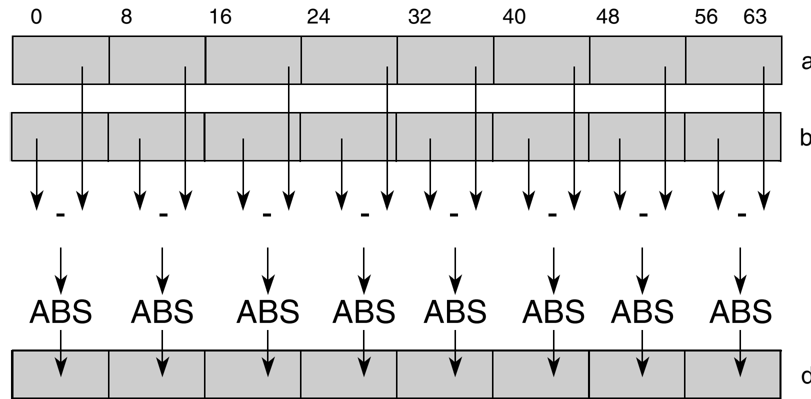


Figure 3-9. Vector Absolute Difference of Signed Bytes (__ev_absdifsb)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evabsdifsb d,a,b**

__ev_absdifsh

__ev_absdifsh

Vector Absolute Difference of Signed Half Words

d = __ev_absdifsh (a,b)

$d_{0:15} \leftarrow ABS(b_{0:15} - a_{0:15})$
$d_{16:31} \leftarrow ABS(b_{16:31} - a_{16:31})$
$d_{32:47} \leftarrow ABS(b_{32:47} - a_{32:47})$
$d_{48:63} \leftarrow ABS(b_{48:63} - a_{48:63})$

The four signed half word elements of parameter **a** are subtracted from the corresponding elements of parameter **b** and the absolute values of the differences are placed into parameter **d**.

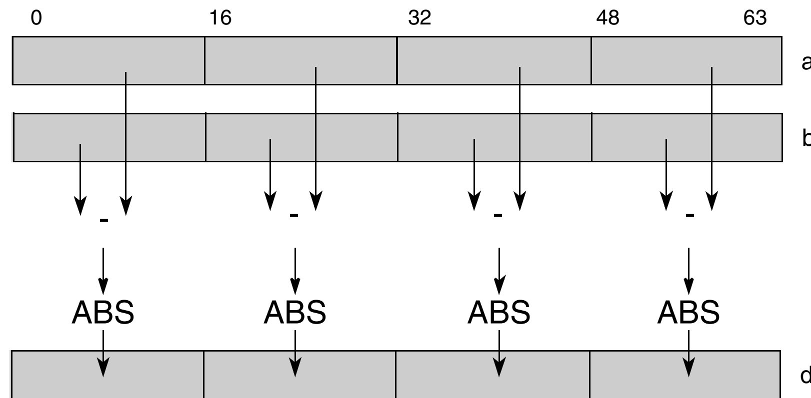


Figure 3-10. Vector Absolute Difference of Signed Half Words (__ev_absdifsh)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evabsdifsh d,a,b**

__ev_absdifsw __ev_absdifsw

Vector Absolute Difference of Signed Words

d = __ev_absdifsw (a,b)

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

The signed word elements of parameter **a** are subtracted from the corresponding elements of parameter **b** and the absolute values of the differences are placed into parameter **d**.

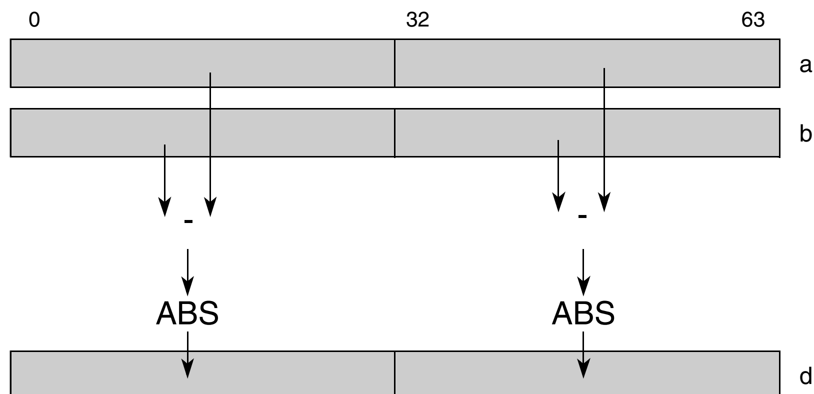


Figure 3-11. Vector Absolute Difference of Signed Words (__ev_absdifsw)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evabsdifsw d,a,b**

__ev_absdifub __ev_absdifub

Vector Absolute Difference of Unsigned Bytes

d = __ev_absdifub (a,b)

$d_{0:7} \leftarrow ABS(b_{0:7} - a_{0:7})$
$d_{8:15} \leftarrow ABS(b_{8:15} - a_{8:15})$
$d_{16:23} \leftarrow ABS(b_{16:23} - a_{16:23})$
$d_{24:31} \leftarrow ABS(b_{24:31} - a_{24:31})$
$d_{32:39} \leftarrow ABS(b_{32:39} - a_{32:39})$
$d_{40:47} \leftarrow ABS(b_{40:47} - a_{40:47})$
$d_{48:55} \leftarrow ABS(b_{48:55} - a_{48:55})$
$d_{56:63} \leftarrow ABS(b_{56:63} - a_{56:63})$

The eight unsigned byte elements of parameter **a** are subtracted from the corresponding elements of parameter **b** and the absolute values of the differences are placed into parameter **d**.

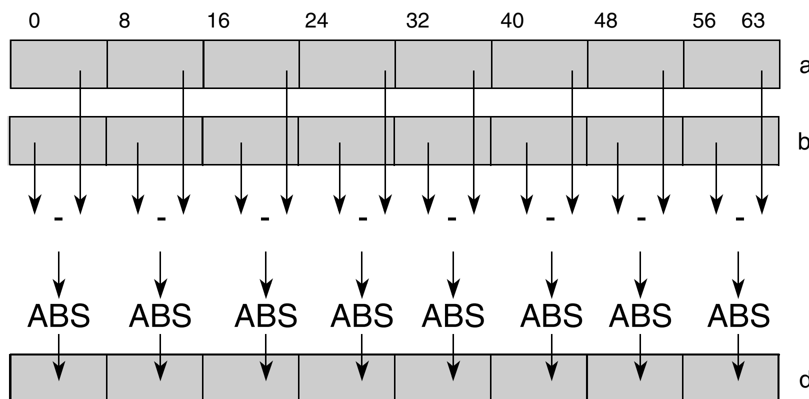


Figure 3-12. Vector Absolute Difference of Unsigned Bytes (__ev_absdifub)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evabsdifub d,a,b**

__ev_absdifuh

__ev_absdifuh

Vector Absolute Difference of Unsigned Half Words

d = __ev_absdifuh (a,b)

$d_{0:15} \leftarrow ABS(b_{0:15} - a_{0:15})$
$d_{16:31} \leftarrow ABS(b_{16:31} - a_{16:31})$
$d_{32:47} \leftarrow ABS(b_{32:47} - a_{32:47})$
$d_{48:63} \leftarrow ABS(b_{48:63} - a_{48:63})$

The four unsigned half word elements of parameter **a** are subtracted from the corresponding elements of parameter **b** and the absolute values of the differences are placed into parameter **d**.

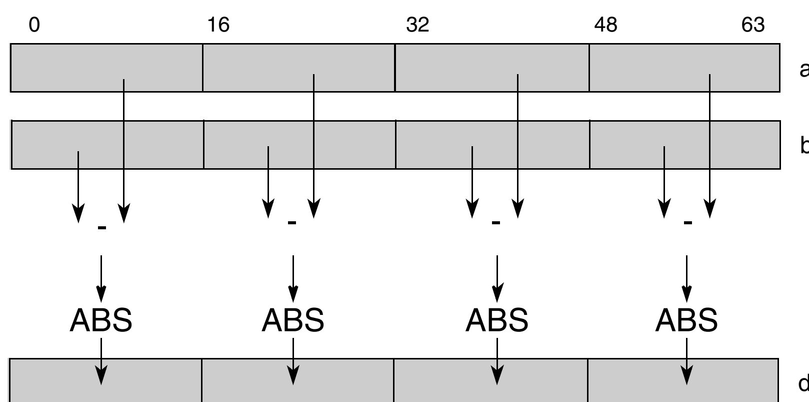


Figure 3-13. Vector Absolute Difference of Unsigned Half Words (__ev_absdifuh)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evabsdifuh d,a,b**

__ev_absdifuw

__ev_absdifuw

Vector Absolute Difference of Unsigned Words

d = __ev_absdifuw (a,b)

```
d0:31 ←ABS(b0:31 - a0:31)
d32:63 ←ABS(b32:63 - a32:63)
```

The unsigned word elements of parameter **a** are subtracted from the corresponding elements of parameter **b** and the absolute values of the differences are placed into parameter **d**.

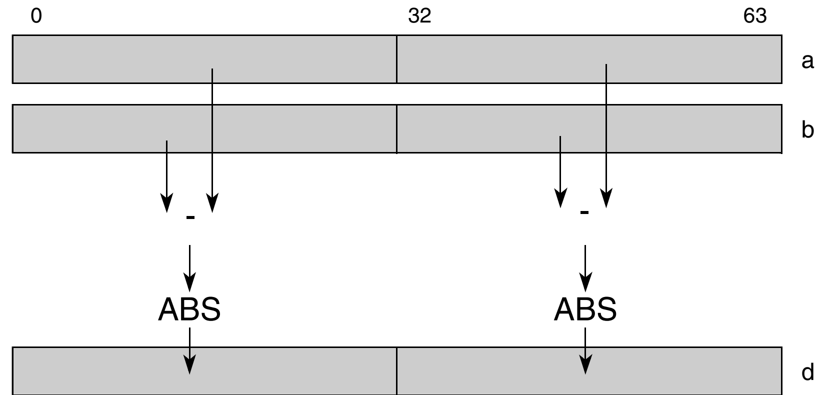


Figure 3-14. Vector Absolute Difference of Unsigned Words (__ev_absdifuw)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evabsdifuw d,a,b**

__ev_absds __ev_absds

Vector Absolute Value Doubleword and Saturate

d = __ev_absds (a)

```
if (a0:63 = 0x8000_0000_0000_0000) then
    d0:63 ←0x7FFF_FFFF_FFFF_FFFF
    ov ←1
else
    d0:63 ←ABS(a0:63)
    ov ←0
endif

SPEFSCROVH ←0
SPEFSCROV ←ov
SPEFSCRSOV ←SPEFSCRSOV | ovl
```

The absolute value of the doubleword in parameter **a** is placed into parameter **d**. The absolute value of 0x8000_0000_0000_0000 (most negative number) returns 0x7FFF_FFFF_FFFF_FFFF. Any overflow is reported in the SPEFSCR overflow and summary overflow bits.

Other registers altered: SPEFSCR

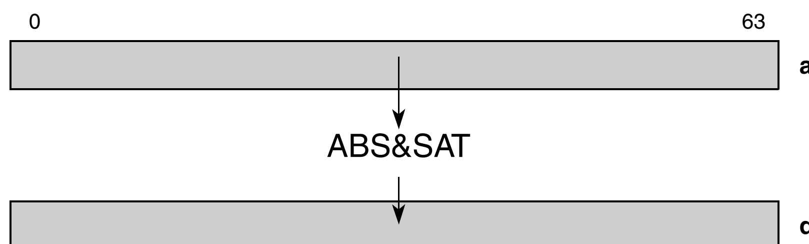


Figure 3-15. Vector Absolute Value Doubleword and Saturate (__ev_absds)

d	a	Maps to
__ev64_opaque__	__ev64_opaque__	**evabsds d,a**

__ev_absh

__ev_absh

Vector Absolute Value Half Word

d = __ev_absh (a)

$d_{0:15} \leftarrow ABS(a_{0:15})$
$d_{16:31} \leftarrow ABS(a_{16:31})$
$d_{32:47} \leftarrow ABS(a_{32:47})$
$d_{48:63} \leftarrow ABS(a_{48:63})$

The absolute value of each half word element of parameter **a** is placed in the corresponding elements of parameter **d**. An absolute value of 0x8000 (most negative number) returns 0x8000. No overflow is detected.

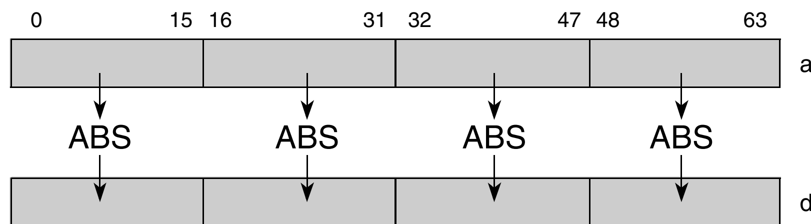


Figure 3-16. Vector Absolute Value Half Word (__ev_absh)

d	a	Maps to
__ev64_opaque__	__ev64_opaque__	**evabsh d,a**

__ev_abshs __ev_abshs

Vector Absolute Value Half Word and Saturate

d = __ev_abshs (a)

```
if (a0:15 = 0x8000) then
    d0:15 ←0x7FFF
    ovh ←1
else
    d0:15 ←ABS(a0:15)
    ovh ←0
endif
if (a16:31 = 0x8000) then
    d16:31 ←0x7FFF
    ovh ←1
else d16:31 ←ABS(a16:31)
if (a32:47 = 0x8000) then
    d32:47 ←0x7FFF
    ovl ←1
else
    d32:47 ←ABS(a32:47)
    ovl ←0
endif
if (a48:63 = 0x8000) then
    d48:63 ←0x7FFF
    ovl ←1
else d48:63 ←ABS(a48:63)

SPEFSCR_OVH ←ovh
SPEFSCR_OV ←ovl
SPEFSCR_SOVH ←SPEFSCR_SOVH | ovh
SPEFSCR_SOV ←SPEFSCR_SOV | ovl
```

The absolute value of each half word element of parameter **a** is placed into parameter **d**. The absolute value of 0x8000 (most negative number) returns 0x7FFF. Any overflow is reported in the SPEFSCR overflow and summary overflow bits.

Other registers altered: SPEFSCR

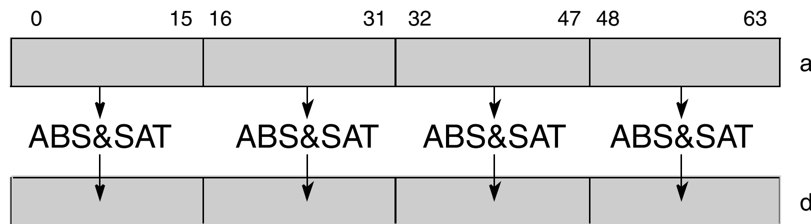


Figure 3-17. Vector Absolute Value Half Word and Saturate (__ev_abshs)

d	a	Maps to
__ev64_opaque__	__ev64_opaque__	**evabshs d,a**

__ev_abss __ev_abss

Vector Absolute Value (Word) and Saturate

d = __ev_abss (a)

```
if (a0:31 = 0x8000_0000) then
    d0:31 ←0x7FFF_FFFF
    ovh ←1
else
    d0:31 ←ABS(a0:31)
    ovh ←0
endif
if (a32:63 = 0x8000_0000) then
    d32:63 ←0x7FFF_FFFF
    ovl ←1
else
    d32:63 ←ABS(a32:63)
    ovl ←0
endif

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

The absolute value of each word element of parameter **a** is placed into parameter **d**. The absolute value of 0x8000_0000 (most negative number) returns 0x7FFF_FFFF. Any overflow is reported in the SPEFSCR overflow and summary overflow bits.

Other registers altered: SPEFSCR

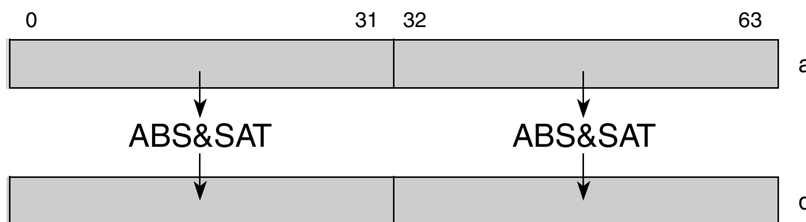


Figure 3-18. Vector Absolute Value (Word) and Saturate (__ev_abss)

d	a	Maps to
__ev64_opaque__	__ev64_opaque__	**evabss d,a**

__ev_add2subf2h __ev_add2subf2h

Vector Add Upper 2 and Subtract from Lower 2 Half Words

d = __ev_add2subf2h (a,b)

```
d0:15 ←b0:15 + a0:15// Modulo sum
d16:31 ←b16:31 + a16:31// Modulo sum
d32:47 ←b32:47 - a32:47// Modulo difference, b - a
d48:63 ←b48:63 - a48:63// Modulo difference, b - a
```

The upper two half word elements of parameter **a** are added to the upper 2 half word elements of parameter **b**, the lower two half word elements of parameter **a** are subtracted from the lower two half word elements of parameter **b**, and the results are placed in parmaeter **d**. The sum and difference are modulo.

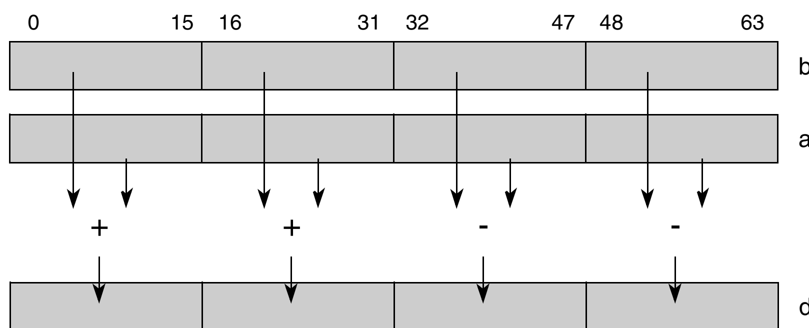


Figure 3-19. Vector Add upper 2 and Subtract from lower 2 Half Words (__ev_add2subf2h)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evadd2subf2h d,a,b**

__ev_add2subf2hss __ev_add2subf2hss

Vector Add Upper 2 and Subtract from Lower 2 Half Words Signed and Saturate

d = __ev_add2subf2hss (a,b)

```
// h0
temp0:31 ←EXTS(b0:15) + EXTS(a0:15)
ovh0 ←temp15 ⊕ temp16
d0:15 ←SATURATE(ovh0, temp15, 0x8000,
                       0x7fff, temp16:31)

// h1
temp0:31 ←EXTS(b16:31) + EXTS(a16:31)
ovh1 ←temp15 ⊕ temp16
d16:31 ←SATURATE(ovh1, temp15, 0x8000,
                       0x7fff, temp16:31)

// h2
temp0:31 ←EXTS(b32:47) - EXTS(a32:47)
ovh2 ←temp15 ⊕ temp16
d32:47 ←SATURATE(ovh2, temp15, 0x8000,
                       0x7fff, temp16:31)

// h3
temp0:31 ←EXTS(b48:63) - EXTS(a48:63)
ovh3 ←temp15 ⊕ temp16
d48:63 ←SATURATE(ovh3, temp15, 0x8000,
                       0x7fff, temp16:31)

ovh ←ovh0 | ovh1
ovl ←ovh2 | ovh3

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

The upper two signed half word elements of parameter **a** are added to the upper two signed half word elements of parameter **b**, the lower two signed half word elements of parameter **a** are subtracted from the lower two signed half word elements of parameter **b**, saturating if overflow or underflow occurs, and the results are placed in parameter **d**. Any overflow or underflow is recorded in the SPEFSCR overflow and summary overflow bits.

Other registers altered: SPEFSCR

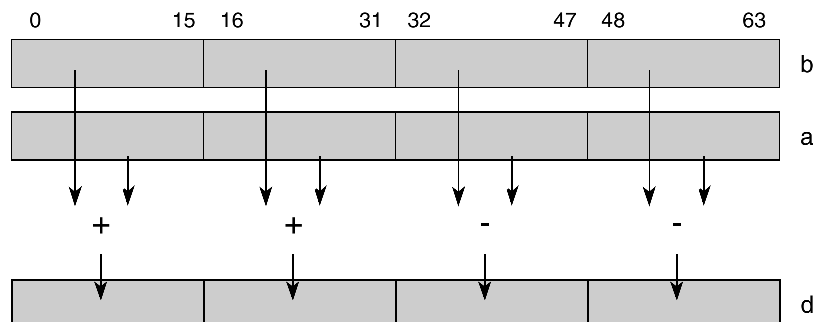


Figure 3-20. Vector Add upper 2 and Subtract from lower 2 Half Words Signed and Saturate (__ev_add2subf2hss)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evadd2subf2hss d,a,b**

__ev_addsmiaa

__ev_addsmiaa

Vector Add to Accumulator

d = __ev_addsmiaa (a)

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

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

The 64-bit value in parameter **a** is added to the accumulator and the results are placed into parameter **d** and the accumulator.

Other registers altered: ACC

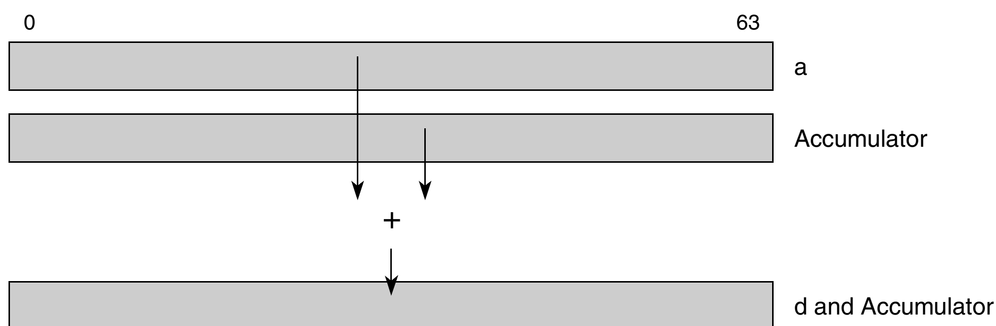


Figure 3-21. Vector Add to Accumulator (__ev_addsmiaa)

d	a	Maps to
__ev64_opaque__	__ev64_opaque__	**evaddsmiaa d,a**

__ev_addb

__ev_addb

Vector Add Bytes

d = __ev_addb (a,b)

$d_{0:7} \leftarrow a_{0:7} + b_{0:7}$ // Modulo sum
$d_{8:15} \leftarrow a_{8:15} + b_{8:15}$ // Modulo sum
$d_{16:23} \leftarrow a_{16:23} + b_{16:23}$ // Modulo sum
$d_{24:31} \leftarrow a_{24:31} + b_{24:31}$ // Modulo sum
$d_{32:39} \leftarrow a_{32:39} + b_{32:39}$ // Modulo sum
$d_{40:47} \leftarrow a_{40:47} + b_{40:47}$ // Modulo sum
$d_{48:55} \leftarrow a_{48:55} + b_{48:55}$ // Modulo sum
$d_{56:63} \leftarrow a_{56:63} + b_{56:63}$ // Modulo sum

The eight byte elements of parameter **a** are added to the corresponding elements of parameter **b** and the results are placed in parameter **d**. The sum is a modulo sum.

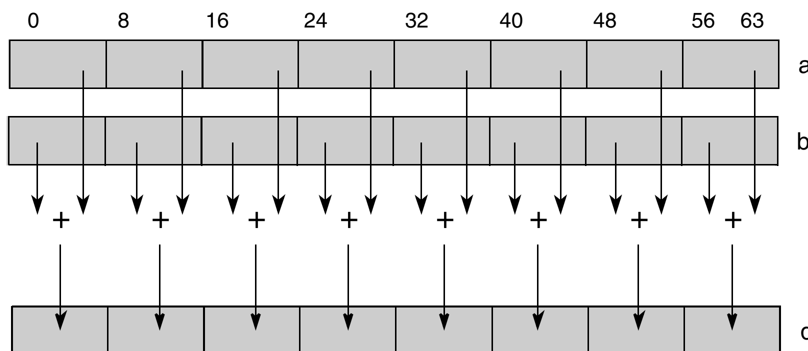


Figure 3-22. Vector Add Bytes (__ev_addb)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evaddb d,a,b**

__ev_addbss

__ev_addbss

Vector Add Bytes Signed and Saturate

d = __ev_addbss (a,b)

```
temp0:8 ←EXTS(a0:7) + EXTS(b0:7)
ovb0 ←temp0 ⊕ temp1
d0:7 ←SATURATE(ovb0,temp0,0x80,0x7f,temp 1:8)

temp0:8 ←EXTS(a8:15) + EXTS(b8:15)
ovb1 ←temp0 ⊕ temp1
d8:15 ←SATURATE(ovb1,temp0,0x80,0x7f,temp1:8)

temp0:8 ←EXTS(a16:23) + EXTS(b16:23)
ovb2 ←temp0 ⊕ temp1
d16:23 ←SATURATE(ovb2,temp0,0x80,0x7f,temp1:8)

temp0:8 ←EXTS(a24:31) + EXTS(b24:31)
ovb3 ←temp0 ⊕ temp1
d24:31 ←SATURATE(ovb3,temp0,0x80,0x7f,temp1:8)

temp0:8 ←EXTS(a32:39) + EXTS(b32:39)
ovb4 ←temp0 ⊕ temp1
d32:39 ←SATURATE(ovb4,temp0,0x80,0x7f,temp1:8)

temp0:8 ←EXTS(a40:47) + EXTS(b40:47)
ovb5 ←temp0 ⊕ temp1
d40:47 ←SATURATE(ovb5,temp0,0x80,0x7f,temp1:8

temp0:8 ←EXTS(a48:55) + EXTS(b48:55)
ovb6 ←temp0 ⊕ temp1
d48:55 ←SATURATE(ovb6,temp0,0x80,0x7f,temp1:

temp0:8 ←EXTS(a56:63) + EXTS(b56:63)
ovb7 ←temp0 ⊕ temp1
d56:63 ←SATURATE(ovb7,temp0,0x80,0x7f,temp1:8

ovh ←ovb0 | ovb1 | ovb2 | ovb3
ovl ←ovh4 | ovh5 | ovb6 | ovb7
SPEFSCROVH ←ovh
SPEFSCROV ←ovl
SPEFSCRSOVH ←SPEFSCRSOVH | ovh
SPEFSCRSOV ←SPEFSCRSOV | ovl
```

The eight signed byte elements of parameter **a** are added to the corresponding signed elements of parameter **b**, saturating if overflow or underflow occurs, and the results are placed in parameter **d**. Any overflow or underflow is recorded in the SPEFSCR overflow and summary overflow bits.

Other registers altered: SPEFSCR

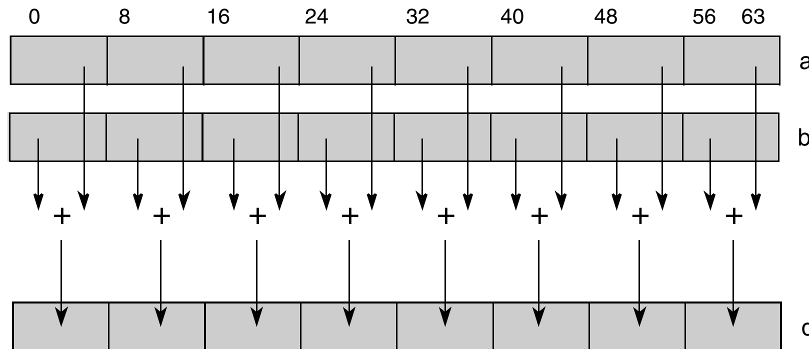


Figure 3-23. Vector Add Bytes Signed and Saturate (__ev_addbss)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evaddbss d,a,b**

__ev_addbus __ev_addbus

Vector Add Bytes Unsigned and Saturate

d = __ev_addbus (a,b)

```
temp0:8 ←EXTS(a0:7) + EXTS(b0:7)
ovb0 ←temp0 ⊕ temp1
d0:7 ←SATURATE(ovb0,temp0,0x80,0x7f,temp 1:8)

temp0:8 ←EXTZ(a0:7) + EXTZ(b0:7)
ovb0 ←temp0
d0:7 ←SATURATE(ovb0,temp0,0xff,0xff,temp1:8)

temp0:8 ←EXTZ(a8:15) + EXTZ(b8:15)
ovb1 ←temp0
d8:15 ←SATURATE(ovb1,temp0,0xff,0xff,temp1:8)

temp0:8 ←EXTZ(a16:23) + EXTZ(b16:23)
ovb2 ←temp0
d16:23 ←SATURATE(ovb2,temp0,0xff,0xff,temp1:8)

temp0:8 ←EXTZ(a24:31) + EXTZ(b24:31)
ovb3 ←temp0
d24:31 ←SATURATE(ovb3,temp0,0xff,0xff,temp1:8)

temp0:8 ←EXTZ(a32:39) + EXTZ(b32:39)
ovb4 ←temp0
d32:39 ←SATURATE(ovb4,temp0,0xff,0xff,temp1:8

temp0:8 ←EXTZ(a40:47) + EXTZ(b40:47)
ovb5 ←temp0
d40:47 ←SATURATE(ovb5,temp0,0xff,0xff,temp1:8

temp0:8 ←EXTZ(a48:55) + EXTZ(b48:55)
ovb6 ←temp0
d48:55 ←SATURATE(ovb6,temp0,0xff,0xff,temp1:8

temp0:8 ←EXTZ(a56:63) + EXTZ(b56:63)
ovb7 ←temp0
d56:63 ←SATURATE(ovb7,temp0,0xff,0xff,temp1:8

ovh ←ovb0 | ovb1 | ovb2 | ovb3
ovl ←ovh4 | ovh5 | ovb6 | ovb7
SPEFSCROVH ←ovh
SPEFSCROV ←ovl
SPEFSCRSOVH ←SPEFSCRSOVH | ovh
SPEFSCRSOV ←SPEFSCRSOV | ovl
```

The eight unsigned byte elements of parameter **a** are added to the corresponding unsigned elements of parameter **b**, saturating if overflow occurs, and the results are placed in parameter **d**. Any overflow is recorded in the SPEFSCR overflow and summary overflow bits.

Other registers altered: SPEFSCR

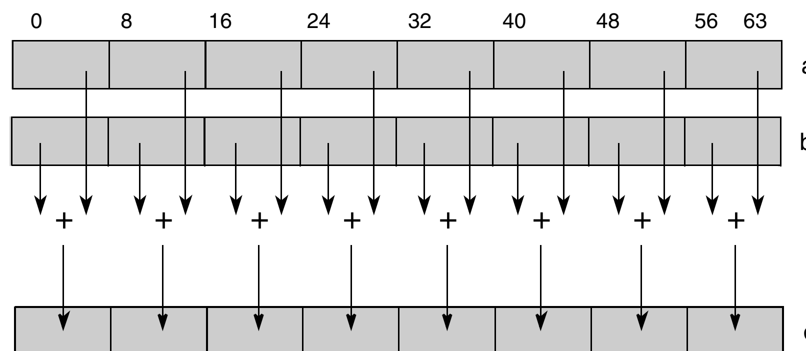


Figure 3-24. Vector Add Bytes Unsigned and Saturate (__ev_addbus)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evaddbus d,a,b**

__ev_addd

__ev_addd

Vector Add Doubleword

d = __ev_addd (a,b)

$d_{0:63} \leftarrow a_{0:63} + b_{0:63}$

The 64-bit value in parameter **a** is added to the 64-bit value in parameter **b** and the results are placed into parameter **d**.

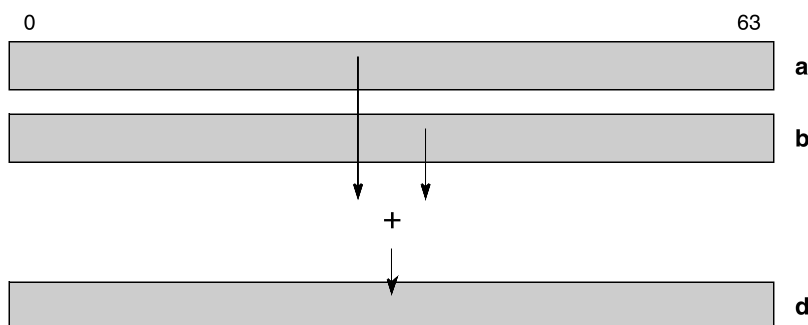


Figure 3-25. Vector Add Doubleword (__ev_addd)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evaddd d,a,b**

__ev_adddss __ev_adddss

Vector Add Doubleword Signed and Saturate

d = __ev_adddss (a,b)

```
temp0:64 ←EXTS65(b0:63) + EXTS65(a0:63)
ov ←temp0 ⊕ temp1
d0:63 ←SATURATE(ov, temp0, 0x8000_0000_0000_0000, 0x7fff_ffff_ffff_ffff, temp1:64)

SPEFSCROVH ←0
SPEFSCROV ←ov
SPEFSCRSOV ←SPEFSCRSOV | ov.
```

The signed doubleword in parameter **a** is added to the signed doubleword in parameter **b**, saturating if overflow or underflow occurs, and the result is placed into parameter **d**. Any overflow or underflow is recorded in the SPEFSCR overflow and summary overflow bits.

Other registers altered: SPEFSCR

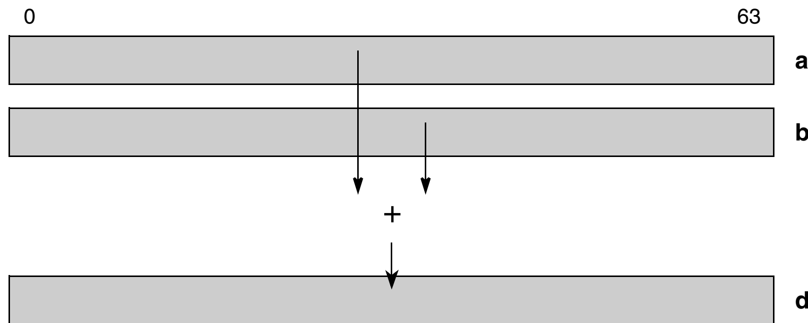


Figure 3-26. Vector Add Doubleword Signed and Saturate (__ev_adddss)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evadddss d,a,b**

__ev_adddus __ev_adddus

Vector Add Doubleword Unsigned and Saturate

d = __ev_adddus (a,b)

```
temp0:64 ←EXTZ65(b0:63) + EXTZ65(a0:63)
ov ←temp0
d0:63 ←SATURATE(ov, temp0, 0xffff_ffff_ffff_ffff, 0xffff_ffff_ffff_ffff, temp1:64)

SPEFSCROVH ←0
SPEFSCROV ←ov
SPEFSCRSOV ←SPEFSCRSOV | ov.
```

The unsigned doubleword in parameter **a** is added to the unsigned doubleword in parameter **b**, saturating if overflow or underflow occurs, and the result is placed into parameter **d**. Any overflow or underflow is recorded in the SPEFSCR overflow and summary overflow bits.

Other registers altered: SPEFSCR

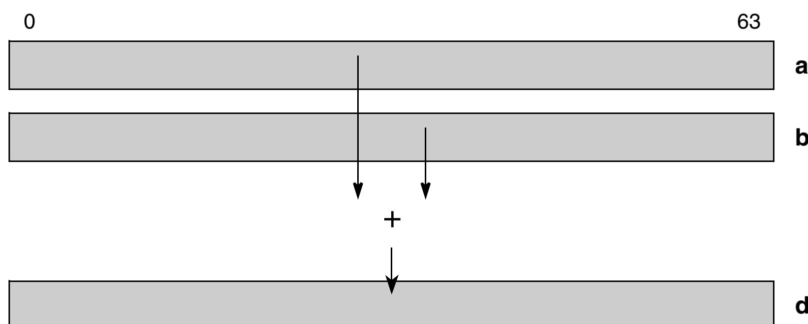


Figure 3-27. Vector Add Doubleword Unsigned and Saturate (__ev_adddus)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evadddus d,a,b**

__ev_addh

__ev_addh

Vector Add Half Words

d = __ev_addh (a,b)

```
d0:15 ←a0:15 + b0:15 // Modulo sum
d16:31 ←a16:31 + b16:31 // Modulo sum
d32:47 ←a32:47 + b32:47 // Modulo sum
d48:63 ←a48:63 + b48:63 // Modulo sum
```

The four half word elements of parameter **a** are added to the four half word elements of parameter **b** and the results are placed in parameter **d**. The sum is a modulo sum.

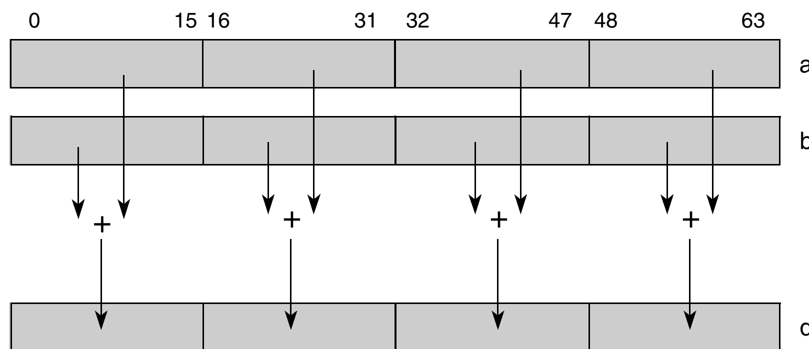


Figure 3-28. Vector Add Half Words (__ev_addh)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evaddh d,a,b**

__ev_addhhisw __ev_addhhisw

Vector Add Half Words High Signed to Words

d = __ev_addhhisw (a,b)

```
d0:31 ←EXTS32(a0:15) + EXTS32(b0:15)// Modulo

d32:63 ←EXTS32(a16:31) + EXTS32(b16:31) // Modulo
```

The high halfword elements of parameter **a** are sign-extended to 32 bits and added to the respective sign-extended high halfword elements of parameter **b** and the 32-bit results are placed into parameter **d**. The sum is a modulo sum.

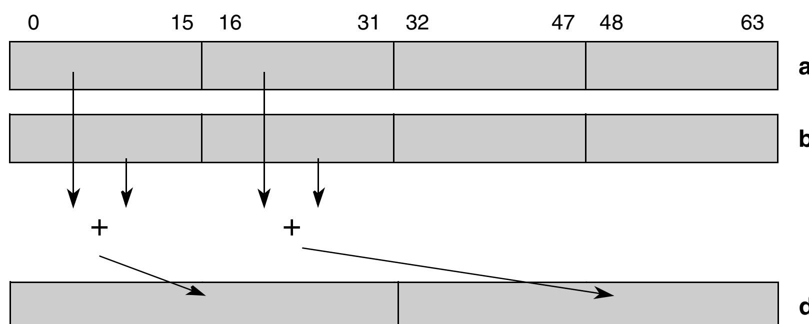


Figure 3-29. Vector Add Halfwords High Signed to Words (__ev_addhhisw)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evaddhhisw d,a,b**

__ev_addhhiuw __ev_addhhiuw

Vector Add Half Words High Unsigned to Words

d = __ev_addhhiuw (a,b)

$d_{0:31} \leftarrow EXTZ_{32}(a_{0:15}) + EXTZ_{32}(b_{0:15})$ // Modulo

$d_{32:63} \leftarrow EXTZ_{32}(a_{16:31}) + EXTZ_{32}(b_{16:31})$ // Modulo

The high halfword elements of parameter **a** are zero-extended to 32 bits and added to the respective zero-extended high halfword elements of parameter **b** and the 32-bit results are placed into parameter **d**. The sum is a modulo sum.

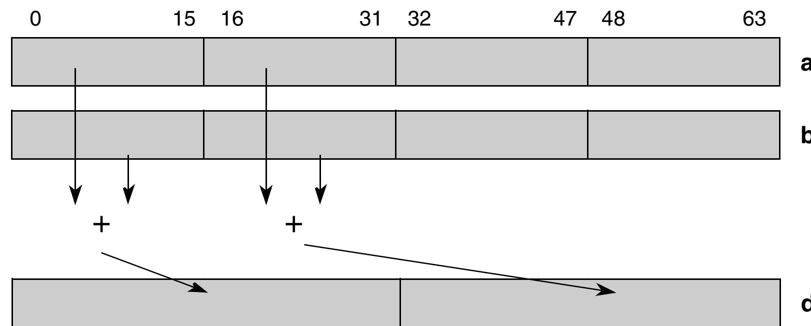


Figure 3-30. Vector Add Halfwords High Unsigned to Words (__ev_addhhiuw)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evaddhhiuw d,a,b**

__ev_addhlosw __ev_addhlosw

Vector Add Half Words Low Signed to Words

d = __ev_addhlosw (a,b)

```
d0:31 ←EXTS32(a32:47) + EXTS32(b32:47)// Modulo

d32:63 ←EXTS32(a48:63) + EXTS32(b48:63) // Modulo
```

The low halfword elements of parameter **a** are sign-extended to 32 bits and added to the respective sign-extended low halfword elements of parameter **b** and the 32-bit results are placed into parameter **d**. The sum is a modulo sum.

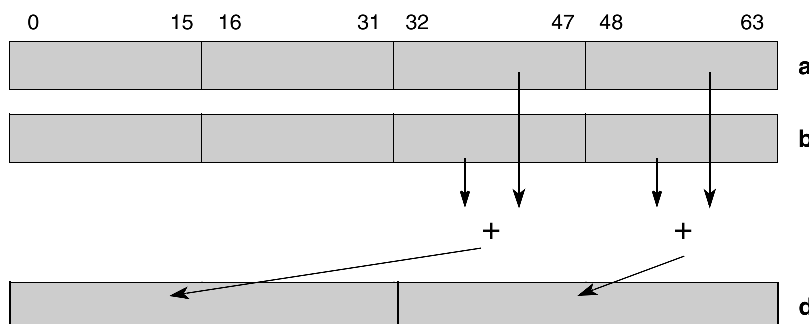


Figure 3-31. Vector Add Halfwords Low Signed to Words (__ev_addhlosw)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evaddhlosw d,a,b**

__ev_addhlouw __ev_addhlouw

Vector Add Half Words Low Unsigned to Words

d = __ev_addhlouw (a,b)

```
d0:31 ←EXTZ32(a32:47) + EXTZ32(b32:47)// Modulo

d32:63 ←EXTZ32(a48:63) + EXTZ32(b48:63) // Modulo
```

The low halfword elements of parameter **a** are zero-extended to 32 bits and added to the respective zero-extended low halfword elements of parameter **b** and the 32-bit results are placed into parameter **d**. The sum is a modulo sum.

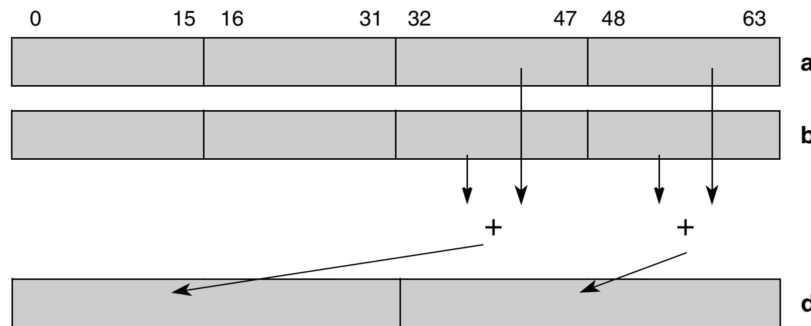


Figure 3-32. Vector Add Halfwords Low Unsigned to Words (__ev_addhlouw)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evaddhlouw d,a,b**

__ev_addhss __ev_addhss

Vector Add Half Words Signed and Saturate

d = __ev_addhss (a,b)

```
// h0
temp0:31 ←EXTS(a0:15) + EXTS(b0:15); ovh0 ←temp15 ⊕ temp16
d0:15 ←SATURATE(ovh0,temp15,0x8000,0x7fff,temp16:31)

// h1
temp0:31 ←EXTS(a16:31) + EXTS(b16:31); ovh1 ←temp15 ⊕ temp16
d16:31 ←SATURATE(ovh1, temp15, 0x8000, 0x7fff, temp16:31)

// h2
temp0:31 ←EXTS(a32:47) + EXTS(b32:47); ovh2 ←temp15 ⊕ temp16
d32:47 ←SATURATE(ovh2, temp15, 0x8000, 0x7fff, temp16:31)

// h3
temp0:31 ←EXTS(a48:63) + EXTS(b48:63); ovh3 ←temp15 ⊕ temp16
d48:63 ←SATURATE(ovh3, temp15, 0x8000, 0x7fff, temp16:31)

ovh ←ovh0 | ovh1
ovl ←ovh2 | ovh3
SPEFSCR_OVH ←ovh
SPEFSCR_OV ←ovl
SPEFSCR_SOVH ←SPEFSCR_SOVH | ovh
SPEFSCR_SOV ←SPEFSCR_SOV | ovl
```

The four signed half word elements of parameter **a** are added to the corresponding four signed half word elements of parameter **b**, saturating if overflow or underflow occurs, and the results are placed in parameter **d**. Any overflow or underflow is recorded in the SPEFSCR overflow and summary overflow bits.

Other registers altered: SPEFSCR

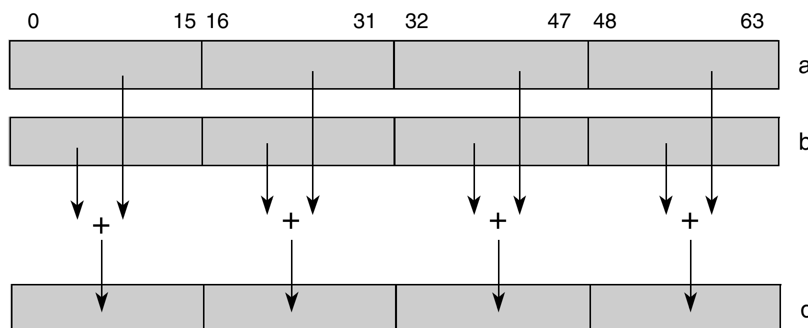


Figure 3-33. Vector Add Half Words Signed and Saturate (__ev_addhss)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evaddhss d,a,b**

__ev_addhus __ev_addhus

Vector Add Half Words Unsigned and Saturate

d = __ev_addhus (a,b)

```
// h0
temp0:31 ←EXTZ(a0:15) + EXTZ(b0:15); ovh0 ←temp15
d0:15 ←SATURATE(ovh0, temp15, 0xffff, 0xffff, temp16:31)

// h1
temp0:31 ←EXTZ(a16:31) + EXTZ(b16:31); ovh1 ←temp15
d16:31 ←SATURATE(ovh1, temp15, 0xffff, 0xffff, temp16:31)

// h2
temp0:31 ←EXTZ(a32:47) + EXTZ(b32:47); ovh2 ←temp15
d32:47 ←SATURATE(ovh2, temp15, 0xffff, 0xffff, temp16:31)

// h3
temp0:31 ←EXTZ(a48:63) + EXTZ(b48:63); ovh3 ←temp15
d48:63 ←SATURATE(ovh3, temp15, 0xffff, 0xffff, temp16:31)

ovh ←ovh0 | ovh1
ovl ←ovh2 | ovh3
SPEFSCROVH ←ovh
SPEFSCROV ←ovl
SPEFSCRSOVH ←SPEFSCRSOVH | ovh
SPEFSCRSOV ←SPEFSCRSOV | ovl
```

The four unsigned half word elements of parameter **a** are added to four unsigned half word elements of parameter **b**, saturating if overflow occurs, and the results are placed in parameter **d**. Any overflow is recorded in the SPEFSCR overflow and summary overflow bits.

Other registers altered: SPEFSCR

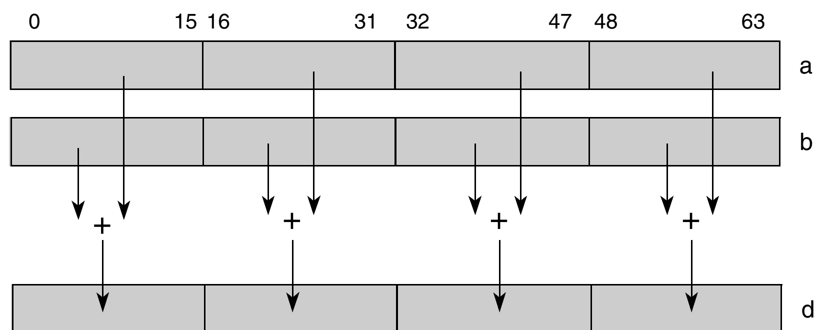


Figure 3-34. Vector Add Half Words Unsigned and Saturate (__ev_addhus)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evaddhus d,a,b**

__ev_addhx

__ev_addhx

Vector Add Half Words Exchanged

d = __ev_addhx (a,b)

```
// h0
d0:15 ←a0:15 + b16:31 // modulo sum
// h1
d16:31 ←a16:31 + b0:15 // modulo sum
// h2
d32:47 ←a32:47 + b48:63 // modulo sum
// h3
d48:63 ←a48:63 + b32:47 // modulo sum
```

The four half word elements of parameter **a** are added to four exchanged half word elements of parameter **b** and the results are placed in parameter **d**. The sum is a modulo sum.

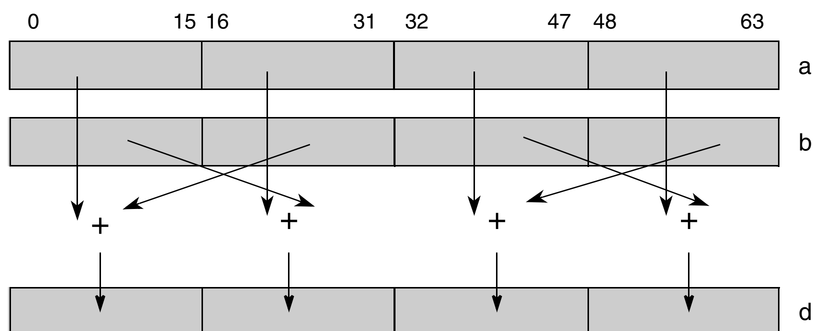


Figure 3-35. Vector Add Half Words Exchanged (__ev_addhx)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evaddhx d,a,b**

__ev_addhxss __ev_addhxss

Vector Add Half Words Exchanged, Signed and Saturate

d = __ev_addhxss (a,b)

```
// h1
temp0:31 ←EXTS(a16:31) + EXTS(b0:15)
ovh1 ←temp15 ⊕ temp16
d16:31 ←SATURATE(ovh1, temp15, 0x8000, 0x7fff, temp16:31)

// h2
temp0:31 ←EXTS(a32:47) + EXTS(b48:63)
ovh2 ←temp15 ⊕ temp16
d32:47 ←SATURATE(ovh2, temp15, 0x8000, 0x7fff, temp16:31)

// h3
temp0:31 ←EXTS(a48:63) + EXTS(b32:47)
ovh3 ←temp15 ⊕ temp16
d48:63 ←SATURATE(ovh3, temp15, 0x8000, 0x7fff, temp16:31)

ovh ←ovh0 | ovh1
ovl ←ovh2 | ovh3
SPEFSCROVH ←ovh
SPEFSCROV ←ovl
SPEFSCRSOVH ←SPEFSCRSOVH | ovh
SPEFSCRSOV ←SPEFSCRSOV | ovl
```

The four signed half word elements of parameter **a** are added to four exchanged signed half word elements of parameter **b**, saturating if overflow or underflow occurs, and the results are placed in parameter **d**. Any overflow or underflow is recorded in the SPEFSCR overflow and summary overflow bits.

Other registers altered: SPEFSCR

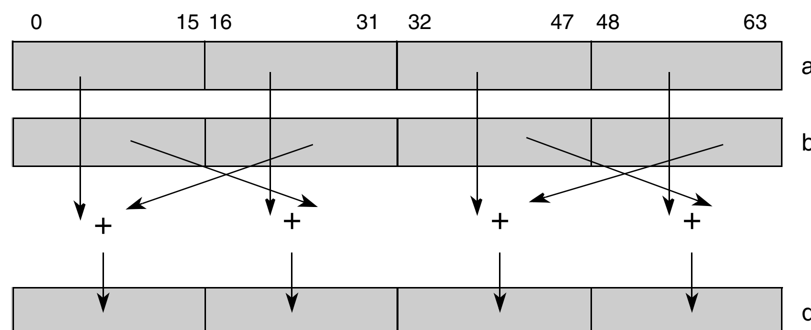


Figure 3-36. Vector Add Half Words Exchanged, Signed and Saturate (__ev_addhxss)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evaddhxss d,a,b**

__ev_addhxus __ev_addhxus

Vector Add Half Words Exchanged, Unsigned and Saturate

d = __ev_addhxus (a,b)

```
// h0
temp0:31 ←EXTZ(a0:15) + EXTZ(b16:31); ovh0 ←temp15
d0:15 ←SATURATE(ovh0, temp15, 0xffff, 0xffff, temp16:31)

// h1
temp0:31 ←EXTZ(a16:31) + EXTZ(b0:15); ovh1 ←temp15
d16:31 ←SATURATE(ovh1, temp15, 0xffff, 0xffff, temp16:31)

// h2
temp0:31 ←EXTZ(a32:47) + EXTZ(b48:63); ovh2 ←temp15
d32:47 ←SATURATE(ovh2, temp15, 0xffff, 0xffff, temp16:31)

// h3
temp0:31 ←EXTZ(a48:63) + EXTZ(b32:47); ovh3 ←temp15
d48:63 ←SATURATE(ovh3, temp15, 0xffff, 0xffff, temp16:31)

ovh ←ovh0 | ovh1
ovl ←ovh2 | ovh3
SPEFSCROVH ←ovh
SPEFSCROV ←ovl
SPEFSCRSOVH ←SPEFSCRSOVH | ovh
SPEFSCRSOV ←SPEFSCRSOV | ovl
```

The four unsigned half word elements of parameter **a** are added to four exchanged unsigned half word elements of parameter **b**, saturating if overflow occurs, and the results are placed in parameter **d**. Any overflow is recorded in the SPEFSCR overflow and summary overflow bits.

Other registers altered: SPEFSCR

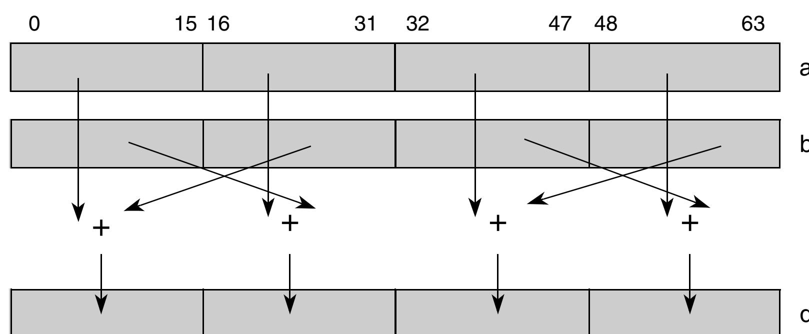


Figure 3-37. Vector Add Half Words Exchanged, Unsigned and Saturate (__ev_addhxus)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evaddhxus d,a,b**

__ev_addib

__ev_addib

Vector Add Immediate Byte

d = __ev_addib (a,b)

$d_{0:7} \leftarrow a_{0:7}$ + EXTZ(b) // Modulo sum
$d_{8:15} \leftarrow a_{8:15}$ + EXTZ(b) // Modulo sum
$d_{16:23} \leftarrow a_{16:23}$ + EXTZ(b) // Modulo sum
$d_{24:31} \leftarrow a_{24:31}$ + EXTZ(b) // Modulo sum
$d_{32:39} \leftarrow a_{32:39}$ + EXTZ(b) // Modulo sum
$d_{40:47} \leftarrow a_{40:47}$ + EXTZ(b) // Modulo sum
$d_{48:55} \leftarrow a_{48:55}$ + EXTZ(b) // Modulo sum
$d_{56:63} \leftarrow a_{56:63}$ + EXTZ(b) // Modulo sum

Parameter **b** is zero-extended and added to the eight byte elements of parameter **a** and the results are placed in parameter **d**.

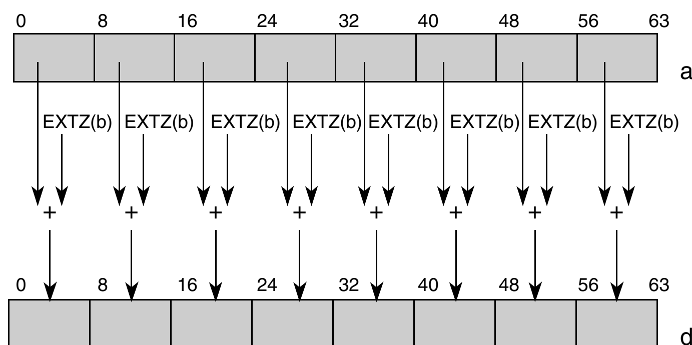


Figure 3-38. Vector Add Immediate Byte (__ev_addib)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	5-bit unsigned literal	**evaddib d,a,b**

__ev_addih __ev_addih

Vector Add Immediate Half Word

d = __ev_addih (a,b)

```
d0:15 ←a0:15 + EXTZ(b)// Modulo sum
d16:31 ←a16:31 + EXTZ(b)// Modulo sum
d32:47 ←a32:47 + EXTZ(b)// Modulo sum
d48:63 ←a48:63 + EXTZ(b)// Modulo sum
```

Parameter **b** is zero-extended and added to the four half word elements of parameter **a** and the results are placed in parameter **d**.

NOTE

The same value is added to all elements of parameter **a**.

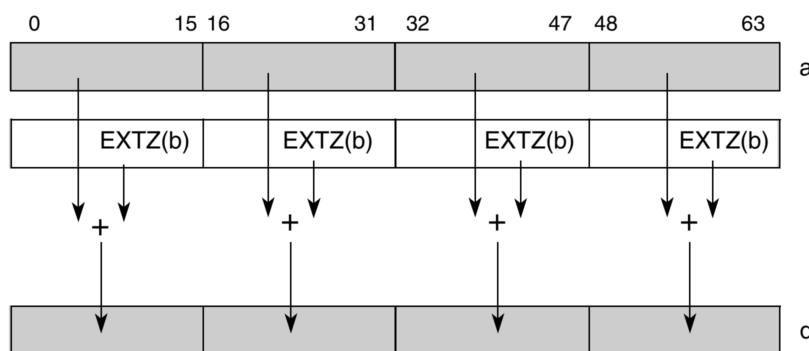


Figure 3-39. Vector Add Immediate Half Word (__ev_addih)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	5-bit unsigned literal	**evaddih d,a,b**

__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 parameter **a**.

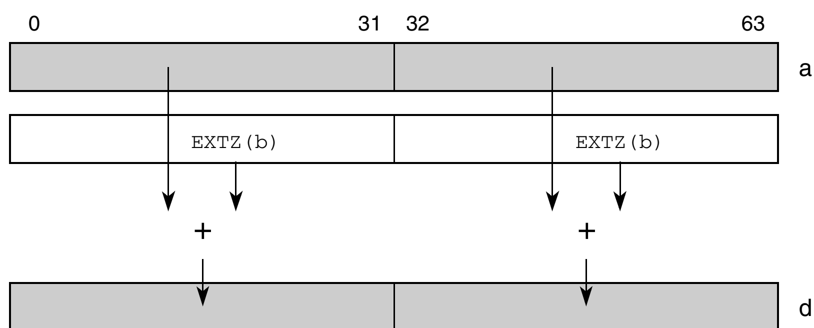


Figure 3-40. Vector Add Immediate Word (__ev_addiw)

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)

$d_{0:31} \leftarrow ACC_{0:31} + a_{0:31}$ // low
$d_{32:63} \leftarrow ACC_{32:63} + a_{32:63}$ // high
// update accumulator
$ACC_{0:63} \leftarrow d_{0:63}$

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 3-41. Vector Add Signed, Modulo, Integer to Accumulator Word (__ev_addsmiaaw)

d	a	Maps to
__ev64_opaque__	__ev64_opaque__	**evaddsmiaaw d,a**

__ev_addssiaa __ev_addssiaa

Vector Add Signed, Saturate, Integer to Accumulator

d = __ev_addssiaa (a)

```
//
temp0:64 ←EXTS(ACC0:63) + EXTS(a0:63)
ov ←temp0 ⊕ temp1
d0:31 ←SATURATE(ov, temp0, 0x8000_0000_0000_0000, 0x7fff_ffff_ffff_ffff, temp1:64)

ACC0:63 ←d0:6
SPEFSCROVH ←0
SPEFSCROV ←ov
SPEFSCRSOV ←SPEFSCRSOV | ov
```

The signed 64-bit value in parameter **a** is added to the accumulator saturating if overflow or underflow occurs, and the results are placed into parameter **d** and the accumulator. Any overflow or underflow is recorded in the SPEFSCR overflow and summary overflow bits.

Other registers altered: SPEFSCR ACC

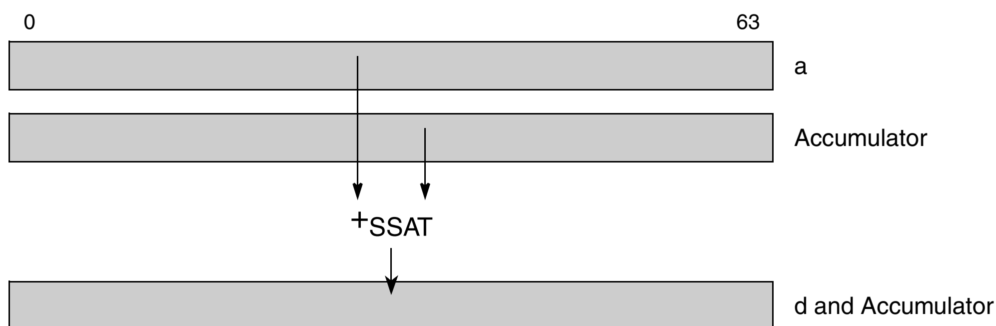


Figure 3-42. Vector Add Signed, Saturate, Integer to Accumulator (__ev_addssiaa)

d	a	Maps to
__ev64_opaque__	__ev64_opaque__	**evaddssiaa 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

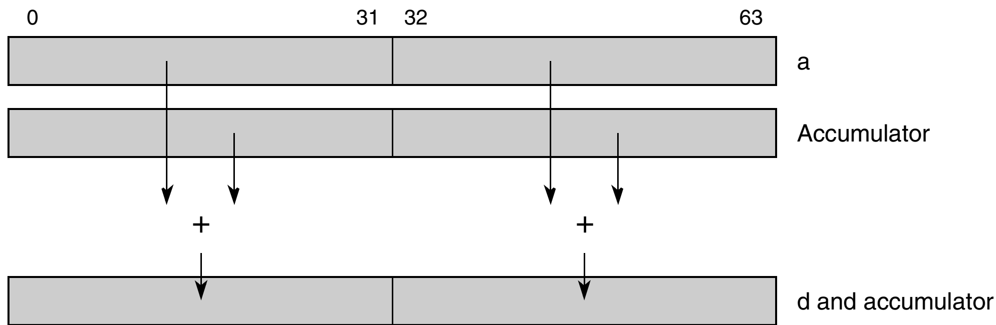


Figure 3-43. Vector Add Signed, Saturate, Integer to Accumulator Word (__ev_addssiaaw)

d	a	Maps to
__ev64_opaque__	__ev64_opaque__	**evaddssiaaw d,a**

__ev_addsubfh

Vector Add / Subtract from Half Word

d = __ev_addsubfh (a,b)

```
d0:15 ←b0:15 + a0:15 // Modulo sum
d16:31 ←b16:31 - a16:31 // Modulo difference, b - a
d32:47 ←b32:47 + a32:47 // Modulo sum
d48:63 ←b48:63 - a48:63 // Modulo difference, b - a
```

The even half word elements of parameter **a** are added to the even half word elements of parameter **b**, the odd half word elements of parameter **a** are subtracted from the odd half word elements of parameter **b**, and the results are placed in parameter **d**. The sum and difference are modulo.

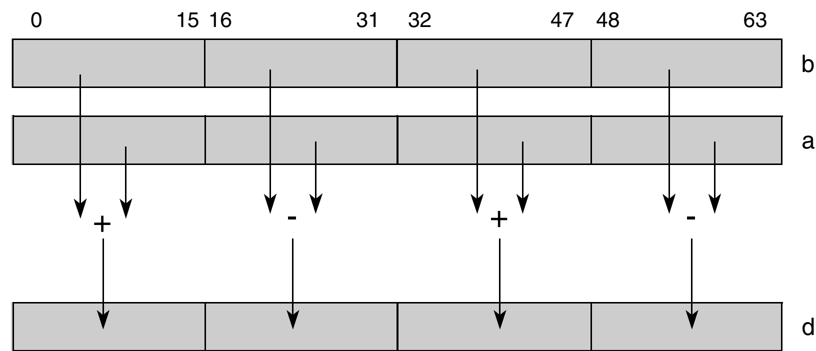


Figure 3-44. Vector Add / Subtract from Half Word (__ev_addsubfh)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evaddsubfh d,a,b**

__ev_addsubfhss __ev_addsubfhss

Vector Add / Subtract from Half Word Signed and Saturate

d = __ev_addsubfhss (a,b)

```
// h0
temp0:31 ←EXTS(b0:15) + EXTS(a0:15); ovh0 ←temp15 ⊕ temp16
d0:15 ←SATURATE(ovh0, temp15, 0x8000, 0x7fff, temp16:31)

// h1
temp0:31 ←EXTS(b16:31) - EXTS(a16:31); ovh1 ←temp15 ⊕ temp16
d16:31 ←SATURATE(ovh1, temp15, 0x8000, 0x7fff, temp16:31)

// h2
temp0:31 ←EXTS(b32:47) + EXTS(a32:47); ovh2 ←temp15 ⊕ temp16
d32:47 ←SATURATE(ovh2, temp15, 0x8000, 0x7fff, temp16:31)

// h3
temp0:31 ←EXTS(b48:63) - EXTS(a48:63); ovh3 ←temp15 ⊕ temp16
d48:63 ←SATURATE(ovh3, temp15, 0x8000, 0x7fff, temp16:31)

ovh ←ovh0 | ovh1
ovl ←ovh2 | ovh3
SPEFSCROVH ←ovh
SPEFSCROV ←ovl
SPEFSCRSOVH ←SPEFSCRSOVH | ovh
SPEFSCRSOV ←SPEFSCRSOV | ovl
```

The even signed half word elements of parameter **a** are added to the even signed half word elements of parameter **b**, the odd signed half word elements of parameter **a** are subtracted from the odd signed half word elements of parameter **b**, saturating if overflow or underflow occurs, and the results are placed in parameter **d**. Any overflow or underflow is recorded in the SPEFSCR overflow and summary overflow bits.

Other registers altered: SPEFSCR

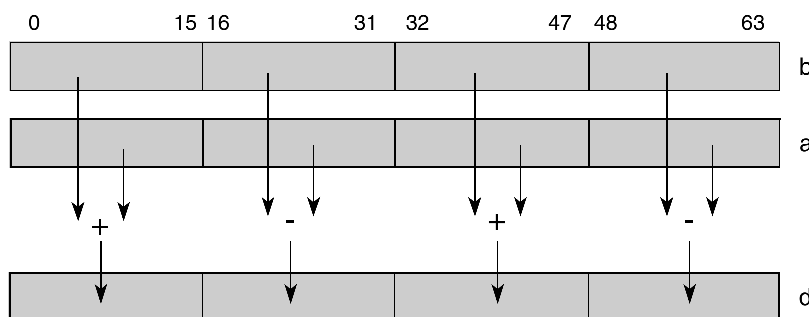


Figure 3-45. Vector Add / Subtract from Half Word Signed and Saturate (__ev_addsubfhss)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evaddsubfhss d,a,b**

__ev_addsubfhx __ev_addsubfhx

Vector Add / Subtract from Half Word Exchanged

d = __ev_addsubfhx (a,b)

```
// h0
d0:15 ←b0:15 + a16:31 // modulo
// h1
d16:31 ←b16:31 - a0:15 // modulo
// h2
d32:47 ←b32:47 + a48:63 // modulo
// h3
d48:63 ←b48:63 - a32:47 // modulo
```

The odd exchanged half word elements of parameter **a** are added to the even half word elements of parameter **b**, the even exchanged half words of parameter **a** are subtracted from the odd half words of parameter **b**, and the results are placed in parameter **d**. The sum and differences are modulo.

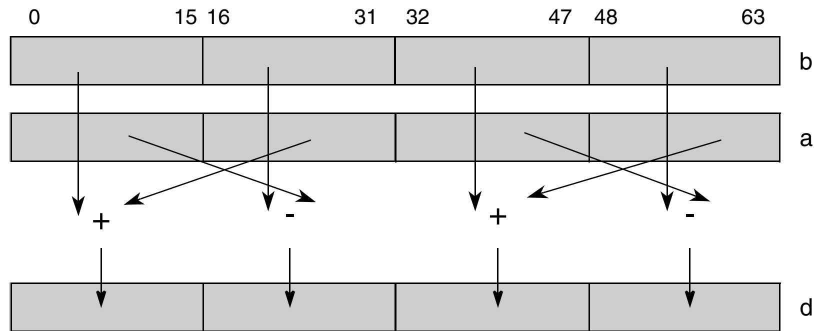


Figure 3-46. Vector Add / Subtract from Half Word Exchanged (__ev_addsubfhx)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evaddsubfhx d,a,b**

__ev_addsubfhxss __ev_addsubfhxss

Vector Add / Subtract from Half Word Exchanged, Signed and Saturate

d = __ev_addsubfhxss (a,b)

```
// h0
temp0:31 ←EXTS(b0:15) + EXTS(a16:31); ovh0 ←temp15 ⊕ temp16
d0:15 ←SATURATE(ovh0, temp15, 0x8000, 0x7fff, temp16:31)
// h1
temp0:31 ←EXTS(b16:31) - EXTS(a0:15); ovh1 ←temp15 ⊕ temp16
d16:31 ←SATURATE(ovh1, temp15, 0x8000, 0x7fff, temp16:31)
// h2
temp0:31 ←EXTS(b32:47) + EXTS(a48:63); ovh2 ←temp15 ⊕ temp16
d32:47 ←SATURATE(ovh2, temp15, 0x8000, 0x7fff, temp16:31)
// h3
temp0:31 ←EXTS(b48:63) - EXTS(a32:47); ovh3 ←temp15 ⊕ temp16
d48:63 ←SATURATE(ovh3, temp15, 0x8000, 0x7fff, temp16:31)

ovh ←ovh0 | ovh1
ovl ←ovh2 | ovh3
SPEFSCROVH ←ovh
SPEFSCROV ←ovl
SPEFSCRSOVH ←SPEFSCRSOVH | ovh
SPEFSCRSOV ←SPEFSCRSOV | ovl
```

The odd exchanged signed half word elements of parameter **a** are added to the even signed half word elements of parameter **b**, the even exchanged signed half words of parameter **a** are subtracted from the odd signed half words of parameter **b**, saturating if overflow or underflow occurs, and the results are placed in parameter **d**. Any overflow or underflow is recorded in the SPEFSCR overflow and summary overflow bits.

Other registers altered: SPEFSCR

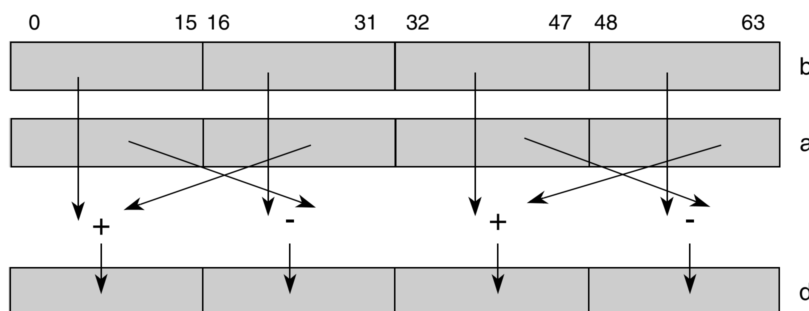


Figure 3-47. Vector Add / Subtract from Half Word Exchanged, Signed and Saturate (__ev_addsubfhxss)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evaddsubfhxss d,a,b**

__ev_addsubfw __ev_addsubfw

Vector Add / Subtract from Word

d = __ev_addsubfw (a,b)

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

The high word element of parameter **a** is added to the high word element of parameter **b**, the low word element of parameter **a** is subtracted from the low word element of parameter **b**, and the results are placed in parameter **d**.

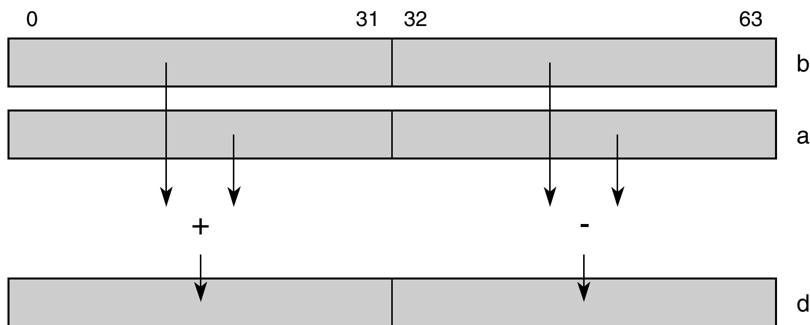


Figure 3-48. Vector Add / Subtract from Word (__ev_addsubfw)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evaddsubfw d,a,b**

__ev_addsubfwss __ev_addsubfwss

Vector Add / Subtract from Word Signed and Saturate

d = __ev_addsubfwss (a,b)

```
// h0
temp0:32 ←EXTS(b0:31) + EXTS(a0:31)
ovh ←temp0 ⊕ temp1
d0:15 ←SATURATE(ovh, temp0, 0x8000_0000, 0x7fff_ffff, temp1:32)

// h1
temp0:31 ←EXTS(b32:63) - EXTS(a32:63)
ovl ←temp0 ⊕ temp1
d16:31 ←SATURATE(ovl, temp0, 0x8000_0000, 0x7fff_ffff, temp1:32)

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

The high word element of parameter **a** is added to the high word element of parameter **b**, saturating if overflow or underflow occurs, the low word element of parameter **a** is subtracted from the low word element of parameter **b**, saturating if overflow or underflow occurs, and the results are placed in parameter **d**. Any overflow or underflow is recorded in the SPEFSCR overflow and summary overflow bits.

Other registers altered: SPEFSCR

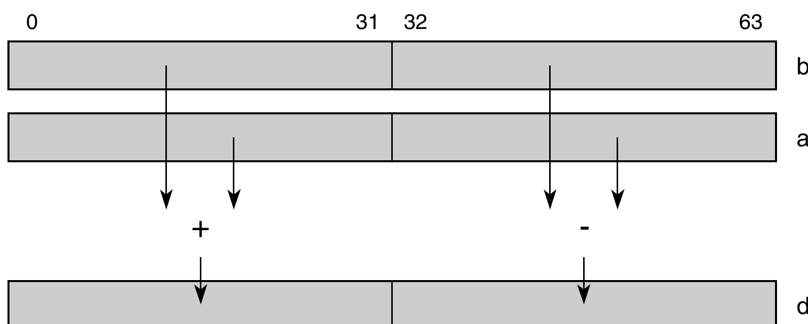


Figure 3-49. Vector Add / Subtract from Word Signed and Saturate (__ev_addsubfwss)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evaddsubfwss d,a,b**

__ev_addsubfwx

__ev_addsubfwx

Vector Add / Subtract from Word Exchanged

d = __ev_addsubfwx (a,b)

$d_{0:31} \leftarrow b_{0:31} + a_{32:63}$ // Modulo
$d_{32:63} \leftarrow b_{32:63} - a_{0:31}$ // Modulo

The low word element of parameter **a** is added to the high word element of parameter **b**, the high word element of parameter **a** is subtracted from the low word element of parameter **b**, and the results are placed in parameter **d**.

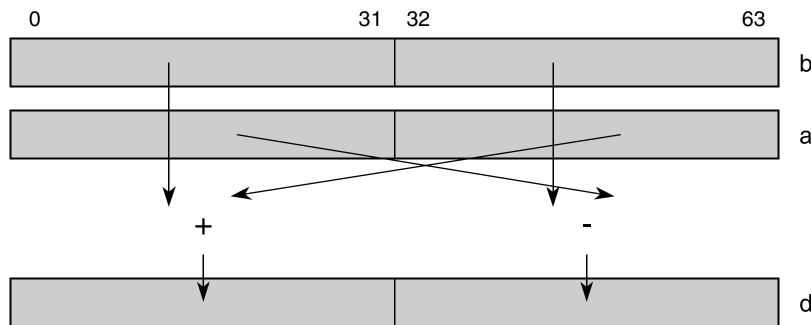


Figure 3-50. Vector Add / Subtract from Word Exchanged (__ev_addsubfwx)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evaddsubfwx d,a,b**

__ev_addsubfwxss __ev_addsubfwxss

Vector Add / Subtract from Word Exchanged Signed and Saturate

d = __ev_addsubfwxss (a,b)

```
// h0
temp0:32 ←EXTS(b0:31) + EXTS(a32:63)
ovh ←temp0 ⊕ temp11
d0:15 ←SATURATE(ovh, temp0, 0x8000_0000, 0x7fff_ffff, temp1:32)

// h1
temp0:31 ←EXTS(b32:63) - EXTS(a0:31)
ovl ←temp0 ⊕ temp1
rD16:31 ←SATURATE(ovl, temp0, 0x8000_0000, 0x7fff_ffff, temp1:32)

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

The low word element of parameter **a** is added to the high word element of parameter **b**, saturating if overflow or underflow occurs, the high word element of parameter **a** is subtracted from the low word element of parameter **b**, saturating if overflow or underflow occurs, and the results are placed in parameter **d**. Any overflow or underflow is recorded in the SPEFSCR overflow and summary overflow bits.

Other registers altered: SPEFSCR

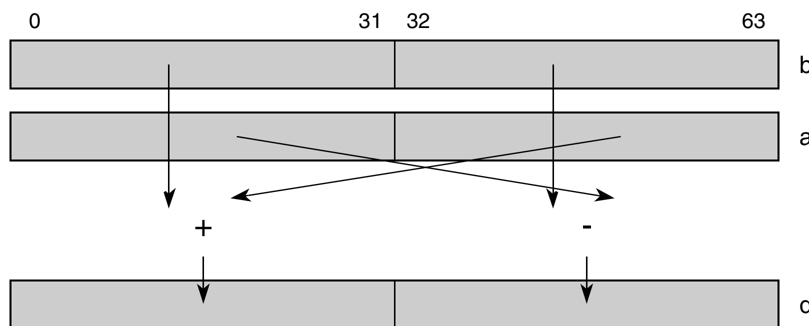


Figure 3-51. Vector Add / Subtract from Word Exchanged Signed and Saturate (__ev_addsubfwxss)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evaddsubfwxss d,a,b**

__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

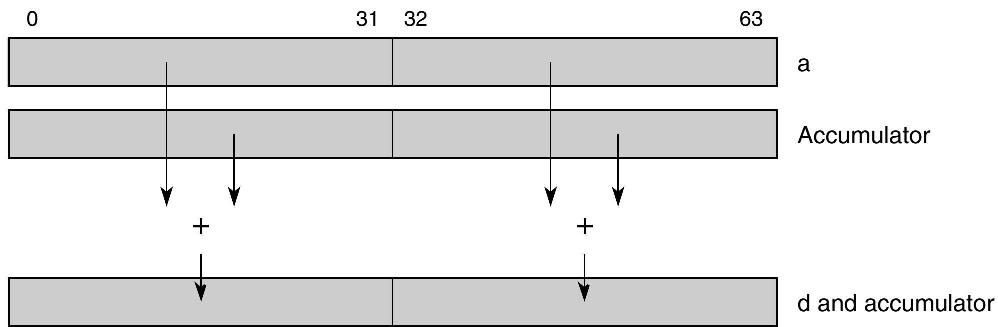


Figure 3-52. Vector Add Unsigned, Modulo, Integer to Accumulator Word (__ev_addumiaaw)

d	a	Maps to
__ev64_opaque__	__ev64_opaque__	**evaddumiaaw d,a**

__ev_addusiaa __ev_addusiaa

Vector Add Unsigned, Saturate, Integer to Accumulator

d = __ev_addusiaa (a)

```
temp0:64←EXTZ(ACC0:63) + EXTZ(a0:63)
ov ←temp0
d0:63 ←SATURATE(ov, temp0, 0xffff_ffff_ffff_ffff, 0xffff_ffff_ffff_ffff, temp1:64)

ACC0:63 ←d0:63

SPEFSCROVH ←0
SPEFSCROV ←ov
SPEFSCRSOV ←SPEFSCRSOV | ov
```

The unsigned 64-bit value in parameter **a** is added to the unsigned value in the accumulator saturating if overflow occurs, and the results are placed into parameter **d** and the accumulator. Any overflow is recorded in the SPEFSCR overflow and summary overflow bits.

Other registers altered: SPEFSCR ACC

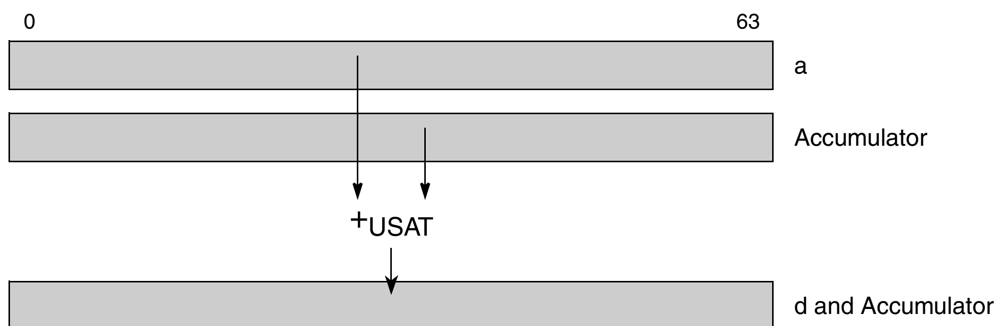


Figure 3-53. Vector Add Unsigned, Saturate, Integer to Accumulator (__ev_addusiaa)

d	a	Maps to
__ev64_opaque__	__ev64_opaque__	**evaddusiaa 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

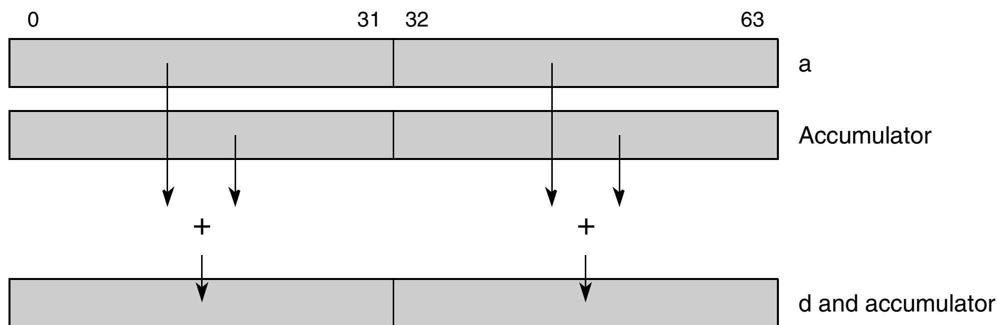


Figure 3-54. Vector Add Unsigned, Saturate, Integer to Accumulator Word (__ev_addusiaaw)

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.

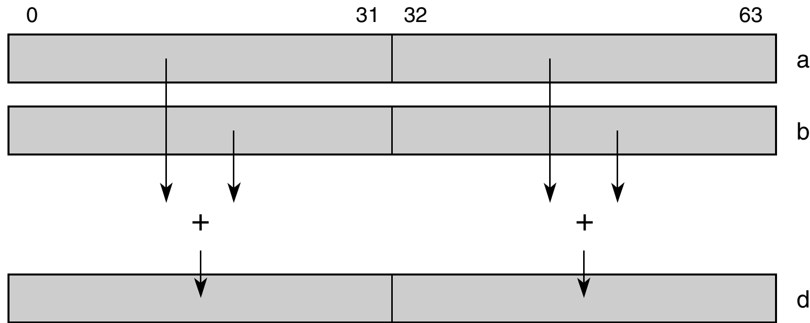


Figure 3-55. Vector Add Word (__ev_addw)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evaddw d,a,b**

__ev_addwegsf __ev_addwegsf

Vector Add Word Even Guarded Signed Fraction

d = __ev_addwegsf (a,b)

$d_{0:63} \leftarrow (EXTS_{48}(b_{0:31}) \,||\, {}^{16}0) + (EXTS_{48}(a_{0:31}) \,||\, {}^{16}0)$

The even word elements of parameter **a** and parameter **b** are sign-extended with 16 guard bits and padded with 16 0's, and then added together to produce a 64-bit sum, and the result is placed into parameter **d**.

Note: __ev_addwegsf is used to add 1.31 fractions to produce a 17.47 fractional sum.

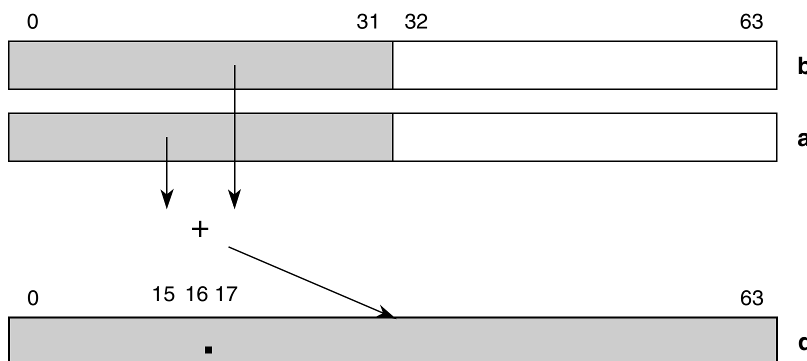


Figure 3-56. Vector Add Word Even Guarded Signed Fraction (__ev_addwegsf)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evaddwegsf d,a,b**

__ev_addwegsi __ev_addwegsi

Vector Add Word Even Guarded Signed Integer

d = __ev_addwegsi (a,b)

$d_{0:63} \leftarrow EXTS_{64}(b_{0:31}) + EXTS_{64}(a_{0:31})$

The even word elements of parameter **a** and parameter **b** are sign-extended to 64 bits and added together to produce a 64-bit sum, and the result is placed into parameter **d**.

Note: __ev_addwegsi can also be used to add 1.31 fractions to produce a 33.31 fractional sum.

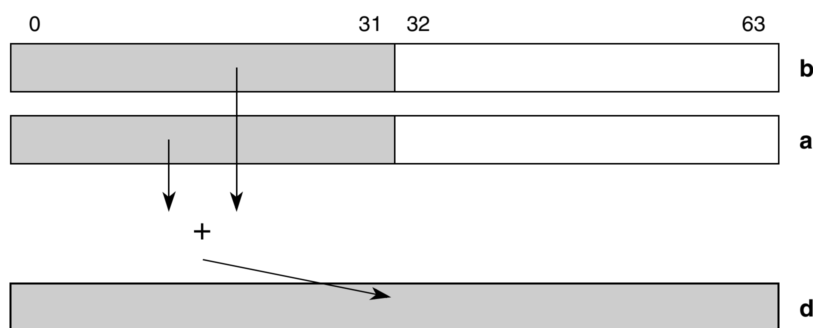


Figure 3-57. Vector Add Word (__ev_addwegsi)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evaddwegsi d,a,b**

__ev_addwogsf __ev_addwogsf

Vector Add Word Odd Guarded Signed Fraction

d = __ev_addwogsf (a,b)

$d_{0:63} \leftarrow (EXTS_{48}(b_{32:63}) \,||\, {}^{16}0) + (EXTS_{48}(a_{32:63}) \,||\, {}^{16}0)$

The odd word elements of parameter **a** and parameter **b** are sign-extended with 16 guard bits and padded with 16 0’s, and then added together to produce a 64-bit sum, and the result is placed into parameter **d**.

Note: __ev_addwogsf is used to add 1.31 fractions to produce a 17.47 fractional sum.

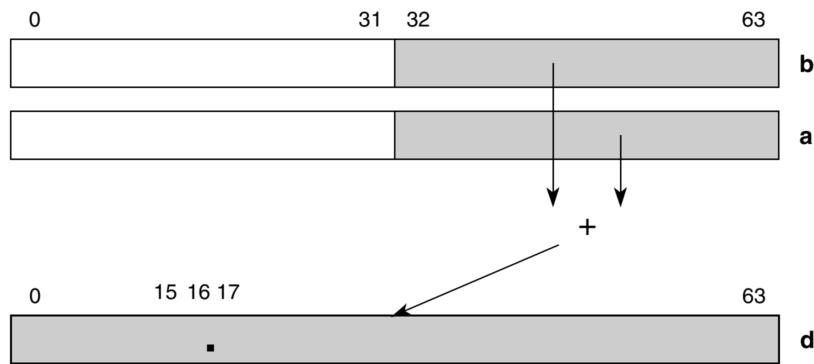


Figure 3-58. Vector Add Word Odd Guarded Signed Fraction (__ev_addwogsf)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evaddwogsf d,a,b**

__ev_addwogsi __ev_addwogsi

Vector Add Word Odd Guarded Signed Integer

d = __ev_addwogsi (a,b)

$d_{0:63} \leftarrow EXTS_{64}(b_{32:63}) + EXTS_{64}(a_{32:63})$

The odd word elements of parameter **a** and parameter **b** are sign-extended to 64 bits and added together to produce a 64-bit sum, and the result is placed into parameter **d**.

Note: __ev_addwogsi can also be used to add 1.31 fractions to produce a 33.31 fractional sum.

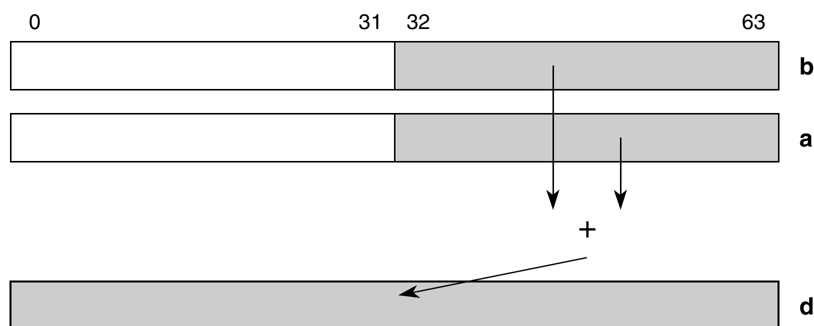


Figure 3-59. Vector Add Word Odd Guarded Signed Integer (__ev_addwogsi)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evaddwogsi d,a,b**

__ev_addwss __ev_addwss

Vector Add Word Signed and Saturate

d = __ev_addwss (a,b)

```
// h0
temp0:32 ←EXTS(b0:31) + EXTS(a0:31)
ovh ←temp0 ⊕ temp1
d0:15 ←SATURATE(ovh, temp0, 0x8000_0000, 0x7fff_ffff, temp1:32)

// h1
temp0:31 ←EXTS(b32:63) + EXTS(a32:63)
ovl ←temp0 ⊕ temp1
d16:31 ←SATURATE(ovl, temp0, 0x8000_0000, 0x7fff_ffff, temp1:32)

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

The high word element of parameter a is added to the high word element of parameter **b**, saturating if overflow or underflow occurs, the low word element of parameter **a** is added to the low word element of parameter **b**, saturating if overflow or underflow occurs, and the results are placed in parameter **d**. Any overflow or underflow is recorded in the SPEFSCR overflow and summary overflow bits.

Other registers altered: SPEFSCR

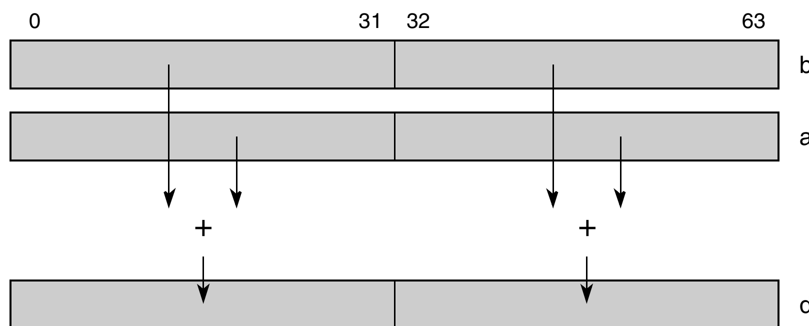


Figure 3-60. Vector Add Word Signed and Saturate (__ev_addwss)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evaddwss d,a,b**

__ev_addwus __ev_addwus

Vector Add Word Unsigned and Saturate

d = __ev_addwus (a,b)

```
// h0
temp0:32 ←EXTZ(b0:31) + EXTZ(a0:31)
ovh ←temp0
d0:15 ←SATURATE(ovh, temp0, 0xffff_ffff, 0xffff_ffff, temp1:32)

// h1
temp0:31 ←EXTZ(b32:63) + EXTZ(a32:63)
ovl ←temp0
d16:31 ←SATURATE(ovl, temp0, 0xffff_ffff, 0xffff_ffff, temp1:32)

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

The high word element of parameter **a** is added to the high word element of parameter **b**, saturating if overflow occurs, the low word element of parameter **a** is added to the low word element of parameter **b**, saturating if overflow occurs, and the results are placed in parameter **d**. Any overflow is recorded in the SPEFSCR overflow and summary overflow bits.

Other registers altered: SPEFSCR

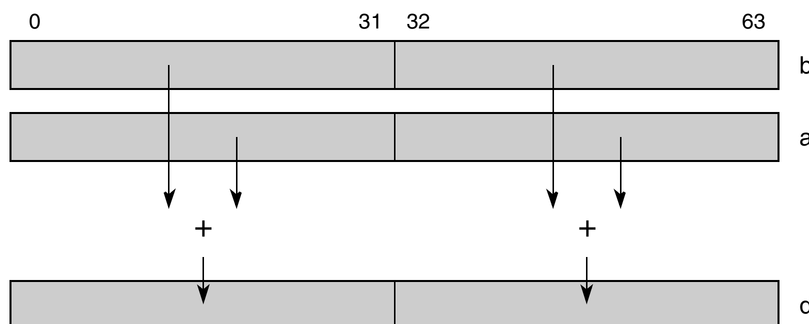


Figure 3-61. Vector Add Word Unsigned and Saturate (__ev_addwus)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evaddwus d,a,b**

__ev_addwx __ev_addwx

Vector Add Word Exchanged

d = __ev_addwx (a,b)

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

The exchanged word elements of parameter **a** are added to the elements of parameter **b** and the results are placed in parameter **d**. The sum is a modulo sum.

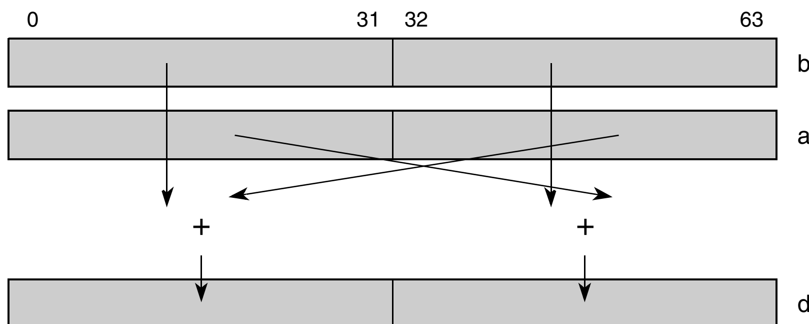


Figure 3-62. Vector Add Word Exchanged (__ev_addwx)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evaddwx d,a,b**

__ev_addwxss __ev_addwxss

Vector Add Word Exchanged Signed and Saturate

d = __ev_addwxss (a,b)

```
// h0
temp0:32 ←EXTS(b0:31) + EXTS(a32:63)
ovh ←temp0 ⊕ temp1
d0:15 ←SATURATE(ovh, temp0, 0x8000_0000, 0x7fff_ffff, temp1:32)

// h1
temp0:31 ←EXTS(b32:63) + EXTS(a0:31)
ovl ←temp0 ⊕ temp1
d16:31 ←SATURATE(ovl, temp0, 0x8000_0000, 0x7fff_ffff, temp1:32)

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

The low word element of parameter **a** is added to the high word element of parameter **b**, saturating if overflow or underflow occurs, the high word element of parameter **a** is added to the low word element of parameter **b**, saturating if overflow or underflow occurs, and the results are placed in parameter **d**. Any overflow or underflow is recorded in the SPEFSCR overflow and summary overflow bits.

Other registers altered: SPEFSCR

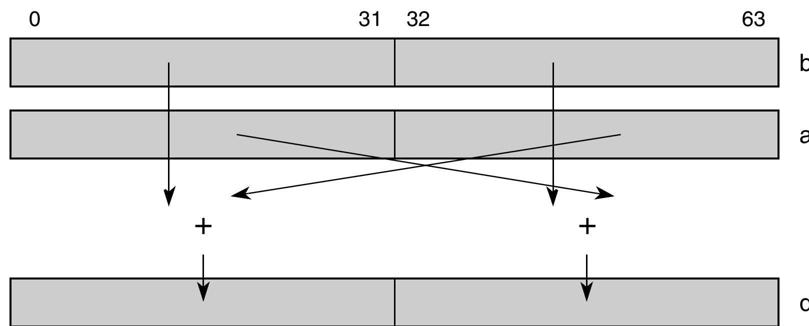


Figure 3-63. Vector Add Word Exchanged Signed and Saturate (__ev_addwxss)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evaddwxss d,a,b**

__ev_addwxus

__ev_addwxus

Vector Add Word Exchanged Unsigned and Saturate

d = __ev_addwxus (a,b)

```
// h0
temp0:32 ←EXTZ(b0:31) + EXTZ(a32:63)
ovh ←temp0
d0:15 ←SATURATE(ovh, temp0, 0xffff_ffff, 0xffff_ffff, temp1:32)

// h1
temp0:31 ←EXTZ(b32:63) + EXTZ(a0:31)
ovl ←temp0
d16:31 ←SATURATE(ovl, temp0, 0xffff_ffff, 0xffff_ffff, temp1:32)

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

The low word element of parameter **a** is added to the high word element of parameter **b**, saturating if overflow occurs, the high word element of parameter **a** is added to the low word element of parameter **b**, saturating if overflow occurs, and the results are placed in parameter **d**. Any overflow is recorded in the SPEFSCR overflow and summary overflow bits.

Other registers altered: SPEFSCR

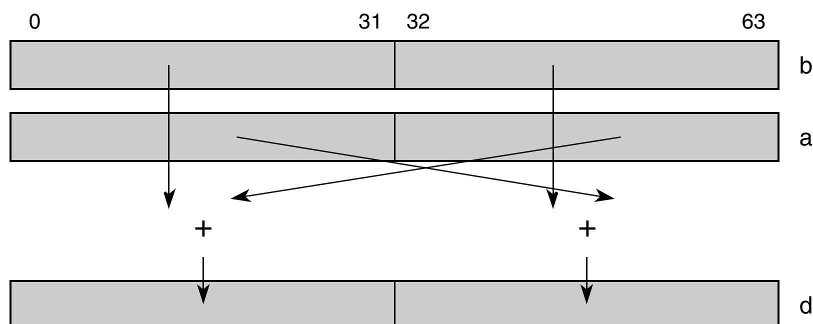


Figure 3-64. Vector Add Word Exchanged Unsigned and Saturate (__ev_addwxus)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evaddwxus 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**.

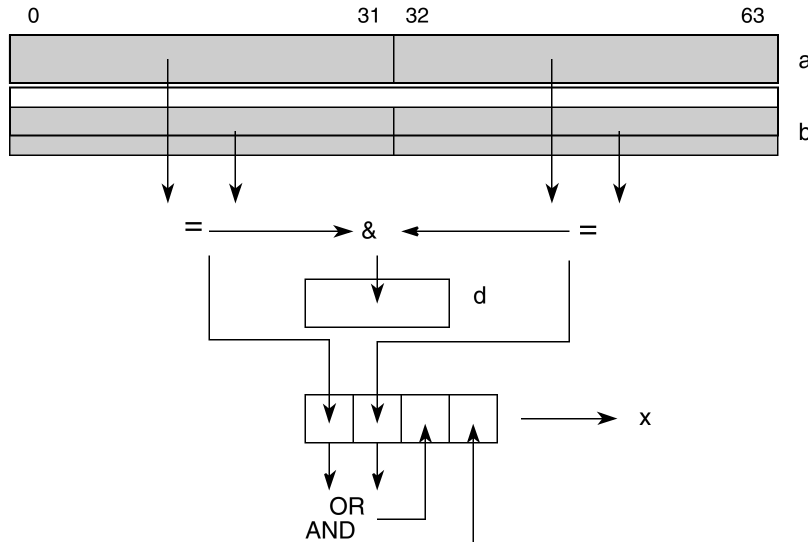


Figure 3-65. Vector All Equal (__ev_all_eq)

d	a	b	Maps to
_Bool	__ev64_opaque__	__ev64_opaque__	**evcmpeq 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**.

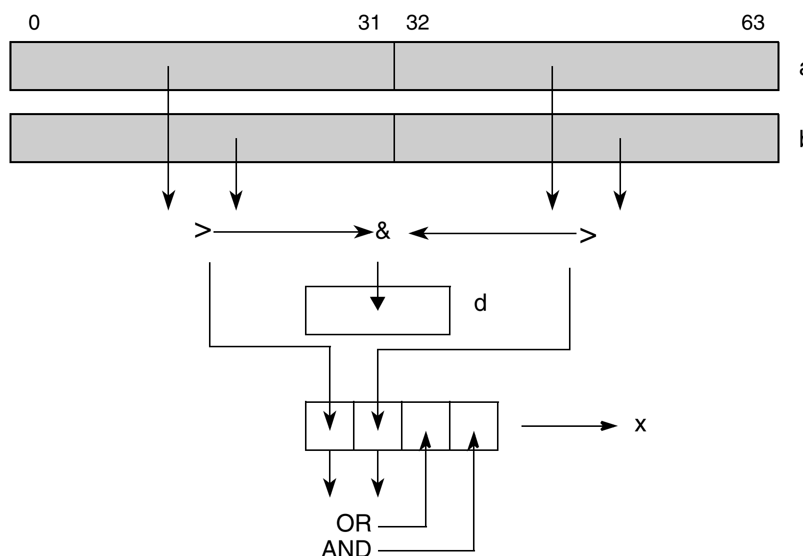


Figure 3-66. Vector All Greater Than Signed (__ev_all_gts)

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**.

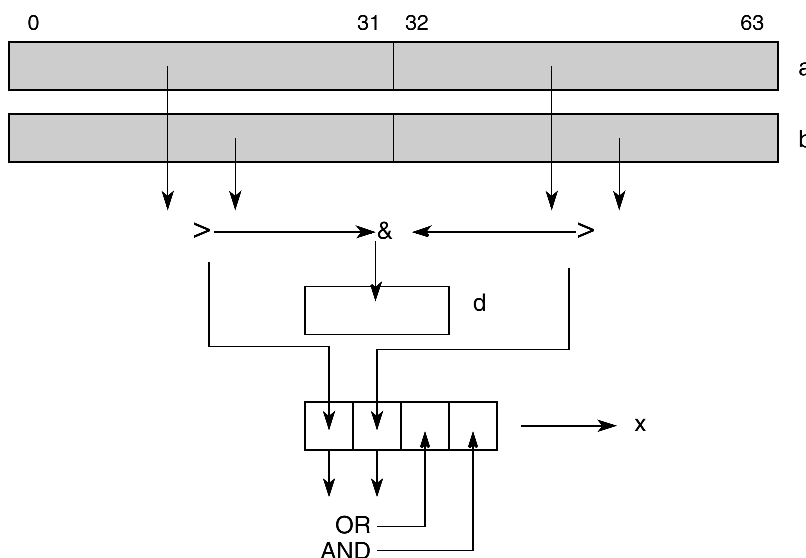


Figure 3-67. Vector All Greater Than Unsigned (__ev_all_gtu)

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**.

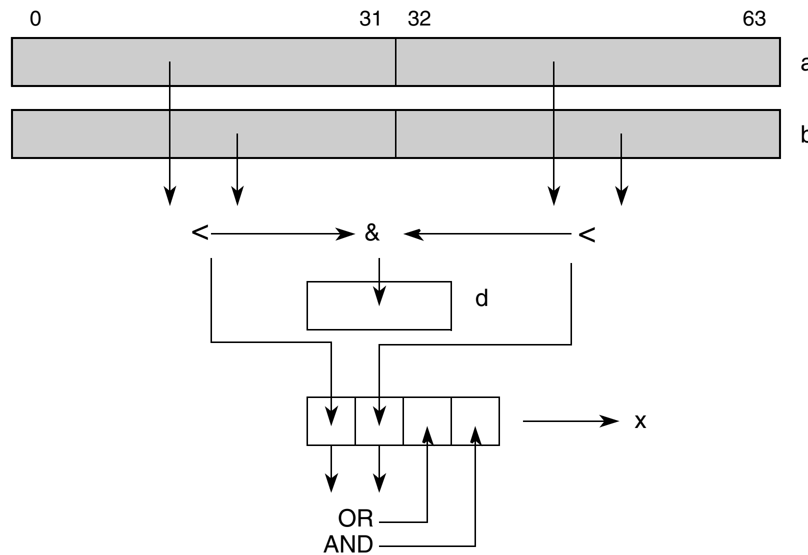


Figure 3-68. Vector All Less Than Signed (__ev_all_lts)

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**.

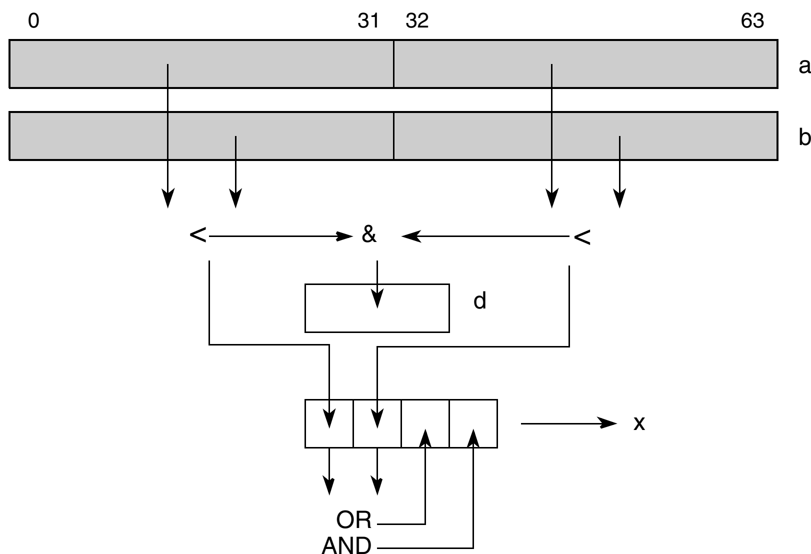


Figure 3-69. Vector All Less Than Unsigned (__ev_all_ltu)

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**.

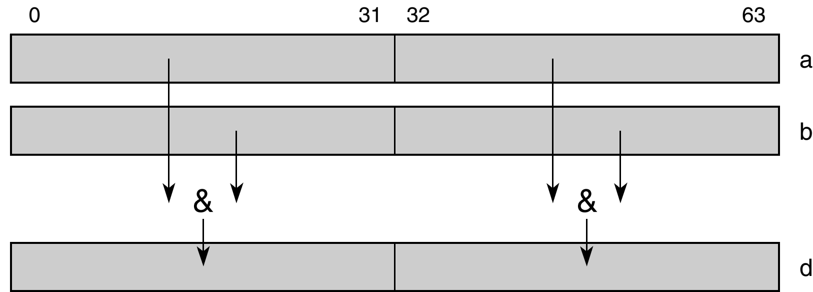


Figure 3-70. Vector AND (__ev_and)

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**.

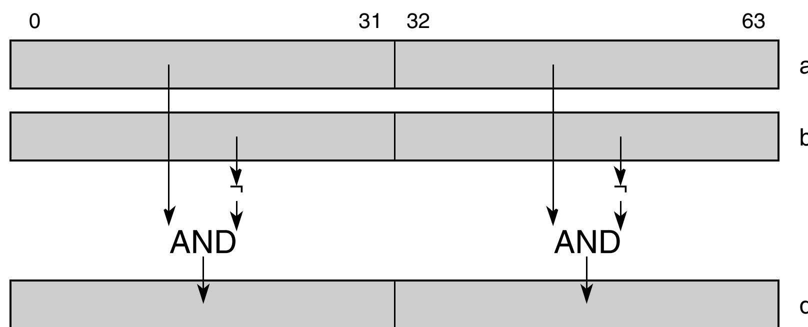


Figure 3-71. Vector AND with Complement (__ev_andc)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evandc d,a,b**

__ev_avgbs

__ev_avgbs

Vector Average Byte Signed

d = __ev_avgbs (a,b)

$temp_{0:8} \leftarrow EXTS(a_{0:7}) + EXTS(b_{0:7})$
$d_{0:7} \leftarrow temp_{0:7}$

$temp_{0:8} \leftarrow EXTS(a_{8:15}) + EXTS(b_{8:15})$
$d_{8:15} \leftarrow temp_{0:7}$

$temp_{0:8} \leftarrow EXTS(a_{16:23}) + EXTS(b_{16:23})$
$d_{16:23} \leftarrow temp_{0:7}$

$temp_{0:8} \leftarrow EXTS(a_{24:31}) + EXTS(b_{24:31})$
$d_{24:31} \leftarrow temp_{0:7}$

$temp_{0:8} \leftarrow EXTS(a_{32:39}) + EXTS(b_{32:39})$
$d_{32:39} \leftarrow temp_{0:7}$

$temp_{0:8} \leftarrow EXTS(a_{40:47}) + EXTS(b_{40:47})$
$d_{40:47} \leftarrow temp_{0:7}$

$temp_{0:8} \leftarrow EXTS(a_{48:55}) + EXTS(b_{48:55})$
$d_{48:55} \leftarrow temp_{0:7}$

$temp_{0:8} \leftarrow EXTS(a_{56:63}) + EXTS(b_{56:63})$
$d_{56:63} \leftarrow temp_{0:7}$

The signed byte elements of parameter **a** are added to the corresponding signed elements of parameter **b**, producing 9-bit signed integer sums. The high-order 8 bits of the results are placed into the corresponding byte elements of parameter **d**.

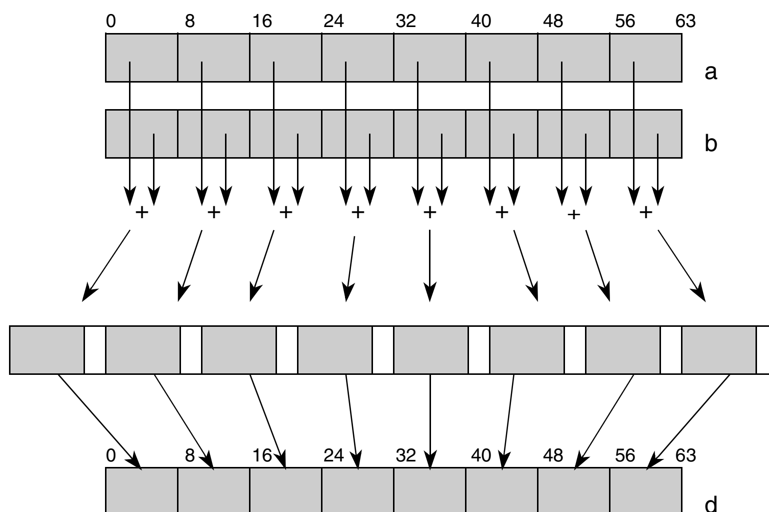


Figure 3-72. Vector Average Byte Signed (__ev_avgbs)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evavgbs d,a,b**

__ev_avgbsr

__ev_avgbsr

Vector Average Byte Signed with Round

d = __ev_avgbsr (a,b)

$temp_{0:8} \leftarrow EXTS(a_{0:7}) + EXTS(b_{0:7}) + 1$
$d_{0:7} \leftarrow temp_{0:7}$

$temp_{0:8} \leftarrow EXTS(a_{8:15}) + EXTS(b_{8:15}) + 1$
$d_{8:15} \leftarrow temp_{0:7}$

$temp_{0:8} \leftarrow EXTS(a_{16:23}) + EXTS(b_{16:23}) + 1$
$d_{16:23} \leftarrow temp_{0:7}$

$temp_{0:8} \leftarrow EXTS(a_{24:31}) + EXTS(b_{24:31}) + 1$
$d_{24:31} \leftarrow temp_{0:7}$

$temp_{0:8} \leftarrow EXTS(a_{32:39}) + EXTS(b_{32:39}) + 1$
$d_{32:39} \leftarrow temp_{0:7}$

$temp_{0:8} \leftarrow EXTS(a_{40:47}) + EXTS(b_{40:47}) + 1$
$d_{40:47} \leftarrow temp_{0:7}$

$temp_{0:8} \leftarrow EXTS(a_{48:55}) + EXTS(b_{48:55}) + 1$
$d_{48:55} \leftarrow temp_{0:7}$

$temp_{0:8} \leftarrow EXTS(a_{56:63}) + EXTS(b_{56:63}) + 1$
$d_{56:63} \leftarrow temp_{0:7}$

The signed byte elements of parameter **a** are added to the corresponding signed elements of parameter **b**, producing 9-bit signed integer sums. The sums are incremented by 1, and the high-order 8 bits of the results are placed into the corresponding byte elements of parameter **d**.

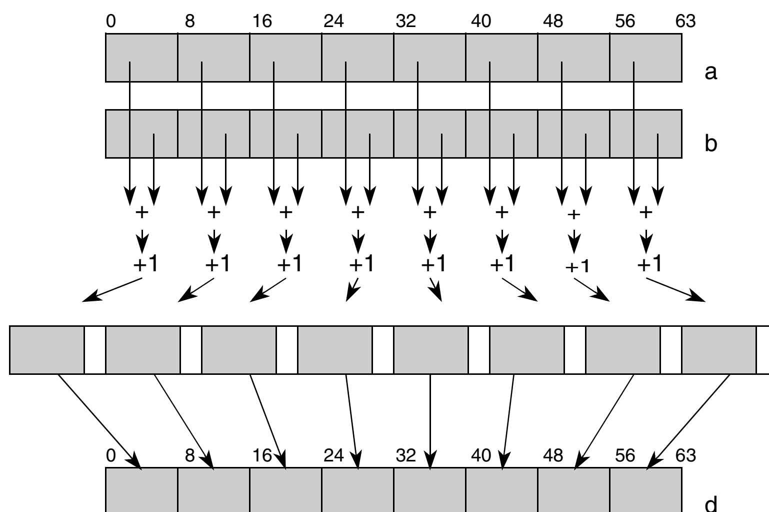


Figure 3-73. Vector Average Byte Signed with Round (__ev_avgbsr)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evavgbsr d,a,b**

__ev_avgbu

__ev_avgbu

Vector Average Byte Unsigned

d = __ev_avgbu (a,b)

$temp_{0:8} \leftarrow EXTZ(a_{0:7}) + EXTZ(b_{0:7})$
$d_{0:7} \leftarrow temp_{0:7}$

$temp_{0:8} \leftarrow EXTZ(a_{8:15}) + EXTZ(b_{8:15})$
$d_{8:15} \leftarrow temp_{0:7}$

$temp_{0:8} \leftarrow EXTZ(a_{16:23}) + EXTZ(b_{16:23})$
$d_{16:23} \leftarrow temp_{0:7}$

$temp_{0:8} \leftarrow EXTZ(a_{24:31}) + EXTZ(b_{24:31})$
$d_{24:31} \leftarrow temp_{0:7}$

$temp_{0:8} \leftarrow EXTZ(a_{32:39}) + EXTZ(b_{32:39})$
$d_{32:39} \leftarrow temp_{0:7}$

$temp_{0:8} \leftarrow EXTZ(a_{40:47}) + EXTZ(b_{40:47})$
$d_{40:47} \leftarrow temp_{0:7}$

$temp_{0:8} \leftarrow EXTZ(a_{48:55}) + EXTZ(b_{48:55})$
$d_{48:55} \leftarrow temp_{0:7}$

$temp_{0:8} \leftarrow EXTZ(a_{56:63}) + EXTZ(b_{56:63})$
$d_{56:63} \leftarrow temp_{0:7}$

The unsigned byte elements of parameter **a** are added to the corresponding unsigned elements of parameter **b**, producing 9-bit unsigned integer sums. The high-order 8 bits of the sums are placed into the corresponding byte elements of parameter **d**.

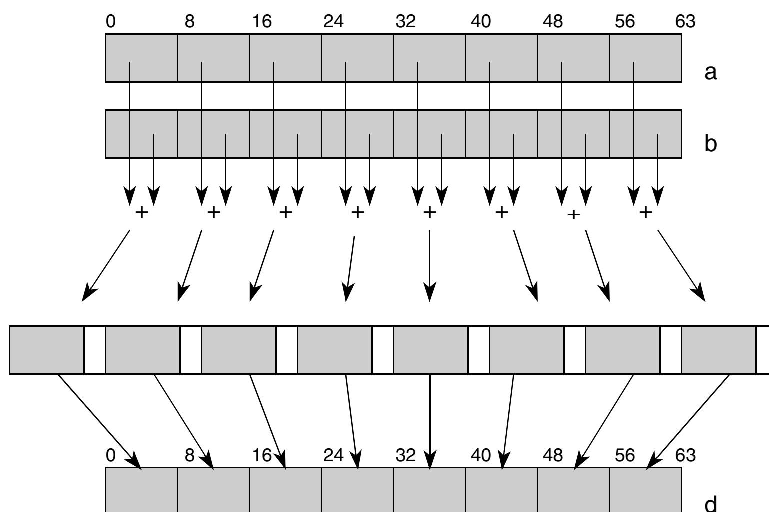


Figure 3-74. Vector Average Byte Unsigned (__ev_avgbu)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evavgbu d,a,b**

__ev_avgbur

__ev_avgbur

Vector Average Byte Unsigned with Round

d = __ev_avgbur (a,b)

$temp_{0:8} \leftarrow EXTZ(a_{0:7}) + EXTZ(b_{0:7}) + 1$
$d_{0:7} \leftarrow temp_{0:7}$

$temp_{0:8} \leftarrow EXTZ(a_{8:15}) + EXTZ(b_{8:15}) + 1$
$d_{8:15} \leftarrow temp_{0:7}$

$temp_{0:8} \leftarrow EXTZ(a_{16:23}) + EXTZ(b_{16:23}) + 1$
$d_{16:23} \leftarrow temp_{0:7}$

$temp_{0:8} \leftarrow EXTZ(a_{24:31}) + EXTZ(b_{24:31}) + 1$
$d_{24:31} \leftarrow temp_{0:7}$

$temp_{0:8} \leftarrow EXTZ(a_{32:39}) + EXTZ(b_{32:39}) + 1$
$d_{32:39} \leftarrow temp_{0:7}$

$temp_{0:8} \leftarrow EXTZ(a_{40:47}) + EXTZ(b_{40:47}) + 1$
$d_{40:47} \leftarrow temp_{0:7}$

$temp_{0:8} \leftarrow EXTZ(a_{48:55}) + EXTZ(b_{48:55}) + 1$
$d_{48:55} \leftarrow temp_{0:7}$

$temp_{0:8} \leftarrow EXTZ(a_{56:63}) + EXTZ(b_{56:63}) + 1$
$d_{56:63} \leftarrow temp_{0:7}$

The unsigned byte elements of parameter **a** are added to the corresponding unsigned elements of parameter **b**, producing 9-bit unsigned integer sums. The sums are incremented by 1 and the high-order 8 bits of the results are placed into the corresponding byte elements of parameter **d**.

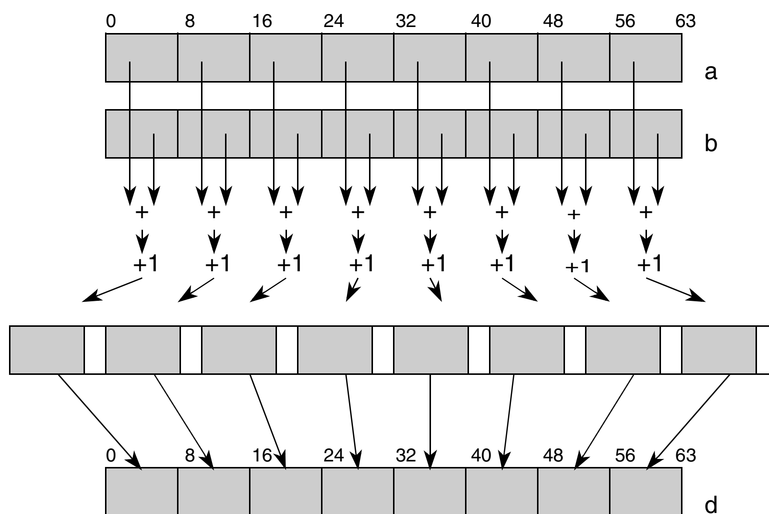


Figure 3-75. Vector Average Byte Unsigned with Round (__ev_avgbur)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evavgbur d,a,b**

__ev_avgds __ev_avgds

Vector Average Doubleword Signed

d = __ev_avgds (a,b)

```
temp0:64 ←EXTS65(a0:63) + EXTS65(b0:63)
d0:63 ←temp0:63
```

The signed doubleword in parameter **a** is added to the signed doubleword in parameter **b**, producing 65-bit signed integer sum. The high-order 64 bits of the result is placed into parameter **d**.

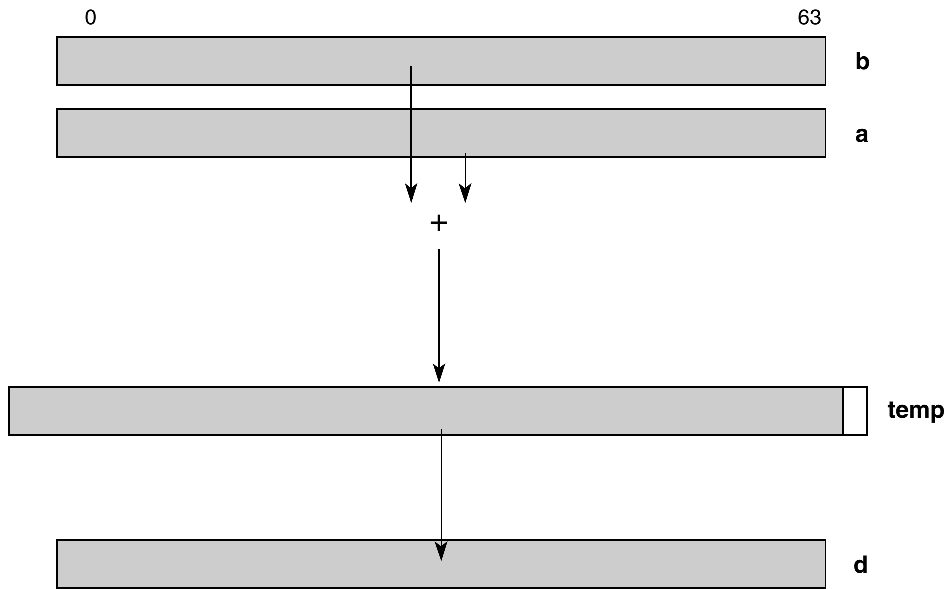


Figure 3-76. Vector Average Doubleword Signed (__ev_avgds)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evavgds d,a,b**

__ev_avgdsr __ev_avgdsr

Vector Average Doubleword Signed with Round

d = __ev_avgdsr (a,b)

```
temp0:64 ←EXTS65(a0:63) + EXTS65(b0:63) + 1
d0:63 ←temp0:63
```

The signed doubleword in parameter **a** is added to the signed doubleword in parameter **b**, producing 65-bit signed integer sum. The sum is incremented by 1 and the high-order 64 bits of the result is placed into parameter **d**.

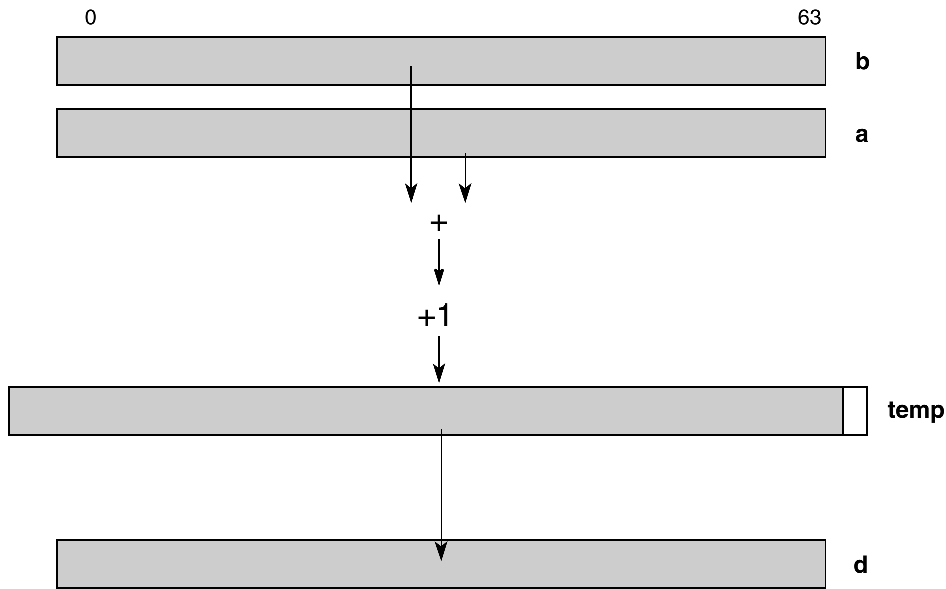


Figure 3-77. Vector Average Doubleword Signed with Round (__ev_avgdsr)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evavgdsr d,a,b**

__ev_avgdu __ev_avgdu

Vector Average Doubleword Unsigned

d = __ev_avgdu (a,b)

```
temp0:64 ←EXTZ65(a0:63) + EXTZ65(b0:63)
d0:63 ←temp0:63
```

The unsigned doubleword in parameter **a** is added to the unsigned doubleword in parameter **b**, producing 65-bit unsigned integer sum. The high-order 64 bits of the result is placed into parameter **d**.

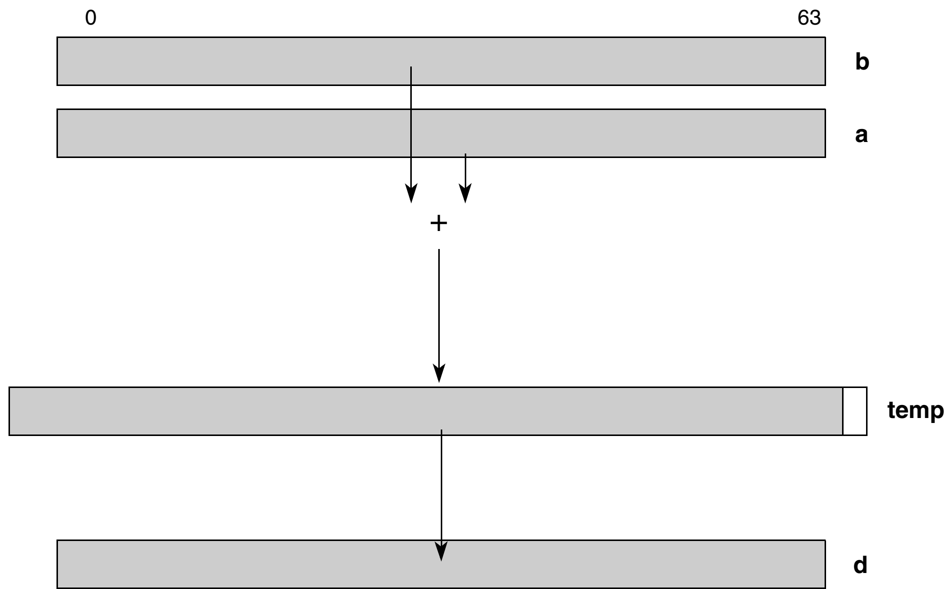


Figure 3-78. Vector Average Doubleword Unsigned (__ev_avgdu)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evavgdu d,a,b**

__ev_avgdur __ev_avgdur

Vector Average Doubleword Unsigned with Round

d = __ev_avgdur (a,b)

```
temp0:64 ←EXTZ65(a0:63) + EXTZ65(b0:63) + 1
d0:63 ←temp0:63
```

The unsigned doubleword in parameter **a** is added to the unsigned doubleword in parameter **b**, producing a 65-bit unsigned integer sum. The sum is incremented by 1 and the high-order 64 bits of the result is placed into parameter **d**.

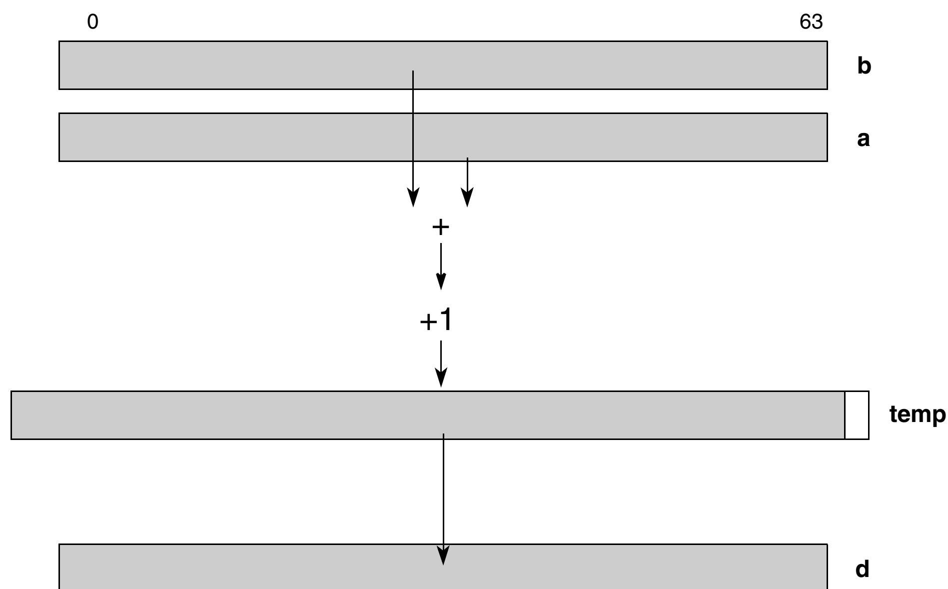


Figure 3-79. Vector Average Doubleword Unsigned with Round (__ev_avgdur)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evavgdur d,a,b**

__ev_avghs

__ev_avghs

Vector Average Half Word Signed

d = __ev_avghs (a,b)

```
temp0:16 ←EXTS(a0:15) + EXTS(b0:15)
d0:15 ←temp0:15

temp0:16 ←EXTS(a16:31) + EXTS(b16:31)
d16:31 ←temp0:15

temp0:16 ←EXTS(a32:47) + EXTS(b32:47)
d32:47 ←temp0:15

temp0:16 ←EXTS(a48:63) + EXTS(b48:63)
d48:63 ←temp0:15
```

The signed half word elements of parameter **a** are added to the signed half word elements of parameter **b**, producing 17-bit signed integer sums. The high-order 16 bits of the results are placed into the corresponding half word elements of parameter **d**.

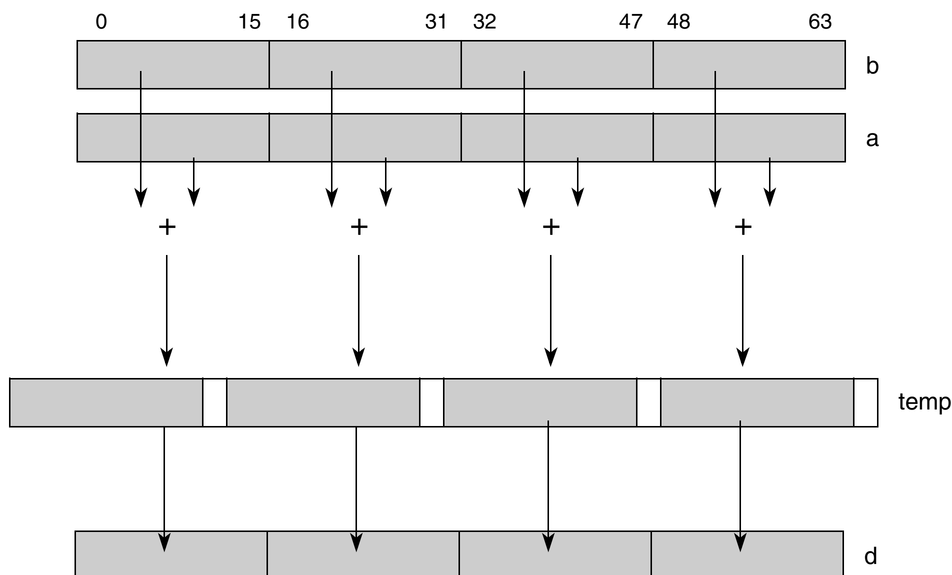


Figure 3-80. Vector Average Half Word Signed (__ev_avghs)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evavghs d,a,b**

__ev_avghsr __ev_avghsr

Vector Average Half Word Signed with Round

d = __ev_avghsr (a,b)

```
temp0:16 ←EXTS(a0:15) + EXTS(b0:15) + 1
d0:15 ←temp0:15

temp0:16 ←EXTS(a16:31) + EXTS(b16:31) + 1
d16:31 ←temp0:15

temp0:16 ←EXTS(a32:47) + EXTS(b32:47) + 1
d32:47 ←temp0:15

temp0:16 ←EXTS(a48:63) + EXTS(b48:63) + 1
d48:63 ←temp0:15
```

The signed half word elements of parameter **a** are added to the signed half word elements of parameter **b**, producing 17-bit signed integer sums. The sums are incremented by 1 and the high-order 16 bits of the results are placed into the corresponding half word elements of parameter **d**.

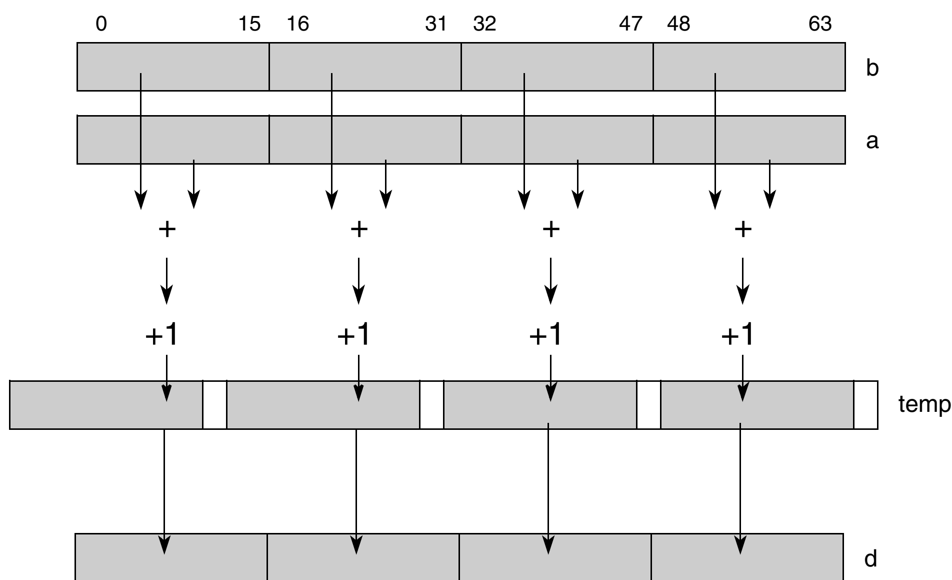


Figure 3-81. Vector Average Half Word Signed with Round (__ev_avghsr)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evavghsr d,a,b**

__ev_avghu __ev_avghu

Vector Average Half Word Unsigned

d = __ev_avghu (a,b)

$temp_{0:16} \leftarrow EXTZ(a_{0:15}) + EXTZ(b_{0:15})$
$d_{0:15} \leftarrow temp_{0:15}$

$temp_{0:16} \leftarrow EXTZ(a_{16:31}) + EXTZ(b_{16:31})$
$d_{16:31} \leftarrow temp_{0:15}$

$temp_{0:16} \leftarrow EXTZ(a_{32:47}) + EXTZ(b_{32:47})$
$d_{32:47} \leftarrow temp_{0:15}$

$temp_{0:16} \leftarrow EXTZ(a_{48:63}) + EXTZ(b_{48:63})$
$d_{48:63} \leftarrow temp_{0:15}$

The unsigned half word elements of parameter **a** are added to the unsigned half word elements of parameter **b**, producing 17-bit unsigned integer sums. The high-order 16 bits of the results are placed into the corresponding half word elements of parameter **d**.

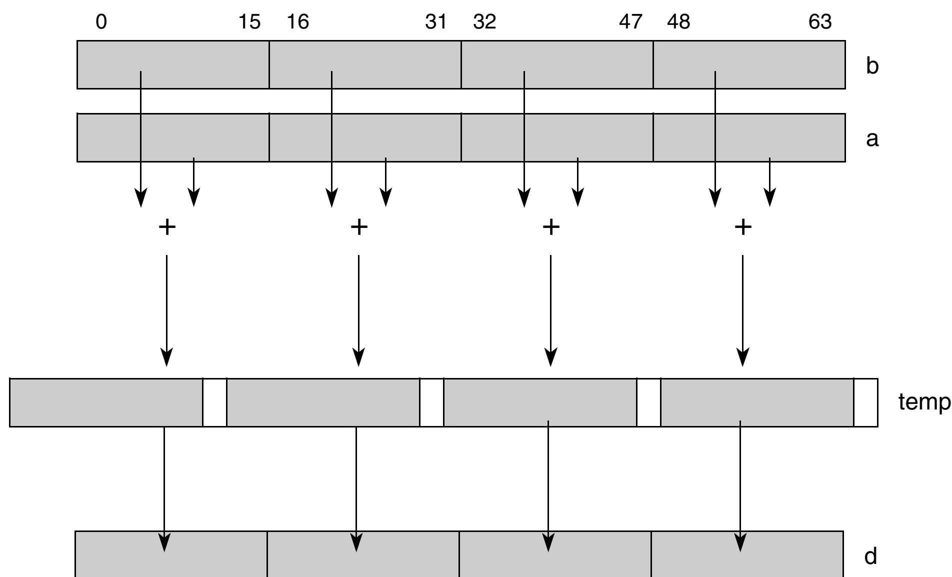


Figure 3-82. Vector Average Half Word Unsigned (__ev_avghu)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evavghu d,a,b**

__ev_avghur

__ev_avghur

Vector Average Half Word Unsigned with Round

d = __ev_avghur (a,b)

$temp_{0:16} \leftarrow EXTZ(a_{0:15}) + EXTZ(b_{0:15}) + 1$
$d_{0:15} \leftarrow temp_{0:15}$

$temp_{0:16} \leftarrow EXTZ(a_{16:31}) + EXTZ(b_{16:31}) + 1$
$d_{16:31} \leftarrow temp_{0:15}$

$temp_{0:16} \leftarrow EXTZ(a_{32:47}) + EXTZ(b_{32:47}) + 1$
$d_{32:47} \leftarrow temp_{0:15}$

$temp_{0:16} \leftarrow EXTZ(a_{48:63}) + EXTZ(b_{48:63}) + 1$
$d_{48:63} \leftarrow temp_{0:15}$

The unsigned half word elements of parameter **a** are added to the unsigned half word elements of parameter **b**, producing 17-bit unsigned integer sums. The sums are incremented by 1 and the high-order 16 bits of the results are placed into the corresponding half word elements of parameter **d**.

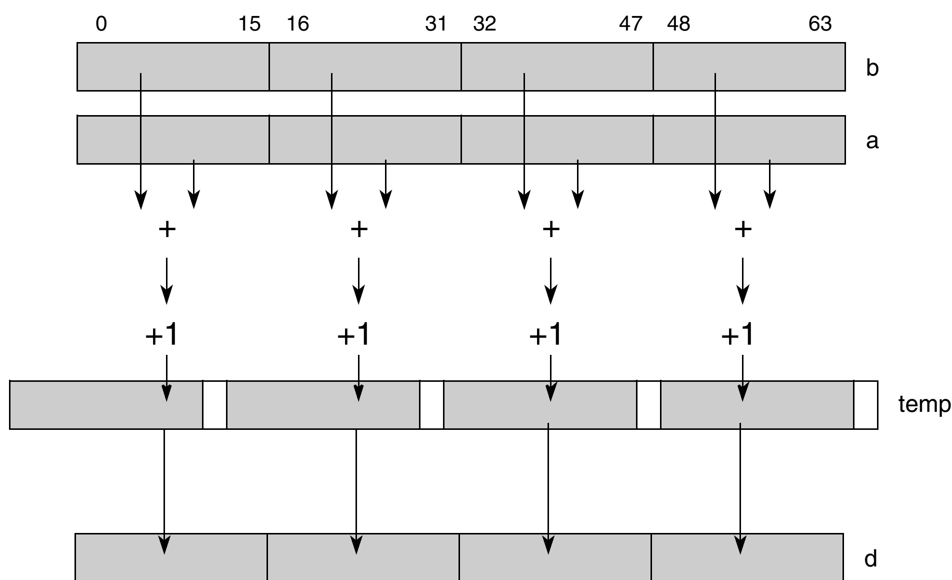


Figure 3-83. Vector Average Half Word Unsigned with Round (__ev_avghur)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evavghur d,a,b**

__ev_avgws

__ev_avgws

Vector Average Word Signed

d = __ev_avgws (a,b)

```
temp0:32 ←EXTS(a0:31) + EXTS(b0:31)
d0:31 ←temp0:31

temp0:32 ←EXTS(a32:63) + EXTS(b32:63)
d32:63 ←temp0:31
```

The signed word elements of parameter **a** are added to the signed word elements of parameter **b**, producing 33-bit signed integer sums. The high-order 32 bits of the results are placed into the corresponding word elements of parameter **d**.

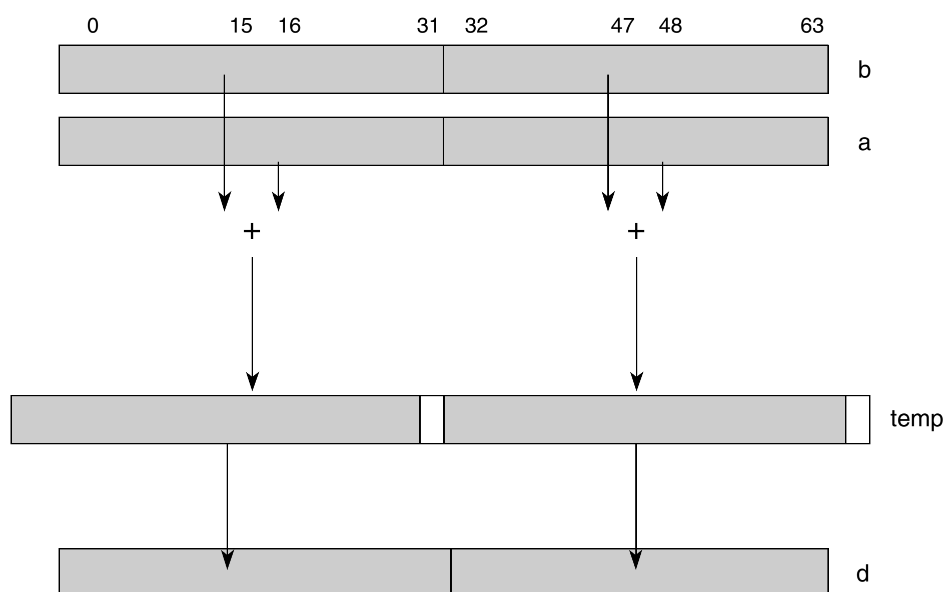


Figure 3-84. Vector Average Word Signed (__ev_avgws)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evavgws d,a,b**

__ev_avgwsr __ev_avgwsr

Vector Average Word Signed with Round

d = __ev_avgwsr (a,b)

```
temp0:32 ←EXTS(a0:31) + EXTS(b0:31) + 1
d0:31 ←temp0:31

temp0:32 ←EXTS(a32:63) + EXTS(b32:63) + 1
d32:63 ←temp0:31
```

The signed word elements of parameter **a** are added to the signed word elements of parameter **b**, producing 33-bit signed integer sums. The sums are incremented by 1 and the high-order 32 bits of the results are placed into the corresponding word elements of parameter **d**.

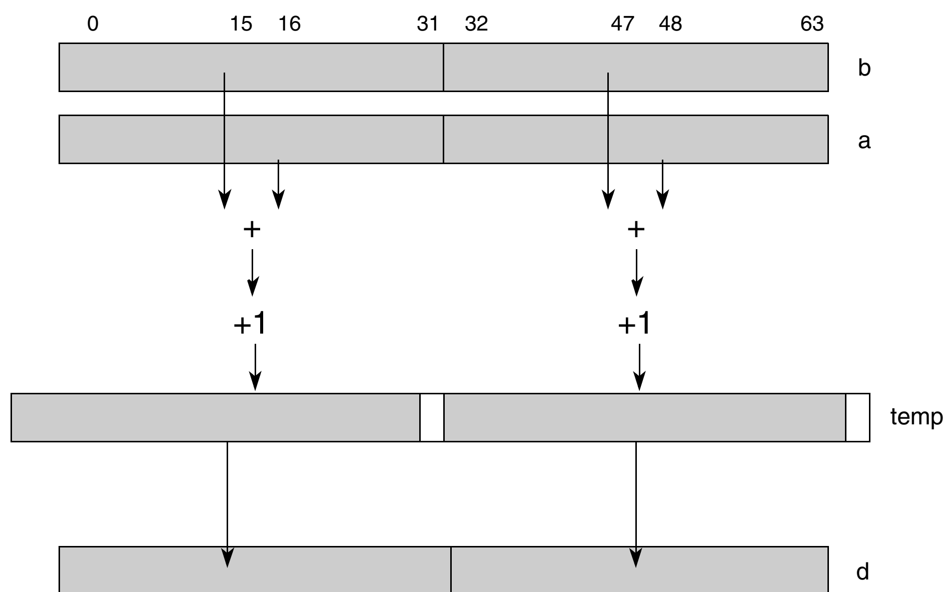


Figure 3-85. Vector Average Word Signed with Round (__ev_avgwsr)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evavgwsr d,a,b**

__ev_avgwu

__ev_avgwu

Vector Average Word Unsigned

d = __ev_avgwu (a,b)

$$temp_{0:32} \leftarrow EXTZ(a_{0:31}) + EXTS(b_{0:31})$$
$$d_{0:31} \leftarrow temp_{0:31}$$

$$temp_{0:32} \leftarrow EXTZ(a_{32:63}) + EXTS(b_{32:63})$$
$$d_{32:63} \leftarrow temp_{0:31}$$

The unsigned word elements of parameter **a** are added to the unsigned word elements of parameter **b**, producing 33-bit unsigned integer sums. The high-order 32 bits of the results are placed into the corresponding word elements of parameter **d**.

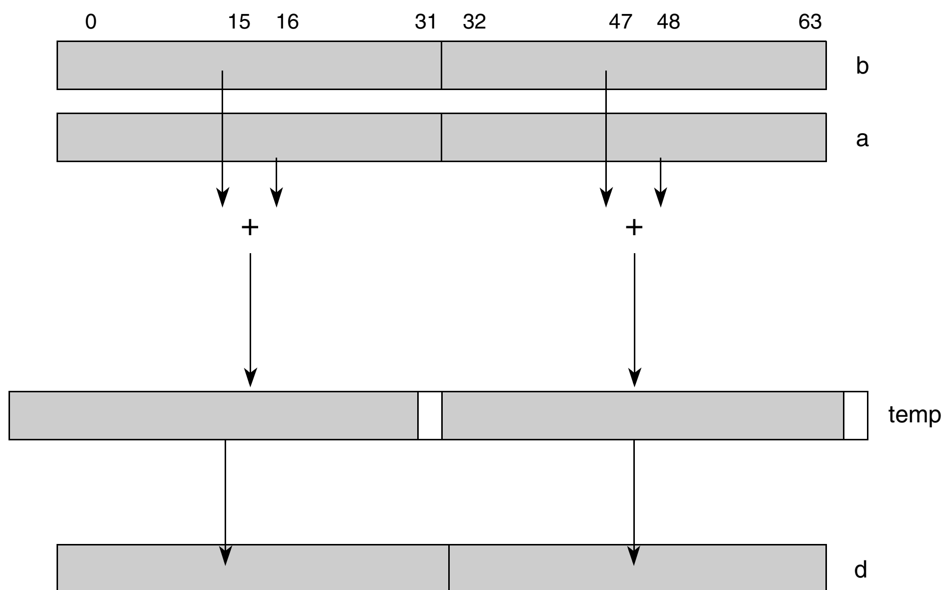


Figure 3-86. Vector Average Word Unsigned (__ev_avgwu)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evavgwu d,a,b**

__ev_avgwur __ev_avgwur

Vector Average Word Unsigned with Round

d = __ev_avgwur (a,b)

```
temp0:32 ←EXTZ(a0:31) + EXTS(b0:31) + 1
d0:31 ←temp0:31

temp0:32 ←EXTZ(a32:63) + EXTS(b32:63) + 1
d32:63 ←temp0:31
```

The unsigned word elements of parameter **a** are added to the unsigned word elements of parameter **b**, producing 33-bit unsigned integer sums. The sums are incremented by 1 and the high-order 32 bits of the results are placed into the corresponding word elements of parameter **d**.

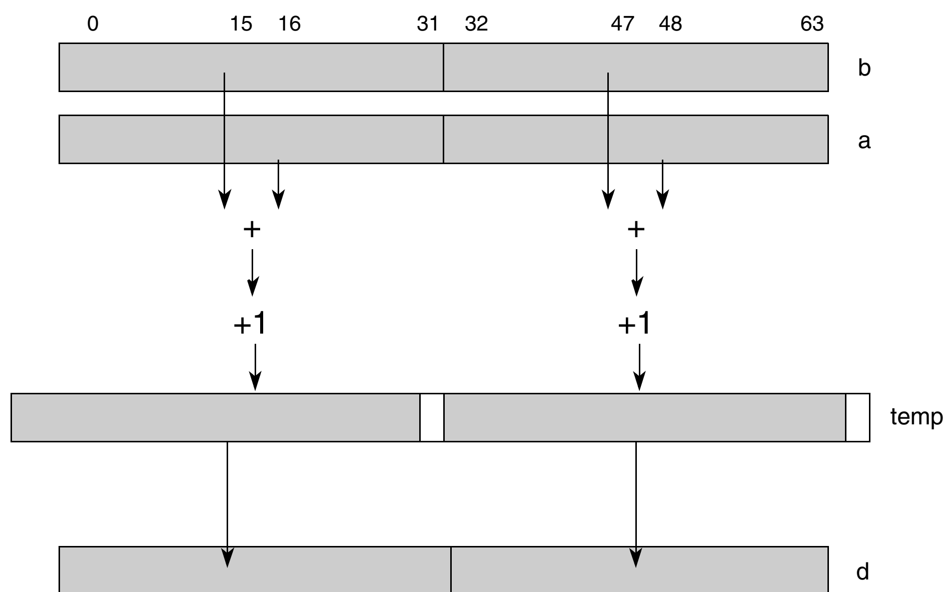


Figure 3-87. Vector Average Word Unsigned with Round (__ev_avgwur)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evavgwur d,a,b**

__ev_clrbe

__ev_clrbe

Vector Clear Bytes Even

d = __ev_clrbe (a,b)

```
d0:15 ←a0:15 & (¬8(mask0) || 81)
d16:31 ←a16:31 & (¬8(mask1) || 81)
d32:47 ←a32:47 & (¬8(mask2) || 81)
d48:63 ←a48:63 & (8(mask3) || 81)
```

Each even byte element in parameter **a** is logically and-ed with the associated mask bit specified by parameter **b** and is placed into the corresponding byte element of parameter **d**, as shown in Figure 3-88. Odd byte elements are placed into the corresponding byte element of parameter **d**. A mask value of '0' causes the corresponding even byte element of parameter **d** to be cleared.

NOTE

Mask bit $mask_3$ is the least significant bit of parameter **b**. A value of 0x8 for parameter **b** sets $mask_0$ to a one and the remaining mask bits to zero.

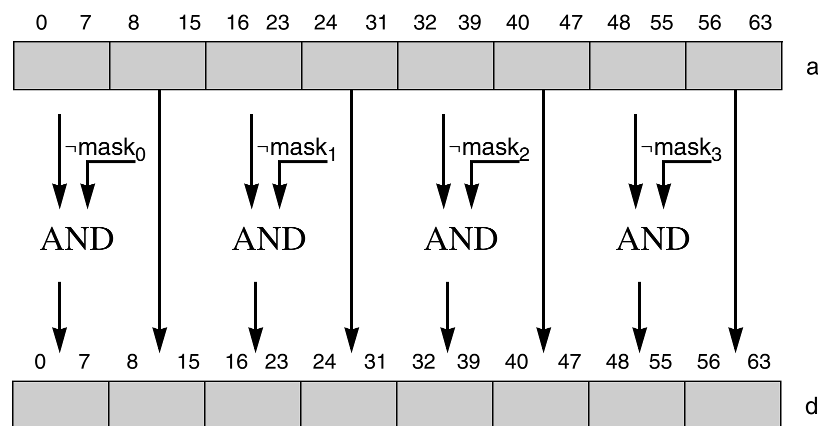


Figure 3-88. Vector Clear Bytes Even (__ev_clrbe)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	4-bit unsigned literal	**evclrbe d,a,b**

__ev_clrbo

__ev_clrbo

Vector Clear Bytes Odd

d = __ev_clrbo (a,b)

$$
\begin{aligned}
d_{0:15} &\leftarrow a_{0:15} \;\&\; (\neg {}^{8}1 \;||\; {}^{8}(\mathrm{mask}_0)) \\
d_{16:31} &\leftarrow a_{16:31} \;\&\; (\neg {}^{8}1 \;||\; {}^{8}(\mathrm{mask}_1)) \\
d_{32:47} &\leftarrow a_{32:47} \;\&\; (\neg {}^{8}1 \;||\; {}^{8}(\mathrm{mask}_2)) \\
d_{48:63} &\leftarrow a_{48:63} \;\&\; (\neg {}^{8}1 \;||\; {}^{8}(\mathrm{mask}_3))
\end{aligned}
$$

Each odd byte element in parameter **a** is logically and-ed with the associated mask bit specified by parameter **b** and is placed into the corresponding byte element of parameter **d**, as shown in Figure 3-89. Even byte elements are placed into the corresponding byte element of parameter **d**. A mask value of '0' causes the corresponding odd byte element of parameter **d** to be cleared.

NOTE

Mask bit mask_3 is the least significant bit of parameter **b**. A value of 0x8 for parameter **b** sets mask_0 to a one and the remaining mask bits to zero.

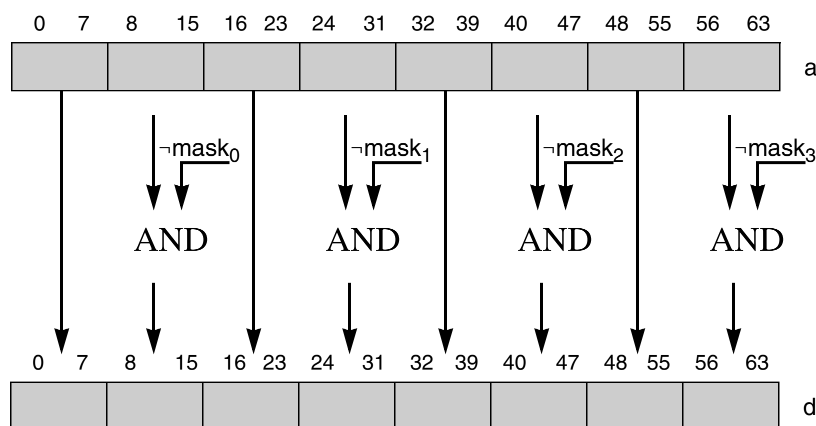


Figure 3-89. Vector Clear Bytes Odd (__ev_clrbo)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	4-bit unsigned literal	**evclrbo d,a,b**

__ev_clrh __ev_clrh

Vector Clear Half Words

d = __ev_clrh (a,b)

$$d_{0:15} \leftarrow a_{0:15} \ \& \ (\neg^{16}(mask_0))$$
$$d_{16:31} \leftarrow a_{16:31} \ \& \ (\neg^{16}(mask_1))$$
$$d_{32:47} \leftarrow a_{32:47} \ \& \ (\neg^{16}(mask_2))$$
$$d_{48:63} \leftarrow a_{48:63} \ \& \ (\neg^{16}(mask_3))$$

Each half word element in parameter **a** is logically and-ed with the associated mask bit specified by parameter **b** and is placed into the corresponding half word element of parameter **d**, as shown in Figure 3-90. A mask value of ‘0’ causes the corresponding half word element of parameter **d** to be cleared.

NOTE

Mask bit $mask_3$ is the least significant bit of parameter **b**. A value of 0x8 for parameter **b** sets $mask_0$ to a one and the remaining mask bits to zero.

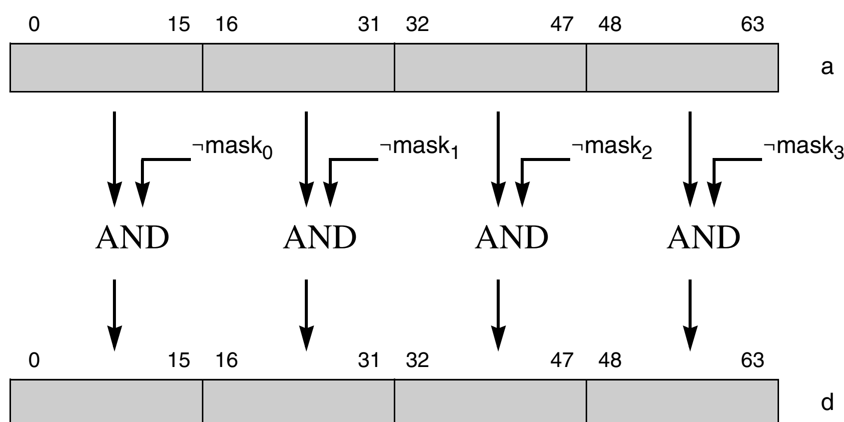


Figure 3-90. Vector Clear Half Words (__ev_clrh)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	4-bit unsigned literal	**evclrh 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**.

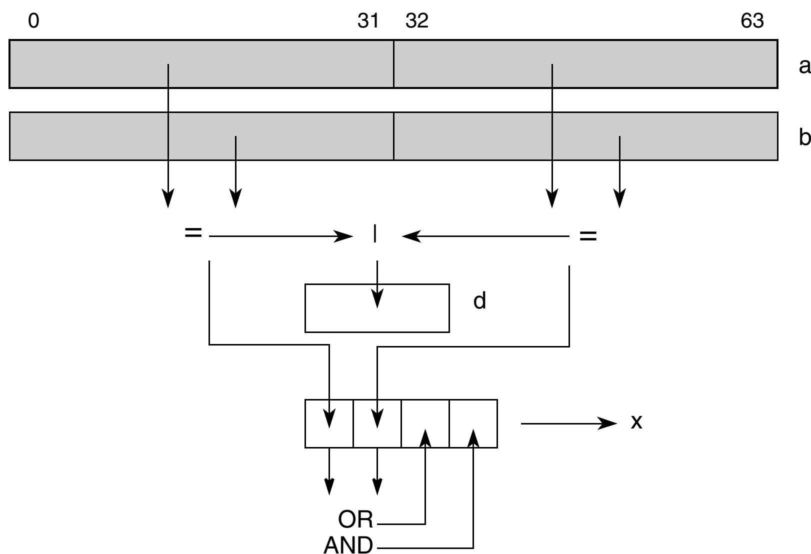


Figure 3-91. Vector Any Equal (__ev_any_eq)

d	a	b	Maps to
_Bool	__ev64_opaque__	__ev64_opaque__	**evcmpeq 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**.

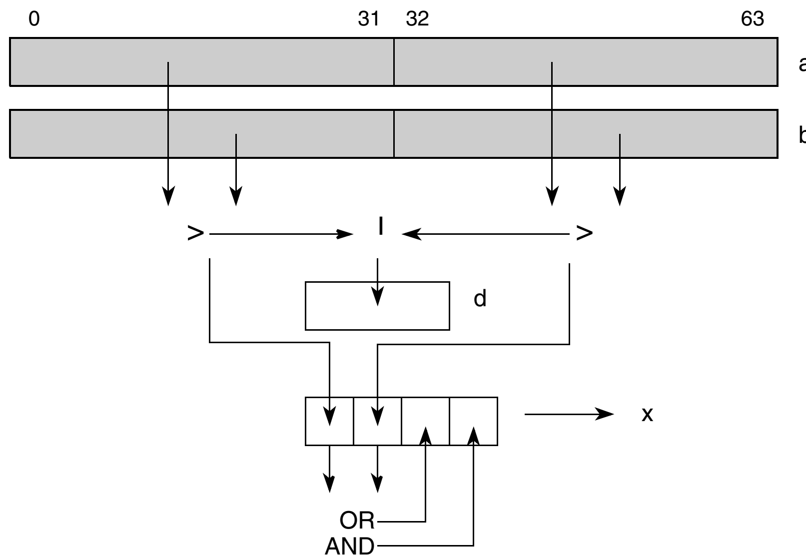


Figure 3-92. Vector Any Greater Than Signed (__ev_any_gts)

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 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**.

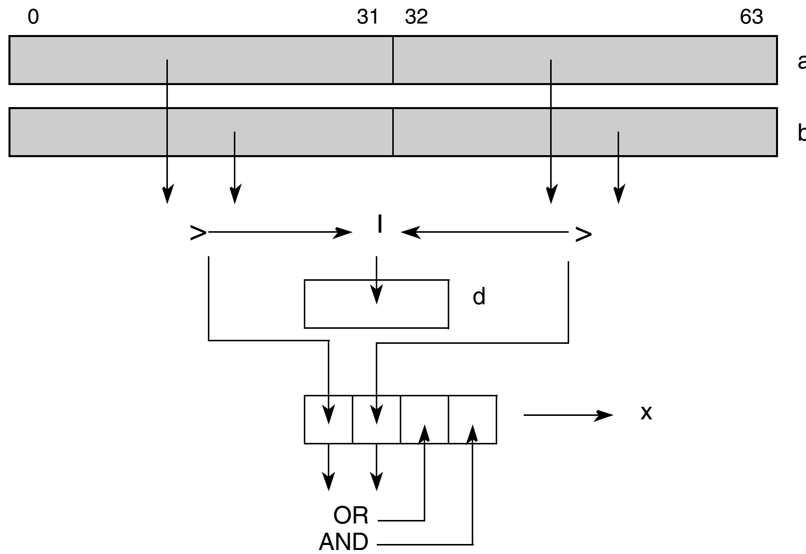


Figure 3-93. Vector Any Greater Than Unsigned (__ev_any_gtu)

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**.

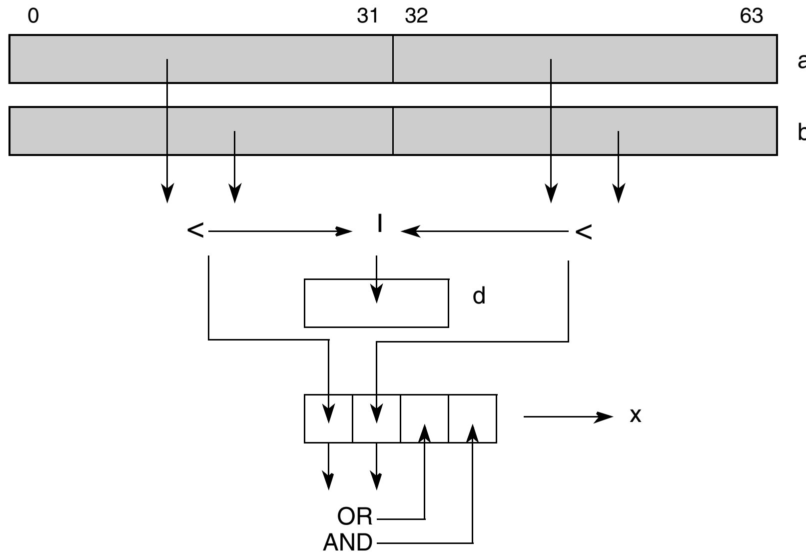


Figure 3-94. Vector Any Less Than Signed(__ev_any_lts)

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**.

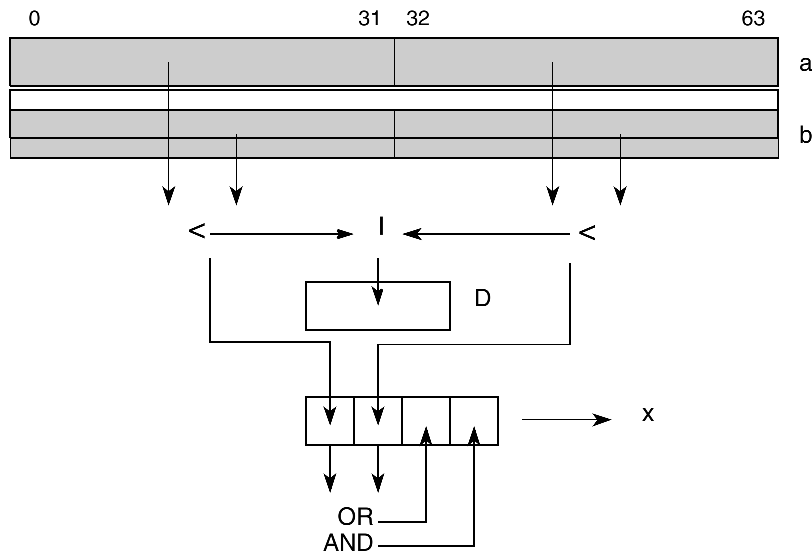


Figure 3-95. Vector Any Less Than Unsigned (__ev_any_ltu)

d	a	b	Maps to
_Bool	__ev64_opaque__	__ev64_opaque__	**evcmpltu x,a,b**

__ev_cntlsh

__ev_cntlsh

Vector Count Leading Signed Bits Half Word

d = __ev_cntlsh (a)

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

evcntlzh is used for unsigned parameters; **evcntlsh** is used for signed parameters.

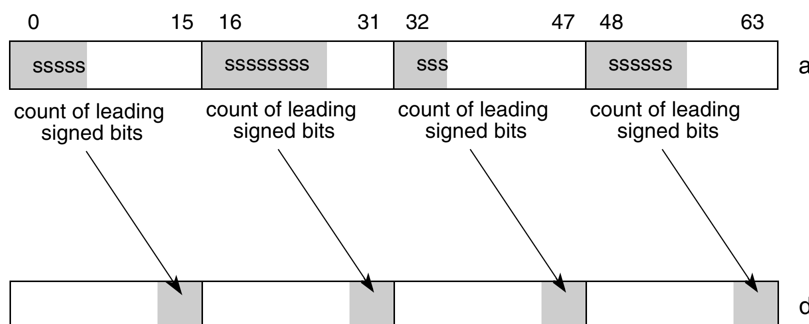


Figure 3-96. Vector Count Leading Signed Bits Half Word (__ev_cntlsh)

d	a	Maps to
__ev64_opaque__	__ev64_opaque__	**evcntlsh d,a**

__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.

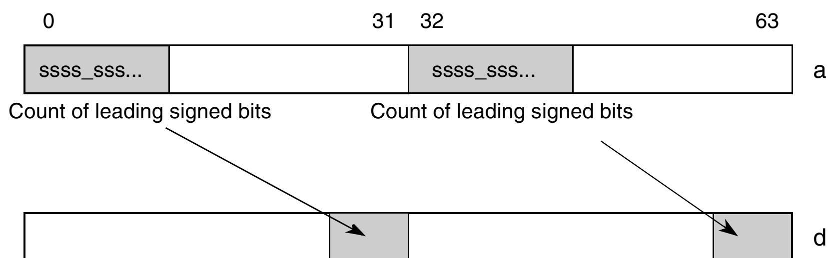


Figure 3-97. Vector Count Leading Signed Bits Word (__ev_cntlsw)

d	a	Maps to
__ev64_opaque__	__ev64_opaque__	**evcntlsw d,a**

__ev_cntlzh __ev_cntlzh

Vector Count Leading Zeros Half Word

d = __ev_cntlzh (a)

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

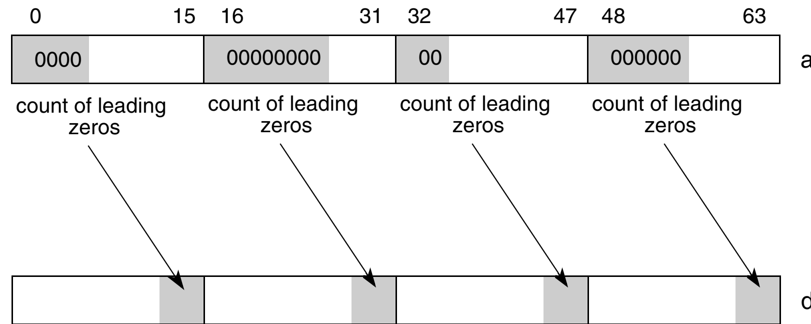


Figure 3-98. Vector Count Leading Zeros Half Word (__ev_cntlzh)

d	a	Maps to
__ev64_opaque__	__ev64_opaque__	**evcntlzh 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**.

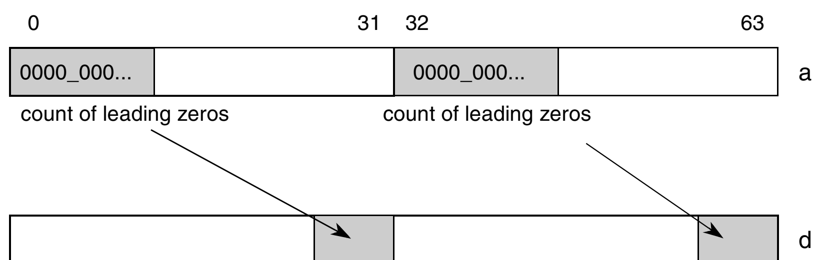


Figure 3-99. Vector Count Leading Zeros Word (__ev_cntlzw)

d	a	Maps to
__ev64_opaque__	__ev64_opaque__	**evcntlzw d,a**

__ev_diff2his[a] __ev_diff2his[a]

Vector Difference of 2 Halfwords Interleaved Signed (to Accumulator)

d = __ev_diff2his (a) (A = 0)

d = __ev_diff2hisa (a) (A = 1)

```
d0:31 ←EXTS32(a0:15) - EXTS32(a32:47)
d32:63 ←EXTS32(a16:31) - EXTS32(a48:63)

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

Groups of two interleaved signed halfword elements of parameter **a** are subtracted, and the results are placed into the word elements of parameter **d**. If A = 1, the result in parameter **d** is also placed into the accumulator.

Other registers altered: ACC (if A=1)

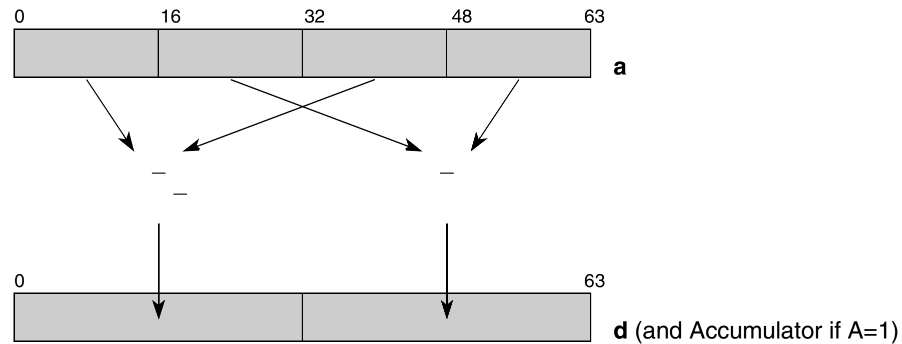


Figure 3-100. Vector Difference of 2 Halfwords Interleaved Signed (to Accumulator) (__ev_diff2his[a])

d	a	Maps to
__ev64_opaque__	__ev64_opaque__	**evdiff2his d,a**
__ev64_opaque__	__ev64_opaque__	**evdiff2hisa d,a**

__ev_diff2hisaaw __ev_diff2hisaaw

Vector Difference of 2 Halfwords Interleaved Signed and Accumulate into Words

d = __ev_diff2hisaaw (a)

```
d0:31 ←ACC0:31 + (EXTS32(a0:15) - EXTS32(a32:47))
d32:63 ←ACC32:63 + (EXTS32(a16:31) - EXTS32(a48:63))

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

Groups of two interleaved signed halfword elements of parameter **a** are subtracted, the differences are added to the corresponding word elements in the accumulator, and the results are placed into the word elements of parameter **d** and the accumulator.

Other registers altered: ACC

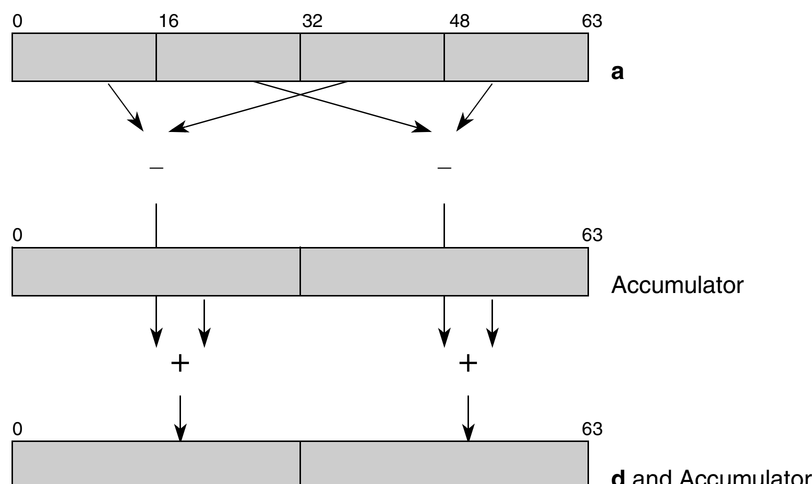


Figure 3-101. Vector Difference of 2 Halfwords Interleaved Signed and Accumulate (__ev_diff2hisaaw)

d	a	Maps to
__ev64_opaque__	__ev64_opaque__	**evdiff2hisaaw d,a**

__ev_divs __ev_divs

Vector Divide Signed

d = __ev_divs (a,b)

```
dividend ←a0:63
divisor ←b0:63
d0:63 ←dividend ÷ divisor

ov←0
if ((dividend < 0) & (divisor = 0)) then
    d0:63 ←0x8000_0000_0000_0000
    ov ←1
else if ((dividend >= 0) & (divisor = 0)) then
    d0:63 ←0x7FFF_FFFF_FFFF_FFFF
    ov ←1
else if ((dividend = 0x8000_0000_0000_0000) & (divisor = 0xFFFF_FFFF_FFFF_FFFF))
    then
        d0:63 ←0x7FFF_FFFF_FFFF_FFFF
        ov ←1
endif

SPEFSCROVH ←0
SPEFSCROV ←ov
SPEFSCRSOV ←SPEFSCRSOV | ov
```

The dividend in parameter **a** is divided by the divisor in parameter **b**. The resulting 64-bit quotient is placed into parameter **d**. The remainder is not supplied. The operands and quotient 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 parameter **d**.

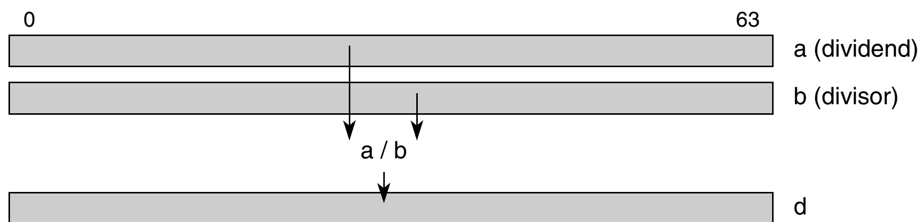


Figure 3-102. Vector Divide Signed (__ev_divs)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evdivs d,a,b**

__ev_divu __ev_divu

Vector Divide Unsigned

d = __ev_divu (a,b)

```
dividend ←a0:63
divisor ←b0:63
d0:63 ←dividend ÷ divisor

ov←0
if (divisor = 0) then
    d0:63 ←0xFFFF_FFFF_FFFF_FFFF
    ov ←1
endif

SPEFSCROVH ←0
SPEFSCROV ←ov
SPEFSCRSOV ←SPEFSCRSOV | ov
```

The dividend in parameter **a** is divided by the divisor in parameter **b**. The resulting 64-bit quotient is placed into parameter **d**. The remainder is not supplied. The operands and quotient are interpreted as unsigned integers. If 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 parameter **d**.

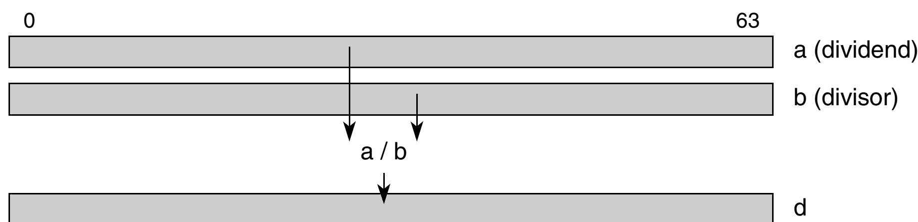


Figure 3-103. Vector Divide Signed (__ev_divs)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evdivu d,a,b**

__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.

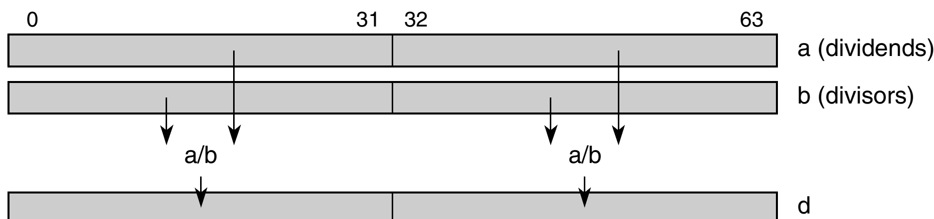


Figure 3-104. Vector Divide Word Signed (__ev_divws)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evdivws d,a,b**

__ev_divwsf __ev_divwsf

Vector Divide Word Signed Fractional

d = __ev_divwsf (a,b)

```
dividendh ←a0:31
dividendl ←a32:63
divisorh ←b0:31
divisorl ←b32:63
d0:31 ←dividendh ÷sf divisorh
d32:63 ←dividendl ÷sf divisorl
ovh ←0
ovl ←0
if (((dividendh0 ^ divisorh0) = 1) & ((|dividendh| > |divisorh|) | (divisorh = 0))) then
    d0:31 ←0x80000000
    ovh ←1
else if ((dividendh0 ^ divisorh0) = 0) & ((|dividendh| >= |divisorh|) | (divisorh = 0))) then
    d0:31 ←0x7FFFFFFF
    ovh ←1
if (((dividendl0 ^ divisorl0) = 1) & ((|dividendl| > |divisorl|) | (divisorl = 0))) then
    d32:63 ←0x80000000
    ovl ←1
else if (((dividendl0 ^ divisorl0) 0) & & ((|dividendl| >= |divisorl|) | (divisorl = 0))) then
    d32:63 ←0x7FFFFFFF
    ovl ←1

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

The two dividends are the two elements contained in parameter **a**. The two divisors are the two elements contained in parameter **b**. The resulting two 32-bit quotients of each element are placed into parameter **d**. The remainders are not supplied. The operands and quotients are interpreted as signed fractions. Each quotient satisfies the equation dividend = (quotient x divisor) + remainder, where the sign of the remainder (if non-zero) is the same as the sign of the dividend. The magnitude of the remainder is less than the magnitude of the divisor. 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 parameter **d**.

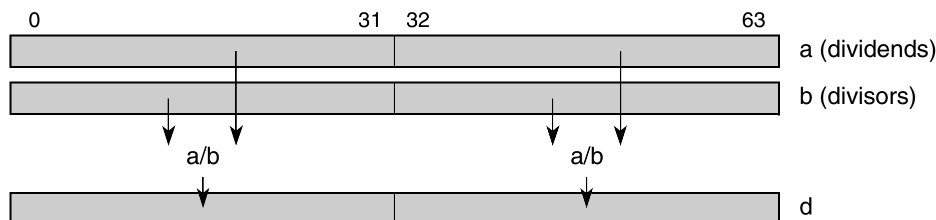


Figure 3-105. Vector Divide Word Signed Fraction (__ev_divwsf)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evdivwsf 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.

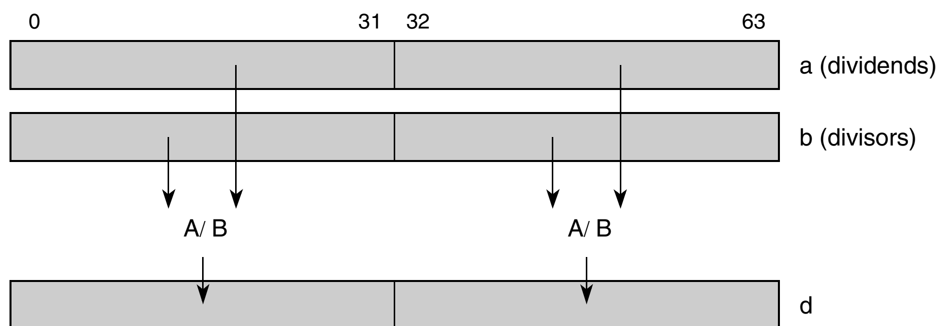


Figure 3-106. Vector Divide Word Unsigned (__ev_divwu)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evdivwu d,a,b**

__ev_divwuf

__ev_divwuf

Vector Divide Word Unsigned Fractional

d = __ev_divwuf (a,b)

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

The two dividends are the two elements contained in parameter **a**. The two divisors are the two elements contained in 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. Operands and quotients are interpreted as unsigned fractions. Each quotient satisfies the equation dividend = (quotient x divisor) + remainder, where the remainder is non-negative and the magnitude of the remainder is less than the magnitude of the divisor. If an overflow (dividend >= divisor) or 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 parameter **d**.

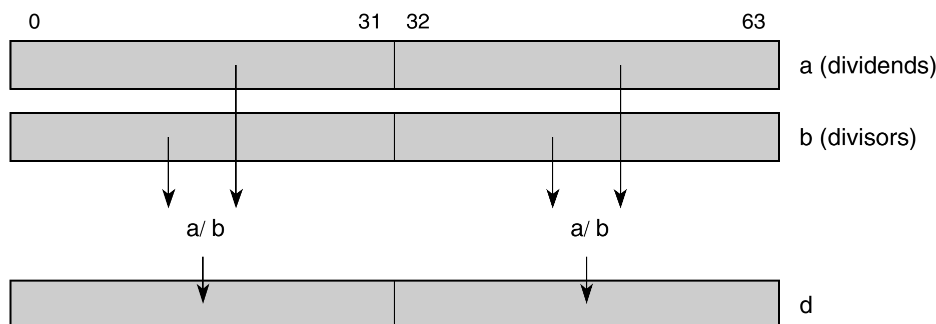


Figure 3-107. Vector Divide Word Unsigned Fractional (__ev_divwuf)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evdivwuf d,a,b**

__ev_dlveb

__ev_dlveb

Vector De-interleave Even Bytes

d = __ev_dlveb (a,b)

$d_{0:63} \leftarrow a_{0:7} \, || \, a_{16:23} \, || \, a_{32:39} \, || \, a_{48:55} \, || \, b_{0:7} \, || \, b_{16:23} \, || \, b_{32:39} \, || \, b_{48:55}$

The even byte elements in parameters **a** and **b** are de-interleaved and placed into parameter **d**.

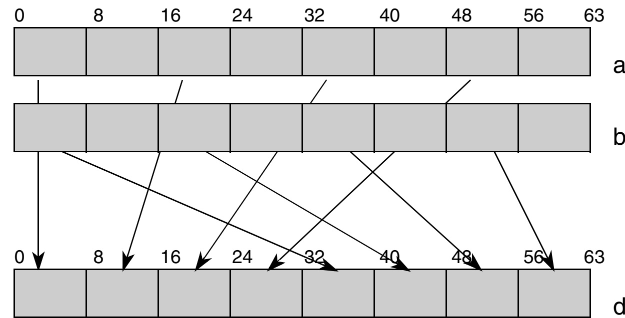


Figure 3-108. Vector De-interleave Even Bytes (__ev_dlveb)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evdlveb d,a,b**

__ev_dlveh __ev_dlveh

Vector De-interleave Even Half Words

d = __ev_dlveh (a,b)

$d_{0:63} \leftarrow a_{0:15} \; || \; a_{32:47} \; || \; b_{0:15} \; || \; b_{32:47}$

The even half word elements in parameters **a** and **b** are de-interleaved and placed into parameter **d**.

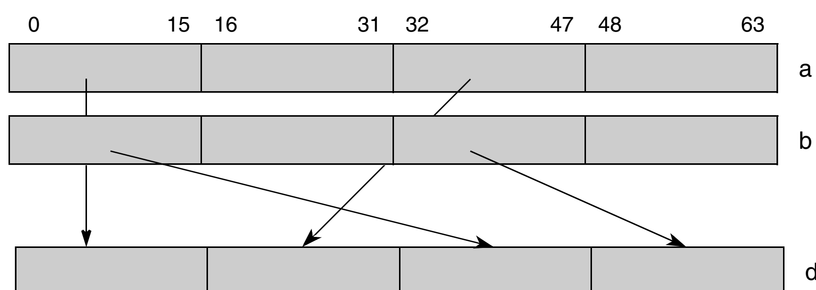


Figure 3-109. Vector De-interleave Even Half Words (__ev_dlveh)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evdlveh d,a,b**

__ev_dlveob

__ev_dlveob

Vector De-interleave Even/Odd Bytes

d = __ev_dlveob (a,b)

$d_{0:63} \leftarrow a_{0:7} \; || \; a_{16:23} \; || \; a_{32:39} \; || \; a_{48:55} \; || \; b_{8:15} \; || \; b_{24:31} \; || \; b_{40:47} \; || \; b_{56:63}$

The even byte elements in parameter **a** and the odd byte elements in parameter **b** are de-interleaved and placed into parameter **d**.

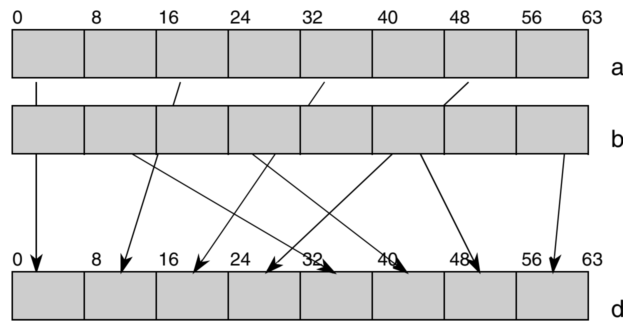


Figure 3-110. Vector De-interleave Even/Odd Bytes (__ev_dlveob)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evdlveob d,a,b**

__ev_dlveoh __ev_dlveoh

Vector De-interleave Even/Odd Half Words

d = __ev_dlveoh (a,b)

$d_{0:63} \leftarrow a_{0:15} \mid\mid a_{32:47} \mid\mid b_{16:31} \mid\mid b_{48:63}$

The even half word elements in parameter **a** and the odd half word elements of parameter **b** are de-interleaved and placed into parameter **d**.

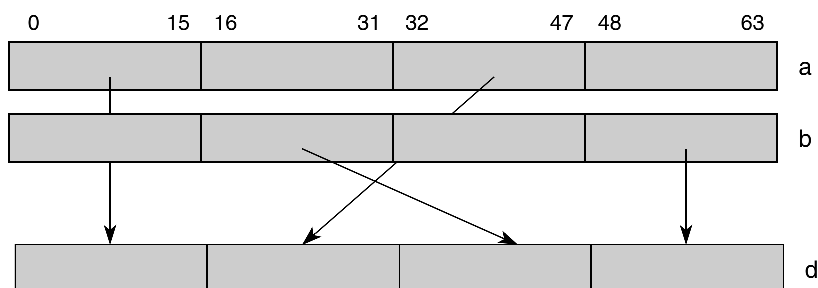


Figure 3-111. Vector De-interleave Even Half Words (__ev_dlveoh)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evdlveoh d,a,b**

__ev_dlvob

__ev_dlvob

Vector De-interleave Odd Bytes

d = __ev_dlvob (a,b)

$d_{0:63} \leftarrow a_{8:15} \,||\, a_{24:31} \,||\, a_{40:47} \,||\, a_{56:63} \,||\, b_{8:15} \,||\, b_{24:31} \,||\, b_{40:47} \,||\, b_{56:63}$

The odd byte elements in parameter **a** and the odd byte elements in parameter **b** are de-interleaved and placed into parameter **d**.

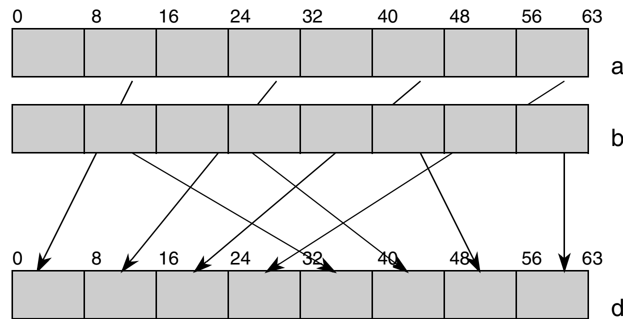


Figure 3-112. Vector De-interleave Odd Bytes (__ev_dlvob)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evdlvob d,a,b**

__ev_dlvoeb __ev_dlvoeb

Vector De-interleave Odd/Even Bytes

d = __ev_dlvoeb (a,b)

$d_{0:63} \leftarrow a_{8:15} \parallel a_{24:31} \parallel a_{40:47} \parallel a_{56:63} \parallel b_{0:7} \parallel b_{16:23} \parallel b_{32:39} \parallel b_{48:55}$

The odd byte elements in parameter **a** and the even byte elements in parameter **b** are de-interleaved and placed into parameter **d**.

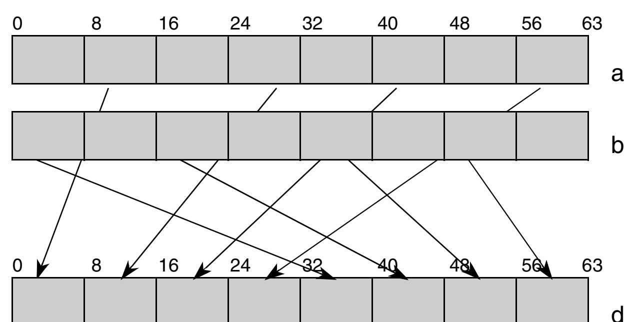


Figure 3-113. Vector De-interleave Odd/Even Bytes (__ev_dlvoeb)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evdlvoeb d,a,b**

__ev_dlvoeh

__ev_dlvoeh

Vector De-interleave Odd/Even Half Words

d = __ev_dlvoeh (a,b)

$d_{0:63} \leftarrow a_{16:31} \;||\; a_{48:63} \;||\; b_{0:15} \;||\; b_{32:47}$

The odd half word elements in parameter **a** and the even half word elements in parameter **b** are de-interleaved and placed into parameter **d**.

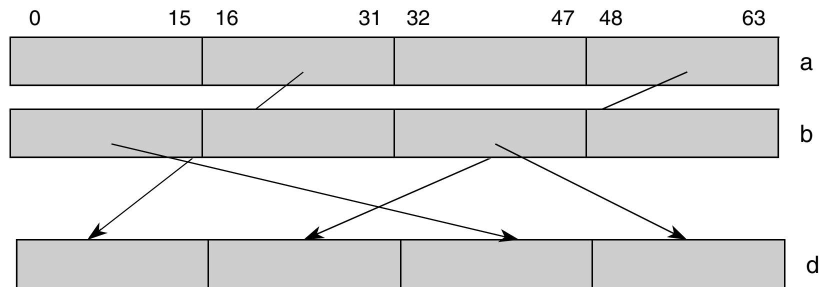


Figure 3-114. Vector De-interleave Odd/Even Half Words (__ev_dlvoeh)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evdlvoeh d,a,b**

__ev_dlvoh

__ev_dlvoh

Vector De-interleave Odd Half Words

d = __ev_dlvoh (a,b)

$d_{0:63} \leftarrow a_{16:31} \; || \; a_{48:63} \; || \; b_{16:31} \; || \; b_{48:63}$

The odd half word elements in parameters **a** and **b** are de-interleaved and placed into parameter **d**.

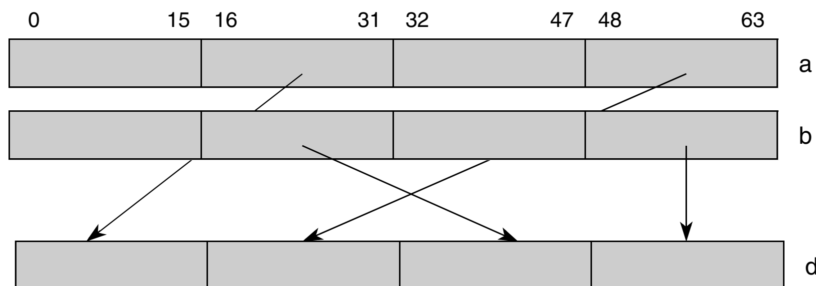


Figure 3-115. Vector De-interleave Odd Half Words (__ev_dlvoh)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evdlvoh d,a,b**

__ev_dotp4hgasmf[a] __ev_dotp4hgasmf[a]

Vector Dot Product of Four Half Words, Guarded, Add, Signed, Modulo, Fractional (to Accumulator)

d = __ev_dotp4hgasmf (a,b) (A = 0)

d = __ev_dotp4hgasmfa (a,b) (A = 1)

```
temp0_{0:32} ← a_{0:15} ×_{sf} b_{0:15}
temp1_{0:32} ← a_{16:31} ×_{sf} b_{16:31}
temp2_{0:32} ← a_{32:47} ×_{sf} b_{32:47}
temp3_{0:32} ← a_{48:63} ×_{sf} b_{48:63}
temp_{0:63} ← EXTS(temp1_{0:31}) + EXTS(temp1_{0:31}) + EXTS(temp2_{0:31}) + EXTS(temp3_{0:31}) //modulo
d_{0:63} ← temp_{0:63}

// update accumulator
if A = 1 then ACC_{0:63} ←d_{0:63}
```

Corresponding half word pairs of signed fractional elements in parameters **a** and **b** are multiplied producing four 33-bit intermediate products. The intermediate 33-bit products are sign-extended and added together to produce a 64-bit result and the sum is placed into parameter **d**. If A = 1, the result in parameter **d** is also placed into the accumulator.

Note: If the two input operands to a multiply are both -1.0, the intermediate product is represented as +1.0, thus the need for a 33-bit intermediate product prior to sign extension.

Other registers altered: ACC (if A=1)

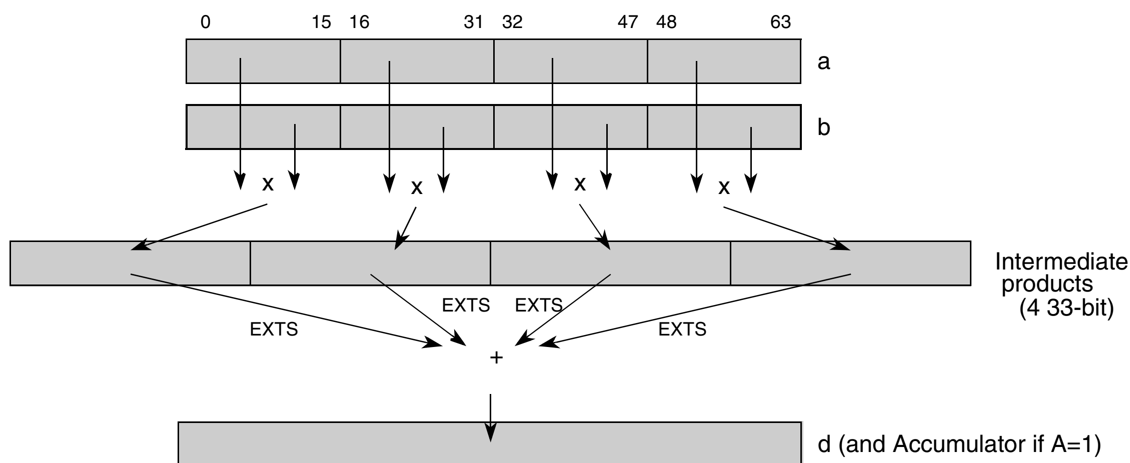


Figure 3-116. Vector Dot Product of Four Half Words, Guarded, Add, Signed, Modulo, Fractional (to Accumulator) (__ev_dotp4hgasmf[a])

A	d	a	b	Maps to
A = 0	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evdotp4hgasmf d,a,b**
A = 1	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evdotp4hgasmfa d,a,b**

__ev_dotp4hgasmfaa __ev_dotp4hgasmfaa

Vector Dot Product of Four Half Words, Guarded, Add, Signed, Modulo, Fractional and Accumulate

d = __ev_dotp4hgasmfaa (a,b)

$$
\begin{aligned}
&temp0_{0:32} \leftarrow a_{0:15} \times_{sf} b_{0:15} \\
&temp1_{0:32} \leftarrow a_{16:31} \times_{sf} b_{16:31} \\
&temp2_{0:32} \leftarrow a_{32:47} \times_{sf} b_{32:47} \\
&temp3_{0:32} \leftarrow a_{48:63} \times_{sf} b_{48:63} \\
&temp_{0:63} \leftarrow EXTS(temp1_{0:31}) + EXTS(temp1_{0:31}) + EXTS(temp2_{0:31}) + EXTS(temp3_{0:31}) + ACC_{0:63} \\
&d_{0:63} \leftarrow temp_{0:63} \\
&ACC_{0:63} \leftarrow d_{0:63}
\end{aligned}
$$

Corresponding half word pairs of signed fractional elements in parameters **a** and **b** are multiplied producing four 33-bit intermediate products. The 33-bit intermediate products are sign-extended and added together with the contents of the accumulator to produce a 64-bit result, and the sum is placed into parameter **d** and the accumulator.

Note: If the two input operands to a multiply are both -1.0, the intermediate product is represented as +1.0, thus the need for a 33-bit intermediate product prior to sign extension.

Other registers altered: ACC

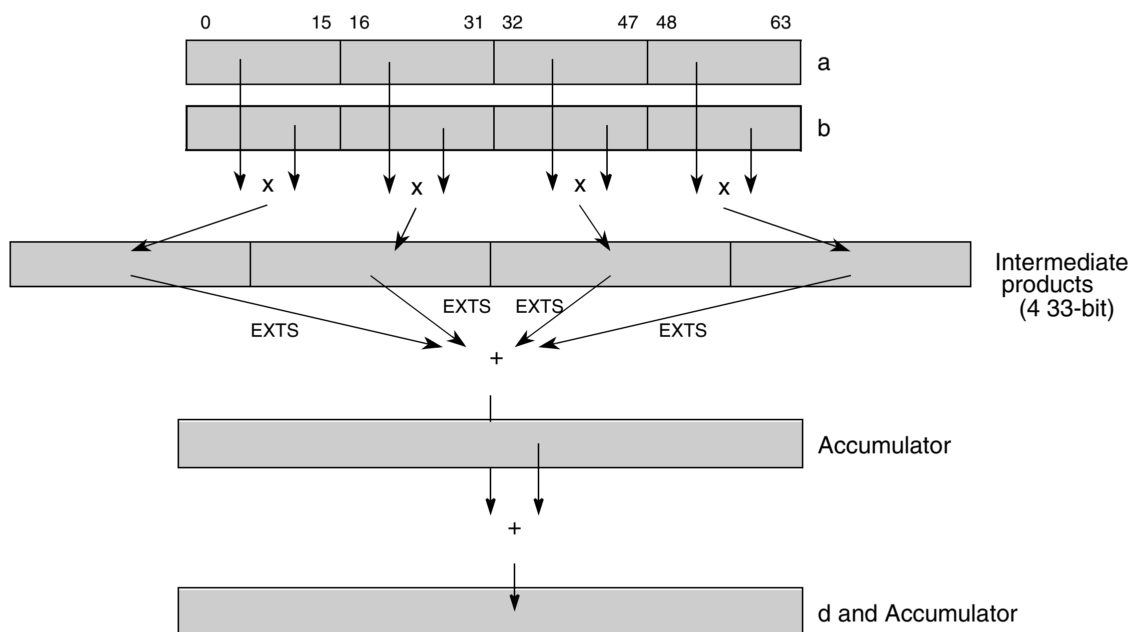


Figure 3-117. Vector Dot Product of Four Half Words, Guarded, Add, Signed, Modulo, Fractional and Accumulate (__ev_dotp4hgasmfaa)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evdotp4hgasmfaa d,a,b**

__ev_dotp4hgasmfaa3 __ev_dotp4hgasmfaa3

Vector Dot Product of Four Halfwords, Guarded, Add, Signed, Modulo, Fractional and Accumulate, 3 operand

d = __ev_dotp4hgasmfaa3 (a,b,c)

$$
\begin{aligned}
&temp0_{0:32} \leftarrow b_{0:15} \times_{sf} c_{0:15} \\
&temp1_{0:32} \leftarrow b_{16:31} \times_{sf} c_{16:31} \\
&temp2_{0:32} \leftarrow b_{32:47} \times_{sf} c_{32:47} \\
&temp3_{0:32} \leftarrow b_{48:63} \times_{sf} c_{48:63} \\
&temp_{0:63} \leftarrow EXTS_{64}(temp1_{0:32}) + EXTS_{64}(temp2_{0:32}) + EXTS_{64}(temp3_{0:32}) + a_{0:63} \\
\\
&d_{0:63} \leftarrow temp_{0:63} \\
&ACC_{0:63} \leftarrow d_{0:63}
\end{aligned}
$$

Corresponding halfword pairs of signed fractional elements in parameters **b** and **c** are multiplied producing four 33-bit intermediate products. The 33-bit intermediate products are sign-extended and added together with the contents of parameter **a** to produce a 64-bit result, and the sum is placed into parameter **d** and the accumulator.

Note: If the two input operands to a multiply are both -1.0, the intermediate product is represented as +1.0, thus the need for a 33-bit intermediate product prior to sign extension.

Other registers altered: ACC

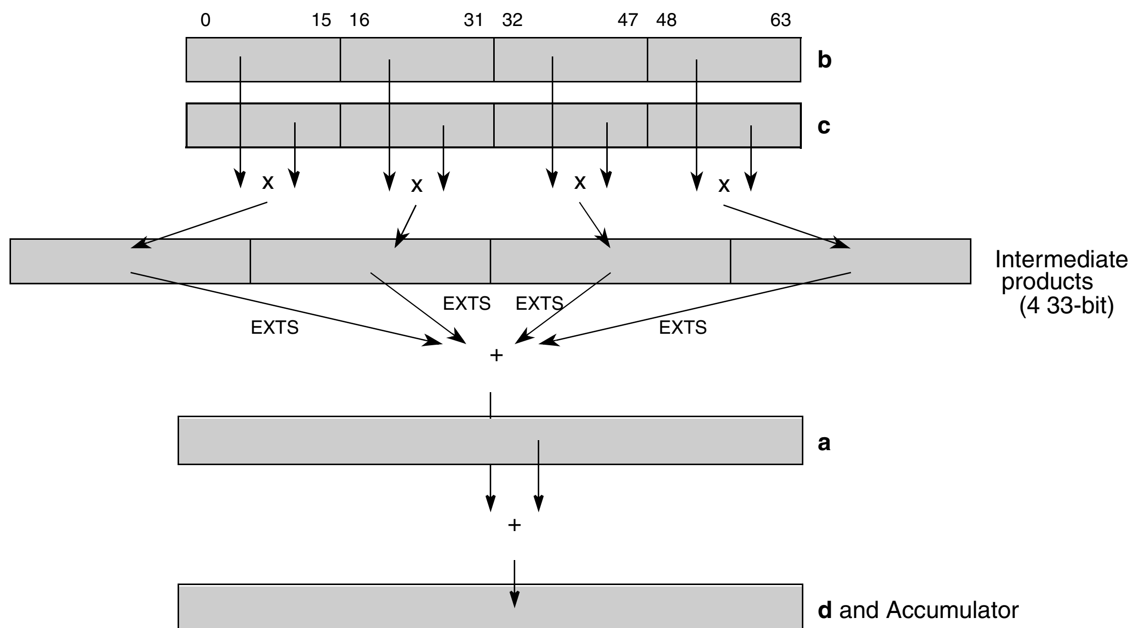


Figure 3-118. Vector Dot Product of Four Halfwords, Guarded, Add, Signed, Modulo, Fractional and Accumulate 3 op (__ev_dotp4hgasmfaa3)

d	a	b	c	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**d ←a** **evdotp4hgasmfaa3 d,b,c**

__ev_dotp4hgasmi[a] __ev_dotp4hgasmi[a]

Vector Dot Product of Four Half Words, Guarded, Add, Signed, Modulo, Integer (to Accumulator)

d = __ev_dotp4hgasmi (a,b) (A = 0)

d = __ev_dotp4hgasmia (a,b) (A = 1)

```
temp0[0:31] ← a[0:15] ×si b[0:15]
temp1[0:31] ← a[16:31] ×si b[16:31]
temp2[0:31] ← a[32:47] ×si b[32:47]
temp3[0:31] ← a[48:63] ×si b[48:63]
temp[0:63] ← EXTS(temp1[0:31]) + EXTS(temp1[0:31]) + EXTS(temp2[0:31]) + EXTS(temp3[0:31]) //modulo
d[0:63] ← temp[0:63]

// update accumulator
if A = 1 then ACC[0:63] ←d[0:63]
```

Corresponding half word pairs of signed integer elements in parameters **a** and **b** are multiplied producing four 32-bit intermediate products. The intermediate 32-bit products are sign-extended and added together to produce a 64-bit result and the sum 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 3-119. Vector Dot Product of Four Half Words, Guarded, Add, Signed, Modulo, Integer (to Accumulator) (__ev_dotp4hgasmi[a])

A	d	a	b	Maps to
A = 0	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evdotp4hgasmi d,a,b**
A = 1	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evdotp4hgasmia d,a,b**

__ev_dotp4hgasmiaa __ev_dotp4hgasmiaa

Vector Dot Product of Four Half Words, Guarded, Add, Signed, Modulo, Integer and Accumulate

d = __ev_dotp4hgasmiaa (a,b)

```
temp0[0:31] ← a[0:15] ×si b[0:15]
temp1[0:31] ← a[16:31] ×si b[16:31]
temp2[0:31] ← a[32:47] ×si b[32:47]
temp3[0:31] ← a[48:63] ×si b[48:63]
temp[0:63] ← EXTS(temp1[0:31]) + EXTS(temp1[0:31]) + EXTS(temp2[0:31]) + EXTS(temp3[0:31]) + ACC[0:63]
d[0:63] ← temp[0:63]

// update accumulator
ACC[0:63] ←d[0:63]
```

Corresponding half word pairs of signed integer elements in paraameter a and parameter **b** are multiplied producing four 32-bit intermediate products. The intermediate 32-bit products are sign-extended and added together with the contents of the accumulator to produce a 64-bit result, and the sum is placed into parameter **d** and the accumulator.

Other registers altered: ACC

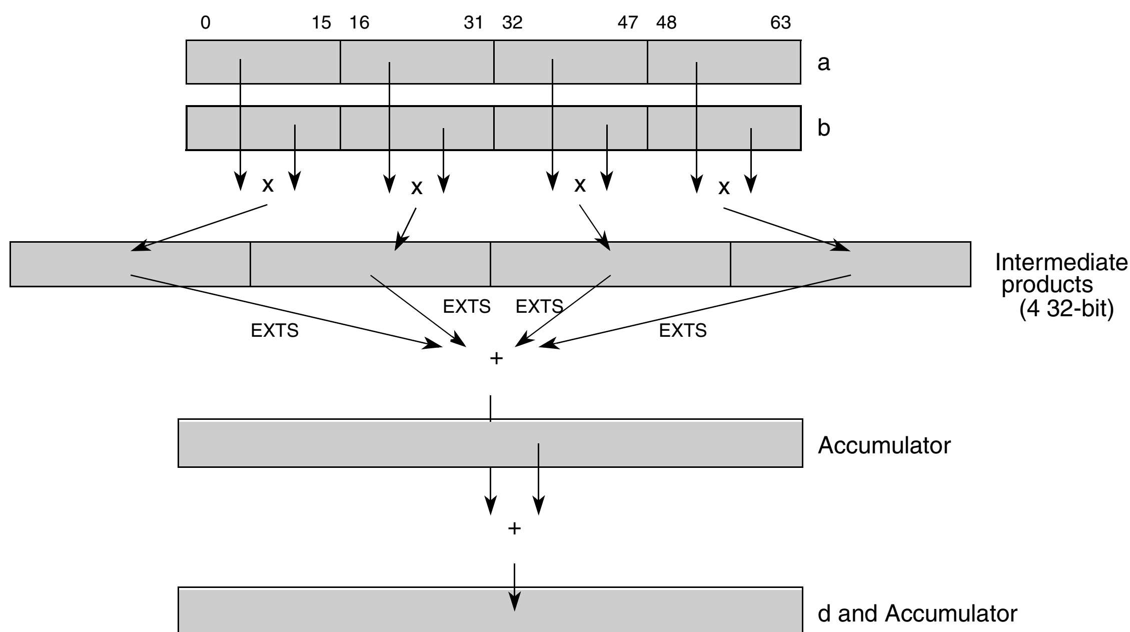


Figure 3-120. Vector Dot Product of Half Words, Guarded, Add, Signed, Modulo, Integer and Accumulate (__ev_dotp4hgasmiaa)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evdotp4hgasmiaa d,a,b**

__ev_dotp4hgasmiaa3 __ev_dotp4hgasmiaa3

Vector Dot Product of Four Halfwords, Guarded, Add, Signed, Modulo, Integer and Accumulate, 3 operand

d = __ev_dotp4hgasmiaa3 (a,b,c)

$$
\begin{aligned}
&temp0_{0:31} \leftarrow b_{0:15} \times_{si} c_{0:15} \\
&temp1_{0:31} \leftarrow b_{16:31} \times_{si} c_{16:31} \\
&temp2_{0:31} \leftarrow b_{32:47} \times_{si} c_{32:47} \\
&temp3_{0:31} \leftarrow b_{48:63} \times_{si} c_{48:63} \\
&temp_{0:63} \leftarrow EXTS_{64}(temp1_{0:31}) + EXTS_{64}(temp1_{0:31}) + EXTS_{64}(temp2_{0:31}) + EXTS_{64}(temp3_{0:31}) + a_{0:63} \\
&d_{0:63} \leftarrow temp_{0:63} \\
&\text{// update accumulator} \\
&ACC_{0:63} \leftarrow d_{0:63}
\end{aligned}
$$

Corresponding halfword pairs of signed integer elements in parameters **b** and **c** are multiplied producing four 32-bit intermediate products. The intermediate 32-bit products are sign-extended and added together with the contents of parameter **a** to produce a 64-bit result, and the sum is placed into parameter **d** and the accumulator.

Other registers altered: ACC

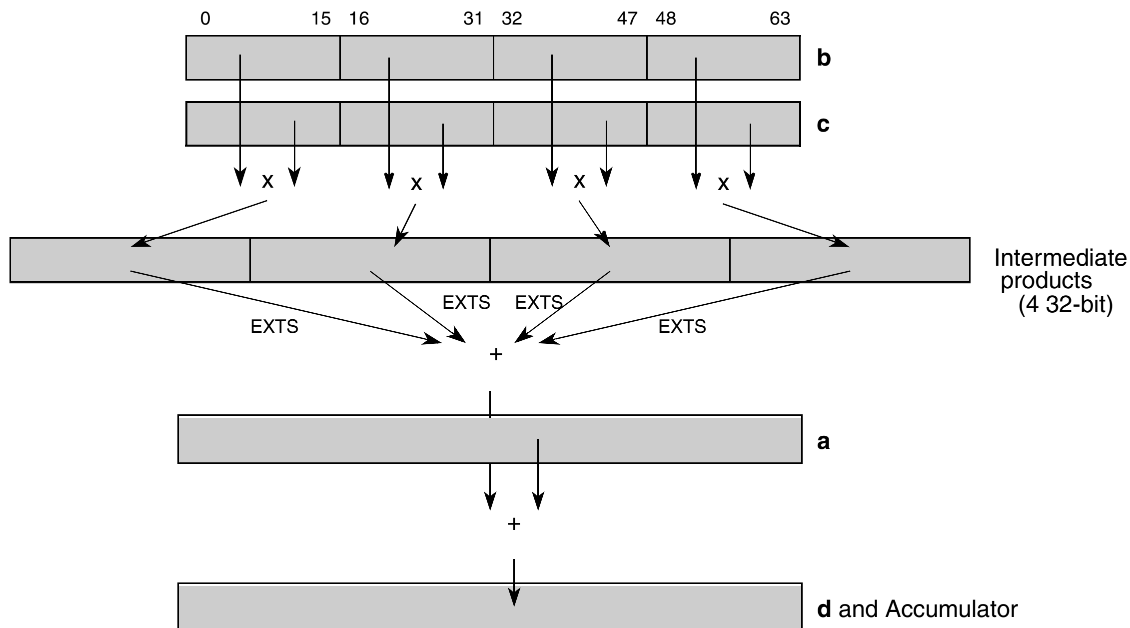


Figure 3-121. Vector Dot Product of Halfwords, Guarded, Add, Signed, Modulo, Integer and Accumulate 3 op (__ev_dotp4hgasmiaa3)

d	a	b	c	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	d ← a **evdotp4hgasmiaa3 d,b,c**

__ev_dotp4hgasumi[a] __ev_dotp4hgasumi[a]

Vector Dot Product of Four Half Words, Guarded, Add, Signed by Unsigned, Modulo, Integer (to Accumulator)

d = __ev_dotp4hgasumi (a,b) (A = 0)

d = __ev_dotp4hgasumia (a,b) (A = 1)

```
temp0[0:31] ← a[0:15] ×sui b[0:15]
temp1[0:31] ← a[16:31] ×sui b[16:31]
temp2[0:31] ← a[32:47] ×sui b[32:47]
temp3[0:31] ← a[48:63] ×sui b[48:63]
temp[0:63] ← EXTS(temp1[0:31]) + EXTS(temp1[0:31]) + EXTS(temp2[0:31]) + EXTS(temp3[0:31]) //modulo
d[0:63] ← temp[0:63]

// update accumulator
if A = 1 then ACC[0:63] ←d[0:63]
```

Corresponding half word pairs of signed integer elements in parameter **a** and unsigned integer elements in parameter **b** are multiplied producing four 32-bit intermediate products. The intermediate 32-bit products are sign-extended and added together to produce a 64-bit result and the sum 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)

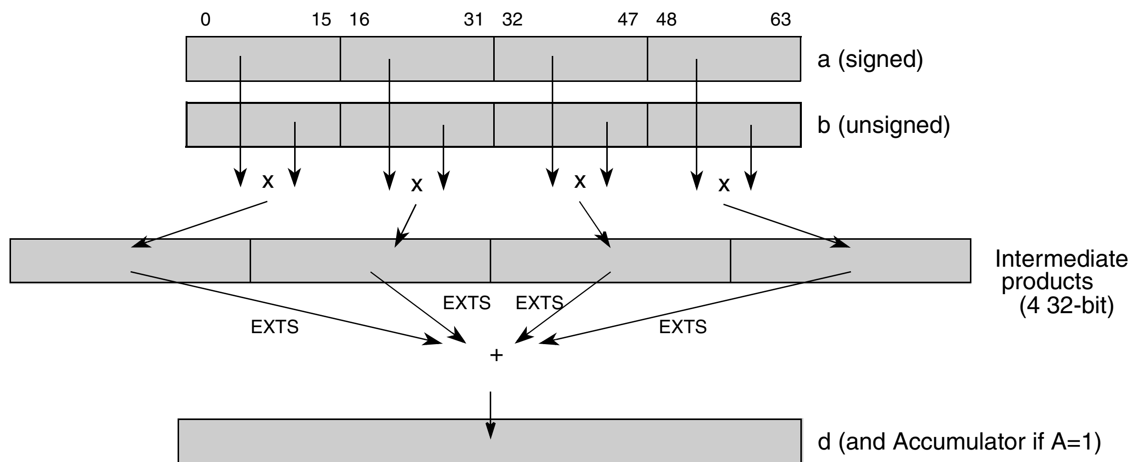


Figure 3-122. Vector Dot Product of Four Half Words, Guarded, Add, Signed by Unsigned, Modulo, Integer (to Accumulator) (__ev_dotp4hgasumi[a])

A	d	a	b	Maps to
A = 0	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evdotp4hgasumi d,a,b**
A = 1	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evdotp4hgasumia d,a,b**

__ev_dotp4hgasumiaa __ev_dotp4hgasumiaa

Vector Dot Product of Four Half Words, Guarded, Add, Signed by Unsigned, Modulo, Integer and Accumulate

d = __ev_dotp4hgasumiaa (a,b)

```
temp0[0:31] ← a[0:15] ×sui b[0:15]
temp1[0:31] ← a[16:31] ×sui b[16:31]
temp2[0:31] ← a[32:47] ×sui b[32:47]
temp3[0:31] ← a[48:63] ×sui b[48:63]
temp[0:63] ← EXTS(temp1[0:31]) + EXTS(temp1[0:31]) + EXTS(temp2[0:31]) + EXTS(temp3[0:31]) + ACC[0:63]
d[0:63] ← temp[0:63]

// update accumulator
ACC[0:63] ←d[0:63]
```

Corresponding half word pairs of signed integer elements in parameter **a** and unsigned integer elements in parameter **b** are multiplied producing four 32-bit intermediate products. The intermediate 32-bit products are sign-extended and added together with the contents of the accumulator to produce a 64-bit result, and the sum is placed into parameter **d** and the accumulator.

Other registers altered: ACC

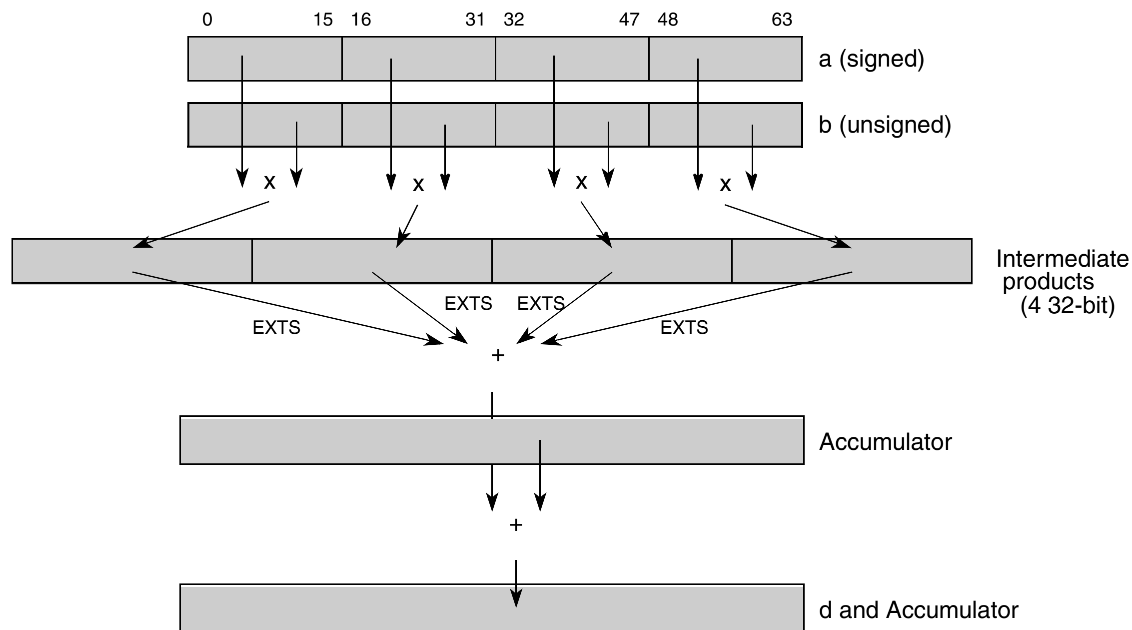


Figure 3-123. Vector Dot Product of Half Words, Guarded, Add, Signed by Unsigned, Modulo, Integer and Accumulate (__ev_dotp4hgasumiaa)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evdotp4hgasumiaa d,a,b**

__ev_dotp4hgasumiaa3 __ev_dotp4hgasumiaa3

Vector Dot Product of Four Halfwords, Guarded, Add, Signed by Unsigned, Modulo, Integer and Accumulate, 3 operand

d = __ev_dotp4hgasumiaa3 (a,b,c)

$$temp0_{0:31} \leftarrow b_{0:15} \times_{sui} c_{0:15}$$
$$temp1_{0:31} \leftarrow b_{16:31} \times_{sui} c_{16:31}$$
$$temp2_{0:31} \leftarrow b_{32:47} \times_{sui} c_{32:47}$$
$$temp3_{0:31} \leftarrow b_{48:63} \times_{sui} c_{48:63}$$
$$temp_{0:63} \leftarrow EXTS_{64}(temp1_{0:31}) + EXTS_{64}(temp1_{0:31}) + EXTS_{64}(temp2_{0:31}) + EXTS_{64}(temp3_{0:31}) + a_{0:63}$$

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

// update accumulator
$$ACC_{0:63} \leftarrow d_{0:63}$$

Corresponding halfword pairs of signed integer elements in parameter **b** and unsigned integer elements in parameter **c** are multiplied producing four 32-bit intermediate products. The intermediate 32-bit products are sign-extended and added together with the contents of parameter **a** to produce a 64-bit result, and the sum is placed into parameter **d** and the accumulator.

Other registers altered: ACC

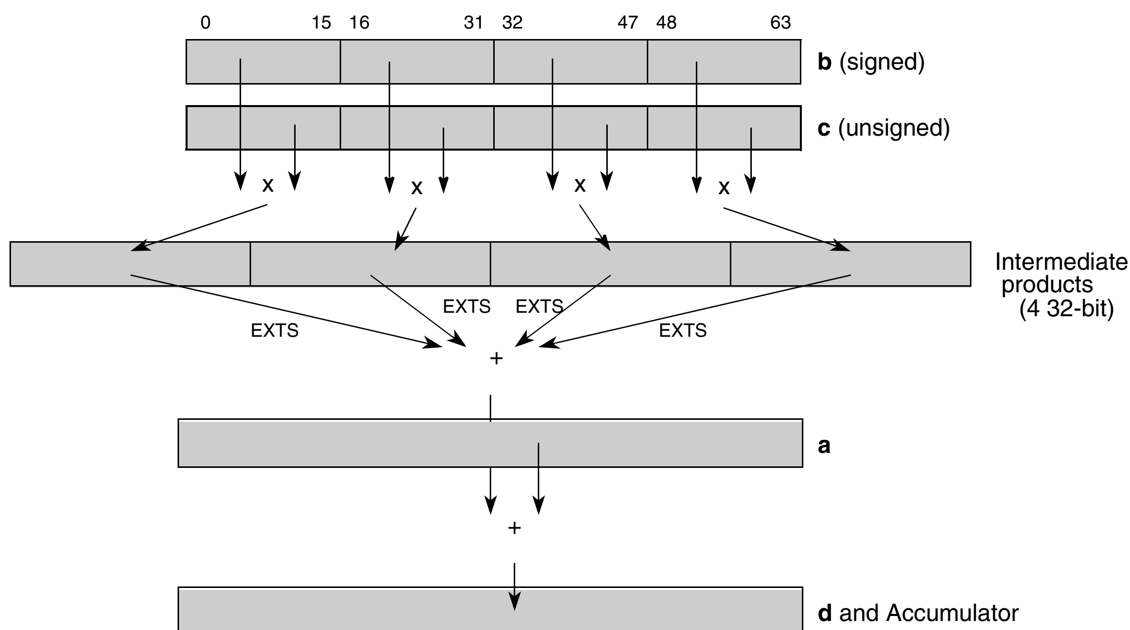


Figure 3-124. Vector Dot Product of Halfwords, Guarded, Add, Signed by Unsigned, Modulo, Integer and Accumulate 3 op (__ev_dotp4hgasumiaa3)

d	a	b	c	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	d ←a evdotp4hgasumiaa3 d,b,c

__ev_dotp4hgaumi[a] __ev_dotp4hgaumi[a]

Vector Dot Product of Four Half Words, Guarded, Add, Unsigned, Modulo, Integer (to Accumulator)

d = __ev_dotp4hgaumi (a,b) (A = 0)

d = __ev_dotp4hgaumia (a,b) (A = 1)

```
temp0[0:31] ← a[0:15] ×ui b[0:15]
temp1[0:31] ← a[16:31] ×ui b[16:31]
temp2[0:31] ← a[32:47] ×ui b[32:47]
temp3[0:31] ← a[48:63] ×ui b[48:63]
temp[0:63] ← EXTZ(temp1[0:31]) + EXTZ(temp1[0:31]) + EXTZ(temp2[0:31]) + EXTZ(temp3[0:31]) //modulo

d[0:63] ← temp[0:63]

// update accumulator
if A = 1 then ACC[0:63] ←d[0:63]
```

Corresponding half word pairs of unsigned integer elements in parameters **a** and **b** are multiplied producing four 32-bit intermediate products. The intermediate 32-bit products are zero-extended and added together to produce a 64-bit result and the sum 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)

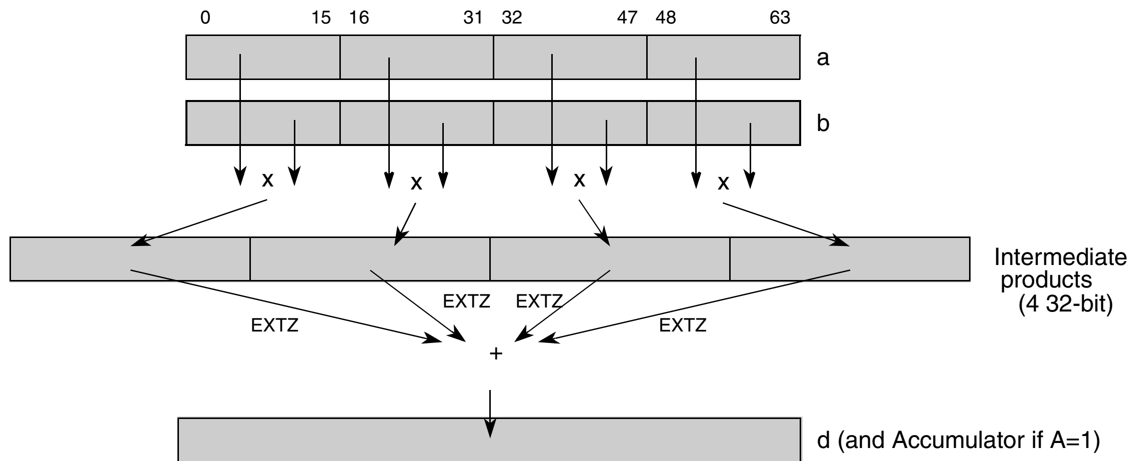


Figure 3-125. Vector Dot Product of Four Half Words, Guarded, Add, Unsigned, Modulo, Integer (to Accumulator) (__ev_dotp4hgaumi[a])

A	d	a	b	Maps to
A = 0	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evdotp4hgaumi d,a,b**
A = 1	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evdotp4hgaumia d,a,b**

__ev_dotp4hgaumiaa __ev_dotp4hgaumiaa

Vector Dot Product of Four Half Words, Guarded, Add, Unsigned, Modulo, Integer and Accumulate

d = __ev_dotp4hgaumiaa (a,b)

```
temp0_{0:31} ← a_{0:15} ×ui b_{0:15}
temp1_{0:31} ← a_{16:31} ×ui b_{16:31}
temp2_{0:31} ← a_{32:47} ×ui b_{32:47}
temp3_{0:31} ← a_{48:63} ×ui b_{48:63}
temp_{0:63} ← EXTZ(temp1_{0:31}) + EXTZ(temp1_{0:31}) + EXTZ(temp2_{0:31}) + EXTZ(temp3_{0:31}) + ACC_{0:63}
d_{0:63} ← temp_{0:63}

// update accumulator
ACC_{0:63} ←d_{0:63}
```

Corresponding half word pairs of unsigned integer elements in parameters **a** and **b** are multiplied producing four 32-bit intermediate products. The intermediate 32-bit products are zero-extended and added together with the contents of the accumulator to produce a 64-bit result, and the sum is placed into parameter **d** and the accumulator.

Other registers altered: ACC

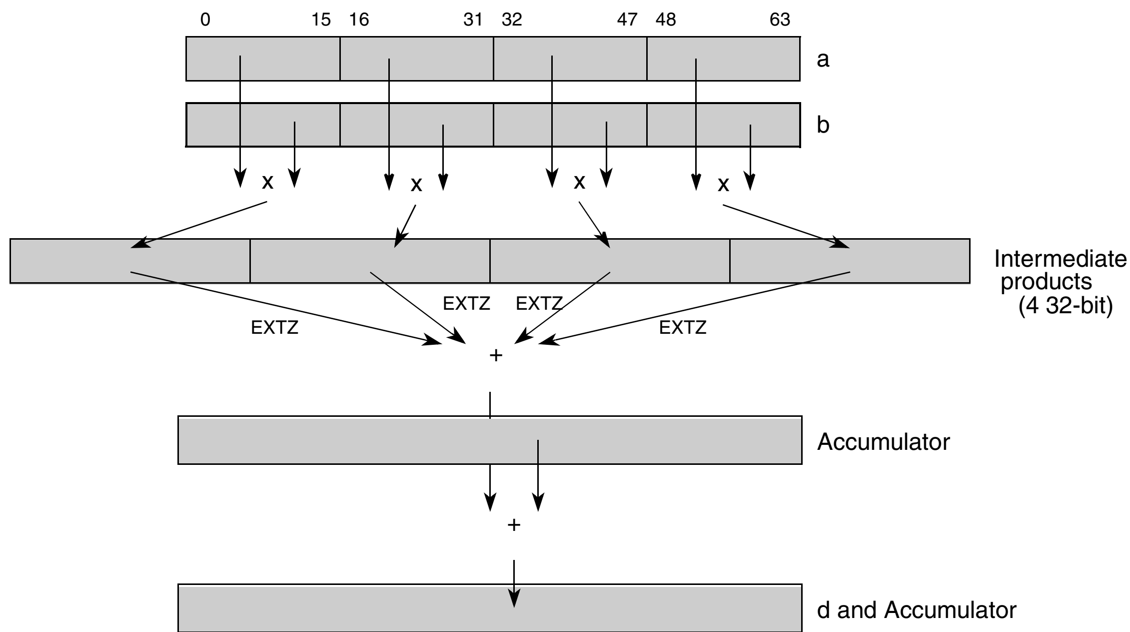


Figure 3-126. Vector Dot Product of Half Words, Guarded, Add, Unsigned, Modulo, Integer and Accumulate (__ev_dotp4hgaumiaa)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evdotp4hgaumiaa d,a,b**

__ev_dotp4hgaumiaa3 __ev_dotp4hgaumiaa3

Vector Dot Product of Four Halfwords, Guarded, Add, Unsigned, Modulo, Integer and Accumulate, 3 operand

d = __ev_dotp4hgaumiaa3 (a,b,c)

```
temp0_{0:31} ← b_{0:15} ×ui c_{0:15}
temp1_{0:31} ← b_{16:31} ×ui c_{16:31}
temp2_{0:31} ← b_{32:47} ×ui c_{32:47}
temp3_{0:31} ← b_{48:63} ×ui c_{48:63}
temp_{0:63} ← EXTZ_64(temp1_{0:31}) + EXTZ_64(temp1_{0:31}) + EXTZ_64(temp2_{0:31}) + EXTZ_64(temp3_{0:31})
            + a_{0:63}
d_{0:63} ← temp_{0:63}

// update accumulator
ACC_{0:63} ←d_{0:63}
```

Corresponding halfword pairs of unsigned integer elements in parameters **b** and **c** are multiplied producing four 32-bit intermediate products. The intermediate 32-bit products are zero-extended and added together with the contents of parameter **a** to produce a 64-bit result, and the sum is placed into parameter **d** and the accumulator.

Other registers altered: ACC

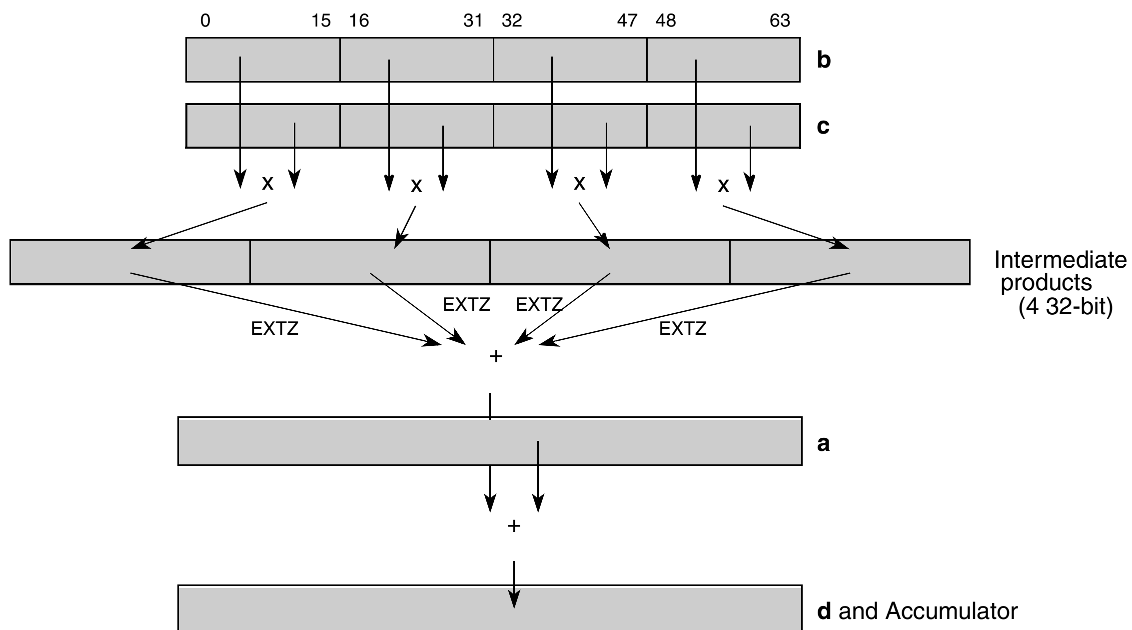


Figure 3-127. Vector Dot Product of Halfwords, Guarded, Add, Unsigned, Modulo, Integer and Accumulate 3 op (__ev_dotp4hgaumiaa3)

d	a	b	c	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	d ←a evdotp4hgaumiaa3 d,b,c

__ev_dotp4hgssmf[a] __ev_dotp4hgssmf[a]

Vector Dot Product of Four Halfwords, Guarded, Subtract, Signed, Modulo, Fractional (to Accumulator)

d = __ev_dotp4hgssmf (a,b) (A = 0)

d = __ev_dotp4hgssmfa (a,b) (A = 1)

```
// high dot - calculate real part of complex product
temph1[0:32] ← a[0:15] ×sf b[0:15]
temph2[0:32] ← a[16:31] ×sf b[16:31]
temph[0:63] ← EXTS64(temph1[0:32]) - EXTS64(temph2[0:32])
//low dot - calculate real part of complex product
templ1[0:32] ← a[32:47] ×si b[32:47]
templ2[0:32] ← a[48:63] ×si b[48:63]
templ[0:63] ← EXTS64(templ1[0:32]) - EXTS64(templ2[0:32])
d[0:63] ← temph[0:63] + templ[0:63]

// update accumulator
if A = 1 then ACC[0:63] ←d[0:63]
```

Halfword pairs of signed fractional elements in the high halfwords of parameter **a** are multiplied with the corresponding high halfwords of parameter **b**, producing a pair of 33-bit products which are sign-extended to 64 bits. The difference of this pair of intermediate products is added to the difference of the corresponding sign-extended 33-bit products of the halfword pairs of signed fractional elements from the low halfwords of parameter **a** and the corresponding low halfwords of parameter **b**, and the sum is placed into parameter **d**. If A = 1, the result in parameter **d** is also placed into the accumulator. This instruction is used to produce the real portion of a guarded complex dot product.

Other registers altered: ACC (if A=1)

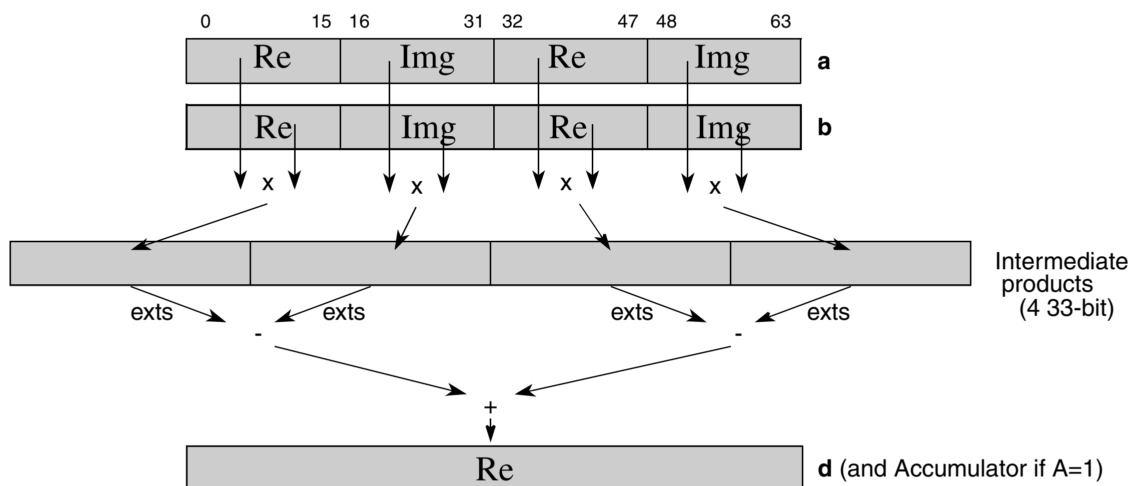


Figure 3-128. Vector Dot Product of Four Halfwords, Guarded, Subtract, Signed, Modulo, Fractional (to Accumulator) (__ev_dotp4hgssmf[a])

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evdotp4hgssmf d,a,b**
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evdotp4hgssmfa d,a,b**

__ev_dotp4hgssmfaa __ev_dotp4hgssmfaa

Vector Dot Product of Four Halfwords, Guarded, Subtract, Signed, Modulo, Fractional and Accumulate

d = __ev_dotp4hgssmfaa (a,b)

```
// high dot - calculate real part of complex product
temph1_{0:32} ← a_{0:15} ×_{sf} b_{0:15}
temph2_{0:32} ← a_{16:31} ×_{sf} b_{16:31}
temph_{0:63} ← EXTS_{64}(temph1_{0:32}) - EXTS_{64}(temph2_{0:32})
//low dot - calculate real part of complex product
templ1_{0:32} ← a_{32:47} ×_{si} b_{32:47}
templ2_{0:32} ← a_{48:63} ×_{si} b_{48:63}
templ_{0:63} ← EXTS_{64}(templ1_{0:32}) - EXTS_{64}(templ2_{0:32})

d_{0:63} ← temph_{0:63} + templ_{0:63} + ACC_{0:63}
ACC_{0:63} ←d_{0:63}
```

Halfword pairs of signed fractional elements in the high halfwords of parameter **a** are multiplied with the corresponding high halfwords of parameter **b**, producing a pair of 33-bit products which are sign-extended to 64 bits. The difference of this pair of intermediate products is added to the difference of the corresponding sign-extended 33-bit products of the halfword pairs of signed fractional elements from the low halfwords of parameter **a** and the corresponding low halfwords of parameter **b**, and then added together with the contents of the accumulator, and the sum is placed into parameter **d** and the accumulator. This instruction is used to produce the real portion of a guarded complex dot product.

Other registers altered: ACC

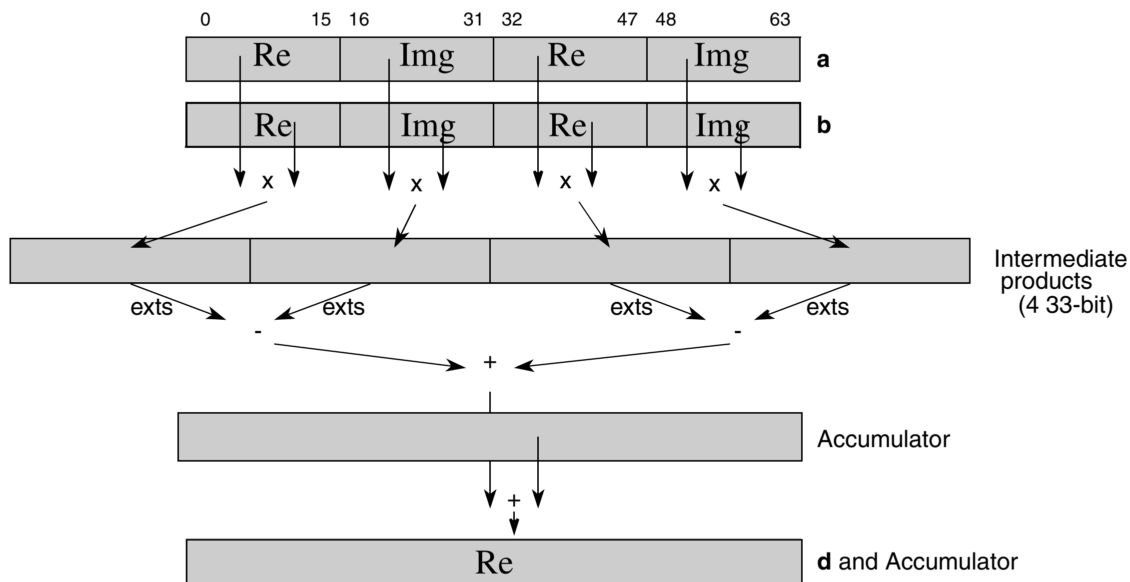


Figure 3-129. Vector Dot Product of Four Halfwords, Guarded, Subtract, Signed, Modulo, Fractional and Accumulate (__ev_dotp4hgssmfaa)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**__ev_dotp4hgssmfaa d,a,b**

__ev_dotp4hgssmfaa3 __ev_dotp4hgssmfaa3

Vector Dot Product of Four Halfwords, Guarded, Subtract, Signed, Modulo, Fractional and Accumulate 3 operand

d = __ev_dotp4hgssmfaa3 (a,b,c)

```
// high dot - calculate real part of complex product
temph1[0:32] ← b[0:15] ×sf c[0:15]
temph2[0:32] ← b[16:31] ×sf c[16:31]
temph[0:63] ← EXTS64(temph1[0:32]) - EXTS64(temph2[0:32])
//low dot - calculate real part of complex product
templ1[0:32] ← b[32:47] ×si c[32:47]
templ2[0:32] ← b[48:63] ×si c[48:63]
templ[0:63] ← EXTS64(templ1[0:32]) - EXTS64(templ2[0:32])

d[0:63] ← temph[0:63] + templ[0:63] + a[0:63]
ACC[0:63] ←d[0:63]
```

Halfword pairs of signed fractional elements in the high halfwords of parameter **b** are multiplied with the corresponding high halfwords of parameter **c**, producing a pair of 33-bit products which are sign-extended to 64 bits. The difference of this pair of intermediate products is added to the difference of the corresponding sign-extended 33-bit products of the halfword pairs of signed fractional elements from the low halfwords of parameter **b** and the corresponding low halfwords of parameter **c**, then added together with the contents of parameter **a** to produce a 64-bit result, and the sum is placed into parameter **d** and the accumulator. This instruction is used to produce the real portion of a guarded complex dot product.

Other registers altered: ACC

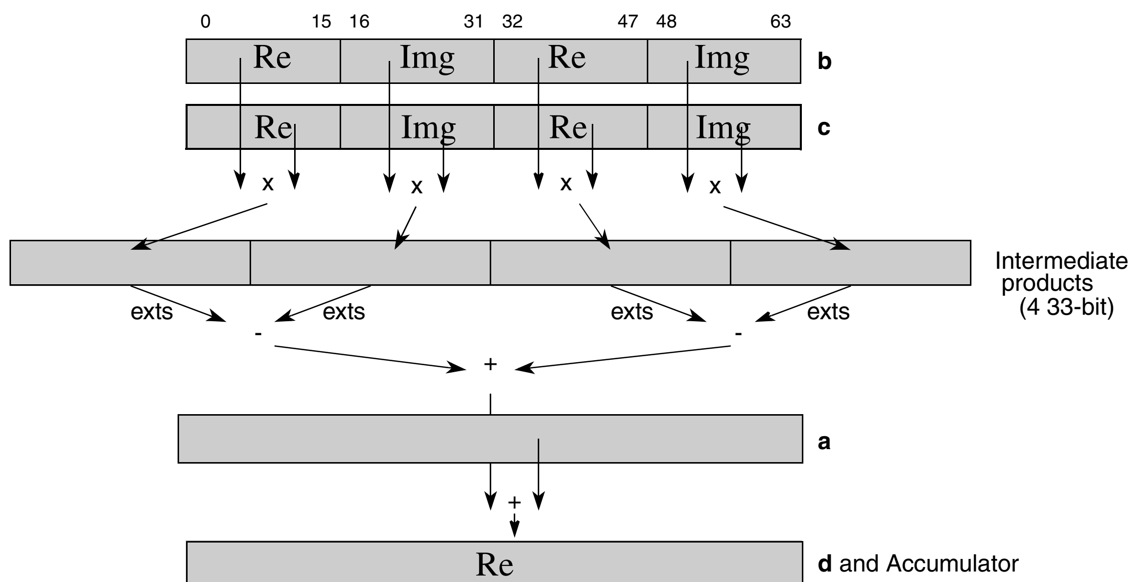


Figure 3-130. Vector Dot Product of Four Halfwords, Guarded, Subtract, Signed, Modulo, Fractional and Accumulate 3op (__ev_dotp4hgssmfaa3)

d	a	b	c	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**d ←a evdotp4hgssmfaa3 d,b,c**

__ev_dotp4hgssmi[a] __ev_dotp4hgssmi[a]

Vector Dot Product of Four Halfwords, Guarded, Subtract, Signed, Modulo, Integer (to Accumulator)

d = __ev_dotp4hgssmi (a,b) (A = 0)

d = __ev_dotp4hgssmia (a,b) (A = 1)

```
// high dot - calculate real part of complex product
temph1_{0:31} ←a_{0:15} ×_{si} b_{0:15}
temph2_{0:31} ←a_{16:31} ×_{si} b_{16:31}
temph_{0:63} ←EXTS_{64}(temph1_{0:31}) - EXTS_{64}(temph2_{0:31})

//low dot - calculate real part of complex product
templ1_{0:31} ←a_{32:47} ×_{si} b_{32:47}
templ2_{0:31} ←a_{48:63} ×_{si} b_{48:63}
templ_{0:63} ←EXTS_{64}(templ1_{0:31}) - EXTS_{64}(templ2_{0:31})
d_{0:63} ←temph_{0:63} + templ_{0:63}

// update accumulator
if A = 1 then ACC_{0:63} ←d_{0:63}
```

Halfword pairs of signed integer elements in the high halfwords of parameter **a** are multiplied with the corresponding high halfwords of parameter **b**, producing a pair of 32-bit products which are sign-extended to 64 bits. The difference of this pair of intermediate products is added to the difference of the corresponding sign-extended products of the halfword pairs of signed integer elements from the low halfwords of parameter **a** and the corresponding low halfwords of parameter **b**, and the sum is placed into parameter **d**. If A = 1, the result in parameter **d** is also placed into the accumulator. This instruction is used to produce the real portion of a guarded complex dot product.

Other registers altered: ACC (if A=1)

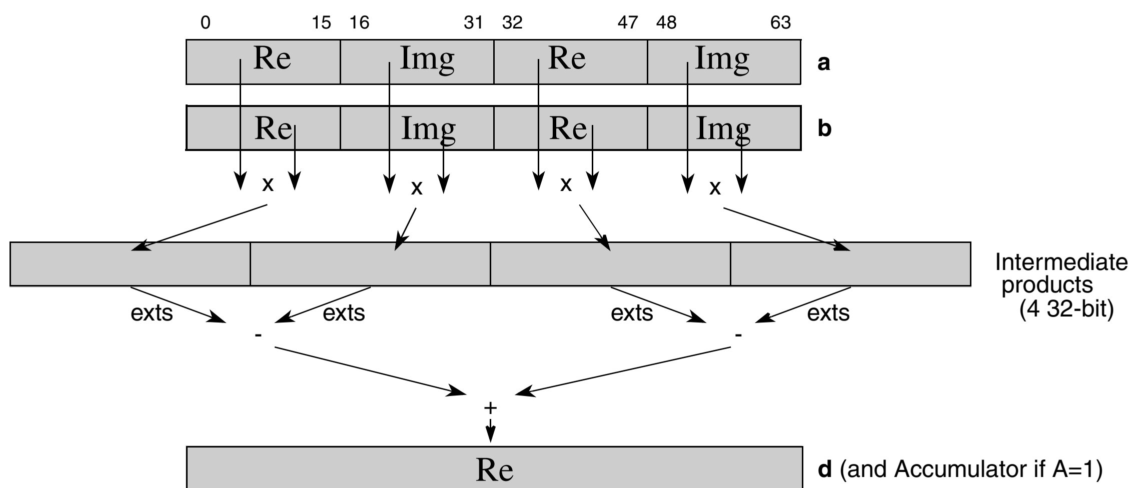


Figure 3-131. Vector Dot Product of Four Halfwords, Guarded, Subtract, Signed, Modulo, Integer (to Accumulator) (__ev_dotp4hgssmi[a])

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evdotp4hgssmi d,a,b**
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evdotp4hgssmia d,a,b**

__ev_dotp4hgssmiaa __ev_dotp4hgssmiaa

Vector Dot Product of Four Halfwords, Guarded, Subtract, Signed, Modulo, Integer and Accumulate

d = __ev_dotp4hgssmiaa (a,b)

```
// high dot - calculate real part of complex product
temph1_0:31 ← a_0:15 ×si b_0:15
temph2_0:31 ← a_16:31 ×si b_16:31
temph_0:63 ← EXTS_64(temph1_0:31) - EXTS_64(temph2_0:31)
//low dot - calculate real part of complex product
templ1_0:31 ← a_32:47 ×si b_32:47
templ2_0:31 ← a_48:63 ×si b_48:63
templ_0:63 ← EXTS_64(templ1_0:31) - EXTS_64(templ2_0:31)

d_0:63 ← temph_0:63 + templ_0:63 + ACC_0:63
ACC_0:63 ←d_0:63
```

Halfword pairs of signed integer elements in the high halfwords of parameter **a** are multiplied with the corresponding high halfwords of parameter **b**, producing a pair of 32-bit products which are sign-extended to 64 bits. The difference of this pair of intermediate products is added to the difference of the corresponding sign-extended products of the halfword pairs of signed integer elements from the low halfwords of parameter **a** and the corresponding low halfwords of parameter **b**, and then added together with the contents of the accumulator, and the sum is placed into parameter **d** and the accumulator. This instruction is used to produce the real portion of a guarded complex dot product.

Other registers altered: ACC

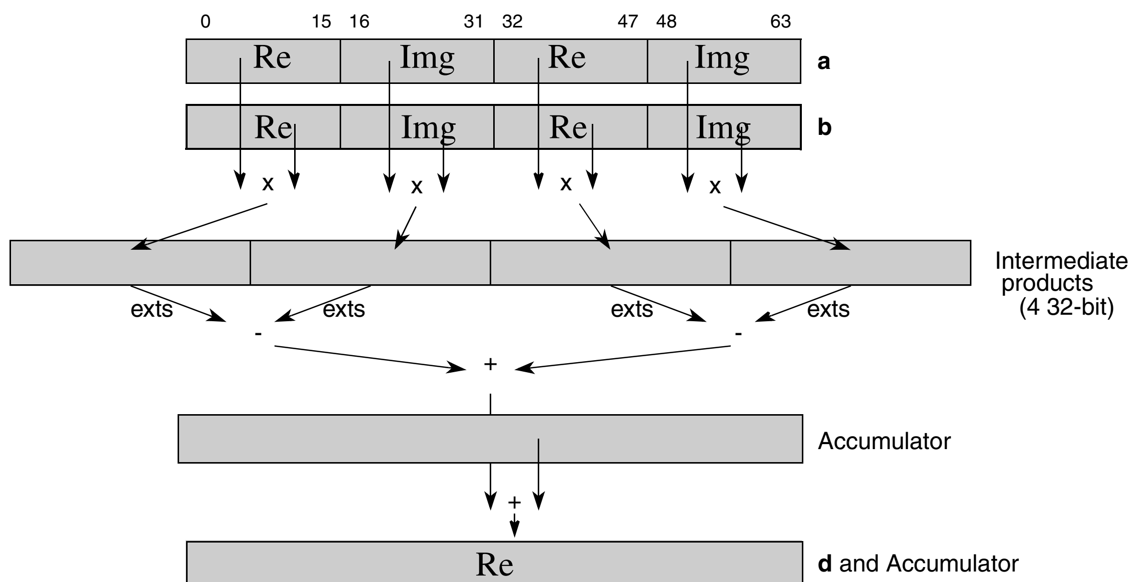


Figure 3-132. Vector Dot Product of Four Halfwords, Guarded, Subtract, Signed, Modulo, Integer and Accumulate (__ev_dotp4hgssmiaa)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evdotp4hgssmiaa d,a,b**

__ev_dotp4hgssmiaa3 __ev_dotp4hgssmiaa3

Vector Dot Product of Four Halfwords, Guarded, Subtract, Signed, Modulo, Integer and Accumulate 3 operand

d = __ev_dotp4hgssmiaa3 (a,b,c)

```
// high dot - calculate real part of complex product
temph1[0:31] ← b[0:15] ×si c[0:15]
temph2[0:31] ← b[16:31] ×si c[16:31]
temph[0:63] ← EXTS64(temph1[0:31]) - EXTS64(temph2[0:31])
//low dot - calculate real part of complex product
templ1[0:31] ← b[32:47] ×si c[32:47]
templ2[0:31] ← b[48:63] ×si c[48:63]
templ[0:63] ← EXTS64(templ1[0:31]) - EXTS64(templ2[0:31])

d[0:63] ← temph[0:63] + templ[0:63] + a[0:63]
ACC[0:63] ←d[0:63]
```

Halfword pairs of signed integer elements in the high halfwords of parameter **b** are multiplied with the corresponding high halfwords of parameter **c**, producing a pair of 32-bit products which are sign-extended to 64 bits. The difference of this pair of intermediate products is added to the difference of the corresponding sign-extended products of the halfword pairs of signed integer elements from the low halfwords of parameter **b** and the corresponding low halfwords of parameter **c**, then added together with the contents of parameter **a** to produce a 64-bit result, and the sum is placed into parameter **d** and the accumulator. This instruction is used to produce the real portion of a guarded complex dot product.

Other registers altered: ACC

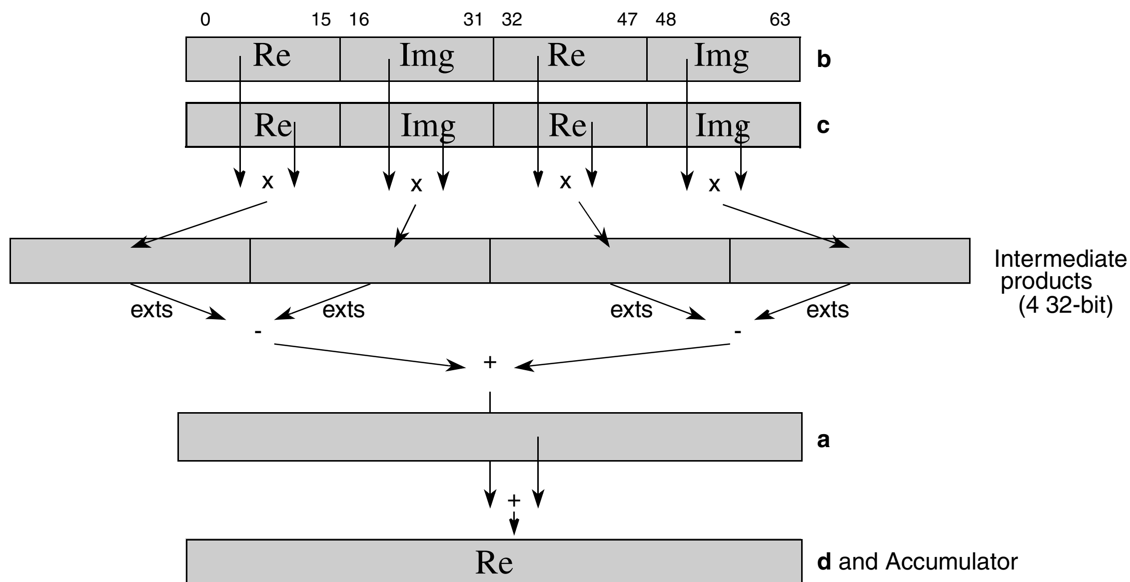


Figure 3-133. Vector Dot Product of Four Halfwords, Guarded, Subtract, Signed, Modulo, Integer and Accumulate 3op (__ev_dotp4hgssmiaa3)

d	a	b	c	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**d ←a** **evdotp4hgssmiaa3 d,b,c**

__ev_dotp4hxgasmf[a] __ev_dotp4hxgasmf[a]

Vector Dot Product of Four Halfwords Exchanged, Guarded, Add, Signed, Modulo, Fractional (to Accumulator)

d = __ev_dotp4hxgasmf (a,b) (A = 0)

d = __ev_dotp4hxgasmfa (a,b) (A = 1)

```
// high dot - calculate imag part of complex product
temph1_{0:32} ← b_{0:15} ×sf a_{16:31}
temph2_{0:32} ← b_{16:31} ×sf a_{0:15}
temph_{0:63} ← EXTS_{64}(temph1_{0:32}) + EXTS_{64}(temph2_{0:32})
//low dot - calculate imag part of complex product
templ1_{0:32} ← b_{32:47} ×sf a_{48:63}
templ2_{0:32} ← b_{48:63} ×sf a_{32:47}
templ_{0:63} ← EXTS_{64}(templ1_{0:32}) + EXTS_{64}(templ2_{0:32})

d_{0:63} ← temph_{0:63} + templ_{0:63}
// update accumulator
if A = 1 then ACC_{0:63} ←d_{0:63}
```

Pairs of signed fractional elements in the high halfwords of parameter **b** are multiplied with the exchanged high halfwords of parameter **a**, producing a pair of 33-bit products which are sign-extended to 64 bits. The sum of this pair of intermediate products is added to the sum of the corresponding sign-extended 33-bit products of the halfword pairs of signed fractional elements from the low halfwords of parameter **b** and the exchanged low halfwords of parameter **a** to produce a 64-bit result, and the sum is placed into parameter **d**. If A = 1, the result in parameter **d** is also placed into the accumulator. This instruction is used to produce the imaginary portion of a guarded complex dot product. Note: If the two input operands to a multiply are both -1.0, the intermediate product is represented as +1.0 ($^{1}0$ || 0x8000_0000).

Other registers altered: ACC (if A=1)

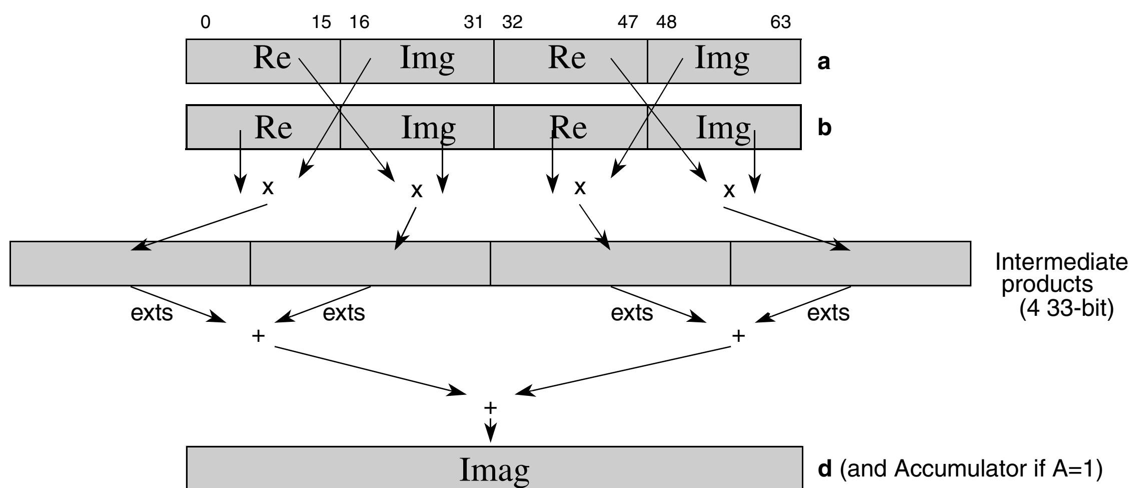


Figure 3-134. Vector Dot Product of Four Halfwords Exchanged, Guarded, Add, Signed, Modulo, Fractional (to Accumulator) (__ev_dotp4hxgasmf[a])

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evdotp4hxgasmf d,a,b**
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evdotp4hxgasmfa d,a,b**

__ev_dotp4hxgasmfaa __ev_dotp4hxgasmfaa

Vector Dot Product of Four Halfwords Exchanged, Guarded, Add, Signed, Modulo, Fractional and Accumulate

d = __ev_dotp4hxgasmfaa (a,b)

```
// high dot - calculate imag part of complex product
temph1_0:32 ← b_0:15 ×sf a_16:31
temph2_0:32 ← b_16:31 ×sf a_0:15
temph_0:63 ← EXTS_64(temph1_0:32) + EXTS_64(temph2_0:32)
//low dot - calculate imag part of complex product
templ1_0:32 ← b_32:47 ×sf a_48:63
templ2_0:32 ← b_48:63 ×sf a_32:47
templ_0:63 ← EXTS_64(templ1_0:32) + EXTS_64(templ2_0:32)

d_0:63 ← temph_0:63 + templ_0:63 + ACC_0:63
ACC_0:63 ←d_0:63
```

Pairs of signed fractional elements in the high halfwords of parameter **b** are multiplied with the exchanged high halfwords of parameter **a**, producing a pair of 33-bit products which are sign-extended to 64 bits. The sum of this pair of intermediate products is added to the sum of the corresponding sign-extended 33-bit products of the halfword pairs of signed fractional elements from the low halfwords of parameter **b** and the exchanged low halfwords of parameter **a**, then added together with the contents of the accumulator to produce a 64-bit result, and the sum is placed into parameter **d** and the accumulator. This instruction is used to produce the imaginary portion of a guarded complex dot product. Note: If the two input operands to a multiply are both -1.0, the intermediate product is represented as +1.0 ($^{1}0$ || 0x8000_0000).

Other registers altered: ACC

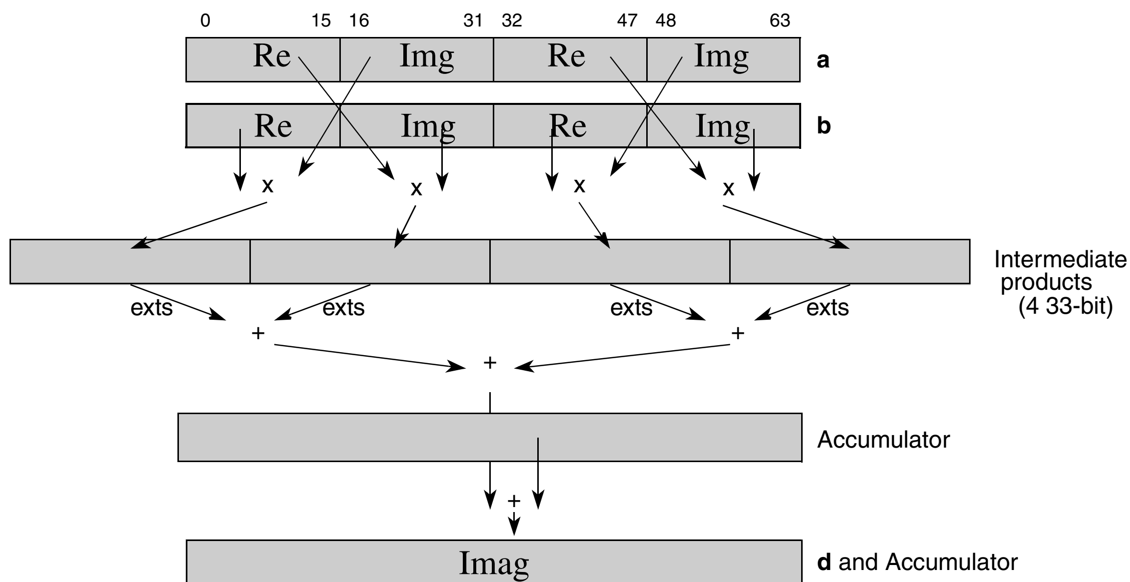


Figure 3-135. Vector Dot Product of Four Halfwords Exchanged, Guarded, Add, Signed, Modulo, Fractional and Accumulate (__ev_dotp4hxgasmfaa)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evdotp4hxgasmfaa d,a,b**

__ev_dotp4hxgasmfaa3 __ev_dotp4hxgasmfaa3

Vector Dot Product of Four Halfwords Exchanged, Guarded, Add, Signed, Modulo, Fractional and Accumulate 3 operand

d = __ev_dotp4hxgasmfaa3 (a,b,c)

```
// high dot - calculate imag part of complex product
temph1[0:32] ← c[0:15] ×sf b[16:31]
temph2[0:32] ← c[16:31] ×sf b[0:15]
temph[0:63] ← EXTS64(temph1[0:32]) + EXTS64(temph2[0:32])
//low dot - calculate imag part of complex product
templ1[0:32] ← c[32:47] ×sf b[48:63]
templ2[0:32] ← c[48:63] ×sf b[32:47]
templ[0:63] ← EXTS64(templ1[0:32]) + EXTS64(templ2[0:32])

d[0:63] ← temph[0:63] + templ[0:63] + a[0:63]
ACC[0:63] ← d[0:63]
```

Pairs of signed fractional elements in the high halfwords of parameter **c** are multiplied with the exchanged high halfwords of parameter **b**, producing a pair of 33-bit products which are sign-extended to 64 bits. The sum of this pair of intermediate products is added to the sum of the corresponding sign-extended 33-bit products of the halfword pairs of signed fractional elements from the low halfwords of parameter **c** and the exchanged low halfwords of parameter **b**, then added together with the contents of parameter **a** to produce a 64-bit result, and the sum is placed into parameter **d** and the accumulator. This instruction is used to produce the imaginary portion of a guarded complex dot product. Note: If the two input operands to a multiply are both -1.0, the intermediate product is represented as +1.0 ($^{1}0 \| 0x8000_0000$).

Other registers altered: ACC

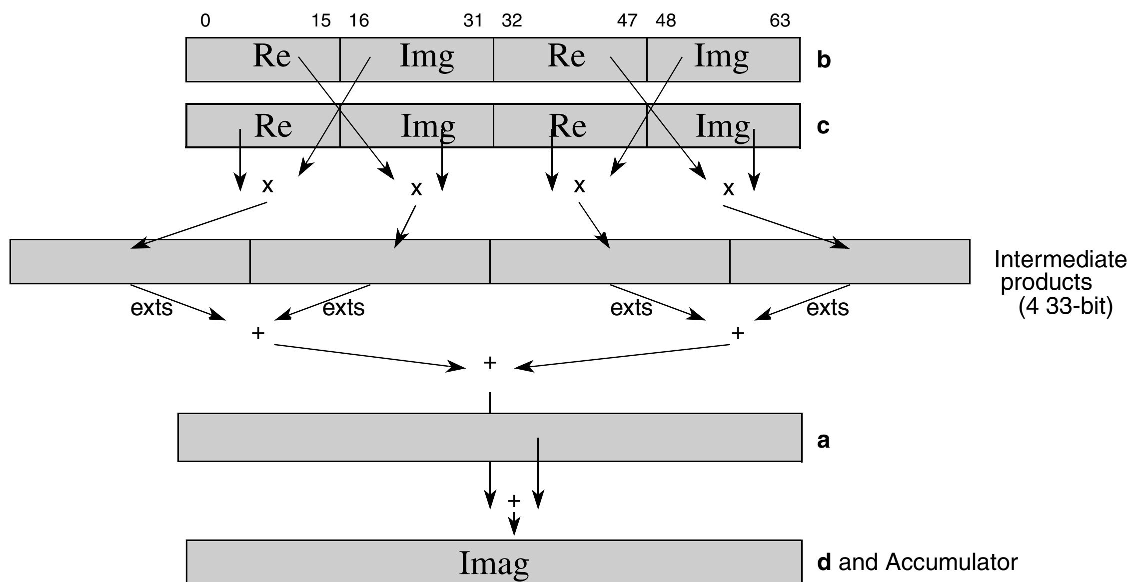


Figure 3-136. Vector Dot Product of Four Halfwords Exchanged, Guarded, Add, Signed, Modulo, Fractional and Accumulate 3 op (__ev_dotp4hxgasmfaa3)

d	a	b	c	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**d ←a evdotp4hxgasmfaa3 d,b,c**

__ev_dotp4hxgasmi[a] __ev_dotp4hxgasmi[a]

Vector Dot Product of Four Halfwords Exchanged, Guarded, Add, Signed, Modulo, Integer (to Accumulator)

d = __ev_dotp4hxgasmi (a,b) (A = 0)

d = __ev_dotp4hxgasmia (a,b) (A = 1)

```
// high dot - calculate imag part of complex product
temph1[0:31] ← b[0:15] ×si a[16:31]
temph2[0:31] ← b[16:31] ×si a[0:15]
temph[0:63] ← EXTS64(temph1[0:31]) + EXTS64(temph2[0:31])

//low dot - calculate imag part of complex product
templ1[0:31] ← b[32:47] ×si a[48:63]
templ2[0:31] ← b[48:63] ×si a[32:47]
templ[0:63] ← EXTS64(templ1[0:31]) + EXTS64(templ2[0:31])
d[0:63] ← temph[0:63] + templ[0:63]

// update accumulator
if A = 1 then ACC[0:63] ←d[0:63]
```

Halfword pairs of signed integer elements in the high halfwords of parameter **b** are multiplied with the exchanged high halfwords of parameter **a**, producing a pair of 32-bit products which are sign-extended to 64 bits. The sum of this pair of intermediate products is added to the sum of the corresponding sign-extended products of the halfword pairs of signed integer elements from the low halfwords of parameter **b** and the exchanged low halfwords of parameter **a** to produce a 64-bit result, and the sum is placed into parameter **d**. If A = 1, the result in parameter **d** is also placed into the accumulator. This instruction is used to produce the imaginary portion of a guarded complex dot product.

Other registers altered: ACC (if A=1)

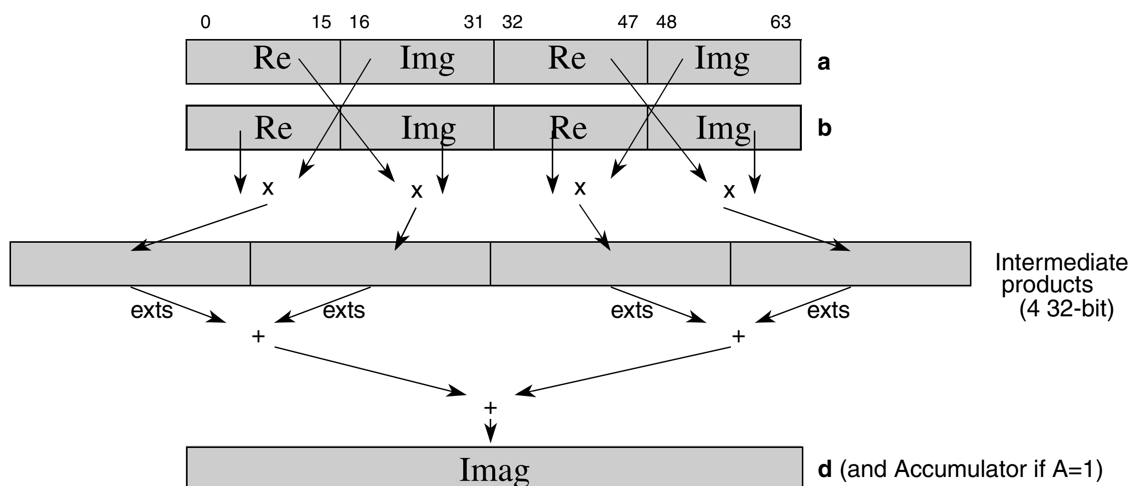


Figure 3-137. Vector Dot Product of Four Halfwords Exchanged, Guarded, Add, Signed, Modulo, Integer (to Accumulator) (__ev_dotp4hxgasmi[a])

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evdotp4hxgasmi d,a,b**
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evdotp4hxgasmia d,a,b**

__ev_dotp4hxgasmiaa __ev_dotp4hxgasmiaa

Vector Dot Product of Four Halfwords Exchanged, Guarded, Add, Signed, Modulo, Integer and Accumulate

d = __ev_dotp4hxgasmiaa (a,b)

```
// high dot - calculate imag part of complex product
temph1[0:31] ← b[0:15] ×si a[16:31]
temph2[0:31] ← b[16:31] ×si a[0:15]
temph[0:63] ← EXTS64(temph1[0:31]) + EXTS64(temph2[0:31])

//low dot - calculate imag part of complex product
templ1[0:31] ← b[32:47] ×si a[48:63]
templ2[0:31] ← b[48:63] ×si a[32:47]
templ[0:63] ← EXTS64(templ1[0:31]) + EXTS64(templ2[0:31])

d[0:63] ← temph[0:63] + templ[0:63] + ACC[0:63]
ACC[0:63] ←d[0:63]
```

Halfword pairs of signed integer elements in the high halfwords of parameter **b** are multiplied with the exchanged high halfwords of parameter **a**, producing a pair of 32-bit products which are sign-extended to 64 bits. The sum of this pair of intermediate products is added to the sum of the corresponding sign-extended products of the halfword pairs of signed integer elements from the low halfwords of parameter **b** and the exchanged low halfwords of parameter **a**, then added together with the contents of the accumulator to produce a 64-bit result, and the sum is placed into parameter **d** and the accumulator. This instruction is used to produce the real portion of a guarded complex dot product.

Other registers altered: ACC

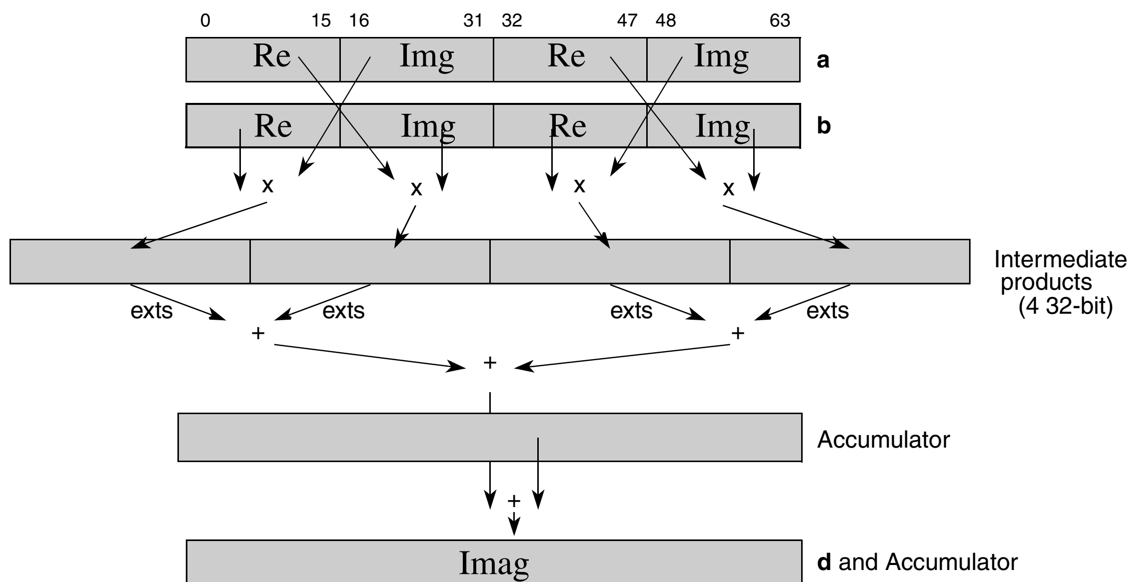


Figure 3-138. Vector Dot Product of Four Halfwords Exchanged, Guarded, Add, Signed, Modulo, Integer and Accumulate (__ev_dotp4hxgasmiaa)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evdotp4hxgasmiaa d,a,b**

__ev_dotp4hxgasmiaa3 __ev_dotp4hxgasmiaa3

Vector Dot Product of Four Halfwords Exchanged, Guarded, Add, Signed, Modulo, Integer and Accumulate 3 operand

d = __ev_dotp4hxgasmiaa3 (a,b,c)

```
// high dot - calculate imag part of complex product
temph1_{0:31} ← c_{0:15} ×_{si} b_{16:31}
temph2_{0:31} ← c_{16:31} ×_{si} b_{0:15}
temph_{0:63} ← EXTS_{64}(temph1_{0:31}) + EXTS_{64}(temph2_{0:31})
//low dot - calculate imag part of complex product
templ1_{0:31} ← c_{32:47} ×_{si} b_{48:63}
templ2_{0:31} ← c_{48:63} ×_{si} b_{32:47}
templ_{0:63} ← EXTS_{64}(templ1_{0:31}) + EXTS_{64}(templ2_{0:31})

d_{0:63} ← temph_{0:63} + templ_{0:63} + a_{0:63}
ACC_{0:63} ← d_{0:63}
```

Halfword pairs of signed integer elements in the high halfwords of parameter **c** are multiplied with the exchanged high halfwords of parameter **b**, producing a pair of 32-bit products which are sign-extended to 64 bits. The sum of this pair of intermediate products is added to the sum of the corresponding sign-extended products of the halfword pairs of signed integer elements from the low halfwords of parameter **c** and the exchanged low halfwords of parameter **b**, then added together with the contents of parameter **a** to produce a 64-bit result, and the sum is placed into parameter **d** and the accumulator. This instruction is used to produce the real portion of a guarded complex dot product.

Other registers altered: ACC

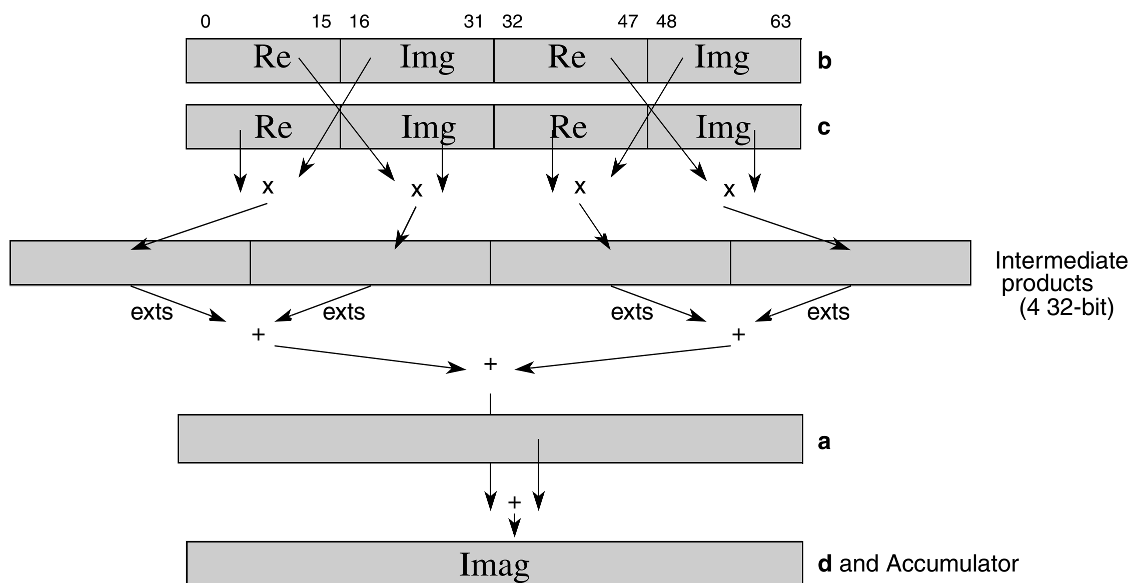


Figure 3-139. Vector Dot Product of Four Halfwords Exchanged, Guarded, Add, Signed, Modulo, Integer and Accumulate 3 op (__ev_dotp4hxgasmiaa3)

d	a	b	c	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**d ←a evdotp4hxgasmiaa3 d,b,c**

__ev_dotp4hxgssmf[a] __ev_dotp4hxgssmf[a]

Vector Dot Product of Four Halfwords Exchanged, Guarded, Subtract, Signed, Modulo, Fractional (to Accumulator)

d = __ev_dotp4hxgssmf (a,b) (A = 0)

d = __ev_dotp4hxgssmfa (a,b) (A = 1)

```
// high dot - calculate imag part of complex x complex conjugate product
temph1[0:32] ← b[0:15] ×sf a[16:31]
temph2[0:32] ← b[16:31] ×sf a[0:15]
temph[0:63] ← EXTS64(temph1[0:32]) - EXTS64(temph2[0:32])
//low dot - calculate imag part of complex x complex conjugate product
templ1[0:32] ← b[32:47] ×sf a[48:63]
templ2[0:32] ← b[48:63] ×sf a[32:47]
templ[0:63] ← EXTS64(templ1[0:32]) - EXTS64(templ2[0:32])

d[0:63] ← temph[0:63] + templ[0:63]
// update accumulator
if A = 1 then ACC[0:63] ←d[0:63]
```

Pairs of signed fractional elements in the high halfwords of parameter **b** are multiplied with the exchanged high halfwords of parameter **a**, producing a pair of 33-bit products which are sign-extended to 64 bits. The difference of this pair of intermediate products is added to the difference of the corresponding sign-extended 33-bit products of the halfword pairs of signed fractional elements from the low halfwords of parameter **b** and the exchanged low halfwords of parameter **a** to produce a 64-bit result, and the sum is placed into parameter **d**. If A = 1, the result in parameter **d** is also placed into the accumulator. This instruction is used to produce the imaginary portion of a guarded complex x complex conjugate dot product. If the two input operands to a multiply are both -1.0, the intermediate product is represented as +1.0 ($^{1}0$ || 0x8000_0000).

Other registers altered: ACC (if A=1)

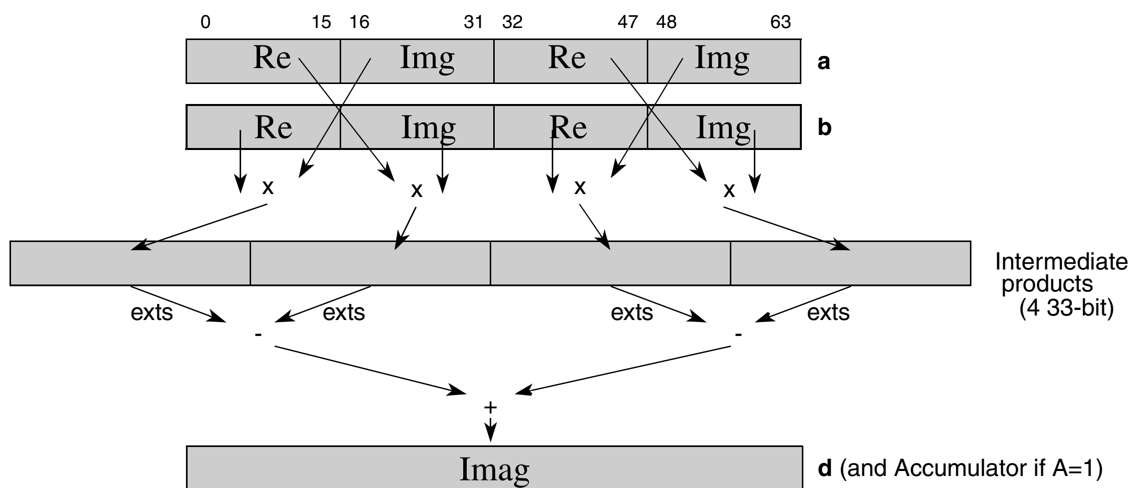


Figure 3-140. Vector Dot Product of Four Halfwords Exchanged, Guarded, Subtract, Signed, Modulo, Fractional (to Accumulator) (__ev_dotp4hxgssmf[a])

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evdotp4hxgssmf d,a,b**
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evdotp4hxgssmfa d,a,b**

__ev_dotp4hxgssmfaa __ev_dotp4hxgssmfaa

Vector Dot Product of Four Halfwords Exchanged, Guarded, Subtract, Signed, Modulo, Fractional and Accumulate

d = __ev_dotp4hxgssmfaa (a,b)

```
// high dot - calculate imag part of complex x complex conjugate product
temph1_{0:32} ← b_{0:15} ×sf a_{16:31}
temph2_{0:32} ← b_{16:31} ×sf a_{0:15}
temph_{0:63} ← EXTS_{64}(temph1_{0:32}) - EXTS_{64}(temph2_{0:32})
//low dot - calculate imag part of complex x complex conjugate product
templ1_{0:32} ← b_{32:47} ×sf a_{48:63}
templ2_{0:32} ← b_{48:63} ×sf a_{32:47}
templ_{0:63} ← EXTS_{64}(templ1_{0:32}) - EXTS_{64}(templ2_{0:32})

d_{0:63} ← temph_{0:63} + templ_{0:63} + ACC_{0:63}
ACC_{0:63} ←d_{0:63}
```

Pairs of signed fractional elements in the high halfwords of parameter **b** are multiplied with the exchanged high halfwords of parameter **a**, producing a pair of 33-bit products which are sign-extended to 64 bits. The difference of this pair of intermediate products is added to the difference of the corresponding sign-extended 33-bit products of the halfword pairs of signed fractional elements from the low halfwords of parameter **b** and the exchanged low halfwords of parameter **a**, then added together with the contents of the accumulator to produce a 64-bit result, and the sum is placed into parameter **d** and the accumulator. This instruction is used to produce the imaginary portion of a guarded complex x complex conjugate dot product. If the two input operands to a multiply are both -1.0, the intermediate product is represented as +1.0 ($^{1}0$ || 0x8000_0000).

Other registers altered: ACC

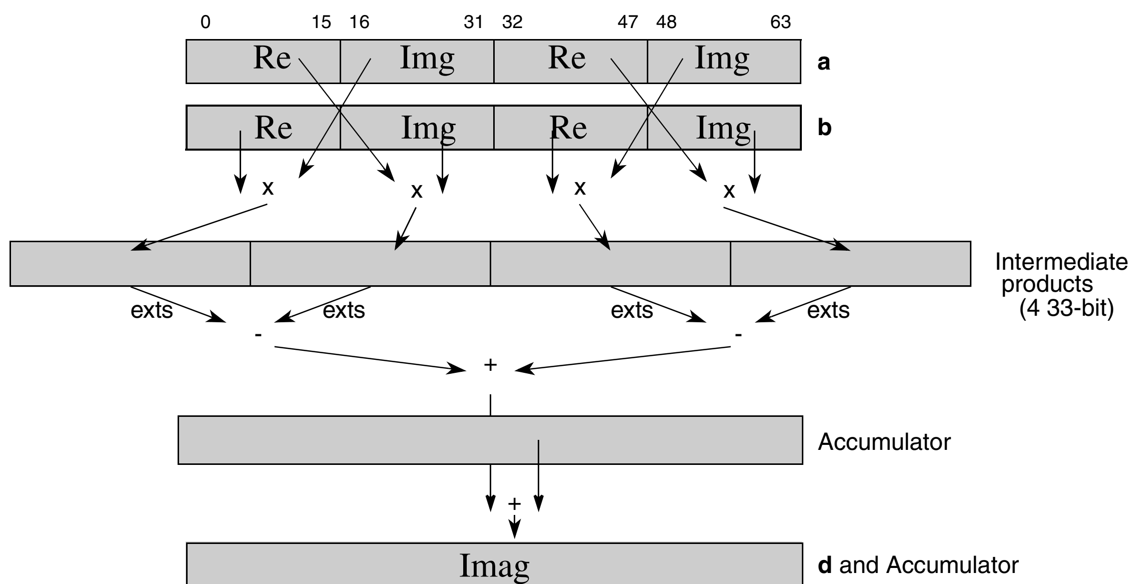


Figure 3-141. Vector Dot Product of Four Halfwords Exchanged, Guarded, Subtract, Signed, Modulo, Fractional and Accumulate (__ev_dotp4hxgssmfaa)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evdotp4hxgssmfaa d,a,b**

__ev_dotp4hxgssmfaa3 __ev_dotp4hxgssmfaa3

Vector Dot Product of Four Halfwords Exchanged, Guarded, Subtract, Signed, Modulo, Fractional and Accumulate 3 operand

d = __ev_dotp4hxgssmfaa3 (a,b,c)

```
// high dot - calculate imag part of complex x complex conjugate product
temph1_{0:32} ← c_{0:15} ×_{sf} b_{16:31}
temph2_{0:32} ← c_{16:31} ×_{sf} b_{0:15}
temph_{0:63} ← EXTS_{64}(temph1_{0:32}) - EXTS_{64}(temph2_{0:32})
//low dot - calculate imag part of complex x complex conjugate product
templ1_{0:32} ← c_{32:47} ×_{sf} b_{48:63}
templ2_{0:32} ← c_{48:63} ×_{sf} b_{32:47}
templ_{0:63} ← EXTS_{64}(templ1_{0:32}) - EXTS_{64}(templ2_{0:32})

d_{0:63} ← temph_{0:63} + templ_{0:63} + a_{0:63}
ACC_{0:63} ←d_{0:63}
```

Pairs of signed fractional elements in the high halfwords of parameter **c** are multiplied with the exchanged high halfwords of parameter **b**, producing a pair of 33-bit products which are sign-extended to 64 bits. The difference of this pair of intermediate products is added to the difference of the corresponding sign-extended 33-bit products of the halfword pairs of signed fractional elements from the low halfwords of parameter **c** and the exchanged low halfwords of parameter **b**, then added together with the contents of parameter **a** to produce a 64-bit result, and the sum is placed into parameter **d** and the accumulator. This instruction is used to produce the imaginary portion of a guarded complex x complex conjugate dot product. Note: If the two input operands to a multiply are both -1.0, the intermediate product is represented as +1.0 ($^{1}0$ || 0x8000_0000).

Other registers altered: ACC

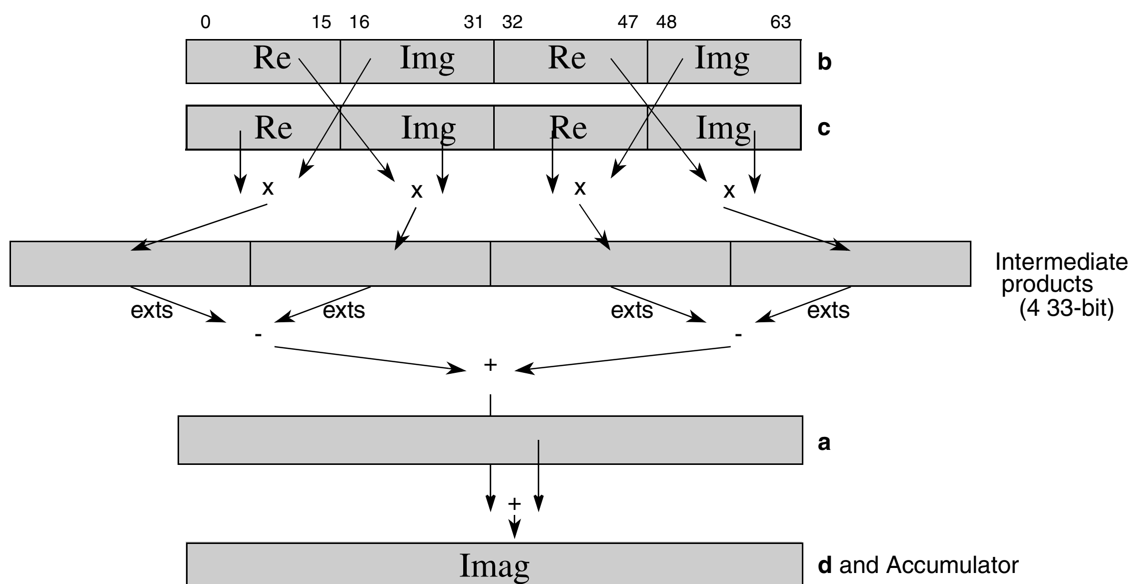


Figure 3-142. Vector Dot Product of Four Halfwords Exchanged, Guarded, Subtract, Signed, Modulo, Fractional and Accumulate 3 op (__ev_dotp4hxgssmfaa3)

d	a	b	c	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**d ←a evdotp4hxgssmfaa3 d,b,c**

__ev_dotp4hxgssmi[a] __ev_dotp4hxgssmi[a]

Vector Dot Product of Four Halfwords Exchanged, Guarded, Subtract, Signed, Modulo, Integer (to Accumulator)

d = __ev_dotp4hxgssmi (a,b) (A = 0)

d = __ev_dotp4hxgssmia (a,b) (A = 1)

```
// high dot - calculate imag part of complex x complex conjugate product
temph1[0:31] ← b[0:15] ×si a[16:31]
temph2[0:31] ← b[16:31] ×si a[0:15]
temph[0:63] ← EXTS64(temph1[0:31]) - EXTS64(temph2[0:31])

//low dot - calculate imag part of complex x complex conjugate product
templ1[0:31] ← b[32:47] ×si a[48:63]
templ2[0:31] ← b[48:63] ×si a[32:47]
templ[0:63] ← EXTS64(templ1[0:31]) - EXTS64(templ2[0:31])
d[0:63] ← temph[0:63] + templ[0:63]

// update accumulator
if A = 1 then ACC[0:63] ←d[0:63]
```

Halfword pairs of signed integer elements in the high halfwords of parameter **b** are multiplied with the exchanged high halfwords of parameter **a**, producing a pair of 32-bit products which are sign-extended to 64 bits. The difference of this pair of intermediate products is added to the difference of the corresponding sign-extended products of the halfword pairs of signed integer elements from the low halfwords of parameter **b** and the exchanged low halfwords of parameter **a** to produce a 64-bit result, and the sum is placed into parameter **d**. If A = 1, the result in parameter **d** is also placed into the accumulator. This instruction is used to produce the imaginary portion of a guarded complex x complex conjugate dot product.

Other registers altered: ACC (if A=1)

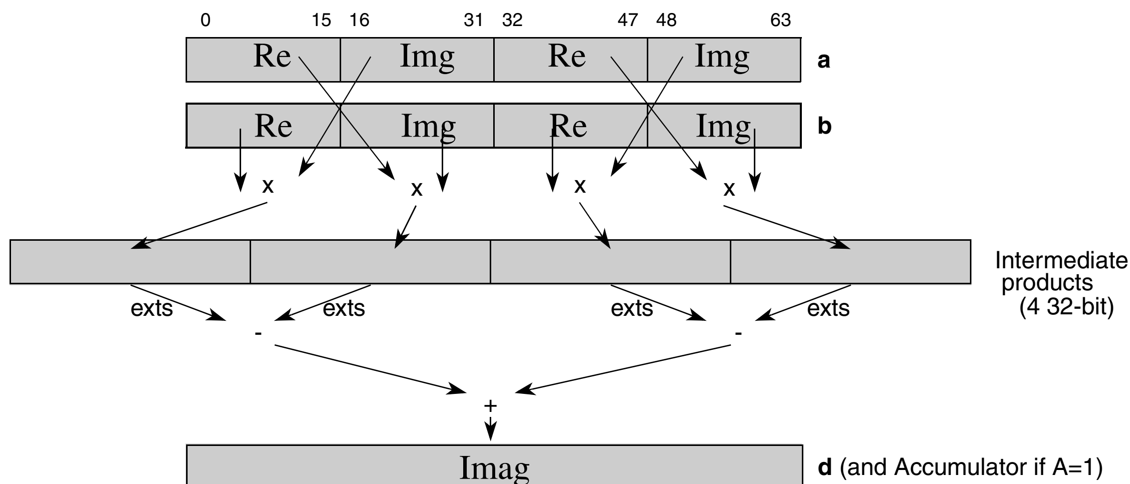


Figure 3-143. Vector Dot Product of Four Halfwords Exchanged, Guarded, Subtract, Signed, Modulo, Integer (to Accumulator) (__ev_dotp4hxgssmi[a])

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evdotp4hxgssmi d,a,b**
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evdotp4hxgssmia d,a,b**

__ev_dotp4hxgssmiaa __ev_dotp4hxgssmiaa

Vector Dot Product of Four Halfwords Exchanged, Guarded, Subtract, Signed, Modulo, Integer and Accumulate

d = __ev_dotp4hxgssmiaa (a,b)

```
// high dot - calculate imag part of complex x complex conjugate product
temph1[0:31] ← b[0:15] ×si a[16:31]
temph2[0:31] ← b[16:31] ×si a[0:15]
temph[0:63] ← EXTS64(temph1[0:31]) - EXTS64(temph2[0:31])

//low dot - calculate imag part of complex x complex conjugate product
templ1[0:31] ← b[32:47] ×si a[48:63]
templ2[0:31] ← b[48:63] ×si a[32:47]
templ[0:63] ← EXTS64(templ1[0:31]) - EXTS64(templ2[0:31])

d[0:63] ← temph[0:63] + templ[0:63] + ACC[0:63]
ACC[0:63] ←d[0:63]
```

Halfword pairs of signed integer elements in the high halfwords of parameter **b** are multiplied with the exchanged high halfwords of parameter **a**, producing a pair of 32-bit products which are sign-extended to 64 bits. The difference of this pair of intermediate products is added to the difference of the corresponding sign-extended products of the halfword pairs of signed integer elements from the low halfwords of parameter **b** and the exchanged low halfwords of parameter **a**, then added together with the contents of the accumulator to produce a 64-bit result, and the sum is placed into parameter **d** and the accumulator. This instruction is used to produce the real portion of a guarded complex x complex conjugate dot product.

Other registers altered: ACC

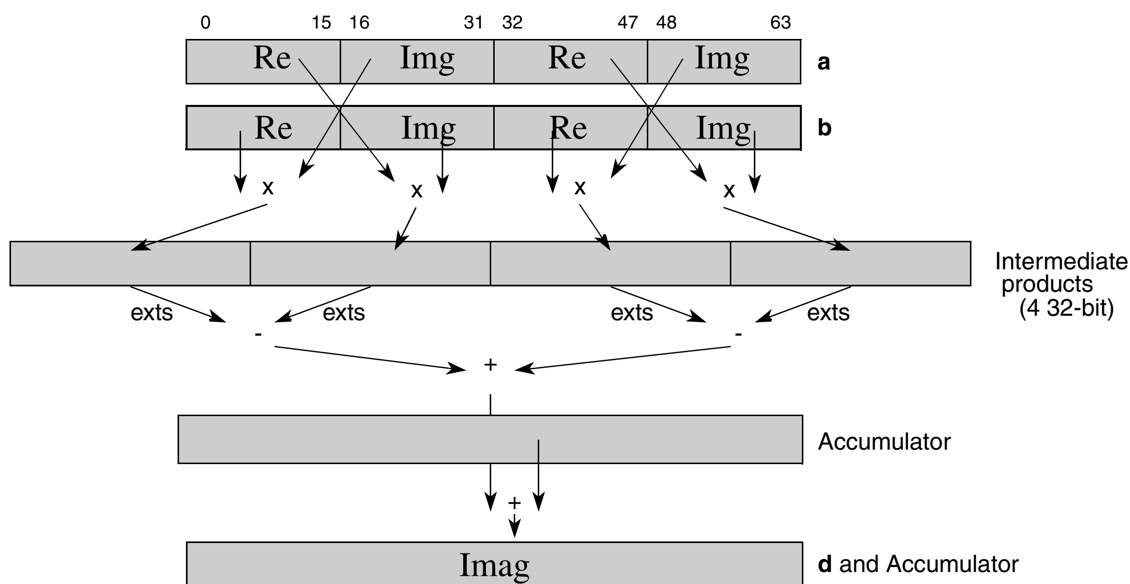


Figure 3-144. Vector Dot Product of Four Halfwords Exchanged, Guarded, Subtract, Signed, Modulo, Integer and Accumulate (__ev_dotp4hxgssmiaa)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evdotp4hxgssmiaa d,a,b**

__ev_dotp4hxgssmiaa3 __ev_dotp4hxgssmiaa3

Vector Dot Product of Four Halfwords Exchanged, Guarded, Subtract, Signed, Modulo, Integer and Accumulate 3 operand

d = __ev_dotp4hxgssmiaa3 (a,b,c)

```
// high dot - calculate imag part of complex x complex conjugate product
temph1[0:31] ← c[0:15] ×si b[16:31]
temph2[0:31] ← c[16:31] ×si b[0:15]
temph[0:63] ← EXTS64(temph1[0:31]) - EXTS64(temph2[0:31])
//low dot - calculate imag part of complex x complex conjugate product
templ1[0:31] ← c[32:47] ×si b[48:63]
templ2[0:31] ← c[48:63] ×si b[32:47]
templ[0:63] ← EXTS64(templ1[0:31]) - EXTS64(templ2[0:31])

d[0:63] ← temph[0:63] + templ[0:63] + a[0:63]
ACC[0:63] ←d[0:63]
```

Halfword pairs of signed integer elements in the high halfwords of parameter **c** are multiplied with the exchanged high halfwords of parameter **b**, producing a pair of 32-bit products which are sign-extended to 64 bits. The difference of this pair of intermediate products is added to the difference of the corresponding sign-extended products of the halfword pairs of signed integer elements from the low halfwords of parameter **c** and the exchanged low halfwords of parameter **b**, then added together with the contents of parameter **a** to produce a 64-bit result, and the sum is placed into parameter **d** and the accumulator. This instruction is used to produce the real portion of a guarded complex x complex conjugate dot product.

Other registers altered: ACC

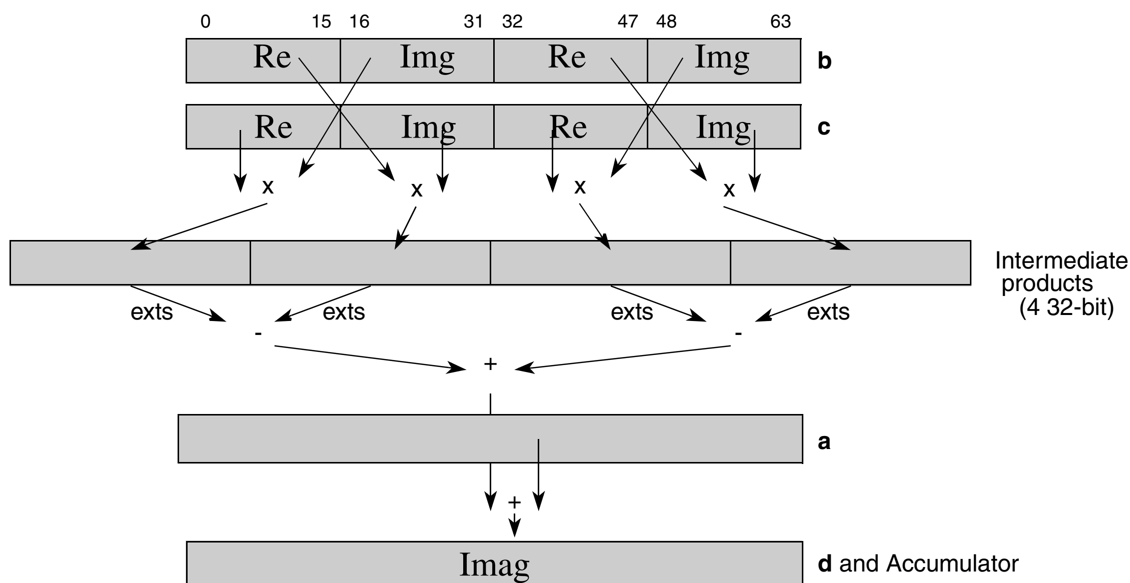


Figure 3-145. Vector Dot Product of Four Halfwords Exchanged, Guarded, Subtract, Signed, Modulo, Integer and Accumulate 3 op (__ev_dotp4hxgssmiaa3)

d	a	b	c	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**d ←a evdotp4hxgssmiaa3 d,b,c**

__ev_dotpbasmi[a] __ev_dotpbasmi[a]

Vector Dot Product of Bytes, Add, Signed, Modulo, Integer (to Accumulator)

d = __ev_dotpbasmi (a,b) (A = 0)

d = __ev_dotpbasmia (a,b) (A = 1)

```
// high dot
temph0:31 ←EXTS(a0:7 ×si b0:7) + EXTS(a8:15 ×si b8:15) + EXTS(a16:23 ×si b16:23) +
          EXTS(a24:31 ×si b24:31)
d0:31 ←temph0:31

//low
templ0:31 ←EXTS(a32:39 ×si b32:39) + EXTS(a40:47 ×si b40:47) + EXTS(a48:55 ×si b48:55)
          + EXTS(a56:63 ×si b56:63)
d32:63 ←templ0:31

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

For each word element in the destination, corresponding byte pairs of signed integer elements in parameters **a** and **b** are multiplied producing four 16-bit products. This quad of intermediate products are sign-extended to 32 bits and added together and the sum is placed into the corresponding parameter **d** word. If A = 1, the result in parameter **d** is also placed into the accumulator.

Other registers altered: ACC (if A=1)

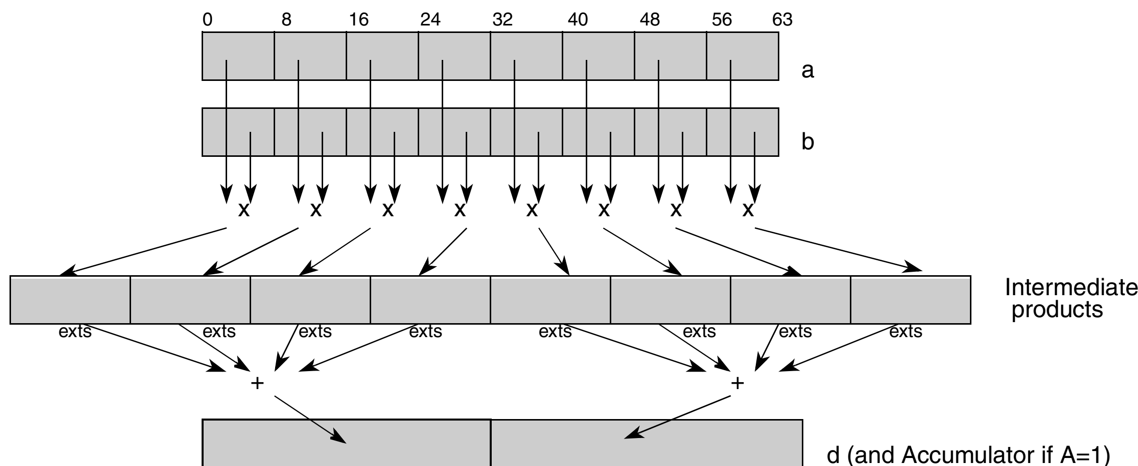


Figure 3-146. Vector Dot Product of Bytes, Add, Signed, Modulo, Integer (to Accumulator) (__ev_dotpbasmi[a])

A	d	a	b	Maps to
A = 0	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evdotpbasmi d,a,b**
A = 1	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evdotpbasmia d,a,b**

__ev_dotpbasmiaaw __ev_dotpbasmiaaw

Vector Dot Product of Bytes, Add, Signed, Modulo, Integer and Accumulate into Words

d = __ev_dotpbasmiaaw (a,b)

```
// high dot
temph0:31 ←EXTS(a0:7 ×si b0:7) + EXTS(a8:15 ×si b8:15) + EXTS(a16:23 ×si b16:23) +
          EXTS(a24:31 ×si b24:31) + ACC0:31
d0:31 ←temph0:31

//low
templ0:31 ←EXTS(a32:39 ×si b32:39) + EXTS(a40:47 ×si b40:47) + EXTS(a48:55 ×si b48:55)
          + EXTS(a56:63 ×si b56:63) + ACC32:63
d32:63 ←templ0:31

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

For each word element in the destination, corresponding byte pairs of signed integer elements in parameters **a** and **b** are multiplied producing four 16-bit products. This quad of intermediate products are sign-extended to 32 bits and added together with the contents of the corresponding accumulator word and the sum is placed into the corresponding parameter **d** and accumulator word.

Other registers altered: ACC

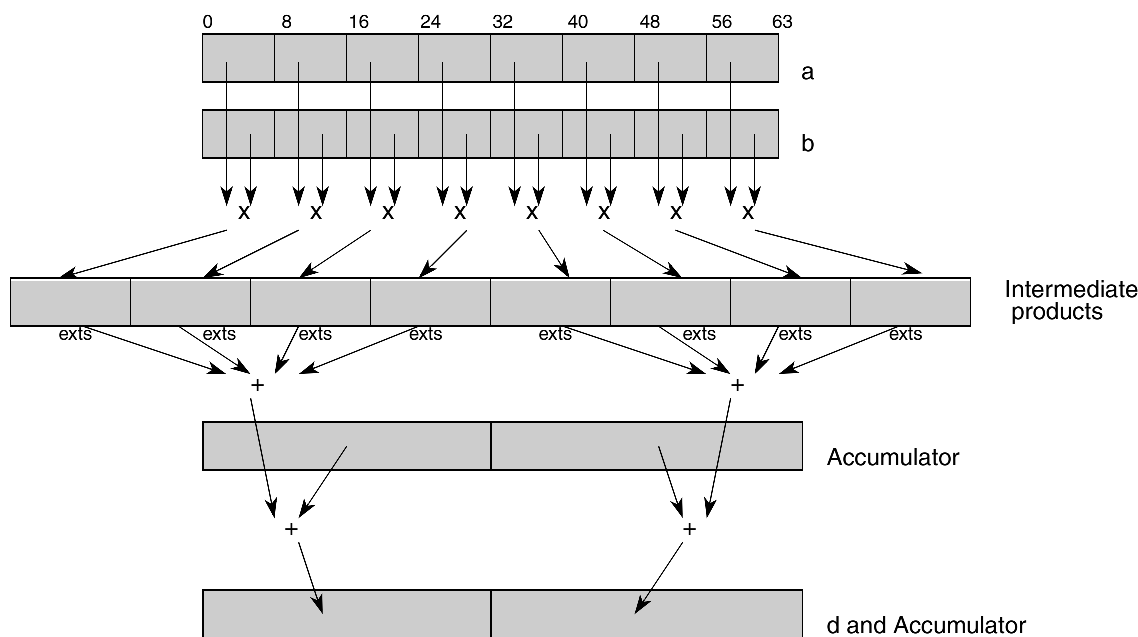


Figure 3-147. Vector Dot Product of Bytes, Add, Signed, Modulo, Integer and Accumulate Words (__ev_dotpbasmiaaw)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evdotpbasmiaaw d,a,b**

__ev_dotpbasmiaaw3 __ev_dotpbasmiaaw3

Vector Dot Product of Bytes, Add, Signed, Modulo, Integer and Accumulate into Words, 3 operand

d = __ev_dotpbasmiaaw3 (a,b,c)

```
// high dot
temph0:31 ← EXTS32(b0:7 ×si c0:7) + EXTS32(b8:15 ×si c8:15) + EXTS32(b16:23 ×si c16:23)
          + EXTS32(b24:31 ×si c24:31) + a0:31
d0:31 ← temph0:31

//low
templ0:31 ← EXTS32(b32:39 ×si c32:39) + EXTS32(b40:47 ×si c40:47)
          + EXTS32(b48:55 ×si c48:55) + EXTS32(b56:63 ×si c56:63) + a32:63
d32:63 ← templ0:31

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

For each word element in the destination, corresponding byte pairs of signed integer elements in parameters **b** and **c** are multiplied producing four 16-bit products. This quad of intermediate products are sign-extended to 32 bits and added together with the contents of the corresponding parameter **a** word and the sum is placed into the corresponding parameter **d** and accumulator word.

Other registers altered: ACC

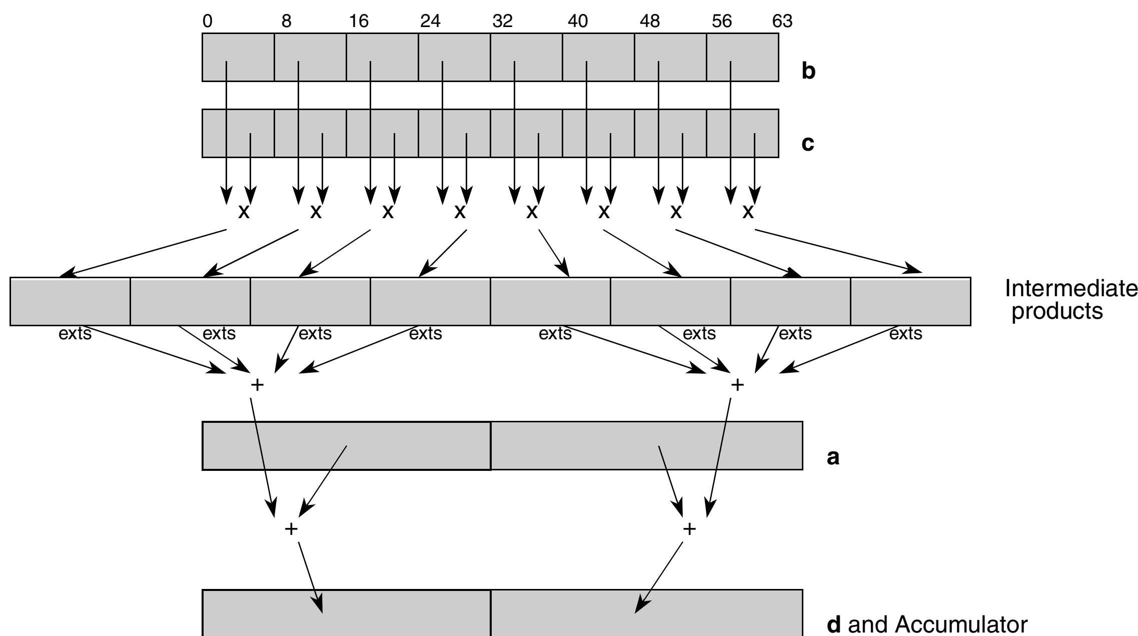


Figure 3-148. Vector Dot Product of Bytes, Add, Signed, Modulo, Integer and Accumulate Words 3 op (__ev_dotpbasmiaaw3)

d	a	b	c	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**d ←a** **evdotpbasmiaaw3 d,b,c**

__ev_dotpbasumi[a] __ev_dotpbasumi[a]

Vector Dot Product of Bytes, Add, Signed by Unsigned, Modulo, Integer (to Accumulator)

d = __ev_dotpbasumi (a,b) (A = 0)

d = __ev_dotpbasumia (a,b) (A = 1)

```
// high dot
temph0:31 ←EXTS(a0:7 ×sui b0:7) + EXTS(a8:15 ×sui b8:15) + EXTS(a16:23 ×sui b16:23) +
        EXTS(a24:31 ×sui b24:31)
d0:31 ←temph0:31

//low
templ0:31 ←EXTS(a32:39 ×sui b32:39) + EXTS(a40:47 ×sui b40:47) +
        EXTS(a48:55 ×sui b48:55) + EXTS(a56:63 ×sui b56:63)
d32:63 ←templ0:31

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

For each word element in the destination, corresponding byte pairs of signed integer elements in parameter **a** and unsigned integer elements in parameter **b** are multiplied producing four 16-bit products. This quad of intermediate products are sign-extended to 32 bits and added together and the sum is placed into the corresponding parameter **d** word. If A = 1, the result in parameter **d** is also placed into the accumulator.

Other registers altered: ACC (if A=1)

Figure 3-149. Vector Dot Product of Bytes, Add, Signed by Unsigned, Modulo, Integer (to Accumulator) (__ev_dotpbasumi[a])

A	d	a	b	Maps to
A = 0	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evdotpbasumi d,a,b**
A = 1	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evdotpbasumia d,a,b**

__ev_dotpbasumiaaw __ev_dotpbasumiaaw

Vector Dot Product of Bytes, Add, Signed by Unsigned, Modulo, Integer and Accumulate into Words

d = __ev_dotpbasumiaaw (a,b)

```
// high dot
temph0:31 ← EXTS(a0:7 ×sui b0:7) + EXTS(a8:15 ×sui b8:15) + EXTS(a16:23 ×sui b16:23) +
            EXTS(a24:31 ×sui b24:31) + ACC0:31
d0:31 ← temph0:31
//low
templ0:31 ← EXTS(a32:39 ×sui b32:39) + EXTS(a40:47 ×sui b40:47) +
            EXTS(a48:55 ×sui b48:55) + EXTS(a56:63 ×sui b56:63) + ACC32:63
d32:63 ← templ0:31
// update accumulator
ACC0:63 ←d0:63
```

For each word element in the destination, corresponding byte pairs of signed integer elements in parameter **a** and unsigned integer elements in parameter **b** are multiplied producing four 16-bit products. This quad of intermediate products are sign-extended to 32 bits and added together with the contents of the corresponding accumulator word and the sum is placed into the corresponding parameter **d** and accumulator word.

Other registers altered: ACC

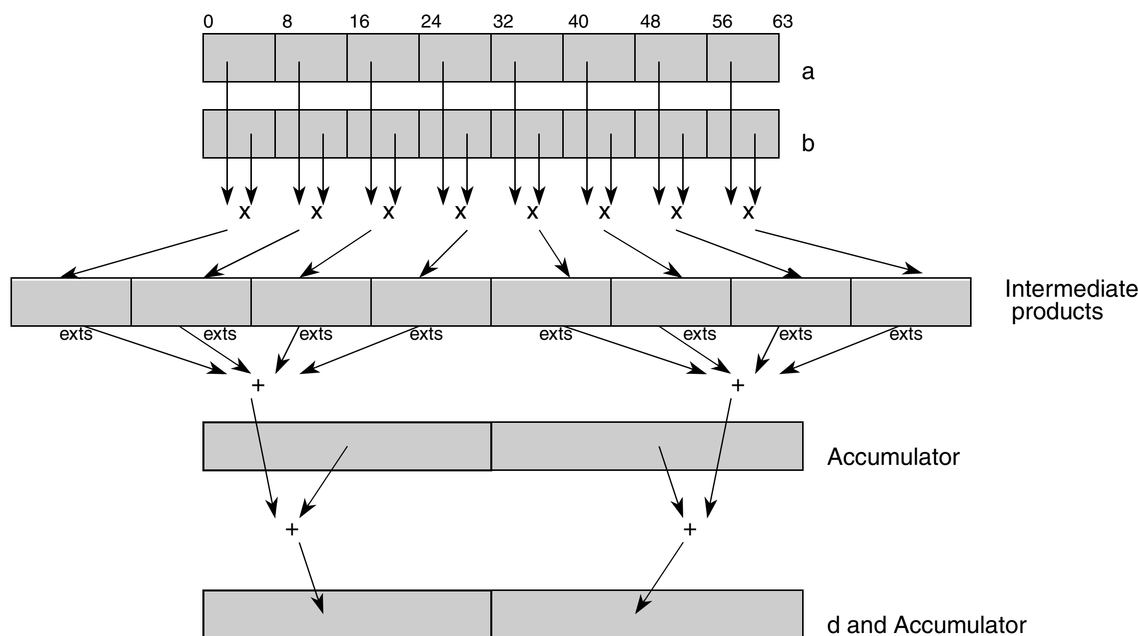


Figure 3-150. Vector Dot Product of Bytes, Add, Signed by Unsigned, Modulo, Integer and Accumulate Words (__ev_dotpbasumiaaw)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evdotpbasumiaaw d,a,b**

__ev_dotpbasumiaaw3 __ev_dotpbasumiaaw3

Vector Dot Product of Bytes, Add, Signed by Unsigned, Modulo, Integer and Accumulate into Words, 3 operand

d = __ev_dotpbasumiaaw3 (a,b,c)

```
// high dot
temph0:31 ← EXTS32(b0:7 ×sui c0:7) + EXTS32(b8:15 ×sui c8:15)
          + EXTS32(b16:23 ×sui c16:23) + EXTS32(b24:31 ×sui c24:31) + a0:31
d0:31 ← temph0:31
//low
templ0:31 ← EXTS32(b32:39 ×sui a32:39) + EXTS32(b40:47 ×sui c40:47)
          + EXTS32(b48:55 ×sui c48:55) + EXTS32(b56:63 ×sui c56:63) + a32:63
d32:63 ← templ0:31

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

For each word element in the destination, corresponding byte pairs of signed integer elements in **b** and unsigned integer elements in parameter **c** are multiplied producing four 16-bit products. This quad of intermediate products are sign-extended to 32 bits and added together with the contents of the corresponding parameter **a** word and the sum is placed into the corresponding parameter **d** and accumulator word.

Other registers altered: ACC

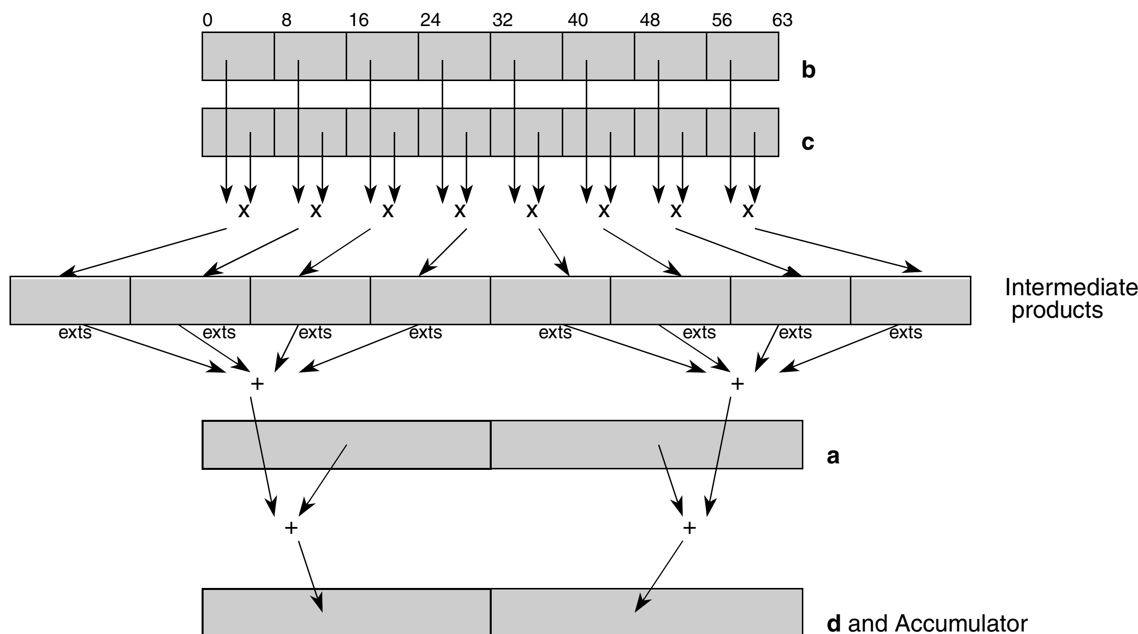


Figure 3-151. Vector Dot Product of Bytes, Add, Signed by Unsigned, Modulo, Integer and Accumulate Words 3 op (__ev_dotpbasumiaaw3)

d	a	b	c	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**d ←a** **evdotpbasumiaaw3 d,b,c**

__ev_dotpbaumi[a] __ev_dotpbaumi[a]

Vector Dot Product of Bytes, Add, Unsigned, Modulo, Integer (to Accumulator)

d = __ev_dotpbaumi (a,b) (A = 0)

d = __ev_dotpbaumia (a,b) (A = 1)

```
// high dot
temph0:31 ←EXTZ(a0:7 ×ui b0:7) + EXTZ(a8:15 ×ui b8:15) + EXTZ(a16:23 ×ui b16:23) +
          EXTZ(a24:31 ×ui b24:31)
d0:31 ←temph0:31

//low
templ0:31 ←EXTZ(a32:39 ×ui b32:39) + EXTZ(a40:47 ×ui b40:47) +
          EXTZ(a48:55 ×ui b48:55) + EXTZ(a56:63 ×ui b56:63)
d32:63 ←templ0:31

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

For each word element in the destination, corresponding byte pairs of unsigned integer elements in parameters **a** and **b** are multiplied producing four 16-bit products. This quad of intermediate products are zero-extended to 32 bits and added together and the sum is placed into the corresponding parameter **d** word. If A = 1, the result in parameter **d** is also placed into the accumulator.

Other registers altered: ACC (if A=1)

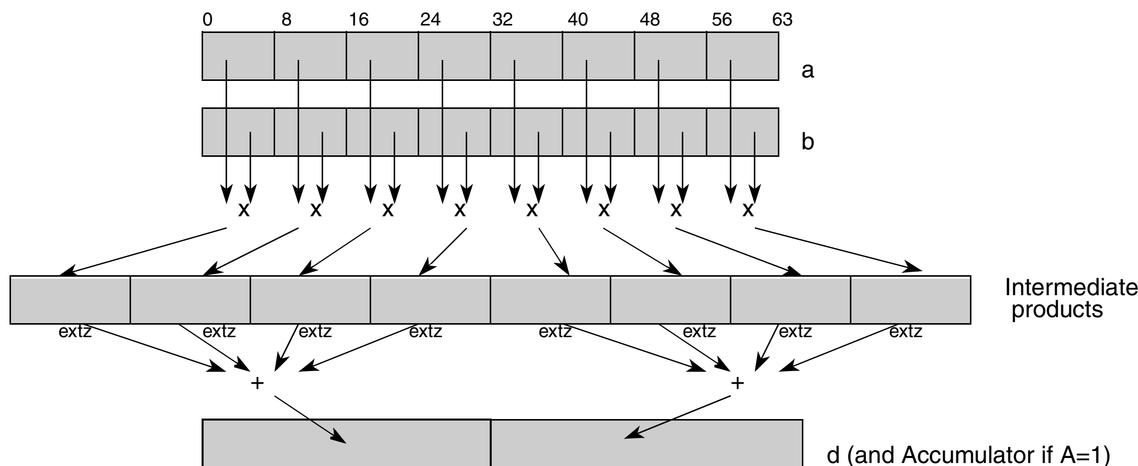


Figure 3-152. Vector Dot Product of Bytes, Add, Unsigned, Modulo, Integer (to Accumulator) (__ev_dotpbaumi[a])

A	d	a	b	Maps to
A = 0	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evdotpbaumi d,a,b**
A = 1	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evdotpbaumia d,a,b**

__ev_dotpbaumiaaw __ev_dotpbaumiaaw

Vector Dot Product of Bytes, Add, Unsigned, Modulo, Integer and Accumulate into Words

d = __ev_dotpbaumiaaw (a,b)

```
// high dot
temph0:31 ←EXTZ(a0:7 ×ui a0:7) + EXTZ(a8:15 ×ui b8:15) + EXTZ(a16:23 ×ui b16:23) +
          EXTZ(a24:31 ×ui b24:31) + ACC0:31
d0:31 ←temph0:31
//low
templ0:31 ←EXTZ(a32:39 ×ui b32:39) + EXTZ(a40:47 ×ui b40:47) +
          EXTZ(a48:55 ×ui b48:55) + EXTZ(a56:63 ×ui b56:63) + ACC32:63
d32:63 ←templ0:31
// update accumulator
ACC0:63 ←d0:63
```

For each word element in the destination, corresponding byte pairs of unsigned integer elements in parameter **a** and parameter **b** are multiplied producing four 16-bit products. This quad of intermediate products are zero-extended to 32 bits and added together with the contents of the corresponding accumulator word and the sum is placed into the corresponding parameter **d** and accumulator word.

Other registers altered: ACC

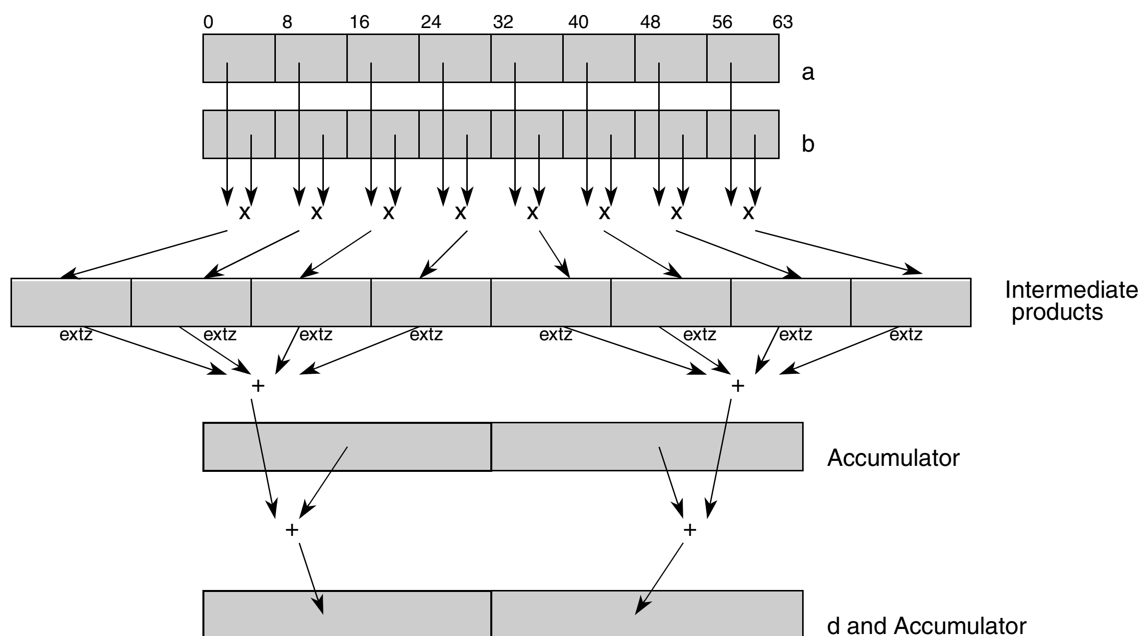


Figure 3-153. Vector Dot Product of Bytes, Add, Signed by Unsigned, Modulo, Integer and Accumulate Words (__ev_dotpbaumiaaw)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evdotpbaumiaaw d,a,b**

__ev_dotpbaumiaaw3 __ev_dotpbaumiaaw3

Vector Dot Product of Bytes, Add, Unsigned, Modulo, Integer and Accumulate into Words, 3 operand

d = __ev_dotpbaumiaaw3 (a,b,c)

```
// high dot
temph0:31 ← EXTZ32(b0:7 ×ui c0:7) + EXTZ32(b8:15 ×ui c8:15) + EXTZ32(b16:23 ×ui c16:23)
        + EXTZ32(b24:31 ×ui c24:31) + a0:31
d0:31 ← temph0:31
//low
templ0:31 ← EXTZ32(b32:39 ×ui c32:39) + EXTZ32(b40:47 ×ui c40:47)
        + EXTZ32(b48:55 ×ui c48:55) + EXTZ32(b56:63 ×ui c56:63) + a32:63
d32:63 ← templ0:31
// update accumulator
ACC0:63 ←d0:63
```

For each word element in the destination, corresponding byte pairs of unsigned integer elements in parameters **b** and **c** are multiplied producing four 16-bit products. This quad of intermediate products are zero-extended to 32 bits and added together with the contents of the corresponding parameter **a** word and the sum is placed into the corresponding parameter **d** and accumulator word.

Other registers altered: ACC

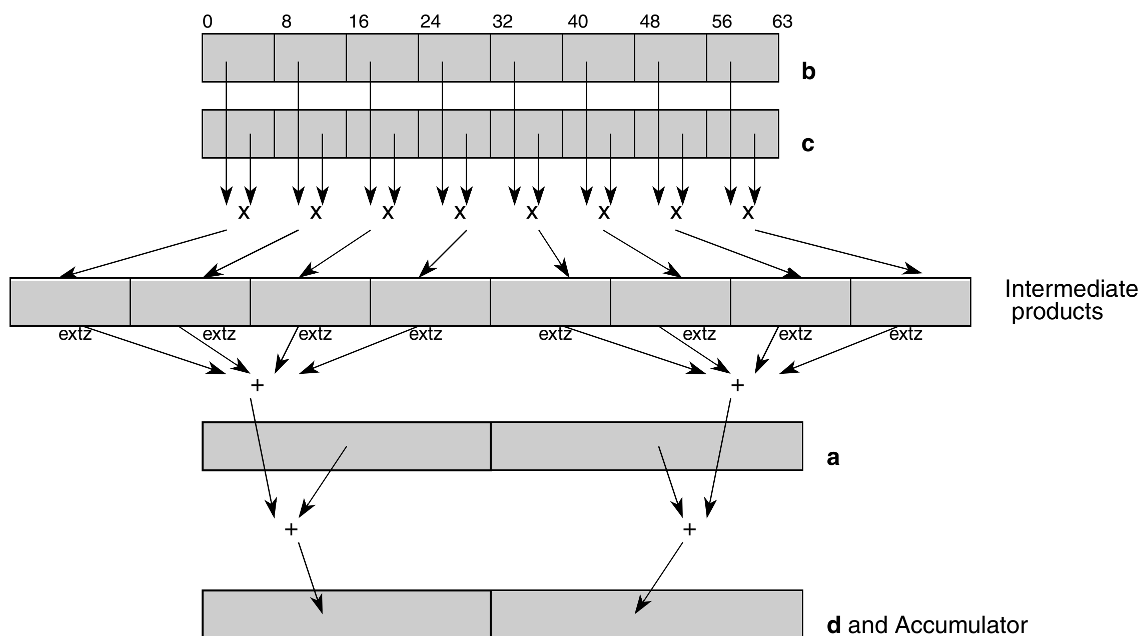


Figure 3-154. Vector Dot Product of Bytes, Add, Signed by Unsigned, Modulo, Integer and Accumulate Words 3 op (__ev_dotpbaumiaaw3)

d	a	b	c	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	d ←a **evdotpbaumiaaw3 d,b,c**

__ev_dotphasmi[a] __ev_dotphasmi[a]

Vector Dot Product of Half Words, Add, Signed, Modulo, Integer (to Accumulator)

d = __ev_dotphasmi (a,b) (A = 0)

d = __ev_dotphasmia (a,b) (A = 1)

```
// high dot
temph1_0:31 ← a_0:15 ×si b_0:15
temph2_0:31 ← a_16:31 ×si b_16:31
temph_0:31 ← temph1_0:31 + temph2_0:31 // modulo sum
d_0:31 ← temph_0:31

//low
templ1_0:31 ← a_32:47 ×si b_32:47
templ2_0:31 ← a_48:63 ×si b_48:63
templ_0:31 ← templ1_0:31 + templ2_0:31 // modulo sum
d_32:63 ← templ_0:31

// update accumulator
if A = 1 then ACC_0:63 ←d_0:63
```

For each word element in the destination, corresponding half word pairs of signed integer elements in parameters **a** and **b** are multiplied producing a pair of 32-bit products. This pair of intermediate 32-bit products is added together and the sum is placed into the corresponding parameter **d** word. 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 Dot Product of Half Words, Add, Signed, Modulo, Integer (to Accumulator) (__ev_dotphasmi[a])

A	d	a	b	Maps to
A = 0	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evdotphasmi d,a,b**
A = 1	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evdotphasmia d,a,b**

__ev_dotphasmiaaw __ev_dotphasmiaaw

Vector Dot Product of Half Words, Add, Signed, Modulo, Integer and Accumulate into Words

d = __ev_dotphasmiaaw (a,b)

```
// high dot
temph1[0:31] ← a[0:15] ×si b[0:15]
temph2[0:31] ← a[16:31] ×si b[16:31]
temph[0:31] ← temph1[0:31] + temph2[0:31] + ACC[0:31] // modulo sum
d[0:31] ← temph[0:31]

//low
templ1[0:31] ← a[32:47] ×si b[32:47]
templ2[0:31] ← a[48:63] ×si b[48:63]
templ[0:31] ← templ1[0:31] + templ2[0:31] + ACC[32:63] // modulo sum
d[32:63] ← templ[0:31]

// update accumulator
ACC[0:63] ←d[0:63]
```

For each word element in the destination, corresponding half word pairs of signed integer elements in parameters **a** and **b** are multiplied producing a pair of 32-bit products. This pair of intermediate 32-bit products is added together with the contents of the corresponding accumulator word and the sum is placed into the corresponding parameter **d** and accumulator word.

Other registers altered: ACC

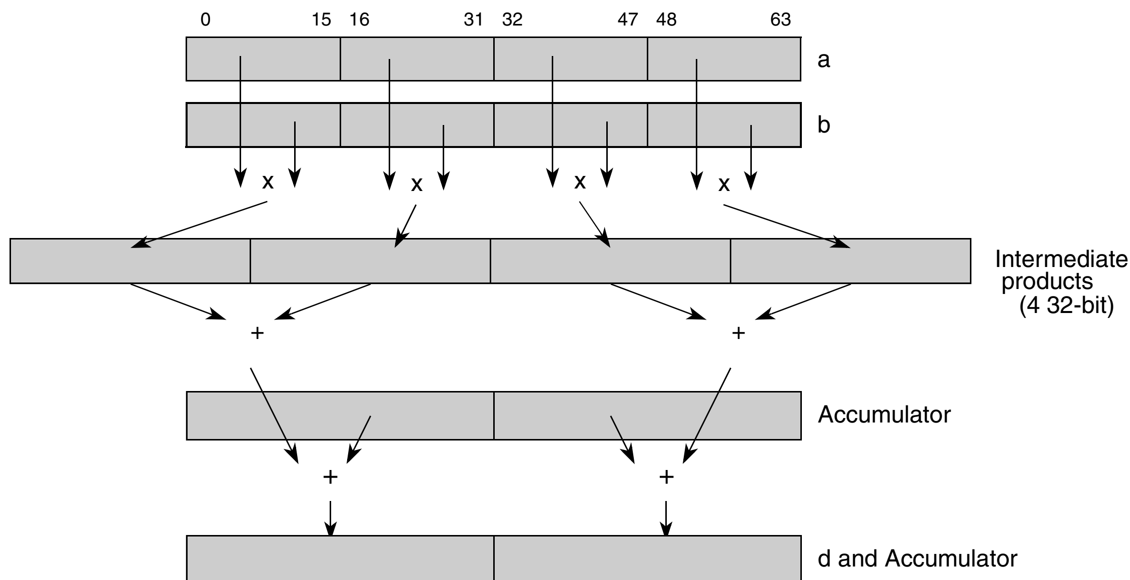


Figure 3-156. Vector Dot Product of Half Words, Add, Signed, Modulo, Integer and Accumulate Words (__ev_dotphasmiaaw)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evdotphasmiaaw d,a,b**

__ev_dotphasmiaaw3 __ev_dotphasmiaaw3

Vector Dot Product of Halfwords, Add, Signed, Modulo, Integer and Accumulate into Words, 3 operand

d = __ev_dotphasmiaaw3 (a,b,c)

```
// high dot
temph1_{0:31} ← b_{0:15} ×_{si} c_{0:15}
temph2_{0:31} ← b_{16:31} ×_{si} c_{16:31}
temph_{0:31} ← temph1_{0:31} + temph2_{0:31} + a_{0:31} // modulo sum
d_{0:31} ← temph_{0:31}

//low
templ1_{0:31} ← b_{32:47} ×_{si} c_{32:47}
templ2_{0:31} ← b_{48:63} ×_{si} c_{48:63}
templ_{0:31} ← templ1_{0:31} + templ2_{0:31} + a_{32:63} // modulo sum
d_{32:63} ← templ_{0:31}

// update accumulator
ACC_{0:63} ←d_{0:63}
```

For each word element in the destination, corresponding halfword pairs of signed integer elements in parameters **b** and **c** are multiplied producing a pair of 32-bit products. This pair of intermediate 32-bit products is added together with the contents of the corresponding parameter **a** word and the sum is placed into the corresponding parameter **d** and accumulator word.

Other registers altered: ACC

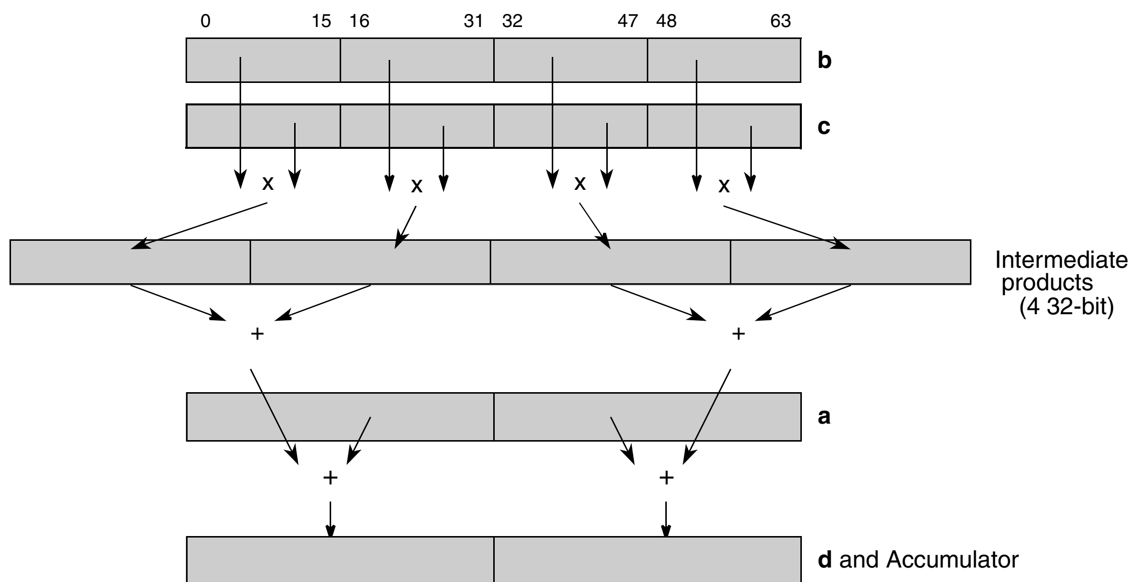


Figure 3-157. Vector Dot Product of Halfwords, Add, Signed, Modulo, Integer and Accumulate Words 3 op (__ev_dotphasmiaaw3)

d	a	b	c	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**d ←a** **evdotphasmiaaw3 d,b,c**

__ev_dotphassf[a] __ev_dotphassf[a]

Vector Dot Product of Half Words, Add, Signed, Saturate, Fractional (to Accumulator)

d = __ev_dotphassf (a,b) (A = 0)

d = __ev_dotphassfa (a,b) (A = 1)

```
// high dot
temph1_{0:31} ← a_{0:15} ×sf b_{0:15}
if (a_{0:15} = 0x8000) & (b_{0:15} = 0x8000) then
    temph1_{0:31} ←0x7FFF_FFFF //saturate
    movh ←1
else
    movh ←0
temph2_{0:31} ← a_{16:31} ×sf b_{16:31}
if (a_{16:31} = 0x8000) & (b_{16:31} = 0x8000) then
    temph2_{0:31} ←0x7FFF_FFFF //saturate
    movh ←1
endif
temph_{0:63} ←EXTS(temph1_{0:31}) + EXTS(temph2_{0:31})
ovh ←temph_{31} ⊕ temph_{32}
d_{0:31} ←SATURATE(ovh, temph_{31}, 0x8000_0000, 0xFFFF_FFFF, temph_{32:63})

//low
templ1_{0:31} ←a_{32:47} ×sf b_{32:47}
if (a_{32:47} = 0x8000) & (b_{32:47} = 0x8000) then
    templ1_{0:31} ←0x7FFF_FFFF //saturate
    movl ←1
else
    movl ←0
templ2_{0:31} ←a_{48:63} ×sf b_{48:63}
if (a_{48:63} = 0x8000) & (b_{48:63} = 0x8000) then
    templ2_{0:31} ←0x7FFF_FFFF //saturate
    movl ←1
endif
templ_{0:63} ←EXTS(templ1_{0:31}) + EXTS(templ2_{0:31})
ovl ←templ_{31} ⊕ templ_{32}
d_{32:63} ←SATURATE(ovl, templ_{31}, 0x8000_0000, 0xFFFF_FFFF, templ_{32:63})
// update accumulator
if A = 1 then ACC_{0:63} ←d_{0:63}
// update SPEFSCR
SPEFSCR_{OVH} ←movh | ovh; SPEFSCR_{OV} ←movl | ovl
SPEFSCR_{SOVH} ←SPEFSCR_{SOVH} | movh | ovh; SPEFSCR_{SOV} ←SPEFSCR_{SOV} | movl | ovl
```

For each word element in the destination, corresponding half word pairs of signed fractional elements in parameters **a** and **b** are multiplied producing a pair of 32-bit products. If both inputs of a multiply are -1.0, the result saturates to 0x7FFF_FFFF. This pair of intermediate 32-bit products is added together, saturating if overflow or underflow occurs, and the sum is placed into the corresponding parameter **d** word. If A = 1, the result in parameter **d** is also placed into the accumulator

The overflow and summary overflow bits are recorded in the SPEFSCR based on an overflow or underflow from either the multiply or the accumulation.

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

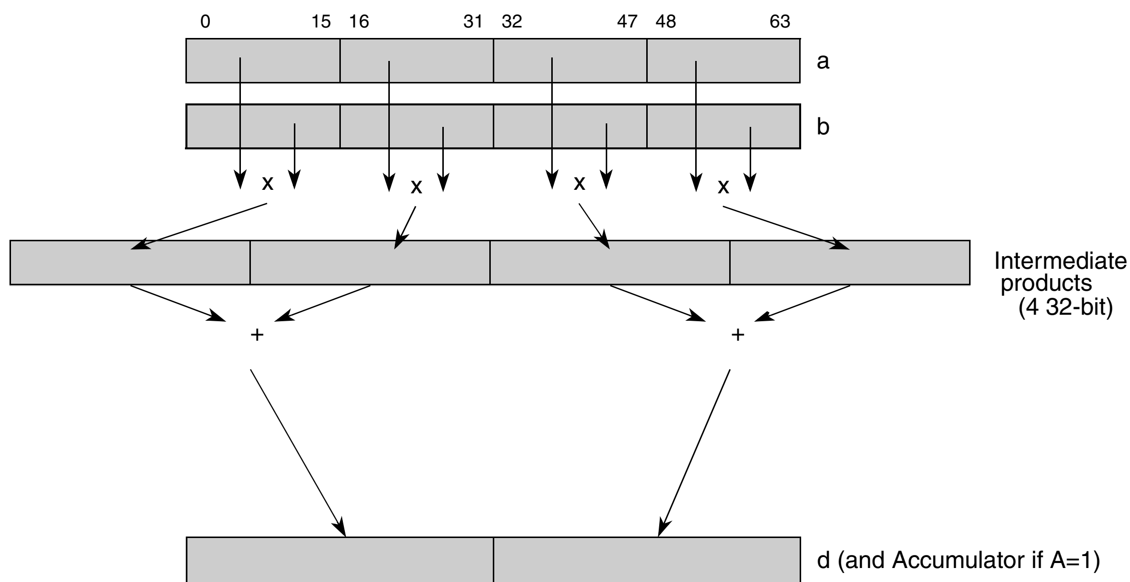


Figure 3-158. Vector Dot Product of Half Words, Add, Signed, Saturate, Fractional (to Accumulator) (__ev_dotphassf[a])

A	d	a	b	Maps to
A = 0	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evdotphassf d,a,b**
A = 1	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evdotphassfa d,a,b**

__ev_dotphassfaaw __ev_dotphassfaaw

Vector Dot Product of Half Words, Add, Signed, Saturate, Fractional and Accumulate into Words

d = __ev_dotphassfaaw (a,b)

```
// high dot
temph1_{0:31} ← a_{0:15} ×_{sf} b_{0:15}
if (a_{0:15} = 0x8000) & (b_{0:15} = 0x8000) then
    temph1_{0:31} ←0x7FFF_FFFF //saturate
    movh ←1
else
    movh ←0
temph2_{0:31} ← a_{16:31} ×_{sf} b_{16:31}
if (a_{16:31} = 0x8000) & (b_{16:31} = 0x8000) then
    temph2_{0:31} ←0x7FFF_FFFF //saturate
    movh ←1
endif
temph_{0:63} ← EXTS(temph1_{0:31}) + EXTS(temph2_{0:31}) + EXTS(ACC_{0:31})
ovh ← temph_{31} ⊕ temph_{32}
d_{0:31} ← SATURATE(ovh, temph_{31}, 0x8000_0000, 0xFFFF_FFFF, temph_{32:63})

//low
templ1_{0:31} ← a_{32:47} ×_{sf} b_{32:47}
if (a_{32:47} = 0x8000) & (b_{32:47} = 0x8000) then
    templ1_{0:31} ←0x7FFF_FFFF //saturate
    movl ←1
else
    movl ←0
templ2_{0:31} ← a_{48:63} ×_{sf} b_{48:63}
if (a_{48:63} = 0x8000) & (b_{48:63} = 0x8000) then
    templ2_{0:31} ←0x7FFF_FFFF //saturate
    movl ←1
endif
templ_{0:63} ← EXTS(templ1_{0:31}) + EXTS(templ2_{0:31}) + EXTS(ACC_{32:63})
ovl ← templ_{31} ⊕ templ_{32}
d_{32:63} ← SATURATE(ovl, templ_{31}, 0x8000_0000, 0xFFFF_FFFF, templ_{32:63})
// update accumulator
ACC_{0:63} ← d_{0:63}
// update SPEFSCR
SPEFSCR_{OVH} ← movh | ovh; SPEFSCR_{OV} ← movl | ovl
SPEFSCR_{SOVH} ← SPEFSCR_{SOVH} | movh | ovh; SPEFSCR_{SOV} ← SPEFSCR_{SOV} | movl | ovl
```

For each word element in the destination, corresponding half word pairs of signed fractional elements in parameter **a** and parameter **b** are multiplied producing a pair of 32-bit products. If both inputs of a multiply are -1.0, the result saturates to 0x7FFF_FFFF. This pair of intermediate 32-bit products is added together with the contents of the corresponding accumulator word, saturating if overflow or underflow occurs, and the sum is placed into the corresponding parameter **d** and accumulator word.

The overflow and summary overflow bits are recorded in the SPEFSCR based on an overflow or underflow from either the multiply or the accumulation.

Other registers altered: SPEFSCR, ACC

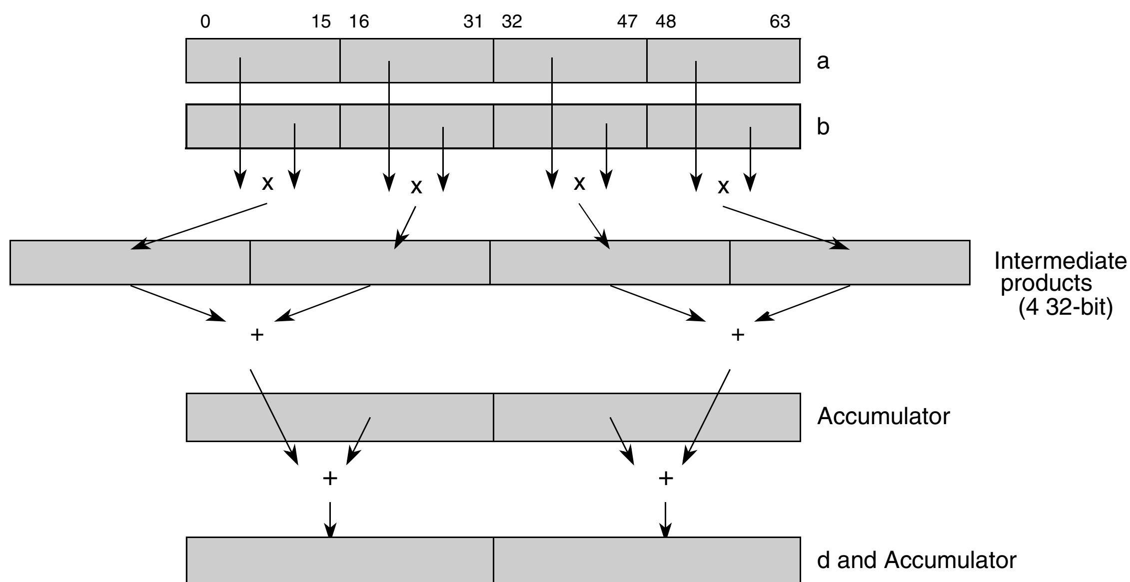


Figure 3-159. Vector Dot Product of Half Words, Add, Signed, Saturate, Fractional and Accumulate Words (__ev_dotphassfaaw)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evdotphassfaaw d,a,b**

__ev_dotphassfaaw3 __ev_dotphassfaaw3

Vector Dot Product of Halfwords, Add, Signed, Saturate, Fractional and Accumulate into Words, 3 operand

d = __ev_dotphassfaaw3 (a,b,c)

```
// high dot
temph1[0:31] ← b[0:15] ×sf c[0:15]
if (b[0:15] = 0x8000) & (c[0:15] = 0x8000) then
    temph1[0:31] ←0x7FFF_FFFF //saturate
    movh ←1
else
    movh ←0
temph2[0:31] ← b[16:31] ×sf c[16:31]
if (b[16:31] = 0x8000) & (c[16:31] = 0x8000) then
    temph2[0:31] ←0x7FFF_FFFF //saturate
    movh ←1
endif
temph[0:63] ← EXTS64(temph1[0:31]) + EXTS64(temph2[0:31]) + EXTS64(a[0:31])
ovh ← chk_ovf(temph[30:32])
d[0:31] ← SATURATE(ovh, temph[30], 0x8000_0000, 0x7FFF_FFFF, temph[32:63])

//low
templ1[0:31] ← b[32:47] ×sf c[32:47]
if (b[32:47] = 0x8000) & (c[32:47] = 0x8000) then
    templ1[0:31] ←0x7FFF_FFFF //saturate
    movl ←1
else
    movl ←0
templ2[0:31] ← b[48:63] ×sf c[48:63]
if (b[48:63] = 0x8000) & (c[48:63] = 0x8000) then
    templ2[0:31] ←0x7FFF_FFFF //saturate
    movl ←1
endif
templ[0:63] ← EXTS64(templ1[0:31]) + EXTS64(templ2[0:31]) + EXTS64(a[32:63])
ovl ← chk_ovf(templ[30:32])
d[32:63] ← SATURATE(ovl, templ[30], 0x8000_0000, 0x7FFF_FFFF, templ[32:63])
// update accumulator
ACC[0:63] ← d[0:63]
// update SPEFSCR
SPEFSCR[OVH] ← movh | ovh; SPEFSCR[OV] ← movl | ovl
SPEFSCR[SOVH] ← SPEFSCR[SOVH] | movh | ovh; SPEFSCR[SOV] ← SPEFSCR[SOV] | movl | ovl
```

For each word element in the destination, corresponding halfword pairs of signed fractional elements in parameters **b** and **c** are multiplied producing a pair of 32-bit products. If both inputs of a multiply are -1.0, the result saturates to 0x7FFF_FFFF. This pair of intermediate 32-bit products is added together with the contents of the corresponding parameter **a** word, saturating if overflow or underflow occurs, and the sum is placed into the corresponding parameter **d** and accumulator word.

The overflow and summary overflow bits are recorded in the SPEFSCR based on an overflow or underflow from either the multiply or the accumulation.

Other registers altered: SPEFSCR, ACC

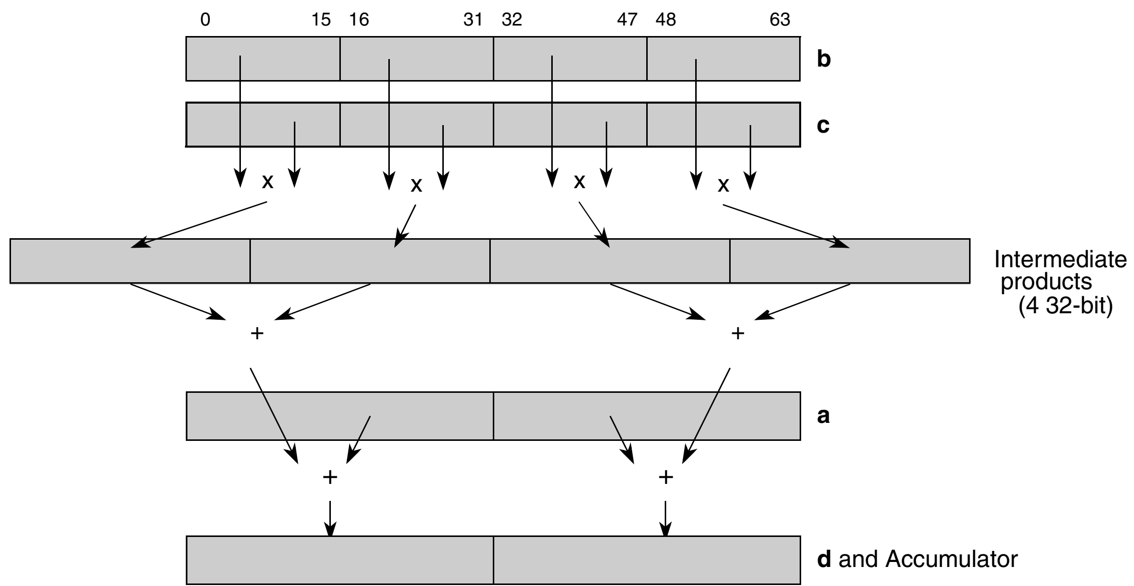


Figure 3-160. Vector Dot Product of Halfwords, Add, Signed, Saturate, Fractional and Accumulate Words 3 op (__ev_dotphassfaaw3)

d	a	b	c	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**d ←a** **evdotphassfaaw3 d,b,c**

__ev_dotphassfr[a]

__ev_dotphassfr[a]

Vector Dot Product of Halfwords, Add, Signed, Saturate, Fractional, Round (to Accumulator)

d = __ev_dotphassfr (a,b) (A = 0)

d = __ev_dotphassfra (a,b) (A = 1)

```
// high dot
temph1_{0:31} ← a_{0:15} ×sf b_{0:15}
if (a_{0:15} = 0x8000) & (b_{0:15} = 0x8000) then
    temph1_{0:31} ←0x7FFF_FFFF //saturate
    movh ←1
else
    movh ←0
temph2_{0:31} ← a_{16:31} ×sf b_{16:31}
if (a_{16:31} = 0x8000) & (b_{16:31} = 0x8000) then
    temph2_{0:31} ←0x7FFF_FFFF //saturate
    movh ←1
endif
temph_{0:63} ←ROUND((EXTS_{64}(temph1_{0:31}) + EXTS_{64}(temph2_{0:31})),16)
ovh ←chk_ovf(temph_{30:32})
d_{0:31} ←SATURATE(ovh, temph_{30}, 0x8000_0000, 0x7FFF_0000, temph_{32:63})

//low
templ1_{0:31} ← a_{32:47} ×sf b_{32:47}
if (a_{32:47} = 0x8000) & (b_{32:47} = 0x8000) then
    templ1_{0:31} ←0x7FFF_FFFF //saturate
    movl ←1
else
    movl ←0
templ2_{0:31} ← a_{48:63} ×sf b_{48:63}
if (a_{48:63} = 0x8000) & (b_{48:63} = 0x8000) then
    templ2_{0:31} ←0x7FFF_FFFF //saturate
    movl ←1
endif
templ_{0:63} ←ROUND((EXTS_{64}(templ1_{0:31}) + EXTS_{64}(templ2_{0:31})),16)
ovl ←chk_ovf(templ_{30:32})
d_{32:63} ←SATURATE(ovl, templ_{30}, 0x8000_0000, 0x7FFF_0000, templ_{32:63})
// update accumulator
if A = 1 then ACC_{0:63} ←d_{0:63}
// update SPEFSCR
SPEFSCR_{OVH} ←movh | ovh; SPEFSCR_{OV} ←movl | ovl
SPEFSCR_{SOVH} ←SPEFSCR_{SOVH} | movh | ovh; SPEFSCR_{SOV} ←SPEFSCR_{SOV} | movl | ovl
```

For each word element in the destination, corresponding halfword pairs of signed fractional elements in parameters **a** and **b** are multiplied producing a pair of 32-bit products. If both inputs of a multiply are -1.0, the result saturates to 0x7FFF_FFFF. This pair of intermediate 32-bit products is added together and rounded to 16 bits using the current fractional rounding mode in SPEFCSR, saturating if overflow or underflow occurs, and placed into the corresponding parameter **d** word. If A = 1, the result in parameter **d** is also placed into the accumulator

The overflow and summary overflow bits are recorded in the SPEFSCR based on an overflow or underflow from either the multiply or the accumulation.

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

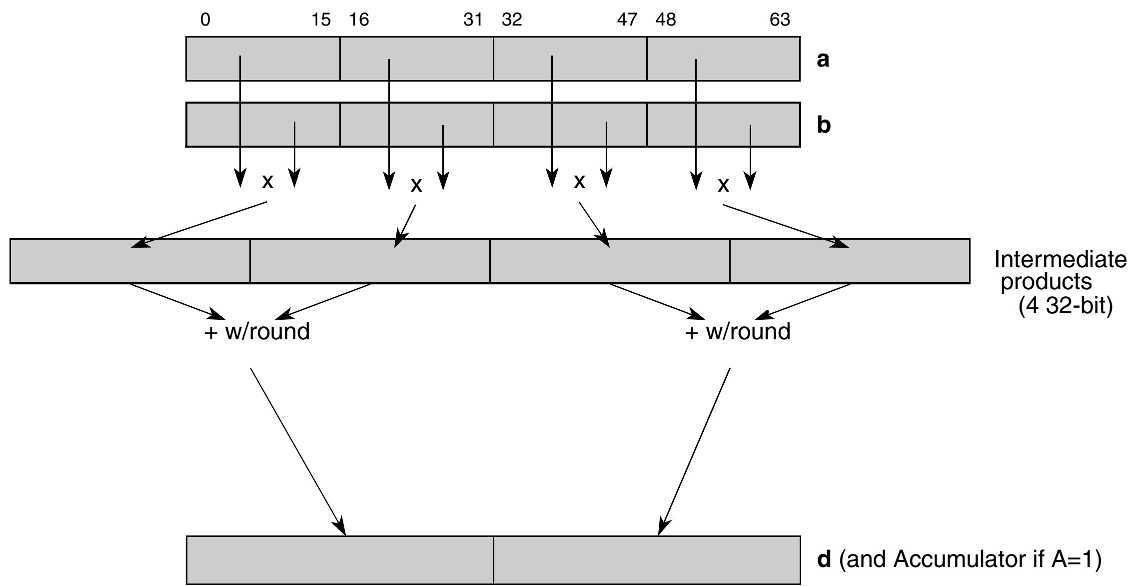


Figure 3-161. Vector Dot Product of Halfwords, Add, Signed, Saturate, Fractional, Round (to Accumulator) (__ev_dotphassfr[a])

A	d	a	b	Maps to
A = 0	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evdotphassfr d,a,b**
A = 1	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evdotphassfra d,a,b**

__ev_dotphassfraaw __ev_dotphassfraaw

Vector Dot Product of Halfwords, Add, Signed, Saturate, Fractional, Round and Accumulate into Words

d = __ev_dotphassfraaw (a,b)

```
// high dot
temph1_{0:31} ←a_{0:15} ×_{sf} b_{0:15}
if (a_{0:15} = 0x8000) & (b_{0:15} = 0x8000) then
    temph1_{0:31} ←0x7FFF_FFFF //saturate
    movh ←1
else
    movh ←0
temph2_{0:31} ←a_{16:31} ×_{sf} b_{16:31}
if (a_{16:31} = 0x8000) & (b_{16:31} = 0x8000) then
    temph2_{0:31} ←0x7FFF_FFFF //saturate
    movh ←1
endif
temph_{0:63} ←ROUND((EXTS_{64}(temph1_{0:31}) + EXTS_{64}(temph2_{0:31}) + EXTS_{64}(ACC_{0:31})),16)
ovh ←chk_ovf(temph_{30:32})
d_{0:31} ←SATURATE(ovh, temph_{30}, 0x8000_0000, 0x7FFF_0000, temph_{32:63})

//low
templ1_{0:31} ←a_{32:47} ×_{sf} b_{32:47}
if (a_{32:47} = 0x8000) & (b_{32:47} = 0x8000) then
    templ1_{0:31} ←0x7FFF_FFFF //saturate
    movl ←1
else
    movl ←0
templ2_{0:31} ←a_{48:63} ×_{sf} b_{48:63}
if (a_{48:63} = 0x8000) & (b_{48:63} = 0x8000) then
    templ2_{0:31} ←0x7FFF_FFFF //saturate
    movl ←1
endif
templ_{0:63} ←ROUND((EXTS_{64}(templ1_{0:31}) + EXTS_{64}(templ2_{0:31}) + EXTS_{64}(ACC_{32:63})),16)
ovl ←chk_ovf(templ_{30:32})
d_{32:63} ←SATURATE(ovl, templ_{30}, 0x8000_0000, 0x7FFF_0000, templ_{32:63})
// update accumulator
ACC_{0:63} ←d_{0:63}
// update SPEFSCR
SPEFSCR_{OVH} ←movh | ovh; SPEFSCR_{OV} ←movl | ovl
SPEFSCR_{SOVH} ←SPEFSCR_{SOVH} | movh | ovh; SPEFSCR_{SOV} ←SPEFSCR_{SOV} | movl | ovl
```

For each word element in the destination, corresponding halfword pairs of signed fractional elements in **a** and **b** are multiplied producing a pair of 32-bit products. If both inputs of a multiply are -1.0, the result saturates to 0x7FFF_FFFF. This pair of intermediate 32-bit products is added together with the contents of the corresponding accumulator word and rounded to 16 bits using the current fractional rounding mode in SPEFCSR, saturating if overflow or underflow occurs, and placed into the corresponding **d** and accumulator word.

The overflow and summary overflow bits are recorded in the SPEFSCR based on an overflow or underflow from either the multiply or the accumulation and round. It is implementation dependent whether the saturation is detected before or after rounding for the accumulate. The pseudocode above assumes detection after rounding.

Other registers altered: SPEFSCR, ACC

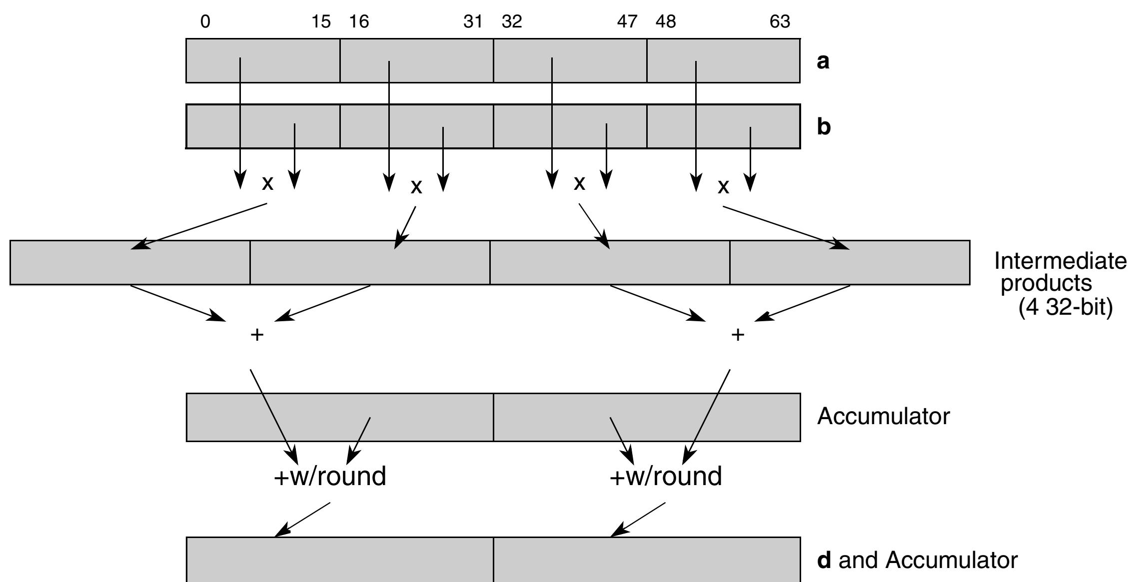


Figure 3-162. Vector Dot Product of Halfwords, Add, Signed, Saturate, Fractional, Round and Accumulate Words (__ev_dotphassfraaw)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evdotphassfraaw d,a,b**

__ev_dotphassfraaw3 __ev_dotphassfraaw3

Vector Dot Product of Halfwords, Add, Signed, Saturate, Fractional, Round and Accumulate into Words, 3 operand

d = __ev_dotphassfraaw3 (a,b,c)

```
// high dot
temph1_{0:31} ← b_{0:15} ×_{sf} c_{0:15}
if (b_{0:15} = 0x8000) & (c_{0:15} = 0x8000) then
    temph1_{0:31} ←0x7FFF_FFFF //saturate
    movh ←1
else
    movh ←0
temph2_{0:31} ← b_{16:31} ×_{sf} c_{16:31}
if (b_{16:31} = 0x8000) & (c_{16:31} = 0x8000) then
    temph2_{0:31} ←0x7FFF_FFFF //saturate
    movh ←1
endif
temph_{0:63} ← ROUND((EXTS_{64}(temph1_{0:31}) + EXTS_{64}(temph2_{0:31}) + EXTS_{64}(a_{0:31})),16)
ovh ← chk_ovf(temph_{30:32})
d_{0:31} ← SATURATE(ovh, temph_{30}, 0x8000_0000, 0x7FFF_0000, temph_{32:63})

//low
templ1_{0:31} ← b_{32:47} ×_{sf} c_{32:47}
if (b_{32:47} = 0x8000) & (c_{32:47} = 0x8000) then
    templ1_{0:31} ←0x7FFF_FFFF //saturate
    movl ←1
else
    movl ←0
templ2_{0:31} ← b_{48:63} ×_{sf} c_{48:63}
if (b_{48:63} = 0x8000) & (c_{48:63} = 0x8000) then
    templ2_{0:31} ←0x7FFF_FFFF //saturate
    movl ←1
endif
templ_{0:63} ← ROUND((EXTS_{64}(templ1_{0:31}) + EXTS_{64}(templ2_{0:31}) + EXTS_{64}(a_{32:63})),16)
ovl ← chk_ovf(templ_{30:32})
d_{32:63} ← SATURATE(ovl, templ_{30}, 0x8000_0000, 0x7FFF_0000, templ_{32:63})
// update accumulator
ACC_{0:63} ← d_{0:63}
// update SPEFSCR
SPEFSCR_{OVH} ← movh | ovh; SPEFSCR_{OV} ← movl | ovl
SPEFSCR_{SOVH} ← SPEFSCR_{SOVH} | movh | ovh; SPEFSCR_{SOV} ← SPEFSCR_{SOV} | movl | ovl
```

For each word element in the destination, corresponding halfword pairs of signed fractional elements in parameters **b** and **c** are multiplied producing a pair of 32-bit products. If both inputs of a multiply are -1.0, the result saturates to 0x7FFF_FFFF. This pair of intermediate 32-bit products is added together with the contents of the corresponding parameter **a** word and rounded to 16 bits using the current fractional rounding mode in SPEFCSR, saturating if overflow or underflow occurs, and placed into the corresponding parameter **d** and accumulator word.

The overflow and summary overflow bits are recorded in the SPEFSCR based on an overflow or underflow from either the multiply or the accumulation and round. It is implementation dependent whether the saturation is detected before or after rounding for the accumulate. The pseudocode above assumes detection after rounding.

Other registers altered: SPEFSCR, ACC

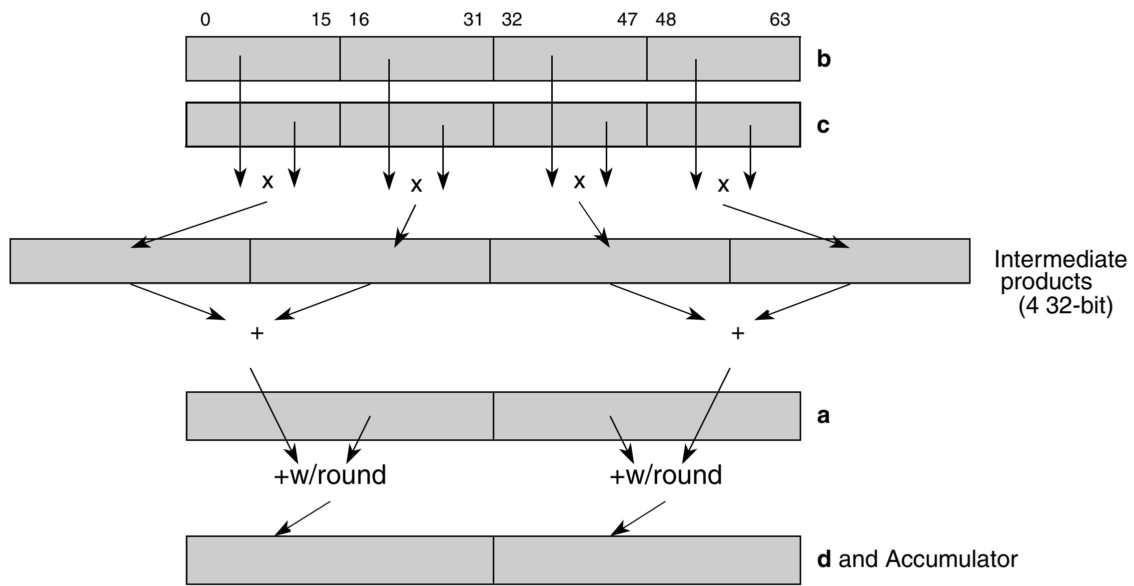


Figure 3-163. Vector Dot Product of Halfwords, Add, Signed, Saturate, Fractional, Round and Accumulate Words 3 op (__ev_dotphassfraaw3)

d	a	b	c	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**d ←a** **evdotphassfraaw3 d,b,c**

__ev_dotphassi[a] __ev_dotphassi[a]

Vector Dot Product of Half Words, Add, Signed, Saturate, Integer (to Accumulator)

d = __ev_dotphassi (a,b) (A = 0)

d = __ev_dotphassia (a,b) (A = 1)

```
// high dot
temph1_{0:31} ← a_{0:15} ×_{si} b_{0:15}; temph2_{0:31} ← a_{16:31} ×_{si} b_{16:31}
temph_{0:63} ← EXTS(temph1_{0:31}) + EXTS(temph2_{0:31})
ovh ← temph_{31} ⊕ temph_{32}
d_{0:31} ← SATURATE(ovh, temph_{31}, 0x8000_0000, 0xFFFF_FFFF, temph_{32:63})
//low
templ1_{0:31} ← a_{32:47} ×_{si} b_{32:47}; templ2_{0:31} ← a_{48:63} ×_{si} b_{48:63}
if (a_{48:63} = 0x8000) & (b_{48:63} = 0x8000) then
templ_{0:63} ← EXTS(templ1_{0:31}) + EXTS(templ2_{0:31})
ovl ← templ_{31} ⊕ templ_{32}
d_{32:63} ← SATURATE(ovl, templ_{31}, 0x8000_0000, 0xFFFF_FFFF, templ_{32:63})
// update accumulator
if A = 1 then 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 destination, corresponding half word pairs of signed integer elements in parameters **a** and **b** are multiplied producing a pair of 32-bit products. This pair of intermediate 32-bit products is added together, saturating if overflow or underflow occurs, and the sum is placed into the corresponding parameter **d** word. If A = 1, the result in parameter **d** is also placed into the accumulator

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

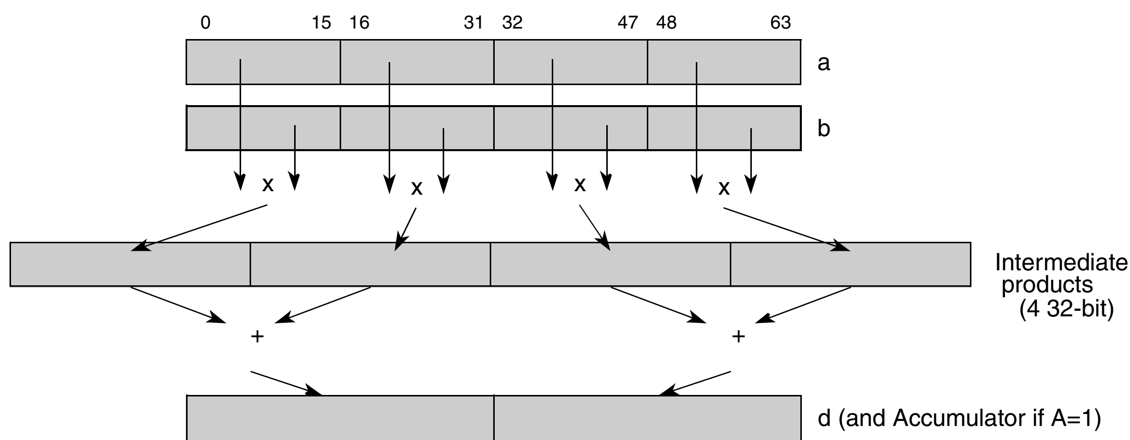


Figure 3-164. Vector Dot Product of Half Words, Add, Signed, Saturate, Integer (to Accumulator) (__ev_dotphassi[a])

A	d	a	b	Maps to
A = 0	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evdotphassi d,a,b**
A = 1	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evdotphassia d,a,b**

__ev_dotphassiaaw __ev_dotphassiaaw

Vector Dot Product of Half Words, Add, Signed, Saturate, Integer and Accumulate into Words

d = __ev_dotphassiaaw (a,b)

```
// high dot
temph1_0:31 ← a_0:15 ×si b_0:15; temph2_0:31 ← a_16:31 ×si b_16:31
temph_0:63 ← EXTS(temph1_0:31) + EXTS(temph2_0:31) + EXTS(ACC_0:31)
ovh ← temph_31 ⊕ temph_32
d_0:31 ← SATURATE(ovh, temph_31, 0x8000_0000, 0xFFFF_FFFF, temph_32:63)
//low
templ1_0:31 ← a_32:47 ×si b_32:47; templ2_0:31 ← a_48:63 ×si b_48:63
templ_0:63 ← EXTS(templ1_0:31) + EXTS(templ2_0:31) + EXTS(ACC_32:63)
ovl ← templ_31 ⊕ templ_32
d_32:63 ← SATURATE(ovl, templ_31, 0x8000_0000, 0xFFFF_FFFF, templ_32:63)
// 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 destination, corresponding half word pairs of signed integer elements in parameters **a** and **b** are multiplied producing a pair of 32-bit products. This pair of intermediate 32-bit products is added together with the contents of the corresponding accumulator word, saturating if overflow or underflow occurs, and the sum is placed into the corresponding parameter **d** and accumulator word.

Other registers altered: SPEFSCR, ACC

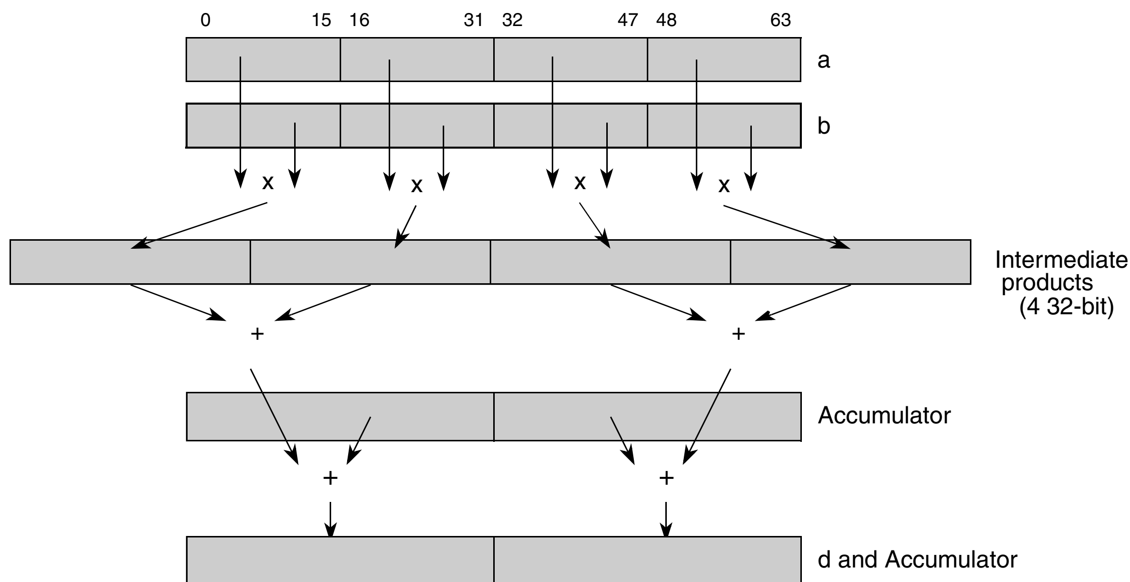


Figure 3-165. Vector Dot Product of Half Words, Add, Signed, Saturate, Integer and Accumulate (__ev_dotphassiaaw)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evdotphassiaaw d,a,b**

__ev_dotphassiaaw3 __ev_dotphassiaaw3

Vector Dot Product of Halfwords, Add, Signed, Saturate, Integer and Accumulate into Words, 3 operand

d = __ev_dotphassiaaw3 (a,b,c)

```
// high dot
temph1[0:31] ← b[0:15] ×si c[0:15]; temph2[0:31] ← b[16:31] ×si c[16:31]
temph[0:63] ← EXTS64(temph1[0:31]) + EXTS64(temph2[0:31]) + EXTS64(a[0:31])
ovh ← chk_ovf(temph[30:32])
d[0:31] ← SATURATE(ovh, temph[30], 0x8000_0000, 0x7FFF_FFFF, temph[32:63])
//low
templ1[0:31] ← b[32:47] ×si c[32:47]; templ2[0:31] ← b[48:63] ×si c[48:63]
templ[0:63] ← EXTS64(templ1[0:31]) + EXTS64(templ2[0:31]) + EXTS64(a[32:63])
ovl ← chk_ovf(templ[30:32])
d[32:63] ← SATURATE(ovl, templ[30], 0x8000_0000, 0x7FFF_FFFF, templ[32:63])
// 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 destination, corresponding halfword pairs of signed integer elements in parameters **b** and **c** are multiplied producing a pair of 32-bit products. This pair of intermediate 32-bit products is added together with the contents of the corresponding parameter **a** word, saturating if overflow or underflow occurs, and the sum is placed into the corresponding parameter **d** and accumulator word.

Other registers altered: SPEFSCR, ACC

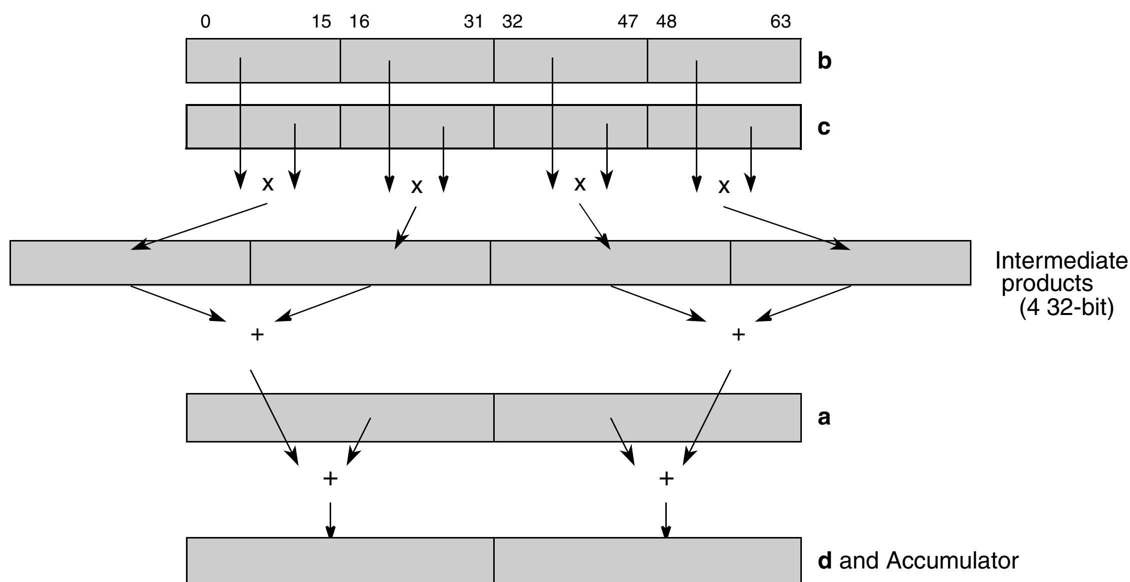


Figure 3-166. Vector Dot Product of Halfwords, Add, Signed, Saturate, Integer and Accumulate 3 op (__ev_dotphassiaaw3)

d	a	b	c	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**d ←a** **evdotphassiaaw3 d,b,c**

__ev_dotphasumi[a] __ev_dotphasumi[a]

Vector Dot Product of Half Words, Add, Signed by Unsigned, Modulo, Integer (to Accumulator)

d = __ev_dotphasumi (a,b) (A = 0)

d = __ev_dotphasumia (a,b) (A = 1)

```
// high dot
temph1[0:31] ← a[0:15] ×sui b[0:15]
temph2[0:31] ← a[16:31] ×sui b[16:31]
temph[0:31] ← temph1[0:31] + temph2[0:31] // modulo sum
d[0:31] ← temph[0:31]

//low
templ1[0:31] ← a[32:47] ×sui b[32:47]
templ2[0:31] ← a[48:63] ×siu b[48:63]
templ[0:31] ← templ1[0:31] + templ2[0:31] // modulo sum
d[32:63] ← templ[0:31]

// update accumulator
if A = 1 then ACC[0:63] ←d[0:63]
```

For each word element in the destination, corresponding half word pairs of signed integer elements in parameter **a** and unsigned integer elements in parameter **b** are multiplied producing a pair of 32-bit products. This pair of intermediate 32-bit products is added together and the sum is placed into the corresponding parameter **d** word. If A = 1, the result in parameter **d** is also placed into the accumulator.

Other registers altered: ACC (if A=1)

Figure 3-167. Vector Dot Product of Half Words, Add, Signed by Unsigned, Modulo, Integer (to Accumulator) (__ev_dotphasumi[a])

A	d	a	b	Maps to
A = 0	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evdotphasumi d,a,b**
A = 1	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evdotphasumia d,a,b**

__ev_dotphasumiaaw __ev_dotphasumiaaw

Vector Dot Product of Half Words, Add, Signed by Unsigned, Modulo, Integer and Accumulate into Words

d = __ev_dotphasumiaaw (a,b)

```
// high dot
temph1_{0:31} ← a_{0:15} ×_{sui} b_{0:15}
temph2_{0:31} ← a_{16:31} ×_{sui} b_{16:31}
temph_{0:31} ← temph1_{0:31} + temph2_{0:31} + ACC_{0:31} // modulo sum
d_{0:31} ← temph_{0:31}
//low
templ1_{0:31} ← a_{32:47} ×_{sui} b_{32:47}
templ2_{0:31} ← a_{48:63} ×_{sui} b_{48:63}
templ_{0:31} ← templ1_{0:31} + templ2_{0:31} + ACC_{32:63} // modulo sum
d_{32:63} ← templ_{0:31}
// update accumulator
ACC_{0:63} ←d_{0:63}
```

For each word element in the destination, corresponding half word pairs of signed integer elements in parameter **a** and unsigned integer elements in parameter **b** are multiplied producing a pair of 32-bit products. This pair of intermediate 32-bit products is added together with the contents of the corresponding accumulator word and the sum is placed into the corresponding parameter **d** and accumulator word.

Other registers altered: ACC

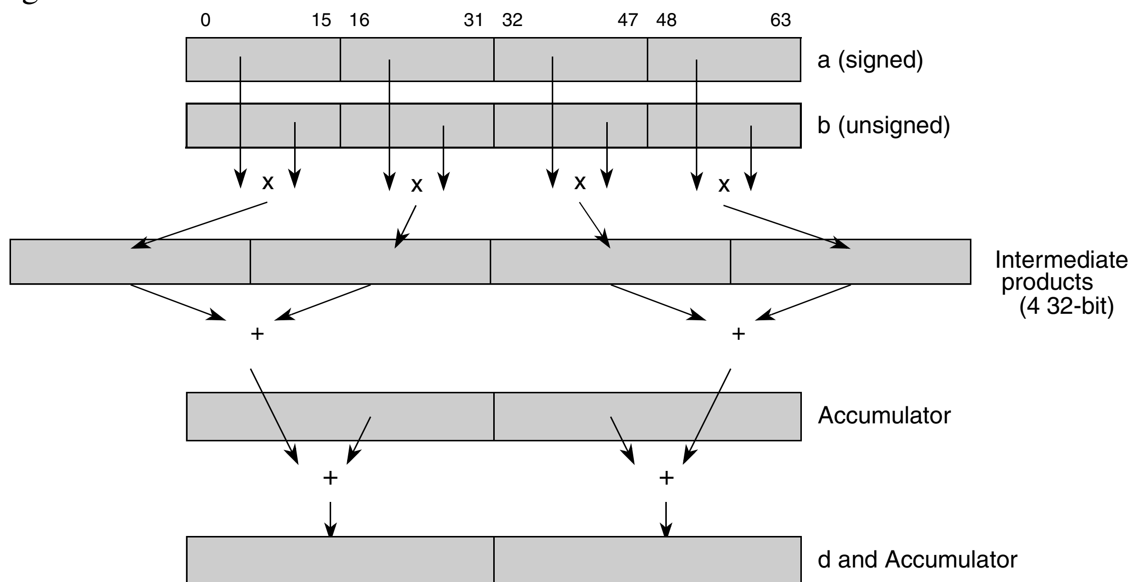


Figure 3-168. Vector Dot Product of Half Words, Add, Signed by Unsigned, Modulo, Integer and Accumulate Words (__ev_dotphasumiaaw)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evdotphasumiaaw d,a,b**

__ev_dotphasumiaaw3 __ev_dotphasumiaaw3

Vector Dot Product of Halfwords, Add, Signed by Unsigned, Modulo, Integer and Accumulate into Words, 3 operand

d = __ev_dotphasumiaaw3 (a,b,c)

```
// high dot
temph1_{0:31} ← b_{0:15} ×_{sui} c_{0:15}
temph2_{0:31} ← b_{16:31} ×_{sui} c_{16:31}
temph_{0:31} ← temph1_{0:31} + temph2_{0:31} + a_{0:31} // modulo sum
d_{0:31} ← temph_{0:31}

//low
templ1_{0:31} ← b_{32:47} ×_{sui} c_{32:47}
templ2_{0:31} ← b_{48:63} ×_{sui} c_{48:63}
templ_{0:31} ← templ1_{0:31} + templ2_{0:31} + a_{32:63} // modulo sum
d_{32:63} ← templ_{0:31}

// update accumulator
ACC_{0:63} ←d_{0:63}
```

For each word element in the destination, corresponding halfword pairs of signed integer elements in parameter **b** and unsigned integer elements in parameter **c** are multiplied producing a pair of 32-bit products. This pair of intermediate 32-bit products is added together with the contents of the corresponding parameter **a** word and the sum is placed into the corresponding parameter **d** and accumulator word.

Other registers altered: ACC

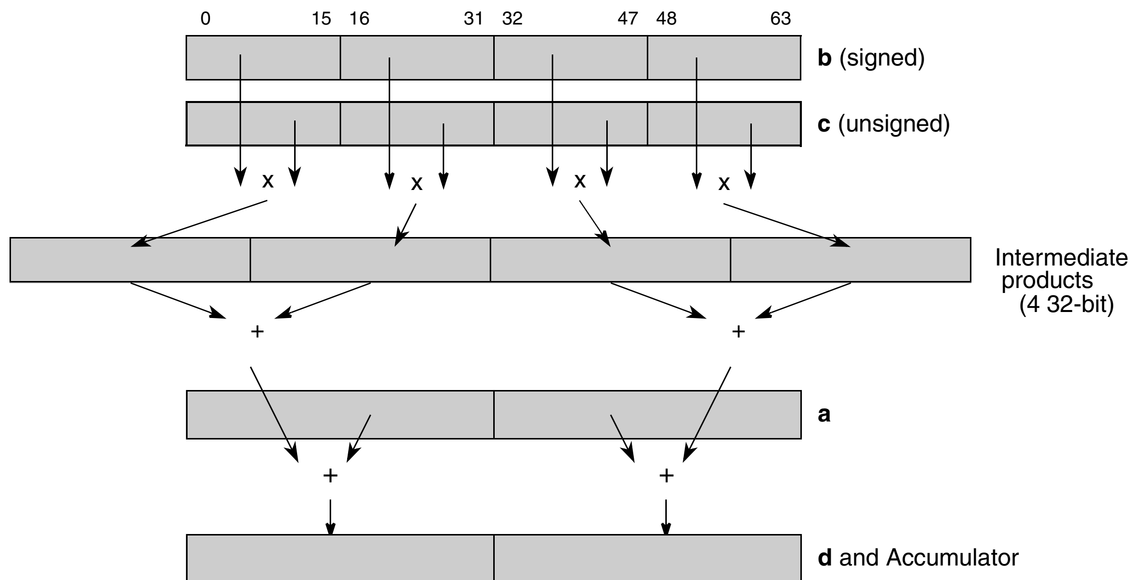


Figure 3-169. Vector Dot Product of Halfwords, Add, Signed by Unsigned, Modulo, Integer and Accumulate Words 3 op (__ev_dotphasumiaaw3)

d	a	b	c	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**d ←a evdotphasumiaaw3 d,b,c**

__ev_dotphasusi[a] __ev_dotphasusi[a]

Vector Dot Product of Half Words, Add, Signed by Unsigned, Saturate, Integer (to Accumulator)

d = __ev_dotphasusi (a,b) (A = 0)

d = __ev_dotphasusia (a,b) (A = 1)

```
// high dot
temph1[0:31] ← a[0:15] ×sui b[0:15]; temph2[0:31] ← a[16:31] ×sui b[16:31]
temph[0:63] ← EXTS(temph1[0:31]) + EXTS(temph2[0:31])
ovh ← temph[31] ⊕ temph[32]
d[0:31] ← SATURATE(ovh, temph[31], 0x8000_0000, 0xFFFF_FFFF, temph[32:63])
//low
templ1[0:31] ← a[32:47] ×sui b[32:47]; templ2[0:31] ← a[48:63] ×sui b[48:63]
templ[0:63] ← EXTS(templ1[0:31]) + EXTS(templ2[0:31])
ovl ← templ[31] ⊕ templ[32]
d[32:63] ← SATURATE(ovl, templ[31], 0x8000_0000, 0xFFFF_FFFF, templ[32:63])
// update accumulator
if A = 1 then 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 destination, corresponding half word pairs of signed integer elements in parameter **a** and unsigned integer elements in parameter **b** are multiplied producing a pair of 32-bit products. This pair of intermediate 32-bit products is added together, saturating if overflow or underflow occurs, and the sum is placed into the corresponding parameter **d** word. If A = 1, the result in parameter **d** is also placed into the accumulator

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

Figure 3-170. Vector Dot Product of Half Words, Add, Signed by Unsigned, Saturate, Integer (to Accumulator) (__ev_dotphasusi[a])

A	d	a	b	Maps to
A = 0	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evdotphasusi d,a,b**
A = 1	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evdotphasusia d,a,b**

__ev_dotphasusiaaw __ev_dotphasusiaaw

Vector Dot Product of Half Words, Add, Singed by Unsigned, Saturate, Integer and Accumulate into Words

d = __ev_dotphasusiaaw (a,b)

```
// high dot
temph1_{0:31} ← a_{0:15} ×_{sui} b_{0:15}; temph2_{0:31} ← a_{16:31} ×_{sui} b_{16:31}
temph_{0:63} ← EXTS(temph1_{0:31}) + EXTS(temph2_{0:31}) + EXTS(ACC_{0:31})
ovh ← temph_{31} ⊕ temph_{32}
d_{0:31} ← SATURATE(ovh, temph_{31}, 0x8000_0000, 0xFFFF_FFFF, temph_{32:63})
//low
templ1_{0:31} ← a_{32:47} ×_{sui} b_{32:47}; templ2_{0:31} ← a_{48:63} ×_{sui} b_{48:63}
templ_{0:63} ← EXTS(templ1_{0:31}) + EXTS(templ2_{0:31}) + EXTS(ACC_{32:63})
ovl ← templ_{31} ⊕ templ_{32}
d_{32:63} ← SATURATE(ovl, templ_{31}, 0x8000_0000, 0xFFFF_FFFF, templ_{32:63})
// 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 destination, corresponding half word pairs of signed integer elements in parameter **a** and unsigned integer elements in parameter **b** are multiplied producing a pair of 32-bit products. This pair of intermediate 32-bit products is added together with the contents of the corresponding accumulator word, saturating if overflow or underflow occurs, and the sum is placed into the corresponding parameter **d** and accumulator word.

Other registers altered: SPEFSCR, ACC

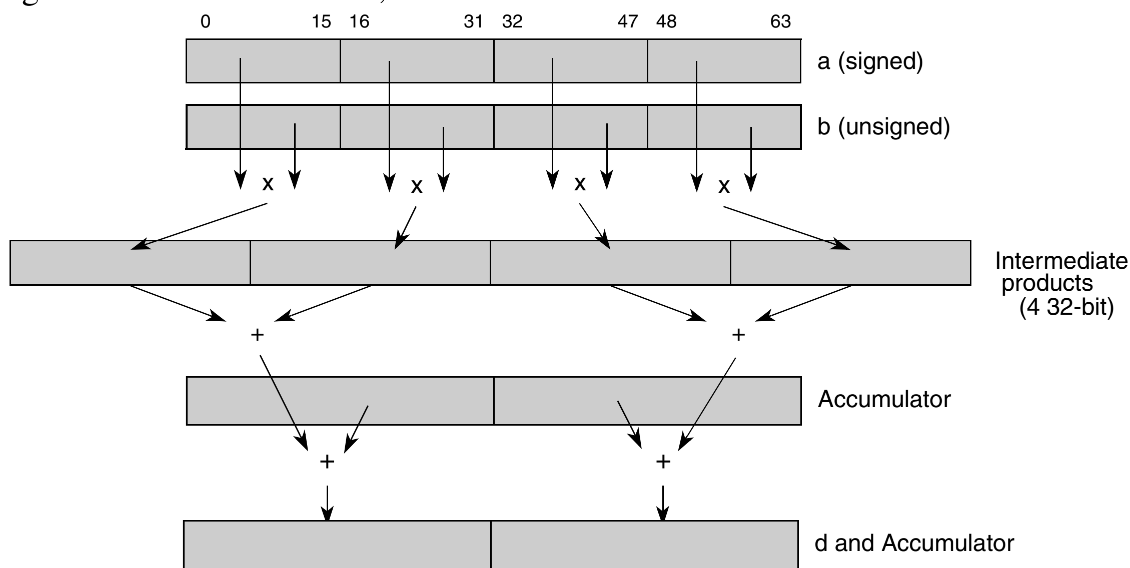


Figure 3-171. Vector Dot Product of Half Words, Add, Signed by Unsigned, Saturate, Integer and Accumulate (__ev_dotphasusiaaw)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evdotphasusiaaw d,a,b**

__ev_dotphasusiaaw3 __ev_dotphasusiaaw3

Vector Dot Product of Halfwords, Add, Signed by Unsigned, Saturate, Integer and Accumulate into Words, 3 operand

d = __ev_dotphasusiaaw3 (a,b,c)

```
// high dot
temph1_0:31 ←b_0:15 ×sui c_0:15; temph2_0:31 ←b_16:31 ×sui c_16:31
temph_0:63 ←EXTS_64(temph1_0:31) + EXTS_64(temph2_0:31) + EXTS_64(a_0:31)
ovh ←chk_ovf(temph_30:32)
d_0:31 ←SATURATE(ovh, temph_30, 0x8000_0000, 0x7FFF_FFFF, temph_32:63)
//low
templ1_0:31 ←b_32:47 ×sui c_32:47; templ2_0:31 ←b_48:63 ×sui c_48:63
templ_0:63 ←EXTS_64(templ1_0:31) + EXTS_64(templ2_0:31) + EXTS_64(a_32:63)
ovl ←chk_ovf(templ_30:32)
d_32:63 ←SATURATE(ovl, templ_30, 0x8000_0000, 0x7FFF_FFFF, templ_32:63)
// 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 destination, corresponding halfword pairs of signed integer elements in parameter **b** and unsigned integer elements in parameter **c** are multiplied producing a pair of 32-bit products. This pair of intermediate 32-bit products is added together with the contents of the corresponding parameter **a** word, saturating if overflow or underflow occurs, and the sum is placed into the corresponding parameter **d** and accumulator word.

Other registers altered: SPEFSCR, ACC

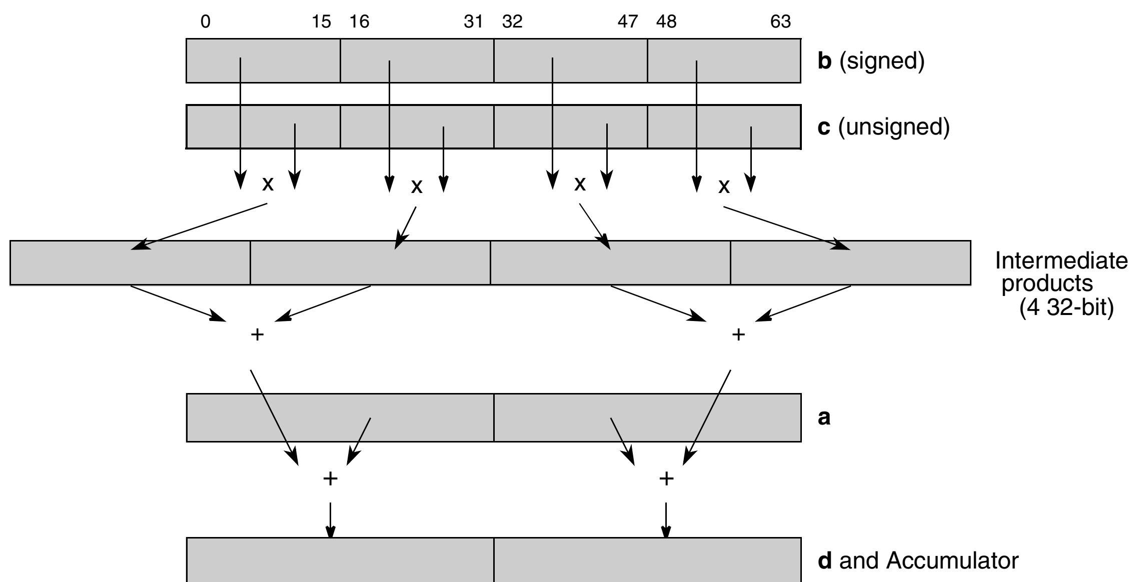


Figure 3-172. Vector Dot Product of Halfwords, Add, Signed by Unsigned, Saturate, Integer and Accumulate 3 op (__ev_dotphasusiaaw3)

d	a	b	c	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**d ←a** **evdotphasusiaaw3 d,b,c**

__ev_dotphaumi[a]

__ev_dotphaumi[a]

Vector Dot Product of Half Words, Add, Unsigned, Modulo, Integer (to Accumulator)

d = __ev_dotphaumi (a,b) (A = 0)

d = __ev_dotphaumia (a,b) (A = 1)

```
// high dot
temph1_{0:31} ← a_{0:15} ×_{ui} b_{0:15}
temph2_{0:31} ← a_{16:31} ×_{ui} b_{16:31}
temph_{0:31} ← temph1_{0:31} + temph2_{0:31} // modulo sum
d_{0:31} ← temph_{0:31}

//low
templ1_{0:31} ← a_{32:47} ×_{ui} b_{32:47}
templ2_{0:31} ← a_{48:63} ×_{ui} b_{48:63}
templ_{0:31} ← templ1_{0:31} + templ2_{0:31} // modulo sum
d_{32:63} ← templ_{0:31}

// update accumulator
if A = 1 then ACC_{0:63} ←d_{0:63}
```

For each word element in the destination, corresponding half word pairs of unsigned integer elements in parameters **a** and **b** are multiplied producing a pair of 32-bit products. This pair of intermediate 32-bit products is added together with a modulo sum, and the result is placed in parameter **d**. If A = 1, the result in parameter **d** is also placed into the accumulator.

Other registers altered: ACC (if A=1)

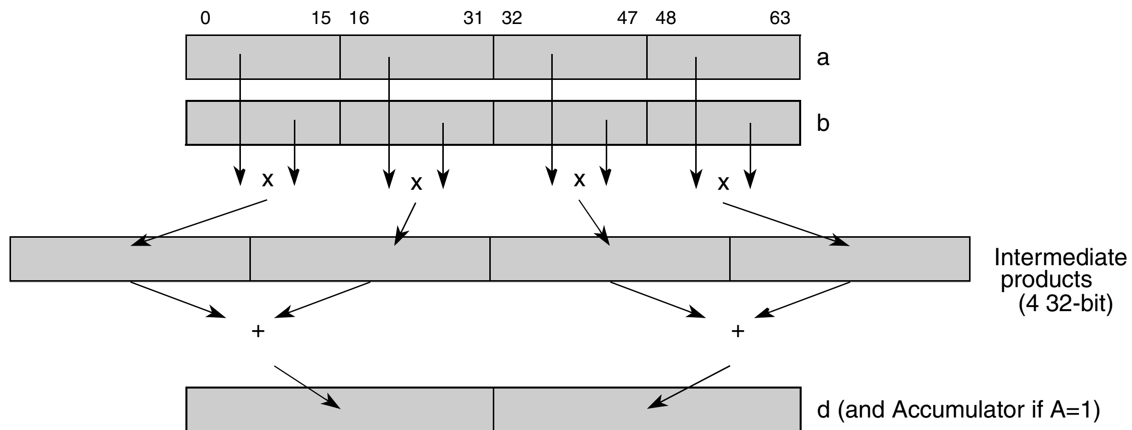


Figure 3-173. Vector Dot Product of Half Words, Add, Unsigned, Modulo, Integer (to Accumulator) (__ev_dotphaumi[a])

A	d	a	b	Maps to
A = 0	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evdotphaumi d,a,b**
A = 1	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evdotphaumia d,a,b**

__ev_dotphaumiaaw __ev_dotphaumiaaw

Vector Dot Product of Half Words, Add, Unsigned, Modulo, Integer and Accumulate into Words

d = __ev_dotphaumiaaw (a,b)

```
// high dot
temph1_{0:31} ← a_{0:15} ×ui b_{0:15}
temph2_{0:31} ← a_{16:31} ×ui b_{16:31}
temph_{0:31} ← temph1_{0:31} + temph2_{0:31} + ACC_{0:31} // modulo sum
d_{0:31} ← temph_{0:31}

//low
templ1_{0:31} ← a_{32:47} ×ui b_{32:47}
templ2_{0:31} ← a_{48:63} ×ui b_{48:63}
templ_{0:31} ← templ1_{0:31} + templ2_{0:31} + ACC_{32:63} // modulo sum
d_{32:63} ← templ_{0:31}

// update accumulator
ACC_{0:63} ←d_{0:63}
```

For each word element in the destination, corresponding half word pairs of unsigned integer elements in parameters **a** and **b** are multiplied producing a pair of 32-bit products. This pair of intermediate 32-bit products is added together with the contents of the corresponding accumulator word and the sum is placed into the corresponding parameter **d** and accumulator word.

Other registers altered: ACC

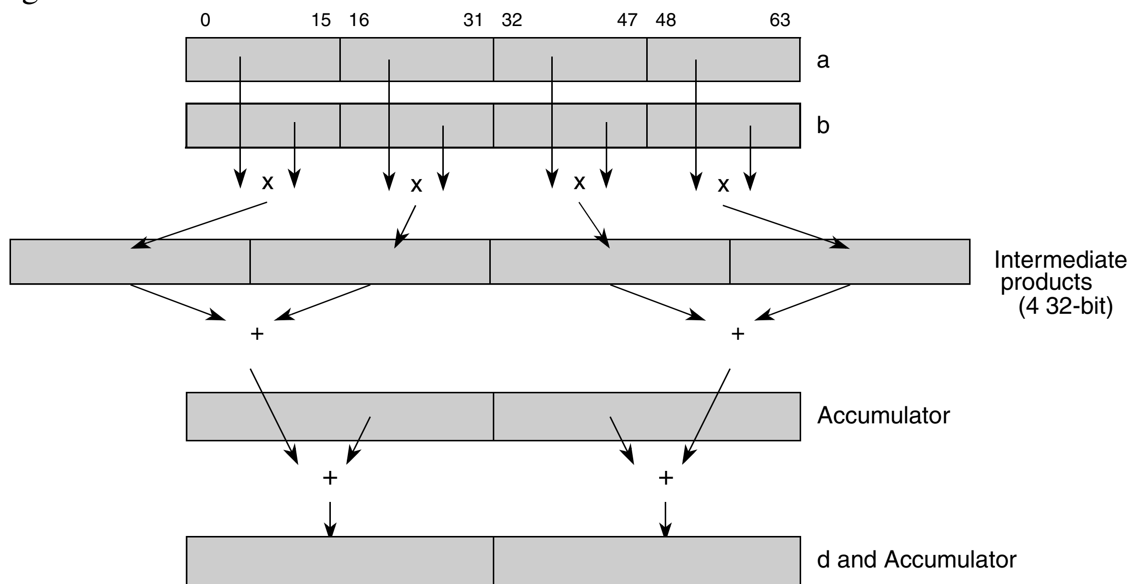


Figure 3-174. Vector Dot Product of Half Words, Add, Unsigned, Modulo, Integer and Accumulate Words (__ev_dotphaumiaaw)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evdotphaumiaaw d,a,b**

__ev_dotphaumiaaw3 __ev_dotphaumiaaw3

Vector Dot Product of Halfwords, Add, Unsigned, Modulo, Integer and Accumulate into Words, 3 operand

d = __ev_dotphaumiaaw3 (a,b,c)

```
// high dot
temph1_{0:31} ←b_{0:15} ×ui c_{0:15}
temph2_{0:31} ←b_{16:31} ×ui c_{16:31}
temph_{0:31} ←temph1_{0:31} + temph2_{0:31} + a_{0:31} // modulo sum
d_{0:31} ←temph_{0:31}

//low
templ1_{0:31} ←b_{32:47} ×ui c_{32:47}
templ2_{0:31} ←b_{48:63} ×ui c_{48:63}
templ_{0:31} ←templ1_{0:31} + templ2_{0:31} + a_{32:63} // modulo sum
d_{32:63} ←templ_{0:31}

// update accumulator
ACC_{0:63} ←d_{0:63}
```

For each word element in the destination, corresponding halfword pairs of unsigned integer elements in parameters **b** and **c** are multiplied producing a pair of 32-bit products. This pair of intermediate 32-bit products is added together with the contents of the corresponding parameter **a** word and the sum is placed into the corresponding parameter **d** and accumulator word.

Other registers altered: ACC

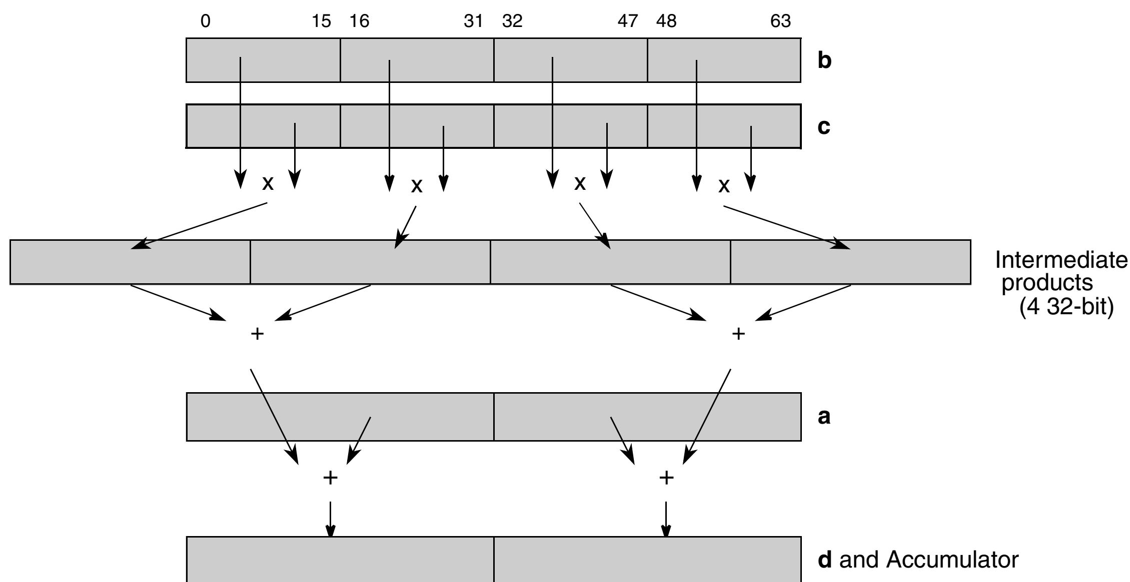


Figure 3-175. Vector Dot Product of Halfwords, Add, Unsigned, Modulo, Integer and Accumulate Words 3 op (__ev_dotphaumiaaw3)

d	a	b	c	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**d ←a** **evdotphaumiaaw3 d,b,c**

__ev_dotphausi[a] __ev_dotphausi[a]

Vector Dot Product of Half Words, Add, Unsigned, Saturate, Integer (to Accumulator)

d = __ev_dotphausi (a,b) (A = 0)

d = __ev_dotphausia (a,b) (A = 1)

```
// high dot
temph1[0:31] ← a[0:15] ×ui b[0:15]; temph2[0:31] ← a[16:31] ×ui b[16:31]
temph[0:63] ← EXTZ(temph1[0:31]) + EXTZ(temph2[0:31])
ovh ← temph[30] | temph[31]
d[0:31] ← SATURATE(ovh, 0, 0xFFFF_FFFF, 0xFFFF_FFFF, temph[32:63])
//low
templ1[0:31] ← a[32:47] ×ui b[32:47]; templ2[0:31] ← a[48:63] ×ui b[48:63]
templ[0:63] ← EXTZ(templ1[0:31]) + EXTZ(templ2[0:31])
ovl ← templ[30] | templ[31]
d[32:63] ← SATURATE(ovl, 0, 0xFFFF_FFFF, 0xFFFF_FFFF, templ[32:63])
// update accumulator
if A = 1 then 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 destination, corresponding half word pairs of unsigned integer elements in parameter **a** and parameter **b** are multiplied producing a pair of 32-bit products. This pair of intermediate 32-bit products is added together, saturating if overflow occurs, and the sum is placed into the corresponding parameter **d** word. If A = 1, the result in parameter **d** is also placed into the accumulator.

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

Figure 3-176. Vector Dot Product of Half Words, Add, Unsigned, Saturate, Integer [to Accumulator] (__ev_dotphausi[a])

A	d	a	b	Maps to
A = 0	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evdotphausi d,a,b**
A = 1	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evdotphausia d,a,b**

__ev_dotphausiaaw __ev_dotphausiaaw

Vector Dot Product of Half Words, Add, Unsigned, Saturate, Integer and Accumulate into Words

d = __ev_dotphausiaaw (a,b)

```
// high dot
temph1_0:31 ← a_0:15 ×ui b_0:15; temph2_0:31 ← a_16:31 ×ui b_16:31
temph_0:63 ← EXTZ(temph1_0:31) + EXTZ(temph2_0:31) + EXTZ(ACC_0:31)
ovh ← temph_30 | temph_31
d_0:31 ← SATURATE(ovh, 0, 0xFFFF_FFFF, 0xFFFF_FFFF, temph_32:63)
//low
templ1_0:31 ← a_32:47 ×ui b_32:47; templ2_0:31 ← a_48:63 ×ui b_48:63
templ_0:63 ← EXTZ(templ1_0:31) + EXTZ(templ2_0:31) + EXTZ(ACC_32:63)
ovl ← templ_30 | templ_31
d_32:63 ← SATURATE(ovl, 0, 0xFFFF_FFFF, 0xFFFF_FFFF, templ_32:63)
// 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 destination, corresponding half word pairs of unsigned integer elements in parameters **a** and **b** are multiplied producing a pair of 32-bit products. This pair of intermediate 32-bit products is added together with the contents of the corresponding accumulator word, saturating if overflow occurs, and the sum is placed into the corresponding parameter **d** and accumulator word.

Other registers altered: SPEFSCR, ACC

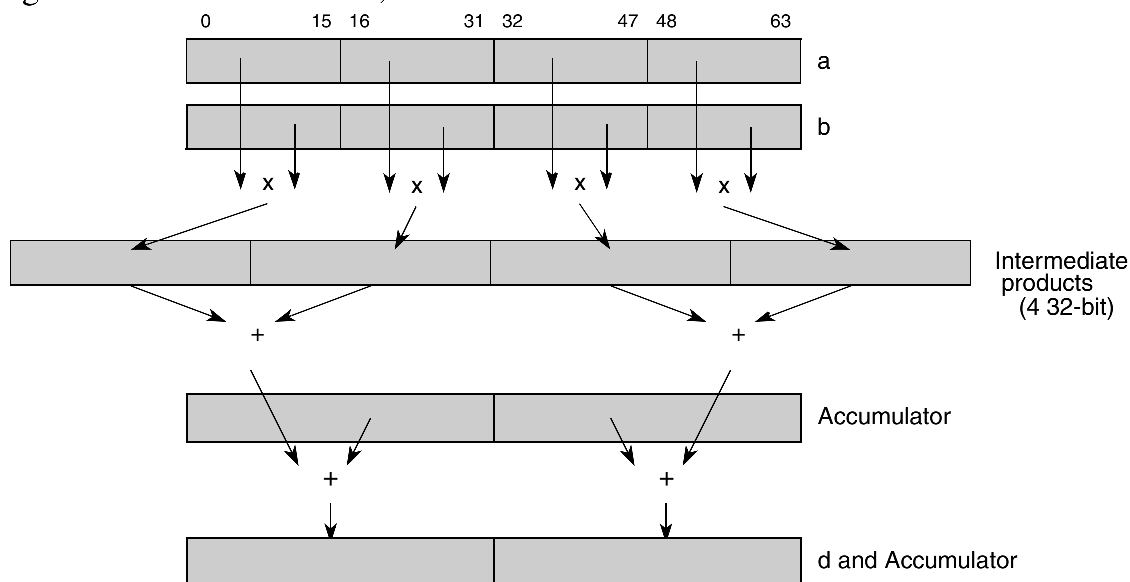


Figure 3-177. Vector Dot Product of Half Words, Add, Unsigned, Saturate, Integer and Accumulate Words (__ev_dotphausiaaw)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evdotphausiaaw d,a,b**

__ev_dotphausiaaw3 __ev_dotphausiaaw3

Vector Dot Product of Halfwords, Add, Unsigned, Saturate, Integer and Accumulate into Words, 3 operand

d = __ev_dotphausiaaw3 (a,b,c)

```
// high dot
temph1[0:31] ← b[0:15] ×ui c[0:15]; temph2[0:31] ← b[16:31] ×ui c[16:31]
temph[0:63] ← EXTZ64(temph1[0:31]) + EXTZ64(temph2[0:31]) + EXTZ64(a[0:31])
ovh ← temph[30] | temph[31]
d[0:31] ← SATURATE(ovh, 0, 0xFFFF_FFFF, 0xFFFF_FFFF, temph[32:63])
//low
templ1[0:31] ← b[32:47] ×ui c[32:47]; templ2[0:31] ← b[48:63] ×ui c[48:63]
templ[0:63] ← EXTZ64(templ1[0:31]) + EXTZ64(templ2[0:31]) + EXTZ64(a[32:63])
ovl ← templ[30] | templ[31]
d[32:63] ← SATURATE(ovl, 0, 0xFFFF_FFFF, 0xFFFF_FFFF, templ[32:63])
// 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 destination, corresponding halfword pairs of unsigned integer elements in parameters **b** and **c** are multiplied producing a pair of 32-bit products. This pair of intermediate 32-bit products is added together with the contents of the corresponding parameter **a** word, saturating if overflow occurs, and the sum is placed into the corresponding parameter **d** and accumulator word.

Other registers altered: SPEFSCR, ACC

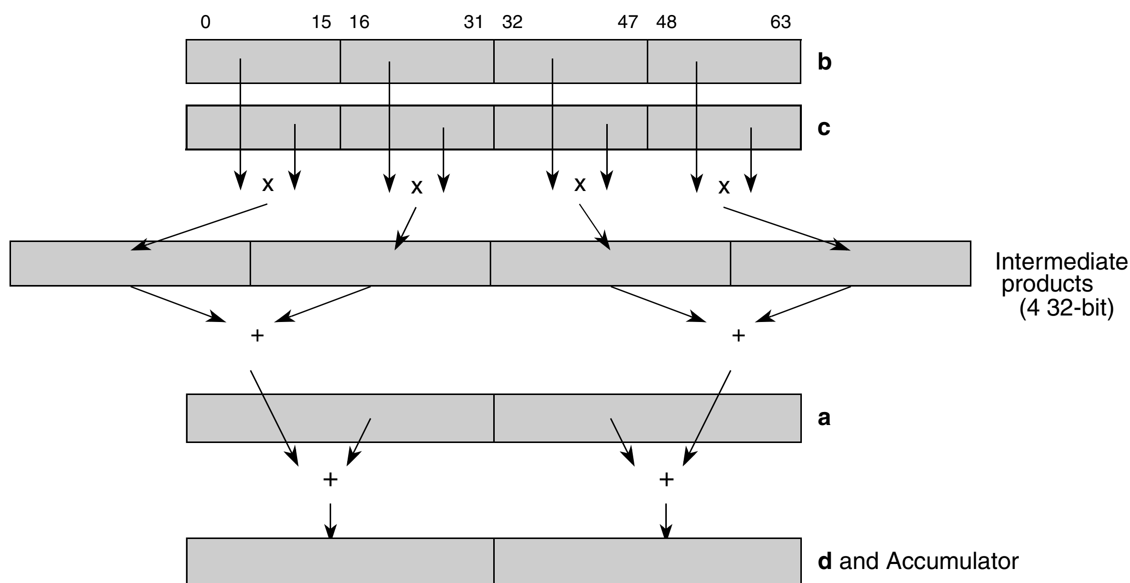


Figure 3-178. Vector Dot Product of Halfwords, Add, Unsigned, Saturate, Integer and Accumulate Words 3 op (__ev_dotphausiaaw3)

d	a	b	c	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**d ←a** **evdotphausiaaw3 d,b,c**

__ev_dotphihcsmi[a]   __ev_dotphihcsmi[a]

Vector Dot Product of High Halfwords, Complex, Signed, Modulo, Integer (to Accumulator)

d = __ev_dotphihcsmi (a,b) (A = 0)

d = __ev_dotphihcsmia (a,b) (A = 1)

```
// high dot - calculate real part of complex product
temph1_0:31 ← a_0:15 ×si b_0:15
temph2_0:31 ← a_16:31 ×si b_16:31
temph_0:31 ← temph1_0:31 - temph2_0:31 // modulo difference
d_0:31 ← temph_0:31

//low dot - calculate imaginary part of complex product
templ1_0:31 ← b_0:15 ×si a_16:31
templ2_0:31 ← a_0:15 ×si b_16:31
templ_0:31 ← templ1_0:31 + templ2_0:31 // modulo sum
d_32:63 ← templ_0:31

// update accumulator
if A = 1 then ACC_0:63 ←d_0:63
```

For the high word element in the destination, corresponding halfword pairs of signed integer elements in the high halfwords of parameters **a** and **b** are multiplied producing a pair of 32-bit products. The difference of this pair of intermediate 32-bit products is placed into the high word of parameter **d**. For the low word element in the destination, halfword pairs of signed integer elements from the high halfwords of parameters **a** and **b** are multiplied after exchanging the high halfwords of parameter **a**, producing a pair of 32-bit products. The sum of this pair of intermediate 32-bit products is placed into the low word of parameter **d**. If A = 1, the result in parameter **d** is also placed into the accumulator.

Other registers altered: ACC (if A=1)

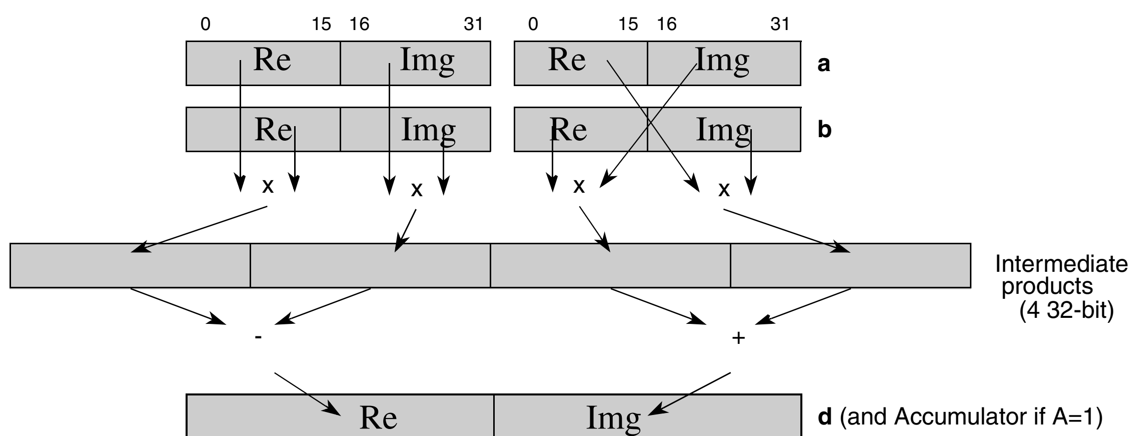


Figure 3-179. Vector Dot Product of High Halfwords, Complex, Signed, Modulo, Integer (to Accumulator) (__ev_dotphihcsmi[a])

A	d	a	b	Maps to
A = 0	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evdotphihcsmi d,a,b**
A = 1	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evdotphihcsmia d,a,b**

__ev_dotphihcsmiaaw __ev_dotphihcsmiaaw

Vector Dot Product of High Halfwords, Complex, Signed, Modulo, Integer and Accumulate into Words

d = __ev_dotphihcsmiaaw (a,b)

```
// high dot - calculate real part of complex product
temph1_{0:31} ← a_{0:15} ×_{si} b_{0:15}
temph2_{0:31} ← a_{16:31} ×_{si} b_{16:31}
temph_{0:31} ← temph1_{0:31} - temph2_{0:31} + ACC_{0:31} // modulo sum
d_{0:31} ← temph_{0:31}

//low dot - calculate imaginary part of complex product
templ1_{0:31} ← b_{0:15} ×_{si} a_{16:31}
templ2_{0:31} ← a_{0:15} ×_{si} b_{16:31}
templ_{0:31} ← templ1_{0:31} + templ2_{0:31} + ACC_{32:63} // modulo sum
d_{32:63} ← templ_{0:31}

// update accumulator
ACC_{0:63} ←d_{0:63}
```

For the high word element in the destination, corresponding halfword pairs of signed integer elements in the high halfwords of parameters **a** and **b** are multiplied producing a pair of 32-bit products. The difference of this pair of intermediate 32-bit products is added together with the contents of the upper accumulator word and the sum is placed into the high word of parameter **d**. For the low word element in the destination, halfword pairs of signed integer elements from the high halfwords of parameters **a** and **b** are multiplied after exchanging the high halfwords of parameter **a**, producing a pair of 32-bit products. The sum of this pair of intermediate 32-bit products is added together with the contents of the low accumulator word and the sum is placed into the low word of parameter **d** and the accumulator.

Other registers altered: ACC

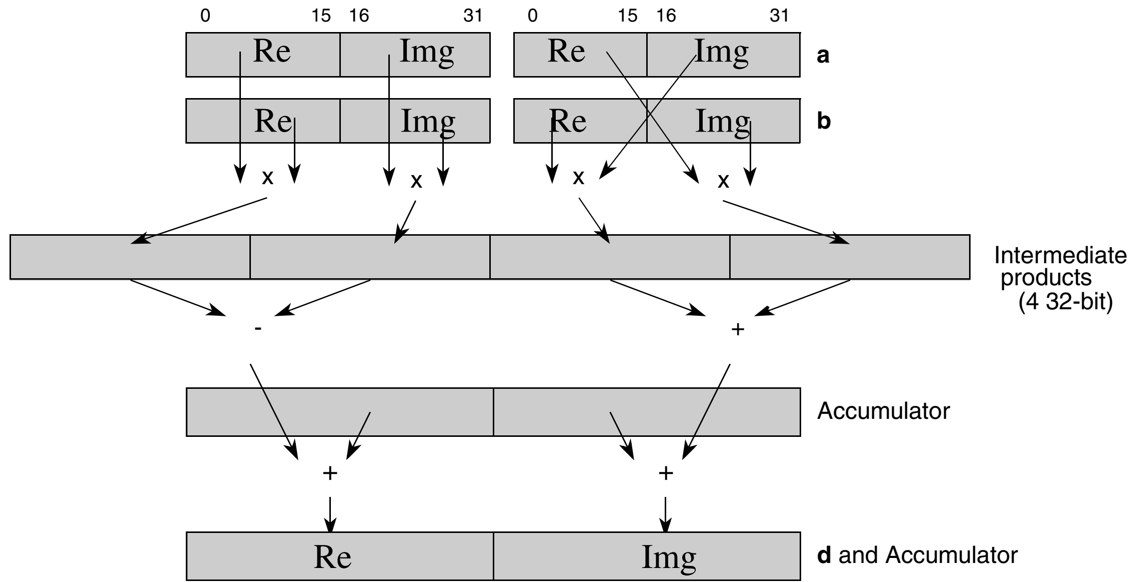


Figure 3-180. Vector Dot Product of High Halfwords, Complex, Signed, Modulo, Integer and Accumulate Words (__ev_dotphihcsmiaaw)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evdotphihcsmiaaw d,a,b**

__ev_dotphihcsmiaaw3 __ev_dotphihcsmiaaw3

Vector Dot Product of High Halfwords, Complex, Signed, Modulo, Integer and Accumulate into Words, 3 operand

d = __ev_dotphihcsmiaaw3 (a,b,c)

```
// high dot - calculate real part of complex product
temph1_{0:31} ← b_{0:15} ×_{si} c_{0:15}
temph2_{0:31} ← b_{16:31} ×_{si} c_{16:31}
temph_{0:31} ← temph1_{0:31} - temph2_{0:31} + a_{0:31} // modulo sum
d_{0:31} ← temph_{0:31}

//low dot - calculate imaginary part of complex product
templ1_{0:31} ← c_{0:15} ×_{si} b_{16:31}
templ2_{0:31} ← b_{0:15} ×_{si} c_{16:31}
templ_{0:31} ← templ1_{0:31} + templ2_{0:31} + a_{32:63} // modulo sum
d_{32:63} ← templ_{0:31}

// update accumulator
ACC_{0:63} ←d_{0:63}
```

For the high word element in the destination, corresponding halfword pairs of signed integer elements in the high halfwords of parameters **b** and **c** are multiplied producing a pair of 32-bit products. The difference of this pair of intermediate 32-bit products is added together with the contents of the upper parameter **a** word and the sum is placed into the high word of parameter **a**. For the low word element in the destination, halfword pairs of signed integer elements from the high halfwords of parameters **b** and **c** are multiplied after exchanging the high halfwords of parameter **b**, producing a pair of 32-bit products. The sum of this pair of intermediate 32-bit products is added together with the contents of the lower parameter **a** word and the sum is placed into the low word of parameter **d** and the accumulator.

Other registers altered: ACC

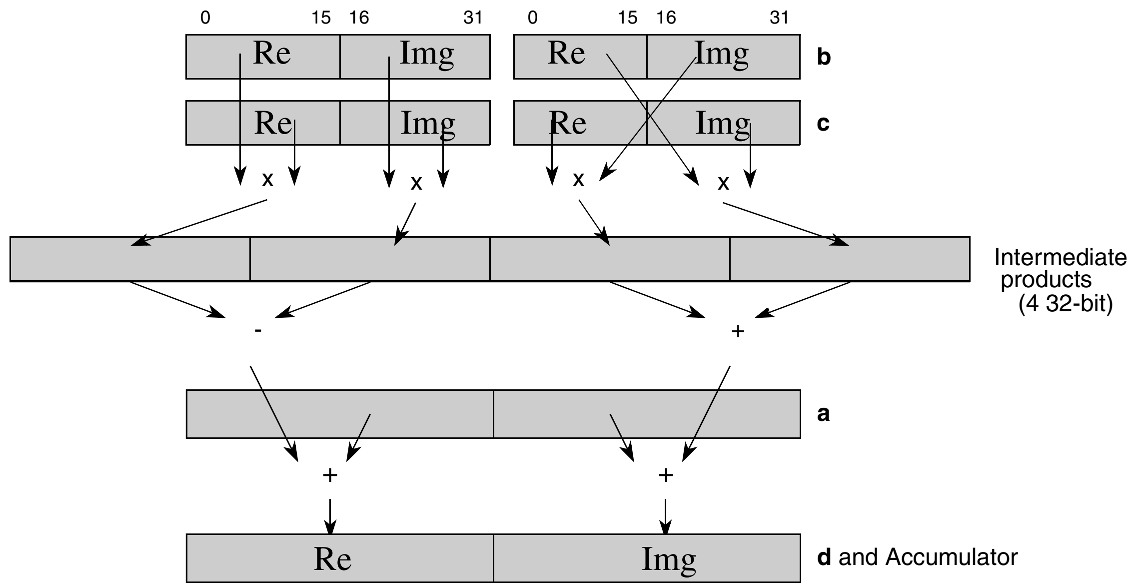


Figure 3-181. Vector Dot Product of High Halfwords, Complex, Signed, Modulo, Integer and Accumulate Words 3 op (__ev_dotphihcsmiaaw3)

d	a	b	c	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**d ←a evdotphihcsmiaaw3 d,b,c**

__ev_dotphihcssf[a] __ev_dotphihcssf[a]

Vector Dot Product of High Halfwords, Complex, Signed, Saturate, Fractional (to Accumulator)

d = __ev_dotphihcssf (a,b) (A = 0)

d = __ev_dotphihcssfa (a,b) (A = 1)

```
// high dot
temph1_0:31 ← a_0:15 ×sf b_0:15
if (a_0:15 = 0x8000) & (b_0:15 = 0x8000) then
    temph1_0:31 ←0x7FFF_FFFF //saturate
    movh ←1
else
    movh ←0
temph2_0:31 ← a_16:31 ×sf b_16:31
if (a_16:31 = 0x8000) & (b_16:31 = 0x8000) then
    temph2_0:31 ←0x7FFF_FFFF //saturate
    movh ←1
endif
temph_0:63 ← EXTS_64(temph1_0:31) - EXTS_64(temph2_0:31)
ovh ← temph_31 ⊕ temph_32
d_0:31 ← SATURATE(ovh, temph_31, 0x8000_0000, 0x7FFF_FFFF, temph_32:63)

//low
templ1_0:31 ← a_16:31 ×sf b_0:15
if (a_16:31 = 0x8000) & (b_0:15 = 0x8000) then
    templ1_0:31 ←0x7FFF_FFFF //saturate
    movl ←1
else
    movl ←0
templ2_0:31 ← a_0:15 ×sf b_16:31
if (a_0:15 = 0x8000) & (b_16:31 = 0x8000) then
    templ2_0:31 ←0x7FFF_FFFF //saturate
    movl ←1
endif
templ_0:63 ← EXTS_64(templ1_0:31) + EXTS_64(templ2_0:31)
ovl ← templ_31 ⊕ templ_32
d_32:63 ← SATURATE(ovl, templ_31, 0x8000_0000, 0x7FFF_FFFF, templ_32:63)
// update accumulator
if A = 1 then ACC_0:63 ←d_0:63
// update SPEFSCR
SPEFSCR_OVH ← movh | ovh; SPEFSCR_OV ← movl | ovl
SPEFSCR_SOVH ← SPEFSCR_SOVH | movh | ovh; SPEFSCR_SOV ← SPEFSCR_SOV | movl | ovl
```

For the high word element in the destination, corresponding halfword pairs of signed fractional elements in the high halfwords of parameters **a** and **b** are multiplied producing a pair of 32-bit products. If both inputs of a multiply are -1.0, the result saturates to 0x7FFF_FFFF. The difference of this pair of intermediate 32-bit products is placed into the high word of parameter **d**, saturating if overflow or underflow occurs. For the low word element in the destination, halfword pairs of signed fractional elements from the high halfwords of parameters **a** and **b** are multiplied after exchanging the high halfwords of parameter **a**, producing a pair of 32-bit products. If both inputs of a multiply are -1.0, the result saturates to 0x7FFF_FFFF. The sum of this pair of intermediate 32-bit products is placed into the low word of parameter **d**, saturating if overflow or underflow occurs. If A = 1, the result in parameter **d** is also placed into the accumulator

The overflow and summary overflow bits are recorded in the SPEFSCR based on an overflow or underflow from either the multiply or the accumulation.

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

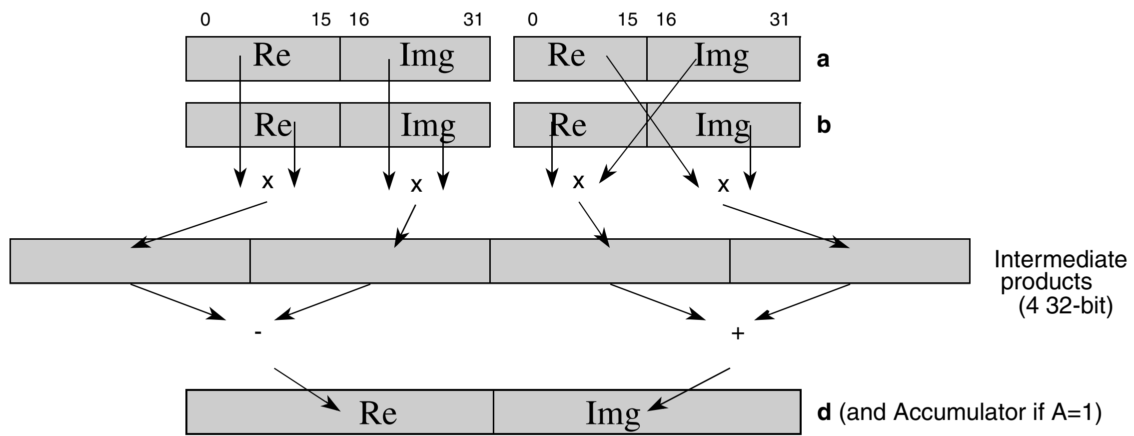


Figure 3-182. Vector Dot Product of High Halfwords, Complex, Signed, Saturate, Fractional (to Accumulator) (__ev_dotphihcssf[a])

A	d	a	b	Maps to
A = 0	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evdotphihcssf d,a,b**
A = 1	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evdotphihcssfa d,a,b**

__ev_dotphihcssfaaw __ev_dotphihcssfaaw

Vector Dot Product of High Halfwords, Complex, Signed, Saturate, Fractional and Accumulate into Words

d = __ev_dotphihcssfaaw (a,b)

```
// high dot
temph1[0:31] ← a[0:15] ×sf b[0:15]
if (a[0:15] = 0x8000) & (b[0:15] = 0x8000) then
    temph1[0:31] ←0x7FFF_FFFF //saturate
    movh ←1
else
    movh ←0
temph2[0:31] ← a[16:31] ×sf b[16:31]
if (a[16:31] = 0x8000) & (b[16:31] = 0x8000) then
    temph2[0:31] ←0x7FFF_FFFF //saturate
    movh ←1
endif
temph[0:63] ← EXTS64(temph1[0:31]) - EXTS64(temph2[0:31]) + EXTS64(ACC[0:31])
ovh ← chk_ovf(temph[30:32])
d[0:31] ← SATURATE(ovh, temph[30], 0x8000_0000, 0x7FFF_FFFF, temph[32:63])

//low
templ1[0:31] ← a[16:31] ×sf b[0:15]
if (a[16:31] = 0x8000) & (b[0:15] = 0x8000) then
    templ1[0:31] ←0x7FFF_FFFF //saturate
    movl ←1
else
    movl ←0
templ2[0:31] ← a[0:15] ×sf b[16:31]
if (a[0:15] = 0x8000) & (b[16:31] = 0x8000) then
    templ2[0:31] ←0x7FFF_FFFF //saturate
    movl ←1
endif
templ[0:63] ← EXTS64(templ1[0:31]) + EXTS64(templ2[0:31]) + EXTS64(ACC[32:63])
ovl ← chk_ovf(templ[30:32])
d[32:63] ← SATURATE(ovl, templ[30], 0x8000_0000, 0x7FFF_FFFF, templ[32:63])
// update accumulator
ACC[0:63] ← d[0:63]
// update SPEFSCR
SPEFSCR[OVH] ← movh | ovh; SPEFSCR[OV] ← movl | ovl
SPEFSCR[SOVH] ← SPEFSCR[SOVH] | movh | ovh; SPEFSCR[SOV] ← SPEFSCR[SOV] | movl | ovl
```

For the high word element in the destination, corresponding halfword pairs of signed fractional elements in the high halfwords of parameters **a** and **b** are multiplied producing a pair of 32-bit products. If both inputs of a multiply are -1.0, the result saturates to 0x7FFF_FFFF. The difference of this pair of intermediate 32-bit products is added to the high word of the accumulator, and the result is placed into the high word of parameter **d** and the accumulator, saturating if overflow or underflow occurs. For the low word element in the destination, halfword pairs of signed fractional elements from the high halfwords of parameters **a** and **b** are multiplied after exchanging the high halfwords of parameter **a**, producing a pair of 32-bit products. If both inputs of a multiply are -1.0, the result saturates to 0x7FFF_FFFF. The sum of this pair of intermediate 32-bit products and the low word of the accumulator is placed into the low word of parameter **d** and the accumulator.

The overflow and summary overflow bits are recorded in the SPEFSCR based on an overflow or underflow from either the multiply or the accumulation.

Other registers altered: SPEFSCR, ACC

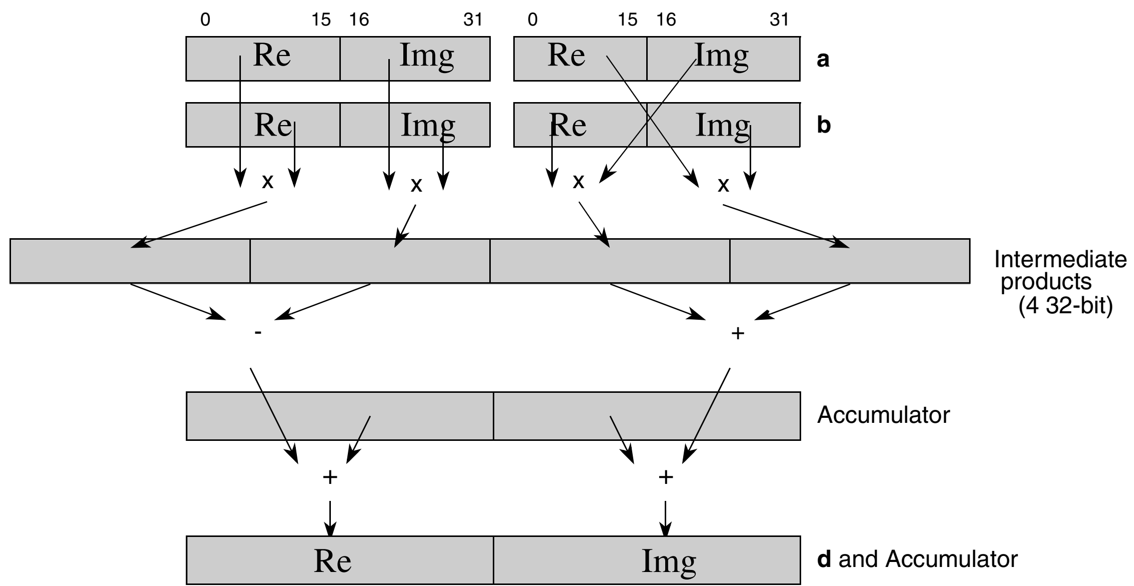


Figure 3-183. Vector Dot Product of High Halfwords, Complex, Signed, Saturate, Fractional and Accumulate Words (__ev_dotphihcssfaaw)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evdotphihcssfaaw d,a,b**

__ev_dotphihcssfaaw3 __ev_dotphihcssfaaw3

Vector Dot Product of High Halfwords, Complex, Signed, Saturate, Fractional and Accumulate into Words, 3 operand

d = __ev_dotphihcssfaaw3 (a,b,c)

```
// high dot
temph1_{0:31} ← b_{0:15} ×_{sf} c_{0:15}
if (b_{0:15} = 0x8000) & (c_{0:15} = 0x8000) then
    temph1_{0:31} ←0x7FFF_FFFF //saturate
    movh ←1
else
    movh ←0
temph2_{0:31} ← b_{16:31} ×_{sf} c_{16:31}
if (b_{16:31} = 0x8000) & (c_{16:31} = 0x8000) then
    temph2_{0:31} ←0x7FFF_FFFF //saturate
    movh ←1
endif
temph_{0:63} ←EXTS_{64}(temph1_{0:31}) - EXTS_{64}(temph2_{0:31}) + EXTS_{64}(a_{0:31})
ovh ←chk_ovf(temph_{30:32})
d_{0:31} ←SATURATE(ovh, temph_{30}, 0x8000_0000, 0x7FFF_FFFF, temph_{32:63})

//low
templ1_{0:31} ←b_{16:31} ×_{sf} c_{0:15}
if (b_{16:31} = 0x8000) & (c_{0:15} = 0x8000) then
    templ1_{0:31} ←0x7FFF_FFFF //saturate
    movl ←1
else
    movl ←0
templ2_{0:31} ←b_{0:15} ×_{sf} c_{16:31}
if (b_{0:15} = 0x8000) & (c_{16:31} = 0x8000) then
    templ2_{0:31} ←0x7FFF_FFFF //saturate
    movl ←1
endif
templ_{0:63} ←EXTS_{64}(templ1_{0:31}) + EXTS_{64}(templ2_{0:31}) + EXTS_{64}(a_{32:63})
ovl ←chk_ovf(templ_{30:32})
d_{32:63} ←SATURATE(ovl, templ_{30}, 0x8000_0000, 0x7FFF_FFFF, templ_{32:63})
// update accumulator
ACC_{0:63} ←d_{0:63}
// update SPEFSCR
SPEFSCR_{OVH} ←movh | ovh; SPEFSCR_{OV} ←movl | ovl
SPEFSCR_{SOVH} ←SPEFSCR_{SOVH} | movh | ovh; SPEFSCR_{SOV} ←SPEFSCR_{SOV} | movl | ovl
```

For the high word element in the destination, corresponding halfword pairs of signed fractional elements in the high halfwords of parameters **b** and **c** are multiplied producing a pair of 32-bit products. If both inputs of a multiply are -1.0, the result saturates to 0x7FFF_FFFF. The difference of this pair of intermediate 32-bit products is added to the high word of parameter **a**, and the result is placed into the high word of parameter **d** and the accumulator, saturating if overflow or underflow occurs. For the low word element in the destination, halfword pairs of signed fractional elements from the high halfwords of parameters **b** and **c** are multiplied after exchanging the high halfwords of parameter **b**, producing a pair of 32-bit products. If both inputs of a multiply are -1.0, the result saturates to 0x7FFF_FFFF. The sum of this pair of intermediate 32-bit products and the low word of parameter **a** is placed into the low word of parameter **d** and the accumulator.

The overflow and summary overflow bits are recorded in the SPEFSCR based on an overflow or underflow from either the multiply or the accumulation.

Other registers altered: SPEFSCR, ACC

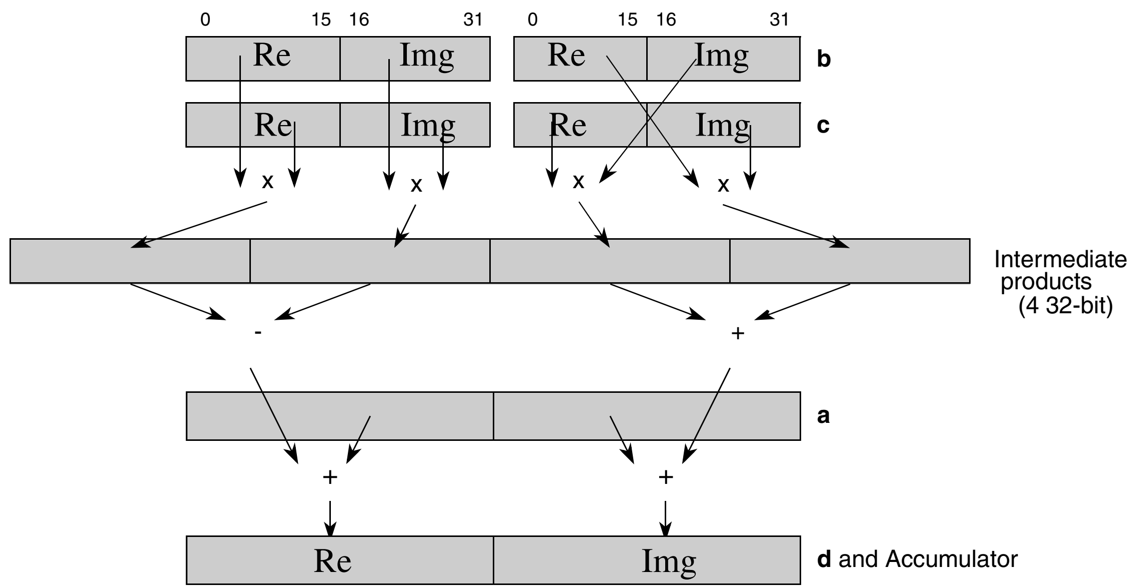


Figure 3-184. Vector Dot Product of High Halfwords, Complex, Signed, Saturate, Fractional and Accumulate Words 3 op (__ev_dotphihcssfaaw3)

d	a	b	c	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	d ←a evdotphihcssfaaw3 d,b,c

__ev_dotphihcssfr[a] __ev_dotphihcssfr[a]

Vector Dot Product of High Halfwords, Complex, Signed, Saturate, Fractional, Round (to Accumulator)

d = __ev_dotphihcssfr (a,b) (A = 0)

d = __ev_dotphihcssfra (a,b) (A = 1)

```
// high dot
temph1[0:31] ← a[0:15] ×sf b[0:15]
if (a[0:15] = 0x8000) & (b[0:15] = 0x8000) then
    temph1[0:31] ←0x7FFF_FFFF //saturate
    movh ←1
else
    movh ←0
temph2[0:31] ← a[16:31] ×sf b[16:31]
if (a[16:31] = 0x8000) & (b[16:31] = 0x8000) then
    temph2[0:31] ←0x7FFF_FFFF //saturate
    movh ←1
endif
temph[0:63] ← ROUND((EXTS64(temph1[0:31]) - EXTS64(temph2[0:31])),16)
ovh ← chk_ovf(temph[30:32])
d[0:31] ← SATURATE(ovh, temph[30], 0x8000_0000, 0x7FFF_0000, temph[32:63])

//low
templ1[0:31] ← a[16:31] ×sf b[0:15]
if (a[16:31] = 0x8000) & (b[0:15] = 0x8000) then
    templ1[0:31] ←0x7FFF_FFFF //saturate
    movl ←1
else
    movl ←0
templ2[0:31] ← a[0:15] ×sf b[16:31]
if (a[0:15] = 0x8000) & (b[16:31] = 0x8000) then
    templ2[0:31] ←0x7FFF_FFFF //saturate
    movl ←1
endif
templ[0:63] ← ROUND((EXTS64(templ1[0:31]) + EXTS64(templ2[0:31])),16)
ovl ← chk_ovf(templ[30:32])
d[32:63] ← SATURATE(ovl, templ[30], 0x8000_0000, 0x7FFF_0000, templ[32:63])
// update accumulator
if A = 1 then ACC[0:63] ←d[0:63]
// update SPEFSCR
SPEFSCR[OVH] ← movh | ovh; SPEFSCR[OV] ← movl | ovl
SPEFSCR[SOVH] ← SPEFSCR[SOVH] | movh | ovh; SPEFSCR[SOV] ← SPEFSCR[SOV] | movl | ovl
```

For the high word element in the destination, corresponding halfword pairs of signed fractional elements in the high halfwords of parameters **a** and **b** are multiplied producing a pair of 32-bit products. If both inputs of a multiply are -1.0, the result saturates to 0x7FFF_FFFF. The difference of this pair of intermediate 32-bit products is rounded to 16 bits using the current fractional rounding mode in SPEFCSR, saturating if overflow or underflow occurs, and placed into the high word of parameter **d**. For the low word element in the destination, halfword pairs of signed fractional elements from the high halfwords of parameters **a** and **b** are multiplied after exchanging the high halfwords of parameter **a**, producing a pair of 32-bit products. If both inputs of a multiply are -1.0, the result saturates to 0x7FFF_FFFF. The sum of this pair of intermediate 32-bit products is rounded to 16 bits using the current fractional rounding mode in SPEFCSR, saturating if overflow or underflow occurs, and placed into the low word of parameter **d**. If A = 1, the result in parameter **d** is also placed into the accumulator

The overflow and summary overflow bits are recorded in the SPEFSCR based on an overflow or underflow from either the multiply or the accumulation.

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

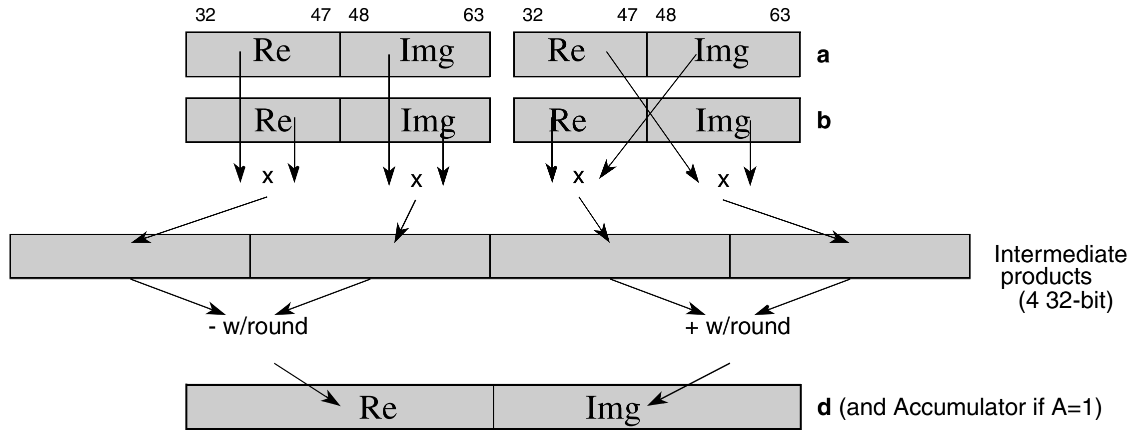


Figure 3-185. Vector Dot Product of High Halfwords, Complex, Signed, Saturate, Fractional, Round (to Accumulator) (__ev_dotphihcssfr[a])

A	d	a	b	Maps to
A = 0	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evdotphihcssfr d,a,b**
A = 1	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evdotphihcssfra d,a,b**

__ev_dotphihcssfraaw __ev_dotphihcssfraaw

Vector Dot Product of High Halfwords, Complex, Signed, Saturate, Fractional, Round and Accumulate into Words

d = __ev_dotphihcssfraaw (a,b)

```
// high dot
temph1[0:31] ← a[0:15] ×sf b[0:15]
if (a[0:15] = 0x8000) & (b[0:15] = 0x8000) then
    temph1[0:31] ←0x7FFF_FFFF //saturate
    movh ←1
else
    movh ←0
temph2[0:31] ← a[16:31] ×sf b[16:31]
if (a[16:31] = 0x8000) & (b[16:31] = 0x8000) then
    temph2[0:31] ←0x7FFF_FFFF //saturate
    movh ←1
endif
temph[0:63] ← ROUND((EXTS64(temph1[0:31]) - EXTS64(temph2[0:31]) + EXTS64(ACC[0:31])),16)
ovh ← chk_ovf(temph[30:32])
d[0:31] ← SATURATE(ovh, temph[30], 0x8000_0000, 0x7FFF_0000, temph[32:63])

//low
templ1[0:31] ← a[16:31] ×sf b[0:15]
if (a[16:31] = 0x8000) & (b[0:15] = 0x8000) then
    templ1[0:31] ←0x7FFF_FFFF //saturate
    movl ←1
else
    movl ←0
templ2[0:31] ← a[0:15] ×sf b[16:31]
if (a[0:15] = 0x8000) & (b[16:31] = 0x8000) then
    templ2[0:31] ←0x7FFF_FFFF //saturate
    movl ←1
endif
templ[0:63] ← ROUND((EXTS64(templ1[0:31]) + EXTS64(templ2[0:31]) + EXTS64(ACC[32:63])),16)
ovl ← chk_ovf(templ[30:32])
d[32:63] ← SATURATE(ovl, templ[30], 0x8000_0000, 0x7FFF_0000, templ[32:63])
// update accumulator
ACC[0:63] ← d[0:63]
// update SPEFSCR
SPEFSCR[OVH] ← movh | ovh; SPEFSCR[OV] ← movl | ovl
SPEFSCR[SOVH] ← SPEFSCR[SOVH] | movh | ovh; SPEFSCR[SOV] ← SPEFSCR[SOV] | movl | ovl
```

For the high word element in the destination, corresponding halfword pairs of signed fractional elements in the high halfwords of parameters **a** and **b** are multiplied producing a pair of 32-bit products. If both inputs of a multiply are -1.0, the result saturates to 0x7FFF_FFFF. The difference of this pair of intermediate 32-bit products is added to the high word of the accumulator and rounded to 16 bits using the current fractional rounding mode in SPEFCSR, saturating if overflow or underflow occurs, and placed into the high word of parameter **d** and the accumulator. For the low word element in the destination, halfword pairs of signed fractional elements from the high halfwords of parameters **a** and **b** are multiplied after exchanging the high halfwords of parameter **a**, producing a pair of 32-bit products. If both inputs of a multiply are -1.0, the result saturates to 0x7FFF_FFFF. The sum of this pair of intermediate 32-bit products and the low word of the accumulator is rounded to 16 bits using the current fractional rounding mode in SPEFCSR, saturating if overflow or underflow occurs, and placed into the low word of parameter **d** and the accumulator.

The overflow and summary overflow bits are recorded in the SPEFSCR based on an overflow or underflow from either the multiply or the accumulation.

Other registers altered: SPEFSCR, ACC

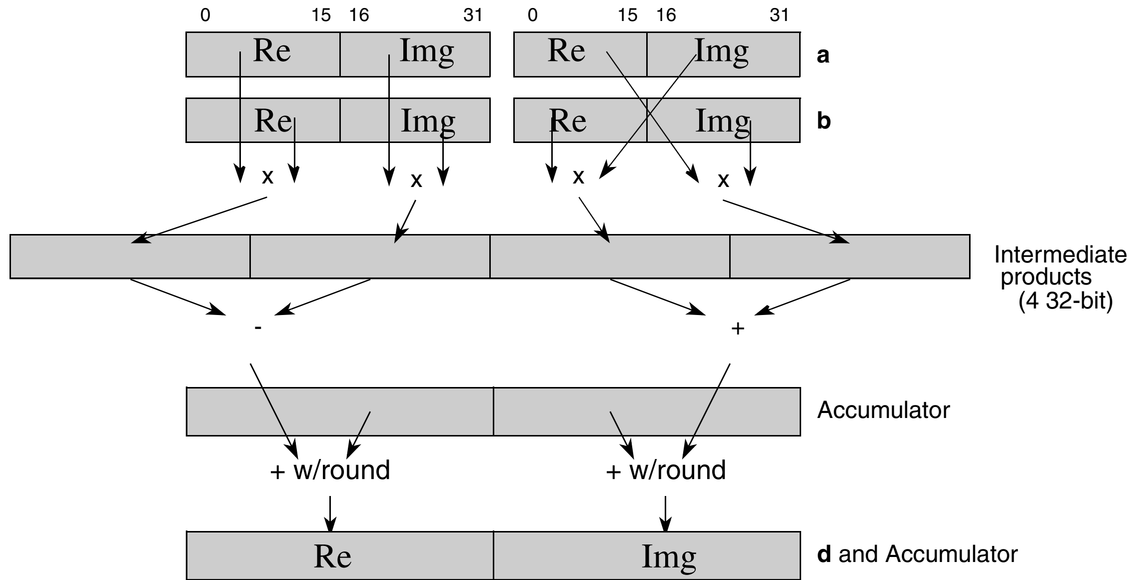


Figure 3-186. Vector Dot Product of High Halfwords, Complex, Signed, Saturate, Fractional, Round and Accumulate Words (__ev_dotphihcssfraaw)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evdotphihcssfraaw d,a,b**

__ev_dotphihcssfraaw3 __ev_dotphihcssfraaw3

Vector Dot Product of High Halfwords, Complex, Signed, Saturate, Fractional, Round and Accumulate into Words, 3 operand

d = __ev_dotphihcssfraaw3 (a,b,c)

```
// high dot
temph1_{0:31} ←b_{0:15} ×sf c_{0:15}
if (b_{0:15} = 0x8000) & (c_{0:15} = 0x8000) then
    temph1_{0:31} ←0x7FFF_FFFF //saturate
    movh ←1
else
    movh ←0
temph2_{0:31} ←b_{16:31} ×sf c_{16:31}
if (b_{16:31} = 0x8000) & (c_{16:31} = 0x8000) then
    temph2_{0:31} ←0x7FFF_FFFF //saturate
    movh ←1
endif
temph_{0:63} ←ROUND((EXTS_{64}(temph1_{0:31}) - EXTS_{64}(temph2_{0:31}) + EXTS_{64}(a_{0:31})),16)
ovh ←chk_ovf(temph_{30:32})
d_{0:31} ←SATURATE(ovh, temph_{30}, 0x8000_0000, 0x7FFF_0000, temph_{32:63})

//low
templ1_{0:31} ←b_{16:31} ×sf c_{0:15}
if (b_{16:31} = 0x8000) & (c_{0:15} = 0x8000) then
    templ1_{0:31} ←0x7FFF_FFFF //saturate
    movl ←1
else
    movl ←0
templ2_{0:31} ←b_{0:15} ×sf c_{16:31}
if (b_{0:15} = 0x8000) & (c_{16:31} = 0x8000) then
    templ2_{0:31} ←0x7FFF_FFFF //saturate
    movl ←1
endif
templ_{0:63} ←ROUND((EXTS_{64}(templ1_{0:31}) + EXTS_{64}(templ2_{0:31}) + EXTS_{64}(a_{32:63})),16)
ovl ←chk_ovf(temph_{30:32})
d_{32:63} ←SATURATE(ovl, templ_{30}, 0x8000_0000, 0x7FFF_0000, templ_{32:63})
// update accumulator
ACC_{0:63} ←d_{0:63}
// update SPEFSCR
SPEFSCR_{OVH} ←movh | ovh; SPEFSCR_{OV} ←movl | ovl
SPEFSCR_{SOVH} ←SPEFSCR_{SOVH} | movh | ovh; SPEFSCR_{SOV} ←SPEFSCR_{SOV} | movl | ovl
```

For the high word element in the destination, corresponding halfword pairs of signed fractional elements in the high halfwords of parameters **b** and **c** are multiplied producing a pair of 32-bit products. If both inputs of a multiply are -1.0, the result saturates to 0x7FFF_FFFF. The difference of this pair of intermediate 32-bit products is added to the high word of parameter **a** and rounded to 16 bits using the current fractional rounding mode in SPEFCSR, saturating if overflow or underflow occurs, and placed into the high word of parameter **d** and the accumulator. For the low word element in the destination, halfword pairs of signed fractional elements from the high halfwords of parameters **b** and **c** are multiplied after exchanging the high halfwords of parameter **b**, producing a pair of 32-bit products. If both inputs of a multiply are -1.0, the result saturates to 0x7FFF_FFFF. The sum of this pair of intermediate 32-bit products and the low word of parameter **a** is rounded to 16 bits using the current fractional rounding mode in SPEFCSR, saturating if overflow or underflow occurs, and placed into the low word of parameter **d** and the accumulator.

The overflow and summary overflow bits are recorded in the SPEFSCR based on an overflow or underflow from either the multiply or the accumulation.

Other registers altered: SPEFSCR, ACC

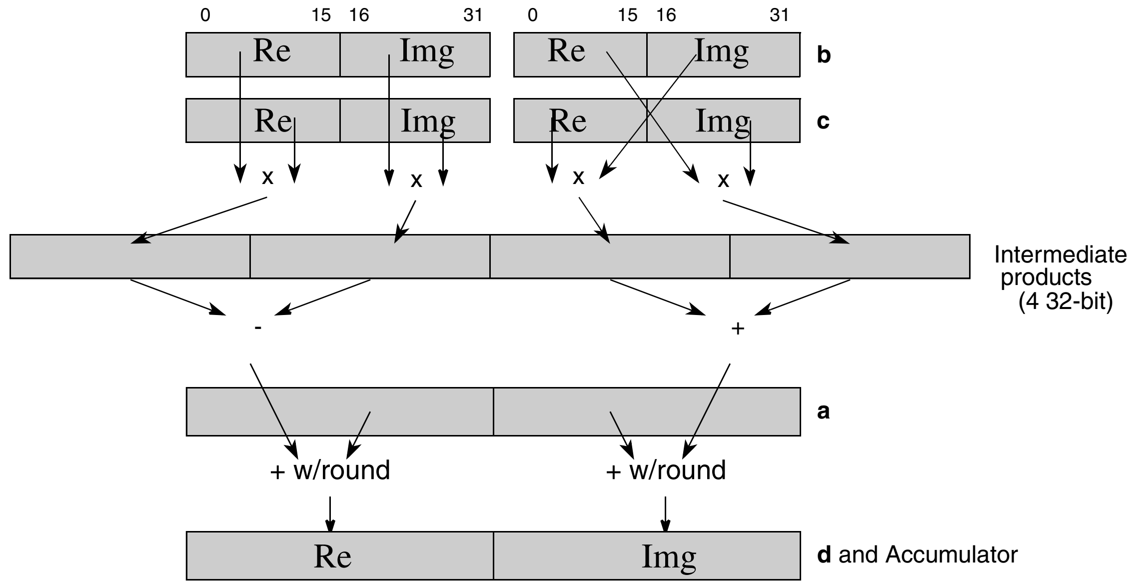


Figure 3-187. Vector Dot Product of High Halfwords, Complex, Signed, Saturate, Fractional, Round and Accumulate Words 3 op (__ev_dotphihcssfraaw3)

d	a	b	c	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**d ←a evdotphihcssfraaw3 d,b,c**

__ev_dotphihcssi[a] __ev_dotphihcssi[a]

Vector Dot Product of High Halfwords, Complex, Signed, Saturate, Integer (to Accumulator)

d = __ev_dotphihcssi (a,b) (A = 0)

d = __ev_dotphihcssia (a,b) (A = 1)

```
// high dot
temph1_0:31 ← a_0:15 ×si b_0:15; temph2_0:31 ← a_16:31 ×si b_16:31
temph_0:63 ← EXTS_64(temph1_0:31) - EXTS_64(temph2_0:31)
ovh ← temph_31 ⊕ temph_32
d_0:31 ← SATURATE(ovh, temph_31, 0x8000_0000, 0x7FFF_FFFF, temph_32:63)
//low
templ1_0:31 ← a_16:31 ×si b_0:15; templ2_0:31 ← a_0:15 ×si b_16:31
templ_0:63 ← EXTS_64(templ1_0:31) + EXTS_64(templ2_0:31)
ovl ← templ_31 ⊕ templ_32
d_32:63 ← SATURATE(ovl, templ_31, 0x8000_0000, 0x7FFF_FFFF, templ_32:63)
// update accumulator
if A = 1 then 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 the high word element in the destination, corresponding halfword pairs of signed integer elements in the high halfwords of parameters **a** and **b** are multiplied producing a pair of 32-bit products. The difference of this pair of intermediate 32-bit products is placed into the high word of parameter **d**, saturating if overflow or underflow occurs. For the low word element in the destination, halfword pairs of signed integer elements from the high halfwords of parameters **a** and **b** are multiplied after exchanging the high halfwords of parameter **a**, producing a pair of 32-bit products. The sum of this pair of intermediate 32-bit products is placed into the low word of parameter **d**, saturating if overflow or underflow occurs. If A = 1, the result in **d** is also placed into the accumulator.

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

Figure 3-188. Vector Dot Product of High Halfwords, Complex, Signed, Saturate, Integer (to Accumulator) (__ev_dotphihcssi[a])

A	d	a	b	Maps to
A = 0	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evdotphihcssi d,a,b**
A = 1	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evdotphihcssia d,a,b**

__ev_dotphihcssiaaw __ev_dotphihcssiaaw

Vector Dot Product of High Halfwords, Complex, Signed, Saturate, Integer and Accumulate into Words

d = __ev_dotphihcssiaaw (a,b)

```
// high dot
temph1_0:31 ← a_0:15 ×si b_0:15; temph2_0:31 ← a_16:31 ×si b_16:31
temph_0:63 ← EXTS_64(temph1_0:31) - EXTS_64(temph2_0:31) + EXTS_64(ACC_0:31)
ovh ← chk_ovf(temph_30:32)
d_0:31 ← SATURATE(ovh, temph_30, 0x8000_0000, 0x7FFF_FFFF, temph_32:63)

//low
templ1_0:31 ← a_16:31 ×si b_0:15; templ2_0:31 ← a_0:15 ×si b_16:31
templ_0:63 ← EXTS_64(templ1_0:31) + EXTS_64(templ2_0:31) + EXTS_64(ACC_32:63)
ovl ← chk_ovf(templ_30:32)
d_32:63 ← SATURATE(ovl, templ_30, 0x8000_0000, 0x7FFF_FFFF, templ_32:63)
// 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 the high word element in the destination, corresponding halfword pairs of signed fractional elements in the high halfwords of parameters **a** and **b** are multiplied producing a pair of 32-bit products. The difference of this pair of intermediate 32-bit products is added to the high word of the accumulator, and the result is placed into the high word of parameter **d** and the accumulator, saturating if overflow or underflow occurs. For the low word element in the destination, halfword pairs of signed fractional elements from the high halfwords of parameters **a** and **b** are multiplied after exchanging the high halfwords of parameter **a**, producing a pair of 32-bit products. The sum of this pair of intermediate 32-bit products and the low word of the accumulator is placed into the low word of parameter **d** and the accumulator, saturating if overflow or underflow occurs.

Other registers altered: SPEFSCR, ACC

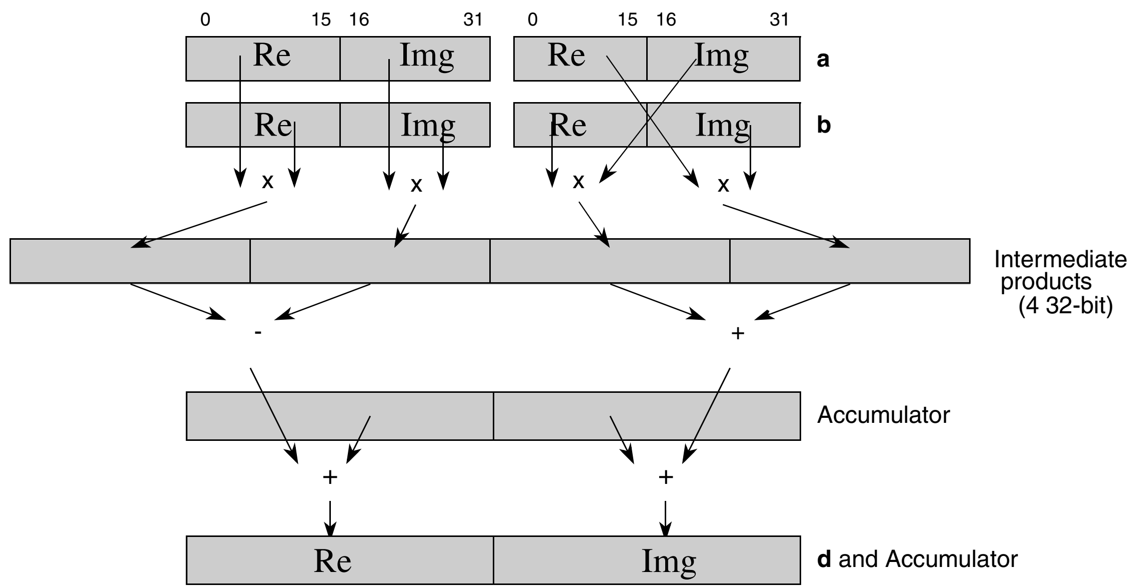


Figure 3-189. Vector Dot Product of Half Words, Subtract, Signed, Saturate, Integer and Accumulate Words (__ev_dotphihcssiaaw)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evdotphihcssiaaw d,a,b**

__ev_dotphihcssiaaw3 __ev_dotphihcssiaaw3

Vector Dot Product of High Halfwords, Complex, Signed, Saturate, Integer and Accumulate into Words, 3 operand

d = __ev_dotphihcssiaaw3 (a,b,c)

```
// high dot
temph1_{0:31} ← b_{0:15} ×_{si} c_{0:15}; temph2_{0:31} ← b_{16:31} ×_{si} c_{16:31}
temph_{0:63} ← EXTS_{64}(temph1_{0:31}) - EXTS_{64}(temph2_{0:31}) + EXTS_{64}(a_{0:31})
ovh ← chk_ovf(temph_{30:32})
d_{0:31} ← SATURATE(ovh, temph_{30}, 0x8000_0000, 0x7FFF_FFFF, temph_{32:63})

//low
templ1_{0:31} ← b_{16:31} ×_{si} c_{0:15}; templ2_{0:31} ← b_{0:15} ×_{si} c_{16:31}
templ_{0:63} ← EXTS_{64}(templ1_{0:31}) + EXTS_{64}(templ2_{0:31}) + EXTS_{64}(a_{32:63})
ovl ← chk_ovf(templ_{30:32})
d_{32:63} ← SATURATE(ovl, templ_{30}, 0x8000_0000, 0x7FFF_FFFF, templ_{32:63})
// 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 the high word element in the destination, corresponding halfword pairs of signed fractional elements in the high halfwords of parameters **b** and **c** are multiplied producing a pair of 32-bit products. The difference of this pair of intermediate 32-bit products is added to the high word of parameter **a**, and the result is placed into the high word of parameter **d** and the accumulator, saturating if overflow or underflow occurs. For the low word element in the destination, halfword pairs of signed fractional elements from the high halfwords of parameters **b** and **c** are multiplied after exchanging the high halfwords of parameter **b**, producing a pair of 32-bit products. The sum of this pair of intermediate 32-bit products and the low word of parameter **a** is placed into the low word of parameter **d** and the accumulator, saturating if overflow or underflow occurs.

Other registers altered: SPEFSCR, ACC

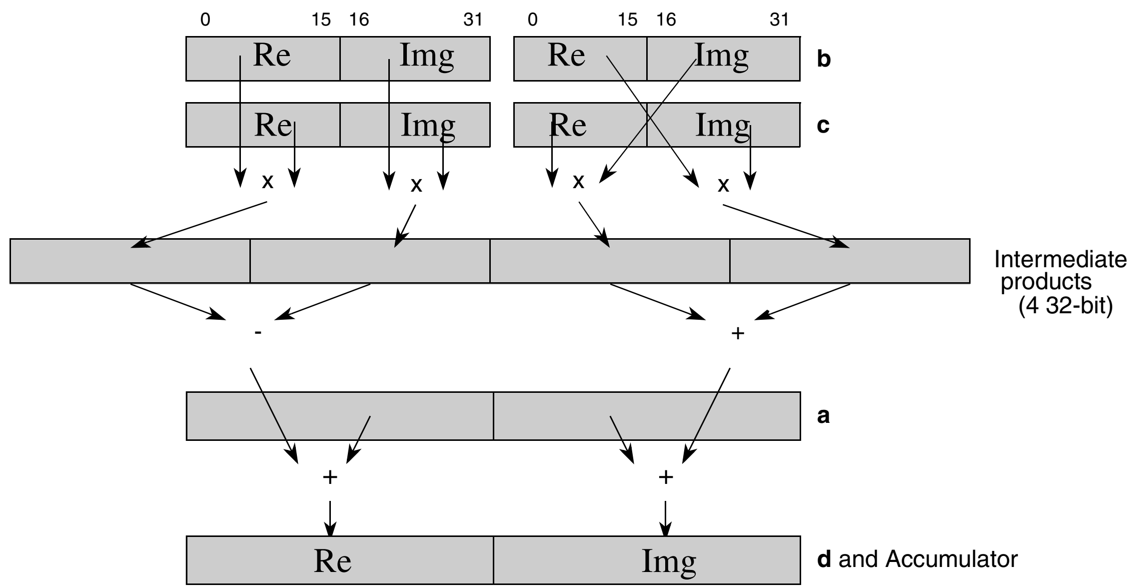


Figure 3-190. Vector Dot Product of High Halfwords, Complex, Signed, Saturate, Integer and Accumulate Words (__ev_dotphihcssiaaw3)

d	a	b	c	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**d ←a evdotphihcssiaaw3 d,b,c**

__ev_dotphssmi[a] __ev_dotphssmi[a]

Vector Dot Product of Halfwords, Subtract, Signed, Modulo, Integer (to Accumulator)

d = __ev_dotphssmi (a,b) (A = 0)

d = __ev_dotphssmia (a,b) (A = 1)

```
// high dot
temph1_0:31 ← a_0:15 ×si b_0:15
temph2_0:31 ← a_16:31 ×si b_16:31
temph_0:31 ← temph1_0:31 - temph2_0:31 // modulo difference
d_0:31 ← temph_0:31

//low
templ1_0:31 ← a_32:47 ×si b_32:47
templ2_0:31 ← a_48:63 ×si b_48:63
templ_0:31 ← templ1_0:31 - templ2_0:31 // modulo difference
d_32:63 ← templ_0:31

// update accumulator
if A = 1 then ACC_0:63 ←d_0:63
```

For each word element in the destination, corresponding halfword pairs of signed integer elements in parameters **a** and **b** are multiplied producing a pair of 32-bit products. The intermediate 32-bit product of the odd halfword is subtracted from the intermediate product of the even halfword with a modulo difference, and the result 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 3-191. Vector Dot Product of Halfwords, Subtract, Signed, Modulo, Integer (to Accumulator) (__ev_dotphssmi[a])

A	d	a	b	Maps to
A = 0	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evdotphssmi d,a,b**
A = 1	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evdotphssmia d,a,b**

__ev_dotphssmiaaw __ev_dotphssmiaaw

Vector Dot Product of Halfwords, Subtract, Signed, Modulo, Integer and Accumulate into Words

d = __ev_dotphssmiaaw (a,b)

```
// high dot
temph1_{0:31} ← a_{0:15} ×_{si} b_{0:15}
temph2_{0:31} ← a_{16:31} ×_{si} b_{16:31}
temph_{0:31} ← temph1_{0:31} - temph2_{0:31} + ACC_{0:31} // modulo difference
d_{0:31} ← temph_{0:31}

//low
templ1_{0:31} ← a_{32:47} ×_{si} b_{32:47}
templ2_{0:31} ← a_{48:63} ×_{si} b_{48:63}
templ_{0:31} ← templ1_{0:31} - templ2_{0:31} + ACC_{32:63} // modulo difference
d_{32:63} ← templ_{0:31}

// update accumulator
ACC_{0:63} ←d_{0:63}
```

For each word element in the destination, corresponding halfword pairs of signed integer elements in parameters **a** and **b** are multiplied producing a pair of 32-bit products. The intermediate 32-bit product of the odd halfword is subtracted from the intermediate product of the even halfword, and the difference is added together with the contents of the corresponding accumulator word and the sum is placed into the corresponding parameter **d** and accumulator word.

Other registers altered: ACC

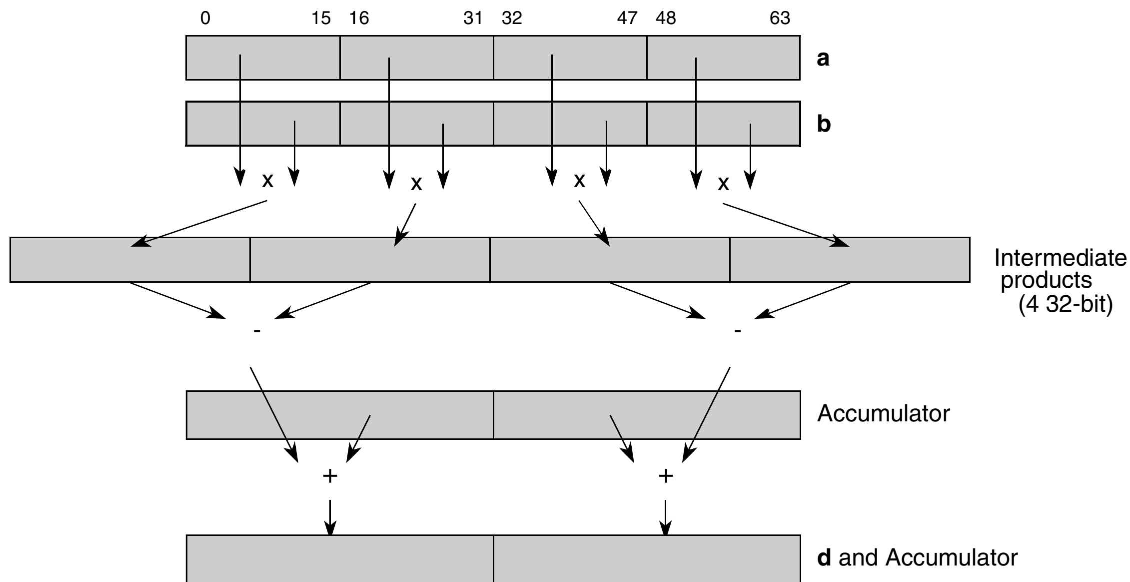


Figure 3-192. Vector Dot Product of Halfwords, Subtract, Signed, Modulo, Integer and Accumulate Words (__ev_dotphssmiaaw)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evdotphssmiaaw d,a,b**

__ev_dotphssmiaaw3 __ev_dotphssmiaaw3

Vector Dot Product of Halfwords, Subtract, Signed, Modulo, Integer and Accumulate into Words, 3 operand

d = __ev_dotphssmiaaw3 (a,b,c)

```
// high dot
temph1[0:31] ← b[0:15] ×si c[0:15]
temph2[0:31] ← b[16:31] ×si c[16:31]
temph[0:31] ← temph1[0:31] - temph2[0:31] + a[0:31] // modulo difference
d[0:31] ← temph[0:31]

//low
templ1[0:31] ← b[32:47] ×si c[32:47]
templ2[0:31] ← b[48:63] ×si c[48:63]
templ[0:31] ← templ1[0:31] - templ2[0:31] + a[32:63] // modulo difference
d[32:63] ← templ[0:31]

// update accumulator
ACC[0:63] ←d[0:63]
```

For each word element in the destination, corresponding halfword pairs of signed integer elements in parameters **b** and **c** are multiplied producing a pair of 32-bit products. The intermediate 32-bit product of the odd halfword is subtracted from the intermediate product of the even halfword, and the difference is added together with the contents of the corresponding parameter **a** word and the sum is placed into the corresponding parameter **d** and accumulator word.

Other registers altered: ACC

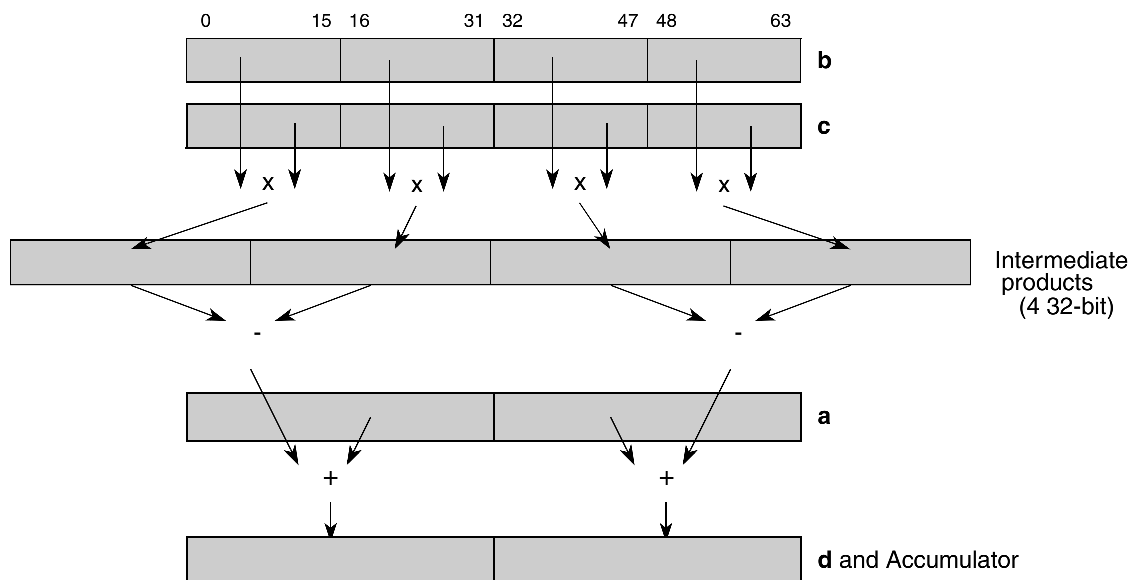


Figure 3-193. Vector Dot Product of Halfwords, Subtract, Signed, Modulo, Integer and Accumulate Words 3 op (__ev_dotphssmiaaw3)

d	a	b	c	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**d ←a** **evdotphssmiaaw3 d,b,c**

__ev_dotphsssf[a] __ev_dotphsssf[a]

Vector Dot Product of Halfwords, Subtract, Signed, Saturate, Fractional (to Accumulator)

d = __ev_dotphsssf (a,b) (A = 0)

d = __ev_dotphsssfa (a,b) (A = 1)

```
// high dot
temph1_{0:31} ← a_{0:15} ×sf b_{0:15}
if (a_{0:15} = 0x8000) & (b_{0:15} = 0x8000) then
    temph1_{0:31} ←0x7FFF_FFFF //saturate
    movh ←1
else
    movh ←0
temph2_{0:31} ← a_{16:31} ×sf b_{16:31}
if (a_{16:31} = 0x8000) & (b_{16:31} = 0x8000) then
    temph2_{0:31} ←0x7FFF_FFFF //saturate
    movh ←1
endif
temph_{0:63} ←EXTS_{64}(temph1_{0:31}) - EXTS_{64}(temph2_{0:31})
ovh ←temph_{31} ⊕ temph_{32}
d_{0:31} ←SATURATE(ovh, temph_{31}, 0x8000_0000, 0x7FFF_FFFF, temph_{32:63})

//low
templ1_{0:31} ← a_{32:47} ×sf b_{32:47}
if (a_{32:47} = 0x8000) & (b_{32:47} = 0x8000) then
    templ1_{0:31} ←0x7FFF_FFFF //saturate
    movl ←1
else
    movl ←0
templ2_{0:31} ← a_{48:63} ×sf b_{48:63}
if (a_{48:63} = 0x8000) & (b_{48:63} = 0x8000) then
    templ2_{0:31} ←0x7FFF_FFFF //saturate
    movl ←1
endif
templ_{0:63} ←EXTS_{64}(templ1_{0:31}) - EXTS_{64}(templ2_{0:31})
ovl ←templ_{31} ⊕ templ_{32}
d_{32:63} ←SATURATE(ovl, templ_{31}, 0x8000_0000, 0x7FFF_FFFF, templ_{32:63})
// update accumulator
if A = 1 then ACC_{0:63} ←d_{0:63}
// update SPEFSCR
SPEFSCR_{OVH} ←movh | ovh; SPEFSCR_{OV} ←movl | ovl
SPEFSCR_{SOVH} ←SPEFSCR_{SOVH} | movh | ovh; SPEFSCR_{SOV} ←SPEFSCR_{SOV} | movl | ovl
```

For each word element in the destination, corresponding halfword pairs of signed fractional elements in parameters **a** and **b** are multiplied producing a pair of 32-bit products. If both inputs of a multiply are -1.0, the result saturates to 0x7FFF_FFFF. The intermediate 32-bit product of the odd halfword is subtracted from the intermediate product of the even halfword, saturating if overflow or underflow occurs, and the difference is placed into the corresponding parameter **d** word. If A = 1, the result in parameter **d** is also placed into the accumulator

The overflow and summary overflow bits are recorded in the SPEFSCR based on an overflow or underflow from either the multiply or the accumulation.

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

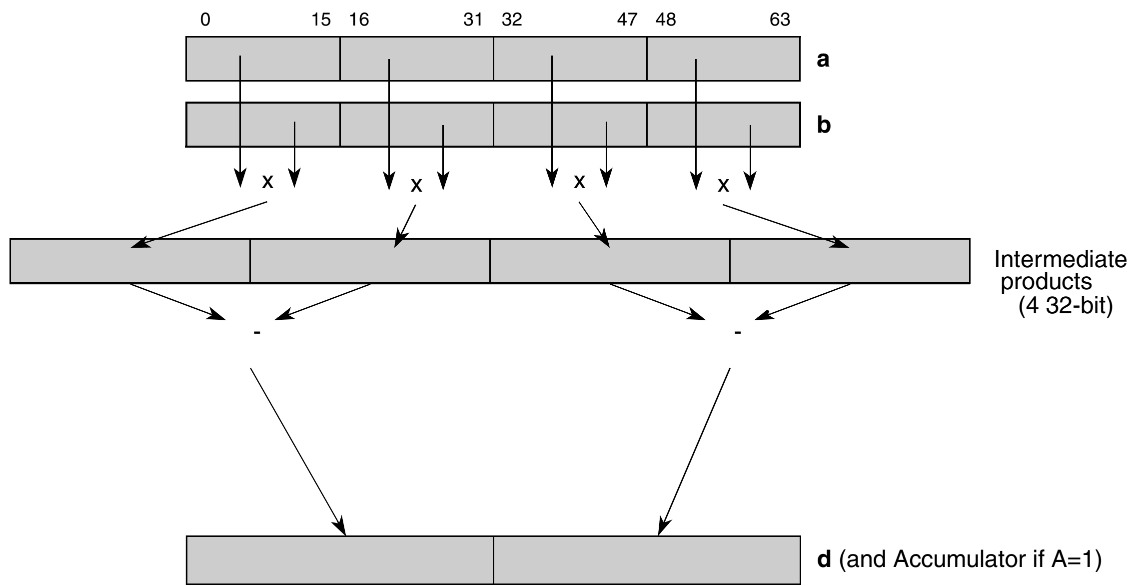


Figure 3-194. Vector Dot Product of Halfwords, Subtract, Signed, Saturate, Fractional (to Accumulator) (__ev_dotphsssf[a])

A	d	a	b	Maps to
A = 0	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evdotphsssf d,a,b**
A = 1	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evdotphsssfa d,a,b**

__ev_dotphsssfaaw __ev_dotphsssfaaw

Vector Dot Product of Halfwords, Subtract, Signed, Saturate, Fractional and Accumulate into Words

d = __ev_dotphsssfaaw (a,b)

```
// high dot
temph1_{0:31} ← a_{0:15} ×_{sf} b_{0:15}
if (a_{0:15} = 0x8000) & (b_{0:15} = 0x8000) then
    temph1_{0:31} ←0x7FFF_FFFF //saturate
    movh ←1
else
    movh ←0
temph2_{0:31} ← a_{16:31} ×_{sf} b_{16:31}
if (a_{16:31} = 0x8000) & (b_{16:31} = 0x8000) then
    temph2_{0:31} ←0x7FFF_FFFF //saturate
    movh ←1
endif
temph_{0:63} ← EXTS_{64}(temph1_{0:31}) - EXTS_{64}(temph2_{0:31}) + EXTS_{64}(ACC_{0:31})
ovh ← chk_ovf(temph_{30:32})
d_{0:31} ← SATURATE(ovh, temph_{30}, 0x8000_0000, 0x7FFF_FFFF, temph_{32:63})

//low
templ1_{0:31} ← a_{32:47} ×_{sf} b_{32:47}
if (a_{32:47} = 0x8000) & (b_{32:47} = 0x8000) then
    templ1_{0:31} ←0x7FFF_FFFF //saturate
    movl ←1
else
    movl ←0
templ2_{0:31} ← a_{48:63} ×_{sf} b_{48:63}
if (a_{48:63} = 0x8000) & (b_{48:63} = 0x8000) then
    templ2_{0:31} ←0x7FFF_FFFF //saturate
    movl ←1
endif
templ_{0:63} ← EXTS_{64}(templ1_{0:31}) - EXTS_{64}(templ2_{0:31}) + EXTS_{64}(ACC_{32:63})
ovl ← chk_ovf(templ_{30:32})
d_{32:63} ← SATURATE(ovl, templ_{30}, 0x8000_0000, 0x7FFF_FFFF, templ_{32:63})
// update accumulator
ACC_{0:63} ← d_{0:63}
// update SPEFSCR
SPEFSCR_{OVH} ← movh | ovh; SPEFSCR_{OV} ← movl | ovl
SPEFSCR_{SOVH} ← SPEFSCR_{SOVH} | movh | ovh; SPEFSCR_{SOV} ← SPEFSCR_{SOV} | movl | ovl
```

For each word element in the destination, corresponding halfword pairs of signed fractional elements in parameters **a** and **b** are multiplied producing a pair of 32-bit products. If both inputs of a multiply are -1.0, the result saturates to 0x7FFF_FFFF. The intermediate 32-bit product of the odd halfword is subtracted from the intermediate product of the even halfword, and the difference is added together with the contents of the corresponding accumulator word, saturating if overflow or underflow occurs, and the sum is placed into the corresponding parameter **d** and accumulator word.

The overflow and summary overflow bits are recorded in the SPEFSCR based on an overflow or underflow from either the multiply or the accumulation.

Other registers altered: SPEFSCR, ACC

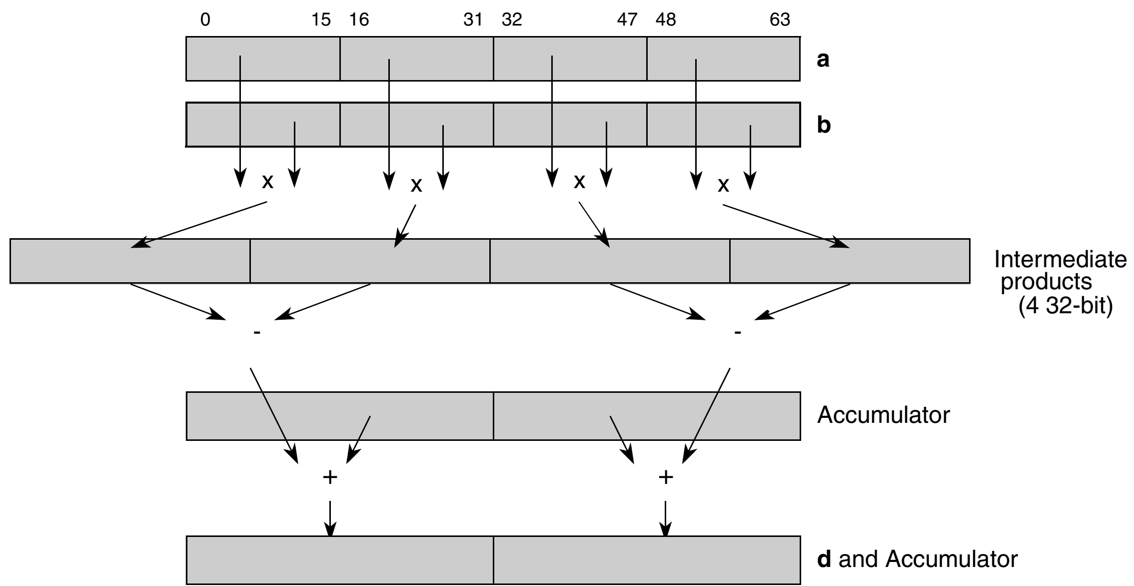


Figure 3-195. Vector Dot Product of Halfwords, Subtract, Signed, Saturate, Fractional and Accumulate Words (__ev_dotphsssfaaw)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evdotphsssfaaw d,a,b**

__ev_dotphsssfaaw3 __ev_dotphsssfaaw3

Vector Dot Product of Halfwords, Subtract, Signed, Saturate, Fractional and Accumulate into Words, 3 operand

d = __ev_dotphsssfaaw3 (a,b,c)

```
// high dot
temph1_{0:31} ←b_{0:15} ×_{sf} c_{0:15}
if (b_{0:15} = 0x8000) & (c_{0:15} = 0x8000) then
    temph1_{0:31} ←0x7FFF_FFFF //saturate
    movh ←1
else
    movh ←0
temph2_{0:31} ←b_{16:31} ×_{sf} c_{16:31}
if (b_{16:31} = 0x8000) & (c_{16:31} = 0x8000) then
    temph2_{0:31} ←0x7FFF_FFFF //saturate
    movh ←1
endif
temph_{0:63} ←EXTS_{64}(temph1_{0:31}) - EXTS_{64}(temph2_{0:31}) + EXTS_{64}(a_{0:31})
ovh ←chk_ovf(temph_{30:32})
d_{0:31} ←SATURATE(ovh, temph_{30}, 0x8000_0000, 0x7FFF_FFFF, temph_{32:63})

//low
templ1_{0:31} ←b_{32:47} ×_{sf} c_{32:47}
if (b_{32:47} = 0x8000) & (c_{32:47} = 0x8000) then
    templ1_{0:31} ←0x7FFF_FFFF //saturate
    movl ←1
else
    movl ←0
templ2_{0:31} ←b_{48:63} ×_{sf} c_{48:63}
if (b_{48:63} = 0x8000) & (c_{48:63} = 0x8000) then
    templ2_{0:31} ←0x7FFF_FFFF //saturate
    movl ←1
endif
templ_{0:63} ←EXTS_{64}(templ1_{0:31}) - EXTS_{64}(templ2_{0:31}) + EXTS_{64}(a_{32:63})
ovl ←chk_ovf(templ_{30:32})
d_{32:63} ←SATURATE(ovl, templ_{30}, 0x8000_0000, 0x7FFF_FFFF, templ_{32:63})
// update accumulator
ACC_{0:63} ←d_{0:63}
// update SPEFSCR
SPEFSCR_{OVH} ←movh | ovh; SPEFSCR_{OV} ←movl | ovl
SPEFSCR_{SOVH} ←SPEFSCR_{SOVH} | movh | ovh; SPEFSCR_{SOV} ←SPEFSCR_{SOV} | movl | ovl
```

For each word element in the destination, corresponding halfword pairs of signed fractional elements in parameters **b** and **c** are multiplied producing a pair of 32-bit products. If both inputs of a multiply are -1.0, the result saturates to 0x7FFF_FFFF. The intermediate 32-bit product of the odd halfword is subtracted from the intermediate product of the even halfword, and the difference is added together with the contents of the corresponding parameter **a** word, saturating if overflow or underflow occurs, and the sum is placed into the corresponding parameter **d** and accumulator word.

The overflow and summary overflow bits are recorded in the SPEFSCR based on an overflow or underflow from either the multiply or the accumulation.

Other registers altered: SPEFSCR, ACC

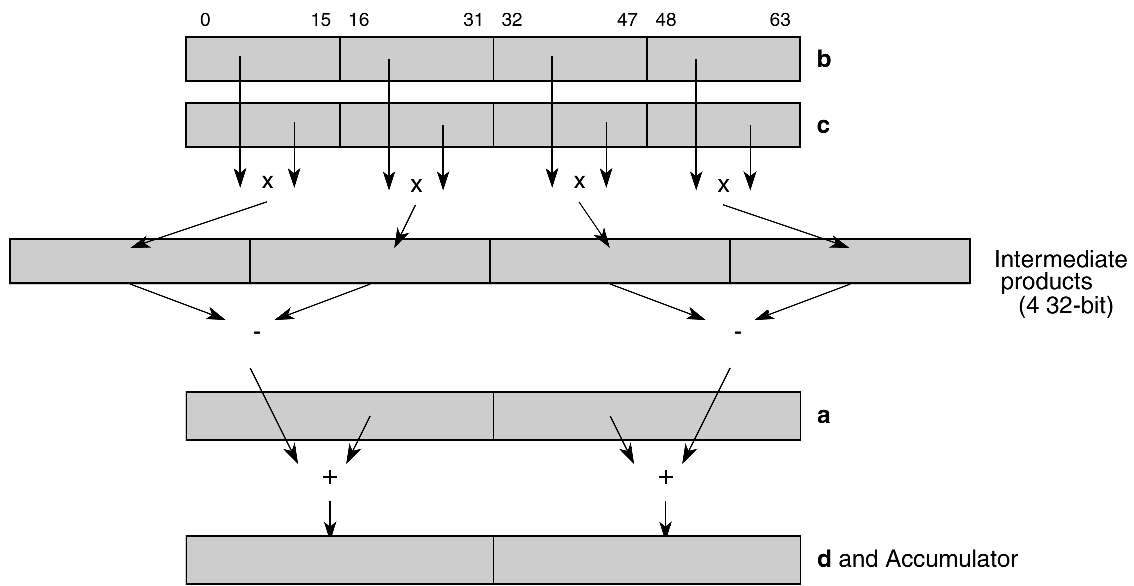


Figure 3-196. Vector Dot Product of Halfwords, Subtract, Signed, Saturate, Fractional and Accumulate Words 3 op (__ev_dotphsssfaaw3)

d	a	b	c	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	d ←a evdotphsssfaaw3 d,b,c

__ev_dotphsssfr[a] __ev_dotphsssfr[a]

Vector Dot Product of Halfwords, Subtract, Signed, Saturate, Fractional, Round (to Accumulator)

d = __ev_dotphsssfr (a,b) (A = 0)

d = __ev_dotphsssfra (a,b) (A = 1)

```
// high dot
temph1_{0:31} ← a_{0:15} ×sf b_{0:15}
if (a_{0:15} = 0x8000) & (b_{0:15} = 0x8000) then
    temph1_{0:31} ←0x7FFF_FFFF //saturate
    movh ←1
else
    movh ←0
temph2_{0:31} ← a_{16:31} ×sf b_{16:31}
if (a_{16:31} = 0x8000) & (b_{16:31} = 0x8000) then
    temph2_{0:31} ←0x7FFF_FFFF //saturate
    movh ←1
endif
temph_{0:63} ←ROUND((EXTS_{64}(temph1_{0:31}) - EXTS_{64}(temph2_{0:31})),16)
ovh ←chk_ovf(temph_{30:32})
d_{0:31} ←SATURATE(ovh, temph_{30}, 0x8000_0000, 0x7FFF_0000, temph_{32:63})

//low
templ1_{0:31} ← a_{32:47} ×sf b_{32:47}
if (a_{32:47} = 0x8000) & (b_{32:47} = 0x8000) then
    templ1_{0:31} ←0x7FFF_FFFF //saturate
    movl ←1
else
    movl ←0
templ2_{0:31} ← a_{48:63} ×sf b_{48:63}
if (a_{48:63} = 0x8000) & (b_{48:63} = 0x8000) then
    templ2_{0:31} ←0x7FFF_FFFF //saturate
    movl ←1
endif
templ_{0:63} ←ROUND((EXTS_{64}(templ1_{0:31}) - EXTS_{64}(templ2_{0:31})),16)
ovl ←chk_ovf(templ_{30:32})
d_{32:63} ←SATURATE(ovl, templ_{30}, 0x8000_0000, 0x7FFF_0000, templ_{32:63})

// update accumulator
if A = 1 then ACC_{0:63} ←d_{0:63}
// update SPEFSCR
SPEFSCR_{OVH} ←movh | ovh; SPEFSCR_{OV} ←movl | ovl
SPEFSCR_{SOVH} ←SPEFSCR_{SOVH} | movh | ovh; SPEFSCR_{SOV} ←SPEFSCR_{SOV} | movl | ovl
```

For each word element in the destination, corresponding halfword pairs of signed fractional elements in parameters **a** and **b** are multiplied producing a pair of 32-bit products. If both inputs of a multiply are -1.0, the result saturates to 0x7FFF_FFFF. The intermediate 32-bit product of the odd halfword is subtracted from the intermediate product of the even halfword, and the difference is and rounded to 16 bits using the current fractional rounding mode in SPEFCSR, saturating if overflow or underflow occurs, and placed into the corresponding parameter **d** word. If A = 1, the result in parameter **d** is also placed into the accumulator

The overflow and summary overflow bits are recorded in the SPEFSCR based on an overflow or underflow from either the multiply or the accumulation.

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

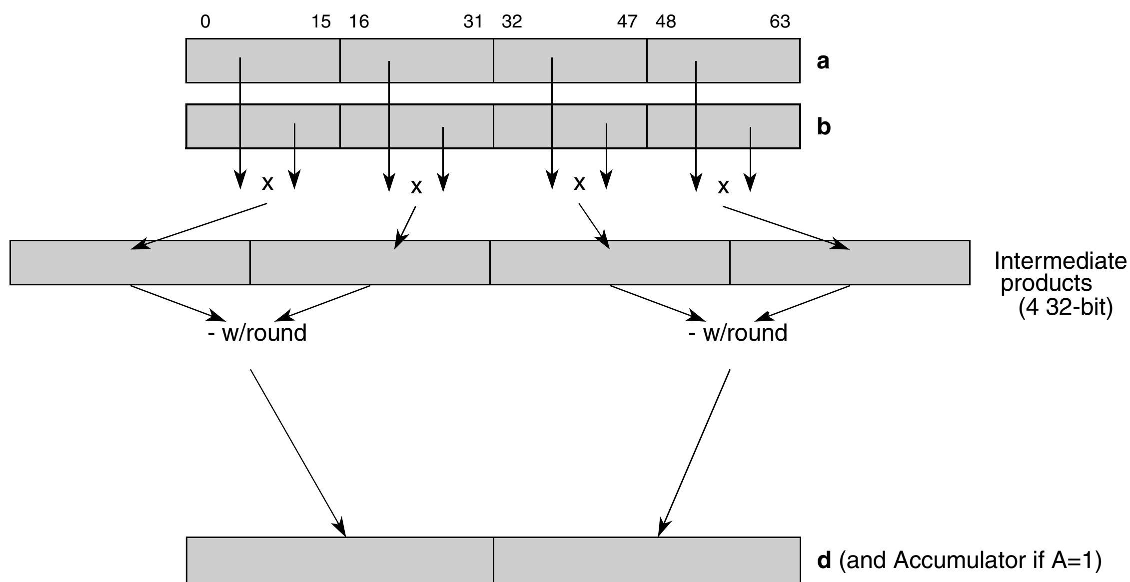


Figure 3-197. Vector Dot Product of Halfwords, Subtract, Signed, Saturate, Fractional, Round (to Accumulator) (__ev_dotphsssfr[a])

A	d	a	b	Maps to
A = 0	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evdotphsssfr d,a,b**
A = 1	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evdotphsssfra d,a,b**

__ev_dotphsssfraaw __ev_dotphsssfraaw

Vector Dot Product of Halfwords, Subtract, Signed, Saturate, Fractional, Round and Accumulate into Words

d = __ev_dotphsssfraaw (a,b)

```
// high dot
temph1[0:31] ← a[0:15] ×sf b[0:15]
if (a[0:15] = 0x8000) & (b[0:15] = 0x8000) then
    temph1[0:31] ←0x7FFF_FFFF //saturate
    movh ←1
else
    movh ←0
temph2[0:31] ← a[16:31] ×sf b[16:31]
if (a[16:31] = 0x8000) & (b[16:31] = 0x8000) then
    temph2[0:31] ←0x7FFF_FFFF //saturate
    movh ←1
endif
temph[0:63] ← ROUND((EXTS64(temph1[0:31]) - EXTS64(temph2[0:31]) + EXTS64(ACC[0:31])),16)
ovh ←chk_ovf(temph[30:32])
d[0:31] ← SATURATE(ovh, temph[30], 0x8000_0000, 0x7FFF_0000, temph[32:63])

//low
templ1[0:31] ← a[32:47] ×sf b[32:47]
if (a[32:47] = 0x8000) & (b[32:47] = 0x8000) then
    templ1[0:31] ←0x7FFF_FFFF //saturate
    movl ←1
else
    movl ←0
templ2[0:31] ← a[48:63] ×sf b[48:63]
if (a[48:63] = 0x8000) & (b[48:63] = 0x8000) then
    templ2[0:31] ←0x7FFF_FFFF //saturate
    movl ←1
endif
templ[0:63] ← ROUND((EXTS64(templ1[0:31]) - EXTS64(templ2[0:31]) + EXTS64(ACC[32:63])),16)
ovl ←chk_ovf(templ[30:32])
d[32:63] ←SATURATE(ovl, templ[30], 0x8000_0000, 0x7FFF_0000, templ[32:63])
// update accumulator
ACC[0:63] ← d[0:63]
// update SPEFSCR
SPEFSCR[OVH] ←movh | ovh; SPEFSCR[OV] ←movl | ovl
SPEFSCR[SOVH] ←SPEFSCR[SOVH] | movh | ovh; SPEFSCR[SOV] ←SPEFSCR[SOV] | movl | ovl
```

For each word element in the destination, corresponding halfword pairs of signed fractional elements in parameters **a** and **b** are multiplied producing a pair of 32-bit products. If both inputs of a multiply are -1.0, the result saturates to 0x7FFF_FFFF. The intermediate 32-bit product of the odd halfword is subtracted from the intermediate product of the even halfword, and the difference is added together with the contents of the corresponding accumulator word and rounded to 16 bits using the current fractional rounding mode in SPEFCSR, saturating if overflow or underflow occurs, and placed into the corresponding parameter **d** and accumulator word.

The overflow and summary overflow bits are recorded in the SPEFSCR based on an overflow or underflow from either the multiply or the accumulation.

Other registers altered: SPEFSCR, ACC

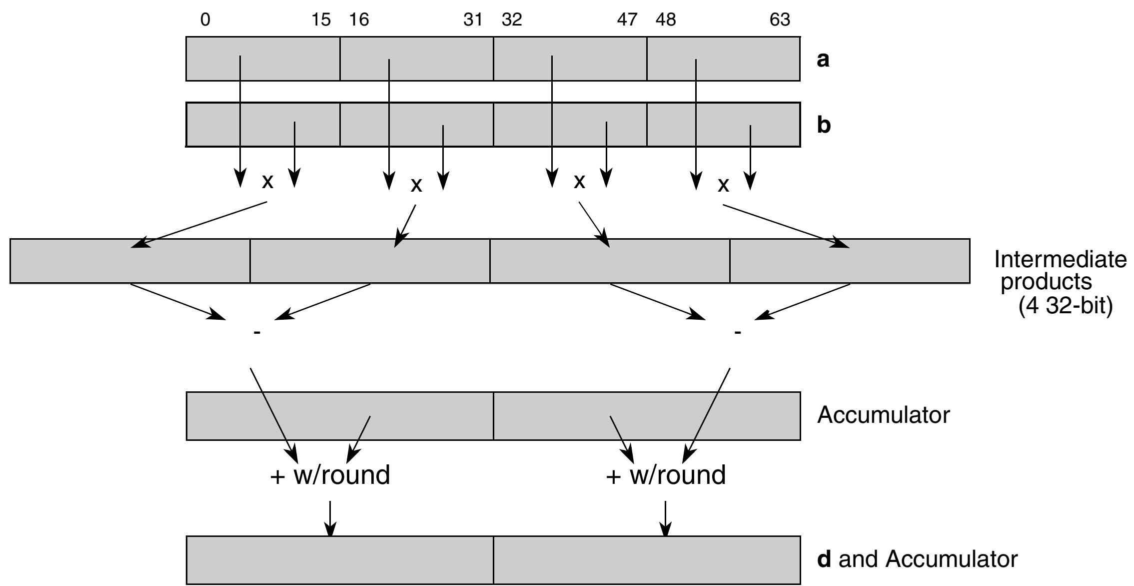


Figure 3-198. Vector Dot Product of Halfwords, Subtract, Signed, Saturate, Fractional, Round and Accumulate Words (__ev_dotphsssfraaw)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evdotphsssfraaw d,a,b**

__ev_dotphsssfraaw3 __ev_dotphsssfraaw3

Vector Dot Product of Halfwords, Subtract, Signed, Saturate, Fractional, Round and Accumulate into Words, 3 operand

d = __ev_dotphsssfraaw3 (a,b,c)

```
// high dot
temph1_0:31 ←b_0:15 ×sf c_0:15
if (b_0:15 = 0x8000) & (c_0:15 = 0x8000) then
    temph1_0:31 ←0x7FFF_FFFF //saturate
    movh ←1
else
    movh ←0
temph2_0:31 ←b_16:31 ×sf c_16:31
if (b_16:31 = 0x8000) & (c_16:31 = 0x8000) then
    temph2_0:31 ←0x7FFF_FFFF //saturate
    movh ←1
endif
temph_0:63 ←ROUND((EXTS_64(temph1_0:31) - EXTS_64(temph2_0:31) + EXTS_64(a_0:31)),16)
ovh ←chk_ovf(temph_30:32)
d_0:31 ←SATURATE(ovh, temph_30, 0x8000_0000, 0x7FFF_0000, temph_32:63)

//low
templ1_0:31 ←b_32:47 ×sf c_32:47
if (b_32:47 = 0x8000) & (c_32:47 = 0x8000) then
    templ1_0:31 ←0x7FFF_FFFF //saturate
    movl ←1
else
    movl ←0
templ2_0:31 ←b_48:63 ×sf c_48:63
if (b_48:63 = 0x8000) & (c_48:63 = 0x8000) then
    templ2_0:31 ←0x7FFF_FFFF //saturate
    movl ←1
endif
templ_0:63 ←ROUND((EXTS_64(templ1_0:31) - EXTS_64(templ2_0:31) + EXTS_64(a_32:63)),16)
ovl ←chk_ovf(templ_30:32)
d_32:63 ←SATURATE(ovl, templ_30, 0x8000_0000, 0x7FFF_0000, templ_32:63)
// update accumulator
ACC_0:63 ←d_0:63
// update SPEFSCR
SPEFSCR_OVH ←movh | ovh; SPEFSCR_OV ←movl | ovl
SPEFSCR_SOVH ←SPEFSCR_SOVH | movh | ovh; SPEFSCR_SOV ←SPEFSCR_SOV | movl | ovl
```

For each word element in the destination, corresponding halfword pairs of signed fractional elements in parameters **b** and **c** are multiplied producing a pair of 32-bit products. If both inputs of a multiply are -1.0, the result saturates to 0x7FFF_FFFF. The intermediate 32-bit product of the odd halfword is subtracted from the intermediate product of the even halfword, and the difference is added together with the contents of the corresponding parameter **a** word and rounded to 16 bits using the current fractional rounding mode in SPEFCSR, saturating if overflow or underflow occurs, and placed into the corresponding parameter **d** and accumulator word.

The overflow and summary overflow bits are recorded in the SPEFSCR based on an overflow or underflow from either the multiply or the accumulation.

Other registers altered: SPEFSCR, ACC

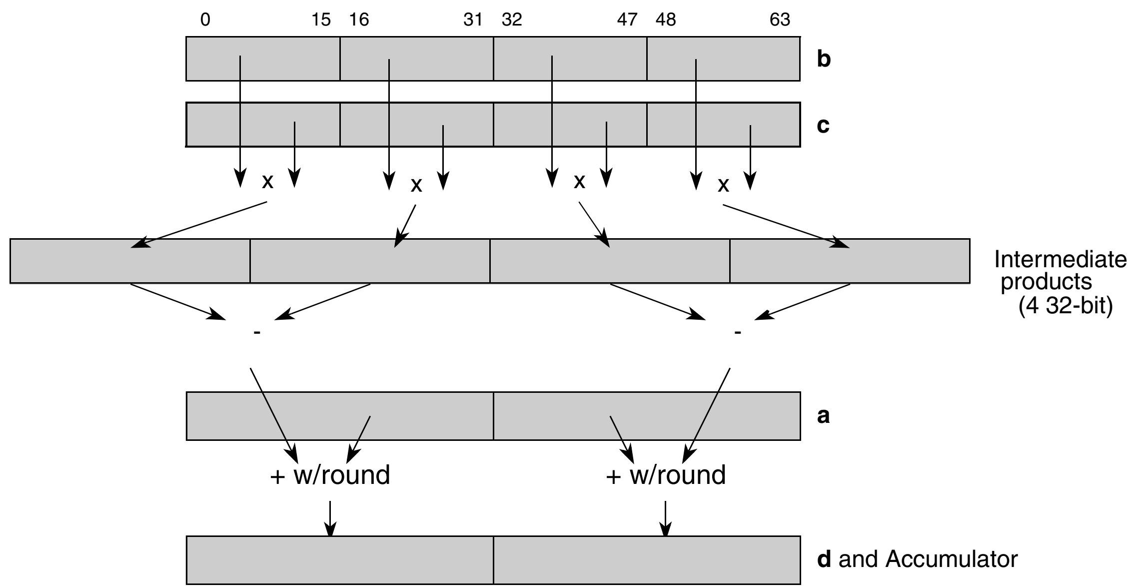


Figure 3-199. Vector Dot Product of Halfwords, Subtract, Signed, Saturate, Fractional, Round and Accumulate Words 3 op (__ev_dotphsssfraaw3)

d	a	b	c	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**d ← a** **evdotphsssfraaw3 d,b,c**

__ev_dotphsssi[a] __ev_dotphsssi[a]

Vector Dot Product of Half Words, Subtract, Signed, Saturate, Integer (to Accumulator)

Maps to **evotphssmi[a]** since no overflow can occur

A	d	a	b	Maps to
A = 0	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evdotphssmi d,a,b**
A = 1	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evdotphssmia d,a,b**

__ev_dotphsssiaaw __ev_dotphsssiaaw

Vector Dot Product of Half Words, Subtract, Signed, Saturate, Integer and Accumulate into Words

d = __ev_dotphsssiaaw (a,b)

```
// high dot
temph1_{0:31} ←a_{0:15} ×_{si} b_{0:15}; temph2_{0:31} ←a_{16:31} ×_{si} b_{16:31}
temph_{0:63} ←EXTS(temph1_{0:31}) - EXTS(temph2_{0:31}) + EXTS(ACC_{0:31})
ovh ←temph_{31} ⊕ temph_{32}
d_{0:31} ←SATURATE(ovh, temph_{31}, 0x8000_0000, 0xFFFF_FFFF, temph_{32:63})
//low
templ1_{0:31} ←a_{32:47} ×_{si} b_{32:47}; templ2_{0:31} ←a_{48:63} ×_{si} b_{48:63}
templ_{0:63} ←EXTS(templ1_{0:31}) - EXTS(templ2_{0:31}) + EXTS(ACC_{32:63})
ovl ←templ_{31} ⊕ templ_{32}
d_{32:63} ←SATURATE(ovl, templ_{31}, 0x8000_0000, 0xFFFF_FFFF, templ_{32:63})
// 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 destination, corresponding half word pairs of signed integer elements in parameters **a** and **b** are multiplied producing a pair of 32-bit products. The intermediate 32-bit product of the odd half word is subtracted from the intermediate product of the even half word, and the difference is added together with the contents of the corresponding accumulator word, saturating if overflow or underflow occurs, and the sum is placed into the corresponding parameter **d** and accumulator word.

Other registers altered: SPEFSCR, ACC

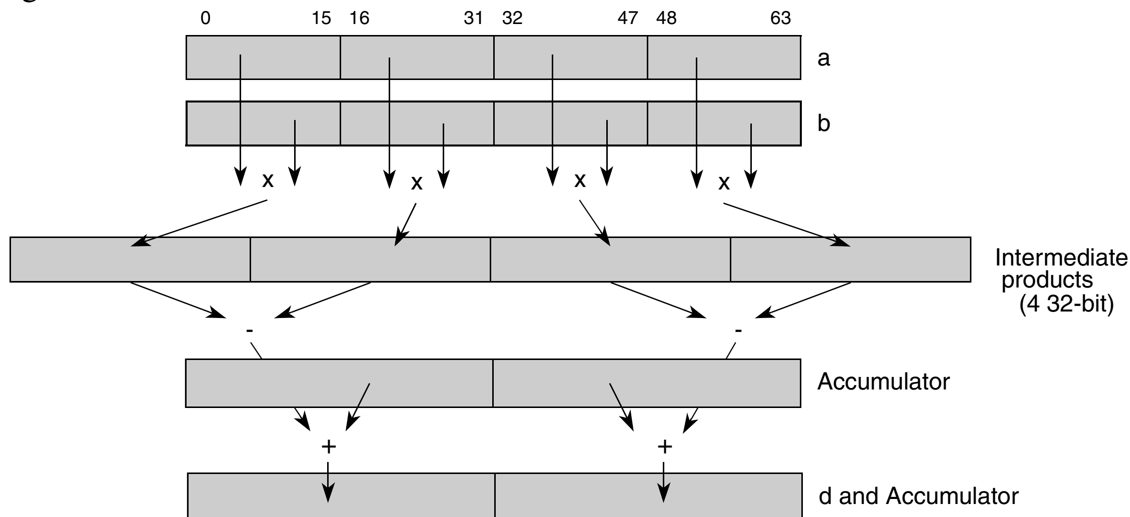


Figure 3-200. Vector Dot Product of Half Words, Subtract, Signed, Saturate, Integer and Accumulate Words (__ev_dotphsssiaaw)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evdotphsssiaaw d,a,b**

__ev_dotphsssiaaw3 __ev_dotphsssiaaw3

Vector Dot Product of Halfwords, Subtract, Signed, Saturate, Integer and Accumulate into Words, 3 operand

d = __ev_dotphsssiaaw3 (a,b,c)

```
// high dot
temph1_{0:31} ← b_{0:15} ×_{si} c_{0:15}; temph2_{0:31} ← b_{16:31} ×_{si} c_{16:31}
temph_{0:63} ← EXTS_{64}(temph1_{0:31}) - EXTS_{64}(temph2_{0:31}) + EXTS_{64}(a_{0:31})
ovh ← chk_ovf(temph_{30:32})
d_{0:31} ← SATURATE(ovh, temph_{30}, 0x8000_0000, 0x7FFF_FFFF, temph_{32:63})
//low
templ1_{0:31} ← b_{32:47} ×_{si} c_{32:47}; templ2_{0:31} ← b_{48:63} ×_{si} c_{48:63}
templ_{0:63} ← EXTS_{64}(templ1_{0:31}) - EXTS_{64}(templ2_{0:31}) + EXTS_{64}(a_{32:63})
ovl ← chk_ovf(templ_{30:32})
d_{32:63} ← SATURATE(ovl, templ_{30}, 0x8000_0000, 0x7FFF_FFFF, templ_{32:63})
// 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 destination, corresponding halfword pairs of signed integer elements in parameters **b** and **c** are multiplied producing a pair of 32-bit products. The intermediate 32-bit product of the odd halfword is subtracted from the intermediate product of the even halfword, and the difference is added together with the contents of the corresponding parameter **a** word, saturating if overflow or underflow occurs, and the sum is placed into the corresponding parameter **d** and accumulator word.

Other registers altered: SPEFSCR, ACC

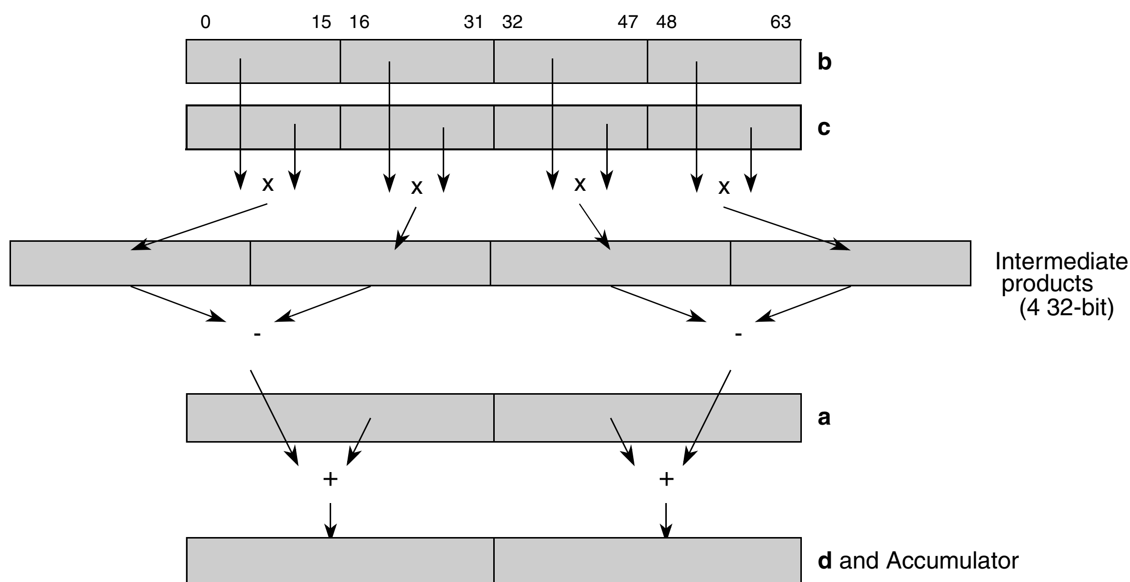


Figure 3-201. Vector Dot Product of Halfwords, Subtract, Signed, Saturate, Integer and Accumulate Words 3 op (__ev_dotphsssiaaw3)

d	a	b	c	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**d ←a** **evdotphsssiaaw3 d,b,c**

__ev_dotplohcsmi[a] __ev_dotplohcsmi[a]

Vector Dot Product of Low Halfwords, Complex, Signed, Modulo, Integer (to Accumulator)

d = __ev_dotplohcsmi (a,b) (A = 0)

d = __ev_dotplohcsmia (a,b) (A = 1)

```
// high dot - calculate real part of complex product
temph1[0:31] ← a[32:47] ×si b[32:47]
temph2[0:31] ← a[48:63] ×si b[48:63]
temph[0:31] ← temph1[0:31] - temph2[0:31] // modulo difference
d[0:31] ← temph[0:31]

//low dot - calculate imaginary part of complex product
templ1[0:31] ← b[32:47] ×si a[48:63]
templ2[0:31] ← a[32:47] ×si b[48:63]
templ[0:31] ← templ1[0:31] + templ2[0:31] // modulo sum
d[32:63] ← templ[0:31]

// update accumulator
if A = 1 then ACC[0:63] ←d[0:63]
```

For the high word element in the destination, corresponding halfword pairs of signed integer elements in the low halfwords of parameters **a** and **b** are multiplied producing a pair of 32-bit products. The difference of this pair of intermediate 32-bit products is placed into the high word of parameter **d**. For the low word element in the destination, halfword pairs of signed integer elements from the low halfwords of parameters **a** and **b** are multiplied after exchanging the low halfwords of parameter **a**, producing a pair of 32-bit products. The sum of this pair of intermediate 32-bit products is placed into the low word of parameter **d**. If A = 1, the result in parameter **d** is also placed into the accumulator.

Other registers altered: ACC (if A=1)

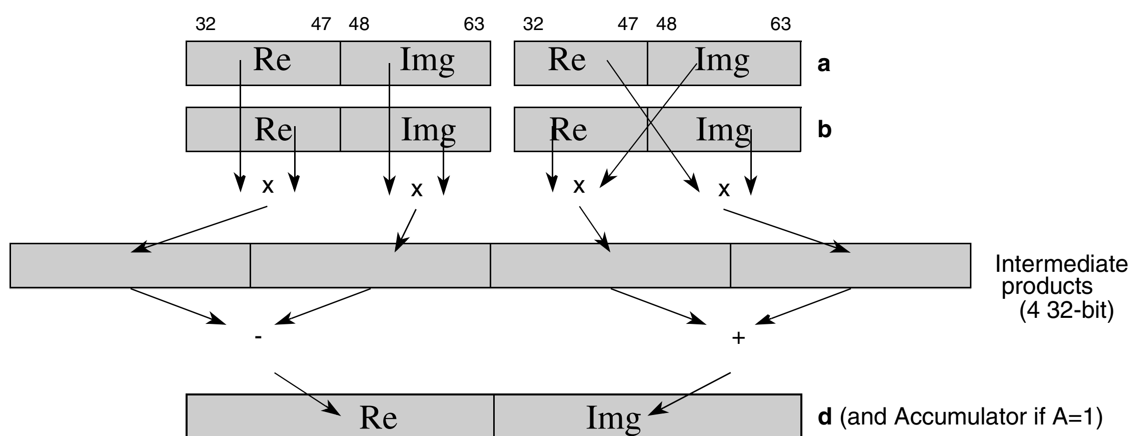


Figure 3-202. Vector Dot Product of Low Halfwords, Complex, Signed, Modulo, Integer (to Accumulator) (__ev_dotplohcsmi[a])

A	d	a	b	Maps to
A = 0	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evdotplohcsmi d,a,b**
A = 1	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evdotplohcsmia d,a,b**

__ev_dotplohcsmiaaw __ev_dotplohcsmiaaw

Vector Dot Product of Low Halfwords, Complex, Signed, Modulo, Integer and Accumulate into Words

d = __ev_dotplohcsmiaaw (a,b)

```
// high dot - calculate real part of complex product
temph1_{0:31} ← a_{32:47} ×_{si} b_{32:47}
temph2_{0:31} ← a_{48:63} ×_{si} b_{48:63}
temph_{0:31} ← temph1_{0:31} - temph2_{0:31} + ACC_{0:31} // modulo sum
d_{0:31} ← temph_{0:31}

//low dot - calculate imaginary part of complex product
templ1_{0:31} ← b_{32:47} ×_{si} a_{48:63}
templ2_{0:31} ← a_{32:47} ×_{si} b_{48:63}
templ_{0:31} ← templ1_{0:31} + templ2_{0:31} + ACC_{32:63} // modulo sum
d_{32:63} ← templ_{0:31}

// update accumulator
ACC_{0:63} ←d_{0:63}
```

For the high word element in the destination, corresponding halfword pairs of signed integer elements in the low halfwords of parameters **a** and **b** are multiplied producing a pair of 32-bit products. The difference of this pair of intermediate 32-bit products is added together with the contents of the upper accumulator word and the sum is placed into the high word of parameter **d**. For the low word element in the destination, halfword pairs of signed integer elements from the high halfwords of parameters **a** and **b** are multiplied after exchanging the high halfwords of parameter **a**, producing a pair of 32-bit products. The sum of this pair of intermediate 32-bit products is added together with the contents of the low accumulator word and the sum is placed into the low word of parameter **d** and the accumulator.

Other registers altered: ACC

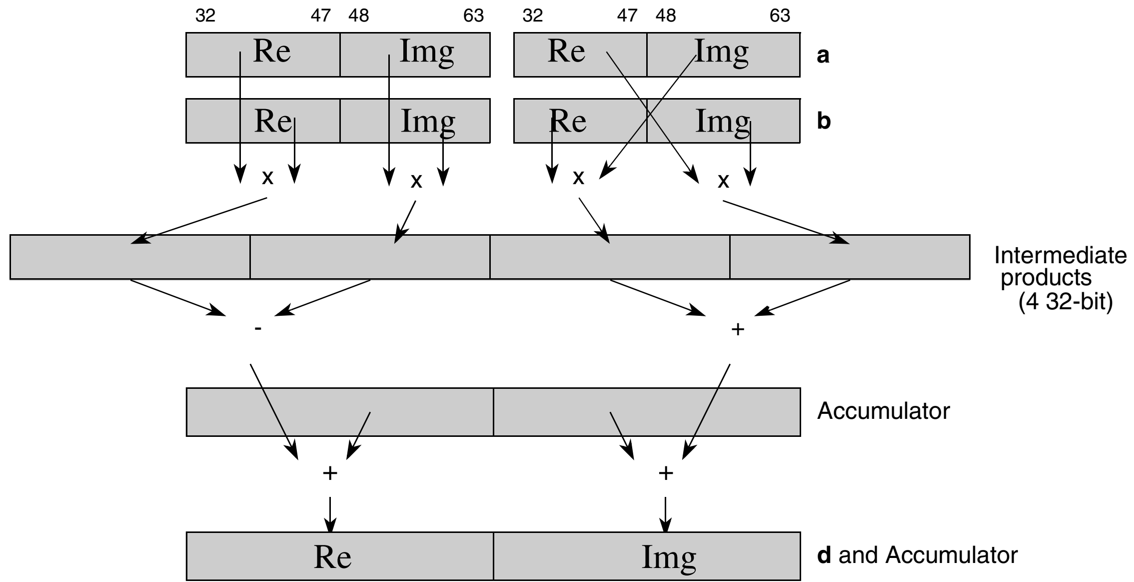


Figure 3-203. Vector Dot Product of Low Halfwords, Complex, Signed, Modulo, Integer and Accumulate Words (__ev_dotplohcsmiaaw)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evdotplohcsmiaaw d,a,b**

__ev_dotplohcsmiaaw3 __ev_dotplohcsmiaaw3

Vector Dot Product of Low Halfwords, Complex, Signed, Modulo, Integer and Accumulate into Words, 3 operand

d = __ev_dotplohcsmiaaw3 (a,b,c)

```
// high dot - calculate real part of complex product
temph1_{0:31} ← b_{32:47} ×_{si} c_{32:47}
temph2_{0:31} ← b_{48:63} ×_{si} c_{48:63}
temph_{0:31} ← temph1_{0:31} - temph2_{0:31} + a_{0:31} // modulo sum
d_{0:31} ← temph_{0:31}

//low dot - calculate imaginary part of complex product
templ1_{0:31} ← c_{32:47} ×_{si} b_{48:63}
templ2_{0:31} ← b_{32:47} ×_{si} c_{48:63}
templ_{0:31} ← templ1_{0:31} + templ2_{0:31} + a_{32:63} // modulo sum
d_{32:63} ← templ_{0:31}

// update accumulator
ACC_{0:63} ←d_{0:63}
```

For the high word element in the destination, corresponding halfword pairs of signed integer elements in the low halfwords of parameters **b** and **c** are multiplied producing a pair of 32-bit products. The difference of this pair of intermediate 32-bit products is added together with the contents of the upper parameter **a** word and the sum is placed into the high word of parameter **d**. For the low word element in the destination, halfword pairs of signed integer elements from the low halfwords of parameters **b** and **c** are multiplied after exchanging the low halfwords of parameter **b**, producing a pair of 32-bit products. The sum of this pair of intermediate 32-bit products is added together with the contents of the lower parameter **a** word and the sum is placed into the low word of parameter **d** and the accumulator.

Other registers altered: ACC

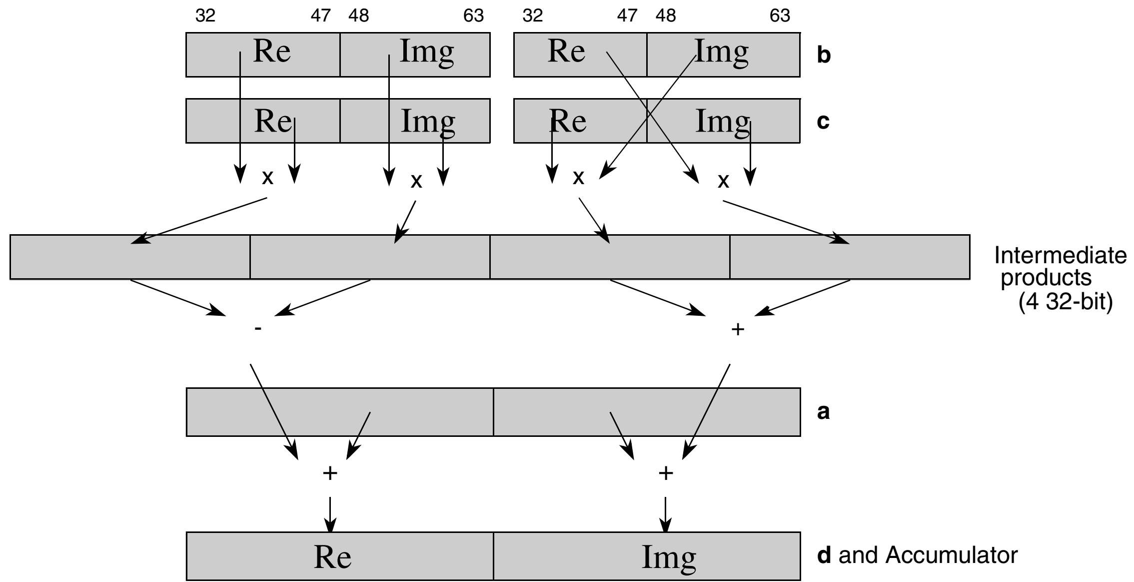


Figure 3-204. Vector Dot Product of Low Halfwords, Complex, Signed, Modulo, Integer and Accumulate Words 3 op (__ev_dotplohcsmiaaw3)

d	a	b	c	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**d ←a** **evdotplohcsmiaaw3 d,b,c**

__ev_dotplohcssf[a] __ev_dotplohcssf[a]

Vector Dot Product of Low Halfwords, Complex, Signed, Saturate, Fractional (to Accumulator)

d = __ev_dotplohcssf (a,b) (A = 0)

d = __ev_dotplohcssfa (a,b) (A = 1)

```
// high dot
temph1_{0:31} ← a_{32:47} ×_{sf} b_{32:47}
if (a_{32:47} = 0x8000) & (b_{32:47} = 0x8000) then
    temph1_{0:31} ←0x7FFF_FFFF //saturate
    movh ←1
else
    movh ←0
temph2_{0:31} ← a_{48:63} ×_{sf} b_{48:63}
if (a_{48:63} = 0x8000) & (b_{48:63} = 0x8000) then
    temph2_{0:31} ←0x7FFF_FFFF //saturate
    movh ←1
endif
temph_{0:63} ←EXTS_{64}(temph1_{0:31}) - EXTS_{64}(temph2_{0:31})
ovh ←temph_{31} ⊕ temph_{32}
d_{0:31} ←SATURATE(ovh, temph_{31}, 0x8000_0000, 0x7FFF_FFFF, temph_{32:63})

//low
templ1_{0:31} ← a_{48:63} ×_{sf} b_{32:47}
if (a_{48:63} = 0x8000) & (b_{32:47} = 0x8000) then
    templ1_{0:31} ←0x7FFF_FFFF //saturate
    movl ←1
else
    movl ←0
templ2_{0:31} ← a_{32:47} ×_{sf} b_{48:63}
if (a_{32:47} = 0x8000) & (b_{48:63} = 0x8000) then
    templ2_{0:31} ←0x7FFF_FFFF //saturate
    movl ←1
endif
templ_{0:63} ←EXTS_{64}(templ1_{0:31}) + EXTS_{64}(templ2_{0:31})
ovl ←templ_{31} ⊕ templ_{32}
d_{32:63} ←SATURATE(ovl, templ_{31}, 0x8000_0000, 0x7FFF_FFFF, templ_{32:63})
// update accumulator
if A = 1 then ACC_{0:63} ←d_{0:63}
// update SPEFSCR
SPEFSCR_{OVH} ←movh | ovh; SPEFSCR_{OV} ←movl | ovl
SPEFSCR_{SOVH} ←SPEFSCR_{SOVH} | movh | ovh; SPEFSCR_{SOV} ←SPEFSCR_{SOV} | movl | ovl
```

For the high word element in the destination, corresponding halfword pairs of signed fractional elements in the low halfwords of parameters **a** and **b** are multiplied producing a pair of 32-bit products. If both inputs of a multiply are -1.0, the result saturates to 0x7FFF_FFFF. The difference of this pair of intermediate 32-bit products is placed into the high word of parameter **d**, saturating if overflow or underflow occurs. For the low word element in the destination, halfword pairs of signed fractional elements from the low halfwords of parameters **a** and **b** are multiplied after exchanging the low halfwords of parameter **a**, producing a pair of 32-bit products. If both inputs of a multiply are -1.0, the result saturates to 0x7FFF_FFFF. The sum of this pair of intermediate 32-bit products is placed into the low word of parameter **d**, saturating if overflow or underflow occurs. If A = 1, the result in parameter **d** is also placed into the accumulator

The overflow and summary overflow bits are recorded in the SPEFSCR based on an overflow or underflow from either the multiply or the accumulation.

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

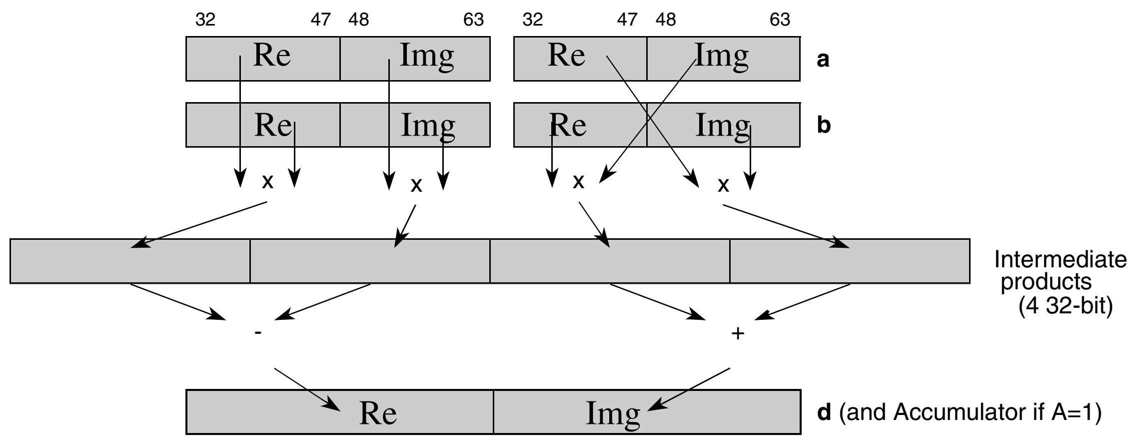


Figure 3-205. Vector Dot Product of Low Halfwords, Complex, Signed, Saturate, Fractional (to Accumulator) (__ev_dotplohcssf[a])

A	d	a	b	Maps to
A = 0	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evdotplohcssf d,a,b**
A = 1	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evdotplohcssfa d,a,b**

__ev_dotplohcssfaaw __ev_dotplohcssfaaw

Vector Dot Product of Low Halfwords, Complex, Signed, Saturate, Fractional and Accumulate into Words

d = __ev_dotplohcssfaaw (a,b)

```
// high dot
temph1_{0:31} ← a_{32:47} ×_{sf} b_{32:47}
if (a_{32:47} = 0x8000) & (b_{32:47} = 0x8000) then
    temph1_{0:31} ←0x7FFF_FFFF //saturate
    movh ←1
else
    movh ←0
temph2_{0:31} ← a_{48:63} ×_{sf} b_{48:63}
if (a_{48:63} = 0x8000) & (b_{48:63} = 0x8000) then
    temph2_{0:31} ←0x7FFF_FFFF //saturate
    movh ←1
endif
temph_{0:63} ← EXTS_{64}(temph1_{0:31}) - EXTS_{64}(temph2_{0:31}) + EXTS_{64}(ACC_{0:31})
ovh ← chk_ovf(temph_{30:32})
d_{0:31} ← SATURATE(ovh, temph_{30}, 0x8000_0000, 0x7FFF_FFFF, temph_{32:63})

//low
templ1_{0:31} ← a_{48:63} ×_{sf} b_{32:47}
if (a_{48:63} = 0x8000) & (b_{32:47} = 0x8000) then
    templ1_{0:31} ←0x7FFF_FFFF //saturate
    movl ←1
else
    movl ←0
templ2_{0:31} ← a_{32:47} ×_{sf} b_{48:63}
if (a_{32:47} = 0x8000) & (b_{48:63} = 0x8000) then
    templ2_{0:31} ←0x7FFF_FFFF //saturate
    movl ←1
endif
templ_{0:63} ← EXTS_{64}(templ1_{0:31}) + EXTS_{64}(templ2_{0:31}) + EXTS_{64}(ACC_{32:63})
ovl ← chk_ovf(templ_{30:32})
d_{32:63} ← SATURATE(ovl, templ_{30}, 0x8000_0000, 0x7FFF_FFFF, templ_{32:63})
// update accumulator
ACC_{0:63} ← d_{0:63}
// update SPEFSCR
SPEFSCR_{OVH} ← movh | ovh; SPEFSCR_{OV} ← movl | ovl
SPEFSCR_{SOVH} ← SPEFSCR_{SOVH} | movh | ovh; SPEFSCR_{SOV} ← SPEFSCR_{SOV} | movl | ovl
```

For the high word element in the destination, corresponding halfword pairs of signed fractional elements in the low halfwords of parameters **a** and **b** are multiplied producing a pair of 32-bit products. If both inputs of a multiply are -1.0, the result saturates to 0x7FFF_FFFF. The difference of this pair of intermediate 32-bit products is added to the high word of the accumulator, and the result is placed into the high word of parameter **d** and the accumulator, saturating if overflow or underflow occurs. For the low word element in the destination, halfword pairs of signed fractional elements from the low halfwords of parameters **a** and **b** are multiplied after exchanging the low halfwords of parameter **a**, producing a pair of 32-bit products. If both inputs of a multiply are -1.0, the result saturates to 0x7FFF_FFFF. The sum of this pair of intermediate 32-bit products and the low word of the accumulator is placed into the low word of parameter **d** and the accumulator.

The overflow and summary overflow bits are recorded in the SPEFSCR based on an overflow or underflow from either the multiply or the accumulation.

Other registers altered: SPEFSCR, ACC

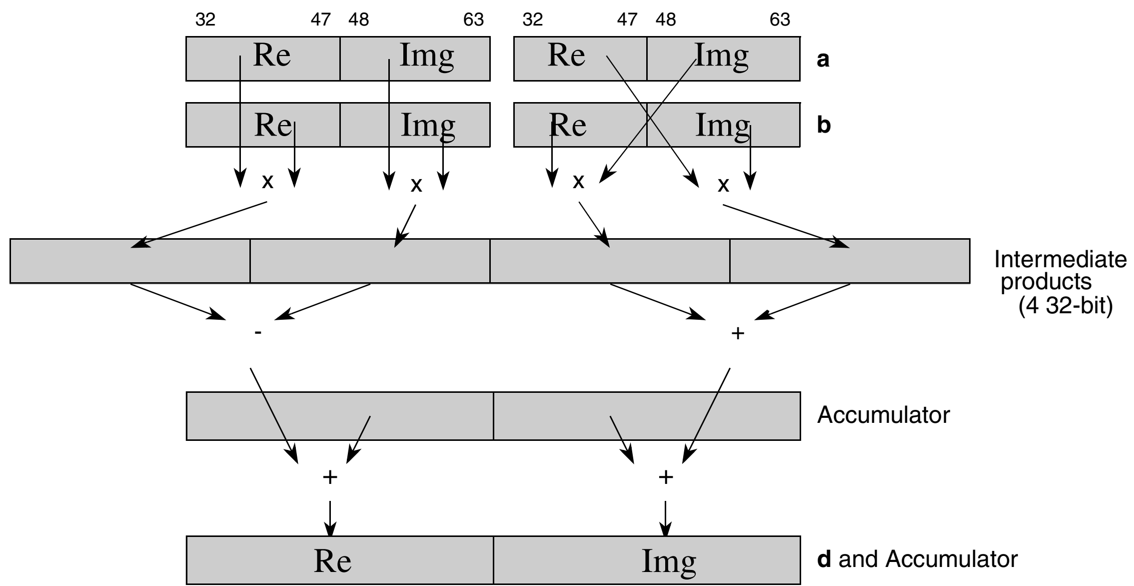


Figure 3-206. Vector Dot Product of Low Halfwords, Complex, Signed, Saturate, Fractional and Accumulate Words (__ev_dotplohcssfaaw)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evdotplohcssfaaw d,a,b**

__ev_dotplohcssfaaw3 __ev_dotplohcssfaaw3

Vector Dot Product of Low Halfwords, Complex, Signed, Saturate, Fractional and Accumulate into Words, 3 operand

d = __ev_dotplohcssfaaw3 (a,b,c)

```
// high dot
temph1_0:31 ←b_32:47 ×sf c_32:47
if (b_32:47 = 0x8000) & (c_32:47 = 0x8000) then
    temph1_0:31 ←0x7FFF_FFFF //saturate
    movh ←1
else
    movh ←0
temph2_0:31 ←b_48:63 ×sf c_48:63
if (b_48:63 = 0x8000) & (c_48:63 = 0x8000) then
    temph2_0:31 ←0x7FFF_FFFF //saturate
    movh ←1
endif
temph_0:63 ←EXTS_64(temph1_0:31) - EXTS_64(temph2_0:31) + EXTS_64(a_0:31)
ovh ←chk_ovf(temph_30:32)
d_0:31 ←SATURATE(ovh, temph_30, 0x8000_0000, 0x7FFF_FFFF, temph_32:63)

//low
templ1_0:31 ←b_48:63 ×sf c_32:47
if (b_48:63 = 0x8000) & (c_32:47 = 0x8000) then
    templ1_0:31 ←0x7FFF_FFFF //saturate
    movl ←1
else
    movl ←0
templ2_0:31 ←b_32:47 ×sf c_48:63
if (b_32:47 = 0x8000) & (c_48:63 = 0x8000) then
    templ2_0:31 ←0x7FFF_FFFF //saturate
    movl ←1
endif
templ_0:63 ←EXTS_64(templ1_0:31) + EXTS_64(templ2_0:31) + EXTS_64(a_32:63)
ovl ←chk_ovf(templ_30:32)
d_32:63 ←SATURATE(ovl, templ_30, 0x8000_0000, 0x7FFF_FFFF, templ_32:63)
// update accumulator
ACC_0:63 ←d_0:63
// update SPEFSCR
SPEFSCR_OVH ←movh | ovh; SPEFSCR_OV ←movl | ovl
SPEFSCR_SOVH ←SPEFSCR_SOVH | movh | ovh; SPEFSCR_SOV ←SPEFSCR_SOV | movl | ovl
```

For the high word element in the destination, corresponding halfword pairs of signed fractional elements in the low halfwords of parameters **b** and **c** are multiplied producing a pair of 32-bit products. If both inputs of a multiply are -1.0, the result saturates to 0x7FFF_FFFF. The difference of this pair of intermediate 32-bit products is added to the high word of parameter **a**, and the result is placed into the high word of parameter **d** and the accumulator, saturating if overflow or underflow occurs. For the low word element in the destination, halfword pairs of signed fractional elements from the low halfwords of parameters **b** and **c** are multiplied after exchanging the low halfwords of parameter **b**, producing a pair of 32-bit products. If both inputs of a multiply are -1.0, the result saturates to 0x7FFF_FFFF. The sum of this pair of intermediate 32-bit products and the low word of parameter **a** is placed into the low word of parameter **d** and the accumulator.

The overflow and summary overflow bits are recorded in the SPEFSCR based on an overflow or underflow from either the multiply or the accumulation.

Other registers altered: SPEFSCR, ACC

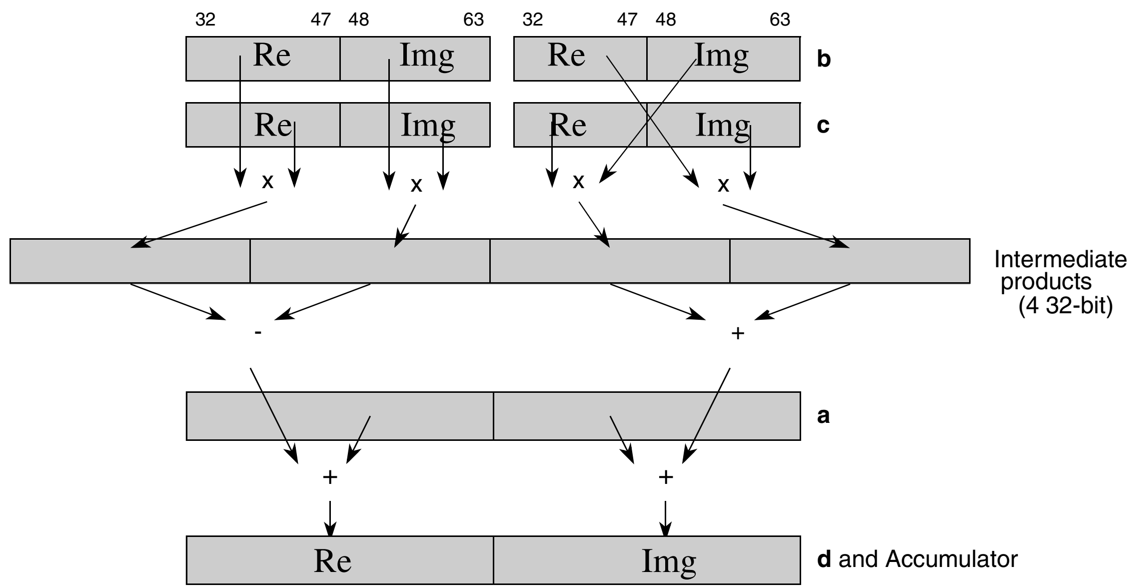


Figure 3-207. Vector Dot Product of Low Halfwords, Complex, Signed, Saturate, Fractional and Accumulate Words 3 op (__ev_dotplohcssfaaw3)

d	a	b	c	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**d ←a evdotplohcssfaaw3 d,b,c**

__ev_dotplohcssfr[a] __ev_dotplohcssfr[a]

Vector Dot Product of Low Halfwords, Complex, Signed, Saturate, Fractional, Round (to Accumulator)

d = __ev_dotplohcssfr (a,b) (A = 0)

d = __ev_dotplohcssfra (a,b) (A = 1)

```
// high dot
temph1_{0:31} ← a_{32:47} ×_{sf} b_{32:47}
if (a_{32:47} = 0x8000) & (b_{32:47} = 0x8000) then
    temph1_{0:31} ←0x7FFF_FFFF //saturate
    movh ←1
else
    movh ←0
temph2_{0:31} ← a_{48:63} ×_{sf} b_{48:63}
if (a_{48:63} = 0x8000) & (b_{48:63} = 0x8000) then
    temph2_{0:31} ←0x7FFF_FFFF //saturate
    movh ←1
endif
temph_{0:63} ← ROUND((EXTS_{64}(temph1_{0:31}) - EXTS_{64}(temph2_{0:31})),16)
ovh ← chk_ovf(temph_{30:32})
d_{0:31} ← SATURATE(ovh, temph_{30}, 0x8000_0000, 0x7FFF_0000, temph_{32:63})

//low
templ1_{0:31} ← a_{48:63} ×_{sf} b_{32:47}
if (a_{48:63} = 0x8000) & (b_{32:47} = 0x8000) then
    templ1_{0:31} ←0x7FFF_FFFF //saturate
    movl ←1
else
    movl ←0
templ2_{0:31} ← a_{32:47} ×_{sf} b_{48:63}
if (a_{32:47} = 0x8000) & (b_{48:63} = 0x8000) then
    templ2_{0:31} ←0x7FFF_FFFF //saturate
    movl ←1
endif
templ_{0:63} ← ROUND((EXTS_{64}(templ1_{0:31}) + EXTS_{64}(templ2_{0:31})),16)
ovl ← chk_ovf(templ_{30:32})
d_{32:63} ← SATURATE(ovl, templ_{30}, 0x8000_0000, 0x7FFF_0000, templ_{32:63})
// update accumulator
if A = 1 then ACC_{0:63} ←d_{0:63}
// update SPEFSCR
SPEFSCR_{OVH} ← movh | ovh; SPEFSCR_{OV} ← movl | ovl
SPEFSCR_{SOVH} ← SPEFSCR_{SOVH} | movh | ovh; SPEFSCR_{SOV} ← SPEFSCR_{SOV} | movl | ovl
```

For the high word element in the destination, corresponding halfword pairs of signed fractional elements in the low halfwords of parameters **a** and **b** are multiplied producing a pair of 32-bit products. If both inputs of a multiply are -1.0, the result saturates to 0x7FFF_FFFF. The difference of this pair of intermediate 32-bit products is rounded to 16 bits using the current fractional rounding mode in SPEFCSR, saturating if overflow or underflow occurs, and placed into the high word of parameter **d**. For the low word element in the destination, halfword pairs of signed fractional elements from the low halfwords of parameters **a** and **b** are multiplied after exchanging the low halfwords of parameter **a**, producing a pair of 32-bit products. If both inputs of a multiply are -1.0, the result saturates to 0x7FFF_FFFF. The sum of this pair of intermediate 32-bit products is rounded to 16 bits using the current fractional rounding mode in SPEFCSR, saturating if overflow or underflow occurs, and placed into the low word of parameter **d**. If A = 1, the result in parameter **d** is also placed into the accumulator

The overflow and summary overflow bits are recorded in the SPEFSCR based on an overflow or underflow from either the multiply or the accumulation.

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

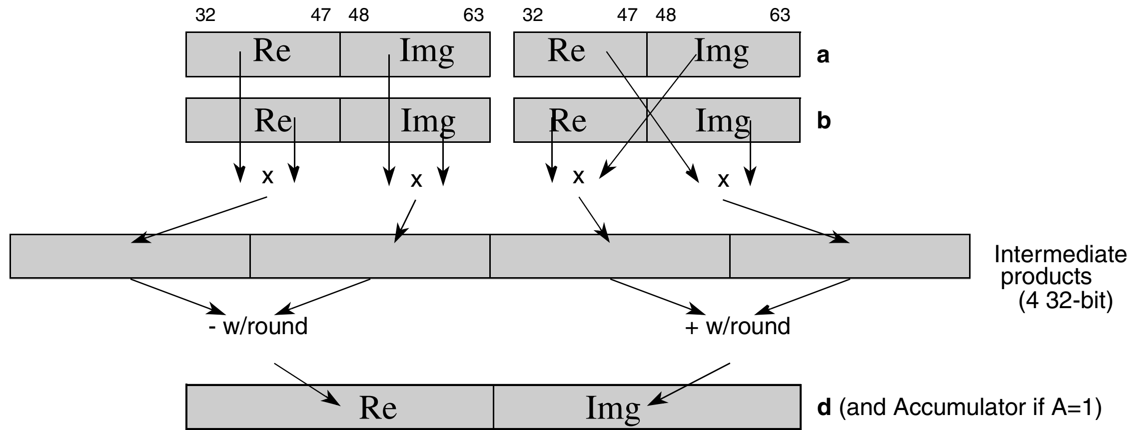


Figure 3-208. Vector Dot Product of Low Halfwords, Complex, Signed, Saturate, Fractional, Round (to Accumulator) (__ev_dotplohcssfr[a])

A	d	a	b	Maps to
A = 0	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evdotplohcssfr d,a,b**
A = 1	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evdotplohcssfra d,a,b**

__ev_dotplohcssfraaw   __ev_dotplohcssfraaw

Vector Dot Product of Low Halfwords, Complex, Signed, Saturate, Fractional, Round and Accumulate into Words

d = __ev_dotplohcssfraaw (a,b)

```
// high dot
temph1_0:31 ← a_32:47 ×sf b_32:47
if (a_32:47 = 0x8000) & (b_32:47 = 0x8000) then
    temph1_0:31 ←0x7FFF_FFFF //saturate
    movh ←1
else
    movh ←0
temph2_0:31 ← a_48:63 ×sf b_48:63
if (a_48:63 = 0x8000) & (b_48:63 = 0x8000) then
    temph2_0:31 ←0x7FFF_FFFF //saturate
    movh ←1
endif
temph_0:63 ← ROUND((EXTS_64(temph1_0:31) - EXTS_64(temph2_0:31) + EXTS_64(ACC_0:31)),16)
ovh ← chk_ovf(temph_30:32)
d_0:31 ← SATURATE(ovh, temph_30, 0x8000_0000, 0x7FFF_0000, temph_32:63)

//low
templ1_0:31 ← a_48:63 ×sf b_32:47
if (a_48:63 = 0x8000) & (b_32:47 = 0x8000) then
    templ1_0:31 ←0x7FFF_FFFF //saturate
    movl ←1
else
    movl ←0
templ2_0:31 ← a_32:47 ×sf b_48:63
if (a_32:47 = 0x8000) & (b_48:63 = 0x8000) then
    templ2_0:31 ←0x7FFF_FFFF //saturate
    movl ←1
endif
templ_0:63 ← ROUND((EXTS_64(templ1_0:31) + EXTS_64(templ2_0:31) + EXTS_64(ACC_32:63)),16)
ovl ← chk_ovf(templ_30:32)
d_32:63 ← SATURATE(ovl, templ_30, 0x8000_0000, 0x7FFF_0000, templ_32:63)
// update accumulator
ACC_0:63 ← d_0:63
// update SPEFSCR
SPEFSCR_OVH ← movh | ovh; SPEFSCR_OV ← movl | ovl
SPEFSCR_SOVH ← SPEFSCR_SOVH | movh | ovh; SPEFSCR_SOV ← SPEFSCR_SOV | movl | ovl
```

For the high word element in the destination, corresponding halfword pairs of signed fractional elements in the low halfwords of parameters **a** and **b** are multiplied producing a pair of 32-bit products. If both inputs of a multiply are -1.0, the result saturates to 0x7FFF_FFFF. The difference of this pair of intermediate 32-bit products is added to the high word of the accumulator and rounded to 16 bits using the current fractional rounding mode in SPEFCSR, saturating if overflow or underflow occurs, and placed into the high word of parameter **d** and the accumulator. For the low word element in the destination, halfword pairs of signed fractional elements from the low halfwords of parameters **a** and **b** are multiplied after exchanging the low halfwords of parameter **a**, producing a pair of 32-bit products. If both inputs of a multiply are -1.0, the result saturates to 0x7FFF_FFFF. The sum of this pair of intermediate 32-bit products and the low word of the accumulator is rounded to 16 bits using the current fractional rounding mode in SPEFCSR, saturating if overflow or underflow occurs, and placed into the low word of parameter **d** and the accumulator.

The overflow and summary overflow bits are recorded in the SPEFSCR based on an overflow or underflow from either the multiply or the accumulation.

Other registers altered: SPEFSCR, ACC

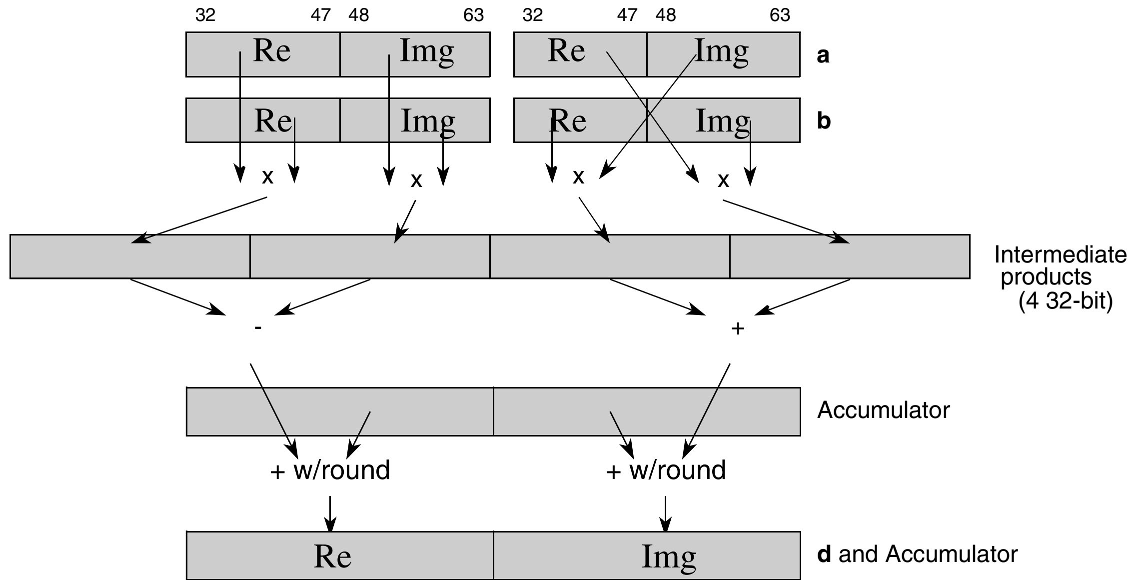


Figure 3-209. Vector Dot Product of Low Halfwords, Complex, Signed, Saturate, Fractional, Round and Accumulate Words (__ev_dotplohcssfraaw)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evdotplohcssfraaw d,a,b**

__ev_dotplohcssfraaw3 __ev_dotplohcssfraaw3

Vector Dot Product of Low Halfwords, Complex, Signed, Saturate, Fractional, Round and Accumulate into Words, 3 operand

d = __ev_dotplohcssfraaw3 (a,b,c)

```
// high dot
temph1_{0:31} ← b_{32:47} ×_{sf} c_{32:47}
if (b_{32:47} = 0x8000) & (c_{32:47} = 0x8000) then
    temph1_{0:31} ←0x7FFF_FFFF //saturate
    movh ←1
else
    movh ←0
temph2_{0:31} ← b_{48:63} ×_{sf} c_{48:63}
if (b_{48:63} = 0x8000) & (c_{48:63} = 0x8000) then
    temph2_{0:31} ←0x7FFF_FFFF //saturate
    movh ←1
endif
temph_{0:63} ← ROUND((EXTS_{64}(temph1_{0:31}) - EXTS_{64}(temph2_{0:31}) + EXTS_{64}(a_{0:31})),16)
ovh ← chk_ovf(temph_{30:32})
d_{0:31} ← SATURATE(ovh, temph_{30}, 0x8000_0000, 0x7FFF_0000, temph_{32:63})

//low
templ1_{0:31} ← b_{48:63} ×_{sf} c_{32:47}
if (b_{48:63} = 0x8000) & (c_{32:47} = 0x8000) then
    templ1_{0:31} ←0x7FFF_FFFF //saturate
    movl ←1
else
    movl ←0
templ2_{0:31} ← b_{32:47} ×_{sf} c_{48:63}
if (b_{32:47} = 0x8000) & (c_{48:63} = 0x8000) then
    templ2_{0:31} ←0x7FFF_FFFF //saturate
    movl ←1
endif
templ_{0:63} ← ROUND((EXTS_{64}(templ1_{0:31}) + EXTS_{64}(templ2_{0:31}) + EXTS_{64}(a_{32:63})),16)
ovl ← chk_ovf(temph_{30:32})
d_{32:63} ← SATURATE(ovl, templ_{30}, 0x8000_0000, 0x7FFF_0000, templ_{32:63})
// update accumulator
ACC_{0:63} ← d_{0:63}
// update SPEFSCR
SPEFSCR_{OVH} ← movh | ovh; SPEFSCR_{OV} ← movl | ovl
SPEFSCR_{SOVH} ← SPEFSCR_{SOVH} | movh | ovh; SPEFSCR_{SOV} ← SPEFSCR_{SOV} | movl | ovl
```

For the high word element in the destination, corresponding halfword pairs of signed fractional elements in the low halfwords of parameters **b** and **c** are multiplied producing a pair of 32-bit products. If both inputs of a multiply are -1.0, the result saturates to 0x7FFF_FFFF. The difference of this pair of intermediate 32-bit products is added to the high word of parameter **a** and rounded to 16 bits using the current fractional rounding mode in SPEFCSR, saturating if overflow or underflow occurs, and placed into the high word of parameter **d** and the accumulator. For the low word element in the destination, halfword pairs of signed fractional elements from the low halfwords of parameters **b** and **c** are multiplied after exchanging the low halfwords of parameter **a**, producing a pair of 32-bit products. If both inputs of a multiply are -1.0, the result saturates to 0x7FFF_FFFF. The sum of this pair of intermediate 32-bit products and the low word of parameter **a** is rounded to 16 bits using the current fractional rounding mode in SPEFCSR, saturating if overflow or underflow occurs, and placed into the low word of parameter **d** and the accumulator.

The overflow and summary overflow bits are recorded in the SPEFSCR based on an overflow or underflow from either the multiply or the accumulation.

Other registers altered: SPEFSCR, ACC

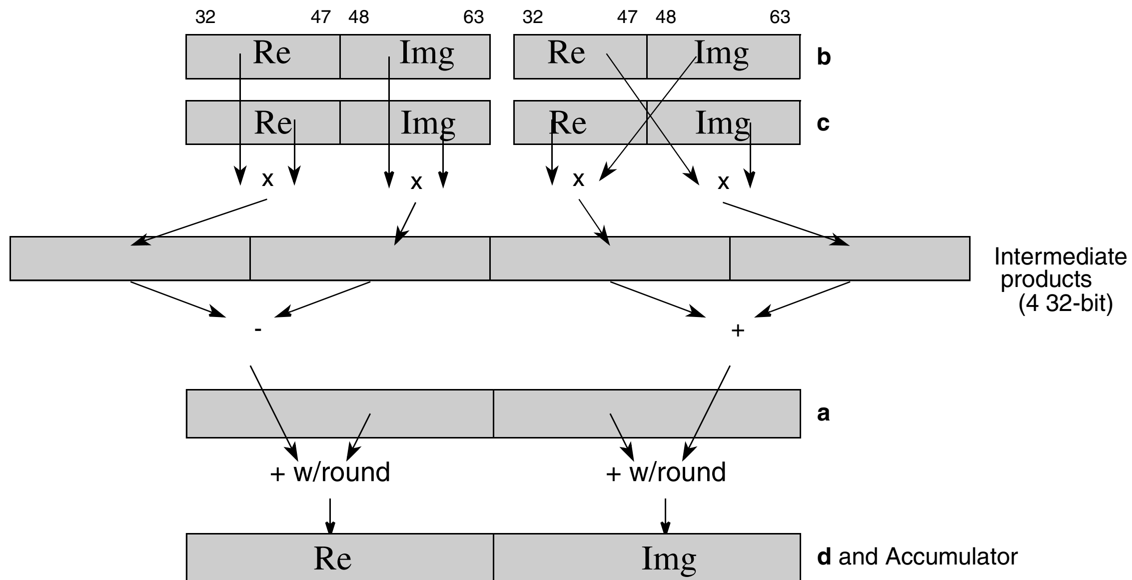


Figure 3-210. Vector Dot Product of Low Halfwords, Complex, Signed, Saturate, Fractional, Round and Accumulate Words 3 op (__ev_dotplohcssfraaw3)

d	a	b	c	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	d ←a evdotplohcssfraaw3 d,b,c

__ev_dotplohcssi[a] __ev_dotplohcssi[a]

Vector Dot Product of Low Halfwords, Complex, Signed, Saturate, Integer (to Accumulator)

d = __ev_dotplohcssi (a,b) (A = 0)

d = __ev_dotplohcssia (a,b) (A = 1)

```
// high dot
temph1_{0:31} ← a_{32:47} ×si b_{32:47}; temph2_{0:31} ← a_{48:63} ×si b_{48:63}
temph_{0:63} ← EXTS_{64}(temph1_{0:31}) - EXTS_{64}(temph2_{0:31})
ovh ← temph_{31} ⊕ temph_{32}
d_{0:31} ← SATURATE(ovh, temph_{31}, 0x8000_0000, 0x7FFF_FFFF, temph_{32:63})
//low
templ1_{0:31} ← a_{48:63} ×si b_{32:47}; templ2_{0:31} ← a_{32:47} ×si b_{48:63}
templ_{0:63} ← EXTS_{64}(templ1_{0:31}) + EXTS_{64}(templ2_{0:31})
ovl ← templ_{31} ⊕ templ_{32}
d_{32:63} ← SATURATE(ovl, templ_{31}, 0x8000_0000, 0x7FFF_FFFF, templ_{32:63})
// update accumulator
if A = 1 then 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 the high word element in the destination, corresponding halfword pairs of signed integer elements in the low halfwords of parameters **a** and **b** are multiplied producing a pair of 32-bit products. The difference of this pair of intermediate 32-bit products is placed into the high word of parameter **d**, saturating if overflow or underflow occurs. For the low word element in the destination, halfword pairs of signed integer elements from the low halfwords of parameters **a** and **b** are multiplied after exchanging the low halfwords of parameter **a**, producing a pair of 32-bit products. The sum of this pair of intermediate 32-bit products is placed into the low word of parameter **d**, saturating if overflow or underflow occurs. If A = 1, the result in parameter **d** is also placed into the accumulator.

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

32 47 48 63 32 47 48 63
Re Img Re Img **a**
Re Img Re Img **b**
x x x x
Intermediate products (4 32-bit)
- +
Re Img **d** (and Accumulator if A=1)

Figure 3-211. Vector Dot Product of Low Halfwords, Complex, Signed, Saturate, Integer (to Accumulator) (__ev_dotplohcssi[a])

A	d	a	b	Maps to
A = 0	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evdotplohcssi d,a,b**
A = 1	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evdotplohcssia d,a,b**

__ev_dotplohcssiaaw __ev_dotplohcssiaaw

Vector Dot Product of Low Halfwords, Complex, Signed, Saturate, Integer and Accumulate into Words

d = __ev_dotplohcssiaaw (a,b)

```
// high dot
temph1[0:31] ← a[32:47] ×si b[32:47]; temph2[0:31] ← a[48:63] ×si b[48:63]
temph[0:63] ← EXTS64(temph1[0:31]) - EXTS64(temph2[0:31]) + EXTS64(ACC[0:31])
ovh ← chk_ovf(temph[30:32])
d[0:31] ← SATURATE(ovh, temph[30], 0x8000_0000, 0x7FFF_FFFF, temph[32:63])

//low
templ1[0:31] ← a[48:63] ×si b[32:47]; templ2[0:31] ← a[32:47] ×si b[48:63]
templ[0:63] ← EXTS64(templ1[0:31]) + EXTS64(templ2[0:31]) + EXTS64(ACC[32:63])
ovl ← chk_ovf(templ[30:32])
d[32:63] ← SATURATE(ovl, templ[30], 0x8000_0000, 0x7FFF_FFFF, templ[32:63])
// 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 the high word element in the destination, corresponding halfword pairs of signed fractional elements in the low halfwords of parameters **a** and **b** are multiplied producing a pair of 32-bit products. The difference of this pair of intermediate 32-bit products is added to the high word of the accumulator, and the result is placed into the high word of parameter **d** and the accumulator, saturating if overflow or underflow occurs. For the low word element in the destination, halfword pairs of signed fractional elements from the low halfwords of parameters **a** and **b** are multiplied after exchanging the low halfwords of parameter **a**, producing a pair of 32-bit products. The sum of this pair of intermediate 32-bit products and the low word of the accumulator is placed into the low word of parameter **d** and the accumulator, saturating if overflow or underflow occurs.

Other registers altered: SPEFSCR, ACC

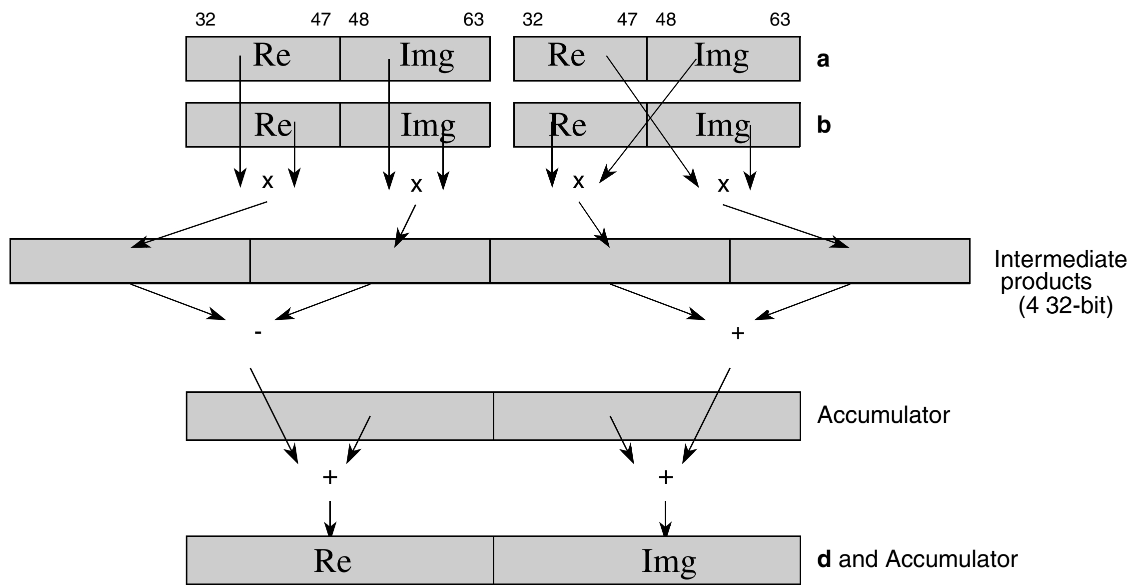


Figure 3-212. Vector Dot Product of Low Halfwords, Complex, Signed, Saturate, Integer and Accumulate Words (__ev_dotplohcssiaaw)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evdotplohcssiaaw d,a,b**

__ev_dotplohcssiaaw3 __ev_dotplohcssiaaw3

Vector Dot Product of Low Halfwords, Complex, Signed, Saturate, Integer and Accumulate into Words, 3 operand

d = __ev_dotplohcssiaaw3 (a,b,c)

```
// high dot
temph1_{0:31} ← b_{32:47} ×si c_{32:47}; temph2_{0:31} ← b_{48:63} ×si c_{48:63}
temph_{0:63} ← EXTS_64(temph1_{0:31}) - EXTS_64(temph2_{0:31}) + EXTS_64(a_{0:31})
ovh ← chk_ovf(temph_{30:32})
d_{0:31} ← SATURATE(ovh, temph_{30}, 0x8000_0000, 0x7FFF_FFFF, temph_{32:63})

//low
templ1_{0:31} ← b_{48:63} ×si c_{32:47}; templ2_{0:31} ← b_{32:47} ×si c_{48:63}
templ_{0:63} ← EXTS_64(templ1_{0:31}) + EXTS_64(templ2_{0:31}) + EXTS_64(a_{32:63})
ovl ← chk_ovf(templ_{30:32})
d_{32:63} ← SATURATE(ovl, templ_{30}, 0x8000_0000, 0x7FFF_FFFF, templ_{32:63})
// 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 the high word element in the destination, corresponding halfword pairs of signed fractional elements in the low halfwords of parameters **b** and **c** are multiplied producing a pair of 32-bit products. The difference of this pair of intermediate 32-bit products is added to the high word of parameter **a**, and the result is placed into the high word of parameter **d** and the accumulator, saturating if overflow or underflow occurs. For the low word element in the destination, halfword pairs of signed fractional elements from the low halfwords of parameters **b** and **c** are multiplied after exchanging the low halfwords of parameter **b**, producing a pair of 32-bit products. The sum of this pair of intermediate 32-bit products and the low word of parameter **a** is placed into the low word of parameter **d** and the accumulator, saturating if overflow or underflow occurs.

Other registers altered: SPEFSCR, ACC

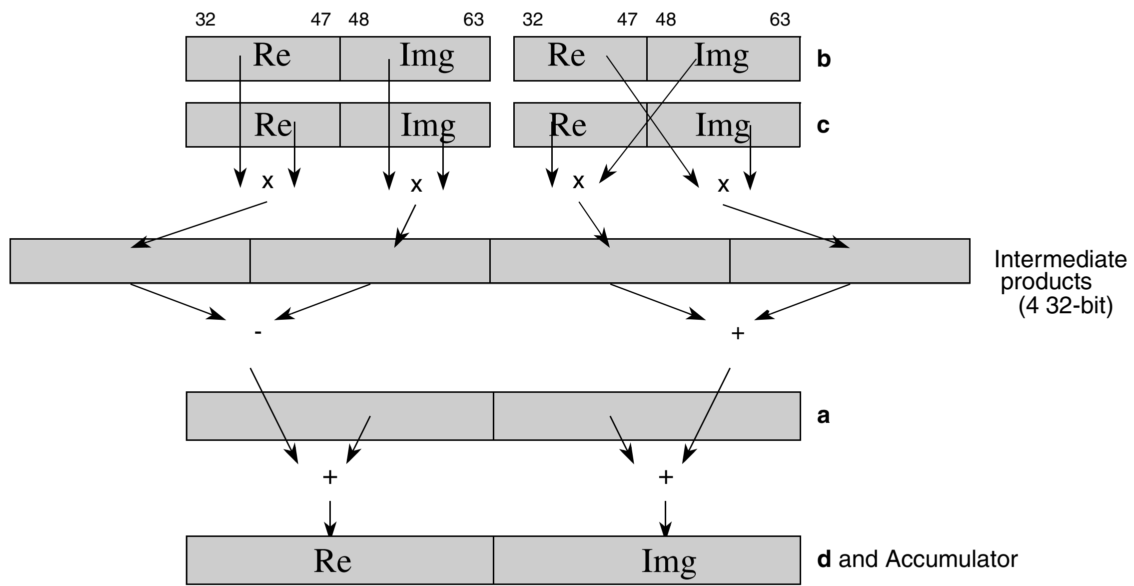


Figure 3-213. Vector Dot Product of Low Halfwords, Complex, Signed, Saturate, Integer and Accumulate Words (__ev_dotplohcssiaaw3)

d	a	b	c	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	d ←a evdotplohcssiaaw3 d,b,c

__ev_dotpwasmi[a] __ev_dotpwasmi[a]

Vector Dot Product of Words, Add, Signed, Modulo, integer (to Accumulator)

d = __ev_dotpwasmi (a,b) (A = 0)

d = __ev_dotpwasmia (a,b) (A = 1)

```
temph0:63 ← a0:31 ×si b0:31
templ0:63 ← a32:63 ×si b32:63
temp0:31 ← temph0:63 + templ0:63 // modulo sum
d0:63 ← temp0:63

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

Corresponding pairs of signed integer word elements in parameters **a** and **b** are multiplied producing a pair of 64-bit products. This pair of intermediate products is added together and the sum 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)

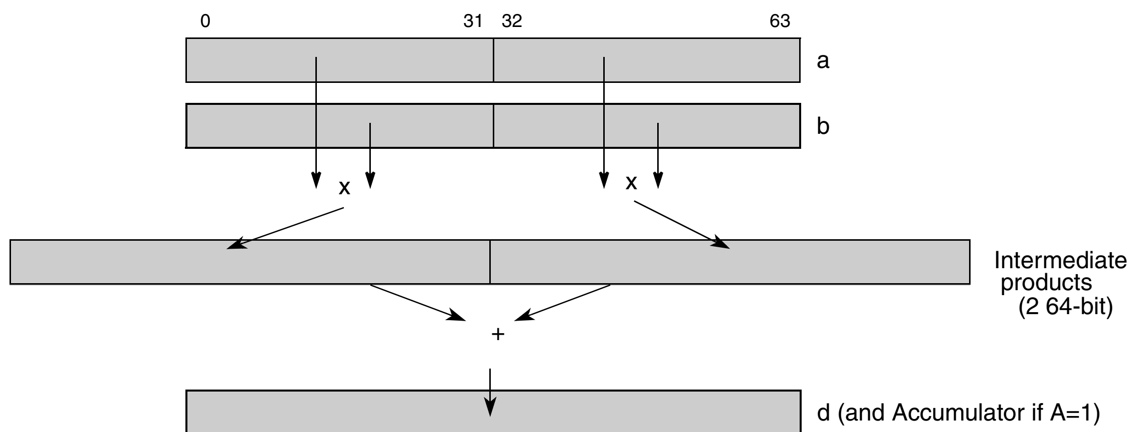


Figure 3-214. Vector Dot Product of Words, Add, Signed, Modulo, Integer (to Accumulator) (__ev_dotpwasmi[a])

A	d	a	b	Maps to
A = 0	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evdotpwasmi d,a,b**
A = 1	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evdotpwasmia d,a,b**

__ev_dotpwasmiaa __ev_dotpwasmiaa

Vector Dot Product of Words, Add, Signed, Modulo, Integer and Accumulate

d = __ev_dotpwasmiaa (a,b)

```
temph0:63 ← a0:31 ×si b0:31
templ0:63 ← a32:63 ×si b32:63
temp0:63 ← temph0:63 + templ0:63 + ACC0:63 // modulo sum
d0:63 ← temp0:63

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

Corresponding pairs of signed integer word elements in parameters **a** and **b** are multiplied producing a pair of 64-bit products. This pair of intermediate products is added together with the contents of the accumulator and the sum is placed into parameter **d** and the accumulator.

Other registers altered: ACC

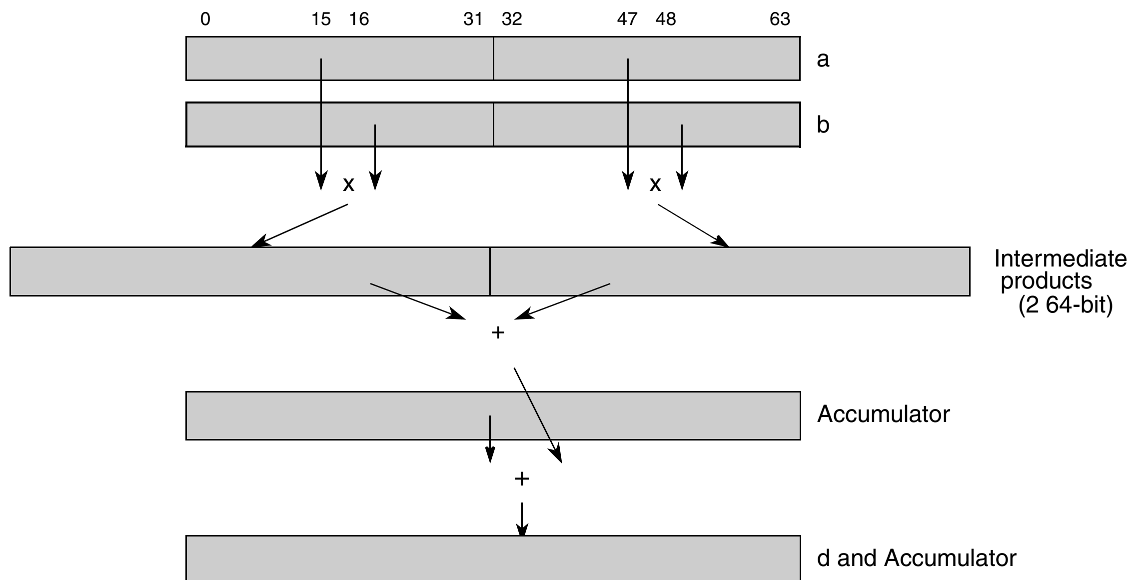


Figure 3-215. Vector Dot Product of Words, Add, Signed, Modulo, Integer and Accumulate (__ev_dotpwasmiaa)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evdotpwasmiaa d,a,b**

__ev_dotpwasmiaa3 __ev_dotpwasmiaa3

Vector Dot Product of Words, Add, Signed, Modulo, Integer and Accumulate, 3 operand

d = __ev_dotpwasmiaa3 (a,b,c)

$temph_{0:63} \leftarrow b_{0:31} \times_{si} c_{0:31}$
$templ_{0:63} \leftarrow b_{32:63} \times_{si} c_{32:63}$
$temp_{0:63} \leftarrow temph_{0:63} + templ_{0:63} + a_{0:63}$ // modulo sum
$d_{0:63} \leftarrow temp_{0:63}$

// update accumulator
$ACC_{0:63} \leftarrow d_{0:63}$

Corresponding pairs of signed integer word elements in parameters **b** and **c** are multiplied producing a pair of 64-bit products. This pair of intermediate products is added together with the contents of parameter **a** and the sum is placed into parameter **d** and the accumulator.

Other registers altered: ACC

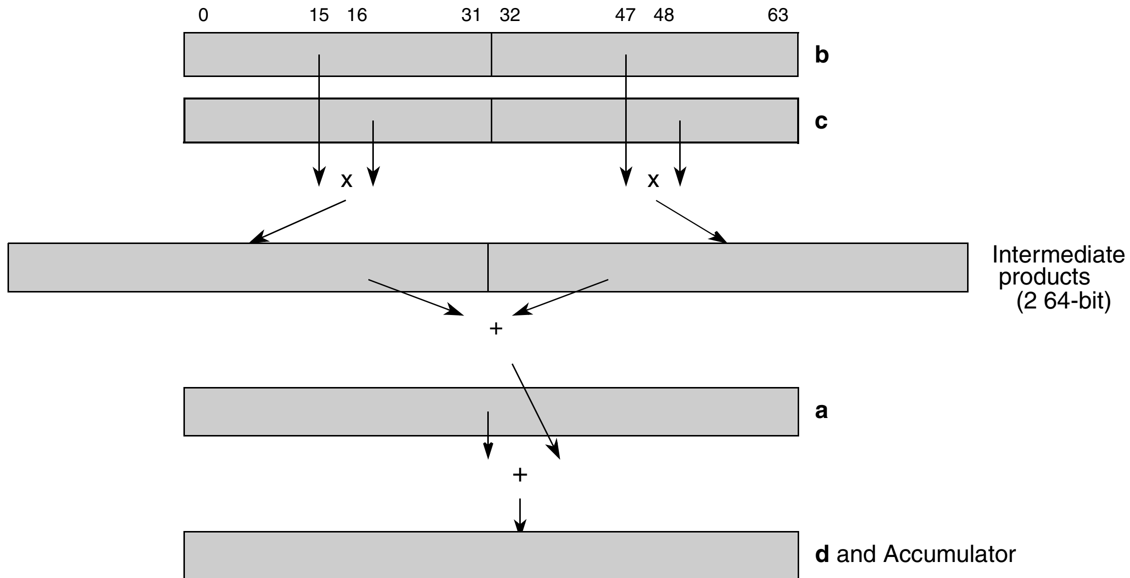


Figure 3-216. Vector Dot Product of Words, Add, Signed, Modulo, Integer and Accumulate 3 op (__ev_dotpwasmiaa3)

d	a	b	c	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	d ←a evdotpwasmiaa3 d,b,c

__ev_dotpwassi[a] __ev_dotpwassi[a]

Vector Dot Product of Words, Add, Signed, Saturate, integer (to Accumulator)

d = __ev_dotpwassi (a,b) (A = 0)

d = __ev_dotpwassia (a,b) (A = 1)

```
temph0:63 ← a0:31 ×si b0:31
templ0:63 ← a32:63 ×si b32:63

temp0:64 ← EXTS(temph0:63) + EXTS(templ0:63)
ov ← (temp0 ⊕ temp1)
d0:63 ← SATURATE(ov, temp0, 0x8000_0000_0000_0000, 0x7FFF_FFFF_FFFF_FFFF, temp1:64)
// update accumulator
if A = 1 then ACC0:63 ←d0:63
// update SPEFSCR
SPEFSCROVH ← 0; SPEFSCROV ← ov
SPEFSCRSOV ← SPEFSCRSOV | ov
```

Corresponding pairs of signed integer word elements in parameters **a** and **b** are multiplied producing a pair of 64-bit products. This pair of intermediate 64-bit products is added together, saturating if overflow or underflow occurs, and the sum is placed into parameter **d**. If A = 1, the result in parameter **d** is also placed into the accumulator.

The overflow and summary overflow bits are recorded in the SPEFSCR based on an overflow or underflow from the accumulation operations.

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

Figure 3-217. Vector Dot Product of Words, Add, Signed, Saturate, Integer [to Accumulator] (__ev_dotpwassi[a])

A	d	a	b	Maps to
A = 0	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evdotpwassi d,a,b**
A = 1	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evdotpwassia d,a,b**

__ev_dotpwassiaa __ev_dotpwassiaa

Vector Dot Product of Words, Add, Signed, Saturate, Integer and Accumulate

d = __ev_dotpwassiaa (a,b)

```
temph0:63 ← a0:31 ×si b0:31
templ0:63 ← a32:63 ×si b32:63

temp0:64 ← EXTS(ACC0:63) + EXTS(temph0:63) + EXTS(templ0:63)
ov ← (temp0 ⊕ temp1)
d0:63 ← SATURATE(ov, temp0, 0x8000_0000_0000_0000, 0x7FFF_FFFF_FFFF_FFFF, temp1:64)
// update accumulator
if A = 1 then ACC0:63 ←d0:63
// update SPEFSCR
SPEFSCROVH ← 0; SPEFSCROV ← mov | ov
SPEFSCRSOV ← SPEFSCRSOV | mov | ov
```

Corresponding pairs of signed integer word elements in parameters **a** and **b** are multiplied producing a pair of 64-bit products. This pair of intermediate 64-bit products is added together with the contents of the accumulator, saturating if overflow or underflow occurs in the final sum, and the sum is placed into parameter **d** and the accumulator.
The overflow and summary overflow bits are recorded in the SPEFSCR based on an overflow or underflow from the accumulation operations.

Other registers altered: SPEFSCR ACC

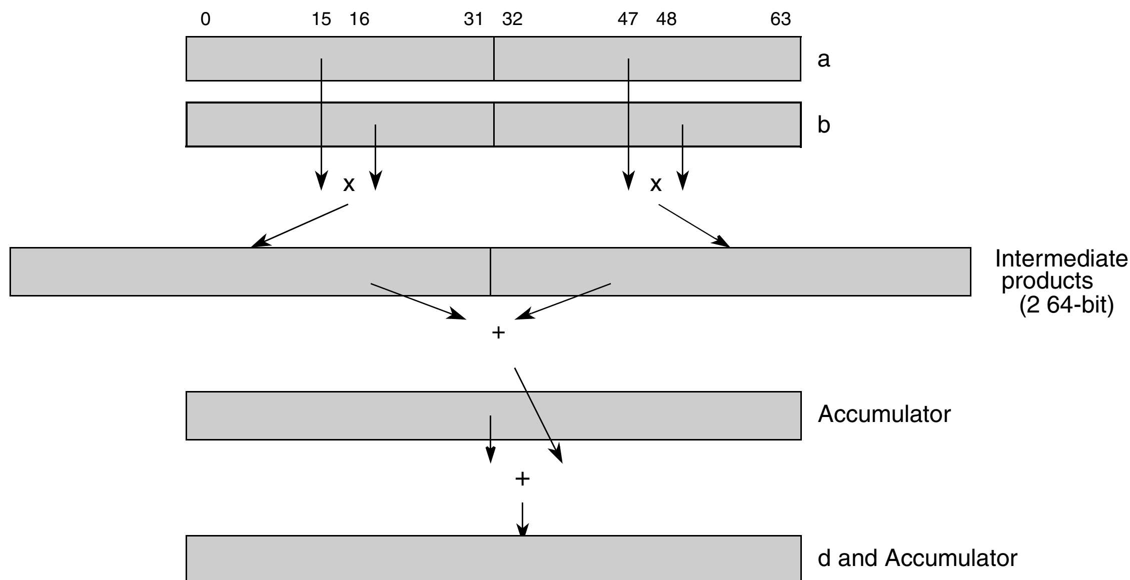


Figure 3-218. Vector Dot Product of Words, Add, Signed, Saturate, Integer and Accumulate (__ev_dotpwassiaa)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evdotpwassiaa d,a,b**

__ev_dotpwassiaa3 __ev_dotpwassiaa3

Vector Dot Product of Words, Add, Signed, Saturate, Integer and Accumulate, 3 operand

d = __ev_dotpwassiaa3 (a,b,c)

```
temph0:63 ←b0:31 ×si c0:31
templ0:63 ←b32:63 ×si c32:63

temp0:66 ←EXTS67(a0:63) + EXTS67(temph0:63) + EXTS67(templ0:63)
ov ←chk_ovf(temp0:3)
d0:63 ←SATURATE(ov, temp0, 0x8000_0000_0000_0000, 0x7FFF_FFFF_FFFF_FFFF, temp3:66)
// update accumulator
ACC0:63 ←d0:63
// update SPEFSCR
SPEFSCROVH ←0; SPEFSCROV ←mov | ov
SPEFSCRSOV ←SPEFSCRSOV | mov | ov
```

Corresponding pairs of signed integer word elements in parameters **b** and **c** are multiplied producing a pair of 64-bit products. This pair of intermediate 64-bit products is added together with the contents of parameter **a**, saturating if overflow or underflow occurs in the final sum, and the sum is placed into parameter **d** and the accumulator.

The overflow and summary overflow bits are recorded in the SPEFSCR based on an overflow or underflow from the accumulation operations.

Other registers altered: SPEFSCR ACC

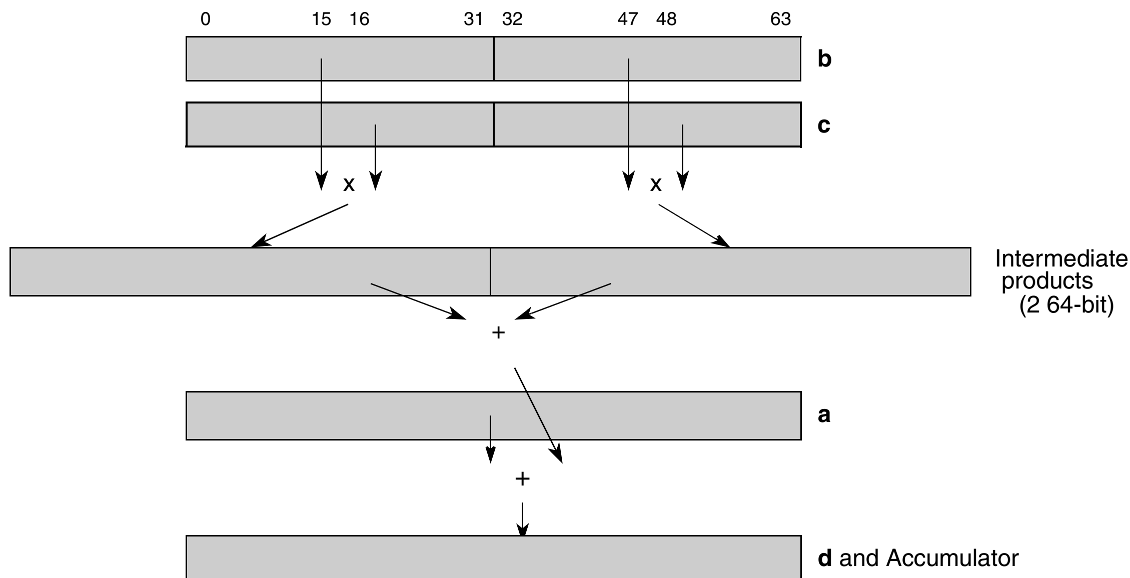


Figure 3-219. Vector Dot Product of Words, Add, Signed, Saturate, Integer and Accumulate 3 op (__ev_dotpwassiaa3)

d	a	b	c	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**d ←a evdotpwassiaa3 d,b,c**

__ev_dotpwasumi[a]   __ev_dotpwasumi[a]

Vector Dot Product of Words, Add, Signed by Unsigned, Modulo, integer (to Accumulator)

d = __ev_dotpwasumi (a,b) (A = 0)

d = __ev_dotpwasumia (a,b) (A = 1)

```
temph0:63 ←a0:31 ×sui b0:31
templ0:63 ←a32:63 ×sui b32:63
temp0:63 ←temph0:63 + templ0:63 // modulo sum
d0:63 ←temp0:63

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

Corresponding pairs of signed integer word elements in parameter **a** and unsigned integer word elements in parameter **b** are multiplied producing a pair of 64-bit products. This pair of intermediate products is added together and the sum 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)

0 31 32 63
a (signed)
b (unsigned)
x x
Intermediate products (2 64-bit)
+
d (and Accumulator if A=1)

Figure 3-220. Vector Dot Product of Words, Add, Signed by Unsigned, Modulo, Integer (to Accumulator) (__ev_dotpwasumi[a])

A	d	a	b	Maps to
A = 0	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evdotpwasumi d,a,b**
A = 1	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evdotpwasumia d,a,b**

__ev_dotpwasumiaa __ev_dotpwasumiaa

Vector Dot Product of Words, Add, Signed by Unsigned, Modulo, Integer and Accumulate

d = __ev_dotpwasumiaa (a,b)

$$temph_{0:63} \leftarrow a_{0:31} \times_{sui} b_{0:31}$$
$$templ_{0:63} \leftarrow a_{32:63} \times_{sui} b_{32:63}$$
$$temp_{0:63} \leftarrow temph_{0:63} + templ_{0:63} + ACC_{0:63} \text{ // modulo sum}$$
$$d_{0:63} \leftarrow temp_{0:63}$$

// update accumulator
$$ACC_{0:63} \leftarrow d_{0:63}$$

Corresponding pairs of signed integer word elements in parameter **a** and unsigned integer word elements in parameter **b** are multiplied producing a pair of 64-bit products. This pair of intermediate products is added together with the contents of the accumulator and the sum is placed into parameter **d** and the accumulator.

Other registers altered: ACC

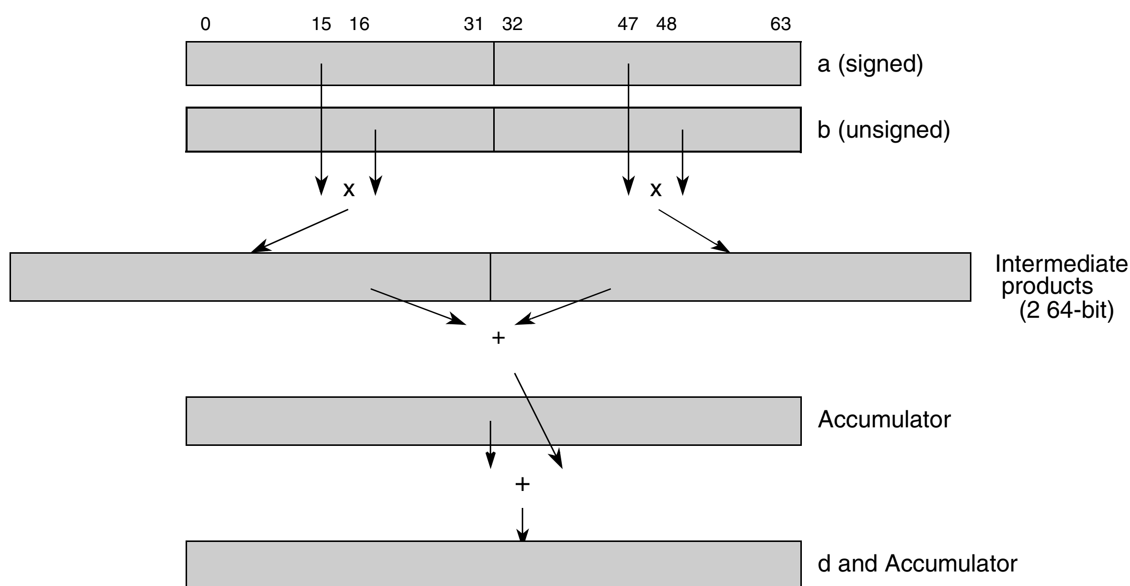


Figure 3-221. Vector Dot Product of Words, Add, Signed by Unsigned, Modulo, Integer and Accumulate (__ev_dotpwasumiaa)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evdotpwasumiaa d,a,b**

__ev_dotpwasumiaa3 __ev_dotpwasumiaa3

Vector Dot Product of Words, Add, Signed by Unsigned, Modulo, Integer and Accumulate, 3 operand

d = __ev_dotpwasumiaa3 (a,b,c)

```
temph0:63 ←b0:31 ×sui c0:31
templ0:63 ←b32:63 ×sui c32:63
temp0:63 ←temph0:63 + templ0:63 + a0:63 // modulo sum
d0:63 ←temp0:63

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

Corresponding pairs of signed integer word elements in parameter **b** and unsigned integer word elements in parameter **c** are multiplied producing a pair of 64-bit products. This pair of intermediate products is added together with the contents of parameter **a** and the sum is placed into parameter **d** and the accumulator.

Other registers altered: ACC

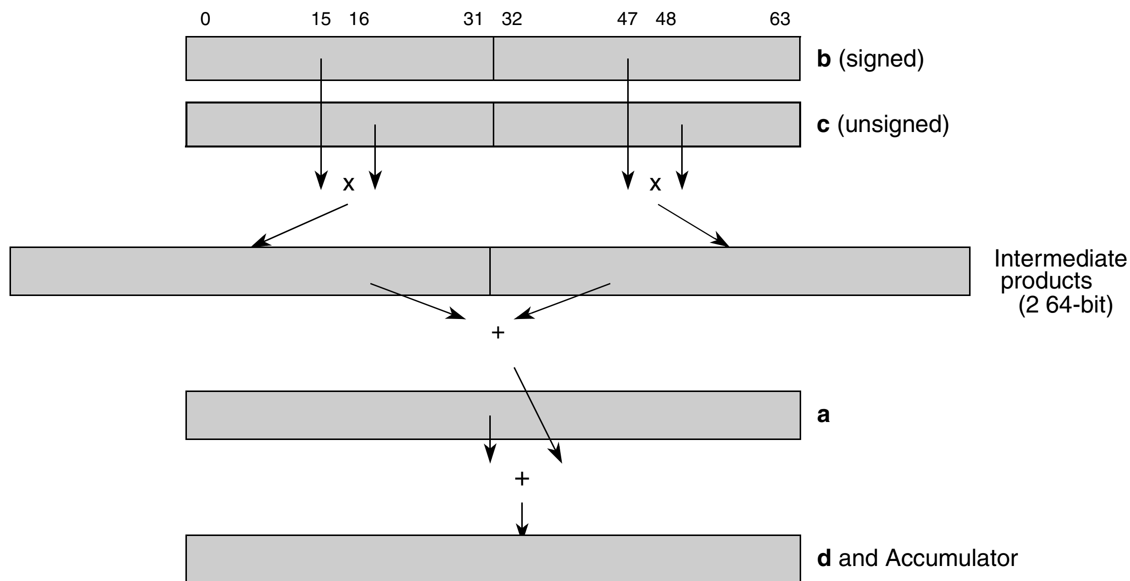


Figure 3-222. Vector Dot Product of Words, Add, Signed by Unsigned, Modulo, Integer and Accumulate 3 op (__ev_dotpwasumiaa3)

d	a	b	c	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	d ←a evdotpwasumiaa3 d,b,c

__ev_dotpwasusi[a] __ev_dotpwasusi[a]

Vector Dot Product of Words, Add, Signed by Unsigned, Saturate, integer (to Accumulator)

d = __ev_dotpwasusi (a,b) (A = 0)

d = __ev_dotpwasusia (a,b) (A = 1)

```
temph0:63 ←a0:31 ×sui b0:31
templ0:63 ←a32:63 ×sui b32:63

temp0:64 ←EXTS(temph0:63) + EXTS(templ0:63)
ov ← (temp0 ⊕ temp1)
d0:63 ←SATURATE(ov, temp0, 0x8000_0000_0000_0000, 0x7FFF_FFFF_FFFF_FFFF, temp1:64)
// update accumulator
if A = 1 then ACC0:63 ←d0:63
// update SPEFSCR
SPEFSCROVH ←0; SPEFSCROV ←ov
SPEFSCRSOV ←SPEFSCRSOV | ov
```

Corresponding pairs of signed integer word elements in parameter **a** and unsigned integer word elements in parameter **b** are multiplied producing a pair of 64-bit products. This pair of intermediate 64-bit products is added together, saturating if overflow or underflow occurs, and the sum is placed into parameter **d**. If A = 1, the result in parameter **d** is also placed into the accumulator.

The overflow and summary overflow bits are recorded in the SPEFSCR based on an overflow or underflow from the accumulation operations.

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

Figure 3-223. Vector Dot Product of Words, Add, Signed by Unsigned, Saturate, Integer [to Accumulator] (__ev_dotpwasusi[a])

A	d	a	b	Maps to
A = 0	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evdotpwasusi d,a,b**
A = 1	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evdotpwasusia d,a,b**

__ev_dotpwasusiaa __ev_dotpwasusiaa

Vector Dot Product of Words, Add, Signed by Unsigned, Saturate, Integer and Accumulate

d = __ev_dotpwasusiaa (a,b)

```
temph0:63 ← a0:31 ×sui b0:31
templ0:63 ← a32:63 ×sui b32:63

temp0:64 ← EXTS(ACC0:63) + EXTS(temph0:63) + EXTS(templ0:63)
ov ← (temp0 ⊕ temp1)
d0:63 ← SATURATE(ov, temp0, 0x8000_0000_0000_0000, 0x7FFF_FFFF_FFFF_FFFF, temp1:64)
// update accumulator
ACC0:63 ←d0:63
// update SPEFSCR
SPEFSCROVH ← 0; SPEFSCROV ← ov
SPEFSCRSOV ← SPEFSCRSOV | ov
```

Corresponding pairs of signed integer word elements in parameter **a** and unsigned integer word elements in parameter **b** are multiplied producing a pair of 64-bit products. This pair of intermediate 64-bit products is added together with the contents of the accumulator, saturating if overflow or underflow occurs in the final sum, and the sum is placed into parameter **d** and the accumulator.

The overflow and summary overflow bits are recorded in the SPEFSCR based on an overflow or underflow from the accumulation operations.

Other registers altered: SPEFSCR ACC

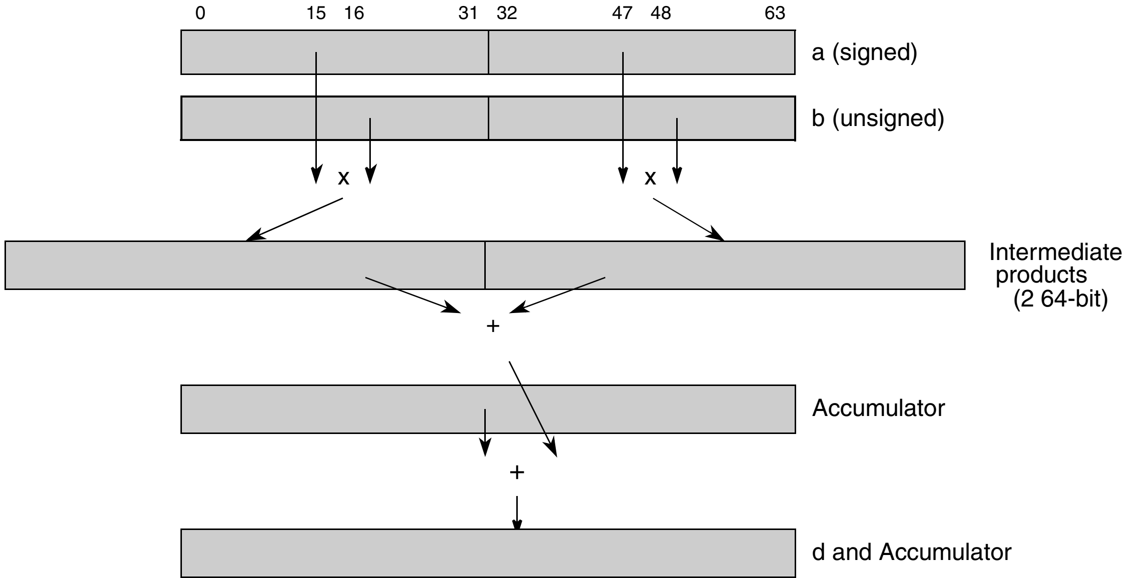


Figure 3-224. Vector Dot Product of Words, Add, Signed by Unsigned, Saturate, Integer and Accumulate (__ev_dotpwasusiaa)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evdotpwasusiaa d,a,b**

__ev_dotpwasusiaa3 __ev_dotpwasusiaa3

Vector Dot Product of Words, Add, Signed by Unsigned, Saturate, Integer and Accumulate, 3 operand

d = __ev_dotpwasusiaa3 (a,b,c)

```
temph0:63 ←b0:31 ×sui c0:31
templ0:63 ←b32:63 ×sui c32:63

temp0:66 ←EXTS67(a0:63) + EXTS67(temph0:63) + EXTS67(templ0:63)
ov ←chk_ovf(temp0:3)
d0:63 ←SATURATE(ov, temp0, 0x8000_0000_0000_0000, 0x7FFF_FFFF_FFFF_FFFF, temp3:66)
// update accumulator
ACC0:63 ←d0:63
// update SPEFSCR
SPEFSCROVH ←0; SPEFSCROV ←ov
SPEFSCRSOV ←SPEFSCRSOV | ov
```

Corresponding pairs of signed integer word elements in parameter **b** and unsigned integer word elements in parameter **c** are multiplied producing a pair of 64-bit products. This pair of intermediate 64-bit products is added together with the contents of parameter **a**, saturating if overflow or underflow occurs in the final sum, and the sum is placed into parameter **d** and the accumulator. The overflow and summary overflow bits are recorded in the SPEFSCR based on an overflow or underflow from the accumulation operations.

Other registers altered: SPEFSCR ACC

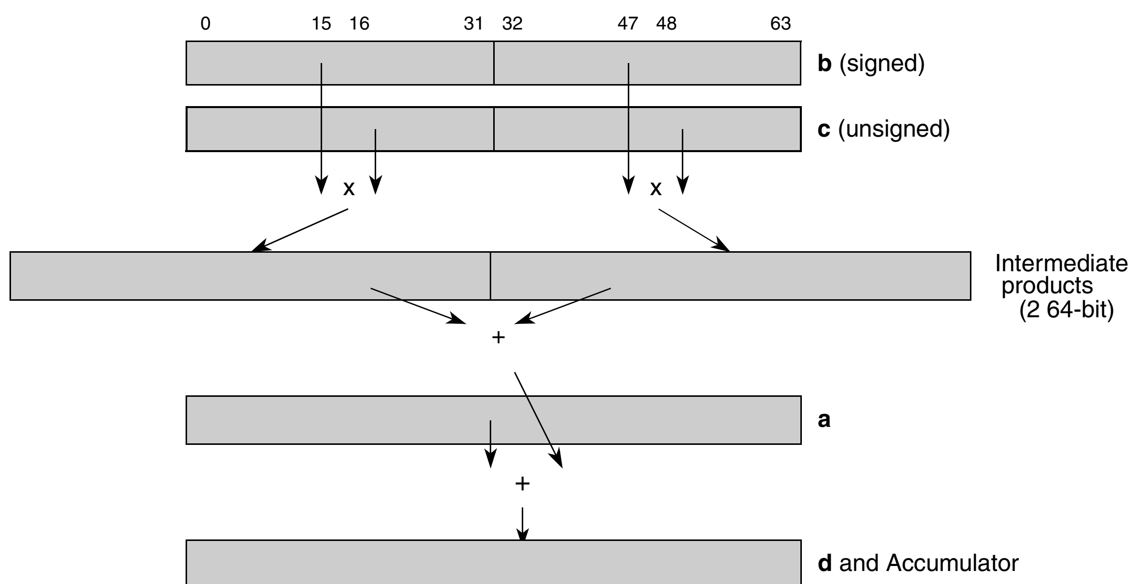


Figure 3-225. Vector Dot Product of Words, Add, Signed by Unsigned, Saturate, Integer and Accumulate 3 op (__ev_dotpwasusiaa3)

d	a	b	c	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**d ←a** **evdotpwasusiaa3 d,b,c**

__ev_dotpwaumi[a] __ev_dotpwaumi[a]

Vector Dot Product of Words, Add, Unsigned, Modulo, integer (to Accumulator)

d = __ev_dotpwaumi (a,b) (A = 0)

d = __ev_dotpwaumia (a,b) (A = 1)

$$temph_{0:63} \leftarrow a_{0:31} \times_{ui} b_{0:31}$$
$$templ_{0:63} \leftarrow a_{32:63} \times_{ui} b_{32:63}$$
$$temp_{0:31} \leftarrow temph_{0:63} + templ_{0:63} \text{ // modulo sum}$$
$$d_{0:63} \leftarrow temp_{0:63}$$

$$\text{// update accumulator}$$
$$\text{if } A = 1 \text{ then } ACC_{0:63} \leftarrow d_{0:63}$$

Corresponding pairs of unsigned integer word elements in parameters **a** and **b** are multiplied producing a pair of 64-bit products. This pair of intermediate products is added together and the sum 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)

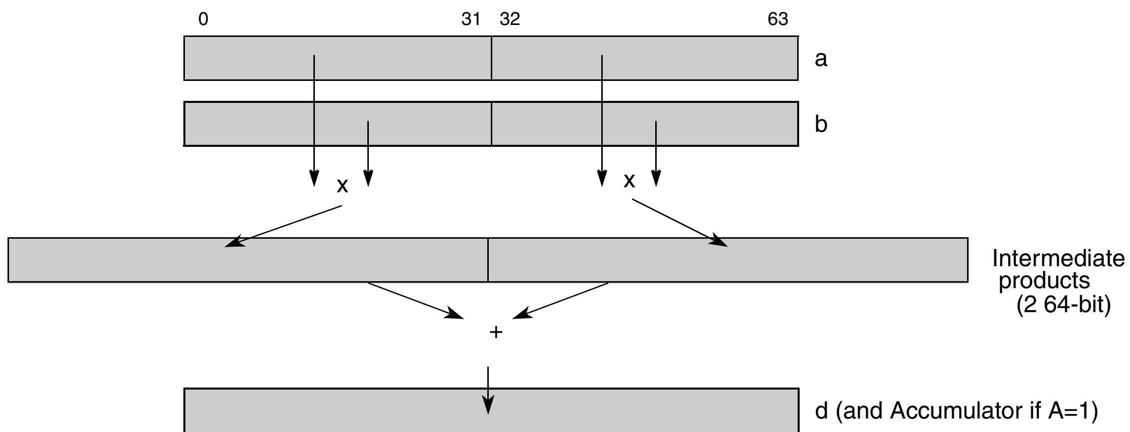


Figure 3-226. Vector Dot Product of Words, Add, Unsigned, Modulo, Integer (to Accumulator) (__ev_dotpwaumi[a])

A	d	a	b	Maps to
A = 0	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evdotpwaumi d,a,b**
A = 1	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evdotpwaumia d,a,b**

__ev_dotpwaumiaa __ev_dotpwaumiaa

Vector Dot Product of Words, Add, Unsigned, Modulo, Integer and Accumulate

d = __ev_dotpwaumiaa (a,b)

$$temph_{0:63} \leftarrow a_{0:31} \times_{ui} b_{0:31}$$
$$templ_{0:63} \leftarrow a_{32:63} \times_{ui} b_{32:63}$$
$$temp_{0:63} \leftarrow temph_{0:63} + templ_{0:63} + ACC_{0:63} \text{ // modulo sum}$$
$$d_{0:63} \leftarrow temp_{0:63}$$

```
// update accumulator
```
$$ACC_{0:63} \leftarrow d_{0:63}$$

Corresponding pairs of unsigned integer word elements in parameters **a** and **b** are multiplied producing a pair of 64-bit products. This pair of intermediate products is added together with the contents of the accumulator and the sum is placed into parameter **d** and the accumulator.

Other registers altered: ACC

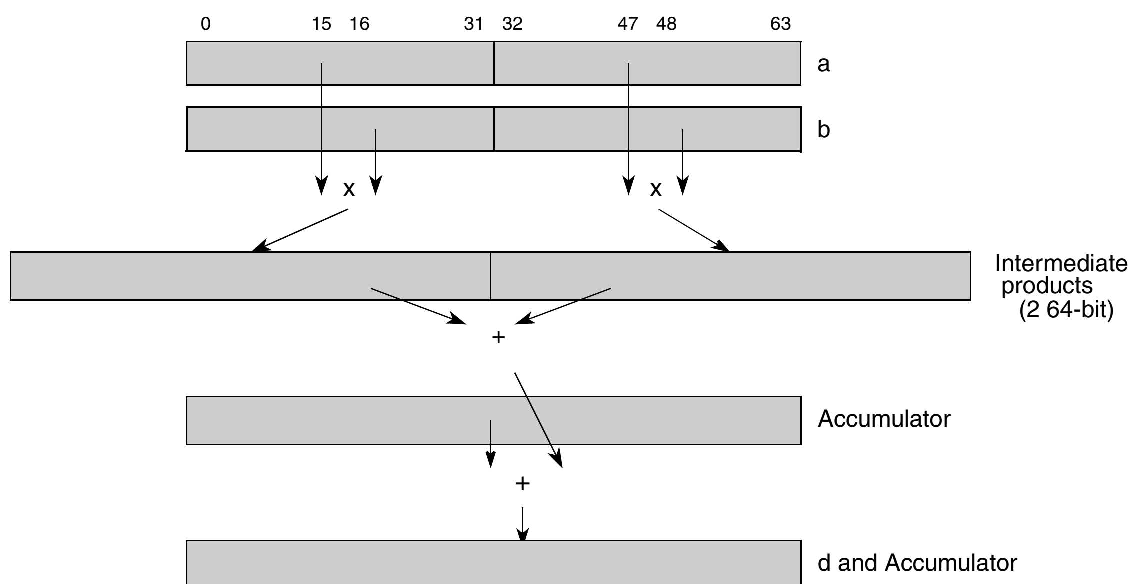


Figure 3-227. Vector Dot Product of Words, Add, Unsigned, Modulo, Integer and Accumulate (__ev_dotpwaumiaa)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evdotpwaumiaa d,a,b**

__ev_dotpwaumiaa3 __ev_dotpwaumiaa3

Vector Dot Product of Words, Add, Unsigned, Modulo, Integer and Accumulate, 3 operand

d = __ev_dotpwaumiaa3 (a,b,c)

```
temph0:63 ←b0:31 ×ui c0:31
templ0:63 ←b32:63 ×ui c32:63
temp0:63 ←temph0:63 + templ0:63 + a0:63 // modulo sum
d0:63 ←temp0:63

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

Corresponding pairs of unsigned integer word elements in parameters **b** and **c** are multiplied producing a pair of 64-bit products. This pair of intermediate products is added together with the contents of parameter **a** and the sum is placed into parameter **d** and the accumulator.

Other registers altered: ACC

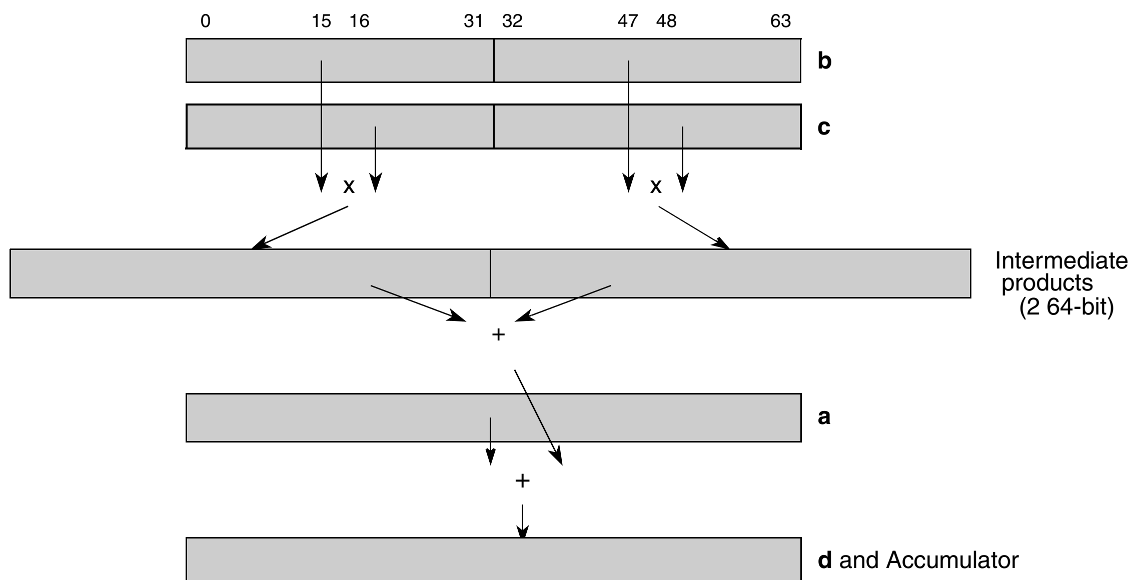


Figure 3-228. Vector Dot Product of Words, Add, Unsigned, Modulo, Integer and Accumulate 3 op (__ev_dotpwaumiaa3)

d	a	b	c	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**d ←a** **evdotpwaumiaa3 d,b,c**

__ev_dotpwausi[a] __ev_dotpwausi[a]

Vector Dot Product of Words, Add, Unsigned, Saturate, integer (to Accumulator)

d = __ev_dotpwausi (a,b) (A = 0)

d = __ev_dotpwausia (a,b) (A = 1)

```
temph0:63 ←a0:31 ×ui b0:31
templ0:63 ←a32:63 ×ui b32:63

temp0:64 ←EXTZ(temph0:63) + EXTZ(templ0:63)
ov ←temp0
d0:63 ←SATURATE(ov, 0, 0xFFFF_FFFF_FFFF_FFFF, 0xFFFF_FFFF_FFFF_FFFF, temp1:64)
// update accumulator
if A = 1 then ACC0:63 ←d0:63
// update SPEFSCR
SPEFSCROVH ←0; SPEFSCROV ←ov
SPEFSCRSOV ←SPEFSCRSOV | ov
```

Corresponding pairs of unsigned integer word elements in parameters **a** and **b** are multiplied producing a pair of 64-bit products. This pair of intermediate 64-bit products is added together with the contents of the accumulator, saturating if overflow occurs, and the sum is placed into parameter **d** and the accumulator.

The overflow and summary overflow bits are recorded in the SPEFSCR based on an overflow from the accumulation operations.

Other registers altered: SPEFSCR ACC

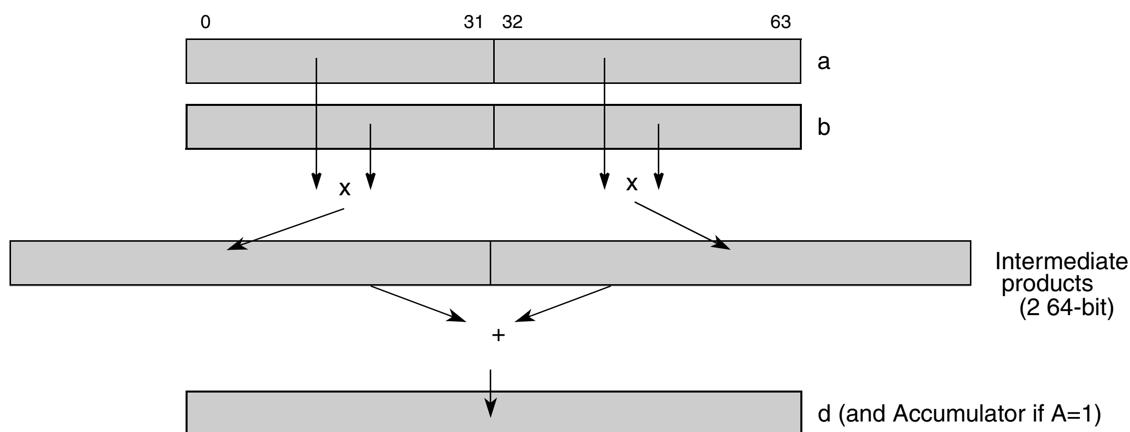


Figure 3-229. Vector Dot Product of Words, Add, Unsigned, Saturate, Integer (to Accumulator) (__ev_dotpwausi[a])

A	d	a	b	Maps to
A = 0	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evdotpwausi d,a,b**
A = 1	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evdotpwausia d,a,b**

__ev_dotpwausiaa __ev_dotpwausiaa

Vector Dot Product of Words, Add, Unsigned, Saturate, Integer and Accumulate

d = __ev_dotpwausiaa (a,b)

```
temph0:63 ← a0:31 ×ui b0:31
templ0:63 ← a32:63 ×ui b2:63

temp0:64 ← EXTZ(ACC0:63) + EXTZ(temph0:63) + EXTZ(templ0:63)
ov ← temp0
d0:63 ← SATURATE(ov, 0, 0xFFFF_FFFF_FFFF_FFFF, 0xFFFF_FFFF_FFFF_FFFF, temp1:64)
// update accumulator
ACC0:63 ←d0:63
// update SPEFSCR
SPEFSCROVH ← 0; SPEFSCROV ←ov
SPEFSCRSOV ← SPEFSCRSOV | ov
```

Corresponding pairs of unsigned integer word elements in parameters **a** and **b** are multiplied producing a pair of 64-bit products. This pair of intermediate 64-bit products is added together with the contents of the accumulator, saturating if overflow occurs in the final sum, and the sum is placed into parameter **d** and the accumulator.

The overflow and summary overflow bits are recorded in the SPEFSCR based on an overflow from the accumulation operations.

Other registers altered: SPEFSCR ACC

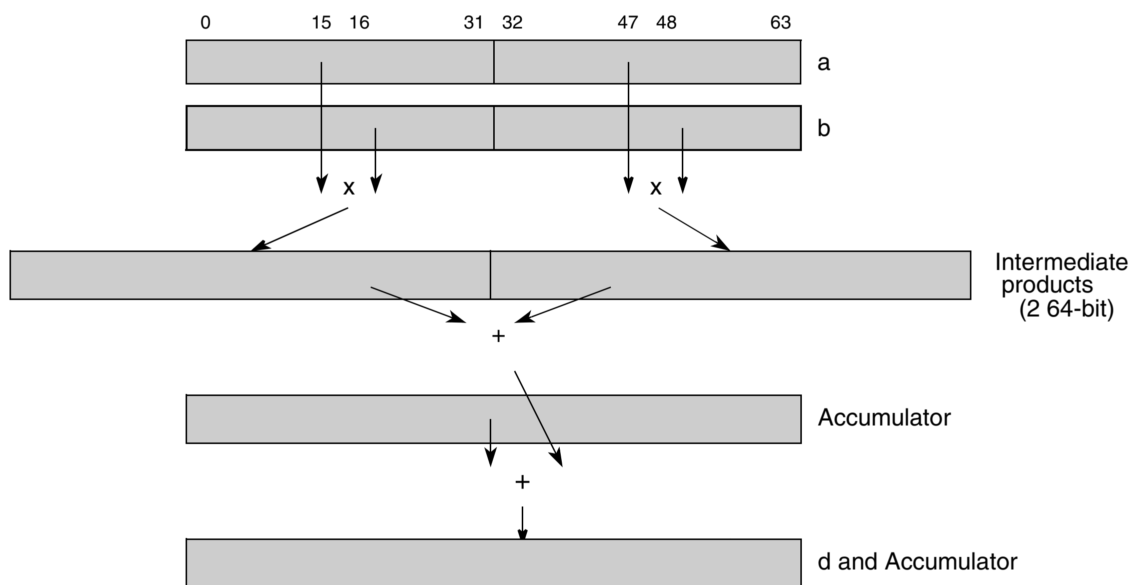


Figure 3-230. Vector Dot Product of Words, Add, Unsigned, Saturate, Integer and Accumulate (__ev_dotpwausiaa)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evdotpwausiaa d,a,b**

__ev_dotpwausiaa3 __ev_dotpwausiaa3

Vector Dot Product of Words, Add, Unsigned, Saturate, Integer and Accumulate, 3 operand

d = __ev_dotpwausiaa3 (a,b,c)

```
temph0:63 ←b0:31 ×ui c0:31
templ0:63 ←b32:63 ×ui c32:63

temp0:66 ←EXTZ67(a0:63) + EXTZ67(temph0:63) + EXTZ67(templ0:63)
ov ←temp0:2 != 30
d0:63 ←SATURATE(ov, 0, 0xFFFF_FFFF_FFFF_FFFF, 0xFFFF_FFFF_FFFF_FFFF, temp3:66)
// update accumulator
ACC0:63 ←d0:63
// update SPEFSCR
SPEFSCROVH ←0; SPEFSCROV ←ov
SPEFSCRSOV ←SPEFSCRSOV | ov
```

Corresponding pairs of unsigned integer word elements in parameters **b** and **c** are multiplied producing a pair of 64-bit products. This pair of intermediate 64-bit products is added together with the contents of parameter **a**, saturating if overflow occurs in the final sum, and the sum is placed into parameter **d** and the accumulator. The overflow and summary overflow bits are recorded in the SPEFSCR based on an overflow from the accumulation operations.

Other registers altered: SPEFSCR ACC

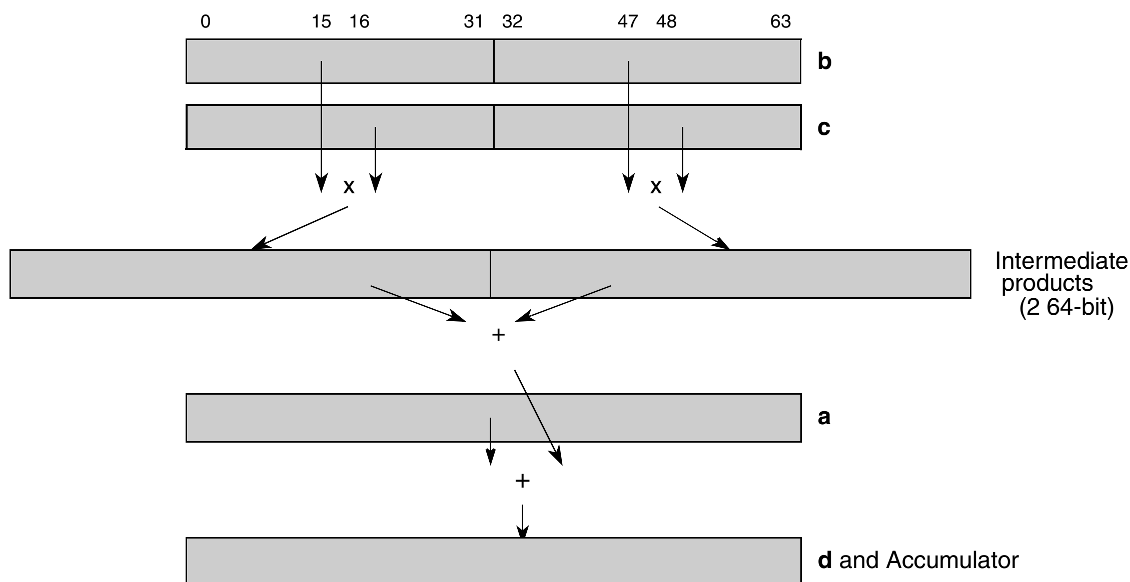


Figure 3-231. Vector Dot Product of Words, Add, Unsigned, Saturate, Integer and Accumulate 3 op (__ev_dotpwausiaa3)

d	a	b	c	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	d ←a **evdotpwausiaa3 d,b,c**

__ev_dotpwcsmi[a] __ev_dotpwcsmi[a]

Vector Dot Product of Words, Complex, Signed, Modulo, Integer (to Accumulator)

d = __ev_dotpwcsmi (a,b) (A = 0)

d = __ev_dotpwcsmia (a,b) (A = 1)

```
// high dot - calculate real part of complex product
temph1[0:63] ← a[0:31] ×si b[0:31]
temph2[0:63] ← a[32:63] ×si b[32:63]
temph[0:63] ← temph1[0:63] - temph2[0:63] // modulo difference
d[0:31] ← temph[32:63]

//low dot - calculate imaginary part of complex product
templ1[0:31] ← a[32:63] ×si b[0:31]
templ2[0:31] ← a[0:31] ×si b[32:63]
templ[0:63] ← templ1[0:63] + templ2[0:63] // modulo sum
d[32:63] ← templ[32:63]

// update accumulator
if A = 1 then ACC[0:63] ←d[0:63]
```

For the high word element in the destination, corresponding word pairs of signed integer elements in parameters **a** and **b** are multiplied producing a pair of 64-bit products. The low-order 32-bits of the difference of this pair of intermediate 64-bit products is placed into the high word of parameter **d**. For the low word element in the destination, word pairs of signed integer elements from the words of parameters **a** and **b** are multiplied after exchanging the words of parameter **a**, producing a pair of 64-bit products. The low-order 32-bits of the sum of this pair of intermediate 64-bit products is placed into the low word of parameter **d**. If A = 1, the result in parameter **d** is also placed into the accumulator. This instruction can be used to perform a complex multiply or dot product by placing {real,img} pairs into the word pairs of parameters **a** and **b**.

Other registers altered: ACC (if A=1)

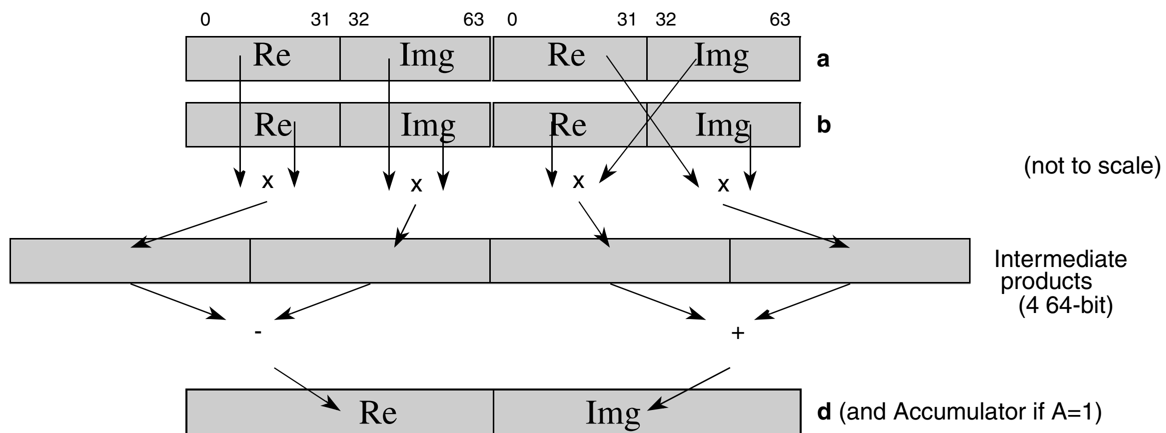


Figure 3-232. Vector Dot Product of Words, Complex, Signed, Modulo, Integer (to Accumulator) (__ev_dotpwcsmi[a])

A	d	a	b	Maps to
A = 0	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evdotpwcsmi d,a,b**
A = 1	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evdotpwcsmia d,a,b**

__ev_dotpwcsmiaaw __ev_dotpwcsmiaaw

Vector Dot Product of Words, Complex, Signed, Modulo, Integer and Accumulate into Words

d = __ev_dotpwcsmiaaw (a,b)

```
// high dot - calculate real part of complex product
temph1[0:63] ← a[0:31] ×si b[0:31]
temph2[0:63] ← a[32:63] ×si b[32:63]
temph[0:63] ← temph1[0:63] - temph2[0:63] // modulo difference
d[0:31] ← temph[32:63] + ACC[0:31]

//low dot - calculate imaginary part of complex product
templ1[0:31] ← a[32:63] ×si b[0:31]
templ2[0:31] ← a[0:31] ×si b[32:63]
templ[0:63] ← templ1[0:63] + templ2[0:63] // modulo sum
d[32:63] ← templ[32:63] + ACC[32:63]

// update accumulator
ACC[0:63] ←d[0:63]
```

For the high word element in the destination, corresponding word pairs of signed integer elements in parameters **a** and **b** are multiplied producing a pair of 64-bit products. The low-order 32-bits of the difference of this pair of intermediate 64-bit products is added together with the contents of the upper accumulator word and the sum is placed into the high words of parameter **d** and the accumulator. For the low word element in the destination, word pairs of signed integer elements from the words of parameters **a** and **b** are multiplied after exchanging the words of parameter **a**, producing a pair of 64-bit products. The low-order 32-bits of the sum of this pair of intermediate 64-bit products is added together with the contents of the low accumulator word and the sum is placed into the low words of parameter **d** and the accumulator. This instruction can be used to perform a complex multiply or dot product by placing {real,img} pairs into the word pairs of parameters **a** and **b**.

Other registers altered: ACC

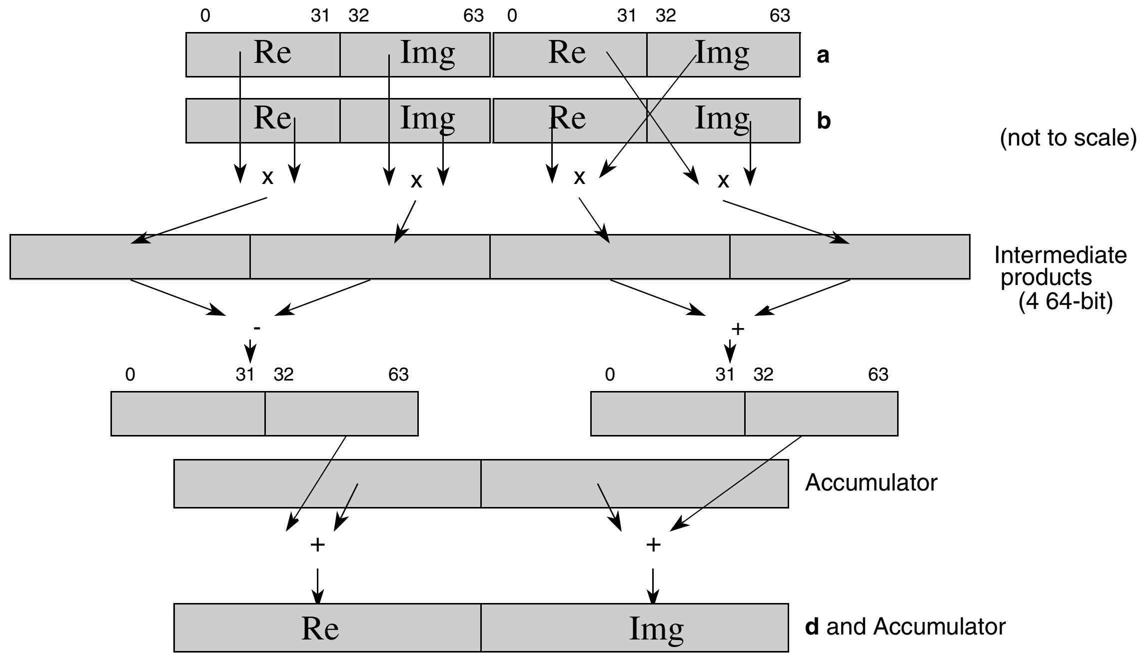


Figure 3-233. Vector Dot Product of Words, Complex, Signed, Modulo, Integer and Accumulate Words (__ev_dotpwcsmiaaw)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evdotpwcsmiaaw d,a,b**

__ev_dotpwcsmiaaw3 __ev_dotpwcsmiaaw3

Vector Dot Product of Words, Complex, Signed, Modulo, Integer and Accumulate into Words 3 operand

d = __ev_dotpwcsmiaaw3 (a,b,c)

```
// high dot - calculate real part of complex product
temph1[0:63] ←b[0:31] ×si c[0:31]
temph2[0:63] ←b[32:63] ×si c[32:63]
temph[0:63] ←temph1[0:63] - temph2[0:63] // modulo difference
d[0:31] ←temph[32:63] + a[0:31]

//low dot - calculate imaginary part of complex product
templ1[0:31] ←b[32:63] ×si c[0:31]
templ2[0:31] ←b[0:31] ×si c[32:63]
templ[0:63] ←templ1[0:63] + templ2[0:63] // modulo sum
d[32:63] ←templ[32:63] + a[32:63]

// update accumulator
ACC[0:63] ←d[0:63]
```

For the high word element in the destination, corresponding word pairs of signed integer elements in parameters **b** and **c** are multiplied producing a pair of 64-bit products. The low-order 32-bits of the difference of this pair of intermediate 64-bit products is added together with the contents of the upper parameter **a** word and the sum is placed into the high words of parameter **d** and the accumulator. For the low word element in the destination, word pairs of signed integer elements from the words of parameters **b** and **c** are multiplied after exchanging the words of parameter **b**, producing a pair of 64-bit products. The low-order 32-bits of the sum of this pair of intermediate 64-bit products is added together with the contents of the low parameter **a** word and the sum is placed into the low words of parameter **d** and the accumulator. This instruction can be used to perform a complex multiply or dot product by placing {real,img} pairs into the word pairs of parameters **b** and **c**.

Other registers altered: ACC

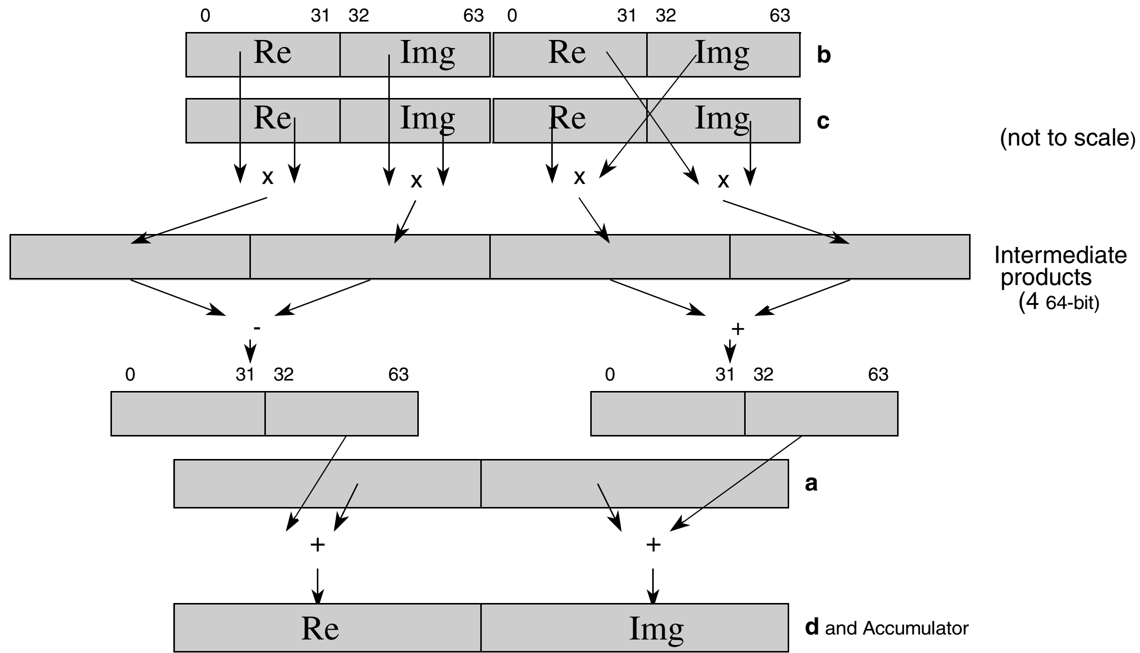


Figure 3-234. Vector Dot Product of Words, Complex, Signed, Modulo, Integer and Accumulate Words 3 op (__ev_dotpwcsmiaaw3)

d	a	b	c	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	d ←a evdotpwcsmiaaw3 d,b,c

__ev_dotpwcssf[a] __ev_dotpwcssf[a]

Vector Dot Product of Words, Complex, Signed, Saturate, Fractional (to Accumulator)

d = __ev_dotpwcssf (a,b) (A = 0)

d = __ev_dotpwcssfa (a,b) (A = 1)

```
// high dot - calculate real part of complex product
temph1[0:63] ← a[0:31] ×sf b[0:31]
if (a[0:31] = 0x8000_0000) & (b[0:31] = 0x8000_0000) then
    temph1[0:63] ←0x7FFF_FFFF_FFFF_FFFF //saturate
    movh ←1
else
    movh ←0
temph2[0:63] ← a[32:63] ×sf b[32:63]
if (a[32:63] = 0x8000_0000) & (b[32:63] = 0x8000_0000) then
    temph2[0:63] ←0x7FFF_FFFF_FFFF_FFFF //saturate
    movh ←1
temph[0:64] ← EXTS65(temph1[0:63]) - EXTS65(temph2[0:63])
ovh ← temph[0] ⊕ temph[1]
d[0:31] ← SATURATE(ovh, temph[0], 0x8000_0000, 0x7FFF_FFFF, temph[1:32])

//low dot - calculate imaginary part of complex product
templ1[0:63] ← a[32:63] ×sf b[0:31]
if (a[32:63] = 0x8000_0000) & (b[0:31] = 0x8000_0000) then
    templ1[0:63] ←0x7FFF_FFFF_FFFF_FFFF //saturate
    movl ←1
else
    movl ←0
templ2[0:63] ← a[0:31] ×sf b[32:63]
if (a[0:31] = 0x8000_0000) & (b[32:63] = 0x8000_0000) then
    templ2[0:63] ←0x7FFF_FFFF_FFFF_FFFF //saturate
    movl ←1
templ[0:64] ← EXTS65(templ1[0:63]) + EXTS65(templ2[0:63])
ovl ← templ[0] ⊕ templ[1]
d[32:63] ← SATURATE(ovl, templ[0], 0x8000_0000, 0x7FFF_FFFF, templ[1:32])

// update accumulator
if A = 1 then ACC[0:63] ←d[0:63]
// update SPEFSCR
SPEFSCR[OVH] ← movh | ovh; SPEFSCR[OV] ← movl | ovl
SPEFSCR[SOVH] ← SPEFSCR[SOVH] | movh | ovh; SPEFSCR[SOV] ← SPEFSCR[SOV] | movl | ovl
```

For the high word element in the destination, corresponding word pairs of signed fractional elements in parameters **a** and **b** are multiplied producing a pair of 64-bit products. The high-order 32-bit difference of this pair of intermediate 64-bit products is placed into the high word of parameter **d**, saturating if overflow or underflow occurs. For the low word element in the destination, word pairs of signed fractional elements from the words of parameters **a** and **b** are multiplied after exchanging the words of parameter **a**, producing a pair of 64-bit products. The high-order 32-bit sum of this pair of intermediate 64-bit products is placed into the low word of parameter **d**, saturating if overflow or underflow occurs. If both elements of a multiply are -1.0, the product is saturated. If A = 1, the result in parameter **d** is also placed into the accumulator. This instruction can be used to perform a complex multiply or dot product by placing {real,img} pairs into the word pairs of parameters **a** and **b**.

Other registers altered: ACC (if A=1)

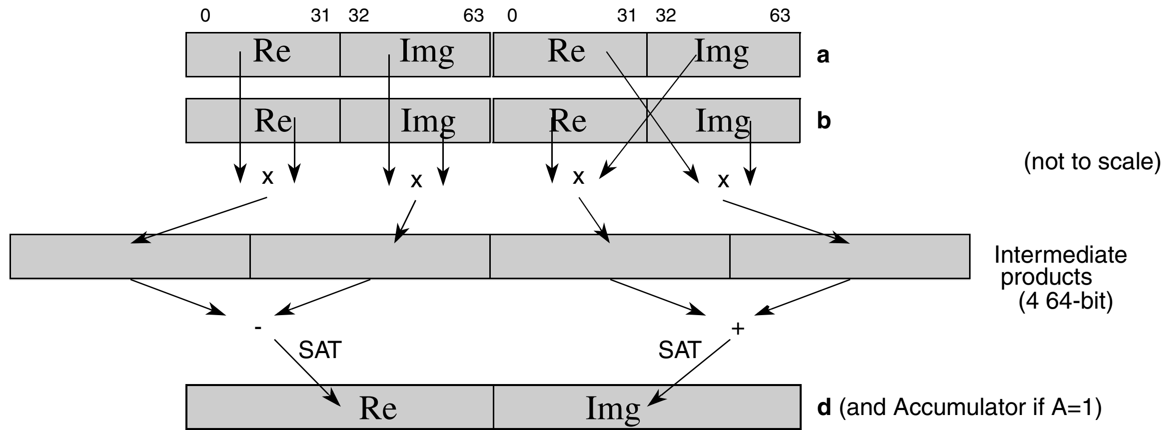


Figure 3-235. Vector Dot Product of Words, Complex, Signed, Saturate, Fractional (to Accumulator) (__ev_dotpwcssf[a])

A	d	a	b	Maps to
A = 0	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evdotpwcssf d,a,b**
A = 1	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evdotpwcssfa d,a,b**

__ev_dotpwcssfaaw __ev_dotpwcssfaaw

Vector Dot Product of Words, Complex, Signed, Saturate, Fractional and Accumulate into Words

d = __ev_dotpwcssfaaw (a,b)

```
// high dot - calculate real part of complex product
temph1[0:63] ← a[0:31] ×sf b[0:31]
if (a[0:31] = 0x8000_0000) & (b[0:31] = 0x8000_0000) then
    temph1[0:63] ←0x7FFF_FFFF_FFFF_FFFF //saturate
    movh ←1
else
    movh ←0
temph2[0:63] ← a[32:63] ×sf b[32:63]
if (a[32:63] = 0x8000_0000) & (b[32:63] = 0x8000_0000) then
    temph2[0:63] ←0x7FFF_FFFF_FFFF_FFFF //saturate
    movh ←1
temph[0:66] ← EXTS67(ACC[0:31] || 320) + EXTS67(temph1[0:63]) - EXTS67(temph2[0:63])
ovh ← chk_ovf(temph[0:3])
d[0:31] ← SATURATE(ovh, temph[0], 0x8000_0000, 0x7FFF_FFFF, temph[3:34])

//low dot - calculate imaginary part of complex product
templ1[0:63] ← a[32:63] ×sf b[0:31]
if (a[32:63] = 0x8000_0000) & (b[0:31] = 0x8000_0000) then
    templ1[0:63] ←0x7FFF_FFFF_FFFF_FFFF //saturate
    movl ←1
else
    movl ←0
templ2[0:63] ← a[0:31] ×sf b[32:63]
if (a[0:31] = 0x8000_0000) & (b[32:63] = 0x8000_0000) then
    templ2[0:63] ←0x7FFF_FFFF_FFFF_FFFF //saturate
    movl ←1
templ[0:66] ← EXTS67(ACC[32:63] || 320) + EXTS67(templ1[0:63]) + EXTS67(templ2[0:63])
ovl ← chk_ovf(templ[0:3])
d[32:63] ← SATURATE(ovl, templ[0], 0x8000_0000, 0x7FFF_FFFF, templ[3:34])

// update accumulator
ACC[0:63] ←d[0:63]
// update SPEFSCR
SPEFSCR[OVH] ← movh | ovh; SPEFSCR[OV] ← movl | ovl
SPEFSCR[SOVH] ← SPEFSCR[SOVH] | movh | ovh; SPEFSCR[SOV] ← SPEFSCR[SOV] | movl | ovl
```

For the high word element in the destination, corresponding word pairs of signed fractional elements in parameters **a** and **b** are multiplied producing a pair of 64-bit products. The high-order 32-bit difference of this pair of intermediate 64-bit products added to the upper word of the accumulator, saturating if overflow or underflow occurs. The 32-bit sum is placed into the high word of parameter **d** and the accumulator. For the low word element in the destination, word pairs of signed fractional elements from the words of parameters **a** and **b** are multiplied after exchanging the words of parameter **a**, producing a pair of 64-bit products. The high-order 32-bit sum of this pair of intermediate 64-bit products is added to the lower word of the accumulator, saturating if overflow or underflow occurs. The 32-bit sum is placed into the low word of parameter **d** and the accumulator. If both elements of a multiply are -1.0, the product is saturated. This instruction can be used to perform a complex multiply or dot product by placing {real,img} pairs into the word pairs of parameters **a** and **b**.

Other registers altered: ACC

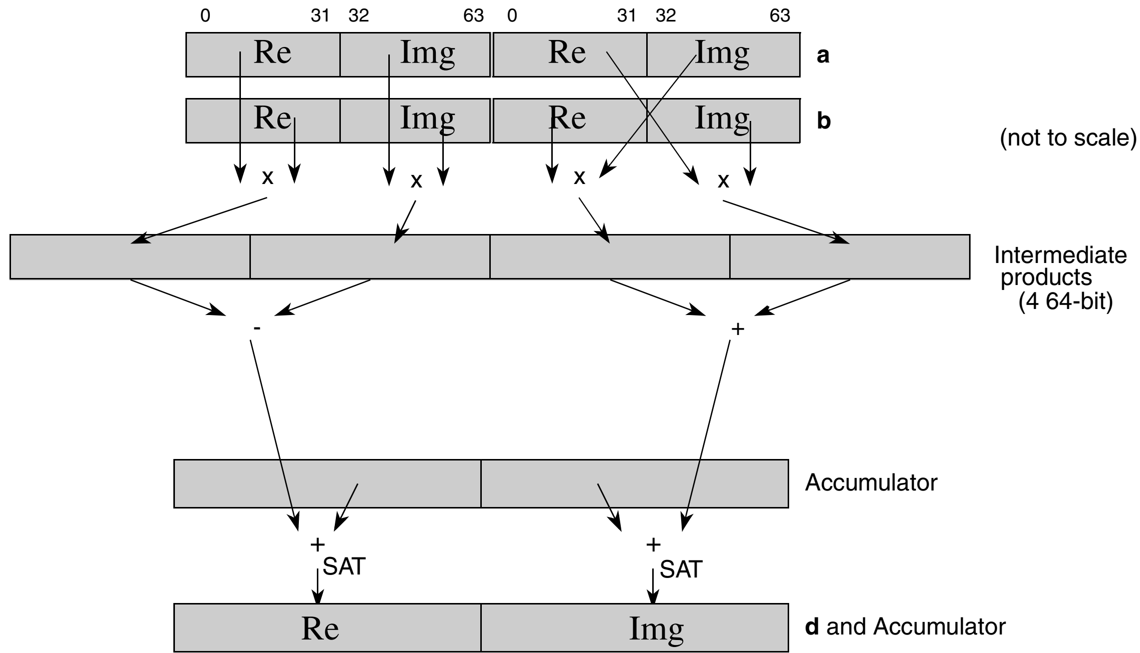


Figure 3-236. Vector Dot Product of Words, Complex, Signed, Saturate, Fractional and Accumulate Words (__ev_dotpwcssfaaw)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evdotpwcssfaaw d,a,b**

__ev_dotpwcssfaaw3 __ev_dotpwcssfaaw3

Vector Dot Product of Words, Complex, Signed, Saturate, Fractional and Accumulate into Words 3 operand

d = __ev_dotpwcssfaaw3 (a,b,c)

```
// high dot - calculate real part of complex product
temph1[0:63] ← b[0:31] ×sf c[0:31]
if (b[0:31] = 0x8000_0000) & (c[0:31] = 0x8000_0000) then
    temph1[0:63] ←0x7FFF_FFFF_FFFF_FFFF //saturate
    movh ←1
else
    movh ←0
temph2[0:63] ← b[32:63] ×sf c[32:63]
if (b[32:63] = 0x8000_0000) & (c[32:63] = 0x8000_0000) then
    temph2[0:63] ←0x7FFF_FFFF_FFFF_FFFF //saturate
    movh ←1
temph[0:66] ← EXTS67(a[0:31] || 32 0) + EXTS67(temph1[0:63]) - EXTS67(temph2[0:63])
ovh ← chk_ovf(temph[0:3])
d[0:31] ← SATURATE(ovh, temph[0], 0x8000_0000, 0x7FFF_FFFF, temph[3:34])

//low dot - calculate imaginary part of complex product
templ1[0:63] ← b[32:63] ×sf c[0:31]
if (b[32:63] = 0x8000_0000) & (c[0:31] = 0x8000_0000) then
    templ1[0:63] ←0x7FFF_FFFF_FFFF_FFFF //saturate
    movl ←1
else
    movl ←0
templ2[0:63] ← b[0:31] ×sf c[32:63]
if (b[0:31] = 0x8000_0000) & (c[32:63] = 0x8000_0000) then
    templ2[0:63] ←0x7FFF_FFFF_FFFF_FFFF //saturate
    movl ←1
templ[0:66] ← EXTS67(a[32:63] || 32 0) + EXTS67(templ1[0:63]) + EXTS67(templ2[0:63])
ovl ← chk_ovf(templ[0:3])
d[32:63] ← SATURATE(ovl, templ[0], 0x8000_0000, 0x7FFF_FFFF, templ[3:34])

// update accumulator
ACC[0:63] ←d[0:63]
// update SPEFSCR
SPEFSCR[OVH] ← movh | ovh; SPEFSCR[OV] ← movl | ovl
SPEFSCR[SOVH] ← SPEFSCR[SOVH] | movh | ovh; SPEFSCR[SOV] ← SPEFSCR[SOV] | movl | ovl
```

For the high word element in the destination, corresponding word pairs of signed fractional elements in parameters **b** and **c** are multiplied producing a pair of 64-bit products. The high-order 32-bit difference of this pair of intermediate 64-bit products added to the upper word of parameter **a**, saturating if overflow or underflow occurs. The 32-bit sum is placed into the high word of parameter **d** and the accumulator. For the low word element in the destination, word pairs of signed fractional elements from the words of parameters **b** and **c** are multiplied after exchanging the words of parameter **b**, producing a pair of 64-bit products. The high-order 32-bit sum of this pair of intermediate 64-bit products is added to the lower word of parameter **a**, saturating if overflow or underflow occurs. The 32-bit sum is placed into the low word of parameter **d** and the accumulator. If both elements of a multiply are -1.0, the product is saturated. This instruction can be used to perform a complex multiply or dot product by placing {real,img} pairs into the word pairs of parameters **b** and **c**.

Other registers altered: ACC

Figure 3-237. Vector Dot Product of Words, Complex, Signed, Saturate, Fractional and Accumulate Words 3 op (__ev_dotpwcssfaaw3)

d	a	b	c	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	d ←a **evdotpwcssfaaw3 d,b,c**

__ev_dotpwcssfr[a] __ev_dotpwcssfr[a]

Vector Dot Product of Words, Complex, Signed, Saturate, Fractional, Round (to Accumulator)

d = __ev_dotpwcssfr (a,b) (A = 0)

d = __ev_dotpwcssfra (a,b) (A = 1)

```
// high dot - calculate real part of complex product
temph1[0:63] ← a[0:31] ×sf b[0:31]
if (a[0:31] = 0x8000_0000) & (b[0:31] = 0x8000_0000) then
    temph1[0:63] ←0x7FFF_FFFF_FFFF_FFFF //saturate
    movh ←1
else
    movh ←0
temph2[0:63] ← a[32:63] ×sf b[32:63]
if (a[32:63] = 0x8000_0000) & (b[32:63] = 0x8000_0000) then
    temph2[0:63] ←0x7FFF_FFFF_FFFF_FFFF //saturate
    movh ←1
temph[0:66] ← ROUND((EXTS67(temph1[0:63]) - EXTS67(temph2[0:63])), 32)
ovh ← chk_ovf(temph[0:3])
d[0:31] ← SATURATE(ovh, temph[0], 0x8000_0000, 0x7FFF_FFFF, temph[3:34])

//low dot - calculate imaginary part of complex product
templ1[0:63] ← a[32:63] ×sf b[0:31]
if (a[32:63] = 0x8000_0000) & (b[0:31] = 0x8000_0000) then
    templ1[0:63] ←0x7FFF_FFFF_FFFF_FFFF //saturate
    movl ←1
else
    movl ←0
templ2[0:63] ← a[0:31] ×sf b[32:63]
if (a[0:31] = 0x8000_0000) & (b[32:63] = 0x8000_0000) then
    templ2[0:63] ←0x7FFF_FFFF_FFFF_FFFF //saturate
    movl ←1
templ[0:66] ← ROUND((EXTS67(templ1[0:63]) + EXTS67(templ2[0:63])),32)
ovl ← chk_ovf(templ[0:3])
d[32:63] ← SATURATE(ovl, templ[0], 0x8000_0000, 0x7FFF_FFFF, templ[3:34])

// update accumulator
if A = 1 then ACC[0:63] ←d[0:63]
// update SPEFSCR
SPEFSCR[OVH] ← movh | ovh; SPEFSCR[OV] ← movl | ovl
SPEFSCR[SOVH] ← SPEFSCR[SOVH] | movh | ovh; SPEFSCR[SOV] ← SPEFSCR[SOV] | movl | ovl
```

For the high word element in the destination, corresponding word pairs of signed fractional elements in parameters **a** and **b** are multiplied producing a pair of 64-bit products. The difference of this pair of intermediate products is rounded to 32 bits using the current fractional rounding mode in SPEFCSR, saturating if overflow or underflow occurs, and placed into the high word of parameter **d**. For the low word element in the destination, word pairs of signed fractional elements from the words of parameters **a** and **b** are multiplied after exchanging the words of parameter **a**, producing a pair of 64-bit products. The sum of this pair of intermediate products is rounded to 32 bits using the current fractional rounding mode in SPEFCSR, saturating if overflow or underflow occurs, and placed into the high word of parameter **d**. If both elements of a multiply are -1.0, the product is saturated. If A = 1, the result in parameter **d** is also placed into the accumulator. This instruction can be used to perform a complex multiply or dot product by placing {real,img} pairs into the word pairs of parameters **a** and **b**.

Other registers altered: ACC (if A=1)

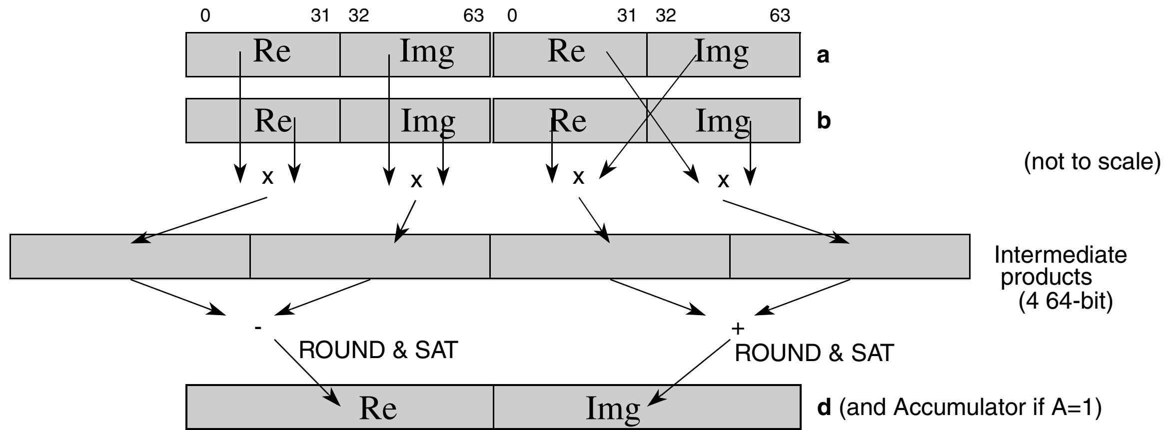


Figure 3-238. Vector Dot Product of Words, Complex, Signed, Saturate, Fractional, Round (to Accumulator) (__ev_dotpwcssfr[a])

A	d	a	b	Maps to
A = 0	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evdotpwcssfr d,a,b**
A = 1	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evdotpwcssfra d,a,b**

__ev_dotpwcssfraaw __ev_dotpwcssfraaw

Vector Dot Product of Words, Complex, Signed, Saturate, Fractional, Round and Accumulate into Words

d = __ev_dotpwcssfraaw (a,b)

```
// high dot - calculate real part of complex product
temph1_{0:63} ← a_{0:31} ×_{sf} b_{0:31}
if (a_{0:31} = 0x8000_0000) & (b_{0:31} = 0x8000_0000) then
    temph1_{0:63} ←0x7FFF_FFFF_FFFF_FFFF //saturate
    movh ←1
else
    movh ←0
temph2_{0:63} ← a_{32:63} ×_{sf} b_{32:63}
if (a_{32:63} = 0x8000_0000) & (b_{32:63} = 0x8000_0000) then
    temph2_{0:63} ←0x7FFF_FFFF_FFFF_FFFF //saturate
    movh ←1
temph_{0:66} ← ROUND((EXTS_{67}(ACC_{0:31} || ^{32}0) + EXTS_{67}(temph1_{0:63}) - EXTS_{67}(temph2_{0:63})),32)
ovl ← chk_ovf(temph_{0:3})
d_{0:31} ← SATURATE(ovh, temph_{0}, 0x8000_0000, 0x7FFF_FFFF, temph_{3:34})

//low dot - calculate imaginary part of complex product
templ1_{0:63} ← a_{32:63} ×_{sf} b_{0:31}
if (a_{32:63} = 0x8000_0000) & (b_{0:31} = 0x8000_0000) then
    templ1_{0:63} ←0x7FFF_FFFF_FFFF_FFFF //saturate
    movl ←1
else
    movl ←0
templ2_{0:63} ← a_{0:31} ×_{sf} b_{32:63}
if (a_{0:31} = 0x8000_0000) & (b_{32:63} = 0x8000_0000) then
    templ2_{0:63} ←0x7FFF_FFFF_FFFF_FFFF //saturate
    movl ←1
templ_{0:66} ← ROUND((EXTS_{67}(ACC_{32:63} || ^{32}0) + EXTS_{67}(templ1_{0:63}) + EXTS_{67}(templ2_{0:63})),32)
ovl ← chk_ovf(templ_{0:3})
d_{32:63} ← SATURATE(ovl, templ_{0}, 0x8000_0000, 0x7FFF_FFFF, templ_{3:34})

// update accumulator
ACC_{0:63} ←d_{0:63}
// update SPEFSCR
SPEFSCR_{OVH} ← movh | ovh; SPEFSCR_{OV} ← movl | ovl
SPEFSCR_{SOVH} ← SPEFSCR_{SOVH} | movh | ovh; SPEFSCR_{SOV} ← SPEFSCR_{SOV} | movl | ovl
```

For the high word element in the destination, corresponding word pairs of signed fractional elements in parameters **a** and **b** are multiplied producing a pair of 64-bit products. The difference of this pair of intermediate 64-bit products is added to the zero-padded upper word of the accumulator, rounded to 32 bits using the current fractional rounding mode in SPEFCSR, saturating if overflow or underflow occurs, and placed into the high word of parameter **d** and the accumulator. For the low word element in the destination, word pairs of signed fractional elements from the words of parameters **a** and **b** are multiplied after exchanging the words of parameter **a**, producing a pair of 64-bit products. The sum of this pair of intermediate 64-bit products is added to the zero-padded lower word of the accumulator, rounded to 32 bits using the current fractional rounding mode in SPEFCSR, saturating if overflow or underflow occurs, and placed into the low word of parameter **d** and the accumulator. If both elements of a multiply are -1.0, the product is saturated. This instruction can be used to perform a complex multiply or dot product by placing {real,img} pairs into the word pairs of parameters **a** and **b**.

Other registers altered: ACC

Figure 3-239. Vector Dot Product of Words, Complex, Signed, Saturate, Fractional, Round and Accumulate Words (__ev_dotpwcssfraaw)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evdotpwcssfraaw d,a,b**

__ev_dotpwcssfraaw3 __ev_dotpwcssfraaw3

Vector Dot Product of Words, Complex, Signed, Saturate, Fractional, Round and Accumulate into Words 3 operand

d = __ev_dotpwcssfraaw3 (a,b,c)

```
// high dot - calculate real part of complex product
temph1_0:63 ← b_0:31 ×sf c_0:31
if (b_0:31 = 0x8000_0000) & (c_0:31 = 0x8000_0000) then
    temph1_0:63 ←0x7FFF_FFFF_FFFF_FFFF //saturate
    movh ←1
else
    movh ←0
temph2_0:63 ← b_32:63 ×sf c_32:63
if (b_32:63 = 0x8000_0000) & (c_32:63 = 0x8000_0000) then
    temph2_0:63 ←0x7FFF_FFFF_FFFF_FFFF //saturate
    movh ←1
temph_0:66 ← ROUND((EXTS_67(a_0:31 || 32 0) + EXTS_67(temph1_0:63) - EXTS_67(temph2_0:63)),32)
ovl ← chk_ovf(temph_0:3)
d_0:31 ← SATURATE(ovh, temph_0, 0x8000_0000, 0x7FFF_FFFF, temph_3:34)

//low dot - calculate imaginary part of complex product
templ1_0:63 ← b_32:63 ×sf a_0:31
if (b_32:63 = 0x8000_0000) & (c_0:31 = 0x8000_0000) then
    templ1_0:63 ←0x7FFF_FFFF_FFFF_FFFF //saturate
    movl ←1
else
    movl ←0
templ2_0:63 ← b_0:31 ×sf c_32:63
if (b_0:31 = 0x8000_0000) & (c_32:63 = 0x8000_0000) then
    templ2_0:63 ←0x7FFF_FFFF_FFFF_FFFF //saturate
    movl ←1
templ_0:66 ← ROUND((EXTS_67(a_32:63 || 32 0) + EXTS_67(templ1_0:63) + EXTS_67(templ2_0:63)),32)
ovl ← chk_ovf(templ_0:3)
d_32:63 ← SATURATE(ovl, templ_0, 0x8000_0000, 0x7FFF_FFFF, templ_3:34)

// update accumulator
ACC_0:63 ←d_0:63
// update SPEFSCR
SPEFSCR_OVH ← movh | ovh; SPEFSCR_OV ← movl | ovl
SPEFSCR_SOVH ← SPEFSCR_SOVH | movh | ovh; SPEFSCR_SOV ← SPEFSCR_SOV | movl | ovl
```

For the high word element in the destination, corresponding word pairs of signed fractional elements in parameters **b** and **c** are multiplied producing a pair of 64-bit products. The difference of this pair of intermediate 64-bit products is added to the zero-padded upper word of parameter **a**, rounded to 32 bits using the current fractional rounding mode in SPEFCSR, saturating if overflow or underflow occurs, and placed into the high word of parameter **d** and the accumulator. For the low word element in the destination, word pairs of signed fractional elements from the words of parameters **b** and **c** are multiplied after exchanging the words of parameter **b**, producing a pair of 64-bit products. The sum of this pair of intermediate 64-bit products is added to the zero-padded lower word of the parameter **a**, rounded to 32 bits using the current fractional rounding mode in SPEFCSR, saturating if overflow or underflow occurs, and placed into the low word of parameter **d** and the accumulator. If both elements of a multiply are -1.0, the product is saturated. This instruction can be used to perform a complex multiply or dot product by placing {real,img} pairs into the word pairs of parameters **b** and **c**.

Other registers altered: ACC

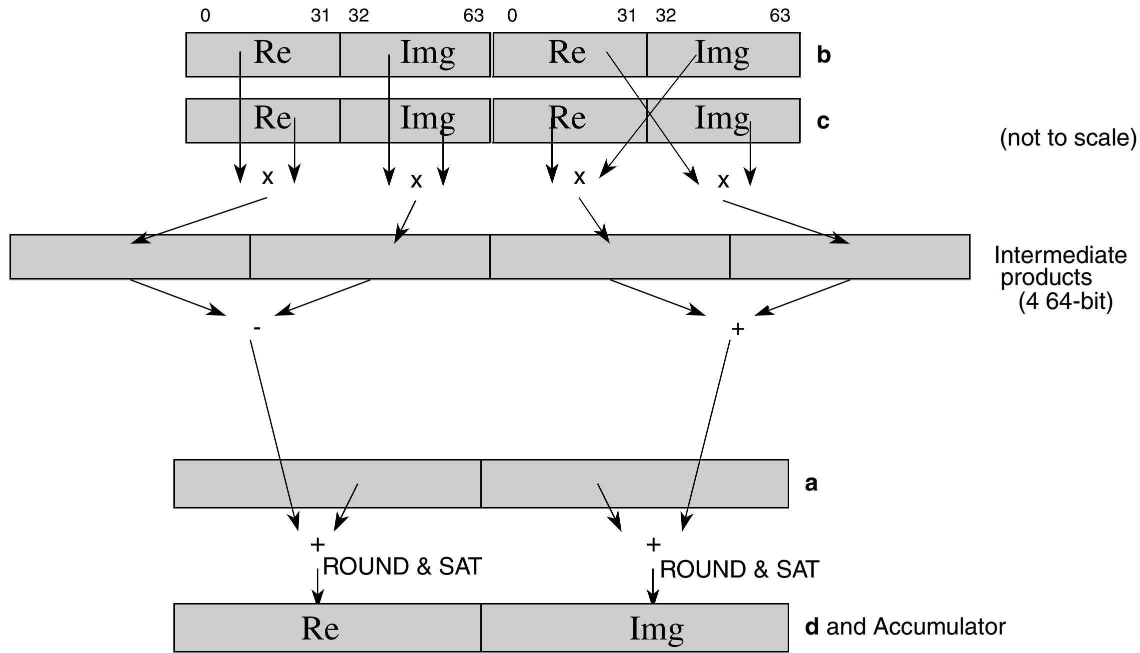


Figure 3-240. Vector Dot Product of Words, Complex, Signed, Saturate, Fractional, Round and Accumulate Words 3 op (__ev_dotpwcssfraaw3)

d	a	b	c	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	d ←a evdotpwcssfraaw3 d,b,c

__ev_dotpwcssi[a] __ev_dotpwcssi[a]

Vector Dot Product of Words, Complex, Signed, Saturate, Integer (to Accumulator)

d = __ev_dotpwcssi (a,b) (A = 0)

d = __ev_dotpwcssia (a,b) (A = 1)

```
// high dot - calculate real part of complex product
temph1_0:63 ← a_0:31 ×si b_0:31
temph2_0:63 ← a_32:63 ×si b_32:63
temph_0:64 ← EXTS_65(temph1_0:63) - EXTS_65(temph2_0:63)
if (temph_0:64 >si EXTS_65(0x7FFF_FFFF)) | (temph_0:64 <si EXTS_65(0x8000_0000)) then
    ovh ←1
else
    ovh ←0
d_0:31 ←SATURATE(ovh, temph_0, 0x8000_0000, 0x7FFF_FFFF, temph_33:64)

//low dot - calculate imaginary part of complex product
templ1_0:31 ← a_32:63 ×si b_0:31
templ2_0:31 ← a_0:31 ×si b_32:63
templ_0:64 ← EXTS_65(templ1_0:63) + EXTS_65(templ2_0:63)
if (templ_0:64 >si EXTS_65(0x7FFF_FFFF)) | (templ_0:64 <si EXTS_65(0x8000_0000)) then
    ovl ←1
else
    ovl ←0
d_32:63 ←SATURATE(ovl, templ_0, 0x8000_0000, 0x7FFF_FFFF, templ_33:64)

// update accumulator
if A = 1 then 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 the high word element in the destination, corresponding word pairs of signed integer elements in parameters **a** and **b** are multiplied producing a pair of 64-bit products. The difference of this pair of intermediate 64-bit products is placed into the high word of parameter **d**, saturating if the difference cannot be represented in 32 bits. For the low word element in the destination, word pairs of signed integer elements from the words of parameters **a** and **b** are multiplied after exchanging the words of parameter **a**, producing a pair of 64-bit products. The low-order 32-bits of the sum of this pair of intermediate 64-bit products is placed into the low word of parameter **d**, saturating if the sum cannot be represented in 32 bits. If A = 1, the result in parameter **d** is also placed into the accumulator. This instruction can be used to perform a complex multiply or dot product by placing {real,img} pairs into the word pairs of parameters **a** and **b**.

Other registers altered: ACC (if A=1)

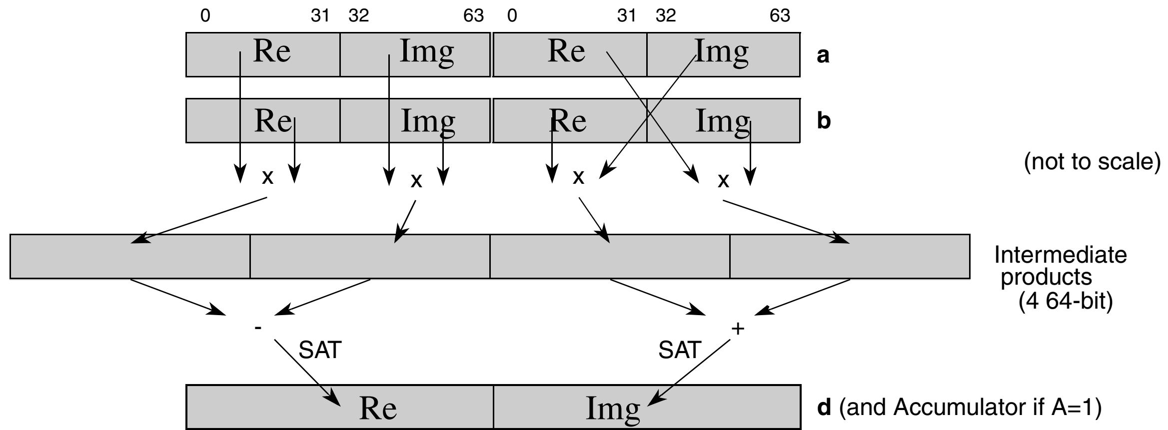


Figure 3-241. Vector Dot Product of Words, Complex, Signed, Saturate, Integer (to Accumulator) (__ev_dotpwcssi[a])

A	d	a	b	Maps to
A = 0	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evdotpwcssi d,a,b**
A = 1	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evdotpwcssia d,a,b**

__ev_dotpwcssiaaw __ev_dotpwcssiaaw

Vector Dot Product of Words, Complex, Signed, Saturate, Integer and Accumulate into Words

d = __ev_dotpwcssiaaw (a,b)

```
// high dot - calculate real part of complex product
temph1[0:63] ← a[0:31] ×si b[0:31]
temph2[0:63] ← a[32:63] ×si b[32:63]
temph[0:66] ← EXTS67(ACC[0:31]) + EXTS67(temph1[0:63]) - EXTS67(temph2[0:63])

if (temph[0:66] >si EXTS67(0x7FFF_FFFF)) | (temph[0:66] <si EXTS67(0x8000_0000)) then
    ovh ←1
else
    ovh ←0
d[0:31] ←SATURATE(ovh, temph[0], 0x8000_0000, 0x7FFF_FFFF, temph[35:66])

//low dot - calculate imaginary part of complex product
templ1[0:31] ← a[32:63] ×si b[0:31]
templ2[0:31] ← a[0:31] ×si b[32:63]
templ[0:66] ← EXTS67(ACC[32:63]) + EXTS67(templ1[0:63]) + EXTS67(templ2[0:63])

if (templ[0:66] >si EXTS67(0x7FFF_FFFF)) | (templ[0:66] <si EXTS67(0x8000_0000)) then
    ovl ←1
else
    ovl ←0
d[32:63] ←SATURATE(ovl, templ[0], 0x8000_0000, 0x7FFF_FFFF, templ[35:66])


// 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 the high word element in the destination, corresponding word pairs of signed integer elements in parameters **a** and **b** are multiplied producing a pair of 64-bit products. The difference of this pair of intermediate 64-bit products is added together with the contents of the sign-extended upper accumulator word, and the sum is placed into the high word of parameter **d** and the accumulator, saturating if the sum cannot be represented in 32 bits. For the low word element in the destination, word pairs of signed integer elements from the words of parameters **a** and **b** are multiplied after exchanging the words of parameter **a**, producing a pair of 64-bit products. The sum of this pair of intermediate 64-bit products is added together with the contents of the sign-extended lower accumulator word and the sum is placed into the low word of parameter **d** and the accumulator, saturating if the sum cannot be represented in 32 bits. This instruction can be used to perform a complex multiply or dot product by placing {real,img} pairs into the word pairs of parameters **a** and **b**.

Other registers altered: ACC

Figure 3-242. Vector Dot Product of Words, Complex, Signed, Saturate, Integer and Accumulate Words (__ev_dotpwcssiaaw)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evdotpwcssiaaw d,a,b**

__ev_dotpwcssiaaw3 __ev_dotpwcssiaaw3

Vector Dot Product of Words, Complex, Signed, Saturate, Integer and Accumulate into Words 3 operand

d = __ev_dotpwcssiaaw3 (a,b,c)

```
// high dot - calculate real part of complex product
temph1[0:63] ← b[0:31] ×si c[0:31]
temph2[0:63] ← b[32:63] ×si c[32:63]
temph[0:66] ← EXTS67(a[0:31]) + EXTS67(temph1[0:63]) - EXTS67(temph2[0:63])

if (temph[0:66] >si EXTS67(0x7FFF_FFFF)) | (temph[0:66] <si EXTS67(0x8000_0000)) then
    ovh ←1
else
    ovh ←0
d[0:31] ←SATURATE(ovh, temph[0], 0x8000_0000, 0x7FFF_FFFF, temph[35:66])

//low dot - calculate imaginary part of complex product
templ1[0:31] ← b[32:63] ×si c[0:31]
templ2[0:31] ← b[0:31] ×si c[32:63]
templ[0:66] ← EXTS67(a[32:63]) + EXTS67(templ1[0:63]) + EXTS67(templ2[0:63])

if (templ[0:66] >si EXTS67(0x7FFF_FFFF)) | (templ[0:66] <si EXTS67(0x8000_0000)) then
    ovl ←1
else
    ovl ←0
d[32:63] ←SATURATE(ovl, templ[0], 0x8000_0000, 0x7FFF_FFFF, templ[35:66])

// 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 the high word element in the destination, corresponding word pairs of signed integer elements in parameters **b** and **c** are multiplied producing a pair of 64-bit products. The difference of this pair of intermediate 64-bit products is added together with the contents of the sign-extended upper parameter **a** word, and the sum is placed into the high word of parameter **d** and the accumulator, saturating if the sum cannot be represented in 32 bits. For the low word element in the destination, word pairs of signed integer elements from the words of parameters **b** and **c** are multiplied after exchanging the words of parameter **b**, producing a pair of 64-bit products. The sum of this pair of intermediate 64-bit products is added together with the contents of the sign-extended low parameter **a** word and the sum is placed into the low word of parameter **d** and the accumulator, saturating if the sum cannot be represented in 32 bits. This instruction can be used to perform a complex multiply or dot product by placing {real,img} pairs into the word pairs of parameters **b** and **c**.

Other registers altered: ACC

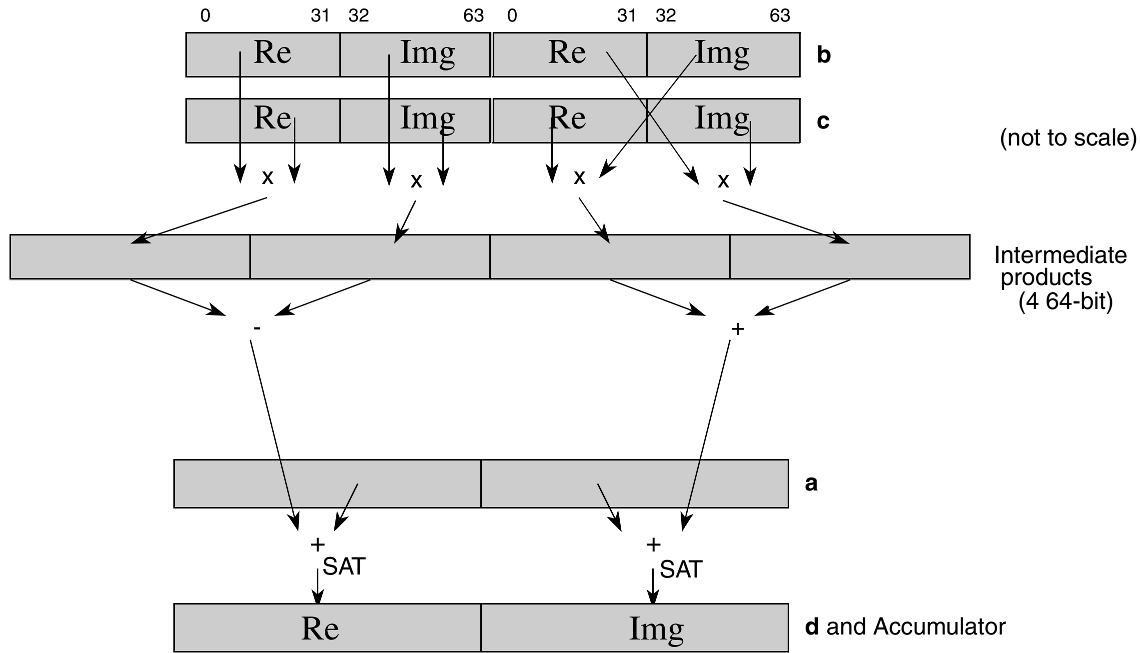


Figure 3-243. Vector Dot Product of Words, Complex, Signed, Saturate, Integer and Accumulate Words 3 op (__ev_dotpwcssiaaw3)

d	a	b	c	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	d ←a evdotpwcssiaaw3 d,b,c

__ev_dotpwgasmf[a] __ev_dotpwgasmf[a]

Vector Dot Product of Words Guarded, Add, Signed, Modulo, Fractional (to Accumulator)

d = __ev_dotpwgasmf (a,b) (A = 0)

d = __ev_dotpwgasmfa (a,b) (A = 1)

```
temph0:64 ←b0:31 ×sf a0:31
templ0:64 ←b32:63 ×sf a32:63

temp0:65 ←EXTS66(temph0:64) + EXTS66(templ0:64)
d0:63 ←EXTS64(temp0:49)

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

The signed fractional word elements in parameter **b** are multiplied with the corresponding words in parameter **a** to produce a pair of 65-bit intermediate fractional products. If both inputs are -1.0, the intermediate product is represented as +1.0 ($^{1}0$ || 0x8000_0000_0000_0000), otherwise the two high-order product bits are the same. The intermediate products are sign-extended to 66 bits, added together, and the high-order 50 bits of the 66-bit sum are sign-extended to 64-bits to produce an intermediate sum in 17.47 fractional format, and the result 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 3-244. Vector Dot Product of Words Guarded, Add, Signed, Modulo, Fractional (to Accumulator) (__ev_dotpwgasmf[a])

A	d	a	b	Maps to
A = 0	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evdotpwgasmf d,a,b**
A = 1	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evdotpwgasmfa d,a,b**

__ev_dotpwgasmfaa __ev_dotpwgasmfaa

Vector Dot Product of Words Guarded, Add, Signed, Modulo, Fractional and Accumulate

d = __ev_dotpwgasmfaa (a,b)

```
temph0:64 ←b0:31 ×sf a0:31
templ0:64 ←b32:63 ×sf a32:63
temp0:65 ←EXTS66(temph0:64) + EXTS66(templ0:64)
d0:63 ←ACC0:63 + EXTS64(temp0:49)
ACC0:63 ←d0:63
```

The signed fractional word elements in parameter **b** are multiplied with the corresponding words in parameter **a** to produce a pair of 65-bit intermediate fractional products. If both inputs are -1.0, the intermediate product is represented as +1.0 ($^{1}0$ || 0x8000_0000_0000_0000), otherwise the two high-order product bits are the same. The intermediate products are sign-extended to 66 bits, added together, and the high-order 50 bits of the 66-bit sum are sign-extended to 64-bits to produce an intermediate sum in 17.47 fractional format which is added to the contents of the accumulator, and the result is placed into parameter **d** and the accumulator.

Other registers altered: ACC

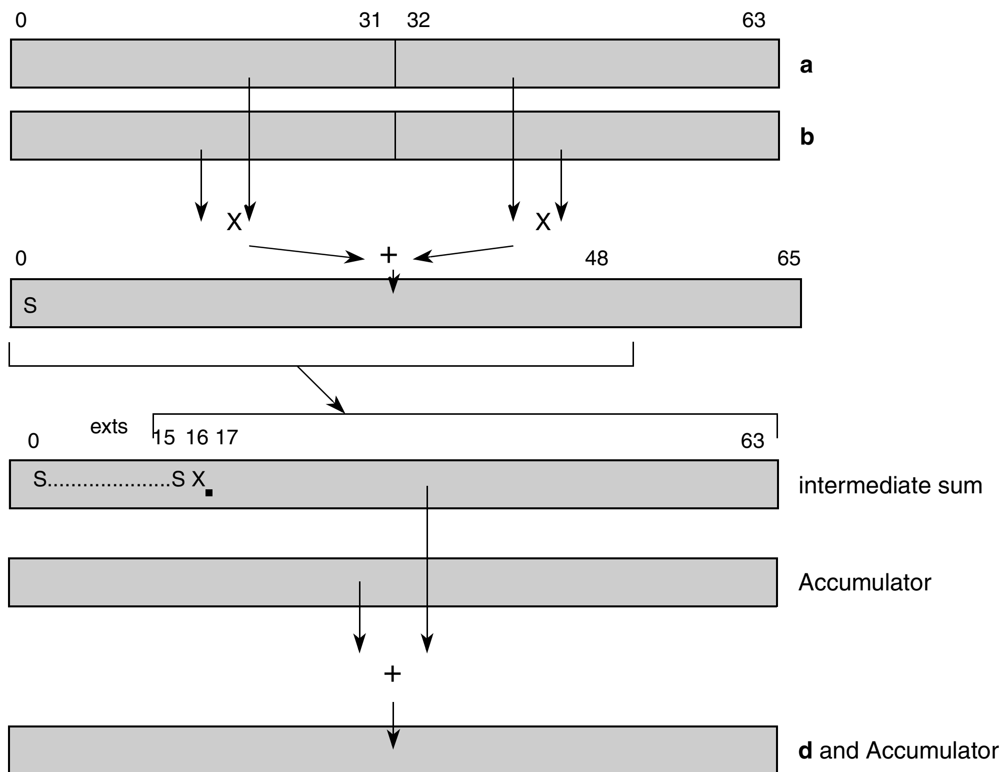


Figure 3-245. Vector Dot Product of Words Guarded, Add, Signed, Modulo, Fractional and Accumulate (__ev_dotpwgasmfaa)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evdotpwgasmfaa d,a,b**

__ev_dotpwgasmfaa3 __ev_dotpwgasmfaa3

Vector Dot Product of Words Guarded, Add, Signed, Modulo, Fractional and Accumulate 3 operand

d = __ev_dotpwgasmfaa3 (a,b,c)

$temph_{0:64} \leftarrow c_{0:31} \times_{sf} b_{0:31}$
$templ_{0:64} \leftarrow c_{32:63} \times_{sf} b_{32:63}$
$temp_{0:65} \leftarrow EXTS_{66}(temph_{0:64}) + EXTS_{66}(templ_{0:64})$
$d_{0:63} \leftarrow a_{0:63} + EXTS_{64}(temp_{0:49})$
$ACC_{0:63} \leftarrow d_{0:63}$

The signed fractional word elements in parameter **c** are multiplied with the corresponding words in parameter **b** to produce a pair of 65-bit intermediate fractional products. If both inputs are -1.0, the intermediate product is represented as +1.0 ($^{1}0$ || 0x8000_0000_0000_0000), otherwise the two high-order product bits are the same. The intermediate products are sign-extended to 66 bits, added together, and the high-order 50 bits of the 66-bit sum are sign-extended to 64-bits to produce an intermediate sum in 17.47 fractional format which is added to the contents of parameter **a**, and the result is placed into parameter **d** and the accumulator.

Other registers altered: ACC

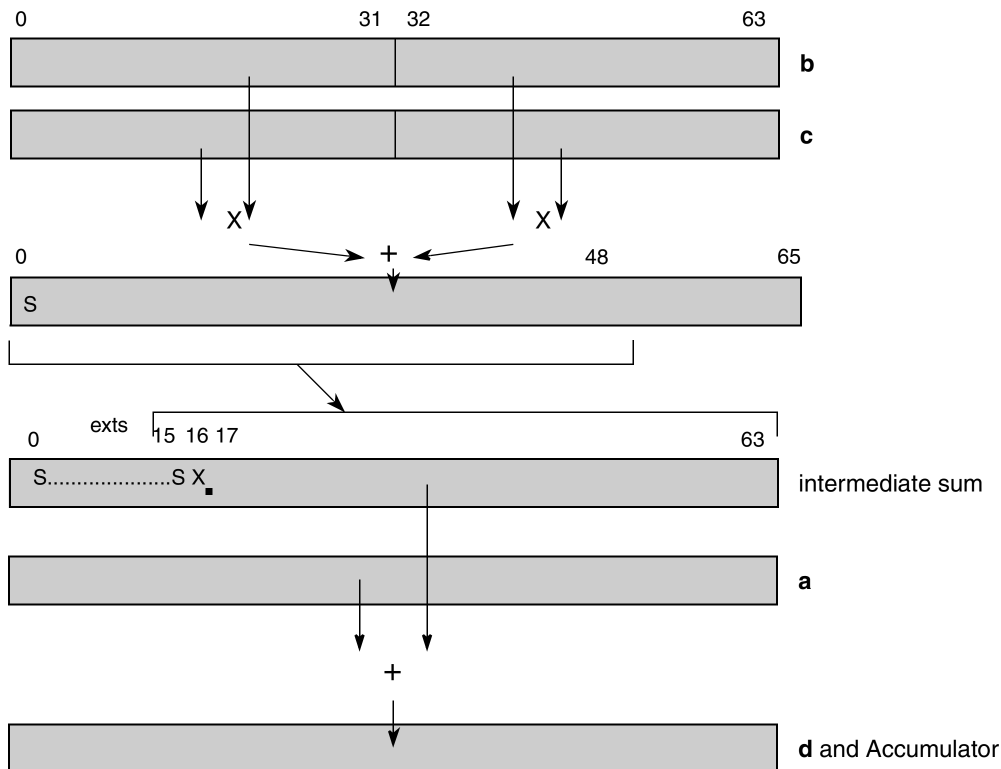


Figure 3-246. Vector Dot Product of Words Guarded, Add, Signed, Modulo, Fractional and Accumulate 3 op (__ev_dotpwgasmfaa3)

d	a	b	c	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	d ←a **evdotpwgasmfaa3 d,b,c**

__ev_dotpwgasmfr[a] __ev_dotpwgasmfr[a]

Vector Dot Product of Words Guarded, Add, Signed, Modulo, Fractional, Round (to Accumulator)

d = __ev_dotpwgasmfr (a,b) (A = 0)

d = __ev_dotpwgasmfra (a,b) (A = 1)

```
temph0:64 ←b0:31 ×sf a0:31
templ0:64 ←b32:63 ×sf a32:63
temp0:66 ←EXTS67(temph0:64) + EXTS67(templ0:64)
tempr0:66 ←ROUND(temp0:66,16)
d0:63 ←EXTS64(tempr0:50)

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

The signed fractional word elements in parameter **b** are multiplied with the corresponding words in parameter **a** to produce a pair of 65-bit intermediate fractional products. If both inputs are -1.0, the intermediate product is represented as +1.0 ($^{1}0$ || 0x8000_0000_0000_0000), otherwise the two high-order product bits are the same. The intermediate products are sign-extended to 67 bits, added together, and rounded using the current rounding mode in SPEFSCR to discard 16 low order bits. The high order 51 bits of the 67-bit rounded sum are sign-extended to 64-bits to produce an intermediate sum in 17.47 fractional format, and the result 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)

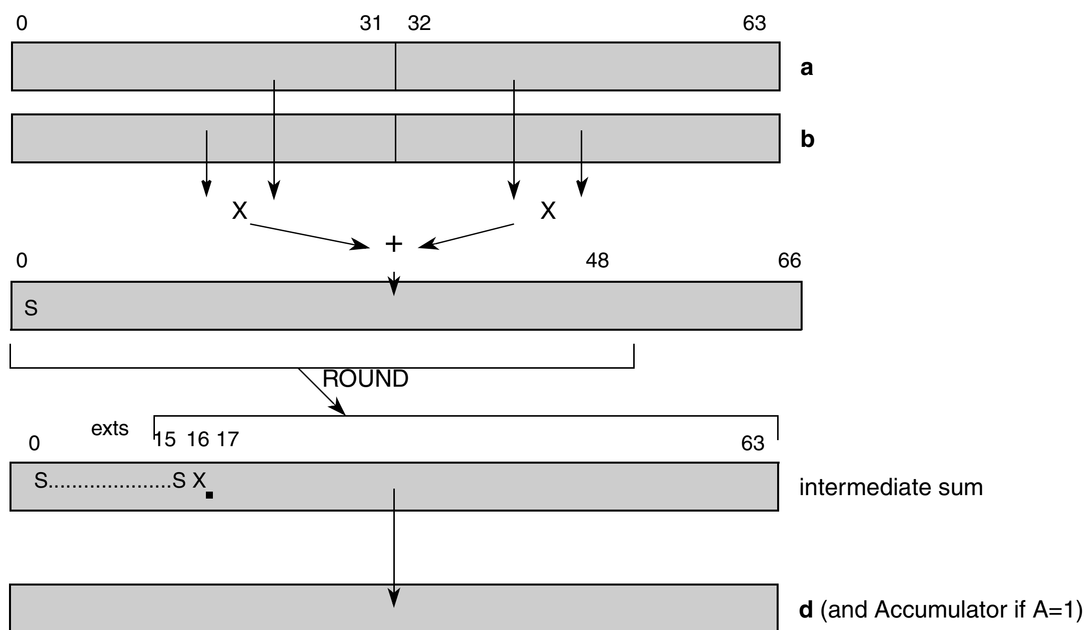


Figure 3-247. Vector Dot Product of Words Guarded, Add, Signed, Modulo, Fractional, Round (to Accumulator) (__ev_dotpwgasmfr[a])

A	d	a	b	Maps to
A = 0	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evdotpwgasmfr d,a,b**
A = 1	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evdotpwgasmfra d,a,b**

__ev_dotpwgasmfraa __ev_dotpwgasmfraa

Vector Dot Product of Words Guarded, Add, Signed, Modulo, Fractional, Round and Accumulate

d = __ev_dotpwgasmfraa (a,b)

$$temph_{0:64} \leftarrow b_{0:31} \times_{sf} a_{0:31};\ templ_{0:64} \leftarrow b_{32:63} \times_{sf} a_{32:63}$$
$$temp_{0:66} \leftarrow EXTS_{67}(temph_{0:64}) + EXTS_{67}(templ_{0:64})$$
$$tempr_{0:66} \leftarrow ROUND(temp_{0:66},16)$$
$$d_{0:63} \leftarrow ACC_{0:63} + EXTS_{64}(tempr_{0:50})$$
$$ACC_{0:63} \leftarrow d_{0:63}$$

The signed fractional word elements in parameter **b** are multiplied with the corresponding words in parameter **a** to produce a pair of 65-bit intermediate fractional products. If both inputs are -1.0, the intermediate product is represented as +1.0 ($^{1}0$ || 0x8000_0000_0000_0000), otherwise the two high-order product bits are the same. The intermediate products are sign-extended to 67 bits, added together, and rounded using the current rounding mode in SPEFSCR to discard 16 low order bits. The high order 51 bits of the 67-bit rounded sum are sign-extended to 64-bits to produce an intermediate sum in 17.47 fractional format. The intermediate sum is added to the contents of the accumulator, and the result is placed into parameter **d** and the accumulator.

Other registers altered: ACC

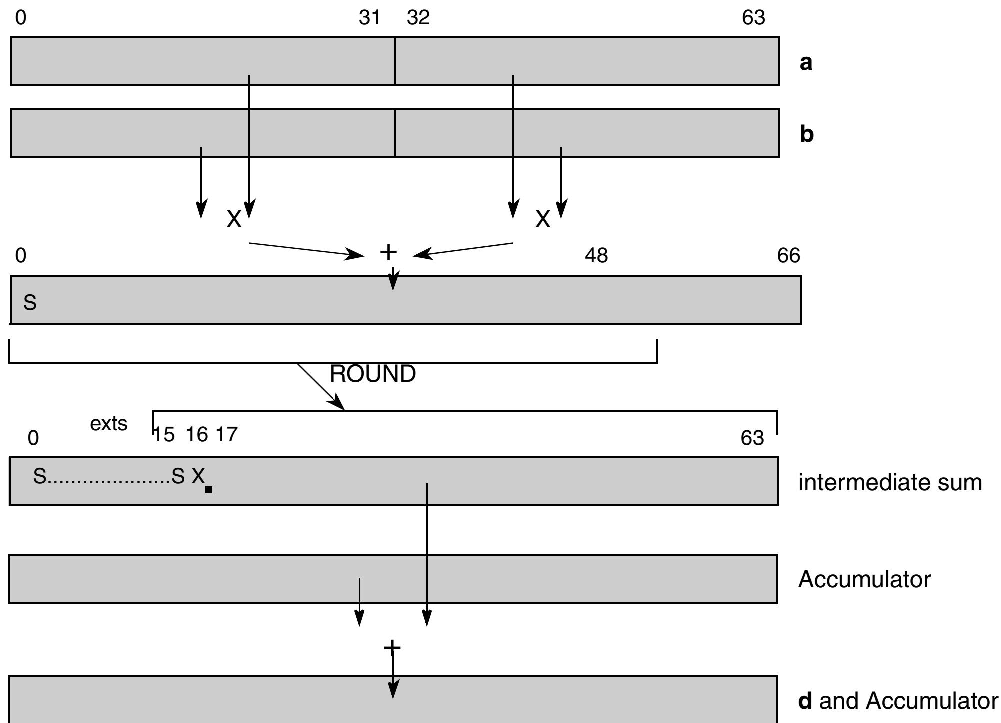


Figure 3-248. Vector Dot Product of Words Guarded, Add, Signed, Modulo, Fractional, Round and Accumulate (__ev_dotpwgasmfraa)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evdotpwgasmfraa d,a,b**

__ev_dotpwgasmfraa3 __ev_dotpwgasmfraa3

Vector Dot Product of Words Guarded, Add, Signed, Modulo, Fractional, Round and Accumulate 3 operand

d = __ev_dotpwgasmfraa3 (a,b,c)

$temph_{0:64} \leftarrow c_{0:31} \times_{sf} b_{0:31};\ templ_{0:64} \leftarrow c_{32:63} \times_{sf} b_{32:63}$
$temp_{0:66} \leftarrow EXTS_{67}(temph_{0:64}) + EXTS_{67}(templ_{0:64})$
$tempr_{0:66} \leftarrow ROUND(temp_{0:66},16)$
$d_{0:63} \leftarrow a_{0:63} + EXTS_{64}(tempr_{0:50})$
$ACC_{0:63} \leftarrow d_{0:63}$

The signed fractional word elements in parameter **c** are multiplied with the corresponding words in parameter **b** to produce a pair of 65-bit intermediate fractional products. If both inputs are -1.0, the intermediate product is represented as +1.0 ($^{1}0$ || 0x8000_0000_0000_0000), otherwise the two high-order product bits are the same. The intermediate products are sign-extended to 67 bits, added together, and rounded using the current rounding mode in SPEFSCR to discard 16 low order bits. The high order 51 bits of the 67-bit rounded sum are sign-extended to 64-bits to produce an intermediate sum in 17.47 fractional format. The intermediate sum is added to the contents of parameter **a**, and the result is placed into parameter **d** and the accumulator.

Other registers altered: ACC

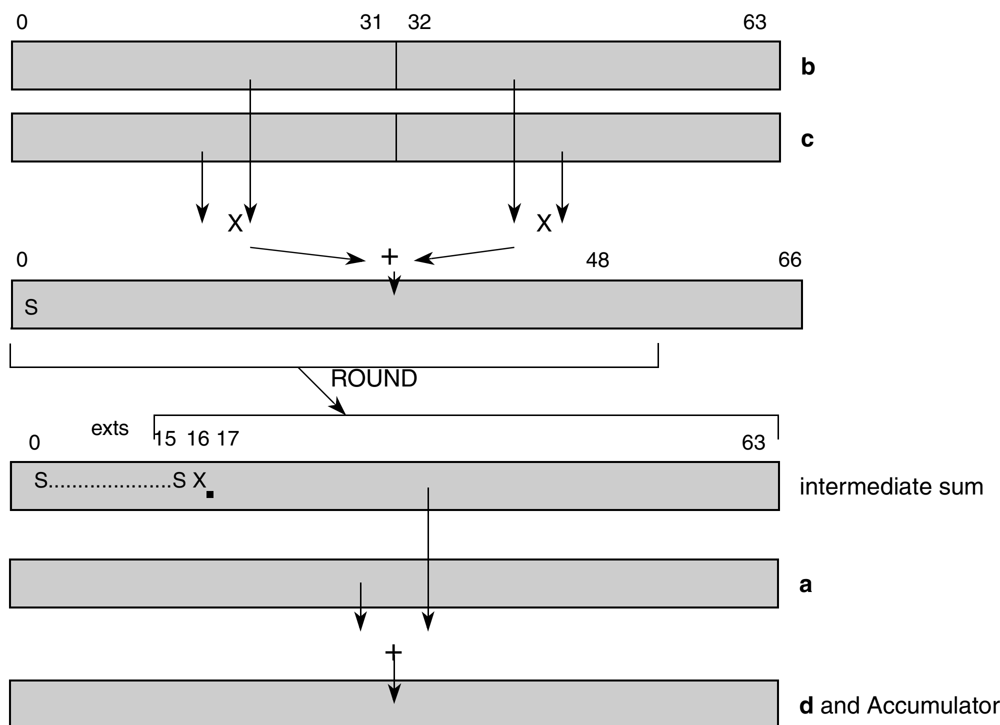


Figure 3-249. Vector Dot Product of Words Guarded, Add, Signed, Modulo, Fractional, Round and Accumulate 3 op (__ev_dotpwgasmfraa3)

d	a	b	c	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**d ←a evdotpwgasmfraa3 d,b,c**

__ev_dotpwgssmf[a] __ev_dotpwgssmf[a]

Vector Dot Product of Words Guarded, Subtract, Signed, Modulo, Fractional (to Accumulator)

d = __ev_dotpwgssmf (a,b) (A = 0)

d = __ev_dotpwgssmfa (a,b) (A = 1)

```
temph0:64 ←a0:31 ×sf b0:31
templ0:64 ←a32:63 ×sf b32:63
temp0:65 ←EXTS66(temph0:64) - EXTS66(templ0:64)
d0:63 ←EXTS64(temp0:49)

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

The signed fractional word elements in parameters **a** and **b** are multiplied to produce a pair of 65-bit intermediate fractional products. If both inputs are -1.0, the intermediate product is represented as +1.0 ($^{1}0$ || 0x8000_0000_0000_0000), otherwise the two high-order product bits are the same. The intermediate products are sign-extended to 66 bits, the low-order product is subtracted from the high-order product, the high-order 50 bits of the 66-bit result are sign-extended to 64-bits to produce an intermediate difference in 17.47 fractional format, and the result 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 3-250. Vector Dot Product of Words Guarded, Subtract, Signed, Modulo, Fractional (to Accumulator) (__ev_dotpwgssmf[a])

A	d	a	b	Maps to
A = 0	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evdotpwgssmf d,a,b**
A = 1	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evdotpwgssmfa d,a,b**

__ev_dotpwgssmfaa   __ev_dotpwgssmfaa

Vector Multiply Word Complex, Real, Guarded, Signed, Modulo, Fractional and Accumulate

d = __ev_dotpwgssmfaa (a,b)

$$temph_{0:64} \leftarrow a_{0:31} \times_{sf} b_{0:31}$$
$$templ_{0:64} \leftarrow a_{32:63} \times_{sf} b_{32:63}$$
$$temp_{0:66} \leftarrow EXTS_{67}(temph_{0:64}) - EXTS_{67}(templ_{0:64})$$
$$d_{0:63} \leftarrow ACC_{0:63} + EXTS_{64}(temp_{0:50})$$
$$ACC_{0:63} \leftarrow d_{0:63}$$

The signed fractional word elements in parameters **a** and **b** are multiplied to produce a pair of 65-bit intermediate fractional products. If both inputs are -1.0, the intermediate product is represented as +1.0 ($^{1}0$ || 0x8000_0000_0000_0000), otherwise the two high-order product bits are the same. The intermediate products are sign-extended to 67 bits, the low-order product is subtracted from the high-order product, the high-order 51 bits of the 67-bit result are sign-extended to 64-bits to produce an intermediate difference in 17.47 fractional format which is added to the contents of the accumulator, and the result is placed into parameter **d** and the accumulator.

Other registers altered: ACC

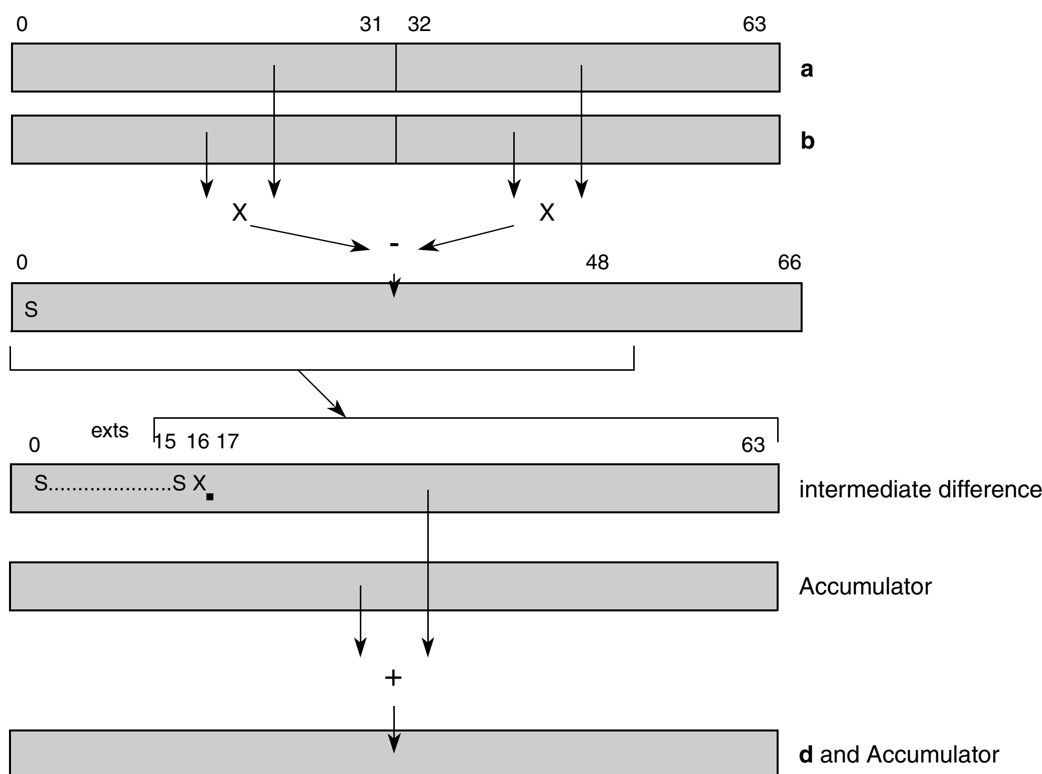


Figure 3-251. Vector Dot Product of Words Guarded, Subtract, Signed, Modulo, Fractional and Accumulate (__ev_dotpwgssmfaa)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evdotpwgssmfaa d,a,b**

__ev_dotpwgssmfaa3 __ev_dotpwgssmfaa3

Vector Multiply Word Complex, Real, Guarded, Signed, Modulo, Fractional and Accumulate 3 operand

d = __ev_dotpwgssmfaa3 (a,b,c)

```
temph0:64 ←b0:31 ×sf c0:31
templ0:64 ←b32:63 ×sf c32:63
temp0:66 ←EXTS67(temph0:64) - EXTS67(templ0:64)
d0:63 ←a0:63 + EXTS64(temp0:50)
ACC0:63 ←d0:63
```

The signed fractional word elements in parameters **b** and **c** are multiplied to produce a pair of 65-bit intermediate fractional products. If both inputs are -1.0, the intermediate product is represented as +1.0 ($^{1}0$ || 0x8000_0000_0000_0000), otherwise the two high-order product bits are the same. The intermediate products are sign-extended to 67 bits, the low-order product is subtracted from the high-order product, the high-order 51 bits of the 67-bit result are sign-extended to 64-bits to produce an intermediate difference in 17.47 fractional format which is added to the contents of parameter **a**, and the result is placed into parameter **d** and the accumulator.

Other registers altered: ACC

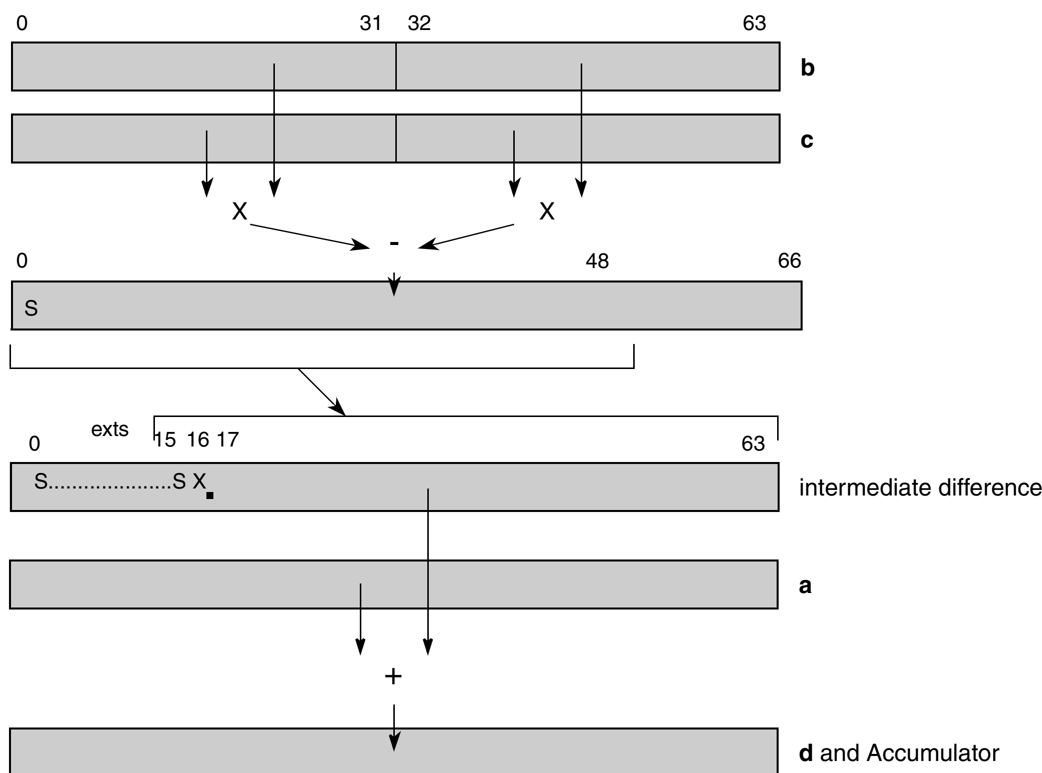


Figure 3-252. Vector Dot Product of Words Guarded, Subtract, Signed, Modulo, Fractional and Accumulate 3 op (__ev_dotpwgssmfaa3)

d	a	b	c	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**d ←a** **evdotpwgssmfaa3 d,b,c**

__ev_dotpwgssmfr[a] __ev_dotpwgssmfr[a]

Vector Dot Product of Words Guarded, Subtract, Signed, Modulo, Fractional, Round (to Accumulator)

d = __ev_dotpwgssmfr (a,b) (A = 0)

d = __ev_dotpwgssmfra (a,b) (A = 1)

```
temph0:64 ←b0:31 ×sf a0:31
templ0:64 ←b32:63 ×sf a32:63
temp0:66 ←EXTS67(temph0:64) - EXTS67(templ0:64)
tempr0:66 ←ROUND(temp0:66,16)
d0:63 ←EXTS64(tempr0:50)
// update accumulator
if A = 1 then ACC0:63 ←d0:63
```

The signed fractional word elements in parameter **b** are multiplied with the corresponding words in parameter **a** to produce a pair of 65-bit intermediate fractional products. If both inputs are -1.0, the intermediate product is represented as +1.0 ($^{1}0$ || 0x8000_0000_0000_0000), otherwise the two high-order product bits are the same. The intermediate products are sign-extended to 67 bits, the low-order product is subtracted from the high-order product and rounded using the current rounding mode in SPEFSCR to discard 16 low order bits. The high order 51 bits of the 67-bit rounded result are sign-extended to 64-bits to produce an intermediate difference in 17.47 fractional format, and the result 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)

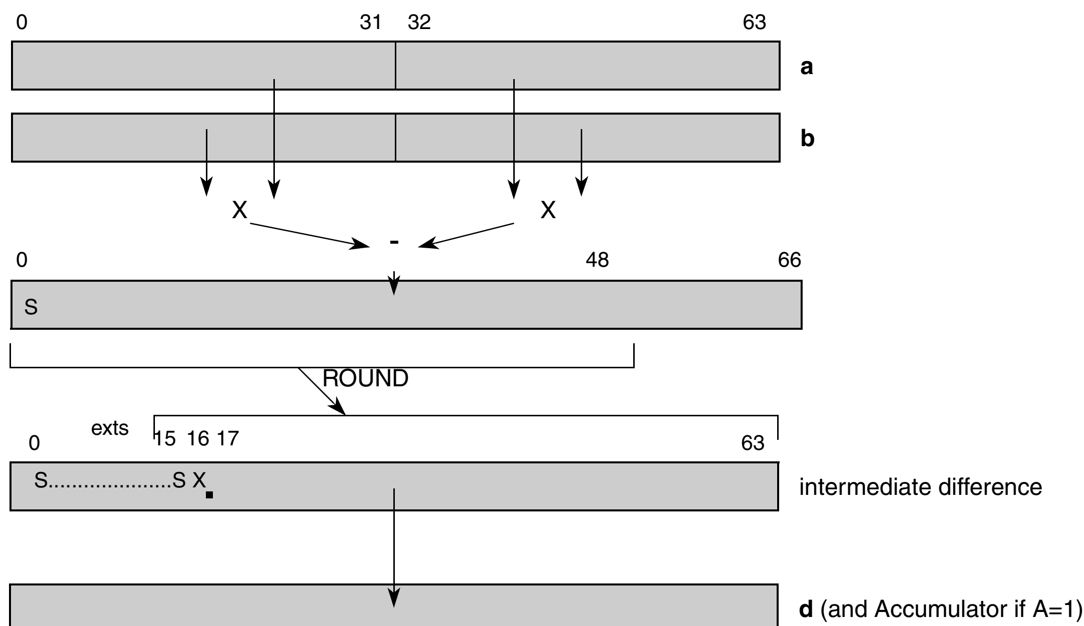


Figure 3-253. Vector Dot Product of Words Guarded, Subtract, Signed, Modulo, Fractional, Round (to Accumulator) (__ev_dotpwgssmfr[a])

A	d	a	b	Maps to
A = 0	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evdotpwgssmfr d,a,b**
A = 1	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evdotpwgssmfra d,a,b**

__ev_dotpwgssmfraa __ev_dotpwgssmfraa

Vector Dot Product of Words Guarded, Subtract, Signed, Modulo, Fractional, Round and Accumulate

d = __ev_dotpwgssmfraa (a,b)

$temph_{0:64} \leftarrow b_{0:31} \times_{sf} a_{0:31}$; $templ_{0:64} \leftarrow b_{32:63} \times_{sf} a_{32:63}$
$temp_{0:66} \leftarrow EXTS_{67}(temph_{0:64}) - EXTS_{67}(templ_{0:64})$
$tempr_{0:66} \leftarrow ROUND(temp_{0:66},16)$
$d_{0:63} \leftarrow ACC_{0:63} + EXTS_{64}(tempr_{0:50})$
$ACC_{0:63} \leftarrow d_{0:63}$

The signed fractional word elements in parameter **b** are multiplied with the corresponding words in parameter **a** to produce a pair of 65-bit intermediate fractional products. If both inputs are -1.0, the intermediate product is represented as +1.0 ($^{1}0$ || 0x8000_0000_0000_0000), otherwise the two high-order product bits are the same. The intermediate products are sign-extended to 67 bits, the low-order product is subtracted from the high-order product and rounded using the current rounding mode in SPEFSCR to discard 16 low order bits. The high order 51 bits of the 67-bit rounded result are sign-extended to 64-bits to produce an intermediate difference in 17.47 fractional format which is added to the contents of the accumulator, and the result is placed into parameter **d** and the accumulator.

Other registers altered: ACC

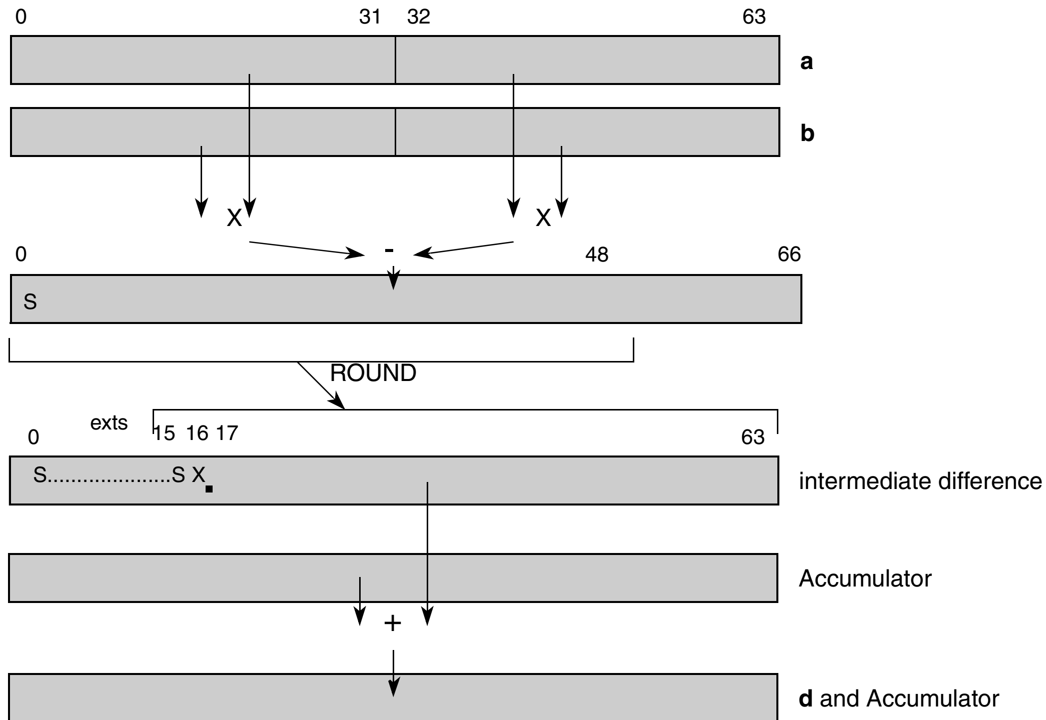


Figure 3-254. Vector Dot Product of Words Guarded, Add, Signed, Modulo, Fractional, Round and Accumulate (__ev_dotpwgssmfraa)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evdotpwgssmfraa d,a,b**

__ev_dotpwgssmfraa3 __ev_dotpwgssmfraa3

Vector Dot Product of Words Guarded, Subtract, Signed, Modulo, Fractional, Round and Accumulate 3 operand

d = __ev_dotpwgssmfraa3 (a,b,c)

$temph_{0:64} \leftarrow c_{0:31} \times_{sf} b_{0:31};\ templ_{0:64} \leftarrow c_{32:63} \times_{sf} b_{32:63}$
$temp_{0:66} \leftarrow EXTS_{67}(temph_{0:64}) - EXTS_{67}(templ_{0:64})$
$tempr_{0:66} \leftarrow ROUND(temp_{0:66}, 16)$
$d_{0:63} \leftarrow a_{0:63} + EXTS_{64}(tempr_{0:50})$
$ACC_{0:63} \leftarrow d_{0:63}$

The signed fractional word elements in parameter **c** are multiplied with the corresponding words in parameter **b** to produce a pair of 65-bit intermediate fractional products. If both inputs are -1.0, the intermediate product is represented as +1.0 ($^{1}0$ || 0x8000_0000_0000_0000), otherwise the two high-order product bits are the same. The intermediate products are sign-extended to 67 bits, the low-order product is subtracted from the high-order product and rounded using the current rounding mode in SPEFSCR to discard 16 low order bits. The high order 51 bits of the 67-bit rounded result are sign-extended to 64-bits to produce an intermediate difference in 17.47 fractional format which is added to the contents of parameter **a**, and the result is placed into parameter **d** and the accumulator.

Other registers altered: ACC

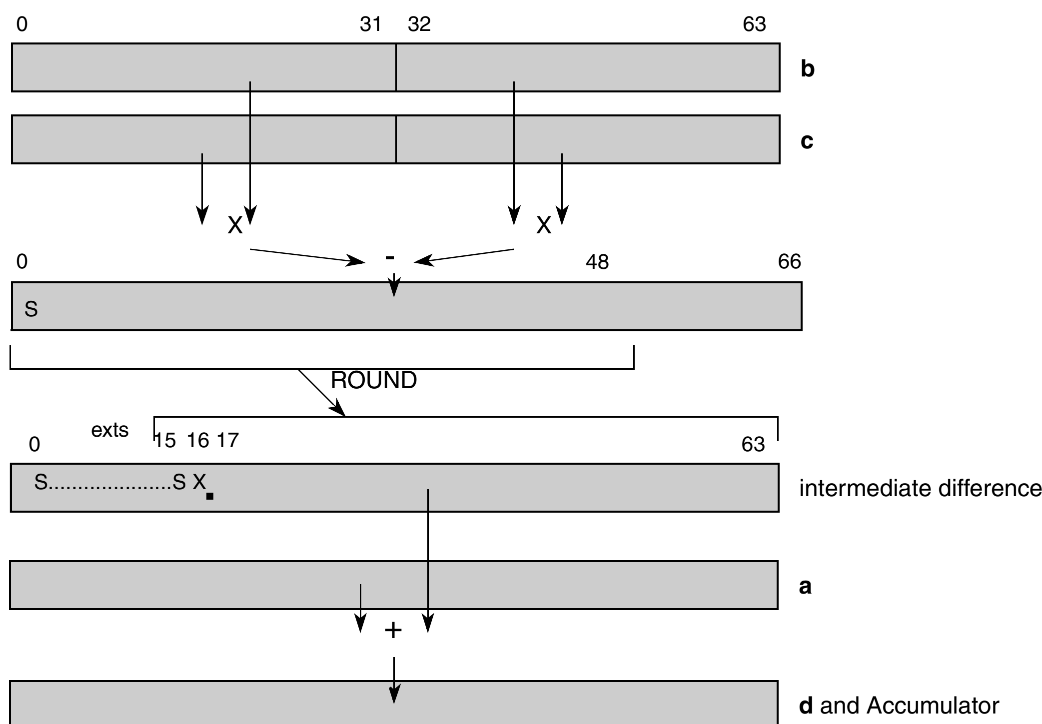


Figure 3-255. Vector Dot Product of Words Guarded, Subtract, Signed, Modulo, Fractional, Round and Accumulate 3 op (__ev_dotpwgssmfraa3)

d	a	b	c	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**d ←a** **evdotpwgssmfraa3 d,b,c**

__ev_dotpwssmi[a] __ev_dotpwssmi[a]

Vector Dot Product of Words, Add, Unsigned, Saturate, integer (to Accumulator)

d = __ev_dotpwssmi (a,b) (A = 0)

d = __ev_dotpwssmia (a,b) (A = 1)

```
temph0:63 ←a0:31 ×si b0:31
templ0:63 ←a32:63 ×si b32:63
temp0:31 ←temph0:63 - templ0:63 // modulo sum
d0:63 ←temp0:63

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

Corresponding pairs of signed integer word elements in parameters **a** and **b** are multiplied producing a pair of 64-bit products. The product of the low word elements is subtracted from the product of the high word elements, and the difference 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 3-256. Vector Dot Product of Words, Subtract, Signed, Modulo, Integer (to Accumulator) (__ev_dotpwssmi[a])

A	d	a	b	Maps to
A = 0	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evdotpwssmi d,a,b**
A = 1	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evdotpwssmia d,a,b**

__ev_dotpwssmiaa __ev_dotpwssmiaa

Vector Dot Product of Words, Subtract, Signed, Modulo, Integer and Accumulate

d = __ev_dotpwssmiaa (a,b)

```
temph0:63 ← a0:31 ×si b0:31
templ0:63 ← a32:63 ×si b32:63
temp0:63 ← temph0:63 - templ0:63 + ACC0:63 // modulo sum
d0:63 ← temp0:63

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

Corresponding pairs of signed integer word elements in parameters **a** and **b** are multiplied producing a pair of 64-bit products. The product of the low word elements is subtracted from the product of the high word elements, and the result is added to the contents of the accumulator and the sum is placed into parameter **d** and the accumulator.

Other registers altered: ACC

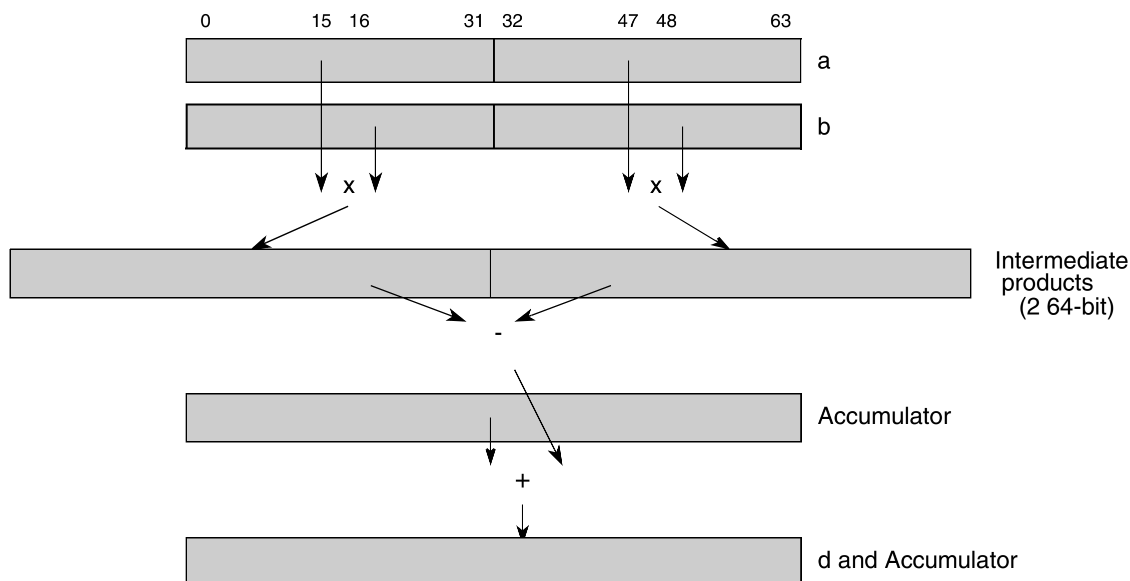


Figure 3-257. Vector Dot Product of Words, Subtract, Signed, Modulo, Integer and Accumulate (__ev_dotpwssmiaa)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evdotpwssmiaa d,a,b**

__ev_dotpwssmiaa3 __ev_dotpwssmiaa3

Vector Dot Product of Words, Subtract, Signed, Modulo, Integer and Accumulate, 3 operand

d = __ev_dotpwssmiaa3 (a,b,c)

```
temph0:63 ← b0:31 ×si c0:31
templ0:63 ← b32:63 ×si c32:63
temp0:63 ← temph0:63 - templ0:63 + a0:63 // modulo sum
d0:63 ← temp0:63

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

Corresponding pairs of signed integer word elements in parameters **b** and **c** are multiplied producing a pair of 64-bit products. The product of the low word elements is subtracted from the product of the high word elements, the result is added to the contents of parameter **a**, and the sum is placed into parameter **d** and the accumulator.

Other registers altered: ACC

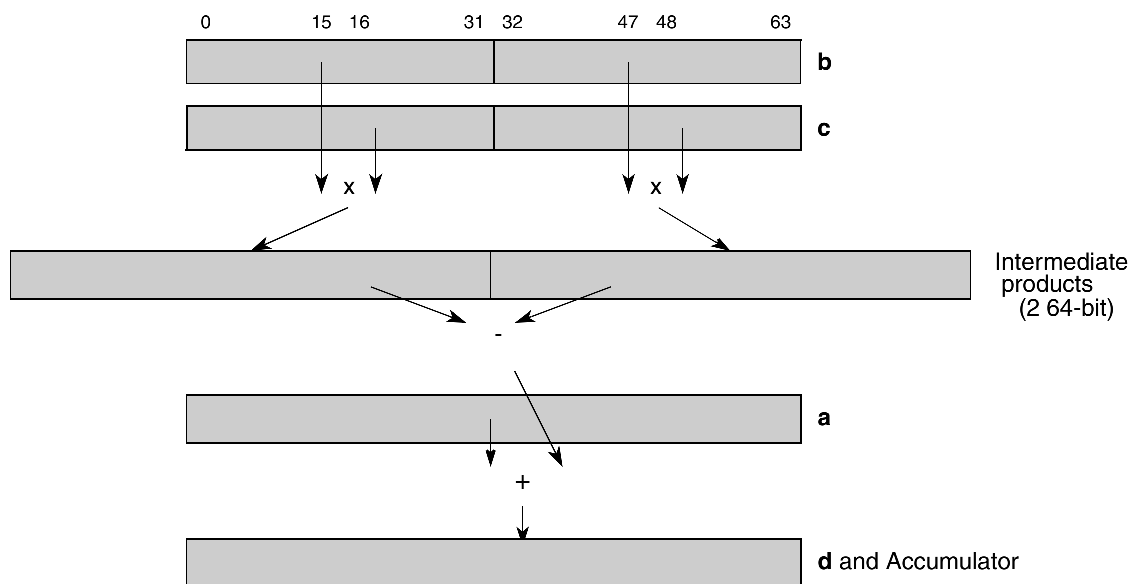


Figure 3-258. Vector Dot Product of Words, Subtract, Signed, Modulo, Integer and Accumulate 3op (__ev_dotpwssmiaa3)

d	a	b	c	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**d ←a** **evdotpwssmiaa3 d,b,c**

__ev_dotpwsssi[a] __ev_dotpwsssi[a]

Vector Dot Product of Words, Subtract, Signed, Saturate, integer (to Accumulator)

d = __ev_dotpwsssi (a,b) (A = 0)

d = __ev_dotpwsssia (a,b) (A = 1)

Maps to **evdotpwssmi** since no overflow can occur

A	d	a	b	Maps to
A = 0	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evdotpwssmi d,a,b**
A = 1	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evdotpwssmia d,a,b**

__ev_dotpwsssiaa __ev_dotpwsssiaa

Vector Dot Product of Words, Subtract, Signed, Saturate, Integer and Accumulate

d = __ev_dotpwsssiaa (a,b)

```
temph0:63 ←a0:31 ×si b0:31
templ0:63 ←a32:63 ×si b32:63

temp0:64 ←EXTS(ACC0:63) + EXTS(temph0:63) - EXTS(templ0:63)
ov ←(temp0 ⊕ temp1)
d0:63 ←SATURATE(ov, temp0, 0x8000_0000_0000_0000, 0x7FFF_FFFF_FFFF_FFFF, temp1:64)
// update accumulator
if A = 1 then ACC0:63 ←d0:63
// update SPEFSCR
SPEFSCROVH ←0; SPEFSCROV ←mov | ov
SPEFSCRSOV ←SPEFSCRSOV | mov | ov
```

Corresponding pairs of signed integer word elements in parameters **a** and **b** are multiplied producing a pair of 64-bit products. The product of the low word elements is subtracted from the product of the high word elements, then added together with the contents of the accumulator, saturating if overflow or underflow occurs in the final sum, and the sum is placed into parameter **d** and the accumulator.

The overflow and summary overflow bits are recorded in the SPEFSCR based on an overflow or underflow from the accumulation operations.

Other registers altered: SPEFSCR ACC

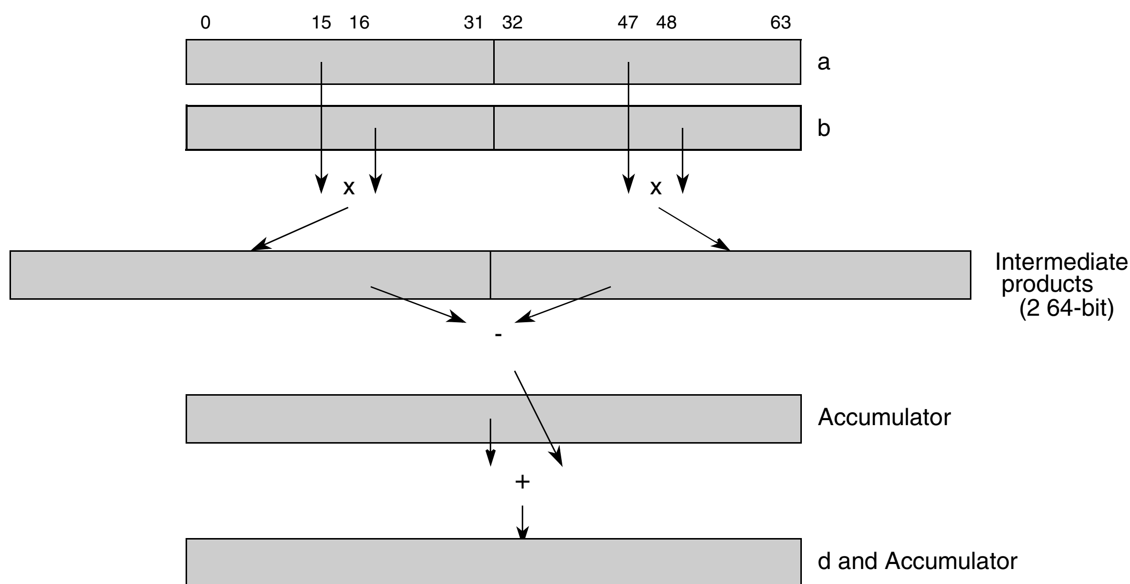


Figure 3-259. Vector Dot Product of Words, Add, Signed, Saturate, Integer and Accumulate (__ev_dotpwsssiaa)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evdotpwsssiaa d,a,b**

__ev_dotpwsssiaa3 __ev_dotpwsssiaa3

Vector Dot Product of Words, Subtract, Signed, Saturate, Integer and Accumulate, 3 operand

d = __ev_dotpwsssiaa3 (a,b,c)

```
temph0:63 ←b0:31 ×si c0:31
templ0:63 ←b32:63 ×si c32:63

temp0:66 ←EXTS67(a0:63) + EXTS67(temph0:63) - EXTS67(templ0:63)
ov ←chk_ovf(temp0:3)
d0:63 ←SATURATE(ov, temp0, 0x8000_0000_0000_0000, 0x7FFF_FFFF_FFFF_FFFF, temp3:66)
// update accumulator
ACC0:63 ←d0:63
// update SPEFSCR
SPEFSCROVH ←0; SPEFSCROV ←ov
SPEFSCRSOV ←SPEFSCRSOV | ov
```

Corresponding pairs of signed integer word elements in parameters **b** and **c** are multiplied producing a pair of 64-bit products. The product of the low word elements is subtracted from the product of the high word elements, then added together with the contents of parameter **a**, saturating if overflow or underflow occurs in the final sum, and the sum is placed into parameter **d** and the accumulator. The overflow and summary overflow bits are recorded in the SPEFSCR based on an overflow or underflow from the accumulation operations.

Other registers altered: SPEFSCR ACC

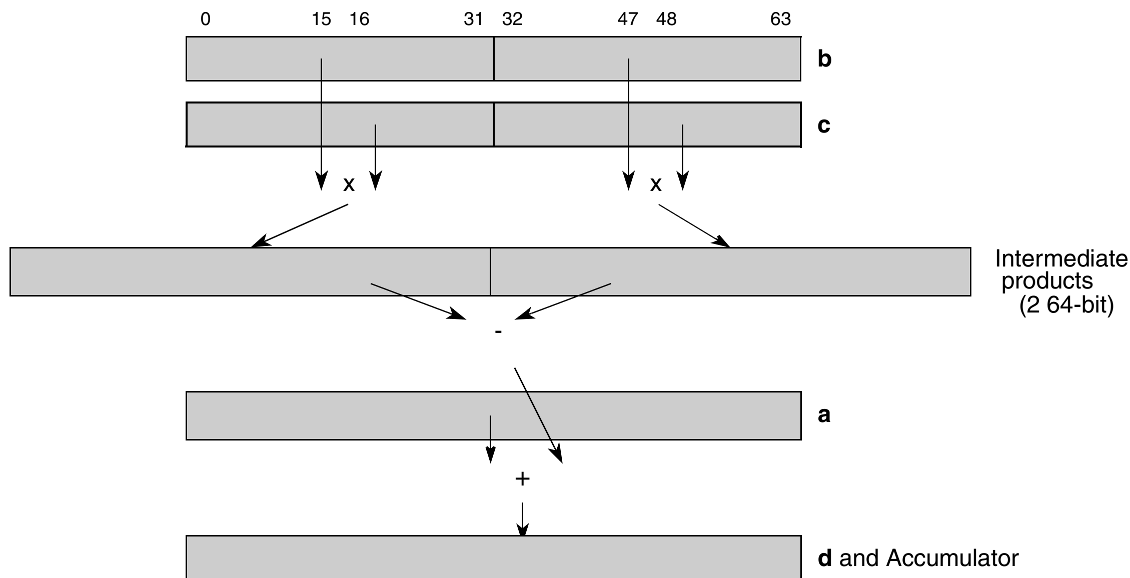


Figure 3-260. Vector Dot Product of Words, Subtract, Signed, Saturate, Integer and Accumulate 3 op (__ev_dotpwsssiaa3)

d	a	b	c	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	d ←a **evdotpwsssiaa3 d,b,c**

__ev_dotpwxgasmf[a] __ev_dotpwxgasmf[a]

Vector Dot Product of Words Exchanged, Guarded, Add, Signed, Modulo, Fractional (to Accumulator)

d = __ev_dotpwxgasmf (a,b) (A = 0)

d = __ev_dotpwxgasmfa (a,b) (A = 1)

```
temph0:64 ←b0:31 ×sf a32:63
templ0:64 ←b32:63 ×sf a0:31

temp0:65 ←EXTS66(temph0:64) + EXTS66(templ0:64)
d0:63 ←EXTS64(temp0:49)

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

The signed fractional word elements in parameter **b** are multiplied with the exchanged words in parameter **a** to produce a pair of 65-bit intermediate fractional products. If both inputs are -1.0, the intermediate product is represented as +1.0 ($^{1}0$ || 0x8000_0000_0000_0000), otherwise the two high-order product bits are the same. The intermediate products are sign-extended to 66 bits, added together, and the high-order 50 bits of the 66-bit sum are sign-extended to 64-bits to produce an intermediate sum in 17.47 fractional format, and the result 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)

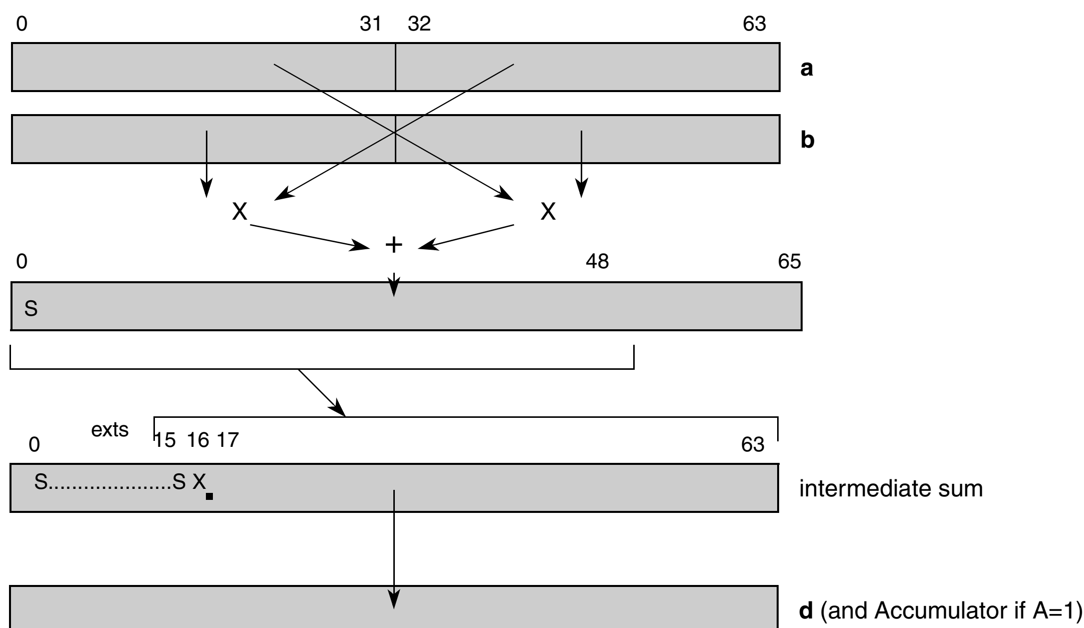


Figure 3-261. Vector Dot Product of Words Exchanged, Guarded, Add, Signed, Modulo, Fractional (to Accumulator) (__ev_dotpwxgasmf[a])

A	d	a	b	Maps to
A = 0	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evdotpwxgasmf d,a,b**
A = 1	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evdotpwxgasmfa d,a,b**

__ev_dotpwxgasmfaa __ev_dotpwxgasmfaa

Vector Dot Product of Words Exchanged, Guarded, Add, Signed, Modulo, Fractional and Accumulate

d = __ev_dotpwxgasmfaa (a,b)

$$temph_{0:64} \leftarrow b_{0:31} \times_{sf} a_{32:63}$$
$$templ_{0:64} \leftarrow b_{32:63} \times_{sf} a_{0:31}$$

$$temp_{0:65} \leftarrow EXTS_{66}(temph_{0:64}) + EXTS_{66}(templ_{0:64})$$
$$d_{0:63} \leftarrow ACC_{0:63} + EXTS_{64}(temp_{0:49})$$
$$ACC_{0:63} \leftarrow d_{0:63}$$

The signed fractional word elements in parameter **b** are multiplied with the exchanged words in parameter **a** to produce a pair of 65-bit intermediate fractional products. If both inputs are -1.0, the intermediate product is represented as +1.0 ($^{1}0$ || 0x8000_0000_0000_0000), otherwise the two high-order product bits are the same. The intermediate products are sign-extended to 66 bits, added together, and the high-order 50 bits of the 66-bit sum are sign-extended to 64-bits to produce an intermediate sum in 17.47 fractional format which is added to the contents of the accumulator, and the result is placed into parameter **d** and the accumulator.

Other registers altered: ACC

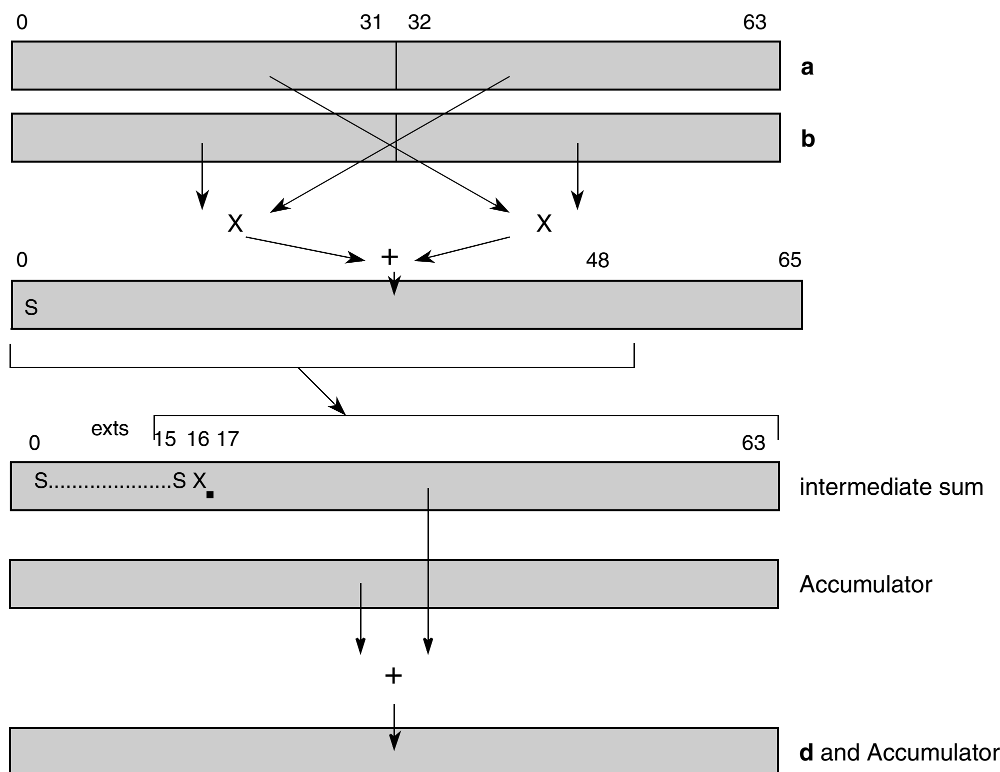


Figure 3-262. Vector Dot Product of Words Exchanged, Guarded, Add, Signed, Modulo, Fractional and Accumulate (__ev_dotpwxgasmfaa)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evdotpwxgasmfaa d,a,b**

__ev_dotpwxgasmfaa3 __ev_dotpwxgasmfaa3

Vector Dot Product of Words Exchanged, Guarded, Add, Signed, Modulo, Fractional and Accumulate 3 operand

d = __ev_dotpwxgasmfaa3 (a,b,c)

$temph_{0:64} \leftarrow c_{0:31} \times_{sf} b_{32:63}$
$templ_{0:64} \leftarrow c_{32:63} \times_{sf} b_{0:31}$

$temp_{0:65} \leftarrow EXTS_{66}(temph_{0:64}) + EXTS_{66}(templ_{0:64})$
$d_{0:63} \leftarrow a_{0:63} + EXTS_{64}(temp_{0:49})$
$ACC_{0:63} \leftarrow d_{0:63}$

The signed fractional word elements in parameter **c** are multiplied with the exchanged words in parameter **b** to produce a pair of 65-bit intermediate fractional products. If both inputs are -1.0, the intermediate product is represented as +1.0 ($^{1}0$ || 0x8000_0000_0000_0000), otherwise the two high-order product bits are the same. The intermediate products are sign-extended to 66 bits, added together, and the high-order 50 bits of the 66-bit sum are sign-extended to 64-bits to produce an intermediate sum in 17.47 fractional format which is added to the contents of parameter **a**, and the result is placed into parameter **d** and the accumulator.

Other registers altered: ACC

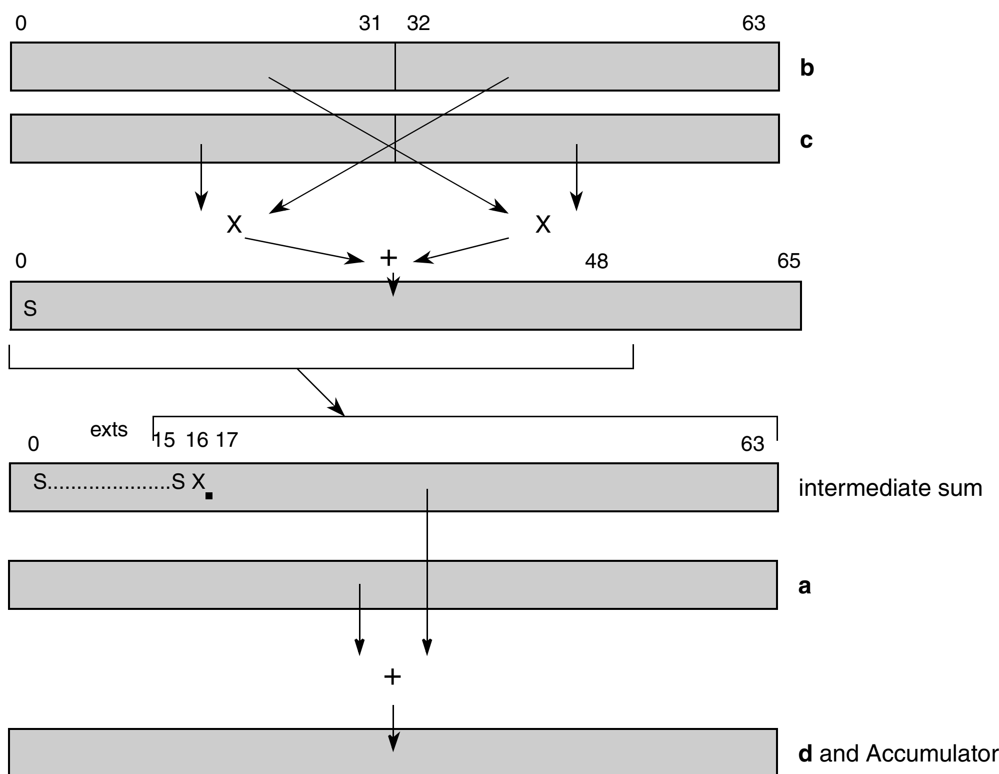


Figure 3-263. Vector Dot Product of Words Exchanged, Guarded, Add, Signed, Modulo, Fractional and Accumulate 3op (__ev_dotpwxgasmfaa3)

d	a	b	c	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**d ←a evdotpwxgasmfaa3 d,b,c**

__ev_dotpwxgasmfr[a] __ev_dotpwxgasmfr[a]

Vector Dot Product of Words Exchanged, Guarded, Add, Signed, Modulo, Fractional, Round (to Accumulator)

d = __ev_dotpwxgasmfr (a,b) (A = 0)

d = __ev_dotpwxgasmfra (a,b) (A = 1)

```
temph0:64 ←b0:31 ×sf a32:63
templ0:64 ←b32:63 ×sf a0:31
temp0:66 ←EXTS67(temph0:64) + EXTS67(templ0:64)
tempr0:66 ←ROUND(temp0:66,16)
d0:63 ←EXTS64(tempr0:50)
// update accumulator
if A = 1 then ACC0:63 ←d0:63
```

The signed fractional word elements in parameter **b** are multiplied with the exchanged words in parameter **a** to produce a pair of 65-bit intermediate fractional products. If both inputs are -1.0, the intermediate product is represented as +1.0 ($^{1}0$ || 0x8000_0000_0000_0000), otherwise the two high-order product bits are the same. The intermediate products are sign-extended to 67 bits, added together, and rounded using the current rounding mode in SPEFSCR to discard 16 low order bits. The high order 51 bits of the 67-bit rounded sum are sign-extended to 64-bits to produce an intermediate sum in 17.47 fractional format, and the result 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)

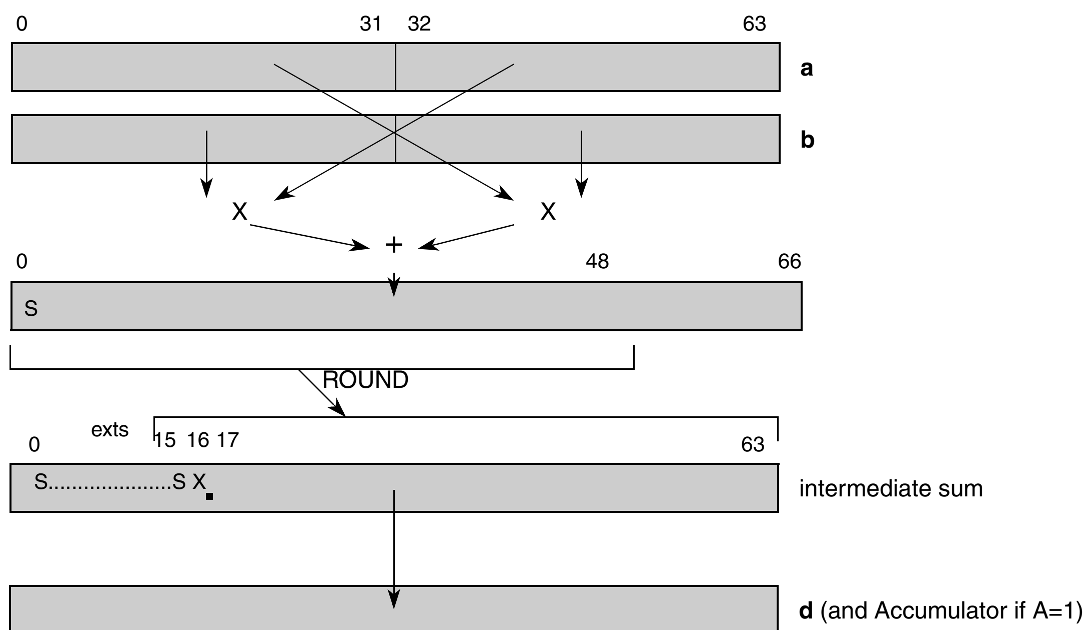


Figure 3-264. Vector Dot Product of Words Exchanged, Guarded, Add, Signed, Modulo, Fractional, Round (to Accumulator) (__ev_dotpwxgasmfr[a])

A	d	a	b	Maps to
A = 0	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evdotpwxgasmfr d,a,b**
A = 1	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evdotpwxgasmfra d,a,b**

__ev_dotpwxgasmfraa __ev_dotpwxgasmfraa

Vector Dot Product of Words Exchanged, Guarded, Add, Signed, Modulo, Fractional, Round and Accumulate

d = __ev_dotpwxgasmfraa (a,b)

$temph_{0:64} \leftarrow b_{0:31} \times_{sf} a_{32:63}$; $templ_{0:64} \leftarrow b_{32:63} \times_{sf} a_{0:31}$
$temp_{0:66} \leftarrow EXTS_{67}(temph_{0:64}) + EXTS_{67}(templ_{0:64})$
$tempr_{0:66} \leftarrow ROUND(temp_{0:66}, 16)$
$d_{0:63} \leftarrow ACC_{0:63} + EXTS_{64}(tempr_{0:50})$
$ACC_{0:63} \leftarrow d_{0:63}$

The signed fractional word elements in parameter **b** are multiplied with the exchanged words in parameter **a** to produce a pair of 65-bit intermediate fractional products. If both inputs are -1.0, the intermediate product is represented as +1.0 ($^{1}0$ || 0x8000_0000_0000_0000), otherwise the two high-order product bits are the same. The intermediate products are sign-extended to 67 bits, added together, and rounded using the current rounding mode in SPEFSCR to discard 16 low order bits. The high order 51 bits of the 67-bit rounded sum are sign-extended to 64-bits to produce an intermediate sum in 17.47 fractional format. The intermediate sum is added to the contents of the accumulator, and the result is placed into parameter **d** and the accumulator.

Other registers altered: ACC

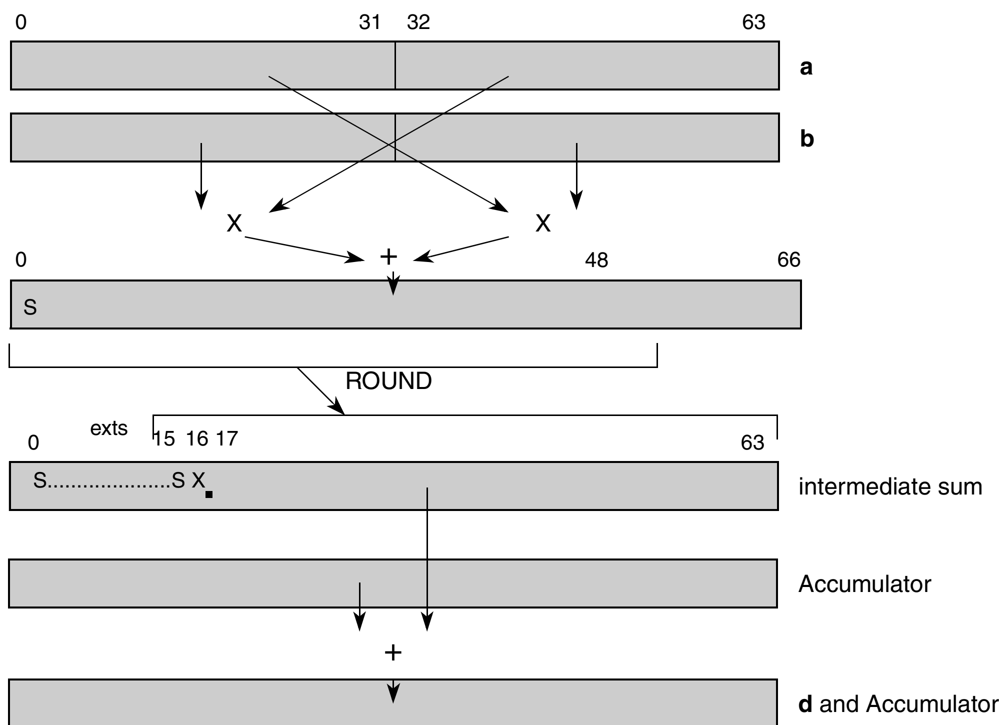


Figure 3-265. Vector Dot Product of Words Exchanged, Guarded, Add, Signed, Modulo, Fractional, Round and Accumulate (__ev_dotpwxgasmfraa)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evdotpwxgasmfraa d,a,b**

__ev_dotpwxgasmfraa3 __ev_dotpwxgasmfraa3

Vector Dot Product of Words Exchanged, Guarded, Add, Signed, Modulo, Fractional, Round and Accumulate 3 operand

d = __ev_dotpwxgasmfraa3 (a,b,c)

$$temph_{0:64} \leftarrow c_{0:31} \times_{sf} b_{32:63};\ templ_{0:64} \leftarrow c_{32:63} \times_{sf} b_{0:31}$$
$$temp_{0:66} \leftarrow EXTS_{67}(temph_{0:64}) + EXTS_{67}(templ_{0:64})$$
$$tempr_{0:66} \leftarrow ROUND(temp_{0:66}, 16)$$
$$d_{0:63} \leftarrow a_{0:63} + EXTS_{64}(tempr_{0:50})$$
$$ACC_{0:63} \leftarrow d_{0:63}$$

The signed fractional word elements in parameter **c** are multiplied with the exchanged words in parameter **b** to produce a pair of 65-bit intermediate fractional products. If both inputs are -1.0, the intermediate product is represented as +1.0 ($^{1}0$ || 0x8000_0000_0000_0000), otherwise the two high-order product bits are the same. The intermediate products are sign-extended to 67 bits, added together, and rounded using the current rounding mode in SPEFSCR to discard 16 low order bits. The high order 51 bits of the 67-bit rounded sum are sign-extended to 64-bits to produce an intermediate sum in 17.47 fractional format. The intermediate sum is added to the contents of parameter **a**, and the result is placed into parameter **d** and the accumulator.

Other registers altered: ACC

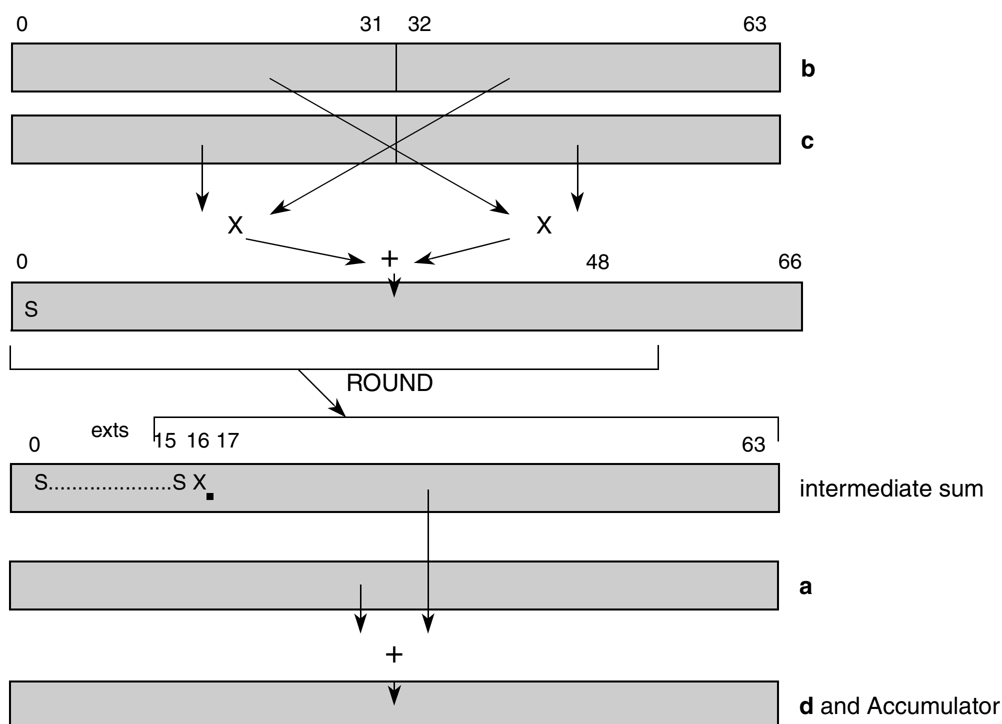


Figure 3-266. Vector Dot Product of Words Exchanged, Guarded, Add, Signed, Modulo, Fractional, Round and Accumulate 3op (__ev_dotpwxgasmfraa3)

d	a	b	c	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	d ←a **evdotpwxgasmfraa3 d,b,c**

__ev_dotpwxgssmf[a] __ev_dotpwxgssmf[a]

Vector Dot Product of Words Exchanged, Guarded, Subtract, Signed, Modulo, Fractional (to Accumulator)

d = __ev_dotpwxgssmf (a,b) (A = 0)

d = __ev_dotpwxgssmfa (a,b) (A = 1)

```
temph0:64 ←b0:31 ×sf a32:63
templ0:64 ←b32:63 ×sf a0:31
temp0:65 ←EXTS66(temph0:64) - EXTS66(templ0:64)
d0:63 ←EXTS64(temp0:49)

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

The signed fractional word elements in parameter **b** are multiplied with the exchanged words in parameter **a** to produce a pair of 65-bit intermediate fractional products. If both inputs are -1.0, the intermediate product is represented as +1.0 ($^{1}0$ || 0x8000_0000_0000_0000), otherwise the two high-order product bits are the same. The intermediate products are sign-extended to 66 bits, the low product is subtracted from the high product, and the high-order 50 bits of the 66-bit difference are sign-extended to 64-bits to produce an intermediate difference in 17.47 fractional format, and the result 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)

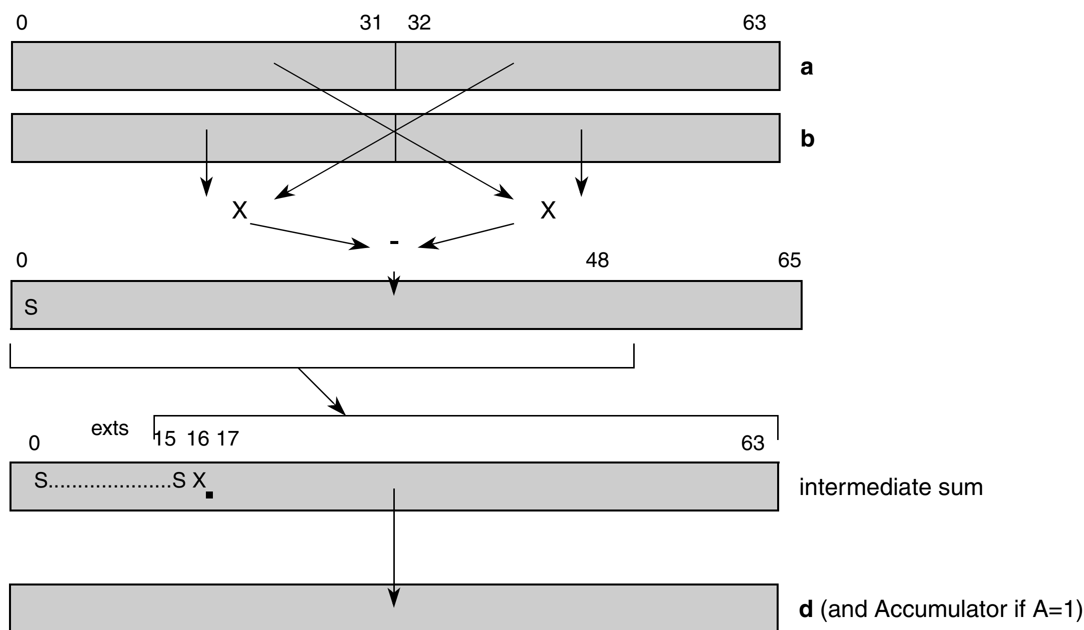


Figure 3-267. Vector Dot Product of Words Exchanged, Guarded, Subtract, Signed, Modulo, Fractional (to Accumulator) (__ev_dotpwxgssmf[a])

A	d	a	b	Maps to
A = 0	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evdotpwxgssmf d,a,b**
A = 1	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evdotpwxgssmfa d,a,b**

__ev_dotpwxgssmfaa __ev_dotpwxgssmfaa

Vector Dot Product of Words Exchanged, Guarded, Subtract, Signed, Modulo, Fractional and Accumulate

d = __ev_dotpwxgssmfaa (a,b)

```
temph0:64 ←b0:31 ×sf a32:63
templ0:64 ←b32:63 ×sf a0:31
temp0:65 ←EXTS66(temph0:64) - EXTS66(templ0:64)
d0:63 ←ACC0:63 + EXTS64(temp0:49)
ACC0:63 ←d0:63
```

The signed fractional word elements in parameter **b** are multiplied with the exchanged words in parameter **a** to produce a pair of 65-bit intermediate fractional products. If both inputs are -1.0, the intermediate product is represented as +1.0 ($^{1}0$ || 0x8000_0000_0000_0000), otherwise the two high-order product bits are the same. The intermediate products are sign-extended to 66 bits, the low product is subtracted from the high product, and the high-order 50 bits of the 66-bit difference are sign-extended to 64-bits to produce an intermediate difference in 17.47 fractional format which is added to the contents of the accumulator, and the result is placed into parameter **d** and the accumulator.

Other registers altered: ACC

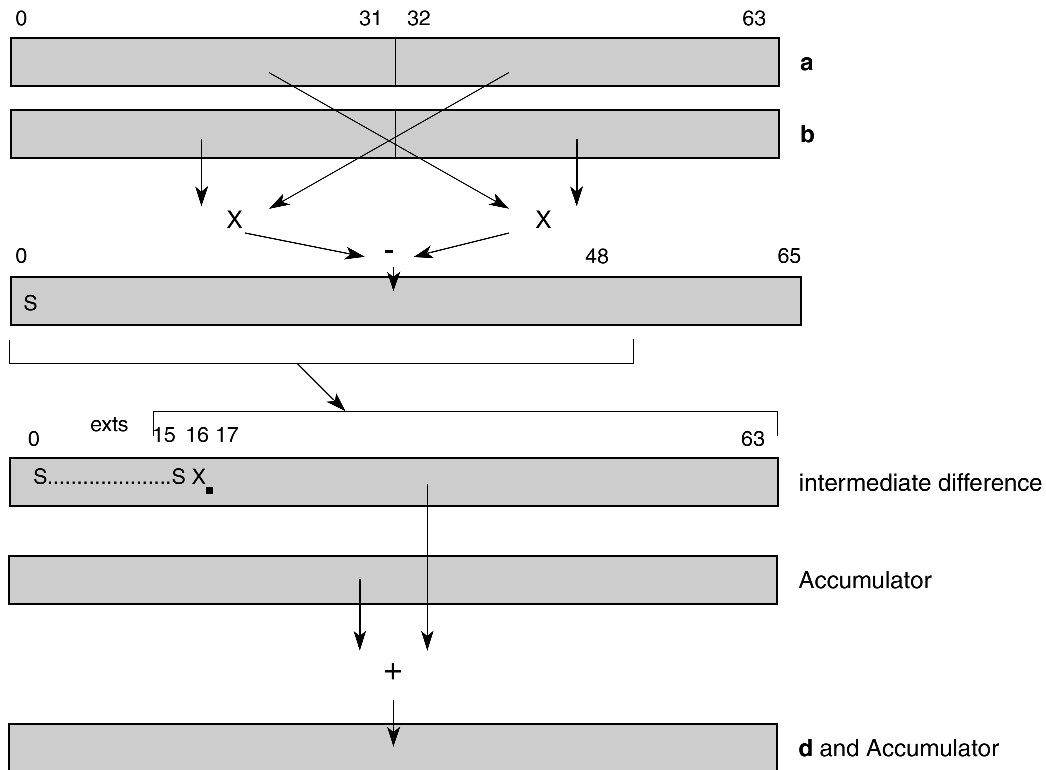


Figure 3-268. Vector Dot Product of Words Exchanged, Guarded, Subtract, Signed, Modulo, Fractional and Accumulate (__ev_dotpwxgssmfaa)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evdotpwxgssmfaa d,a,b**

__ev_dotpwxgssmfaa3 __ev_dotpwxgssmfaa3

Vector Dot Product of Words Exchanged, Guarded, Subtract, Signed, Modulo, Fractional and Accumulate 3 operand

d = __ev_dotpwxgssmfaa3 (a,b,c)

$temph_{0:64} \leftarrow c_{0:31} \times_{sf} b_{32:63}$
$templ_{0:64} \leftarrow c_{32:63} \times_{sf} b_{0:31}$
$temp_{0:65} \leftarrow EXTS_{66}(temph_{0:64}) + EXTS_{66}(templ_{0:64})$
$d_{0:63} \leftarrow a_{0:63} + EXTS_{64}(temp_{0:49})$
$ACC_{0:63} \leftarrow d_{0:63}$

The signed fractional word elements in parameter **c** are multiplied with the exchanged words in parameter **b** to produce a pair of 65-bit intermediate fractional products. If both inputs are -1.0, the intermediate product is represented as +1.0 ($^{1}0$ || 0x8000_0000_0000_0000), otherwise the two high-order product bits are the same. The intermediate products are sign-extended to 66 bits, the low product is subtracted from the high product, and the high-order 50 bits of the 66-bit difference are sign-extended to 64-bits to produce an intermediate difference in 17.47 fractional format which is added to the contents of parameter **a**, and the result is placed into parameter **d** and the accumulator.

Other registers altered: ACC

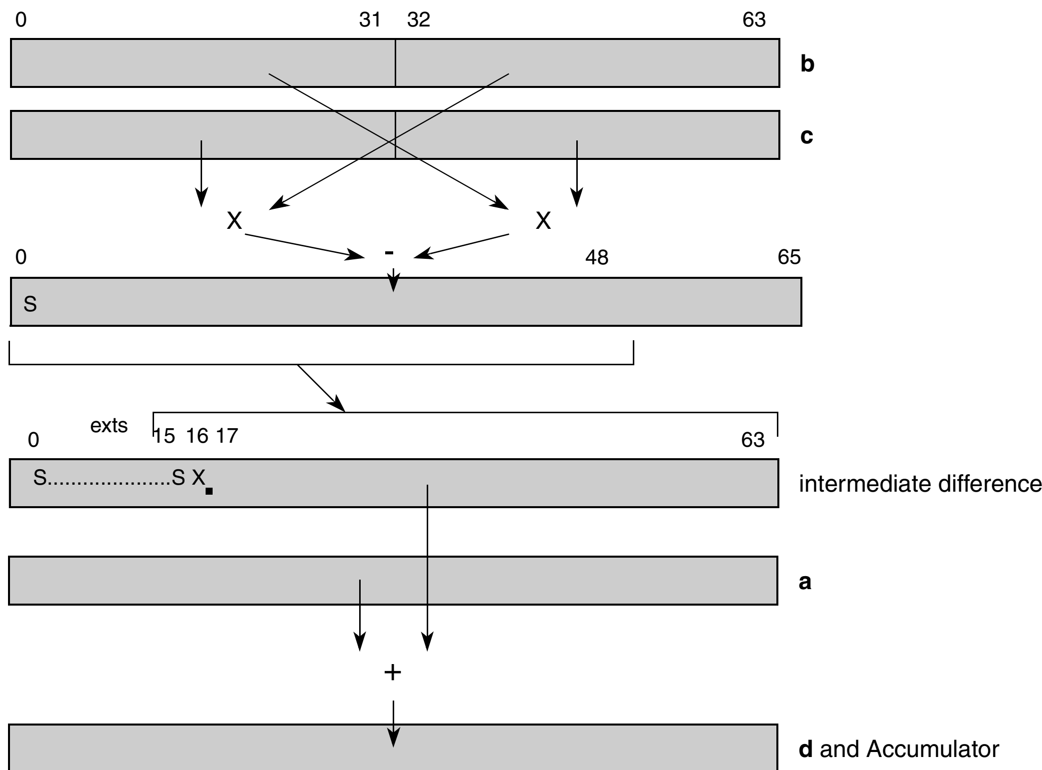


Figure 3-269. Vector Dot Product of Words Exchanged, Guarded, Subtract, Signed, Modulo, Fractional and Accumulate 3op (__ev_dotpwxgssmfaa3)

d	a	b	c	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**d ←a evdotpwxgssmfaa3 d,b,c**

__ev_dotpwxgssmfr[a] __ev_dotpwxgssmfr[a]

Vector Dot Product of Words Exchanged, Guarded, Add, Signed, Modulo, Fractional, Round (to Accumulator)

d = __ev_dotpwxgssmfr (a,b) (A = 0)

d = __ev_dotpwxgssmfra (a,b) (A = 1)

```
temph0:64 ←b0:31 ×sf a32:63
templ0:64 ←b32:63 ×sf a0:31
temp0:66 ←EXTS67(temph0:64) - EXTS67(templ0:64)
tempr0:66 ←ROUND(temp0:66,16)
d0:63 ←EXTS64(tempr0:50)
// update accumulator
if A = 1 then ACC0:63 ←d0:63
```

The signed fractional word elements in parameter **b** are multiplied with the exchanged words in parameter **a** to produce a pair of 65-bit intermediate fractional products. If both inputs are -1.0, the intermediate product is represented as +1.0 ($^{1}0$ || 0x8000_0000_0000_0000), otherwise the two high-order product bits are the same. The intermediate products are sign-extended to 67 bits, the low product is subtracted from the high product, and the difference is rounded using the current rounding mode in SPEFSCR to discard 16 low order bits. The high order 51 bits of the 67-bit rounded difference are sign-extended to 64-bits to produce an intermediate difference in 17.47 fractional format, and the result 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)

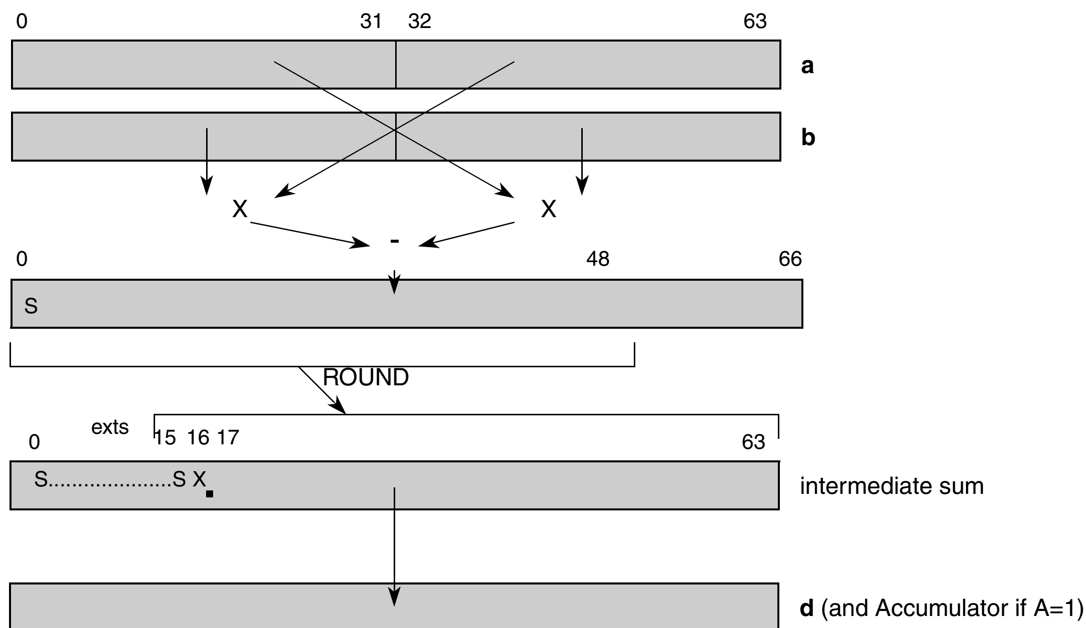


Figure 3-270. Vector Dot Product of Words Exchanged, Guarded, Subtract, Signed, Modulo, Fractional, Round (to Accumulator) (__ev_dotpwxgssmfr[a])

A	d	a	b	Maps to
A = 0	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evdotpwxgssmfr d,a,b**
A = 1	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evdotpwxgssmfra d,a,b**

__ev_dotpwxgssmfraa __ev_dotpwxgssmfraa

Vector Dot Product of Words Exchanged, Guarded, Subtract, Signed, Modulo, Fractional, Round and Accumulate

d = __ev_dotpwxgssmfraa (a,b)

$temph_{0:64} \leftarrow b_{0:31} \times_{sf} a_{32:63}$; $templ_{0:64} \leftarrow b_{32:63} \times_{sf} a_{0:31}$
$temp_{0:66} \leftarrow EXTS_{67}(temph_{0:64}) - EXTS_{67}(templ_{0:64})$
$tempr_{0:66} \leftarrow ROUND(temp_{0:66},16)$
$d_{0:63} \leftarrow ACC_{0:63} + EXTS_{64}(tempr_{0:50})$
$ACC_{0:63} \leftarrow d_{0:63}$

The signed fractional word elements in parameter **b** are multiplied with the exchanged words in parameter **a** to produce a pair of 65-bit intermediate fractional products. If both inputs are -1.0, the intermediate product is represented as +1.0 ($^{1}0$ || 0x8000_0000_0000_0000), otherwise the two high-order product bits are the same. The intermediate products are sign-extended to 67 bits, the low product is subtracted from the high product, and the difference is rounded using the current rounding mode in SPEFSCR to discard 16 low order bits. The high order 51 bits of the 67-bit rounded difference are sign-extended to 64-bits to produce an intermediate difference in 17.47 fractional format. The intermediate difference is added to the contents of the accumulator, and the result is placed into parameter **d** and the accumulator.

Other registers altered: ACC

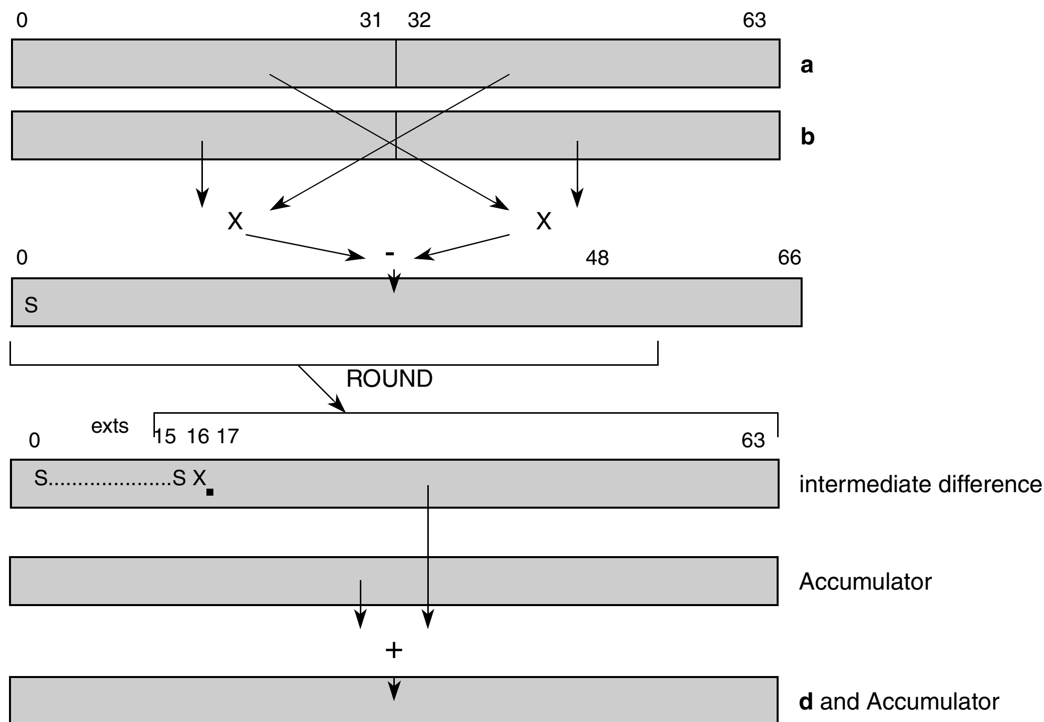


Figure 3-271. Vector Dot Product of Words Exchanged, Guarded, Subtract, Signed, Modulo, Fractional, Round and Accumulate (__ev_dotpwxgssmfraa)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evdotpwxgssmfraa d,a,b**

__ev_dotpwxgssmfraa3 __ev_dotpwxgssmfraa3

Vector Dot Product of Words Exchanged, Guarded, Subtract, Signed, Modulo, Fractional, Round and Accumulate 3 operand

d = __ev_dotpwxgssmfraa3 (a,b,c)

$temph_{0:64} \leftarrow c_{0:31} \times_{sf} b_{32:63}$; $templ_{0:64} \leftarrow c_{32:63} \times_{sf} b_{0:31}$
$temp_{0:66} \leftarrow EXTS_{67}(temph_{0:64}) - EXTS_{67}(templ_{0:64})$
$tempr_{0:66} \leftarrow ROUND(temp_{0:66}, 16)$
$d_{0:63} \leftarrow a_{0:63} + EXTS_{64}(tempr_{0:50})$
$ACC_{0:63} \leftarrow d_{0:63}$

The signed fractional word elements in parameter **c** are multiplied with the exchanged words in parameter **b** to produce a pair of 65-bit intermediate fractional products. If both inputs are -1.0, the intermediate product is represented as +1.0 ($^{1}0$ || 0x8000_0000_0000_0000), otherwise the two high-order product bits are the same. The intermediate products are sign-extended to 67 bits, the low product is subtracted from the high product, and the difference is rounded using the current rounding mode in SPEFSCR to discard 16 low order bits. The high order 51 bits of the 67-bit rounded difference are sign-extended to 64-bits to produce an intermediate difference in 17.47 fractional format. The intermediate difference is added to the contents of parameter **a**, and the result is placed into parameter **d** and the accumulator.

Other registers altered: ACC

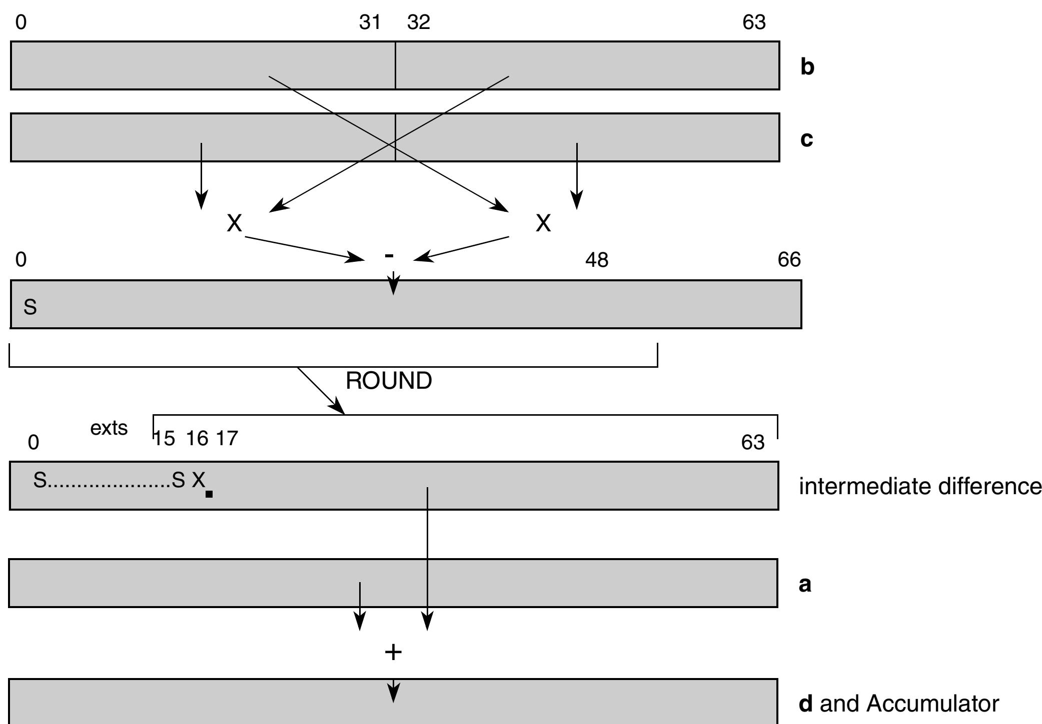


Figure 3-272. Vector Dot Product of Words Exchanged, Guarded, Subtract, Signed, Modulo, Fractional, Round and Accumulate 3op (__ev_dotpwxgssmfraa3)

d	a	b	c	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	d ←a **evdotpwxgssmfraa3 d,b,c**

__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**.

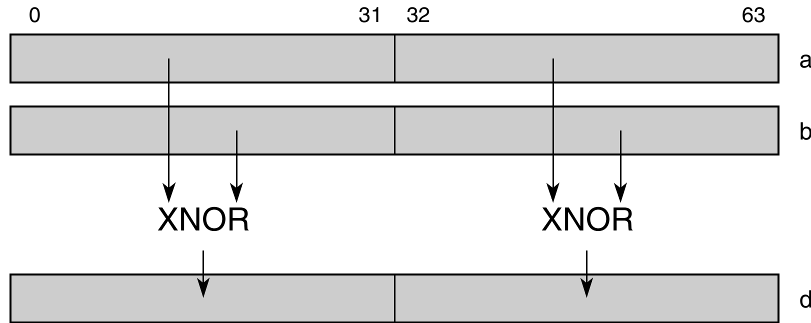


Figure 3-273. Vector Equivalent (__ev_eqv)

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)

```
d0:31  ← EXTS(a24:31)
d32:63 ← EXTS(a56:63)
```

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 3-274. Vector Extend Sign Byte (__ev_extsb)

d	a	Maps to
__ev64_opaque__	__ev64_opaque__	**evextsb d,a**

__ev_extsbh __ev_extsbh

Vector Extend Sign of Bytes (Odd) to Half Words

d = __ev_extsbh (a)

```
d0:15 ←EXTS(a8:15)
d16:31 ←EXTS(a24:31)
d32:47 ←EXTS(a40:47)
d48:63 ←EXTS(a56:63)
```

Each of the odd byte elements of parameter **a** are sign-extended to half words and placed into corresponding half word elements of parameter **d**.

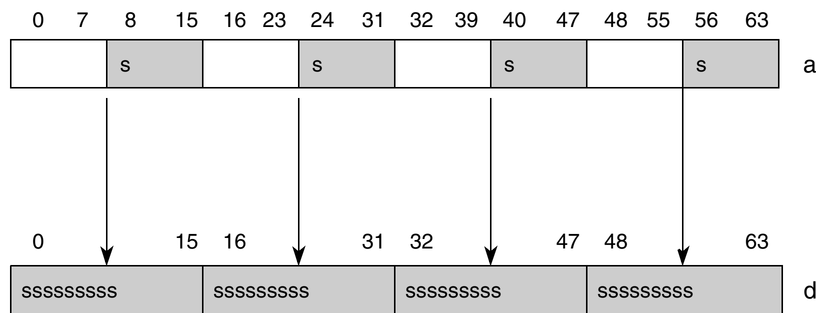


Figure 3-275. Vector Extend Sign Bytes (Odd) to Half Words (__ev_extsbh)

d	a	**Maps to**
__ev64_opaque__	__ev64_opaque__	**evextsbh d,a**

__ev_extsh __ev_extsh

Vector Extend Sign Half Word

d = __ev_extsh (a)

```
d0:31  ← EXTS(a16:31)
d32:63 ← EXTS(a48:63)
```

The odd half word elements in parameter **a** are sign-extended, and the results are placed in parameter **d**.

Figure 3-276. Vector Extend Sign Half Word (__ev_extsh)

d	a	Maps to
__ev64_opaque__	__ev64_opaque__	**evextsh d,a**

__ev_extsw __ev_extsw

Vector Extend Sign Word

d = __ev_extsw (a)

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

The low word element in parameter **a** is sign-extended, and the result is placed in parameter **d**.

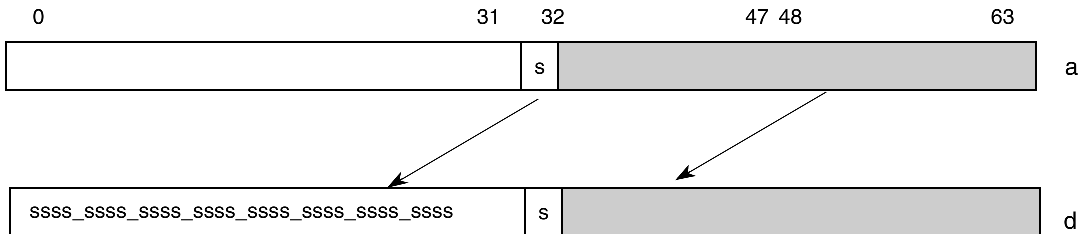


Figure 3-277. Vector Extend Sign Word (__ev_extsw)

d	a	Maps to
__ev64_opaque__	__ev64_opaque__	**evextsw d,a**

__ev_extzb __ev_extzb

Vector Extend Zero Byte

d = __ev_extzb (a)

```
d0:31  ←EXTZ(a24:31)
d32:63 ←EXTZ(a56:63)
```

The low-order byte in each of the elements in parameter **a** is zero-extended, and the results are placed in parameter **d**.

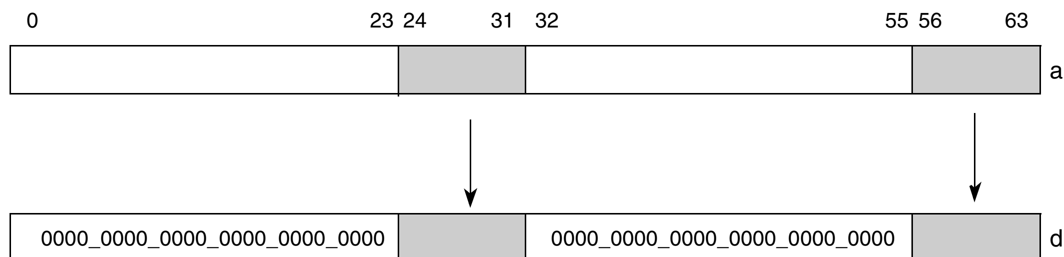


Figure 3-278. Vector Extend Zero Byte (__ev_extzb)

d	a	Maps to
__ev64_opaque__	__ev64_opaque__	**evextzb d,a**

__ev_insb __ev_insb

Vector Insert Byte

d = __ev_insb (a,b,c,s)

```
temp0:7 ←b0+(c*8):7+(c*8)
if (s=0) then d0:7 ←temp0:7 else d0:7 ←a0:7
if (s=1) then d8:15 ←temp0:7 else d8:15 ←a8:15
if (s=2) then d16:23 ←temp0:7 else d16:23 ←a16:23
if (s=3) then d24:31 ←temp0:7 else d24:31 ←a24:31
if (s=4) then d32:39 ←temp0:7 else d32:39 ←a32:39
if (s=5) then d40:47 ←temp0:7 else d40:47 ←a40:47
if (s=6) then d48:55 ←temp0:7 else d48:55 ←a48:55
if (s=7) then d56:63 ←temp0:7 else d56:63 ←a56:63
```

The byte element of parameter **b** specified by parameter **c** is placed into the byte element of parameter **d** specified by parameter **s**. The remaining bytes of parameter **d** are copied from parameter **a**. Byte 0 is the most-significant byte.

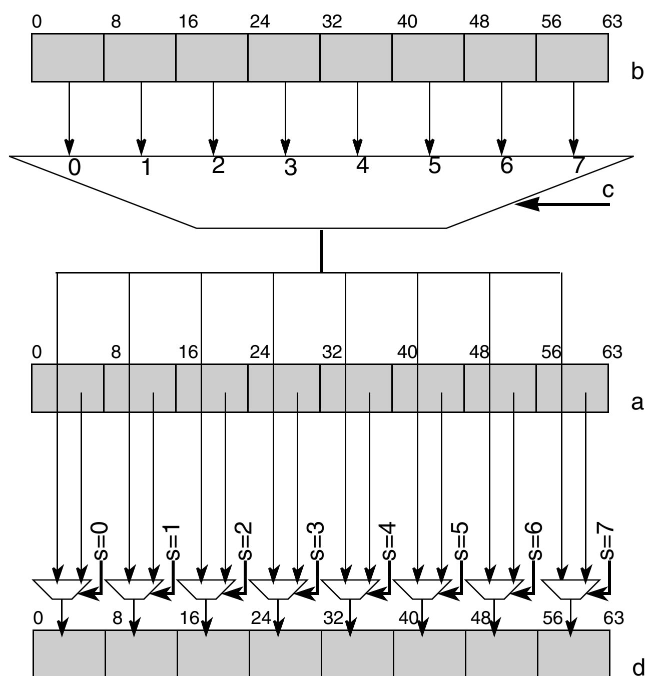


Figure 3-279. Vector Insert Byte (__ev_insb)

d	a	b	c	s	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	3-bit unsigned literal	3-bit unsigned literal	**d ←a** **evinsb d,b,s,c**

__ev_insh

__ev_insh

Vector Insert Half Word

d = __ev_insh (a,b,c,s)

```
temp0:15 ←b0+(c*16):15+(c*16)
if (s=0) then d0:15 ←temp0:15 else d0:15 ←a0:15
if (s=1) then d16:31 ←temp0:15 else d16:31 ←a16:31
if (s=2) then d32:47 ←temp0:15 else d32:47 ←a32:47
if (s=3) then d48:63 ←temp0:15 else d48:63 ←a48:63
```

The half word element of parameter **b** specified by parameter **c** is placed into the half word element of parameter **d** specified by parameter **s**. The remaining half words of parameter **d** are copied from parameter **a**. Half word 0 is the most-significant half word.

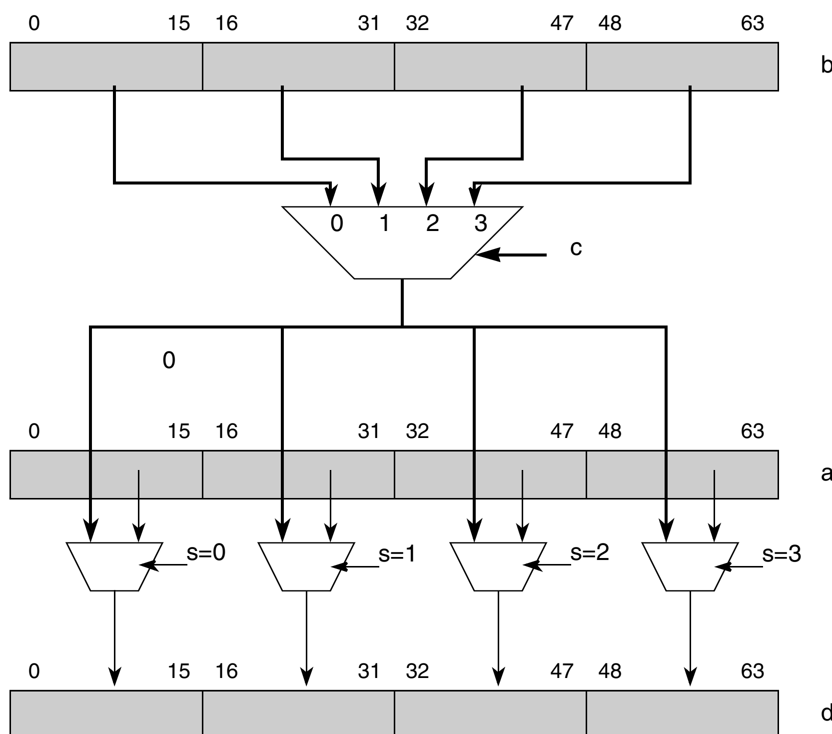


Figure 3-280. Vector Insert Half Word (__ev_insh)

d	a	b	c	s	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	2-bit unsigned literal	2-bit unsigned literal	**d ←a evinsh d,b,s,c**

__ev_ilveh

__ev_ilveh

Vector Interleave Even Half Words

d = __ev_ilveh (a,b)

$d_{0:63} \leftarrow a_{0:15} \,||\, b_{0:15} \,||\, a_{32:47} \,||\, b_{32:47}$

The even half word elements in parameter **a** are interleaved with the even half word elements in parameter **b** and placed into parameter **d**.

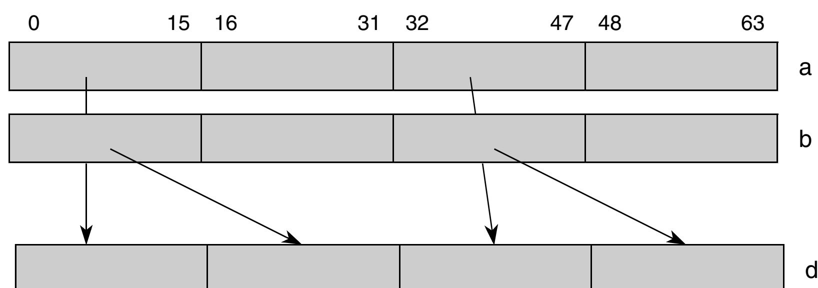


Figure 3-281. Vector Interleave Even Half Words (__ev_ilveh)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evilveh d,a,b**

__ev_ilveoh

__ev_ilveoh

Vector Interleave Even/Odd Half Words

d = __ev_ilveoh (a,b)

$d_{0:63} \leftarrow a_{0:15} \,||\, b_{0:15} \,||\, a_{32:47} \,||\, b_{32:47}$

The even half word elements in parameter **a** are interleaved with the odd half word elements in parameter **b** and placed into parameter **d**.

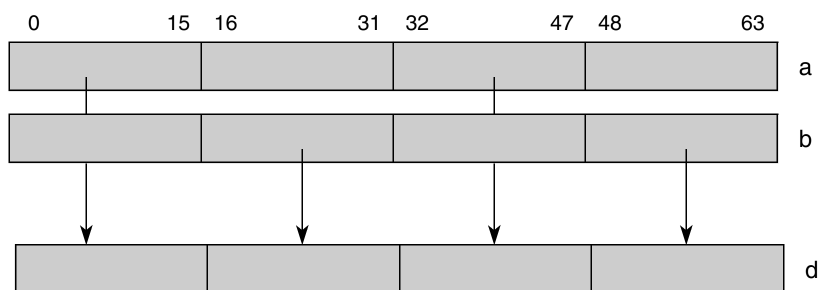


Figure 3-282. Vector Interleave Even/Odd Half Words (__ev_ilveoh)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evilveoh d,a,b**

__ev_ilvhih __ev_ilvhih

Vector Interleave High Half Words

d = __ev_ilvhih (a,b)

$d_{0:63} \leftarrow a_{0:15} \;||\; b_{0:15} \;||\; a_{16:31} \;||\; b_{16:31}$

The most significant two half word elements in parameter **a** are interleaved with the most significant two half word elements in parameter **b** and placed into parameter **d**.

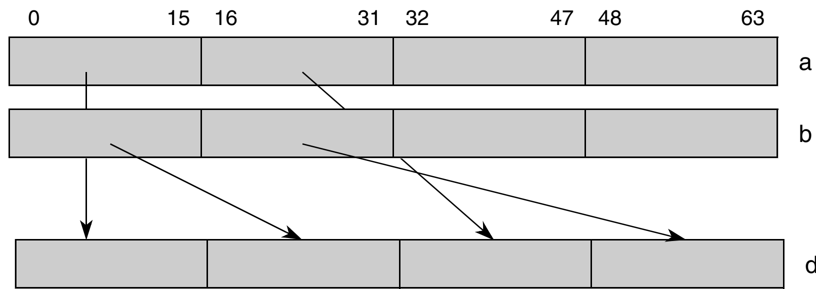


Figure 3-283. Vector Interleave High Half Words (__ev_ilvhih)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evilvhih d,a,b**

__ev_ilvhiloh

__ev_ilvhiloh

Vector Interleave High/Low Half Words

d = __ev_ilvhiloh (a,b)

$d_{0:63} \leftarrow a_{0:15} \mathbin{||} b_{32:47} \mathbin{||} a_{16:31} \mathbin{||} b_{48:63}$

The most significant two half word elements in parameter **a** are interleaved with the least significant two half word elements in parameter **b** and placed into parameter **d**.

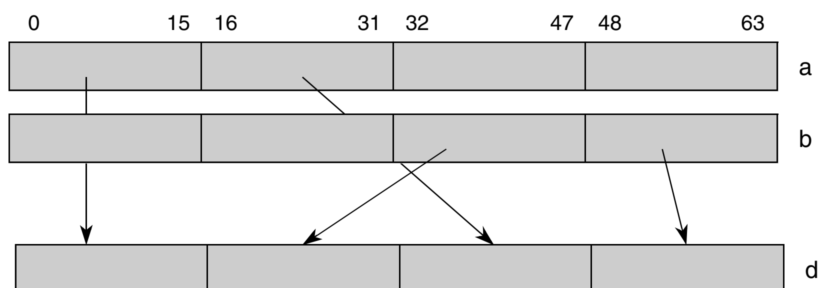


Figure 3-284. Vector Interleave High/Low Half Words (__ev_ilvhiloh)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evilvhiloh d,a,b**

__ev_ilvloh __ev_ilvloh

Vector Interleave Low Half Words

d = __ev_ilvloh (a,b)

```
d0:63 ←a32:47 || b32:47 || a48:63 || b48:63
```

The least significant two half word elements in parameter **a** are interleaved with the least significant two half word elements in parameter **b** and placed into parameter **d**.

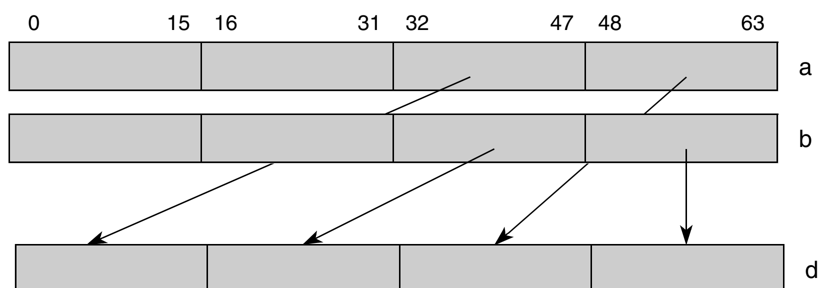


Figure 3-285. Vector Interleave Low Half Words (__ev_ilvloh)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evilvloh d,a,b**

__ev_ilvlohih

__ev_ilvlohih

Vector Interleave Low/High Half Words

d = __ev_ilvlohih (a,b)

$d_{0:63} \leftarrow a_{32:47} \parallel b_{0:15} \parallel a_{48:63} \parallel b_{16:31}$

The least significant two half word elements in parameter **a** are interleaved with the most significant two half word elements in parameter **b** and placed into parameter **d**.

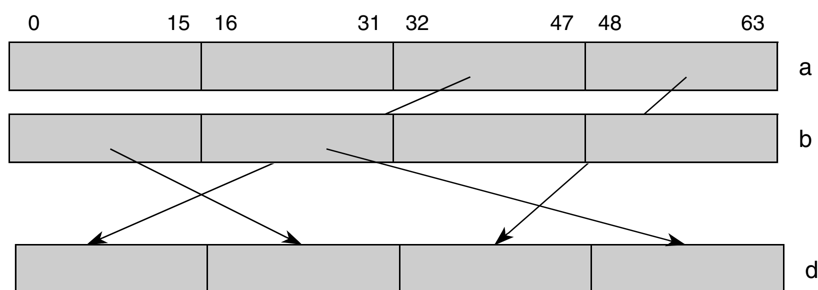


Figure 3-286. Vector Interleave Low/High Half Words (__ev_ilvlohih)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evilvlohih d,a,b**

__ev_ilvoeh

__ev_ilvoeh

Vector Interleave Odd/Even Half Words

d = __ev_ilvoeh (a,b)

$d_{0:63} \leftarrow a_{16:31} \,||\, b_{0:15} \,||\, a_{48:63} \,||\, b_{32:47}$

The odd half word elements in parameter **b** are interleaved with the even half word elements in parameter **b** and placed into parameter **d**.

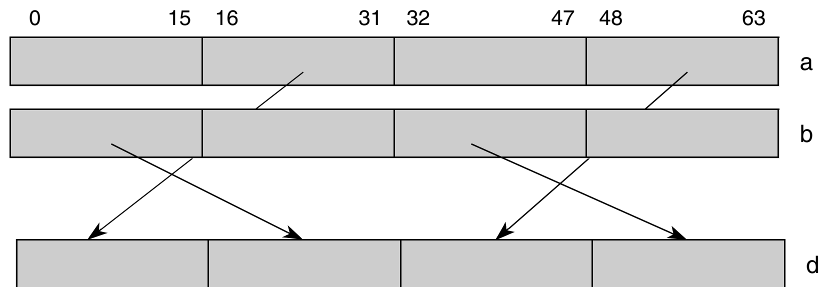


Figure 3-287. Vector Interleave Odd/Even Half Words (__ev_ilvoeh)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evilvoeh d,a,b**

__ev_ilvoh

__ev_ilvoh

Vector Interleave Odd Half Words

d = __ev_ilvoh (a,b)

$d_{0:63} \leftarrow a_{16:31} \, || \, b_{16:31} \, || \, a_{48:63} \, || \, b_{48:63}$

The odd half word elements in parameter **a** are interleaved with the odd half word elements in parameter **b** and placed into parameter **d**.

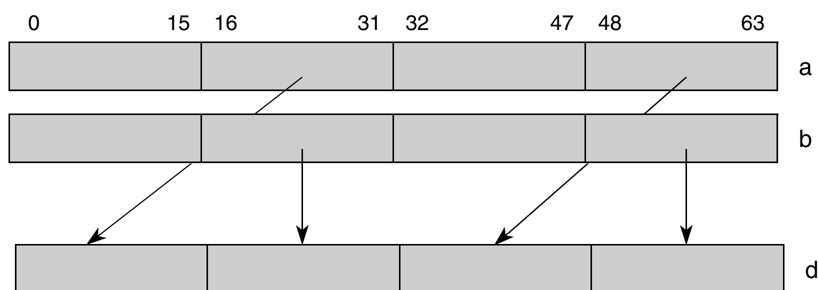


Figure 3-288. Vector Interleave Odd Half Words (__ev_ilvoh)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evilvoh d,a,b**

__ev_lbbsplatb[u] __ev_lbbsplatb[u]

Vector Load Byte into Byte and Splat Bytes [with Update]

d = __ev_lbbsplatb (a,b) (U = 0)

d = __ev_lbbsplatbu (a,b) (U = 1)

```
if (a = r0) then temp ← 0
else temp ← (a)
EA ← temp + EXTZ(b)
d0:7 ← MEM(EA,1)
d8:15 ← MEM(EA,1)
d16:23 ← MEM(EA,1)
d23:31 ← MEM(EA,1)
d32:39 ← MEM(EA,1)
d40:47 ← MEM(EA,1)
d48:55 ← MEM(EA,1)
d56:63 ← MEM(EA,1)

if (U=1) then a ← EA
```

The byte addressed by the EA is loaded from memory and placed into each byte element of parameter **d**.

If U=1 (‘with update’), EA is placed into parameter **a**.

Figure 3-289 shows how bytes are loaded into parameter **d** as determined by the endian mode.

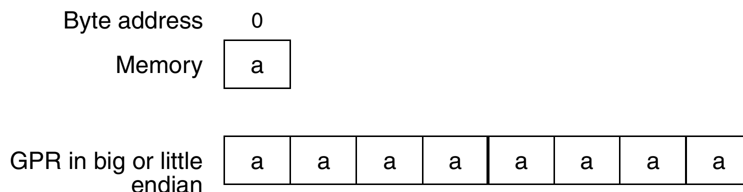


Figure 3-289. __ev_lbbsplatb[u] Results in Big- and Little-Endian Modes

NOTE

If machine register r0 is used to pass parameter **a**, then a zero value is used as the base value when calculating EA. For the ‘with update’ form (U=1), parameter **a** is both an input and output operand. Also, for the ‘with update’ form (U=1), parameter **b** cannot be 0.

U	d	a	b	Maps to
U = 0	__ev64_opaque__	uint8_t *	5-bit unsigned literal	**evlbbsplatb d,b(a)**
U = 1	__ev64_opaque__	uint8_t *&	5-bit unsigned literal	**evlbbsplatbu d,b(a)**

__ev_lbbsplatb[m]x __ev_lbbsplatb[m]x

Vector Load Byte into Byte and Splat Bytes [with Modify] Indexed

d = __ev_lbbsplatbx (a,b) (M = 0)

d = __ev_lbbsplatbmx (a,b) (M = 1)

```
if a=0 & M=1 then take_illegal_exception
if a=0 & M=0 then temp0 ←0
else temp0 ←(a)
EA ←temp0 + (b)
d0:7 ←MEM(EA,1)
d8:15 ←MEM(EA,1)
d16:23 ←MEM(EA,1)
d23:31 ←MEM(EA,1)
d32:39 ←MEM(EA,1)
d40:47 ←MEM(EA,1)
d48:55 ←MEM(EA,1)
d56:63 ←MEM(EA,1)

if (M=1) then a32:63 ←calc_a_update(a,b)
```

The byte addressed by EA is loaded from memory and placed into each byte element of parameter **d**.

If M=1 (‘with modify’), $\mathbf{a}_{32:63}$ is updated with an address value determined by the mode specifier in $\mathbf{a}_{0:3}$. See 3.2.3, “Addressing Modes - Modify forms.

Figure 3-290 shows how bytes are loaded into **d** as determined by the endian mode.

Byte address	0							
Memory	a							
GPR in big or little endian	a	a	a	a	a	a	a	a

Figure 3-290. __ev_lbbsplatb[m]x Results in Big- and Little-Endian Modes

NOTE

If machine register r0 is used to pass parameter **a**, then a zero value is used as the base value when calculating EA. Also, for the ‘with modify’ form (M=1), parameter **a** is both an input and output operand.

M	d	a	b	Maps to
M = 0	__ev64_opaque__	uint8_t *	int32_t	**evlbbsplatbx d,a,b**
M = 1	__ev64_opaque__	uint8_t *&	int32_t	**evlbbsplatbmx d,a,b**

__ev_ldb[u] __ev_ldb[u]

Vector Load Double into Eight Bytes [with Update]

d = __ev_ldb (a,b) (U = 0)

d = __ev_ldbu (a,b) (U = 1)

```
if (a = r0) then temp ← 0
else temp ← (a)
EA ← temp + EXTZ(b*8)
d0:7 ← MEM(EA, 1)
d8:15 ← MEM(EA+1,1)
d16:23 ← MEM(EA+2,1)
d24:31 ← MEM(EA+3,1)
d32:39 ← MEM(EA+4, 1)
d40:47 ← MEM(EA+5,1)
d48:55 ← MEM(EA+6,1)
d56:63 ← MEM(EA+7,1)

if (U=1) then a ← EA
```

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

If U=1 (‘with update’), EA is placed into parameter **a**.

Figure 3-291 shows how bytes are loaded into parameter **d** as determined by the endian mode.

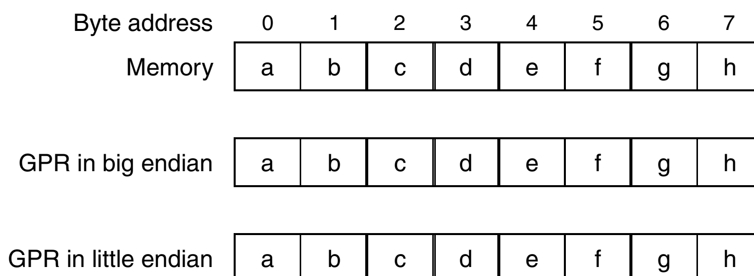


Figure 3-291. __ev_ldb[u] Results in Big- and Little-Endian Modes

NOTE

An alignment exception may occur if the effective address (EA) is not double-word aligned. Also, if machine register r0 is used to pass parameter **a**, then a zero value is used as the base value when calculating EA. For the ‘with update’ form (U=1), parameter **a** is both an input and output operand. Also, for the ‘with update’ form (U=1), parameter **b** cannot be 0.

U	d	a	b	Maps to
U = 0	__ev64_opaque__	__ev64_opaque__ *	5-bit unsigned literal	**evldb d,b(a)**
U = 1	__ev64_opaque__	__ev64_opaque__ *&	5-bit unsigned literal	**evldbu d,b(a)**

__ev_ldb[m]x

__ev_ldb[m]x

Vector Load Double into Eight Bytes [with Modify] Indexed

d = __ev_ldbx (a,b) (M = 0)

d = __ev_ldbmx (a,b) (M = 1)

```
if a=0 & M=1 then take_illegal_exception
if a=0 & M=0 then temp0 ←0
else temp0 ←(a)
EA ←temp0 + (b)
d0:7 ←MEM(EA,1)
d8:15 ←MEM(EA+1,1)*
d16:23 ←MEM(EA+2,1)*
d24:31 ←MEM(EA+3,1)*
d32:39 ←MEM(EA+4,1)*
d40:47 ←MEM(EA+5,1)*
d48:55 ←MEM(EA+6,1)*
d56:63 ←MEM(EA+7,1)*

if (M=1) then a32:63 ←calc_a_update(a,b)

        * - may wrap at length boundary for M=1 and mode 1000.
```

The doubleword addressed by EA is loaded from memory and placed into parameter **d**.

If M=1 (‘with modify’), $\mathbf{a}_{32:63}$ is updated with an address value determined by the mode specifier in $\mathbf{a}_{0:3}$. See 3.2.3, “Addressing Modes - Modify forms.

Figure 3-292 shows how bytes are loaded into parameter **d** as determined by the endian mode.

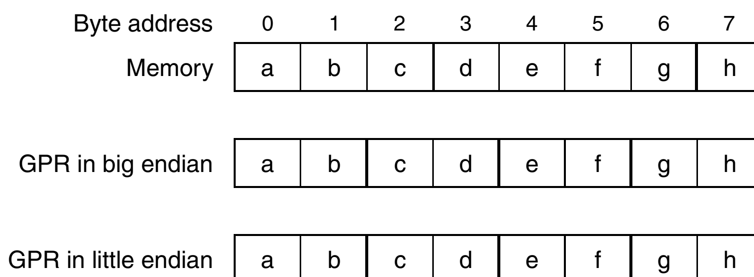


Figure 3-292. __ev_ldb[m]x Results in Big- and Little-Endian Modes

NOTE

An alignment exception may occur if the effective address (EA) is not double-word aligned. Also, if machine register r0 is used to pass parameter **a**, then a zero value is used as the base value when calculating EA. And, for the ‘with modify’ form (M=1), parameter **a** is both an input and output operand.

M	d	a	b	Maps to
M = 0	__ev64_opaque__	__ev64_opaque__ *	int32_t	**evldbx d,a,b**
M = 1	__ev64_opaque__	__ev64_opaque__ *&	int32_t	**evldbmx d,a,b**

__ev_ldd[u] __ev_ldd[u]

Vector Load Double Word into Double Word [with Update]

d = __ev_ldd (a,b) (U = 0)

d = __ev_lddu (a,b) (U = 1)

```
if (a = r0) then temp ← 0
else temp ← (a)
UIMM ← b
EA ← temp + EXTZ(UIMM*8)
d ← MEM(EA, 8)

if (U=1) then a ← EA
```

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

If U=1 ('with update'), EA is placed into parameter **a**.

Figure 3-293 shows how bytes are loaded into parameter **d** as determined by the endian mode.

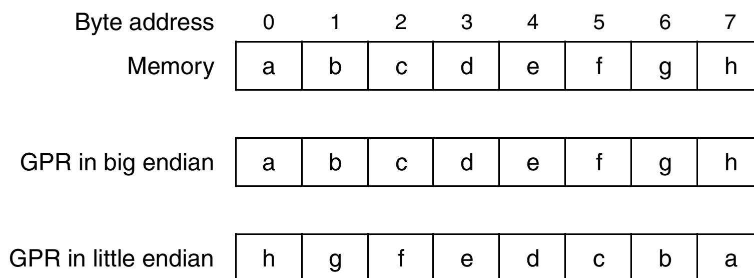


Figure 3-293. __ev_ldd[u] Results in Big- and Little-Endian Modes

NOTE

An alignment exception may occur if the effective address (EA) is not double-word aligned. Also, if machine register r0 is used to pass parameter **a**, then a zero value is used as the base value when calculating EA. For the 'with update' form (U=1), parameter **a** is both an input and output operand. Also, for the 'with update' form (U=1), parameter **b** cannot be 0.

U	d	a	b	Maps to
U = 0	__ev64_opaque__	__ev64_opaque__ *	5-bit unsigned literal	**evldd d,b(a)**
U = 1	__ev64_opaque__	__ev64_opaque__ *&	5-bit unsigned literal	**evlddu d,b(a)**

__ev_ldd[m]x __ev_ldd[m]x

Vector Load Double Word into Double Word [with Modify] Indexed

d = __ev_lddx (a,b) (M = 0)

d = __ev_lddmx (a,b) (M = 1)

```
if a=0 & M=1 then take_illegal_exception
if a=0 & M=0 then temp0 ←0
else temp0 ←(a)
EA ←temp0 + (b)
d ←MEM(EA,8)*

if (M=1) then a32:63 ←calc_a_update(a,b)

        * - may wrap at length boundary for M=1 and mode 1000.
```

The doubleword addressed by EA is loaded from memory and placed into parameter **d**.

If M=1 (‘with modify’), $\mathbf{a}_{32:63}$ is updated with an address value determined by the mode specifier in $\mathbf{a}_{0:3}$. See 3.2.3, “Addressing Modes - Modify forms.

Figure 3-294 shows how bytes are loaded into parameter **d** as determined by the endian mode.

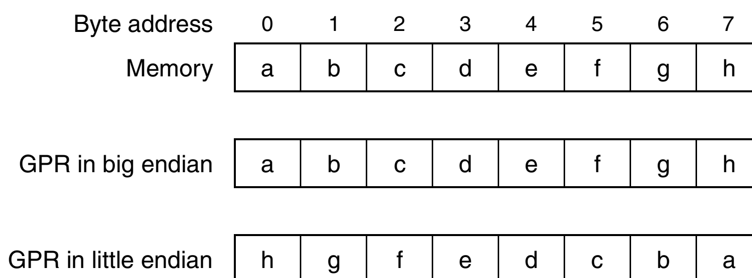


Figure 3-294. __ev_ldd[m]x Results in Big- and Little-Endian Modes

NOTE

An alignment exception may occur if the effective address (EA) is not double-word aligned. Also, if machine register r0 is used to pass parameter **a**, then a zero value is used as the base value when calculating EA. And, for the ‘with modify’ form (M=1), parameter **a** is both an input and output operand.

M	d	a	b	Maps to
M = 0	__ev64_opaque__	__ev64_opaque__*	int32_t	**evlddx d,a,b**
M = 1	__ev64_opaque__	__ev64_opaque__*&	int32_t	**evlddmx d,a,b**

__ev_ldh[u] __ev_ldh[u]

Vector Load Double into Four Half Words [with Update]

d = __ev_ldh (a,b) (U=0)

d = __ev_ldhu (a,b) (U=1)

```
if (a = r0) then temp ← 0
else temp ← (a)
UIMM ← b
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)

if (U=1) then a ← EA
```

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

If U=1 (‘with update’), EA is placed into parameter **a**.

Figure 3-295 shows how bytes are loaded into parameter **d** as determined by the endian mode.

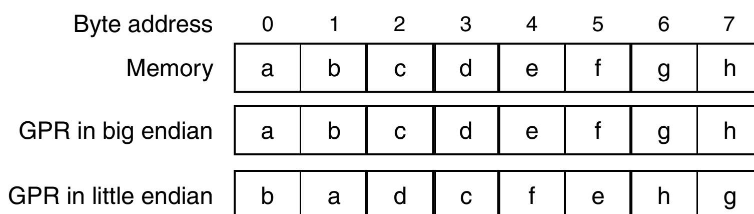


Figure 3-295. __ev_ldh[u] Results in Big- and Little-Endian Modes

NOTE

An alignment exception may occur if the effective address (EA) is not double-word aligned. Also, if machine register r0 is used to pass parameter **a**, then a zero value is used as the base value when calculating EA. For the ‘with update’ form (U=1), parameter **a** is both an input and output operand. Also, for the ‘with update’ form (U=1), parameter **b** cannot be 0.

U	d	a	b	Maps to
U = 0	__ev64_opaque__	__ev64_opaque__ *	5-bit unsigned literal	**evldh d,b(a)**
U = 1	__ev64_opaque__	__ev64_opaque__ *&	5-bit unsigned literal	**evldhu d,b(a)**

__ev_ldh[m]x __ev_ldh[m]x

Vector Load Double into Four Halfwords [with Modify] Indexed

d = __ev_ldhx (a,b) (M=0)

d = __ev_ldhmx (a,b) (M=1)

```
if a=0 & M=1 then take_illegal_exception
if a=0 & M=0 then temp0 ←0
else temp0 ←(a)
EA ←temp0 + (a)
d0:15 ←MEM(EA,2)*
d16:31 ←MEM(EA+2,2)*
d32:47 ←MEM(EA+4,2)*
d48:63 ←MEM(EA+6,2)*

if (M=1) then a32:63 ←calc_a_update(a,b)

        * - may wrap at length boundary for M=1 and mode 1000.
```

The doubleword addressed by EA is loaded from memory and placed into parameter **d**.

If M=1 (‘with modify’), $\mathbf{a}_{32:63}$ is updated with an address value determined by the mode specifier in $\mathbf{a}_{0:3}$. See 3.2.3, “Addressing Modes - Modify forms.

Figure 3-296 shows how bytes are loaded into parameter **d** as determined by the endian mode.

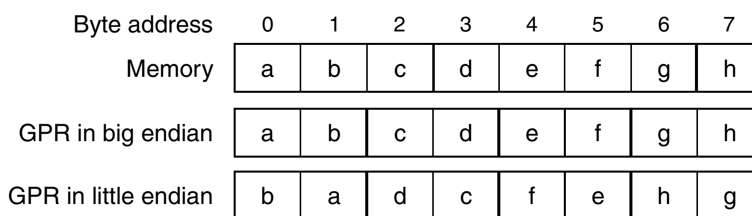


Figure 3-296. __ev_ldhx Results in Big- and Little-Endian Modes

NOTE

An alignment exception may occur if the effective address (EA) is not double-word aligned. Also, if machine register r0 is used to pass parameter **a**, then a zero value is used as the base value when calculating EA. And, for the ‘with modify’ form (M=1), parameter **a** is both an input and output operand.

M	d	a	b	Maps to
M = 0	__ev64_opaque__	__ev64_opaque__ *	int32_t	**evldhx d,a,b**
M = 1	__ev64_opaque__	__ev64_opaque__ *&	int32_t	**evldhmx d,a,b**

__ev_ldw[u] __ev_ldw[u]

Vector Load Double into Two Words [with Update]

d = __ev_ldw (a,b) (U=0)

d = __ev_ldwu (a,b) (U=1)

```
if (a = r0) then temp ← 0
else temp ← (a)
UIMM ← b
EA ← temp + EXTZ(UIMM*8)
d0:31 ← MEM(EA, 4)
d32:63 ← MEM(EA+4, 4)

if (U=1) then a ← EA
```

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

If U=1 (‘with update’), EA is placed into parameter **a**.

Figure 3-297 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 3-297. __ev_ldw[u] Results in Big- and Little-Endian Modes

NOTE

An alignment exception may occur if the effective address (EA) is not double-word aligned. Also, if machine register r0 is used to pass parameter **a**, then a zero value is used as the base value when calculating EA. For the ‘with update’ form (U=1), parameter **a** is both an input and output operand. Also, for the ‘with update’ form (U=1), parameter **b** cannot be 0.

U	d	a	b	Maps to
U = 0	__ev64_opaque__	__ev64_opaque__ *	5-bit unsigned literal	**evldw d,b(a)**
U = 1	__ev64_opaque__	__ev64_opaque__ *&	5-bit unsigned literal	**evldwu d,b(a)**

__ev_ldw[m]x __ev_ldw[m]x

Vector Load Double into Two Words [with Modify] Indexed

d = __ev_ldwx (a,b) (M=0)

d = __ev_ldwmx (a,b) (M=1)

```
if a=0 & M=1 then take_illegal_exception
if a=0 & M=0 then temp0 ←0
else temp0 ←(a)
EA ←temp0 + (b)
d0:31 ←MEM(EA,4)*
d32:63 ←MEM(EA+4,4)*

if (M=1) then a32:63 ←calc_a_update(a,b)

        * - may wrap at length boundary for M=1 and mode 1000.
```

The doubleword addressed by EA is loaded from memory and placed into parameter **d**.

If M=1 (‘with modify’), $\mathbf{a}_{32:63}$ is updated with an address value determined by the mode specifier in $\mathbf{a}_{0:3}$. See 3.2.3, “Addressing Modes - Modify forms.

Figure 3-298 shows how bytes are loaded into parameter **d** as determined by the endian mode.

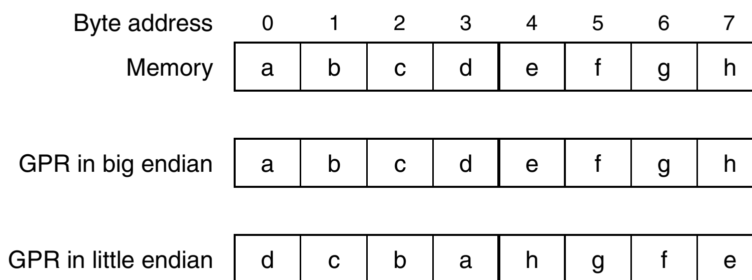


Figure 3-298. __ev_ldwx Results in Big- and Little-Endian Modes

NOTE

An alignment exception may occur if the effective address (EA) is not double-word aligned. Also, if machine register r0 is used to pass parameter **a**, then a zero value is used as the base value when calculating EA. And, for the ‘with modify’ form (M=1), parameter **a** is both an input and output operand.

M	d	a	b	Maps to
M = 0	__ev64_opaque__	__ev64_opaque__ *	int32_t	**evldwx d,a,b**
M = 1	__ev64_opaque__	__ev64_opaque__ *&	int32_t	**evldwmx d,a,b**

__ev_lhhesplat[u] __ev_lhhesplat[u]

Vector Load Half Word into Half Words Even and Splat [with Update]

d = __ev_lhhesplat (a,b) (U=0)

d = __ev_lhhesplatu (a,b) (U=1)

```
if (a = r0) then temp ← 0
else temp ← (a)
UIMM ← b
EA ← temp + EXTZ(UIMM*2)
d0:15 ← MEM(EA,2)
d16:31 ← 0x0000
d32:47 ← MEM(EA,2)
d48:63 ← 0x0000

if (U=1) then a ← EA
```

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

If U=1 (‘with update’), EA is placed into parameter **a**.

Figure 3-299 shows how bytes are loaded into parameter **d** as determined by the endian mode.

Byte address	0								
Memory	a	b							
GPR in big endian	a	b	Z	Z	a	b	Z	Z	Z = zero
GPR in little endian	b	a	Z	Z	b	a	Z	Z	Z = zero

Figure 3-299. __ev_lhhesplat[u] Results in Big- and Little-Endian Modes

NOTE

An alignment exception may occur if the effective address (EA) is not half-word aligned. Also, if machine register r0 is used to pass parameter **a**, then a zero value is used as the base value when calculating EA. For the ‘with update’ form (U=1), parameter **a** is both an input and output operand. Also, for the ‘with update’ form (U=1), parameter **b** cannot be 0.

U	d	a	b	Maps to
U = 0	__ev64_opaque__	uint16_t *	5-bit unsigned literal	**evlhhesplat d,b(a)**
U = 1	__ev64_opaque__	uint16_t *&	5-bit unsigned literal	**evlhhesplatu d,b(a)**

__ev_lhhesplat[m]x __ev_lhhesplat[m]x

Vector Load Halfword into Halfwords Even and Splat [with Modify] Indexed

d = __ev_lhhesplatx (a,b) (M=0)

d = __ev_lhhesplatmx (a,b) (M=1)

```
if a=0 & M=1 then take_illegal_exception
if a=0 & M=0 then temp0 ←0
else temp0 ←(a)
EA ←temp0 + (b)
d0:15 ←MEM(EA,2)*
d16:31 ←0x0000
d32:47 ←MEM(EA,2)*
d48:63 ←0x0000

if (M=1) then a32:63 ←calc_a_update(a,b)
        * - may wrap at length boundary for M=1 and mode 1000.
```

The halfword addressed by EA is loaded from memory and placed in the even halfwords of each word element of parameter **d**. The odd halfwords are zeroed.

If M=1 (‘with modify’), $\mathbf{a}_{32:63}$ is updated with an address value determined by the mode specifier in $\mathbf{a}_{0:3}$. See 3.2.3, “Addressing Modes - Modify forms.

Figure 3-300 shows how bytes are loaded into **d** as determined by the endian mode.

Byte address	0	1							
Memory	a	b							
GPR in big endian	a	b	Z	Z	a	b	Z	Z	Z = zero
GPR in little endian	b	a	Z	Z	b	a	Z	Z	Z = zero

Figure 3-300. __ev_lhhesplat[m]x Results in Big- and Little-Endian Modes

NOTE

An alignment exception may occur if the effective address (EA) is not half-word aligned. Also, if machine register r0 is used to pass parameter **a**, then a zero value is used as the base value when calculating EA. And, for the ‘with modify’ form (M=1), parameter **a** is both an input and output operand.

M	d	a	b	Maps to
M = 0	__ev64_opaque__	uint16_t *	int32_t	**evlhhesplatx d,a,b**
M = 1	__ev64_opaque__	uint16_t *&	int32_t	**evlhhesplatmx d,a,b**

__ev_lhhossplat[u] __ev_lhhossplat[u]

Vector Load Half Word into Half Word Odd Signed and Splat [with Update]

d = __ev_lhhossplat (a,b) (U=0)

d = __ev_lhhossplatu (a,b) (U=1)

```
if (a = r0) then temp ← 0
else temp ← (a)
UIMM ← b
EA ← temp + EXTZ(UIMM*2)
d0:31 ← EXTS(MEM(EA,2))
d32:63 ← EXTS(MEM(EA,2))

if (U=1) then a ← EA
```

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**.

If U=1 (‘with update’), EA is placed into parameter **a**.

Figure 3-301 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.

Byte address	0	1							
Memory	a	b							
GPR in big endian	S	S	a	b	S	S	a	b	S = sign
GPR in little endian	S	S	b	a	S	S	b	a	S = sign

Figure 3-301. __ev_lhhossplat[u] Results in Big- and Little-Endian Modes

NOTE

An alignment exception may occur if the effective address (EA) is not half-word aligned. Also, if machine register r0 is used to pass parameter **a**, then a zero value is used as the base value when calculating EA. For the ‘with update’ form (U=1), parameter **a** is both an input and output operand. Also, for the ‘with update’ form (U=1), parameter **b** cannot be 0.

U	d	a	b	Maps to
U = 0	__ev64_opaque__	uint16_t *	5-bit unsigned literal	**evlhhossplat d,b(a)**
U = 1	__ev64_opaque__	uint16_t *&	5-bit unsigned literal	**evlhhossplatu d,b(a)**

__ev_lhhossplat[m]x __ev_lhhossplat[m]x

Vector Load Halfword into Halfword Odd Signed and Splat [with Modify] Indexed

d = __ev_lhhossplatx (a,b) (M=0)

d = __ev_lhhossplatmx (a,b) (M=1)

```
if a=0 & M=1 then take_illegal_exception
if a=0 & M=0 then temp0 ←0
else temp0 ←(a)
EA ←temp0 + (b)
d0:31 ←EXTS32(MEM(EA,2)*)
d32:63 ←EXTS32(MEM(EA,2)*)

if (M=1) then a32:63 ←calc_a_update(a,b)

        * - may wrap at length boundary for M=1 and mode 1000.
```

The halfword addressed by EA is loaded from memory and placed in the odd halfwords, sign-extended into each word element of parameter **d**.

If M=1 (‘with modify’), parameter $\mathbf{a}_{32:63}$ is updated with an address value determined by the mode specifier in parameter $\mathbf{a}_{0:3}$. See 3.2.3, “Addressing Modes - Modify forms.

Figure 3-302 shows how bytes are loaded into parameter **d** as determined by the endian mode.

Byte address	0	1							
Memory	a	b							
GPR in big endian	S	S	a	b	S	S	a	b	S = sign
GPR in little endian	S	S	b	a	S	S	b	a	S = sign

Figure 3-302. __ev_lhhossplat[m]x Results in Big- and Little-Endian Modes

In big-endian memory, the most significant bit of **a** is sign extended. In little-endian memory, the most significant bit of parameter **b** is sign extended.

Implementation dependent note: If the EA is not halfword aligned, an alignment exception may occur.

NOTE

An alignment exception may occur if the effective address (EA) is not half-word aligned. Also, if machine register r0 is used to pass parameter **a**, then a zero value is used as the base value when calculating EA. And, for the ‘with modify’ form (M=1), parameter **a** is both an input and output operand.

M	d	a	b	Maps to
M = 0	__ev64_opaque__	uint16_t *	int32_t	**evlhhossplatx d,a,b**
M = 1	__ev64_opaque__	uint16_t *&	int32_t	**evlhhossplatmx d,a,b**

__ev_lhhousplat[u] __ev_lhhousplat[u]

Vector Load Half Word into Half Word Odd Unsigned and Splat [with Update]

d = __ev_lhhousplat (a,b) (U=0)

d = __ev_lhhousplatu (a,b) (U=1)

```
if (a = r0) then temp ← 0
else temp ← (a)
UIMM ← b
EA ← temp + EXTZ(UIMM*2)
d0:15 ← 0x0000
d16:31 ← MEM(EA,2)
d32:47 ← 0x0000
d48:63 ← MEM(EA,2)

if (U=1) then a ← EA
```

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**.

If U=1 (‘with update’), EA is placed into parameter **a**.

Figure 3-303 shows how bytes are loaded into parameter **d** as determined by the endian mode.

Byte address	0	1							
Memory	a	b							
GPR in big endian	Z	Z	a	b	Z	Z	a	b	Z = zero
GPR in little endian	Z	Z	b	a	Z	Z	b	a	Z = zero

Figure 3-303. __ev_lhhousplat[u] Results in Big- and Little-Endian Modes

NOTE

An alignment exception may occur if the effective address (EA) is not half-word aligned. Also, if machine register r0 is used to pass parameter **a**, then a zero value is used as the base value when calculating EA. For the ‘with update’ form (U=1), parameter **a** is both an input and output operand. Also, for the ‘with update’ form (U=1), parameter **b** cannot be 0.

U	d	a	b	Maps to
U = 0	__ev64_opaque__	uint16_t *	5-bit unsigned literal	**evlhhousplat d,b(a)**
U = 1	__ev64_opaque__	uint16_t *&	5-bit unsigned literal	**evlhhousplatu d,b(a)**

__ev_lhhousplat[m]x __ev_lhhousplat[m]x

Vector Load Halfword into Halfword Odd Unsigned and Splat [with Modify] Indexed

d = __ev_lhhousplatx (a,b) (M=0)

d = __ev_lhhousplatmx (a,b) (M=1)

```
if a=0 & M=1 then take_illegal_exception
if a=0 & M=0 then temp0 ←0
else temp0 ←(a)
EA ←temp0 + (b)
d0:15 ←0x0000
d16:31 ←MEM(EA,2)*
d32:47 ←0x0000
d48:63 ←MEM(EA,2)*

if (M=1) then a32:63 ←calc_a_update(a,b)

        * - may wrap at length boundary for M=1 and mode 1000.
```

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

If M=1 ('with modify'), $\mathbf{a}_{32:63}$ is updated with an address value determined by the mode specifier in $\mathbf{a}_{0:3}$. See 3.2.3, "Addressing Modes - Modify forms.

Figure 3-304 shows how bytes are loaded into parameter **d** as determined by the endian mode.

Byte address	0	1							
Memory	a	b							
GPR in big endian	Z	Z	a	b	Z	Z	a	b	Z = zero
GPR in little endian	Z	Z	b	a	Z	Z	b	a	Z = zero

Figure 3-304. __ev_lhhousplatx Results in Big- and Little-Endian Modes

NOTE

An alignment exception may occur if the effective address (EA) is not half-word aligned. Also, if machine register r0 is used to pass parameter **a**, then a zero value is used as the base value when calculating EA. And, for the 'with modify' form (M=1), parameter **a** is both an input and output operand.

M	d	a	b	Maps to
M = 0	__ev64_opaque__	uint16_t *	int32_t	**evlhhousplatx d,a,b**
M = 1	__ev64_opaque__	uint16_t *&	int32_t	**evlhhousplatmx d,a,b**

__ev_lhhsplath[u] __ev_lhhsplath[u]

Vector Load Half Word into Half Word and Splat Half Words [with Update]

d = __ev_lhhsplath (a,b) (U=0)

d = __ev_lhhsplathu (a,b) (U=1)

```
if (a = r0) then temp ← 0
else temp ← (a)
EA ←b + EXTZ(b*2)
d0:15 ←MEM(EA,2)
d16:31 ←MEM(EA,2)
d32:47 ←MEM(EA,2)
d48:63 ←MEM(EA,2)

if (U=1) then a ←EA
```

The half word addressed by EA is loaded from memory and placed into each half word element of parameter **d**.

If U=1 (‘with update’), EA is placed into parameter **a**.

Figure 3-305 shows how bytes are loaded into parameter **d** as determined by the endian mode.

Byte address	0	1						
Memory	a	b						
GPR in big endian	a	b	a	b	a	b	a	b
GPR in little endian	b	a	b	a	b	a	b	a

Figure 3-305. __ev_lhhsplath[u] Results in Big- and Little-Endian Modes

NOTE

An alignment exception may occur if the effective address (EA) is not half-word aligned. Also, if machine register r0 is used to pass parameter **a**, then a zero value is used as the base value when calculating EA. For the ‘with update’ form (U=1), parameter **a** is both an input and output operand. Also, for the ‘with update’ form (U=1), parameter **b** cannot be 0.

U	d	a	b	Maps to
U = 0	__ev64_opaque__	uint16_t *	5-bit unsigned literal	**evlhhsplath d,b(a)**
U = 1	__ev64_opaque__	uint16_t *&	5-bit unsigned literal	**evlhhsplathu d,b(a)**

__ev_lhhsplath[m]x __ev_lhhsplath[m]x

Vector Load Halfword into Halfword and Splat Halfwords [with Modify] Indexed

d = __ev_lhhsplathx (a,b) (M=0)

d = __ev_lhhsplathmx (a,b) (M=1)

```
if a=0 & M=1 then take_illegal_exception
if a=0 & M=0 then temp0 ←0
else temp0 ←(a)
EA ←temp0 + (b)
    d0:15 ←MEM(EA,2)*
d16:31 ←MEM(EA,2)*
d32:47 ←MEM(EA,2)*
d48:63 ←MEM(EA,2)*

if (M=1) then a32:63 ←calc_a_update(a,b)
        * - may wrap at length boundary for M=1 and mode 1000.
```

The halfword addressed by EA is loaded from memory and placed into each halfword element of parameter **d**.

If M=1 (‘with modify’), $\mathbf{a}_{32:63}$ is updated with an address value determined by the mode specifier in $\mathbf{a}_{0:3}$. See 3.2.3, “Addressing Modes - Modify forms.

Figure 3-306 shows how bytes are loaded into parameter **d** as determined by the endian mode.

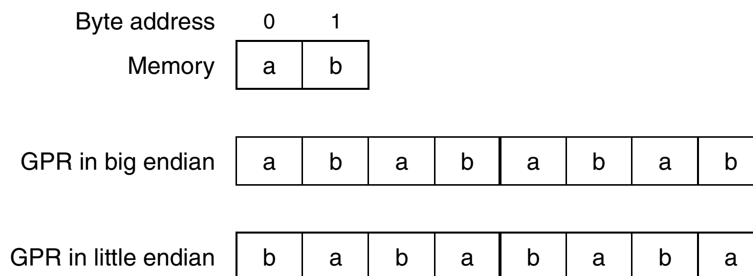


Figure 3-306. __ev_lhhsplathx Results in Big- and Little-Endian Modes

NOTE

An alignment exception may occur if the effective address (EA) is not half-word aligned. Also, if machine register r0 is used to pass parameter **a**, then a zero value is used as the base value when calculating EA. And, for the ‘with modify’ form (M=1), parameter **a** is both an input and output operand.

M	d	a	b	Maps to
M = 0	__ev64_opaque__	uint16_t *	int32_t	**evlhhsplathx d,a,b**
M = 1	__ev64_opaque__	uint16_t *&	int32_t	**evlhhsplathmx 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**.

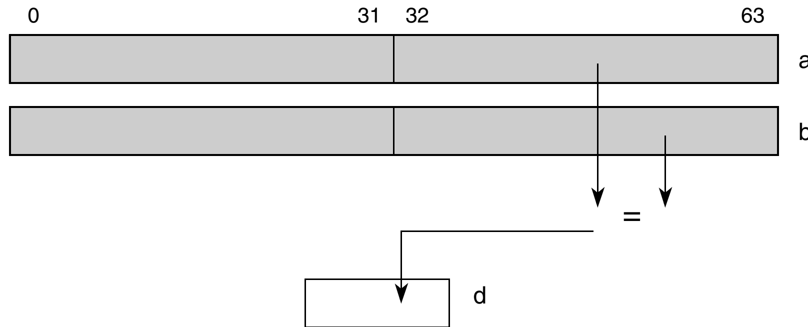


Figure 3-307. Vector Lower Equal (__ev_lower_eq)

d	a	b	Maps to
_Bool	__ev64_opaque__	__ev64_opaque__	**evcmpeq 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**.

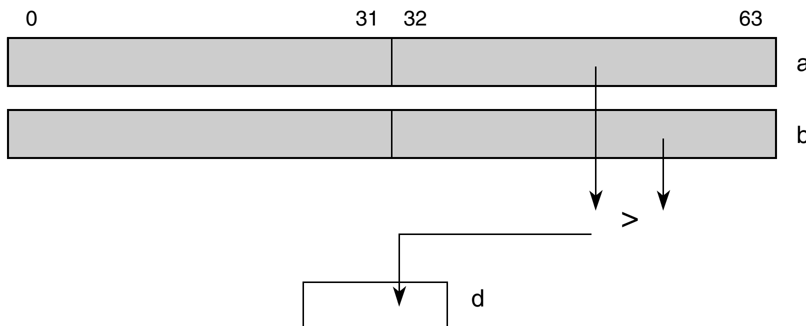


Figure 3-308. Vector Lower Greater Than Signed (__ev_lower_gts)

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**.

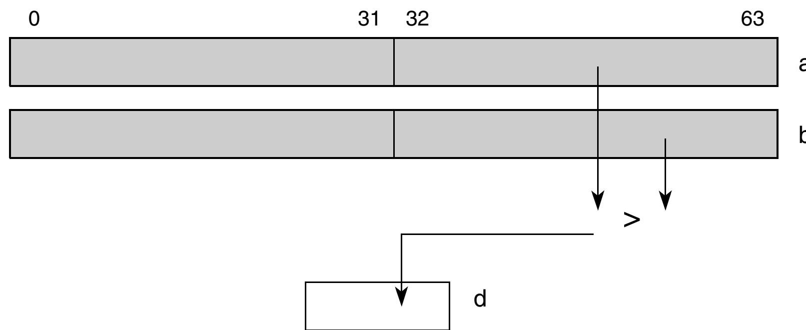


Figure 3-309. Vector Lower Greater Than Unsigned (__ev_lower_gtu)

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**.

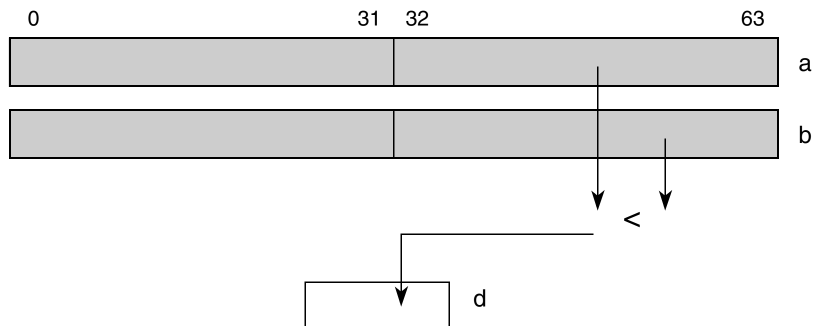


Figure 3-310. Vector Lower Less Than Signed (__ev_lower_lts)

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**.

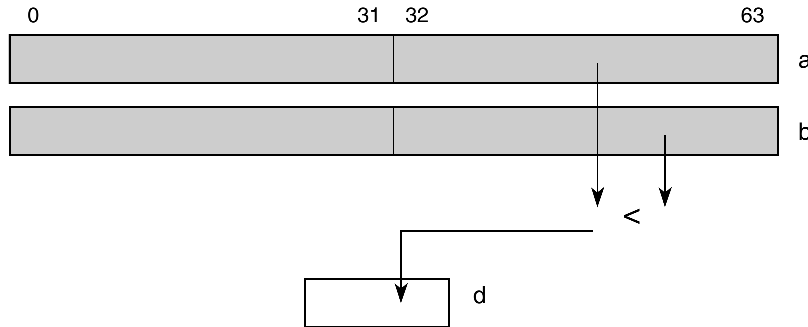


Figure 3-311. Vector Lower Less Than Unsigned (__ev_lower_ltu)

d	a	b	Maps to
_Bool	__ev64_opaque__	__ev64_opaque__	**evcmpltu x,a,b**

__ev_lvsl __ev_lvsl

Load Vector for Shift Left

d = __ev_lvsl (a,b)

```
if (a = 0) then temp ← 32 0
else temp ← (a32:63)
EA0:31 ← temp + (b32:63)
sh ← EA29:31

if (sh=0) then d0:63 ←0x0001020304050607
if (sh=1) then d0:63 ←0x0102030405060708
if (sh=2) then d0:63 ←0x0203040506070809
if (sh=3) then d0:63 ←0x030405060708090A
if (sh=4) then d0:63 ←0x0405060708090A0B
if (sh=5) then d0:63 ←0x05060708090A0B0C
if (sh=6) then d0:63 ←0x060708090A0B0C0D
if (sh=7) then d0:63 ←0x0708090A0B0C0D0E
```

The contents of parameters **a** and **b** are used to calculate an effective address. A control vector is calculated based on the byte offset of the EA and placed into parameter **d**. The control vector may be used by an **__ev_perm2** instruction to do a simulated alignment of a misaligned big-endian double word, or may be used for other computations such as a double length shift or rotate in conjunction with **__ev_lvsr**.

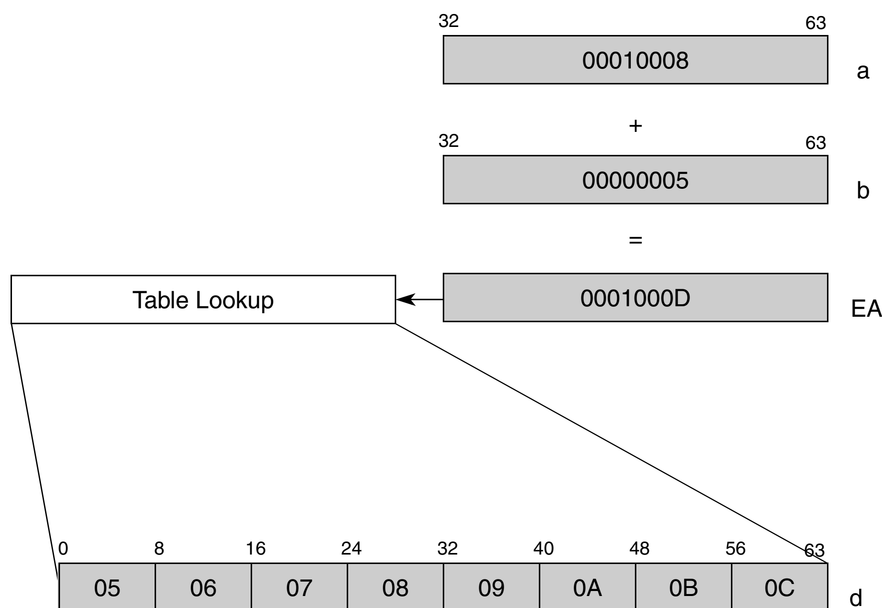


Figure 3-312. Load Vector for Shift Left (__ev_lvsl)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evlvsl d,a,b**

__ev_lvsr

__ev_lvsr

Load Vector for Shift Right

d = __ev_lvsr (a,b)

```
if (a = 0) then temp ← 32 0
else temp ← (a32:63)
EA0:31 ←temp + (b32:63)
sh ←EA29:31

if (sh=0) then d0:63 ←0x08090A0B0C0D0E0F
if (sh=1) then d0:63 ←0x0708090A0B0C0D0E
if (sh=2) then d0:63 ←0x060708090A0B0C0D
if (sh=3) then d0:63 ←0x05060708090A0B0C
if (sh=4) then d0:63 ←0x0405060708090A0B
if (sh=5) then d0:63 ←0x030405060708090A
if (sh=6) then d0:63 ←0x0203040506070809
if (sh=7) then d0:63 ←0x0102030405060708
```

The contents of parameters **a** and **b** are used to calculate an effective address. A control vector is calculated based on the byte offset of the EA and placed into parameter **d**. The control vector may be used by an **__ev_perm2** instruction to do a simulated alignment of a misaligned little-endian double word, or may be used for other computations such as a double length shift or rotate in conjunction with **__ev_lvsl**.

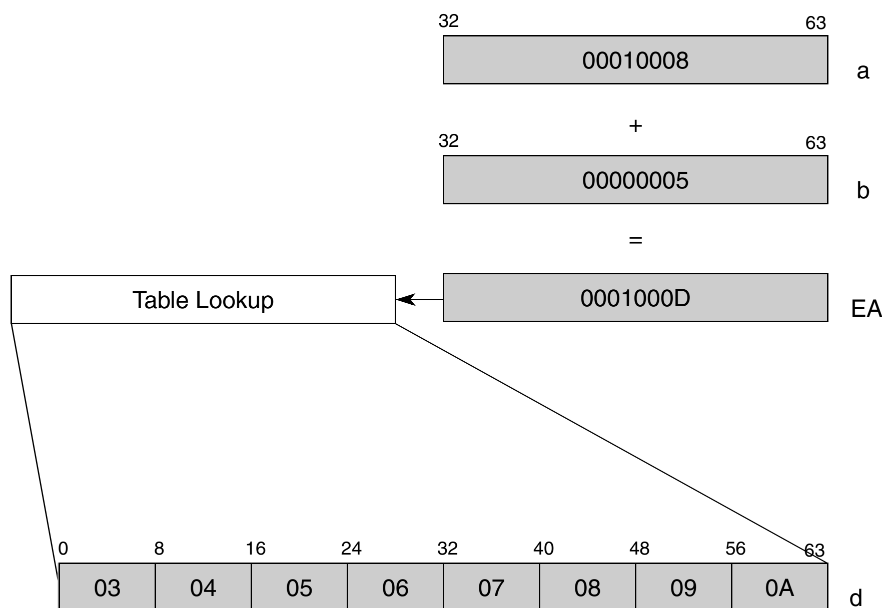


Figure 3-313. Load Vector for Shift Right (__ev_lvsr)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evlvsr d,a,b**

__ev_lwbe[u] __ev_lwbe[u]

Vector Load Word into Four Bytes Even [with Update]

d = __ev_lwbe (a,b) (U=0)

d = __ev_lwbeu (a,b) (U=1)

```
if (a = r0) then temp ← 0
else temp ← (a)
EA ← temp + EXTZ(b*4)
d0:7 ← MEM(EA, 1)
d8:15 ← 0x0000
d16:23 ← MEM(EA+1,1)
d24:31 ← 0x0000
d32:39 ← MEM(EA+2, 1)
d40:47 ← 0x0000
d48:55 ← MEM(EA+3,1)
d56:63 ← 0x0000

if (U=1) then a ← EA
```

The word addressed by EA is loaded from memory and placed into the even bytes of parameter **d**. The odd bytes are zeroed.

If U=1 (‘with update’), EA is placed into parameter **a**.

Figure 3-314 shows how bytes are loaded into parameter **d** as determined by the endian mode.

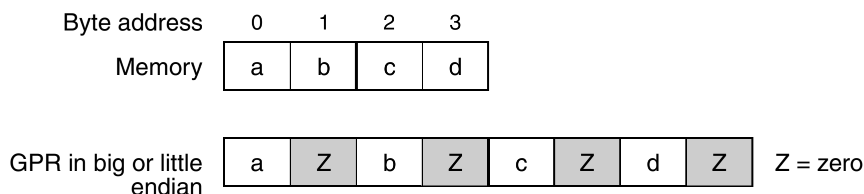


Figure 3-314. __ev_lwbe[u] Results in Big- and Little-Endian Modes

NOTE

An alignment exception may occur if the effective address (EA) is not word aligned. Also, if machine register r0 is used to pass parameter **a**, then a zero value is used as the base value when calculating EA. For the ‘with update’ form (U=1), parameter **a** is both an input and output operand. Also, for the ‘with update’ form (U=1), parameter **b** cannot be 0.

U	d	a	b	Maps to
U = 0	__ev64_opaque__	uint32_t *	5-bit unsigned literal	**evlwbe d,b(a)**
U = 1	__ev64_opaque__	uint32_t *&	5-bit unsigned literal	**evlwbeu d,b(a)**

__ev_lwbe[m]x __ev_lwbe[m]x

Vector Load Word into Four Bytes Even [with Modify] Indexed

d = __ev_lwbex (a,b) (M=0)

d = __ev_lwbemx (a,b) (M=1)

```
if a=0 & M=1 then take_illegal_exception
if a=0 & M=0 then temp ←0
else temp ←(a)
EA ←temp + (b)
d0:7 ←MEM(EA,1)
d8:15 ←0x0000
d16:23 ←MEM(EA+1,1)*
d24:31 ←0x0000
d32:39 ←MEM(EA+2,1)*
d40:47 ←0x0000
d48:55 ←MEM(EA+3,1)*
d56:63 ←0x0000

if (M=1) then a32:63 ←calc_a_update(a,b)

        * - may wrap at length boundary for M=1 and mode 1000.
```

The word addressed by EA is loaded from memory and placed in the even bytes in parameter **d**. The odd bytes are zeroed.

If M=1 (‘with modify’), parameter **a** is updated with an address value determined by the mode specifier in parameter **b**. See 3.2.3, “Addressing Modes - Modify forms.

Figure 3-315 shows how bytes are loaded into parameter **d** as determined by the endian mode.

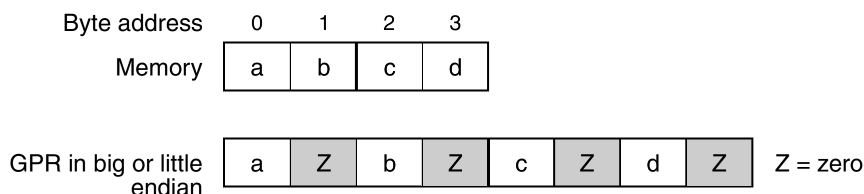


Figure 3-315. __ev_lwbe[u]x Results in Big- and Little-Endian Modes

NOTE

An alignment exception may occur if the effective address (EA) is not word aligned. Also, if machine register r0 is used to pass parameter **a**, then a zero value is used as the base value when calculating EA. And, for the ‘with modify’ form (M=1), parameter **a** is both an input and output operand.

U	d	a	b	Maps to
M = 0	__ev64_opaque__	uint32_t *	int32_t	**evlwbex d,a,b**
M = 1	__ev64_opaque__	uint32_t *&	int32_t	**evlwbemx d,a,b**

__ev_lwbos[u] __ev_lwbos[u]

Vector Load Word into Four Bytes Odd Signed [with Update]

d = __ev_lwbos (a,b) (U=0)

d = __ev_lwbosu (a,b) (U=1)

```
if (a = r0) then temp ← 0
else temp ← (a)
EA ← temp + EXTZ(b*4)
d0:15 ← EXTS(MEM(EA,1))
d16:31 ← EXTS(MEM(EA+1,1))
d32:47 ← EXTS(MEM(EA+2,1))
d48:63 ← EXTS(MEM(EA+3,1))

if (U=1) then a ← EA
```

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

If U=1 (‘with update’), EA is placed into parameter **a**.

Figure 3-316 shows how bytes are loaded into parameter **d** as determined by the endian mode.

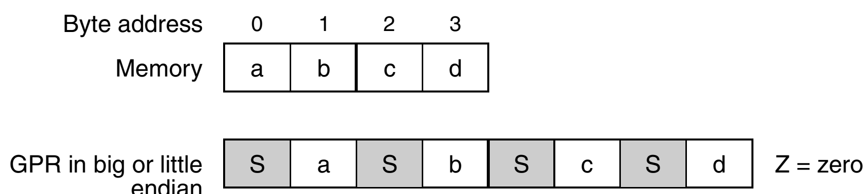


Figure 3-316. __ev_lwbos[u] Results in Big- and Little-Endian Modes

NOTE

An alignment exception may occur if the effective address (EA) is not word aligned. Also, if machine register r0 is used to pass parameter **a**, then a zero value is used as the base value when calculating EA. For the ‘with update’ form (U=1), parameter **a** is both an input and output operand. Also, for the ‘with update’ form (U=1), parameter **b** cannot be 0.

U	d	a	b	Maps to
U = 0	__ev64_opaque__	uint32_t *	5-bit unsigned literal	**evlwbos d,b(a)**
U = 1	__ev64_opaque__	uint32_t *&	5-bit unsigned literal	**evlwbosu d,b(a)**

__ev_lwbos[m]x __ev_lwbos[m]x

Vector Load Word into Four Bytes Odd Signed [with Modify] Indexed

d = __ev_lwbosx (a,b) (M=0)

d = __ev_lwbosmx (a,b) (M=1)

```
if a=0 & M=1 then take_illegal_exception
if a=0 & M=0 then temp ←0
else temp ←(a)
EA ←temp + (b)
d0:15 ←EXTS16(MEM(EA,1))
d16:31 ←EXTS16(MEM(EA+1,1)*)
d32:47 ←EXTS16(MEM(EA+2,1)*)
d48:63 ←EXTS16(MEM(EA+3,1)*)

if (M=1) then a32:63 ←calc_a_update(a,b)
        * - may wrap at length boundary for M=1 and mode 1000.
```

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

If M=1 (‘with modify’), parameter **a** is updated with an address value determined by the mode specifier in parameter **b**. See 3.2.3, “Addressing Modes - Modify forms.

Figure 3-317 shows how bytes are loaded into parameter **d** as determined by the endian mode.

Byte address	0	1	2	3					
Memory	a	b	c	d					
GPR in big or little endian	S	a	S	b	S	c	S	d	S = sign

Figure 3-317. __ev_lwbos[u]x Results in Big- and Little-Endian Modes

NOTE

An alignment exception may occur if the effective address (EA) is not word aligned. Also, if machine register r0 is used to pass parameter **a**, then a zero value is used as the base value when calculating EA. And, for the ‘with update’ form (U=1), parameter **a** is both an input and output operand.

U	d	a	b	Maps to
M = 0	__ev64_opaque__	uint32_t *	int32_t	**evlwbosx d,a,b**
M = 1	__ev64_opaque__	uint32_t *&	int32_t	**evlwbosmx d,a,b**

__ev_lwbou[u]   __ev_lwbou[u]

Vector Load Word into Four Bytes Odd Unsigned [with Update]

d = __ev_lwbou (a,b) (U=0)

d = __ev_lwbouu (a,b) (U=1)

```
if (a = r0) then temp ← 0
else temp ← (a)
EA ← temp + EXTZ(b*4)
d0:15 ← EXTZ(MEM(EA,1))
d16:31 ← EXTZ(MEM(EA+1,1))
d32:47 ← EXTZ(MEM(EA+2,1))
d48:63 ← EXTZ(MEM(EA+3,1))

if (U=1) then a ← EA
```

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

If U=1 (‘with update’), EA is placed into parameter **a**.

Figure 3-318 shows how bytes are loaded into parameter **d** as determined by the endian mode.

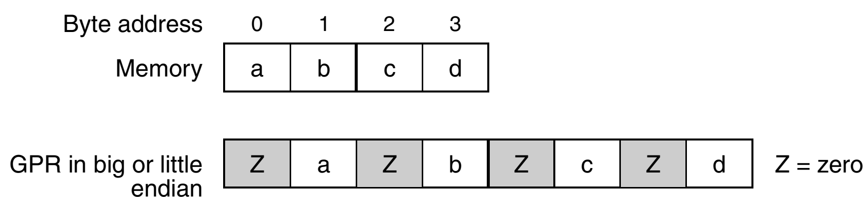


Figure 3-318. __ev_lwbou[u] Results in Big- and Little-Endian Modes

NOTE

An alignment exception may occur if the effective address (EA) is not word aligned. Also, if machine register r0 is used to pass parameter **a**, then a zero value is used as the base value when calculating EA. For the ‘with update’ form (U=1), parameter **a** is both an input and output operand. Also, for the ‘with update’ form (U=1), parameter **b** cannot be 0.

U	d	a	b	Maps to
U = 0	__ev64_opaque__	uint32_t *	5-bit unsigned literal	**evlwbou d,b(a)**
U = 1	__ev64_opaque__	uint32_t *&	5-bit unsigned literal	**evlwbouu d,b(a)**

__ev_lwbou[m]x __ev_lwbou[m]x

Vector Load Word into Four Bytes Odd Unsigned [with Modify] Indexed

d = __ev_lwboux (a,b) (M=0)

d = __ev_lwboumx (a,b) (M=1)

```
if a=0 & M=1 then take_illegal_exception
if a=0 & M=0 then temp ←0
else temp ←(a)
EA ←temp + (b)
d0:15 ←EXTZ16(MEM(EA,1))
d16:31 ←EXTZ16(MEM(EA+1,1)*)
d32:47 ←EXTZ16(MEM(EA+2,1)*)
d48:63 ←EXTZ16(MEM(EA+3,1)*)

if (M=1) then a32:63 ←calc_a_update(a,b)
        * - may wrap at length boundary for M=1 and mode 1000.
```

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

If M=1 (‘with modify’), parameter **a** is updated with an address value determined by the mode specifier in parameter **b**. See 3.2.3, “Addressing Modes - Modify forms.

Figure 3-319 shows how bytes are loaded into parameter **d** as determined by the endian mode.

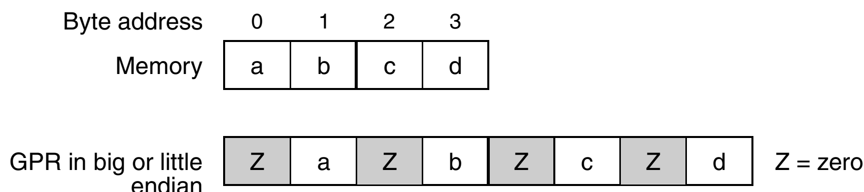


Figure 3-319. __ev_lwbou[m]x Results in Big- and Little-Endian Modes

NOTE

An alignment exception may occur if the effective address (EA) is not word aligned. Also, if machine register r0 is used to pass parameter **a**, then a zero value is used as the base value when calculating EA. And, for the ‘with modify’ form (M=1), parameter **a** is both an input and output operand.

U	d	a	b	Maps to
M = 0	__ev64_opaque__	uint32_t *	int32_t	**evlwboux d,a,b**
M = 1	__ev64_opaque__	uint32_t *&	int32_t	**evlwboumx d,a,b**

__ev_lwbsplatw[u] __ev_lwbsplatw[u]

Vector Load Word as Bytes and Splat Words [with Update]

d = __ev_lwbsplatw (a,b) (U=0)

d = __ev_lwbsplatwu (a,b) (U=1)

```
if (a = r0) then temp ← 0
else temp ← (a)
EA ← temp + EXTZ(b*4)
d0:7 ← MEM(EA,1)
d8:15 ← MEM(EA+1,1)
d16:23 ← MEM(EA+2,1)
d24:31 ← MEM(EA+3,1)
d32:39 ← MEM(EA,1)
d40:47 ← MEM(EA+1,1)
d48:55 ← MEM(EA+2,1)
d56:63 ← MEM(EA+3,1)
if U=1 then a ← EA
```

The word addressed by EA is loaded from memory as a quad of bytes and placed in both word elements of parameter **d**.

If U=1 (‘with update’), EA is placed into parameter **a**.

Figure 3-320 shows how bytes are loaded into parameter **d** as determined by the endian mode.

Byte address	0	1	2	3				
Memory	a	b	c	d				
GPR in big and little endian	a	b	c	d	a	b	c	d

Figure 3-320. __ev_lwbsplatw[u] Results in Big- and Little-Endian Modes

NOTE

An alignment exception may occur if the effective address (EA) is not word aligned. Also, if machine register r0 is used to pass parameter **a**, then a zero value is used as the base value when calculating EA. For the ‘with update’ form (U=1), parameter **a** is both an input and output operand. Also, for the ‘with update’ form (U=1), parameter **b** cannot be 0.

U	d	a	b	Maps to
U = 0	__ev64_opaque__	uint32_t *	5-bit unsigned literal	**evlwbsplatw d,b(a)**
U = 1	__ev64_opaque__	uint32_t *&	5-bit unsigned literal	**evlwbsplatwu d,b(a)**

__ev_lwbsplatw[m]x __ev_lwbsplatw[m]x

Vector Load Word as Bytes and Splat Words [with Modify] Indexed

d = __ev_lwbsplatwx (a,b) (M=0)

d = __ev_lwbsplatwmx (a,b) (M=1)

```
if a=0 & M=1 then take_illegal_exception
if a=0 & M=0 then temp ←0
else temp ←(a)
EA ←temp+ (b)
d0:7 ←MEM(EA,1)
d8:15 ←MEM(EA+1,1)*
d16:23 ←MEM(EA+2,1)*
d24:31 ←MEM(EA+3,1)*
d32:63 ←d0:31

if (M=1) then a32:63 ←calc_a_update(a,b)
         * - may wrap at length boundary for M=1 and mode 1000.
```

The word addressed by EA is loaded from memory as a quad of bytes and placed in both word elements of parameter **d**.

If U=1 (‘with modify’), parameter **a** is updated with an address value determined by the mode specifier in parameter **b**. See 3.2.3, “Addressing Modes - Modify forms.

Figure 3-321 shows how bytes are loaded into parameter **d** as determined by the endian mode.

Byte address	0	1	2	3				
Memory	a	b	c	d				
GPR in big and little endian	a	b	c	d	a	b	c	d

Figure 3-321. __ev_lwbsplatw[m]x Results in Big- and Little-Endian Modes

NOTE

An alignment exception may occur if the effective address (EA) is not word aligned. Also, if machine register r0 is used to pass parameter **a**, then a zero value is used as the base value when calculating EA. And, for the ‘with modify’ form (M=1), parameter **a** is both an input and output operand.

U	d	a	b	Maps to
M = 0	__ev64_opaque__	uint32_t *	int32_t	**evlwbsplatwx d,a,b**
M = 1	__ev64_opaque__	uint32_t *&	int32_t	**evlwbsplatwmx d,a,b**

__ev_lwhe[u] __ev_lwhe[u]

Vector Load Word into Two Half Words Even [with Update]

d = __ev_lwhe (a,b) (U=0)

d = __ev_lwheu (a,b) (U=1)

```
if (a = r0) then temp ← 0
else temp ← (a)
EA ← temp + EXTZ(b*4)
d0:15 ← MEM(EA,2)
d16:31 ← 0x0000
d32:47 ← MEM(EA+2,2)
d48:63 ← 0x0000

if (U=1) then a ← EA
```

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

If U=1 (‘with update’), EA is placed into parameter **a**.

Figure 3-322 shows how bytes are loaded into parameter **d** as determined by the endian mode.

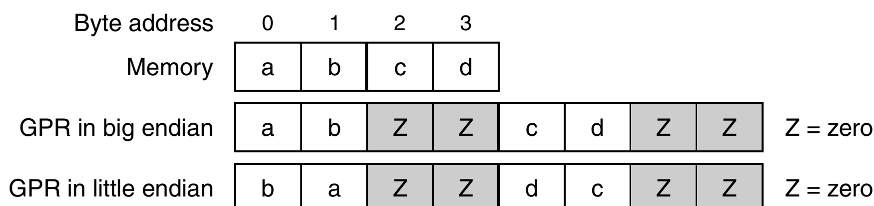


Figure 3-322. __ev_lwhe[u] Results in Big- and Little-Endian Modes

NOTE

An alignment exception may occur if the effective address (EA) is not word aligned. Also, if machine register r0 is used to pass parameter **a**, then a zero value is used as the base value when calculating EA. For the ‘with update’ form (U=1), parameter **a** is both an input and output operand. Also, for the ‘with update’ form (U=1), parameter **b** cannot be 0.

U	d	a	b	Maps to
U = 0	__ev64_opaque__	uint32_t *	5-bit unsigned literal	**evlwhe d,b(a)**
U = 1	__ev64_opaque__	uint32_t *&	5-bit unsigned literal	**evlwheu d,b(a)**

__ev_lwhe[m]x __ev_lwhe[m]x

Vector Load Word into Two Half Words Even [with Modify] Indexed

d = __ev_lwhex (a,b) (M=0)

d = __ev_lwhemx (a,b) (M=1)

```
if a=0 & M=1 then take_illegal_exception
if a=0 & M=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

if (M=1) then a32:63 ←calc_a_update(a,b)
        * - may wrap at length boundary for M=1 and mode 1000.
```

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

If U=1 (‘with modify’), parameter **a** is updated with an address value determined by the mode specifier in parameter **b**. See 3.2.3, “Addressing Modes - Modify forms.

Figure 3-323 shows how bytes are loaded into parameter **d** as determined by the endian mode.

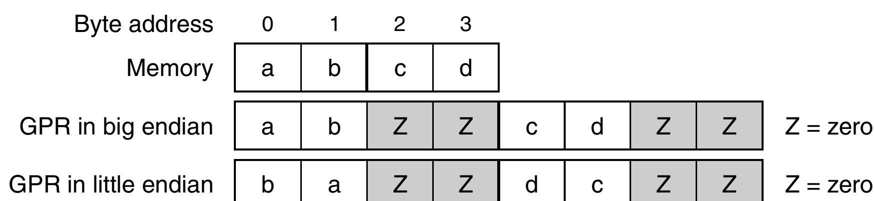


Figure 3-323. __ev_lwhe[m]x Results in Big- and Little-Endian Modes

NOTE

An alignment exception may occur if the effective address (EA) is not word aligned. Also, if machine register r0 is used to pass parameter **a**, then a zero value is used as the base value when calculating EA. And, for the ‘with modify’ form (M=1), parameter **a** is both an input and output operand.

U	d	a	b	Maps to
M = 0	__ev64_opaque__	uint32_t *	int32_t	**evlwhex d,a,b**
M = 1	__ev64_opaque__	uint32_t *&	int32_t	**evlwhemx d,a,b**

__ev_lwhos[u] __ev_lwhos[u]

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

d = __ev_lwhos (a,b) (U=0)

d = __ev_lwhosu (a,b) (U=1)

```
if (a = r0) then temp ← 0
else temp ← (a)
EA ← temp + EXTZ(b*4)
d0:31 ← EXTS(MEM(EA,2))
d32:63 ← EXTS(MEM(EA+2,2))

if (U=1) then a ← EA
```

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

Figure 3-324 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.

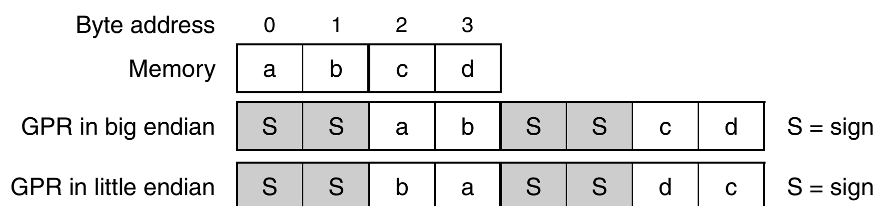


Figure 3-324. __ev_lwhos[u] Results in Big- and Little-Endian Modes

NOTE

An alignment exception may occur if the effective address (EA) is not word aligned. Also, if machine register r0 is used to pass parameter **a**, then a zero value is used as the base value when calculating EA. For the 'with update' form (U=1), parameter **a** is both an input and output operand. Also, for the 'with update' form (U=1), parameter **b** cannot be 0.

U	d	a	b	Maps to
U = 0	__ev64_opaque__	uint32_t *	5-bit unsigned literal	**evlwhos d,b(a)**
U = 1	__ev64_opaque__	uint32_t *&	5-bit unsigned literal	**evlwhosu d,b(a)**

__ev_lwhos[m]x __ev_lwhos[m]x

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

d = __ev_lwhosx (a,b) (M=0)

d = __ev_lwhosmx (a,b) (M=1)

```
if a=0 & M=1 then take_illegal_exception
if a=0 & M=0 then temp←0
else temp ←(a)
EA ←temp + (b)
d0:31 ←EXTS32(MEM(EA,2)*)
d32:63 ←EXTS32(MEM(EA+2,2)*)
if (M=1) then a32:63 ←calc_a_update(a,b)
        * - may wrap at length boundary for M=1 and mode 1000.
```

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

If M=1 (‘with modify’), parameter **a** is updated with an address value determined by the mode specifier in parameter **b**. See 3.2.3, “Addressing Modes - Modify forms.

Figure 3-325 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.

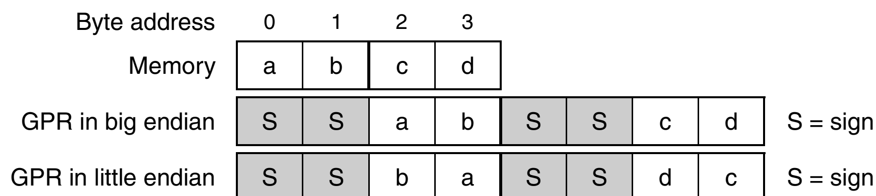


Figure 3-325. __ev_lwhos[m]x Results in Big- and Little-Endian Modes

NOTE

An alignment exception may occur if the effective address (EA) is not word aligned. Also, if machine register r0 is used to pass parameter **a**, then a zero value is used as the base value when calculating EA. And, for the ‘with modify’ form (M=1), parameter **a** is both an input and output operand.

U	d	a	b	Maps to
M = 0	__ev64_opaque__	uint32_t *	int32_t	**evlwhosx d,a,b**
M = 1	__ev64_opaque__	uint32_t *&	int32_t	**evlwhosmx d,a,b**

__ev_lwhou[u]   __ev_lwhou[u]

Vector Load Word into Two Half Words Odd Unsigned (zero-extended) [with Update]

d = __ev_lwhou (a,b) (U=0)

d = __ev_lwhouu (a,b) (U=1)

```
if (a = r0) then temp ← 0
else temp ← (a)
EA ← temp + EXTZ(b*4)
d0:15 ← 0x0000
d16:31 ← MEM(EA,2)
d32:47 ← 0x0000
d48:63 ← MEM(EA+2,2)

if (U=1) then a ← EA
```

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

If U=1 (‘with update’), EA is placed into parameter **a**.

Figure 3-326 shows how bytes are loaded into parameter **d** as determined by the endian mode.

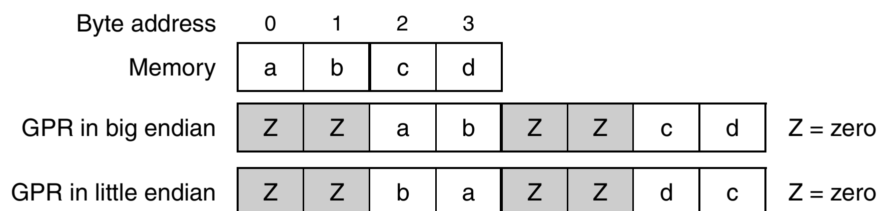


Figure 3-326. __ev_lwhou[u] Results in Big- and Little-Endian Modes

NOTE

An alignment exception may occur if the effective address (EA) is not word aligned. Also, if machine register r0 is used to pass parameter **a**, then a zero value is used as the base value when calculating EA. For the ‘with update’ form (U=1), parameter **a** is both an input and output operand. Also, for the ‘with update’ form (U=1), parameter **b** cannot be 0.

U	d	a	b	Maps to
U = 0	__ev64_opaque__	uint32_t *	5-bit unsigned literal	**evlwhou d,b(a)**
U = 1	__ev64_opaque__	uint32_t *&	5-bit unsigned literal	**evlwhouu d,b(a)**

__ev_lwhou[m]x __ev_lwhou[m]x

Vector Load Word into Two Half Words Odd Unsigned [with Modify] Indexed (zero-extended)

d = __ev_lwhoux (a,b) (M=0)

d = __ev_lwhoumx (a,b) (M=1)

```
if a=0 & M=1 then take_illegal_exception
if a=0 & M=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)*

if (M=1) then a32:63 ←calc_a_update(a,b)

        * - may wrap at length boundary for M=1 and mode 1000.
```

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

If M=1 (‘with modify’), parameter **a** is updated with an address value determined by the mode specifier in parameter **b**. See 3.2.3, “Addressing Modes - Modify forms.

Figure 3-327 shows how bytes are loaded into parameter **d** as determined by the endian mode.

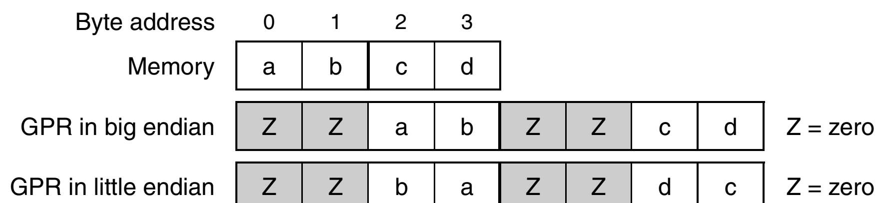


Figure 3-327. __ev_lwhou[m]x Results in Big- and Little-Endian Modes

NOTE

An alignment exception may occur if the effective address (EA) is not word aligned. Also, if machine register r0 is used to pass parameter **a**, then a zero value is used as the base value when calculating EA. And, for the ‘with modify’ form (M=1), parameter **a** is both an input and output operand.

U	d	a	b	Maps to
M = 0	__ev64_opaque__	uint32_t *	int32_t	**evlwhoux d,a,b**
M = 1	__ev64_opaque__	uint32_t *&	int32_t	**evlwhoumx d,a,b**

__ev_lwhsplat[u] __ev_lwhsplat[u]

Vector Load Word into Two Half Words and Splat [with Update]

d = __ev_lwhsplat (a,b) (U=0)

d = __ev_lwhsplatu (a,b) (U=1)

```
if (a = r0) then temp ← 0
else temp ← (a)
EA ← temp + EXTZ(b*4)
d0:15 ← MEM(EA,2)
d16:31 ← MEM(EA,2)
d32:47 ← MEM(EA+2,2)
d48:63 ← MEM(EA+2,2)

if U=1 then a ← EA
```

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

If U=1 ('with update'), EA is placed into parameter **a**.

Figure 3-328 shows how bytes are loaded into parameter **d** as determined by the endian mode.

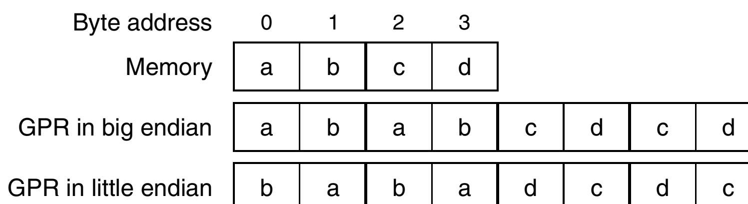


Figure 3-328. __ev_lwhsplat[u] Results in Big- and Little-Endian Modes

NOTE

An alignment exception may occur if the effective address (EA) is not word aligned. Also, if machine register r0 is used to pass parameter **a**, then a zero value is used as the base value when calculating EA. For the 'with update' form (U=1), parameter **a** is both an input and output operand. Also, for the 'with update' form (U=1), parameter **b** cannot be 0.

U	d	a	b	Maps to
U = 0	__ev64_opaque__	uint32_t *	5-bit unsigned literal	**evlwhsplat d,b(a)**
U = 1	__ev64_opaque__	uint32_t *&	5-bit unsigned literal	**evlwhsplatu d,b(a)**

__ev_lwhsplatw[u] __ev_lwhsplatw[u]

Vector Load Word as Half Words and Splat Words [with Update]

d = __ev_lwhsplatw (a,b) (U=0)

d = __ev_lwhsplatwu (a,b) (U=1)

```
if (a = r0) then temp ← 0
else temp ← (a)
EA ←b + EXTZ(b*4)
d0:15 ←MEM(EA,2)
d16:31 ←MEM(EA+2,2)
d32:47 ←MEM(EA,2)
d48:63 ←MEM(EA+2,2)

if (U=1) then a ←EA
```

The word addressed by EA is loaded from memory as a pair of half words and placed in both word elements of parameter **d**.

If U=1 (‘with update’), EA is placed into parameter **a**.

Figure 3-329 shows how bytes are loaded into parameter **d** as determined by the endian mode.

Byte address	0	1	2	3				
Memory	a	b	c	d				
GPR in big endian	a	b	c	d	a	b	c	d
GPR in little endian	b	a	d	c	b	a	d	c

Figure 3-329. __ev_lwhsplatw[u] Results in Big- and Little-Endian Modes

NOTE

An alignment exception may occur if the effective address (EA) is not word aligned. Also, if machine register r0 is used to pass parameter **a**, then a zero value is used as the base value when calculating EA. For the ‘with update’ form (U=1), parameter **a** is both an input and output operand. Also, for the ‘with update’ form (U=1), parameter **b** cannot be 0.

U	d	a	b	Maps to
U = 0	__ev64_opaque__	uint32_t *	5-bit unsigned literal	**evlwhsplatw d,b(a)**
U = 1	__ev64_opaque__	uint32_t *&	5-bit unsigned literal	**evlwhsplatwu d,b(a)**

__ev_lwhsplatw[m]x __ev_lwhsplatw[m]x

Vector Load Word as Half Words and Splat Words [with Modify] Indexed

d = __ev_lwhsplatwx (a,b) (M=0)

d = __ev_lwhsplatwmx (a,b) (M=1)

```
if a=0 & M=1 then take_illegal_exception
if a=0 & M=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,2)*
d48:63 ←MEM(EA+2,2)*

if (M=1) then a32:63 ←calc_a_update(a,b)
        * - may wrap at length boundary for M=1 and mode 1000.
```

The word addressed by EA is loaded from memory as a pair of half words and placed in both word elements of parameter **d**.

If M=1 ('with modify'), parameter **a** is updated with an address value determined by the mode specifier in parameter **b**. See 3.2.3, "Addressing Modes - Modify forms.

Figure 3-330 shows how bytes are loaded into parameter **d** as determined by the endian mode.

Byte address	0	1	2	3				
Memory	a	b	c	d				
GPR in big endian	a	b	c	d	a	b	c	d
GPR in little endian	b	a	d	c	b	a	d	c

Figure 3-330. __ev_lwhsplatw[m]x Results in Big- and Little-Endian Modes

NOTE

An alignment exception may occur if the effective address (EA) is not word aligned. Also, if machine register r0 is used to pass parameter **a**, then a zero value is used as the base value when calculating EA. And, for the 'with modify' form (M=1), parameter **a** is both an input and output operand.

U	d	a	b	Maps to
M = 0	__ev64_opaque__	uint32_t *	int32_t	**evlwhsplatwx d,a,b**
M = 1	__ev64_opaque__	uint32_t *&	int32_t	**evlwhsplatwmx d,a,b**

__ev_lwhsplat[m]x __ev_lwhsplat[m]x

Vector Load Word into Two Half Words and Splat [with Modify] Indexed

d = __ev_lwhsplatx (a,b) (M=0)

d = __ev_lwhsplatmx (a,b) (M=1)

```
if a=0 & M=1 then take_illegal_exception
if a=0 & M=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)*

if (M=1) then a32:63 ←calc_a_update(a,b)

        * - may wrap at length boundary for M=1 and mode 1000.
```

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

If M=1 (‘with modify’), parameter **a** is updated with an address value determined by the mode specifier in parameter **b**. See 3.2.3, “Addressing Modes - Modify forms.

Figure 3-331 shows how bytes are loaded into parameter **d** as determined by the endian mode.

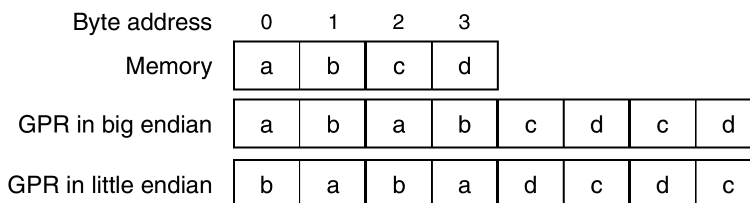


Figure 3-331. __ev_lwhsplat[m]x Results in Big- and Little-Endian Modes

NOTE

An alignment exception may occur if the effective address (EA) is not word aligned. Also, if machine register r0 is used to pass parameter **a**, then a zero value is used as the base value when calculating EA. And, for the ‘with modify’ form (M=1), parameter **a** is both an input and output operand.

U	d	a	b	Maps to
M = 0	__ev64_opaque__	uint32_t *	int32_t	**evlwhsplatx d,a,b**
M = 1	__ev64_opaque__	uint32_t *&	int32_t	**evlwhsplatmx d,a,b**

__ev_lwwsplat[u] __ev_lwwsplat[u]

Vector Load Word into Word and Splat [with Update]

d = __ev_lwwsplat (a,b) (U=0)

d = __ev_lwwsplatu (a,b) (U=1)

```
if (a = r0) then temp ← 0
else temp ← (a)
UIMM ← b
EA ← temp + EXTZ(UIMM*4)
d0:31 ← MEM(EA,4)
d32:63 ← MEM(EA,4)

if (U=1) then a ← EA
```

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

If U=1 (‘with update’), EA is placed into parameter **a**.

Figure 3-332 shows how bytes are loaded into parameter **d** as determined by the endian mode.

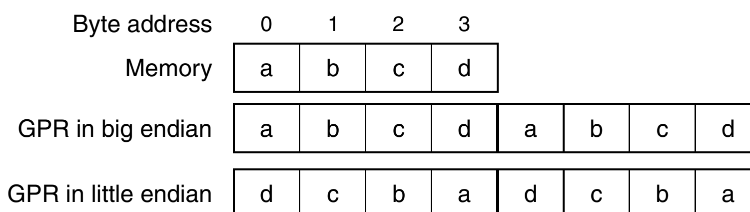


Figure 3-332. __ev_lwwsplat[u] Results in Big- and Little-Endian Modes

NOTE

An alignment exception may occur if the effective address (EA) is not word aligned. Also, if machine register r0 is used to pass parameter **a**, then a zero value is used as the base value when calculating EA. For the ‘with update’ form (U=1), parameter **a** is both an input and output operand. Also, for the ‘with update’ form (U=1), parameter **b** cannot be 0.

U	d	a	b	Maps to
U = 0	__ev64_opaque__	uint32_t *	5-bit unsigned literal	**evlwwsplat d,b(a)**
U = 1	__ev64_opaque__	uint32_t *&	5-bit unsigned literal	**evlwwsplatu d,b(a)**

__ev_lwwsplat[m]x __ev_lwwsplat[m]x

Vector Load Word into Word and Splat [with Modify] Indexed

d = __ev_lwwsplatx (a,b) (M=0)

d = __ev_lwwsplatmx (a,b) (M=1)

```
if a=0 & M=1 then take_illegal_exception
if a=0 & M=0 then temp ←0
else temp ←(a)
EA ←temp + (b)
d0:31 ←MEM(EA,4)*
d32:63 ←MEM(EA,4)*

if (M=1) then a32:63 ←calc_a_update(a,b)
        * - may wrap at length boundary for M=1 and mode 1000.
```

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

If M=1 (‘with modify’), parameter **a** is updated with an address value determined by the mode specifier in parameter **b**. See 3.2.3, “Addressing Modes - Modify forms.

Figure 3-333 shows how bytes are loaded into parameter **d** as determined by the endian mode.

Byte address	0	1	2	3				
Memory	a	b	c	d				
GPR in big endian	a	b	c	d	a	b	c	d
GPR in little endian	d	c	b	a	d	c	b	a

Figure 3-333. __ev_lwwsplat[m]x Results in Big- and Little-Endian Modes

NOTE

An alignment exception may occur if the effective address (EA) is not word aligned. Also, if machine register r0 is used to pass parameter **a**, then a zero value is used as the base value when calculating EA. And, for the ‘with modify’ form (M=1), parameter **a** is both an input and output operand.

U	d	a	b	Maps to
M = 0	__ev64_opaque__	uint32_t *	int32_t	**evlwwsplatx d,a,b**
M = 1	__ev64_opaque__	uint32_t *&	int32_t	**evlwwsplatmx d,a,b**

__ev_mar __ev_mar

Store Accumulator

d = __ev_mar ()

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

The contents of the accumulator are copied into parameter **d**. This is the method for saving the accumulator.

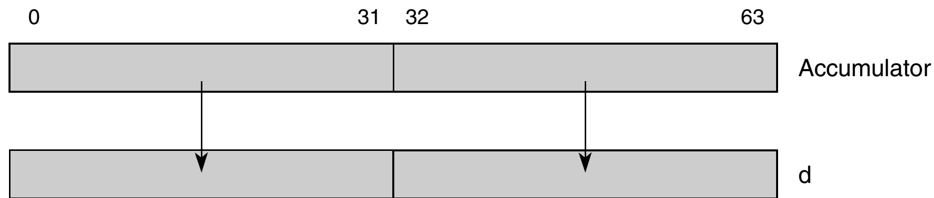


Figure 3-334. Store Accumulator (__ev_mar)

d	Maps to
__ev64_opaque__	**evmar d**

__ev_maxbpsh __ev_maxbpsh

Vector Maximum of Byte Pairs Signed to Half Word

d = __ev_maxbpsh (a)

$d_{0:15} \leftarrow EXTS(MAX_{si}(a_{0:7}, a_{8:15}))$
$d_{16:31} \leftarrow EXTS(MAX_{si}(a_{16:23}, a_{24:31}))$
$d_{32:47} \leftarrow EXTS(MAX_{si}(a_{32:39}, a_{40:47}))$
$d_{48:63} \leftarrow EXTS(MAX_{si}(a_{48:55}, a_{56:63}))$

Even/odd pairs of signed-integer byte elements of parameter **a** are compared and the larger of the two signed-integer byte elements is sign-extended and placed into the corresponding half word elements of parameter **d**.

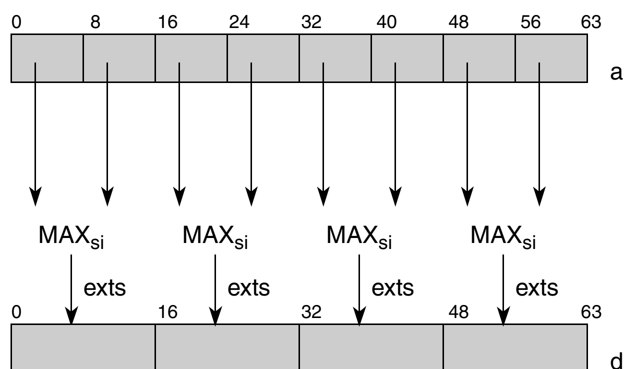


Figure 3-335. Vector Maximum of Byte Pairs Signed to Half Word (__ev_maxbpsh)

d	a	Maps to
__ev64_opaque__	__ev64_opaque__	**evmaxbpsh d,a**

__ev_maxbpuh __ev_maxbpuh

Vector Maximum of Byte Pairs Unsigned to Half Word

d = __ev_maxbpuh (a)

$d_{0:15} \leftarrow EXTZ(MAX_{si}(a_{0:7}, a_{8:15}))$
$d_{16:31} \leftarrow EXTZ(MAX_{si}(a_{16:23}, a_{24:31}))$
$d_{32:47} \leftarrow EXTZ(MAX_{si}(a_{32:39}, a_{40:47}))$
$d_{48:63} \leftarrow EXTZ(MAX_{si}(a_{48:55}, a_{56:63}))$

Even/odd pairs of unsigned-integer byte elements of parameter **a** are compared and the larger of the two unsigned-integer byte elements is zero-extended and placed into the corresponding half word element of parameter **d**.

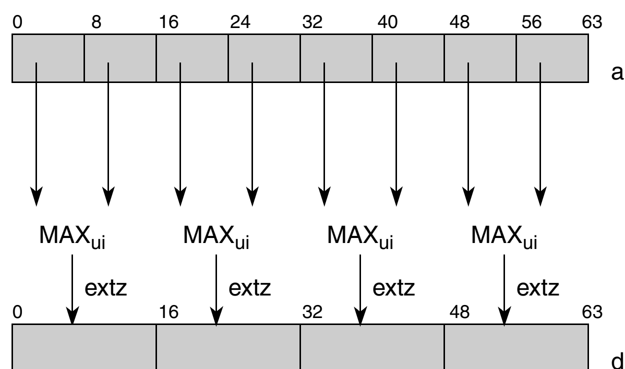


Figure 3-336. Vector Maximum of Byte Pairs Unsigned to Half Word (__ev_maxbpuh)

d	a	Maps to
__ev64_opaque__	__ev64_opaque__	**evmaxbpuh d,a**

__ev_maxbs __ev_maxbs

Vector Maximum Byte Signed

d = __ev_maxbs (a,b)

$d_{0:7} \leftarrow MAX_{si}(a_{0:7}, b_{0:7})$
$d_{8:15} \leftarrow MAX_{si}(a_{8:15}, b_{8:15})$
$d_{16:23} \leftarrow MAX_{si}(a_{16:23}, b_{16:23})$
$d_{24:31} \leftarrow MAX_{si}(a_{24:31}, b_{24:31})$
$d_{32:39} \leftarrow MAX_{si}(a_{32:39}, b_{32:39})$
$d_{40:47} \leftarrow MAX_{si}(a_{40:47}, b_{40:47})$
$d_{48:55} \leftarrow MAX_{si}(a_{48:55}, b_{48:55})$
$d_{56:63} \leftarrow MAX_{si}(a_{56:63}, b_{56:63})$

Each signed-integer byte element of parameter **a** is compared to the corresponding signed-integer byte element of parameter **b** and the larger of the two signed-integer elements is placed into the corresponding byte element of parameter **d**.

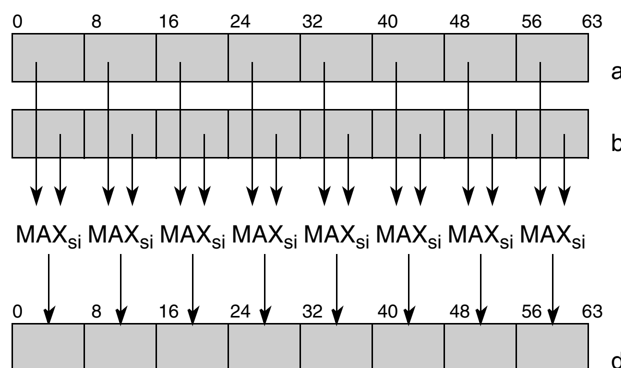


Figure 3-337. Vector Maximum Byte Signed (__ev_maxbs)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evmaxbs d,a,b**

__ev_maxbu __ev_maxbu

Vector Maximum Byte Unsigned

d = __ev_maxbu (a,b)

```
d0:7 ←MAXui(a0:7,b0:7)
d8:15 ←MAXui(a8:15,b8:15)
d16:23 ←MAXui(a16:23,b16:23)
d24:31 ←MAXui(a24:31,b24:31)
d32:39 ←MAXui(a32:39,b32:39)
d40:47 ←MAXui(a40:47,b40:47)
d48:55 ←MAXui(a48:55,b48:55)
d56:63 ←MAXui(a56:63,b56:63)
```

Each signed-integer byte element of parameter **a** is compared to the corresponding signed-integer byte element of parameter **b** and the larger of the two signed-integer elements is placed into the corresponding byte element of parameter **d**.

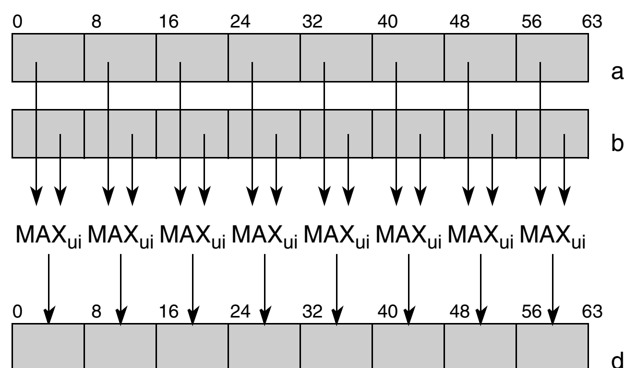


Figure 3-338. Vector Maximum Byte Unsigned (__ev_maxbu)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evmaxbu d,a,b**

__ev_maxds __ev_maxds

Vector Maximum Doubleword Signed

d = __ev_maxds (a,b)

$d_{0:63} \leftarrow MAX_{si}(a_{0:63}, b_{0:63})$

The signed-integer doubleword in parameter **a** is compared to the signed-integer doubleword in parameter **b** and the larger of the two doublewords is placed into parameter **d**.

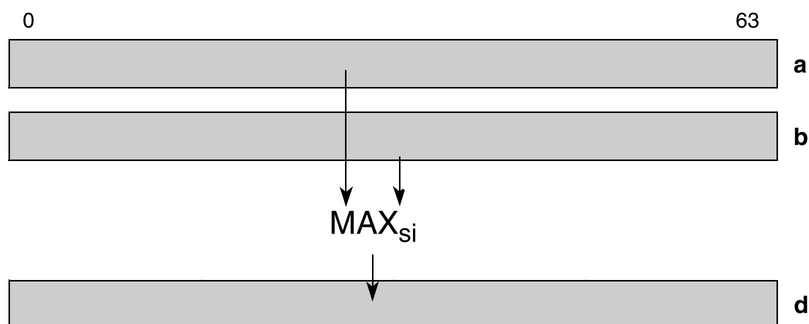


Figure 3-339. Vector Maximum Doubleword Signed (__ev_maxds)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evmaxds d,a,b**

__ev_maxdu __ev_maxdu

Vector Maximum Doubleword Unsigned

d = __ev_maxdu (a,b)

$d_{0:63} \leftarrow MAX_{ui}(a_{0:63}, b_{0:63})$

The unsigned-integer doubleword in parameter **a** is compared to the unsigned-integer doubleword in parameter **b** and the larger of the two doublewords is placed into parameter **d**.

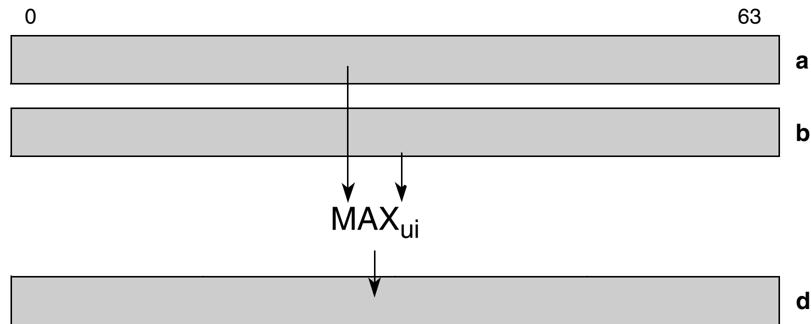


Figure 3-340. Vector Maximum Doubleword Unsigned (__ev_maxdu)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evmaxdu d,a,b**

__ev_maxhpsw

__ev_maxhpsw

Vector Maximum of Half Word Pairs Signed to Word

d = __ev_maxhpsw (a)

$d_{0:31} \leftarrow EXTS(MAX_{si}(a_{0:15}, a_{16:31}))$

$d_{32:63} \leftarrow EXTS(MAX_{si}(a_{32:47}, a_{48:63}))$

Even/odd pairs of signed-integer half word elements of parameter **a** are compared and the larger of the two signed-integer half word elements is sign-extended and placed into the corresponding word element of parameter **d**.

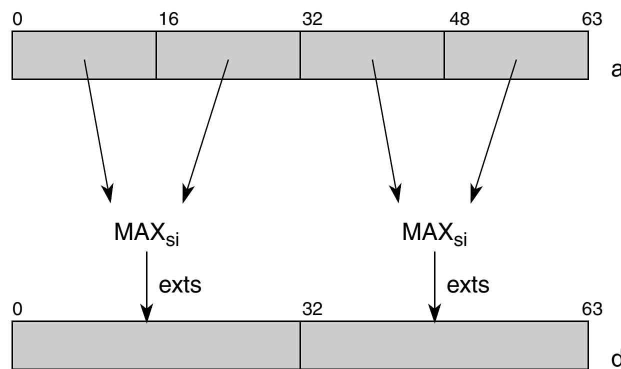


Figure 3-341. Vector Maximum of Half Word Pairs Signed to Word (__ev_maxhpsw)

d	a	Maps to
__ev64_opaque__	__ev64_opaque__	**evmaxhpsw d,a**

__ev_maxhpuw __ev_maxhpuw

Vector Maximum of Half Word Pairs Unsigned to Word

d = __ev_maxhpuw (a)

$d_{0:31} \leftarrow EXTZ(MAX_{si}(a_{0:15}, a_{16:31}))$

$d_{32:63} \leftarrow EXTZ(MAX_{si}(a_{32:47}, a_{48:63}))$

Even/odd pairs of unsigned-integer half word elements of parameter **a** are compared and the larger of the two unsigned-integer half word elements is zero-extended and placed into the corresponding word element of parameter **d**.

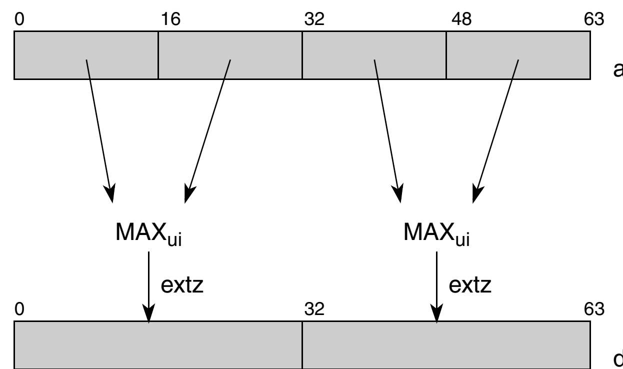


Figure 3-342. Vector Maximum of Half Word Pairs Unsigned to Word (__ev_maxhpuw)

d	a	Maps to
__ev64_opaque__	__ev64_opaque__	**evmaxhpuw d,a**

__ev_maxhs

Vector Maximum Half Word Signed

d = __ev_maxhs (a,b)

$d_{0:15} \leftarrow MAX_{si}(a_{0:15}, b_{0:15})$
$d_{16:31} \leftarrow MAX_{si}(a_{16:31}, b_{16:31})$
$d_{32:47} \leftarrow MAX_{si}(a_{32:47}, b_{32:47})$
$d_{48:63} \leftarrow MAX_{si}(a_{48:63}, b_{48:63})$

Each signed-integer half word element of parameter **a** is compared to the corresponding signed-integer half word element of parameter **b** and the larger of the two signed-integer elements is placed into the corresponding half word element of parameter **d**.

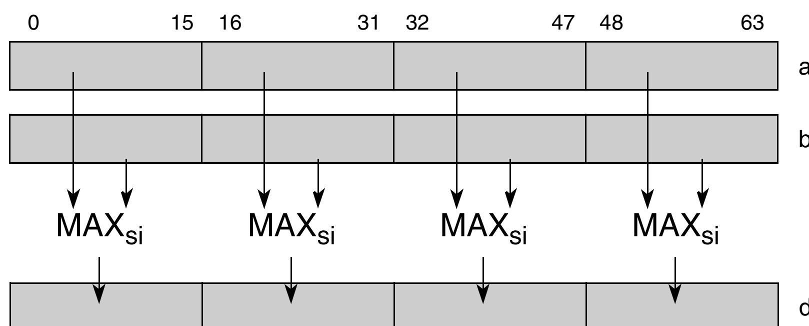


Figure 3-343. Vector Maximum Half Word Signed (__ev_maxhs)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evmaxhs d,a,b**

__ev_maxhu __ev_maxhu

Vector Maximum Half Word Unsigned

d = __ev_maxhu (a,b)

$d_{0:15} \leftarrow MAX_{ui}(a_{0:15}, b_{0:15})$
$d_{16:31} \leftarrow MAX_{ui}(a_{16:31}, b_{16:31})$
$d_{32:47} \leftarrow MAX_{ui}(a_{32:47}, b_{32:47})$
$d_{48:63} \leftarrow MAX_{ui}(a_{48:63}, b_{48:63})$

Each unsigned-integer half word element of parameter **a** is compared to the corresponding unsigned-integer half word element of parameter **b** and the larger of the two unsigned-integer elements is placed into the corresponding half word element of parameter **d**.

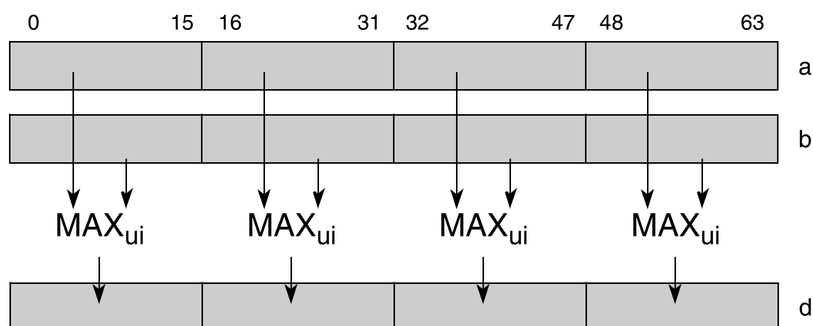


Figure 3-344. Vector Maximum Half Word Unsigned (__ev_maxhu)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evmaxhu d,a,b**

__ev_maxmagws __ev_maxmagws

Vector Maximum Magnitude Word Signed

d = __ev_maxmagws (a,b)

```
if ((ABS(a0:31) > ABS(b0:31)) | ((a0:31>0) & (ABS(a0:31) = ABS(b0:31)))) then
    temph10:31 ←a0:31
else
    temph10:31 ←b0:31
endif
if ((ABS(a32:63) > ABS(b32:63)) | ((a32:63>0) & (ABS(a32:63) = ABS(b32:63)))) then
    templ10:31 ←a32:63
else
    templ10:31 ←b32:63
endif
d0:31 ←temph10:31; d32:63 ←templ10:31
```

The magnitude of each signed-integer word element of parameter **a** is compared to the magnitude of the corresponding signed-integer word element of parameter **b**. The word element from parameter **a** or **b** with the larger magnitude is placed into the corresponding word element of parameter **d**. If the magnitudes of parameters **a** and **b** are equal, a positive value is selected if either word element is positive.

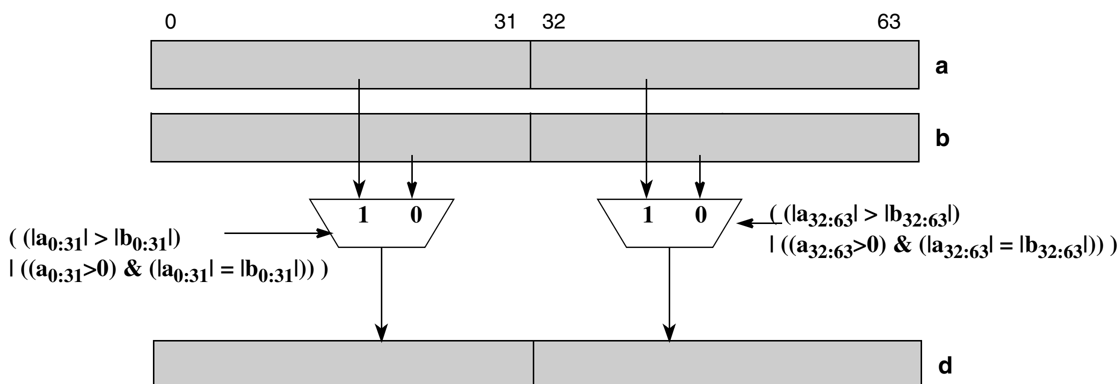


Figure 3-345. Vector Maximum Magnitude Word Signed (__ev_maxmagws)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evmaxmagws d,a,b**

__ev_maxwpsd __ev_maxwpsd

Vector Maximum of Word Pair Signed to Double Word

d = __ev_maxwpsd (a)

$d_{0:63} \leftarrow EXTS(MAX_{si}(a_{0:31}, a_{32:63}))$

The signed-integer word elements of parameter **a** are compared and the larger of the two signed-integer word elements is sign-extended and placed into parameter **d**.

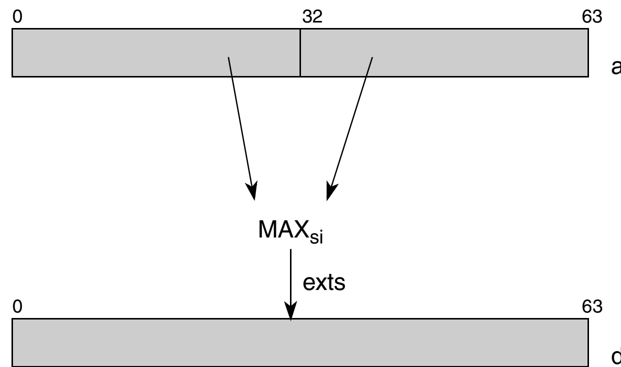


Figure 3-346. Vector Maximum of Word Pairs Signed to Doubleword (__ev_maxwpsd)

d	a	Maps to
__ev64_opaque__	__ev64_opaque__	**evmaxwpsd d,a**

__ev_maxwpud __ev_maxwpud

Vector Maximum of Word Pair Unsigned to Double Word

d = __ev_maxwpud (a)

$d_{0:63} \leftarrow EXTZ(MAX_{ui}(a_{0:31}, a_{32:63}))$

The unsigned-integer word elements of parameter **a** are compared and the larger of the two unsigned-integer word elements is zero-extended and placed into parameter **d**.

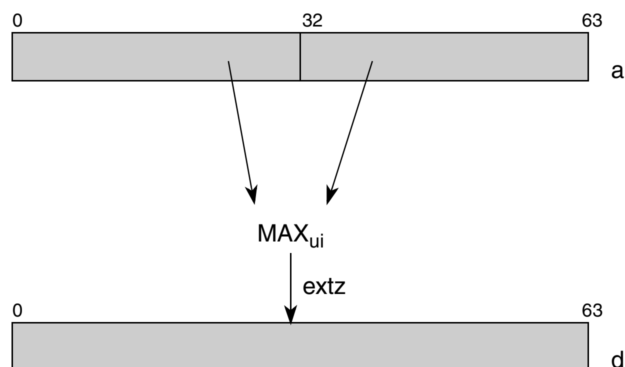


Figure 3-347. Vector Maximum of Word Pairs Unsigned to Double Word (__ev_maxwpud)

d	a	Maps to
__ev64_opaque__	__ev64_opaque__	**evmaxwpud d,a**

__ev_maxws

Vector Maximum Word Signed

d = __ev_maxws (a,b)

$d_{0:31} \leftarrow MAX_{si}(a_{0:31}, b_{0:31})$

$d_{32:63} \leftarrow MAX_{si}(a_{32:63}, b_{32:63})$

Each signed-integer word element of parameter **a** is compared to the corresponding signed-integer word element of parameter **b** and the larger of the two signed-integer elements is placed into the corresponding word element of parameter **d**.

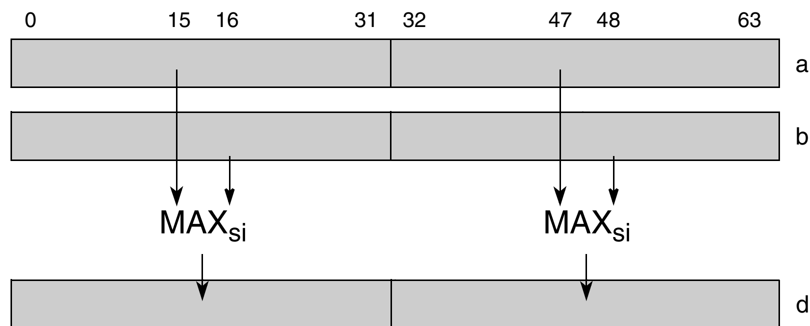


Figure 3-348. Vector Maximum Word Signed (__ev_maxws)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evmaxws d,a,b**

__ev_maxwu __ev_maxwu

Vector Maximum Word Unsigned

d = __ev_maxwu (a,b)

$d_{0:31} \leftarrow MAX_{ui}(a_{0:31}, b_{0:31})$

$d_{32:63} \leftarrow MAX_{ui}(a_{32:63}, b_{32:63})$

Each unsigned-integer word element of parameter **a** is compared to the corresponding unsigned-integer word element of parameter **b** and the larger of the two unsigned-integer elements is placed into the corresponding word element of parameter **d**.

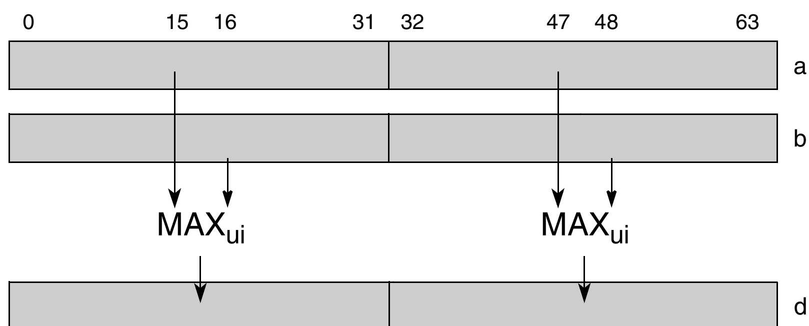


Figure 3-349. Vector Maximum Word Unsigned (__ev_maxwu)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evmaxwu d,a,b**

__ev_mbesmi[a]   __ev_mbesmi[a]

Vector Multiply Bytes Even, Signed, Modulo, Integer (to Accumulator)

d = __ev_mbesmi (a,b) (A=0)

d = __ev_mbesmia (a,b) (A=1)

```
d0:15 ←a0:7 ×si b0:7
d16:31 ←a16:23 ×si b16:23
d32:47 ←a32:39 ×si b32:39
d48:63 ←a48:55 ×si b48:55

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

For each half word element in the accumulator, the corresponding even-numbered byte signed integer elements in parameters **a** and **b** are multiplied. Each intermediate 16-bit product is added to the contents of the accumulator half words to form intermediate sums, which are placed into the corresponding half words in parameter **d**. If A = 1, the result in parameter **d** is also placed into the accumulator.

Other registers altered: ACC (if A=1)

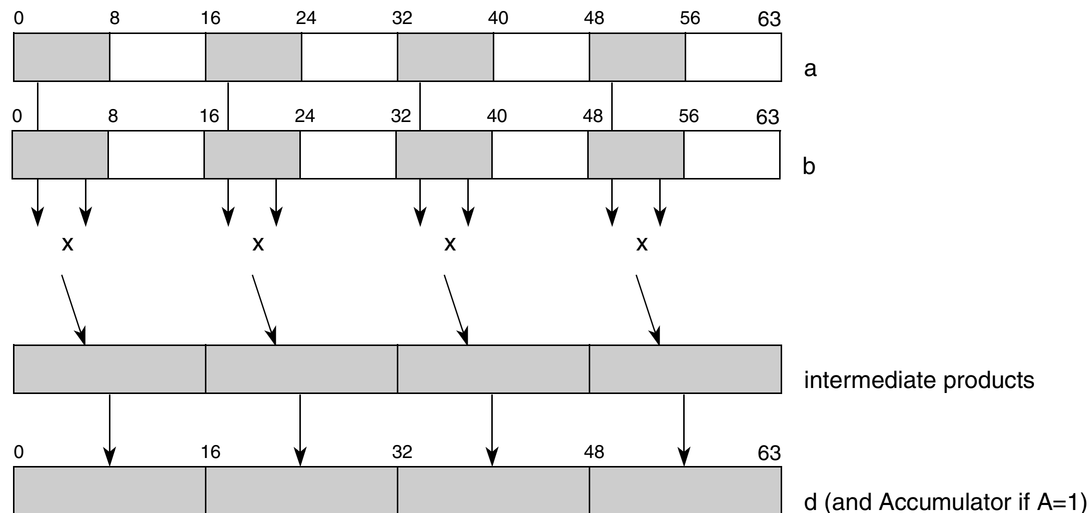


Figure 3-350. Vector Multiply Bytes Even, Signed, Modulo, Integer (to Accumulator) (__ev_mbesmi[a])

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evmbesmi d,a,b**
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evmbesmia d,a,b**

__ev_mbesmiaah __ev_mbesmiaah

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

d = __ev_mbesmiaah (a,b)

$temp0_{0:15} \leftarrow a_{0:7} \times_{si} b_{0:7}$
$d_{0:15} \leftarrow temp0_{0:15} + ACC_{0:15}$
$temp1_{0:15} \leftarrow a_{16:23} \times_{si} b_{16:23}$
$d_{16:31} \leftarrow temp1_{0:15} + ACC_{16:31}$
$temp2_{0:15} \leftarrow a_{32:39} \times_{si} b_{32:39}$
$d_{32:47} \leftarrow temp2_{0:15} + ACC_{32:47}$
$temp3_{0:15} \leftarrow a_{48:55} \times_{si} b_{48:55}$
$d_{48:63} \leftarrow temp3_{0:15} + ACC_{48:63}$

// update accumulator
$ACC_{0:63} \leftarrow d_{0:63}$

For each half word element in the accumulator, the corresponding even-numbered byte signed integer elements in parameters **a** and **b** are multiplied. Each intermediate 16-bit product is added to the contents of the accumulator half words to form intermediate sums, which are placed into the corresponding parameter **d** half words and into the accumulator.

Other registers altered: ACC

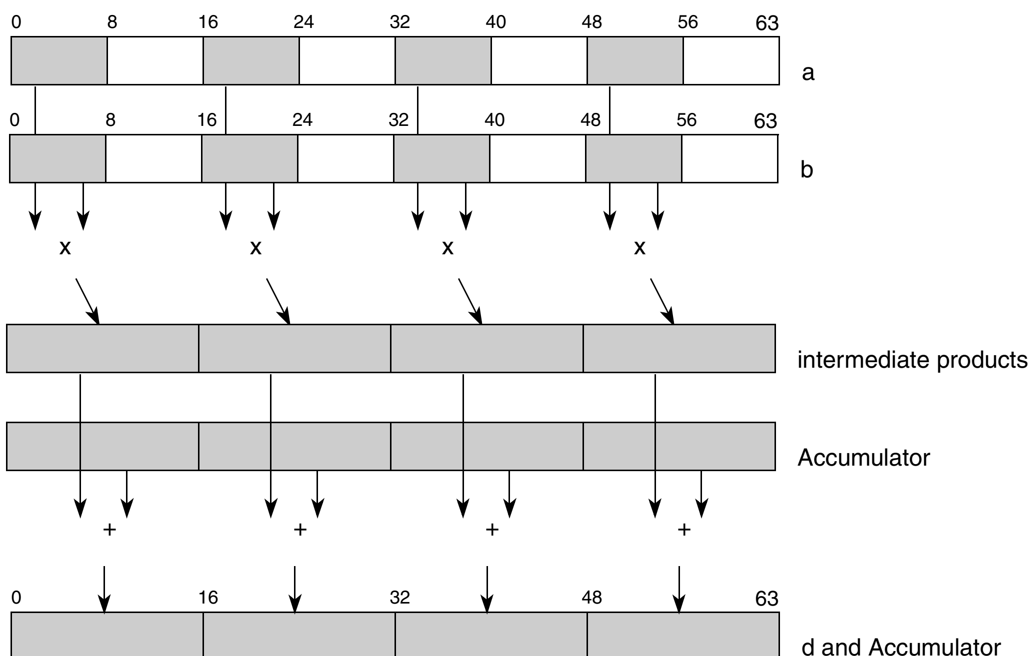


Figure 3-351. Vector Multiply Bytes Even, Signed, Modulo, Integer and Accumulate Half Words (__ev_mbesmiaah)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evmbesmiaah d,a,b**

__ev_mbesmianh __ev_mbesmianh

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

d = __ev_mbesmianh (a,b)

$$
\begin{aligned}
&temp0_{0:15} \leftarrow a_{0:7} \times_{si} b_{0:7}\\
&d_{0:15} \leftarrow ACC_{0:15} - temp0_{0:15}\\
&temp1_{0:15} \leftarrow a_{16:23} \times_{si} b_{16:23}\\
&d_{16:31} \leftarrow ACC_{16:31} - temp1_{0:15}\\
&temp2_{0:15} \leftarrow a_{32:39} \times_{si} b_{32:39}\\
&d_{32:47} \leftarrow ACC_{32:47} - temp2_{0:15}\\
&temp3_{0:15} \leftarrow a_{48:55} \times_{si} b_{48:55}\\
&d_{48:63} \leftarrow ACC_{48:63} - temp3_{0:15}
\end{aligned}
$$

// update accumulator

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

For each half word element in the accumulator, the corresponding even-numbered byte signed integer elements in parameters **a** and **b** are multiplied. Each intermediate 16-bit product is subtracted from the contents of the accumulator half words to form intermediate differences, which are placed into the corresponding parameter **d** half words and into the accumulator.

Other registers altered: ACC

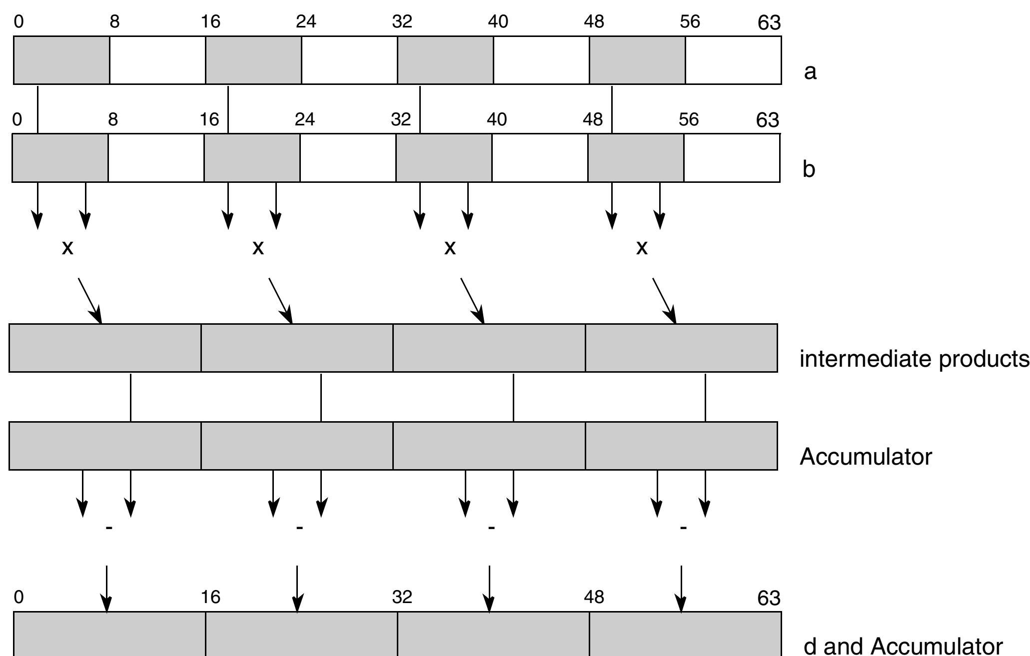


Figure 3-352. Vector Multiply Bytes Even, Signed, Modulo, Integer and Accumulate Negative Half Words (__ev_mbesmianh)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evmbesmianh d,a,b**

__ev_mbessiaah __ev_mbessiaah

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

d = __ev_mbessiaah (a,b)

```
temp0:15 ←a0:7 ×si b0:7
temp0:31 ←EXTS(ACC0:15) + EXTS(temp0:15)
ovh0 ←(temp15 ⊕ temp16)
d0:15 ←SATURATE(ovh0, temp15, 0x8000, 0x7FFF, temp16:31)

temp0:15 ←a16:23 ×si b16:23
temp0:31 ←EXTS(ACC16:31) + EXTS(temp0:15)
ovh1 ←(temp15 ⊕ temp16)
d16:31 ←SATURATE(ovh1, temp15, 0x8000, 0x7FFF, temp16:31)

temp0:15 ←a32:39 ×si b32:39
temp0:31 ←EXTS(ACC32:47) + EXTS(temp0:15)
ovl0 ←(temp15 ⊕ temp16)
d32:47 ←SATURATE(ovl0, temp15, 0x8000, 0x7FFF, temp16:31)

temp0:15 ←a48:55 ×si b48:55
temp0:31 ←EXTS(ACC48:63) + EXTS(temp0:15)
ovl1 ←(temp15 ⊕ temp16)
d48:63 ←SATURATE(ovl1, temp15, 0x8000, 0x7FFF, temp16:31)

// update accumulator
ACC0:63 ←d0:63

// update SPEFSCR
SPEFSCROVH ←ovh0 | ovh1
SPEFSCROV ←ovl0 | ovl1
SPEFSCRSOVH ←SPEFSCRSOVH | ovh0 | ovh1
SPEFSCRSOV ←SPEFSCRSOV | ovl0 | ovl1
```

The corresponding even-numbered byte signed integer elements in parameters **a** and **b** are multiplied, producing a 16-bit intermediate product. Each 16-bit product is then added to the corresponding half 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

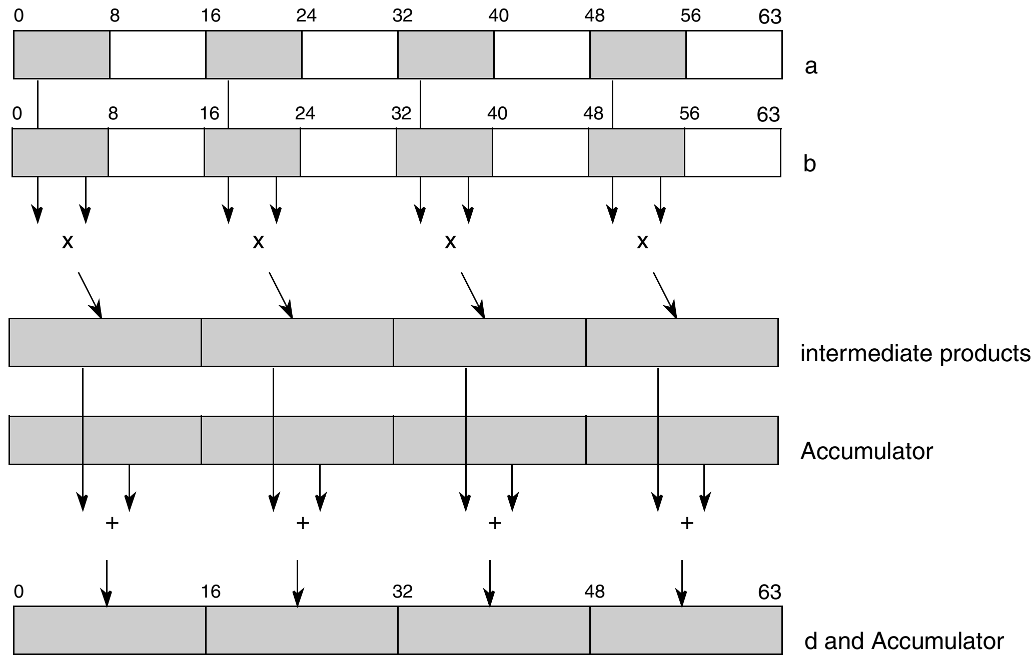


Figure 3-353. Vector Multiply Bytes Even, Signed, Saturate, Integer and Accumulate Half Words (__ev_mbessiaah)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evmbessiaah d,a,b**

__ev_mbessianh __ev_mbessianh

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

d = __ev_mbessianh (a,b)

```
temp0:15 ←a0:7 ×si b0:7
temp0:31 ←EXTS(ACC0:15) - EXTS(temp0:15)
ovh0 ←(temp15 ⊕ temp16)
d0:15 ←SATURATE(ovh0, temp15, 0x8000, 0x7FFF, temp16:31)

temp0:15 ←a16:23 ×si b16:23
temp0:31 ←EXTS(ACC16:31) - EXTS(temp0:15)
ovh1 ←(temp15 ⊕ temp16)
d16:31 ←SATURATE(ovh1, temp15, 0x8000, 0x7FFF, temp16:31)

temp0:15 ←a32:39 ×si b32:39
temp0:31 ←EXTS(ACC32:47) - EXTS(temp0:15)
ovl0 ←(temp15 ⊕ temp16)
d32:47 ←SATURATE(ovl0, temp15, 0x8000, 0x7FFF, temp16:31)

temp0:15 ←a48:55 ×si b48:55
temp0:31 ←EXTS(ACC48:63) - EXTS(temp0:15)
ovl1 ←(temp15 ⊕ temp16)
d48:63 ←SATURATE(ovl1, temp15, 0x8000, 0x7FFF, temp16:31)

// update accumulator
ACC0:63 ←d0:63

// update SPEFSCR
SPEFSCROVH ←ovh0 | ovh1
SPEFSCROV ←ovl0 | ovl1
SPEFSCRSOVH ←SPEFSCRSOVH | ovh0 | ovh1
SPEFSCRSOV ←SPEFSCRSOV | ovl0 | ovl1
```

The corresponding even-numbered byte signed integer elements in parameters **a** and **b** are multiplied, producing a 16-bit intermediate product. Each 16-bit product is then subtracted from the corresponding half 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 subtraction, the overflow and summary overflow bits are recorded in the SPEFSCR.

Other registers altered: SPEFSCR ACC

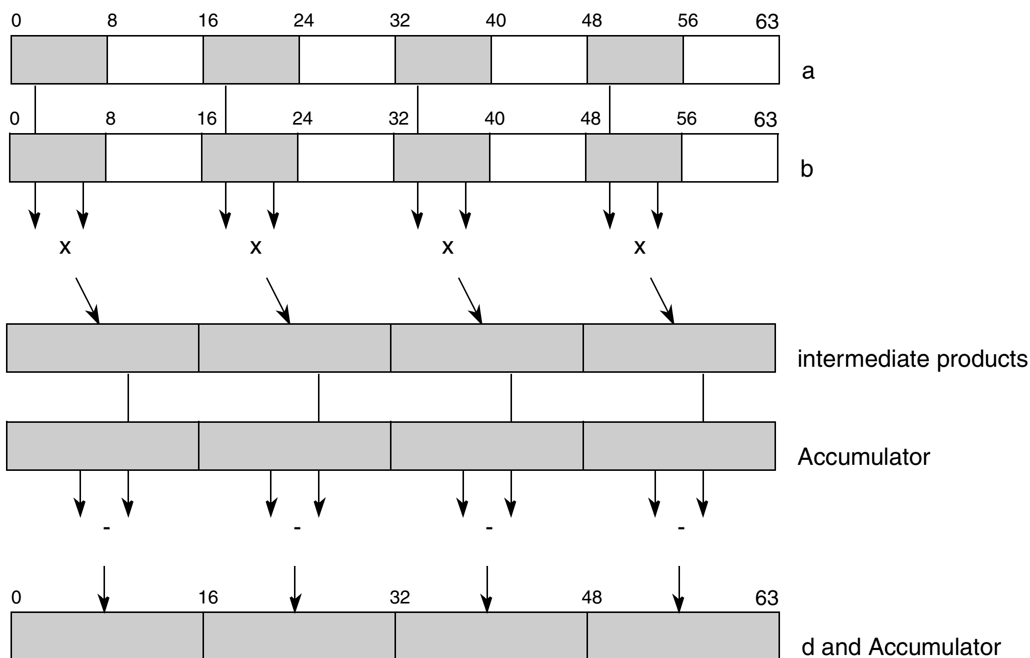


Figure 3-354. Vector Multiply Bytes Even, Signed, Saturate, Integer and Accumulate Negative Half Words (__ev_mbessianh)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evmbessianh d,a,b**

__ev_mbesumi[a] __ev_mbesumi[a]

Vector Multiply Bytes Even, Signed by Unsigned, Modulo, Integer (to Accumulator)

d = __ev_mbesumi (a,b) (A=0)

d = __ev_mbesumia (a,b) (A=1)

$d_{0:15} \leftarrow a_{0:7} \times_{su} b_{0:7}$
$d_{16:31} \leftarrow a_{16:23} \times_{su} b_{16:23}$
$d_{32:47} \leftarrow a_{32:39} \times_{su} b_{32:39}$
$d_{48:63} \leftarrow a_{48:55} \times_{su} b_{48:55}$

// update accumulator
if A = 1, then $ACC_{0:63} \leftarrow d_{0:63}$

For each half word element in the accumulator, the corresponding even-numbered byte signed integer element in parameter **a** and unsigned integer element in parameter **b** are multiplied. Each intermediate 16-bit product is added to the contents of the accumulator half words to form intermediate sums, which are placed into the corresponding half words in parameter **d**. If A = 1, the result in parameter **d** is also placed into the accumulator.

Other registers altered: ACC (if A=1)

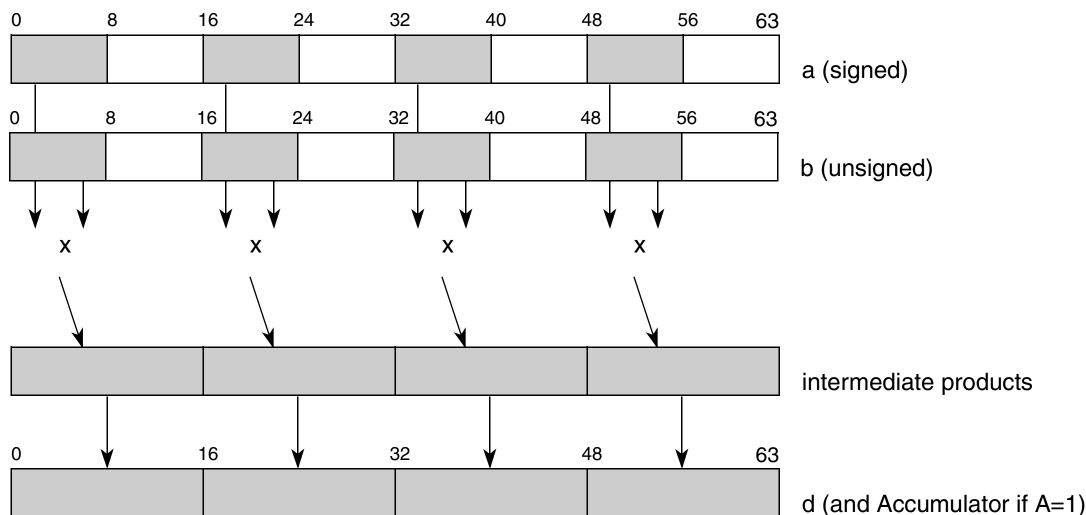


Figure 3-355. Vector Multiply Bytes Even, Signed by Unsigned, Modulo, Integer (to Accumulator) (__ev_mbesumi[a])

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evmbesumi d,a,b**
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evmbesumia d,a,b**

__ev_mbesumiaah __ev_mbesumiaah

Vector Multiply Byte Even, Signed by Unsigned, Modulo, Integer and Accumulate Half Words

d = __ev_mbesumiaah (a,b)

```
temp0_{0:15} ←a_{0:7} ×_{su} b_{0:7}
d_{0:15} ←temp0_{0:15} +ACC_{0:15}
temp1_{0:15} ←a_{16:23} ×_{su} b_{16:23}
d_{16:31} ←temp1_{0:15} +ACC_{16:31}
temp2_{0:15} ←a_{32:39} ×_{su} b_{32:39}
d_{32:47} ←temp2_{0:15} +ACC_{32:47}
temp3_{0:15} ←a_{48:55} ×_{su} b_{48:55}
d_{48:63} ←temp3_{0:15} +ACC_{48:63}

// update accumulator
ACC_{0:63} ←d_{0:63}
```

For each half word element in the accumulator, the corresponding even-numbered byte signed integer element in parameter **a** and unsigned integer element in parameter **b** are multiplied. Each intermediate 16-bit product is added to the contents of the accumulator half words to form intermediate sums, which are placed into the corresponding parameter **d** half words and into the accumulator.

Other registers altered: ACC

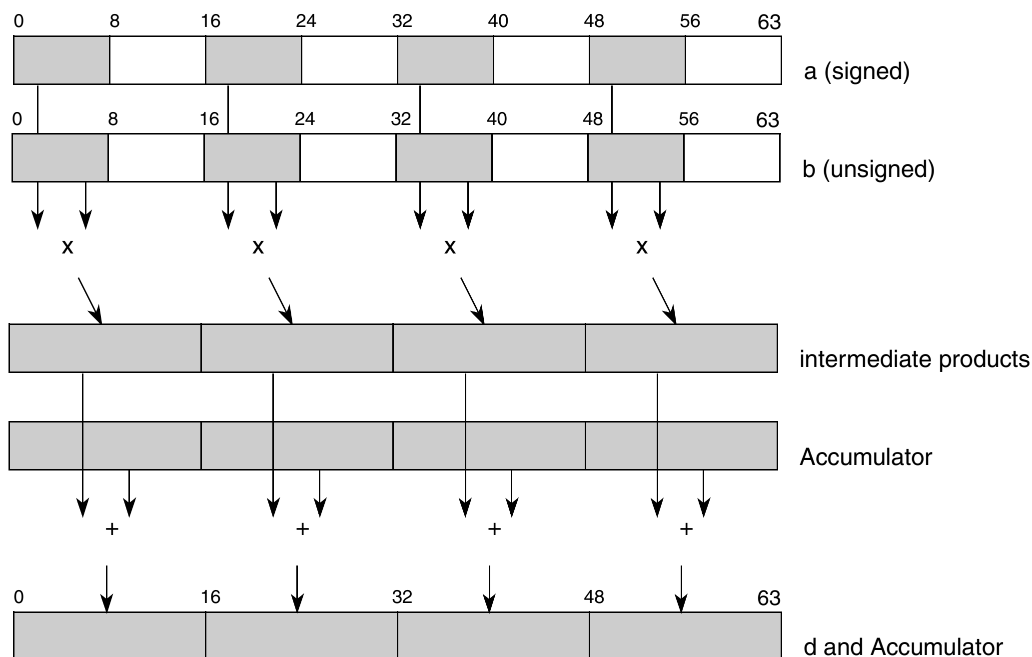


Figure 3-356. Vector Multiply Bytes Even, Signed by Unsigned, Modulo, Integer and Accumulate Half Words (__ev_mbesumiaah)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evmbesumiaah d,a,b**

__ev_mbesumianh __ev_mbesumianh

Vector Multiply Byte Even, Signed by Unsigned, Modulo, Integer and Accumulate Negative Half Words

d = __ev_mbesumianh (a,b)

$$
\begin{aligned}
&temp0_{0:15} \leftarrow a_{0:7} \times_{su} b_{0:7} \\
&d_{0:15} \leftarrow ACC_{0:15} - temp0_{0:15} \\
&temp1_{0:15} \leftarrow a_{16:23} \times_{su} b_{16:23} \\
&d_{16:31} \leftarrow ACC_{16:31} - temp1_{0:15} \\
&temp2_{0:15} \leftarrow a_{32:39} \times_{su} b_{32:39} \\
&d_{32:47} \leftarrow ACC_{32:47} - temp2_{0:15} \\
&temp3_{0:15} \leftarrow a_{48:55} \times_{su} b_{48:55} \\
&d_{48:63} \leftarrow ACC_{48:63} - temp3_{0:15}
\end{aligned}
$$

// update accumulator

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

For each half word element in the accumulator, the corresponding even-numbered byte signed integer element in parameter **a** and unsigned integer element in parameter **b** are multiplied. Each intermediate 16-bit product is subtracted from the contents of the accumulator half words to form intermediate differences, which are placed into the corresponding parameter **d** half words and into the accumulator.

Other registers altered: ACC

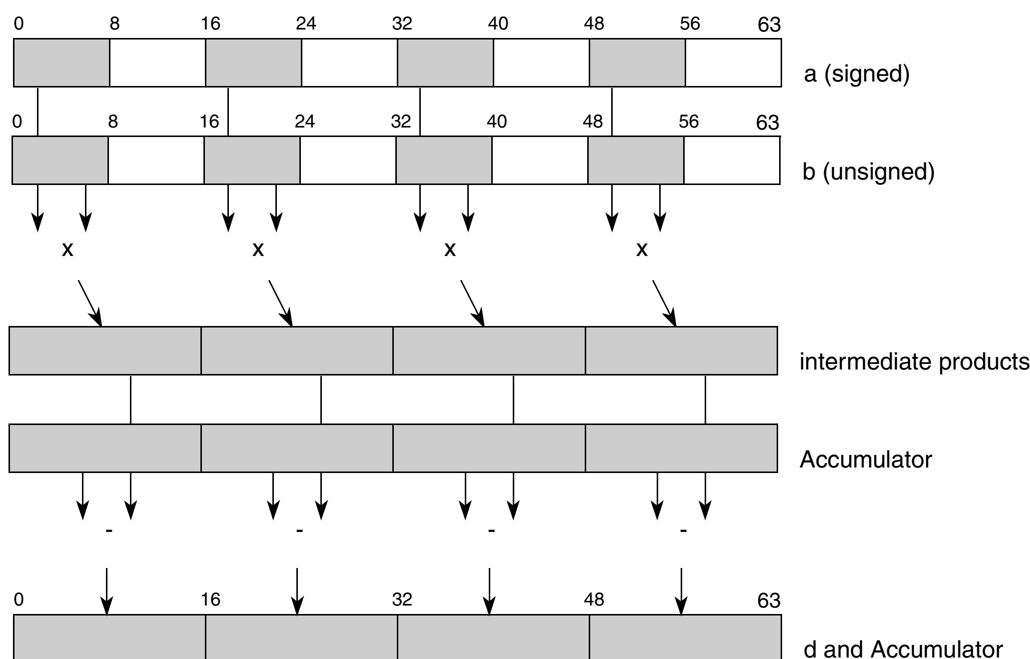


Figure 3-357. Vector Multiply Bytes Even, Signed by Unsigned, Modulo, Integer and Accumulate Negative Half Words (evmbesumianh)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evmbesumianh d,a,b**

__ev_mbesusiaah __ev_mbesusiaah

Vector Multiply Byte Even, Signed by Unsigned, Saturate, Integer and Accumulate Half Words

d = __ev_mbesusiaah (a,b)

```
temp0:15 ←a0:7 ×su b0:7
temp0:31 ←EXTS(ACC0:15) + EXTS(temp0:15)
ovh0 ←(temp15 ⊕ temp16)
d0:15 ←SATURATE(ovh0, temp15, 0x8000, 0x7FFF, temp16:31)

temp0:15 ←a16:23 ×su b16:23
temp0:31 ←EXTS(ACC16:31) + EXTS(temp0:15)
ovh1 ←(temp15 ⊕ temp16)
d16:31 ←SATURATE(ovh1, temp15, 0x8000, 0x7FFF, temp16:31)

temp0:15 ←a32:39 ×su b32:39
temp0:31 ←EXTS(ACC32:47) + EXTS(temp0:15)
ovl0 ←(temp15 ⊕ temp16)
d32:47 ←SATURATE(ovl0, temp15, 0x8000, 0x7FFF, temp16:31)

temp0:15 ←a48:55 ×su b48:55
temp0:31 ←EXTS(ACC48:63) + EXTS(temp0:15)
ovl1 ←(temp15 ⊕ temp16)
d48:63 ←SATURATE(ovl1, temp15, 0x8000, 0x7FFF, temp16:31)

// update accumulator
ACC0:63 ←d0:63

// update SPEFSCR
SPEFSCROVH ←ovh0 | ovh1
SPEFSCROV ←ovl0 | ovl1
SPEFSCRSOVH ←SPEFSCRSOVH | ovh0 | ovh1
SPEFSCRSOV ←SPEFSCRSOV | ovl0 | ovl1
```

The corresponding even-numbered byte signed integer elements in parameter **a** and unsigned integer element in parameter **b** are multiplied, producing a 16-bit intermediate product. Each 16-bit product is then added to the corresponding half 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

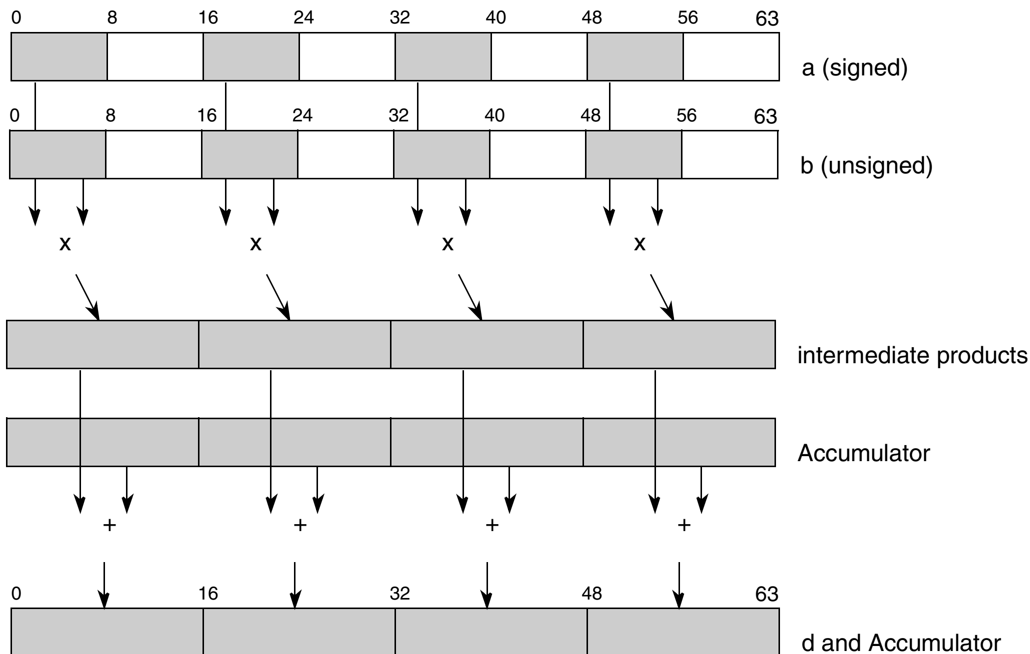


Figure 3-358. Vector Multiply Bytes Even, Signed by Unsigned, Saturate, Integer and Accumulate Half Words (__ev_mbesusiaah)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evmbesusiaah d,a,b**

__ev_mbesusianh __ev_mbesusianh

Vector Multiply Byte Even, Signed by Unsigned, Saturate, Integer and Accumulate Negative Half Words

d = __ev_mbesusianh (a,b)

```
temp0:15 ←a0:7 ×su b0:7
temp0:31 ←EXTS(ACC0:15) - EXTS(temp0:15)
ovh0 ←(temp15 ⊕ temp16)
d0:15 ←SATURATE(ovh0, temp15, 0x8000, 0x7FFF, temp16:31)

temp0:15 ←a16:23 ×su b16:23
temp0:31 ←EXTS(ACC16:31) - EXTS(temp0:15)
ovh1 ←(temp15 ⊕ temp16)
d16:31 ←SATURATE(ovh1, temp15, 0x8000, 0x7FFF, temp16:31)

temp0:15 ←a32:39 ×su b32:39
temp0:31 ←EXTS(ACC32:47) - EXTS(temp0:15)
ovl0 ←(temp15 ⊕ temp16)
d32:47 ←SATURATE(ovl0, temp15, 0x8000, 0x7FFF, temp16:31)

temp0:15 ←a48:55 ×su b48:55
temp0:31 ←EXTS(ACC48:63) - EXTS(temp0:15)
ovl1 ←(temp15 ⊕ temp16)
d48:63 ←SATURATE(ovl1, temp15, 0x8000, 0x7FFF, temp16:31)

// update accumulator
ACC0:63 ←d0:63

// update SPEFSCR
SPEFSCROVH ←ovh0 | ovh1
SPEFSCROV ←ovl0 | ovl1
SPEFSCRSOVH ←SPEFSCRSOVH | ovh0 | ovh1
SPEFSCRSOV ←SPEFSCRSOV | ovl0 | ovl1
```

The corresponding even-numbered byte signed integer element in parameter **a** and unsigned integer element in parameter **b** are multiplied, producing a 16-bit intermediate product. Each 16-bit product is then subtracted from the corresponding half 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 subtraction, the overflow and summary overflow bits are recorded in the SPEFSCR.

Other registers altered: SPEFSCR ACC

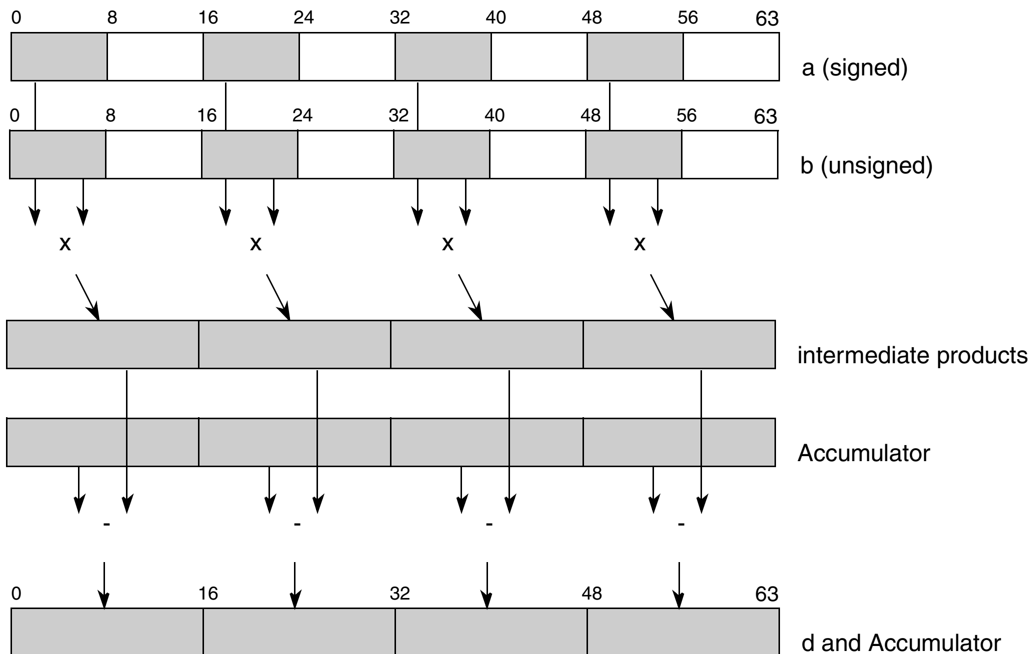


Figure 3-359. Vector Multiply Bytes Even, Signed by Unsigned, Saturate, Integer and Accumulate Negative Half Words (__ev_mbesusianh)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evmbesusianh d,a,b**

__ev_mbeumi[a] __ev_mbeumi[a]

Vector Multiply Bytes Even, Unsigned, Modulo, Integer (to Accumulator)

d = __ev_mbeumi (a,b) (A=0)

d = __ev_mbeumia (a,b) (A=1)

```
d0:15  ←a0:7   ×ui b0:7
d16:31 ←a16:23 ×ui b16:23
d32:47 ←a32:39 ×ui b32:39
d48:63 ←a48:55 ×ui b48:55

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

For each half word element in the accumulator, the corresponding even-numbered byte unsigned integer elements in parameters **a** and **b** are multiplied. Each intermediate 16-bit product is added to the contents of the accumulator half words to form intermediate sums, which are placed into the corresponding half words in parameter **d**. If A = 1, the result in parameter **d** is also placed into the accumulator.

Other registers altered: ACC (if A=1)

Figure 3-360. Vector Multiply Bytes Even, Unsigned, Modulo, Integer (to Accumulator) (__ev_mbeumi[a])

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evmbeumi d,a,b**
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evmbeumia d,a,b**

__ev_mbeumiaah __ev_mbeumiaah

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

d = __ev_mbeumiaah (a,b)

```
temp0_{0:15} ←a_{0:7} ×ui b_{0:7}
d_{0:15} ←temp0_{0:15} +ACC_{0:15}
temp1_{0:15} ←a_{16:23} ×ui b_{16:23}
d_{16:31} ←temp1_{0:15} +ACC_{16:31}
temp2_{0:15} ←a_{32:39} ×ui b_{32:39}
d_{32:47} ←temp2_{0:15} +ACC_{32:47}
temp3_{0:15} ←a_{48:55} ×ui b_{48:55}
d_{48:63} ←temp3_{0:15} +ACC_{48:63}

// update accumulator
ACC_{0:63} ←rD_{0:63}
```

For each half word element in the accumulator, the corresponding even-numbered byte unsigned integer elements in parameters **a** and **b** are multiplied. Each intermediate 16-bit product is added to the contents of the accumulator half words to form intermediate sums, which are placed into the corresponding parameter **d** half words and into the accumulator.

Other registers altered: ACC

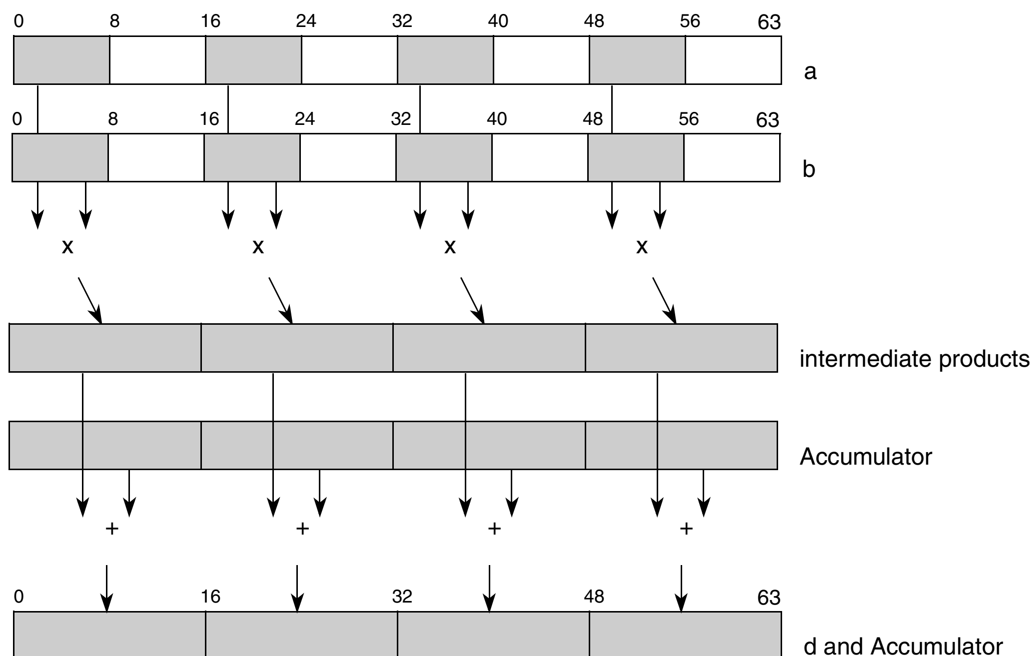


Figure 3-361. Vector Multiply Bytes Even, Unsigned, Modulo, Integer and Accumulate Half Words (__ev_mbeumiaah)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evmbeumiaah d,a,b**

__ev_mbeumianh __ev_mbeumianh

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

d = __ev_mbeumianh (a,b)

```
temp0_{0:15} ←a_{0:7} ×ui b_{0:7}
d_{0:15} ←ACC_{0:15} - temp0_{0:15}

temp1_{0:15} ←a_{16:23} ×ui b_{16:23}
d_{16:31} ←ACC_{16:31} - temp1_{0:15}

temp2_{0:15} ←a_{32:39} ×ui b_{32:39}
d_{32:47} ←ACC_{32:47} - temp2_{0:15}

temp3_{0:15} ←a_{48:55} ×ui b_{48:55}
d_{48:63} ←ACC_{48:63} - temp3_{0:15}

// update accumulator
ACC_{0:63} ←d_{0:63}
```

For each half word element in the accumulator, the corresponding even-numbered byte unsigned integer elements in parameters **a** and **b** are multiplied. Each intermediate 16-bit product is subtracted from the contents of the accumulator half words to form intermediate differences, which are placed into the corresponding parameter **d** half words and into the accumulator.

Other registers altered: ACC

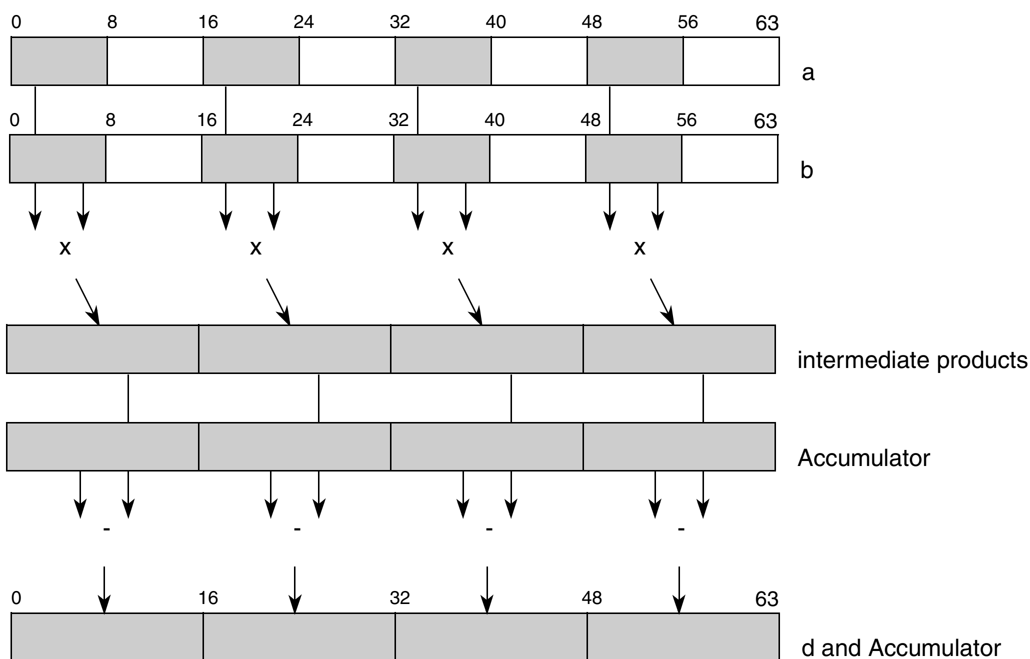


Figure 3-362. Vector Multiply Bytes Even, Unsigned, Modulo, Integer and Accumulate Negative Half Words (__ev_mbeumianh)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evmbeumianh d,a,b**

__ev_mbeusiaah __ev_mbeusiaah

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

d = __ev_mbeusiaah (a,b)

```
temp0:15 ←a0:7 ×ui b0:7
temp0:31 ←EXTZ(ACC0:15) + EXTZ(temp0:15)
ovh0 ←temp15
d0:15 ←SATURATE(ovh0, 0, 0xFFFF, 0xFFFF, temp16:31)

temp0:15 ←a16:23 ×ui b16:23
temp0:31 ←EXTZ(ACC16:31) + EXTZ(temp0:15)
ovh1 ←temp15
d16:31 ←SATURATE(ovh1, 0, 0xFFFF, 0xFFFF, temp16:31)

temp0:15 ←a32:39 ×ui b32:39
temp0:31 ←EXTZ(ACC32:47) + EXTZ(temp0:15)
ovl0 ←temp15
d32:47 ←SATURATE(ovl0, 0, 0xFFFF, 0xFFFF, temp16:31)

temp0:15 ←a48:55 ×ui b48:55
temp0:31 ←EXTZ(ACC48:63) + EXTZ(temp0:15)
ovl1 ←temp15
d48:63 ←SATURATE(ovl1, 0, 0xFFFF, 0xFFFF, temp16:31)

// update accumulator
ACC0:63 ←d0:63

// update SPEFSCR
SPEFSCROVH ←ovh0 | ovh1
SPEFSCROV ←ovl0 | ovl1
SPEFSCRSOVH ←SPEFSCRSOVH | ovh0 | ovh1
SPEFSCRSOV ←SPEFSCRSOV | ovl0 | ovl1
```

The corresponding even-numbered byte unsigned integer elements in parameters **a** and **b** are multiplied, producing a 16-bit intermediate product. Each 16-bit product is then added to the corresponding half 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

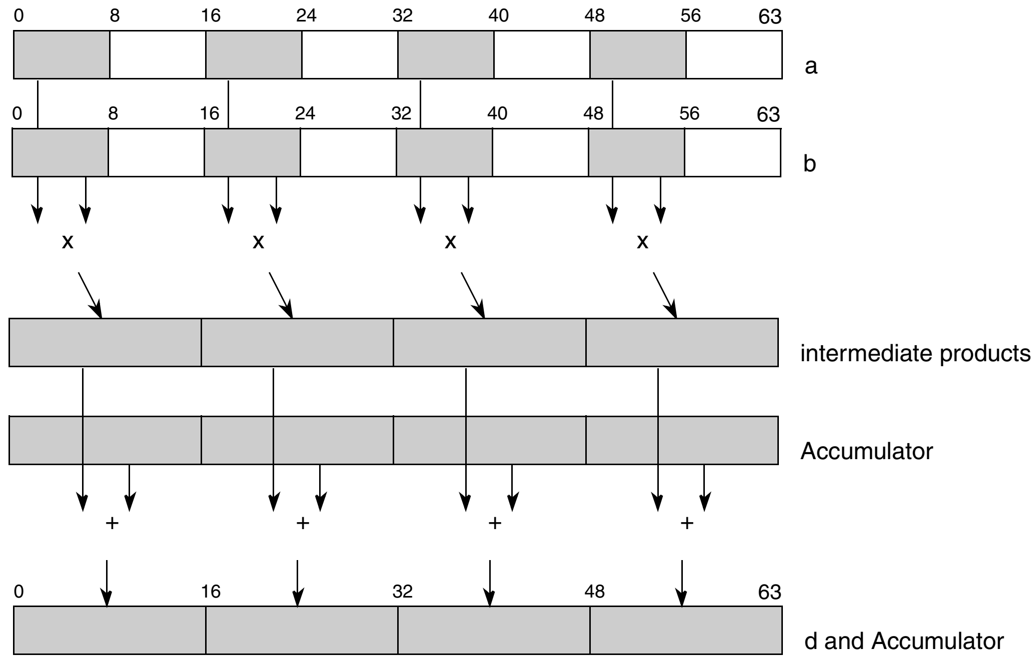


Figure 3-363. Vector Multiply Bytes Even, Unsigned, Saturate, Integer and Accumulate Half Words (__ev_mbeusiaah)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evmbeusiaah d,a,b**

__ev_mbeusianh __ev_mbeusianh

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

d = __ev_mbeusianh (a,b)

```
temp0:15 ←a0:7 ×ui b0:7
temp0:31 ←EXTZ(ACC0:15) - EXTZ(temp0:15)
ovh0 ←temp15
d0:15 ←SATURATE(ovh0, 0, 0x0000, 0x0000, temp16:31)

temp0:15 ←a16:23 ×ui b16:23
temp0:31 ←EXTZ(ACC16:31) - EXTZ(temp0:15)
ovh1 ←temp15
d16:31 ←SATURATE(ovh1, 0, 0x0000, 0x0000, temp16:31)

temp0:15 ←a32:39 ×ui b32:39
temp0:31 ←EXTZ(ACC32:47) - EXTZ(temp0:15)
ovl0 ←temp15
d32:47 ←SATURATE(ovl0, 0, 0x0000, 0x0000, temp16:31)

temp0:15 ←a48:55 ×ui b48:55
temp0:31 ←EXTZ(ACC48:63) - EXTZ(temp0:15)
ovl1 ←temp15
d48:63 ←SATURATE(ovl1, 0, 0x0000, 0x0000, temp16:31)

// update accumulator
ACC0:63 ←d0:63

// update SPEFSCR
SPEFSCROVH ←ovh0 | ovh1
SPEFSCROV ←ovl0 | ovl1
SPEFSCRSOVH ←SPEFSCRSOVH | ovh0 | ovh1
SPEFSCRSOV ←SPEFSCRSOV | ovl0 | ovl1
```

The corresponding even-numbered byte unsigned integer elements in parameters **a** and **b** are multiplied, producing a 16-bit intermediate product. Each 16-bit product is then subtracted from the corresponding half 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

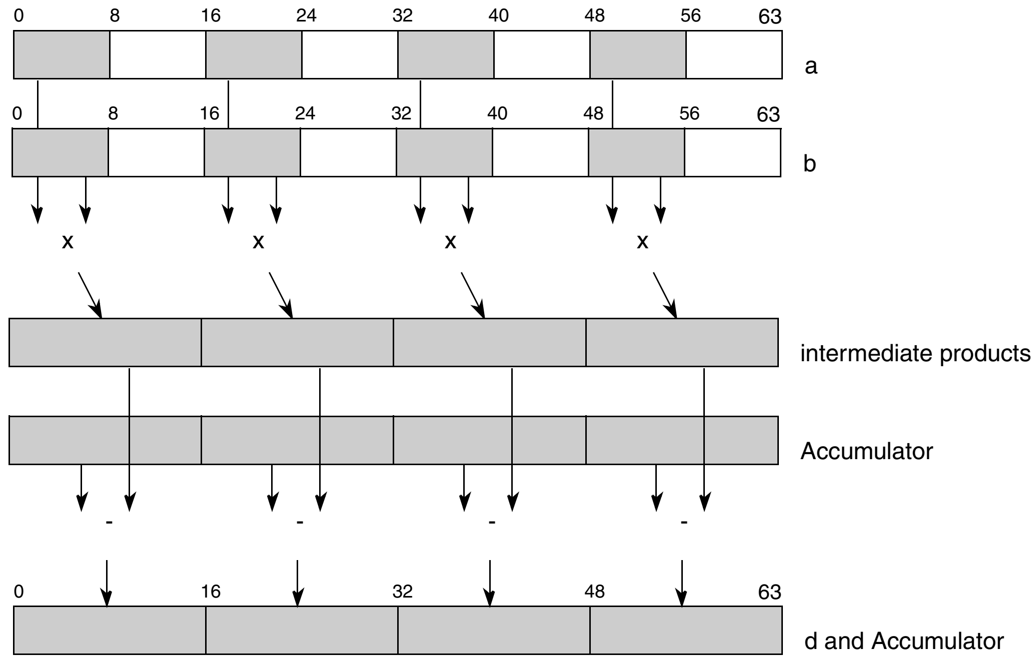


Figure 3-364. Vector Multiply Bytes Even, Unsigned, Saturate, Integer and Accumulate Negative Half Words (__ev_mbeusianh)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evmbeusianh d,a,b**

__ev_mbosmi[a] __ev_mbosmi[a]

Vector Multiply Bytes Odd, Signed, Modulo, Integer (to Accumulator)

d = __ev_mbosmi (a,b) (A=0)

d = __ev_mbosmia (a,b) (A=1)

```
d0:15 ←a8:15 ×si b8:15
d16:31 ←a24:31 ×si b24:31
d32:47 ←a40:47 ×si b40:47
d48:63 ←a56:63 ×si b56:63

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

For each half word element in the accumulator, the corresponding odd-numbered byte signed integer elements in parameters **a** and **b** are multiplied. Each intermediate 16-bit product is added to the contents of the accumulator half words to form intermediate sums, which are placed into the corresponding half words in parameter **d**. If A = 1, the result in parameter **d** is also placed into the accumulator.

Other registers altered: ACC (if A=1)

Figure 3-365. Vector Multiply Bytes Odd, Signed, Modulo, Integer (to Accumulator) (__ev_mbosmi[a])

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evmbosmi d,a,b**
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evmbosmia d,a,b**

__ev_mbosmiaah __ev_mbosmiaah

Vector Multiply Byte Odd, Signed, Modulo, Integer and Accumulate Half Words

d = __ev_mbosmiaah (a,b)

$temp0_{0:15} \leftarrow a_{8:15} \times_{si} b_{8:15}$
$d_{0:15} \leftarrow temp0_{0:15} + ACC_{0:15}$
$temp1_{0:15} \leftarrow a_{24:31} \times_{si} b_{24:31}$
$d_{16:31} \leftarrow temp1_{0:15} + ACC_{16:31}$
$temp2_{0:15} \leftarrow a_{40:47} \times_{si} b_{40:47}$
$d_{32:47} \leftarrow temp2_{0:15} + ACC_{32:47}$
$temp3_{0:15} \leftarrow a_{56:63} \times_{si} b_{56:63}$
$d_{48:63} \leftarrow temp3_{0:15} + ACC_{48:63}$

// update accumulator
$ACC_{0:63} \leftarrow d_{0:63}$

For each half word element in the accumulator, the corresponding odd-numbered byte signed integer elements in parameters **a** and **b** are multiplied. Each intermediate 16-bit product is added to the contents of the accumulator half words to form intermediate sums, which are placed into the corresponding parameter **d** half words and into the accumulator.

Other registers altered: ACC

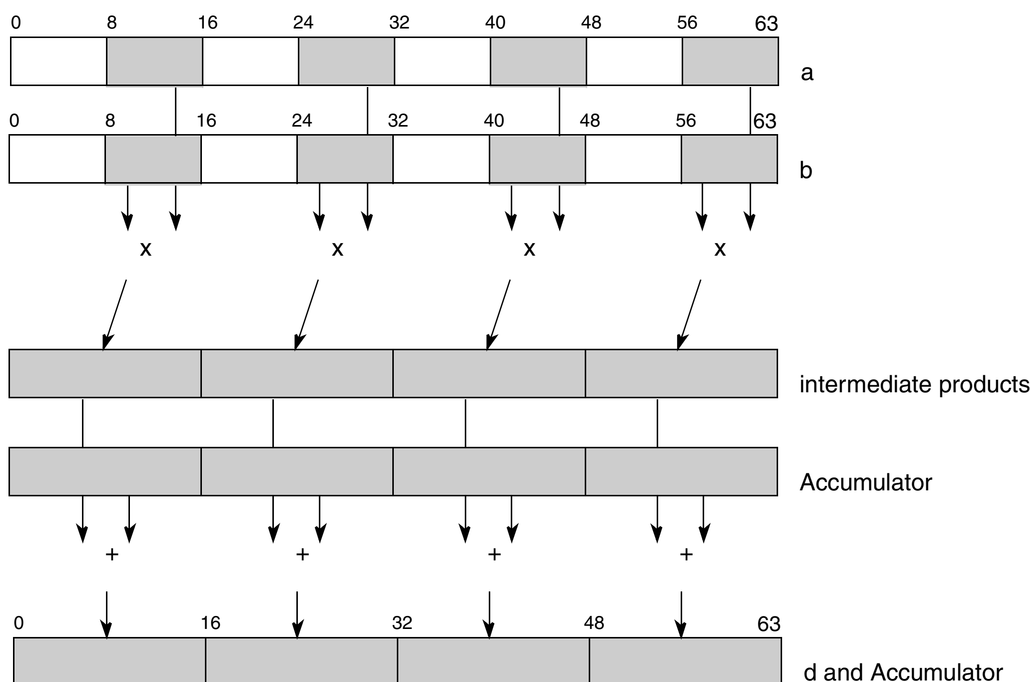


Figure 3-366. Vector Multiply Bytes Odd, Signed, Modulo, Integer and Accumulate Half Words (__ev_mbosmiaah)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evmbosmiaah d,a,b**

__ev_mbosmianh __ev_mbosmianh

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

d = __ev_mbosmianh (a,b)

```
temp0_{0:15} ←a_{8:15} ×_{si} b_{8:15}
d_{0:15} ←ACC_{0:15} - temp0_{0:15}
temp1_{0:15} ←a_{24:31} ×_{si} b_{24:31}
d_{16:31} ←ACC_{16:31} - temp1_{0:15}
temp2_{0:15} ←a_{40:47} ×_{si} b_{40:47}
d_{32:47} ←ACC_{32:47} - temp2_{0:15}
temp3_{0:15} ←a_{56:63} ×_{si} b_{56:63}
d_{48:63} ←ACC_{48:63} - temp3_{0:15}

// update accumulator
ACC_{0:63} ←d_{0:63}
```

For each half word element in the accumulator, the corresponding odd-numbered byte signed integer elements in parameters **a** and **b** are multiplied. Each intermediate 16-bit product is subtracted from the contents of the accumulator half words to form intermediate differences, which are placed into the corresponding parameter **d** half words and into the accumulator.

Other registers altered: ACC

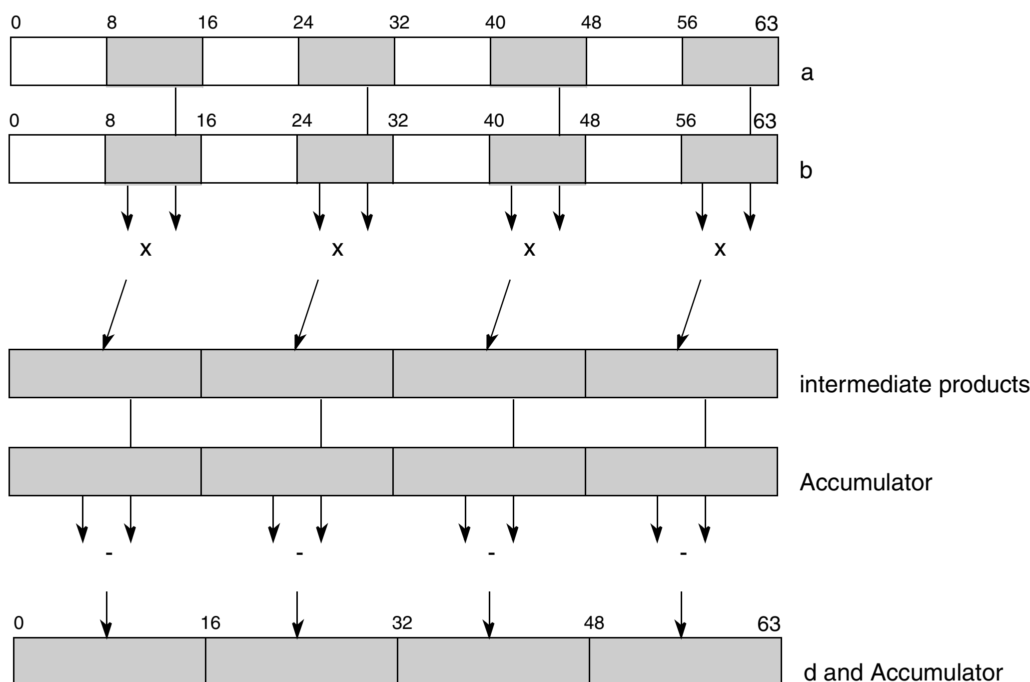


Figure 3-367. Vector Multiply Bytes Odd, Signed, Modulo, Integer and Accumulate Negative Half Words (__ev_mbosmianh)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evmbosmianh d,a,b**

__ev_mbossiaah __ev_mbossiaah

Vector Multiply Byte Odd, Signed, Saturate, Integer and Accumulate Half Words

d = __ev_mbossiaah (a,b)

```
temp0:15 ←a8:15 ×si b8:15
temp0:31 ←EXTS(ACC0:15) + EXTS(temp0:15)
ovh0 ←(temp15 ⊕ temp16)
d0:15 ←SATURATE(ovh0, temp15, 0x8000, 0x7FFF, temp16:31)

temp0:15 ←a24:31 ×si b24:31
temp0:31 ←EXTS(ACC16:31) + EXTS(temp0:15)
ovh1 ←(temp15 ⊕ temp16)
d16:31 ←SATURATE(ovh1, temp15, 0x8000, 0x7FFF, temp16:31)

temp0:15 ←a40:47 ×si b40:47
temp0:31 ←EXTS(ACC32:47) + EXTS(temp0:15)
ovl0 ←(temp15 ⊕ temp16)
d32:47 ←SATURATE(ovl0, temp15, 0x8000, 0x7FFF, temp16:31)

temp0:15 ←a56:63 ×si b56:63
temp0:31 ←EXTS(ACC48:63) + EXTS(temp0:15)
ovl1 ←(temp15 ⊕ temp16)
d48:63 ←SATURATE(ovl1, temp15, 0x8000, 0x7FFF, temp16:31)

// update accumulator
ACC0:63 ←d0:63

// update SPEFSCR
SPEFSCROVH ←ovh0 | ovh1
SPEFSCROV ←ovl0 | ovl1
SPEFSCRSOVH ←SPEFSCRSOVH | ovh0 | ovh1
SPEFSCRSOV ←SPEFSCRSOV | ovl0 | ovl1
```

The corresponding odd-numbered byte signed integer elements in parameters **a** and **b** are multiplied, producing a 16-bit intermediate product. Each 16-bit product is then added to the corresponding half 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

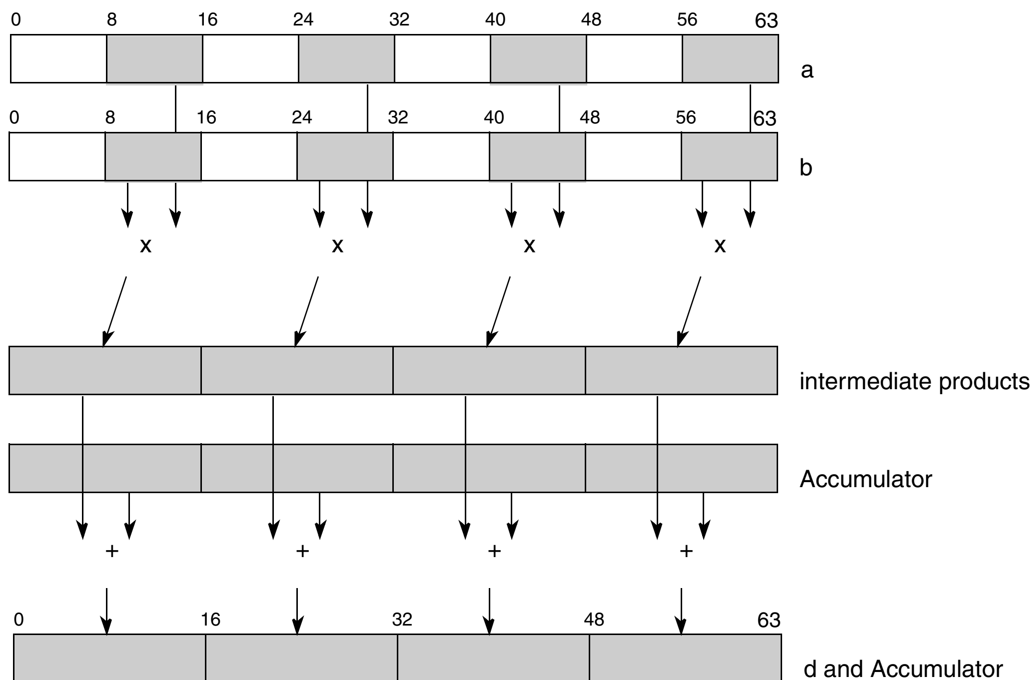


Figure 3-368. Vector Multiply Bytes Odd, Signed, Saturate, Integer and Accumulate Half Words (__ev_mbossiaah)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evmbossiaah d,a,b**

__ev_mbossianh __ev_mbossianh

Vector Multiply Byte Odd, Signed, Saturate, Integer and Accumulate Negative Half Words

d = __ev_mbossianh (a,b)

```
temp0:15 ←a8:15 ×si b8:15
temp0:31 ←EXTS(ACC0:15) - EXTS(temp0:15)
ovh0 ←(temp15 ⊕ temp16)
d0:15 ←SATURATE(ovh0, temp15, 0x8000, 0x7FFF, temp16:31)

temp0:15 ←a24:31 ×si b24:31
temp0:31 ←EXTS(ACC16:31) - EXTS(temp0:15)
ovh1 ←(temp15 ⊕ temp16)
d16:31 ←SATURATE(ovh1, temp15, 0x8000, 0x7FFF, temp16:31)

temp0:15 ←a40:47 ×si b40:47
temp0:31 ←EXTS(ACC32:47) - EXTS(temp0:15)
ovl0 ←(temp15 ⊕ temp16)
d32:47 ←SATURATE(ovl0, temp15, 0x8000, 0x7FFF, temp16:31)

temp0:15 ←a56:63 ×si b56:63
temp0:31 ←EXTS(ACC48:63) - EXTS(temp0:15)
ovl1 ←(temp15 ⊕ temp16)
d48:63 ←SATURATE(ovl1, temp15, 0x8000, 0x7FFF, temp16:31)

// update accumulator
ACC0:63 ←d0:63

// update SPEFSCR
SPEFSCROVH ←ovh0 | ovh1
SPEFSCROV ←ovl0 | ovl1
SPEFSCRSOVH ←SPEFSCRSOVH | ovh0 | ovh1
SPEFSCRSOV ←SPEFSCRSOV | ovl0 | ovl1
```

The corresponding odd-numbered byte signed integer elements in parameters **a** and **b** are multiplied, producing a 16-bit intermediate product. Each 16-bit product is then subtracted from the corresponding half 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 subtraction, the overflow and summary overflow bits are recorded in the SPEFSCR.

Other registers altered: SPEFSCR ACC

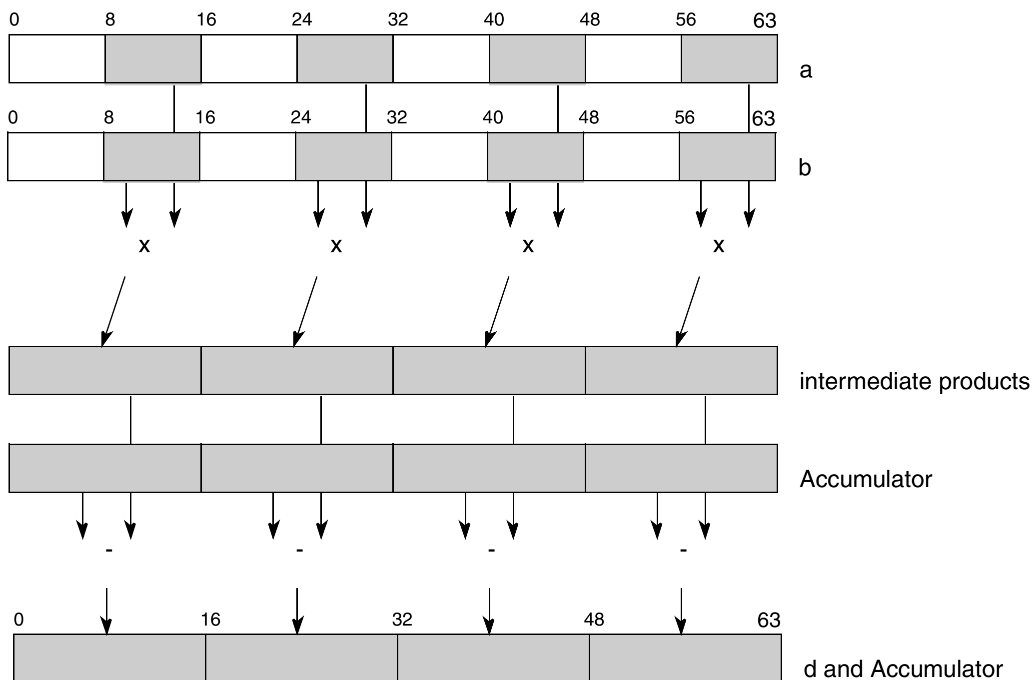


Figure 3-369. Vector Multiply Bytes Odd, Signed, Saturate, Integer and Accumulate Negative Half Words (__ev_mbossianh)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evmbossianh d,a,b**

__ev_mbosumi[a] __ev_mbosumi[a]

Vector Multiply Bytes Odd, Signed by Unsigned, Modulo, Integer (to Accumulator)

d = __ev_mbosumi (a,b) (A=0)

d = __ev_mbosumia (a,b) (A=1)

$$d_{0:15} \leftarrow a_{8:15} \times_{su} b_{8:15}$$
$$d_{16:31} \leftarrow a_{24:31} \times_{su} b_{24:31}$$
$$d_{32:47} \leftarrow a_{40:47} \times_{su} b_{40:47}$$
$$d_{48:63} \leftarrow a_{56:63} \times_{su} b_{56:63}$$

```
// update accumulator
```

if A = 1, then $ACC_{0:63} \leftarrow d_{0:63}$

For each half word element in the accumulator, the corresponding odd-numbered byte signed integer element in parameter **a** and unsigned integer element in parameter **b** are multiplied. Each intermediate 16-bit product is added to the contents of the accumulator half words to form intermediate sums, which are placed into the corresponding half words in parameter **d**. If A = 1, the result in parameter **d** is also placed into the accumulator.

Other registers altered: ACC (if A=1)

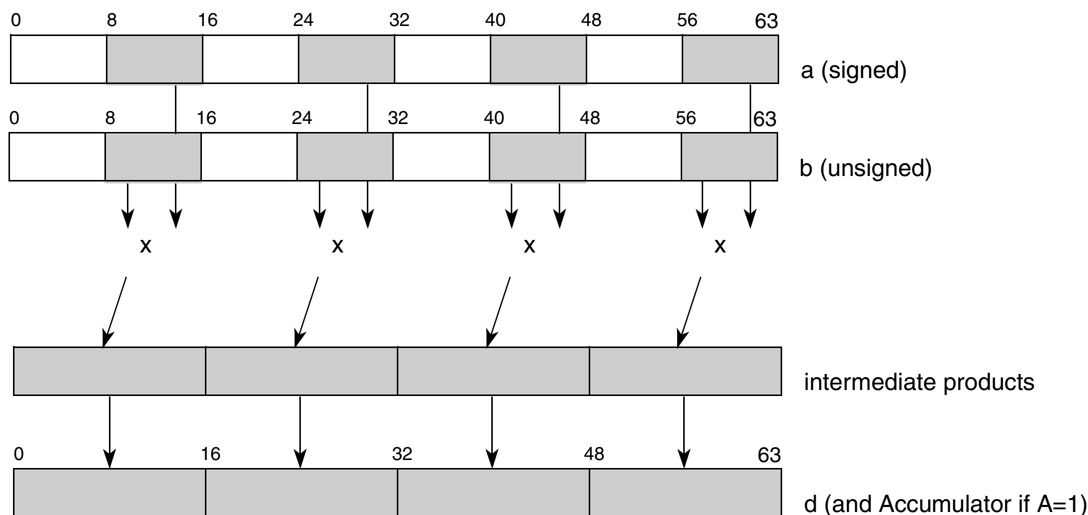


Figure 3-370. Vector Multiply Bytes Odd, Signed by Unsigned, Modulo, Integer (to Accumulator) (__ev_mbosumi[a])

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evmbosumi d,a,b**
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evmbosumia d,a,b**

__ev_mbosumiaah __ev_mbosumiaah

Vector Multiply Byte Odd, Signed by Unsigned, Modulo, Integer and Accumulate Half Words

d = __ev_mbosumiaah (a,b)

```
temp0[0:15] ← a[8:15] ×su b[8:15]
d[0:15] ← temp0[0:15] + ACC[0:15]
temp1[0:15] ← a[24:31] ×su b[24:31]
d[16:31] ← temp1[0:15] + ACC[16:31]
temp2[0:15] ← a[40:47] ×su b[40:47]
d[32:47] ← temp2[0:15] + ACC[32:47]
temp3[0:15] ← a[56:63] ×su b[56:63]
d[48:63] ← temp3[0:15] + ACC[48:63]

// update accumulator
ACC[0:63] ← d[0:63]
```

For each half word element in the accumulator, the corresponding odd-numbered byte signed integer element in parameter **a** and unsigned integer element in parameter **b** are multiplied. Each intermediate 16-bit product is added to the contents of the accumulator half words to form intermediate sums, which are placed into the corresponding parameter **d** half words and into the accumulator.

Other registers altered: ACC

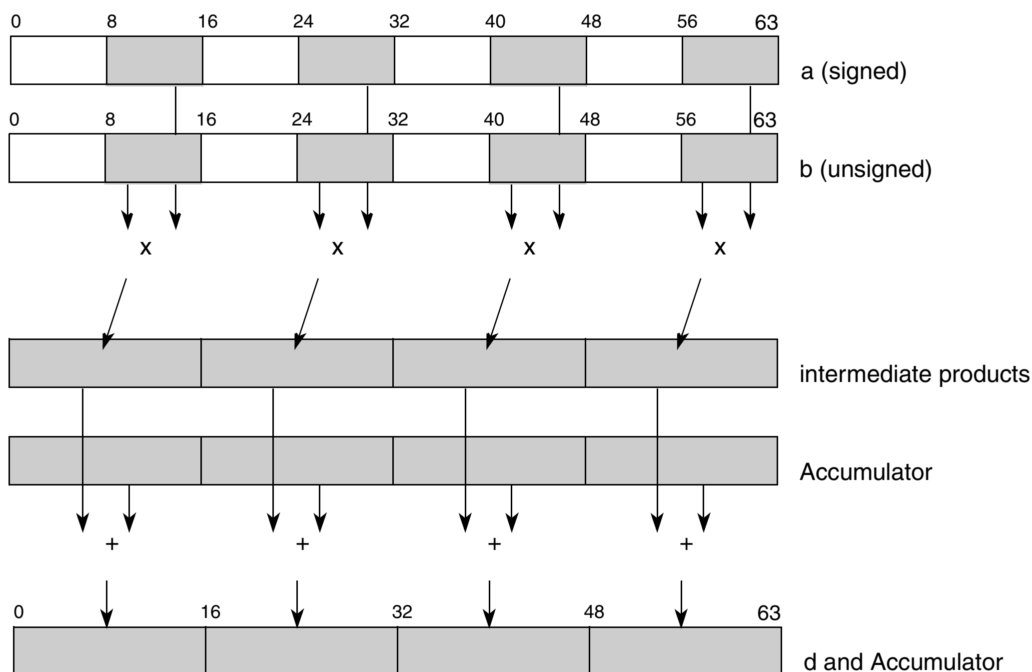


Figure 3-371. Vector Multiply Bytes Odd, Signed by Unsigned, Modulo, Integer and Accumulate Half Words (__ev_mbosumiaah)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evmbosumiaah d,a,b**

__ev_mbosumianh __ev_mbosumianh

Vector Multiply Byte Odd, Signed by Unsigned, Modulo, Integer and Accumulate Negative Half Words

d = __ev_mbosumianh (a,b)

$temp0_{0:15} \leftarrow a_{8:15} \times_{su} b_{8:15}$
$d_{0:15} \leftarrow ACC_{0:15} - temp0_{0:15}$
$temp1_{0:15} \leftarrow a_{24:31} \times_{su} b_{24:31}$
$d_{16:31} \leftarrow ACC_{16:31} - temp1_{0:15}$
$temp2_{0:15} \leftarrow a_{40:47} \times_{su} b_{40:47}$
$d_{32:47} \leftarrow ACC_{32:47} - temp2_{0:15}$
$temp3_{0:15} \leftarrow a_{56:63} \times_{su} b_{56:63}$
$d_{48:63} \leftarrow ACC_{48:63} - temp3_{0:15}$

// update accumulator
$ACC_{0:63} \leftarrow d_{0:63}$

For each half word element in the accumulator, the corresponding odd-numbered byte signed integer element in parameter **a** and unsigned integer element in parameter **b** are multiplied. Each intermediate 16-bit product is subtracted from the contents of the accumulator half words to form intermediate differences, which are placed into the corresponding parameter **d** half words and into the accumulator.

Other registers altered: ACC

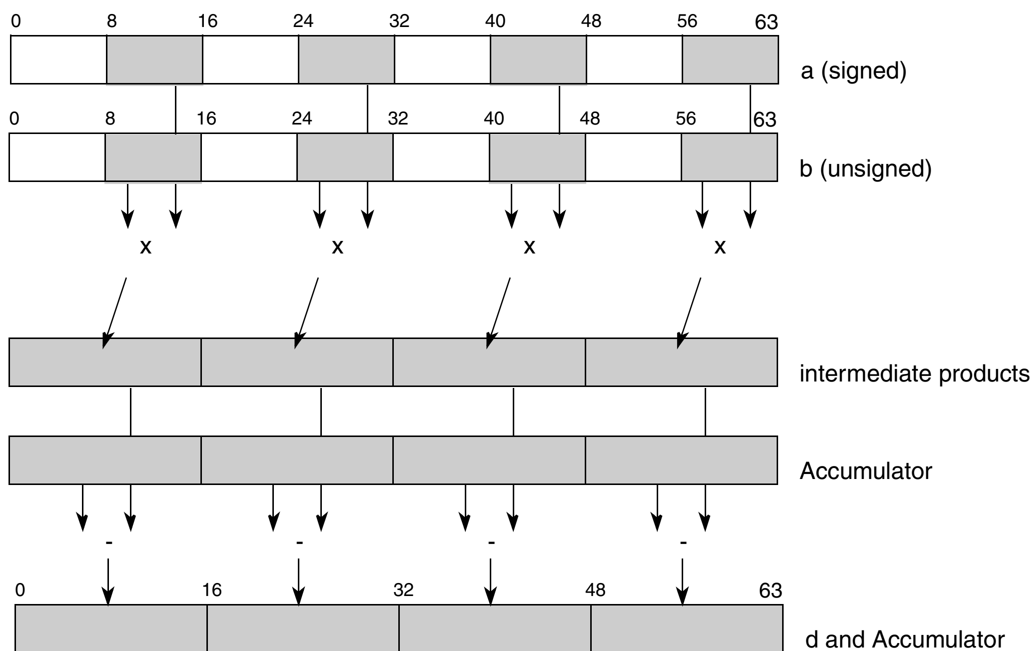


Figure 3-372. Vector Multiply Bytes Odd, Signed by Unsigned, Modulo, Integer and Accumulate Negative Half Words (__ev_mbosumianh)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evmbosumianh d,a,b**

__ev_mbosusiaah __ev_mbosusiaah

Vector Multiply Byte Odd, Signed by Unsigned, Saturate, Integer and Accumulate Half Words

d = __ev_mbosusiaah (a,b)

```
temp0:15 ←a8:15 ×su b8:15
temp0:31 ←EXTS(ACC0:15) + EXTS(temp0:15)
ovh0 ←(temp15 ⊕ temp16)
d0:15 ←SATURATE(ovh0, temp15, 0x8000, 0x7FFF, temp16:31)

temp0:15 ←a24:31 ×su b24:31
temp0:31 ←EXTS(ACC16:31) + EXTS(temp0:15)
ovh1 ←(temp15 ⊕ temp16)
d16:31 ←SATURATE(ovh1, temp15, 0x8000, 0x7FFF, temp16:31)

temp0:15 ←a40:47 ×su b40:47
temp0:31 ←EXTS(ACC32:47) + EXTS(temp0:15)
ovl0 ←(temp15 ⊕ temp16)
d32:47 ←SATURATE(ovl0, temp15, 0x8000, 0x7FFF, temp16:31)

temp0:15 ←a56:63 ×su b56:63
temp0:31 ←EXTS(ACC48:63) + EXTS(temp0:15)
ovl1 ←(temp15 ⊕ temp16)
d48:63 ←SATURATE(ovl1, temp15, 0x8000, 0x7FFF, temp16:31)

// update accumulator
ACC0:63 ←d0:63

// update SPEFSCR
SPEFSCROVH ←ovh0 | ovh1
SPEFSCROV ←ovl0 | ovl1
SPEFSCRSOVH ←SPEFSCRSOVH | ovh0 | ovh1
SPEFSCRSOV ←SPEFSCRSOV | ovl0 | ovl1
```

The corresponding odd-numbered byte signed integer elements in parameter **a** and unsigned integer element in parameter **b** are multiplied, producing a 16-bit intermediate product. Each 16-bit product is then added to the corresponding half 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

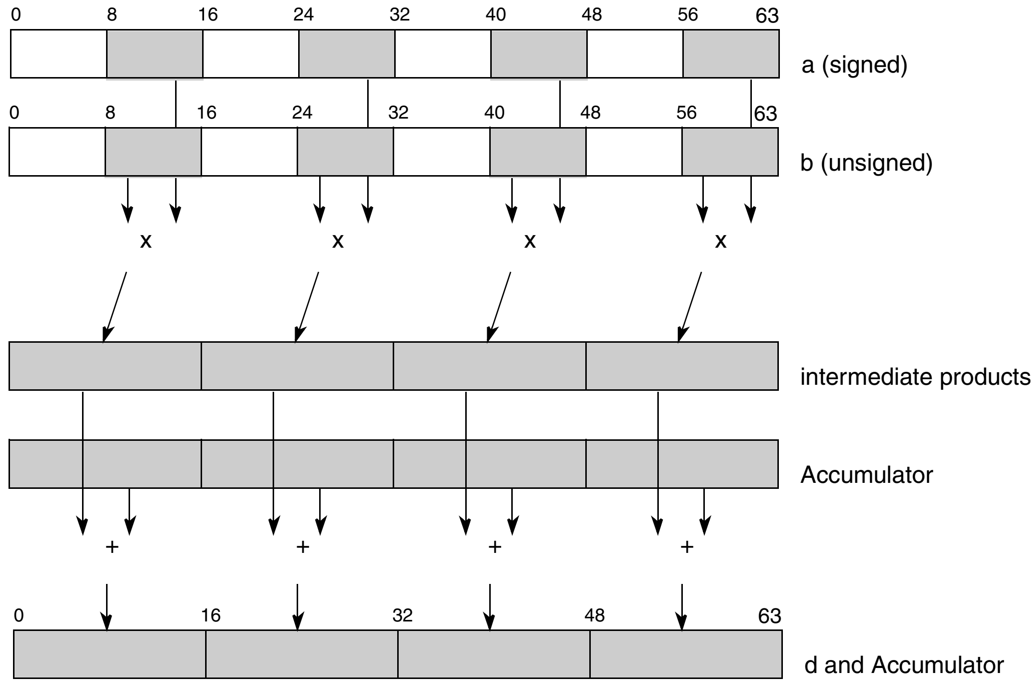


Figure 3-373. Vector Multiply Bytes Odd, Signed by Unsigned, Saturate, Integer and Accumulate Half Words (__ev_mbosusiaah)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evmbosusiaah d,a,b**

__ev_mbosusianh __ev_mbosusianh

Vector Multiply Byte Odd, Signed by Unsigned, Saturate, Integer and Accumulate Negative Half Words

d = __ev_mbosusianh (a,b)

```
temp0:15 ←a8:15 ×su b8:15
temp0:31 ←EXTS(ACC0:15) - EXTS(temp0:15)
ovh0 ←(temp15 ⊕ temp16)
d0:15 ←SATURATE(ovh0, temp15, 0x8000, 0x7FFF, temp16:31)

temp0:15 ←a24:31 ×su b24:31
temp0:31 ←EXTS(ACC16:31) - EXTS(temp0:15)
ovh1 ←(temp15 ⊕ temp16)
d16:31 ←SATURATE(ovh1, temp15, 0x8000, 0x7FFF, temp16:31)

temp0:15 ←a40:47 ×su b40:47
temp0:31 ←EXTS(ACC32:47) - EXTS(temp0:15)
ovl0 ←(temp15 ⊕ temp16)
d32:47 ←SATURATE(ovl0, temp15, 0x8000, 0x7FFF, temp16:31)

temp0:15 ←a56:63 ×su b56:63
temp0:31 ←EXTS(ACC48:63) - EXTS(temp0:15)
ovl1 ←(temp15 ⊕ temp16)
d48:63 ←SATURATE(ovl1, temp15, 0x8000, 0x7FFF, temp16:31)

// update accumulator
ACC0:63 ←d0:63

// update SPEFSCR
SPEFSCROVH ←ovh0 | ovh1
SPEFSCROV ←ovl0 | ovl1
SPEFSCRSOVH ←SPEFSCRSOVH | ovh0 | ovh1
SPEFSCRSOV ←SPEFSCRSOV | ovl0 | ovl1
```

The corresponding odd-numbered byte signed integer element in parameter **a** and unsigned integer element in parameter **b** are multiplied, producing a 16-bit intermediate product. Each 16-bit product is then subtracted from the corresponding half 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 subtraction, the overflow and summary overflow bits are recorded in the SPEFSCR.

Other registers altered: SPEFSCR ACC

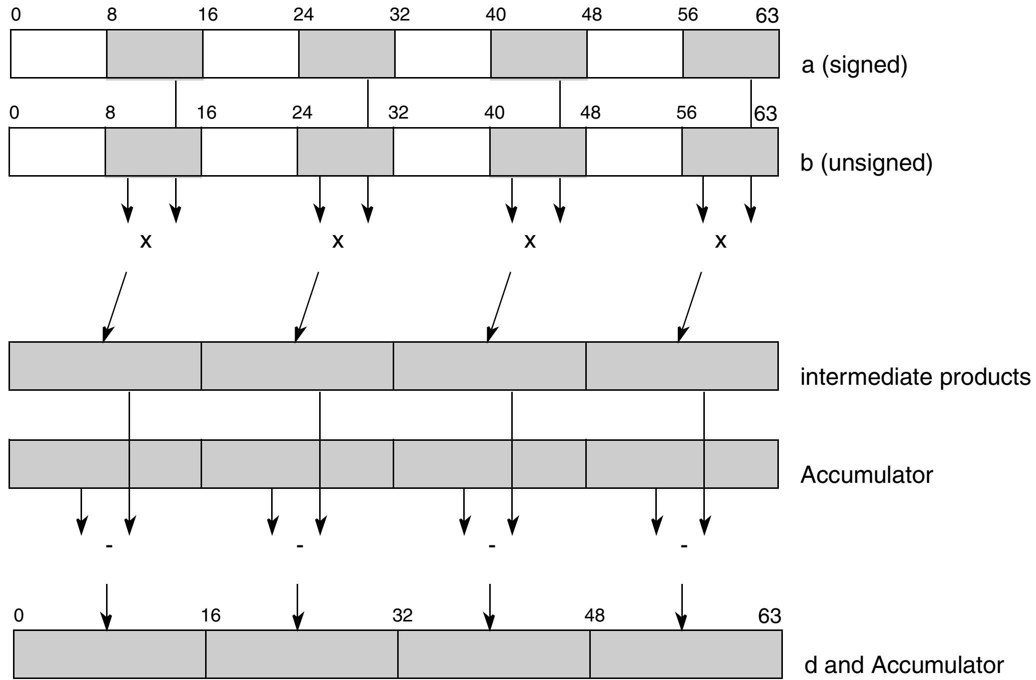


Figure 3-374. Vector Multiply Bytes Odd, Signed by Unsigned, Saturate, Integer and Accumulate Negative Half Words (__ev_mbosusianh)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evmbosusianh d,a,b**

__ev_mboumi[a] __ev_mboumi[a]

Vector Multiply Bytes Odd, Unsigned, Modulo, Integer (to Accumulator)

d = __ev_mboumi (a,b) (A=0)

d = __ev_mboumia (a,b) (A=1)

```
d0:15 ←a8:15 ×ui b8:15
d16:31 ←a24:31 ×ui b24:31
d32:47 ←a40:47 ×ui b40:47
d48:63 ←a56:63 ×ui b56:63

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

For each half word element in the accumulator, the corresponding odd-numbered byte unsigned integer elements in parameters **a** and **b** are multiplied. Each intermediate 16-bit product is added to the contents of the accumulator half words to form intermediate sums, which are placed into the corresponding half words in parameter **d**. If A = 1, the result in parameter **d** is also placed into the accumulator.

Other registers altered: ACC (if A=1)

Figure 3-375. Vector Multiply Bytes Odd, Unsigned, Modulo, Integer (to Accumulator) (__ev_mboumi)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evmboumi d,a,b**
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evmboumia d,a,b**

__ev_mboumiaah __ev_mboumiaah

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

d = __ev_mboumiaah (a,b)

```
temp0_{0:15} ←a_{8:15} ×_{ui} b_{8:15}
d_{0:15} ←temp0_{0:15} +ACC_{0:15}
temp1_{0:15} ←a_{24:31} ×_{ui} b_{24:31}
rD_{16:31} ←temp1_{0:15} +ACC_{16:31}
temp2_{0:15} ←a_{40:47} ×_{ui} b_{40:47}
d_{32:47} ←temp2_{0:15} +ACC_{32:47}
temp3_{0:15} ←a_{56:63} ×_{ui} b_{56:63}
d_{48:63} ←temp3_{0:15} +ACC_{48:63}

// update accumulator
ACC_{0:63} ←d_{0:63}
```

For each half word element in the accumulator, the corresponding odd-numbered byte unsigned integer elements in parameters **a** and **b** are multiplied. Each intermediate 16-bit product is added to the contents of the accumulator half words to form intermediate sums, which are placed into the corresponding parameter **d** half words and into the accumulator.

Other registers altered: ACC

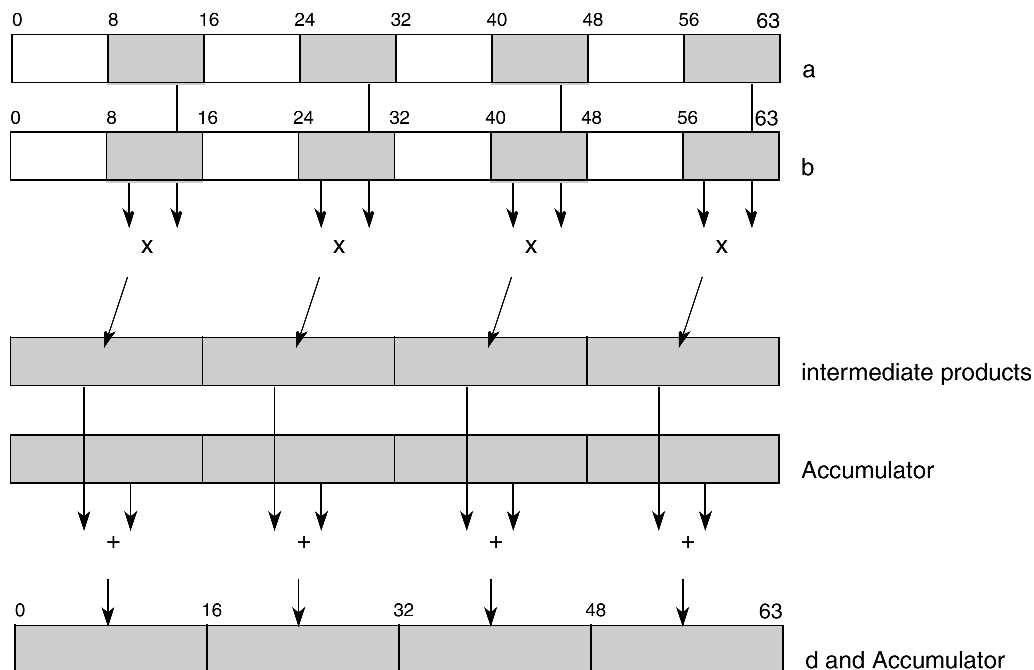


Figure 3-376. Vector Multiply Bytes Odd, Unsigned, Modulo, Integer and Accumulate Half Words (__ev_mboumiaah)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evmboumiaah d,a,b**

__ev_mboumianh __ev_mboumianh

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

d = __ev_mboumianh (a,b)

$temp0_{0:15} \leftarrow a_{8:15} \times_{ui} b_{8:15}$
$d_{0:15} \leftarrow ACC_{0:15} - temp0_{0:15}$

$temp1_{0:15} \leftarrow a_{24:31} \times_{ui} b_{24:31}$
$d_{16:31} \leftarrow ACC_{16:31} - temp1_{0:15}$

$temp2_{0:15} \leftarrow a_{40:47} \times_{ui} b_{40:47}$
$d_{32:47} \leftarrow ACC_{32:47} - temp2_{0:15}$

$temp3_{0:15} \leftarrow a_{56:63} \times_{ui} b_{56:63}$
$d_{48:63} \leftarrow ACC_{48:63} - temp3_{0:15}$

// update accumulator
$ACC_{0:63} \leftarrow d_{0:63}$

For each half word element in the accumulator, the corresponding odd-numbered byte unsigned integer elements in parameters **a** and **b** are multiplied. Each intermediate 16-bit product is subtracted from the contents of the accumulator half words to form intermediate differences, which are placed into the corresponding parameter **d** half words and into the accumulator.

Other registers altered: ACC

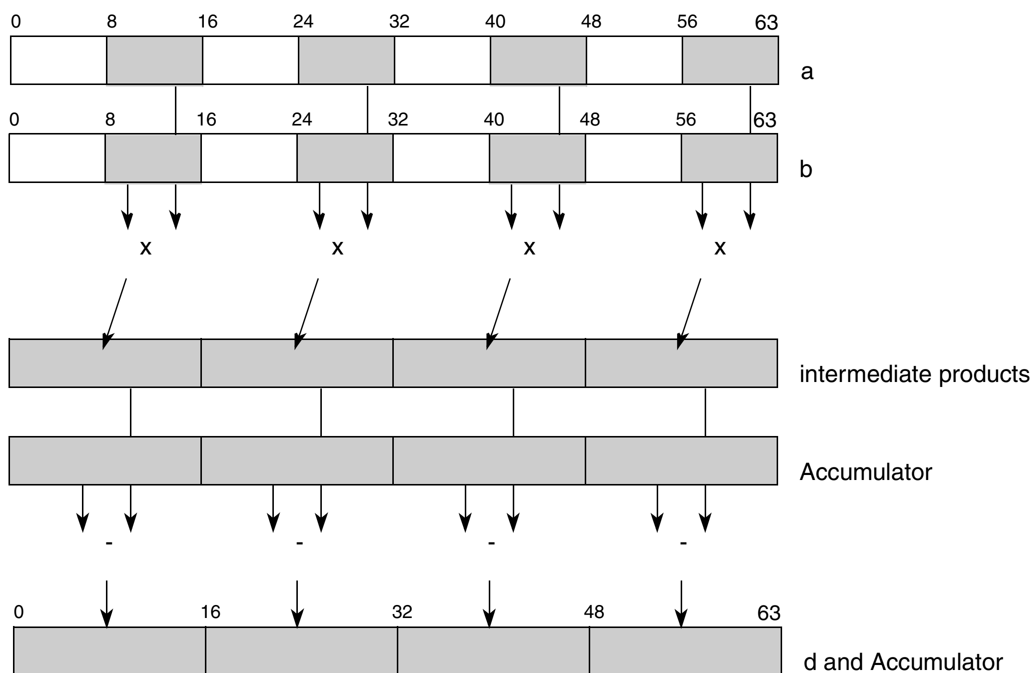


Figure 3-377. Vector Multiply Bytes Odd, Unsigned, Modulo, Integer and Accumulate Negative Half Words (__ev_mboumianh)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evmboumianh d,a,b**

__ev_mbousiaah __ev_mbousiaah

Vector Multiply Byte Odd, Unsigned, Saturate, Integer and Accumulate Half Words

d = __ev_mbousiaah (a,b)

$temp_{0:15} \leftarrow a_{8:15} \times_{ui} b_{8:15}$
$temp_{0:31} \leftarrow EXTZ(ACC_{0:15}) + EXTZ(temp_{0:15})$
$ovh0 \leftarrow temp_{15}$
$d_{0:15} \leftarrow SATURATE(ovh0, 0, 0xFFFF, 0xFFFF, temp_{16:31})$

$temp_{0:15} \leftarrow a_{24:31} \times_{ui} b_{24:31}$
$temp_{0:31} \leftarrow EXTZ(ACC_{16:31}) + EXTZ(temp_{0:15})$
$ovh1 \leftarrow temp_{15}$
$d_{16:31} \leftarrow SATURATE(ovh1, 0, 0xFFFF, 0xFFFF, temp_{16:31})$

$temp_{0:15} \leftarrow a_{40:47} \times_{ui} b_{40:47}$
$temp_{0:31} \leftarrow EXTZ(ACC_{32:47}) + EXTZ(temp_{0:15})$
$ovl0 \leftarrow temp_{15}$
$d_{32:47} \leftarrow SATURATE(ovl0, 0, 0xFFFF, 0xFFFF, temp_{16:31})$

$temp_{0:15} \leftarrow a_{56:63} \times_{ui} b_{56:63}$
$temp_{0:31} \leftarrow EXTZ(ACC_{48:63}) + EXTZ(temp_{0:15})$
$ovl1 \leftarrow temp_{15}$
$d_{48:63} \leftarrow SATURATE(ovl1, 0, 0xFFFF, 0xFFFF, temp_{16:31})$

// update accumulator
$ACC_{0:63} \leftarrow d_{0:63}$

// update SPEFSCR
$SPEFSCR_{OVH} \leftarrow ovh0 \mid ovh1$
$SPEFSCR_{OV} \leftarrow ovl0 \mid ovl1$
$SPEFSCR_{SOVH} \leftarrow SPEFSCR_{SOVH} \mid ovh0 \mid ovh1$
$SPEFSCR_{SOV} \leftarrow SPEFSCR_{SOV} \mid ovl0 \mid ovl1$

The corresponding odd-numbered byte unsigned integer elements in parameters **a** and **b** are multiplied, producing a 16-bit intermediate product. Each 16-bit product is then added to the corresponding half 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

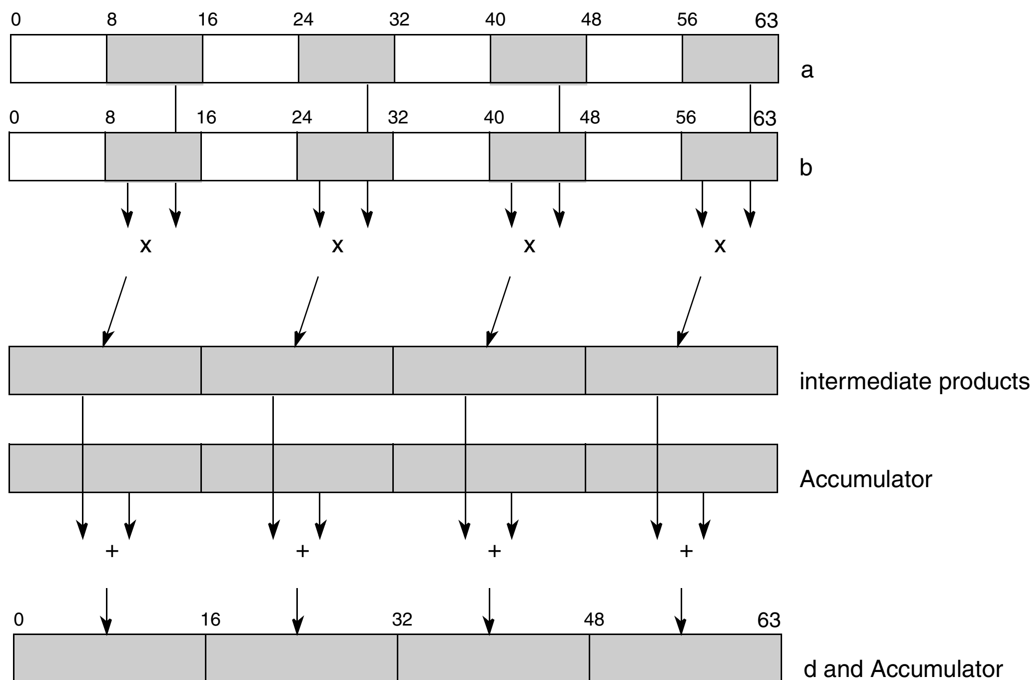


Figure 3-378. Vector Multiply Bytes Odd, Unsigned, Saturate, Integer and Accumulate Half Words (__ev_mbousiaah)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evmbousiaah d,a,b**

__ev_mbousianh __ev_mbousianh

Vector Multiply Byte Odd, Unsigned, Saturate, Integer and Accumulate Negative Half Words

d = __ev_mbousianh (a,b)

```
temp0:15 ←a8:15 ×ui b8:15
temp0:31 ←EXTZ(ACC0:15) - EXTZ(temp0:15)
ovh0 ←temp15
d0:15 ←SATURATE(ovh0, 0, 0x0000, 0x0000, temp16:31)

temp0:15 ←a24:31 ×ui b24:31
temp0:31 ←EXTZ(ACC16:31) - EXTZ(temp0:15)
ovh1 ←temp15
d16:31 ←SATURATE(ovh1, 0, 0x0000, 0x0000, temp16:31)

temp0:15 ←a40:47 ×ui b40:47
temp0:31 ←EXTZ(ACC32:47) - EXTZ(temp0:15)
ovl0 ←temp15
d32:47 ←SATURATE(ovl0, 0, 0x0000, 0x0000, temp16:31)

temp0:15 ←a56:63 ×ui b56:63
temp0:31 ←EXTZ(ACC48:63) - EXTZ(temp0:15)
ovl1 ←temp15
d48:63 ←SATURATE(ovl1, 0, 0x0000, 0x0000, temp16:31)

// update accumulator
ACC0:63 ←d0:63

// update SPEFSCR
SPEFSCROVH ←ovh0 | ovh1
SPEFSCROV ←ovl0 | ovl1
SPEFSCRSOVH ←SPEFSCRSOVH | ovh0 | ovh1
SPEFSCRSOV ←SPEFSCRSOV | ovl0 | ovl1
```

The corresponding odd-numbered byte unsigned integer elements in parameters **a** and **b** are multiplied, producing a 16-bit intermediate product. Each 16-bit product is then subtracted from the corresponding half 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

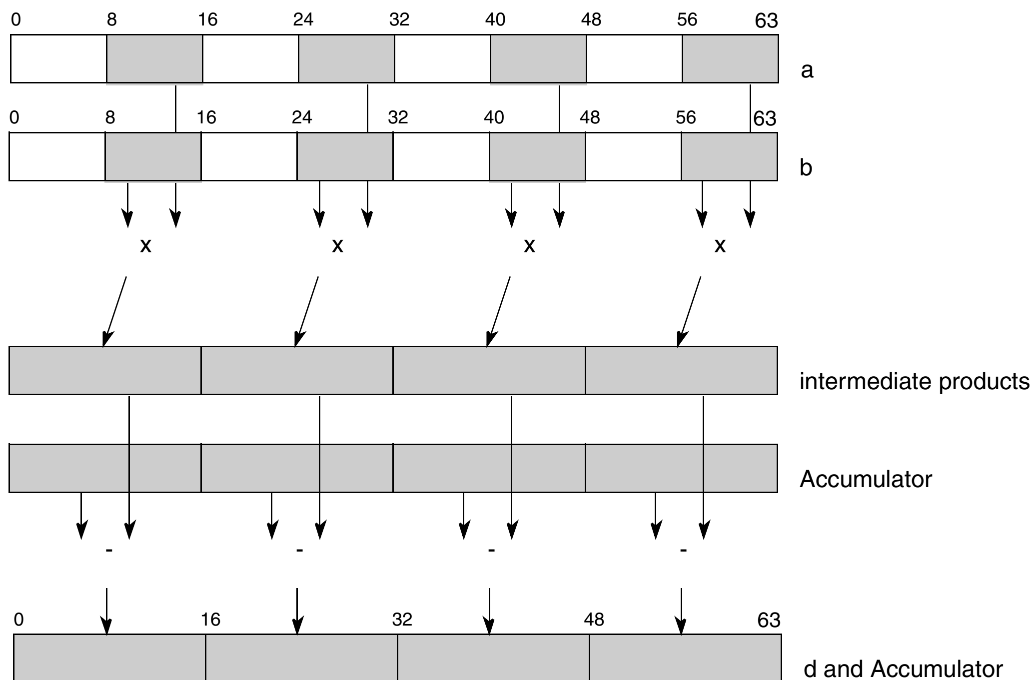


Figure 3-379. Vector Multiply Bytes Odd, Unsigned, Saturate, Integer and Accumulate Negative Half Words (__ev_mbousianh)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evmbousianh 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 3-380.

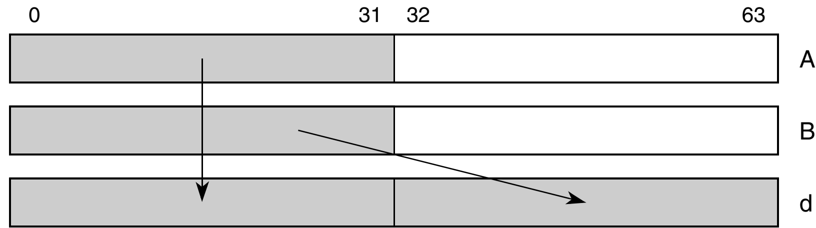


Figure 3-380. High-Order Element Merging (__ev_mergehi)

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 3-381.

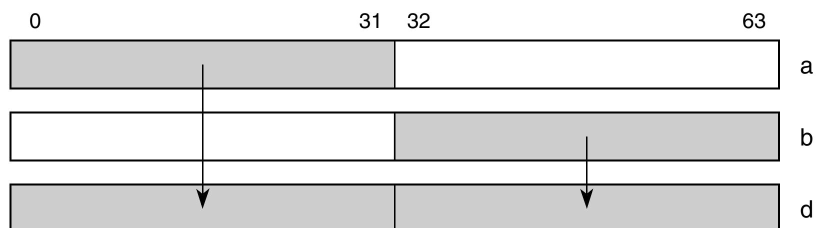


Figure 3-381. High-Order Element Merging (__ev_mergehilo)

Application note: With appropriate specification of parameters **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 3-382.

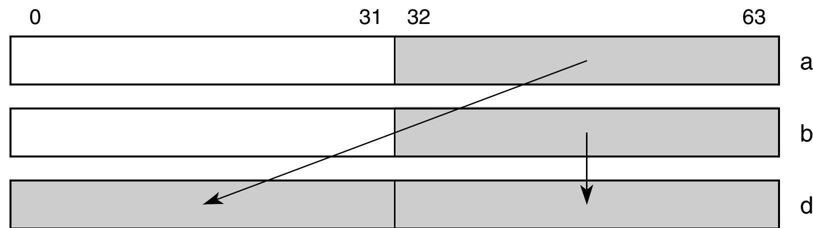


Figure 3-382. Low-Order Element Merging (__ev_mergelo)

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 3-383.

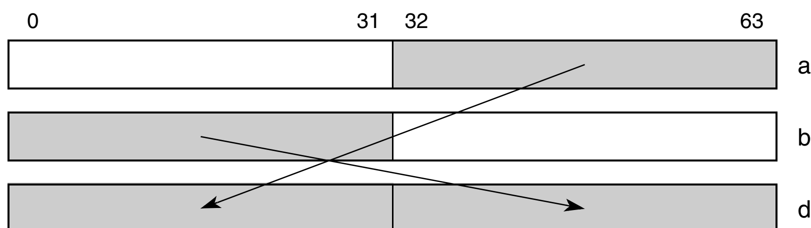


Figure 3-383. Low-Order Element Merging (__ev_mergelohi)

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)

$temp_{0:31} \leftarrow a_{32:47} \times_{sf} b_{32:47}$
$temp_{0:63} \leftarrow EXTS(temp_{0:31})$
$d_{0:63} \leftarrow ACC_{0:63} + temp_{0:63}$
// update accumulator
$ACC_{0:63} \leftarrow d_{0:63}$

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. If the two input operands are both -1.0, the intermediate product is represented as +1.0.

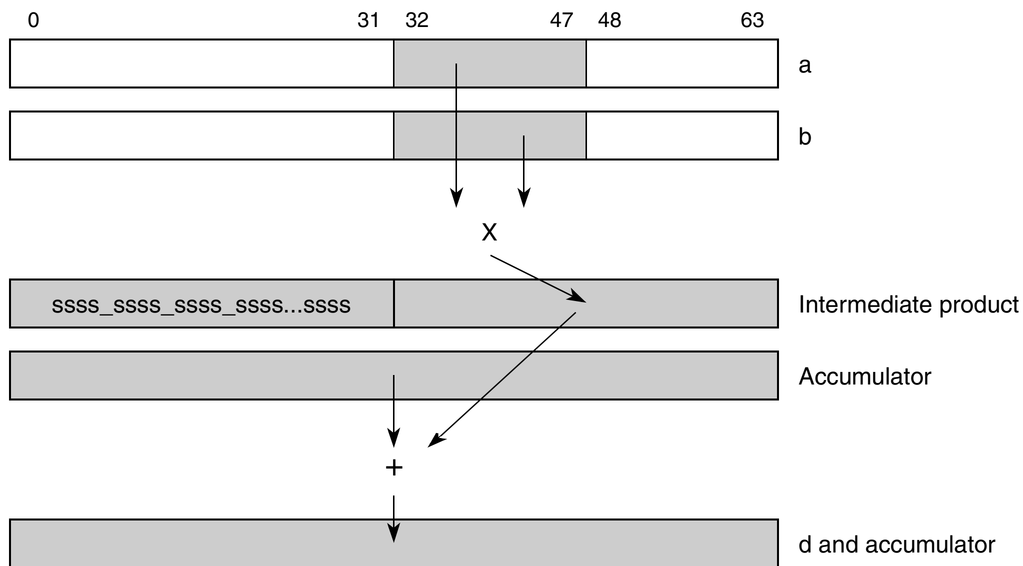


Figure 3-384. __ev_mhegsmfaa (Even Form)

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. If the two input operands are both -1.0, the intermediate product is represented as +1.0.

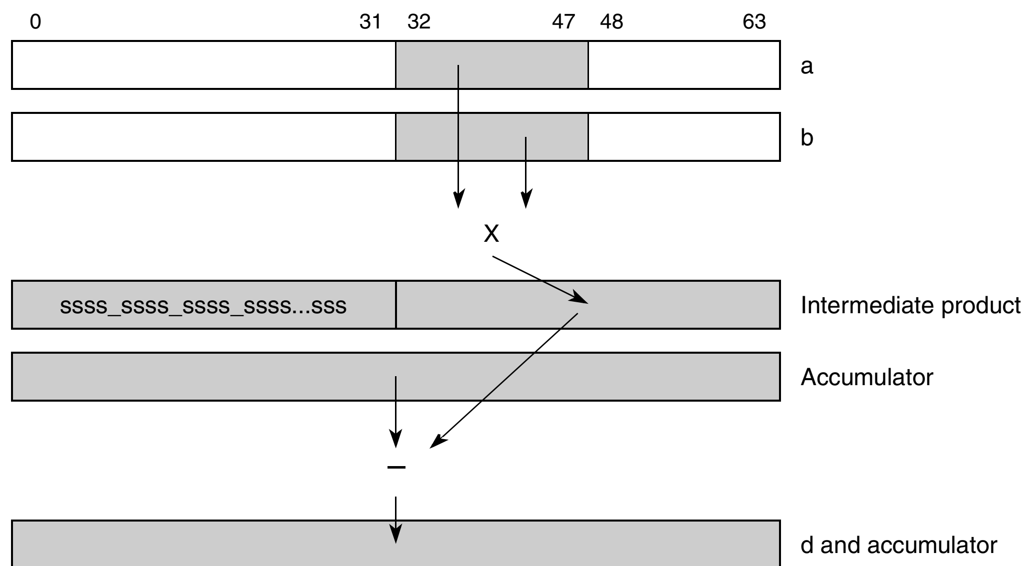


Figure 3-385. __ev_mhegsmfan (Even Form)

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.

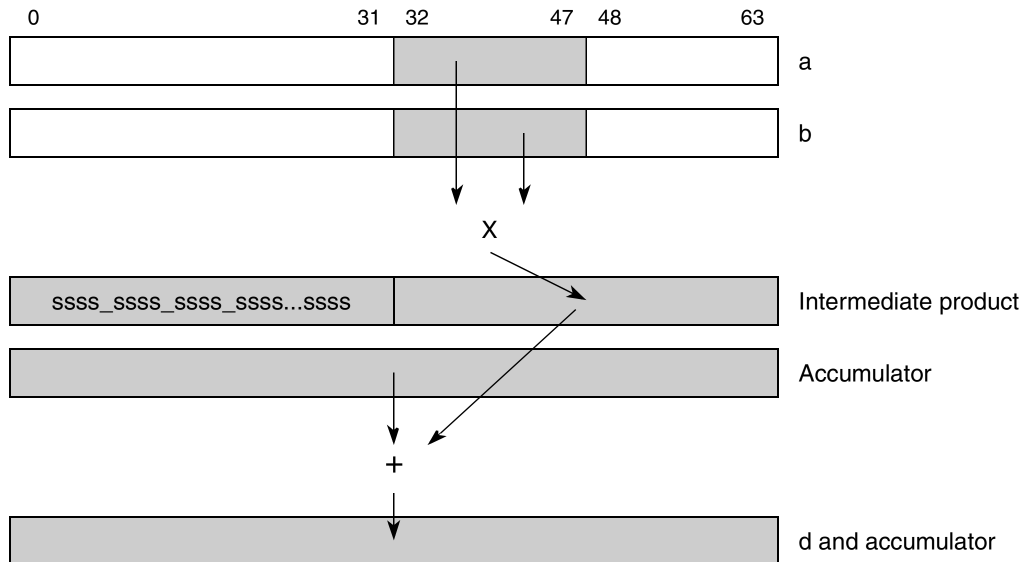


Figure 3-386. __ev_mhegsmiaa (Even Form)

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)

$$temp_{0:31} \leftarrow a_{32:47} \times_{si} b_{32:47}$$
$$temp_{0:63} \leftarrow EXTS(temp_{0:31})$$
$$d_{0:63} \leftarrow ACC_{0:63} - temp_{0:63}$$
```
// update accumulator
```
$$ACC_{0:63} \leftarrow d_{0:63}$$

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.

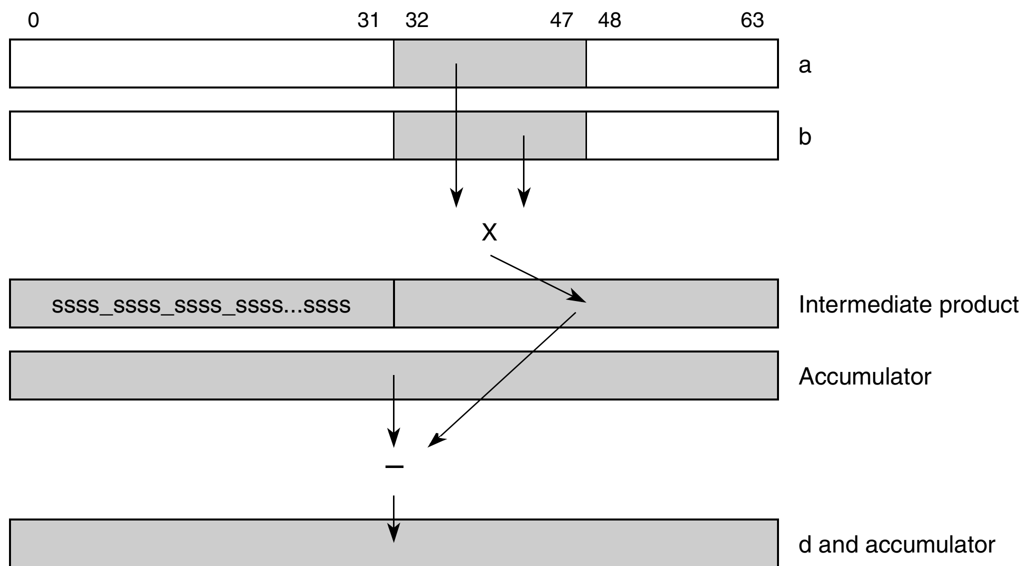


Figure 3-387. __ev_mhegsmian (Even Form)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evmhegsmian 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.

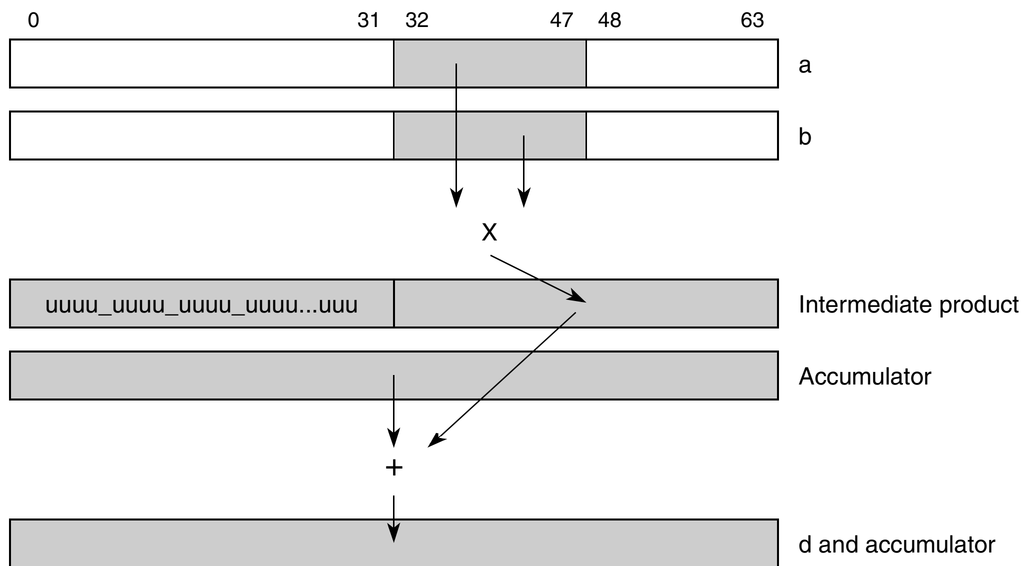


Figure 3-388. __ev_mhegumiaa (Even Form)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evmhegumiaa d,a,b**

__ev_mhegumian __ev_mhegumian

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

d = __ev_mhegumian (a,b)

$$temp_{0:31} \leftarrow a_{32:47} \times_{ui} b_{32:47}$$
$$temp_{0:63} \leftarrow EXTZ(temp_{0:31})$$
$$d_{0:63} \leftarrow ACC_{0:63} - temp_{0:63}$$
```
// update accumulator
```
$$ACC_{0:63} \leftarrow d_{0:63}$$

The corresponding low even-numbered unsigned integer 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. Any overflow of the 64-bit difference is not recorded into the SPEFSCR.

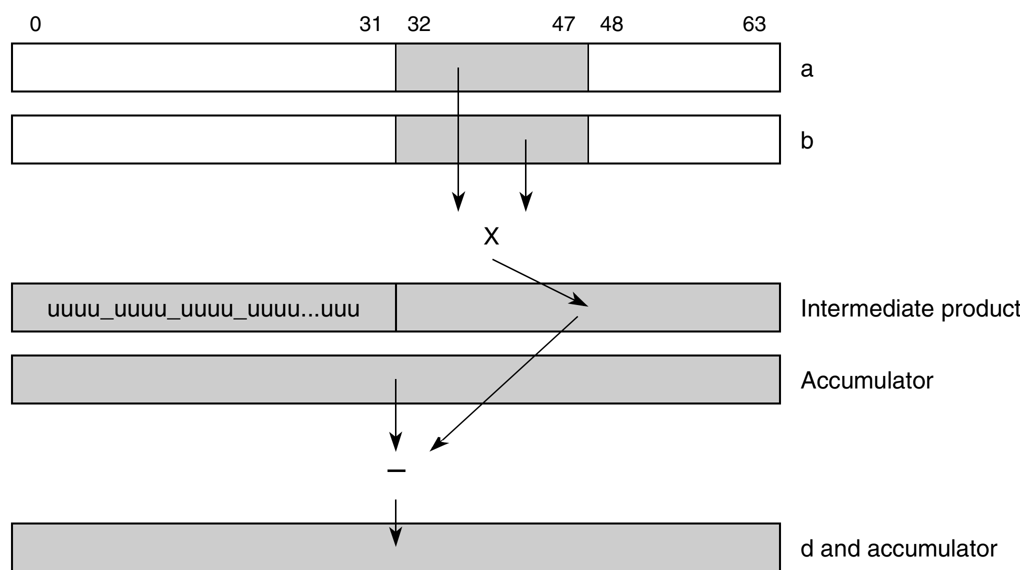


Figure 3-389. __ev_mhegumian (Even Form)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evmhegumian d,a,b**

__ev_mhesmf[a] __ev_mhesmf[a]

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)

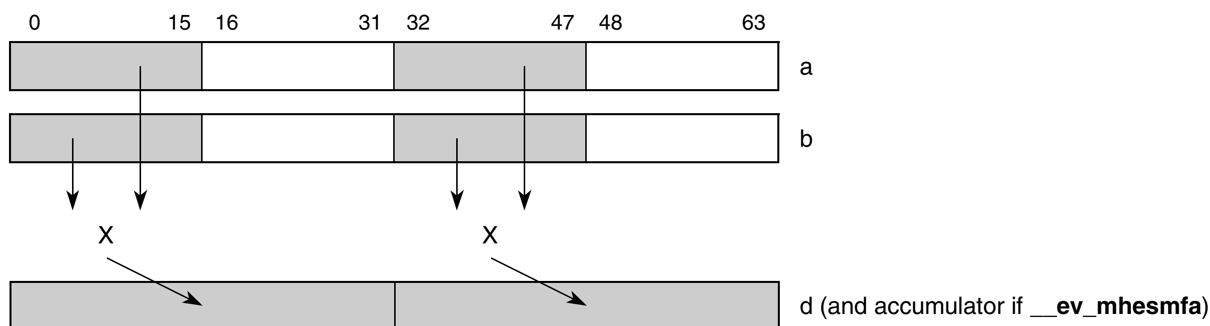


Figure 3-390. Even Multiply of Two Signed Modulo Fractional Elements (to Accumulator) (__ev_mhesmf[a])

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

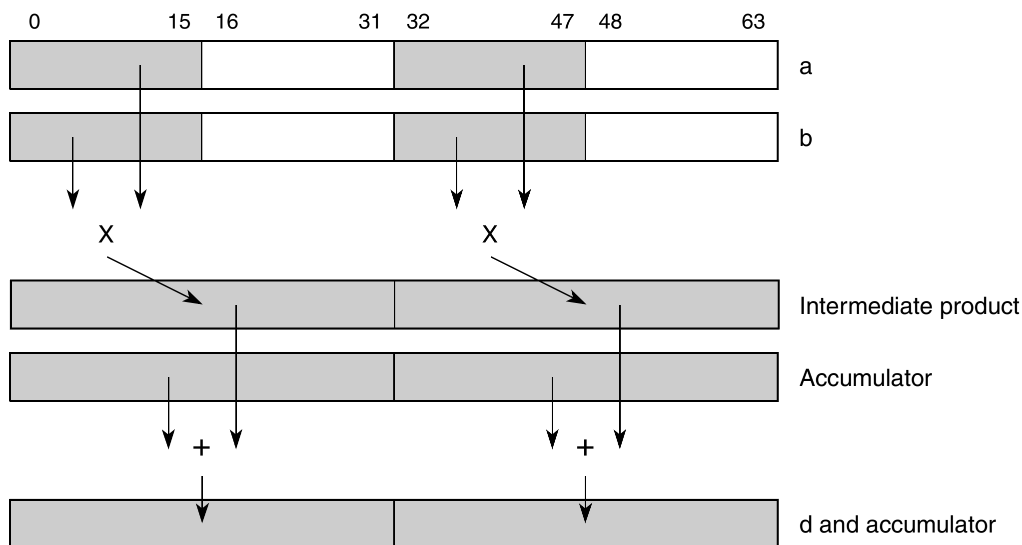


Figure 3-391. Even Form of Vector Half-Word Multiply (__ev_mhesmfaaw)

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

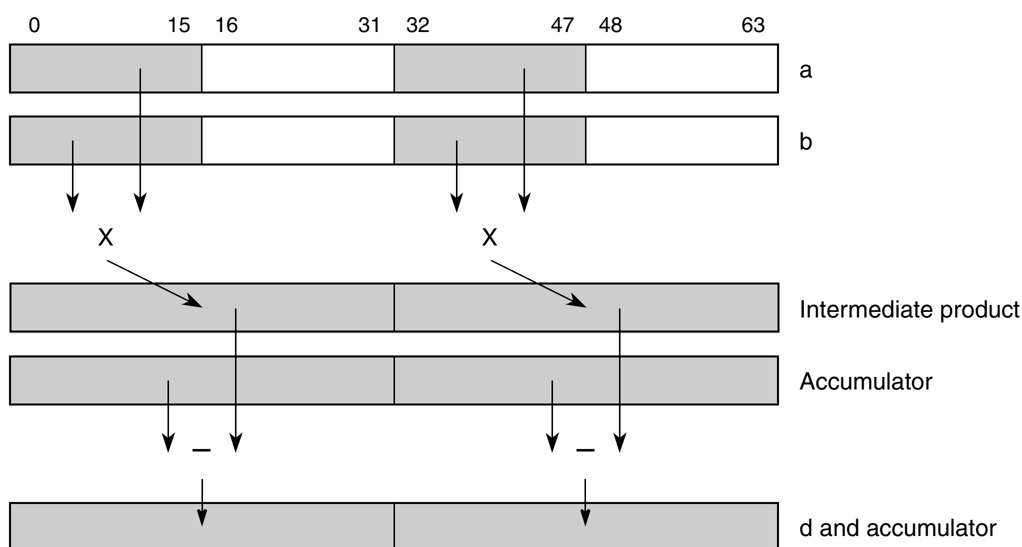


Figure 3-392. Even Form of Vector Half-Word Multiply (__ev_mhesmfanw)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evmhesmfanw d,a,b**

__ev_mhesmi[a] __ev_mhesmi[a]

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 3-393. Even Form for Vector Multiply (to Accumulator) (__ev_mhesmi[a])

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

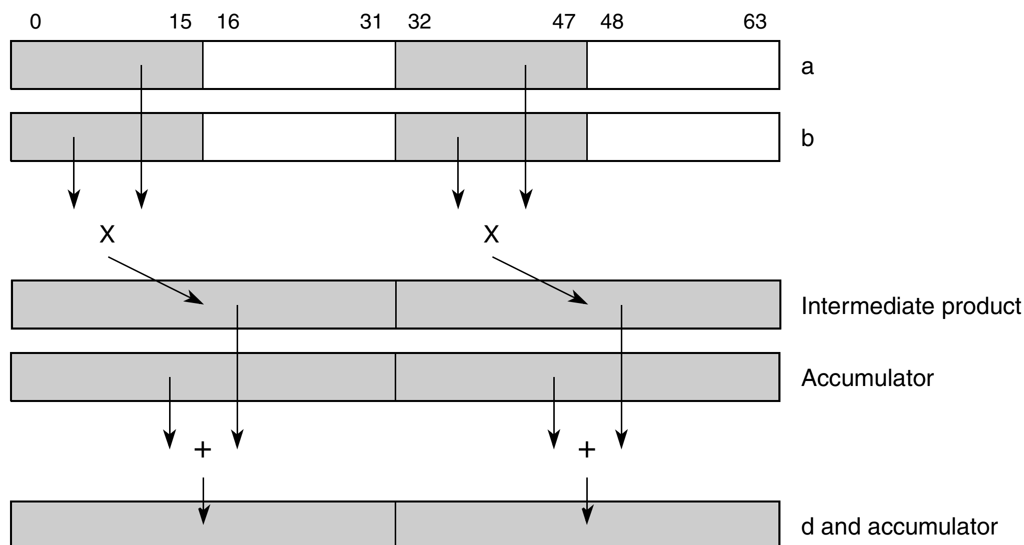


Figure 3-394. Even Form of Vector Half-Word Multiply (__ev_mhesmiaaw)

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

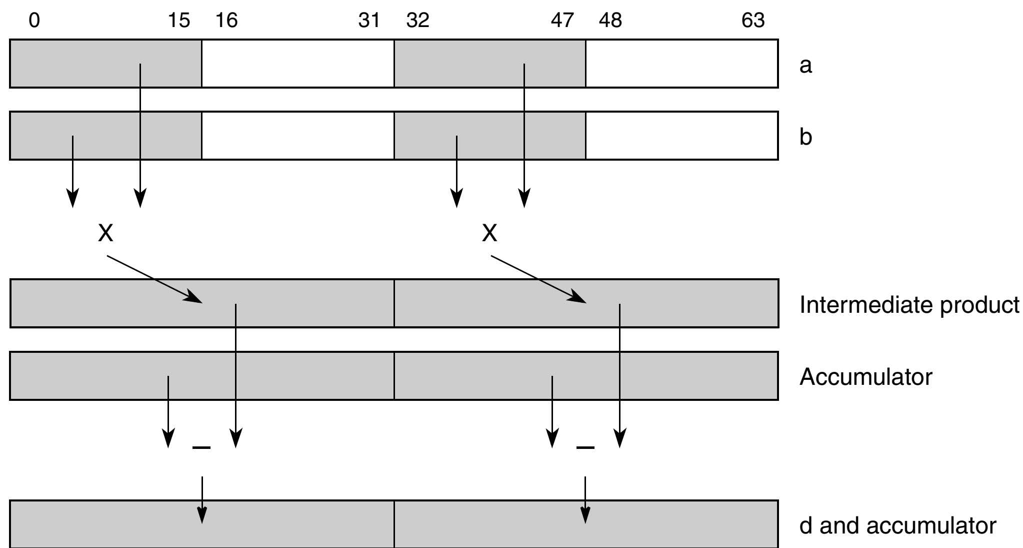


Figure 3-395. Even Form of Vector Half-Word Multiply (__ev_mhesmianw)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evmhesmianw d,a,b**

__ev_mhessf[a] __ev_mhessf[a]

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)

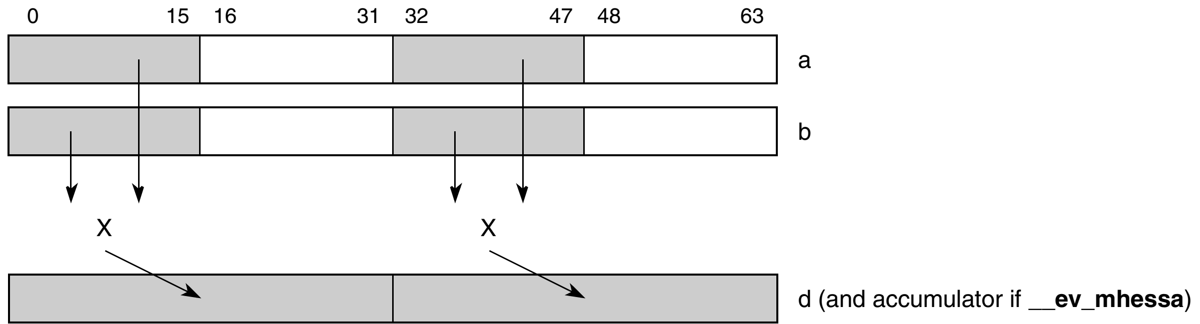


Figure 3-396. Even Multiply of Two Signed Saturate Fractional Elements (to Accumulator) (__ev_mhessf[a])

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 ←ovh | movh
SPEFSCROV ←ovl | 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

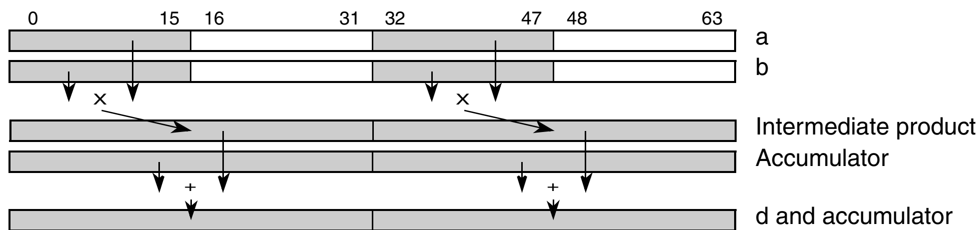


Figure 3-397. Even Form of Vector Half-Word Multiply (__ev_mhessfaaw)

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 ←ovh | movh
SPEFSCROV ←ovl | 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

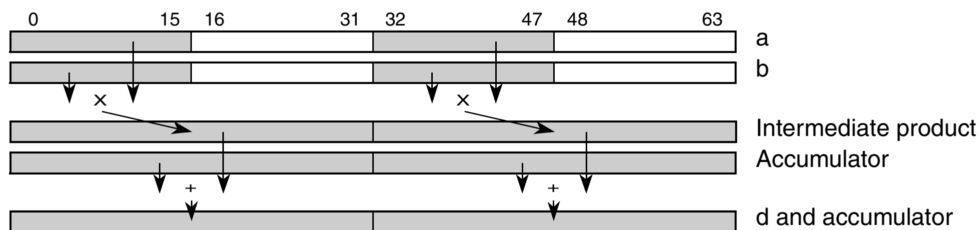


Figure 3-398. Even Form of Vector Half-Word Multiply (__ev_mhessfanw)

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 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

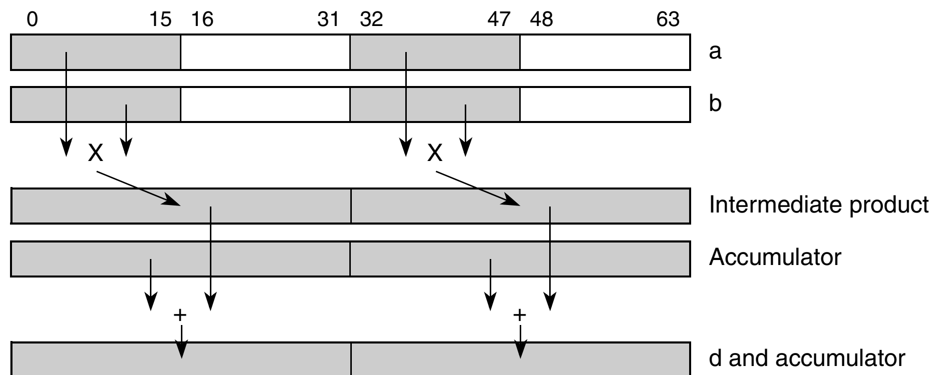


Figure 3-399. Even Form of Vector Half-Word Multiply (__ev_mhessiaaw)

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 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

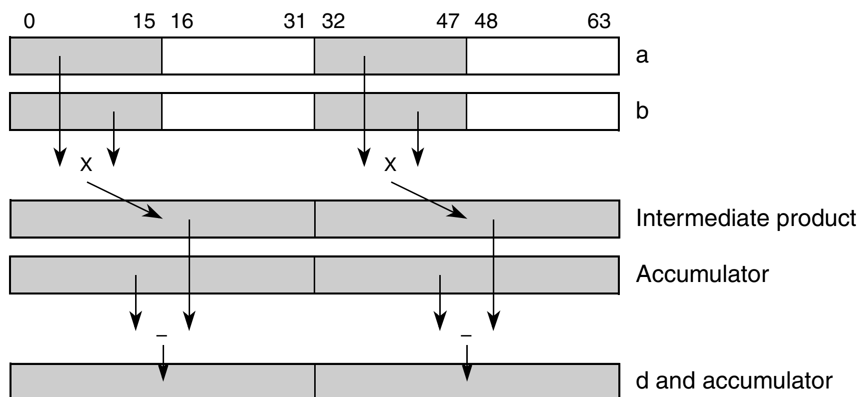


Figure 3-400. Even Form of Vector Half-Word Multiply (__ev_mhessianw)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evmhessianw d,a,b**

__ev_mhesumi[a] __ev_mhesumi[a]

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

d = __ev_mhesumi (a,b) (A = 0)

d = __ev_mhesumia (a,b) (A = 1)

```
// high
d0:31 ←(a0:15 ×su b0:15)

// low
d32:63←(a32:47 ×su b32:47)

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

The corresponding even-numbered half word signed integer elements in parameter **a** and unsigned integer elements in parameter **b** are multiplied then 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)

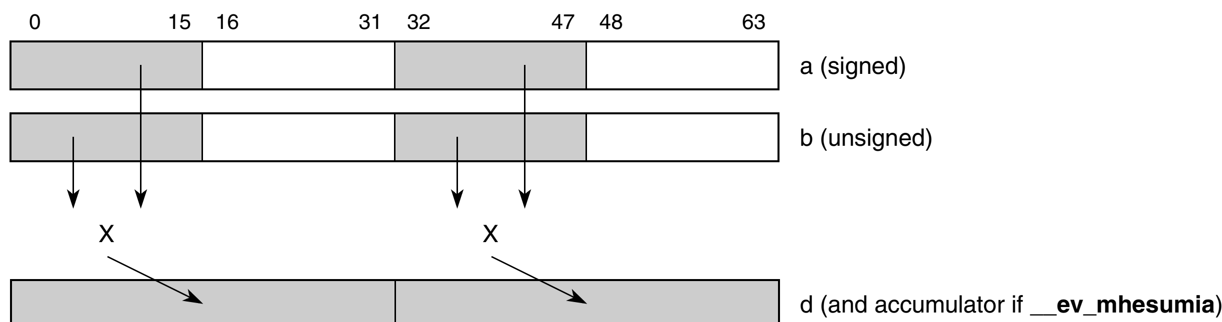


Figure 3-401. Even Multiply of Signed and Unsigned Modulo Integer Elements (to Accumulator) (__ev_mhesumi)

A	d	a	b	Maps to
A = 0	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evmhesumi d,a,b**
A = 1	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evmhesumia d,a,b**

__ev_mhesumiaaw __ev_mhesumiaaw

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

d = __ev_mhesumiaaw (a,b)

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

// low
temp0:31 ← (a32:47 ×su 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 element in parameter **a** and unsigned integer element in parameter **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

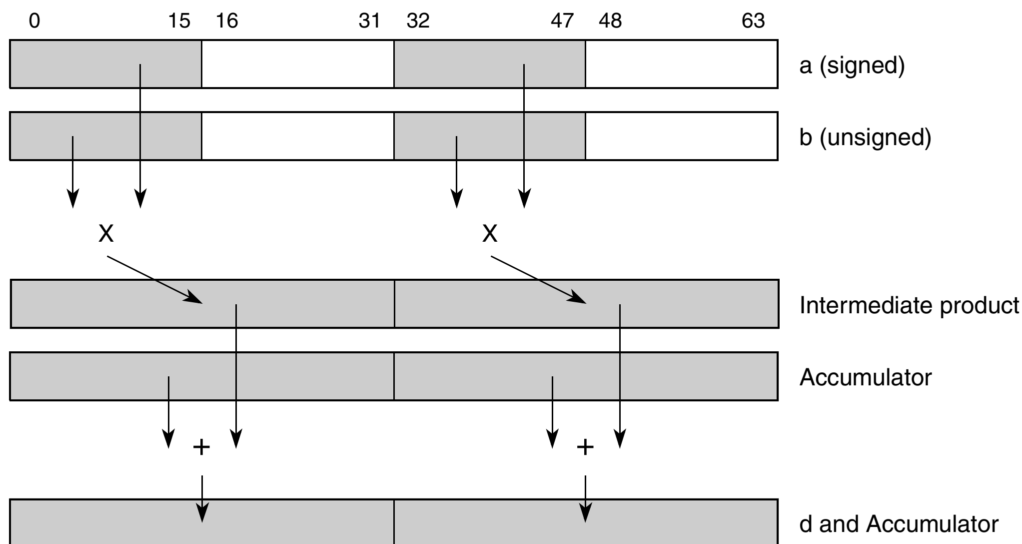


Figure 3-402. Even Form of Vector Half Word Multiply Signed by Unsigned Integer and Accumulate (__ev_mhesumiaaw)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evmhesumiaaw d,a,b**

__ev_mhesumianw __ev_mhesumianw

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

d = __ev_mhesumianw (a,b)

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

// low
temp0:31 ← a32:47 ×su 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 element in parameter **a** and unsigned integer element in parameter **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

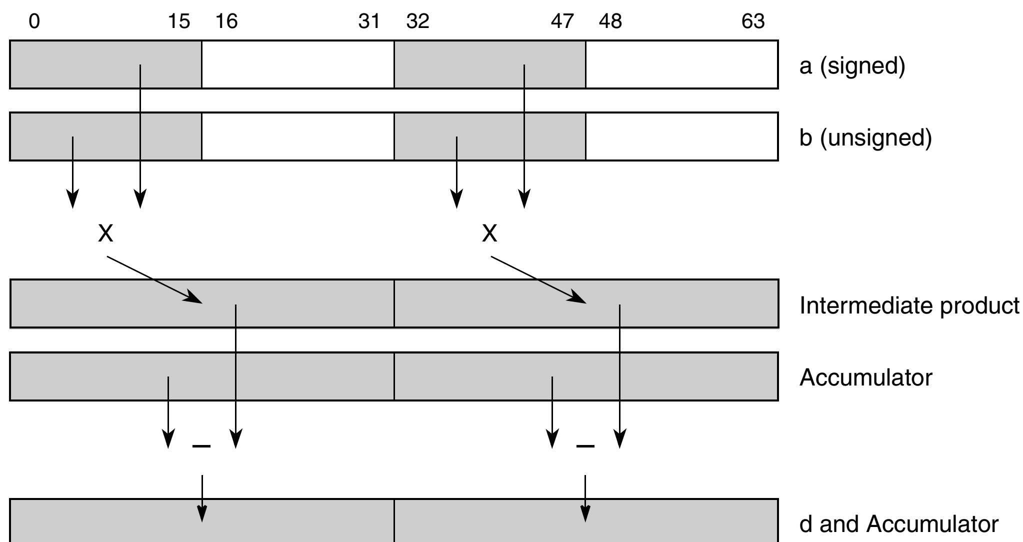


Figure 3-403. Even Form of Vector Half Word Multiply Signed by Unsigned Integer and Accumulate Negated (__ev_mhesumianw)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evmhesumianw d,a,b**

__ev_mhesusiaaw __ev_mhesusiaaw

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

d = __ev_mhesusiaaw (a,b)

```
// high
temph0:31 ←a0:15 ×su b0:15
temp0:63 ←EXTS(ACC0:31) + EXTS(temph0:31)
ovh ←(temp31 ⊕ temp32)
d0:31 ←SATURATE(ovh, temp31, 0x8000_0000, 0x7FFF_FFFF, temp32:63)
// low
templ0:31 ←a32:47 ×su b32:47
temp0:63 ←EXTS(ACC32:63) + EXTS(templ0:31)
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 element in parameter **a** and unsigned integer element in parameter **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 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

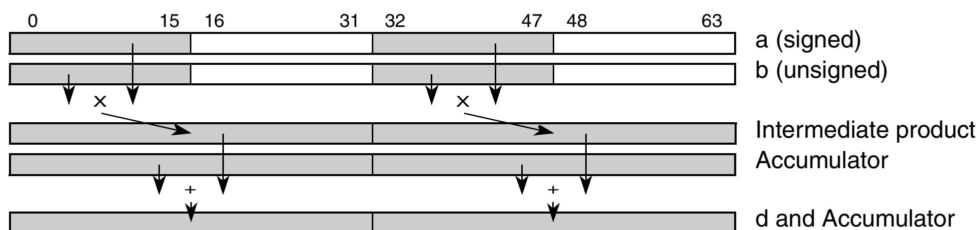


Figure 3-404. Even Form of Vector Half Word Multiply Signed by Unsigned Saturate Integer and Accumulate (__ev_mhesusiaaw)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evmhesusiaaw d,a,b**

__ev_mhesusianw __ev_mhesusianw

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

d = __ev_mhesusianw (a,b)

```
// high
temph0:31 ←a0:15 ×su b0:15
temp0:63 ←EXTS(ACC0:31) - EXTS(temph0:31)
ovh ←(temp31 ⊕ temp32)
d0:31 ←SATURATE(ovh, temp31, 0x8000_0000, 0x7FFF_FFFF, temp32:63)
// low
templ0:31 ←a32:47 ×su b32:47
temp0:63 ←EXTS(ACC32:63) - EXTS(templ0:31)
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 element in parameter **a** and unsigned integer element in parameter **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 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

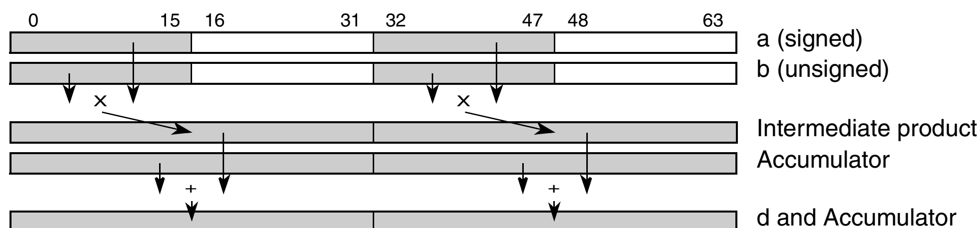


Figure 3-405. Even Form of Vector Half Word Multiply Signed by Unsigned Integer and Accumulate Negated (__ev_mhesusianw)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evmhesusianw d,a,b**

__ev_mheumi[a] __ev_mheumi[a]

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 3-406. Vector Multiply Half Words, Even, Unsigned, Modulo, Integer (to Accumulator) (__ev_mheumi[a])

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_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

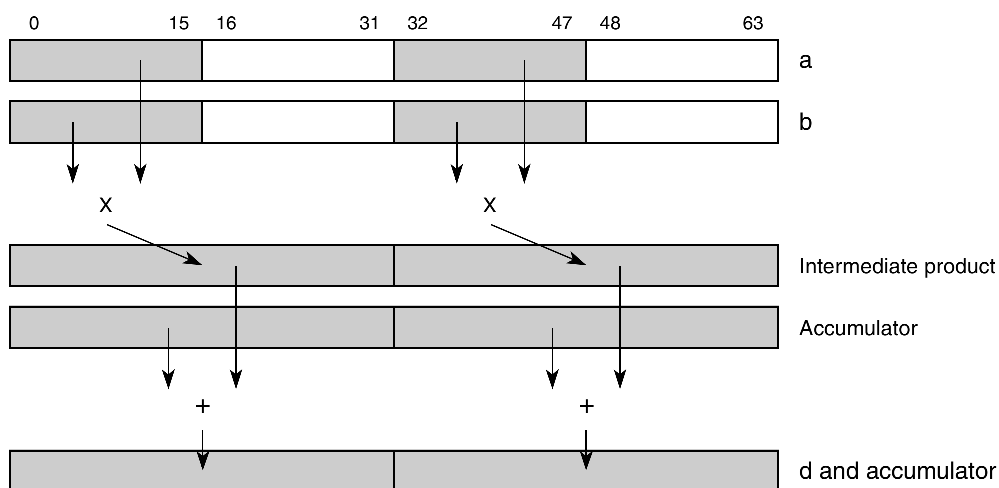


Figure 3-407. Even Form of Vector Half-Word Multiply (__ev_mheumiaaw)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evmheumiaaw d,a,b**

__ev_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

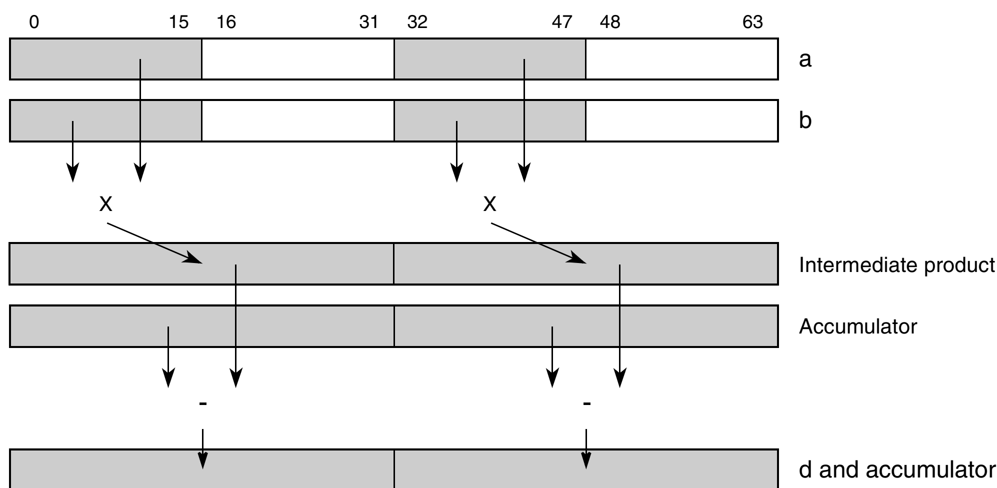


Figure 3-408. Even Form of Vector Half-Word Multiply (__ev_mheumianw)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evmheumianw d,a,b**

__ev_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

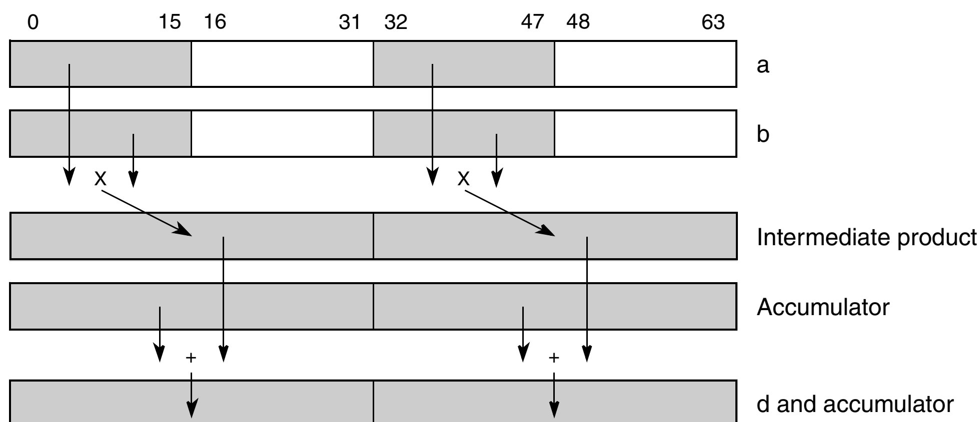


Figure 3-409. Even Form of Vector Half-Word Multiply (__ev_mheusiaaw)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evmheusiaaw d,a,b**

__ev_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

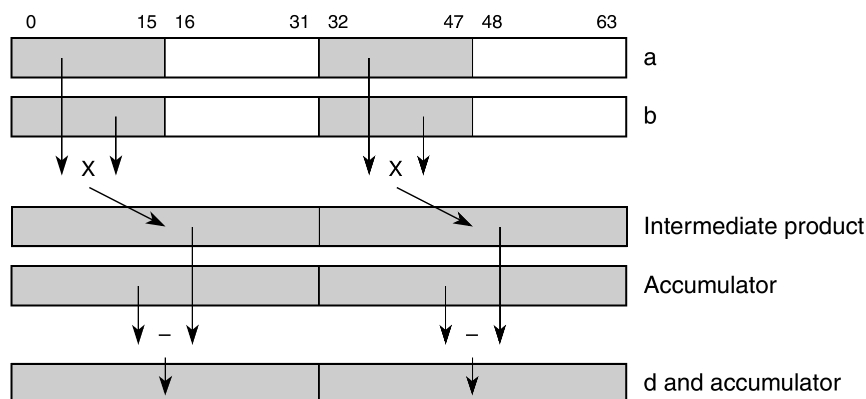


Figure 3-410. Even Form of Vector Half-Word Multiply (__ev_mheusianw)

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)

$$temp_{0:31} \leftarrow a_{48:63} \times_{sf} b_{48:63}$$
$$temp_{0:63} \leftarrow EXTS(temp_{0:31})$$
$$d_{0:63} \leftarrow ACC_{0:63} + temp_{0:63}$$
// update accumulator
$$ACC_{0:63} \leftarrow d_{0:63}$$

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.

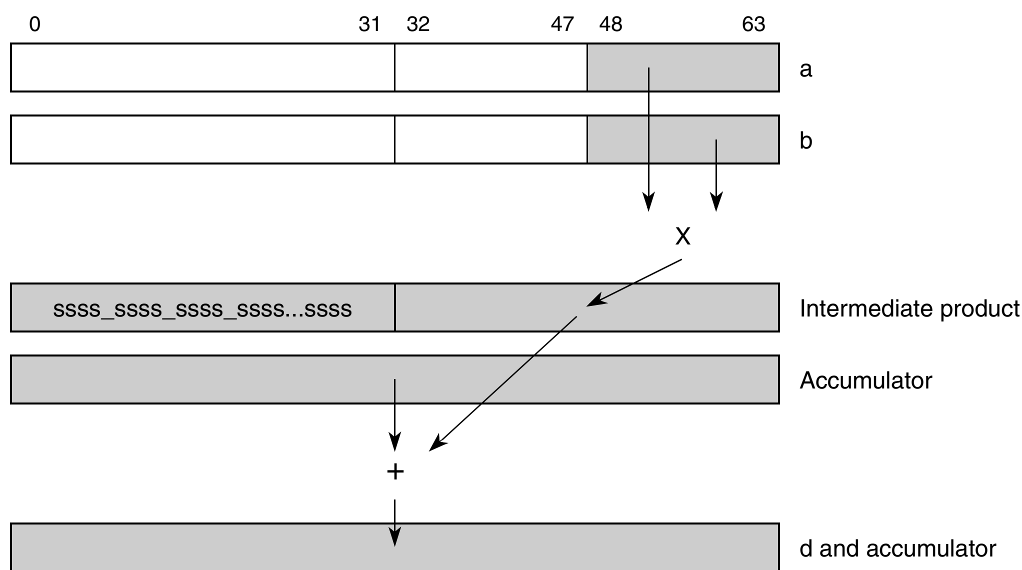


Figure 3-411. __ev_mhogsmfaa (Odd Form)

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)

$$temp_{0:31} \leftarrow a_{48:63} \times_{sf} b_{48:63}$$
$$temp_{0:63} \leftarrow EXTS(temp_{0:31})$$
$$d_{0:63} \leftarrow ACC_{0:63} - temp_{0:63}$$
// update accumulator
$$ACC_{0:63} \leftarrow d_{0:63}$$

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. If the two input operands are both -1.0, the intermediate product is represented as +1.0.

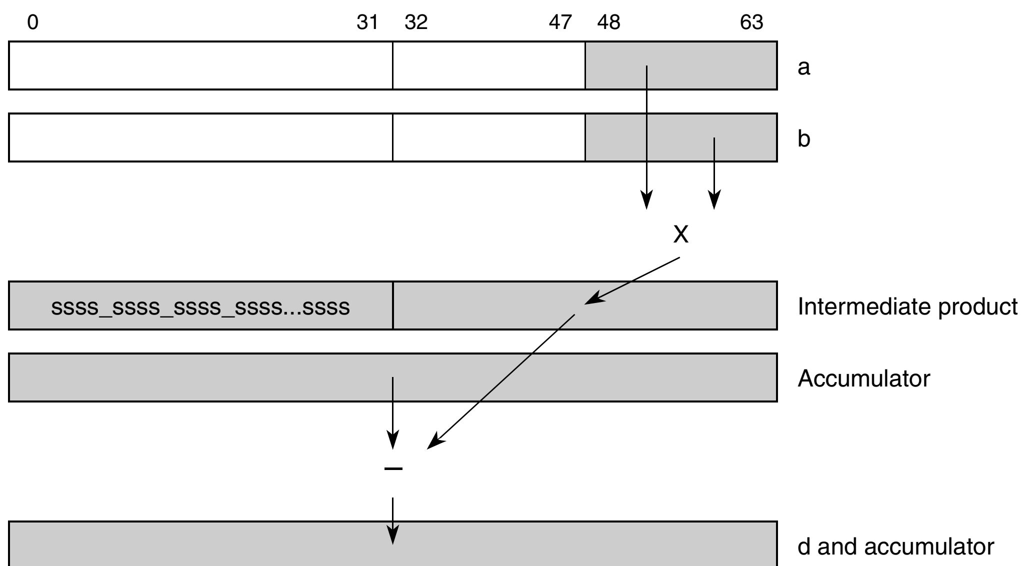


Figure 3-412. __ev_mhogsmfan (Odd Form)

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)

$$temp_{0:31} \leftarrow a_{48:63} \times_{si} b_{48:63}$$
$$temp_{0:63} \leftarrow EXTS(temp_{0:31})$$
$$d_{0:63} \leftarrow ACC_{0:63} + temp_{0:63}$$
// update accumulator
$$ACC_{0:63} \leftarrow d_{0:63}$$

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.

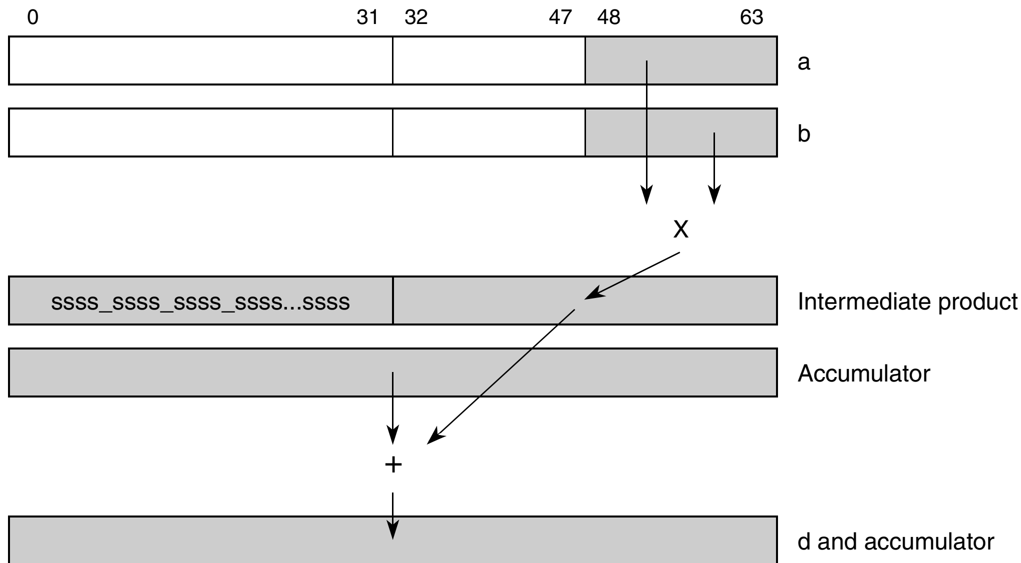


Figure 3-413. __ev_mhogsmiaa (Odd Form)

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.

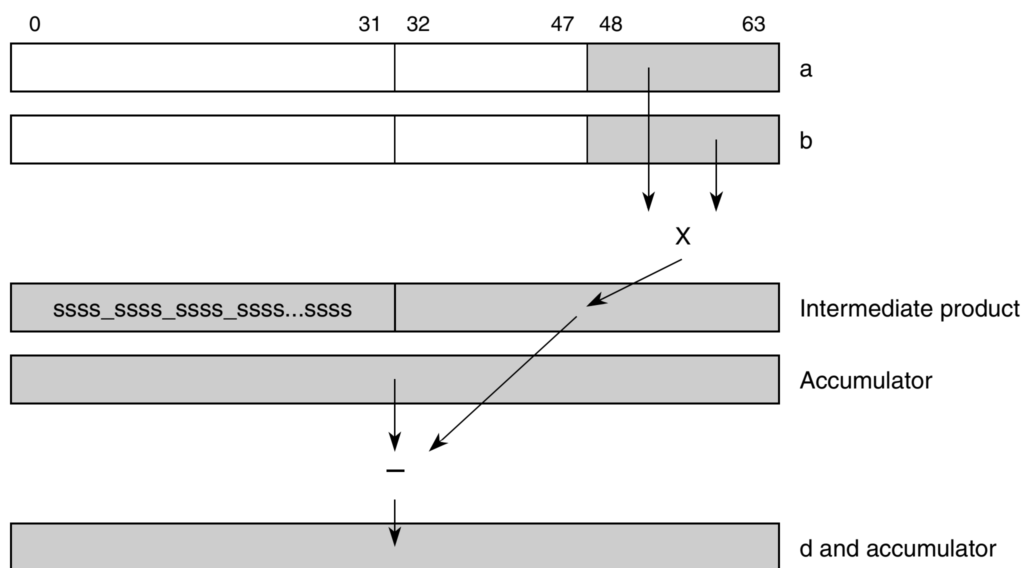


Figure 3-414. __ev_mhogsmian (Odd Form)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evmhogsmian d,a,b**

__ev_mhogumiaa __ev_mhogumiaa

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

d = __ev_mhogumiaa (a,b)

$temp_{0:31} \leftarrow a_{48:63} \times_{ui} b_{48:63}$
$temp_{0:63} \leftarrow EXTZ(temp_{0:31})$
$d_{0:63} \leftarrow ACC_{0:63} + temp_{0:63}$
// update accumulator
$ACC_{0:63} \leftarrow d_{0:63}$

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.

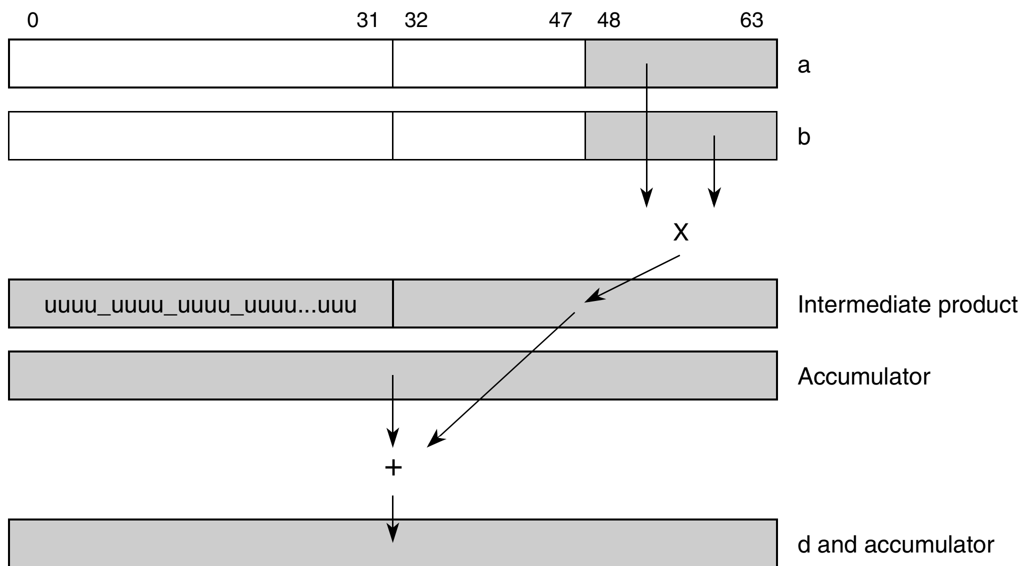


Figure 3-415. __ev_mhogumiaa (Odd Form)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evmhogumiaa 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.

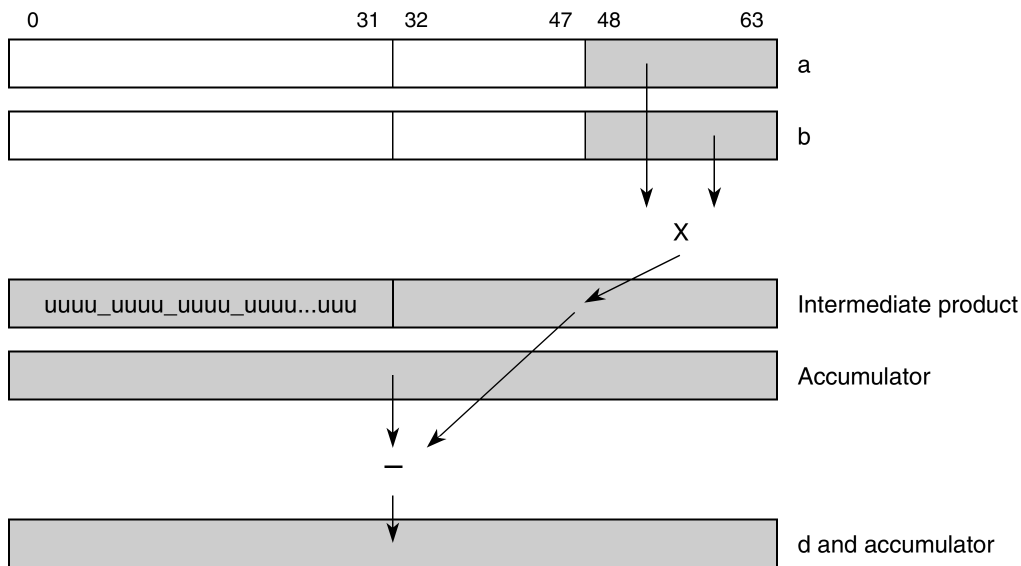


Figure 3-416. __ev_mhogumian (Odd Form)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evmhogumian d,a,b**

__ev_mhosmf[a] __ev_mhosmf[a]

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)

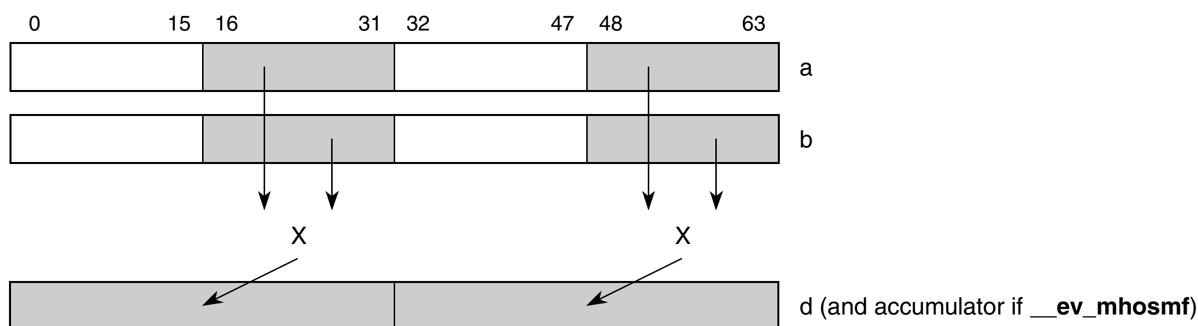


Figure 3-417. Vector Multiply Half Words, Odd, Signed, Modulo, Fractional (to Accumulator) (__ev_mhosmf[a])

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

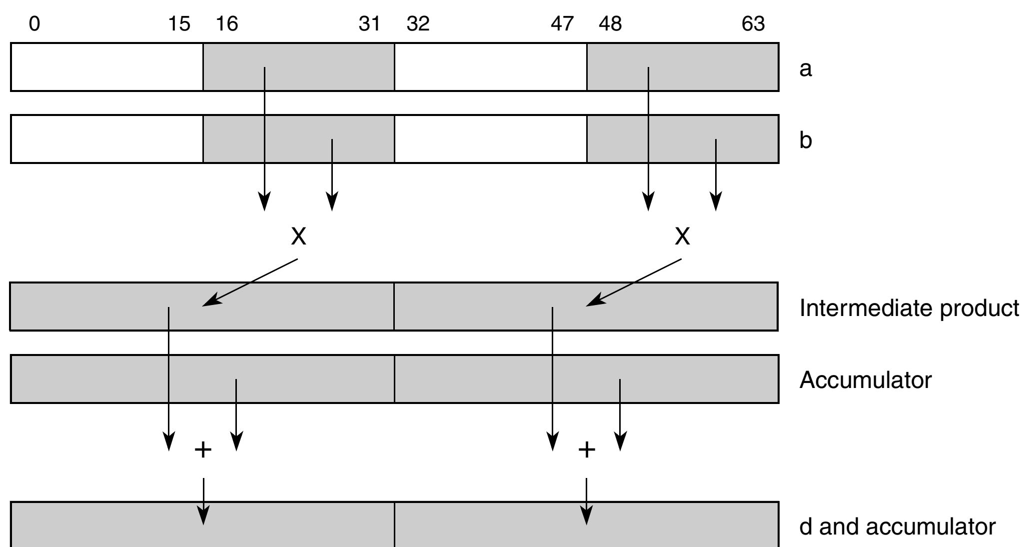


Figure 3-418. Odd Form of Vector Half-Word Multiply (__ev_mhosmfaaw)

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

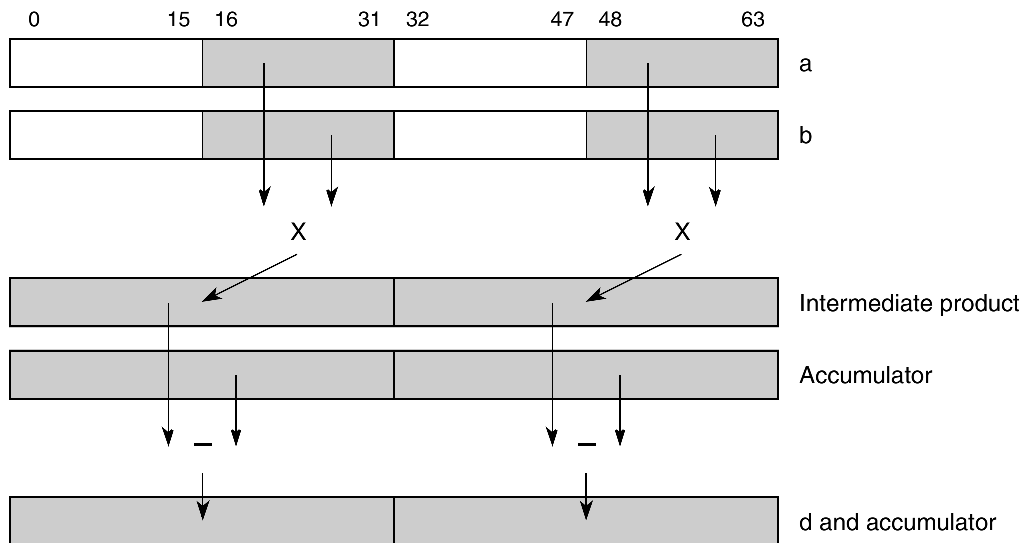


Figure 3-419. Odd Form of Vector Half-Word Multiply (__ev_mhosmfanw)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evmhosmfanw d,a,b**

__ev_mhosmi[a] __ev_mhosmi[a]

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)

0 15 16 31 32 47 48 63

a

b

X X

d (and accumulator if **__ev_mhosmia**)

Figure 3-420. Vector Multiply Half Words, Odd, Signed, Modulo, Integer (to Accumulator) (__ev_mhosmi[a])

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

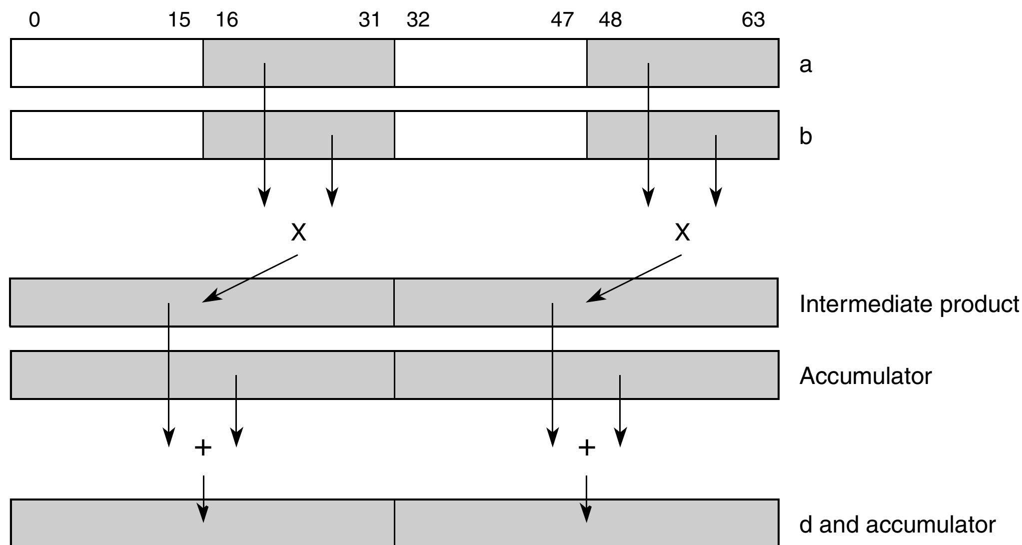


Figure 3-421. Odd Form of Vector Half-Word Multiply (__ev_mhosmiaaw)

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

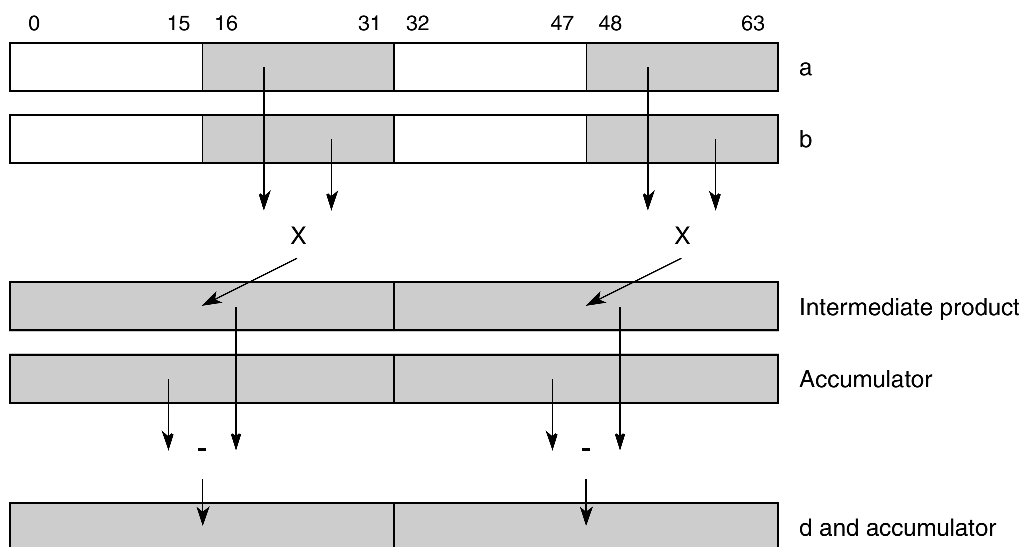


Figure 3-422. Odd Form of Vector Half-Word Multiply (__ev_mhosmianw)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evmhosmianw d,a,b**

__ev_mhossf[a] __ev_mhossf[a]

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)

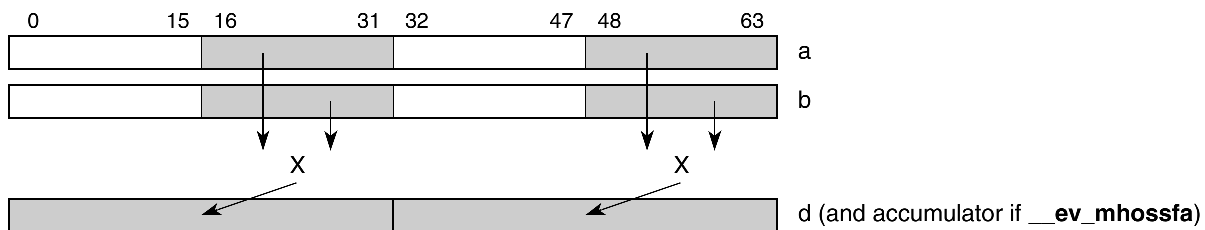


Figure 3-423. Vector Multiply Half Words, Odd, Signed, Saturate, Fractional (to Accumulator) (__ev_mhossf[a])

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 ←ovh | movh ; SPEFSCROV ←ovl | 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

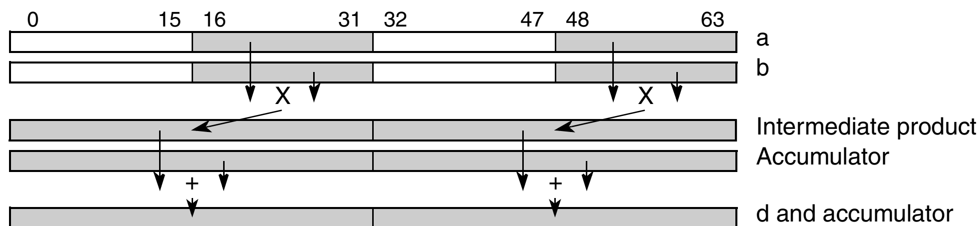


Figure 3-424. Odd Form of Vector Half-Word Multiply (__ev_mhossfaaw)

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 ←ovh | movh ; SPEFSCROV ←ovl | 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

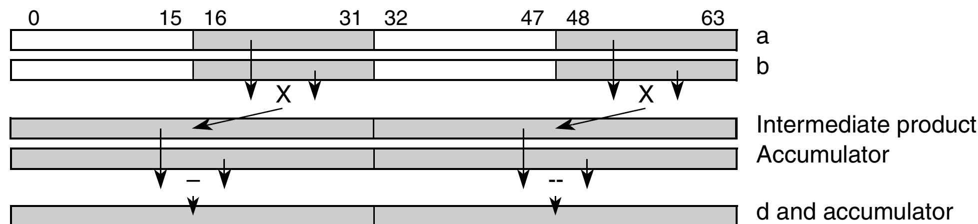


Figure 3-425. Odd Form of Vector Half-Word Multiply (__ev_mhossfanw)

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 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

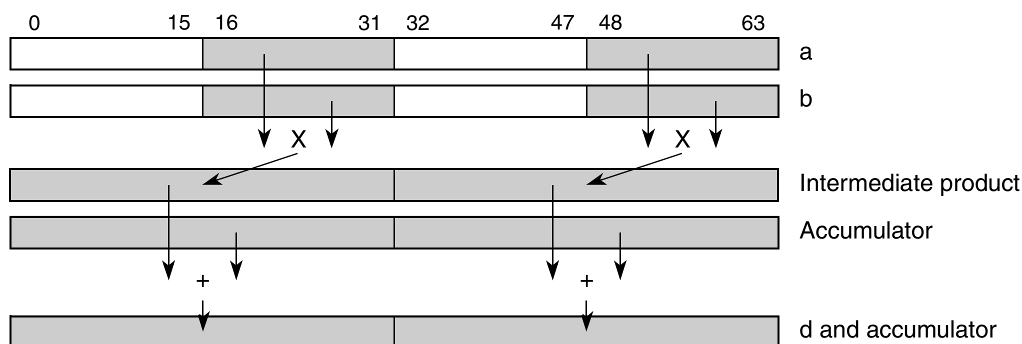


Figure 3-426. Odd Form of Vector Half-Word Multiply (__ev_mhossiaaw)

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 or underflow 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

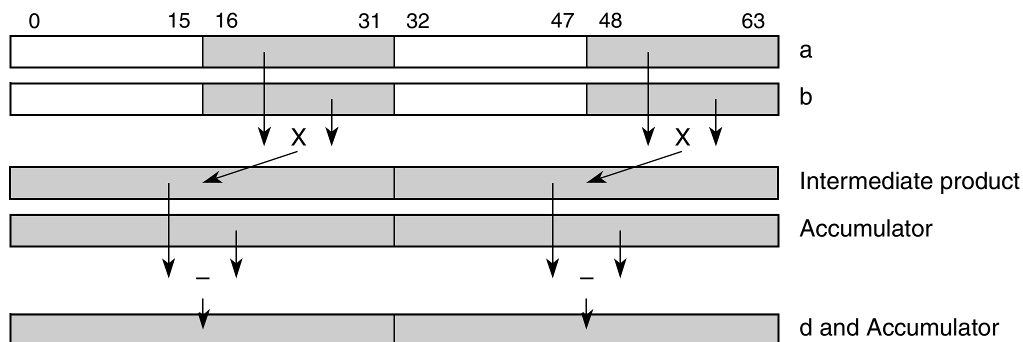


Figure 3-427. Odd Form of Vector Half-Word Multiply (__ev_mhossianw)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evmhossianw d,a,b**

__ev_mhosumi[a] __ev_mhosumi[a]

Vector Multiply Half Words, Odd, Signed by Unsigned, Modulo, Integer (to Accumulator)

d = __ev_mhosumi (a,b) (A = 0)

d = __ev_mhosumia (a,b) (A = 1)

```
// high
d0:31 ←(a16:31 ×su b16:31)

// low
d32:63 ←(a48:63 ×su b48:63)

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

The corresponding odd-numbered, half word signed integer element in parameter **a** and unsigned integer element in parameter **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)

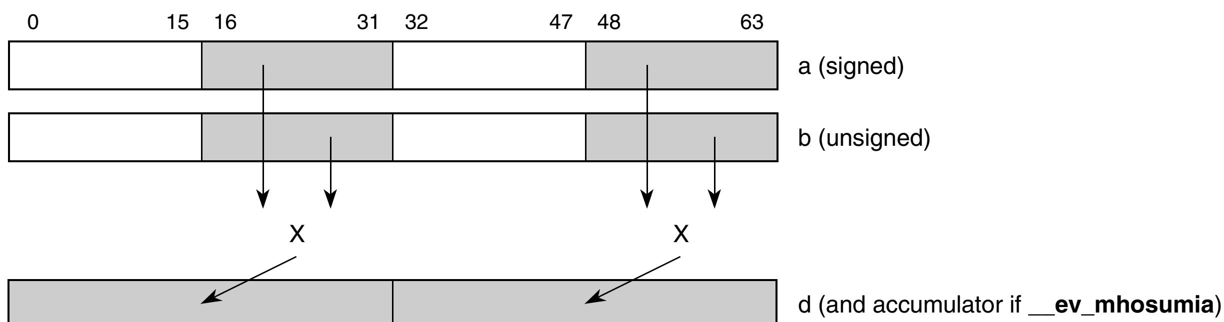


Figure 3-428. Vector Multiply Half Words, Odd, Signed by Unsigned, Modulo, Integer (to Accumulator) (__ev_mhosumi[a])

A	d	a	b	Maps to
A = 0	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evmhosumi d,a,b**
A = 1	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evmhosumia d,a,b**

__ev_mhosumiaaw __ev_mhosumiaaw

Vector Multiply Half Words, Odd, Signed by Unsigned, Modulo, Integer and Accumulate into Words

d = __ev_mhosumiaaw (a,b)

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

// low
temp0:31 ←a48:63 ×su 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 element in parameter **a** and unsigned integer element in parameter **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

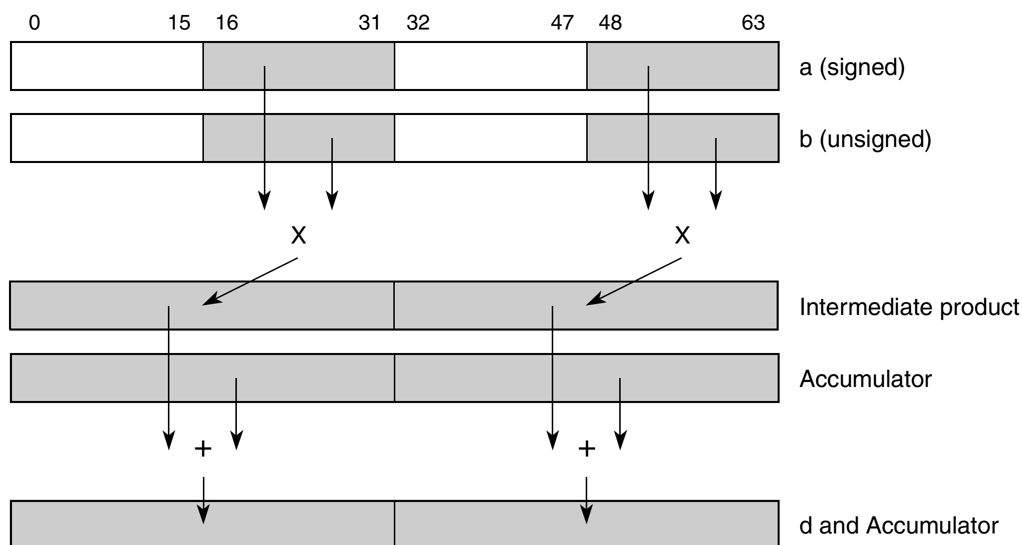


Figure 3-429. Odd Form of Vector Half Word Multiply (__ev_mhosumiaaw)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evmhosumiaaw d,a,b**

__ev_mhosumianw __ev_mhosumianw

Vector Multiply Half Words, Odd, Signed by Unsigned, Modulo, Integer and Accumulate Negative into Words

d = __ev_mhosumianw (a,b)

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

// low
temp0:31 ← a48:63 ×su 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 element in parameter **a** and unsigned integer element in parameter **b** are multiplied. The 32 bits of each intermediate 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

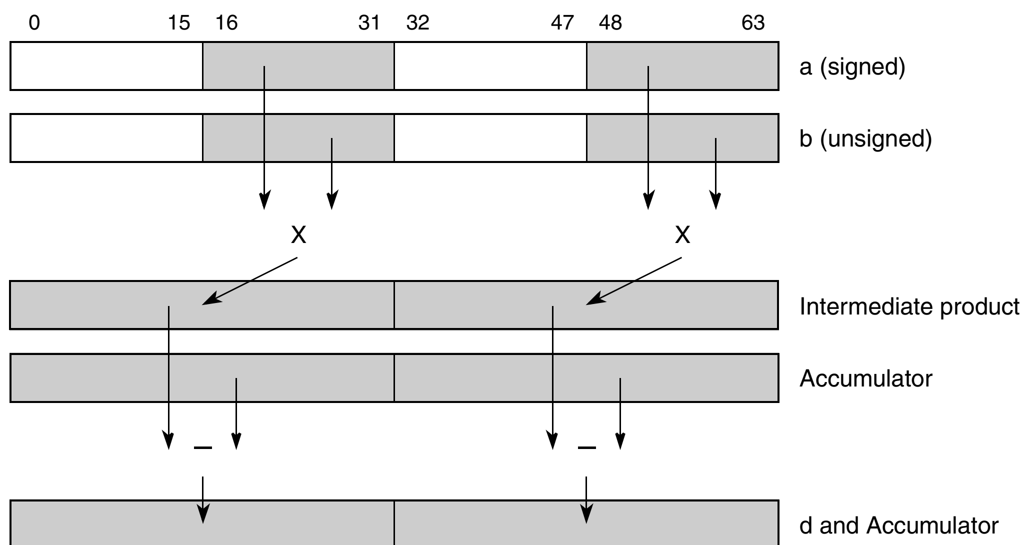


Figure 3-430. Odd Form of Vector half word Multiply (__ev_mhosumianw)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evmhosumianw d,a,b**

__ev_mhosusiaaw __ev_mhosusiaaw

Vector Multiply Half Words, Odd, Signed by Unsigned, Saturate, Integer and Accumulate into Words

d = __ev_mhosusiaaw (a,b)

```
// high
temph0:31 ←a16:31 ×su b16:31
temp0:63 ←EXTS(ACC0:31) + EXTS(temph0:31)
ovh ←(temp31 ⊕ temp32)
d0:31 ←SATURATE(ovh, temp31, 0x8000_0000, 0x7FFF_FFFF, temp32:63)

// low
templ0:31 ←a48:63 ×su b48:63
temp0:63 ←EXTS(ACC32:63) + EXTS(templ0:31)
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 element in parameter **a** and unsigned integer element in parameter **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 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

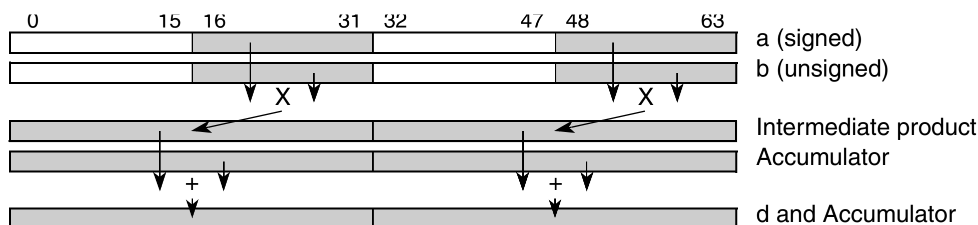


Figure 3-431. Odd Form of Vector half word Multiply (__ev_mhosusiaaw)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evmhosusiaaw d,a,b**

__ev_mhosusianw __ev_mhosusianw

Vector Multiply Half Words, Odd, Signed by Unsigned, Saturate, Integer and Accumulate Negative into Words

d = __ev_mhosusianw (a,b)

```
// high
temph0:31 ←a16:31 ×su b16:31
temp0:63 ←EXTS(ACC0:31) - EXTS(temph0:31)
ovh ←(temp31 ⊕ temp32)
d0:31 ←SATURATE(ovh, temp31, 0x8000_0000, 0x7FFF_FFFF, temp32:63)

// low
templ0:31 ←a48:63 ×su b48:63
temp0:63 ←EXTS(ACC32:63) - EXTS(templ0:31)
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 element in parameter **a** and unsigned integer element in parameter **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 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

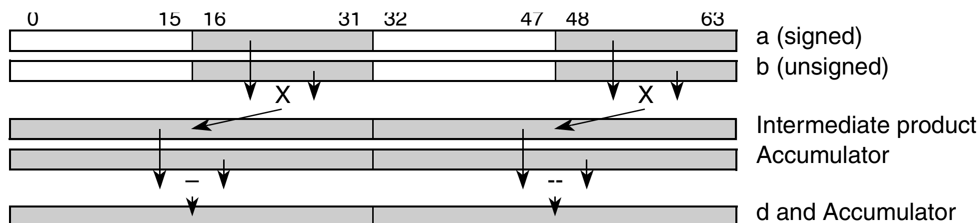


Figure 3-432. Odd Form of Vector half word Multiply (__ev_mhosusianw)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evmhosusianw d,a,b**

__ev_mhoumi[a] __ev_mhoumi[a]

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 3-433. Vector Multiply Half Words, Odd, Unsigned, Modulo, Integer (to Accumulator) (__ev_mhoumi[a])

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_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

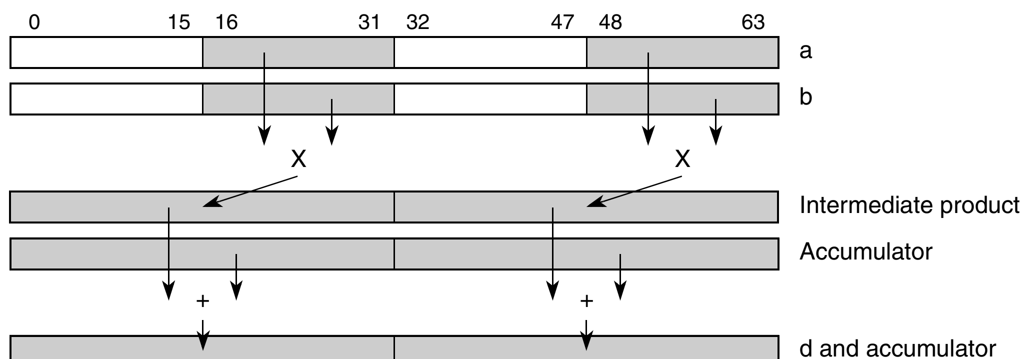


Figure 3-434. Odd Form of Vector Half-Word Multiply (__ev_mhoumiaaw)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evmhoumiaaw d,a,b**

__ev_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

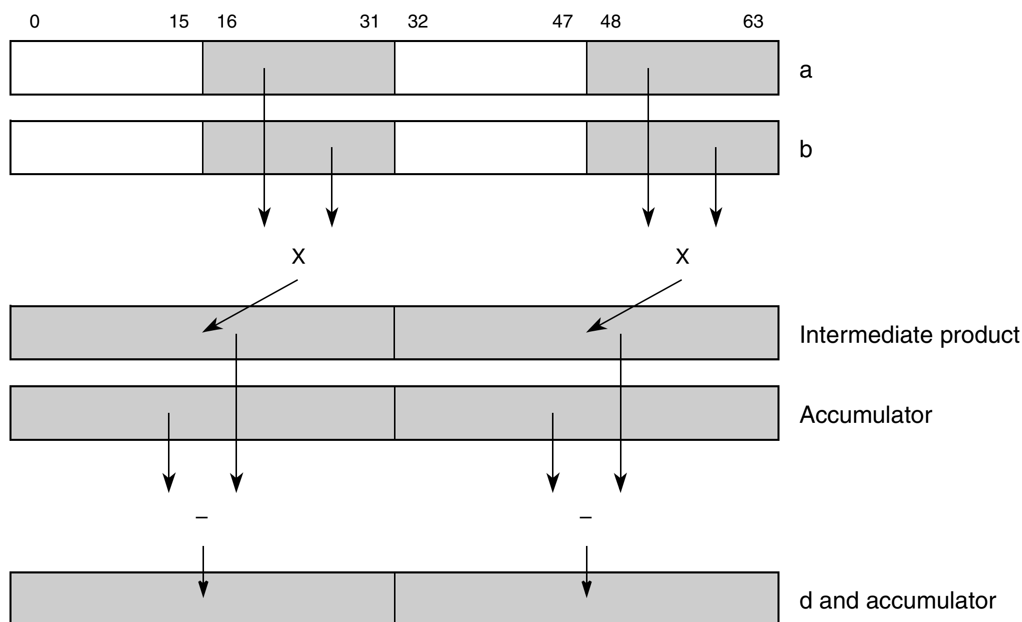


Figure 3-435. Odd Form of Vector Half-Word Multiply (__ev_mhoumianw)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evmhoumianw d,a,b**

__ev_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

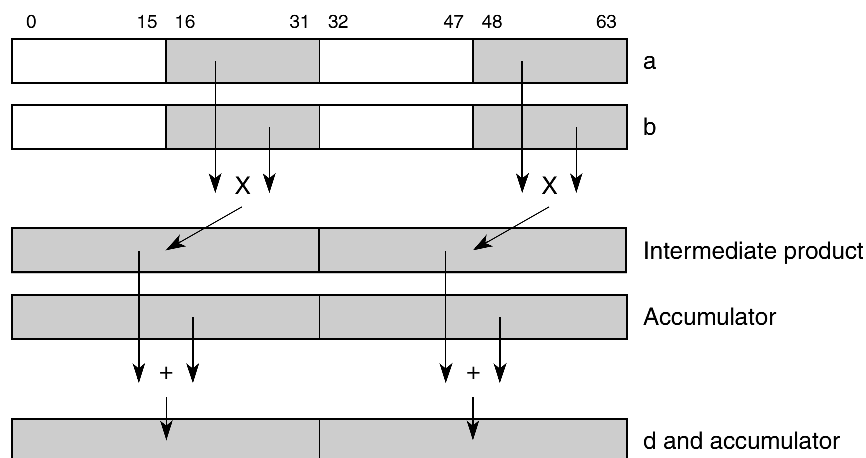


Figure 3-436. Odd Form of Vector Half Word Multiply (__ev_mhousiaaw)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evmhousiaaw d,a,b**

__ev_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

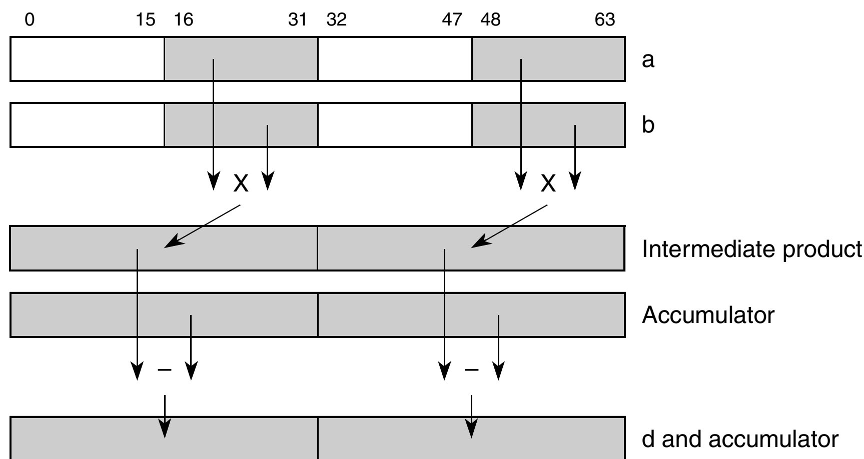


Figure 3-437. Odd Form of Vector Half Word Multiply (__ev_mhousianw)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evmhousianw d,a,b**

__ev_mhssf __ev_mhssf

Vector Multiply Half Word Signed, Saturate, Fractional

d = __ev_mhssf (a,b)

```
temp0_{0:31} ← a_{0:15} ×_{sf} b_{0:15}
if (a_{0:15} = 0x8000) & (b_{0:15} = 0x8000) then
    temp0_{0:31} ←0x7FFF_FFFF //saturate
    movh ←1
else
    movh ←0
temp1_{0:31} ← a_{16:31} ×_{sf} b_{16:31}
if (a_{16:31} = 0x8000) & (b_{16:31} = 0x8000) then
    temp1_{0:31} ←0x7FFF_FFFF //saturate
    movh ←1
temp2_{0:31} ← a_{32:47} ×_{sf} b_{32:47}
if (a_{32:47} = 0x8000) & (b_{32:47} = 0x8000) then
    temp2_{0:31} ←0x7FFF_FFFF //saturate
    movl ←1
else
    movl ←0
temp3_{0:31} ← a_{48:63} ×_{sf} b_{48:63}
if (a_{48:63} = 0x8000) & (b_{48:63} = 0x8000) then
    temp3_{0:31} ←0x7FFF_FFFF //saturate
    movl ←1
d_{0:15} ←temp0_{0:15} ; d_{16:31} ←temp1_{0:15}; d_{32:47} ←temp2_{0:15}; d_{48:63} ←temp3_{0:15}
// update SPEFSCR
SPEFSCR_{OVH} ←movh; SPEFSCR_{OV} ←movl
SPEFSCR_{SOVH} ←SPEFSCR_{SOVH} | movh; SPEFSCR_{SOV} ←SPEFSCR_{SOV} | movl
```

For each half word element in the destination, corresponding half word pairs of signed fractional elements in parameters **a** and **b** are multiplied, and the result is placed into the corresponding parameter **d** half word. If both inputs of a multiply are -1.0, the result saturates to 0x7FFF. The overflow and summary overflow bits are recorded in the SPEFSCR based on an underflow from the multiply.

Other registers altered: SPEFSCR

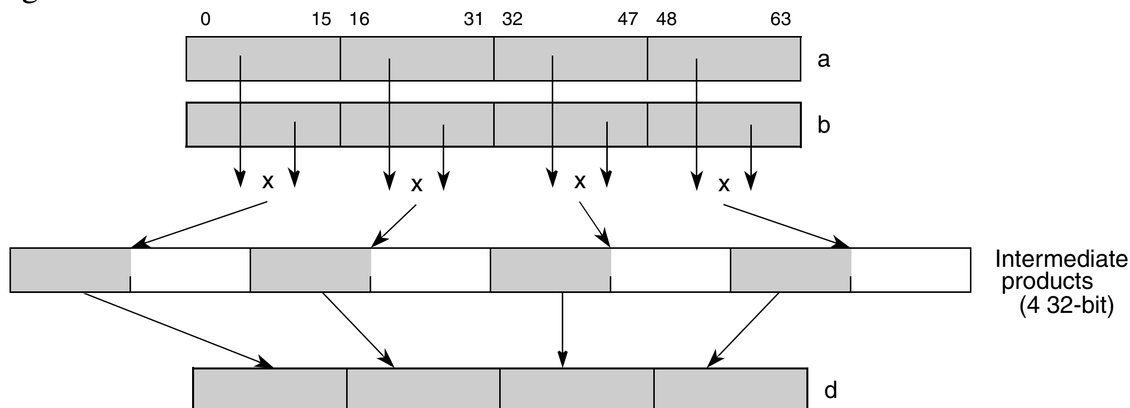


Figure 3-438. Vector Multiply Half Word, Signed, Saturate, Fractional (__ev_mhssf)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evmhssf d,a,b**

__ev_mhssfr __ev_mhssfr

Vector Multiply Half Word Signed, Saturate, Fractional and Round

d = __ev_mhssfr (a,b)

```
temp0[0:31] ← (a[0:15] ×sf b[0:15]) + 0x0000_8000
if (a[0:15] = 0x8000) & (b[0:15] = 0x8000) then
    temp0[0:31] ←0x7FFF_FFFF //saturate
    movh ←1
else
    movh ←0
temp1[0:31] ← (a[16:31] ×sf b[16:31]) + 0x0000_8000
if (a[16:31] = 0x8000) & (b[16:31] = 0x8000) then
    temp1[0:31] ←0x7FFF_FFFF //saturate
    movh ←1
temp2[0:31] ← (a[32:47] ×sf b[32:47]) + 0x0000_8000
if (a[32:47] = 0x8000) & (b[32:47] = 0x8000) then
    temp2[0:31] ←0x7FFF_FFFF //saturate
    movl ←1
else
    movl ←0
temp3[0:31] ← (a[48:63] ×sf b[48:63]) + 0x0000_8000
if (a[48:63] = 0x8000) & (b[48:63] = 0x8000) then
    temp3[0:31] ←0x7FFF_FFFF //saturate
    movl ←1

d[0:15] ←temp0[0:15]
d[16:31] ←temp1[0:15]
d[32:47] ←temp2[0:15]
d[48:63] ←temp3[0:15]

// update SPEFSCR
SPEFSCR[OVH] ←movh
SPEFSCR[OV] ←movl
SPEFSCR[SOVH] ←SPEFSCR[SOVH] | movh
SPEFSCR[SOV] ←SPEFSCR[SOV] | movl
```

For each half word element in the destination, corresponding half word pairs of signed fractional elements in parameters **a** and **b** are multiplied, then rounded by adding 1/2 lsb, and the result is placed into the corresponding parameter **d** half word. If both inputs of a multiply are -1.0, the result saturates to 0x7FFF. The overflow and summary overflow bits are recorded in the SPEFSCR based on an underflow from the multiply.

Other registers altered: SPEFSCR

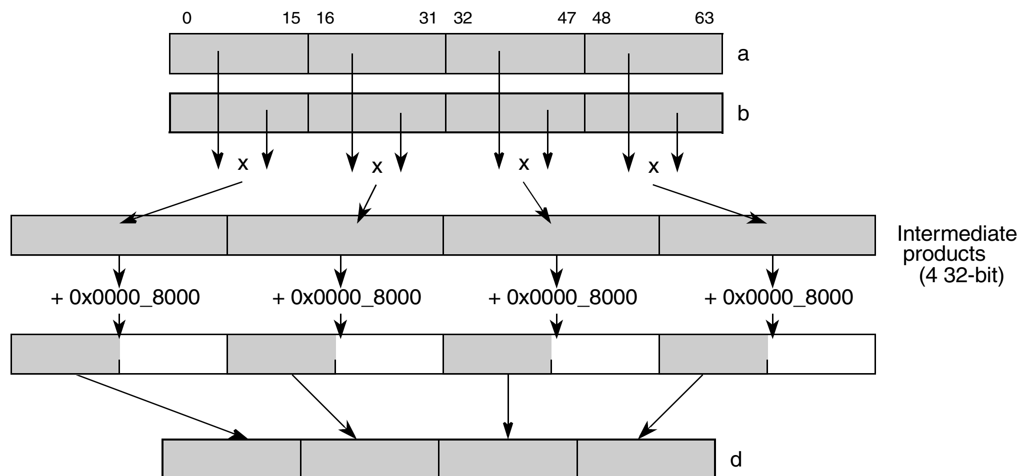


Figure 3-439. Vector Multiply Half Word, Signed, Saturate, Fractional and Round (__ev_mhssfr)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evmhssfr d,a,b**

__ev_mhssi __ev_mhssi

Vector Multiply Half Word Signed, Saturate, Integer

d = __ev_mhssi (a,b)

```
temp0_0:31 ← a_0:15 ×si b_0:15
if (temp0_0:31 > 0x0000_7FFF) | (temp0_0:31 < 0xFFFF_8000) then
    movh ←1
    temp0_16:31 ←SATURATE(movh, temp0_0, 0x8000, 0x7FFF, temp0_16:31)
else
    movh ←0
temp1_0:31 ← a_16:31 ×si b_16:31
if (temp1_0:31 > 0x0000_7FFF) | (temp1_0:31 < 0xFFFF_8000) then
    movh ←1
    temp1_16:31 ←SATURATE(movh, temp1_0, 0x8000, 0x7FFF, temp1_16:31)
temp2_0:31 ← a_32:47 ×si b_32:47
if (temp2_0:31 > 0x0000_7FFF) | (temp2_0:31 < 0xFFFF_8000) then
    movl ←1
    temp2_16:31 ←SATURATE(movl, temp2_0, 0x8000, 0x7FFF, temp2_16:31)
else
    movl ←0
temp3_0:31 ← a_48:63 ×si b_48:63
if (temp3_0:31 > 0x0000_7FFF) | (temp3_0:31 < 0xFFFF_8000) then
    movl ←1
    temp3_16:31 ←SATURATE(movl, temp3_0, 0x8000, 0x7FFF, temp3_16:31)

d_0:15 ←temp0_16:31 ; d_16:31 ←temp1_16:31; d_32:47 ←temp2_16:31; d_48:63 ←temp3_16:31
// update SPEFSCR
SPEFSCR_OVH ←movh; SPEFSCR_OV ←movl
SPEFSCR_SOVH ← SPEFSCR_SOVH | movh; SPEFSCR_SOV ← SPEFSCR_SOV | movl
```

For each half word element in the destination, corresponding half word pairs of signed integer elements in parameters **a** and **b** are multiplied, and the result is placed into the corresponding parameter **d** half word. If the result exceeds the magnitude of a half word, the result saturates. The overflow and summary overflow bits are recorded in the SPEFSCR based on an underflow or overflow from the multiply.

Other registers altered: SPEFSCR

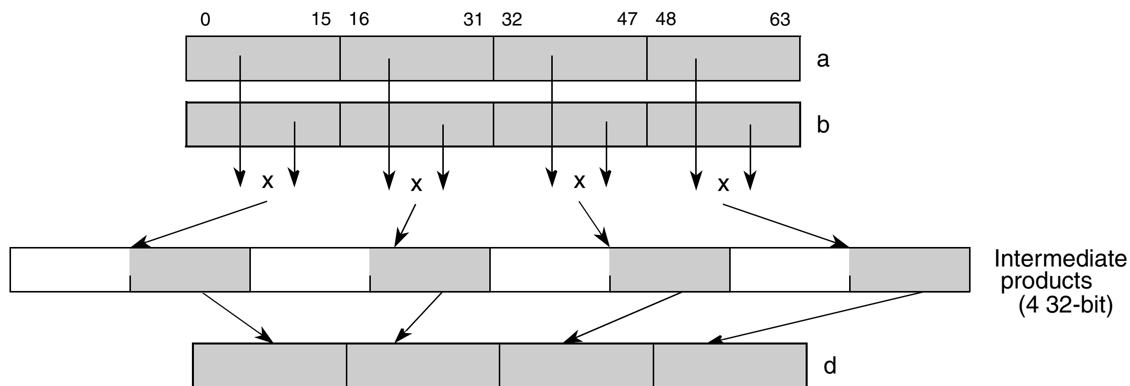


Figure 3-440. Vector Multiply Half Word, Signed, Saturate, Integer (__ev_mhssi)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evmhssi d,a,b**

__ev_mhsusi __ev_mhsusi

Vector Multiply Half Word Signed by Unsigned, Saturate, Integer

d = __ev_mhsusi (a,b)

```
temp0[0:31] ← a[0:15] ×sui b[0:15]
if (temp0[0:31] > 0x0000_7FFF) | (temp0[0:31] < 0xFFFF_8000) then
    movh ←1
    temp0[16:31] ←SATURATE(movh, temp0[0], 0x8000, 0x7FFF, temp0[16:31])
else
    movh ←0
temp1[0:31] ← a[16:31] ×sui b[16:31]
if (temp1[0:31] > 0x0000_7FFF) | (temp1[0:31] < 0xFFFF_8000) then
    movh ←1
    temp1[16:31] ←SATURATE(movh, temp1[0], 0x8000, 0x7FFF, temp1[16:31])
temp2[0:31] ← a[32:47] ×sui b[32:47]
if (temp2[0:31] > 0x0000_7FFF) | (temp2[0:31] < 0xFFFF_8000) then
    movl ←1
    temp2[16:31] ←SATURATE(movl, temp2[0], 0x8000, 0x7FFF, temp2[16:31])
else
    movl ←0
temp3[0:31] ← a[48:63] ×sui b[48:63]
if (temp3[0:31] > 0x0000_7FFF) | (temp3[0:31] < 0xFFFF_8000) then
    movl ←1
    temp3[16:31] ←SATURATE(movl, temp3[0], 0x8000, 0x7FFF, temp3[16:31])

d[0:15] ←temp0[16:31] ; d[16:31] ←temp1[16:31]; d[32:47] ←temp2[16:31]; d[48:63] ←temp3[16:31]
// update SPEFSCR
SPEFSCR[OVH] ←movh; SPEFSCR[OV] ←movl
SPEFSCR[SOVH] ←SPEFSCR[SOVH] | movh; SPEFSCR[SOV] ←SPEFSCR[SOV] | movl
```

For each half word element in the destination, corresponding half word pairs of signed integer elements in parameter **a** and unsigned integer elements in parameter **b** are multiplied, and the result is placed into the corresponding parameter **d** half word. If the result exceeds the magnitude of a signed half word, the result saturates. The overflow and summary overflow bits are recorded in the SPEFSCR based on an underflow or overflow from the multiply.

Other registers altered: SPEFSCR

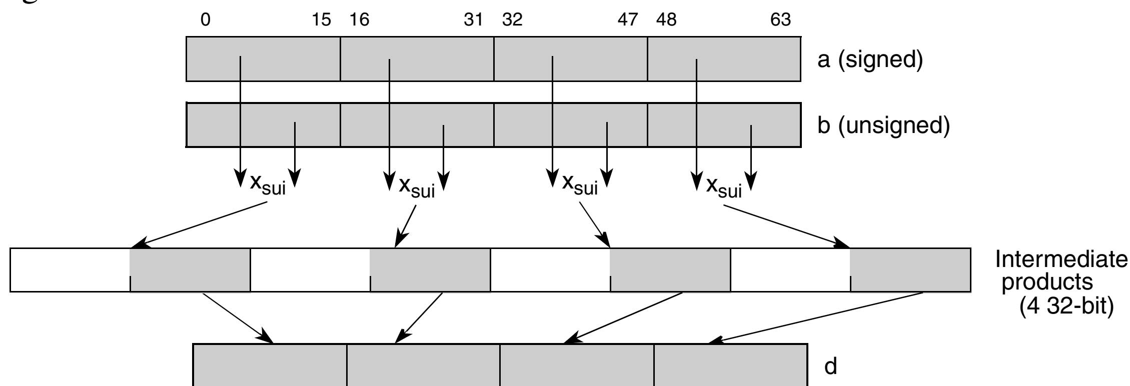


Figure 3-441. Vector Multiply Half Word, Signed by Unsigned, Saturate, Integer (__ev_mhsusi)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evmhsusi d,a,b**

__ev_mhumi

__ev_mhumi

Vector Multiply Half Word Unsigned, Modulo, Integer

d = __ev_mhumi (a,b)

$temph0_{0:31} \leftarrow (a_{0:15} \times_{ui} b_{0:15})$
$temph1_{0:31} \leftarrow (a_{16:31} \times_{ui} b_{16:31})$
$temph2_{0:31} \leftarrow (a_{32:47} \times_{ui} b_{32:47})$
$temph3_{0:31} \leftarrow (a_{48:63} \times_{ui} b_{48:63})$

$d_{0:15} \leftarrow temph0_{16:31}$
$d_{16:31} \leftarrow temph1_{16:31}$
$d_{32:47} \leftarrow temph2_{16:31}$
$d_{48:63} \leftarrow temph3_{16:31}$

For each half word element in the destination, corresponding half word pairs of unsigned integer elements in parameters **a** and **b** are multiplied producing a 32-bit product. The low order 16 bits of the product is placed into the corresponding parameter **d** half word.

Note: The least significant 16 bits of the products are independent of whether the half word elements in parameters **a** and **b** are treated as signed or unsigned 16-bit integers.

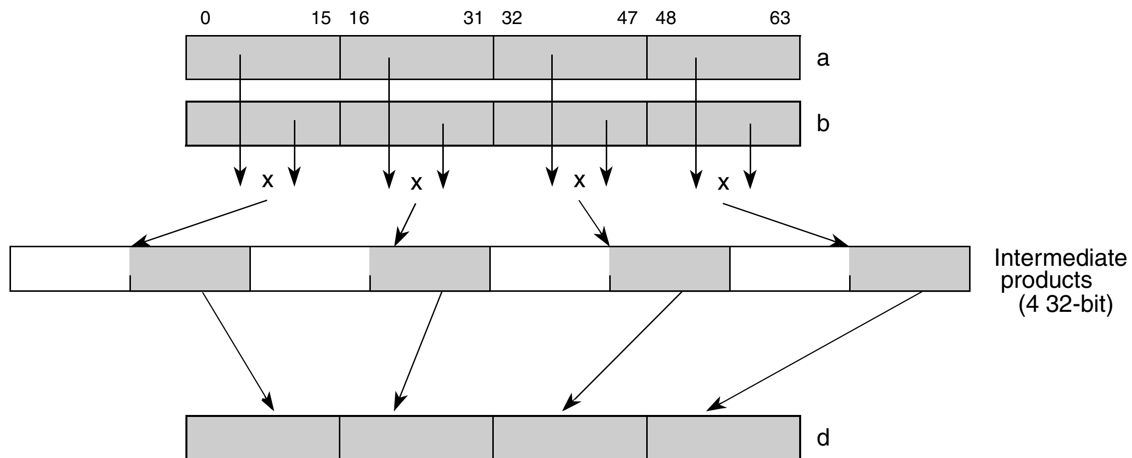


Figure 3-442. Vector Multiply Half Word, Unsigned, Modulo, Integer (__ev_mhumi)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evmhumi d,a,b**

__ev_mhusi __ev_mhusi

Vector Multiply Half Word Unsigned, Saturate, Integer

d = __ev_mhusi (a,b)

```
temp0[0:31] ← a[0:15] ×ui b[0:15]
if (temp0[0:31] > 0x0000_FFFF) then
    movh ←1
    temp0[16:31] ←SATURATE(movh, 0, 0xFFFF, 0xFFFF, temp0[16:31])
else
    movh ←0
temp1[0:31] ← a[16:31] ×ui b[16:31]
if (temp1[0:31] > 0x0000_FFFF) then
    movh ←1
    temp1[16:31] ←SATURATE(movh, 0, 0xFFFF, 0xFFFF, temp1[16:31])
temp2[0:31] ← a[32:47] ×ui b[32:47]
if (temp2[0:31] > 0x0000_FFFF) then
    movl ←1
    temp2[16:31] ←SATURATE(movl, 0, 0xFFFF, 0xFFFF, temp2[16:31])
else
    movl ←0
temp3[0:31] ← a[48:63] ×ui b[48:63]
if (temp3[0:31] > 0x0000_FFFF) then
    movl ←1
    temp3[16:31] ←SATURATE(movl, 0, 0xFFFF, 0xFFFF, temp3[16:31])

d[0:15] ←temp0[16:31] ; d[16:31] ←temp1[16:31]; d[32:47] ←temp2[16:31]; d[48:63] ←temp3[16:31]
// update SPEFSCR
SPEFSCR[OVH] ← movh; SPEFSCR[OV] ← movl
SPEFSCR[SOVH] ← SPEFSCR[SOVH] | movh; SPEFSCR[SOV] ← SPEFSCR[SOV] | movl
```

For each half word element in the destination, corresponding half word pairs of unsigned integer elements in parameters **a** and **b** are multiplied, and the result is placed into the corresponding parameter **d** half word. If the result exceeds the magnitude of a half word, the result saturates. The overflow and summary overflow bits are recorded in the SPEFSCR based on an overflow from the multiply.

Other registers altered: SPEFSCR

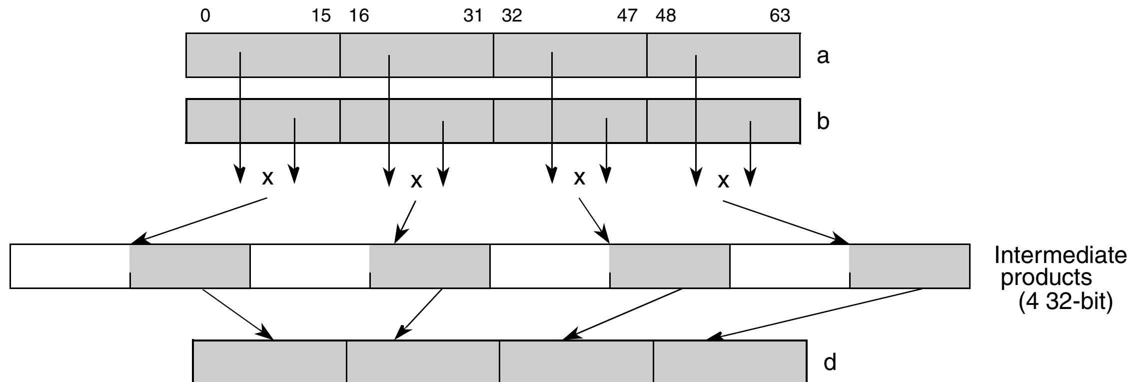


Figure 3-443. Vector Multiply Half Word, Unsigned, Saturate, Integer (__ev_mhusi)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evmhusi d,a,b**

__ev_minbpsh

__ev_minbpsh

Vector Minimum of Byte Pairs Signed to Half Word

d = __ev_minbpsh (a)

$d_{0:15} \leftarrow EXTS(MIN_{si}(a_{0:7}, a_{8:15}))$
$d_{16:31} \leftarrow EXTS(MIN_{si}(a_{16:23}, a_{24:31}))$
$d_{32:47} \leftarrow EXTS(MIN_{si}(a_{32:39}, a_{40:47}))$
$d_{48:63} \leftarrow EXTS(MIN_{si}(a_{48:55}, a_{56:63}))$

Even/odd pairs of signed-integer byte elements of parameter **a** are compared and the smaller of the two signed-integer byte elements is sign-extended and placed into the corresponding half word element of parameter **d**.

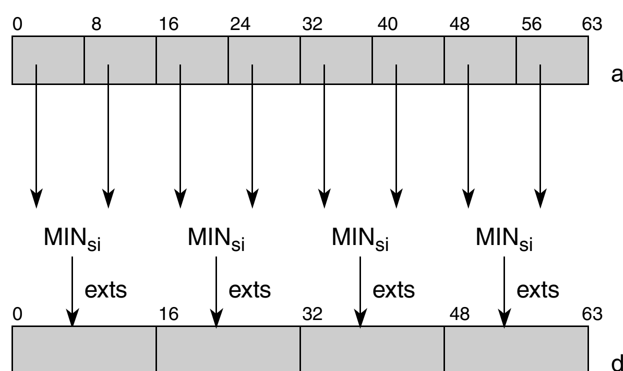


Figure 3-444. Vector Minimum of Byte Pairs Signed to Half Word (__ev_minbpsh)

d	a	Maps to
__ev64_opaque__	__ev64_opaque__	**evminbpsh d,a**

__ev_minbpuh   __ev_minbpuh

Vector Minimum of Byte Pairs Unsigned to Half Word

d = __ev_minbpuh (a)

$d_{0:15} \leftarrow \text{EXTZ}(\text{MIN}_{ui}(a_{0:7}, a_{8:15}))$
$d_{16:31} \leftarrow \text{EXTZ}(\text{MIN}_{ui}(a_{16:23}, a_{24:31}))$
$d_{32:47} \leftarrow \text{EXTZ}(\text{MIN}_{ui}(a_{32:39}, a_{40:47}))$
$d_{48:63} \leftarrow \text{EXTZ}(\text{MIN}_{ui}(a_{48:55}, a_{56:63}))$

Even/odd pairs of unsigned-integer byte elements of parameter **a** are compared and the smaller of the two unsigned-integer byte elements is zero-extended and placed into the corresponding half word element of parameter **d**.

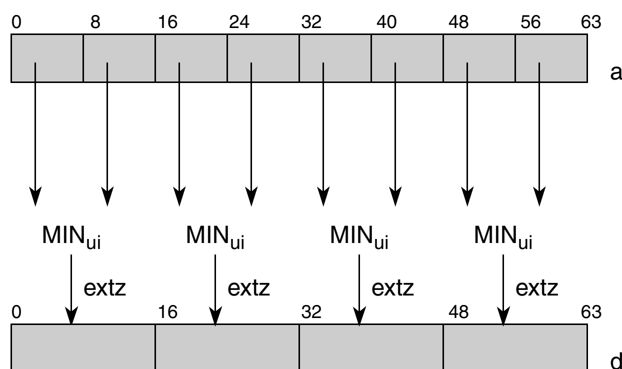


Figure 3-445. Vector Minimum of Byte Pairs Unsigned to Half Word (__ev_minbpuh)

d	a	Maps to
__ev64_opaque__	__ev64_opaque__	**evminbpuh d,a**

__ev_minbs __ev_minbs

Vector Minimum Byte Signed

d = __ev_minbs (a,b)

$d_{0:7} \leftarrow MIN_{si}(a_{0:7}, b_{0:7})$
$d_{8:15} \leftarrow MIN_{si}(a_{8:15}, b_{8:15})$
$d_{16:23} \leftarrow MIN_{si}(a_{16:23}, b_{16:23})$
$d_{24:31} \leftarrow MIN_{si}(a_{24:31}, b_{24:31})$
$d_{32:39} \leftarrow MIN_{si}(a_{32:39}, b_{32:39})$
$d_{40:47} \leftarrow MIN_{si}(a_{40:47}, b_{40:47})$
$d_{48:55} \leftarrow MIN_{si}(a_{48:55}, b_{48:55})$
$d_{56:63} \leftarrow MIN_{si}(a_{56:63}, b_{56:63})$

Each signed-integer byte element of parameter **a** is compared to the corresponding signed-integer byte element of parameter **b** and the smaller of the two signed-integer elements is placed into the corresponding byte element of parameter **d**.

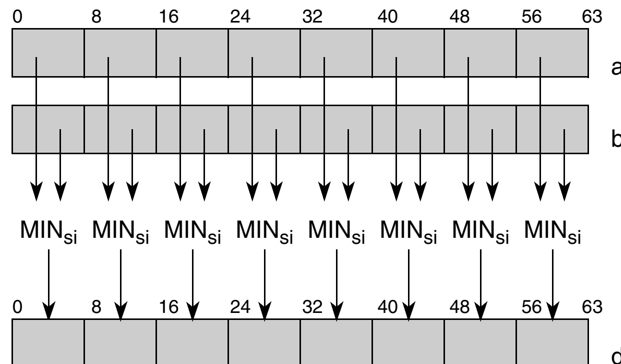


Figure 3-446. Vector Minimum Byte Signed (__ev_minbs)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evminbs d,a,b**

__ev_minbu __ev_minbu

Vector Minimum Byte Unsigned

d = __ev_minbu (a,b)

$d_{0:7} \leftarrow MIN_{ui}(a_{0:7}, b_{0:7})$
$d_{8:15} \leftarrow MIN_{ui}(a_{8:15}, b_{8:15})$
$d_{16:23} \leftarrow MIN_{ui}(a_{16:23}, b_{16:23})$
$d_{24:31} \leftarrow MIN_{ui}(a_{24:31}, b_{24:31})$
$d_{32:39} \leftarrow MIN_{ui}(a_{32:39}, b_{32:39})$
$d_{40:47} \leftarrow MIN_{ui}(a_{40:47}, b_{40:47})$
$d_{48:55} \leftarrow MIN_{ui}(a_{48:55}, b_{48:55})$
$d_{56:63} \leftarrow MIN_{ui}(a_{56:63}, b_{56:63})$

Each signed-integer byte element of parameter **a** is compared to the corresponding signed-integer byte element of parameter **b** and the smaller of the two signed-integer elements is placed into the corresponding byte element of parameter **d**.

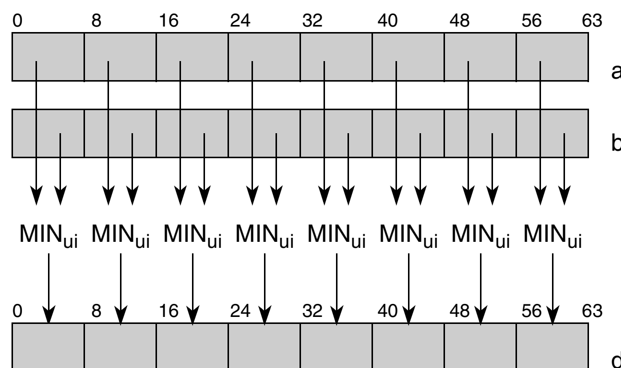


Figure 3-447. Vector Minimum Byte Unsigned (__ev_minbu)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evminbu d,a,b**

__ev_minds __ev_minds

Vector Minimum Download Signed

d = __ev_minds (a,b)

$d_{0:63} \leftarrow MIN_{si}(a_{0:63},\ b_{0:63})$

The signed-integer doubleword in parameter **a** is compared to the signed-integer doubleword in parameter **b** and the smaller of the two doublewords is placed into parameter **d**.

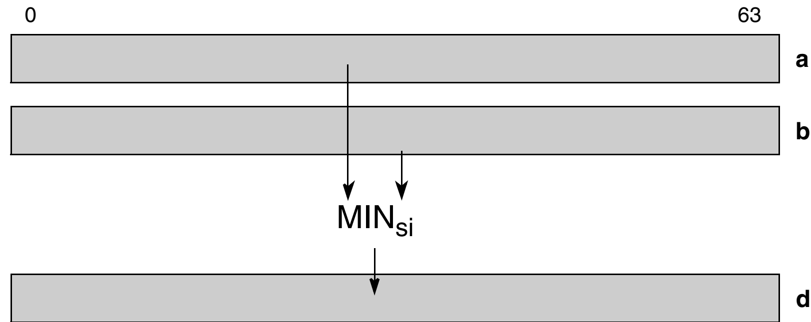


Figure 3-448. Vector Minimum Doubleword Signed (__ev_minds)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evminds d,a,b**

__ev_mindu __ev_mindu

Vector Minimum Download Unsigned

d = __ev_mindu (a,b)

$d_{0:63} \leftarrow MIN_{ui}(a_{0:63}, b_{0:63})$

The unsigned-integer doubleword in parameter **a** is compared to the unsigned-integer doubleword in parameter **b** and the smaller of the two doublewords is placed into parameter **d**.

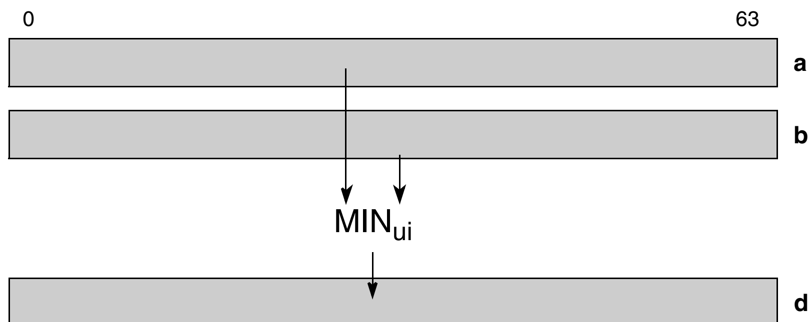


Figure 3-449. Vector Minimum Doubleword Unsigned (__ev_mindu)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evmindu d,a,b**

__ev_minhpsw
__ev_minhpsw

Vector Minimum of Half Word Pairs Signed to Word

d = __ev_minhpsw (a)

$d_{0:31} \leftarrow EXTS(MIN_{si}(a_{0:15}, a_{16:31}))$

$d_{32:63} \leftarrow EXTS(MIN_{si}(a_{32:47}, a_{48:63}))$

Even/odd pairs of signed-integer half word elements of parameter **a** are compared and the smaller of the two signed-integer half word elements is sign-extended and placed into the corresponding word element of parameter **d**.

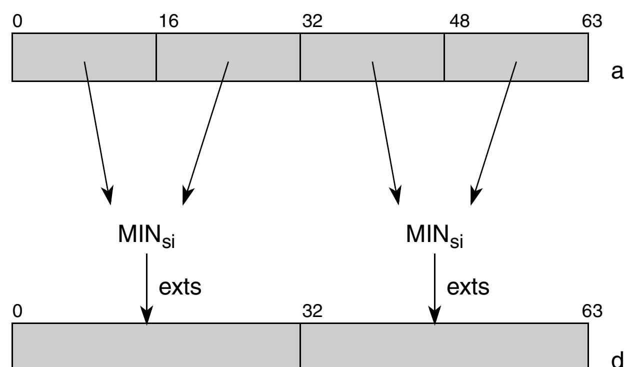


Figure 3-450. Vector Minimum of Half Word Pairs Signed to Word (evminhpsw)

d	a	Maps to
__ev64_opaque__	__ev64_opaque__	**evminhpsw d,a**

__ev_minhpuw __ev_minhpuw

Vector Minimum of Half Word Pairs Unsigned to Word

d = __ev_minhpuw (a)

$d_{0:31} \leftarrow EXTZ(MIN_{ui}(a_{0:15}, a_{16:31}))$

$d_{32:63} \leftarrow EXTZ(MIN_{ui}(a_{32:47}, a_{48:63}))$

Even/odd pairs of unsigned-integer half word elements of parameter **a** are compared and the smaller of the two unsigned-integer half word elements is zero-extended and placed into the corresponding word element of parameter **d**.

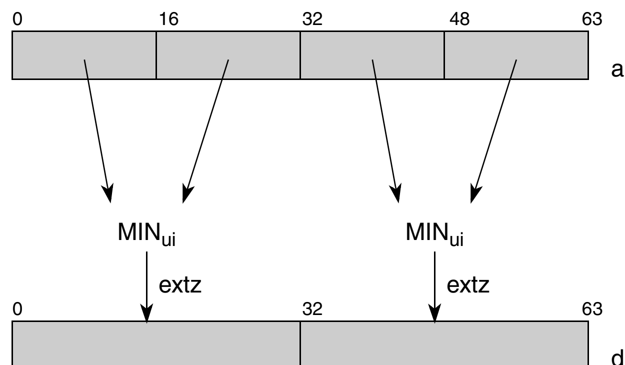


Figure 3-451. Vector Minimum of Half Word Pairs Unsigned to Word (__ev_minhpuw)

d	a	Maps to
__ev64_opaque__	__ev64_opaque__	**evminhpuw d,a**

__ev_minhs

__ev_minhs

Vector Minimum Half Word Signed

d = __ev_minhs (a,b)

$d_{0:15} \leftarrow MIN_{si}(a_{0:15}, b_{0:15})$
$d_{16:31} \leftarrow MIN_{si}(a_{16:31}, b_{16:31})$
$d_{32:47} \leftarrow MIN_{si}(a_{32:47}, b_{32:47})$
$d_{48:63} \leftarrow MIN_{si}(a_{48:63}, b_{48:63})$

Each signed-integer half word element of parameter **a** is compared to the corresponding signed-integer half word element of parameter **b** and the smaller of the two signed-integer elements is placed into the corresponding half word element of parameter **d**.

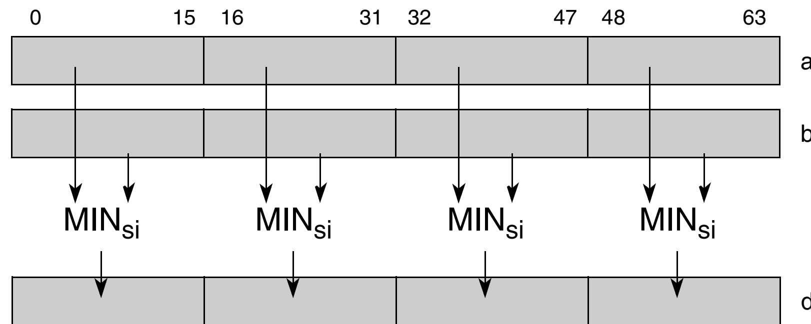


Figure 3-452. Vector Minimum Half Word Signed (__ev_minhs)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evminhs d,a,b**

__ev_minhu __ev_minhu

Vector Minimum Half Word Unsigned

d = __ev_minhu (a,b)

$d_{0:15} \leftarrow MIN_{ui}(a_{0:15}, b_{0:15})$
$d_{16:31} \leftarrow MIN_{ui}(a_{16:31}, b_{16:31})$
$d_{32:47} \leftarrow MIN_{ui}(a_{32:47}, b_{32:47})$
$d_{48:63} \leftarrow MIN_{ui}(a_{48:63}, b_{48:63})$

Each unsigned-integer half word element of parameter **a** is compared to the corresponding unsigned-integer half word element of parameter **b** and the smaller of the two unsigned-integer elements is placed into the corresponding half word element of parameter **d**.

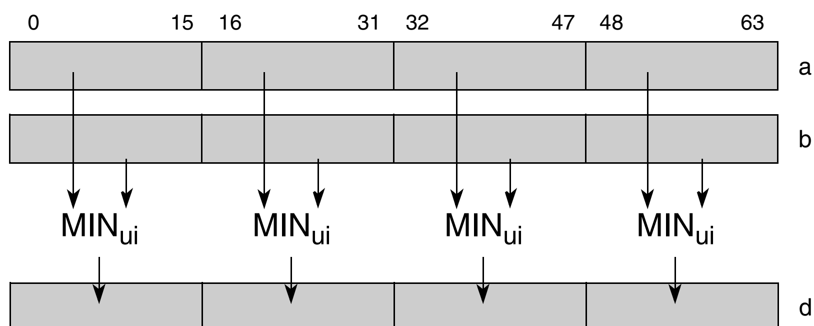


Figure 3-453. Vector Minimum Half Word Unsigned (__ev_minhu)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evminhu d,a,b**

__ev_minwpsd __ev_minwpsd

Vector Minimum of Word Pair Signed to Double Word

d = __ev_minwpsd (a)

$d_{0:63} \leftarrow EXTS(MIN_{si}(a_{0:31}, a_{32:63}))$

The signed-integer word elements of parameter **a** are compared and the smaller of the two signed-integer word elements is sign-extended and placed into parameter **d**.

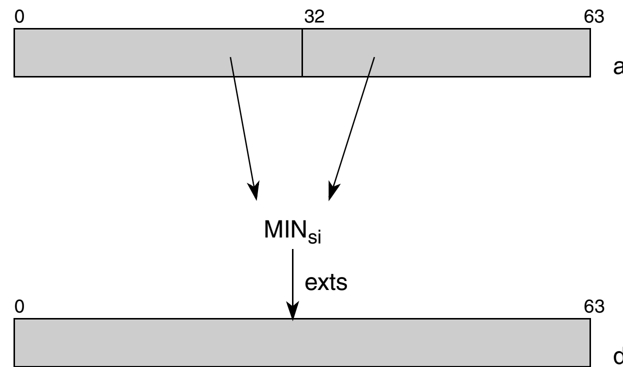


Figure 3-454. Vector Minimum of Word Pairs Signed to Double Word (__ev_minwpsd)

d	a	Maps to
__ev64_opaque__	__ev64_opaque__	**evminwpsd d,a**

__ev_minwpud __ev_minwpud

Vector Minimum of Word Pair Unsigned to Double Word

d = __ev_minwpud (a)

$d_{0:63} \leftarrow EXTZ(MIN_{ui}(a_{0:31}, a_{32:63}))$

The unsigned-integer word elements of parameter **a** are compared and the smaller of the two unsigned-integer word elements is zero-extended and placed into parameter **d**.

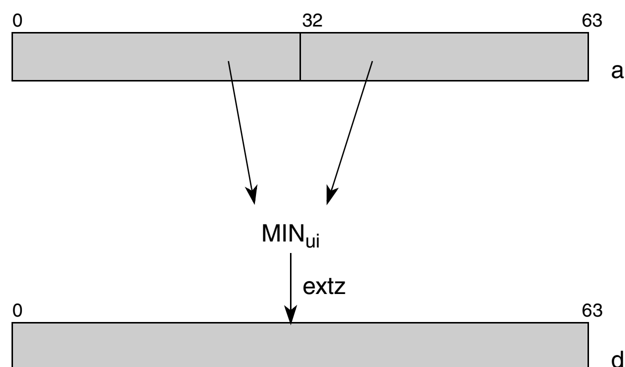


Figure 3-455. Vector Minimum of Word Pairs Unsigned to Double Word (__ev_minwpud)

d	a	Maps to
__ev64_opaque__	__ev64_opaque__	**evminwpud d,a**

__ev_minws __ev_minws

Vector Minimum Word Signed

d = __ev_minws (a,b)

$d_{0:31} \leftarrow MIN_{si}(a_{0:31}, b_{0:31})$

$d_{32:63} \leftarrow MIN_{si}(a_{32:63}, b_{32:63})$

Each signed-integer word element of parameter **a** is compared to the corresponding signed-integer word element of parameter **b** and the smaller of the two signed-integer elements is placed into the corresponding word element of parameter **d**.

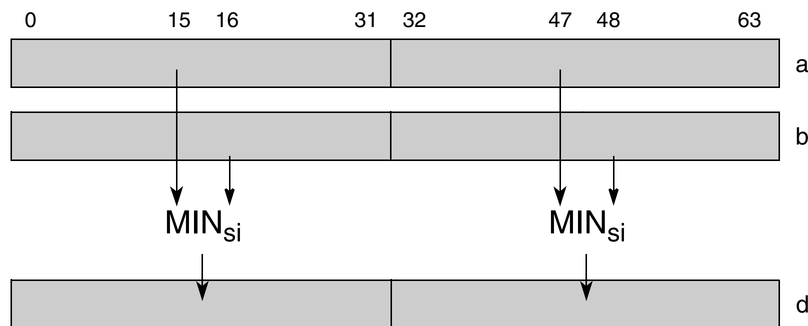


Figure 3-456. Vector Minimum Word Signed (__ev_minws)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evminws d,a,b**

__ev_minwu __ev_minwu

Vector Minimum Word Unsigned

d = __ev_minwu (a,b)

$d_{0:31} \leftarrow MIN_{ui}(a_{0:31}, b_{0:31})$

$d_{32:63} \leftarrow MIN_{ui}(a_{32:63}, b_{32:63})$

Each unsigned-integer word element of parameter **a** is compared to the corresponding unsigned-integer word element of parameter **b** and the smaller of the two unsigned-integer elements is placed into the corresponding word element of parameter **d**.

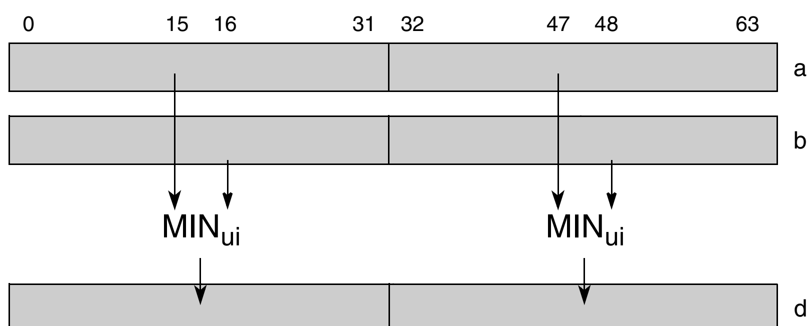


Figure 3-457. Vector Minimum Word Unsigned (evminwu)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evminwu d,a,b**

__ev_mra __ev_mra

Initialize Accumulator

d = __ev_mra (a)

```
ACC0:63 ←a0:63
d0:63 ←a0:63
```

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

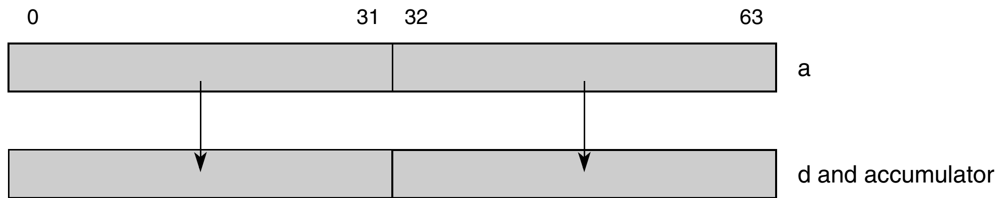


Figure 3-458. Initialize Accumulator (__ev_mra)

d	a	Maps to
__ev64_opaque__	__ev64_opaque__	**evmra d,a**

__ev_mwehgsmf[a] __ev_mwehgsmf[a]

Vector Multiply Word Even High Guarded Signed, Modulo, Fractional (to Accumulator)

d = __ev_mwehgsmf (a,b) (A = 0)

d = __ev_mwehgsmfa (a,b) (A = 1)

```
temp0:63 ←a0:31 ×sf b0:31

if (a0:31 = 0x8000_0000) &(b0:31 = 0x8000_0000) then
    d0:63 ←0x0000_8000_0000_0000
else
    d0:63 ←EXTS64(temp0:47)

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

The even word signed fractional elements in parameters **a** and **b** are multiplied. The high order 48 bits of the 64-bit product are sign-extended to 64-bits to produce an intermediate product in 17.47 fractional format. If both inputs are -1.0, the intermediate product is represented as +1.0. The intermediate product is then 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)

0 31 32 63
a
b
X
0 47
S
0 exts 15 16 17 63
S......................S
d (and accumulator if A=1)

Figure 3-459. Vector Multiply Word Even, High, Guarded, Signed, Modulo, Fractional (to Accumulator) (__ev_mwehgsmf[a])

A	d	a	b	Maps to
A = 0	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evmwehgsmf d,a,b**
A = 1	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evmwehgsmfa d,a,b**

__ev_mwehgsmfaa __ev_mwehgsmfaa

Vector Multiply Word Even High Guarded Signed, Modulo, Fractional and Accumulate

d = __ev_mwehgsmfaa (a,b)

```
temp0:63 ←a0:31 ×sf b0:31

if (a0:31 = 0x8000_0000) &(b0:31 = 0x8000_0000) then
    d0:63 ←0x0000_8000_0000_0000 + ACC0:63
else
    d0:63 ←EXTS64(temp0:47) + ACC0:63

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

The even word signed fractional elements in parameters **a** and **b** are multiplied. The high order 48 bits of the 64-bit product are sign-extended to 64-bits to produce an intermediate product in 17.47 fractional format. If both inputs are -1.0, the intermediate product is represented as +1.0. The intermediate product is then added to the contents of the accumulator and the result is placed into parameter **d** and the accumulator.

Other registers altered: ACC

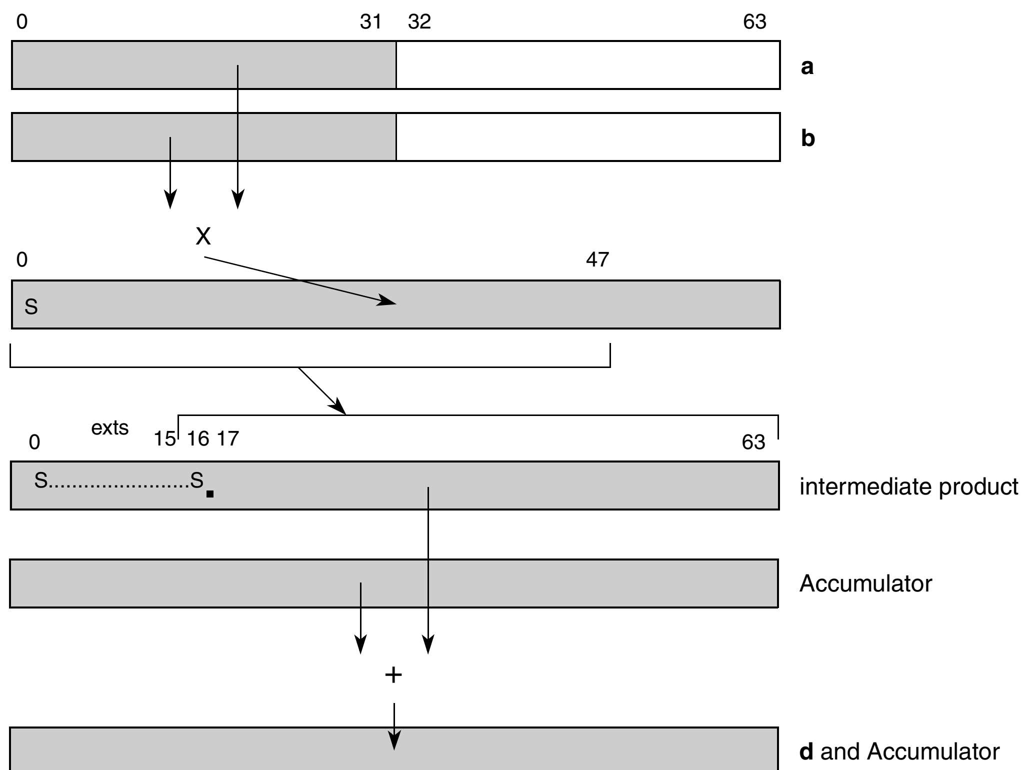


Figure 3-460. Vector Multiply Word Even, High, Guarded, Signed, Modulo, Fractional and Accumulate (__ev_mwehgsmfaa)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evmwehgsmfaa d,a,b**

__ev_mwehgsmfan __ev_mwehgsmfan

Vector Multiply Word Even High, Guarded, Signed, Modulo, Fractional and Accumulate Negative

d = __ev_mwehgsmfan (a,b)

```
temp0:63 ←a0:31 ×sf b0:31

if (a0:31 = 0x8000_0000) &(b0:31 = 0x8000_0000) then
    d0:63 ←ACC0:63 −0x0000_8000_0000_0000
else
    d0:63 ←ACC0:63 - EXTS64(temp0:47)

ACC0:63 ←d0:63
```

The even word signed fractional elements in parameters **a** and **b** are multiplied. The high order 48 bits of the 64-bit product are sign-extended to 64-bits to produce an intermediate product in 17.47 fractional format. If both inputs are -1.0, the intermediate product is represented as +1.0. The intermediate product is then subtracted from the contents of the accumulator and the result is placed into parameter **d** and the accumulator.

Other registers altered: ACC

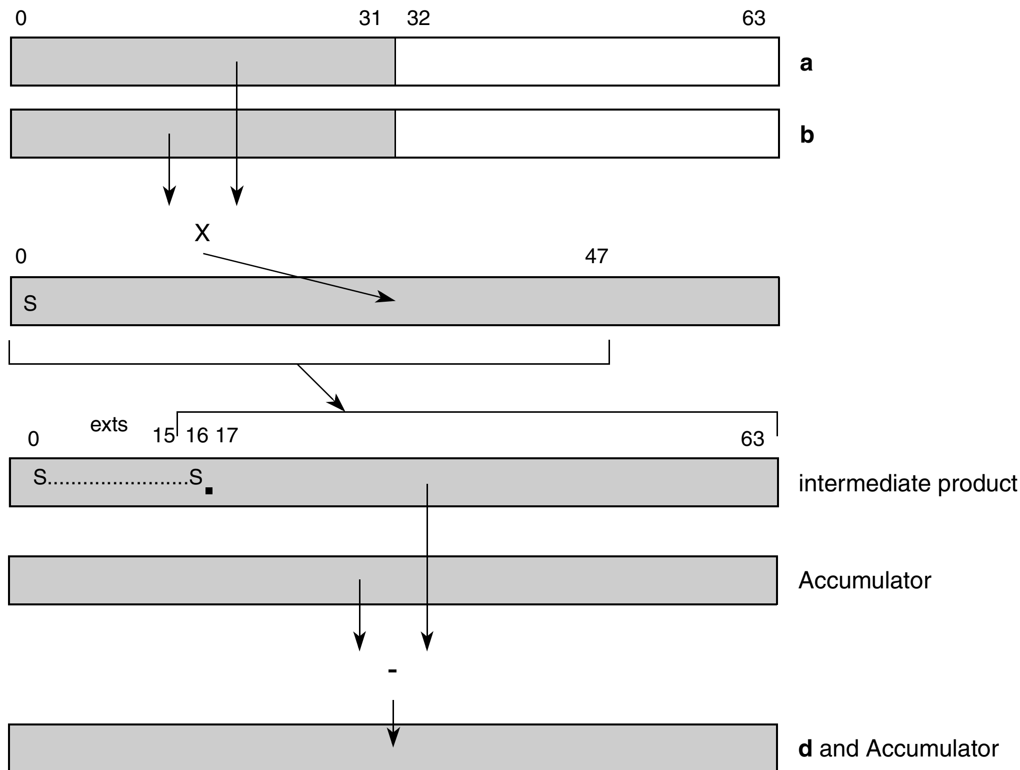


Figure 3-461. Vector Multiply Word Even, High, Guarded, Signed, Modulo, Fractional and Accumulate Negative (__ev_mwehgsmfan)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evmwehgsmfan d,a,b**

__ev_mwehgsmfr[a] __ev_mwehgsmfr[a]

Vector Multiply Word Even High Guarded Signed, Modulo, Fractional, Round (to Accumulator)

d = __ev_mwehgsmfr (a,b) (A = 0)

d = __ev_mwehgsmfra (a,b) (A = 1)

```
temp0:63 ←a0:31 ×sf b0:31

if (a0:31 = 0x8000_0000) &(b0:31 = 0x8000_0000) then
    d0:63 ←0x0000_8000_0000_0000
else
    temp0:64 ←EXTS65(temp0:63)
    tempr0:64 ←ROUND(temp0:64,16)
    d0:63 ←EXTS64(tempr0:48)

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

The even word signed fractional elements in parameters **a** and **b** are multiplied. The 64-bit fractional product is sign-extended to 65 bits, rounded to 49-bits using the current rounding mode in SPEFSCR, and the 49-bit value is sign-extended to 64-bits to produce a rounded intermediate product in 17.47 fractional format. If both inputs are -1.0, the intermediate product is represented as +1.0. The intermediate product is then 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 3-462. Vector Multiply Word Even, High, Guarded, Signed, Modulo, Fractional, Round (to Accumulator) (__ev_mwehgsmfr[a])

A	d	a	b	Maps to
A = 0	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evmwehgsmfr d,a,b**
A = 1	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evmwehgsmfra d,a,b**

__ev_mwehgsmfraa __ev_mwehgsmfraa

Vector Multiply Word Even High Guarded Signed, Modulo, Fractional, Round and Accumulate

d = __ev_mwehgsmfraa (a,b)

```
temp0:63 ←a0:31 ×sf b0:31
if (a0:31 = 0x8000_0000) &(b0:31 = 0x8000_0000) then
    d0:63 ←0x0000_8000_0000_0000 + ACC0:63
else
    temp0:64 ←EXTS65(temp0:63)
    tempr0:64 ←ROUND(temp0:64,16)
    d0:63 ←EXTS64(tempr0:48) + ACC0:63
// update accumulator
ACC0:63 ←d0:63
```

The even word signed fractional elements in parameters **a** and **b** are multiplied. The 64-bit fractional product is sign-extended to 65 bits, rounded to 49-bits using the current rounding mode in SPEFSCR, and the 49-bit value is sign-extended to 64-bits to produce an intermediate product in 17.47 fractional format. If both inputs are -1.0, the intermediate product is represented as +1.0. The intermediate product is then added to the contents of the accumulator and the result is placed into parameter **d** and the accumulator.

Other registers altered: ACC

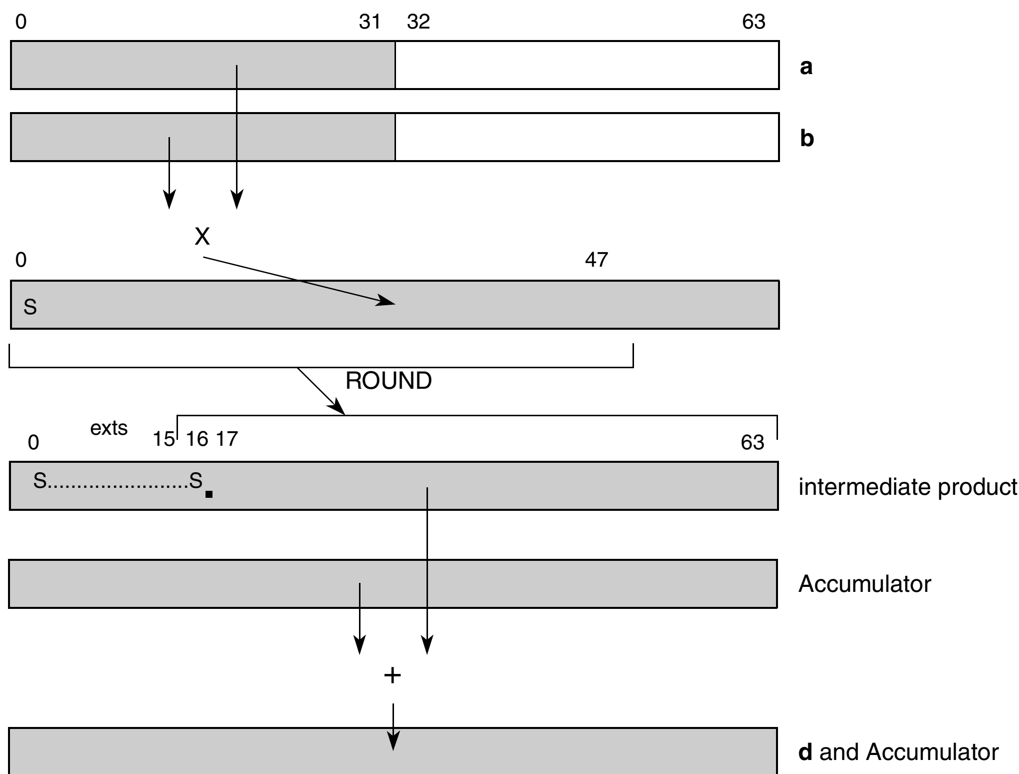


Figure 3-463. Vector Multiply Word Even, High, Guarded, Signed, Modulo, Fractional, Round and Accumulate (__ev_mwehgsmfraa)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evmwehgsmfraa d,a,b**

__ev_mwehgsmfran __ev_mwehgsmfran

Vector Multiply Word Even High, Guarded, Signed, Modulo, Fractional, Round and Accumulate Negative

d = __ev_mwehgsmfran (a,b)

```
temp0:63 ←a0:31 ×sf b0:31
if (a0:31 = 0x8000_0000) &(b0:31 = 0x8000_0000) then
    d0:63 ←ACC0:63 - 0x0000_8000_0000_0000
else
    temp0:64 ←EXTS65(temp0:63)
    tempr0:64 ←ROUND(temp0:64,16)
    d0:63 ←ACC0:63 - EXTS64(tempr0:48)
ACC0:63 ←d0:63
```

The even word signed fractional elements in parameters **a** and **b** are multiplied. The 64-bit fractional product is sign-extended to 65 bits, rounded to 49-bits using the current rounding mode in SPEFSCR, and the 49-bit value is sign-extended to 64-bits to produce an intermediate product in 17.47 fractional format. If both inputs are -1.0, the intermediate product is represented as +1.0. The intermediate product is then subtracted from the contents of the accumulator and the result is placed into parameter **d** and the accumulator.

Other registers altered: ACC

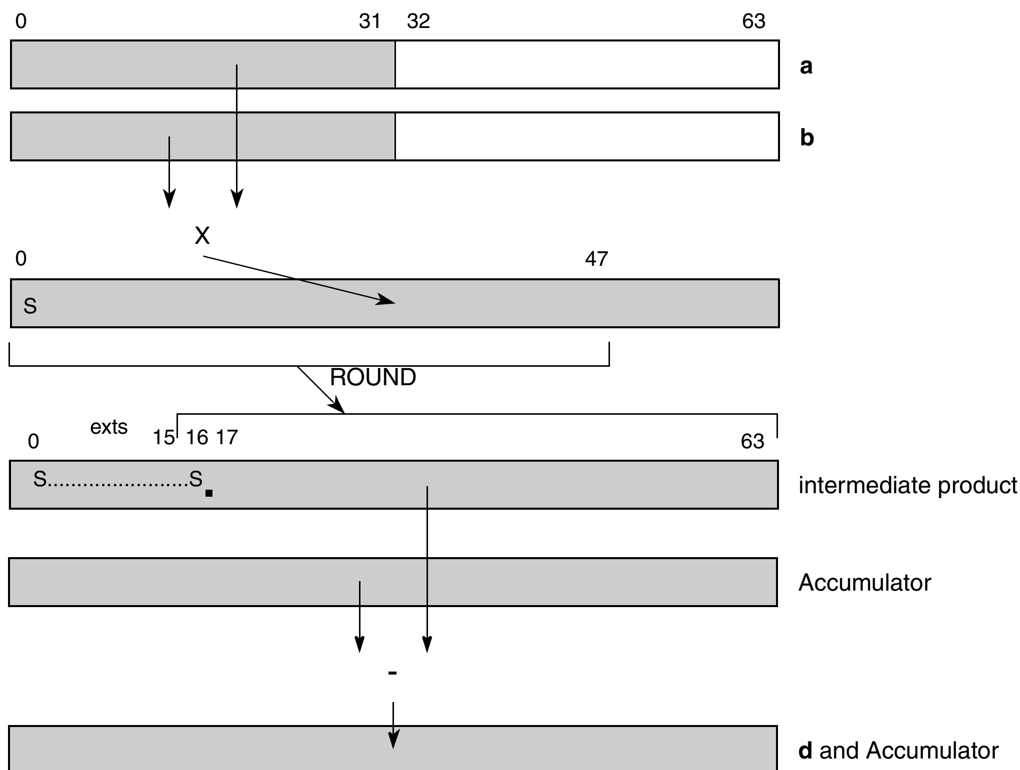


Figure 3-464. Vector Multiply Word Even, High, Guarded, Signed, Modulo, Fractional, Round, and Accumulate Negative (__ev_mwehgsmfran)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evmwehgsmfran d,a,b**

__ev_mwhsmf[a] __ev_mwhsmf[a]

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 3-465. Vector Multiply Word High Signed, Modulo, Fractional (to Accumulator) (__ev_mwhsmf[a])

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[a] __ev_mwhsmi[a]

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 3-466. Vector Multiply Word High Signed, Modulo, Integer (to Accumulator) (__ev_mwhsmi[a])

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[a] __ev_mwhssf[a]

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)

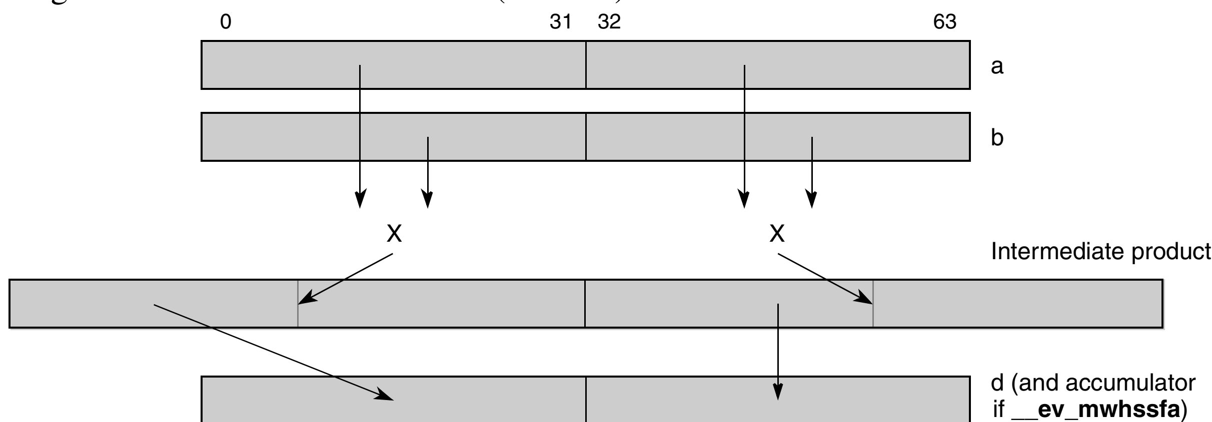


Figure 3-467. Vector Multiply Word High Signed, Saturate, Fractional (to Accumulator) (__ev_mwhssf[a])

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_mwhssfaaw __ev_mwhssfaaw

Vector Multiply Word High Signed, Saturate, Fractional and Accumulate into Words

d = __ev_mwhssfaaw (a,b)

```
// high
if (a0:31 = 0x8000_0000) &(b0:31 = 0x8000_0000) then
    temph0:63 ←0x7FFF_FFFF_FFFF_FFFF //saturate
    movh ←1
else
    temph0:63 ←a0:31 ×sf b0:31
    movh ←0
temp0:63 ←EXTS(ACC0:31) + EXTS(temph0:31)
ovh ←(temp31 ⊕ temp32)
d0:31 ←SATURATE(ovh, temp31, 0x8000_0000, 0x7FFF_FFFF, temp32:63)

// low
if (a32:63 = 0x8000_0000) &(b32:63 = 0x8000_0000) then
    d32:63 ←0x7FFF_FFFF_FFFF_FFFF //saturate
    movl ←1
else
    templ0:63 ←a32:63 ×sf b32:63
    movl ←0
temp0:63 ←EXTS(ACC32:63) + EXTS(templ0:31)
ovl ←(temp31 ⊕ temp32)
d32:63 ←SATURATE(ovl, temp31, 0x8000_0000, 0x7FFF_FFFF, temp32:63)

// update accumulator
ACC0:63 ←d0:63

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

The corresponding signed fractional word elements in parameters **a** and **b** are multiplied producing a 32-bit product. If both inputs are -1.0, the result saturates to the largest positive signed fraction. Bits 0-31 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 either the multiply or the addition, the overflow and summary overflow bits are recorded in the SPEFSCR.

Other registers altered: SPEFSCR ACC

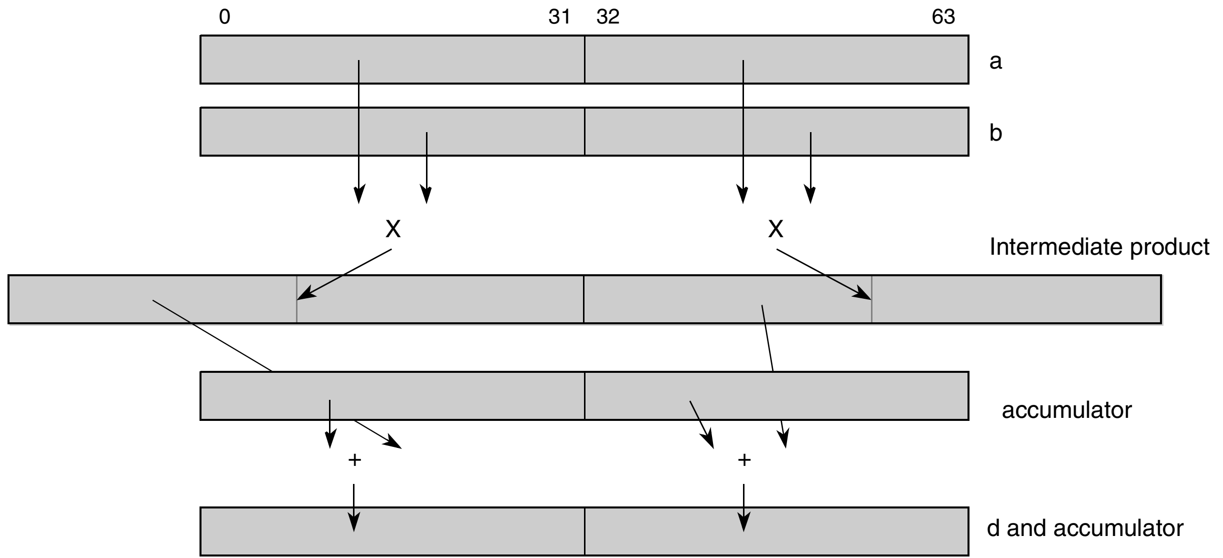


Figure 3-468. Vector Multiply Word High Signed, Saturate, Fractional and Accumulate into Words (__ev_mwhssfaaw)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evmwhssfaaw d,a,b**

__ev_mwhssfaaw3 __ev_mwhssfaaw3

Vector Multiply Word High Signed, Saturate, Fractional and Accumulate into Words 3 operand

d = __ev_mwhssfaaw3 (a,b,c)

```
// high
if (b0:31 = 0x8000_0000) &(c0:31 = 0x8000_0000) then
    temph0:63 ←0x7FFF_FFFF_FFFF_FFFF //saturate
    movh ←1
else
    temph0:63 ←b0:31 ×sf c0:31
    movh ←0
temp0:63 ←EXTS64(a0:31) + EXTS64(temph0:31)
ovh ←(temp31 ⊕ temp32)
d0:31 ←SATURATE(ovh, temp31, 0x8000_0000, 0x7FFF_FFFF, temp32:63)

// low
if (b32:63 = 0x8000_0000) &(c32:63 = 0x8000_0000) then
    a32:63 ←0x7FFF_FFFF_FFFF_FFFF //saturate
    movl ←1
else
    templ0:63 ←b32:63 ×sf c32:63
    movl ←0
temp0:63 ←EXTS64(a32:63) + EXTS64(templ0:31)
ovl ←(temp31 ⊕ temp32)
d32:63 ←SATURATE(ovl, temp31, 0x8000_0000, 0x7FFF_FFFF, temp32:63)

// update accumulator
ACC0:63 ←d0:63

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

The corresponding signed fractional word elements in parameters **b** and **c** are multiplied producing a 32-bit product. If both inputs are -1.0, the result saturates to the largest positive signed fraction. Bits 0:31 of each product are then added to the corresponding word in parameter **a**, saturating if overflow or underflow occurs, and the result is placed into 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

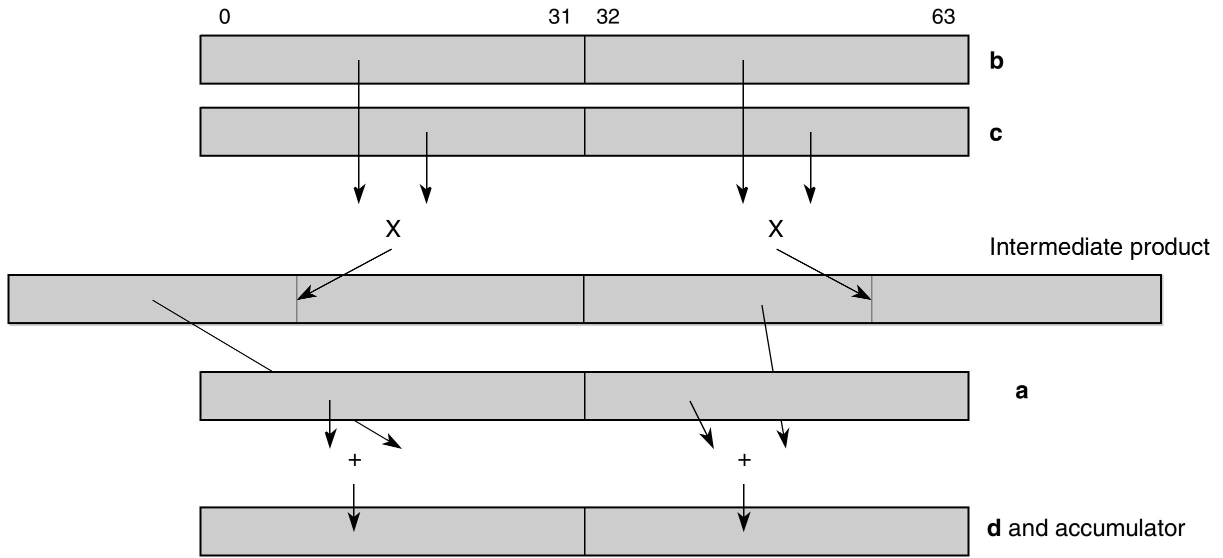


Figure 3-469. Vector Multiply Word High Signed, Saturate, Fractional and Accumulate into Words 3 op (__ev_mwhssfaaw3)

d	a	b	c	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**d ←a evmwhssfaaw3 d,b,c**

__ev_mwhssfanw __ev_mwhssfanw

Vector Multiply Word High Signed, Saturate, Fractional and Accumulate Negative into Words

d = __ev_mwhssfanw (a,b)

```
// high
if (a0:31 = 0x8000_0000) &(b0:31 = 0x8000_0000) then
    temph0:63 ←0x7FFF_FFFF_FFFF_FFFF //saturate
    movh ←1
else
    temph0:63 ←a0:31 ×sf b0:31
    movh ←0
temp0:63 ←EXTS(ACC0:31) - EXTS(temph0:31)
ovh ←(temp31 ⊕ temp32)
d0:31 ←SATURATE(ovh, temp31, 0x8000_0000, 0x7FFF_FFFF, temp32:63)

// low
if (a32:63 = 0x8000_0000) &(b32:63 = 0x8000_0000) then
    d32:63 ←0x7FFF_FFFF_FFFF_FFFF //saturate
    movl ←1
else
    templ0:63 ←a32:63 ×sf b32:63
    movl ←0
temp0:63 ←EXTS(ACC32:63) - EXTS(templ0:31)
ovl ←(temp31 ⊕ temp32)
d32:63 ←SATURATE(ovl, temp31, 0x8000_0000, 0x7FFF_FFFF, temp32:63)

// update accumulator
ACC0:63 ←d0:63

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

The corresponding signed fractional word elements in parameters **a** and **b** are multiplied producing a 32-bit product. If both inputs are -1.0, the result saturates to the largest positive signed fraction. Bits 0-31 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 either the multiply or the addition, the overflow and summary overflow bits are recorded in the SPEFSCR.

Other registers altered: SPEFSCR ACC

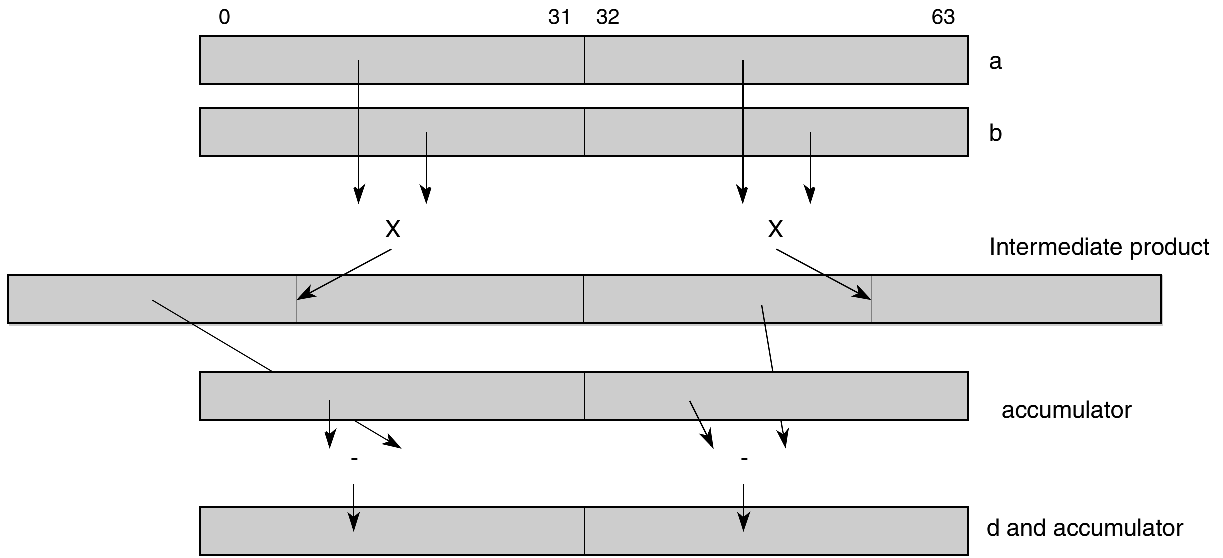


Figure 3-470. Vector Multiply Word High Signed, Saturate, Fractional and Accumulate Negative into Words (__ev_mwhssfanw)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evmwhssfanw d,a,b**

__ev_mwhssfanw3 __ev_mwhssfanw3

Vector Multiply Word High Signed, Saturate, Fractional and Accumulate Negative into Words 3 operand

d = __ev_mwhssfanw3 (a,b,c)

```
// high
if (b0:31 = 0x8000_0000) &(c0:31 = 0x8000_0000) then
    temph0:63 ←0x7FFF_FFFF_FFFF_FFFF //saturate
    movh ←1
else
    temph0:63 ←b0:31 ×sf c0:31
    movh ←0
temp0:63 ←EXTS64(d0:31) - EXTS64(temph0:31)
ovh ←(temp31 ⊕ temp32)
d0:31 ←SATURATE(ovh, temp31, 0x8000_0000, 0x7FFF_FFFF, temp32:63)

// low
if (b32:63 = 0x8000_0000) &(c32:63 = 0x8000_0000) then
    a32:63 ←0x7FFF_FFFF_FFFF_FFFF //saturate
    movl ←1
else
    templ0:63 ←b32:63 ×sf c32:63
    movl ←0
temp0:63 ←EXTS64(a32:63) - EXTS64(templ0:31)
ovl ←(temp31 ⊕ temp32)
d32:63 ←SATURATE(ovl, temp31, 0x8000_0000, 0x7FFF_FFFF, temp32:63)

// update accumulator
ACC0:63 ←d0:63

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

The corresponding signed fractional word elements in parameters **b** and **c** are multiplied producing a 32-bit product. If both inputs are -1.0, the result saturates to the largest positive signed fraction. Bits 0:31 of each product are then subtracted from the corresponding word in parameter **a**, saturating if overflow or underflow occurs, and the result is placed into 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

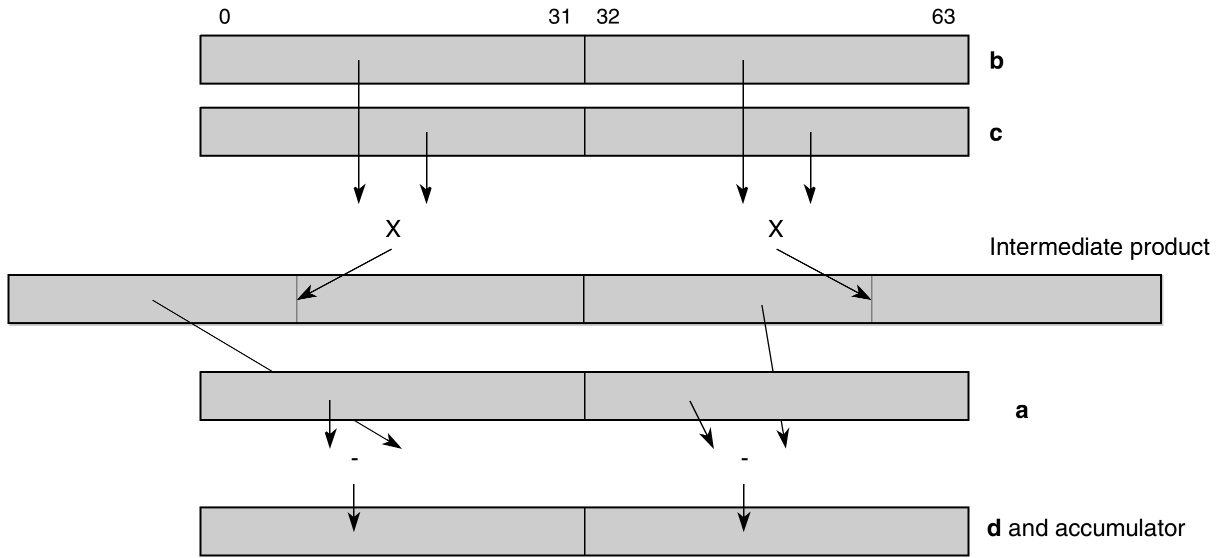


Figure 3-471. Vector Multiply Word High Signed, Saturate, Fractional and Accumulate Negative into Words 3 op (__ev_mwhssfanw3)

d	a	b	c	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**d ←a** **evmwhssfanw3 d,b,c**

__ev_mwhssfr[a] __ev_mwhssfr[a]

Vector Multiply Word High Signed, Saturate, Fractional and Round (to Accumulator)

d = __ev_mwhssfr (a,b) (A = 0)

d = __ev_mwhssfra (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
    tempr0:32 ←temp0:32 + 1
    d0:31 ←tempr0:31
    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
    tempr0:32 ←temp0:32 + 1
    d32:63 ←tempr0:31
    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 0-32 of each product are incremented, and bits 0-31 of the sum are placed into the corresponding words of parameter **d**, and the accumulator if A = 1. If both inputs are -1.0, the result saturates to 0x7FFF_FFFF, and the overflow and summary overflow bits are recorded in the SPEFSCR.

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

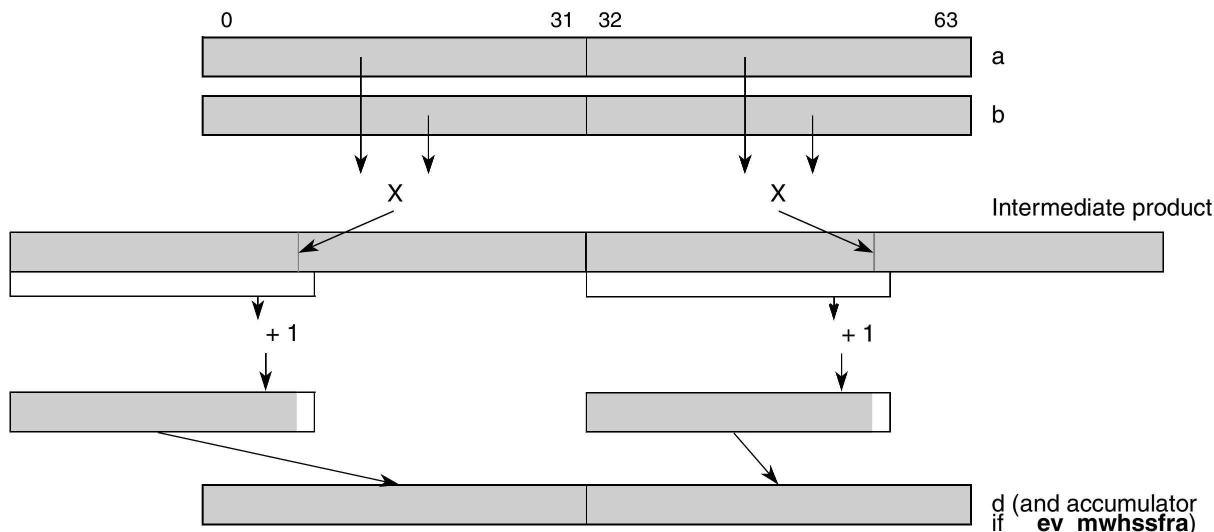


Figure 3-472. Vector Multiply Word High Signed, Saturate, Fractional and Round (to Accumulator) (__ev_mwhssfr[a])

A	d	a	b	Maps to
A = 0	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evmwhssfr d,a,b**
A = 1	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evmwhssfra d,a,b**

__ev_mwhssfraaw __ev_mwhssfraaw

Vector Multiply Word High Signed, Saturate, Fractional, Round and Accumulate into Words

d = __ev_mwhssfraaw (a,b)

```
// high
if (a0:31 = 0x8000_0000) &(b0:31 = 0x8000_0000) then {
    tempr0:63 ←0x7FFF_FFFF_FFFF_FFFF //saturate
    movh ←1}
else {
    temph0:63 ←a0:31 ×sf b0:31
    tempr0:32 ←temph0:32 + 1
    movh ←0}
temp0:63 ←EXTS(ACC0:31) + EXTS(tempr0:31)
ovh ←(temp31 ⊕ temp32)
d0:31 ←SATURATE(ovh, temp31, 0x8000_0000, 0x7FFF_FFFF, temp32:63)

// low
if (a32:63 = 0x8000_0000) &(b32:63 = 0x8000_0000) then {
    tempr0:63 ←0x7FFF_FFFF_FFFF_FFFF //saturate
    movl ←1}
else {
    templ0:63 ←a32:63 ×sf b32:63
    tempr0:32 ←templ0:32 + 1
    movl ←0}
temp0:63 ←EXTS(ACC32:63) + EXTS(tempr0:31)
ovl ←(temp31 ⊕ temp32)
d32:63 ←SATURATE(ovl, temp31, 0x8000_0000, 0x7FFF_FFFF, temp32:63)

// update accumulator
ACC0:63 ←d0:63

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

The corresponding signed fractional word elements in parameters **a** and **b** are multiplied. Bits 0-32 of each product are incremented to obtain an intermediate result. If both inputs are -1.0, the intermediate result saturates to the largest positive signed fraction. Bits 0-31 of each intermediate result 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 either the multiply or the addition, the overflow and summary overflow bits are recorded in the SPEFSCR.

Other registers altered: SPEFSCR ACC

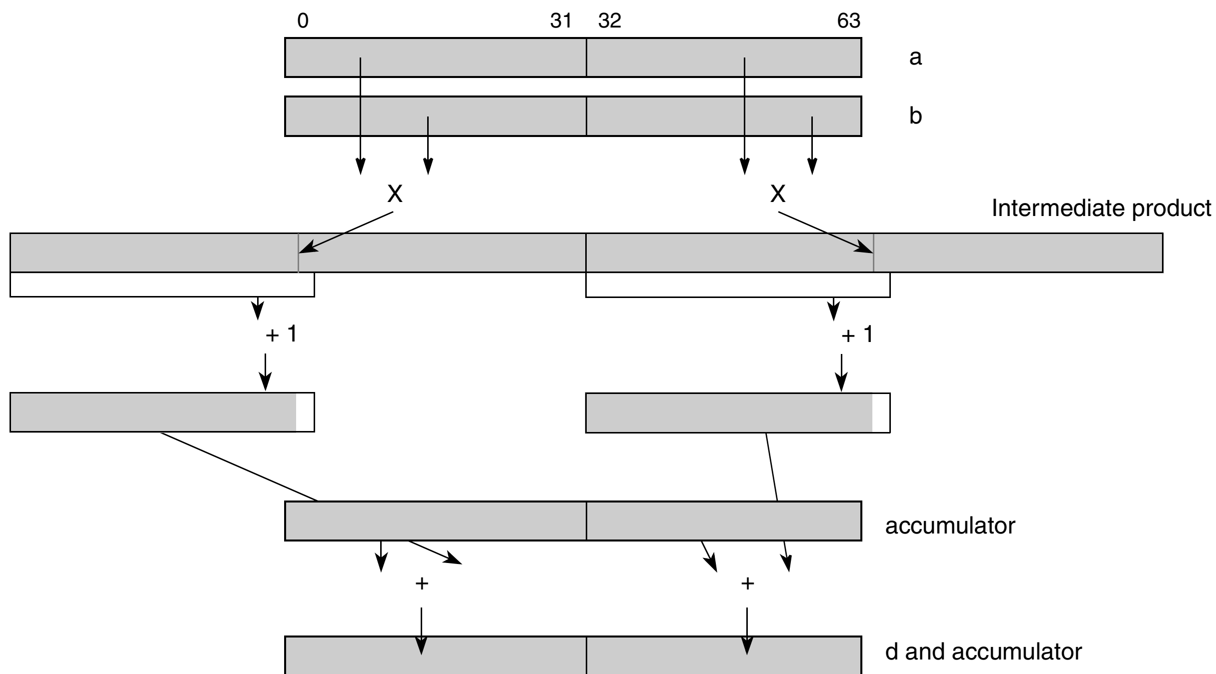


Figure 3-473. Vector Multiply Word High Signed, Saturate, Fractional, Round, and Accumulate into Words (__ev_mwhssfraaw)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evmwhssfraaw d,a,b**

__ev_mwhssfraaw3 __ev_mwhssfraaw3

Vector Multiply Word High Signed, Saturate, Fractional, Round and Accumulate into Words 3 operand

d = __ev_mwhssfraaw3 (a,b,c)

```
// high
temph0:63 ←b0:31 ×sf c0:31
if (b0:31 = 0x8000_0000) &(c0:31 = 0x8000_0000) then {
    temph0:63 ←0x7FFF_FFFF_0000_0000 //saturate
    movh ←1}
else
    movh ←0
temph0:66 ←ROUND((EXTS67(a0:31 || 320) + EXTS67(temph0:63)),32)
ovh ←chk_ovf(temph0:3)
d0:31 ←SATURATE(ovh, temph0, 0x8000_0000, 0x7FFF_FFFF, temph3:34)

// low
templ0:63 ←b32:63 ×sf c32:63
if (b32:63 = 0x8000_0000) &(c32:63 = 0x8000_0000) then {
    templ0:63 ←0x7FFF_FFFF_0000_0000 //saturate
    movl ←1}
else
    movl ←0
templ0:66 ←ROUND((EXTS67(a32:63 || 320) + EXTS67(templ0:63)),32)
ovl ←chk_ovf(templ0:3)
d32:63 ←SATURATE(ovl, templ0, 0x8000_0000, 0x7FFF_FFFF, templ3:34)

// update accumulator
ACC0:63 ←d0:63

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

The corresponding signed fractional word elements in parameters **b** and **c** are multiplied. If both inputs are -1.0, the intermediate result saturates to the largest positive signed fraction. The 64-bit products are added to the corresponding zero-padded word of parameter **a**, the sums are rounded to 32 bits using the current fractional rounding mode in SPEFCSR, saturating if overflow or underflow occurs, and the results are placed into the corresponding word of parameter **d** and the accumulator. If there is an overflow or underflow from either the multiply or the addition with round, the overflow and summary overflow bits are recorded in the SPEFSCR.

Other registers altered: SPEFSCR ACC

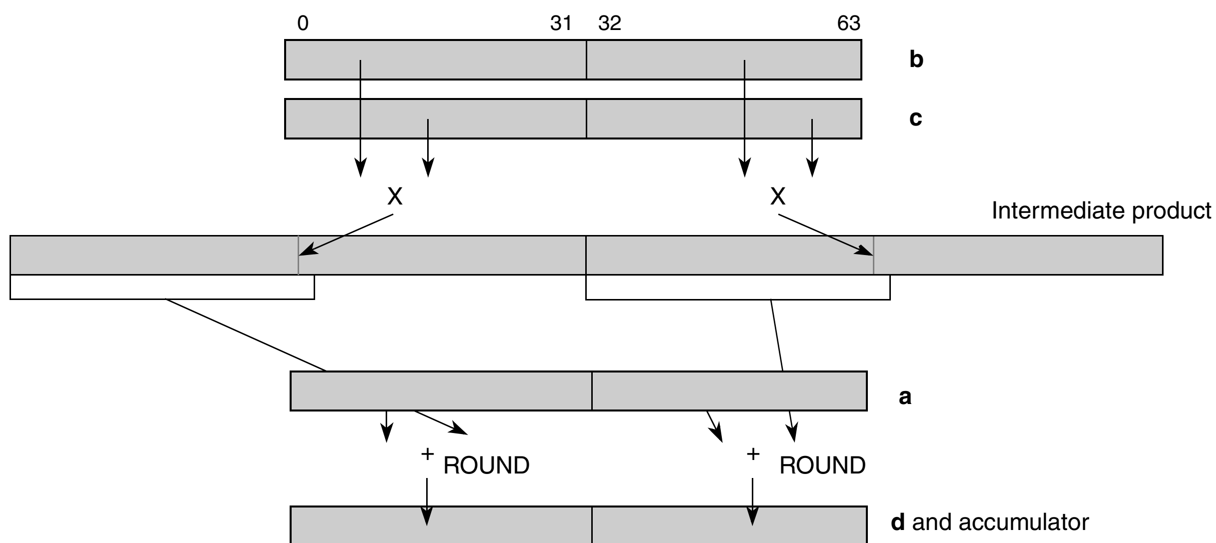


Figure 3-474. Vector Multiply Word High Signed, Saturate, Fractional, Round, and Accumulate into Words 3 op (__ev_mwhssfraaw3)

d	a	b	c	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**d ←a** **evmwhssfraaw3 d,b,c**

__ev_mwhssfranw __ev_mwhssfranw

Vector Multiply Word High Signed, Saturate, Fractional, Round and Accumulate Negative into Words

d = __ev_mwhssfranw (a,b)

```
// high
if (a0:31 = 0x8000_0000) &(b0:31 = 0x8000_0000) then {
    tempr0:63 ←0x7FFF_FFFF_FFFF_FFFF //saturate
    movh ←1}
else {
    temph0:63 ←a0:31 ×sf b0:31
    tempr0:32 ←temph0:32 + 1
    movh ←0}
temp0:63 ←EXTS(ACC0:31) - EXTS(tempr0:31)
ovh ←(temp31 ⊕ temp32)
d0:31 ←SATURATE(ovh, temp31, 0x8000_0000, 0x7FFF_FFFF, temp32:63)

// low
if (a32:63 = 0x8000_0000) &(b32:63 = 0x8000_0000) then {
    tempr0:63 ←0x7FFF_FFFF_FFFF_FFFF //saturate
    movl ←1}
else {
    templ0:63 ←a32:63 ×sf b32:63
    tempr0:32 ←templ0:32 + 1
    movl ←0}
temp0:63 ←EXTS(ACC32:63) - EXTS(tempr0:31)
ovl ←(temp31 ⊕ temp32)
d32:63 ←SATURATE(ovl, temp31, 0x8000_0000, 0x7FFF_FFFF, temp32:63)

// update accumulator
ACC0:63 ←d0:63

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

The corresponding signed fractional word elements in parameters **a** and **b** are multiplied. Bits 0-32 of each product are incremented to obtain an intermediate result. If both inputs are -1.0, the intermediate result saturates to the largest positive signed fraction. Bits 0-31 of each intermediate result 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 either the multiply or the subtract, the overflow and summary overflow bits are recorded in the SPEFSCR.

Other registers altered: SPEFSCR ACC

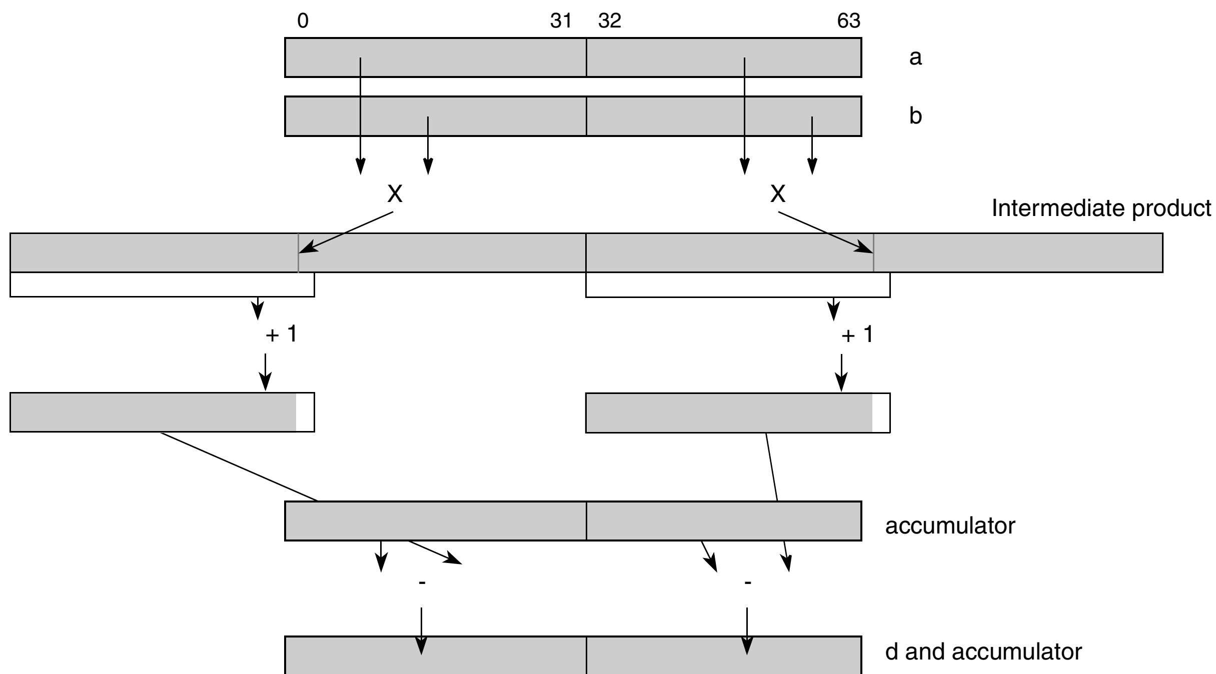


Figure 3-475. Vector Multiply Word High Signed, Saturate, Fractional, Round, and Accumulate Negative into Words (__ev_mwhssfranw)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evmwhssfranw d,a,b**

__ev_mwhssfranw3 __ev_mwhssfranw3

Vector Multiply Word High Signed, Saturate, Fractional, Round and Accumulate Negative into Words 3 operand

d = __ev_mwhssfranw3 (a,b,c)

```
// high
temph0:63 ←b0:31 ×sf c0:31
if (b0:31 = 0x8000_0000) &(c0:31 = 0x8000_0000) then {
    temph0:63 ←0x7FFF_FFFF_0000_0000 //saturate
    movh ←1}
else
    movh ←0
temph0:66 ←ROUND((EXTS67(a0:31 || 320) - EXTS67(temph0:63)),32)
ovh ←chk_ovf(temph0:3)
d0:31 ←SATURATE(ovh, temph0, 0x8000_0000, 0x7FFF_FFFF, temph3:34)

// low
templ0:63 ←b32:63 ×sf c32:63
if (b32:63 = 0x8000_0000) &(c32:63 = 0x8000_0000) then {
    templ0:63 ←0x7FFF_FFFF_0000_0000 //saturate
    movl ←1}
else
    movl ←0
templ0:66 ←ROUND((EXTS67(a32:63 || 320) - EXTS67(templ0:63)),32)
ovl ←chk_ovf(templ0:3)
d32:63 ←SATURATE(ovl, templ0, 0x8000_0000, 0x7FFF_FFFF, templ3:34)

// update accumulator
ACC0:63 ←d0:63

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

The corresponding signed fractional word elements in parameters **b** and **c** are multiplied. If both inputs are -1.0, the intermediate result saturates to the largest positive signed fraction. The 64-bit products are subtracted from the corresponding zero-padded word of **a**, the differences are rounded to 32 bits using the current fractional rounding mode in SPEFCSR, saturating if overflow or underflow occurs, and the results are placed into the corresponding word of **d** and the accumulator. If there is an overflow or underflow from either the multiply or the subtraction with round, the overflow and summary overflow bits are recorded in the SPEFSCR.

Other registers altered: SPEFSCR ACC

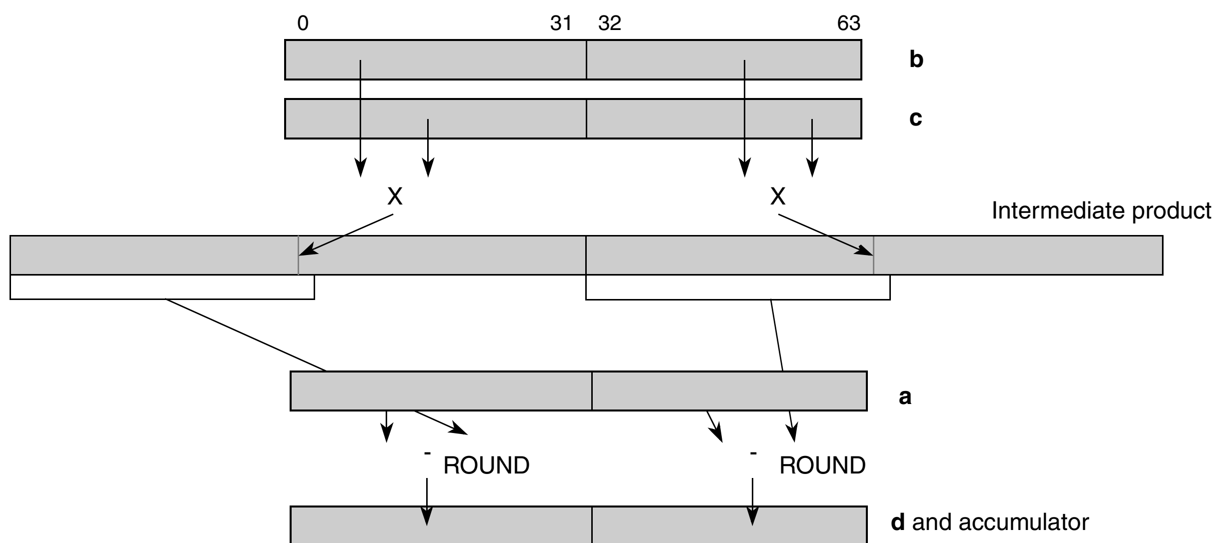


Figure 3-476. Vector Multiply Word High Signed, Saturate, Fractional, Round, and Accumulate Negative into Words 3 op (__ev_mwhssfranw3)

d	a	b	c	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	d ←a evmwhssfranw3 d,b,c

__ev_mwhumi[a] __ev_mwhumi[a]

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 3-477. Vector Multiply Word High Unsigned, Modulo, Integer (to Accumulator) (__ev_mwhumi[a])

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

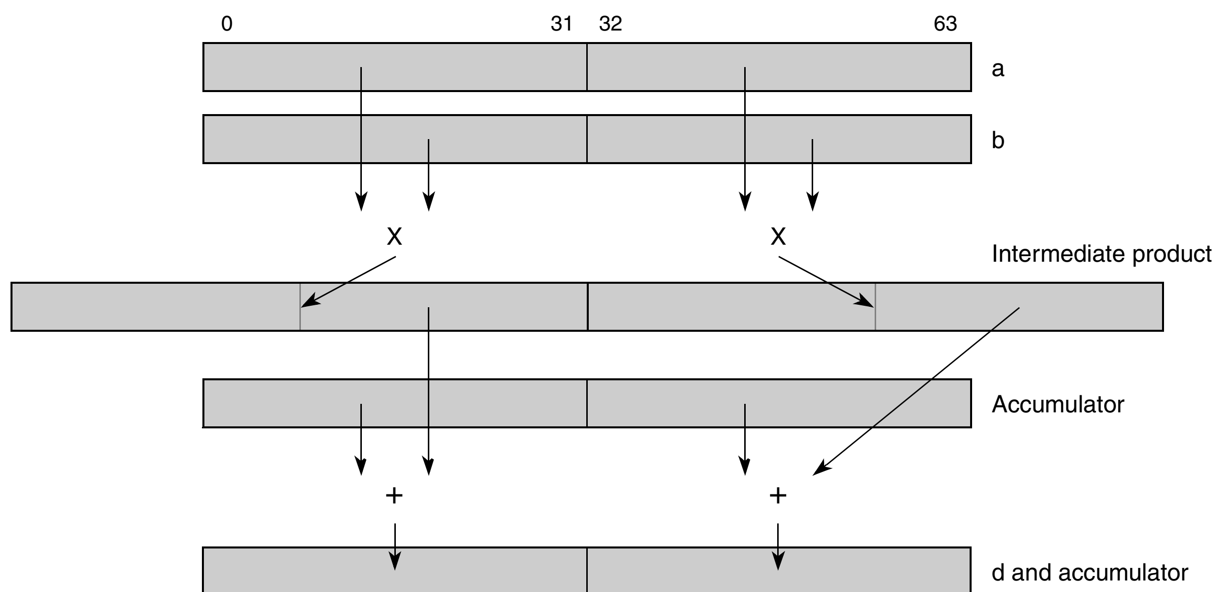


Figure 3-478. Vector Multiply Word Low Signed, Modulo, Integer and Accumulate in Words (__ev_mwlsmiaaw)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evmwlsmiaaw d,a,b**

__ev_mwlsmiaaw3 __ev_mwlsmiaaw3

Vector Multiply Word Low Signed, Modulo, Integer and Accumulate in Words 3 operand

d = __ev_mwlsmiaaw3 (a,b,c)

```
// high
temp0:63 ←b0:31 ×si c0:31
d0:31 ←a0:31 + temp32:63

// low
temp0:63 ←b32:63 ×si c32:63
d32:63 ←a32:63 + temp32:63

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

For each word element in the accumulator, the corresponding word signed integer elements in parameters **b** and **c** are multiplied. The least significant 32 bits of each intermediate product is added to the contents of the corresponding parameter **a** words, and the result is placed into parameter **d** and the accumulator.

Other registers altered: ACC

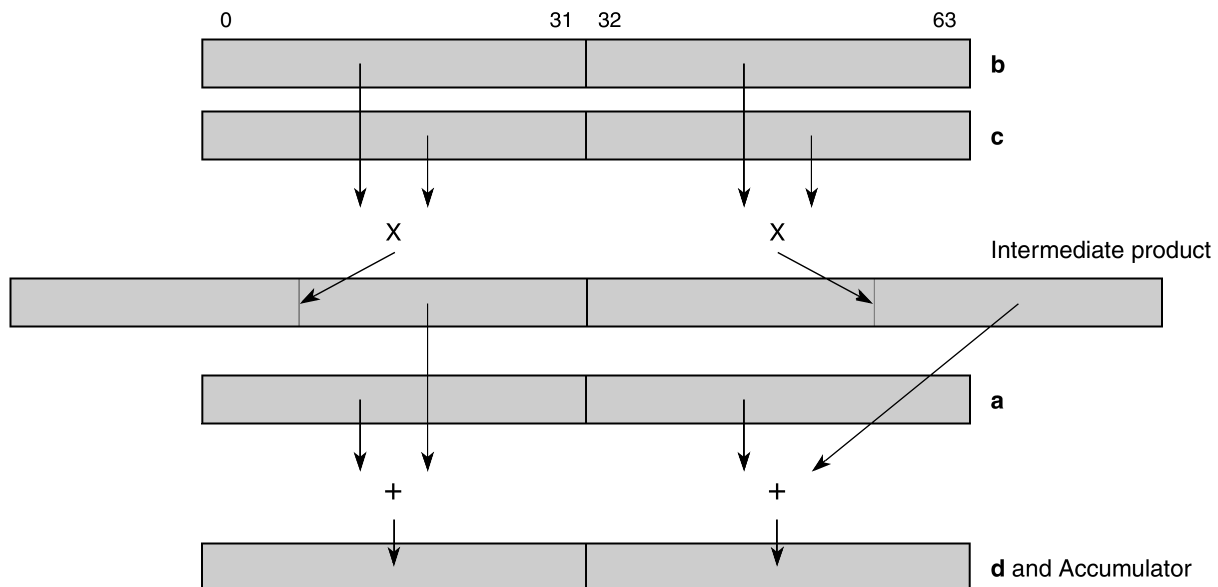


Figure 3-479. Vector Multiply Word Low Signed, Modulo, Integer and Accumulate in Words 3 op (__ev_mwlsmiaaw3)

d	a	b	c	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**d ←a evmwlsmiaaw3 d,b,c**

__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

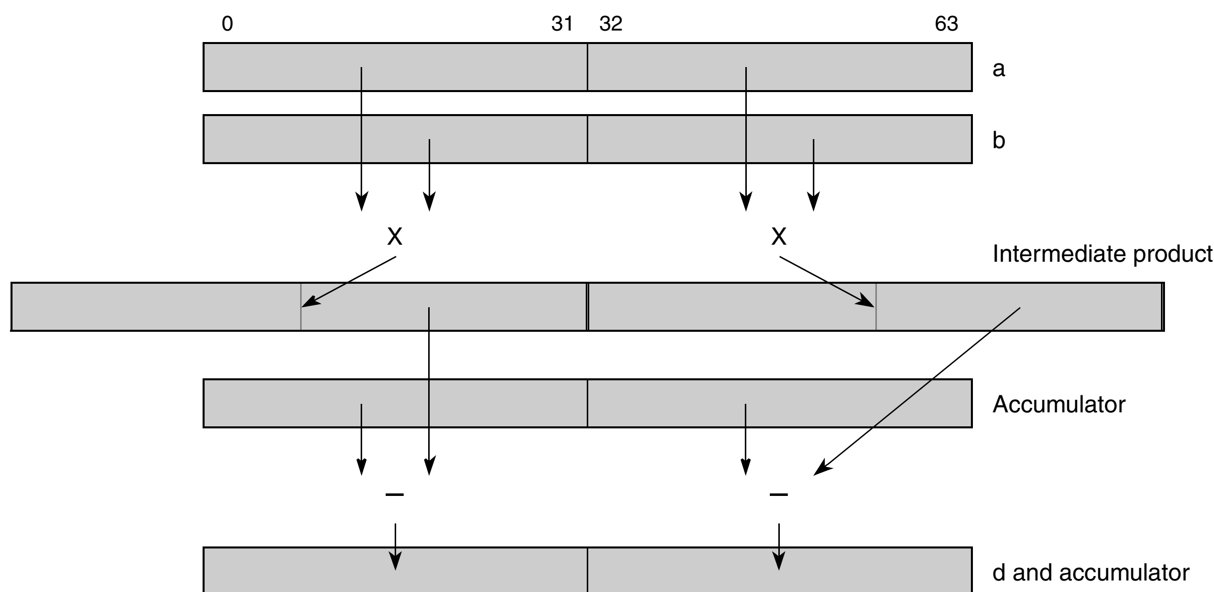


Figure 3-480. Vector Multiply Word Low Signed, Modulo, Integer and Accumulate Negative in Words (__ev_mwlsmianw)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evmwlsmfanw d,a,b**

__ev_mwlsmianw3 __ev_mwlsmianw3

Vector Multiply Word Low Signed, Modulo, Integer and Accumulate Negative in Words 3 operand

d = __ev_mwlsmianw3 (a,b)

```
// high
temp0:63 ←b0:31 ×si c0:31
d0:31 ←a0:31 - temp32:63

// low
temp0:63 ←b32:63 ×si c32:63
d32:63 ←a32:63 - temp32:63

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

For each word element in the accumulator, the corresponding word elements in parameters **b** and **c** 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 into parameter **d** and the accumulator.

Other registers altered: ACC

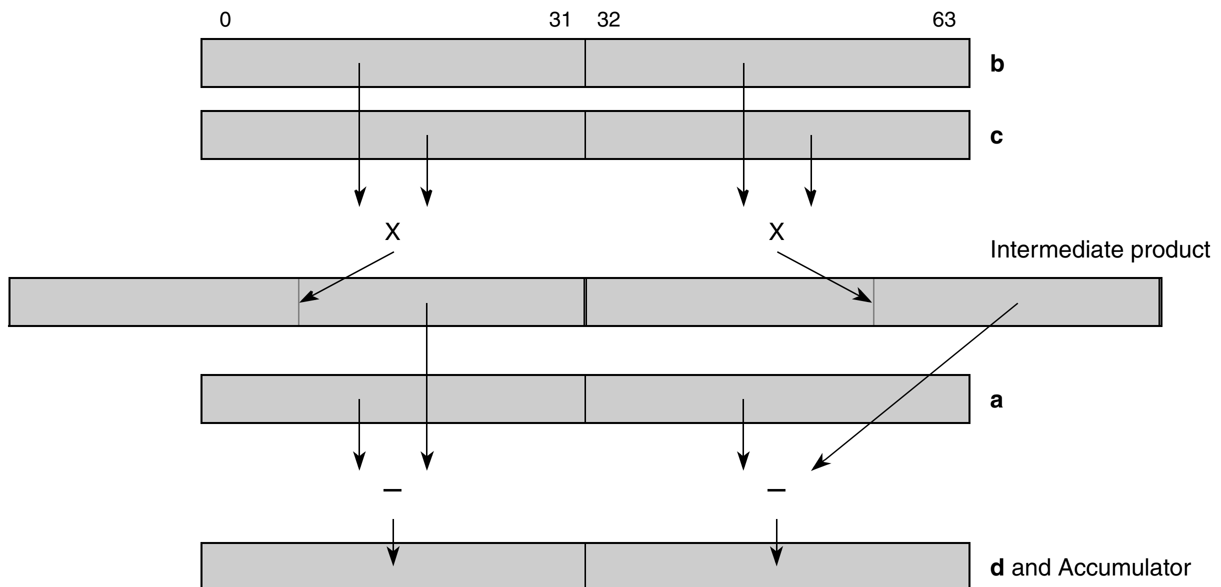


Figure 3-481. Vector Multiply Word Low Signed, Modulo, Integer and Accumulate Negative in Words 3 op (__ev_mwlsmianw3)

d	a	b	c	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	d ←a **evmwlsmianw3 d,b,c**

__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 ←EXTS64(ACC0:31) + EXTS64(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 ←EXTS64(ACC32:63) + EXTS64(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 is then added to the corresponding word in the accumulator saturating if overflow or underflow occurs, and the result is placed into 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

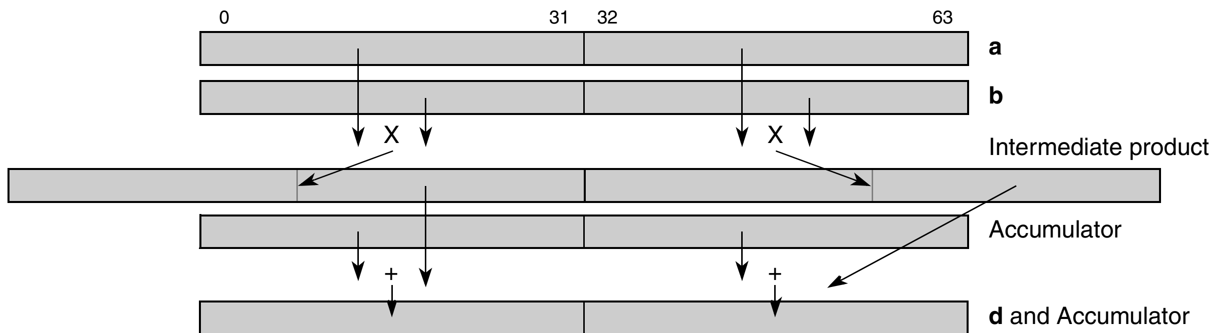


Figure 3-482. Vector Multiply Word Low Signed, Saturate, Integer and Accumulate in Words (__ev_mwlssiaaw)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evmwlssiaaw d,a,b**

__ev_mwlssiaaw3 __ev_mwlssiaaw3

Vector Multiply Word Low Signed, Saturate, Integer and Accumulate in Words

d = __ev_mwlssiaaw3 (a,b,c)

```
// high
temp0:63 ←b0:31 ×si c0:31
temp0:63 ←EXTS64(a0:31) + EXTS64(temp32:63)
ovh ←(temp31 ⊕ temp32)
d0:31 ←SATURATE(ovh, temp31, 0x8000_0000, 0x7FFF_FFFF, temp32:63)

// low
temp0:63 ←b32:63 ×si c32:63
temp0:63 ←EXTS64(a32:63) + EXTS64(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 **b** and **c** are multiplied producing a 64-bit product. The least significant 32 bits of each product is then added to the corresponding word in parameter **a** saturating if overflow or underflow occurs, and the result is placed into 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

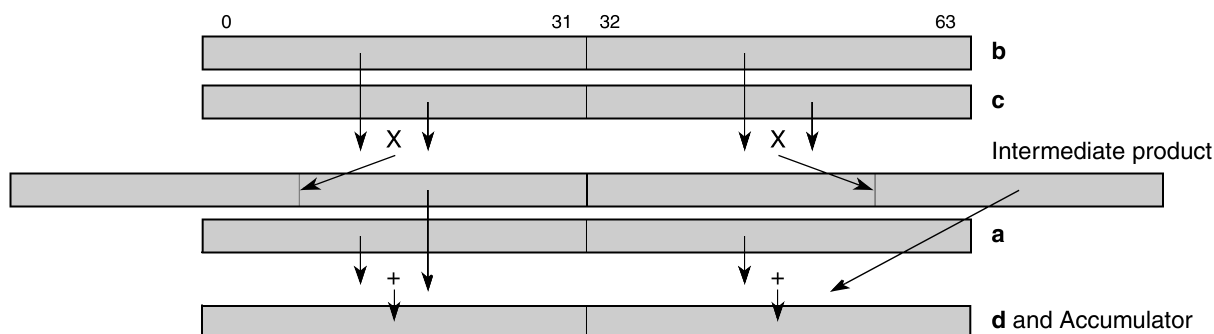


Figure 3-483. Vector Multiply Word Low Signed, Saturate, Integer and Accumulate in Words 3 op (__ev_mwlssiaaw3)

d	a	b	c	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	d ←a __ev_mwlssiaaw3 d,b,c

__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 ←EXTS64(ACC0:31) - EXTS64(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 ←EXTS64(ACC32:63) - EXTS64(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 is then subtracted from the corresponding word in the accumulator saturating if overflow or underflow occurs, and the result is placed into 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

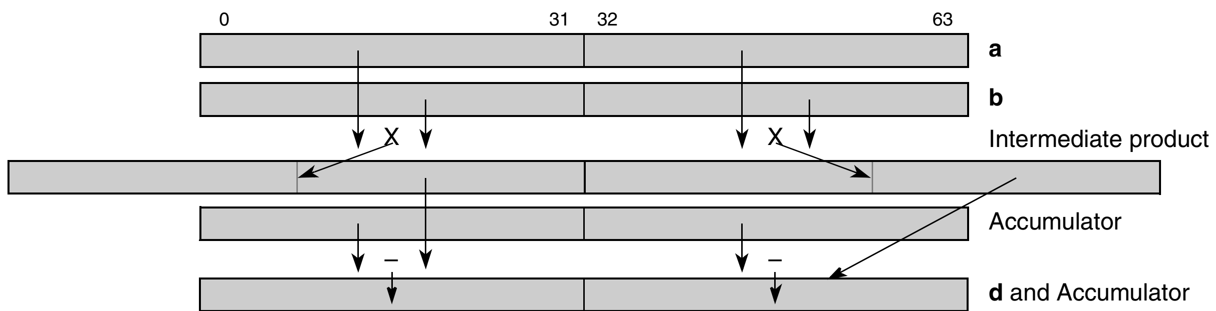


Figure 3-484. Vector Multiply Word Low Signed, Saturate, Integer and Accumulate Negative in Words (__ev_mwlssianw)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evmwlssianw d,a,b**

__ev_mwlssianw3 __ev_mwlssianw3

Vector Multiply Word Low Signed, Saturate, Integer and Accumulate Negative in Words 3 operand

d = __ev_mwlssianw3 (a,b,c)

```
// high
temp0:63 ←b0:31 ×si c0:31
temp0:63 ←EXTS64(a0:31) - EXTS64(temp32:63)
ovh ←(temp31 ⊕ temp32)
d0:31 ←SATURATE(ovh, temp31, 0x8000_0000, 0x7FFF_FFFF, temp32:63)

// low
temp0:63 ←b32:63 ×si c32:63
temp0:63 ←EXTS64(a32:63) - EXTS64(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 **b** and **c** are multiplied producing a 64-bit product. The least significant 32 bits of each product is then subtracted from the corresponding word in parameter **a** saturating if overflow or underflow occurs, and the result is placed into 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

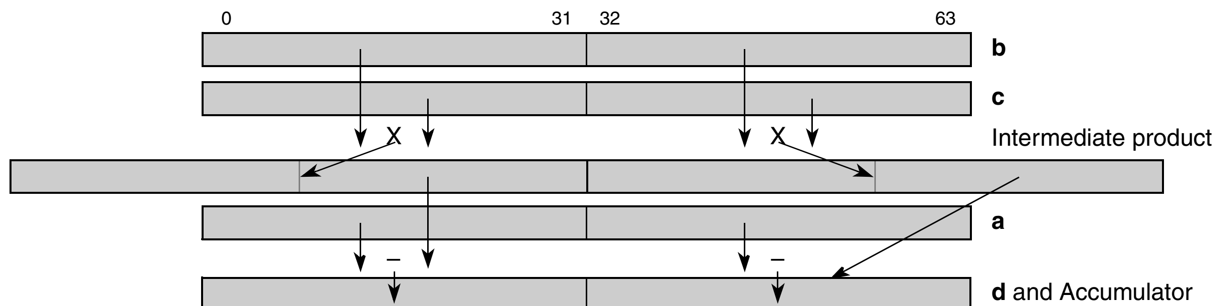


Figure 3-485. Vector Multiply Word Low Signed, Saturate, Integer and Accumulate Negative in Words 3 op (__ev_mwlssianw3)

d	a	b	c	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**d ←a evmwlssianw3 d,b,c**

__ev_mwlumi[a] __ev_mwlumi[a]

Vector Multiply Word Low Unsigned, Modulo, Integer

d = __ev_mwlumi (a,b) (A = 0)

d = __ev_mwlumia (a,b) (A = 1)

```
// 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.

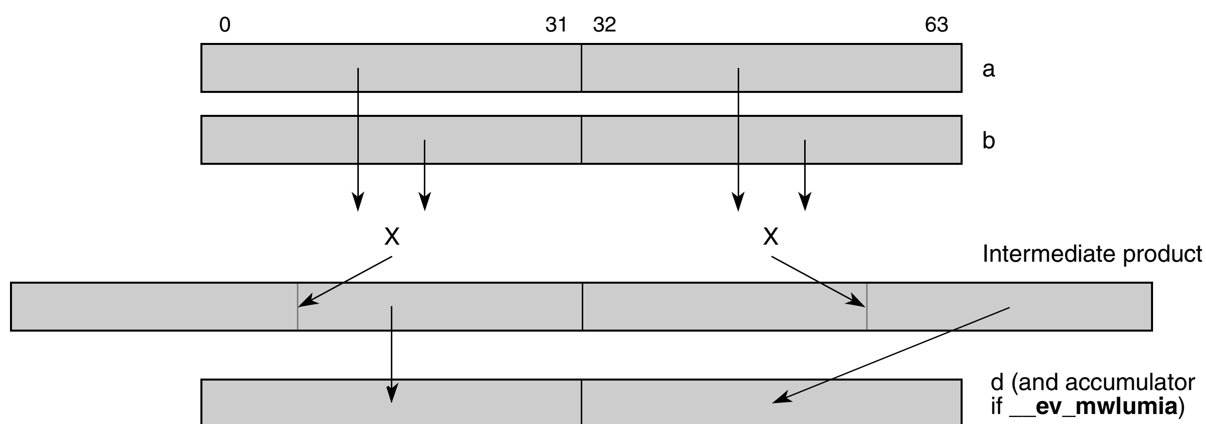


Figure 3-486. Vector Multiply Word Low Unsigned, Modulo, Integer (__ev_mwlumi[a])

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

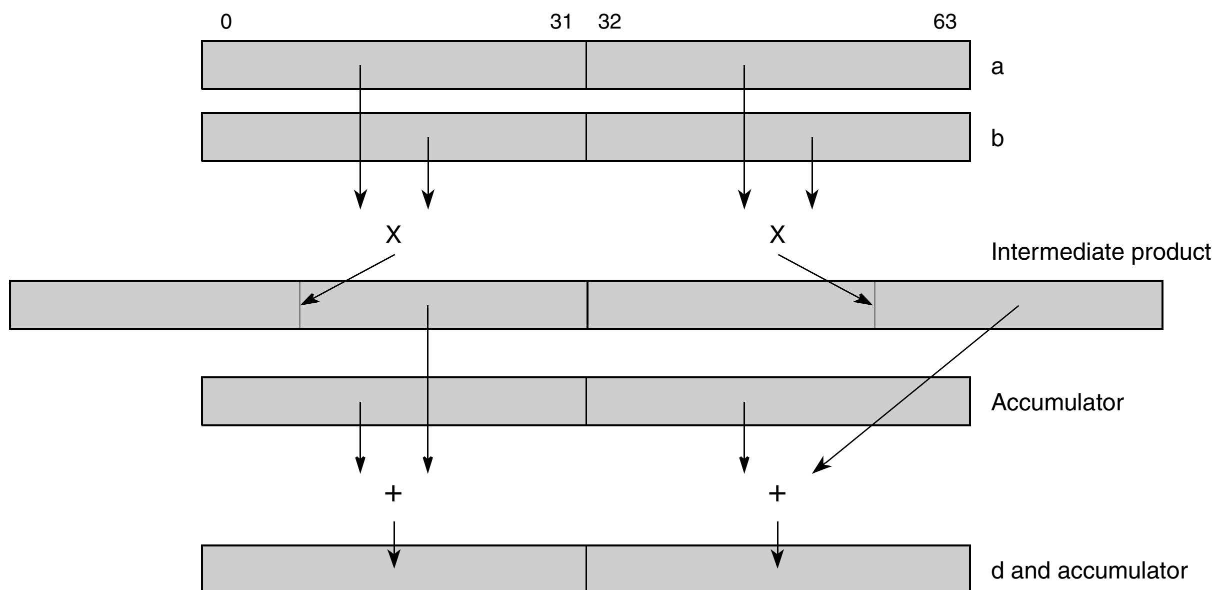


Figure 3-487. Vector Multiply Word Low Unsigned, Modulo, Integer and Accumulate in Words (__ev_mwlumiaaw)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evmwlumiaaw d,a,b**

__ev_mwlumiaaw3 __ev_mwlumiaaw3

Vector Multiply Word Low Unsigned, Modulo, Integer and Accumulate in Words 3 operand

d = __ev_mwlumiaaw3 (a,b,c)

```
// high
temp0:63 ←b0:31 ×ui c0:31
d0:31 ←a0:31 + temp32:63

// low
temp0:63 ←b32:63 ×ui c32:63
d32:63 ←a32:63 + temp32:63

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

For each word element in the accumulator, the corresponding word unsigned integer elements in parameters **b** and **c** are multiplied. The least significant 32 bits of each product is added to the contents of the corresponding parameter **a** word and the result is placed into parameter **d** and the accumulator.

Other registers altered: ACC

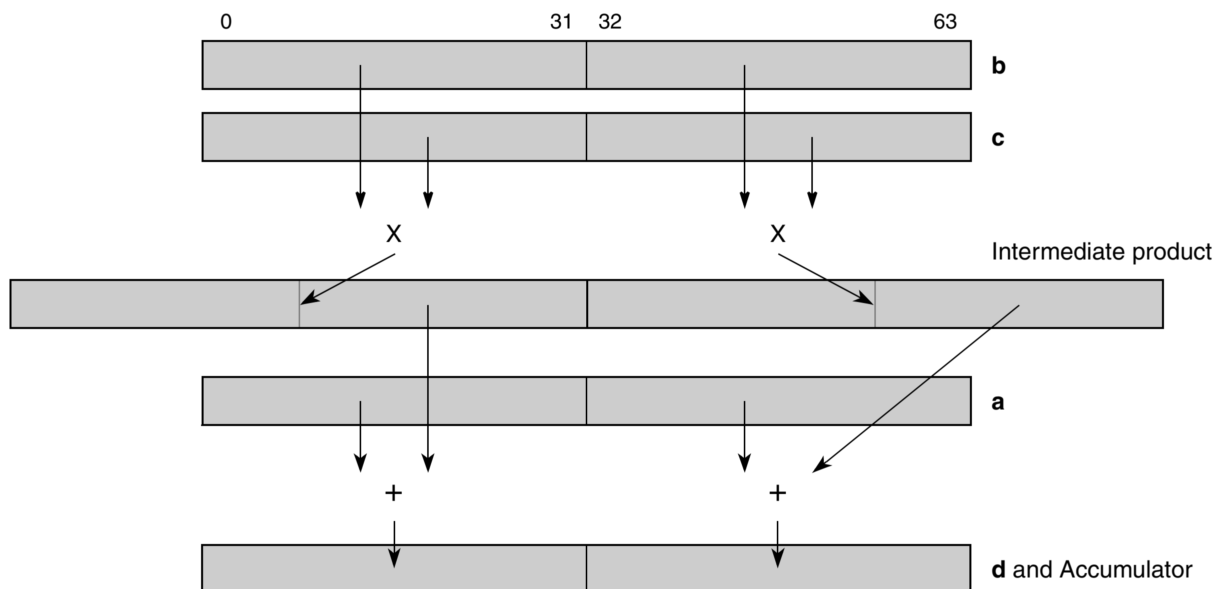


Figure 3-488. Vector Multiply Word Low Unsigned, Modulo, Integer and Accumulate in Words 3 op (__ev_mwlumiaaw3)

d	a	b	c	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**d ←a** **evmwlumiaaw3 d,b,c**

__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

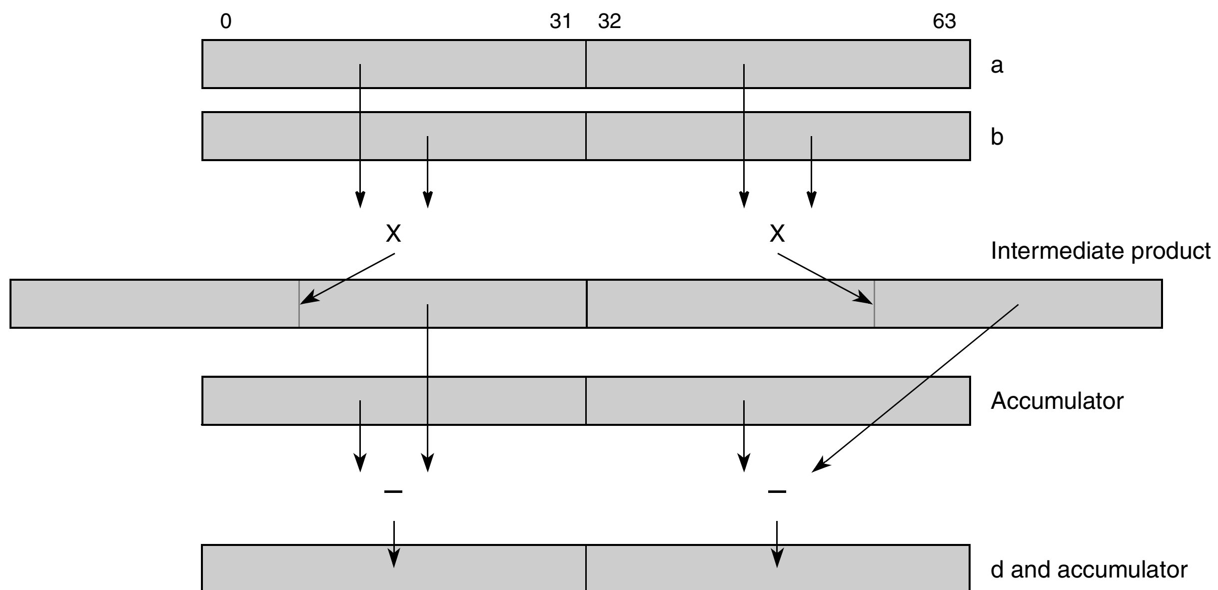


Figure 3-489. Vector Multiply Word Low Unsigned, Modulo, Integer and Accumulate Negative in Words (__ev_mwlumianw)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evmwlumianw d,a,b**

__ev_mwlumianw3 __ev_mwlumianw3

Vector Multiply Word Low Unsigned, Modulo, Integer and Accumulate Negative in Words 3 operand

d = __ev_mwlumianw3 (a,b,c)

```
// high
temp0:63 ←b0:31 ×ui c0:31
d0:31 ←a0:31 - temp32:63

// low
temp0:63 ←b32:63 ×ui c32:63
d32:63 ←a32:63 - temp32:63

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

For each word element in the accumulator, the corresponding word unsigned integer elements in parameters **b** and **c** are multiplied. The least significant 32 bits of each product is subtracted from the contents of the corresponding parameter **a** word and the result is placed into parameter **d** and the accumulator.

Other registers altered: ACC

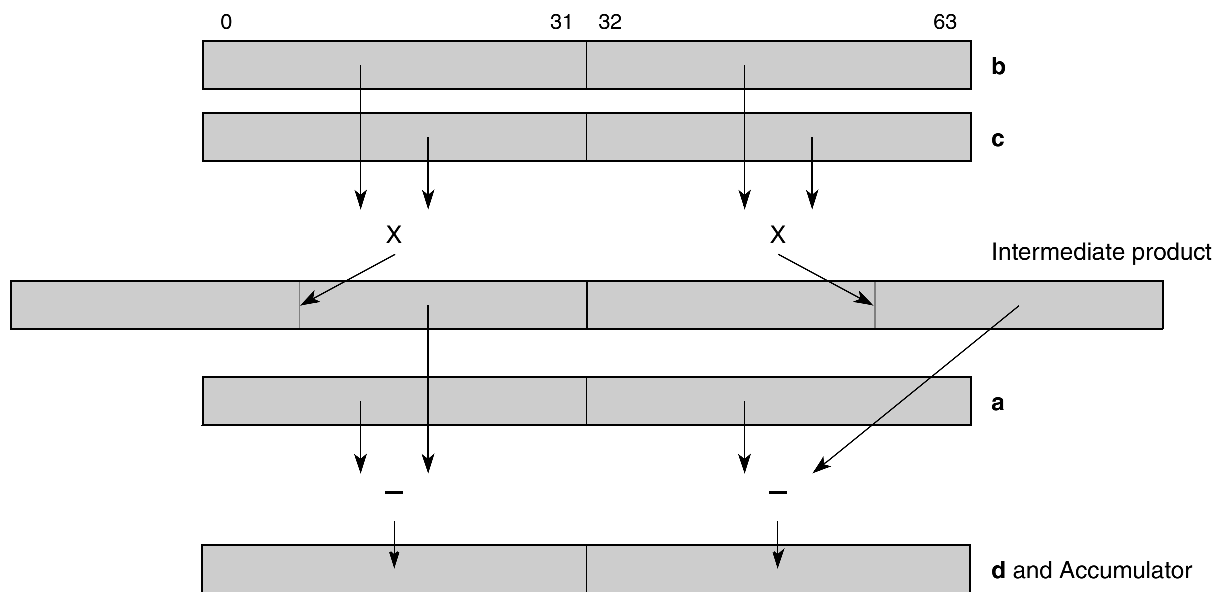


Figure 3-490. Vector Multiply Word Low Unsigned, Modulo, Integer and Accumulate Negative in Words 3 op (__ev_mwlumianw3)

d	a	b	c	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	d ←a **evmwlumianw3 d,b,c**

__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

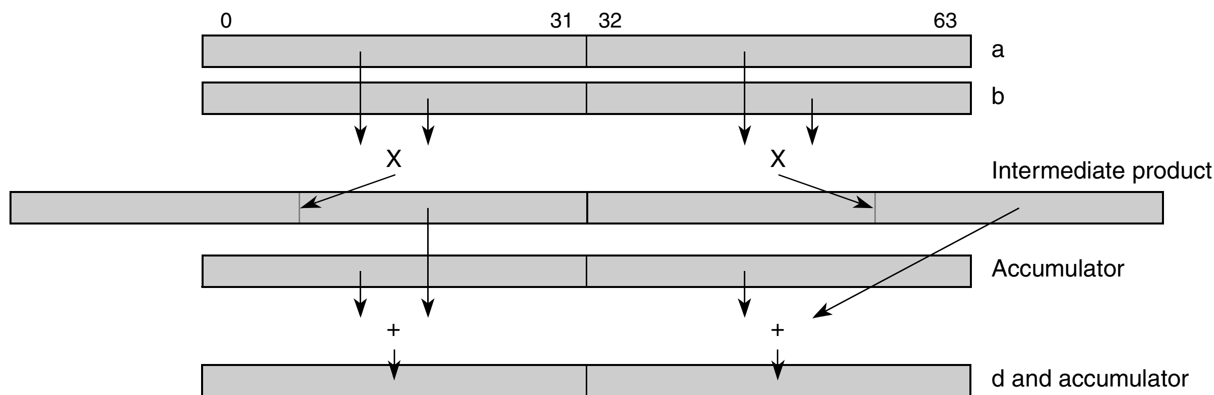


Figure 3-491. Vector Multiply Word Low Unsigned, Saturate, Integer and Accumulate in Words (__ev_mwlusiaaw)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evmwlusiaaw d,a,b**

__ev_mwlusiaaw3 __ev_mwlusiaaw3

Vector Multiply Word Low Unsigned, Saturate, Integer and Accumulate in Words 3 operand

d = __ev_mwlusiaaw3 (a,b,c)

```
// high
temp0:63 ←b0:31 ×ui c0:31
temp0:63 ←EXTZ64(a0:31) + EXTZ64(temp32:63)
ovh ←temp31
d0:31 ←SATURATE(ovh, 0, 0xFFFF_FFFF, 0xFFFF_FFFF, temp32:63)

//low
temp0:63 ←b32:63 ×ui c32:63
temp0:63 ←EXTZ64(a32:63) + EXTZ64(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 **b** and **c** are multiplied producing a 64-bit product. The least significant 32 bits of each product is then added to the corresponding word in parameter **a** saturating if overflow occurs, and the result is placed into 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

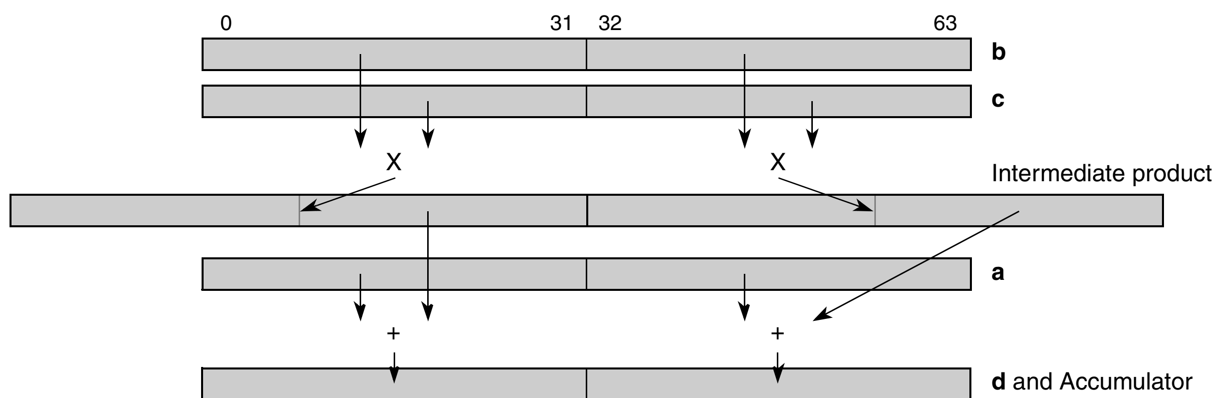


Figure 3-492. Vector Multiply Word Low Unsigned, Saturate, Integer and Accumulate in Words 3 op (__ev_mwlusiaaw3)

d	a	b	c	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	d ←a **evmwlusiaaw3 d,b,c**

__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

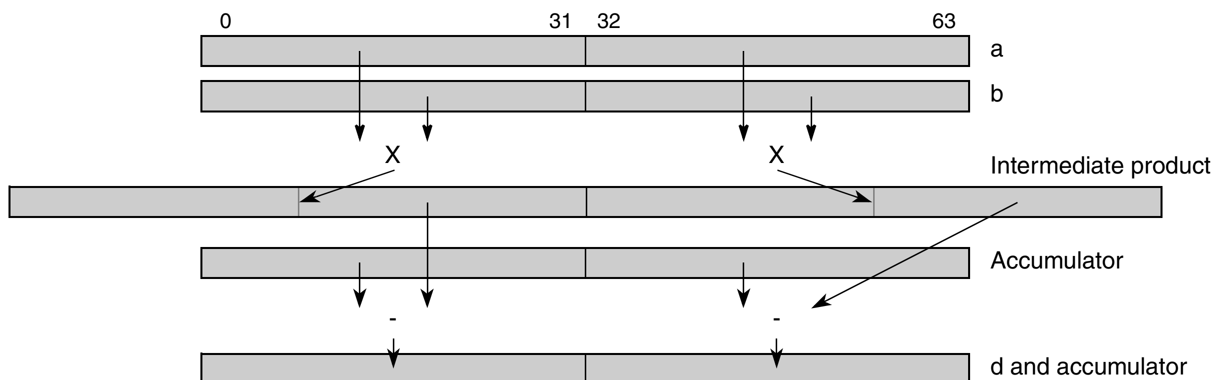


Figure 3-493. Vector Multiply Word Low Unsigned, Saturate, Integer and Accumulate Negative in Words (__ev_mwlusianw)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evmwlusianw d,a,b**

__ev_mwlusianw3 __ev_mwlusianw3

Vector Multiply Word Low Unsigned, Saturate, Integer and Accumulate Negative in Words 3 operand

d = __ev_mwlusianw3 (a,b,c)

```
// high
temp0:63 ←b0:31 ×ui c0:31
temp0:63 ←EXTZ64(a0:31) - EXTZ64(temp32:63)
ovh ←temp31
d0:31 ←SATURATE(ovh, 0, 0x0000_0000, 0x0000_0000, temp32:63)

//low
temp0:63 ←b32:63 ×ui c32:63
temp0:63 ←EXTZ64(a32:63) - EXTZ64(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 **b** and **c** are multiplied producing a 64-bit product. The least significant 32 bits of each product is then subtracted from the corresponding word in parameter **a** saturating if underflow occurs, and the result is placed into 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

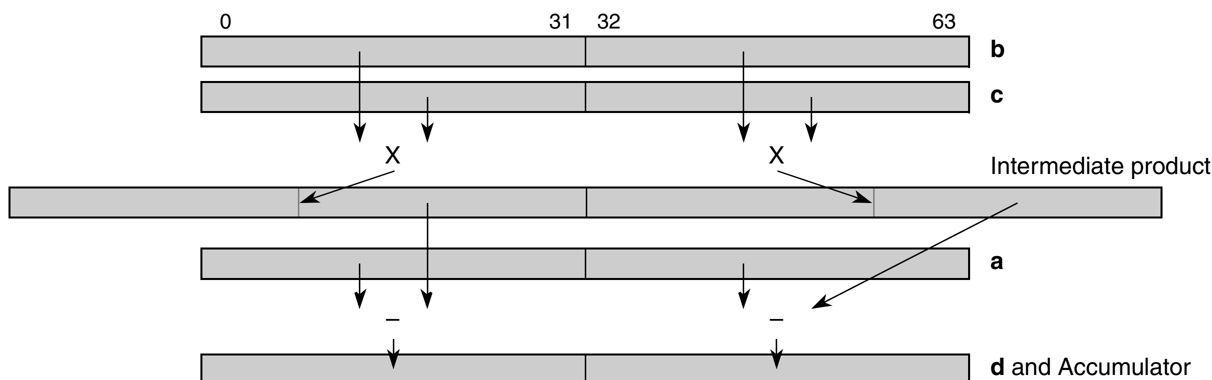


Figure 3-494. Vector Multiply Word Low Unsigned, Saturate, Integer and Accumulate Negative in Words 3 op (__ev_mwlusianw3)

d	a	b	c	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**d ←a** **evmwlusianw3 d,b,c**

__ev_mwohgsmf[a] __ev_mwohgsmf[a]

Vector Multiply Word Odd, High, Guarded, Signed, Modulo, Fractional (to Accumulator)

d = __ev_mwohgsmf (a,b) (A = 0)

d = __ev_mwohgsmfa (a,b) (A = 1)

```
temp0:63 ←a32:63 ×sf b32:63

if (a32:63 = 0x8000_0000) &(b32:63 = 0x8000_0000) then
    d0:63 ←0x0000_8000_0000_0000
else
    d0:63 ←EXTS64(temp0:47)

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

The odd word signed fractional elements in parameters **a** and **b** are multiplied. The high order 48 bits of the 64-bit fractional product are sign-extended to 64-bits to produce an intermediate product in 17.47 fractional format. If both inputs are -1.0, the intermediate product is represented as +1.0. The intermediate product is then 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 3-495. Vector Multiply Word Odd, High, Guarded, Signed, Modulo, Fractional (to Accumulator) (__ev_mwohgsmf[a])

A	d	a	b	Maps to
A = 0	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evmwohgsmf d,a,b**
A = 1	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evmwohgsmfa d,a,b**

__ev_mwohgsmfaa __ev_mwohgsmfaa

Vector Multiply Word Odd High Guarded Signed, Modulo, Fractional and Accumulate

d = __ev_mwohgsmfaa (a,b)

```
temp0:63 ←a32:63 ×sf b32:63

if (a32:63 = 0x8000_0000) &(b32:63 = 0x8000_0000) then
    d0:63 ←0x0000_8000_0000_0000 + ACC0:63
else
    d0:63 ←EXTS64(temp0:47) + ACC0:63

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

The odd word signed fractional elements in parameters **a** and **b** are multiplied. The high order 48 bits of the 64-bit product are sign-extended to 64-bits to produce an intermediate product in 17.47 fractional format. If both inputs are -1.0, the intermediate product is represented as +1.0. The intermediate product is then added to the contents of the accumulator and the result is placed into parameter **d** and the accumulator.

Other registers altered: ACC

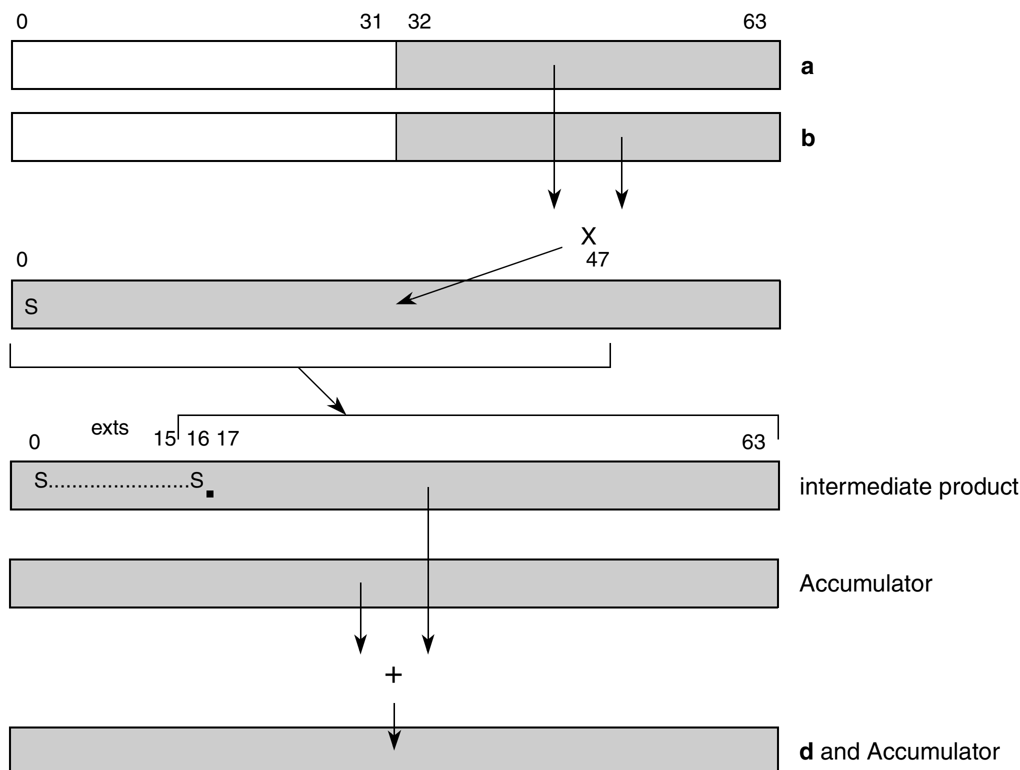


Figure 3-496. Vector Multiply Word Odd, High, Guarded, Signed, Modulo, Fractional and Accumulate (__ev_mwohgsmfaa)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evmwohgsmfaa d,a,b**

__ev_mwohgsmfan __ev_mwohgsmfan

Vector Multiply Word Odd, High, Guarded, Signed, Modulo, Fractional and Accumulate Negative

d = __ev_mwohgsmfan (a,b)

```
temp0:63 ←a32:63 ×sf b32:63
if (a32:63 = 0x8000_0000) &(b32:63 = 0x8000_0000) then
    d0:63 ←0x0000_8000_0000_0000 + ACC0:63
else
    d0:63 ←ACC0:63 - EXTS64(temp0:47)

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

The odd word signed fractional elements in parameters **a** and **b** are multiplied. The high order 48 bits of the 64-bit product are sign-extended to 64-bits to produce an intermediate product in 17.47 fractional format. If both inputs are -1.0, the intermediate product is represented as +1.0. The intermediate product is then subtracted from the contents of the accumulator and the result is placed into parameter **d** and the accumulator.

Other registers altered: ACC

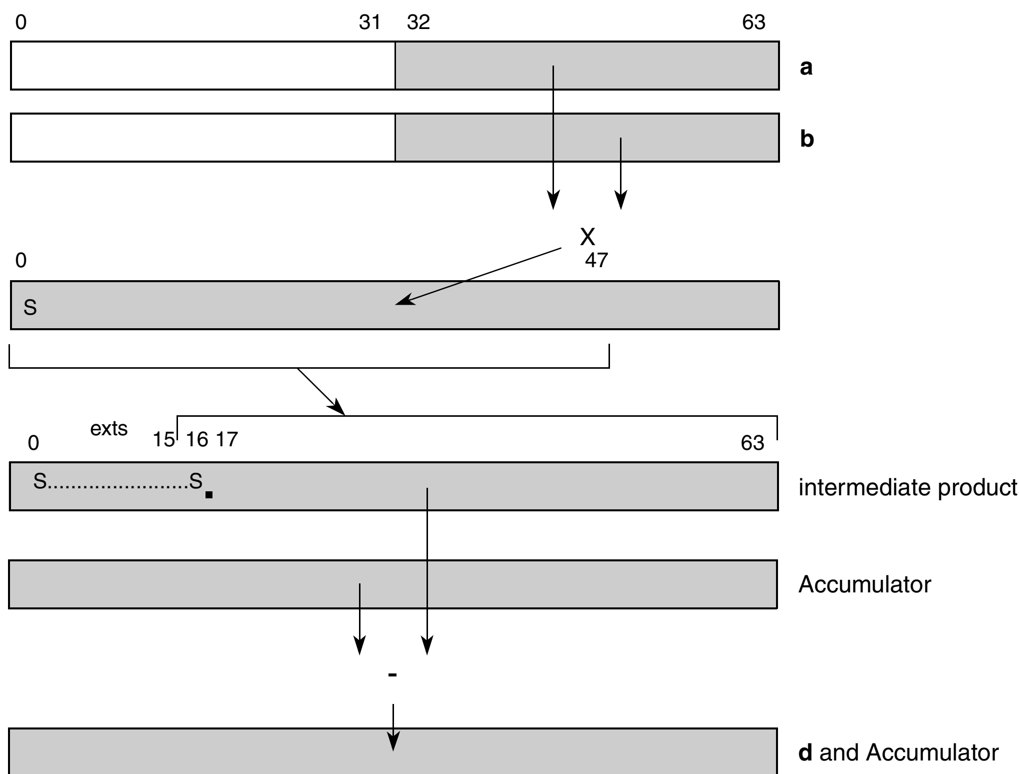


Figure 3-497. Vector Multiply Word Odd, High, Guarded, Signed, Modulo, Fractional and Accumulate Negative (__ev_mwohgsmfan)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evmwohgsmfan d,a,b**

__ev_mwohgsmfr[a] __ev_mwohgsmfr[a]

Vector Multiply Word Odd, High, Guarded, Signed, Modulo, Fractional, Round (to Accumulator)

d = __ev_mwohgsmfr (a,b) (A = 0)

d = __ev_mwohgsmfra (a,b) (A = 1)

```
temp0:63 ←a32:63 ×sf b32:63

if (a32:63 = 0x8000_0000) &(b32:63 = 0x8000_0000) then
    d0:63 ←0x0000_8000_0000_0000
else
    temp0:64 ←EXTS65(temp0:63)
    tempr0:64 ←ROUND(temp0:64,16)
    d0:63 ←EXTS64(tempr0:48)

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

The odd word signed fractional elements in parameters **a** and **b** are multiplied. The 64-bit fractional product is sign-extended to 65 bits, rounded to 49-bits using the current rounding mode in SPEFSCR, and the 49-bit value is sign-extended to 64-bits to produce a rounded intermediate product in 17.47 fractional format. If both inputs are -1.0, the intermediate product is represented as +1.0. The intermediate product is then 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)

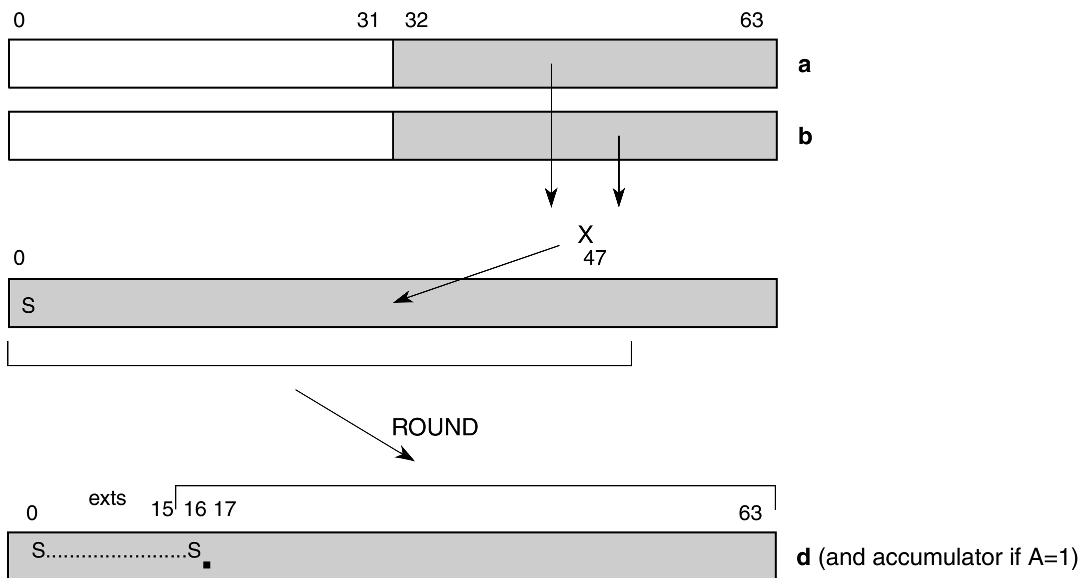


Figure 3-498. Vector Multiply Word Odd, High, Guarded, Signed, Modulo, Fractional, Round (to Accumulator) (__ev_mwohgsmfr[a])

A	d	a	b	Maps to
A = 0	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evmwohgsmfr d,a,b**
A = 1	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evmwohgsmfra d,a,b**

__ev_mwohgsmfraa __ev_mwohgsmfraa

Vector Multiply Word Odd High Guarded Signed, Modulo, Fractional, Round and Accumulate

d = __ev_mwohgsmfraa (a,b)

```
temp0:63 ←a32:63 ×sf b32:63
if (a32:63 = 0x8000_0000) &(b32:63 = 0x8000_0000) then
    d0:63 ←0x0000_8000_0000_0000 + ACC0:63
else
    temp0:64 ←EXTS65(temp0:63)
    tempr0:64 ←ROUND(temp0:64,16)
    d0:63 ←EXTS64(tempr0:48) + ACC0:63
// update accumulator
ACC0:63 ←d0:63
```

The odd word signed fractional elements in parameters **a** and **b** are multiplied. The 64-bit fractional product is sign-extended to 65 bits, rounded to 49-bits using the current rounding mode in SPEFSCR, and the 49-bit value is sign-extended to 64-bits to produce an intermediate product in 17.47 fractional format. If both inputs are -1.0, the intermediate product is represented as +1.0. The intermediate product is then added to the contents of the accumulator and the result is placed into parameter **d** and the accumulator.

Other registers altered: ACC

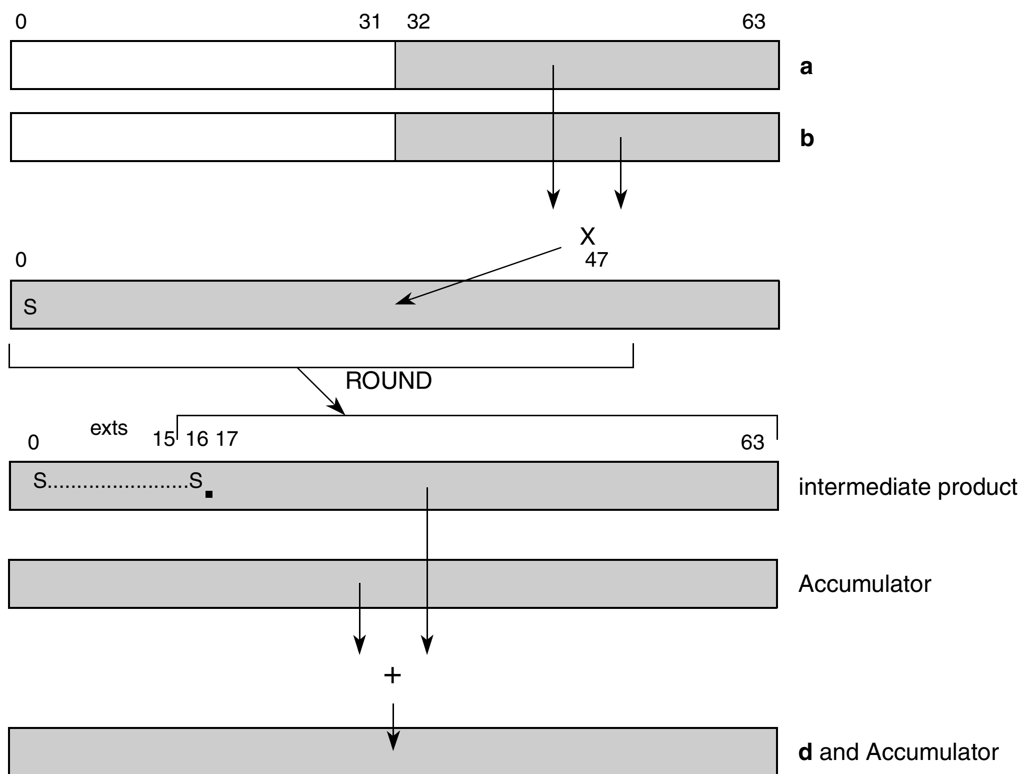


Figure 3-499. Vector Multiply Word Odd, High, Guarded, Signed, Modulo, Fractional, Round and Accumulate (__ev_mwohgsmfraa)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evmwohgsmfraa d,a,b**

__ev_mwohgsmfran __ev_mwohgsmfran

Vector Multiply Word Odd, High, Guarded, Signed, Modulo, Fractional, Round and Accumulate Negative

d = __ev_mwohgsmfran (a,b)

```
temp0:63 ←a32:63 ×sf b32:63
if (a32:63 = 0x8000_0000) &(b32:63 = 0x8000_0000) then
    d0:63 ←ACC0:63 - 0x0000_8000_0000_0000
else
    temp0:64 ←EXTS65(temp0:63)
    tempr0:64 ←ROUND(temp0:64,16)
    d0:63 ←ACC0:63 - EXTS64(tempr0:48)
ACC0:63 ←d0:63
```

The odd word signed fractional elements in parameters **a** and **b** are multiplied. The 64-bit fractional product is sign-extended to 65 bits, rounded to 49-bits using the current rounding mode in SPEFSCR, and the 49-bit value is sign-extended to 64-bits to produce an intermediate product in 17.47 fractional format. If both inputs are -1.0, the intermediate product is represented as +1.0. The intermediate product is then subtracted from the contents of the accumulator and the result is placed into parameter **d** and the accumulator.

Other registers altered: ACC

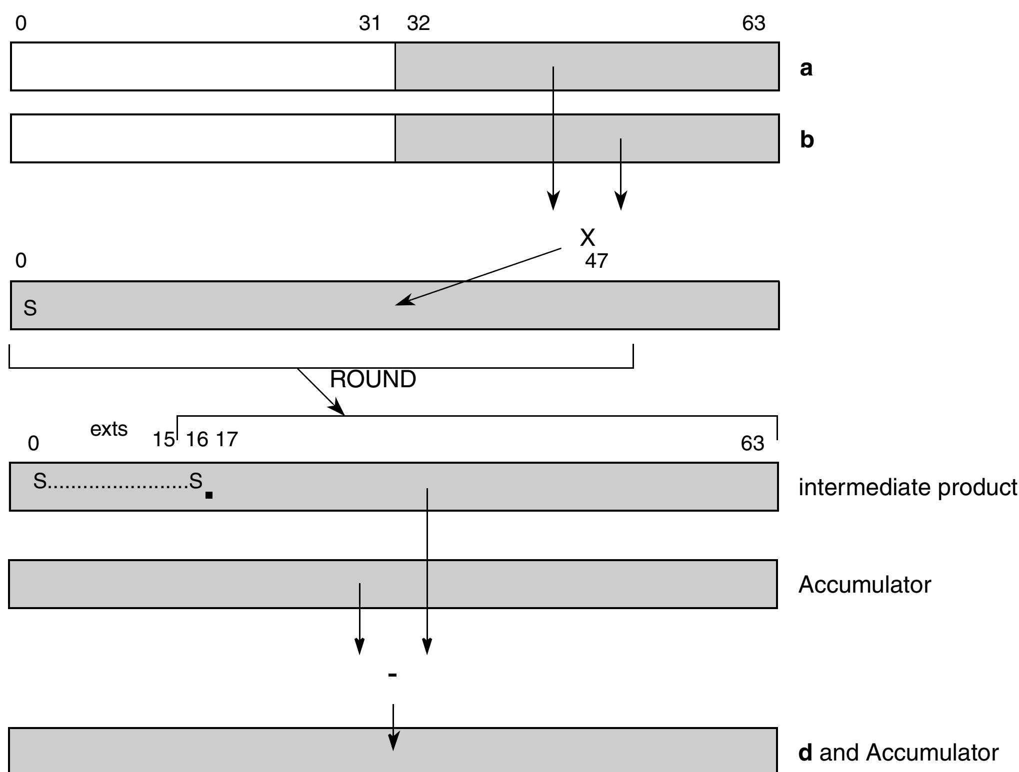


Figure 3-500. Vector Multiply Word Odd, High, Guarded, Signed, Modulo, Fractional, Round and Accumulate Negative (__ev_mwohgsmfran)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evmwohgsmfran d,a,b**

__ev_mwsmf[a] __ev_mwsmf[a]

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)

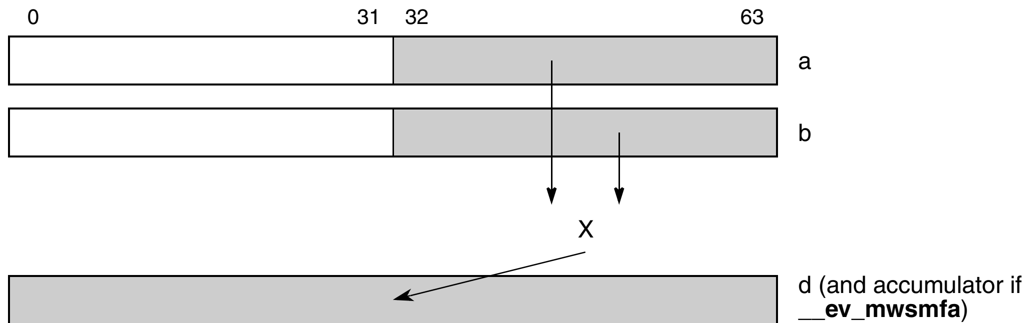


Figure 3-501. Vector Multiply Word Signed, Modulo, Fractional (to Accumulator) (__ev_mwsmf[a])

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)

$$temp_{0:63} \leftarrow a_{32:63} \times_{sf} b_{32:63}$$
$$d_{0:63} \leftarrow ACC_{0:63} + temp_{0:63}$$
// update accumulator
$$ACC_{0:63} \leftarrow d_{0:63}$$

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

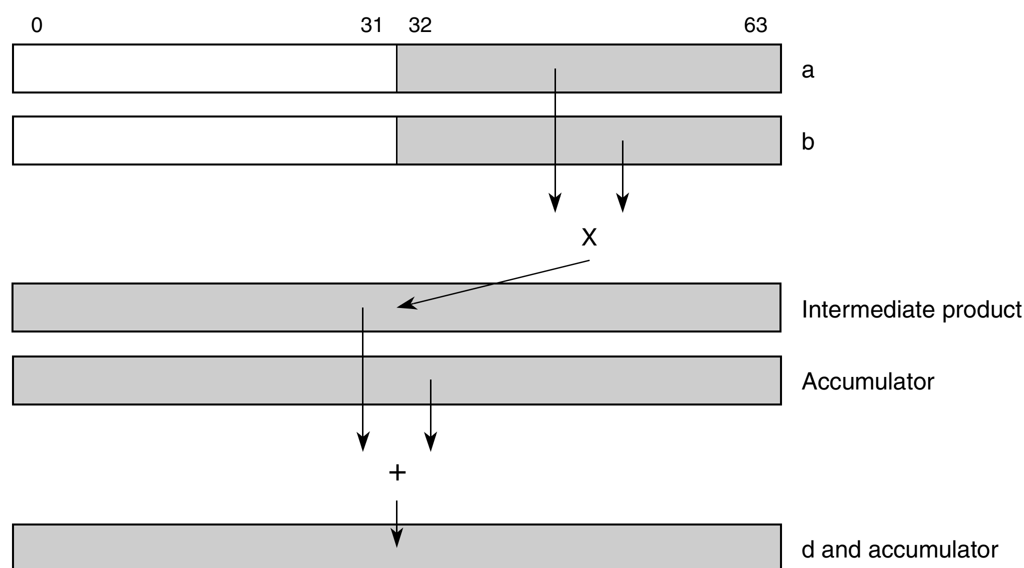


Figure 3-502. Vector Multiply Word Signed, Modulo, Fractional and Accumulate (__ev_mwsmfaa)

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

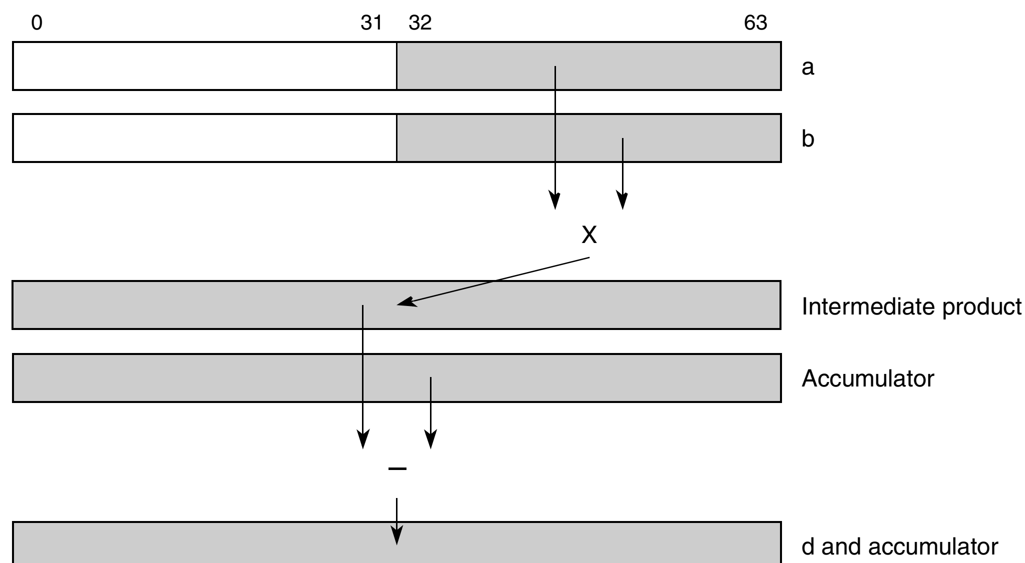


Figure 3-503. Vector Multiply Word Signed, Modulo, Fractional, and Accumulate Negative (__ev_mwsmfan)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evmwsmfan d,a,b**

__ev_mwsmi[a] __ev_mwsmi[a]

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)

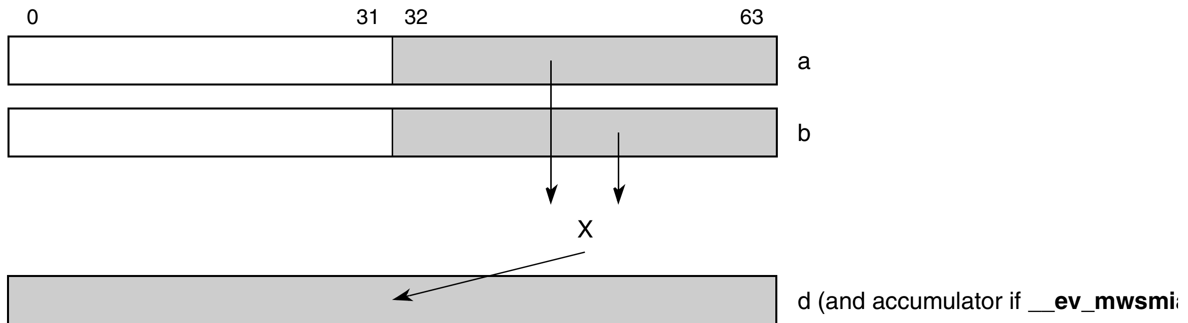


Figure 3-504. Vector Multiply Word Signed, Modulo, Integer (to Accumulator) (__ev_mwsmi[a])

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

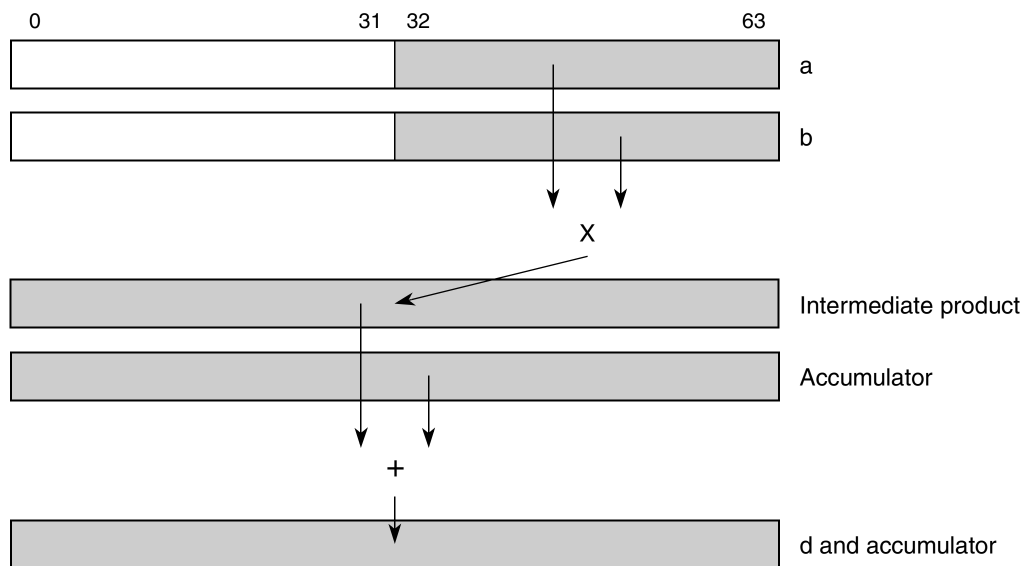


Figure 3-505. Vector Multiply Word Signed, Modulo, Integer and Accumulate (__ev_mwsmiaa)

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)

$temp_{0:63} \leftarrow a_{32:63} \times_{si} b_{32:63}$
$d_{0:63} \leftarrow ACC_{0:63} - temp_{0:63}$
// update accumulator
$ACC_{0:63} \leftarrow d_{0:63}$

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

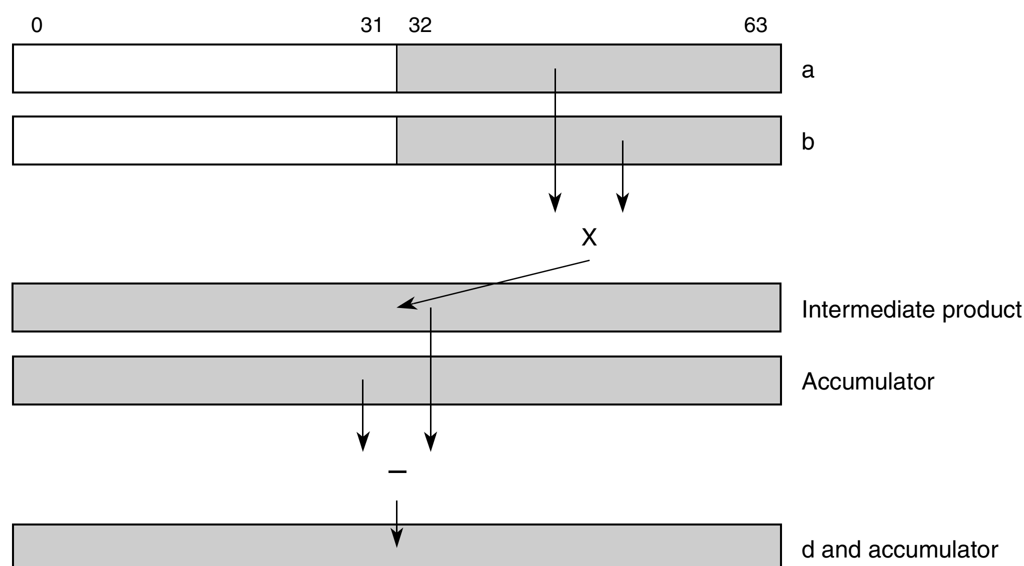


Figure 3-506. Vector Multiply Word Signed, Modulo, Integer and Accumulate Negative (__ev_mwsmian)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evmwsmian d,a,b**

__ev_mwssf[a] __ev_mwssf[a]

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 3-507. Vector Multiply Word Signed, Saturate, Fractional (to Accumulator) (__ev_mwssf[a])

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 = SATURATE(ov, temp0, 0x8000000000000000, 0x7FFFFFFFFFFFFFFF, temp1:64)
// update accumulator
ACC0:63 ←d0:63
// update SPEFSCR
SPEFSCROVH ←0 ; SPEFSCROV ←mov | ov ; SPEFSCRSOV ←SPEFSCRSOV | mov | ov ;
```

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 (0x7FFFFFFFFFFFFFFF). The 64-bit product is then added to the accumulator to form an intermediate sum. If the intermediate sum has overflowed, the appropriate saturation value (0x7FFFFFFFFFFFFFFF if positive overflow or 0x8000000000000000 if negative overflow) is placed into the accumulator word and the corresponding parameter **d** word. Otherwise, the low 64 bits of the intermediate sum are placed into the accumulator word and the corresponding parameter **d** word. The overflow and summary overflow bits are recorded to indicate occurrence of saturation either the multiply or the addition.

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

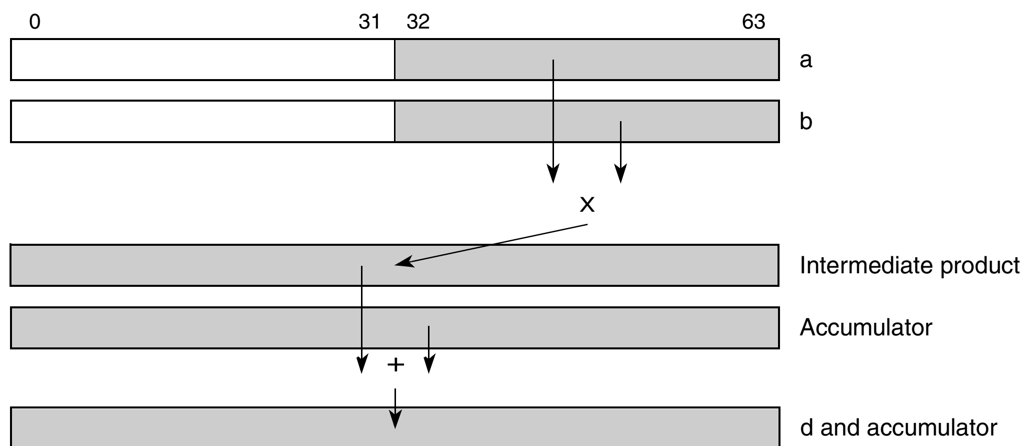


Figure 3-508. Vector Multiply Word Signed, Saturate, Fractional and Accumulate (__ev_mwssfaa)

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 = SATURATE(ov, temp0, 0x8000000000000000, 0x7FFFFFFFFFFFFFFF, temp1:64)
// update accumulator
ACC0:63 ←d0:63
// update SPEFSCR
SPEFSCROVH ←0 ; SPEFSCROV ←mov | ov ; SPEFSCRSOV ←SPEFSCRSOV | mov | ov
```

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 (0x7FFFFFFFFFFFFFFF). The 64-bit product is then subtracted from the accumulator to form an intermediate difference. If the intermediate difference has overflowed, the appropriate saturation value (0x7FFFFFFFFFFFFFFF if positive overflow or 0x8000000000000000 if negative overflow) is placed into the accumulator word and the corresponding parameter **d** word. Otherwise, the low 64 bits of the intermediate difference are placed into the accumulator word and the corresponding parameter **d** word. The overflow and summary overflow bits are recorded to indicate occurrence of saturation either the multiply or the subtraction.

Other registers altered: SPEFSCR ACC

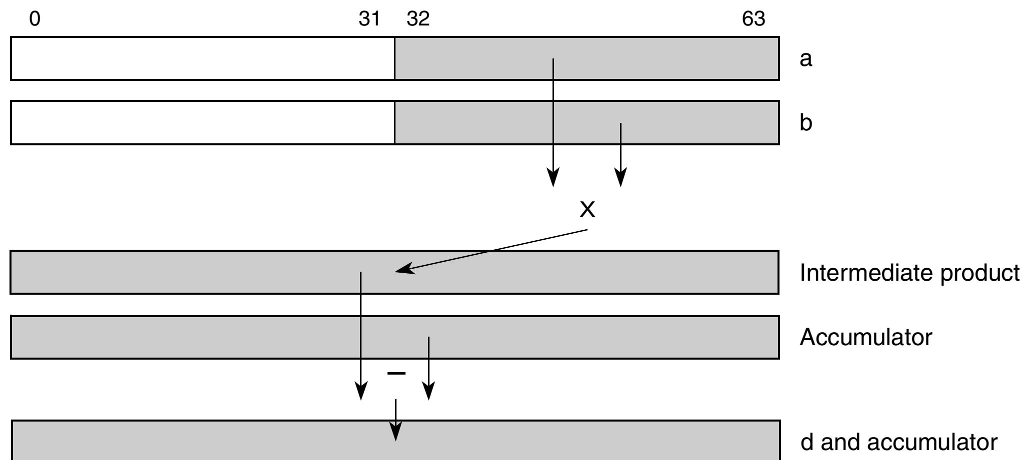


Figure 3-509. Vector Multiply Word Signed, Saturate, Fractional and Accumulate Negative (__ev_mwssfan)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evmwssfan d,a,b**

__ev_mwssiaa __ev_mwssiaa

Vector Multiply Word Signed, Saturate, Integer and Accumulate

d = __ev_mwssiaa (a,b)

```
temp1[0:63] ←a[32:63] ×si b[32:63]
temp2[0:64] ←EXTS(ACC[0:63]) + EXTS(temp1[0:63])
ov ←(temp2[0] ⊕ temp2[1])
d[0:63] = SATURATE(ov, temp2[0], 0x8000000000000000, 0x7FFFFFFFFFFFFFFF, temp2[1:64])
// update accumulator
ACC[0:63] ←d[0:63]
// update SPEFSCR
SPEFSCR[OVH] ←0
SPEFSCR[OV] ←ov
SPEFSCR[SOV] ←SPEFSCR[SOV] | ov
```

The low word signed integer elements in parameters **a** and **b** are multiplied producing a 64-bit product. The 64-bit product is then added to the accumulator to form an intermediate sum. If the intermediate sum has overflowed, the appropriate saturation value (`0x7FFFFFFFFFFFFFFF` if positive overflow or `0x8000000000000000` if negative overflow) is placed into the accumulator word and the corresponding parameter **d** word. Otherwise, the low 64 bits of the intermediate sum are placed into the accumulator word and the corresponding parameter **d** word. The overflow and summary overflow bits are recorded to indicate occurrence of saturation in the addition.

Other registers altered: SPEFSCR ACC

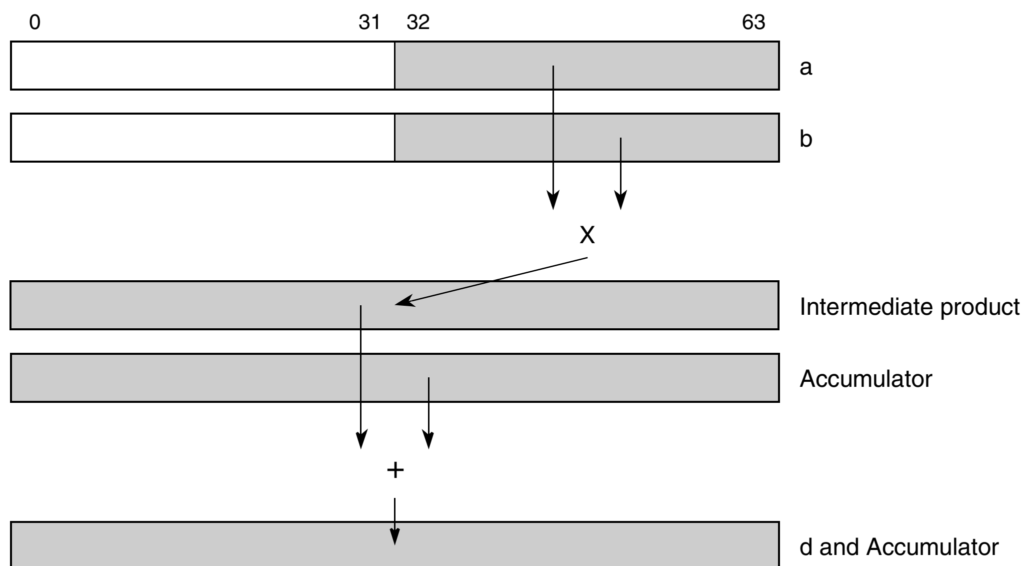


Figure 3-510. Vector Multiply Word Signed, Saturate, Integer and Accumulate (__ev_mwssiaa)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evmwssiaa d,a,b**

__ev_mwssian __ev_mwssian

Vector Multiply Word Signed, Saturate, Integer and Accumulate Negative

d = __ev_mwssian (a,b)

```
temp1_{0:63} ←a_{32:63} ×_{si} b_{32:63}
temp2_{0:64} ←EXTS(ACC_{0:63}) - EXTS(temp1_{0:63})
ov ←(temp2_0 ⊕ temp2_1)
d_{0:63} = SATURATE(ov, temp2_0, 0x8000000000000000, 0x7FFFFFFFFFFFFFFF, temp2_{1:64})
// update accumulator
ACC_{0:63} ←d_{0:63}
// update SPEFSCR
SPEFSCR_{OVH} ←0
SPEFSCR_{OV} ←ov
SPEFSCR_{SOV} ←SPEFSCR_{SOV} | ov
```

The low word signed integer elements in parameters **a** and **b** are multiplied producing a 64-bit product. The 64-bit product is then subtracted from the accumulator to form an intermediate difference. If the intermediate difference has overflowed, the appropriate saturation value (0x7FFFFFFFFFFFFFFF if positive overflow or 0x8000000000000000 if negative overflow) is placed into the accumulator word and the corresponding parameter **d** word. Otherwise, the low 64 bits of the intermediate difference are placed into the accumulator word and the corresponding parameter **d** word. The overflow and summary overflow bits are recorded to indicate occurrence of saturation in the addition.

Other registers altered: SPEFSCR ACC

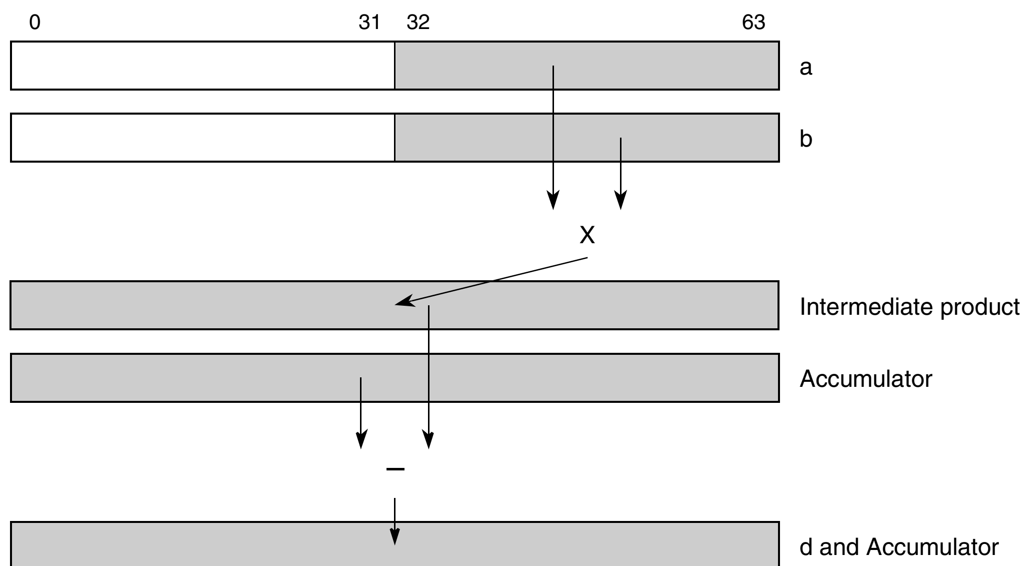


Figure 3-511. Vector Multiply Word Signed, Saturate, Integer and Accumulate Negative (__ev_mwssian)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evmwssian d,a,b**

__ev_mwssiw __ev_mwssiw

Vector Multiply Word Signed, Saturate, Integer Word

d = __ev_mwssiw (a,b)

```
// high
temp0_{0:63} ← a_{0:31} ×_{si} b_{0:31}
if (temp0_{0:63} > 0x0000_0000_7FFF_FFFF) | (temp0_{0:63} < 0xFFFF_FFFF_FFFF_8000) then
    movh ←1
    temp0_{32:63} ←SATURATE(movh, temp0_0, 0x8000_0000, 0x7FFF_FFFF, temp0_{32:63})
else
    movh ←0

// low
temp1_{0:63} ← a_{0:31} ×_{si} b_{0:31}
if (temp1_{0:63} > 0x0000_0000_7FFF_FFFF) | (temp1_{0:63} < 0xFFFF_FFFF_FFFF_8000) then
    movl ←1
    temp1_{32:63} ←SATURATE(movl, temp1_0, 0x8000_0000, 0x7FFF_FFFF, temp1_{32:63})
else
    movl ←0

d_{0:31} ←temp0_{32:63} ; d_{32:63} ←temp1_{32:63}
// update SPEFSCR
SPEFSCR_{OVH} ←movh; SPEFSCR_{OV} ←movl
SPEFSCR_{SOVH} ←SPEFSCR_{SOVH} | movh; SPEFSCR_{SOV} ←SPEFSCR_{SOV} | movl
```

For each word element in the destination, corresponding signed word integer elements in parameters **a** and **b** are multiplied producing a 64-bit intermediate product. The least significant 32 bits of each product is then placed in the corresponding word in parameter **d**, saturating if overflow occurs.

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

Other registers altered: SPEFSCR ACC

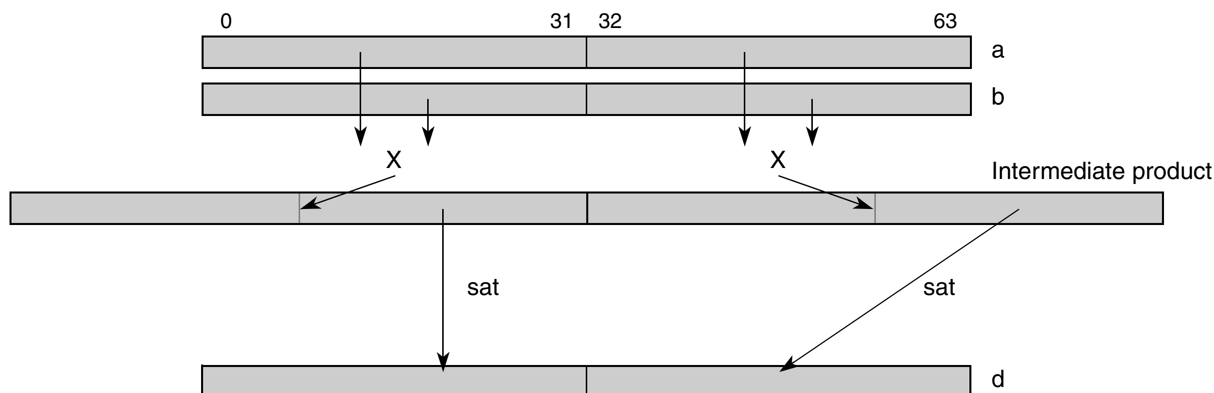


Figure 3-512. Vector Multiply Word Signed, Saturate, Integer Word (__ev_mwssiw)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evmwssiw d,a,b**

__ev_mwumi[a] __ev_mwumi[a]

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)

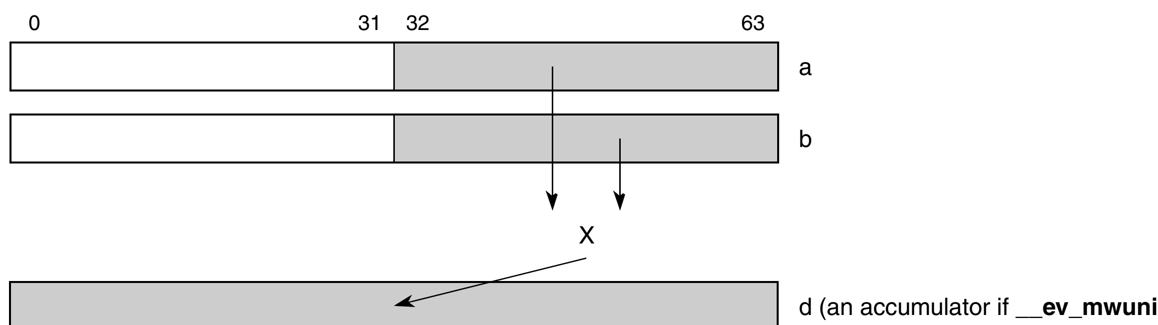


Figure 3-513. Vector Multiply Word Unsigned, Modulo, Integer (to Accumulator) (__ev_mwumi[a])

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

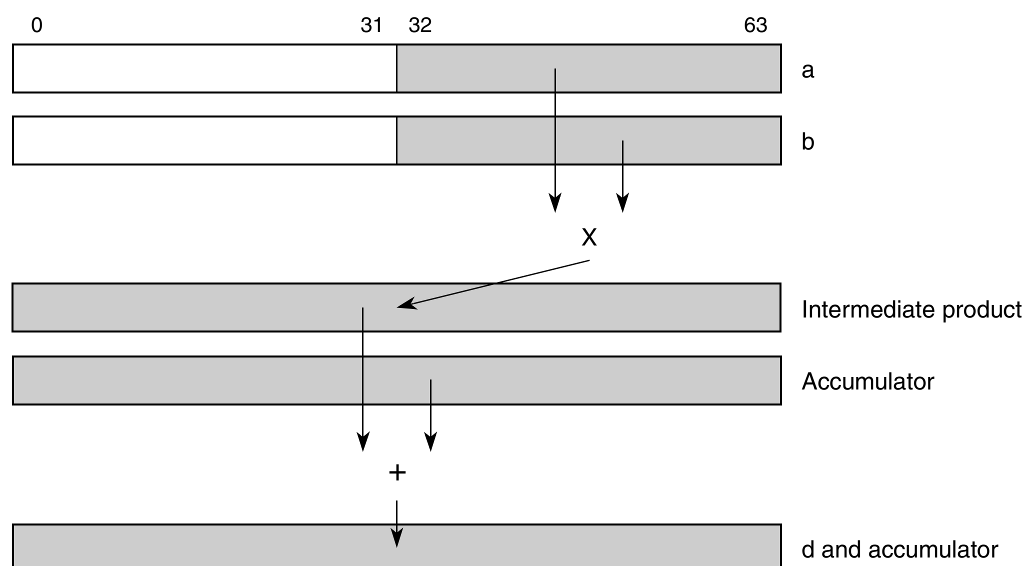


Figure 3-514. Vector Multiply Word Unsigned, Modulo, Integer and Accumulate (__ev_mwumiaa)

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 3-515. 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_mwusiaa __ev_mwusiaa

Vector Multiply Word Unsigned, Saturate, Integer and Accumulate

d = __ev_mwusiaa (a,b)

```
temp1[0:63] ←a[32:63] ×ui b[32:63]
temp2[0:64] ←EXTZ(ACC[0:63]) + EXTZ(temp1[0:63])
ov ←temp2[0]
d[0:63] = SATURATE(ov, temp2[0], 0xFFFFFFFFFFFFFFFF, 0xFFFFFFFFFFFFFFFF, temp2[1:64])
// update accumulator
ACC[0:63] ←d[0:63]
// update SPEFSCR
SPEFSCR[OVH] ←0
SPEFSCR[OV] ←ov
SPEFSCR[SOV] ←SPEFSCR[SOV] | ov
```

The low word unsigned integer elements in parameters **a** and **b** are multiplied producing a 64-bit product. The 64-bit product is then added to the accumulator to form an intermediate sum. If the intermediate sum has overflowed, the saturation value 0xFFFFFFFFFFFFFFFF is placed into the accumulator word and the corresponding parameter **d** word. Otherwise, the low 64 bits of the intermediate sum are placed into the accumulator word and the corresponding parameter **d** word. The overflow and summary overflow bits are recorded to indicate occurrence of saturation in the addition.

Other registers altered: SPEFSCR ACC

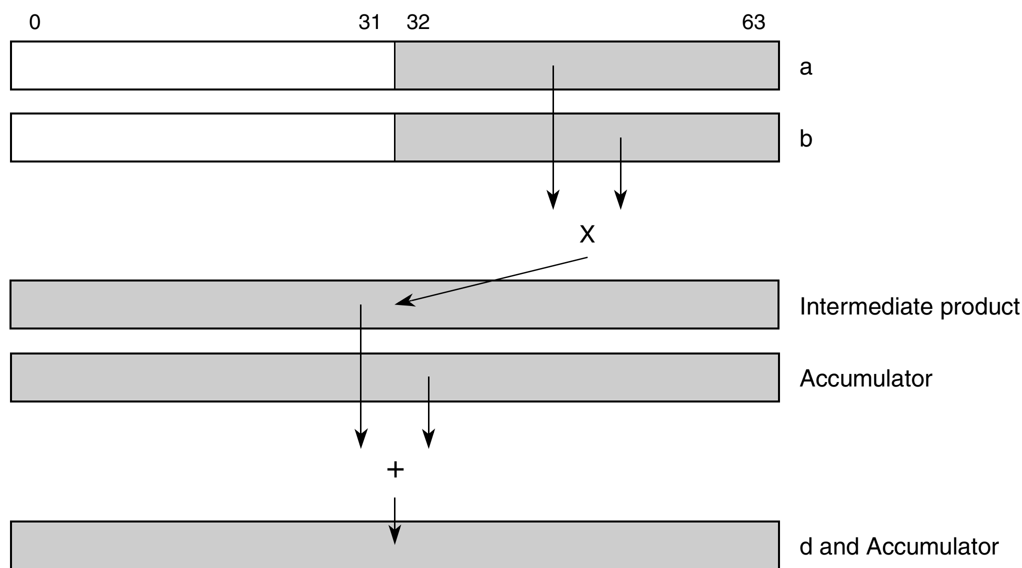


Figure 3-516. Vector Multiply Word Unsigned, Saturate, Integer and Accumulate (__ev_mwusiaa)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evmwusiaa d,a,b**

__ev_mwusian __ev_mwusian

Vector Multiply Word Unsigned, Saturate, Integer and Accumulate Negative

d = __ev_mwusian (a,b)

```
temp1_{0:63} ←a_{32:63} ×_{ui} b_{32:63}
temp2_{0:64} ←EXTZ(ACC_{0:63}) - EXTZ(temp1_{0:63})
ov ←temp2_0
d_{0:63} = SATURATE(ov, 0, 0x0000000000000000, 0x0000000000000000, temp2_{1:64})
// update accumulator
ACC_{0:63} ←d_{0:63}
// update SPEFSCR
SPEFSCR_{OVH} ←0
SPEFSCR_{OV} ←ov
SPEFSCR_{SOV} ←SPEFSCR_{SOV} | ov
```

The low word signed integer elements in parameters **a** and **b** are multiplied producing a 64-bit product. The 64-bit product is then subtracted from the accumulator to form an intermediate difference. If the intermediate difference has underflowed, the saturation value `0x0000000000000000` is placed into the accumulator word and the corresponding parameter **d** word. Otherwise, the low 64 bits of the intermediate difference are placed into the accumulator word and the corresponding parameter **d** word. The overflow and summary overflow bits are recorded to indicate occurrence of saturation in the subtraction.

Other registers altered: SPEFSCR ACC

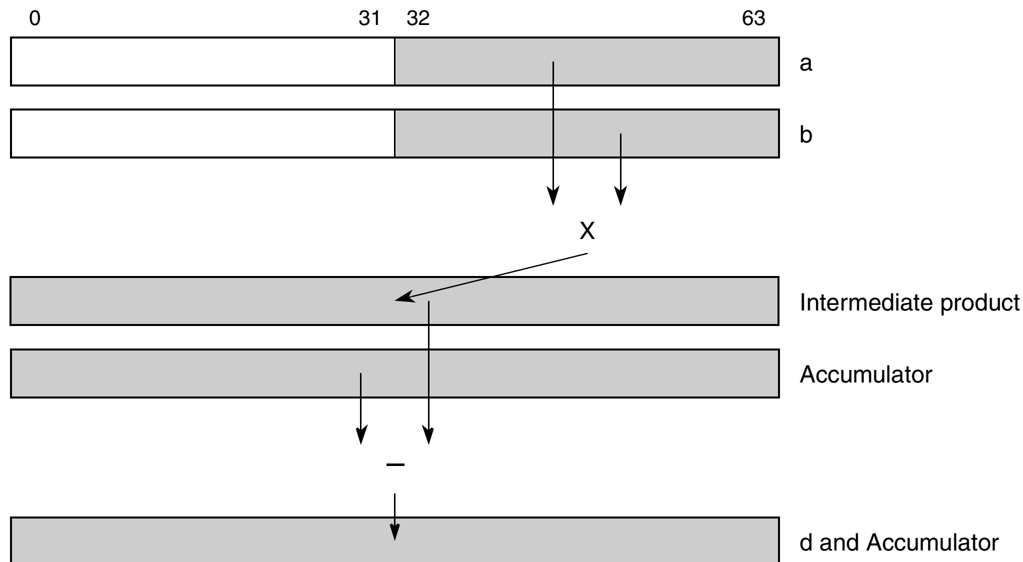


Figure 3-517. Vector Multiply Word Unsigned, Saturate, Integer and Accumulate Negative (__ev_mwusian)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evmwusian d,a,b**

__ev_mwusiw __ev_mwusiw

Vector Multiply Word Unsigned, Saturate, Integer Word

d = __ev_mwusiw (a,b)

```
// high
temp0[0:63] ← a[0:31] ×ui b[0:31]
if (temp0[0:63] > 0x0000_0000_FFFF_FFFF) then
    movh ←1
    temp0[32:63] ←SATURATE(movh, 0, 0xFFFF_FFFF, 0xFFFF_FFFF, temp0[32:63])
else
    movh ←0

// low
temp1[0:63] ← a[0:31] ×ui b[0:31]
if (temp1[0:63] > 0x0000_0000_FFFF_FFFF) then
    movl ←1
    temp1[32:63] ←SATURATE(movl, 0, 0xFFFF_FFFF, 0xFFFF_FFFF, temp1[32:63])
else
    movl ←0

d[0:31] ←temp0[32:63] ; d[32:63] ←temp1[32:63]
// update SPEFSCR
SPEFSCR[OVH] ←movh; SPEFSCR[OV] ←movl
SPEFSCR[SOVH] ←SPEFSCR[SOVH] | movh; SPEFSCR[SOV] ←SPEFSCR[SOV] | movl
```

For each word element in the destination, corresponding word unsigned word integer elements in parameters **a** and **b** are multiplied producing a 64-bit intermediate product. The least significant 32 bits of each product is then placed in the corresponding word in parameter **d**, saturating if overflow occurs.

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

Other registers altered: SPEFSCR ACC

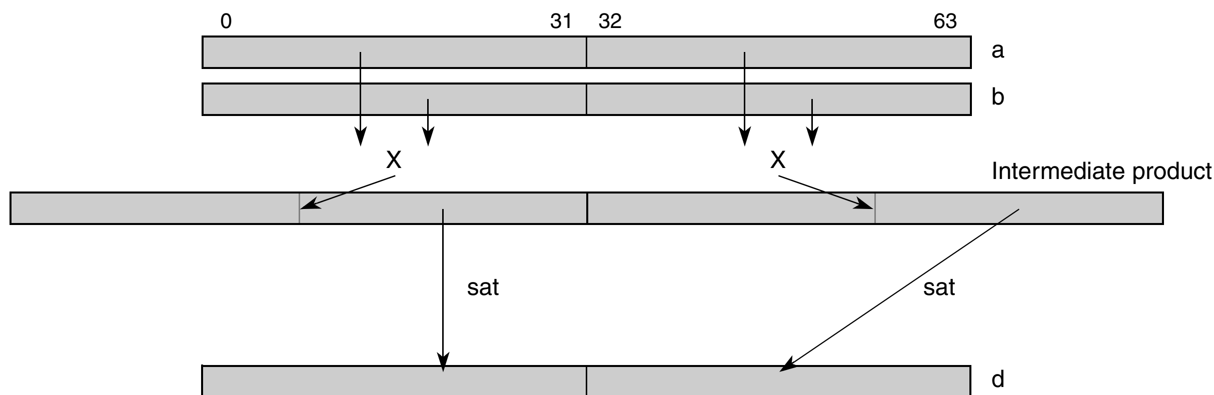


Figure 3-518. Vector Multiply Word Unsigned, Saturate, Integer Word (__ev_mwusiw)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evmwusiw 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**.

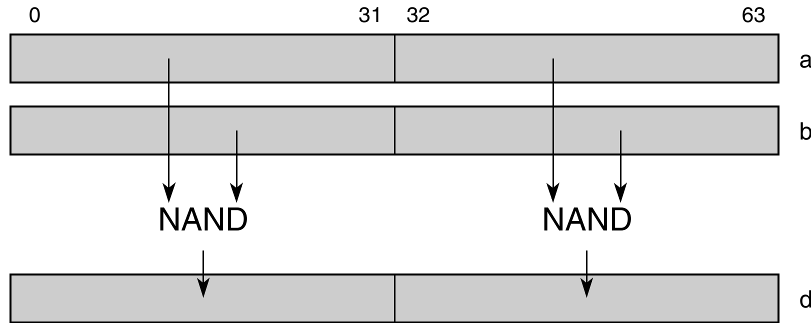


Figure 3-519. Vector NAND (__ev_nand)

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)

```
d0:31 ←NEG(a0:31)
d32:63 ←NEG(a32:63)
```

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.

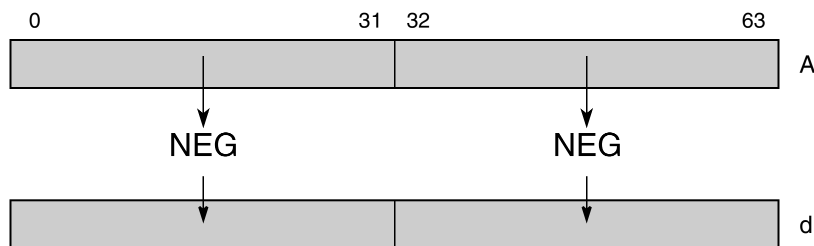


Figure 3-520. Vector Negate (__ev_neg)

d	a	Maps to
__ev64_opaque__	__ev64_opaque__	**evneg d,a**

__ev_negb __ev_negb

Vector Negate Byte

d = __ev_negb (a)

```
d0:7 ←NEG(a0:7)
d8:15 ←NEG(a8:15)
d16:23 ←NEG(a16:23)
d24:31 ←NEG(a24:31)
d32:39 ←NEG(a32:39)
d40:47 ←NEG(a40:47)
d48:55 ←NEG(a48:55)
d56:63 ←NEG(a56:63)
```

The negated value of each byte element of parameter **a** is placed in the corresponding element of parameter **d**. An initial value of 0x80 (most negative number) returns 0x80. No overflow is detected.

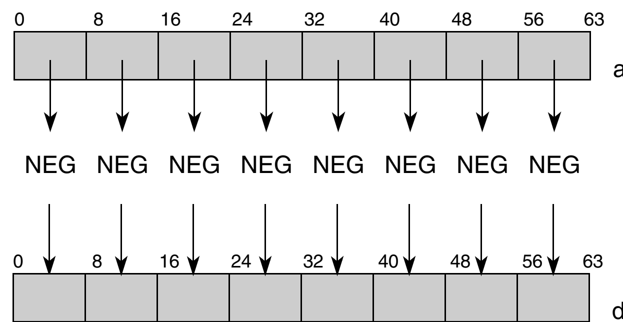


Figure 3-521. Vector Negate Byte (evnegb)

d	a	Maps to
__ev64_opaque__	__ev64_opaque__	**evnegb d,a**

__ev_negbo __ev_negbo

Vector Negate Byte Odd

d = __ev_negbo (a)

$d_{0:7} \leftarrow a_{0:7}$
$d_{8:15} \leftarrow NEG(a_{8:15})$
$d_{16:23} \leftarrow a_{16:23}$
$d_{24:31} \leftarrow NEG(a_{24:31})$
$d_{32:39} \leftarrow a_{32:39}$
$d_{40:47} \leftarrow NEG(a_{40:47})$
$d_{48:55} \leftarrow a_{48:55}$
$d_{56:63} \leftarrow NEG(a_{56:63})$

The even byte elements of parameter **a** are placed into the corresponding bytes of parameter **d** unchanged. The negated values of the odd byte elements of parameter **a** are placed into the corresponding bytes of parameter **d**. An initial value of 0x80 (most negative number) returns 0x80. No overflow is detected.

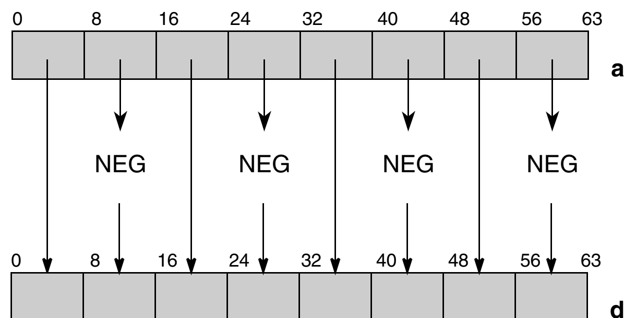


Figure 3-522. Vector Negate Byte Odd (__ev_negbo)

d	a	Maps to
__ev64_opaque__	__ev64_opaque__	**evnegbo d,a**

__ev_negbos __ev_negbos

Vector Negate Byte Odd and Saturate

d = __ev_negbos (a)

```
ovh ←0
d0:7 ←a0:7

if (a8:15 = 0x80) then
    d8:15 ←0x7F
    ovh ←1
else d8:15 ←NEG(a8:15)
endif

d16:23 ←a16:23

if (a24:31 = 0x80) then
    d24:31 ←0x7F
    ovh ←1
else d24:31 ←NEG(a24:31)
endif

ovl ←0
d32:39 ←a32:39

if (a40:47 = 0x80) then
    d40:47 ←0x7F
    ovl ←1
else d40:47 ←NEG(a40:47)
endif
d48:55 ←a48:55

if (a56:63 = 0x80) then
    d56:63 ←0x7F
    ovl ←1
else d56:63 ←NEG(a56:63)
endif

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

The even byte elements of parameter **a** are placed into the corresponding bytes of parameter **d** unchanged. The negated values of the odd byte elements of **a** are placed into the corresponding bytes of parameter **d**. The negated value of 0x80 (most negative number) returns 0x7F. Any overflow is reported in the SPEFSCR overflow and summary overflow bits.

Other registers altered: SPEFSCR

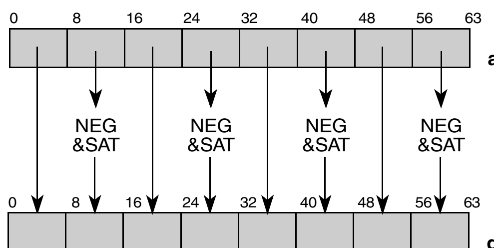


Figure 3-523. Vector Negate Byte Odd and Saturate (__ev_negbos)

d	a	Maps to
__ev64_opaque__	__ev64_opaque__	**evnegbos d,a**

__ev_negbs __ev_negbs

Vector Negate Byte and Saturate

d = __ev_negbs (a)

```
if (a0:7 = 0x80) then
    d0:7 ←0x7F
    ovh ←1
else
    d0:7 ←NEG(a0:7)
    ovh ←0
if (a8:15 = 0x80) then
    d8:15 ←0x7F
    ovh ←1
else d8:15 ←NEG(a8:15)
if (a16:23 = 0x80) then
    d16:23 ←0x7F
    ovh ←1
else d16:23 ←NEG(a16:23)
if (a24:31 = 0x80) then
    d24:31 ←0x7F
    ovh ←1
else d24:31 ←NEG(a24:31)
if (a32:39 = 0x80) then
    d32:39 ←0x7F
    ovl←1
else
    d32:39 ←NEG(a32:39)
    ovl ←0
if (a40:47 = 0x80) then
    d40:47 ←0x7F
    ovl ←1
else d40:47 ←NEG(a40:47)
if (a48:55 = 0x80) then
    d48:55 ←0x7F
    ovl ←1
else d48:55 ←NEG(a48:55)
if (a56:63 = 0x80) then
    d56:63 ←0x7F
    ovl ←1
else d56:63 ←NEG(a56:63)

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

The negated value of each signed byte element of parameter **a** is placed in parameter **d**. The negated value of 0x80 (most negative number) returns 0x7F. Any overflow is reported in the SPEFSCR overflow and summary overflow bits.

Other registers altered: SPEFSCR

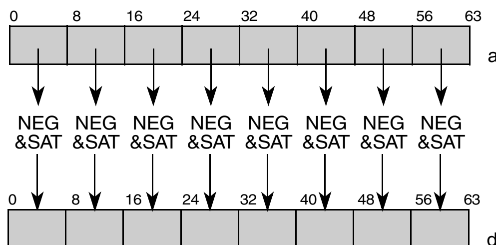


Figure 3-524. Vector Negate Byte and Saturate (__ev_negbs)

d	a	Maps to
__ev64_opaque__	__ev64_opaque__	**evnegbs d,a**

__ev_negd __ev_negd

Vector Negate Doubleword

d = __ev_negd (a)

$d_{0:63} \leftarrow NEG(a_{0:63})$

The negated value of the doubleword in parameter **a** is placed into parameter **d**. The negative of 0x8000_0000_0000_0000 (most negative number) returns 0x8000_0000_0000_0000. No overflow is detected.

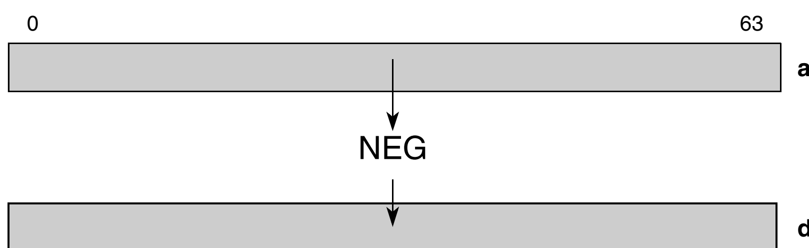


Figure 3-525. Vector Negate Doubleword (__ev_negd)

d	a	Maps to
__ev64_opaque__	__ev64_opaque__	**evnegd d,a**

__ev_negds __ev_negds

Vector Negate Doubleword and Saturate

d = __ev_negds (a)

```
if (a0:63 = 0x8000_0000_0000_0000) then
    d0:63 ←0x7FFF_FFFF_FFFF_FFFF
    ov ←1
else
    d0:63 ←NEG(a0:63)
    ov ←0
endif

SPEFSCROVH ←0
SPEFSCROV ←ov
SPEFSCRSOV ←SPEFSCRSOV | ov
```

The negated value of the doubleword in parameter **a** is placed into parameter **d**. The negative of 0x8000_0000_0000_0000 (most negative number) returns 0x7FFF_FFFF_FFFF_FFFF. Any overflow is reported in the SPEFSCR overflow and summary overflow bits.

Other registers altered: SPEFSCR

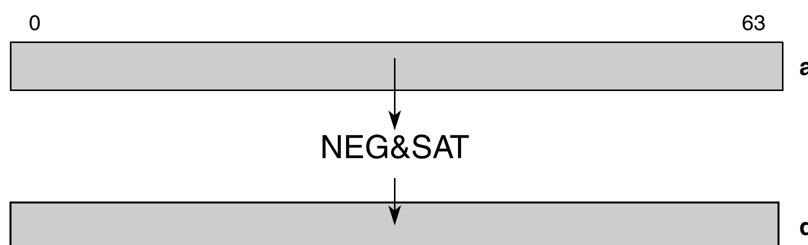


Figure 3-526. Vector Negate Doubleword and Saturate (__ev_negds)

d	a	Maps to
__ev64_opaque__	__ev64_opaque__	**evnegds d,a**

__ev_negh __ev_negh

Vector Negate Half Word

d = __ev_negh (a)

```
d0:15 ←NEG(a0:15)
d16:31 ←NEG(a16:31)
d32:47 ←NEG(a32:47)
d48:63 ←NEG(a48:63)
```

The negative of each element of parameter **a** is placed in parameter **d**. The negative of 0x8000 (most negative number) returns 0x8000. No overflow is detected.

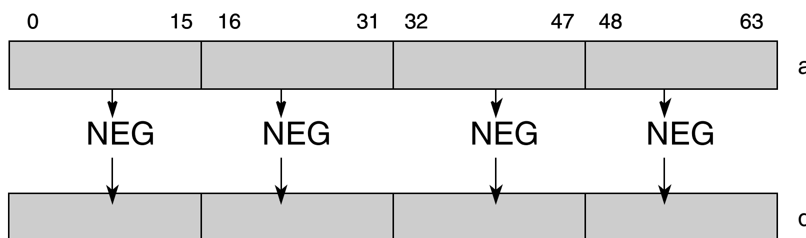


Figure 3-527. Vector Negate Half Word (__ev_negh)

d	a	Maps to
__ev64_opaque__	__ev64_opaque__	**evnegh d,a**

__ev_negho __ev_negho

Vector Negate Halfwords Odd

d = __ev_negho (a)

```
d0:15 ←a0:15
d16:31 ←NEG(a16:31)
d32:47 ←a32:47
d48:63 ←NEG(a48:63)
```

The even halfword elements of parameter **a** are placed into the corresponding halfwords of parameter **d** unchanged. The negated values of the odd halfword elements of parameter **a** are placed into the corresponding halfwords of parameter **d**. The negative of 0x8000 (most negative number) returns 0x8000. No overflow is detected.

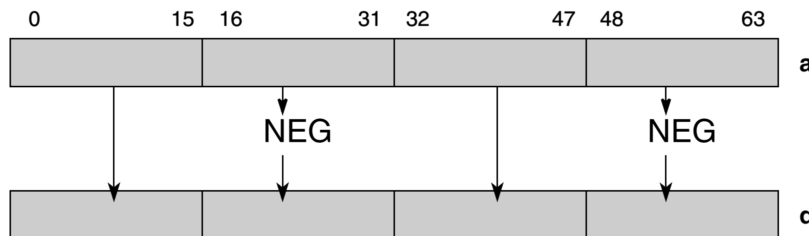


Figure 3-528. Vector Negate Halfwords Odd (__ev_negho)

d	a	Maps to
__ev64_opaque__	__ev64_opaque__	**evnegho d,a**

__ev_neghos __ev_neghos

Vector Negate Halfwords Odd and Saturate

d = __ev_neghos (a)

```
d0:15 ←a0:15

if (a16:31 = 0x8000) then
    d16:31 ←0x7FFF
    ovh ←1
else
    d16:31 ←NEG(a16:31)
    ovh ←0
endif

d32:47 ←a32:47

if (a48:63 = 0x8000) then
    d48:63 ←0x7FFF
    ovl ←1
else
    d48:63 ←NEG(a48:63)
    ovl ←0
endif

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

The even halfword elements of parameter **a** are placed into the corresponding halfwords of parameter **d** unchanged. The negated values of the odd halfword elements of parameter **a** are placed into the corresponding halfwords of parameter **d**. The negative of 0x8000 (most negative number) returns 0x7FFF. Any overflow is reported in the SPEFSCR overflow and summary overflow bits.

Other registers altered: SPEFSCR

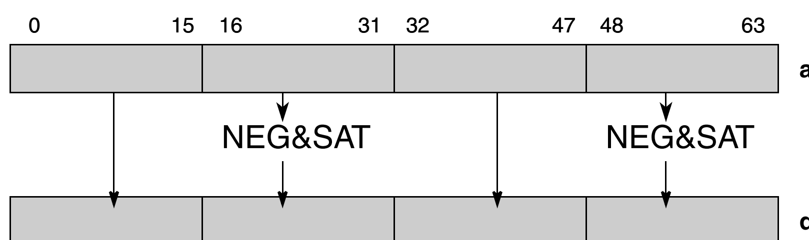


Figure 3-529. Vector Negate Halfwords Odd and Saturate (__ev_neghos)

d	a	Maps to
__ev64_opaque__	__ev64_opaque__	**evneghos d,a**

__ev_neghs

__ev_neghs

Vector Negate Half Word and Saturate

d = __ev_neghs (a)

```
if (a0:15 = 0x8000) then
    d0:15 ←0x7FFF
    ovh ←1
else
    d0:15 ←NEG(a0:15)
    ovh ←0
endif
if (a16:31 = 0x8000) then
    d16:31 ←0x7FFF
    ovh ←1
else d16:31 ←NEG(a16:31)
if (a32:47 = 0x8000) then else
    d32:47 ←0x7FFF
    ovl←1
else
    d32:47 ←NEG(a32:47)
    ovl ←0
endif
if (a48:63 = 0x8000) then
    d48:63 ←0x7FFF
    ovl ←1
else d48:63 ←NEG(a48:63)

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

The negative of each half word element of parameter **a** is placed in parameter **d**. The negative of 0x8000 (most negative number) returns 0x7FFF. Any overflow is reported in the SPEFSCR overflow and summary overflow bits.

Other registers altered: SPEFSCR

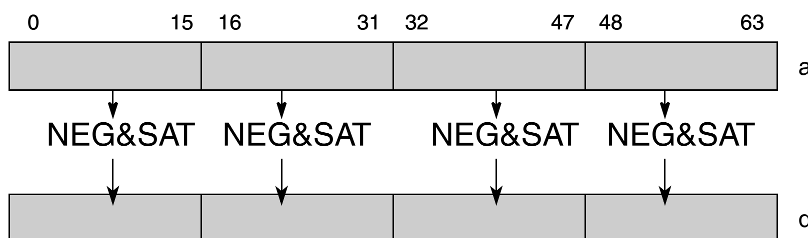


Figure 3-530. Vector Negate Half Word and Saturate (__ev_neghs)

d	a	Maps to
__ev64_opaque__	__ev64_opaque__	**evneghs d,a**

__ev_negs __ev_negs

Vector Negate (Word) and Saturate

d = __ev_negs (a)

```
if (a0:31 = 0x8000_0000) then
    d0:31 ←0x7FFF_FFFF
    ovh ←1
else
    d0:31 ←NEG(a0:31)
    ovh ←0
endif

if (a32:63 = 0x8000_0000) then
    d32:63 ←0x7FFF_FFFF
    ovl ←1
else
    d32:63 ←NEG(a32:63)
    ovl ←0
endif

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

The negative of each word element of parameter **a** is placed in parameter **d**. The negative of 0x8000_0000 (most negative number) returns 0x7FFF_FFFF. Any overflow is reported in the SPEFSCR overflow and summary overflow bits.

Other registers altered: SPEFSCR

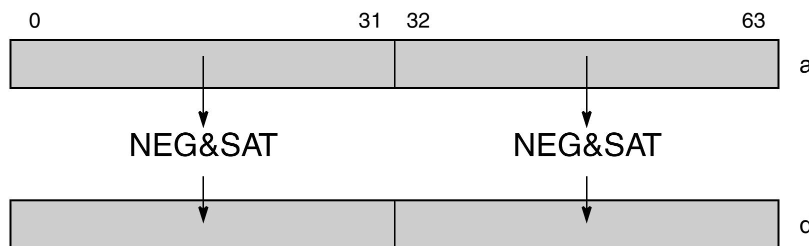


Figure 3-531. Vector Negate (Word) and Saturate (__ev_negs)

d	a	Maps to
__ev64_opaque__	__ev64_opaque__	**evnegs d,a**

__ev_negwo __ev_negwo

Vector Negate Word Odd

d = __ev_negwo (a)

```
d0:31 ←a0:31
d32:63 ←NEG(a32:63)
```

The value of the even word element of parameter **a** is placed into the even word of parameter **d** unchanged, and the negated value of the odd word element of parameter **a** is placed into the odd word of parameter **d**. The negative of 0x8000_0000 (most negative number) returns 0x8000_0000. No overflow is detected.

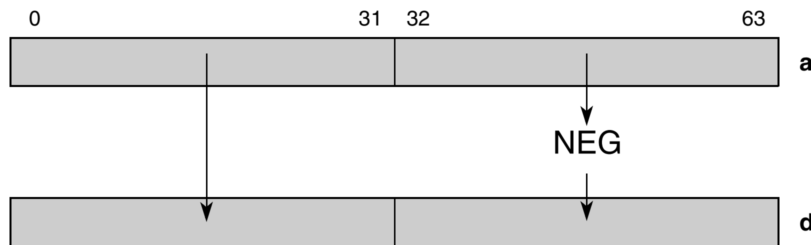


Figure 3-532. Vector Negate Word Odd (__ev_negwo)

d	a	Maps to
__ev64_opaque__	__ev64_opaque__	**evnegwo d,a**

__ev_negwos __ev_negwos

Vector Negate Word Odd and Saturate

d = __ev_negwos (a)

```
d0:31 ←a0:31

if (a32:63 = 0x8000_0000) then
    d32:63 ←0x7FFF_FFFF
    ovl ←1
else
    d32:63 ←NEG(a32:63)
    ovl ←0
endif

SPEFSCROVH ←0
SPEFSCROV ←ovl
SPEFSCRSOV ←SPEFSCRSOV | ovl
```

The value of the even word element of parameter **a** is placed into the even word of parameter **d** unchanged, and the negated value of the odd word element of parameter **a** is placed into the odd word of parameter **d**. The negative of 0x8000_0000 (most negative number) returns 0x7FFF_FFFF. Any overflow is reported in the SPEFSCR overflow and summary overflow bits.

Other registers altered: SPEFSCR

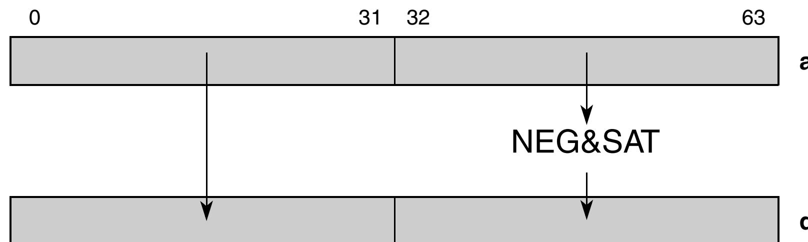


Figure 3-533. Vector Negate Word Odd and Saturate (__ev_negwos)

d	a	Maps to
__ev64_opaque__	__ev64_opaque__	**evnegwos d,a**

__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**.

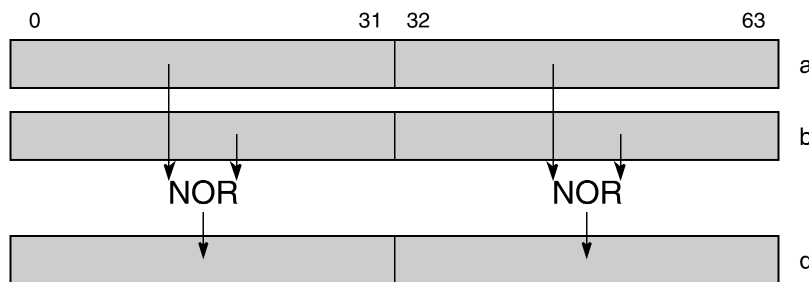


Figure 3-534. Vector NOR (__ev_nor)

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**.

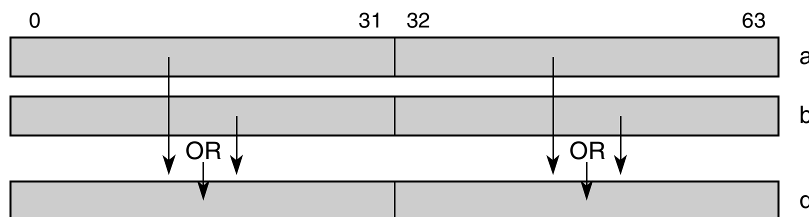


Figure 3-535. Vector OR (__ev_or)

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**.

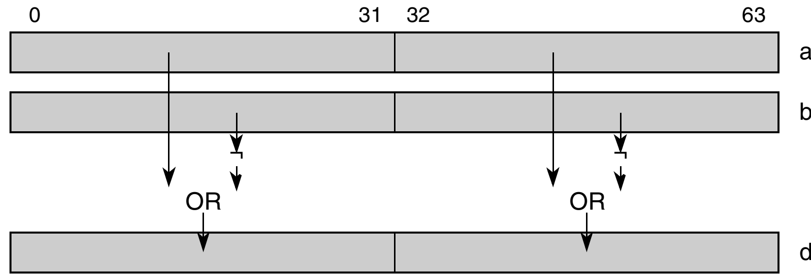


Figure 3-536. Vector OR with Complement (__ev_orc)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evorc d,a,b**

__ev_perm

__ev_perm

Vector Permute

d = __ev_perm (a,b)

```
sa ←b_{0:7}
sb ←b_{8:15}
sc ←b_{16:23}
sd ←b_{24:31}
se ←b_{32:39}
sf ←b_{40:47}
sg ←b_{48:55}
sh ←b_{56:63}

if (sa <_{ui} 8) then d_{0:7} ←a_{a*8:(a*8)+7} elseif (sa_0=1) then d_{0:7} ←0xFF else d_{0:7} ←0x00
if (sb <_{ui} 8) then d_{8:15} ←a_{b*8:(b*8)+7} elseif (sb_0=1) then d_{8:15}←0xFF else d_{8:15}←0x00
if (sc <_{ui} 8) then d_{16:23} ←a_{c*8:(c*8)+7} elseif (sc_0=1) then d_{16:23}←0xFF else d_{16:23}←0x00
if (sd <_{ui} 8) then d_{24:31} ←a_{d*8:(d*8)+7} elseif (sd_0=1) then d_{24:31}←0xFF else d_{24:31}←0x00
if (se <_{ui} 8) then d_{32:39} ←a_{e*8:(e*8)+7} elseif (se_0=1) then d_{32:39}←0xFF else d_{32:39}←0x00
if (sf <_{ui} 8) then d_{40:47} ←a_{f*8:(f*8)+7} elseif (sf_0=1) then d_{40:47}←0xFF else d_{40:47}←0x00
if (sg <_{ui} 8) then d_{48:55} ←a_{g*8:(g*8)+7} elseif (sg_0=1) then d_{48:23}←0xFF else d_{48:55}←0x00
if (sh <_{ui} 8) then d_{56:63} ←a_{h*8:(h*8)+7} elseif (sh_0=1) then d_{16:23}←0xFF else d_{56:63}←0x00
```

The contents of parameter **b** are used as a select vector. For each byte in the destination vector, a byte is selected from either parameter **a**, a constant of 0xFF, or a constant of 0x00 by the corresponding byte of the select vector. The selected byte values are placed into parameter **d**.

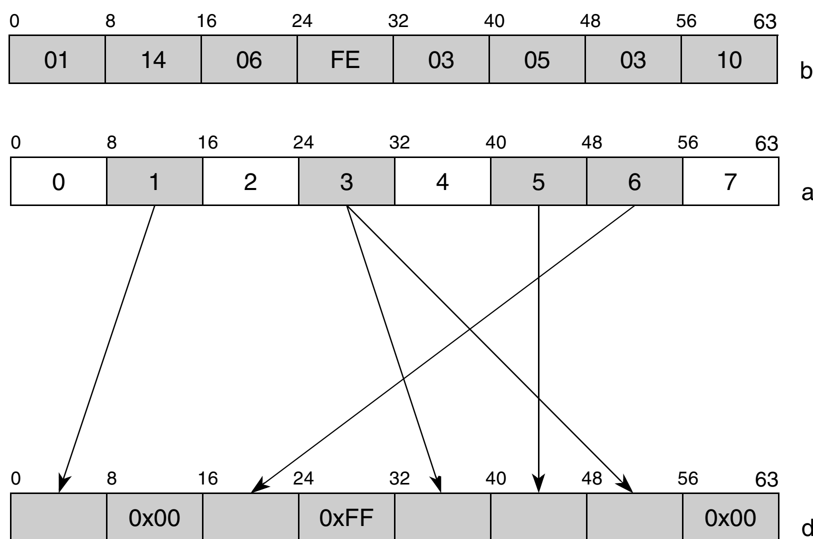


Figure 3-537. Vector Permute (__ev_perm)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evperm d,a,b**

__ev_perm2

__ev_perm2

Vector Permute (form 2)

d = __ev_perm2 (a,b,c)

```
temp0:127 ←b0:63 || c0:63
tempctl0:63 ←a0:63
sa ←tempctl0:7
sb ←tempctl8:15
sc ←tempctl16:23
sd ←tempctl24:31
se ←tempctl32:39
sf ←tempctl40:47
sg ←tempctl48:55
sh ←tempctl56:63

if (sa <ui 16) then d0:7 ←tempsa*8:(sa*8)+7 else d0:7 ←0x00
if (sb <ui 16) then d8:15 ←tempsb*8:(sb*8)+7 else d8:15 ←0x00
if (sc <ui 16) then d16:23 ←tempsc*8:(sc*8)+7 else d16:23 ←0x00
if (sd <ui 16) then d24:31 ←tempsd*8:(sd*8)+7 else d24:31 ←0x00
if (se <ui 16) then d32:39 ←tempse*8:(se*8)+7 else d32:39 ←0x00
if (sf <ui 16) then d40:47 ←tempsf*8:(sf*8)+7 else d40:47 ←0x00
if (sg <ui 16) then d48:55 ←tempsg*8:(sg*8)+7 else d48:55 ←0x00
if (sh <ui 16) then d56:63 ←tempsh*8:(sh*8)+7 else d56:63 ←0x00
```

The contents of parameters **b** and **c** are concatenated into a 128-bit source vector consisting of bytes 0-15. The contents of parameter **a** are copied to a temporary select vector. For each byte in the destination vector, a byte is selected from either the source vector or a constant of 0x00 by the corresponding byte of the select vector. The selected byte values are placed into parameter **d**.

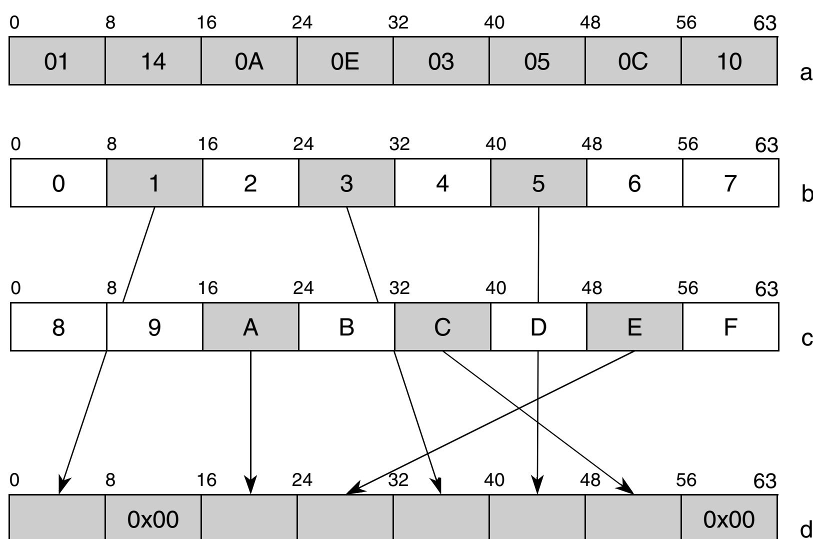


Figure 3-538. Vector Permute (form 2) (__ev_perm2)

d	a	b	c	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	d ←a evperm2 d,b,c

__ev_perm3 __ev_perm3

Vector Permute (form 3)

d = __ev_perm3 (a,b,c)

```
temp0:127 ←a0:63 || c0:63
tempctl0:63 ←b0:63
sa ←tempctl0:7
sb ←tempctl8:15
sc ←tempctl16:23
sd ←tempctl24:31
se ←tempctl32:39
sf ←tempctl40:47
sg ←tempctl48:55
sh ←tempctl56:63

if (sa <ui 16) then d0:7 ←tempsa*8:(sa*8)+7 else d0:7 ←0x00
if (sb <ui 16) then d8:15 ←tempsb*8:(sb*8)+7 else d8:15 ←0x00
if (sc <ui 16) then d16:23 ←tempsc*8:(sc*8)+7 else d16:23 ←0x00
if (sd <ui 16) then d24:31 ←tempsd*8:(sd*8)+7 else d24:31 ←0x00
if (se <ui 16) then d32:39 ←tempse*8:(se*8)+7 else d32:39 ←0x00
if (sf <ui 16) then d40:47 ←tempsf*8:(sf*8)+7 else d40:47 ←0x00
if (sg <ui 16) then d48:55 ←tempsg*8:(sg*8)+7 else d48:55 ←0x00
if (sh <ui 16) then d56:63 ←tempsh*8:(sh*8)+7 else d56:63 ←0x00
```

The contents of parameters **a** and **c** are concatenated into a 128-bit source vector consisting of bytes 0:15. The contents of parameter **b** are copied to a temporary select vector. For each byte in the destination vector, a byte is selected from either the source vector or a constant of 0x00 by the corresponding byte of the select vector. The selected byte values are placed into parameter **d**.

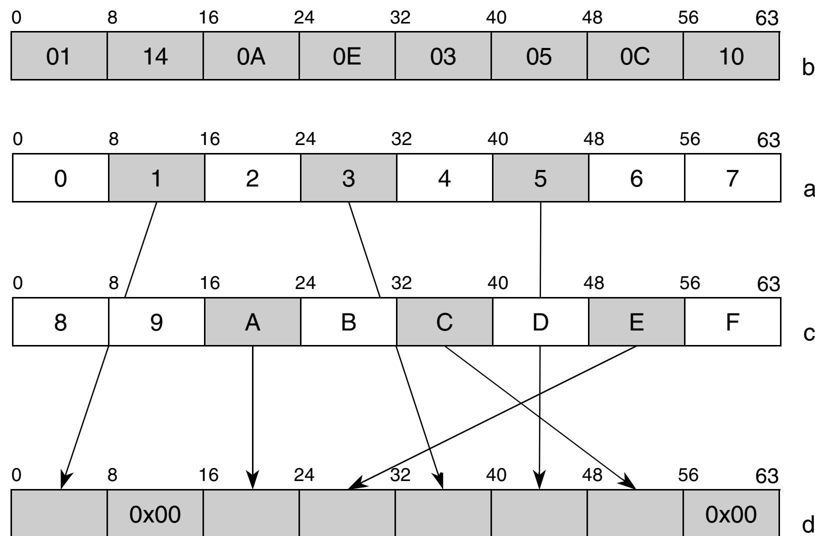


Figure 3-539. Vector Permute (form 3) (__ev_perm3)

d	a	b	c	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**d ←a** **evperm3 d,b,c**

__ev_pksdshefrs __ev_pksdshefrs

Vector Pack Signed Doublewords to Signed Halfwords Even, Fractional, Round, and Saturate

d = __ev_pksdshefrs (a,b)

```
// h0
if (a0:63 >=si 0x0000_0000_7FFF_8000) | (a0:63 <si 0xFFFF_FFFF_7FFF_8000) then
    ovh ←1
    temph0:31 ←SATURATE(ovh, a0, 0x8000_0000, 0x7fff_0000, --------)
else
    ovh ←0
    tempr0:63 ←ROUND(a0:63,16)
    temph0:31 ←tempr32:47 || 160
d0:31 ←temph0:31
// h2
if (b0:63 >=si 0x0000_0000_7FFF_8000) | (b0:63 <si 0xFFFF_FFFF_7FFF_8000) then
    ovl ←1
    templ0:31 ←SATURATE(ovl, b0, 0x8000_0000, 0x7fff_0000, --------)
else
    ovl ←0
    tempr0:63 ←ROUND(b0:63,16)
    templ0:31 ←tempr32:47 || 160
d32:63 ←templ0:31
SPEFSCROVH ←ovh
SPEFSCROV ←ovl
SPEFSCRSOVH ←SPEFSCRSOVH | ovh
SPEFSCRSOV ←SPEFSCRSOV | ovl
```

The signed 64-bit guarded fractional elements of parameters **a** and **b** are rounded and saturated to 16 bits. The results in 1.15 fractional format are packed into the even halfwords of parameter **d**, zeroing the odd halfwords. The original values are assumed to be in 33.31 fractional format as a result of one or more guarded halfword fractional operations. Any saturation is recorded in the SPEFSCR overflow and summary overflow bits.

Other registers altered: SPEFSCR

Figure 3-540. Vector Pack Signed Doublewords to Signed Halfwords Even Fractional, Round, and Saturate (__ev_pksdshefrs)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evpksdshefrs d,a,b**

__ev_pksdswfrs __ev_pksdswfrs

Vector Pack Signed Doublewords to Signed Words Fractional, Round and Saturate

d = __ev_pksdswfrs (a,b)

```
// w0
if (a0:63 >=si 0x7FFF_FFFF_8000_0000) then
    temph0:63 ←0x7FFF_FFFF_0000_0000
    ovh ←1
else
    temph0:63 ←ROUND(a0:63,32)
    ovh ←0
d0:31 ←temph0:31

// w1
if (b0:63 >=si 0x7FFF_FFFF_8000_0000) then
    templ0:63 ←0x7FFF_FFFF_0000_0000
    ovl ←1
else
    templ0:63 ←ROUND(b0:63,32)
    ovl ←0
d32:63 ←templ0:31

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

The signed 64-bit fractional elements of parameters **a** and **b** are rounded and saturated to 32 bits using the current rounding mode in SPEFSCR. The 32-bit results are packed into parameter **d**. Any saturation is recorded in the SPEFSCR overflow and summary overflow bits.

Other registers altered: SPEFSCR

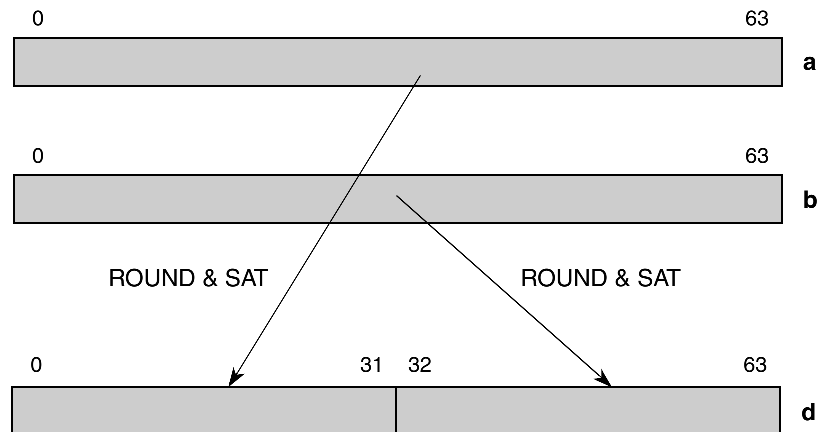


Figure 3-541. Vector Pack Signed Doublewords to Signed Words Fractional, Round and Saturate (__ev_pksdswfrs)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evpksdswfrs d,a,b**

__ev_pksdsws __ev_pksdsws

Vector Pack Signed Doublewords to Signed Words and Saturate

d = __ev_pksdsws (a,b)

```
// w0
if ((a0:63 <si 0xFFFF_FFFF_8000_0000) | (a0:63 >si 0x0000_0000_7FFF_FFFF)) then ovh=1 else
ovh=0;
d0:31 ←SATURATE(ovh, a0, 0x8000_0000, 0x7fff_ffff, a32:63)

// w1
if ((b0:63 <si 0xFFFF_FFFF_8000_0000) | (b0:63 >si 0x0000_0000_7FFF_FFFF)) then ovl=1 else
ovl=0;
d32:63 ←SATURATE(ovl, b0, 0x8000_0000, 0x7fff_ffff, b32:63)


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

The signed 64-bit elements of parameters **a** and **b** are saturated to 32 bits. The 32-bit results are packed into parameter **d**. Any saturation is recorded in the SPEFSCR overflow and summary overflow bits.

Other registers altered: SPEFSCR

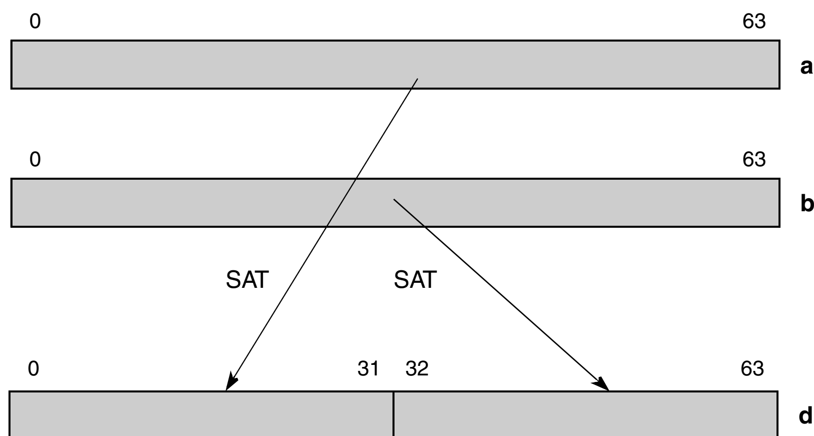


Figure 3-542. Vector Pack Signed Doublewords to Signed Words and Saturate (__ev_pksdsws)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evpksdsws d,a,b**

__ev_pkshsbs __ev_pkshsbs

Vector Pack Signed Half Words to Signed Bytes and Saturate

d = __ev_pkshsbs (a,b)

```
// b0
if ((a0:15 <si 0xFF80) | (a0:15 >si 0x007F)) then ovb0=1 else ovb0=0;
d0:7 ←SATURATE(ovb0, a0, 0x80, 0x7f, a8:15)
// b1
if ((a16:31 <si 0xFF80) | (a16:31 >si 0x007F)) then ovb1=1 else ovb1=0;
d8:15 ←SATURATE(ovb1, a16, 0x80, 0x7f, a24:31)
// b2
if ((a32:47 <si 0xFF80) | (a32:47 >si 0x007F)) then ovb2=1 else ovb2=0;
d16:23 ←SATURATE(ovb2, a32, 0x80, 0x7f, a40:47)
// b3
if ((a48:63 <si 0xFF80) | (a48:63 >si 0x007F)) then ovb3=1 else ovb3=0;
d24:31 ←SATURATE(ovb3, a48, 0x80, 0x7f, a56:63)
// b4
if ((b0:15 <si 0xFF80) | (b0:15 >si 0x007F)) then ovb4=1 else ovb4=0;
d32:39 ←SATURATE(ovb4, b0, 0x80, 0x7f, b8:15)
// b5
if ((b16:31 <si 0xFF80) | (b16:31 >si 0x007F)) then ovb5=1 else ovb5=0;
d40:47 ←SATURATE(ovb5, b16, 0x80, 0x7f, b24:31)
// b6
if ((b32:47 <si 0xFF80) | (b32:47 >si 0x007F)) then ovb6=1 else ovb6=0;
d48:55 ←SATURATE(ovb6, b32, 0x80, 0x7f, b40:47)
// b7
if ((b48:63 <si 0xFF80) | (b48:63 >si 0x007F)) then ovb7=1 else ovb7=0;
d56:63 ←SATURATE(ovb7, b48, 0x80, 0x7f, b56:63)

ovh ←ovb0 | ovb1 | ovb2 | ovb3; ovl ←ovb4 | ovb5 | ovb6 | ovb7;
SPEFSCROVH ←ovh; SPEFSCROV ←ovl;
SPEFSCRSOVH ←SPEFSCRSOVH | ovh; SPEFSCRSOV ←SPEFSCRSOV | ovl;
```

The signed 16-bit elements of parameters **a** and **b** are saturated to 8 bits. The 8-bit results are packed into parameter **d**. Any saturation is recorded in the SPEFSCR overflow and summary overflow bits.

Other registers altered: SPEFSCR

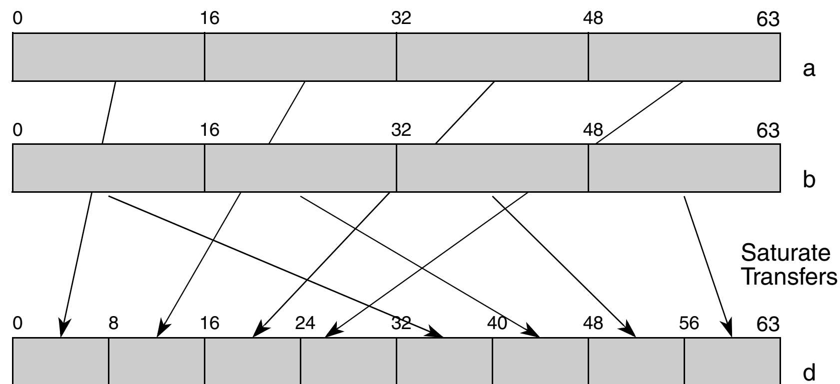


Figure 3-543. Vector Pack Signed Half Words to Signed Bytes and Saturate (__ev_pkshsbs)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evpkshsbs d,a,b**

__ev_pkshubs __ev_pkshubs

Vector Pack Signed Half Words to Unsigned Bytes and Saturate

d = __ev_pkshubs (a,b)

```
// b0
if ((a0:15 <si 0x0000) | (a0:15 >si 0x00FF)) then ovb0=1 else ovb0=0;
d0:7 ←SATURATE(ovb0, a0, 0x00, 0xFF, a8:15)
// b1
if ((a16:31 <si 0x000) | (a16:31 >si 0x00FF)) then ovb1=1 else ovb1=0;
d8:15 ←SATURATE(ovb1, a16, 0x00, 0xFF, a24:31)
// b2
if ((a32:47 <si 0x0000) | (a32:47 >si 0x00FF)) then ovb2=1 else ovb2=0;
d16:23 ←SATURATE(ovb2, a32, 0x00, 0xFF, a40:47)
// b3
if ((a48:63 <si 0x0000) | (a48:63 >si 0x00FF)) then ovb3=1 else ovb3=0;
d24:31 ←SATURATE(ovb3, a48, 0x00, 0xFF, a56:63)
// b4
if ((b0:15 <si 0x0000) | (b0:15 >si 0x00FF)) then ovb4=1 else ovb4=0;
d32:39 ←SATURATE(ovb4, b0, 0x00, 0xFF, b8:15)
// b5
if ((b16:31 <si 0x0000) | (b16:31 >si 0x00FF)) then ovb5=1 else ovb5=0;
d40:47 ←SATURATE(ovb5, b16, 0x00, 0xFF, b24:31)
// b6
if ((b32:47 <si 0x0000) | (b32:47 >si 0x00FF)) then ovb6=1 else ovb6=0;
d48:55 ←SATURATE(ovb6, b32, 0x00, 0xFF, b40:47)
// b7
if ((b48:63 <si 0x0000) | (b48:63 >si 0x00FF)) then ovb7=1 else ovb7=0;
d56:63 ←SATURATE(ovb7, b48, 0x00, 0xFF, b56:63)

ovh ←ovb0 | ovb1 | ovb2 | ovb3 ; ovl ←ovb4 | ovb5 | ovb6 | ovb7
SPEFSCROVH ←ovh ; SPEFSCROV ←ovl
SPEFSCRSOVH ←SPEFSCRSOVH | ovh ; SPEFSCRSOV ←SPEFSCRSOV | ovl.
```

The signed 16-bit elements of parameters **a** and **b** are saturated to 8-bit unsigned elements. Negative elements saturate to 0. The 8-bit results are packed into parameter **d**. Any saturation is recorded in the SPEFSCR overflow and summary overflow bits.

Other registers altered: SPEFSCR

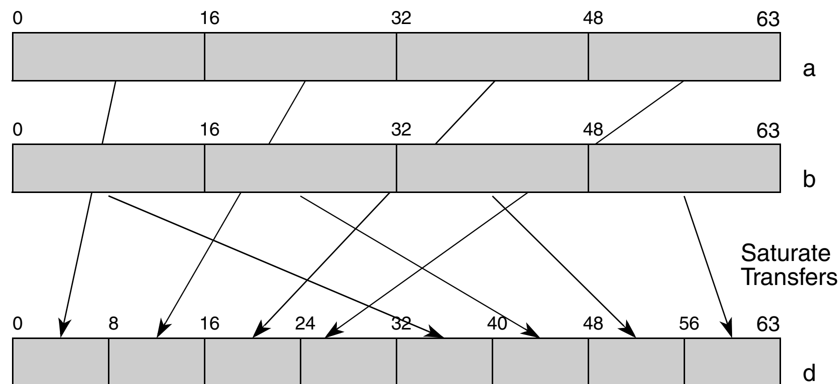


Figure 3-544. Vector Pack Signed Half Words to Unsigned Bytes and Saturate (__ev_pkshubs)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evpkshubs d,a,b**

__ev_pkswgshefrs __ev_pkswgshefrs

Vector Pack Signed Words Guarded to Signed Halfwords Even Fractional Round and Saturate

d = __ev_pkswgshefrs (a,b)

```
// h0
if (a0:63 >=si 0x0000_7FFF_8000_0000) | (a0:63 <si 0xFFFF_7FFF_8000_0000) then
    ovh ←1
    temph0:31 ←SATURATE(ovh, a0, 0x8000_0000, 0x7fff_0000, --------)
else
    ovh ←0
    tempr0:63 ←ROUND(a0:63,32)
    temph0:31 ←tempr16:31 || 160
d0:31 ←temph0:31
// h2
if (b0:63 >=si 0x0000_7FFF_8000_0000) | (b0:63 <si 0xFFFF_7FFF_8000_0000) then
    ovl ←1
    templ0:31 ←SATURATE(ovl, b0, 0x8000_0000, 0x7fff_0000, --------)
else
    ovl ←0
    tempr0:63 ←ROUND(b0:63,32)
    templ0:31 ←tempr16:31 || 160
d32:63 ←templ0:31
SPEFSCROVH ←ovh
SPEFSCROV ←ovl
SPEFSCRSOVH ←SPEFSCRSOVH | ovh
SPEFSCRSOV ←SPEFSCRSOV | ovl.
```

The signed 64-bit elements of parameters **a** and **b** are rounded and saturated to 16 bits. The 16-bit results in 1.15 fractional format are packed into the even halfwords of parameter **d**. The original values are assumed to be in 17.47 fractional format as a result of one or more guarded word fractional operations. Any saturation is recorded in the SPEFSCR overflow and summary overflow bits.

Other registers altered: SPEFSCR

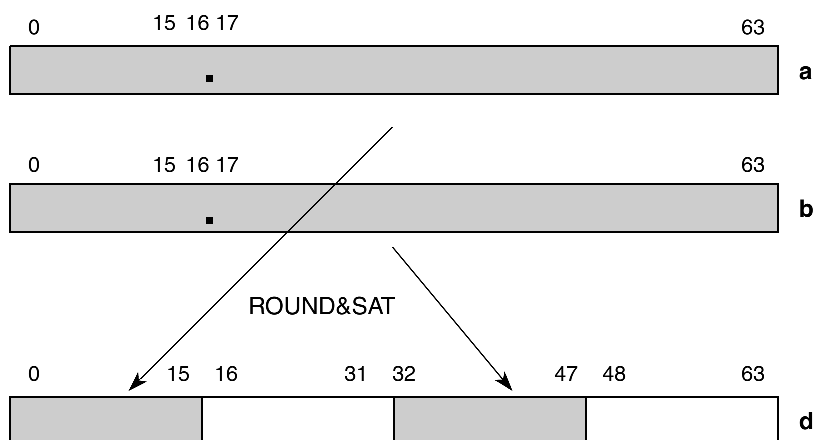


Figure 3-545. Vector Pack Signed Words Guarded to Signed Words Fractional Round and Saturate (__ev_pkswgswfrs)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evpkswgswfrs d,a,b**

__ev_pkswgswfrs __ev_pkswgswfrs

Vector Pack Signed Words Guarded to Signed Words Fractional Round and Saturate

d = __ev_pkswgswfrs (a,b)

```
// w0
if ((a0:63 <si 0xFFFF_7FFF_FFFF_8000) | (a0:63 >=si 0x0000_7FFF_FFFF_8000)) then
    ovh=1
    temph0:31 ←SATURATE(ovh, a0, 0x8000_0000, 0x7fff_ffff, -----)
else
    ovh=0
    tempr0:63 ←ROUND(a0:63, 16)
    temph0:31 ←tempr16:47
d0:31 ←temph0:31

// w1
if ((b0:63 <si 0xFFFF_7FFF_FFFF_8000) | (b0:63 >=si 0x0000_7FFF_FFFF_8000)) then
    ovl=1
    templ0:31 ←SATURATE(ovl, b0, 0x8000_0000, 0x7fff_ffff, -----)
else
    ovl=0
    tempr0:63 ←ROUND(b0:63, 16)
    templ0:31 ←tempr16:47
d32:63 ←templ0:31

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

The signed 64-bit elements of parameters **a** and **b** are saturated to 32 bits. The 32-bit results in 1.31 fractional format are packed into parameter **d**. The original values are assumed to be in 17.47 fractional format as a result of one or more guarded word fractional operations. Any saturation is recorded in the SPEFSCR overflow and summary overflow bits.

Other registers altered: SPEFSCR

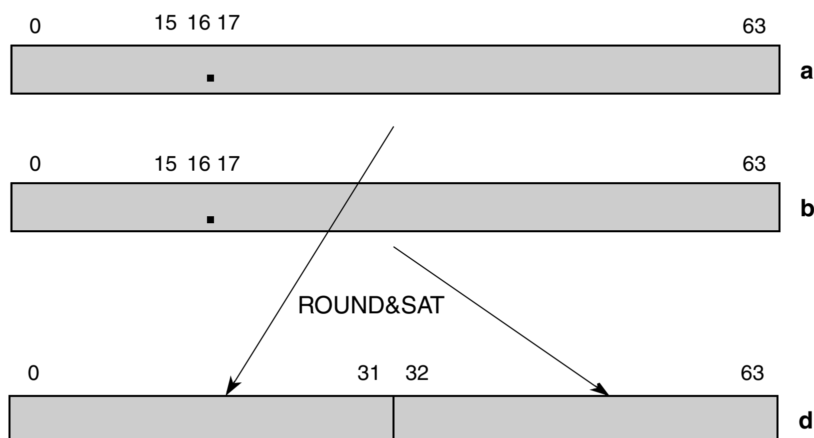


Figure 3-546. Vector Pack Signed Words Guarded to Signed Words Fractional Round and Saturate (__ev_pkswgswfrs)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evpkswgswfrs d,a,b**

__ev_pkswshfrs __ev_pkswshfrs

Vector Pack Signed Words to Signed Halfwords Fractional, Round and Saturate

d = __ev_pkswshfrs (a,b)

```
if (a0:31 >=si 0x7FFF_8000) then
    ovh0 ←1; temph00:15 ←0x7FFF
else
    ovh0 ←0; tempr0:63 ←ROUND(EXTS64(a0:31),16); temph00:15 ←tempr32:47
d0:15 ←temph00:15

if (a32:63 >=si 0x7FFF_8000) then
    ovh1 ←1; temph10:15 ←0x7FFF
else
    ovh1 ←0; tempr0:63 ←ROUND(EXTS64(a32:63),16); temph10:15 ←tempr32:47
d16:31 ←temph10:15

if (b0:31 >=si 0x7FFF_8000) then
    ovh2 ←1; temph20:15 ←0x7FFF
else
    ovh2 ←0; tempr0:63 ←ROUND(EXTS64(b0:31),16); temph20:15 ←tempr32:47
d32:47 ←temph20:15

if (b32:63 >=si 0x7FFF_8000) then
    ovh3 ←1; temph30:15 ←0x7FFF
else
    ovh3 ←0; tempr0:63 ←ROUND(EXTS64(b32:63),16); temph30:15 ←tempr32:47
d48:63 ←temph30:15

ovh ←ovh0 | ovh1; ovl ←ovh2 | ovh3

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

The signed 32-bit fractional elements of parameters **a** and **b** are rounded and saturated to 16 bits using the current rounding mode in SPEFSCR. The 16-bit results are packed into parameter **d**. Any saturation is recorded in the SPEFSCR overflow and summary overflow bits.

Other registers altered: SPEFSCR

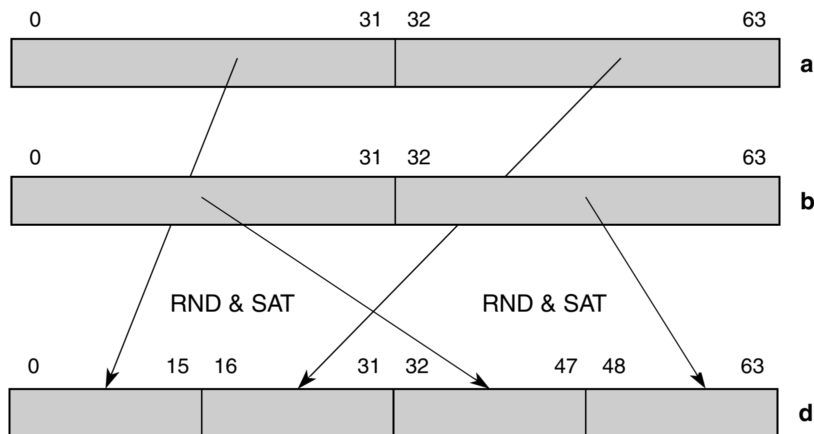


Figure 3-547. Vector Pack Signed Words to Signed Halfwords Fractional, Round and Saturate (__ev_pkswshfrs)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evpkswshfrs d,a,b**

__ev_pkswshilvfrs __ev_pkswshilvfrs

Vector Pack Signed Words to Signed Halfwords Interleaved, Fractional, Round and Saturate

d = __ev_pkswshilvfrs (a,b)

```
if (a0:31 >=si 0x7FFF_8000) then
    ovh0 ←1; temph00:15 ←0x7FFF
else
    ovh0 ←0; tempr0:63 ←ROUND(EXTS64(a0:31),16); temph00:15 ←tempr32:47
d0:15 ←temph00:15

if (a32:63 >=si 0x7FFF_8000) then
    ovh2 ←1; temph20:15 ←0x7FFF
else
    ovh2 ←0; tempr0:63 ←ROUND(EXTS64(a32:63),16); temph20:15 ←tempr32:47
d32:47 ←temph20:15

if (b0:31 >=si 0x7FFF_8000) then
    ovh1 ←1; temph20:15 ←0x7FFF
else
    ovh1 ←0; tempr0:63 ←ROUND(EXTS64(b0:31),16); temph10:15 ←tempr32:47
d16:31 ←temph10:15

if (b32:63 >=si 0x7FFF_8000) then
    ovh3 ←1; temph30:15 ←0x7FFF
else
    ovh3 ←0; tempr0:63 ←ROUND(EXTS64(b32:63),16); temph30:15 ←tempr32:47
d48:63 ←temph30:15

ovh ←ovh0 | ovh1; ovl ←ovh2 | ovh3

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

The signed 32-bit fractional elements of parameters **a** and **b** are rounded and saturated to 16 bits using the current rounding mode in SPEFSCR. The 16-bit results are packed into parameter **d** with interleaving. Any saturation is recorded in the SPEFSCR overflow and summary overflow bits.

Other registers altered: SPEFSCR

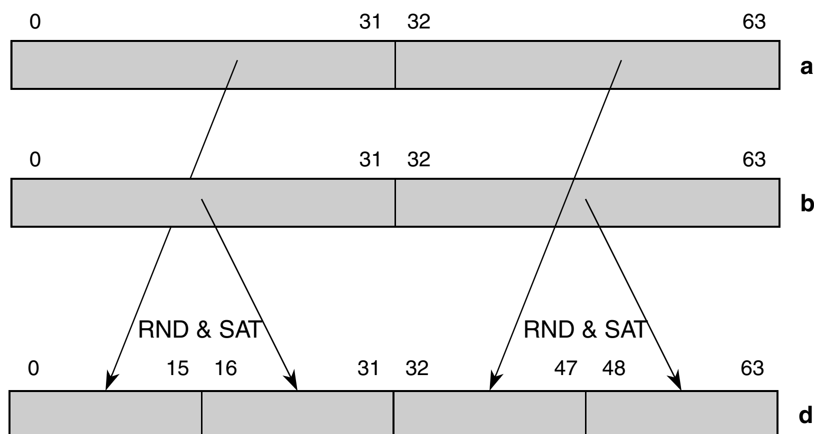


Figure 3-548. Vector Pack Signed Words to Signed Halfwords Interleaved Fractional, Round and Saturate (__ev_pkswshilvfrs)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evpkswshilvfrs d,a,b**

__ev_pkswshilvs __ev_pkswshilvs

Vector Pack Signed Words to Signed Halfwords Interleaved and Saturate

d = __ev_pkswshilvs (a,b)

```
// h0
if ((a0:31 <si 0xFFFF8000) | (a0:31 >si 0x00007FFF)) then ovh0=1 else ovh0=0;
d0:15 ←SATURATE(ovh0, a0, 0x8000, 0x7fff, a16:31)

// h2
if ((a32:63 <si 0xFFFF8000) | (a32:63 >si 0x00007FFF)) then ovh2=1 else ovh2=0;
d32:47 ←SATURATE(ovh2, a32, 0x8000, 0x7fff, a48:63)

// h1
if ((b0:31 <si 0xFFFF8000) | (b0:31 >si 0x00007FFF)) then ovh1 else ovh1=0;
d16:23 ←SATURATE(ovh1, b0, 0x8000, 0x7fff, b16:31)

// h3
if ((b32:63 <si 0xFFFF8000) | (b32:63 >si 0x00007FFF)) then ovh3=1 else ovh3=0;
d48:63 ←SATURATE(ovh3, b32, 0x8000, 0x7fff, b48:63)

ovh ←ovh0 | ovh1
ovl ←ovh2 | ovh3
SPEFSCROVH ←ovh
SPEFSCROV ←ovl
SPEFSCRSOVH ←SPEFSCRSOVH | ovh
SPEFSCRSOV ←SPEFSCRSOV | ovl.
```

The signed 32-bit elements of parameters **a** and **b** are saturated to 16 bits. The 16-bit results are packed into parameter **d** with interleaving. Any saturation is recorded in the SPEFSCR overflow and summary overflow bits.

Other registers altered: SPEFSCR

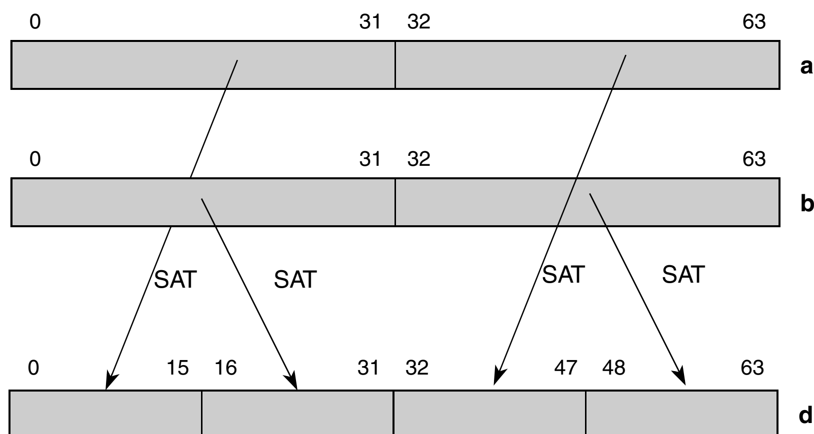


Figure 3-549. Vector Pack Signed Words to Signed Halfwords Interleaved and Saturate (__ev_pkswshilvs)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evpkswshilvs d,a,b**

__ev_pkswshs

Vector Pack Signed Words to Signed Half Words and Saturate

d = __ev_pkswshs (a,b)

```
// h0
if ((a0:31 <si 0xFFFF8000) | (a0:31 >si 0x00007FFF)) then ovh0=1 else ovh0=0;
d0:15 ←SATURATE(ovh0, a0, 0x8000, 0x7fff, a16:31)

// h1
if ((a32:63 <si 0xFFFF8000) | (a32:63 >si 0x00007FFF)) then ovh1=1 else ovh1=0;
d16:31 ←SATURATE(ovh1, a32, 0x8000, 0x7fff, a48:63)

// h2
if ((b0:31 <si 0xFFFF8000) | (b0:31 >si 0x00007FFF)) then ovh21 else ovh2=0;
d32:47 ←SATURATE(ovh2, b0, 0x8000, 0x7fff, b16:31)

// h3
if ((b32:63 <si 0xFFFF8000) | (b32:63 >si 0x00007FFF)) then ovh3=1 else ovh3=0;
d48:63 ←SATURATE(ovh3, b32, 0x8000, 0x7fff, b48:63)

ovh ←ovh0 | ovh1
ovl ←ovh2 | ovh3

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

The signed 32-bit elements of parameters **a** and **b** are saturated to 16 bits. The 16-bit results are packed into parameter **d**. Any saturation is recorded in the SPEFSCR overflow and summary overflow bits.

Other registers altered: SPEFSCR

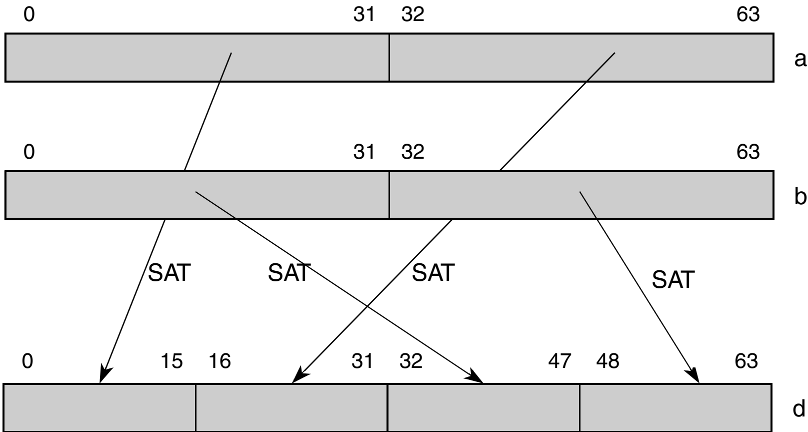


Figure 3-550. Vector Pack Signed Words to Signed Half Words and Saturate (__ev_pkswshs)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evpkswshs d,a,b**

__ev_pkswuhs __ev_pkswuhs

Vector Pack Signed Words to Unsigned Half Words and Saturate

d = __ev_pkswuhs (a,b)

```
// h0
if ((a0:31 <si 0x00000000) | (a0:31 >si 0x0000FFFF)) then ovh0=1 else ovh0=0;
d0:15 ←SATURATE(ovh0, a0, 0x0000, 0xFFFF, a16:31)

// h1
if ((a32:63 <si 0x00000000) | (a32:63 >si 0x0000FFFF)) then ovh1=1 else ovh1=0;
d16:31 ←SATURATE(ovh1, a32, 0x0000, 0xFFFF, a48:63)

// h2
if ((b0:31 <si 0x00000000) | (b0:31 >si 0x0000FFFF)) then ovh21 else ovh2=0;
d32:47 ←SATURATE(ovh2, b0, 0x0000, 0xFFFF, b16:31)

// h3
if ((b32:63 <si 0x00000000) | (b32:63 >si 0x0000FFFF)) then ovh3=1 else ovh3=0;
d48:63 ←SATURATE(ovh3, b32, 0x0000, 0xFFFF, b48:63)

ovh ←ovh0 | ovh1
ovl ←ovh2 | ovh3

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

The signed 32-bit elements of parameters **a** and **b** are saturated to 16-bit unsigned elements. Negative elements saturate to 0. Any saturation is recorded in the SPEFSCR overflow and summary overflow bits.

Other registers altered: SPEFSCR

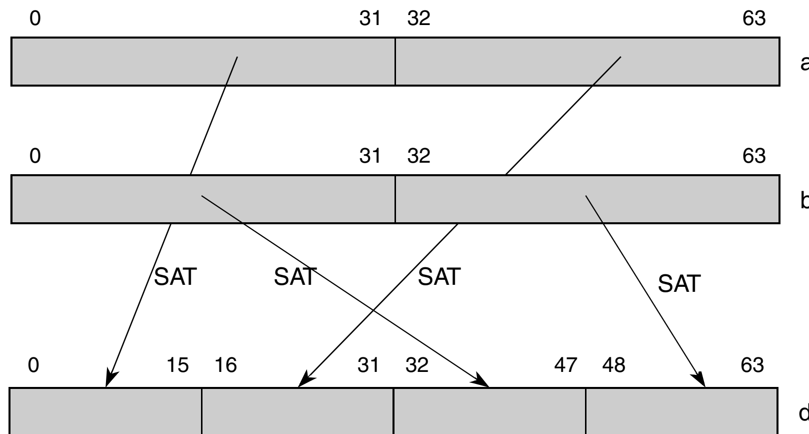


Figure 3-551. Vector Pack Signed Words to Unsigned Half Words and Saturate (__ev_pkswuhs)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evpkswuhs d,a,b**

__ev_pkuduws __ev_pkuduws

Vector Pack Unsigned Doublewords to Unsigned Words and Saturate

d = __ev_pkuduws (a,b)

```
// w0
if (a0:63 >ui 0x0000_0000_FFFF_FFFF) then ovh=1 else ovh=0;
d0:31 ←SATURATE(ovh, a0, 0xffff_ffff, 0xffff_ffff, a32:63)

// w1
if (b0:63 >ui 0x0000_0000_FFFF_FFFF) then ovl=1 else ovl=0;
d32:63 ←SATURATE(ovl, b0, 0xffff_ffff, 0xffff_ffff, b32:63)


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

The unsigned 64-bit elements of parameters **a** and **b** are saturated to 32 bits. The 32-bit results are packed into parameter **d**. Any saturation is recorded in the SPEFSCR overflow and summary overflow bits.

Other registers altered: SPEFSCR

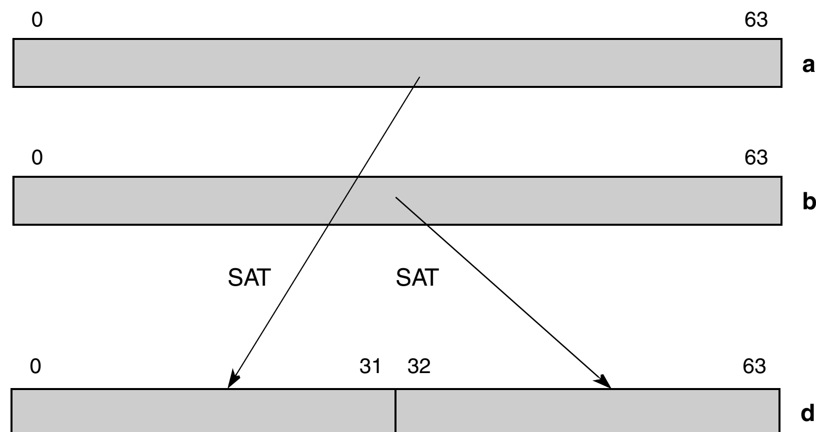


Figure 3-552. Vector Pack Unsigned Doublewords to Unsigned Words and Saturate (__ev_pkuduws)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evpkuduws d,a,b**

__ev_pkuhubs __ev_pkuhubs

Vector Pack Unsigned Half Words to Unsigned Bytes and Saturate

d = __ev_pkuhubs (a,b)

```
// b0
if (a0:15 >ui 0x00FF) then ovb0=1 else ovb0=0;
d0:7 ←SATURATE(ovb0, 0, 0xFF, 0xFF, a8:15)
// b1
if (a16:31 >ui 0x00FF) then ovb1=1 else ovb1=0;
d8:15 ←SATURATE(ovb1, 0, 0xFF, 0xFF, a24:31)
// b2
if (a32:47 >ui 0x00FF) then ovb2=1 else ovb2=0;
d16:23 ←SATURATE(ovb2, 0, 0xFF, 0xFF, a40:47)
// b3
if (a48:63 >ui 0x00FF) then ovb3=1 else ovb3=0;
d24:31 ←SATURATE(ovb3, 0, 0xFF, 0xFF, a56:63)
// b4
if (b0:15 >ui 0x00FF) then ovb4=1 else ovb4=0;
d32:39 ←SATURATE(ovb4, 0, 0xFF, 0xFF, b8:15)
// b5
if (b16:31 >ui 0x00FF) then ovb5=1 else ovb5=0;
d40:47 ←SATURATE(ovb5, 0, 0xFF, 0xFF, b24:31)
// b6
if (b32:47 >ui 0x00FF) then ovb6=1 else ovb6=0;
d48:55 ←SATURATE(ovb6, 0, 0xFF, 0xFF, b40:47)
// b7
if (b48:63 >ui 0x00FF) then ovb7=1 else ovb7=0;
d56:63 ←SATURATE(ovb7, 0, 0xFF, 0xFF, b56:63)

ovh ←ovb0 | ovb1 | ovb2 | ovb3 ; ovl ←ovb4 | ovb5 | ovb6 | ovb7
SPEFSCROVH ←ovh ; SPEFSCROV ←ovl
SPEFSCRSOVH ←SPEFSCRSOVH | ovh ; SPEFSCRSOV ←SPEFSCRSOV | ovl.
```

The unsigned 16-bit elements of parameters **a** and **b** are saturated to 8 bits. The 8-bit unsigned results are packed into parameter **d**. Any saturation is recorded in the SPEFSCR overflow and summary overflow bits.

Other registers altered: SPEFSCR

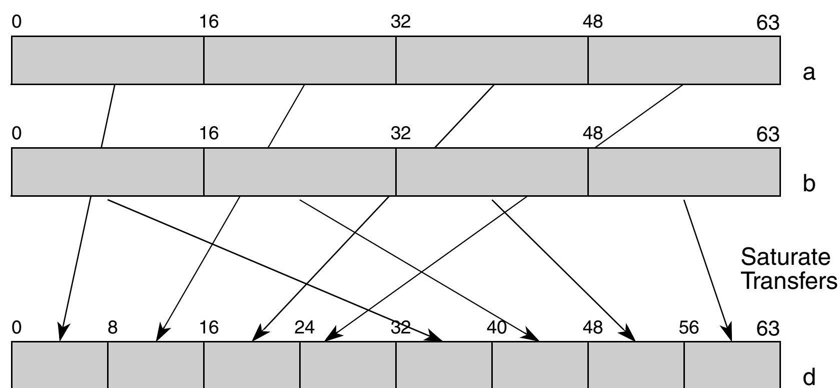


Figure 3-553. Vector Pack Unsigned Half Words to Unsigned Bytes and Saturate (__ev_pkuhubs)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evpkuhubs d,a,b**

__ev_pkuwuhs __ev_pkuwuhs

Vector Pack Unsigned Words to Unsigned Half Words and Saturate

d = __ev_pkuwuhs (a,b)

```
// h0
if (a0:31 >ui 0x0000FFFF) then ovh0=1 else ovh0=0;
d0:15 ←SATURATE(ovh0, 0, 0xFFFF, 0xFFFF, a16:31)

// h1
if ((a32:63 >ui 0x0000FFFF) then ovh1=1 else ovh1=0;
d16:31 ←SATURATE(ovh1, 0, 0xFFFF, 0xFFFF, a48:63)

// h2
if ((b0:31 >ui 0x0000FFFF) then ovh2=1 else ovh2=0;
d32:47 ←SATURATE(ovh2, 0, 0xFFFF, 0xFFFF, b16:31)

// h3
if ((b32:63 >ui 0x0000FFFF) then ovh3=1 else ovh3=0;
d48:63 ←SATURATE(ovh3, 0, 0xFFFF, 0xFFFF, b48:63)

ovh ←ovh0 | ovh1
ovl ←ovh2 | ovh3

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

The unsigned 32-bit elements of parameters **a** and **b** are saturated to 16 bits. The 16-bit unsigned results are packed into parameter **d**. Any saturation is recorded in the SPEFSCR overflow and summary overflow bits.

Other registers altered: SPEFSCR

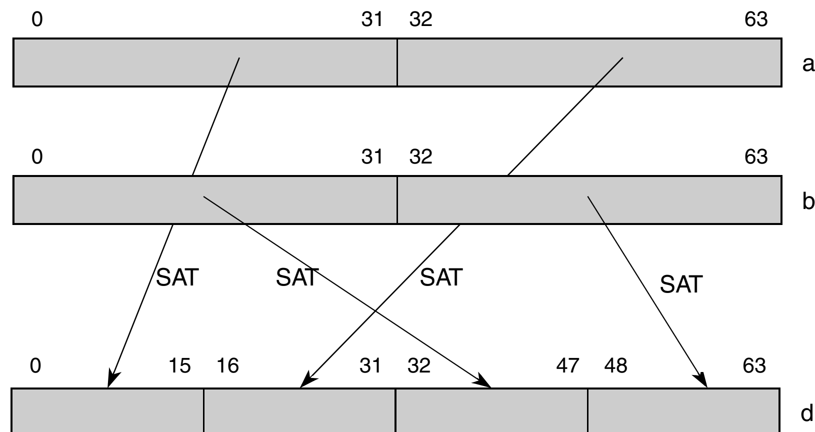


Figure 3-554. Vector Pack Unsigned Words to Unsigned Half Words and Saturate (__ev_pkuwuhs)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evpkuwuhs d,a,b**

__ev_popcntb

__ev_popcntb

Vector Population Count Byte

d = __ev_popcntb (a)

$d_{0:7} \leftarrow \text{POPCNT}(a_{0:7})$
$d_{8:15} \leftarrow \text{POPCNT}(a_{8:15})$
$d_{16:23} \leftarrow \text{POPCNT}(a_{16:23})$
$d_{24:31} \leftarrow \text{POPCNT}(a_{24:31})$
$d_{32:39} \leftarrow \text{POPCNT}(a_{32:39})$
$d_{40:47} \leftarrow \text{POPCNT}(a_{40:47})$
$d_{48:55} \leftarrow \text{POPCNT}(a_{48:55})$
$d_{56:63} \leftarrow \text{POPCNT}(a_{56:63})$

The number of set bits in each byte element of parameter **a** are counted, and the results are placed into the corresponding element of parameter **d**.

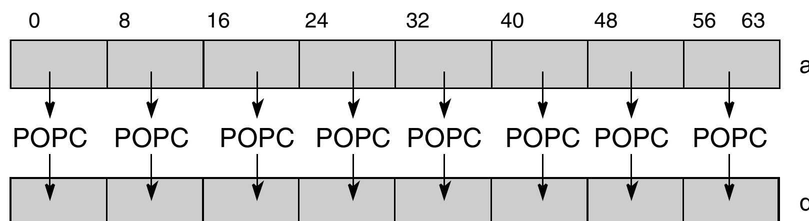


Figure 3-555. Vector Population Count Byte (__ev_popcntb)

d	a	Maps to
__ev64_opaque__	__ev64_opaque__	**evpopcntb d,a**

__ev_rlb

__ev_rlb

Vector Rotate Left Byte

d = __ev_rlb (a,b)

```
nb0 ←b5:7
nb1 ←b13:15
nb2 ←b21:23
nb3 ←b29:31
nb4 ←b37:39
nb5 ←b45:47
nb6 ←b53:55
nb7 ←b61:63

d0:7 ←ROTL(a0:7, nb0)
d8:15 ←ROTL(a8:15, nb1)
d16:23 ←ROTL(a16:23, nb2)
d24:31 ←ROTL(a24:31, nb3)
d32:39 ←ROTL(a32:39, nb4)
d40:47 ←ROTL(a40:47, nb5)
d48:55 ←ROTL(a48:55, nb6)
d56:63 ←ROTL(a56:63, nb7)
```

Each of the byte elements of parameter **a** are rotated left by an amount specified in the lower 3 bits of the corresponding byte elements of parameter **b**. The result is placed into parameter **d**. The separate rotate amounts for each element are specified by the lower 3 bits in each byte element of parameter **b** that lie in bit positions 5:7, 13:15, 21:23, 29:31, 37:39, 45:47 53:55, and 61:63.

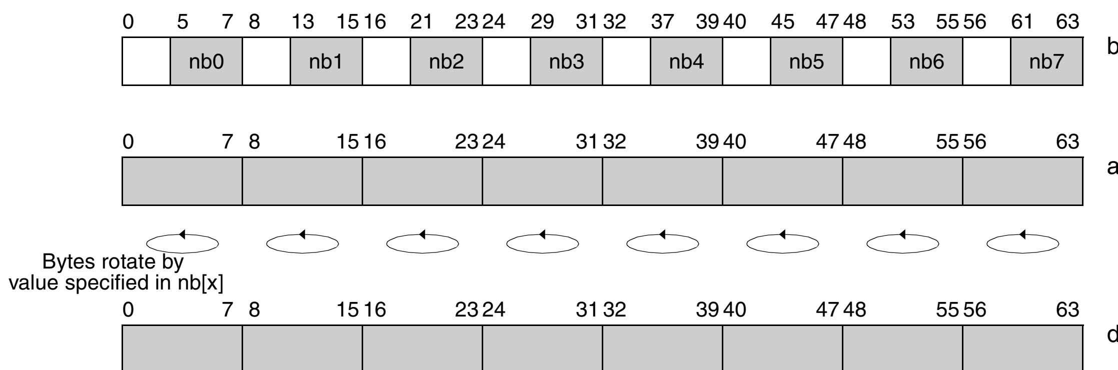


Figure 3-556. Vector Rotate Left Byte (__ev_rlb)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evrlb d,a,b**

__ev_rlbi

__ev_rlbi

Vector Rotate Left Byte Immediate

d = __ev_rlbi (a,b)

$$
\begin{aligned}
&UIMM \leftarrow EXTZ(b)\\
&n \leftarrow UIMM\\
&d_{0:7} \leftarrow ROTL(a_{0:7},\ n)\\
&d_{8:15} \leftarrow ROTL(a_{8:15},\ n)\\
&d_{16:23} \leftarrow ROTL(a_{16:23},\ n)\\
&d_{24:31} \leftarrow ROTL(a_{24:31},\ n)\\
&d_{32:39} \leftarrow ROTL(a_{32:39},\ n)\\
&d_{40:47} \leftarrow ROTL(a_{40:47},\ n)\\
&d_{48:55} \leftarrow ROTL(a_{48:55},\ n)\\
&d_{56:63} \leftarrow ROTL(a_{56:63},\ n)
\end{aligned}
$$

Each of the byte elements of parameter **a** are rotated left by the UIMM value formed from parameter **b** and the results are placed in parameter **d**.

NOTE

Values greater than 7 are illegal for the UIMM value contained in parameter **b**.

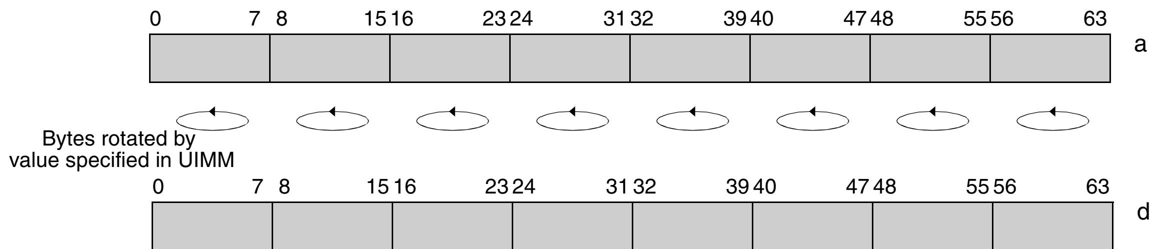


Figure 3-557. Vector Rotate Left Byte Immediate (__ev_rlbi)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	5-bit unsigned literal	**evrlbi d,a,b**

__ev_rlh

__ev_rlh

Vector Rotate Left Half Word

d = __ev_rlh (a,b)

$nh0 \leftarrow b_{12:15}$
$nh1 \leftarrow b_{28:31}$
$nh2 \leftarrow b_{44:47}$
$nh3 \leftarrow b_{60:63}$

$d_{0:15} \leftarrow ROTL(a_{0:15}, nh0)$
$d_{16:31} \leftarrow ROTL(a_{16:31}, nh1)$
$d_{32:47} \leftarrow ROTL(a_{32:47}, nh2)$
$d_{48:63} \leftarrow ROTL(a_{48:63}, nh3)$

Each of the half word elements of parameter **a** are rotated left by an amount specified in the low order four bits or the corresponding half word elements of parameter **b**. The result is placed into parameter **d**. The separate rotate amounts for each element are specified by 4 bits in parameter **b** that lie in bit positions 12-15, 28-31, 44-47 and 60-63.

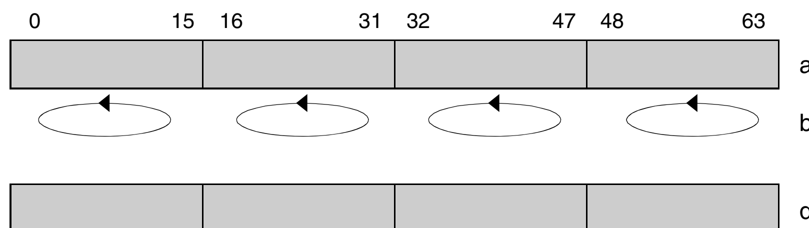


Figure 3-558. Vector Rotate Left Half Word (__ev_rlh)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evrlh d,a,b**

__ev_rlhi

Vector Rotate Left Half Word Immediate

d = __ev_rlhi (a,b)

```
UIMM ←EXTZ(b)
n ←UIMM
d0:15 ←ROTL(a0:15,n)
d16:31 ←ROTL(a16:31,n)
d32:47 ←ROTL(a32:47,n)
d48:63 ←ROTL(a48:63,n)
```

Each of the half word elements of parameter **a** are rotated left by the immediate value specified in UIMM and formed from parameter **b**. The result is placed into parameter **d**.

NOTE

Values greater than 15 are illegal for the UIMM value contained in parameter **b**.

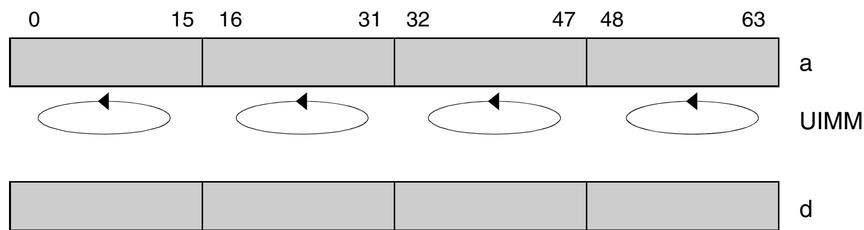


Figure 3-559. Vector Rotate Left Half Word Immediate (evrlhi)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	5-bit unsigned literal	**evrlhi 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].

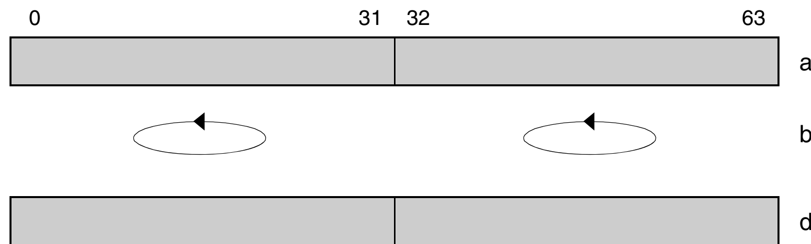


Figure 3-560. Vector Rotate Left Word (__ev_rlw)

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 contained in parameter **b**. The results are placed in parameter **d**.

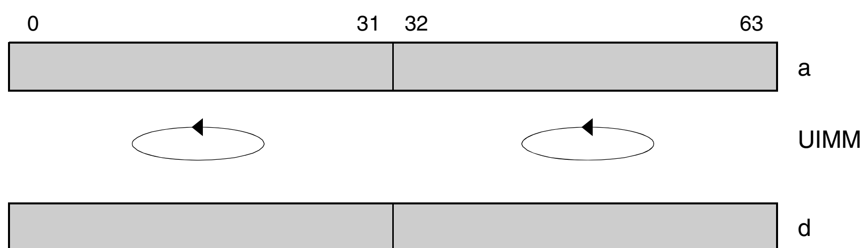


Figure 3-561. Vector Rotate Left Word Immediate (__ev_rlwi)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	5-bit unsigned literal	**evrlwi d,a,b**

__ev_rnddnw __ev_rnddnw

Vector Round Doubleword to Nearest Even Word

d = __ev_rnddnw (a)

```
if (a31:63 = 10 || 0x8000_0000) then temp0:32 ←a0:32 // check for even 0.5
else temp0:32 ← a0:32 + 1 // modulo sum

d0:63 ←temp0:31 || 320
```

The 64-bit value in parameter **a** is rounded into 32 bits using round_to_nearest_even rounding. The result is placed into the most significant 32 bits of parameter **d**, zeroing out the low order 32 bits of parameter **d**.

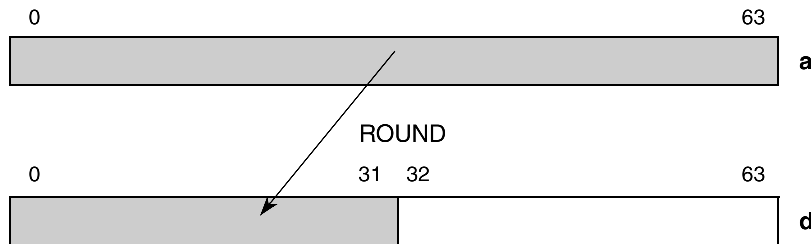


Figure 3-562. Vector Round Doubleword to Nearest Even Word (__ev_rnddnw)

d	a	Maps to
__ev64_opaque__	__ev64_opaque__	**evrnddnw d,a**

__ev_rnddnwss __ev_rnddnwss

Vector Round Doubleword to Nearest Even Word Signed and Saturate

d = __ev_rnddnwss (a)

```
if (a0:63 >=si 0x7FFF_FFFF_8000_0000) then
    temp0:32 ←0x7FFF_FFFF || 10
    ov ←1
else
    ov ←0
    if (a31:63 = 10 || 0x8000_0000) then temp0:32 ←a0:32 // check for even 0.5
    else temp0:32 ← a0:32 + 1 // modulo sum

d0:63 ←temp0:31 || 320
// update SPEFSCR
SPEFSCROVH ← 0; SPEFSCROV ← ov
SPEFSCRSOV ← SPEFSCRSOV | ov
```

The signed 64-bit value in parameter **a** is rounded with saturation into 32 bits using round_to_nearest_even rounding. The 32-bit result is placed into the most significant word element of parameter **d**, zeroing out the low word element. The overflow and summary overflow bits are recorded in the SPEFSCR based on an overflow from the round operation.

Other registers altered: SPEFSCR

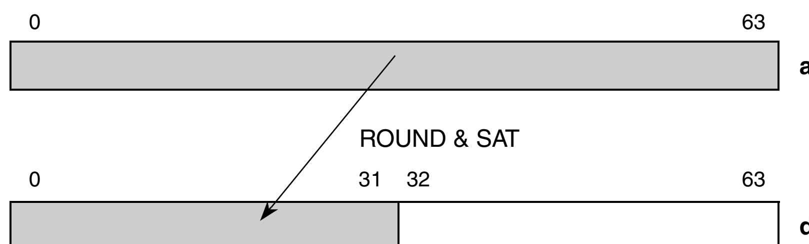


Figure 3-563. Vector Round Doubleword to Nearest Even Word Signed and Saturate (__ev_rnddnwss)

d	a	Maps to
__ev64_opaque__	__ev64_opaque__	**evrnddnwss d,a**

__ev_rnddnwus __ev_rnddnwus

Vector Round Doubleword to Nearest Even Word Unsigned and Saturate

d = __ev_rnddnwus (a)

```
if (a0:63 >=ui 0xFFFF_FFFF_8000_0000) then
    temp0:32 ←0xFFFF_FFFF || 10
    ov ←1
else
    ov ←0
    if (a31:63 = 10 || 0x8000_0000) then temp0:32 ←a0:32 // check for even 0.5
    else temp0:32 ← a0:32 + 1 // modulo sum

d0:63 ←temp0:31 || 320
// update SPEFSCR
SPEFSCROVH ← 0; SPEFSCROV ← ov
SPEFSCRSOV ← SPEFSCRSOV | ov
```

The unsigned 64-bit value in parameter **a** is rounded with saturation into 32 bits using round_to_nearest_even rounding. The 32-bit result is placed into the most significant word element of parameter **d**, zeroing out the low word element. The overflow and summary overflow bits are recorded in the SPEFSCR based on an overflow from the round operation.

Other registers altered: SPEFSCR

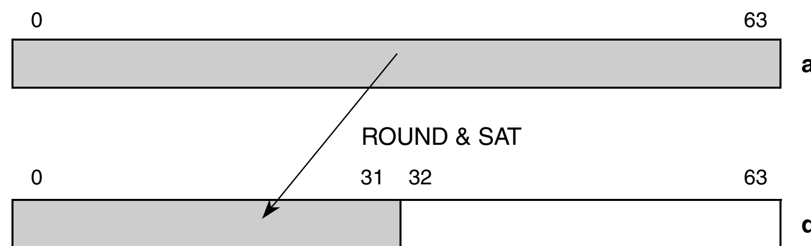


Figure 3-564. Vector Round Doubleword to Nearest Even Word Unsigned and Saturate (__ev_rnddnwus)

d	a	Maps to
__ev64_opaque__	__ev64_opaque__	**evrnddnwus d,a**

__ev_rnddw   __ev_rnddw

Vector Round Doubleword to Word

d = __ev_rnddw (a)

```
d0:63 ←(a0:63+0x0000_0000_8000_0000) & 0xFFFF_FFFF_0000_0000 // Modulo sum
```

The 64-bit value in parameter **a** is rounded into 32 bits. The result is placed into the most significant 32 bits of parameter **d**, zeroing out the low order 32 bits of parameter **d**.

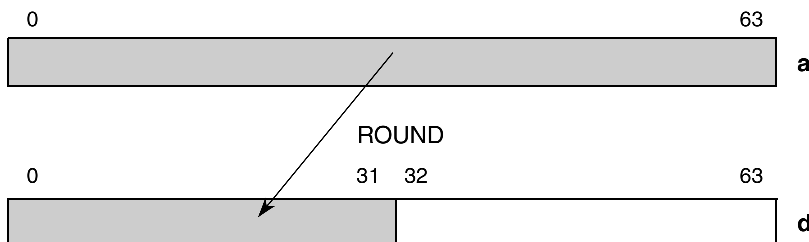


Figure 3-565. Vector Round Doubleword to Word (__ev_rnddw)

d	a	Maps to
__ev64_opaque__	__ev64_opaque__	**evrnddw d,a**

__ev_rnddwss __ev_rnddwss

Vector Round Doubleword to Word Signed and Saturate

d = __ev_rnddwss (a)

```
if (a0:63 >=si 0x7FFF_FFFF_8000_0000) then
    temp0:32 ←0x7FFF_FFFF || 10
    ov ←1
else
    temp0:32 ←(a0:32 + 1)
    ov ←0
d0:63 ←temp0:31 || 320
// update SPEFSCR
SPEFSCROVH ←0; SPEFSCROV ←ov
SPEFSCRSOV ←SPEFSCRSOV | ov
```

The signed 64-bit value in parameter **a** is rounded with saturation into 32 bits. The 32-bit result is placed into the most significant word element of parameter **d**, zeroing out the low word element. The overflow and summary overflow bits are recorded in the SPEFSCR based on an overflow from the round operation.

Other registers altered: SPEFSCR

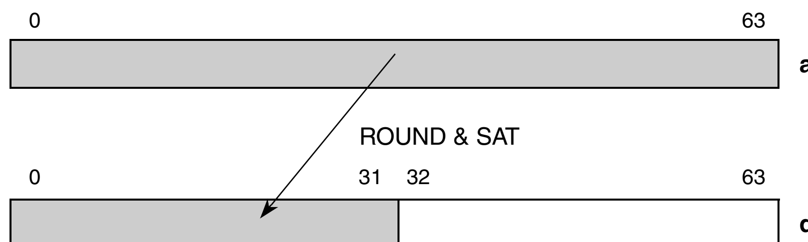


Figure 3-566. Vector Round Doubleword to Word Signed and Saturate (__ev_rnddwss)

d	a	Maps to
__ev64_opaque__	__ev64_opaque__	**evrnddwss d,a**

__ev_rnddwus __ev_rnddwus

Vector Round Doubleword to Word Unsigned and Saturate

d = __ev_rnddwus (a)

```
if (a0:63 >=ui 0xFFFF_FFFF_8000_0000) then
    temp0:32 ←0xFFFF_FFFF || 10
    ov ←1
else
    temp0:32 ←(a0:32 + 1)
    ov ←0

d0:63 ←temp0:31 || 320
// update SPEFSCR
SPEFSCROVH ← 0; SPEFSCROV ← ov
SPEFSCRSOV ← SPEFSCRSOV | ov
```

The unsigned 64-bit value in parameter **a** is rounded with saturation into 32 bits. The 32-bit result is placed into the most significant word element of parameter **d**, zeroing out the low word element. The overflow and summary overflow bits are recorded in the SPEFSCR based on an overflow from the round operation.

Other registers altered: SPEFSCR

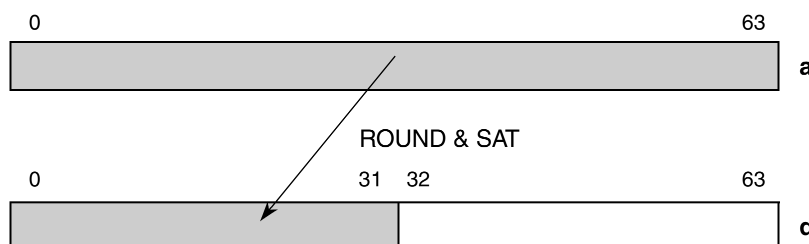


Figure 3-567. Vector Round Doubleword to Word Unsigned and Saturate (__ev_rnddwus)

d	a	Maps to
__ev64_opaque__	__ev64_opaque__	**evrnddwus d,a**

__ev_rndhb

__ev_rndhb

Vector Round Halfword to Byte

d = __ev_rndhb (a)

$d_{0:15} \leftarrow (a_{0:15}+0x0080)\ \&\ 0xFF00$

$d_{16:31} \leftarrow (a_{16:31}+0x0080)\ \&\ 0xFF00$

$d_{32:47} \leftarrow (a_{32:47}+0x0080)\ \&\ 0xFF00$

$d_{48:63} \leftarrow (a_{48:63}+0x0080)\ \&\ 0xFF00$

The 16-bit elements of parameter **a** are rounded into 8 bits. The 8-bit results are placed in the most significant 8 bits of each halfword element of parameter **d**, zeroing out the low order 8 bits of each halfword element.

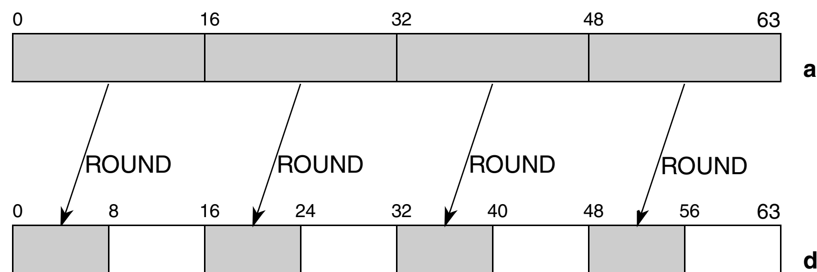


Figure 3-568. Vector Round Halfword to Byte (__ev_rndhb)

d	a	Maps to
__ev64_opaque__	__ev64_opaque__	**evrndhb d,a**

__ev_rndhbss __ev_rndhbss

Vector Round Halfword to Byte Signed and Saturate

d = __ev_rndhbss (a)

```
if (a0:15 >=si 0x7F80) then
    temp00:15 = 0x7F00; ovh0 ←1
else
    temp00:15 = (a0:15+0x0080); ovh0 ←0
d0:15 ←temp00:15 & 0xFF00

if (a16:31 >=si 0x7F80) then
    temp10:15 = 0x7F00; ovh1 ←1
else
    temp10:15 = (a16:31+0x0080); ovh1 ←0
d16:31 ←temp10:15 & 0xFF00

if (a32:47 >=si 0x7F80) then
    temp20:15 = 0x7F00; ovh2 ←1
else
    temp20:15 = (a32:47+0x0080); ovh2 ←0
d32:47 ←temp20:15 & 0xFF00

if (a48:63 >=si 0x7F80) then
    temp30:15 = 0x7F00; ovh3 ←1
else
    temp30:15 = (a48:63+0x0080); ovh3 ←0
d48:63 ←temp30:15 & 0xFF00


ovh ←ovh0 | ovh1
ovl ←ovh2 | ovh3
SPEFSCROVH ←ovh
SPEFSCROV ←ovl
SPEFSCRSOVH ←SPEFSCRSOVH | ovh
SPEFSCRSOV ←SPEFSCRSOV | ovl.
```

The signed 16-bit elements of parameter **a** are rounded with saturation into 8 bits. The 8-bit results are placed in the most significant 8 bits of each halfword element of parameter **d**, zeroing out the low order 8 bits of each halfword element. Any saturation is recorded in the SPEFSCR overflow and summary overflow bits.

Other registers altered: SPEFSCR

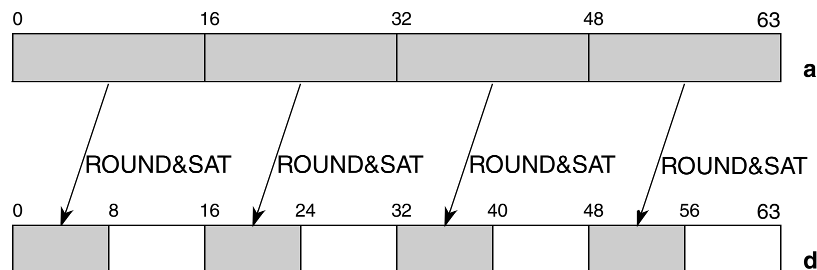


Figure 3-569. Vector Round Halfword to Byte Signed and Saturate (__ev_rndhbss)

d	a	Maps to
__ev64_opaque__	__ev64_opaque__	**evrndhbss d,a**

__ev_rndhbus __ev_rndhbus

Vector Round Half Word to Byte Unsigned and Saturate

d = __ev_rndhbus (a)

```
if (a0:15 >=ui 0xFF80) then
    temp00:15 = 0xFF00; ovh0 ←1
else
    temp00:15 = (a0:15+0x0080); ovh0 ←0
d0:15 ←temp00:15 & 0xFF00

if (a16:31 >=ui 0xFF80) then
    temp10:15 = 0xFF00; ovh1 ←1
else
    temp10:15 = (a16:31+0x0080); ovh1 ←0
d16:31 ←temp10:15 & 0xFF00

if (a32:47 >=ui 0xFF80) then
    temp20:15 = 0xFF00; ovh02 ←1
else
    temp20:15 = (a32:47+0x0080); ovh2 ←0
d32:47 ←temp20:15 & 0xFF00

if (a48:63 >=ui 0xFF80) then
    temp30:15 = 0xFF00; ovh3 ←1
else
    temp30:15 = (a48:63+0x0080); ovh3 ←0
d48:63 ←temp30:15 & 0xFF00

ovh ←ovh0 | ovh1
ovl ←ovh2 | ovh3
SPEFSCROVH ←ovh
SPEFSCROV ←ovl
SPEFSCRSOVH ←SPEFSCRSOVH | ovh
SPEFSCRSOV ←SPEFSCRSOV | ovl
```

The unsigned 16-bit elements of parameter **a** are rounded with saturation into 8 bits. The 8-bit results are placed in the most significant 8 bits of each halfword element of parameter **d**, zeroing out the low order 8 bits of each halfword element. Any saturation is recorded in the SPEFSCR overflow and summary overflow bits.

Other registers altered: SPEFSCR

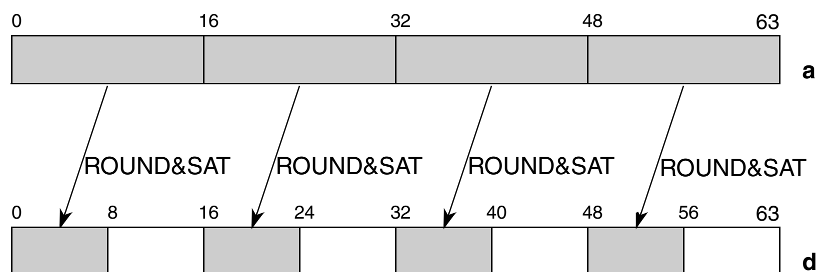


Figure 3-570. Vector Round Half Word to Byte Unsigned and Saturate (__ev_rndhbus)

d	a	Maps to
__ev64_opaque__	__ev64_opaque__	**evrndhbus d,a**

__ev_rndhnb __ev_rndhnb

Vector Round Halfword to Nearest Even Byte

d = __ev_rndhnb (a)

```
if (a7:15 = 10 || 0x80) then temp0:8 ←a0:8 // check for even 0.5
else temp0:8 ←a0:8 + 1 // modulo sum
d0:15 ←temp0:7 || 80

if (a23:31 = 10 || 0x80) then temp0:8 ←a16:24 // check for even 0.5
else temp0:8 ← a16:24 + 1 // modulo sum
d16:31 ←temp0:7 || 80

if (a39:47 = 10 || 0x80) then temp0:8 ←a32:40 // check for even 0.5
else temp0:8 ← a32:40 + 1 // modulo sum
d32:47 ←temp0:7 || 80

if (a55:63 = 10 || 0x80) then temp0:8 ←a48:56 // check for even 0.5
else temp0:8 ← a48:56 + 1 // modulo sum
d48:63 ←temp0:7 || 80
```

The 16-bit elements of parameter **a** are rounded into 8 bits using round_to_nearest_even rounding. The 8-bit results are placed in the most significant 8 bits of each halfword element of parameter **d**, zeroing out the low order 8 bits of each halfword element.

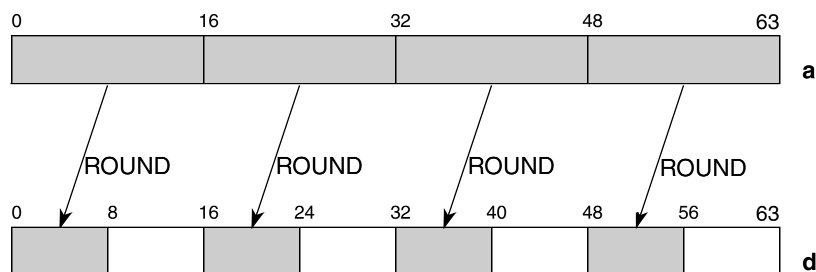


Figure 3-571. Vector Round Halfword to Nearest Even Byte (__ev_rndhnb)

d	a	Maps to
__ev64_opaque__	__ev64_opaque__	**evrndhnb d,a**

__ev_rndhnbss __ev_rndhnbss

Vector Round Halfword to Nearest Even Byte Signed and Saturate

d = __ev_rndhnbss (a)

```
if (a0:15 >=si 0x7F80) then temp0:8 = (0x7F || 10); ovh0 ←1
else
    ovh0 ←0
    if (a7:15 = 10 || 0x80) then temp0:8 ←a0:8 // check for even 0.5
    else temp0:8 ←a0:8 + 1 // modulo sum
d0:15 ←temp0:7 || 80

if (a16:31 >=si 0x7F80) then temp0:8 = (0x7F || 10); ovh1 ←1
else
    ovh1 ←0
    if (a23:31 = 10 || 0x80) then temp0:8 ←a16:24 // check for even 0.5
    else temp0:8 ←a16:24 + 1 // modulo sum
d16:31 ←temp0:7 || 80

if (a32:47 >=si 0x7F80) then temp0:8 = (0x7F || 10); ovh2 ←1
else
    ovh2 ←0
    if (a39:47 = 10 || 0x80) then temp0:8 ←a32:40 // check for even 0.5
    else temp0:8 ←a32:40 + 1 // modulo sum
d32:47 ←temp0:7 || 80

if (a48:63 >=si 0x7F80) then temp0:8 = (0x7F || 10); ovh3 ←1
else
    ovh3 ←0
    if (a55:63 = 10 || 0x80) then temp0:8 ←a48:56 // check for even 0.5
    else temp0:8 ←a48:56 + 1 // modulo sum
d48:63 ←temp0:7 || 80

ovh ←ovh0 | ovh1; ovl ←ovh2 | ovh3
SPEFSCROVH ←ovh; SPEFSCROV ←ovl
SPEFSCRSOVH ←SPEFSCRSOVH | ovh; SPEFSCRSOV ←SPEFSCRSOV | ovl
```

The signed 16-bit elements of parameter **a** are rounded with saturation into 8 bits using round_to_nearest_even rounding. The 8-bit results are placed in the most significant 8 bits of each halfword element of parameter **d**, zeroing out the low order 8 bits of each halfword element. Any saturation is recorded in the SPEFSCR overflow and summary overflow bits.

Other registers altered: SPEFSCR

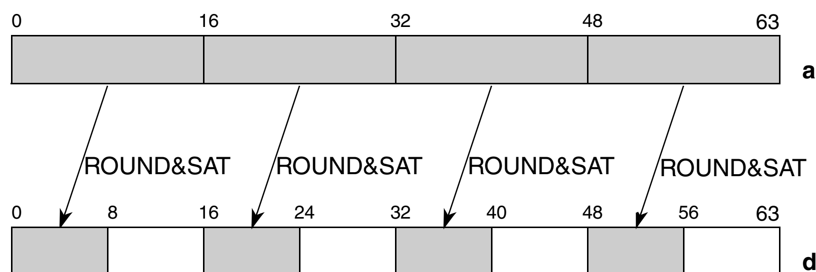


Figure 3-572. Vector Round Halfword to Nearest Even Byte Signed and Saturate (__ev_rndhnbss)

d	a	Maps to
__ev64_opaque__	__ev64_opaque__	**evrndhnbss d,a**

__ev_rndhnbus __ev_rndhnbus

Vector Round Halfword to Nearest Byte Unsigned and Saturate

d = __ev_rndhnbus (a)

```
if (a0:15 >=ui 0xFF80) then temp0:8 = (0xFF || 10); ovh0 ←1
else
    ovh0 ←0
    if (a7:15 = 10 || 0x80) then temp0:8 ←a0:8 // check for even 0.5
    else temp0:8 ←a0:8 + 1 // modulo sum
d0:15 ←temp0:7 || 80

if (a16:31 >=ui 0xFF80) then temp0:8 = (0xFF || 10); ovh1 ←1
else
    ovh1 ←0
    if (a23:31 = 10 || 0x80) then temp0:8 ←a16:24 // check for even 0.5
    else temp0:8 ← a16:24 + 1 // modulo sum
d16:31 ←temp0:7 || 80

if (a32:47 >=ui 0xFF80) then temp0:8 = (0xFF || 10); ovh2 ←1
else
    ovh2 ←0
    if (a39:47 = 10 || 0x80) then temp0:8 ←a32:40 // check for even 0.5
    else temp0:8 ← a32:40 + 1 // modulo sum
d32:47 ←temp0:7 || 80

if (a48:63 >=ui 0xFF80) then temp0:8 = (0xFF || 10); ovh3 ←1
else
    ovh3 ←0
    if (a55:63 = 10 || 0x80) then temp0:8 ←a48:56 // check for even 0.5
    else temp0:8 ← a48:56 + 1 // modulo sum
d48:63 ←temp0:7 || 80


ovh ←ovh0 | ovh1; ovl ←ovh2 | ovh3
SPEFSCROVH ←ovh; SPEFSCROV ←ovl
SPEFSCRSOVH ←SPEFSCRSOVH | ovh; SPEFSCRSOV ←SPEFSCRSOV | ovl
```

The unsigned 16-bit elements of parameter **a** are rounded with saturation into 8 bits using round_to_nearest_even rounding. The 8-bit results are placed in the most significant 8 bits of each halfword element of parameter **d**, zeroing out the low order 8 bits of each halfword element. Any saturation is recorded in the SPEFSCR overflow and summary overflow bits.

Other registers altered: SPEFSCR

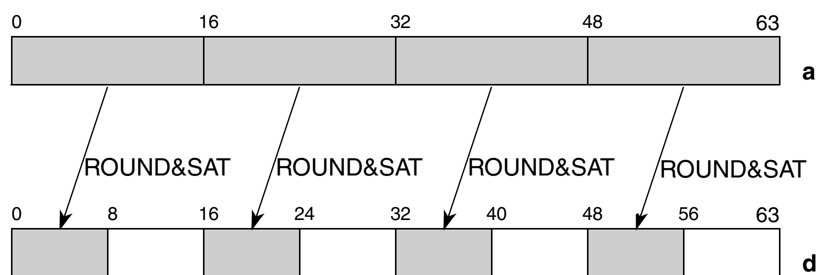


Figure 3-573. Vector Round Halfword to Nearest Even Byte Unsigned and Saturate (__ev_rndhnbus)

d	a	Maps to
__ev64_opaque__	__ev64_opaque__	**evrndhnbus d,a**

__ev_rndwh

__ev_rndwh

Vector Round Word to Halfword

d = __ev_rndw (a)

d = __ev_rndwh (a)

$d_{0:31} \leftarrow (a_{0:31}$ + 0x0000_8000) & 0xFFFF_0000 // Modulo sum
$d_{32:63} \leftarrow (a_{32:63}$ + 0x0000_8000) & 0xFFFF_0000 // 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.

Note: __ev_rndw and __ev_rndwh are the same instruction

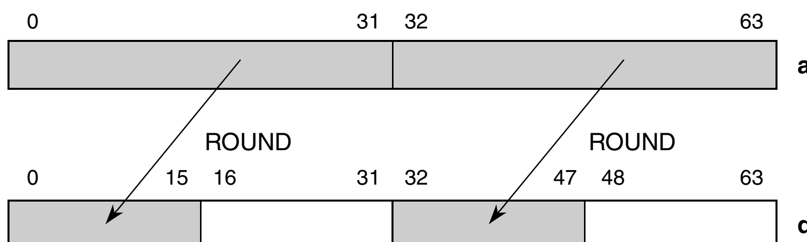


Figure 3-574. Vector Round Word to Halfword (__ev_rndwh(__ev_rndw))

d	a	Maps to
__ev64_opaque__	__ev64_opaque__	**evrnd d,a**
__ev64_opaque__	__ev64_opaque__	**evrndwh d,a**

__ev_rndwhss

Vector Round Word to Halfword Signed and Saturate

d = __ev_rndwhss (a)

```
if (a0:31 >=si 0x7FFF_8000) then
    temph0:31 = 0x7FFF_0000; ovh ←1
else
    temph0:31 = (a0:31 + 0x0000_8000); ovh ←0
d0:31 ←temph0:31 & 0xFFFF_0000

if (a32:63 >=si 0x7FFF_8000) then
    templ0:31 = 0x7FFF_0000; ovl ←1
else
    templ0:31 = (a32:63 + 0x0000_8000); ovl ←0
d32:63 ←templ0:31 & 0xFFFF_0000


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

The signed 32-bit elements of parameter **a** are rounded with saturation into 16 bits. The 16-bit results are placed in the most significant 16 bits of each word element of parameter **d**, zeroing out the low order 16 bits of each word element. Any saturation is recorded in the SPEFSCR overflow and summary overflow bits.

Other registers altered: SPEFSCR

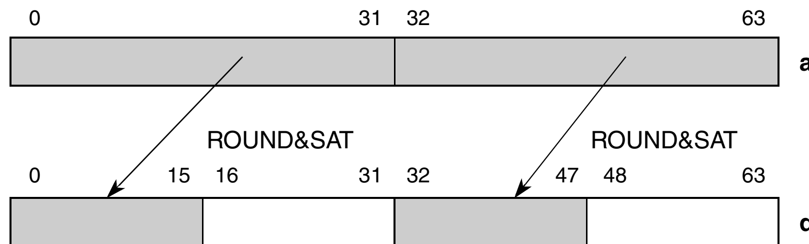


Figure 3-575. Vector Round Word to Halfword Signed and Saturate (__ev_rndwhss)

d	a	Maps to
__ev64_opaque__	__ev64_opaque__	**evrndwhss d,a**

__ev_rndwhus __ev_rndwhus

Vector Round Word to Halfword Unsigned and Saturate

d = __ev_rndwhus (a)

```
if (a0:31 >=ui 0xFFFF_8000) then
    temph0:31 = 0xFFFF_0000; ovh ←1
else
    temph0:31 = (a0:31 + 0x0000_8000); ovh ←0
d0:31 ←temph0:31 & 0xFFFF_0000

if (a32:63 >=ui 0xFFFF_8000) then
    templ0:31 = 0xFFFF_0000; ovl ←1
else
    templ0:31 = (a32:63 + 0x0000_8000); ovl ←0
d32:63 ←templ0:31 & 0xFFFF_0000

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

The unsigned 32-bit elements of parameter **a** are rounded with saturation into 16 bits. The 16-bit results are placed in the most significant 16 bits of each word element of parameter **d**, zeroing out the low order 16 bits of each word element. Any saturation is recorded in the SPEFSCR overflow and summary overflow bits.

Other registers altered: SPEFSCR

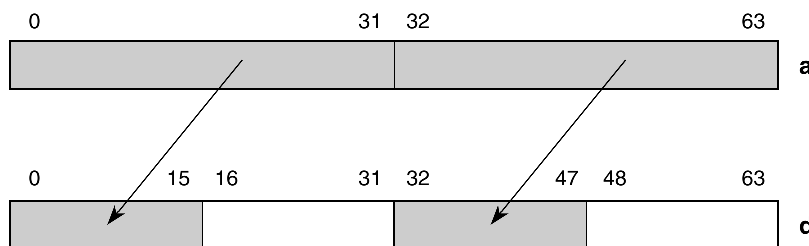


Figure 3-576. Vector Round Word to Halfword Unsigned and Saturate (__ev_rndwhus)

d	a	Maps to
__ev64_opaque__	__ev64_opaque__	**evrndwhus d,a**

__ev_rndwnh

__ev_rndwnh

Vector Round Word to Nearest Even Halfword

d = __ev_rndwnh (a)

```
if (a15:31 = 10 || 0x8000) then temp0:16 ←a0:16 // check for even 0.5
else temp0:16 ← a0:16 + 1 // modulo sum

d0:31 ←temp0:15 || 160

if (a47:63 = 10 || 0x8000) then temp0:16 ←a32:48 // check for even 0.5
else temp0:16 ← a32:48 + 1 // modulo sum

d32:63 ←temp0:15 || 160
```

The 32-bit elements of parameter **a** are rounded into 16 bits using round_to_nearest_even rounding. 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.

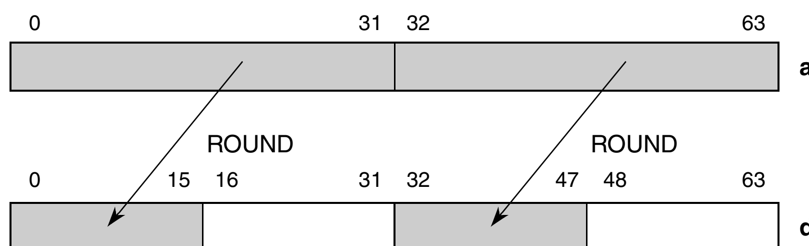


Figure 3-577. Vector Round Word to Nearest Even Halfword (__ev_rndwnh)

d	a	Maps to
__ev64_opaque__	__ev64_opaque__	**evrndwnh d,a**

__ev_rndwnhss __ev_rndwnhss

Vector Round Word to Nearest Even Halfword Signed and Saturate

d = __ev_rndwnhss (a)

```
if (a0:31 >=si 0x7FFF_8000) then
    temph0:16 = 0x7FFF || 10; ovh ←1
else
    ovh ←0
    if (a15:31 = 10 || 0x8000) then temph0:16 ←a0:16 // check for even 0.5
    else temph0:16 ←a0:16 + 1 // modulo sum
d0:31 ←temph0:15 || 160

if (a32:63 >=si 0x7FFF_8000) then
    templ0:16 = 0x7FFF || 10; ovl ←1
else
    ovl ←0
    if (a47:63 = 10 || 0x8000) then templ0:16 ←a32:48 // check for even 0.5
    else templ0:16 ←a32:48 + 1 // modulo sum
d32:63 ←templ0:15 || 160


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

The signed 32-bit elements of parameter **a** are rounded with saturation into 16 bits using round_to_nearest_even rounding. The 16-bit results are placed in the most significant 16 bits of each word element of parameter **d**, zeroing out the low order 16 bits of each word element. Any saturation is recorded in the SPEFSCR overflow and summary overflow bits.

Other registers altered: SPEFSCR

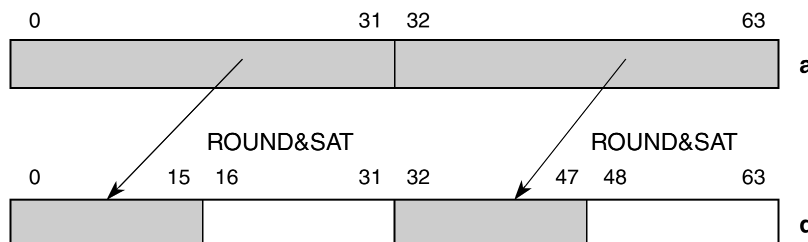


Figure 3-578. Vector Round Word to Nearest Even Halfword Signed and Saturate (__ev_rndwnhss)

d	a	Maps to
__ev64_opaque__	__ev64_opaque__	**evrndwnhss d,a**

__ev_rndwnhus __ev_rndwnhus

Vector Round Word to Nearest Even Halfword Unsigned and Saturate

d = __ev_rndwnhus (a)

```
if (a0:31 >=ui 0xFFFF_8000) then
    temph0:16 = 0xFFFF || 10; ovh ←1
else
    ovh ←0
    if (a15:31 = 10 || 0x8000) then temph0:16 ←a0:16 // check for even 0.5
    else temph0:16 ←a0:16 + 1 // modulo sum
d0:31 ←temph0:15 || 160

if (a32:63 >=ui 0xFFFF_8000) then
    templ0:16 = 0xFFFF || 10; ovl ←1
else
    ovl ←0
    if (a47:63 = 10 || 0x8000) then templ0:16 ←a32:48 // check for even 0.5
    else templ0:16 ←a32:48 + 1 // modulo sum
d32:63 ←templ0:15 || 160


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

The unsigned 32-bit elements of parameter **a** are rounded with saturation into 16 bits. The 16-bit results are placed in the most significant 16 bits of each word element of parameter **d**, zeroing out the low order 16 bits of each word element. Any saturation is recorded in the SPEFSCR overflow and summary overflow bits.

Other registers altered: SPEFSCR

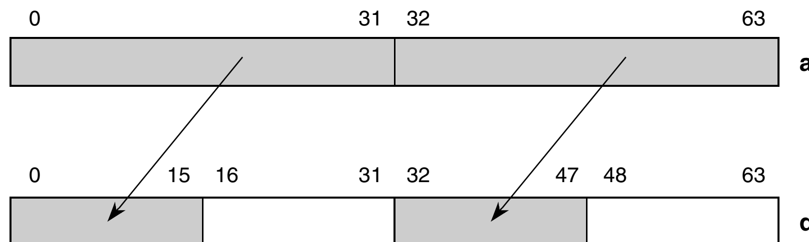


Figure 3-579. Vector Round Word to Nearest Even Halfword Unsigned and Saturate (__ev_rndwnhus)

d	a	Maps to
__ev64_opaque__	__ev64_opaque__	**evrndwnhus d,a**

__ev_sad2sh[a] __ev_sad2sh[a]

Vector Sum of Absolute Differences of 2 Signed Half Words (to Accumulator)

d = __ev_sad2sh (a,b) (A = 0)

d = __ev_sad2sha (a,b) (A = 1)

```
temp0[0:15] ←ABS(b[0:15] - a[0:15])
temp1[0:15] ←ABS(b[16:31] - a[16:31])
temp2[0:15] ←ABS(b[32:47] - a[32:47])
temp3[0:15] ←ABS(b[48:63] - a[48:63])

d[0:31] ←EXTZ(temp0[0:15]) + EXTZ(temp1[0:15])
d[32:63] ←EXTZ(temp2[0:15]) + EXTZ(temp3[0:15])

// update accumulator
if A = 1 then ACC[0:63] ←d[0:63]
```

Groups of two signed half word elements of parmeter a are subtracted from the corresponding elements of parameter **b**, the absolute values of the 17-bit differences from each group are summed together, and the results are placed into the word elements 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 3-580. Vector Sum of Absolute Differences of 2 Signed Half Words (to Accumulator) (__ev_sad2sh[a])

A	d	a	b	Maps to
A = 0	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evsad2sh d,a,b**
A = 1	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evsad2sha d,a,b**

__ev_sad2shaaw __ev_sad2shaaw

Vector Sum of Absolute Differences of 2 Signed Half Words and Accumulate into Words

d = __ev_sad2shaaw (a,b)

```
temp0[0:15] ←ABS(b[0:15] - a[0:15])
temp1[0:15] ←ABS(b[16:31] - a[16:31])
temp2[0:15] ←ABS(b[32:47] - a[32:47])
temp3[0:15] ←ABS(b[48:63] - a[48:63])

d[0:31] ←ACC[0:31] + EXTZ(temp0[0:15]) + EXTZ(temp1[0:15])
d[32:63] ←ACC[32:63] + EXTZ(temp2[0:15]) + EXTZ(temp3[0:15])

// update accumulator
ACC[0:63] ←d[0:63]
```

Groups of two signed half word elements of parameter **a** are subtracted from the corresponding elements of parameter **b**, the absolute values of the 17-bit differences from each group are summed together, then added to the corresponding word elements in the accumulator, and the results are placed into the word elements of parameter **d** and the accumulator.

Other registers altered: ACC

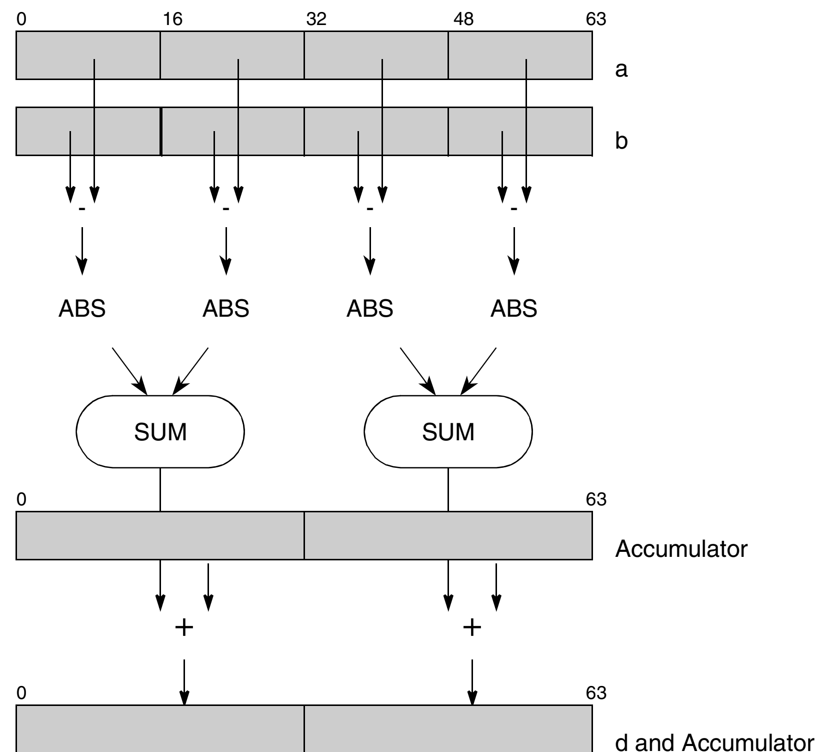


Figure 3-581. Vector Sum of Absolute Differences of 2 Signed Half Words and Accumulate (__ev_sad2shaaw)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evsad2shaaw d,a,b**

__ev_sad2uh[a] __ev_sad2uh[a]

Vector Sum of Absolute Differences of 2 Unsigned Half Words (to Accumulator)

d = __ev_sad2uh (a,b) (A = 0)

d = __ev_sad2uha (a,b) (A = 1)

```
temp0[0:15] ←ABS(b[0:15] - a[0:15])
temp1[0:15] ←ABS(b[16:31] - a[16:31])
temp2[0:15] ←ABS(b[32:47] - a[32:47])
temp3[0:15] ←ABS(b[48:63] - a[48:63])

d[0:31] ←EXTZ(temp0[0:15]) + EXTZ(temp1[0:15])
d[32:63] ←EXTZ(temp2[0:15]) + EXTZ(temp3[0:15])

// update accumulator
if A = 1 then ACC[0:63] ←d[0:63]
```

Groups of two unsigned half word elements of parameter **a** are subtracted from the corresponding elements of parameter **b**, the absolute values of the 16-bit differences from each group are summed together, and the results are placed into the word elements 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 3-582. Vector Sum of Absolute Differences of 2 Unsigned Half Words (to Accumulator) (__ev_sad2uh[a])

A	d	a	b	Maps to
A = 0	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evsad2uh d,a,b**
A = 1	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evsad2uha d,a,b**

__ev_sad2uhaaw __ev_sad2uhaaw

Vector Sum of Absolute Differences of 2 Unsigned Half Words and Accumulate into Words

d = __ev_sad2uhaaw (a,b)

```
temp0[0:15] ←ABS(b[0:15] - a[0:15])
temp1[0:15] ←ABS(b[16:31] - a[16:31])
temp2[0:15] ←ABS(b[32:47] - a[32:47])
temp3[0:15] ←ABS(b[48:63] - a[48:63])

d[0:31] ←ACC[0:31] + EXTZ(temp0[0:15]) + EXTZ(temp1[0:15])
d[32:63] ←ACC[32:63] + EXTZ(temp2[0:15]) + EXTZ(temp3[0:15])

// update accumulator
ACC[0:63] ←d[0:63]
```

Groups of two unsigned half word elements of parameter **a** are subtracted from the corresponding elements of parameter **b**, the absolute values of the 16-bit differences from each group are summed together, then added to the corresponding word elements in the accumulator, and the results are placed into the word elements of parameter **d** and the accumulator.

Other registers altered: ACC

Figure 3-583. Vector Sum of Absolute Differences of 2 Unsigned Half Words and Accumulate (__ev_sad2uhaaw)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evsad2uhaaw d,a,b**

__ev_sad4sb[a] __ev_sad4sb[a]

Vector Sum of Absolute Differences of 4 Signed Bytes (to Accumulator)

d = __ev_sad4sb (a,b) (A = 0)

d = __ev_sad4sba (a,b) (A = 1)

```
temp0[0:7] ←ABS(b[0:7] - a[0:7])
temp1[0:7] ←ABS(b[8:15] - a[8:15])
temp2[0:7] ←ABS(b[16:23] - a[16:23])
temp3[0:7] ←ABS(b[24:31] - a[24:31])
temp4[0:7] ←ABS(b[32:39] - a[32:39])
temp5[0:7] ←ABS(b[40:47] - a[40:47])
temp6[0:7] ←ABS(b[48:55] - a[48:55])
temp7[0:7] ←ABS(b[56:63] - a[56:63])
d[0:31] ←EXTZ(temp0[0:7]) + EXTZ(temp1[0:7]) + EXTZ(temp2[0:7]) + EXTZ(temp3[0:7])
d[32:63] ←EXTZ(temp4[0:7]) + EXTZ(temp5[0:7]) + EXTZ(temp6[0:7]) + EXTZ(temp7[0:7])
// update accumulator
if A = 1 then ACC[0:63] ←d[0:63]
```

Groups of four signed byte elements of parameter **a** are subtracted from the corresponding elements of parameter **b**, the absolute values of the 8-bit differences from each group are summed together, and the results are placed into the word elements 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 3-584. Vector Sum of Absolute Differences of 4 Signed Bytes (to Accumulator) (__ev_sad4sb[a])

A	d	a	b	Maps to
A = 0	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evsad4sb d,a,b**
A = 1	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evsad4sba d,a,b**

__ev_sad4sbaaw

__ev_sad4sbaaw

Vector Sum of Absolute Differences of 4 Signed Bytes and Accumulate into Words

d = __ev_sad4sbaaw (a,b)

```
temp0[0:7] ←ABS(b[0:7] - a[0:7])
temp1[0:7] ←ABS(b[8:15] - a[8:15])
temp2[0:7] ←ABS(b[16:23] - a[16:23])
temp3[0:7] ←ABS(b[24:31] - a[24:31])
temp4[0:7] ←ABS(b[32:39] - a[32:39])
temp5[0:7] ←ABS(b[40:47] - a[40:47])
temp6[0:7] ←ABS(b[48:55] - a[48:55])
temp7[0:7] ←ABS(b[56:63] - a[56:63])
d[0:31] ←ACC[0:31] + EXTZ(temp0[0:7]) + EXTZ(temp1[0:7]) + EXTZ(temp2[0:7]) + EXTZ(temp3[0:7])
d[32:63] ←ACC[32:63] + EXTZ(temp4[0:7]) + EXTZ(temp5[0:7]) + EXTZ(temp6[0:7]) + EXTZ(temp7[0:7])
// update accumulator
ACC[0:63] ←d[0:63]
```

Groups of four signed byte elements of parameter **a** are subtracted from the corresponding elements of parameter **b**, the absolute values of the 8-bit differences from each group are summed together, then added to the respective word elements of the accumulator, and the result is placed in parameter **d** and the accumulator.

Other registers altered: ACC

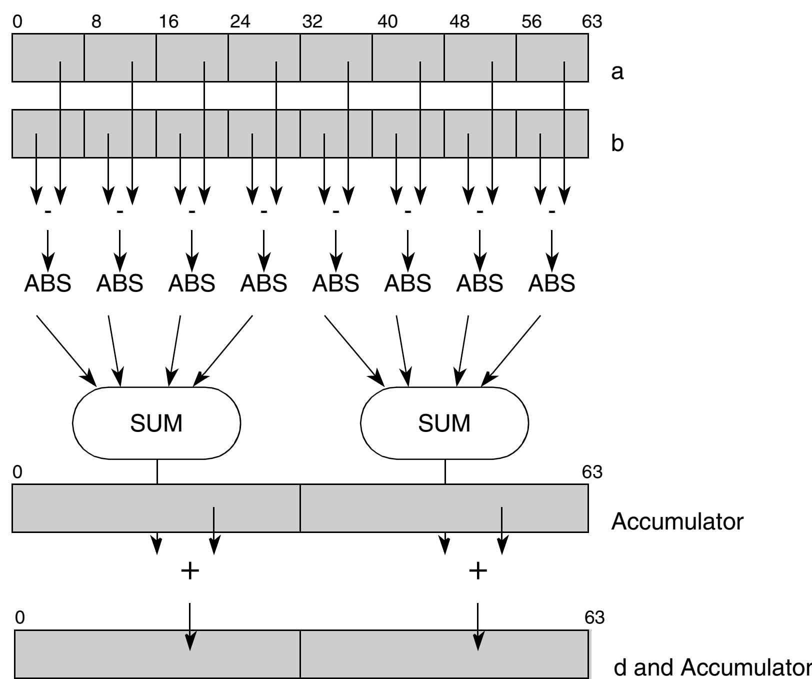


Figure 3-585. Vector Sum of Absolute Differences of 4 Signed Bytes and Accumulate Words (__ev_sad4sbaaw)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evsad4sbaaw d,a,b**

__ev_sad4ub[a] __ev_sad4ub[a]

Vector Sum of Absolute Differences of 4 Unsigned Bytes (to Accumulator)

d = __ev_sad4ub (a,b) (A = 0)

d = __ev_sad4uba (a,b) (A = 1)

```
temp0[0:7] ←ABS(b[0:7] - a[0:7])
temp1[0:7] ←ABS(b[8:15] - a[8:15])
temp2[0:7] ←ABS(b[16:23] - a[16:23])
temp3[0:7] ←ABS(b[24:31] - raA[24:31])
temp4[0:7] ←ABS(b[32:39] - a[32:39])
temp5[0:7] ←ABS(b[40:47] - a[40:47])
temp6[0:7] ←ABS(b[48:55] - a[48:55])
temp7[0:7] ←ABS(b[56:63] - a[56:63])
d[0:31] ←EXTZ(temp0[0:7]) + EXTZ(temp1[0:7]) + EXTZ(temp2[0:7]) + EXTZ(temp3[0:7])
d[32:63] ←EXTZ(temp4[0:7]) + EXTZ(temp5[0:7]) + EXTZ(temp6[0:7]) + EXTZ(temp7[0:7])
// update accumulator
if A = 1 then ACC[0:63] ←d[0:63]
```

Groups of four signed byte elements of parameter **a** are subtracted from the corresponding elements of parameter **b**, the absolute values of the 9-bit differences from each group are summed together, and the results are placed into the word elements of parameter **d**. If A = 1, the result in parameter **d** is also placed into the accumulator.

Other registers altered: ACC (if A=1)

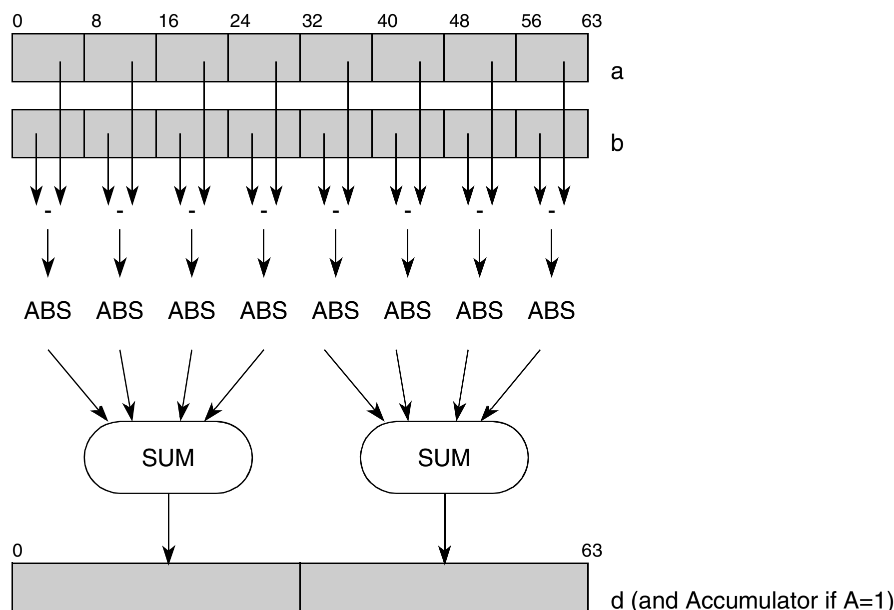


Figure 3-586. Vector Sum of Absolute Differences of 4 Unsigned Bytes (to Accumulator (__ev_sad4ub[a])

A	d	a	b	Maps to
A = 0	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evsad4ub d,a,b**
A = 1	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evsad4uba d,a,b**

__ev_sad4ubaaw __ev_sad4ubaaw

Vector Sum of Absolute Differences of 4 Unsigned Bytes and Accumulate into Words

d = __ev_sad4ubaaw (a,b)

```
temp0[0:7] ←ABS(b[0:7] - a[0:7])
temp1[0:7] ←ABS(b[8:15] - a[8:15])
temp2[0:7] ←ABS(b[16:23] - a[16:23])
temp3[0:7] ←ABS(b[24:31] - a[24:31])
temp4[0:7] ←ABS(b[32:39] - a[32:39])
temp5[0:7] ←ABS(b[40:47] - a[40:47])
temp6[0:7] ←ABS(b[48:55] - a[48:55])
temp7[0:7] ←ABS(b[56:63] - a[56:63])
d[0:31] ←ACC[0:31] + EXTZ(temp0[0:7]) + EXTZ(temp1[0:7]) + EXTZ(temp2[0:7]) + EXTZ(temp3[0:7])
d[32:63] ←ACC[32:63] + EXTZ(temp4[0:7]) + EXTZ(temp5[0:7]) + EXTZ(temp6[0:7]) + EXTZ(temp7[0:7])
// update accumulator
ACC[0:63] ←d[0:63]
```

Groups of four unsigned byte elements of parameter **a** are subtracted from the corresponding elements of parameter **b**, the absolute values of the 9-bit differences from each group are summed together, then added to the respective word elements of the accumulator, and the result is placed in parameter **d** and the accumulator.

Other registers altered: ACC

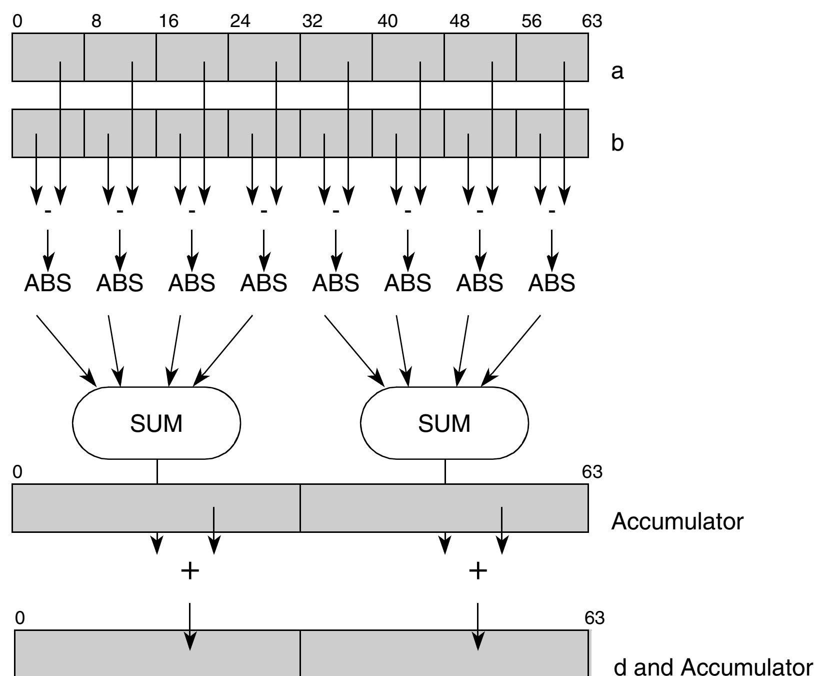


Figure 3-587. Vector Sum of Absolute Differences of 4 Unsigned Bytes and Accumulate Words (__ev_sad4ubaaw)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evsad4ubaaw d,a,b**

__ev_sadsw[a] __ev_sadsw[a]

Vector Sum of Absolute Differences of Signed Words (to Accumulator)

d = __ev_sadsw (a,b) (A = 0)

d = __ev_sadswa (a,b) (A = 1)

```
temph0:31 ←ABS(b0:31 - a0:31)
templ0:31 ←ABS(b32:63 - a32:63)

d0:63 ←EXTZ(temph0:31) + EXTZ(temph0:31)

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

The signed word elements of parameter **a** are subtracted from the corresponding elements of parameter **b**, the absolute values of the 32-bit differences are summed together, and the result is placed in parameter **d**. If A = 1, the result in parameter **d** is also placed into the accumulator.

Other registers altered: ACC (if A=1)

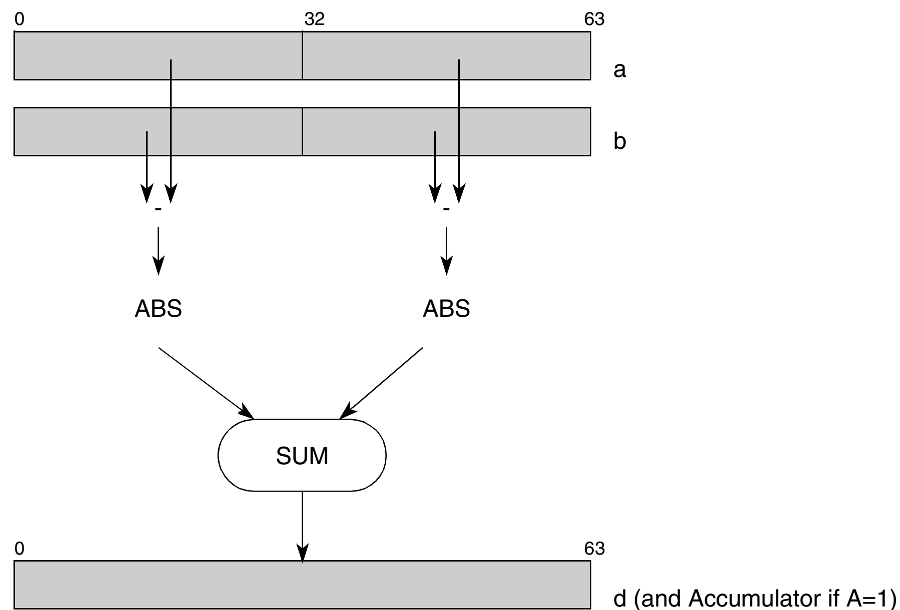


Figure 3-588. Vector Sum of Absolute Differences of Signed Words (to Accumulator) (__ev_sadsw[a])

A	d	a	b	Maps to
A = 0	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evsadsw d,a,b**
A = 1	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evsadswa d,a,b**

__ev_sadswaa __ev_sadswaa

Vector Sum of Absolute Differences of Signed Words and Accumulate

d = __ev_sadswaa (a,b)

```
temph0:31 ←ABS(b0:31 - a0:31)
templ0:31 ←ABS(b32:63 - a32:63)

d0:63 ←EXTZ(temph0:31) + EXTZ(temph0:31) + ACC0:63

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

The signed word elements of parameter **a** are subtracted from the corresponding elements of parameter **b**, the absolute values of the 32-bit differences are summed together then added to the contents of the accumulator, and the result is placed in parameter **d** and the accumulator.

Other registers altered: ACC

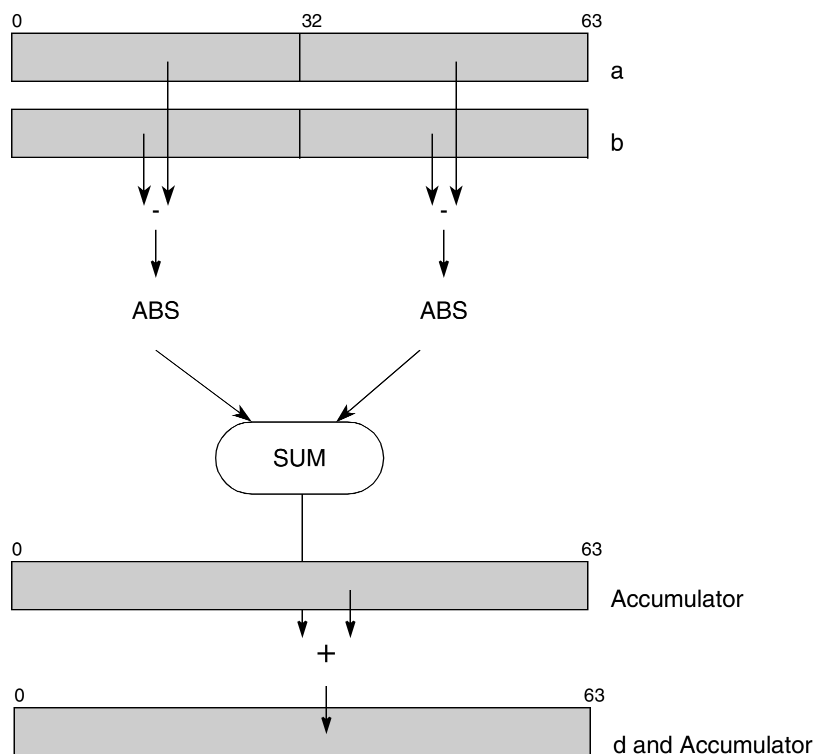


Figure 3-589. Vector Sum of Absolute Differences of Signed Words and Accumulate (__ev_sadswaa)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evsadswaa d,a,b**

__ev_saduw[a] __ev_saduw[a]

Vector Sum of Absolute Differences of Unsigned Words (to Accumulator)

d = __ev_saduw (a,b) (A = 0)

d = __ev_saduwa (a,b) (A = 1)

```
temph0:31 ←ABS(b0:31 - a0:31)
templ0:31 ←ABS(b32:63 - a32:63)

d0:63 ←EXTZ(temph0:31) + EXTZ(temph0:31)

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

The unsigned word elements of parameter **a** are subtracted from the corresponding elements of parameter **b**, the absolute values of the 33-bit differences are summed together, and the result is placed in parameter **d**. If A = 1, the result in parameter **d** is also placed into the accumulator.

Other registers altered: ACC (if A=1)

Figure 3-590. Vector Sum of Absolute Differences of Unsigned Words (to Accumulator) (__ev_saduw[a])

A	d	a	b	Maps to
A = 0	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evsaduw d,a,b**
A = 1	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evsaduwa d,a,b**

__ev_saduwaa __ev_saduwaa

Vector Sum of Absolute Differences of Unsigned Words and Accumulate

d = __ev_saduwaa (a,b)

```
temph0:31 ←ABS(b0:31 - a0:31)
templ0:31 ←ABS(b32:63 - a32:63)

d0:63 ←EXTZ(temph0:31) + EXTZ(temph0:31) + ACC0:63

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

The unsigned word elements of parameter **a** are subtracted from the corresponding elements of parameter **b**, the absolute values of the 33-bit differences are summed together then added to the contents of the accumulator, and the result is placed in parameter **d** and the accumulator.

Other registers altered: ACC

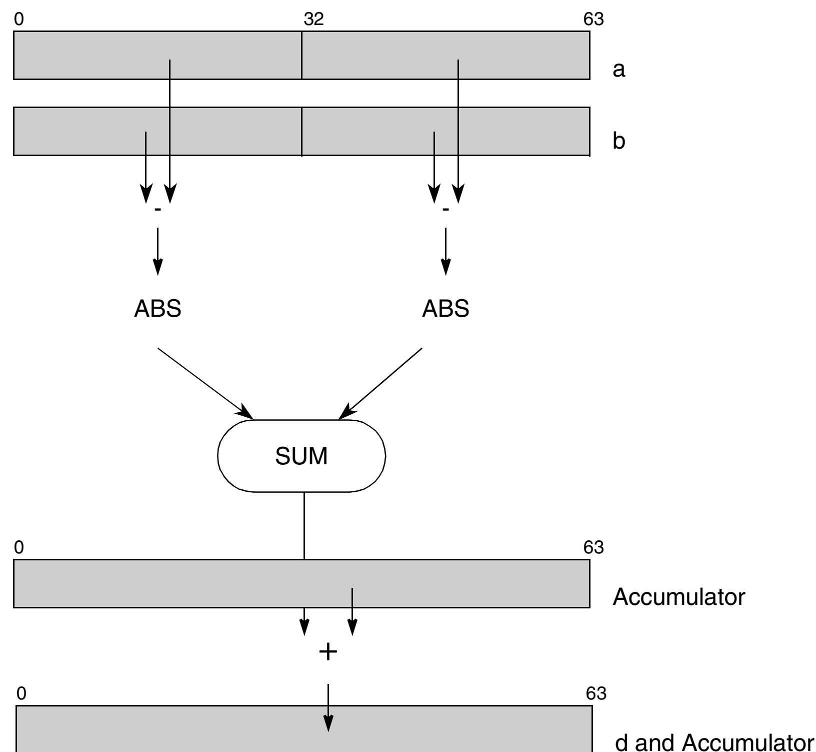


Figure 3-591. Vector Sum of Absolute Differences of Unsigned Words and Accumulate (__ev_saduwaa)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evsaduwaa d,a,b**

__ev_satsbub

__ev_satsbub

Vector Saturate Signed Bytes to Unsigned Byte Range

d = __ev_satsbub (a)

```
// b0
if (a0:7 <si 0x00) then ovb0=1 else ovb0=0;
d0:7 ←SATURATE(ovb0, 0, 0x00, 0x00, a0:7)
// b1
if (a8:15 <si 0x00) then ovb1=1 else ovb1=0;
d8:15 ←SATURATE(ovb1, 0, 0x00, 0x00, a8:15)
// b2
if (a16:23 <si 0x00) then ovb2=1 else ovb2=0;
d16:23 ←SATURATE(ovb2, 0, 0x00, 0x00, a16:23)
// b3
if (a24:31 <si 0x00) then ovb3=1 else ovb3=0;
d24:31 ←SATURATE(ovb3, 0, 0x00, 0x00, a24:31)
// b4
if (a32:39 <si 0x00) then ovb4=1 else ovb4=0;
d32:39 ←SATURATE(ovb4, 0, 0x00, 0x00, a32:39)
// b5
if (a40:47 <si 0x00) then ovb5=1 else ovb5=0;
d40:47 ←SATURATE(ovb5, 0, 0x00, 0x00, a40:47)
// b6
if (a48:55 <si 0x00) then ovb6=1 else ovb6=0;
d48:55 ←SATURATE(ovb6, 0, 0x00, 0x00, a48:55)
// b7
if (a56:63 <si 0x00) then ovb7=1 else ovb7=0;
d56:63 ←SATURATE(ovb7, 0, 0x00, 0x00, a56:63)

ovh ←ovb0 | ovb1 | ovb2 | ovb3
ovl ←ovh4 | ovb5 | ovb6 | ovb7
SPEFSCROVH ←ovh
SPEFSCROV ←ovl
SPEFSCRSOVH ←SPEFSCRSOVH | ovh
SPEFSCRSOV ←SPEFSCRSOV | ovl.
```

The signed 8-bit elements of parameter **a** are saturated to 8-bit unsigned values. Negative elements saturate to 0. The 8-bit results are placed into parameter **d**. Any saturation is recorded in the SPEFSCR overflow and summary overflow bits.

Other registers altered: SPEFSCR

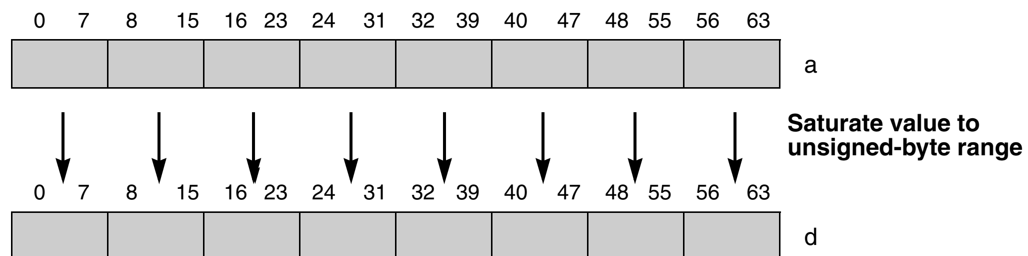


Figure 3-592. Vector Saturate Signed Byte to Unsigned Byte Range (__ev_satsbub)

d	a	Maps to
__ev64_opaque__	__ev64_opaque__	**evsatsbub d,a**

__ev_satsdsw __ev_satsdsw

Vector Saturate Signed Double Word to Signed Word Range

d = __ev_satsdsw (a)

```
if ((a0:63 <si 0xFFFF_FFFF_FFFF_8000) | (a0:63 >si 0x0000_0000_7FFF_FFFF)) then ov=1
else ov=0;
d0:63 ←EXTS(SATURATE(ov, a0, 0x8000_0000, 0x7fff_ffff, a32:63))

SPEFSCROVH ←0
SPEFSCROV ←ov
SPEFSCRSOV ←SPEFSCRSOV | ov.
```

The signed 64-bit value in parameter **a** is saturated to a 32-bit signed value. The 32-bit value is sign-extended to 64 bits and placed into parameter **d**. Any saturation is recorded in the SPEFSCR overflow and summary overflow bits.

Other registers altered: SPEFSCR

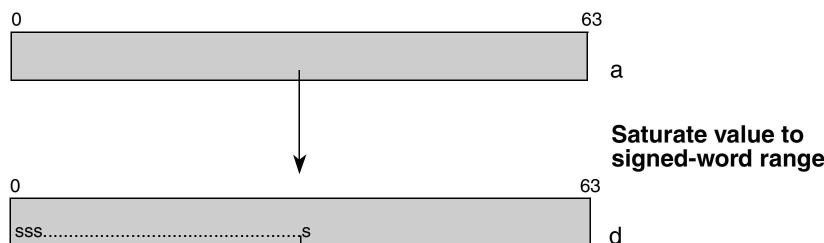


Figure 3-593. Vector Saturate Signed Doubleword to Signed Word Range (__ev_satsdsw)

d	a	Maps to
__ev64_opaque__	__ev64_opaque__	**evsatsdsw d,a**

__ev_satsduw __ev_satsduw

Vector Saturate Signed Double Word to Unsigned Word Range

d = __ev_satsduw (a)

```
if ((a0:63 <si 0x0000_0000_0000_0000) | (a0:63 >si 0x0000_0000_FFFF_FFFF)) then ov=1
else ov=0;
d0:63 ←EXTZ(SATURATE(ov, a0, 0x0000_0000, 0xFFFF_FFFF, a32:63))

SPEFSCROVH ←0
SPEFSCROV ←ov
SPEFSCRSOV ←SPEFSCRSOV | ov.
```

The signed 64-bit value in parameter **a** is saturated to a 32-bit unsigned value. The 32-bit value is zero-extended to 64 bits and placed into parameter **d**. Any saturation is recorded in the SPEFSCR overflow and summary overflow bits.

Other registers altered: SPEFSCR

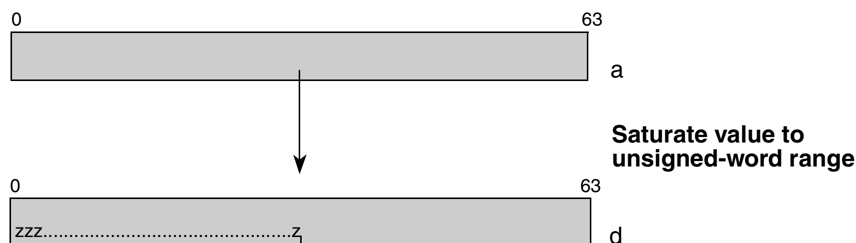


Figure 3-594. Vector Saturate Signed Double Word to Unsigned Word Range (__ev_satsduw)

d	a	Maps to
__ev64_opaque__	__ev64_opaque__	**evsatsduw d,a**

__ev_satshsb   __ev_satshsb

Vector Saturate Signed Half Words to Signed Byte Range

d = __ev_satshsb (a)

```
// h0
if ((a0:15 <si 0xFF80) | (a0:15 >si 0x007F)) then ovh0=1 else ovh0=0;
d0:15 ←EXTS(SATURATE(ovh0, a0, 0x80, 0x7f, a8:15))
// h1
if ((a16:31 <si 0xFF80) | (a16:31 >si 0x007F)) then ovh1=1 else ovh1=0;
d16:31 ←EXTS(SATURATE(ovh1, a16, 0x80, 0x7f, a24:31))
// h2
if ((a32:47 <si 0xFF80) | (a32:47 >si 0x007F)) then ovh2=1 else ovh2=0;
d32:47 ←EXTS(SATURATE(ovh2, a32, 0x80, 0x7f, a40:47))
// h3
if ((a48:63 <si 0xFF80) | (a48:63 >si 0x007F)) then ovh3=1 else ovh3=0;
d48:63 ←EXTS(SATURATE(ovh3, a48, 0x80, 0x7f, a56:63))


ovh ←ovh0 | ovh1
ovl ←ovb2 | ovb3
SPEFSCROVH ←ovh
SPEFSCROV ←ovl
SPEFSCRSOVH ←SPEFSCRSOVH | ovh
SPEFSCRSOV ←SPEFSCRSOV | ovl.
```

The signed 16-bit elements of parameter **a** are saturated to 8-bit signed values. The 8-bit values are sign-extended to half words and placed into parameter **d**. Any saturation is recorded in the SPEFSCR overflow and summary overflow bits.

Other registers altered: SPEFSCR

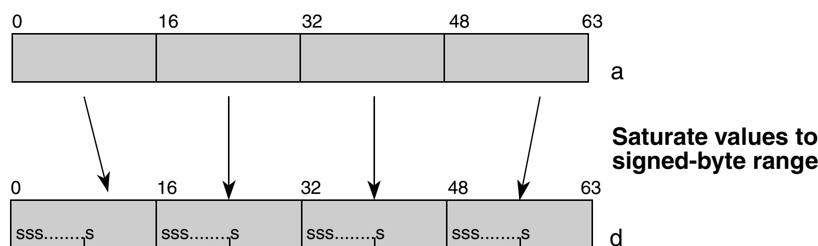


Figure 3-595. Vector Saturate Signed Half Words to Signed Byte Range (__ev_satshsb)

d	a	Maps to
__ev64_opaque__	__ev64_opaque__	**evsatshsb d,a**

__ev_satshub __ev_satshub

Vector Saturate Signed Half Words to Unsigned Byte Range

d = __ev_satshub (a)

```
// b0
if ((a0:15 <si 0x0000) | (a0:15 >si 0x00FF)) then ovh0=1 else ovh0=0;
d0:15 ←EXTZ(SATURATE(ovh0, a0, 0x00, 0xFF, a8:15))
// b1
if ((a16:31 <si 0x000) | (a16:31 >si 0x00FF)) then ovh1=1 else ovh1=0;
d16:31 ←EXTZ(SATURATE(ovh1, a16, 0x00, 0xFF, a24:31))
// b2
if ((a32:47 <si 0x0000) | (a32:47 >si 0x00FF)) then ovh2=1 else ovh2=0;
d32:47 ←EXTZ(SATURATE(ovh2, a32, 0x00, 0xFF, a40:47))
// b3
if ((a48:63 <si 0x0000) | (a48:63 >si 0x00FF)) then ovh3=1 else ovh3=0;
d48:63 ←EXTZ(SATURATE(ovh3, a48, 0x00, 0xFF, a56:63))

ovh ←ovh0 | ovh1
ovl ←ovh2 | ovh3
SPEFSCROVH ←ovh
SPEFSCROV ←ovl
SPEFSCRSOVH ←SPEFSCRSOVH | ovh
SPEFSCRSOV ←SPEFSCRSOV | ovl.
```

The signed 16-bit elements of parameter **a** are saturated to 8-bit unsigned values. Negative elements saturate to 0. The 8-bit results are zero-extended to half words and placed into parameter **d**. Any saturation is recorded in the SPEFSCR overflow and summary overflow bits.

Other registers altered: SPEFSCR

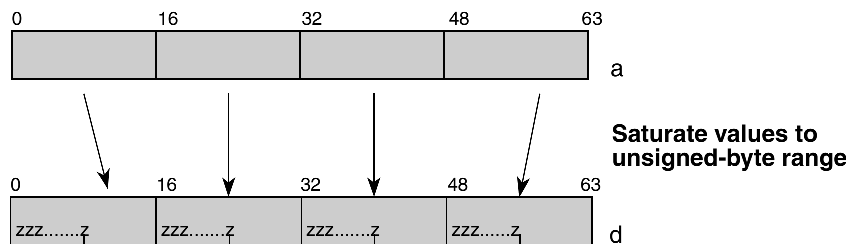


Figure 3-596. Vector Saturate Signed Half Words to Unsigned Byte Range (__ev_satshub)

d	a	Maps to
__ev64_opaque__	__ev64_opaque__	**evsatshub d,a**

__ev_satshuh __ev_satshuh

Vector Saturate Signed Half Words to Unsigned Half Word Range

d = __ev_satshuh (a)

```
// h0
if (a0:15 <si 0x0000) then ovh0=1 else ovh0=0;
d0:15 ←SATURATE(ovh0, 0, 0x0000, 0x0000, a0:15)
// h1
if (a16:31 <si 0x000) then ovh1=1 else ovh1=0;
d16:31 ←SATURATE(ovh1, 0, 0x0000, 0x0000, a16:31)
// h2
if (a32:47 <si 0x0000) then ovh2=1 else ovh2=0;
d32:47 ←SATURATE(ovh2, 0, 0x0000, 0x0000, a32:47)
// h3
if (a48:63 <si 0x0000) then ovh3=1 else ovh3=0;
d48:63 ←SATURATE(ovh3, 0, 0x0000, 0x0000, a48:63)

ovh ←ovh0 | ovh1
ovl ←ovh2 | ovh3
SPEFSCROVH ←ovh
SPEFSCROV ←ovl
SPEFSCRSOVH ←SPEFSCRSOVH | ovh
SPEFSCRSOV ←SPEFSCRSOV | ovl.
```

The signed 16-bit elements of parameter **a** are saturated to 16-bit unsigned values. Negative elements saturate to 0. The 16-bit results are placed into parameter **d**. Any saturation is recorded in the SPEFSCR overflow and summary overflow bits.

Other registers altered: SPEFSCR

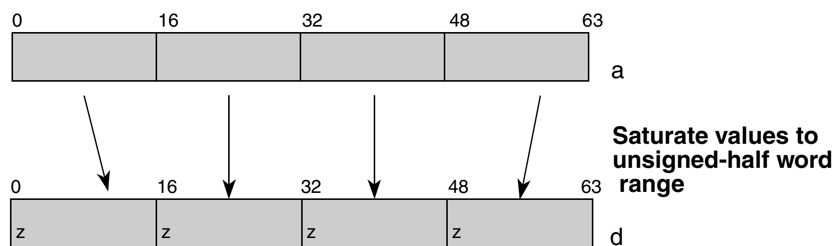


Figure 3-597. Vector Saturate Signed Half Words to Unsigned Half Word Range (__ev_satshuh)

d	a	Maps to
__ev64_opaque__	__ev64_opaque__	**evsatshuh d,a**

__ev_satswgsdf __ev_satswgsdf

Vector Saturate Signed Word Guarded to Signed Doubleword Fractional

d = __ev_satswgsdf (a)

```
if ((a0:63 <si 0xFFFF_8000_0000_0000) | (a0:63 >si 0x0000_7FFF_FFFF_FFFF)) then
    ov=1
    temp0:63 ←SATURATE(ov, a0, 0x8000_0000_0000_0000, 0x7fff_ffff_ffff_0000, -----)
else
    ov=0
    temp0:63 ←a16:63 || 160)

d0:63 ←temp0:63

SPEFSCROVH ←0
SPEFSCROV ←ov
SPEFSCRSOV ←SPEFSCRSOV | ov
```

The signed 64-bit guarded word fraction in 17.47 fractional format in parameter **a** is saturated to a 64-bit fraction in 1.63 fractional format, zeroing the low-order 16 bits. The result is placed into parameter **d**. The original value is assumed to be in 17.47 fractional format as a result of one or more guarded word fractional operations. Any saturation is recorded in the SPEFSCR overflow and summary overflow bits.

Other registers altered: SPEFSCR

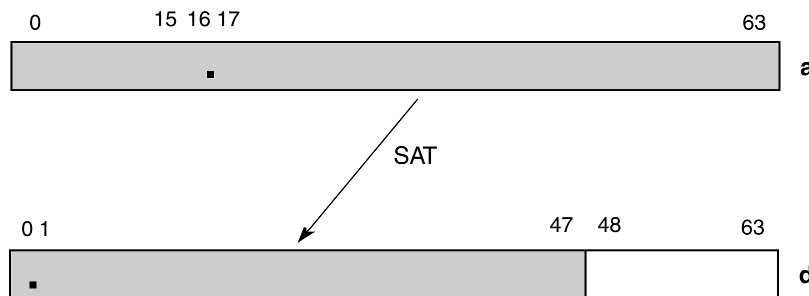


Figure 3-598. Vector Saturate Signed Word Guarded to Signed Doubleword Fractional (__ev_satswgsdf)

d	a	Maps to
__ev64_opaque__	__ev64_opaque__	**evsatswgsdf d,a**

__ev_satswsh __ev_satswsh

Vector Saturate Signed Words to Signed Half Word Range

d = __ev_satswsh (a)

```
// w0
if ((a0:31 <si 0xFFFF8000) | (a0:31 >si 0x00007FFF)) then ovh=1 else ovh=0;
d0:31 ←EXTS(SATURATE(ovh, a0, 0x8000, 0x7fff, a16:31))

// w1
if ((a32:63 <si 0xFFFF8000) | (a32:63 >si 0x00007FFF)) then ovl=1 else ovl=0;
d32:63 ←EXTS(SATURATE(ovl, a32, 0x8000, 0x7fff, a48:63))

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

The signed 32-bit elements of parameter **a** are saturated to 16-bit signed values. The 16-bit results are sign-extended to 32-bits and placed into parameter **d**. Any saturation is recorded in the SPEFSCR overflow and summary overflow bits.

Other registers altered: SPEFSCR

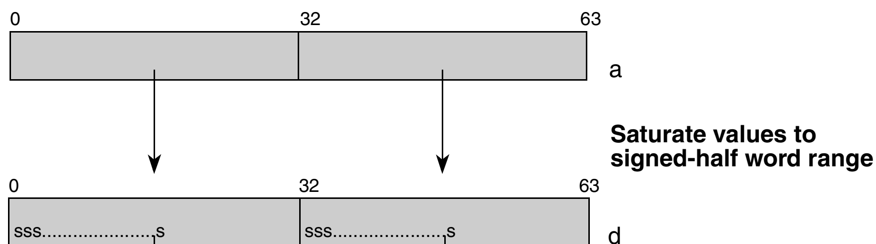


Figure 3-599. Vector Saturate Signed Words to Signed Half Word Range (__ev_satswsh)

d	a	Maps to
__ev64_opaque__	__ev64_opaque__	**evsatswsh d,a**

__ev_satswuh __ev_satswuh

Vector Saturate Signed Words to Unsigned Half Word Range

d = __ev_satswuh (a)

```
// w0
if ((a0:31 <si 0x00000000) | (a0:31 >si 0x0000FFFF)) then ovh=1 else ovh=0;
d0:31 ←EXTZ(SATURATE(ovh, a0, 0x0000, 0xFFFF, a16:31))

// w1
if ((a32:63 <si 0x00000000) | (a32:63 >si 0x0000FFFF)) then ovl=1 else ovl=0;
d32:63 ←EXTZ(SATURATE(ovl, a32, 0x0000, 0xFFFF, a48:63))

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

The signed 32-bit elements of parameter **a** are saturated to 16-bit unsigned values. Negative elements saturate to 0. The values are zero-extended to 32-bits and placed into parameter **d**. Any saturation is recorded in the SPEFSCR overflow and summary overflow bits.

Other registers altered: SPEFSCR

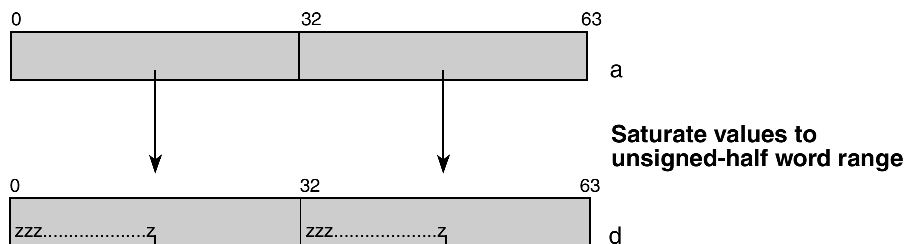


Figure 3-600. Vector Saturate Signed Words to Unsigned Half Word Range (__ev_satswuh)

d	a	Maps to
__ev64_opaque__	__ev64_opaque__	**evsatswuh d,a**

__ev_satswuw __ev_satswuw

Vector Pack Signed Words to Unsigned Word Range

d = __ev_satswuw (a)

```
// w0
if (a0:31 <si 0x00000000) then ovh=1 else ovh=0;
d0:31 ←SATURATE(ovh, 0, 0x00000000, 0x00000000, a0:31)

// w1
if (a32:63 <si 0x00000000) then ovl=1 else ovl=0;
d32:63 ←SATURATE(ovl, 0, 0x00000000, 0x00000000, a32:63)

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

The signed 32-bit elements of parameter **a** are saturated to 32-bit unsigned values. Negative elements saturate to 0. The results are placed into parameter **d**. Any saturation is recorded in the SPEFSCR overflow and summary overflow bits.

Other registers altered: SPEFSCR

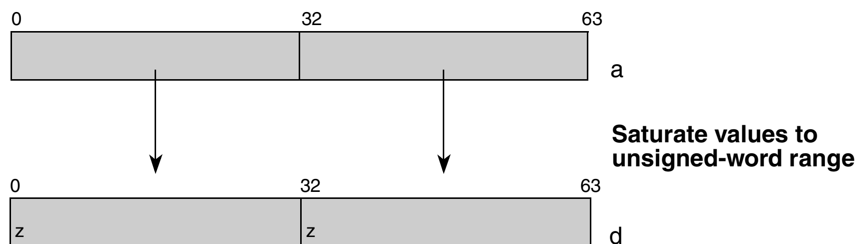


Figure 3-601. Vector Saturate Signed Words to Unsigned Word Range (__ev_satswuw)

d	a	Maps to
__ev64_opaque__	__ev64_opaque__	**evsatswuw d,a**

__ev_satuduw __ev_satuduw

Vector Saturate Unsigned Double Word to Unsigned Word Range

d = __ev_satuduw (a)

```
if (a0:63 >ui 0x0000_0000_FFFF_FFFF) then ov=1 else ov=0;
d0:63 ←EXTZ(SATURATE(ov, 0, 0xFFFF_FFFF, 0xFFFF_FFFF, a32:63))

SPEFSCROVH ←0
SPEFSCROV ←ov
SPEFSCRSOV ←SPEFSCRSOV | ov.
```

The unsigned 64-bit value in parameter **a** is saturated to a 32-bit unsigned value. The 32-bit value is zero-extended to 64 bits and placed into parameter **d**. Any saturation is recorded in the SPEFSCR overflow and summary overflow bits.

Other registers altered: SPEFSCR

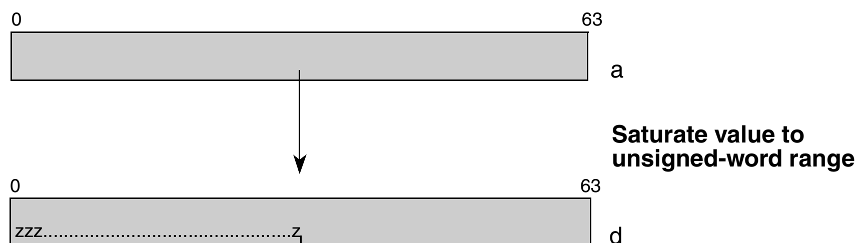


Figure 3-602. Vector Saturate Unsigned Double Word to Unsigned Word Range (__ev_satuduw)

d	a	Maps to
__ev64_opaque__	__ev64_opaque__	**evsatuduw d,a**

__ev_satubsb

__ev_satubsb

Vector Saturate Unsigned Bytes to Signed Byte Range

d = __ev_satubsb (a)

```
// b0
if (a0:7 >ui 0x7F) then ovb0=1 else ovb0=0;
d0:7 ←SATURATE(ovb0, 0, 0x7F, 0x7F, a0:7)
// b1
if (a8:15 >ui 0x7F) then ovb1=1 else ovb1=0;
d8:15 ←SATURATE(ovb1, 0, 0x7F, 0x7F, a8:15)
// b2
if (a16:23 >ui 0x7F) then ovb2=1 else ovb2=0;
d16:23 ←SATURATE(ovb2, 0, 0x7F, 0x7F, a16:23)
// b3
if (a24:31 >ui 0x7F) then ovb3=1 else ovb3=0;
d24:31 ←SATURATE(ovb3, 0, 0x7F, 0x7F, a24:31)
// b4
if (a32:39 >ui 0x7F) then ovb4=1 else ovb4=0;
d32:39 ←SATURATE(ovb4, 0, 0x7F, 0x7F, a32:39)
// b5
if (a40:47 >ui 0x7F) then ovb5=1 else ovb5=0;
d40:47 ←SATURATE(ovb5, 0, 0x7F, 0x7F, a40:47)
// b6
if (a48:55 >ui 0x7F) then ovb6=1 else ovb6=0;
d48:55 ←SATURATE(ovb6, 0, 0x7F, 0x7F, a48:55)
// b7
if (a56:63 >ui 0x7F) then ovb7=1 else ovb7=0;
d56:63 ←SATURATE(ovb7, 0, 0x7F, 0x7F, a56:63)

ovh ←ovb0 | ovb1 | ovb2 | ovb3
ovl ←ovh4 | ovb5 | ovb6 | ovb7
SPEFSCROVH ←ovh
SPEFSCROV ←ovl
SPEFSCRSOVH ←SPEFSCRSOVH | ovh
SPEFSCRSOV ←SPEFSCRSOV | ovl.
```

The unsigned 8-bit elements of parameter **a** are saturated to 8-bit signed values. The 8-bit results are placed into parameter **d**. Any saturation is recorded in the SPEFSCR overflow and summary overflow bits.

Other registers altered: SPEFSCR

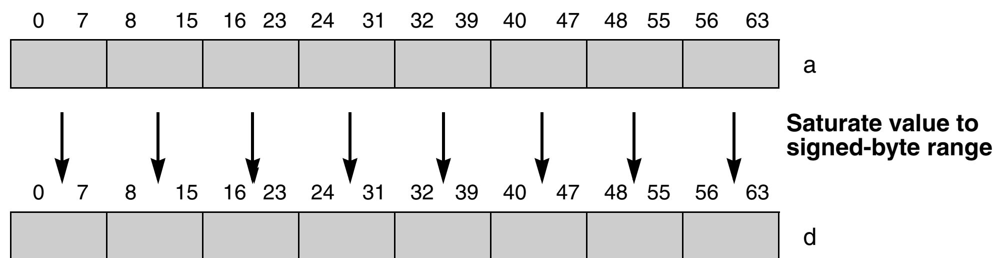


Figure 3-603. Vector Saturate Unsigned Byte to Signed Byte Range (__ev_satubsb)

d	a	Maps to
__ev64_opaque__	__ev64_opaque__	**evsatubsb d,a**

__ev_satuhsh __ev_satuhsh

Vector Saturate Unsigned Half Words to Signed Half Word Range

d = __ev_satuhsh (a)

```
// h0
if (a0:15 >ui 0x7FFF) then ovh0=1 else ovh0=0;
d0:15 ←SATURATE(ovh0, 0, 0x7FFF, 0x7FFF, a0:15)
// h1
if (a16:31 >ui 0x7FFF) then ovh1=1 else ovh1=0;
d16:31 ←SATURATE(ovh1, 0, 0x7FFF, 0x7FFF, a16:31)
// h2
if (a32:47 >ui 0x7FFF) then ovh2=1 else ovh2=0;
d32:47 ←SATURATE(ovh2, 0, 0x7FFF, 0x7FFF, a32:47)
// h3
if (a48:63 >ui 0x7FFF) then ovh3=1 else ovh3=0;
d48:63 ←SATURATE(ovh3, 0, 0x7FFF, 0x7FFF, a48:63)

ovh ←ovh0 | ovh1
ovl ←ovh2 | ovh3
SPEFSCROVH ←ovh
SPEFSCROV ←ovl
SPEFSCRSOVH ←SPEFSCRSOVH | ovh
SPEFSCRSOV ←SPEFSCRSOV | ovl.
```

The unsigned 16-bit elements of parameter **a** are saturated to 16-bit signed values. The 16-bit results are placed into parameter **d**. Any saturation is recorded in the SPEFSCR overflow and summary overflow bits.

Other registers altered: SPEFSCR

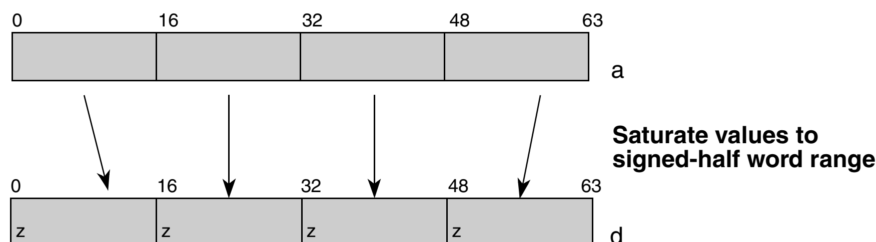


Figure 3-604. Vector Saturate Unsigned Half Words to Signed Half Word Range (__ev_satuhsh)

d	a	Maps to
__ev64_opaque__	__ev64_opaque__	**evsatuhsh d,a**

__ev_satuhub __ev_satuhub

Vector Saturate Unsigned Half Words to Unsigned Byte Range

d = __ev_satuhub (a)

```
// h0
if (a0:15 >ui 0x00FF) then ovh0=1 else ovh0=0;
d0:15 ←EXTZ(SATURATE(ovh0, 0, 0xFF, 0xFF, a8:15))
// h1
if (a16:31 >ui 0x00FF) then ovh1=1 else ovh1=0;
d16:31 ←EXTZ(SATURATE(ovh1, 0, 0xFF, 0xFF, a24:31))
// h2
if (a32:47 >ui 0x00FF) then ovh2=1 else ovh2=0;
d32:47 ←EXTZ(SATURATE(ovh2, 0, 0xFF, 0xFF, a40:47))
// h3
if (a48:63 >ui 0x00FF) then ovh3=1 else ovh3=0;
d48:63 ←EXTZ(SATURATE(ovh3, 0, 0xFF, 0xFF, a56:63))

ovh ←ovh0 | ovh1
ovl ←ovh2 | ovh3
SPEFSCROVH ←ovh
SPEFSCROV ←ovl
SPEFSCRSOVH ←SPEFSCRSOVH | ovh
SPEFSCRSOV ←SPEFSCRSOV | ovl.
```

The unsigned 16-bit elements of parameter **a** are saturated to 8-bit unsigned values. The 8-bit results are zero-extended to half words and placed into parameter **d**. Any saturation is recorded in the SPEFSCR overflow and summary overflow bits.

Other registers altered: SPEFSCR

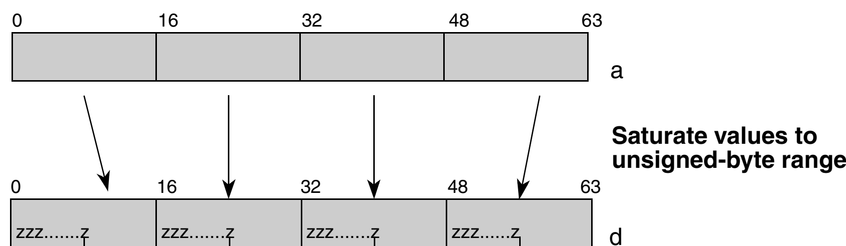


Figure 3-605. Vector Saturate Unsigned Half Words to Unsigned Byte Range (__ev_satuhub)

d	a	Maps to
__ev64_opaque__	__ev64_opaque__	**evsatuhub d,a**

__ev_satuwsw __ev_satuwsw

Vector Saturate Unsigned Words to Signed Word Range

d = __ev_satuwsw (a)

```
// w0
if (a0:31 >ui 0x7FFFFFFF) then ovh=1 else ovh=0;
d0:31 ←SATURATE(ovh, 0, 0x7FFFFFFF, 0x7FFFFFFF, a0:31)

// w1
if ((a32:63 >ui 0x7FFFFFFFF) then ovl=1 else ovl=0;
d32:63 ←SATURATE(ovl, 0, 0x7FFFFFFF, 0x7FFFFFFF, a32:63)

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

The unsigned 32-bit elements of parameter **a** are saturated to 32-bit signed values. The 32-bit results are placed into parameter **d**. Any saturation is recorded in the SPEFSCR overflow and summary overflow bits.

Other registers altered: SPEFSCR

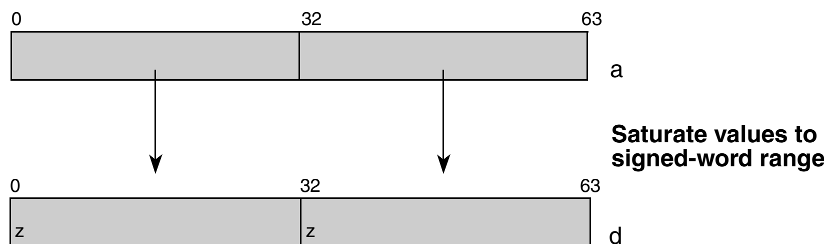


Figure 3-606. Vector Saturate Unsigned Words to Signed Word Range (__ev_satuwsw)

d	a	Maps to
__ev64_opaque__	__ev64_opaque__	**evsatuwsw d,a**

__ev_satuwuh __ev_satuwuh

Vector Saturate Unsigned Words to Unsigned Half Word Range

d = __ev_satuwuh (a)

```
// w0
if (a0:31 >ui 0x0000FFFF) then ovh=1 else ovh=0;
d0:15 ←EXTZ(SATURATE(ovh, 0, 0xFFFF, 0xFFFF, a16:31))

// w1
if ((a32:63 >ui 0x0000FFFF) then ovl=1 else ovl=0;
d16:31 ←EXTZ(SATURATE(ovl, 0, 0xFFFF, 0xFFFF, a48:63))

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

The unsigned 32-bit elements of parameters **a** and **b** are saturated to 16-bit unsigned values. The 16-bit results are zero-extended into words and placed into parameter **d**. Any saturation is recorded in the SPEFSCR overflow and summary overflow bits.

Other registers altered: SPEFSCR

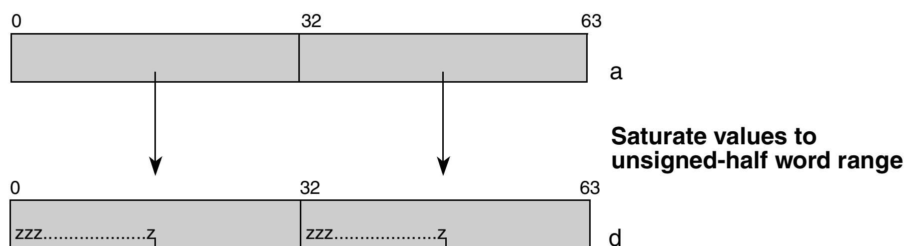


Figure 3-607. Vector Saturate Unsigned Words to Unsigned Half Word Range (__ev_satuwuh)

d	a	Maps to
__ev64_opaque__	__ev64_opaque__	**evsatuwuh 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.

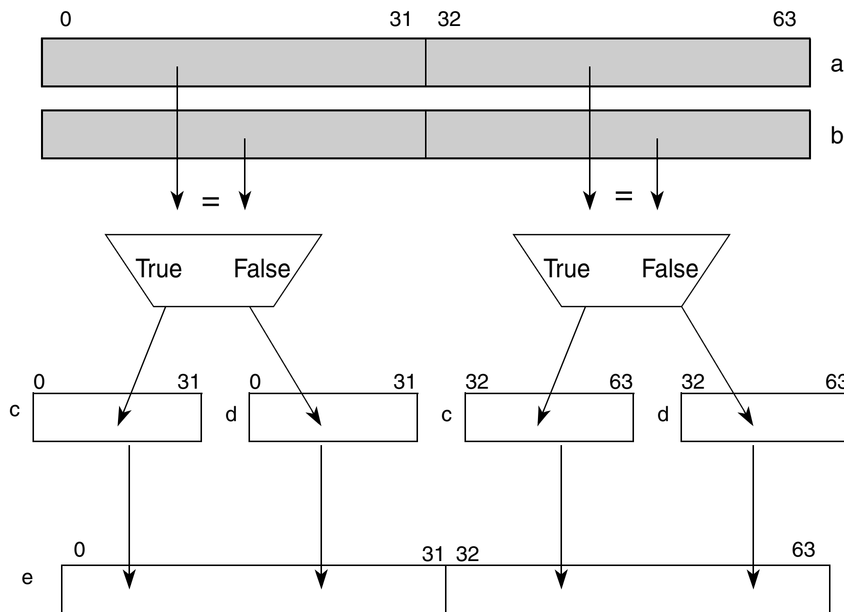


Figure 3-608. Vector Select Equal (__ev_select_eq)

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_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.

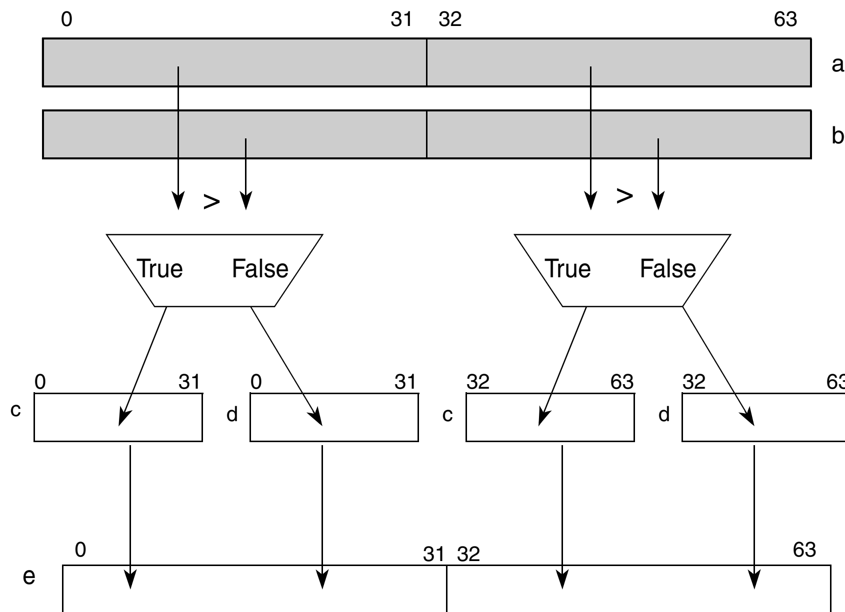


Figure 3-609. Vector Select Greater Than Signed (__ev_select_gts)

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.

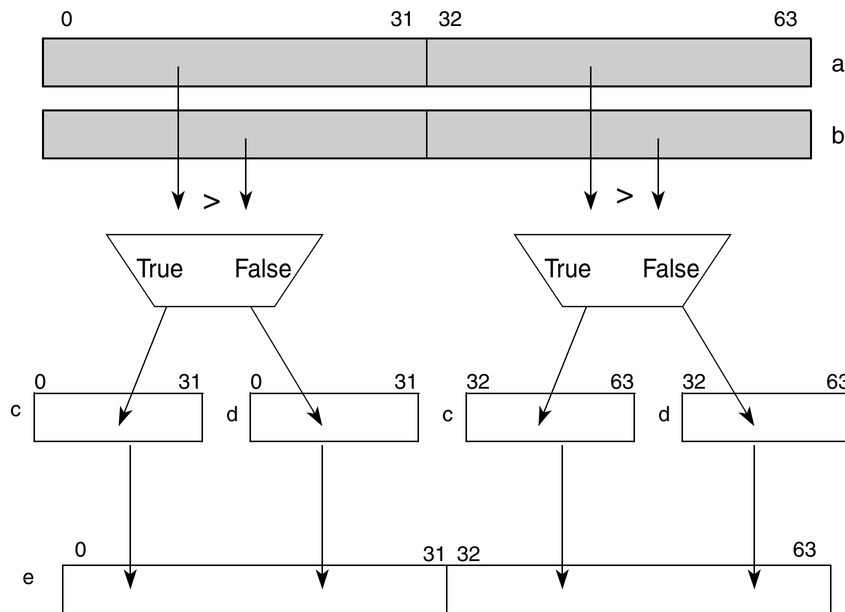


Figure 3-610. Vector Select Greater Than Unsigned (__ev_select_gtu)

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)

$$\begin{aligned}&\text{if } (a_{0:31} <_{signed} b_{0:31}) \text{ then } e_{0:31} \leftarrow c_{0:31}\\&\text{else } e_{0:31} \leftarrow d_{0:31}\\[6pt]&\text{if } (a_{32:63} <_{signed} b_{32:63}) \text{ then } e \leftarrow c_{32:63}\\&\text{else } e \leftarrow d_{32:63}\end{aligned}$$

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.

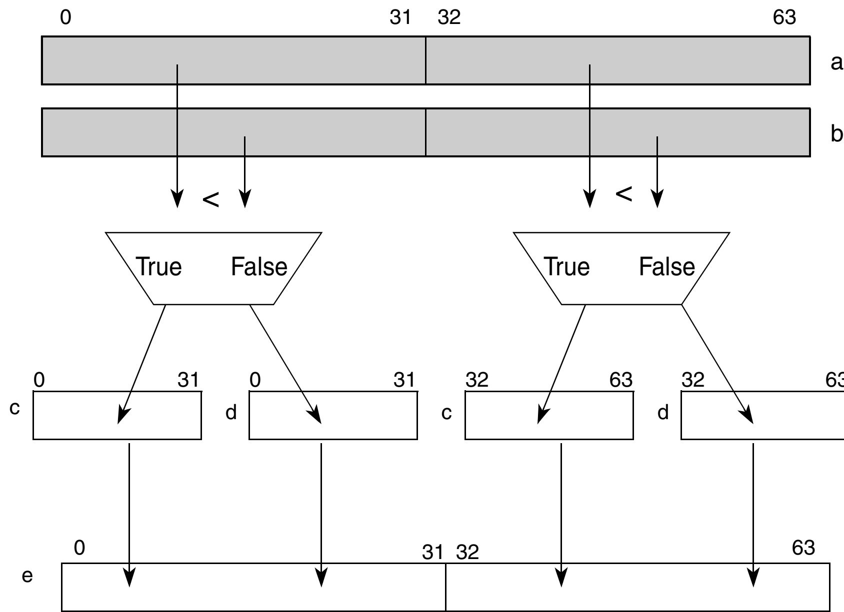


Figure 3-611. Vector Select Less Than Signed (__ev_select_lts)

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.

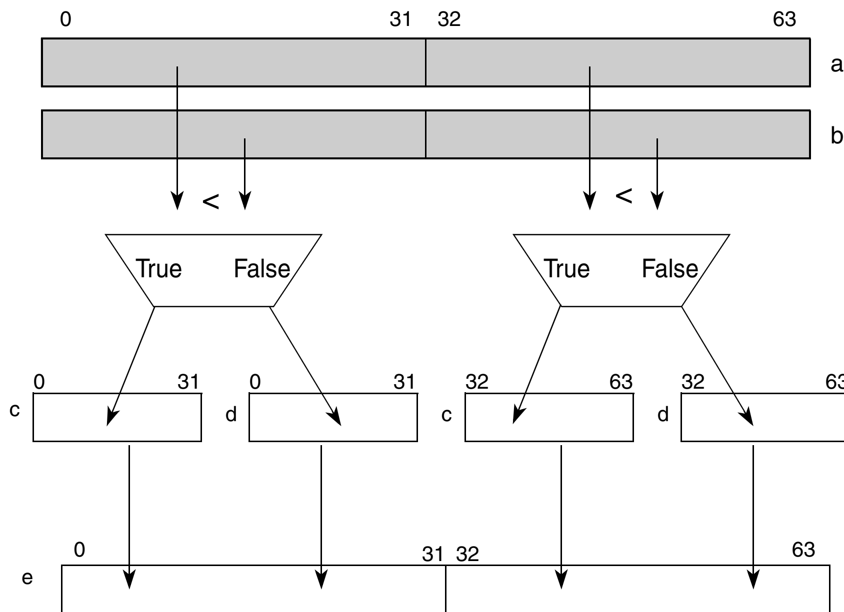


Figure 3-612. Vector Select Less Than Unsigned (__ev_select_ltu)

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_selbit __ev_selbit

Vector Select Bit

d = __ev_selbit (a,b,c)

```
temp0:63 ←64 0

do i=0 to 63
    if (ai = 0) then tempi ←bi
    else tempi ←ci
end
d0:63 ←temp0:63
```

For each bit in parameter **a** that contains the value 0, the corresponding bit in parameter **b** is selected. For each bit in parameter **a** that contains the value 1, the corresponding bit in parameter **c** is selected. The selected bits are then placed into parameter **d**.

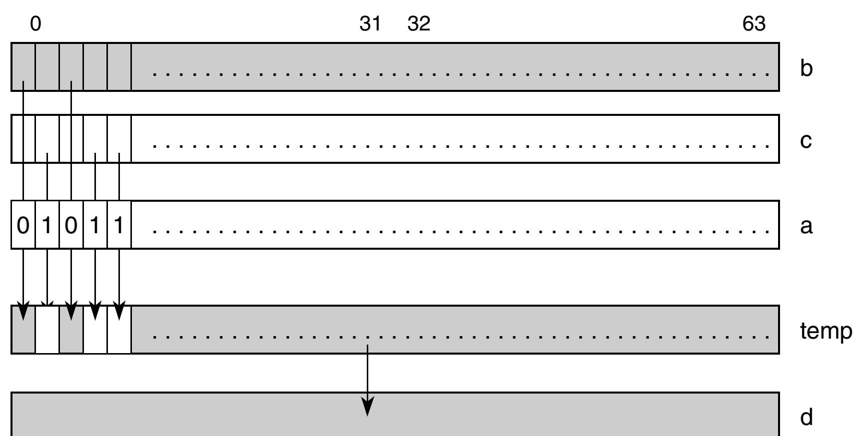


Figure 3-613. Vector Select Bit (__ev_selbit)

d	a	b	c	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	d ←a evselbit d,b,c

__ev_selbitm0 __ev_selbitm0

Vector Select Bit if Mask is 0

d = __ev_selbitm0 (a,b,c)

```
temp0:63 ←64 0

do i=0 to 63
    if (bi = 0) then tempi ←ci
    else tempi ←ai
end
d0:63 ←temp0:63
```

For each bit in parameter **b** that contains the value 0, the corresponding bit in parameter **c** is selected. For each bit in parameter **b** that contains the value 1, the corresponding bit in parameter **a** is selected. The selected bits are then placed into parameter **d**.

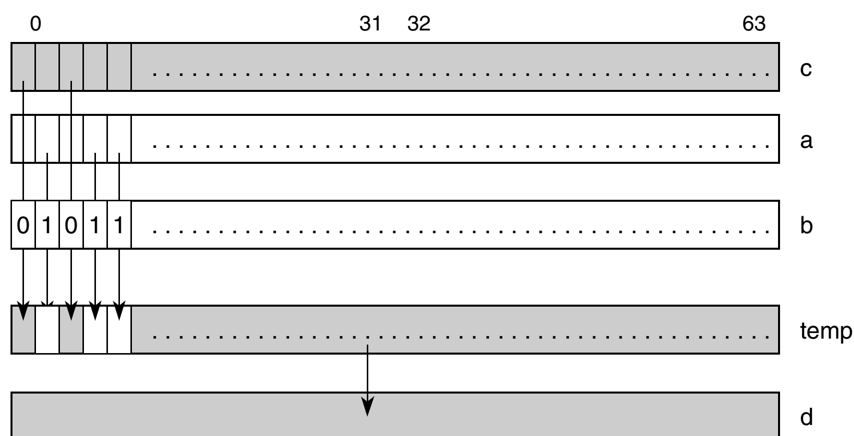


Figure 3-614. Vector Select Bit if Mask is 0 (__ev_selbitm0)

d	a	b	c	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	d ←a evselbitm0 d,b,c

__ev_selbitm1 __ev_selbitm1

Vector Select Bit if Mask is 1

d = __ev_selbitm1 (a,b,c)

```
temp0:63 ←64 0

do i=0 to 63
    if (bi = 0) then tempi ←ai
    else tempi ←ci
end
d0:63 ←temp0:63
```

For each bit in parameter **b** that contains the value 1, the corresponding bit in parameter **c** is selected. For each bit in parameter **b** that contains the value 0, the corresponding bit in parameter **a** is selected. The selected bits are then placed into parameter **d**.

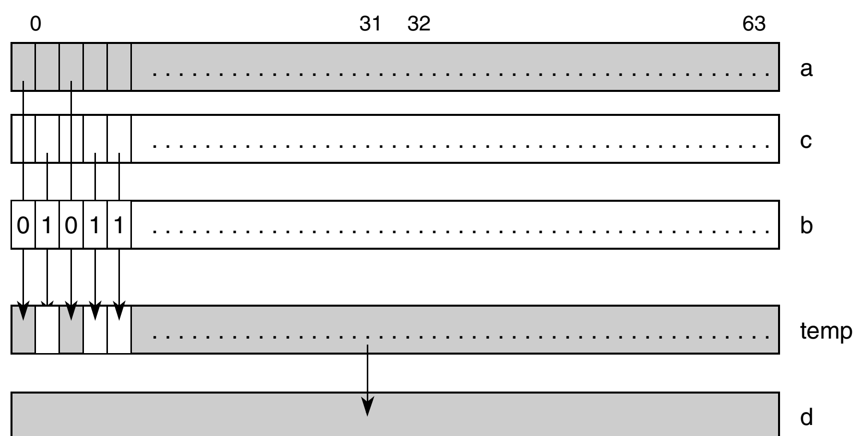


Figure 3-615. Vector Select Bit if Mask is 1 (__ev_selbitm1)

d	a	b	c	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	d ←a **evselbitm1 d,b,c**

__ev_seteqb[_rc] __ev_seteqb[_rc]

Vector Set if Equal Byte [and Record]

d = __ev_seteqb (a,b) (Rc = 0)

d = __ev_seteqb_rc (a,b) (Rc = 1)

```
do i=0 to 63 by 8
    if a_{i:i+7} = b_{i:i+7} then d_{i:i+7} = ^8 1 else d_{i:i+7} = ^8 0
end
c0 ←(a_{0:7} = b_{0:7}) | (a_{8:15} = b_{8:15}) | (a_{16:23} = b_{16:23}) | (a_{24:31} = b_{24:31})
c1 ←(a_{0:7} = b_{0:7}) & (a_{8:15} = b_{8:15}) & (a_{16:23} = b_{16:23}) & (a_{24:31} = b_{24:31})
c2 ←(a_{0:7} = b_{0:7}) | (a_{8:15} = b_{8:15}) | (a_{16:23} = b_{16:23}) | (a_{24:31} = b_{24:31}) |
      (a_{32:39} = b_{32:39}) | (a_{40:47} = b_{40:47}) | (a_{48:55} = b_{48:55}) | (a_{56:63} = b_{56:63})
c3 ←(a_{0:7} = b_{0:7}) & (a_{8:15} = b_{8:15}) & (a_{16:23} = b_{16:23}) & (a_{24:31} = b_{24:31}) &
      (a_{32:39} = b_{32:39}) & (a_{40:47} = b_{40:47}) & (a_{48:55} = b_{48:55}) & (a_{56:63} = b_{56:63})
if (Rc = 1) then CR_{0:3} ←c0 || c1 || c2 || c3
```

Each byte element in parameter **a** is compared to the corresponding element in parameter **b**. The corresponding element of parameter **d** is set to all ‘1’s if the element in parameter **a** is equal to the element in parameter **b**, and is cleared to all ‘0’s otherwise.

If Rc=1, then **cr**0 is updated according to the comparison results. The most significant bit in **cr**0 is set if any of the high-order four elements of parameter **a** is equal to the corresponding element of parameter **b**; it is cleared otherwise. The next bit in **cr**0 is set if all of the high-order four elements of parameter **a** are equal to the corresponding element of parameter **b**; it is cleared otherwise. The third bit of **cr**0 is set if any element of parameter **a** is equal to the corresponding element of parameter **b**; it is cleared otherwise. The last bit of **cr**0 is set if all of the elements of parameter **a** are equal to the corresponding elements of parameter **b**; it is cleared otherwise.
Other registers altered: Condition Register (CR0) (if Rc=1)

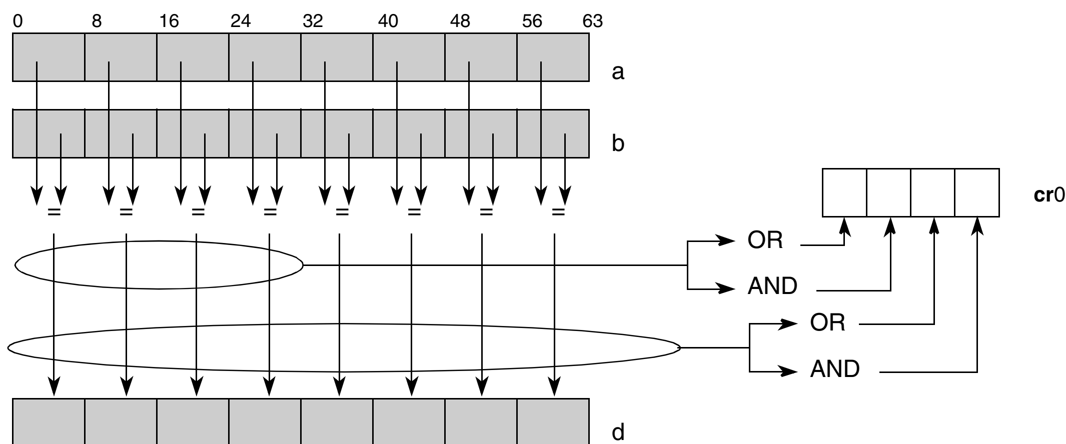


Figure 3-616. Vector Set if Equal Byte [and Record] (__ev_seteqb[_rc])

Rc	d	a	b	Maps to
Rc = 0	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evseteqb d,a,b**
Rc = 1	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evseteqb. d,a,b**

__ev_seteqh[_rc] __ev_seteqh[_rc]

Vector Set if Equal Half Word [and Record]

d = __ev_seteqh (a,b) (Rc = 0)

d = __ev_seteqh_rc (a,b) (Rc = 1)

```
do i=0 to 63 by 16
    if a_{i:i+15} = b_{i:i+15} then d_{i:i+15} = ^{16}1 else d_{i:i+15} = ^{16}0
end
c0 ←(a_{0:15} = b_{0:15}) | (a_{16:31} = b_{16:31})
c1 ←(a_{0:15} = b_{0:15}) & (a_{16:31} = b_{16:31})
c2 ←(a_{0:15} = b_{0:15}) | (a_{16:31} = b_{16:31}) | (a_{32:47} = b_{32:47}) | (a_{48:63} = b_{48:63})
c3 ←(a_{0:15} = b_{0:15}) & (a_{16:31} = b_{16:31}) & (a_{32:47} = b_{32:47}) & (a_{48:63} = b_{48:63})
if (Rc = 1) then CR_{0:3} ←c0 || c1 || c2 || c3
```

Each half word element in parameter **a** is compared to the corresponding element in parameter **b**. The corresponding element of parameter **d** is set to all ‘1’s if the element in parameter **a** is equal to the element in parameter **b**, and is cleared to all ‘0’s otherwise.

If Rc=1, then **cr**0 is updated according to the comparison results. The most significant bit in **cr**0 is set if either of the high-order two elements of parameter **a** is equal to the corresponding element of parameter **b**; it is cleared otherwise. The next bit in **cr**0 is set if both of the high-order two elements of parameter **a** are equal to the corresponding element of parameter **b**; it is cleared otherwise. The third bit of **cr**0 is set if any element of parameter **a** is equal to the corresponding element of parameter **b**; it is cleared otherwise. The last bit of **cr**0 is set if all of the elements of parameter **a** are equal to the corresponding elements of parameter **b**; it is cleared otherwise.

Other registers altered: Condition Register (CR0) (if Rc=1)

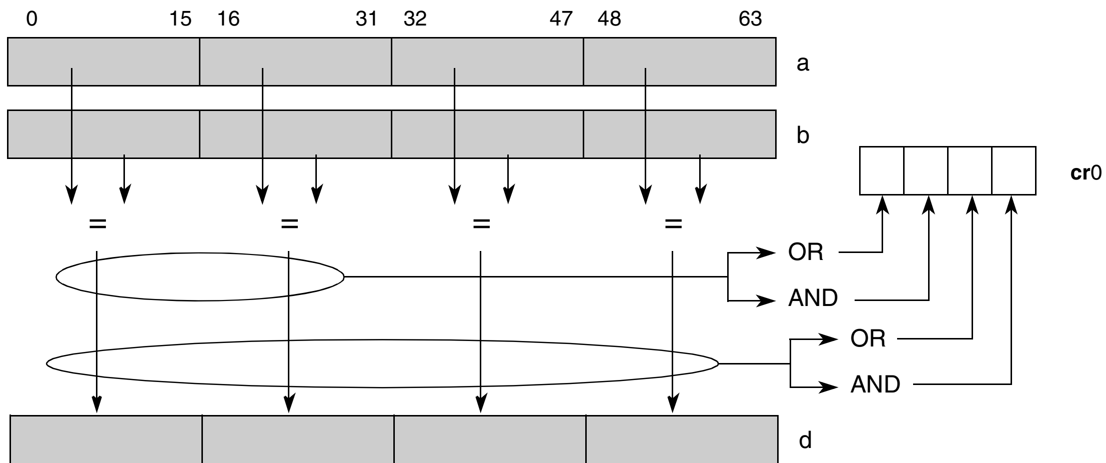


Figure 3-617. Vector Set if Equal Half Word [and Record] (__ev_seteqh[_rc])

Rc	d	a	b	Maps to
Rc = 0	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evseteqh d,a,b**
Rc = 1	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evseteqh. d,a,b**

__ev_seteqw[_rc]

__ev_seteqw[_rc]

Vector Set if Equal Word [and Record]

d = __ev_seteqw (a,b) (Rc = 0)

d = __ev_seteqw_rc (a,b) (Rc = 1)

```
if a0:31 = b0:31 then d0:31 = 321 else d0:31 = 320
if a32:63 = b32:63 then d32:63 = 321 else d32:63 = 320
c0 ←(a0:31 = b0:31)
c1 ←(a32:63 = b32:63)
c2 ←(a0:31 = b0:31) | (a32:63 = b32:63)
c3 ←(a0:31 = b0:31) & (a32:63 = b32:63)
if (Rc = 1) then CR0:3 ←c0 || c1 || c2 || c3
```

Each word element in parameter **a** is compared to the corresponding element in parameter **b**. The corresponding element of parameter **d** is set to all ‘1’s if the element in parameter **a** is equal to the element in parameter **b**, and is cleared to all ‘0’s otherwise.

If Rc=1, then **cr**0 is updated according to the comparison results. The most significant bit in **cr**0 is set if the high-order element of parameter **a** is equal to the high-order element of parameter **b**; it is cleared otherwise. The next bit in **cr**0 is set if the low-order element of parameter **a** is equal to the low-order element of parameter **b**; it is cleared otherwise. The third bit of **cr**0 is set if any element of parameter **a** is equal to the corresponding element of parameter **b**; it is cleared otherwise. The last bit of **cr**0 is set if both elements of parameter **a** are equal to the corresponding elements of parameter **b**; it is cleared otherwise.

Other registers altered: Condition Register (CR0) (if Rc=1)

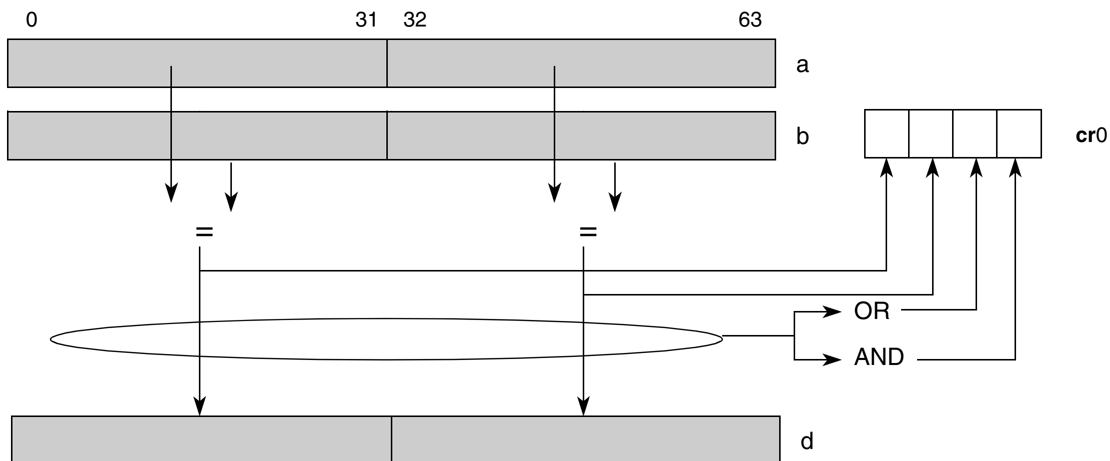


Figure 3-618. Vector Set if Equal Word (__ev_seteqw[_rc])

Rc	d	a	b	Maps to
Rc = 0	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evseteqw d,a,b**
Rc = 1	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evseteqw. d,a,b**

__ev_setgtbs[_rc] __ev_setgtbs[_rc]

Vector Set if Greater Than Byte Signed [and Record]

d = __ev_setgtbs (a,b) (Rc = 0)

d = __ev_setgtbs_rc (a,b) (Rc = 1)

```
do i=0 to 63 by 8
    if a_{i:i+7} >si b_{i:i+7} then d_{i:i+7} = ^8 1 else d_{i:i+7} = ^8 0
end
c0 ←(a_{0:7} >si b_{0:7}) | (a_{8:15} >si b_{8:15}) | (a_{16:23} >si b_{16:23}) | (a_{24:31} >si b_{24:31})
c1 ←(d_{0:7} >si b_{0:7}) & (a_{8:15} >si b_{8:15}) & (a_{16:23} >si b_{16:23}) & (a_{24:31} >si b_{24:31})
c2 ←(a_{0:7} >si b_{0:7}) | (a_{8:15} >si b_{8:15}) | (a_{16:23} >si b_{16:23}) | (a_{24:31} >si b_{24:31}) |
      (a_{32:39} >si b_{32:39}) | (a_{40:47} >si b_{40:47}) | (a_{48:55} >si b_{48:55}) | (a_{56:63} >si b_{56:63})
c3 ←(a_{0:7} >si b_{0:7}) & (a_{8:15} >si b_{8:15}) & (a_{16:23} >si b_{16:23}) & (a_{24:31} >si b_{24:31}) &
      (a_{32:39} >si b_{32:39}) & (a_{40:47} >si b_{40:47}) & (a_{48:55} >si b_{48:55}) & (a_{56:63} >si b_{56:63})
if (Rc = 1) then CR_{0:3} ←c0 || c1 || c2 || c3
```

Each signed byte element in parameter **a** is compared to the corresponding signed byte element in parameter **b**. The corresponding element of parameter **d** is set to all ‘1’s if the element in parameter **a** is greater than the element in parameter **b**, and is cleared to all ‘0’s otherwise.

If Rc=1, then **cr**0 is updated according to the comparison results. The most significant bit in **cr**0 is set if any of the high-order four elements of parameter **a** is greater than the corresponding element of parameter **b**; it is cleared otherwise. The next bit in **cr**0 is set if all of the high-order four elements of parameter **a** are greater than the corresponding element of parameter **b**; it is cleared otherwise. The third bit of **cr**0 is set if any element of parameter **a** is greater than the corresponding element of parameter **b**; it is cleared otherwise. The last bit of **cr**0 is set if all of the elements of parameter **a** are greater than the corresponding elements of parameter **b**; it is cleared otherwise.

Other registers altered: Condition Register (CR0) (if Rc=1)

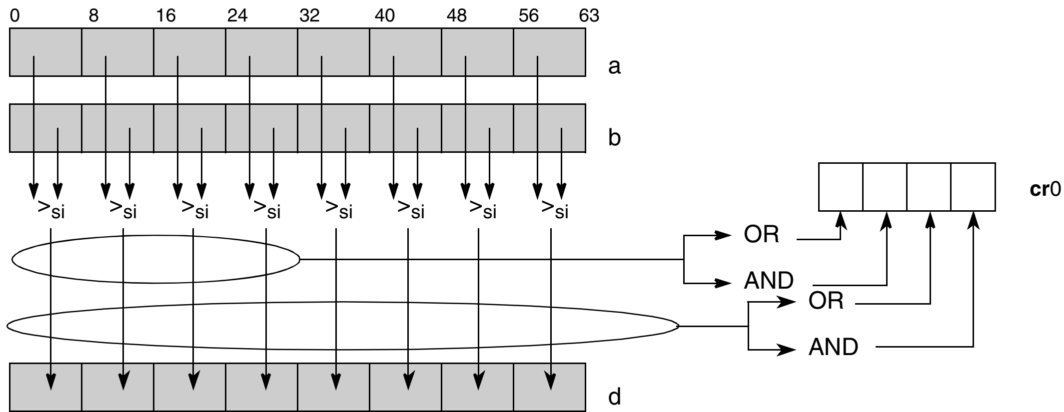


Figure 3-619. Vector Set if Greater than Byte Signed [and Record] (__ev_setgtbs[_rc])

Rc	d	a	b	Maps to
Rc = 0	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evsetgtbs d,a,b**
Rc = 1	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evsetgtbs. d,a,b**

__ev_setgtbu[_rc] __ev_setgtbu[_rc]

Vector Set if Greater Than Byte Unsigned [and Record]

d = __ev_setgtbu (a,b) (Rc = 0)

d = __ev_setgtbu_rc (a,b) (Rc = 1)

```
do i=0 to 63 by 8
    if a_{i:i+7} >ui b_{i:i+7} then d_{i:i+7} = ^8 1 else d_{i:i+7} = ^8 0
end
c0 ←(a_{0:7} >ui b_{0:7}) | (a_{8:15} >ui b_{8:15}) | (a_{16:23} >ui b_{16:23}) | (a_{24:31} >ui b_{24:31})
c1 ←(a_{0:7} >ui b_{0:7}) & (a_{8:15} >ui b_{8:15}) & (a_{16:23} >ui b_{16:23}) & (a_{24:31} >ui b_{24:31})
c2 ←(a_{0:7} >ui b_{0:7}) | (a_{8:15} >ui b_{8:15}) | (a_{16:23} >ui b_{16:23}) | (a_{24:31} >ui b_{24:31}) |
      (a_{32:39} >ui b_{32:39}) | (a_{40:47} >ui b_{40:47}) | (a_{48:55} >ui b_{48:55}) | (a_{56:63} >ui b_{56:63})
c3 ←(a_{0:7} >ui b_{0:7}) & (a_{8:15} >ui b_{8:15}) & (a_{16:23} >ui b_{16:23}) & (a_{24:31} >ui b_{24:31}) &
      (a_{32:39} >ui b_{32:39}) & (a_{40:47} >ui b_{40:47}) & (a_{48:55} >ui b_{48:55}) & (a_{56:63} >ui b_{56:63})
if (Rc = 1) then CR_{0:3} ←c0 || c1 || c2 || c3
```

Each unsigned byte element in parameter **a** is compared to the corresponding unsigned byte element in parameter **b**. The corresponding element of parameter **d** is set to all ‘1’s if the element in parameter **a** is greater than the element in parameter **b**, and is cleared to all ‘0’s otherwise.

If Rc=1, then **cr**0 is updated according to the comparison results. The most significant bit in **cr**0 is set if any of the high-order four elements of parameter **a** is greater than the corresponding element of parameter **b**; it is cleared otherwise. The next bit in **cr**0 is set if all of the high-order four elements of parameter **a** are greater than the corresponding element of parameter **b**; it is cleared otherwise. The third bit of **cr**0 is set if any element of parameter **a** is greater than the corresponding element of parameter **b**; it is cleared otherwise. The last bit of **cr**0 is set if all of the elements of parameter **a** are greater than the corresponding elements of parameter **b**; it is cleared otherwise.

Other registers altered: Condition Register (CR0) (if Rc=1)

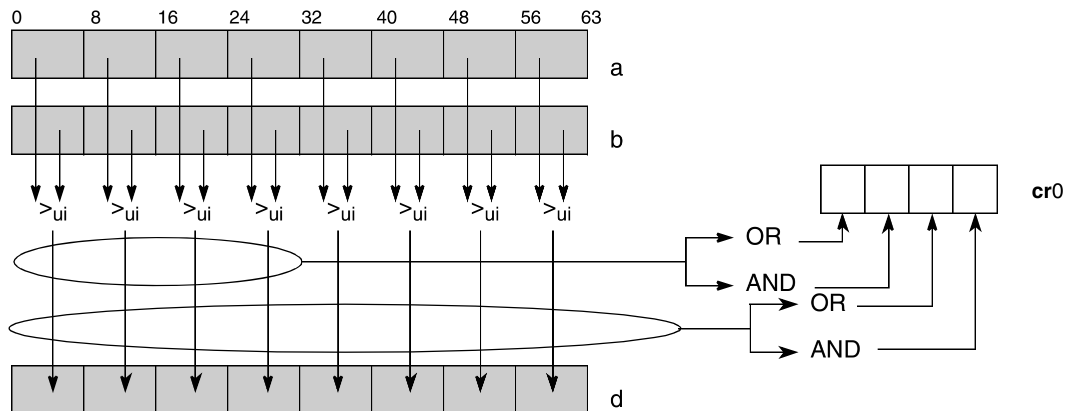


Figure 3-620. Vector Set if Greater than Byte Unsigned [and Record] (__ev_setgtbu[_rc])

Rc	d	a	b	Maps to
Rc = 0	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evsetgtbu d,a,b**
Rc = 1	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evsetgtbu. d,a,b**

__ev_setgths[_rc] __ev_setgths[_rc]

Vector Set if Greater Than Half Word SIgned [and Record]

d = __ev_setgths (a,b) (Rc = 0)

d = __ev_setgths_rc (a,b) (Rc = 1)

```
do i=0 to 63 by 16
    if a_{i:i+15} >si b_{i:i+15} then d_{i:i+15} = ^16 1 else d_{i:i+15} = ^16 0
end
c0 ←(a_{0:15} >si b_{0:15}) | (a_{16:31} >si b_{16:31})
c1 ←(a_{0:15} >si b_{0:15}) & (a_{16:31} >si b_{16:31})
c2 ←(a_{0:15} >si b_{0:15}) | (a_{16:31} >si b_{16:31}) | (a_{32:47} >si b_{32:47}) | (a_{48:63} >si b_{48:63})
c3 ←(a_{0:15} >si b_{0:15}) & (a_{16:31} >si b_{16:31}) & (a_{32:47} >si b_{32:47}) & (a_{48:63} >si b_{48:63})
if (Rc = 1) then CR_{0:3} ←c0 || c1 || c2 || c3
```

Each signed half word element in parameter **a** is compared to the corresponding signed element in parameter **b**. The corresponding element of parameter **d** is set to all ‘1’s if the element in parameter **a** is greater than the element in parameter **b**, and is cleared to all ‘0’s otherwise.

If Rc=1, then **cr**0 is updated according to the comparison results. The most significant bit in **cr**0 is set if either of the high-order two elements of parameter **a** is greater than the corresponding element of parameter **b**; it is cleared otherwise. The next bit in **cr**0 is set if both of the high-order two elements of parameter **a** are greater than the corresponding element of parameter **b**; it is cleared otherwise. The third bit of **cr**0 is set if any element of parameter **a** is greater than the corresponding element of parameter **b**; it is cleared otherwise. The last bit of **cr**0 is set if all of the elements of parameter **a** are greater than the corresponding elements of parameter **b**; it is cleared otherwise.

Other registers altered: Condition Register (CR0) (if Rc=1)

Figure 3-621. Vector Set if Greater than Half Word Signed [and Record] (__ev_setgths[_rc])

Rc	d	a	b	Maps to
Rc = 0	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evsetgths d,a,b**
Rc = 1	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evsetgths. d,a,b**

__ev_setgthu[_rc] __ev_setgthu[_rc]

Vector Set if Greater Than Half Word Unsigned [and Record]

d = __ev_setgthu (a,b) (Rc = 0)

d = __ev_setgthu_rc (a,b) (Rc = 1)

```
do i=0 to 63 by 16
    if a_i:i+15 >ui b_i:i+15 then d_i:i+15 = 16 1 else d_i:i+15 = 16 0
end
c0 ←(a_0:15 >ui b_0:15) | (a_16:31 >ui b_16:31)
c1 ←(a_0:15 >ui b_0:15) & (a_16:31 >ui b_16:31)
c2 ←(a_0:15 >ui b_0:15) | (a_16:31 >ui b_16:31) | (a_32:47 >ui b_32:47) | (a_48:63 >ui b_48:63)
c3 ←(a_0:15 >ui b_0:15) & (a_16:31 >ui b_16:31) & (a_32:47 >ui b_32:47) & (a_48:63 >ui b_48:63)
if (Rc = 1) then CR_0:3 ←c0 || c1 || c2 || c3
```

Each unsigned half word element in parameter **a** is compared to the corresponding unsigned element in parameter **b**. The corresponding element of parameter **d** is set to all ‘1’s if the element in parameter **a** is greater than the element in parameter **b**, and is cleared to all ‘0’s otherwise.

If Rc=1, then **cr**0 is updated according to the comparison results. The most significant bit in **cr**0 is set if either of the high-order two elements of parameter **a** is greater than the corresponding element of parameter **b**; it is cleared otherwise. The next bit in **cr**0 is set if both of the high-order two elements of parameter **a** are greater than the corresponding element of parameter **b**; it is cleared otherwise. The third bit of **cr**0 is set if any element of parameter **a** is greater than the corresponding element of parameter **b**; it is cleared otherwise. The last bit of **cr**0 is set if all of the elements of parameter **a** are greater than the corresponding elements of parameter **b**; it is cleared otherwise.

Other registers altered: Condition Register (CR0) (if Rc=1)

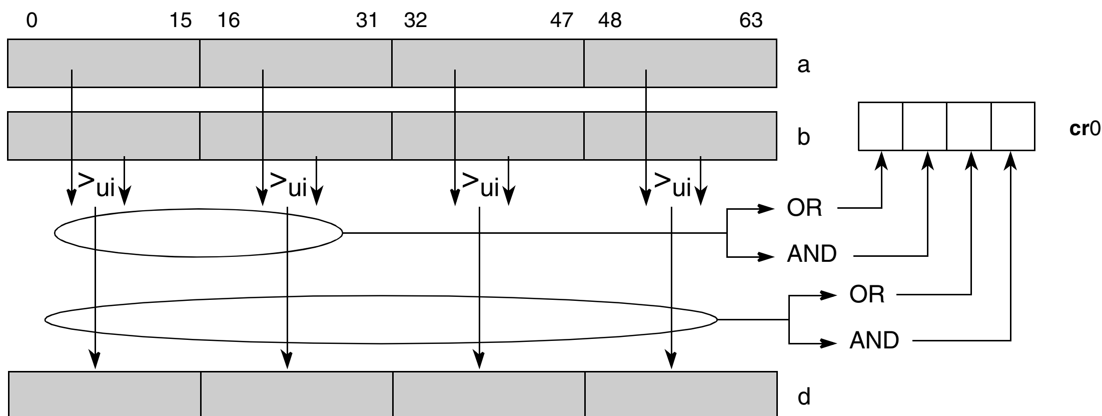


Figure 3-622. Vector Set if Greater than Half Word Unsigned [and Record] (__ev_setgthu[_rc])

Rc	d	a	b	Maps to
Rc = 0	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evsetgthu d,a,b**
Rc = 1	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evsetgthu. d,a,b**

__ev_setgtws[_rc] __ev_setgtws[_rc]

Vector Set if Greater Than Word Signed [and Record]

d = __ev_setgtws (a,b) (Rc = 0)

d = __ev_setgtws_rc (a,b) (Rc = 1)

```
if a0:31 >si b0:31 then d0:31 = 321 else d0:31 = 320
if a32:63 >si b32:63 then d32:63 = 321 else d32:63 = 320
c0 ←(a0:31 >si b0:31)
c1 ←(a32:63 >si b32:63)
c2 ←(a0:31 >si b0:31) | (a32:63 >si b32:63)
c3 ←(a0:31 >si b0:31) & (a32:63 >si b32:63)
if (Rc = 1) then CR0:3 ←c0 || c1 || c2 || c3
```

Each signed word element in parameter **a** is compared to the corresponding signed word element in parameter **b**. The corresponding element of parameter **d** is set to all ‘1’s if the element in parameter **a** is greater than the element in parameter **b**, and is cleared to all ‘0’s otherwise.

If Rc=1, then **cr**0 is updated according to the comparison results. The most significant bit in **cr**0 is set if the high-order element of parameter **a** is greater than the high-order element of parameter **b**; it is cleared otherwise. The next bit in **cr**0 is set if the low-order element of parameter **a** is greater than the low-order element of parameter **b**; it is cleared otherwise. The third bit of **cr**0 is set if any element of parameter **a** is greater than the corresponding element of parameter **b**; it is cleared otherwise. The last bit of **cr**0 is set if both elements of parameter **a** are greater than the corresponding elements of parameter **b**; it is cleared otherwise.

Other registers altered: Condition Register (CR0) (if Rc=1)

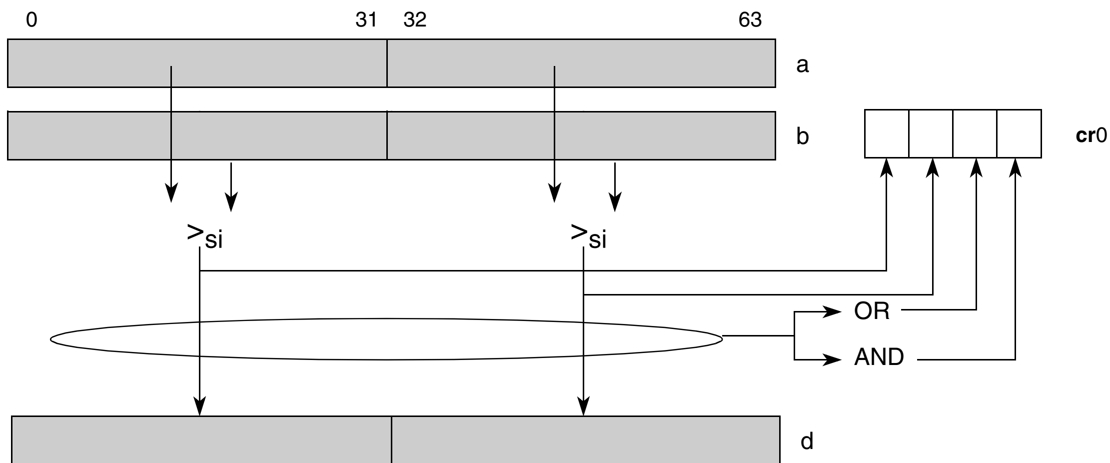


Figure 3-623. Vector Set if Greater than Word Signed (__ev_setgtws[_rc])

Rc	d	a	b	Maps to
Rc = 0	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evsetgtws d,a,b**
Rc = 1	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evsetgtws. d,a,b**

__ev_setgtwu[_rc] __ev_setgtwu[_rc]

Vector Set if Greater Than Word Unsigned [and Record]

d = __ev_setgtwu (a,b) (Rc = 0)

d = __ev_setgtwu_rc (a,b) (Rc = 1)

```
if a0:31 >ui b0:31 then d0:31 = 321 else d0:31 = 320
if a32:63 >ui b32:63 then d32:63 = 321 else d32:63 = 320
c0 ←(a0:31 >ui b0:31)
c1 ←(a32:63 >ui b32:63)
c2 ←(a0:31 >ui b0:31) | (a32:63 >ui b32:63)
c3 ←(a0:31 >ui b0:31) & (a32:63 >ui b32:63)
if (Rc = 1) then CR0:3 ←c0 || c1 || c2 || c3
```

Each unsigned word element in parameter **a** is compared to the corresponding unsigned word element in parameter **b**. The corresponding element of parameter **d** is set to all ‘1’s if the element in parameter **a** is greater than the element in parameter **b**, and is cleared to all ‘0’s otherwise.

If Rc=1, then **cr**0 is updated according to the comparison results. The most significant bit in **cr**0 is set if the high-order element of parameter **a** is greater than the high-order element of parameter **b**; it is cleared otherwise. The next bit in **cr**0 is set if the low-order element of parameter **a** is greater than the low-order element of parameter **b**; it is cleared otherwise. The third bit of **cr**0 is set if any element of parameter **a** is greater than the corresponding element of parameter **b**; it is cleared otherwise. The last bit of **cr**0 is set if both elements of parameter **a** are greater than the corresponding elements of parameter **b**; it is cleared otherwise.

Other registers altered: Condition Register (CR0) (if Rc=1)

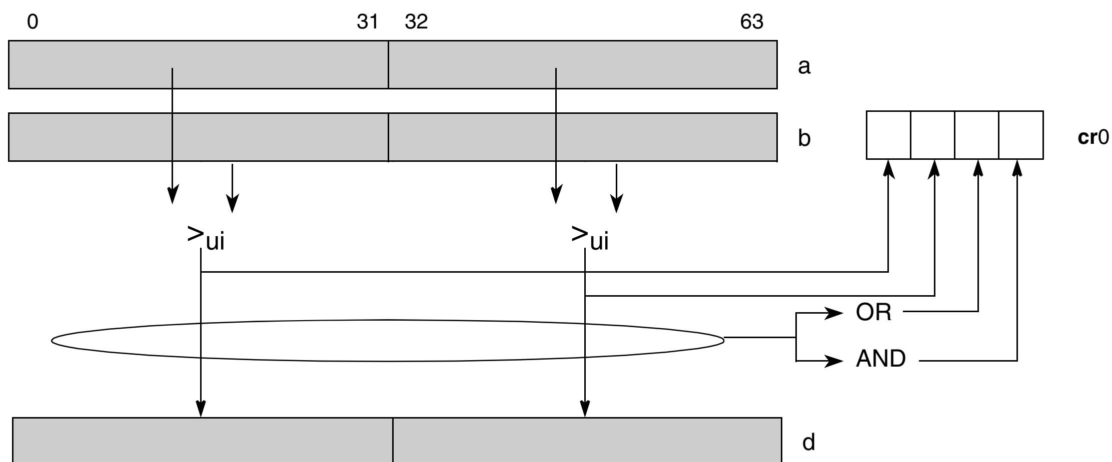


Figure 3-624. Vector Set if Greater than Word Unsigned (__ev_setgtwu[_rc])

Rc	d	a	b	Maps to
Rc = 0	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evsetgtwu d,a,b**
Rc = 1	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evsetgtwu. d,a,b**

__ev_setltbs[_rc] __ev_setltbs[_rc]

Vector Set if Greater Than Byte Signed [and Record]

d = __ev_setltbs (a,b) (Rc = 0)

d = __ev_setltbs_rc (a,b) (Rc = 1)

```
do i=0 to 63 by 8
    if a_i:i+7 <si b_i:i+7 then d_i:i+7 = 8_1 else d_i:i+7 = 8_0
end
c0 ←(a_0:7 <si b_0:7) | (a_8:15 <si b_8:15) | (a_16:23 <si b_16:23) | (a_24:31 <si b_24:31)
c1 ←(a_0:7 <si b_0:7) & (a_8:15 <si b_8:15) & (a_16:23 <si b_16:23) & (a_24:31 <si b_24:31)
c2 ←(a_0:7 <si b_0:7) | (a_8:15 <si b_8:15) | (a_16:23 <si b_16:23) | (a_24:31 <si b_24:31) |
      (a_32:39 <si b_32:39) | (a_40:47 <si b_40:47) | (a_48:55 <si b_48:55) | (a_56:63 <si b_56:63)
c3 ←(a_0:7 <si b_0:7) & (a_8:15 <si b_8:15) & (a_16:23 <si b_16:23) & (a_24:31 <si b_24:31) &
      (a_32:39 <si b_32:39) & (a_40:47 <si b_40:47) & (a_48:55 <si b_48:55) & (a_56:63 <si b_56:63)
if (Rc = 1) then CR_0:3 ←c0 || c1 || c2 || c3
```

Each signed byte element in parameter **a** is compared to the corresponding signed byte element in parameter **b**. The corresponding element of parameter **d** is set to all ‘1’s if the element in parameter **a** is greater than the element in parameter **b**, and is cleared to all ‘0’s otherwise.

If Rc=1, then **cr**0 is updated according to the comparison results. The most significant bit in **cr**0 is set if any of the high-order four elements of parameter **a** is less than the corresponding element of parameter **b**; it is cleared otherwise. The next bit in **cr**0 is set if all of the high-order four elements of parameter **a** are less than the corresponding element of parameter **b**; it is cleared otherwise. The third bit of **cr**0 is set if any element of parameter **a** is less than the corresponding element of parameter **b**; it is cleared otherwise. The last bit of **cr**0 is set if all of the elements of parameter **a** are less than the corresponding elements of parameter **b**; it is cleared otherwise.

Other registers altered: Condition Register (CR0) (if Rc=1)

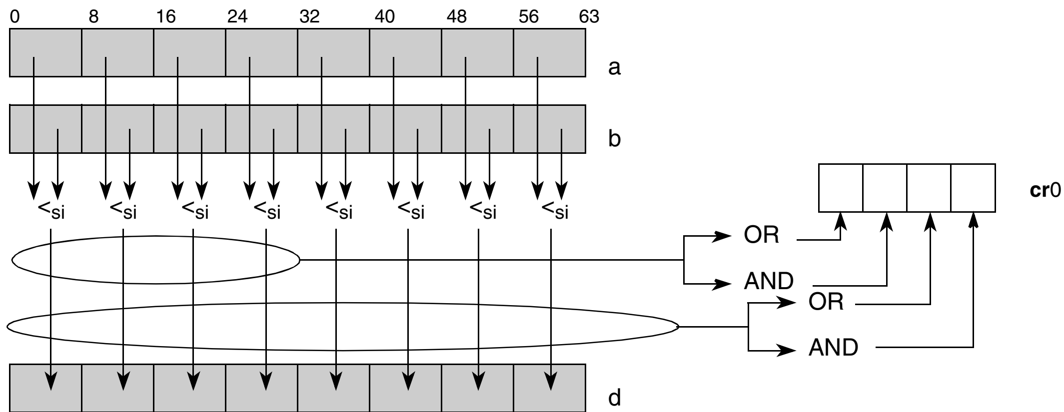


Figure 3-625. Vector Set if Less Than Byte Signed [and Record] (__ev_setltbs[_rc])

Rc	d	a	b	Maps to
Rc = 0	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evsetltbs d,a,b**
Rc = 1	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evsetltbs. d,a,b**

__ev_setltbu[_rc] __ev_setltbu[_rc]

Vector Set if Less Than Byte Unsigned [and Record]

d = __ev_setltbu (a,b) (Rc = 0)

d = __ev_setltbu_rc (a,b) (Rc = 1)

```
do i=0 to 63 by 8
    if a_{i:i+7} <ui b_{i:i+7} then d_{i:i+7} = ^8 1 else d_{i:i+7} = ^8 0
end
c0 ←(a_{0:7} <ui b_{0:7}) | (a_{8:15} <ui b_{8:15}) | (a_{16:23} <ui b_{16:23}) | (a_{24:31} <ui b_{24:31})
c1 ←(a_{0:7} <ui b_{0:7}) & (a_{8:15} <ui b_{8:15}) & (a_{16:23} <ui b_{16:23}) & (a_{24:31} <ui b_{24:31})
c2 ←(a_{0:7} <ui b_{0:7}) | (a_{8:15} <ui b_{8:15}) | (a_{16:23} <ui b_{16:23}) | (a_{24:31} <ui b_{24:31}) |
      (a_{32:39} <ui b_{32:39}) | (a_{40:47} <ui b_{40:47}) | (a_{48:55} <ui b_{48:55}) | (a_{56:63} <ui b_{56:63})
c3 ←(a_{0:7} <ui b_{0:7}) & (a_{8:15} <ui b_{8:15}) & (a_{16:23} <ui b_{16:23}) & (a_{24:31} <ui b_{24:31}) &
      (a_{32:39} <ui b_{32:39}) & (a_{40:47} <ui b_{40:47}) & (a_{48:55} <ui b_{48:55}) & (a_{56:63} <ui b_{56:63})
if (Rc = 1) then CR_{0:3} ←c0 || c1 || c2 || c3
```

Each unsigned byte element in parameter **a** is compared to the corresponding unsigned byte element in parameter **b**. The corresponding element of parameter **d** is set to all ‘1’s if the element in parameter **a** is less than the element in parameter **b**, and is cleared to all ‘0’s otherwise.

If Rc=1, then **cr**0 is updated according to the comparison results. The most significant bit in **cr**0 is set if any of the high-order four elements of parameter **a** is less than the corresponding element of parameter **b**; it is cleared otherwise. The next bit in **cr**0 is set if all of the high-order four elements of parameter **a** are less than the corresponding element of parameter **b**; it is cleared otherwise. The third bit of **cr**0 is set if any element of parameter **a** is less than the corresponding element of parameter **b**; it is cleared otherwise. The last bit of **cr**0 is set if all of the elements of parameter **a** are less than the corresponding elements of parameter **b**; it is cleared otherwise.

Other registers altered: Condition Register (CR0) (if Rc=1)

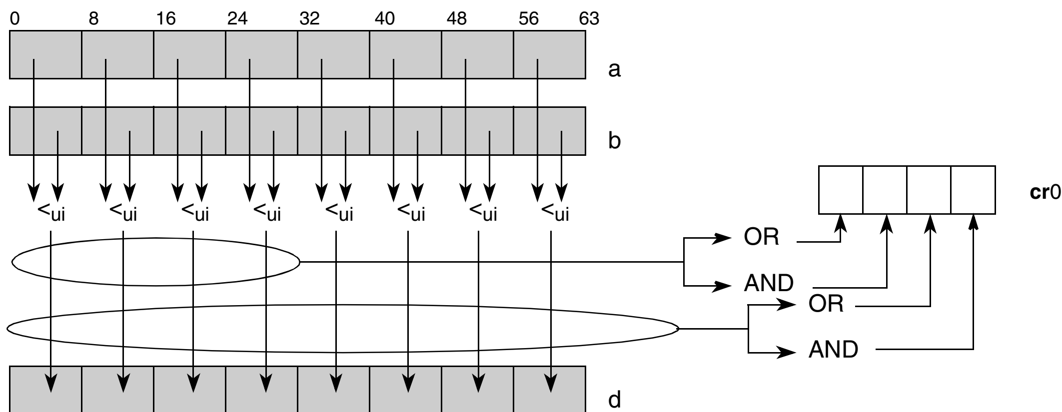


Figure 3-626. Vector Set if Less Than Byte Unsigned [and Record] (__ev_setltbu[_rc])

Rc	d	a	b	Maps to
Rc = 0	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evsetltbu d,a,b**
Rc = 1	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evsetltbu. d,a,b**

__ev_setlths[_rc] __ev_setlths[_rc]

Vector Set if Greater Than Half Word SIgned [and Record]

d = __ev_setlths (a,b) (Rc = 0)

d = __ev_setlths_rc (a,b) (Rc = 1)

```
do i=0 to 63 by 16
    if a[i:i+15] <si b[i:i+15] then d[i:i+15] = 16(1) else d[i:i+15] = 16(0)
end
c0 ←(a[0:15] <si b[0:15]) | (a[16:31] <si b[16:31])
c1 ←(a[0:15] <si b[0:15]) & (a[16:31] <si b[16:31])
c2 ←(a[0:15] <si b[0:15]) | (a[16:31] <si b[16:31]) | (a[32:47] <si b[32:47]) | (a[48:63] <si b[48:63])
c3 ←(a[0:15] <si b[0:15]) & (a[16:31] <si b[16:31]) & (a[32:47] <si b[32:47]) & (a[48:63] <si b[48:63])
if (Rc = 1) then CR[0:3] ←c0 || c1 || c2 || c3
```

Each signed half word element in parameter **a** is compared to the corresponding signed element in parameter **b**. The corresponding element of parameter **d** is set to all ‘1’s if the element in parameter **a** is less than the element in parameter **b**, and is cleared to all ‘0’s otherwise.

If Rc=1, then **cr**0 is updated according to the comparison results. The most significant bit in **cr**0 is set if either of the high-order two elements of parameter **a** is less than the corresponding element of parameter **b**; it is cleared otherwise. The next bit in **cr**0 is set if both of the high-order two elements of parameter **a** are less than the corresponding element of parameter **b**; it is cleared otherwise. The third bit of **cr**0 is set if any element of parameter **a** is less than the corresponding element of parameter **b**; it is cleared otherwise. The last bit of **cr**0 is set if all of the elements of parameter **a** are less than the corresponding elements of parameter **b**; it is cleared otherwise.

Other registers altered: Condition Register (CR0) (if Rc=1)

Figure 3-627. Vector Set if Less than Half Word Signed [and Record] (__ev_setlths[_rc])

Rc	d	a	b	Maps to
Rc = 0	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evsetlths d,a,b**
Rc = 1	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evsetlths. d,a,b**

__ev_setlthu[_rc] __ev_setlthu[_rc]

Vector Set if Less Than Half Word Unsigned [and Record]

d = __ev_setlthu (a,b) (Rc = 0)

d = __ev_setlthu_rc (a,b) (Rc = 1)

```
do i=0 to 63 by 16
    if a_{i:i+15} <ui b_{i:i+15} then d_{i:i+15} = 16_1 else d_{i:i+15} = 16_0
end
c0 ←(a_{0:15} <ui b_{0:15}) | (a_{16:31} <ui b_{16:31})
c1 ←(a_{0:15} <ui b_{0:15}) & (a_{16:31} <ui b_{16:31})
c2 ←(a_{0:15} <ui b_{0:15}) | (a_{16:31} <ui b_{16:31}) | (a_{32:47} <ui b_{32:47}) | (a_{48:63} <ui b_{48:63})
c3 ←(a_{0:15} <ui b_{0:15}) & (a_{16:31} <ui b_{16:31}) & (a_{32:47} <ui b_{32:47}) & (a_{48:63} <ui b_{48:63})
if (Rc = 1) then CR_{0:3} ←c0 || c1 || c2 || c3
```

Each unsigned half word element in parameter **a** is compared to the corresponding unsigned element in parameter **b**. The corresponding element of parameter **d** is set to all ‘1’s if the element in parameter **a** is less than the element in parameter **b**, and is cleared to all ‘0’s otherwise.

If Rc=1, then **cr**0 is updated according to the comparison results. The most significant bit in **cr**0 is set if either of the high-order two elements of parameter **a** is less than the corresponding element of parameter **b**; it is cleared otherwise. The next bit in **cr**0 is set if both of the high-order two elements of parameter **a** are less than the corresponding element of parameter **b**; it is cleared otherwise. The third bit of **cr**0 is set if any element of parameter **a** is less than the corresponding element of parameter **b**; it is cleared otherwise. The last bit of **cr**0 is set if all of the elements of parameter **a** are less than the corresponding elements of parameter **b**; it is cleared otherwise.

Other registers altered: Condition Register (CR0) (if Rc=1)

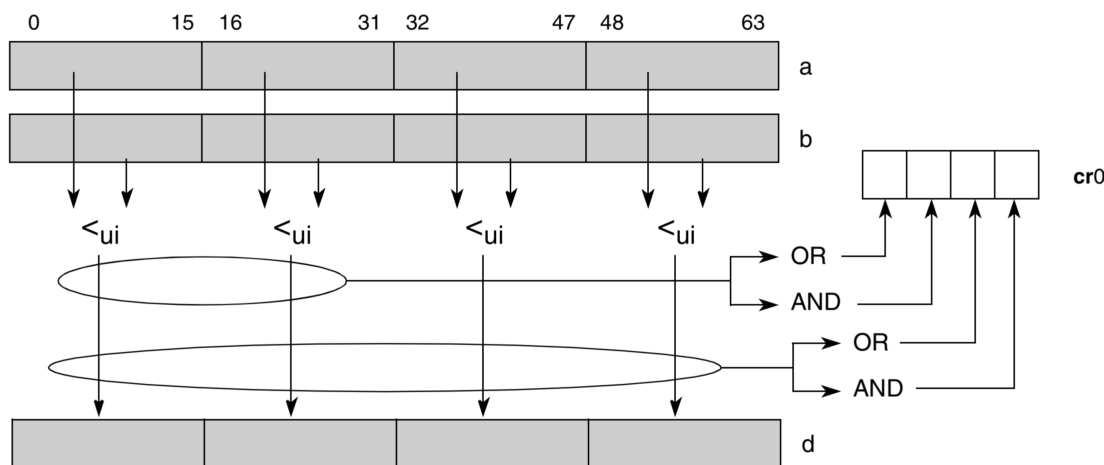


Figure 3-628. Vector Set if Less than Half Word Unsigned [and Record] (__ev_setlthu[_rc])

Rc	d	a	b	Maps to
Rc = 0	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evsetlthu d,a,b**
Rc = 1	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evsetlthu. d,a,b**

__ev_setltws[_rc] __ev_setltws[_rc]

Vector Set if Less Than Word Signed [and Record]

d = __ev_setltws (a,b) (Rc = 0)

d = __ev_setltws_rc (a,b) (Rc = 1)

```
if a0:31 <si b0:31 then d0:31 = 321 else d0:31 = 320
if a32:63 <si b32:63 then d32:63 = 321 else d32:63 = 320
c0 ←(a0:31 <si b0:31)
c1 ←(a32:63 <si b32:63)
c2 ←(a0:31 <si b0:31) | (a32:63 <si b32:63)
c3 ←(a0:31 <si b0:31) & (a32:63 <si b32:63)
if (Rc = 1) then CR0:3 ←c0 || c1 || c2 || c3
```

Each signed word element in parameter **a** is compared to the corresponding signed word element in parameter **b**. The corresponding element of parameter **d** is set to all ‘1’s if the element in parameter **a** is less than the element in parameter **b**, and is cleared to all ‘0’s otherwise.

If Rc=1, then **cr**0 is updated according to the comparison results. The most significant bit in **cr**0 is set if the high-order element of parameter **a** is less than the high-order element of parameter **b**; it is cleared otherwise. The next bit in **cr**0 is set if the low-order element of parameter **a** is less than the low-order element of parameter **b**; it is cleared otherwise. The third bit of **cr**0 is set if any element of parameter **a** is less than the corresponding element of parameter **b**; it is cleared otherwise. The last bit of **cr**0 is set if both elements of parameter **a** are less than the corresponding elements of parameter **b**; it is cleared otherwise.

Other registers altered: Condition Register (CR0) (if Rc=1)

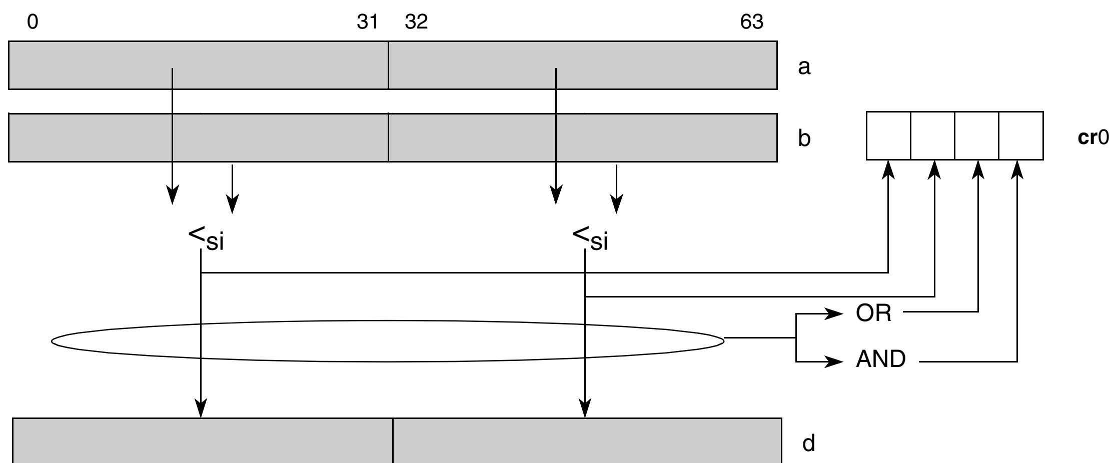


Figure 3-629. Vector Set if Less than Word Signed (__ev_setltws[_rc])

Rc	d	a	b	Maps to
Rc = 0	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evsetltws d,a,b**
Rc = 1	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evsetltws. d,a,b**

__ev_setltwu[_rc] __ev_setltwu[_rc]

Vector Set if Less Than Word Unsigned [and Record]

d = __ev_setltwu (a,b) (Rc = 0)

d = __ev_setltwu_rc (a,b) (Rc = 1)

$$\begin{array}{l}
\text{if } a_{0:31} <_{ui} b_{0:31} \text{ then } d_{0:31} = {}^{16}1 \text{ else } d_{0:31} = {}^{16}0 \\
\text{if } a_{32:63} <_{ui} b_{32:63} \text{ then } d_{32:63} = {}^{16}1 \text{ else } d_{32:63} = {}^{16}0 \\
c0 \leftarrow (a_{0:31} <_{ui} b_{0:31}) \\
c1 \leftarrow (a_{32:63} <_{ui} b_{32:63}) \\
c2 \leftarrow (a_{0:31} <_{ui} b_{0:31}) \mid (a_{32:63} <_{ui} b_{32:63}) \\
c3 \leftarrow (a_{0:31} <_{ui} b_{0:31}) \;\&\; (a_{32:63} <_{ui} b_{32:63}) \\
\text{if } (Rc = 1) \text{ then } CR_{0:3} \leftarrow c0 \;||\; c1 \;||\; c2 \;||\; c3
\end{array}$$

Each unsigned word element in parameter **a** is compared to the corresponding unsigned word element in parameter **b**. The corresponding element of parameter **d** is set to all ‘1’s if the element in parameter **a** is less than the element in parameter **b**, and is cleared to all ‘0’s otherwise.

If Rc=1, then **cr**0 is updated according to the comparison results. The most significant bit in **cr**0 is set if the high-order element of parameter **a** is less than the high-order element of parameter **b**; it is cleared otherwise. The next bit in **cr**0 is set if the low-order element of parameter **a** is less than the low-order element of parameter **b**; it is cleared otherwise. The third bit of **cr**0 is set if any element of parameter **a** is less than the corresponding element of parameter **b**; it is cleared otherwise. The last bit of **cr**0 is set if both elements of parameter **a** are less than the corresponding elements of parameter **b**; it is cleared otherwise.

Other registers altered: Condition Register (CR0) (if Rc=1)

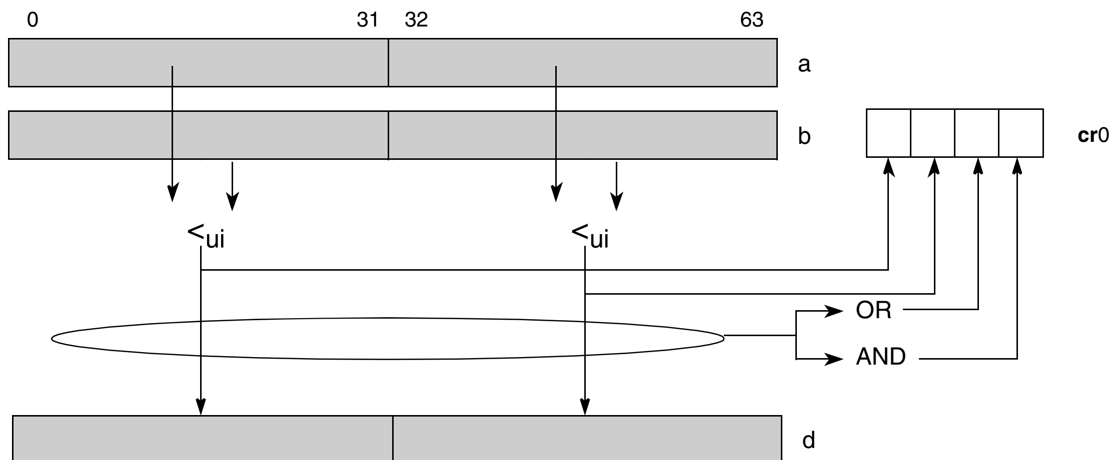


Figure 3-630. Vector Set if Less than Word Unsigned (__ev_setltwu[_rc])

Rc	d	a	b	Maps to
Rc = 0	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evsetltwu d,a,b**
Rc = 1	__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evsetltwu. d,a,b**

__ev_sl

__ev_sl

Vector Shift Left

d = __ev_sl (a,b)

```
sh ←b57:63

if b57 = 0 then
    d0:63 ←EXTZ(a0:63-sh)
else
    d ←64 0
```

The value in parameter **a** is shifted left by ‘sh’ bit positions specified in parameter $\mathbf{b}_{57:63}$, filling vacated bit positions with zeros, and the result is placed into parameter **d**. Shift amounts of 64 to 127 give a zero result.

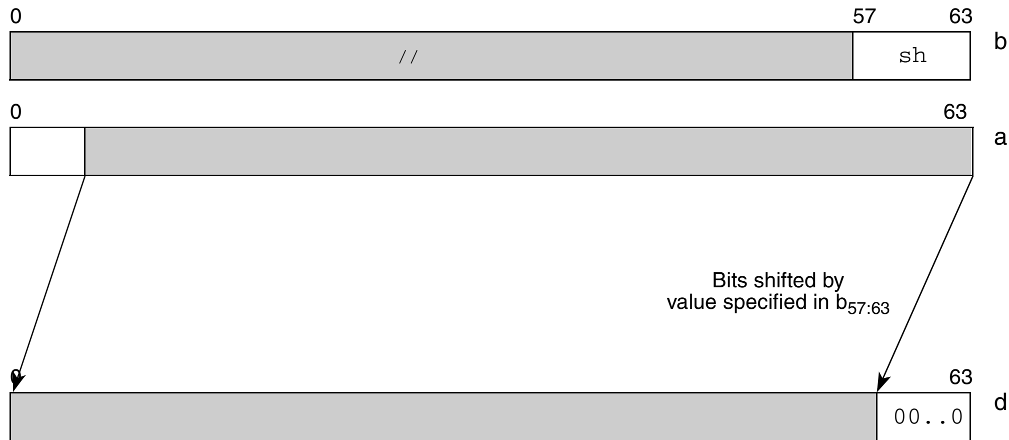


Figure 3-631. Vector Shift Left (__ev_sl)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evsl d,a,b**

__ev_sli

__ev_sli

Vector Shift Left Immediate

d = __ev_sli (a,b)

```
UIMM ←b
n ←UIMM

d0:63 ←EXTZ(a0:63-n)
```

The value in parameter **a** is shifted left by ‘n’ bit positions specified by the 5-bit unsigned literal (UIMM) provided by parameter **b**, filling vacated bit positions with zeros, and the result is placed into parameter **d**.

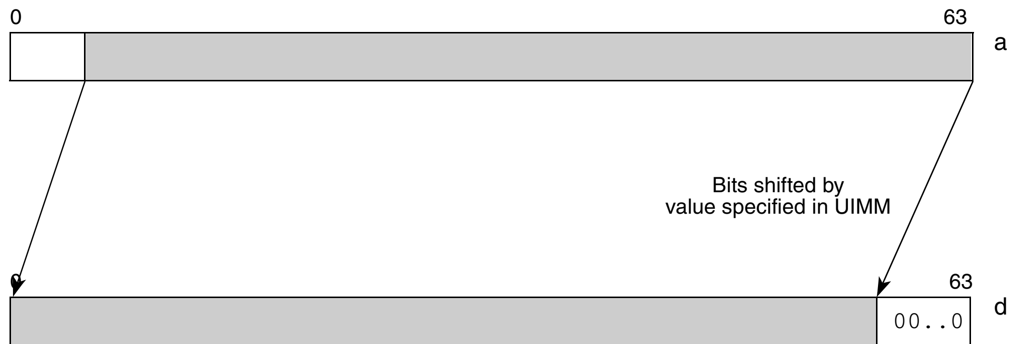


Figure 3-632. Vector Shift Left Immediate (__ev_sli)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	5-bit unsigned literal	**evsli d,a,b**

__ev_slb __ev_slb

Vector Shift Left Byte

d = __ev_slb (a,b)

$$
\begin{aligned}
&nb0 \leftarrow b_{4:7}\\
&nb1 \leftarrow b_{12:15}\\
&nb2 \leftarrow b_{20:23}\\
&nb3 \leftarrow b_{28:31}\\
&nb4 \leftarrow b_{36:39}\\
&nb5 \leftarrow b_{44:47}\\
&nb6 \leftarrow b_{52:55}\\
&nb7 \leftarrow b_{60:63}\\
\\
&d_{0:7} \leftarrow SL(a_{0:7},\ nb0)\\
&d_{8:15} \leftarrow SL(a_{8:15},\ nb1)\\
&d_{16:23} \leftarrow SL(a_{16:23},\ nb2)\\
&d_{24:31} \leftarrow SL(a_{24:31},\ nb3)\\
&d_{32:39} \leftarrow SL(a_{32:39},\ nb4)\\
&d_{40:47} \leftarrow SL(a_{40:47},\ nb5)\\
&d_{48:55} \leftarrow SL(a_{48:55},\ nb6)\\
&d_{56:63} \leftarrow SL(a_{56:63},\ nb7)
\end{aligned}
$$

Each of the byte elements of parameter **a** are shifted left by an amount specified in the corresponding byte elements of parameter **b**. The result is placed into parameter **d**. The separate shift amounts for each element are specified by the lower 5 bits in each byte element of parameter **b** that lie in bit positions 4-7, 12-15, 20-23, 28-31, 36-39, 44-47, 52-55, and 60-63.

Shift amounts from 8 to 15 give a zero result.

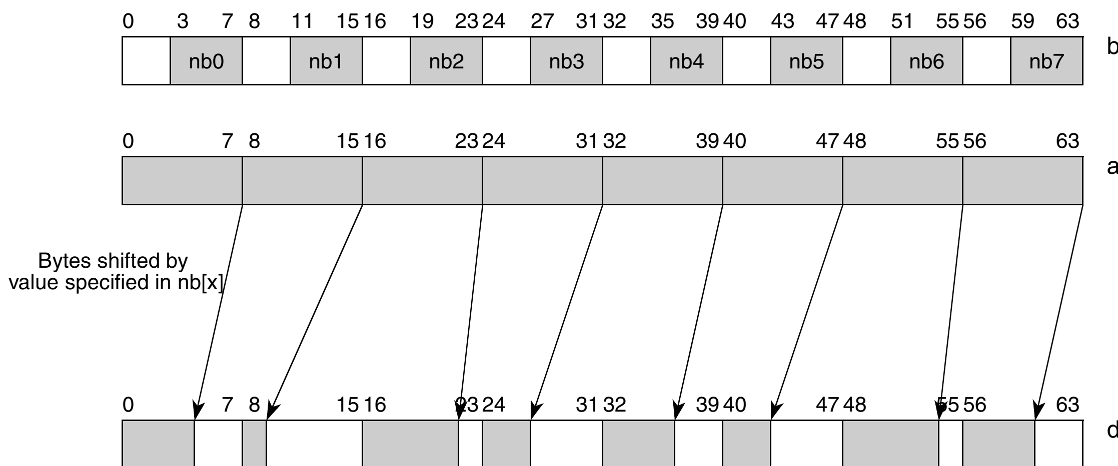


Figure 3-633. Vector Shift Left Byte (__ev_slb)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evslb d,a,b**

__ev_slbi

__ev_slbi

Vector Shift Left Byte Immediate

d = __ev_slbi (a,b)

```
UIMM ←b
n ←UIMM

d0:7 ←SL(a0:7, n)
d8:15 ←SL(a8:15, n)
d16:23 ←SL(a16:23, n)
d24:31 ←SL(a24:31, n)
d32:39 ←SL(a32:39, n)
d40:47 ←SL(a40:47, n)
d48:55 ←SL(a48:55, n)
d56:63 ←SL(a56:63, n)
```

Each of the byte elements of parameter **a** are shifted left by the unsigned literal value (UIMM) provided by parameter **b** (range of 0-7) and the results are placed in parameter **d**.

Shift amounts greater than 7 are illegal.

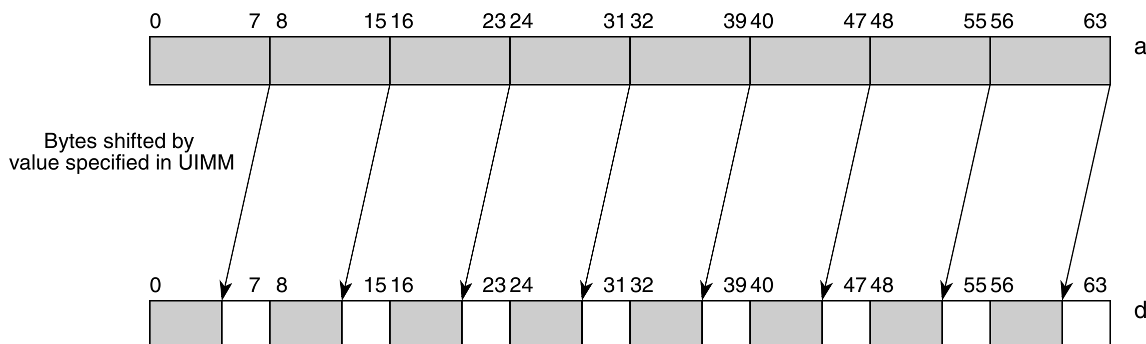


Figure 3-634. Vector Shift Left Byte Immediate (__ev_slbi)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	5-bit unsigned literal	**evslbi d,a,b**

__ev_slh __ev_slh

Vector Shift Left Half Word

d = __ev_slh (a,b)

$$nh0 \leftarrow b_{11:15}$$
$$nh1 \leftarrow b_{27:31}$$
$$nh2 \leftarrow b_{43:47}$$
$$nh3 \leftarrow b_{59:63}$$

$$d_{0:15} \leftarrow SL(a_{0:15},\ nh0)$$
$$d_{16:31} \leftarrow SL(a_{16:31},\ nh1)$$
$$d_{32:47} \leftarrow SL(a_{32:47},\ nh2)$$
$$d_{48:63} \leftarrow SL(a_{48:63},\ nh3)$$

Each of the half word elements of parameter **a** are shifted left by an amount specified in the corresponding half word elements of parameter **b**. The result is placed into parameter **d**. The separate shift amounts for each element are specified by 5 bits in parameter **b** that lie in bit positions 11-15, 27-31, 43-47 and 59-63.

Shift amounts from 16 to 31 give a zero result.

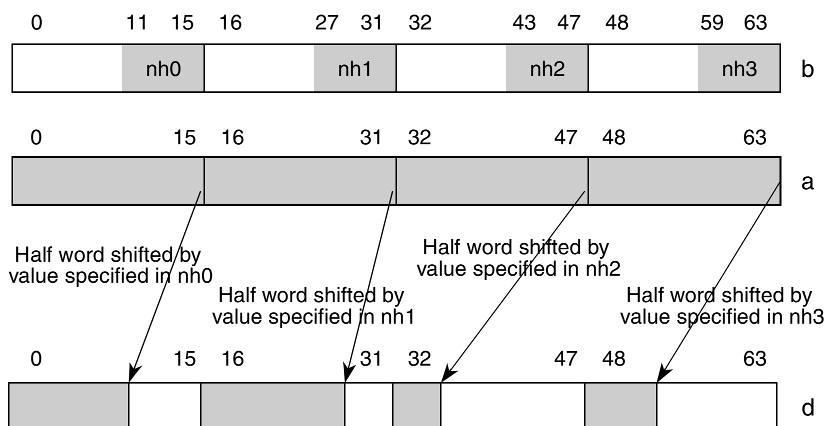


Figure 3-635. Vector Shift Left Half Word (__ev_slh)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evslh d,a,b**

__ev_slhi

__ev_slhi

Vector Shift Left Half Word Immediate

d = __ev_slhi (a,b)

```
UIMM ←b
n ←UIMM
d0:15 ←SL(a0:15, n)
d16:31 ←SL(a16:31, n)
d32:47 ←SL(a32:47, n)
d48:63 ←SL(a48:63, n)
```

Each of the half word elements of parameter **a** are shifted left by the unsigned literal value (UIMM) provided by parameter **b** (range of 0-15) and the results are placed in parameter **d**.

Shift amounts greater than 15 are illegal.

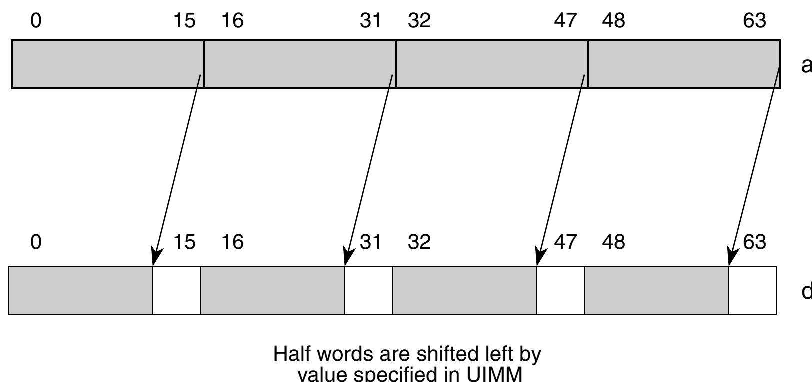


Figure 3-636. Vector Shift Left Half Word (__ev_slhi)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	5-bit unsigned literal	**evslhi d,a,b**

__ev_sloi __ev_sloi

Vector Shift Left by Octet Immediate

d = __ev_sloi (a,b)

```
n ←b * 8
d0:63 ←SL(a0:63, n)
```

The value in parameter **a** is shifted left by the number of byte positions contained in parameter **b**, filling vacated byte positions with zeros, and the result is placed into parameter **d**.

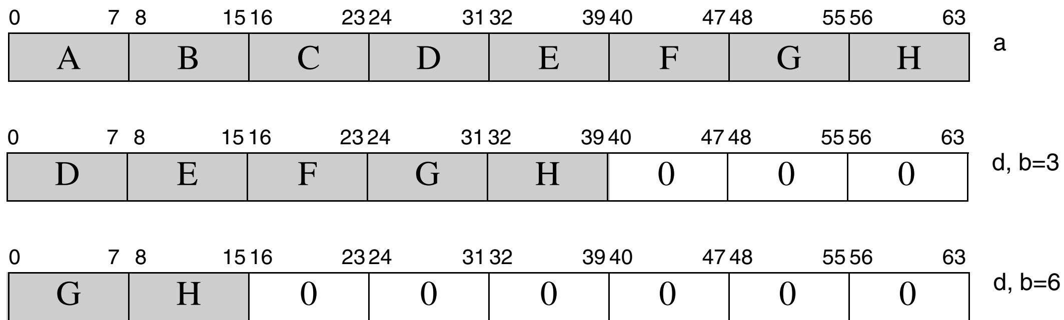


Figure 3-637. Vector Shift Left by Octet Immediate (__ev_sloi)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	3-bit unsigned literal	**evsloi d,a,b**

__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.

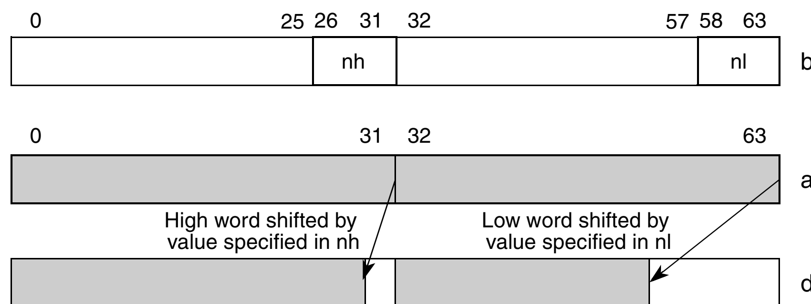


Figure 3-638. Vector Shift Left Word (__ev_slw)

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)

```
UIMM ←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 unsigned literal (UIMM) provided by parameter **b**, and the results are placed in parameter **d**.

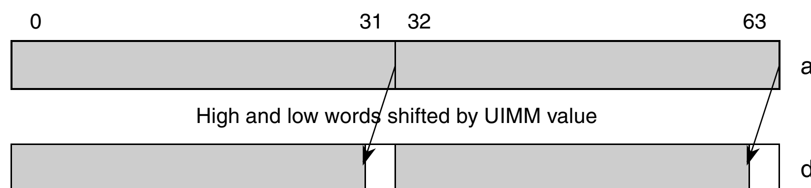


Figure 3-639. Vector Shift Left Word Immediate (__ev_slwi)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	5-bit unsigned literal	**evslwi d,a,b**

__ev_splatb __ev_splatb

Vector Splat Byte

d = __ev_splatb (a,b)

```
temp0:7 ←a0+(b*8):7+(b*8)
d0:7 ←temp0:7
d8:15 ←temp0:7
d16:23 ←temp0:7
d24:31 ←temp0:7
d32:39 ←temp0:7
d40:47 ←temp0:7
d48:55 ←temp0:7
d56:63 ←temp0:7
```

The byte element in parameter **a** specified by parameter **b** is placed into all byte elements of parameter **d**. Byte 0 is the most-significant byte.

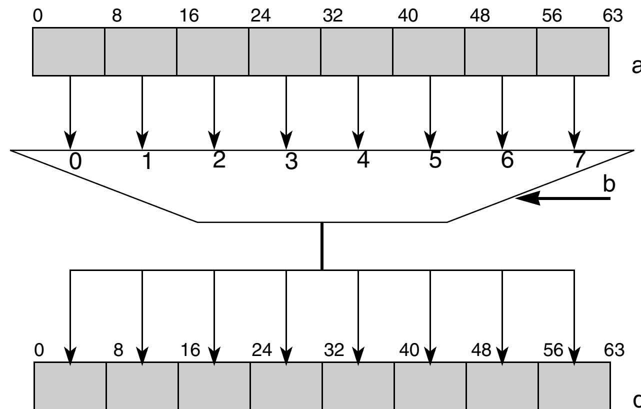


Figure 3-640. Vector Splat Byte (__ev_splatb)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	3-bit unsigned literal	**evsplatb d,a,b**

__ev_splatfi[a] __ev_splatfi[a]

Vector Splat Fractional Immediate (to Accumulator)

d = __ev_splatfi (a) **(A = 0)**

d = __ev_splatfia (a) **(A = 1)**

```
SIMM ←a
d0:31 ←SIMM || 270
d32:63 ←SIMM || 270

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

The 5-bit immediate value is padded with trailing zeros and placed in both elements of parameter **d**, as shown in Figure 3-641. The SIMM ends up in bit positions $\mathbf{d}_{0:4}$ and $\mathbf{d}_{32:36}$. If A = 1, the result in parameter **d** is also placed into the accumulator.

Other registers altered: ACC (if A=1)

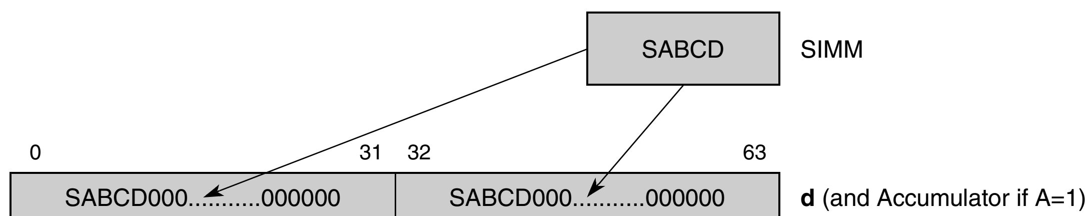


Figure 3-641. Vector Splat Fractional Immediate (to Accumulator) (__ev_splatfi[a])

d	a	Maps to
__ev64_opaque__	5-bit signed literal	**evsplatfi d,a**
__ev64_opaque__	5-bit signed literal	**evsplatfia d,a**

__ev_splatfib[a] __ev_splatfib[a]

Vector Splat Fractional Immediate Byte (to Accumulator)

d = __ev_splatfib (a) **(A = 0)**

d = __ev_splatfiba (a) **(A = 1)**

```
SIMM ←a
d0:7 ←SIMM || 30
d8:15 ←SIMM || 30
d16:23 ←SIMM || 30)
d24:31 ←SIMM || 30
d32:39 ←SIMM || 30
d40:47 ←SIMM || 30
d48:55 ←SIMM || 30
d56:63 ←SIMM || 30

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

The 5-bit immediate value is padded with trailing zeros and placed into all byte elements of parameter **d**. If A = 1, the result in parameter **d** is also placed into the accumulator.

Other registers altered: ACC (if A=1)

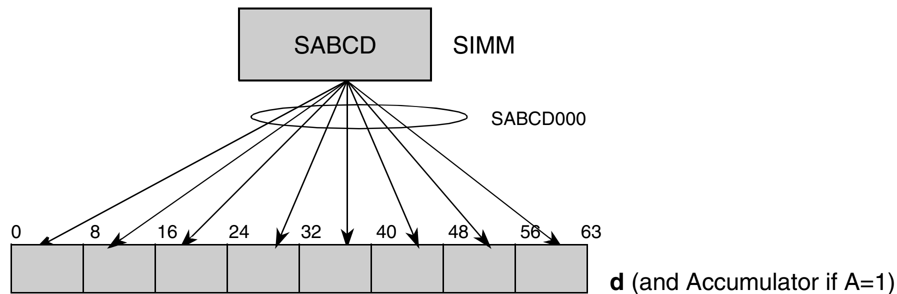


Figure 3-642. Vector Splat Fractional Immediate Byte (to Accumulator) (__ev_splatfib[a])

d	a	Maps to
__ev64_opaque__	5-bit signed literal	**evsplatfib d,a**
__ev64_opaque__	5-bit signed literal	**evsplatfiba d,a**

__ev_splatfibo[a] __ev_splatfibo[a]

Vector Splat Fractional Immediate Byte Odd (to Accumulator)

d = __ev_splatfibo (a) **(A = 0)**

d = __ev_splatfiboa (a) **(A = 1)**

```
SIMM ←a
d0:7 ←80
d8:15 ←SIMM || 30
d16:23 ←80
d24:31 ←SIMM || 30
d32:39 ←80
d40:47 ←SIMM || 30
d48:55 ←80
d56:63 ←SIMM || 30

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

The 5-bit immediate value is padded with trailing zeros and placed into all odd byte elements of parameter **d**, zeroing the even byte elements. If A = 1, the result in parameter **d** is also placed into the accumulator.

Other registers altered: ACC (if A=1)

Figure 3-643. Vector Splat Fractional Immediate Byte Odd (to Accumulator) (__ev_splatfibo[a])

d	a	Maps to
__ev64_opaque__	5-bit signed literal	**evsplatfibo d,a**
__ev64_opaque__	5-bit signed literal	**evsplatfiboa d,a**

__ev_splatfid[a] __ev_splatfid[a]

Vector Splat Fractional Immediate Doubleword (to Accumulator)

d = __ev_splatfid (a) **(A = 0)**

d = __ev_splatfida (a) **(A = 1)**

```
SIMM ←a
d0:63 ←SIMM || 590

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

The 5-bit immediate value is padded with trailing zeros and placed into parameter **d**. The SIMM ends up in bit positions $\mathbf{d}_{0:4}$. If A = 1, the result in parameter **d** is also placed into the accumulator.

Other registers altered: ACC (if A=1)

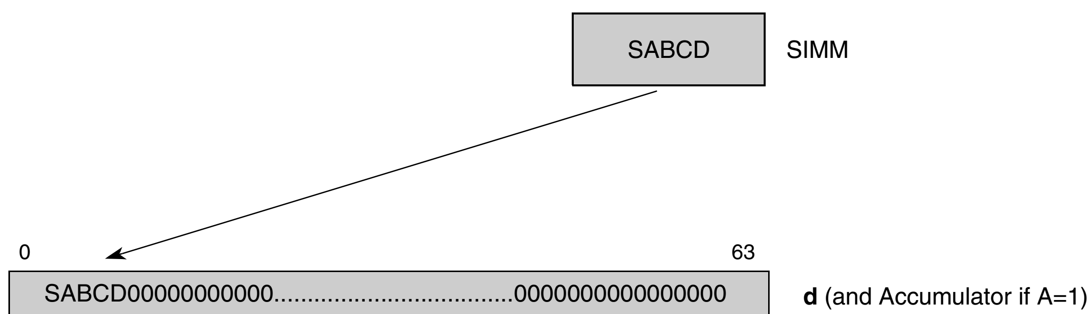


Figure 3-644. Vector Splat Fractional Immediate Doubleword (to Accumulator) (__ev_splatfid[a])

d	a	Maps to
__ev64_opaque__	5-bit signed literal	**evsplatfid d,a**
__ev64_opaque__	5-bit signed literal	**evsplatfida d,a**

__ev_splatfih[a] __ev_splatfih[a]

Vector Splat Fractional Immediate Half Word (to Accumulator)

d = __ev_splatfih (a) (A = 0)

d = __ev_splatfiha (a) (A = 1)

```
SIMM ←a
d0:15 ←SIMM || 110
d16:31 ←SIMM || 110
d32:47 ←SIMM || 110
d48:63 ←SIMM || 110

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

The 5-bit immediate value is padded with trailing zeros and placed into all halfword elements of parameter **d**, as shown in Figure 3-645. The SIMM ends up in bit positions $\mathbf{d}_{0:4}$, $\mathbf{d}_{16:20}$, $\mathbf{d}_{32:36}$, and $\mathbf{d}_{48:52}$. If A = 1, the result in parameter **d** is also placed into the accumulator.

Other registers altered: ACC (if A=1)

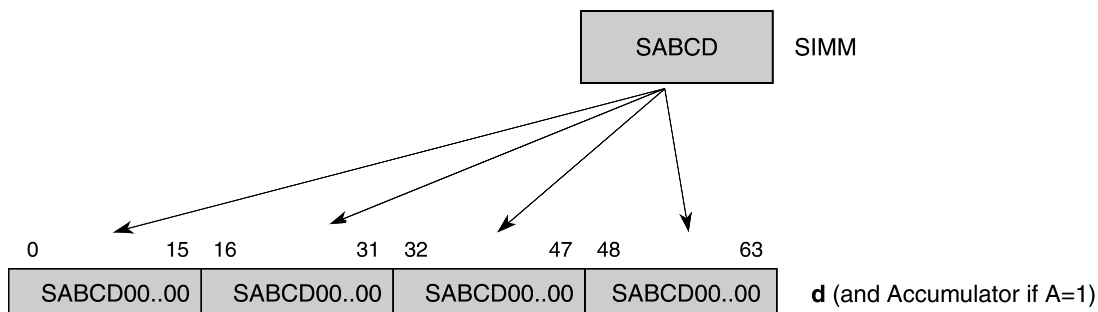


Figure 3-645. Vector Splat Fractional Immediate Halfword (to Accumulator) (__ev_splatfih[a])

d	a	Maps to
__ev64_opaque__	5-bit signed literal	**evsplatfih d,a**
__ev64_opaque__	5-bit signed literal	**evsplatfiha d,a**

__ev_splatfiho[a]   __ev_splatfiho[a]

Vector Splat Fractional Immediate Half Word Odd (to Accumulator)

d = __ev_splatfiho (a) (A = 0)

d = __ev_splatfihoa (a) (A = 1)

```
SIMM ←a
d0:15 ←16 0
d16:31 ←SIMM || 11 0
d32:47 ←16 0
d48:63 ←SIMM || 11 0

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

The 5-bit immediate value is padded with trailing zeros and placed into the odd halfword elements of parameter **d**, zeroing the even halfword elements. The SIMM ends up in bit positions $\mathbf{d}_{16:20}$ and $\mathbf{d}_{48:52}$. If A = 1, the result in parameter **d** is also placed into the accumulator.

Other registers altered: ACC (if A=1)

SABCD SIMM

0 15 16 31 32 47 48 63

0000........0000	SABCD00..00	0000........0000	SABCD00..00

d (and Accumulator if A=1)

Figure 3-646. Vector Splat Fractional Immediate Halfword Odd (to Accumulator) (__ev_splatfiho[a])

d	a	Maps to
__ev64_opaque__	5-bit signed literal	**evsplatfiho d,a**
__ev64_opaque__	5-bit signed literal	**evsplatfihoa d,a**

__ev_splatfio[a]

__ev_splatfio[a]

Vector Splat Fractional Immediate Odd (to Accumulator)

d = __ev_splatfio (a) (A = 0)

d = __ev_splatfioa (a) (A = 1)

```
SIMM ←a
d0:31 ←32 0
d32:63 ←SIMM || 27 0

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

The 5-bit immediate value is padded with trailing zeros and placed into the odd word element of parameter **d**, zeroing the even word element. The SIMM ends up in bit positions $\mathbf{d}_{32:36}$. If A = 1, the result in parameter **d** is also placed into the accumulator.

Other registers altered: ACC (if A=1)

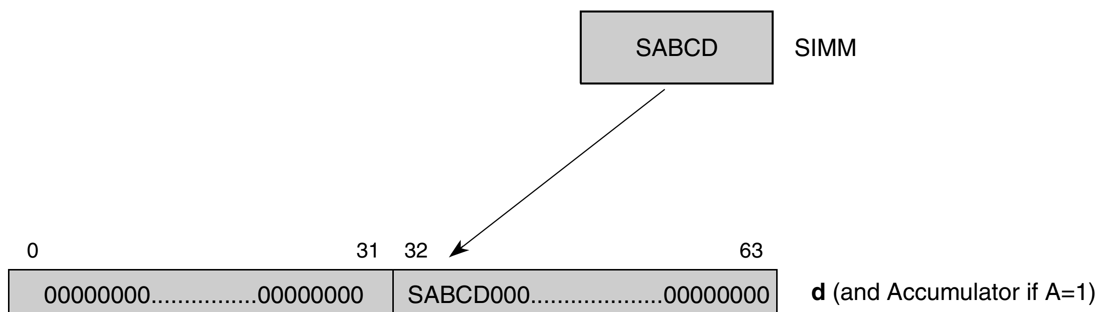


Figure 3-647. Vector Splat Fractional Immediate Odd (to Accumulator) (__ev_splatfio[a])

d	a	Maps to
__ev64_opaque__	5-bit signed literal	**evsplatfio d,a**
__ev64_opaque__	5-bit signed literal	**evsplatfioa d,a**

__ev_splath __ev_splath

Vector Splat Half Word

d = __ev_splath (a,b)

```
hh ←b
temp0:15 ←a0+(hh*16):15+(hh*16)
d0:15 ←temp0:15
d16:31 ←temp0:15
d32:47 ←temp0:15
d48:63 ←temp0:15
```

The half word element in parameter **a** specified by the 2-bit unsigned value in parameter **b** (“hh”) is placed into all half word elements of parameter **d**, as shown in Figure 3-648. Half word 0 is the most-significant half word.

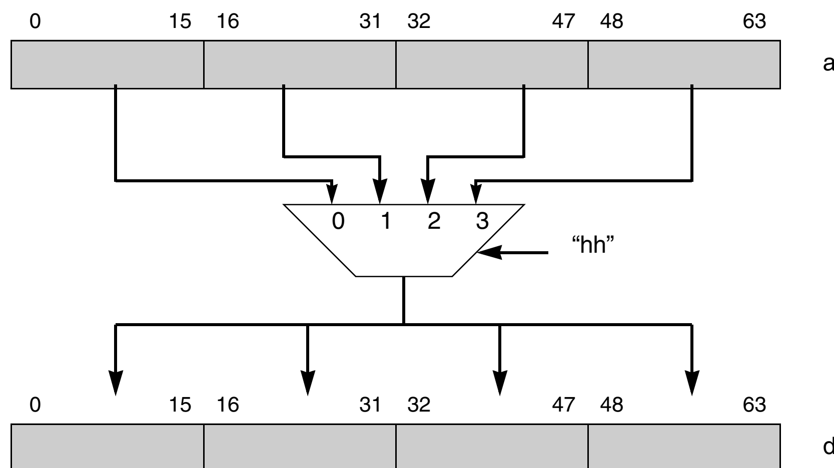


Figure 3-648. Vector Splat Half Word (__ev_splath)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	2-bit unsigned literal	**evsplath d,a,b**

__ev_splati[a] __ev_splati[a]

Vector Splat Immediate (to Accumulator)

d = __ev_splati (a) (A = 0)

d = __ev_splatia (a) (A = 1)

```
SIMM ←a
d0:31 ←EXTS32(SIMM)
d32:63 ←EXTS32(SIMM)

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

The 5-bit immediate value is sign extended and placed in both elements of parameter **d**. If A = 1, the result in parameter **d** is also placed into the accumulator.

Other registers altered: ACC (if A=1)

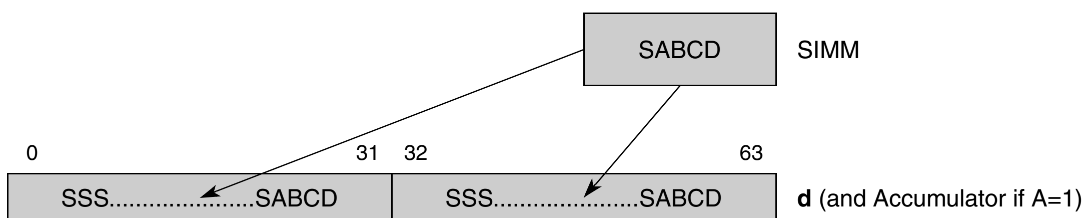


Figure 3-649. __ev_splati[a] Sign Extend (to Accumulator)

d	a	Maps to
__ev64_opaque__	5-bit signed literal	**evsplati d,a**
__ev64_opaque__	5-bit signed literal	**evsplatia d,a**

__ev_splatib[a] __ev_splatib[a]

Vector Splat Immediate Byte (to Accumulator)

d = __ev_splatib (a) (A = 0)

d = __ev_splatiba (a) (A = 1)

```
SIMM ←a
d0:7 ←EXTS8(SIMM)
d8:15 ←EXTS8(SIMM)
d16:23 ←EXTS8(SIMM)
d24:31 ←EXTS8(SIMM)
d32:39 ←EXTS8(SIMM)
d40:47 ←EXTS8(SIMM)
d48:55 ←EXTS8(SIMM)
d56:63 ←EXTS8(SIMM)

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

The 5-bit immediate value is sign-extended and placed into all byte elements of parameter **d**. If A = 1, the result in parameter **d** is also placed into the accumulator.

Other registers altered: ACC (if A=1)

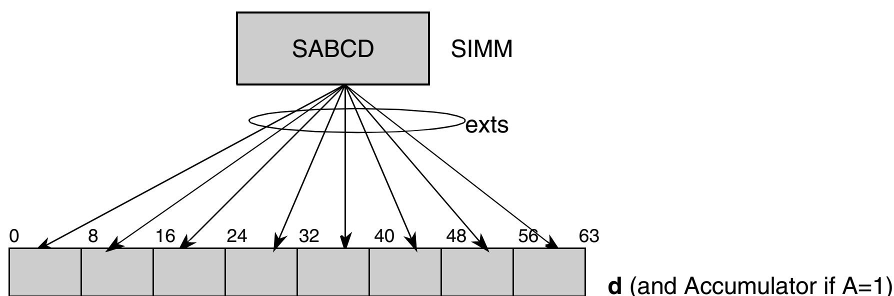


Figure 3-650. Vector Splat Immediate Byte (to Accumulator) (__ev_splatib[a])

d	a	Maps to
__ev64_opaque__	5-bit signed literal	**evsplatib d,a**
__ev64_opaque__	5-bit signed literal	**evsplatiba d,a**

__ev_splatibe[a] __ev_splatibe[a]

Vector Splat Immediate Byte Even (to Accumulator)

d = __ev_splatibe (a) (A = 0)

d = __ev_splatibea (a) (A = 1)

```
SIMM ←a
d0:7 ←EXTS8(SIMM)
d8:15 ←80
d16:23 ←EXTS8(SIMM)
d24:31 ←80
d32:39 ←EXTS8(SIMM)
d40:47 ←80
d48:55 ←EXTS8(SIMM)
d56:63 ←80

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

The 5-bit immediate value is sign-extended and placed into the even byte elements of parameter **d**, zeroing the odd byte elements. If A = 1, the result in parameter **d** is also placed into the accumulator.

Other registers altered: ACC (if A=1)

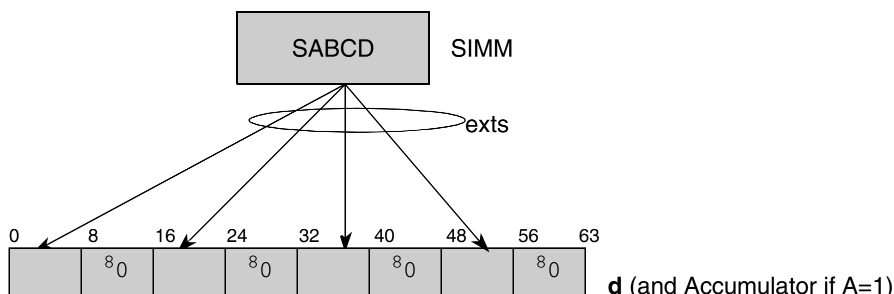


Figure 3-651. Vector Splat Immediate Byte Even (to Accumulator) (__ev_splatibe[a])

d	a	Maps to
__ev64_opaque__	5-bit signed literal	**evsplatibe d,a**
__ev64_opaque__	5-bit signed literal	**evsplatibea d,a**

__ev_splatid[a] __ev_splatid[a]

Vector Splat Immediate Doubleword (to Accumulator)

d = __ev_splatid (a) (A = 0)

d = __ev_splatida (a) (A = 1)

```
SIMM ←a
d0:63 ←EXTS64(SIMM)

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

The 5-bit immediate value is sign extended and 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)

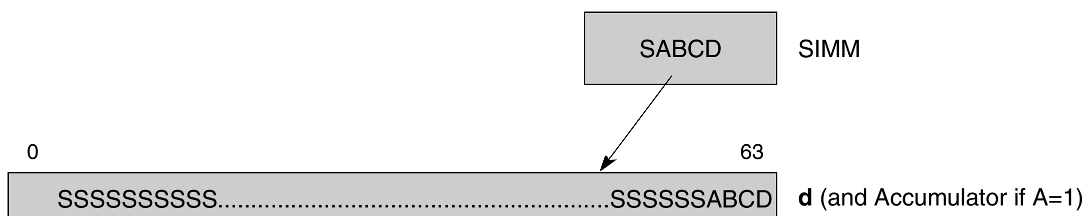


Figure 3-652. Vector Splat Immediate Doubleword (to Accumulator) (__ev_splatid[a])

d	a	Maps to
__ev64_opaque__	5-bit signed literal	**evsplatid d,a**
__ev64_opaque__	5-bit signed literal	**evsplatida d,a**

__ev_splatie[a] __ev_splatie[a]

Vector Splat Immediate Even (to Accumulator)

d = __ev_splatie (a) (A = 0)

d = __ev_splatiea (a) (A = 1)

```
SIMM ←a
d0:31 ←EXTS32(SIMM)
d32:63 ←320

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

The 5-bit immediate value is sign extended and placed in the even word element of parameter **d**, zeroing the odd word element. If A = 1, the result in parameter **d** is also placed into the accumulator.

Other registers altered: ACC (if A=1)

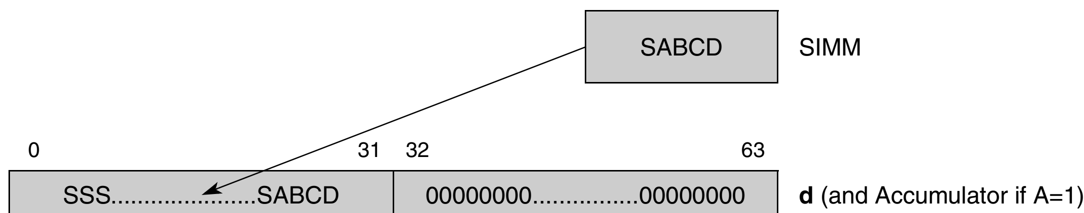


Figure 3-653. Vector Splat Immediate Even (to Accumulator) (__ev_splatie[a])

d	a	Maps to
__ev64_opaque__	5-bit signed literal	**evsplatie d,a**
__ev64_opaque__	5-bit signed literal	**evsplatiea d,a**

__ev_splatih[a] __ev_splatih[a]

Vector Splat Immediate Half Word (to Accumulator)

d = __ev_splatih (a) (A = 0)

d = __ev_splatiha (a) (A = 1)

```
SIMM ←a
d0:15 ←EXTS16(SIMM)
d16:31 ←EXTS16(SIMM)
d32:47 ←EXTS16(SIMM)
d48:63 ←EXTS16(SIMM)

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

The 5-bit immediate value is sign extended and placed into all halfword elements of parameter **d**. If A = 1, the result in parameter **d** is also placed into the accumulator.

Other registers altered: ACC (if A=1)

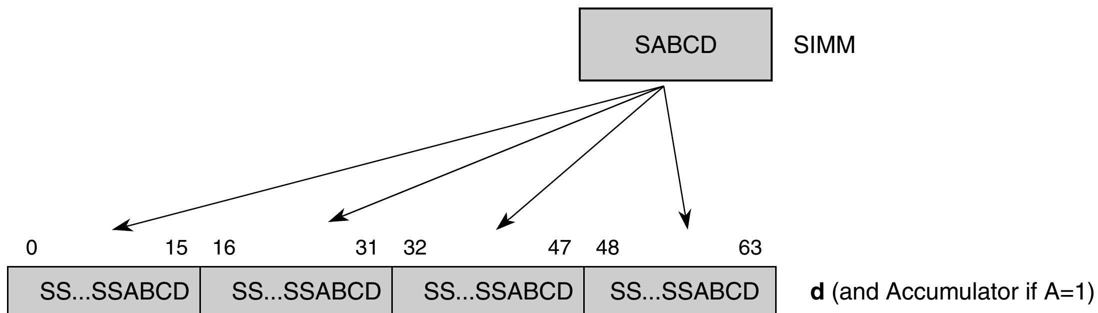


Figure 3-654. Vector Splat Immediate Halfword (to Accumulator) (__ev_splatih[a])

d	a	Maps to
__ev64_opaque__	5-bit signed literal	**evsplatih d,a**
__ev64_opaque__	5-bit signed literal	**evsplatiha d,a**

__ev_splatihe[a] __ev_splatihe[a]

Vector Splat Immediate Halfword Even (to Accumulator)

d = __ev_splatihe (a) (A = 0)

d = __ev_splatihea (a) (A = 1)

```
SIMM ←a
d0:15 ←EXTS16(SIMM)
d16:31 ←160
d32:47 ←EXTS16(SIMM)
d48:63 ←160

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

The 5-bit immediate value is sign extended and placed into the even halfword elements of parameter **d**, zeroing the odd halfword elements. If A = 1, the result in parameter **d** is also placed into the accumulator.

Other registers altered: ACC (if A=1)

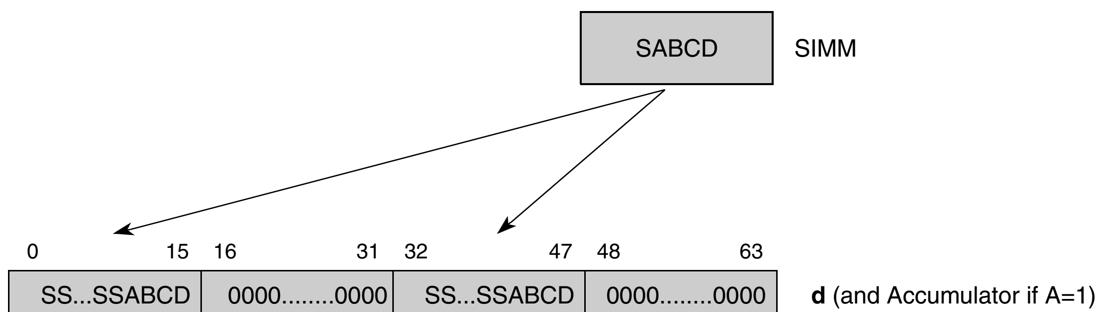


Figure 3-655. Vector Splat Immediate Halfword Even (to Accumulator) (__ev_splatihe[a])

d	a	Maps to
__ev64_opaque__	5-bit signed literal	**evsplatihe d,a**
__ev64_opaque__	5-bit signed literal	**evsplatihea d,a**

__ev_srbis

__ev_srbis

Vector Shift Right Byte Immediate Signed

d = __ev_srbis (a,b)

```
UIMM ←b
n ←UIMM

d0:7 ←EXTS(a0:7-n)
d8:15 ←EXTS(a8:15-n)
d16:23 ←EXTS(a16:23-n)
d24:31 ←EXTS(a24:31-n)
d32:39 ←EXTS(a32:39-n)
d40:47 ←EXTS(a40:47-n)
d48:55 ←EXTS(a48:55-n)
d56:63 ←EXTS(a56:63-n)
```

Each of the byte elements of parameter **a** are shifted right by the unsigned literal value (UIMM) in parameter **b** (range 0-7). Bits in the most significant positions vacated by the shift are filled with a copy of the sign bit.

Shift amounts greater than 7 are illegal.

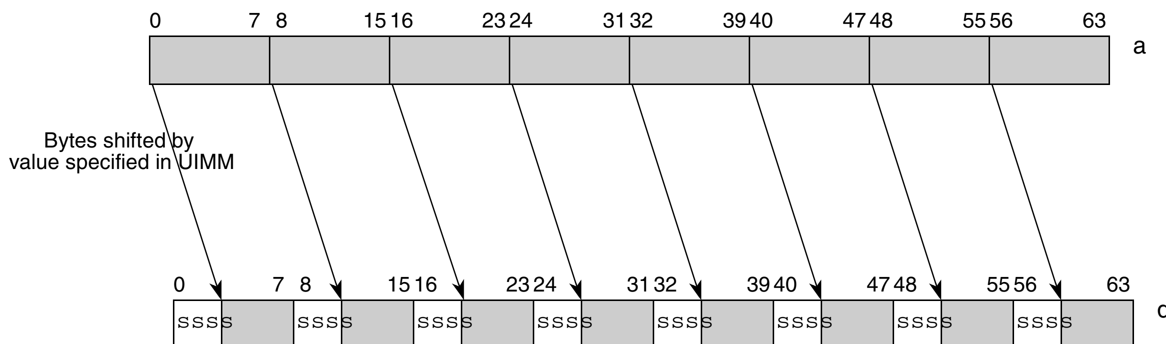


Figure 3-656. Vector Shift Right Byte Immediate Signed (__ev_srbis)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	5-bit unsigned literal	**evsrbis d,a,b**

__ev_srbiu __ev_srbiu

Vector Shift Right Byte Immediate Unsigned

d = __ev_srbiu (a,b)

```
UIMM ←b
n ←UIMM

d0:7 ←EXTZ(a0:7-n)
d8:15 ←EXTZ(a8:15-n)
d16:23 ←EXTZ(a16:23-n)
d24:31 ←EXTZ(a24:31-n)
d32:39 ←EXTZ(a32:39-n)
d40:47 ←EXTZ(a40:47-n)
d48:55 ←EXTZ(a48:55-n)
d56:63 ←EXTZ(a56:63-n)
```

Each of the byte elements of parameter **a** are shifted right by the unsigned literal value (UIMM) in parameter **b** (range 0-7). Bits in the most significant positions vacated by the shift are filled with zeros.

Shift amounts greater than 7 are illegal.

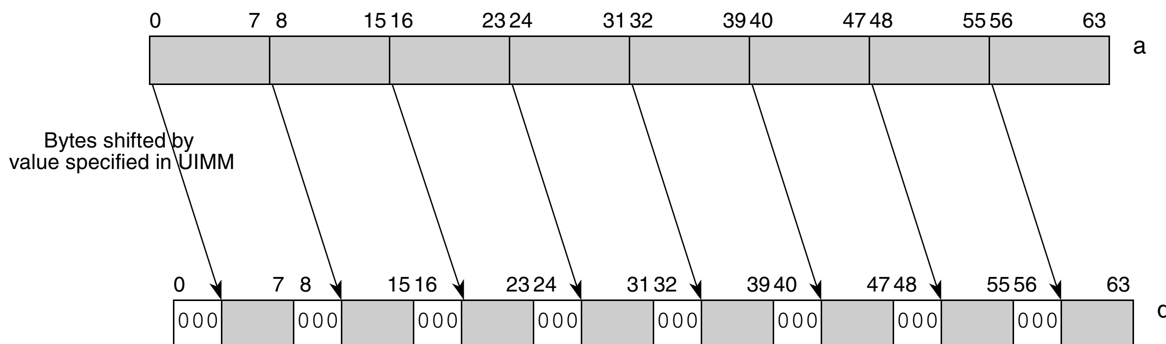


Figure 3-657. Vector Shift Right Byte Immediate Unsigned (__ev_srbiu)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	5-bit unsigned literal	**evsrbiu d,a,b**

__ev_srbs

Vector Shift Right Byte Signed

__ev_srbs

d = __ev_srbs (a,b)

```
nb0 ←b4:7
nb1 ←b12:15
nb2 ←b20:23
nb3 ←b28:31
nb4 ←b36:39
nb5 ←b44:47
nb6 ←b52:55
nb7 ←b60:63

d0:7 ←EXTS(a0:7-nb0)
d8:15 ←EXTS(a8:15-nb1)
d16:23 ←EXTS(a16:23-nb2)
d24:31 ←EXTS(a24:31-nb3)
d32:39 ←EXTS(a32:39-nb4)
d40:47 ←EXTS(a40:47-nb5)
d48:55 ←EXTS(a48:55-nb6)
d56:63 ←EXTS(a56:63-nb7)
```

Each of the byte elements of parameter **a** are shifted right by an amount specified in the corresponding byte elements of parameter **b**. Bits in the most significant positions vacated by the shift are filled with a copy of the sign bit and the result is placed into parameter **d**. The separate shift amounts for each element are specified by the lower 5 bits in each byte element of parameter **b** that lie in bit positions 4-7, 12-15, 20-23, 28-31, 36-39, 44-47, 52-55, and 60-63.

Shift amounts from 8 to 15 give a result of 8 sign bits.

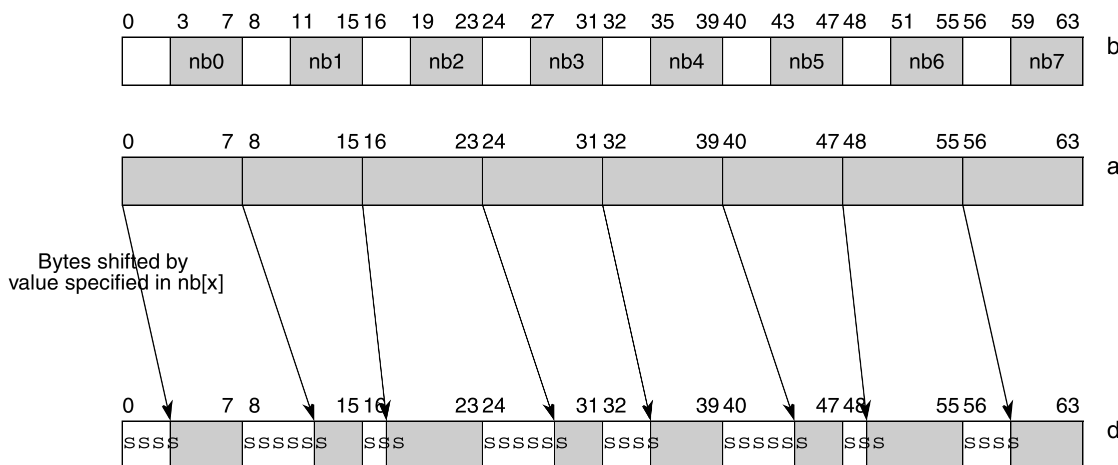


Figure 3-658. Vector Shift Right Byte Signed (__ev_srbs)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evsrbs d,a,b**

__ev_srbu __ev_srbu

Vector Shift Right Byte Unsigned

d = __ev_srbu (a,b)

```
nb0 ←b4:7
nb1 ←b12:15
nb2 ←b20:23
nb3 ←b28:31
nb4 ←b36:39
nb5 ←b44:47
nb6 ←b52:55
nb7 ←b60:63

d0:7 ←EXTZ(a0:7-nb0)
d8:15 ←EXTZ(a8:15-nb1)
d16:23 ←EXTZ(a16:23-nb2)
d24:31 ←EXTZ(a24:31-nb3)
d32:39 ←EXTZ(a32:39-nb4)
d40:47 ←EXTZ(a40:47-nb5)
d48:55 ←EXTZ(a48:55-nb6)
d56:63 ←EXTZ(a56:63-nb7)
```

Each of the byte elements of parameter **a** are shifted right by an amount specified in the corresponding byte elements of parameter **b**. Bits in the most significant positions vacated by the shift are filled with zeros and the result is placed into parameter **d**. The separate shift amounts for each element are specified by the lower 5 bits in each byte element of parameter **b** that lie in bit positions 4-7, 12-15, 20-23, 28-31, 36-39, 44-47, 52-55, and 60-63.

Shift amounts from 8 to 31 give a result of zero.

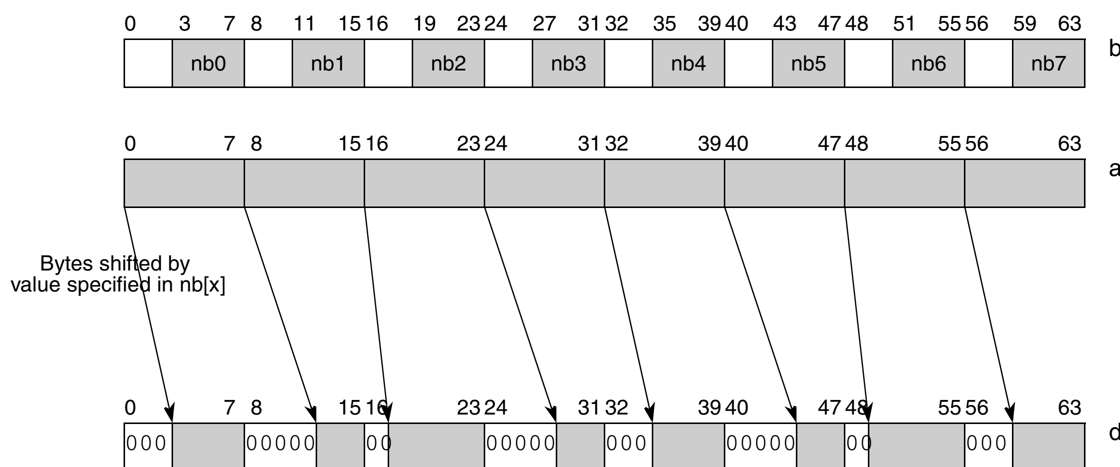


Figure 3-659. Vector Shift Right Byte Unsigned (__ev_srbu)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evsrbu d,a,b**

__ev_srhis __ev_srhis

Vector Shift Right Half Word Immediate Signed

d = __ev_srhis (a,b)

```
UIMM ←b
n ←UIMM

d0:15 ←EXTS(a0:15-n)
d16:31 ←EXTS(a16:31-n)
d32:47 ←EXTS(a32:47-n)
d48:63 ←EXTS(a48:63-n)
```

Each of the half word elements of parameter **a** are shifted right by the unsigned literal value (UIMM) in parameter **b** (range 0-15). Bits in the most significant positions vacated by the shift are filled with a copy of the sign bit.

Shift amounts greater than 15 are illegal.

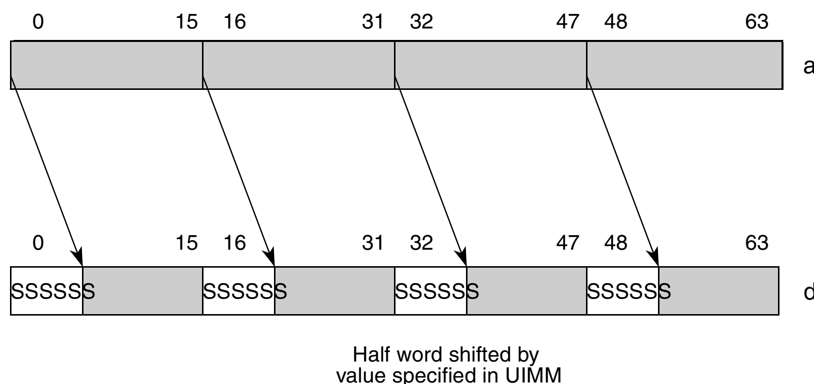


Figure 3-660. Vector Shift Right Half Word Immediate Signed (__ev_srhis)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	5-bit unsigned literal	**evsrhis d,a,b**

__ev_srhiu __ev_srhiu

Vector Shift Right Half Word Immediate Unsigned

d = __ev_srhiu (a,b)

```
UIMM ←b
n ←UIMM

d0:15 ←EXTZ(a0:15-n)
d16:31 ←EXTZ(a16:31-n)
d32:47 ←EXTZ(a32:47-n)
d48:63 ←EXTZ(a48:63-n)
```

Each of the half word elements of parameter **a** are shifted right by the unsigned literal value (UIMM) in parameter **b** (range 0-15). Bits in the most significant positions vacated by the shift are filled with zeros. The results are placed into parameter **d**.

Shift amounts greater than 15 are illegal.

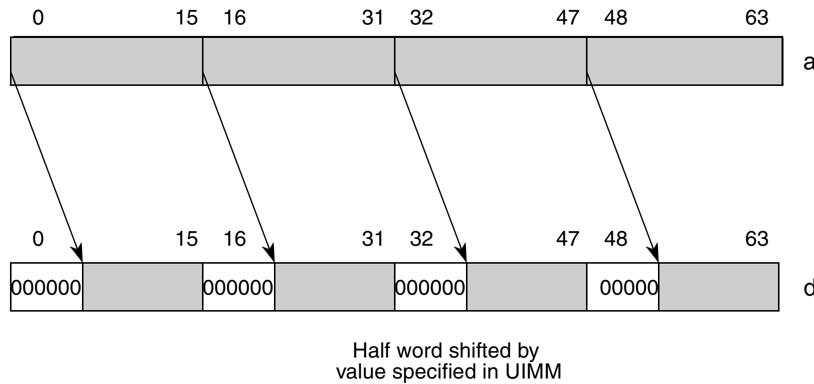


Figure 3-661. Vector Shift Right Half Word Immediate Unsigned (__ev_srhiu)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	5-bit unsigned literal	**evsrhiu d,a,b**

__ev_srhs __ev_srhs

Vector Shift Right Half Word Signed

d = __ev_srhs (a,b)

```
nh0 ←b11:15
nh1 ←b27:31
nh2 ←b43:47
nh3 ←b59:63

d0:15 ←EXTS(a0:15-nh0)
d16:31 ←EXTS(a16:31-nh1)
d32:47 ←EXTS(a32:47-nh2)
d48:63 ←EXTS(a48:63-nh3)
```

Each of the half word elements of parameter **a** are shifted right by an amount specified in the corresponding half word elements of parameter **b**. The result is placed into parameter **d**. The separate shift amounts for each element are specified by 5 bits in parameter **b** that lie in bit positions 11-15, 27-31, 43-47 and 59-63. Bits in the most significant positions vacated by the shift are filled with a copy of the sign bit

Shift amounts from 16 to 31 give a result of 16 sign bits.

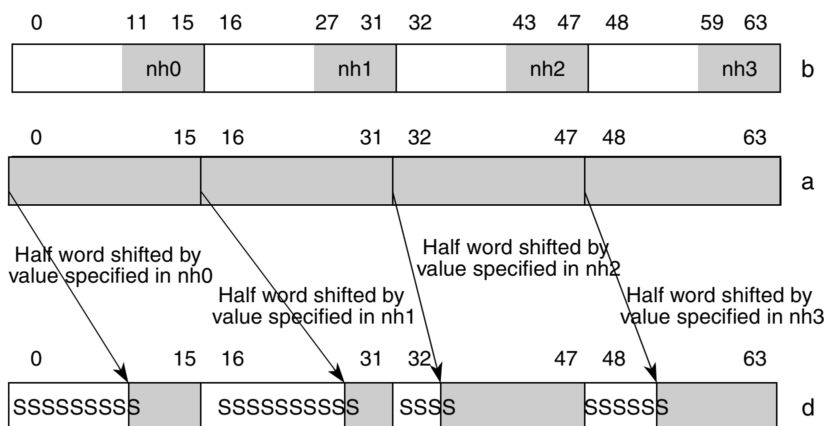


Figure 3-662. Vector Shift Right Half Word Signed (__ev_srhs)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evsrhs d,a,b**

__ev_srhu __ev_srhu

Vector Shift Right Half Word Unsigned

d = __ev_srhu (a,b)

$nh0 \leftarrow b_{11:15}$
$nh1 \leftarrow b_{27:31}$
$nh2 \leftarrow b_{43:47}$
$nh3 \leftarrow b_{59:63}$

$d_{0:15} \leftarrow EXTZ(a_{0:15-nh0})$
$d_{16:31} \leftarrow EXTZ(a_{16:31-nh1})$
$d_{32:47} \leftarrow EXTZ(a_{32:47-nh2})$
$d_{48:63} \leftarrow EXTZ(a_{48:63-nh3})$

Each of the half word elements of parameter **a** are shifted right by an amount specified in the corresponding half word elements of parameter **b**. The result is placed into parameter **d**. The separate shift amounts for each element are specified by 5 bits in parameter **b** that lie in bit positions 11-15, 27-31, 43-47 and 59-63. Bits in the most significant positions vacated by the shift are filled with zeros.

Shift amounts from 16 to 31 give a zero result.

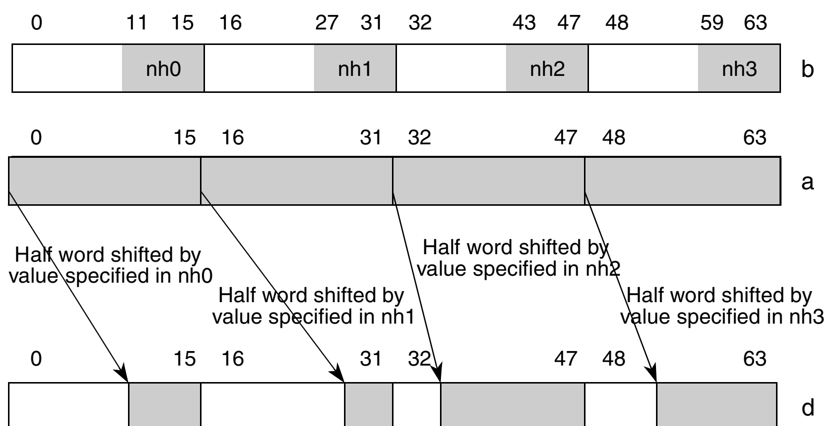


Figure 3-663. Vector Shift Right Half Word Unsigned (__ev_srhu)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evsrhu d,a,b**

__ev_sris __ev_sris

Vector Shift Right Immediate Signed

d = __ev_sris (a,b)

```
n ←b
d0:63 ←EXTS(a0:63-n)
```

The value in parameter **a** is shifted right by the number of bit positions specified by parameter **b**, filling vacated bit positions with the sign of parameter **a**, and the result is placed into parameter **d**.

Note: arbitrary shifts may be performed by using **__ev_sris** in conjunction with **__ev_srois**.

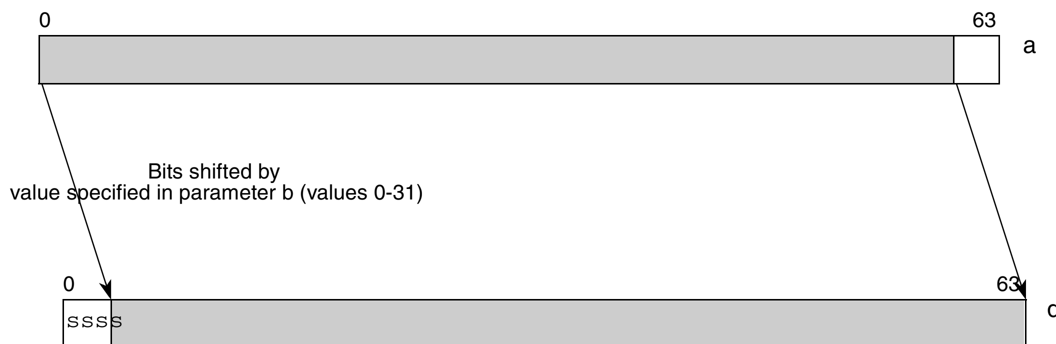


Figure 3-664. Vector Shift Right Immediate Signed (__ev_sris)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	5-bit unsigned literal	**evsris d,a,b**

__ev_sriu

__ev_sriu

Vector Shift Right Immediate

d = __ev_sriu (a,b)

```
n ←b
d0:63 ←EXTZ(a0:63-n)
```

The value in parameter **a** is shifted right by the number of bit positions specified by parameter **b**, filling vacated bit positions with zeros, and the result is placed into parameter **d**.

Note: arbitrary shifts may be performed by using **__ev_sriu** in conjunction with **__ev_sroiu**.

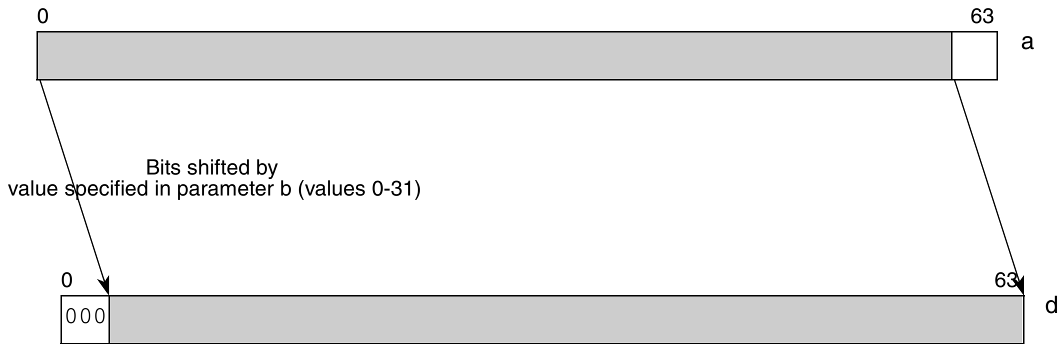


Figure 3-665. Vector Shift Right Immediate (evsri)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	5-bit unsigned literal	**evsriu d,a,b**

__ev_srois __ev_srois

Vector Shift Right by Octet Immediate Signed

d = __ev_srois (a,b)

```
n ←b * 8
d0:63 ←EXTS(an:63)
```

The value in parameter **a** is shifted right by the number of byte positions contained in parameter **b**, filling vacated byte positions with the sign of parameter **a**, and the result is placed into parameter **d**.

Note: arbitrary shifts may be performed by using **__ev_sris** in conjunction with **__ev_srois**.

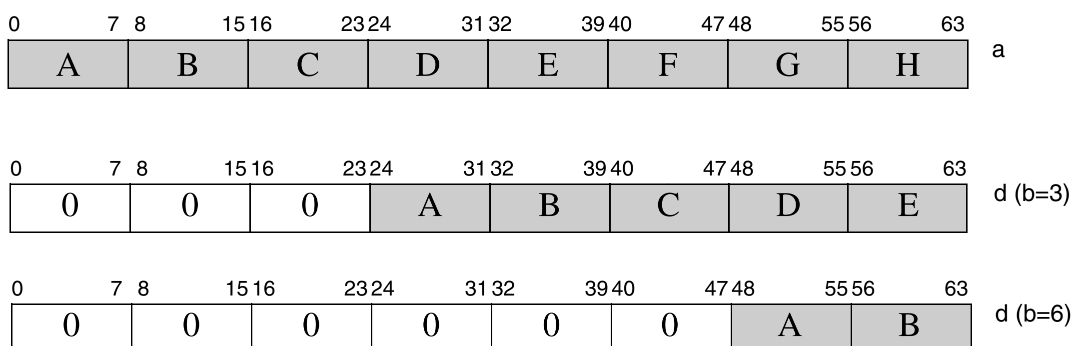


Figure 3-666. Vector Shift Right by Octet Immediate Signed (__ev_srois)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	3-bit unsigned literal	**evsrois d,a,b**

__ev_sroiu __ev_sroiu

Vector Shift Right by Octet Immediate Unsigned

d = __ev_sroiu (a,b)

```
n ←b * 8
d0:63 ←EXTZ(an:63)
```

The value in parameter **a** is shifted right by the number of byte positions contained in parameter **b**, filling vacated byte positions with zeros, and the result is placed into parameter **d**.

Note: arbitrary shifts may be performed by using **__ev_sriu** in conjunction with **__ev_sroiu**.

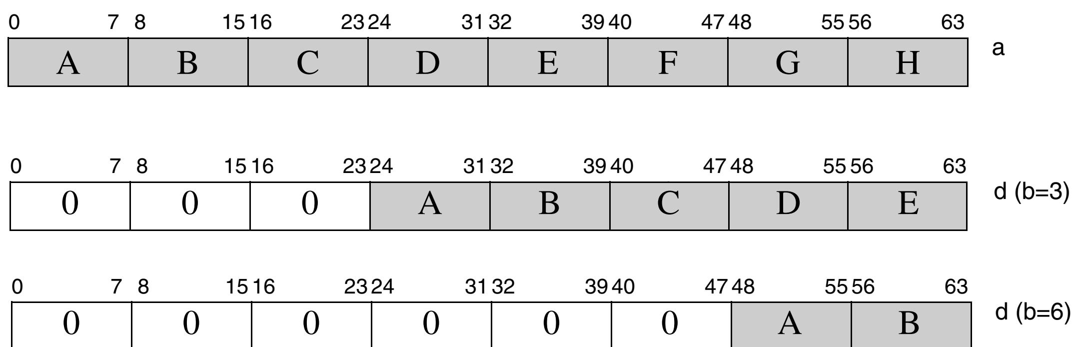


Figure 3-667. Vector Shift Right by Octet Immediate (__ev_sroiu)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	3-bit unsigned literal	**evsroiu d,a,b**

__ev_srs __ev_srs

Vector Shift Right Signed

d = __ev_srs (a,b)

```
sh ←b57:63
s ←a0

if b57 = 0 then
    d0:63 ←EXTS(a0:63-sh)
else
    d ←64s
```

The value in parameter **a** is shifted right arithmetically by ‘sh’ bit positions specified in parameter $\mathbf{b}_{57:63}$, filling vacated bit positions with the sign of parameter **a**, and the result is placed into parameter **d**. Shift amounts of 64 to 127 give a result of 0 if parameter **a** is positive, or -1 if parameter **a** is negative.

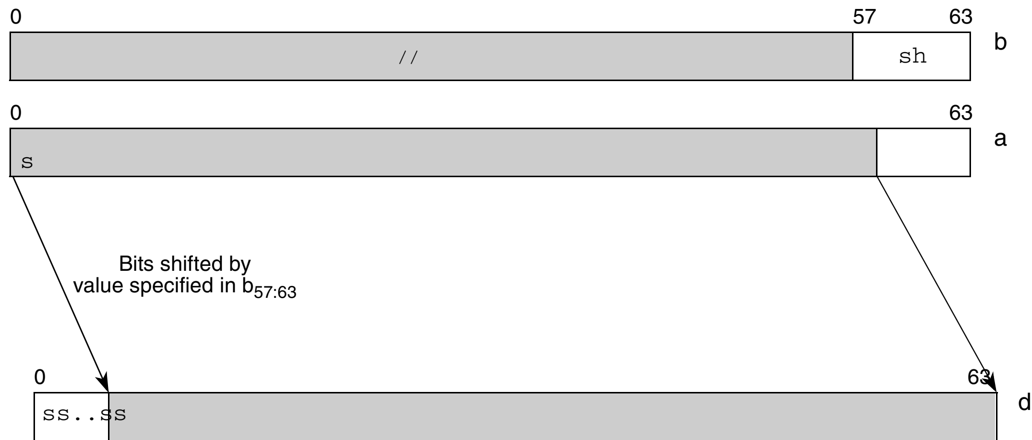


Figure 3-668. Vector Shift Right Signed (__ev_srs)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evsrs d,a,b**

__ev_sru __ev_sru

Vector Shift Right Unsigned

d = __ev_sru (a,b)

```
sh ←b57:63

if b57 = 0 then
    d0:63 ←EXTZ(a0:63-sh)
else
    d ←640
```

The value in parameter **a** is shifted right by 'sh' bit positions specified in parameter $\mathbf{b}_{57:63}$, filling vacated bit positions with zeros, and the result is placed into parameter **d**. Shift amounts of 64 to 127 give a zero result.

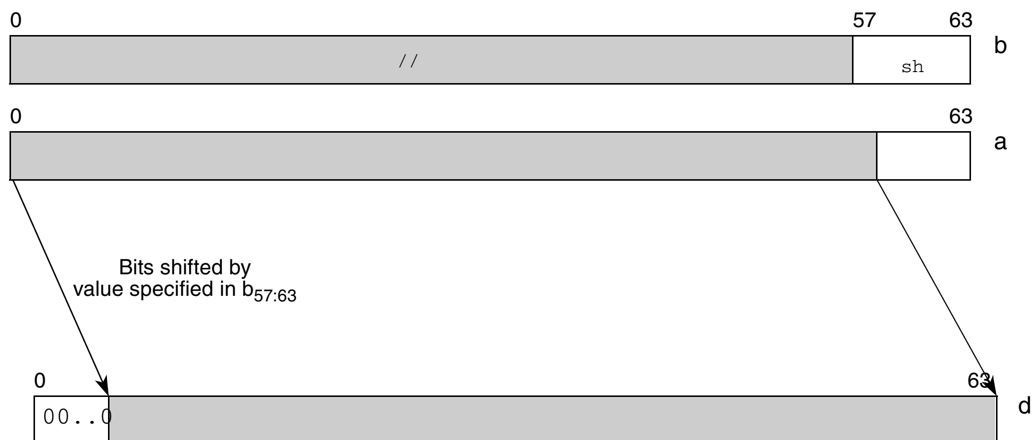


Figure 3-669. Vector Shift Right Unsigned (__ev_sru)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evsru d,a,b**

__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 unsigned immediate (UIMM) value in parameter **b**. Bits in the most significant positions vacated by the shift are filled with a copy of the sign bit. The results are placed into parameter **d**.

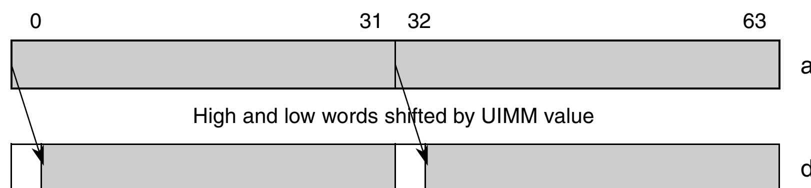


Figure 3-670. Vector Shift Right Word Immediate Signed (__ev_srwis)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	5-bit unsigned literal	**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 unsigned immediate (UIMM) value in parameter **b**; 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. The results are placed into parameter **d**.

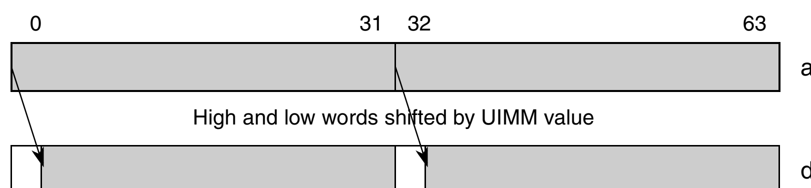


Figure 3-671. Vector Shift Right Word Immediate Unsigned (__ev_srwiu)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	5-bit unsigned literal	**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.

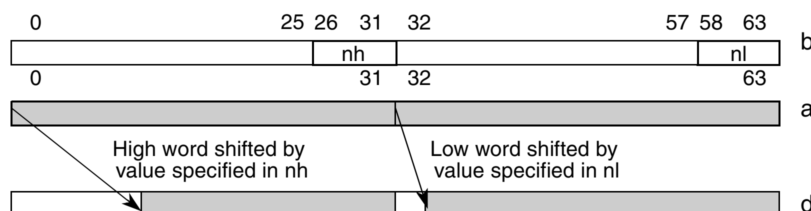


Figure 3-672. Vector Shift Right Word Signed (__ev_srws)

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.

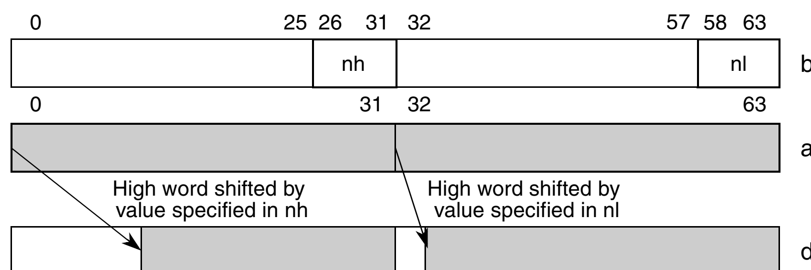


Figure 3-673. Vector Shift Right Word Unsigned (__ev_srwu)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evsrwu d,a,b**

__ev_stdb[u] __ev_stdb[u]

Vector Store Double of Eight Bytes [with Update]

__ev_stdb (a,b,c) (U = 0)

__ev_stdbu (a,b,c) (U = 1)

```
if (b = r0) then temp ←0
else temp ←(b)
EA ←temp + EXTZ(c*8)

MEM(EA,1) ←a0:7
MEM(EA+1,1) ←a8:15
MEM(EA+2,1) ←a16:23
MEM(EA+3,1) ←a24:31
MEM(EA+4,1) ←a32:39
MEM(EA+5,1) ←a40:47
MEM(EA+6,1) ←a48:55
MEM(EA+7,1) ←a56:63

if (U=1) then b ←EA
```

The contents of parameter **a** are stored as eight bytes in storage addressed by the effective address (EA).

If U=1 (‘with update’), EA is placed into parameter **b**.

Figure 3-674 shows how bytes are stored in memory as determined by the endian mode.

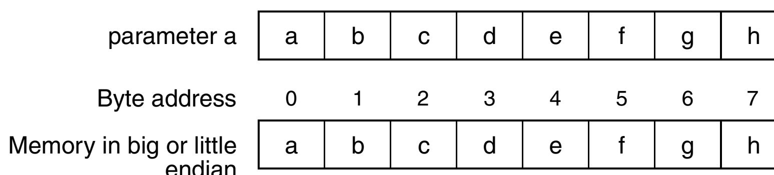


Figure 3-674. __ev_stdb[u] Results in Big- and Little-Endian Modes

NOTE

An alignment exception may occur if the effective address (EA) is not double-word aligned. Also, if machine register r0 is used to pass parameter **b**, then a zero value is used as the base value when calculating EA. For the ‘with update’ form (U=1), parameter **b** is both an input and output operand. Also, for the ‘with update’ form (U=1), parameter **c** cannot be 0.

U	returns	a	b	c	Maps to
U = 0	*void*	__ev64_opaque__	__ev64_opaque__ *	5-bit unsigned literal	**evstdb a,c(b)**
U = 1	*void*	__ev64_opaque__	__ev64_opaque__ *&	5-bit unsigned literal	**evstdbu a,c(b)**

__ev_stdb[m]x __ev_stdb[m]x

Vector Store Double of Eight Bytes [with Modify] Indexed

__ev_stdbx (a,b,c) (M = 0)

__ev_stdbmx (a,b,c) (M = 1)

```
if b=0 & M=1 then take_illegal_exception
if b=0 & M=0 then temp ←0
else temp ←(b)
EA ←temp + (c)
MEM(EA,1) ←a0:7
MEM(EA+1,1)* ←a8:15
MEM(EA+2,1)* ←a16:23
MEM(EA+3,1)* ←a24:31
MEM(EA+4,1)* ←a32:39
MEM(EA+5,1)* ←a40:47
MEM(EA+6,1)* ←a48:55
MEM(EA+7,1)* ←a56:63

if (M=1) then b32:63 ←calc_b_update(b,c)

        * - may wrap at length boundary for M=1 and mode 1000.
```

The contents of parameter **a** are stored as eight bytes in storage addressed by EA.

If M=1 (‘with modify’), $\mathbf{b}_{32:63}$ is updated with an address value determined by the mode specifier in $\mathbf{b}_{0:3}$. See Section 3.2.3, “Addressing Modes - Modify forms.

Figure 3-675 shows how bytes are stored in memory as determined by the endian mode.

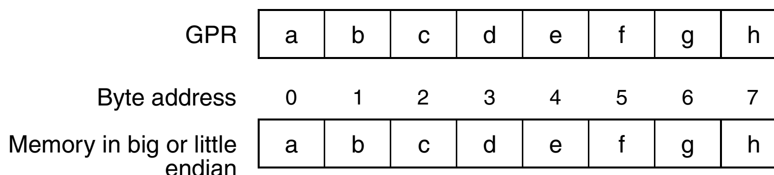


Figure 3-675. __ev_stdb[m]x Results in Big- and Little-Endian Modes

NOTE

An alignment exception may occur if the effective address (EA) is not double-word aligned. Also, if machine register r0 is used to pass parameter **c**, then a zero value is used as the base value when calculating EA. And, for the ‘with modify’ form (M=1), parameter **c** is both an input and output operand.

M	returns	a	b	c	Maps to
M = 0	*void*	__ev64_opaque__	__ev64_opaque__ *	int32_t	**evstdbx a,b,c**
M = 1	*void*	__ev64_opaque__	__ev64_opaque__ *&	int32_t	**evstdbmx a,b,c**

__ev_stdd[u]

__ev_stdd[u]

Vector Store Double of Double [with Update]

__ev_stdd (a,b,c) (U = 0)

__ev_stddu (a,b,c) (U = 1)

```
if (b = r0) then temp ←0
else temp ←(b)
EA ←temp + EXTZ(c*8)
MEM(EA,8) ←a0:63

if (U=1) then b ← EA
```

The contents of parameter **a** are stored as a double word in storage addressed by EA.

If U=1 (‘with update’), EA is placed into parameter **b**.

Figure 3-676 shows how bytes are stored in memory as determined by the endian mode.

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

Figure 3-676. __ev_stdd[u] Results in Big- and Little-Endian Modes

NOTE

An alignment exception may occur if the effective address (EA) is not double-word aligned. Also, if machine register r0 is used to pass parameter **b**, then a zero value is used as the base value when calculating EA. For the ‘with update’ form (U=1), parameter **b** is both an input and output operand. Also, for the ‘with update’ form (U=1), parameter **c** cannot be 0.

U	returns	a	b	c	Maps to
U = 0	*void*	__ev64_opaque__	__ev64_opaque__ *	5-bit unsigned literal	**evstdd a,c(b)**
U = 1	*void*	__ev64_opaque__	__ev64_opaque__ *&	5-bit unsigned literal	**evstddu a,c(b)**

__ev_stdd[m]x

__ev_stdd[m]x

Vector Store Double of Double [with Modify] Indexed

__ev_stddx (a,b,c) (M = 0)

__ev_stddmx (a,b,c) (M = 1)

```
if b=0 & M=1 then take_illegal_exception
if b=0 & M=0 then temp ←0
else temp←(b)
EA ←temp + (c)
MEM(EA,8)* ←a0:63
if (M=1) then b32:63 ←calc_b_update(b,c)
        * - may wrap at length boundary for M=1 and mode 1000.
```

The contents of parameter **a** are stored as a doubleword in storage addressed by EA.

If M=1 (‘with modify’), $\mathbf{b}_{32:63}$ is updated with an address value determined by the mode specifier in $\mathbf{b}_{0:3}$. See Section 3.2.3, “Addressing Modes - Modify forms.

Figure 3-677 shows how bytes are stored in memory as determined by the endian mode.

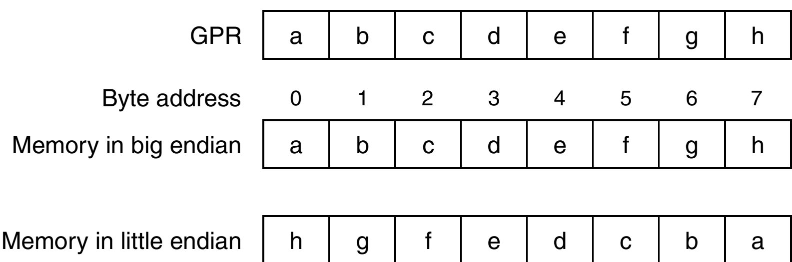


Figure 3-677. __ev_stdd[m]x Results in Big- and Little-Endian Modes

NOTE

An alignment exception may occur if the effective address (EA) is not double-word aligned. Also, if machine register r0 is used to pass parameter **c**, then a zero value is used as the base value when calculating EA. And, for the ‘with modify’ form (M=1), parameter **c** is both an input and output operand.

U	returns	a	b	c	Maps to
M = 0	*void*	__ev64_opaque__	__ev64_opaque__ *	int32_t	**evstddx a,b,c**
M = 1	*void*	__ev64_opaque__	__ev64_opaque__ *&	int32_t	**evstddmx a,b,c**

__ev_stdh[u]   __ev_stdh[u]

Vector Store Double of Four Half Words [with Update]

__ev_stdh (a,b,c) (U = 0)

__ev_stdhu (a,b,c) (U = 1)

```
if (b = r0) then temp ←0
else temp ←b
EA ←temp + EXTZ(c*8)
MEM(EA,2) ←a0:15
MEM(EA+2,2) ←a16:31
MEM(EA+4,2) ←a32:47
MEM(EA+6,2) ←a48:63

if (U=1) then b ← EA
```

The contents of parameter **a** are stored as four half words in storage addressed by EA.

If U=1 (‘with update’), EA is placed into parameter **b**.

Figure 3-678 shows how bytes are stored in memory as determined by the endian mode.

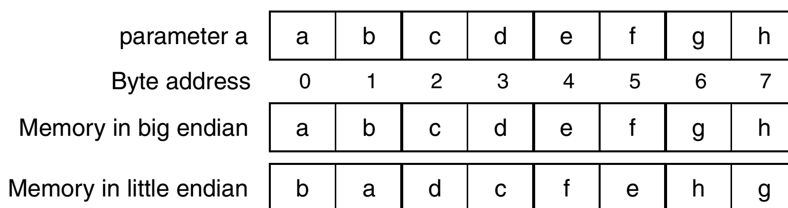


Figure 3-678. __ev_stdh[u] Results in Big- and Little-Endian Modes

NOTE

An alignment exception may occur if the effective address (EA) is not double-word aligned. Also, if machine register r0 is used to pass parameter **b**, then a zero value is used as the base value when calculating EA. For the ‘with update’ form (U=1), parameter **b** is both an input and output operand. Also, for the ‘with update’ form (U=1), parameter **c** cannot be 0.

U	returns	a	b	c	Maps to
U = 0	*void*	__ev64_opaque__	__ev64_opaque__ *	5-bit unsigned literal	**evstdh a,c(b)**
U = 1	*void*	__ev64_opaque__	__ev64_opaque__ *&	5-bit unsigned literal	**evstdhu a,c(b)**

__ev_stdh[m]x __ev_stdh[m]x

Vector Store Double of Four Half Words [with Modify] Indexed

__ev_stdhx (a,b,c) (M = 0)

__ev_stdhmx (a,b,c) (M = 1)

```
if b=0 & M=1 then take_illegal_exception
if b=0 & M=0 then temp ←0
else temp ←(b)
EA ←temp + (c)
MEM(EA,2)* ←a0:15
MEM(EA+2,2)* ←a16:31
MEM(EA+4,2)* ←a32:47
MEM(EA+6,2)* ←a48:63

if (M=1) then b32:63 ←calc_b_update(b,c)

        * - may wrap at length boundary for M=1 and mode 1000.
```

The contents of parameter **a** are stored as four halfwords in storage addressed by EA.

If M=1 (‘with modify’), $\mathbf{b}_{32:63}$ is updated with an address value determined by the mode specifier in $\mathbf{b}_{0:3}$. See Section 3.2.3, “Addressing Modes - Modify forms.

Figure 3-679 shows how bytes are stored in memory as determined by the endian mode.

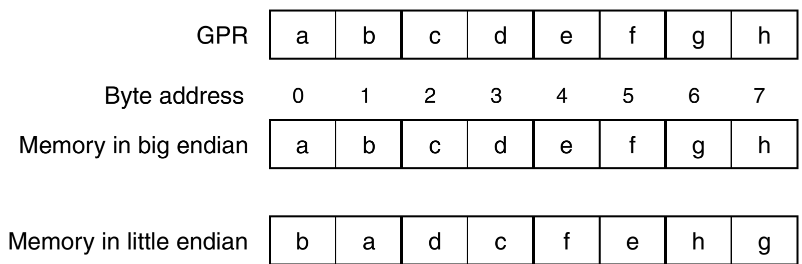


Figure 3-679. __ev_stdh[m]x Results in Big- and Little-Endian Modes

NOTE

An alignment exception may occur if the effective address (EA) is not double-word aligned. Also, if machine register r0 is used to pass parameter **c**, then a zero value is used as the base value when calculating EA. And, for the ‘with modify’ form (M=1), parameter **c** is both an input and output operand.

M	returns	a	b	c	Maps to
M = 0	*void*	__ev64_opaque__	__ev64_opaque__ *	int32_t	**evstdhx a,b,c**
M = 1	*void*	__ev64_opaque__	__ev64_opaque__ *&	int32_t	**evstdhmx a,b,c**

__ev_stdw[u] __ev_stdw[u]

Vector Store Double of Two Words [with Update]

__ev_stdw (a,b,c) (U = 0)

__ev_stdwu (a,b,c) (U = 1)

```
if (b = r0) then temp ←0
else temp ←(b)
EA ←temp + EXTZ(c*8)
MEM(EA,4) ←a0:31
MEM(EA+4,4) ←a32:63

if (U=1) then b ← EA
```

The contents of parameter **a** are stored as two words in storage addressed by EA.

If U=1 (‘with update’), EA is placed into parameter **b**.

Figure 3-680 shows how bytes are stored in memory as determined by the endian mode.

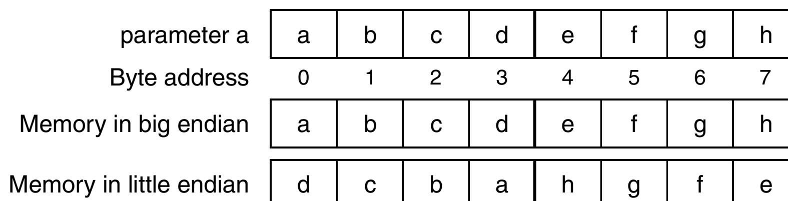


Figure 3-680. __ev_stdw[u] Results in Big- and Little-Endian Modes

NOTE

An alignment exception may occur if the effective address (EA) is not double-word aligned. Also, if machine register r0 is used to pass parameter **b**, then a zero value is used as the base value when calculating EA. For the ‘with update’ form (U=1), parameter **b** is both an input and output operand. Also, for the ‘with update’ form (U=1), parameter **c** cannot be 0.

U	returns	a	b	c	Maps to
U = 0	*void*	__ev64_opaque__	__ev64_opaque__ *	5-bit unsigned literal	**evstdw a,c(b)**
U = 1	*void*	__ev64_opaque__	__ev64_opaque__ *&	5-bit unsigned literal	**evstdwu a,c(b)**

__ev_stdw[m]x

__ev_stdw[m]x

Vector Store Double of Two Words [with Modify] Indexed

__ev_stdwx (a,b,c) (M = 0)

__ev_stdwmx (a,b,c) (M = 1)

```
if b=0 & M=1 then take_illegal_exception
if b=0 & M=0 then temp ←0
else temp ←(b)
EA ←temp + (c)
MEM(EA,4)* ←a0:31
MEM(EA+4,4)* ←a32:63

if (M=1) then b32:63 ←calc_b_update(b,c)

        * - may wrap at length boundary for M=1 and mode 1000.
```

The contents of parameter **a** are stored as two words in storage addressed by EA.

If M=1 (‘with modify’), parameter $\mathbf{b}_{32:63}$ is updated with an address value determined by the mode specifier in parameter $\mathbf{b}_{0:3}$. See Section 3.2.3, “Addressing Modes - Modify forms.

Figure 3-681 shows how bytes are stored in memory as determined by the endian mode.

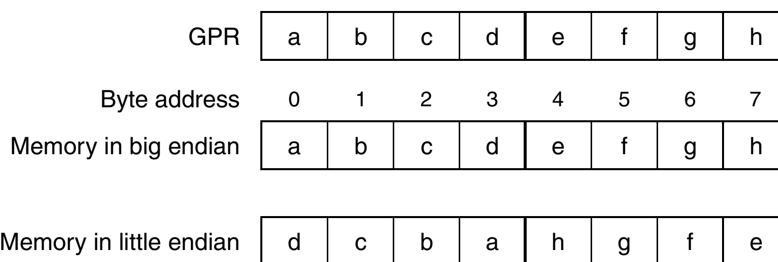


Figure 3-681. __ev_stdw[m]x Results in Big- and Little-Endian Modes

NOTE

An alignment exception may occur if the effective address (EA) is not double-word aligned. Also, if machine register r0 is used to pass parameter **c**, then a zero value is used as the base value when calculating EA. And, for the ‘with modify’ form (M=1), parameter **c** is both an input and output operand.

M	returns	a	b	c	Maps to
M = 0	*void*	__ev64_opaque__	__ev64_opaque__ *	int32_t	**evstdwx a,b,c**
M = 1	*void*	__ev64_opaque__	__ev64_opaque__ *&	int32_t	**evstdwmx a,b,c**

__ev_sthb[u] __ev_sthb[u]

Vector Store Half Word of Two Bytes [with Update]

__ev_sthb (a,b,c) (U = 0)

__ev_sthbu (a,b,c) (U = 1)

```
if (b = r0) then temp ←0
else temp ←(b)
EA ←temp + EXTZ(c*2)

MEM(EA,1) ←a48:55
MEM(EA+1,1) ←a56:63

if U=1 then b ← EA
```

The two lower byte elements of parameter **a** are stored as two bytes in storage addressed by EA.

If U=1 (‘with update’), EA is placed into parameter **b**.

Figure 3-682 shows how bytes are stored in memory as determined by the endian mode.

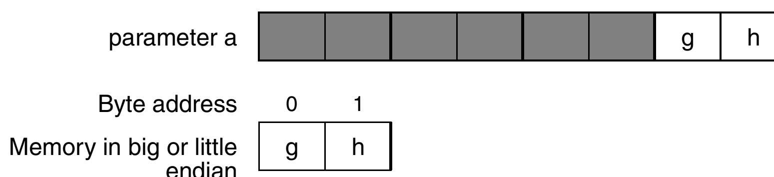


Figure 3-682. __ev_sthb[u] Results in Big- and Little-Endian Modes

NOTE

An alignment exception may occur if the effective address (EA) is not half word aligned. Also, if machine register r0 is used to pass parameter **b**, then a zero value is used as the base value when calculating EA. For the ‘with update’ form (U=1), parameter **b** is both an input and output operand. Also, for the ‘with update’ form (U=1), parameter **c** cannot be 0.

U	returns	a	b	c	Maps to
U = 0	*void*	__ev64_opaque__	uint16_t *	5-bit unsigned literal	**evsthb a,c(b)**
U = 1	*void*	__ev64_opaque__	uint16_t *&	5-bit unsigned literal	**evsthbu a,c(b)**

__ev_sthb[m]x __ev_sthb[m]x

Vector Store Half Word of Two Bytes [with Modify] Indexed

__ev_sthbx (a,b,c) (M = 0)

__ev_sthbmx (a,b,c) (M = 1)

```
if b=0 & M=1 then take_illegal_exception
if b=0 & M=0 then temp ←0
else temp←(b)
EA ←temp + (c)

MEM(EA,1) ←a48:55
MEM(EA+1,1)* ←a56:63

if (M=1) then b32:63 ←calc_b_update(b,c)
        * - may wrap at length boundary for M=1 and mode 1000.
```

The two lower byte elements of parameter **a** are stored as two bytes in storage addressed by EA.

If M=1 (‘with modify’), $\mathbf{b}_{32:63}$ is updated with an address value determined by the mode specifier in $\mathbf{b}_{0:3}$. See Section 3.2.3, “Addressing Modes - Modify forms.

Figure 3-683 shows how bytes are stored in memory as determined by the endian mode.

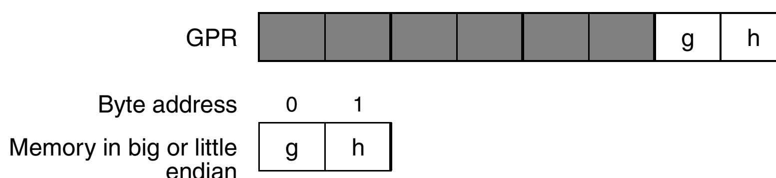


Figure 3-683. __ev_sthb[m]x Results in Big- and Little-Endian Modes

NOTE

An alignment exception may occur if the effective address (EA) is not half word aligned. Also, if machine register r0 is used to pass parameter **c**, then a zero value is used as the base value when calculating EA. And, for the ‘with modify’ form (M=1), parameter **c** is both an input and output operand.

M	returns	a	b	c	Maps to
M = 0	*void*	__ev64_opaque__	uint16_t *	int32_t	**evsthbx a,b,c**
M = 1	*void*	__ev64_opaque__	uint16_t *&	int32_t	**evsthbmx a,b,c**

__ev_stwb[u] __ev_stwb[u]

Vector Store Word of Four Bytes [with Update]

d = __ev_stwb (a,b,c) (U = 0)

d = __ev_stwbu (a,b,c) (U = 1)

```
if (b = r0) then temp ←0
else temp ←(b)
EA ←temp + EXTZ(c*4)
MEM(EA,1) ←a32:39
MEM(EA+1,1) ←a40:47
MEM(EA+2,1) ←a48:55
MEM(EA+3,1) ←a56:63

if U=1 then b ← EA
```

The four lower byte elements of parameter **a** are stored as four bytes in storage addressed by EA.

If U=1 (‘with update’), EA is placed into parameter **b**.

Figure 3-684 shows how bytes are stored in memory as determined by the endian mode.

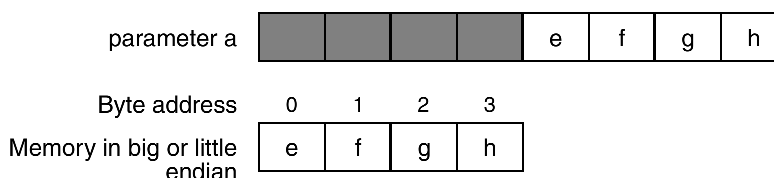


Figure 3-684. __ev_stwb[u] Results in Big- and Little-Endian Modes

NOTE

An alignment exception may occur if the effective address (EA) is not word aligned. Also, if machine register r0 is used to pass parameter **b**, then a zero value is used as the base value when calculating EA. For the ‘with update’ form (U=1), parameter **b** is both an input and output operand. Also, for the ‘with update’ form (U=1), parameter **c** cannot be 0.

U	returns	a	b	c	Maps to
U = 0	*void*	__ev64_opaque__	uint32_t *	5-bit unsigned literal	**evstwb a,c(b)**
U = 1	*void*	__ev64_opaque__	uint32_t *&	5-bit unsigned literal	**evstwbu a,c(b)**

__ev_stwb[m]x   __ev_stwb[m]x

Vector Store Word of Four Bytes [with Modify] Indexed

__ev_stwbx (a,b,c) (M = 0)

__ev_stwbmx (a,b,c) (M = 1)

```
if b=0 & M=1 then take_illegal_exception
if b=0 & M=0 then temp ←0
else temp ←(b)
EA ←temp + (c)
MEM(EA,1) ←a32:39
MEM(EA+1,1)* ←a40:47
MEM(EA+2,1)* ←a48:55
MEM(EA+3,1)* ←a56:63

if (M=1) then b32:63 ←calc_b_update(b,c)

        * - may wrap at length boundary for M=1 and mode 1000.
```

The four lower byte elements of parameter **a** are stored as four bytes in storage addressed by EA.

If M=1 (‘with modify’), $\mathbf{b}_{32:63}$ is updated with an address value determined by the mode specifier in $\mathbf{b}_{0:3}$. See Section 3.2.3, “Addressing Modes - Modify forms.

Figure 3-685 shows how bytes are stored in memory as determined by the endian mode.

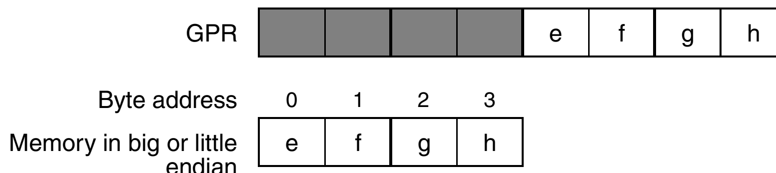


Figure 3-685. __ev_stwb[m]x Results in Big- and Little-Endian Modes

NOTE

An alignment exception may occur if the effective address (EA) is not word aligned. Also, if machine register r0 is used to pass parameter **c**, then a zero value is used as the base value when calculating EA. And, for the ‘with modify’ form (M=1), parameter **c** is both an input and output operand.

M	returns	a	b	c	Maps to
M = 0	*void*	__ev64_opaque__	uint32_t *	int32_t	**evstwbx a,b,c**
M = 1	*void*	__ev64_opaque__	uint32_t *&	int32_t	**evstwbmx a,b,c**

__ev_stwbe[u] __ev_stwbe[u]

Vector Store Word of Four Bytes from Even [with Update]

__ev_stwbe (a,b,c) (U = 0)

__ev_stwbeu (a,b,c) (U = 1)

```
if (b = r0) then temp ←0
else temp ←(b)
EA ←temp + EXTZ(c*4)
MEM(EA,1) ←a0:7
MEM(EA+1,1) ←a16:23
MEM(EA+2,1) ←a32:39
MEM(EA+3,1) ←a48:55

if (U=1) then b ← EA
```

The even byte elements of parameter **a** are stored as four bytes in storage addressed by EA.

If U=1 (‘with update’), EA is placed into parameter **b**.

Figure 3-686 shows how bytes are stored in memory as determined by the endian mode.

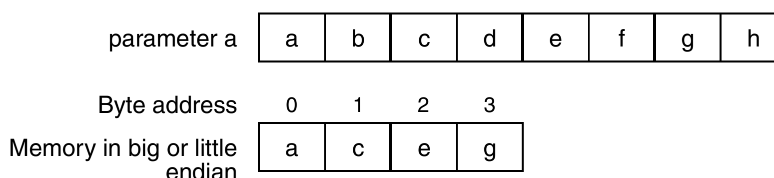


Figure 3-686. __ev_stwbe[u] Results in Big- and Little-Endian Modes

NOTE

An alignment exception may occur if the effective address (EA) is not word aligned. Also, if machine register r0 is used to pass parameter **b**, then a zero value is used as the base value when calculating EA. For the ‘with update’ form (U=1), parameter **b** is both an input and output operand. Also, for the ‘with update’ form (U=1), parameter **c** cannot be 0.

U	returns	a	b	c	Maps to
U = 0	*void*	__ev64_opaque__	uint32_t *	5-bit unsigned literal	**evstwbe a,c(b)**
U = 1	*void*	__ev64_opaque__	uint32_t *&	5-bit unsigned literal	**evstwbeu a,c(b)**

__ev_stwbe[m]x __ev_stwbe[m]x

Vector Store Word of Four Bytes from Even [with Modify] Indexed

__ev_stwbex (a,b,c) (M = 0)

__ev_stwbemx (a,b,c) (M = 1)

```
if b=0 & M=1 then take_illegal_exception
if b=0 & M=0 then temp ←0
else temp ←(b)
EA ←temp + (c)
MEM(EA,1) ←a0:7
MEM(EA+1,1)* ←a16:23
MEM(EA+2,1)* ←a32:39
MEM(EA+3,1)* ←a48:55

if (M=1) then b32:63 ←calc_b_update(b,c)

        * - may wrap at length boundary for M=1 and mode 1000.
```

The even byte elements of parameter **a** are stored as four bytes in storage addressed by EA.

If M=1 (‘with modify’), $\mathbf{b}_{32:63}$ is updated with an address value determined by the mode specifier in $\mathbf{b}_{0:3}$. See Section 3.2.3, “Addressing Modes - Modify forms.

Figure 3-687 shows how bytes are stored in memory as determined by the endian mode.

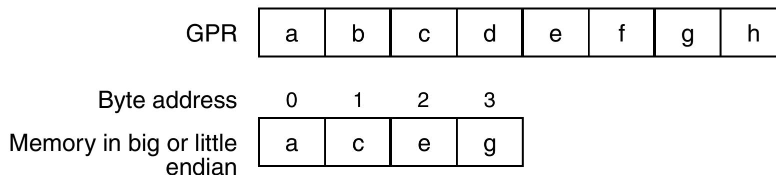


Figure 3-687. __ev_stwbe[m]x Results in Big- and Little-Endian Modes

NOTE

An alignment exception may occur if the effective address (EA) is not word aligned. Also, if machine register r0 is used to pass parameter **c**, then a zero value is used as the base value when calculating EA. And, for the ‘with modify’ form (M=1), parameter **c** is both an input and output operand.

M	returns	a	b	c	Maps to
M = 0	*void*	__ev64_opaque__	uint32_t *	int32_t	**evstwbex a,b,c**
M = 1	*void*	__ev64_opaque__	uint32_t *&	int32_t	**evstwbemx a,b,c**

__ev_stwbo[u]
__ev_stwbo[u]

Vector Store Word of Four Bytes from Odd [with Update]

__ev_stwbo (a,b,c) (U = 0)

__ev_stwbou (a,b,c) (U = 1)

```
if (b = r0) then temp ←0
else temp ←(b)
EA ←temp + EXTZ(c*4)
MEM(EA,1) ←a8:15
MEM(EA+1,1) ←a24:31
MEM(EA+2,1) ←a40:47
MEM(EA+3,1) ←a56:63

if (U=1) then b ← EA
```

The odd byte elements of parameter **a** are stored as four bytes in storage addressed by EA.

If U=1 ('with update'), EA is placed into parameter **b**.

Figure 3-688 shows how bytes are stored in memory as determined by the endian mode.

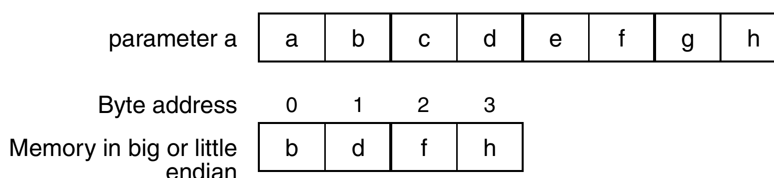


Figure 3-688. __ev_stwbo[u] Results in Big- and Little-Endian Modes

NOTE

An alignment exception may occur if the effective address (EA) is not word aligned. Also, if machine register r0 is used to pass parameter **b**, then a zero value is used as the base value when calculating EA. For the 'with update' form (U=1), parameter **b** is both an input and output operand. Also, for the 'with update' form (U=1), parameter **c** cannot be 0.

U	returns	a	b	c	Maps to
U = 0	*void*	__ev64_opaque__	uint32_t *	5-bit unsigned literal	**evstwbo a,c(b)**
U = 1	*void*	__ev64_opaque__	uint32_t *&	5-bit unsigned literal	**evstwbou a,c(b)**

__ev_stwbo[m]x __ev_stwbo[m]x

Vector Store Word of Four Bytes from Odd [with Modify] Indexed

__ev_stwbox (a,b,c) (M = 0)

__ev_stwbomx (a,b,c) (M = 1)

```
if b=0 & M=1 then take_illegal_exception
if b=0 & M=0 then temp ←0
else temp ←(b)
EA ←temp + (c)
MEM(EA,1) ←a8:15
MEM(EA+1,1)* ←a24:31
MEM(EA+2,1)* ←a40:47
MEM(EA+3,1)* ←a56:63

if (M=1) then b32:63 ←calc_b_update(b,c)

        * - may wrap at length boundary for M=1 and mode 1000.
```

The odd byte elements of parameter **a** are stored as four bytes in storage addressed by EA.

If M=1 (‘with modify’), $\mathbf{b}_{32:63}$ is updated with an address value determined by the mode specifier in $\mathbf{b}_{0:3}$. See Section 3.2.3, “Addressing Modes - Modify forms.

Figure 3-689 shows how bytes are stored in memory as determined by the endian mode.

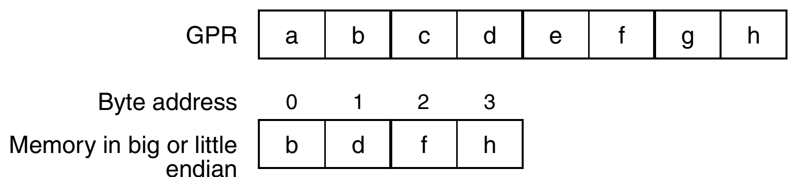


Figure 3-689. __ev_stwbo[m]x Results in Big- and Little-Endian Modes

NOTE

An alignment exception may occur if the effective address (EA) is not word aligned. Also, if machine register r0 is used to pass parameter **c**, then a zero value is used as the base value when calculating EA. And, for the ‘with modify’ form (M=1), parameter **c** is both an input and output operand.

M	returns	a	b	c	Maps to
M = 0	*void*	__ev64_opaque__	uint32_t *	int32_t	**evstwbox a,b,c**
M = 1	*void*	__ev64_opaque__	uint32_t *&	int32_t	**evstwbomx a,b,c**

__ev_stwhe[u] __ev_stwhe[u]

Vector Store Word of Two Half Words from Even [with Update]

__ev_stwhe (a,b,c) (U = 0)

__ev_stwheu (a,b,c) (U = 1)

```
if (b = r0) then temp ←0
else temp ←(b)
EA ←temp + EXTZ(c*4)
MEM(EA,2) ←a0:15
MEM(EA+2,2) ←a32:47

if (U=1) then b ← EA
```

The even half words from each element of parameter **a** are stored as two half words in storage addressed by EA.

If U=1 (‘with update’), EA is placed into parameter **b**.

Figure 3-690 shows how bytes are stored in memory as determined by the endian mode.

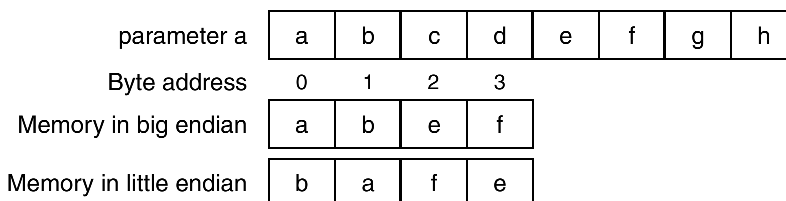


Figure 3-690. __ev_stwhe[u] Results in Big- and Little-Endian Modes

NOTE

An alignment exception may occur if the effective address (EA) is not word aligned. Also, if machine register r0 is used to pass parameter **b**, then a zero value is used as the base value when calculating EA. For the ‘with update’ form (U=1), parameter **b** is both an input and output operand. Also, for the ‘with update’ form (U=1), parameter **c** cannot be 0.

U	returns	a	b	c	Maps to
U = 0	*void*	__ev64_opaque__	uint32_t *	5-bit unsigned literal	**evstwhe a,c(b)**
U = 1	*void*	__ev64_opaque__	uint32_t *&	5-bit unsigned literal	**evstwheu a,c(b)**

__ev_stwhe[m]x

Vector Store Word of Two Half Words from Even [with Modify] Indexed

__ev_stwhex (a,b,c) (M = 0)

__ev_stwhemx (a,b,c) (M = 1)

```
if b=0 & M=1 then take_illegal_exception
if b=0 & M=0 then temp ←0
else temp ←(b)
EA ←temp + (c)
MEM(EA,2)* ←a0:15
MEM(EA+2,2)* ←a32:47

if (M=1) then b32:63 ←calc_b_update(b,c)

        * - may wrap at length boundary for M=1 and mode 1000.
```

The even halfwords from each element of parameter **a** are stored as two halfwords in storage addressed by EA.

If M=1 (‘with modify’), parameter $\mathbf{b}_{32:63}$ is updated with an address value determined by the mode specifier in parameter $\mathbf{b}_{0:3}$. See Section 3.2.3, “Addressing Modes - Modify forms.

Figure 3-691 shows how bytes are stored in memory as determined by the endian mode.

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

Figure 3-691. __ev_stwhe[m]x Results in Big- and Little-Endian Modes

NOTE

An alignment exception may occur if the effective address (EA) is not word aligned. Also, if machine register r0 is used to pass parameter **c**, then a zero value is used as the base value when calculating EA. And, for the ‘with modify’ form (M=1), parameter **c** is both an input and output operand.

M	returns	a	b	c	Maps to
M = 0	*void*	__ev64_opaque__	uint32_t *	int32_t	**evstwhex a,b,c**
M = 1	*void*	__ev64_opaque__	uint32_t *&	int32_t	**evstwhemx a,b,c**

__ev_stwho[u] __ev_stwho[u]

Vector Store Word of Two Half Words from Odd [with Update]

__ev_stwho (a,b,c) (U = 0)

__ev_stwhou (a,b,c) (U = 1)

```
if (b = r0) then temp ←0
else temp ←(b)
EA ←temp + EXTZ(c*4)
MEM(EA,2) ←a16:31
MEM(EA+2,2) ←a48:63

if (U=1) then b ← EA
```

The odd half words from each element of parameter **a** are stored as two half words in storage addressed by EA.

If U=1 (‘with update’), EA is placed into parameter **b**.

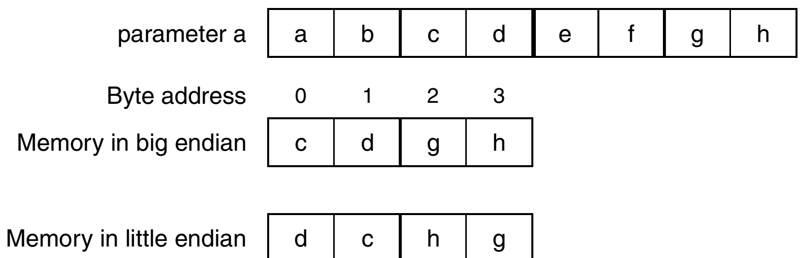


Figure 3-692. __ev_stwho[u] Results in Big- and Little-Endian Modes

NOTE

An alignment exception may occur if the effective address (EA) is not word aligned. Also, if machine register r0 is used to pass parameter **b**, then a zero value is used as the base value when calculating EA. For the ‘with update’ form (U=1), parameter **b** is both an input and output operand. Also, for the ‘with update’ form (U=1), parameter **c** cannot be 0.

U	returns	a	b	c	Maps to
U = 0	*void*	__ev64_opaque__	uint32_t *	5-bit unsigned literal	**evstwho a,c(b)**
U = 1	*void*	__ev64_opaque__	uint32_t *&	5-bit unsigned literal	**evstwhou a,c(b)**

__ev_stwho[m]x

__ev_stwho[m]x

Vector Store Word of Two Half Words from Odd [with Modify] Indexed

__ev_stwhox (a,b,c) (M = 0)

__ev_stwhomx (a,b,c) (M = 1)

```
if b=0 & M=1 then take_illegal_exception
if b=0 & M=0 then temp ←0
else temp ←(b)
EA ←temp + (c)
MEM(EA,2)* ←a16:31
MEM(EA+2,2)* ←a48:63

if (M=1) then b32:63 ←calc_b_update(b,c)

        * - may wrap at length boundary for M=1 and mode 1000.
```

The odd halfwords from each element of parameter **a** are stored as two halfwords in storage addressed by EA.

If M=1 ('with modify'), $\mathbf{b}_{32:63}$ is updated with an address value determined by the mode specifier in $\mathbf{b}_{0:3}$. See Section 3.2.3, "Addressing Modes - Modify forms.

Figure 3-693 shows how bytes are stored in memory as determined by the endian mode.

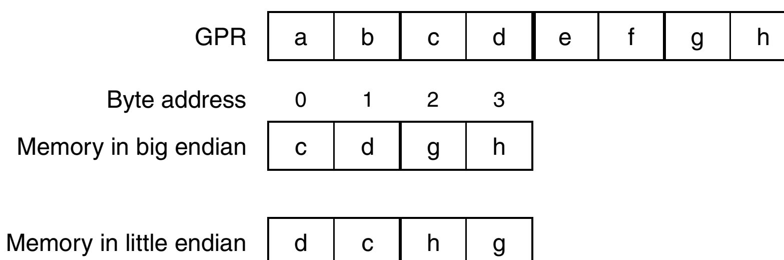


Figure 3-693. __ev_stwho[m]x Results in Big- and Little-Endian Modes

NOTE

An alignment exception may occur if the effective address (EA) is not word aligned. Also, if machine register r0 is used to pass parameter **c**, then a zero value is used as the base value when calculating EA. And, for the 'with modify' form (M=1), parameter **c** is both an input and output operand.

M	returns	a	b	c	Maps to
M = 0	*void*	__ev64_opaque__	uint32_t *	int32_t	**evstwhox a,b,c**
M = 1	*void*	__ev64_opaque__	uint32_t *&	int32_t	**evstwhomx a,b,c**

__ev_stwwe[u] __ev_stwwe[u]

Vector Store Word of Word from Even [with Update]

__ev_stwwe (a,b,c) (U = 0)

__ev_stwweu (a,b,c) (U = 1)

```
if (b = r0) then temp ←0
else temp ←(b)
EA ←temp + EXTZ(c*4)
MEM(EA,4) ←a0:31

if (U=1) then b ← EA
```

The even word of parameter **a** is stored in storage addressed by EA.

Figure 3-694 shows how bytes are stored in memory as determined by the endian mode.

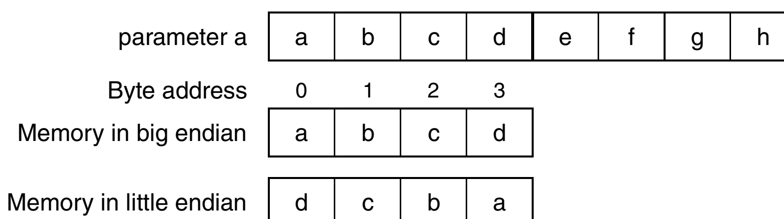


Figure 3-694. __ev_stwwe[u] Results in Big- and Little-Endian Modes

NOTE

An alignment exception may occur if the effective address (EA) is not word aligned. Also, if machine register r0 is used to pass parameter **b**, then a zero value is used as the base value when calculating EA. For the ‘with update’ form (U=1), parameter **b** is both an input and output operand. Also, for the ‘with update’ form (U=1), parameter **c** cannot be 0.

U	returns	a	b	c	Maps to
U = 0	*void*	__ev64_opaque__	uint32_t *	5-bit unsigned literal	**evstwwe a,c(b)**
U = 1	*void*	__ev64_opaque__	uint32_t *&	5-bit unsigned literal	**evstwweu a,c(b)**

__ev_stwwe[m]x __ev_stwwe[m]x

Vector Store Word of Word from Even [with Modify] Indexed

__ev_stwwex (a,b,c) (M = 0)

__ev_stwwemx (a,b,c) (M = 1)

```
if b=0 & M=1 then take_illegal_exception
if b=0 & M=0 then temp ←0
else temp ←(b)
EA ←temp + (c)
MEM(EA,4)* ←a0:31

if (M=1) then b32:63 ←calc_b_update(b,c)

        * - may wrap at length boundary for M=1 and mode 1000.
```

The even word of parameter **a** is stored in storage addressed by EA.

If M=1 (‘with modify’), parameter $\mathbf{b}_{32:63}$ is updated with an address value determined by the mode specifier in parameter $\mathbf{b}_{0:3}$. See Section 3.2.3, “Addressing Modes - Modify forms.

Figure 3-695 shows how bytes are stored in memory as determined by the endian mode.

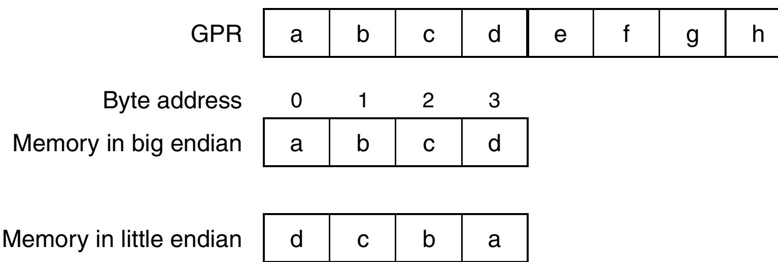


Figure 3-695. __ev_stwwe[m]x Results in Big- and Little-Endian Modes

NOTE

An alignment exception may occur if the effective address (EA) is not word aligned. Also, if machine register r0 is used to pass parameter **c**, then a zero value is used as the base value when calculating EA. And, for the ‘with modify’ form (M=1), parameter **c** is both an input and output operand.

M	returns	a	b	Maps to
M = 0	*void*	__ev64_opaque__	uint32_t *	**evstwwex a,b,c**
M = 1	*void*	__ev64_opaque__	uint32_t *&	**evstwwemx a,b,c**

__ev_stwwo[u] __ev_stwwo[u]

Vector Store Word of Word from Odd [with Update]

__ev_stwwo (a,b,c) (U = 0)

__ev_stwwou (a,b,c) (U = 1)

```
if (b = r0) then temp ←0
else temp ←(b)
EA ←temp + EXTZ(c*4)
MEM(EA,4) ←a32:63

if (U=1) then b ← EA
```

The odd word of parameter **a** is stored in storage addressed by EA.

Figure 3-696 shows how bytes are stored in memory as determined by the endian mode.

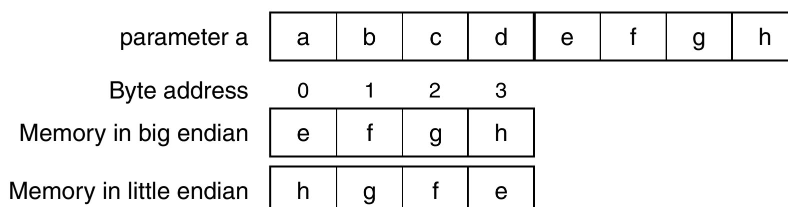


Figure 3-696. __ev_stwwo[u] Results in Big- and Little-Endian Modes

NOTE

An alignment exception may occur if the effective address (EA) is not word aligned. Also, if machine register r0 is used to pass parameter **b**, then a zero value is used as the base value when calculating EA. For the ‘with update’ form (U=1), parameter **b** is both an input and output operand. Also, for the ‘with update’ form (U=1), parameter **c** cannot be 0.

U	returns	a	b	c	Maps to
U = 0	*void*	__ev64_opaque__	uint32_t *	5-bit unsigned literal	**evstwwo a,c(b)**
U = 1	*void*	__ev64_opaque__	uint32_t *&	5-bit unsigned literal	**evstwwou a,c(b)**

__ev_stwwo[m]x __ev_stwwo[m]x

Vector Store Word of Word from Odd [with Modify] Indexed

__ev_stwwox (a,b,c) (M = 0)

__ev_stwwomx (a,b,c) (M = 1)

```
if b=0 & M=1 then take_illegal_exception
if b=0 & M=0 then temp ←0
else temp ←(b)
EA ←temp + (c)
MEM(EA,4)* ←a32:63
if (M=1) then b32:63 ←calc_b_update(b,c)
        * - may wrap at length boundary for M=1 and mode 1000.
```

The odd word of parameter **a** is stored in storage addressed by EA.

If M=1 (‘with modify’), parameter $\mathbf{b}_{32:63}$ is updated with an address value determined by the mode specifier in parameter $\mathbf{b}_{0:3}$. See Section 3.2.3, “Addressing Modes - Modify forms.

Figure 3-697 shows how bytes are stored in memory as determined by the endian mode.

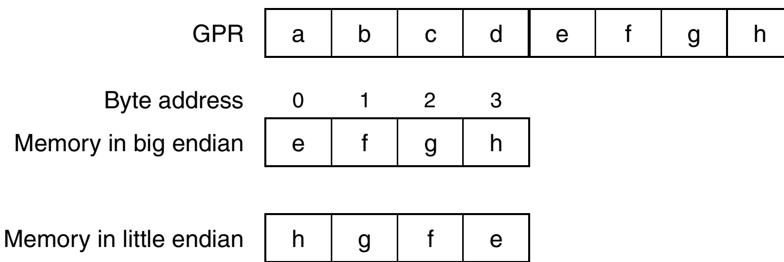


Figure 3-697. __ev_stwwo[m]x Results in Big- and Little-Endian Modes

NOTE

An alignment exception may occur if the effective address (EA) is not word aligned. Also, if machine register r0 is used to pass parameter **c**, then a zero value is used as the base value when calculating EA. And, for the ‘with modify’ form (M=1), parameter **c** is both an input and output operand.

M	returns	a	b	c	Maps to
M = 0	*void*	__ev64_opaque__	uint32_t *	int32_t	**evstwwox a,b,c**
M = 1	*void*	__ev64_opaque__	uint32_t *&	int32_t	**evstwwomx a,b,c**

__ev_subf2add2h __ev_subf2add2h

Vector Subtract from Upper 2 / Add Lower 2 Half Words

d = __ev_subf2add2h (a,b)

```
d0:15 ←b0:15 - a0:15// Modulo
d16:31 ←b16:31 - a16:31// Modulo
d32:47 ←b32:47 + a32:47// Modulo
d48:63 ←b48:63 + a48:63// Modulo
```

The upper two half word elements of parameter **a** are subtracted from the upper 2 half word elements of parameter **b**, the lower two half word elements of parameter **a** are added to the lower two half word elements of parameter **b**, and the results are placed in parameter **d**. The sum and difference are modulo.

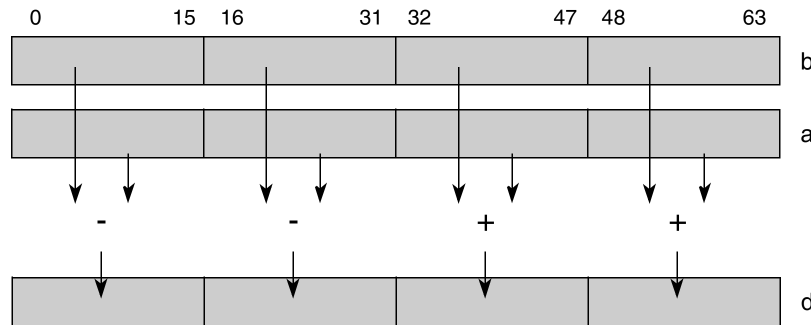


Figure 3-698. Vector Subtract from Upper 2 / Add Lower 2 Half Words (__ev_subf2add2h)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evsubf2add2h d,a,b**

__ev_subf2add2hss __ev_subf2add2hss

Vector Subtract from Upper 2 / Add Lower 2 Half Words Signed and Saturate

d = __ev_subf2add2hss (a,b)

```
// h0
temp0:31 ←EXTS(b0:15) - EXTS(a0:15)
ovh0 ←temp15 ⊕ temp16
d0:15 ←SATURATE(ovh0, temp15, 0x8000, 0x7fff, temp16:31)
// h1
temp0:31 ←EXTS(b16:31) - EXTS(a16:31)
ovh1 ←temp15 ⊕ temp16
d16:31 ←SATURATE(ovh1, temp15, 0x8000, 0x7fff, temp16:31)
// h2
temp0:31 ←EXTS(b32:47) + EXTS(a32:47)
ovh2 ←temp15 ⊕ temp16
d32:47 ←SATURATE(ovh2, temp15, 0x8000, 0x7fff, temp16:31)
// h3
temp0:31 ←EXTS(b48:63) + EXTS(a48:63)
ovh3 ←temp15 ⊕ temp16
d48:63 ←SATURATE(ovh3, temp15, 0x8000, 0x7fff, temp16:31)
ovh ←ovh0 | ovh1
ovl ←ovh2 | ovh3

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

The upper two signed half word elements of parameter **a** are subtracted from the upper two signed half word elements of parameter **b**, the lower two signed half word elements of parameter **a** are added to the lower two signed half word elements of parameter **b**, saturating if overflow or underflow occurs, and the results are placed in parameter **d**. Any overflow or underflow is recorded in the SPEFSCR overflow and summary overflow bits.

Other registers altered: SPEFSCR

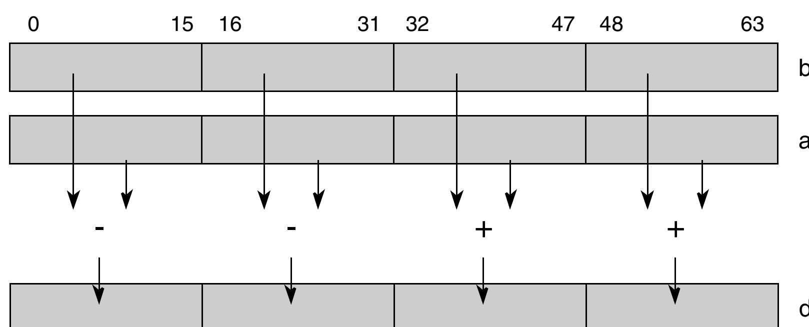


Figure 3-699. Vector Subtract from Upper 2 / Add Lower 2 Half Words Signed and Saturate (__ev_subf2add2hss)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evsubf2add2hss d,a,b**

__ev_subfaddh

__ev_subfaddh

Vector Subtract from / Add Half Words

d = __ev_subfaddh (a,b)

```
d0:15 ←b0:15 - a0:15// Modulo difference, b - a
d16:31 ←b16:31 + a16:31// Modulo sum
d32:47 ←b32:47 - a32:47// Modulo difference, b - a
d48:63 ←b48:63 + a48:63// Modulo sum
```

The even half word elements of parameter **a** are subtracted from the even half word elements of parameter **b**, the odd half word elements of parameter **a** are added to the odd half word elements of parameter **b**, and the results are placed in parameter **d**. The sum and difference are modulo.

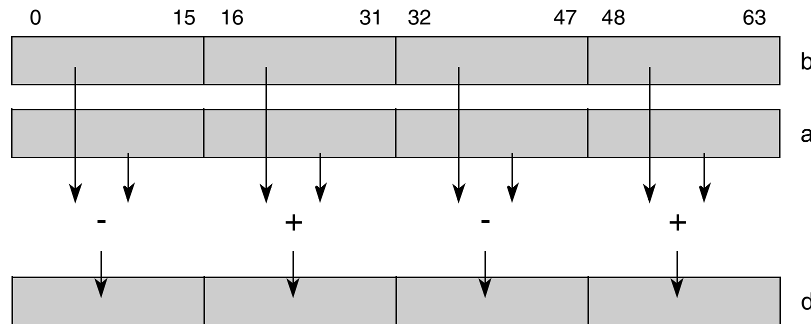


Figure 3-700. Vector Subtract from / Add Half Words (__ev_subfaddh)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evsubfaddh d,a,b**

__ev_subfaddhss __ev_subfaddhss

Vector Subtract from / Add Half Words Signed and Saturate

d = __ev_subfaddhss (a,b)

```
// h0
temp0:31 ←EXTS(b0:15) - EXTS(a0:15)
ovh0 ←temp15 ⊕ temp16
d0:15 ←SATURATE(ovh0, temp15, 0x8000, 0x7fff, temp16:31)
// h1
temp0:31 ←EXTS(b16:31) + EXTS(a16:31)
ovh1 ←temp15 ⊕ temp16
d16:31 ←SATURATE(ovh1, temp15, 0x8000, 0x7fff, temp16:31)
// h2
temp0:31 ←EXTS(b32:47) - EXTS(a32:47)
ovh2 ←temp15 ⊕ temp16
d32:47 ←SATURATE(ovh2, temp15, 0x8000, 0x7fff, temp16:31)
// h3
temp0:31 ←EXTS(b48:63) + EXTS(a48:63)
ovh3 ←temp15 ⊕ temp16
d48:63 ←SATURATE(ovh3, temp15, 0x8000, 0x7fff, temp16:31)

ovh ←ovh0 | ovh1
ovl ←ovh2 | ovh3

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

The even signed half word elements of parameter **a** are subtracted from the even signed half word elements of parameter **b**, the odd signed half word elements of parameter **a** are added to the odd signed half word elements of parameter **b**, saturating if overflow or underflow occurs, and the results are placed in parameter **d**. Any overflow or underflow is recorded in the SPEFSCR overflow and summary overflow bits.

Other registers altered: SPEFSCR

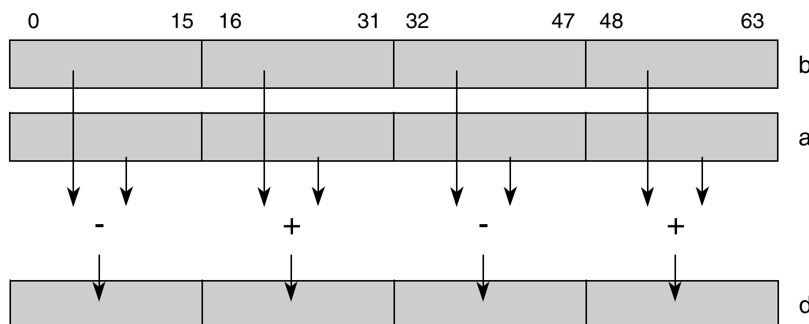


Figure 3-701. Vector Subtract from / Add Half Words Signed and Saturate (__ev_subfaddhss)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evsubfaddhss d,a,b**

__ev_subfaddhx __ev_subfaddhx

Vector Subtract from / Add Half Words Exchanged

d = __ev_subfaddhx (a,b)

```
// h0
d0:15 ←b0:15 - a16:31 // modulo difference

// h1
d16:31 ←b16:31 + a0:15 // modulo sum

// h2
d32:47 ←b32:47 - a48:63 // modulo difference

// h3
d48:63 ←b48:63 + a32:47 // modulo sum
```

The odd exchanged half word elements of parameter **a** are subtracted from the even half word elements of parameter **b**, the even exchanged half words of parameter **a** are added to the odd half words of parameter **b**, and the results are placed in parameter **d**. The sum and differences are modulo.

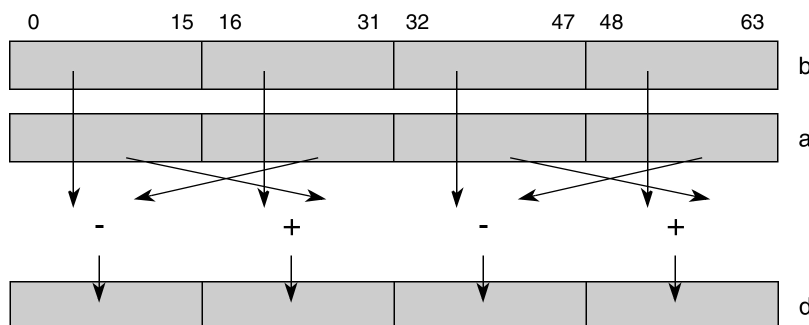


Figure 3-702. Vector Subtract from / Add Half Words Exchanged (__ev_subfaddhx)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evsubfaddhx d,a,b**

__ev_subfaddhxss __ev_subfaddhxss

Vector Subtract from / Add Half Words Exchanged, Signed and Saturate

d = __ev_subfaddhxss (a,b)

```
// h0
temp0:31 ←EXTS(b0:15) - EXTS(a16:31)
ovh0 ←temp15 ⊕ temp16
d0:15 ←SATURATE(ovh0, temp15, 0x8000, 0x7fff, temp16:31)
// h1
temp0:31 ←EXTS(b16:31) + EXTS(a0:15)
ovh1 ←temp15 ⊕ temp16
d16:31 ←SATURATE(ovh1, temp15, 0x8000, 0x7fff, temp16:31)
// h2
temp0:31 ←EXTS(b32:47) - EXTS(a48:63)
ovh2 ←temp15 ⊕ temp16
d32:47 ←SATURATE(ovh2, temp15, 0x8000, 0x7fff, temp16:31)
// h3
temp0:31 ←EXTS(b48:63) + EXTS(a32:47)
ovh3 ←temp15 ⊕ temp16
d48:63 ←SATURATE(ovh3, temp15, 0x8000, 0x7fff, temp16:31)

ovh ←ovh0 | ovh1
ovl ←ovh2 | ovh3

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

The odd exchanged signed half word elements of parameter **a** are subtracted from the even signed half word elements of parameter **b**, the even exchanged signed half words of parameter **a** are added to the odd signed half words of parameter **b**, saturating if overflow or underflow occurs, and the results are placed in parameter **d**. Any overflow or underflow is recorded in the SPEFSCR overflow and summary overflow bits.

Other registers altered: SPEFSCR

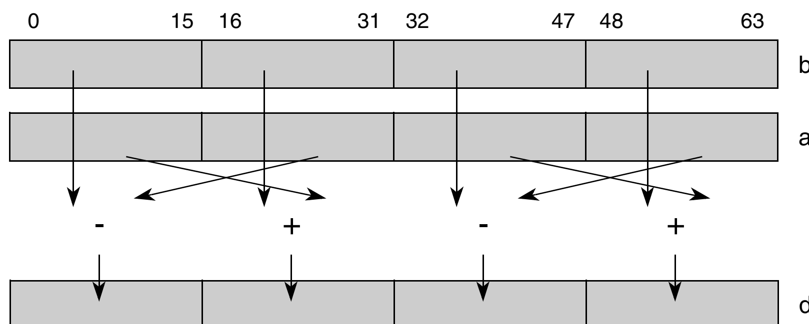


Figure 3-703. Vector Subtract from / Add Half Words Exchanged, Signed and Saturate (__ev_subfaddhxss)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evsubfaddhxss d,a,b**

__ev_subfaddw __ev_subfaddw

Vector Subtract from / Add Word

d = __ev_subfaddw (a,b)

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

The high word element of parameter **a** is subtracted from the high word element of parameter **b**, the low word element of parameter **a** is added to the low word element of parameter **b**, and the results are placed in parameter **d**.

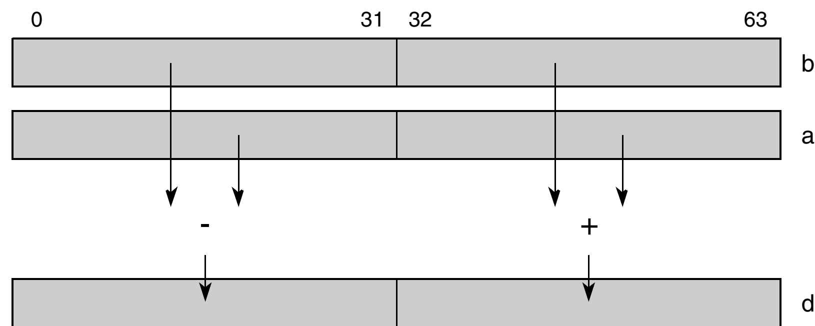


Figure 3-704. Vector Subtract from / Add Word (__ev_subfaddw)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evsubfaddw d,a,b**

__ev_subfaddwss __ev_subfaddwss

Vector Subtract from / Add Word Signed and Saturate

d = __ev_subfaddwss (a,b)

```
// h0
temp0:32 ←EXTS(b0:31) - EXTS(a0:31)
ovh ←temp0 ⊕ temp1
d0:15 ←SATURATE(ovh, temp0, 0x8000_0000, 0x7fff_ffff, temp1:32)

// h1
temp0:31 ←EXTS(b32:63) + EXTS(a32:63)
ovl ←temp0 ⊕ temp1
d16:31 ←SATURATE(ovl, temp0, 0x8000_0000, 0x7fff_ffff, temp1:32)

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

The high word element of parameter **a** is subtracted from the high word element of parameter **b**, saturating if overflow or underflow occurs, the low word element of parameter **a** is added to the low word element of parameter **b**, saturating if overflow or underflow occurs, and the results are placed in parameter **d**. Any overflow or underflow is recorded in the SPEFSCR overflow and summary overflow bits.

Other registers altered: SPEFSCR

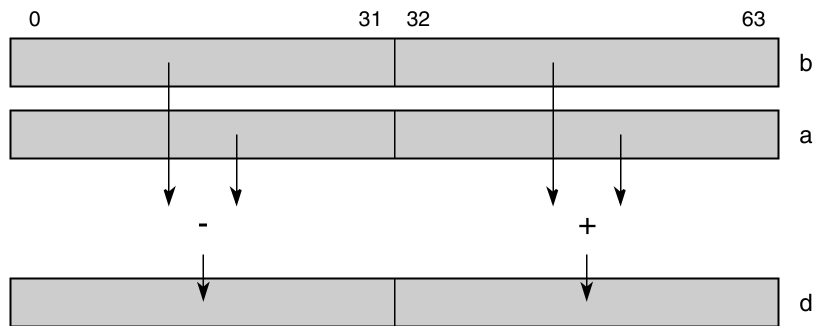


Figure 3-705. Vector Subtract from / Add Word Signed and Saturate (__ev_subfaddwss)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evsubfaddwss d,a,b**

__ev_subfaddwx

__ev_subfaddwx

Vector Add / Subtract from Word Exchanged

d = __ev_subfaddwx (a,b)

$d_{0:31} \leftarrow b_{0:31} - a_{32:63}$ // Modulo
$d_{32:63} \leftarrow b_{32:63} + a_{0:31}$ // Modulo

The low word element of parameter **a** is subtracted from the high word element of parameter **b**, the high word element of parameter **a** is added to the low word element of parameter **b**, and the results are placed in parameter **d**.

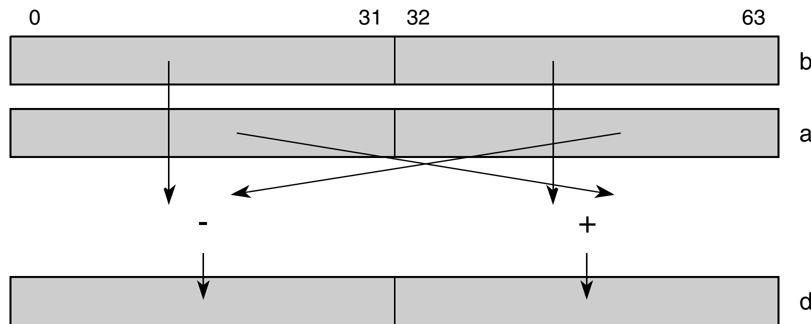


Figure 3-706. Vector Subtract from / Add Word Exchanged (__ev_subfaddwx)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evsubfaddwx d,a,b**

__ev_subfaddwxss __ev_subfaddwxss

Vector Add / Subtract from Word Exchanged Signed and Saturate

d = __ev_subfaddwxss (a,b)

```
// h0
temp0:32 ←EXTS(b0:31) - EXTS(a32:63)
ovh ←temp0 ⊕ temp11
d0:15 ←SATURATE(ovh, temp0, 0x8000_0000, 0x7fff_ffff, temp1:32)

// h1
temp0:31 ←EXTS(b32:63) + EXTS(a0:31)
ovl ←temp0 ⊕ temp1
d16:31 ←SATURATE(ovl, temp0, 0x8000_0000, 0x7fff_ffff, temp1:32)

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

The low word element of parameter **a** is subtracted from the high word element of parameter **b**, saturating if overflow or underflow occurs, the high word element of parameter **a** is added to the low word element of parameter **b**, saturating if overflow or underflow occurs, and the results are placed in parameter **d**. Any overflow or underflow is recorded in the SPEFSCR overflow and summary overflow bits.

Other registers altered: SPEFSCR

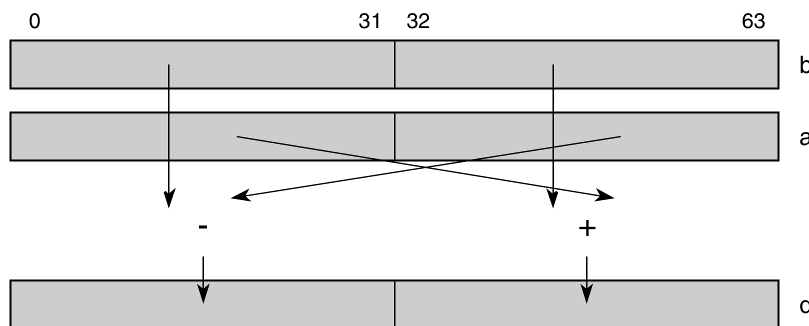


Figure 3-707. Vector Add / Subtract from Word Exchanged Signed and Saturate (__ev_addsubfwxss)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evaddsubfwxss d,a,b**

__ev_subfb __ev_subfb

Vector Subtract from Byte

d = __ev_subfb (a,b)

```
d0:7 ←b0:7 - a0:7// Modulo difference
d8:15 ←b8:15 - a8:15// Modulo difference
d16:23 ←b16:23 - a16:23// Modulo difference
d24:31 ←b24:31 - a24:31// Modulo difference
d32:39 ←b32:39 - a32:39// Modulo difference
d40:47 ←b40:47 - a40:47// Modulo difference
d48:55 ←b48:55 - a48:55// Modulo difference
d56:63 ←b56:63 - a56:63// Modulo difference
```

The eight byte elements of parameter **a** are subtracted from the corresponding elements of parameter **b** and the results are placed in parameter **d**. The difference is a modulo difference.

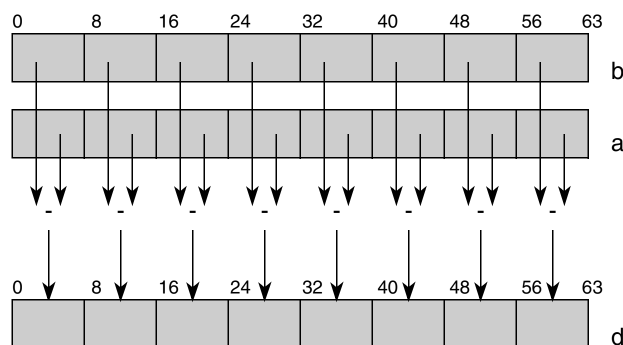


Figure 3-708. Vector Subtract from Byte (__ev_subfb)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evsubfb d,a,b**

__ev_subfbss __ev_subfbss

Vector Subtract from Byte Signed and Saturate

d = __ev_subfbss (a,b)

```
temp0:8 ←EXTS(b0:7) - EXTS(a0:7); ovb0 ←temp0 ⊕ temp1
d0:7 ←SATURATE(ovb0, temp0, 0x80, 0x7f, temp1:8)
temp0:8 ←EXTS(b8:15) - EXTS(a8:15); ovb1 ←temp0 ⊕ temp1
d8:15 ←SATURATE(ovb1, temp0, 0x80, 0x7f, temp1:8)
temp0:8 ←EXTS(b16:23) - EXTS(a16:23); ovb2 ←temp0 ⊕ temp1
d16:23 ←SATURATE(ovb2, temp0, 0x80, 0x7f, temp1:8)
temp0:8 ←EXTS(b24:31) - EXTS(a24:31); ovb3 ←temp0 ⊕ temp1
d24:31 ←SATURATE(ovb3, temp0, 0x80, 0x7f, temp1:8)
temp0:8 ←EXTS(b32:39) - EXTS(a32:39); ovb4 ←temp0 ⊕ temp1
d32:39 ←SATURATE(ovb4, temp0, 0x80, 0x7f, temp1:8)
temp0:8 ←EXTS(b40:47) - EXTS(a40:47); ovb5 ←temp0 ⊕ temp1
d40:47 ←SATURATE(ovb5, temp0, 0x80, 0x7f, temp1:8)
temp0:8 ←EXTS(b48:55) - EXTS(a48:55); ovb6 ←temp0 ⊕ temp1
d48:55 ←SATURATE(ovb6, temp0, 0x80, 0x7f, temp1:8)
temp0:8←EXTS(b56:63) - EXTS(a56:63); ovb7 ←temp0 ⊕ temp1
d56:63 ←SATURATE(ovb7, temp0, 0x80, 0x7f, temp1:8)

ovh ←ovb0 | ovb1 | ovb2 | ovb3
ovl ←ovh4 | ovh5 | ovb6 | ovb7
SPEFSCROVH ←ovh
SPEFSCROV ←ovl
SPEFSCRSOVH ←SPEFSCRSOVH | ovh
SPEFSCRSOV ←SPEFSCRSOV | ovl
```

The eight signed byte elements of parameter **a** are subtracted from the corresponding signed elements of parameter **b**, saturating if overflow or underflow occurs, and the results are placed in parameter **d**. Any overflow or underflow is recorded in the SPEFSCR overflow and summary overflow bits.

Other registers altered: SPEFSCR

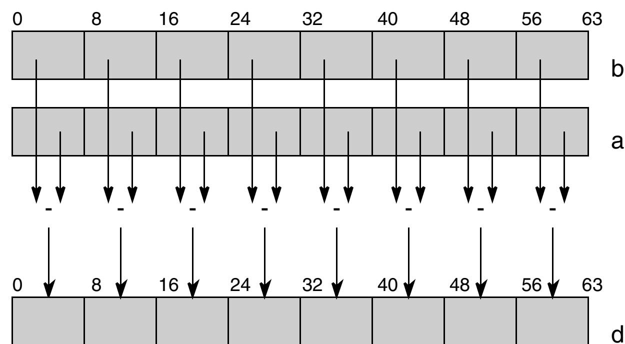


Figure 3-709. Vector Subtract from Byte Signed and Saturate (__ev_subfbss)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evsubfbss d,a,b**

__ev_subfbus __ev_subfbus

Vector Subtract from Byte Unsigned and Saturate

d = __ev_subfbus (a,b)

```
temp0:8 ←EXTZ(b0:7) - EXTZ(a0:7); ovb0 ←temp0
d0:7 ←SATURATE(ovb0, temp0, 0x0, 0x0, temp1:8)
temp0:8 ←EXTZ(b8:15) - EXTZ(a8:15); ovb1 ←temp0
d8:15 ←SATURATE(ovb1, temp0, 0x0, 0x0, temp1:8)
temp0:8 ←EXTZ(b16:23) - EXTZ(a16:23); ovb2 ←temp0
d16:23 ←SATURATE(ovb2, temp0, 0x0, 0x0, temp1:8)
temp0:8 ←EXTZ(b24:31) - EXTZ(a24:31); ovb3 ←temp0
d24:31 ←SATURATE(ovb3, temp0, 0x0, 0x0, temp1:8)
temp0:8 ←EXTZ(b32:39) - EXTZ(a32:39); ovb4 ←temp0
d32:39 ←SATURATE(ovb4, temp0, 0x0, 0x0, temp1:8)
temp0:8 ←EXTZ(b40:47) - EXTZ(a40:47); ovb5 ←temp0
d40:47 ←SATURATE(ovb5, temp0, 0x0, 0x0, temp1:8)
temp0:8 ←EXTZ(b48:55) - EXTZ(a48:55); ovb6 ←temp0
d48:55 ←SATURATE(ovb6, temp0, 0x0, 0x0, temp1:8)
temp0:8←EXTZ(b56:63) - EXTZ(a56:63); ovb7 ←temp0
d56:63 ←SATURATE(ovb7, temp0, 0x0, 0x0, temp1:8)

ovh ←ovb0 | ovb1 | ovb2 | ovb3
ovl ←ovh4 | ovh5 | ovb6 | ovb7
SPEFSCROVH ←ovh
SPEFSCROV ←ovl
SPEFSCRSOVH ←SPEFSCRSOVH | ovh
SPEFSCRSOV ←SPEFSCRSOV | ovl
```

The eight unsigned byte elements of parameter **a** are subtracted from the corresponding unsigned elements of parameter **b**, saturating if underflow occurs, and the results are placed in parameter **d**. Any underflow is recorded in the SPEFSCR overflow and summary overflow bits.

Other registers altered: SPEFSCR

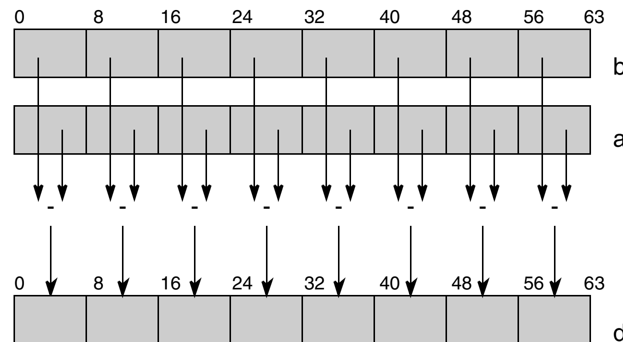


Figure 3-710. Vector Subtract from Byte Unsigned and Saturate (__ev_subfbus)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evsubfbus d,a,b**

__ev_subfd __ev_subfd

Vector Subtract from Doubleword

d = __ev_subfd (a,b)

$d_{0:63} \leftarrow b_{0:63} - a_{0:63}$

The 64-bit value in **a** is subtracted from the 64-bit value in parameter **b** and the results are placed into parameter **d**.

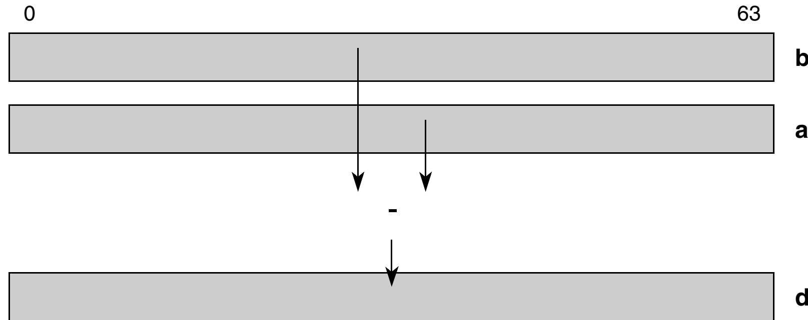


Figure 3-711. Vector Subtract from Doubleword (__ev_subfd)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evsubfd d,a,b**

__ev_subfdss __ev_subfdss

Vector Subtract from Doubleword Signed and Saturate

d = __ev_subfdss (a,b)

```
temp0:64 ←EXTS65(b0:63) - EXTS65(a0:63)
ov ←temp0 ⊕ temp1
d0:63 ←SATURATE(ov, temp0, 0x8000_0000_0000_0000, 0x7fff_ffff_ffff_ffff, temp1:64)

SPEFSCROVH ←0
SPEFSCROV ←ov
SPEFSCRSOV ←SPEFSCRSOV | ov.
```

The signed doubleword in parameter **a** is subtracted from the signed doubleword in parameter **b**, saturating if overflow or underflow occurs, and the result is placed into parameter **d**. Any overflow or underflow is recorded in the SPEFSCR overflow and summary overflow bits.

Other registers altered: SPEFSCR

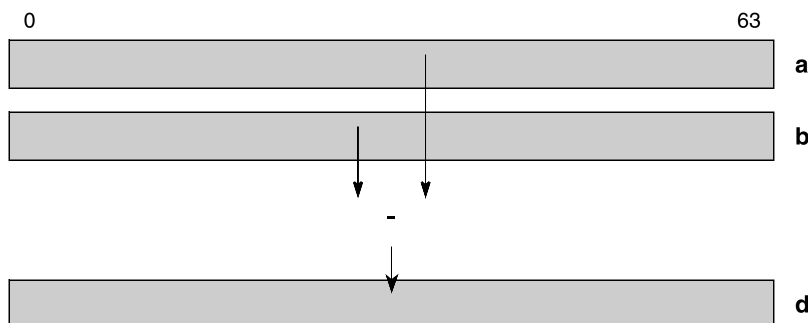


Figure 3-712. Vector Subtract from Doubleword Signed and Saturate (__ev_subfdss)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evsubfdss d,a,b**

__ev_subfdus __ev_subfdus

Vector Subtract from Doubleword Unsigned and Saturate

d = __ev_subfdus (a,b)

```
temp0:64 ←EXTZ65(b0:63) - EXTZ65(a0:63)
ov ←temp0
d0:63 ←SATURATE(ov, temp0, 0x0000_0000_0000_0000, 0x0000_0000_0000_0000, temp1:64)

SPEFSCROVH ←0
SPEFSCROV ←ov
SPEFSCRSOV ←SPEFSCRSOV | ov.
```

The unsigned doubleword in parameter **a** is subtracted from the unsigned doubleword in parameter **b**, saturating if underflow occurs, and the result is placed into parameter **d**. Any underflow is recorded in the SPEFSCR overflow and summary overflow bits.

Other registers altered: SPEFSCR

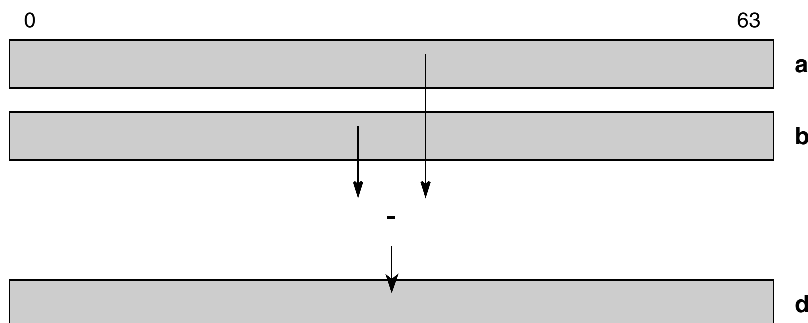


Figure 3-713. Vector Subtract from Doubleword Unsigned and Saturate (__ev_subfdus)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evsubfdus d,a,b**

__ev_subfh

__ev_subfh

Vector Subtract from Half Word

d = __ev_subfh (a,b)

```
d0:15 ←b0:15 - a0:15// Modulo difference
d16:31 ←b16:31 - a16:31// Modulo difference
d32:47 ←b32:47 - a32:47// Modulo difference
d48:63 ←b48:63 - a48:63// Modulo difference
```

The four half word elements of parameter **a** are subtracted from the four half word elements of parameter **b** and the results are placed in parameter **d**. The difference is a modulo difference.

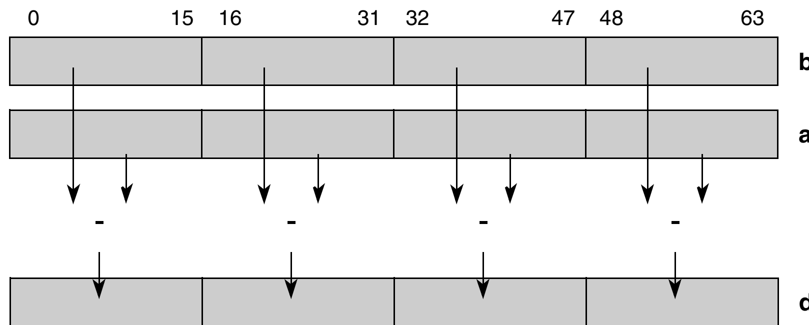


Figure 3-714. Vector Subtract from Half Word (__ev_subfh)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evsubfh d,a,b**

__ev_subfhhisw __ev_subfhhisw

Vector Subtract from Halfwords High Signed to Words

d = __ev_subfhhisw (a,b)

$d_{0:31} \leftarrow EXTS_{32}(b_{0:15})$ - $EXTS_{32}(a_{0:15})$ // Modulo

$d_{32:63} \leftarrow EXTS_{32}(b_{16:31})$ - $EXTS_{32}(a_{16:31})$ // Modulo

The high halfword elements of parameter **a** are sign-extended to 32 bits and subtracted from the respective sign-extended high halfword elements of parameter **b** and the 32-bit results are placed into parameter **d**. The difference is a modulo difference.

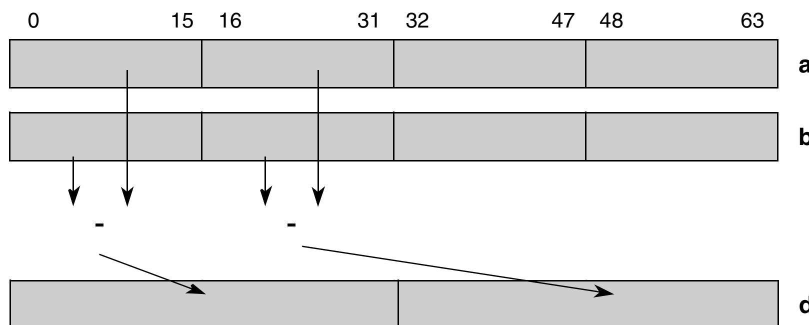


Figure 3-715. Vector Subtract from Halfwords High Signed to Words (__ev_subfhhisw)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evsubfhhisw d,a,b**

__ev_subfhhiuw __ev_subfhhiuw

Vector Subtract from Half Words High Unsigned to Words

d = __ev_subfhhiuw (a,b)

$d_{0:31} \leftarrow EXTZ_{32}(b_{0:15})$ - $EXTZ_{32}(a_{0:15})$ // Modulo

$d_{32:63} \leftarrow EXTZ_{32}(b_{16:31})$ - $EXTZ_{32}(a_{16:31})$ // Modulo

The high halfword elements of parameter **a** are zero-extended to 32 bits and subtracted from the respective zero-extended high halfword elements of parameter **b** and the 32-bit results are placed into parameter **d**. The difference is a modulo difference.

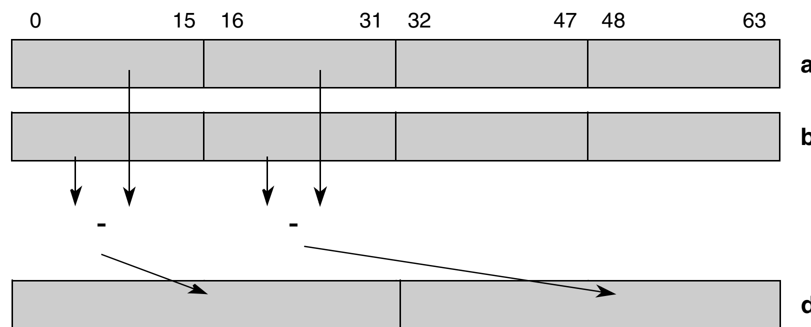


Figure 3-716. Vector Subtract from Halfwords High Unsigned to Words (__ev_subfhhiuw)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evsubfhhiuw d,a,b**

__ev_subfhlosw __ev_subfhlosw

Vector Subtract from Halfwords Low Signed to Words

d = __ev_subfhlosw (a,b)

$d_{0:31} \leftarrow EXTS_{32}(b_{32:47})$ - $EXTS_{32}(a_{32:47})$ // Modulo

$d_{32:63} \leftarrow EXTS_{32}(b_{48:63})$ - $EXTS_{32}(a_{48:63})$ // Modulo

The low halfword elements of parameter **a** are sign-extended to 32 bits and subtracted from the respective sign-extended low halfword elements of parameter **b** and the 32-bit results are placed into parameter **d**. The sum is a modulo sum.

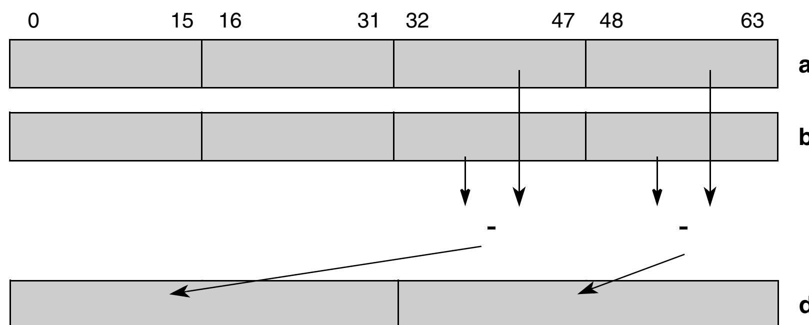


Figure 3-717. Vector Subtract from Halfwords Low Signed to Words (__ev_subfhlosw)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evsubfhlosw d,a,b**

__ev_subfhlouw __ev_subfhlouw

Vector Subtract from Halfwords Low Unsigned to Words

d = __ev_subfhlouw (a,b)

$d_{0:31} \leftarrow EXTZ_{32}(b_{32:47})$ - $EXTZ_{32}(a_{32:47})$ // Modulo

$d_{32:63} \leftarrow EXTZ_{32}(b_{48:63})$ - $EXTZ_{32}(a_{48:63})$ // Modulo

The low halfword elements of parameter **a** are zero-extended to 32 bits and subtracted from the respective zero-extended low halfword elements of parameter **b** and the 32-bit results are placed into parameter **d**. The sum is a modulo sum.

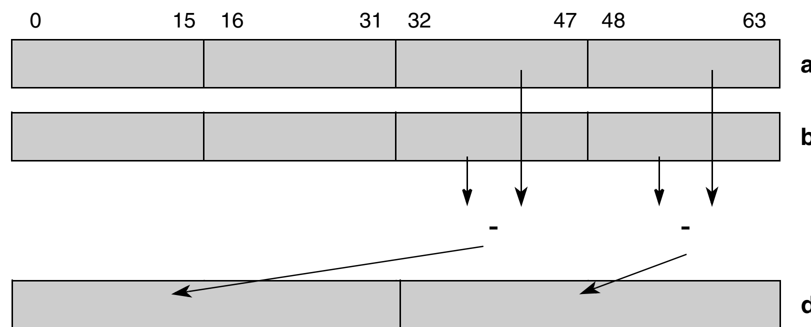


Figure 3-718. Vector Subtract from Halfwords Low Unsigned to Words (__ev_subfhlouw)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evsubfhlouw d,a,b**

__ev_subfhss __ev_subfhss

Vector Subtract from Half Words Signed and Saturate

d = __ev_subfhss (a,b)

```
// h0
temp0:31 ←EXTS(b0:15) - EXTS(a0:15)
ovh0 ←temp15 ⊕ temp16
d0:15 ←SATURATE(ovh0, temp15, 0x8000, 0x7fff, temp16:31)

// h1
temp0:31 ←EXTS(b16:31) - EXTS(a16:31)
ovh1 ←temp15 ⊕ temp16
d16:31 ←SATURATE(ovh1, temp15, 0x8000, 0x7fff, temp16:31)

// h2
temp0:31 ←EXTS(b32:47) - EXTS(a32:47)
ovh2 ←temp15 ⊕ temp16
d32:47 ←SATURATE(ovh2, temp15, 0x8000, 0x7fff, temp16:31)

// h3
temp0:31 ←EXTS(b48:63) - EXTS(a48:63)
ovh3 ←temp15 ⊕ temp16
d48:63 ←SATURATE(ovh3, temp15, 0x8000, 0x7fff, temp16:31)

ovh ←ovh0 | ovh1
ovl ←ovh2 | ovh3

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

The four signed half word elements of parameter **a** are subtracted from the corresponding four signed half word elements of parameter **b**, saturating if overflow or underflow occurs, and the results are placed in parameter **d**. Any overflow or underflow is recorded in the SPEFSCR overflow and summary overflow bits.

Other registers altered: SPEFSCR

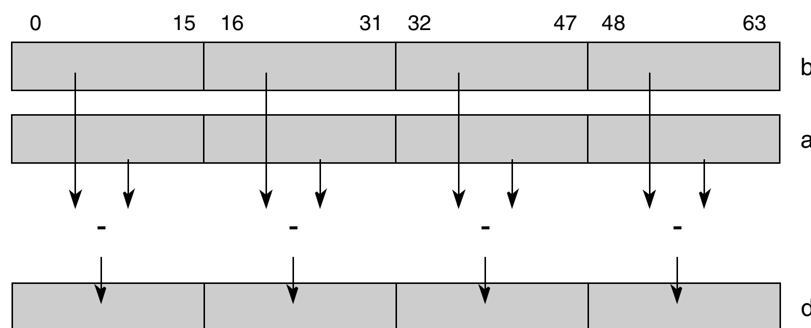


Figure 3-719. Vector Subtract from Half Words Signed and Saturate (__ev_subfhss)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evsubfhss d,a,b**

__ev_subfhus __ev_subfhus

Vector Subtract from Half Words Unsigned and Saturate

d = __ev_subfhus (a,b)

```
// h0
temp0:31 ←EXTZ(b0:15) - EXTZ(a0:15)
ovh0 ←temp15
d0:15 ←SATURATE(ovh0, temp15, 0x0000, 0x0000, temp16:31)

// h1
temp0:31 ←EXTZ(b16:31) - EXTZ(a16:31)
ovh1 ←temp15
d16:31 ←SATURATE(ovh1, temp15, 0x0000, 0x0000, temp16:31)

// h2
temp0:31 ←EXTZ(b32:47) - EXTZ(a32:47)
ovh2 ←temp15
d32:47 ←SATURATE(ovh2, temp15, 0x0000, 0x0000, temp16:31)

// h3
temp0:31 ←EXTZ(b48:63) - EXTZ(a48:63)
ovh3 ←temp15
d48:63 ←SATURATE(ovh3, temp15, 0x0000, 0x0000, temp16:31)

ovh ←ovh0 | ovh1
ovl ←ovh2 | ovh3

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

The four unsigned half word elements of parameter **a** are subtracted from four exchanged unsigned half word elements of parameter **b**, saturating if underflow occurs, and the results are placed in parameter **d**. Any underflow is recorded in the SPEFSCR overflow and summary overflow bits.

Other registers altered: SPEFSCR

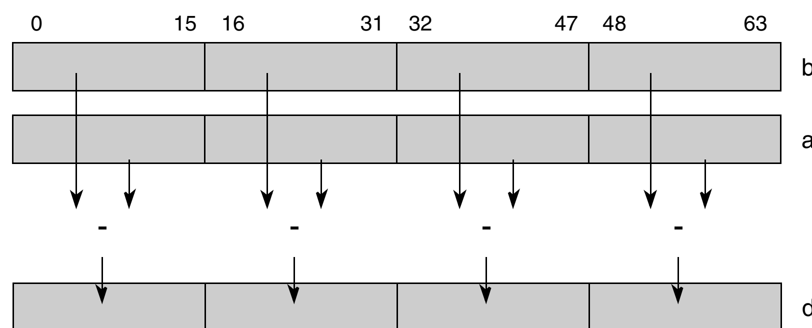


Figure 3-720. Vector Subtract from Half Words Unsigned and Saturate (__ev_subfhus)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evsubfhus d,a,b**

__ev_subfhx __ev_subfhx

Vector Subtract from Half Words Exchanged

d = __ev_subfhx (a,b)

```
// h0
d0:15 ←b0:15 - a16:31 // modulo difference

// h1
d16:31 ←b16:31 - a0:15 // modulo difference

// h2
d32:47 ←b32:47 - a48:63 // modulo difference

// h3
d48:63 ←b48:63 - a32:47 // modulo difference
```

The four exchanged half word elements of parameter **a** are subtracted from the four half word elements of parameter **b**. The results are placed into parameter **d**. The difference is a modulo difference.

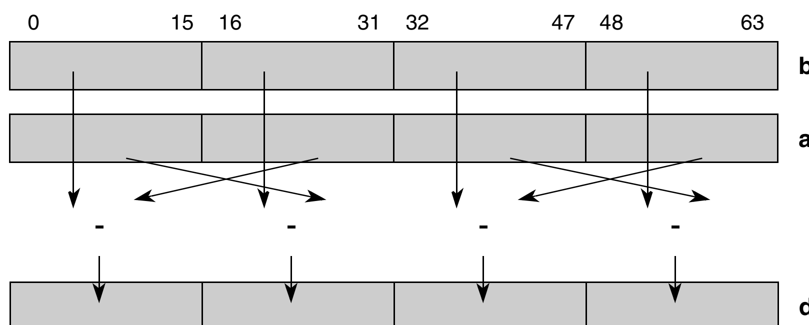


Figure 3-721. Vector Subtract from Half Words Exchanged (__ev_subfhx)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evsubfhx d,a,b**

__ev_subfhxss __ev_subfhxss

Vector Subtract from Half Words Exchanged, Signed and Saturate

d = __ev_subfhxss (a,b)

```
// h0
temp0:31 ←EXTS(b0:15) - EXTS(a16:31)
ovh0 ←temp15 ⊕ temp16
d0:15 ←SATURATE(ovh0, temp15, 0x8000, 0x7fff, temp16:31)

// h1
temp0:31 ←EXTS(b16:31) - EXTS(a0:15)
ovh1 ←temp15 ⊕ temp16
d16:31 ←SATURATE(ovh1, temp15, 0x8000, 0x7fff, temp16:31)

// h2
temp0:31 ←EXTS(b32:47) - EXTS(a48:63)
ovh2 ←temp15 ⊕ temp16
d32:47 ←SATURATE(ovh2, temp15, 0x8000, 0x7fff, temp16:31)

// h3
temp0:31 ←EXTS(b48:63) - EXTS(a32:47)
ovh3 ←temp15 ⊕ temp16
d48:63 ←SATURATE(ovh3, temp15, 0x8000, 0x7fff, temp16:31)

ovh ←ovh0 | ovh1
ovl ←ovh2 | ovh3

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

The four exchanged signed half word elements of parameter **a** are subtracted from four signed half word elements of parameter **b**, saturating if overflow or underflow occurs, and the results are placed in parameter **d**. Any overflow or underflow is recorded in the SPEFSCR overflow and summary overflow bits.

Other registers altered: SPEFSCR

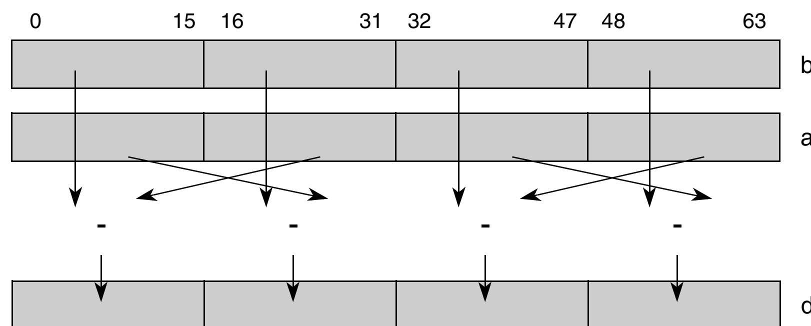


Figure 3-722. Vector Subtract from Half Words Exchanged Signed and Saturate (__ev_subfhxss)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evsubfhxss d,a,b**

__ev_subfhxus __ev_subfhxus

Vector Subtract from Half Words Exchanged, Unsigned and Saturate

d = __ev_subfhxus (a,b)

```
// h0
temp0:31 ←EXTZ(b0:15) - EXTZ(a16:31)
ovh0 ←temp15
d0:15 ←SATURATE(ovh0, temp15, 0x0000, 0x0000, temp16:31)

// h1
temp0:31 ←EXTZ(b16:31) - EXTZ(a0:15)
ovh1 ←temp15
d16:31 ←SATURATE(ovh1, temp15, 0x0000, 0x0000, temp16:31)

// h2
temp0:31 ←EXTZ(b32:47) - EXTZ(a48:63)
ovh2 ←temp15
d32:47 ←SATURATE(ovh2, temp15, 0x0000, 0x0000, temp16:31)

// h3
temp0:31 ←EXTZ(b48:63) - EXTZ(a32:47)
ovh3 ←temp15
d48:63 ←SATURATE(ovh3, temp15, 0x0000, 0x0000, temp16:31)

ovh ←ovh0 | ovh1
ovl ←ovh2 | ovh3

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

The four exchanged unsigned half word elements of parameter **a** are subtracted from four unsigned half word elements of parameter **b**, saturating if underflow occurs, and the results are placed in parameter **d**. Any underflow is recorded in the SPEFSCR overflow and summary overflow bits.

Other registers altered: SPEFSCR

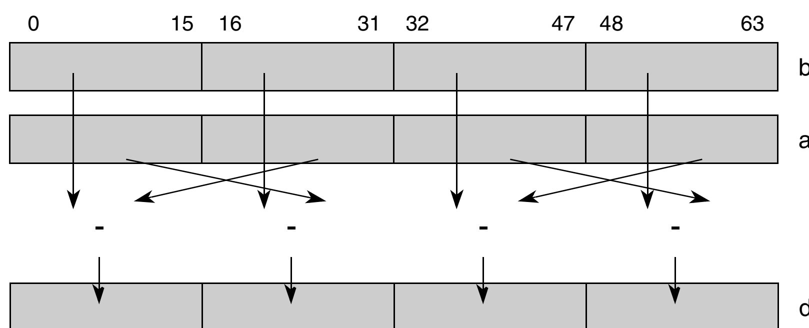


Figure 3-723. Vector Subtract from Exchanged Half Words Unsigned and Saturate (__ev_subfhxus)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evsubfhxus d,a,b**

__ev_subfsmiaa __ev_subfsmiaa

Vector Subtract from Accumulator to Accumulator

d = __ev_subfsmiaa (a)

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

```
// update accumulator
```
$ACC_{0:63} \leftarrow d_{0:63}$

The 64-bit value in parameter **a** is subtracted from the value in the accumulator and the difference is placed into parameter **d** and the accumulator.

This instruction can be used for signed or unsigned integers or fractions.

Other registers altered: ACC

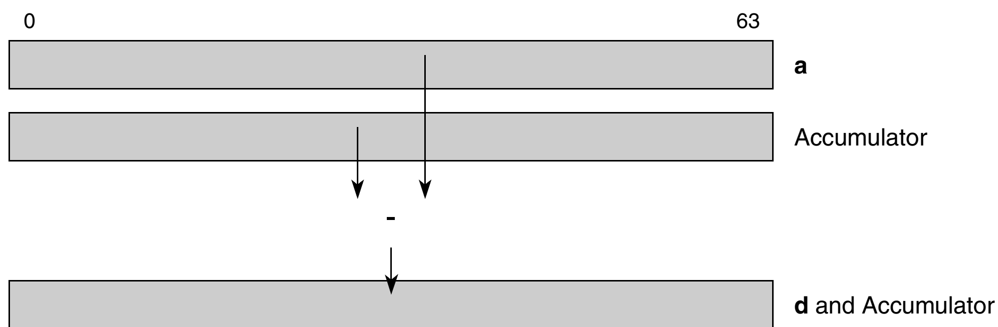


Figure 3-724. Vector Subtract from Accumulator to Accumulator (__ev_subfsmiaa)

d	a	Maps to
__ev64_opaque__	__ev64_opaque__	**evsubfsmiaa d,a**

__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

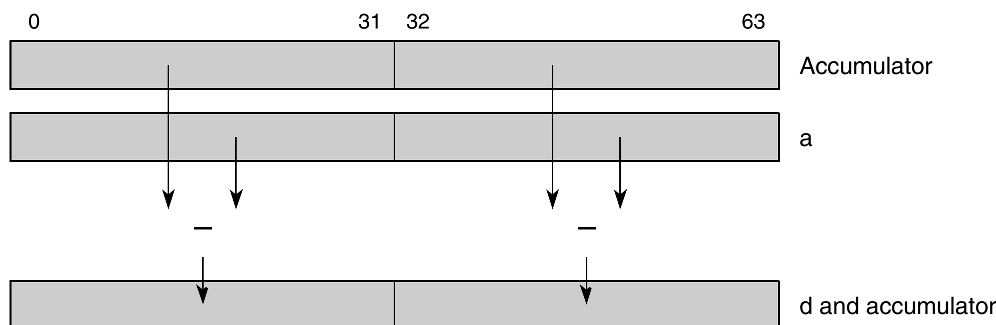


Figure 3-725. Vector Subtract Signed, Modulo, Integer to Accumulator Word (__ev_subfsmiaaw)

d	a	Maps to
__ev64_opaque__	__ev64_opaque__	**evsubfsmiaaw d,a**

__ev_subfssiaa __ev_subfssiaa

Vector Subtract from Signed, Saturate, Integer to Accumulator

d = __ev_subfssiaa (a)

```
temp0:64 ←EXTS(ACC0:63) - EXTS(a0:63)
ov ←temp0 ⊕ temp1
d0:63 ←SATURATE(ov, temp0, 0x8000_0000_0000_0000, 0x7fff_ffff_ffff_ffff, temp1:64)

// update accumulator
ACC0:63 ←d0:63

SPEFSCROVH ←0
SPEFSCROV ←ov
SPEFSCRSOV ←SPEFSCRSOV | ov
```

The signed 64-bit value in parameter **a** is subtracted from the value in the accumulator saturating if overflow or underflow 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

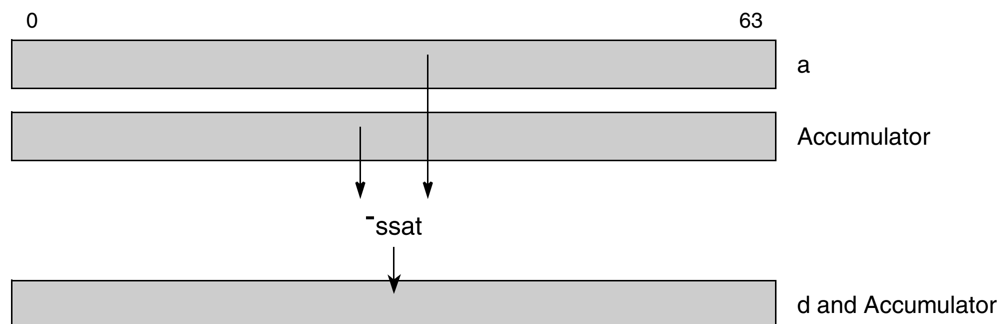


Figure 3-726. Vector Subtract from Signed, Saturate, Integer to Accumulator (__ev_subfssiaa)

d	a	Maps to
__ev64_opaque__	__ev64_opaque__	**evsubfssiaa 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 or underflow 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

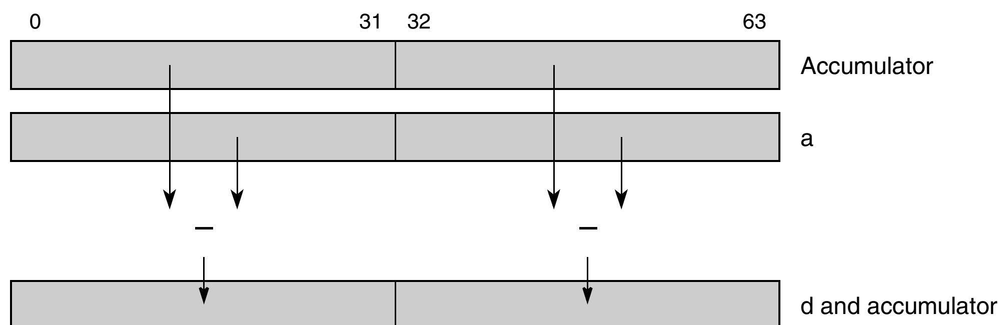


Figure 3-727. Vector Subtract Signed, Saturate, Integer to Accumulator Word (__ev_subfssiaaw)

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

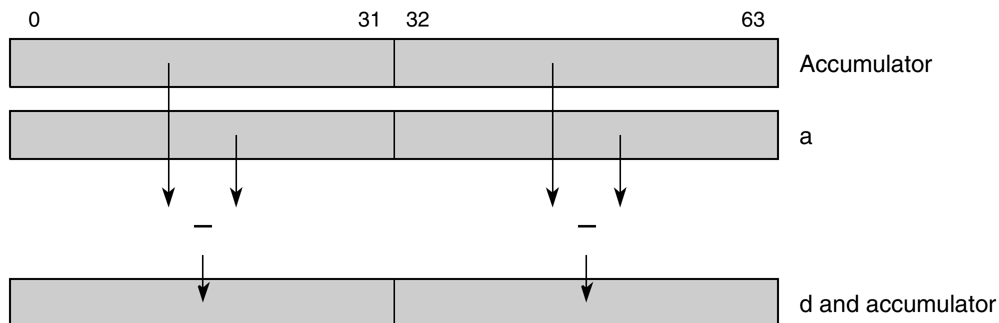


Figure 3-728. Vector Subtract Unsigned, Modulo, Integer to Accumulator Word (__ev_subfumiaaw)

d	a	Maps to
__ev64_opaque__	__ev64_opaque__	**evsubfumiaaw d,a**

__ev_subfusiaa __ev_subfusiaa

Vector Subtract from (Accumulator) Unsigned, Saturate, Integer to Accumulator

d = __ev_subfusiaa (a)

```
temp0:64 ←EXTZ(ACC0:63) - EXTZ(a0:63)
ov ←temp0
d0:63 ←SATURATE(ov, temp0, 0xffff_ffff_ffff_ffff, 0xffff_ffff_ffff_ffff, temp1:64)

// update accumulator
ACC0:63 ←d0:63

SPEFSCROVH ←0
SPEFSCROV ←ov
SPEFSCRSOV ←SPEFSCRSOV | ov
```

The unsigned 64-bit value in parameter **a** is subtracted from the value 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

0
63
a
Accumulator
-usat
d and Accumulator

Figure 3-729. Vector Subtract from Unsigned, Saturate, Integer to Accumulator (__ev_subfusiaa)

d	a	Maps to
__ev64_opaque__	__ev64_opaque__	**evsubfusiaa 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 ←EXTZ(ACC32:63) - EXTZ(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

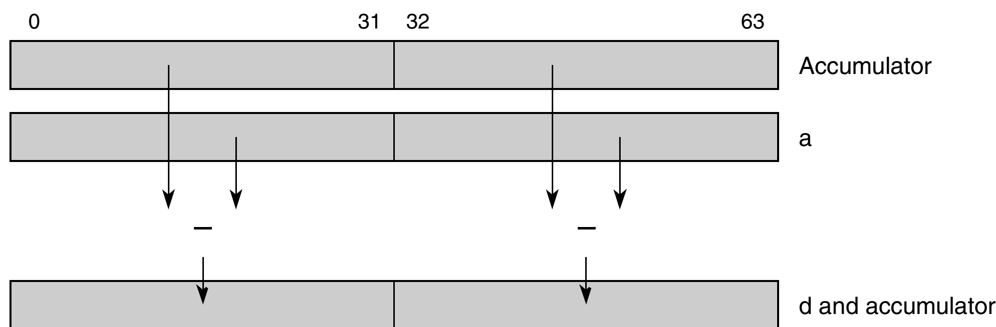


Figure 3-730. Vector Subtract Unsigned, Saturate, Integer to Accumulator Word (__ev_subfusiaaw)

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**.

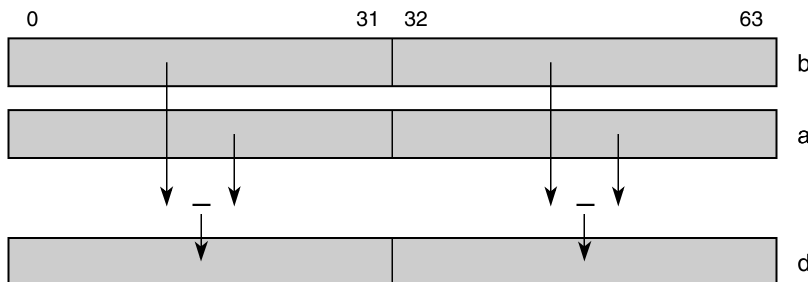


Figure 3-731. Vector Subtract from Word (__ev_subfw)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evsubfw d,a,b**

__ev_subfwegsf __ev_subfwegsf

Vector Subtract from Word Even Guarded Signed Fraction

d = __ev_subfwegsf (a,b)

$$d_{0:63} \leftarrow (EXTS_{48}(b_{0:31}) \ || \ {}^{16}0) - (EXTS_{48}(a_{0:31}) \ || \ {}^{16}0)$$

The even word elements of parameters **a** and **b** are sign-extended with 16 guard bits and padded with 16 0’s, and then the parameter **a** value is subtracted from the parameter **b** value to produce a 64-bit difference, and the result is placed into parameter **d**.

Note: __ev_subfwegsf is used to subtract 1.31 fractions to produce a 17.47 fractional difference.

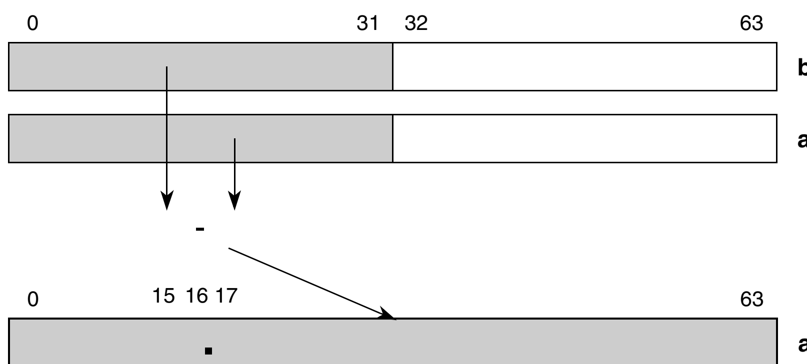


Figure 3-732. Vector Subtract from Word Even Guarded Signed Fraction (__ev_subfwegsf)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evsubfwegsf d,a,b**

__ev_subfwegsi __ev_subfwegsi

Vector Subtract from Word Even Guarded Signed Integer

d = __ev_subfwegsi (a,b)

$$d_{0:63} \leftarrow EXTS_{64}(b_{0:31}) - EXTS_{64}(a_{0:31})$$

The even word element of parameter **a** is sign-extended to 64 bits and subtracted from the sign-extended even word element of parameter **b** to produce a 64-bit difference, and the result is placed into parameter **d**.

Note: __ev_subfwegsi can also be used to subtract 1.31 fractions to produce a 33.31 fractional difference.

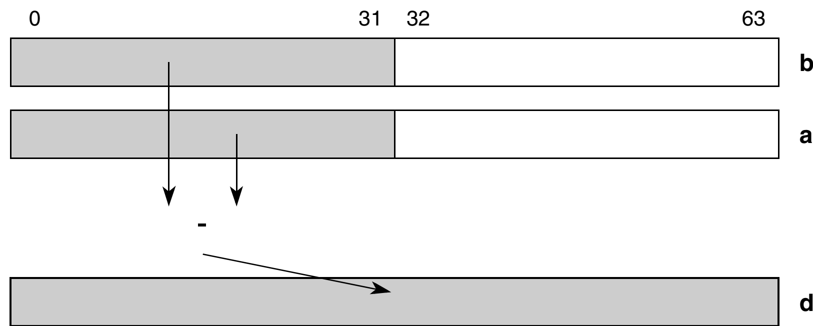


Figure 3-733. Vector Subtract from Word Even Guarded Signed Integer (__ev_subfwegsi)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evsubfwegsi d,a,b**

__ev_subfwogsf __ev_subfwogsf

Vector Subtract from Word Odd Guarded Signed Fraction

d = __ev_subfwogsf (a,b)

$d_{0:63} \leftarrow (EXTS_{48}(b_{32:63}) \,||\, {}^{16}0) - (EXTS_{48}(a_{32:63}) \,||\, {}^{16}0)$

The odd word elements of parameters **a** and **b** are sign-extended with 16 guard bits and padded with 16 0's, and then the parameter **a** value is subtracted from the parameter **b** value to produce a 64-bit difference, and the result is placed into parameter **d**.

Note: __ev_subfwogsf is used to subtract 1.31 fractions to produce a 17.47 fractional difference.

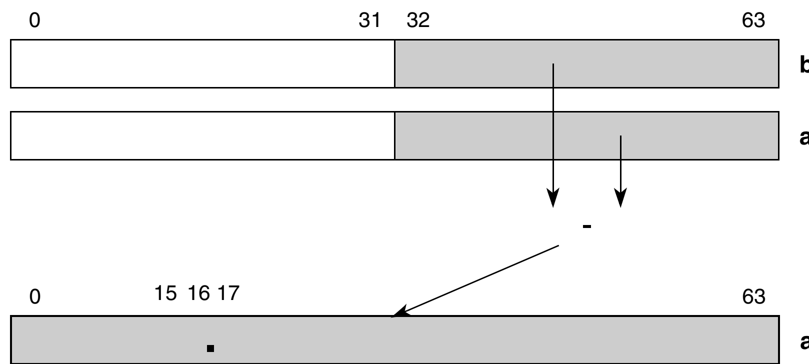


Figure 3-734. Vector Subtract from Word Odd Guarded Signed Fraction (__ev_subfwogsf)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evsubfwogsf d,a,b**

__ev_subfwogsi

Vector Subtract from Word Odd Guarded Signed Integer

d = __ev_subfwogsi (a,b)

$$d_{0:63} \leftarrow EXTS_{64}(b_{32:63}) - EXTS_{64}(a_{32:63})$$

The odd word element of parameter **a** is sign-extended to 64 bits and subtracted from the sign-extended odd word element of parameter **b** to produce a 64-bit difference, and the result is placed into parameter **d**.

Note: __ev_subfwogsi can also be used to subtract 1.31 fractions to produce a 33.31 fractional difference.

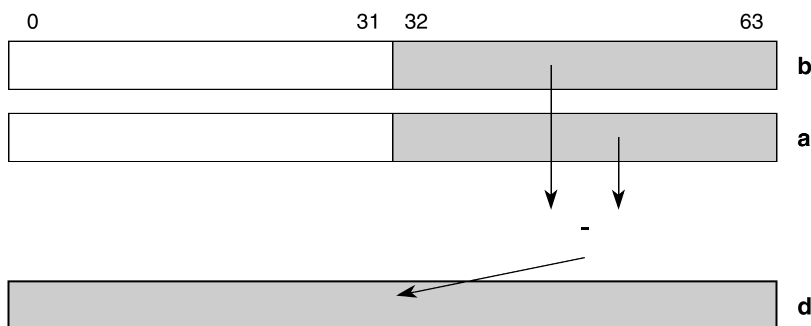


Figure 3-735. Vector Subtract from Word Odd Guarded Signed Integer (__ev_subfwogsi)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**__ev_subfwogsi d,a,b**

__ev_subfwss   __ev_subfwss

Vector Subtract from Word

d = __ev_subfwss (a,b)

```
// h0
temp0:32 ←EXTS(b0:31) - EXTS(a0:31)
ovh ←temp0 ⊕ temp1
d0:31 ←SATURATE(ovh, temp0, 0x8000_0000, 0x7fff_ffff, temp1:32)

// h1
temp0:32 ←EXTS(b32:63) - EXTS(a32:63)
ovl ←temp0 ⊕ temp1
d32:63 ←SATURATE(ovl, temp0, 0x8000_0000, 0x7fff_ffff, temp1:32)

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

The high word element of parameter **a** is subtracted from the high word element of parameter **b**, saturating if overflow or underflow occurs, the low word element of parameter **a** is subtracted from the low word element of parameter **b**, saturating if overflow or underflow occurs, and the results are placed in parameter **d**. Any overflow or underflow is recorded in the SPEFSCR overflow and summary overflow bits.

Other registers altered: SPEFSCR

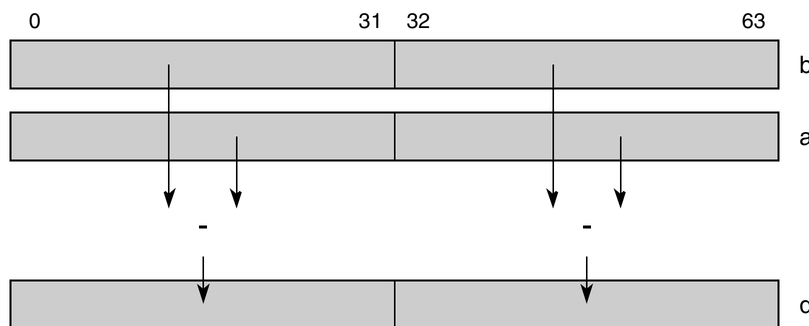


Figure 3-736. Vector Subtract from Word Signed and Saturate (__ev_subfwss)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evsubfwss d,a,b**

__ev_subfwus __ev_subfwus

Vector Subtract from Word Unsigned and Saturate

d = __ev_subfwus (a,b)

```
// h0
temp0:32 ←EXTZ(b0:31) - EXTZ(a0:31)
ovh ←temp0
d0:31 ←SATURATE(ovh, temp0, 0x0000_0000, 0x0000_0000, temp1:32)

// h1
temp0:32 ←EXTZ(b32:63) - EXTZ(a32:63)
ovl ←temp0
d32:63 ←SATURATE(ovl, temp0, x0000_0000, 0x0000_0000, temp1:32)

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

The high word element of parameter **a** is subtracted from the high word element of parameter **b**, saturating if underflow occurs, the low word element of parameter **a** is subtracted from the low word element of parameter **b**, saturating if underflow occurs, and the results are placed in parameter **d**. Any underflow is recorded in the SPEFSCR overflow and summary overflow bits.

Other registers altered: SPEFSCR

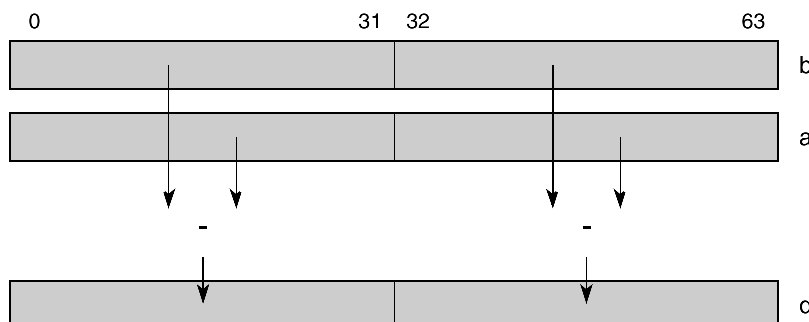


Figure 3-737. Vector Subtract from Word Unsigned and Saturate (__ev_subfwus)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evsubfwus d,a,b**

__ev_subfwx __ev_subfwx

Vector Subtract from Word Exchanged

d = __ev_subfwx (a,b)

```
d0:31 ←b0:31 - a32:63                                  // Modulo difference
d32:63 ←b32:63 - a0:31                                 // Modulo difference
```

The exchanged signed integer word elements of parameter **a** are subtracted from the corresponding elements of parameter **b** and the results are placed into parameter **d**.

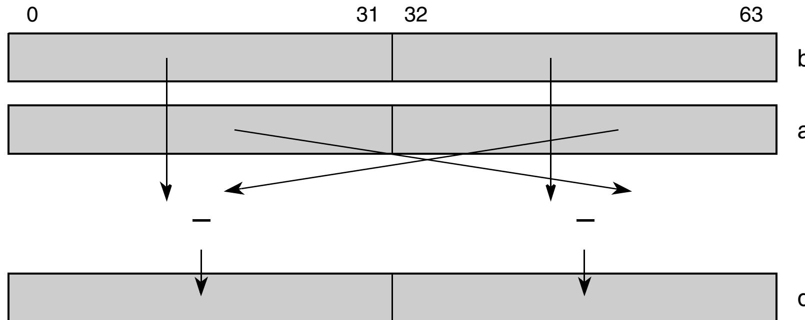


Figure 3-738. Vector Subtract from Word Exchanged (__ev_subfwx)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evsubfwx d,a,b**

__ev_subfwxss __ev_subfwxss

Vector Subtract from Word Exchanged Signed and Saturate

d = __ev_subfwxss (a,b)

```
// h0
temp0:32 ←EXTS(b0:31) - EXTS(a32:63)
ovh ←temp0 ⊕ temp1
d0:15 ←SATURATE(ovh, temp0, 0x8000_0000, 0x7fff_ffff, temp1:32)

// h1
temp0:31 ←EXTS(b32:63) - EXTS(a0:31)
ovl ←temp0 ⊕ temp1
d16:31 ←SATURATE(ovl, temp0, 0x8000_0000, 0x7fff_ffff, temp1:32)

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

The low word element of parameter **a** is subtracted from the high word element of parameter **b**, saturating if overflow or underflow occurs, the high word element of parameter **a** is subtracted from the low word element of parameter **b**, saturating if overflow or underflow occurs, and the results are placed in parameter **d**. Any overflow or underflow is recorded in the SPEFSCR overflow and summary overflow bits.

Other registers altered: SPEFSCR

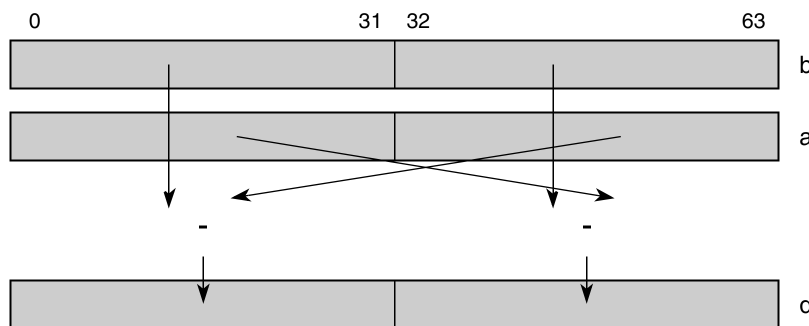


Figure 3-739. Vector Subtract from Word Exchanged Signed and Saturate (__ev_subfwxss)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evsubfwxss d,a,b**

__ev_subfwxus __ev_subfwxus

Vector Subtract from Word Exchanged Unsigned and Saturate

d = __ev_subfwxus (a,b)

```
// h0
temp0:32 ←EXTZ(b0:31) - EXTZ(a32:63)
ovh ←temp0
d0:15 ←SATURATE(ovh, temp0, 0x0000_0000, 0x0000_0000, temp1:32)

// h1
temp0:31 ←EXTZ(b32:63) - EXTZ(a0:31)
ovl ←temp0
d16:31 ←SATURATE(ovl, temp0, x0000_0000, 0x0000_0000, temp1:32)

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

The high word element of parameter **a** is subtracted from the high word element of parameter **b**, saturating if underflow occurs, the low word element of parameter **a** is subtracted from the low word element of parameter **b**, saturating if underflow occurs, and the results are placed in parameter **d**. Any underflow is recorded in the SPEFSCR overflow and summary overflow bits.

Other registers altered: SPEFSCR

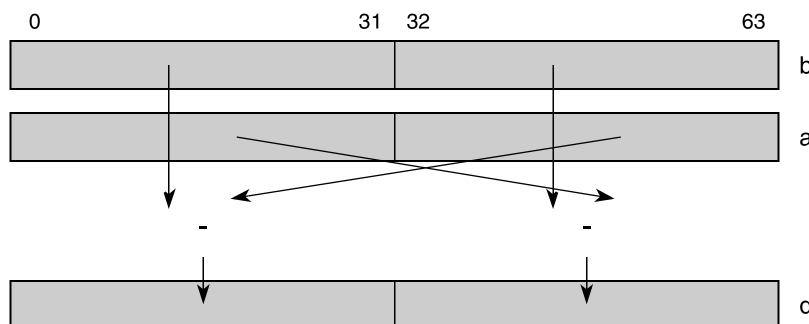


Figure 3-740. Vector Subtract from Word Exchanged Unsigned and Saturate (__ev_subfwxus)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evsubfwxus d,a,b**

__ev_subifb __ev_subifb

Vector Subtract Immediate from Byte

d = __ev_subifb (a,b)

```
UIMM ← a

d0:7 ←b0:7 - EXTZ(UIMM) // Modulo
d8:15 ←b8:15 - EXTZ(UIMM) // Modulo
d16:23 ←b16:23 - EXTZ(UIMM) // Modulo
d4:31 ←b24:31 - EXTZ(UIMM) // Modulo
d32:39 ←b32:39 - EXTZ(UIMM) // Modulo
d40:47 ←b40:47 - EXTZ(UIMM) // Modulo
d48:55 ←b48:55 - EXTZ(UIMM) // Modulo
d56:63 ←b56:63 - EXTZ(UIMM) // Modulo
```

The 5-bit unsigned literal provided by parameter **a** (UIMM) is zero-extended and subtracted from the eight byte elements of parameter **b** and the results are placed in parameter **d**. Note that the same value is subtracted from all elements of parameter **b**. UIMM is 5 bits.

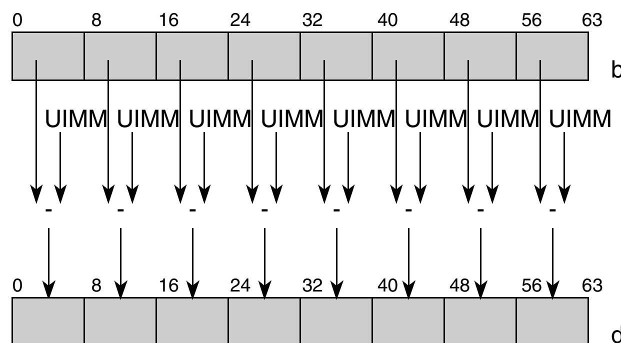


Figure 3-741. Vector Subtract Immediate from Byte (__ev_subifb)

d	a	b	Maps to
__ev64_opaque__	5-bit unsigned literal	__ev64_opaque__	**evsubifb d,a,b**

__ev_subifh __ev_subifh

Vector Subtract Immediate from Half Word

d = __ev_subifh (a,b)

```
UIMM ← a

d0:15 ←b0:15 - EXTZ(UIMM)// Modulo difference
d16:31 ←b16:31 - EXTZ(UIMM)// Modulo difference
d32:47 ←b32:47 - EXTZ(UIMM)// Modulo difference
d48:63 ←b48:63 - EXTZ(UIMM)// Modulo difference
```

The 5-bit unsigned literal provided by parameter **a** (UIMM) is zero-extended and subtracted from the four half word elements of parameter **b** and the results are placed in parameter **d**. Note that the same value is subtracted from all elements of parameter **b**. UIMM is 5 bits.

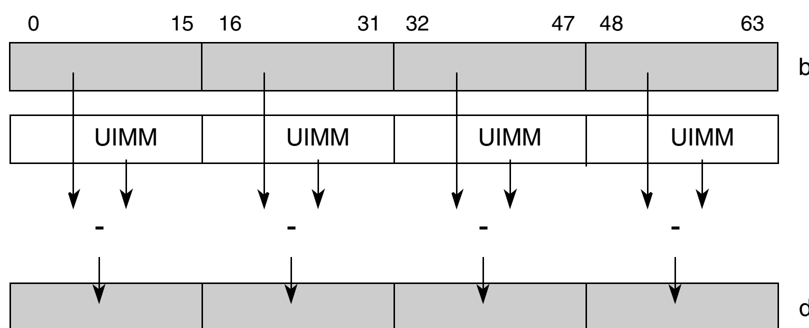


Figure 3-742. Vector Subtract Immediate from Half Word (__ev_subifh)

d	a	b	Maps to
__ev64_opaque__	5-bit unsigned literal	__ev64_opaque__	**evsubifh d,a,b**

__ev_subifw

__ev_subifw

Vector Subtract Immediate from Word

d = __ev_subifw (a,b)

```
UIMM ← a

d0:31 ←b0:31 - EXTZ(UIMM) // Modulo difference
d32:63 ←b32:63 - EXTZ(UIMM)// Modulo difference
```

The 5-bit unsigned literal provided by parameter **a** (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 parameter **b**. UIMM is 5 bits.

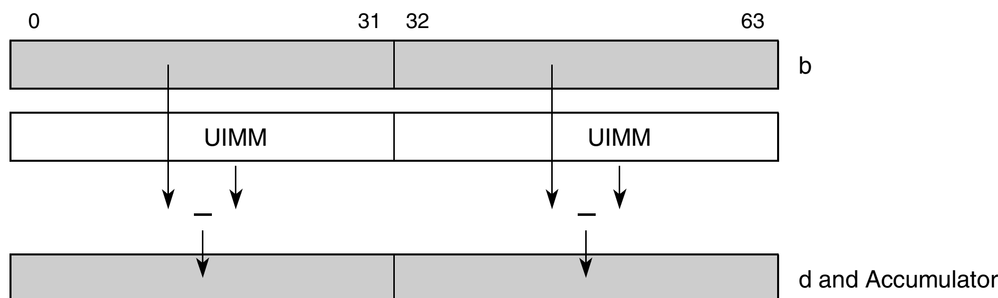


Figure 3-743. Vector Subtract Immediate from Word (__ev_subifw)

d	a	b	Maps to
__ev64_opaque__	5-bit unsigned literal	__ev64_opaque__	**evsubifw d,a,b**

__ev_sum2his[a] __ev_sum2his[a]

Vector Sum of 2 Half Words Interleaved Signed (to Accumulator)

d = __ev_sum2his (a) (A = 0)

d = __ev_sum2hisa (a) (A = 1)

```
d0:31 ←EXTS(a0:15) + EXTS(a32:47)
d32:63 ←EXTS(a16:31) + EXTS(a48:63)

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

Groups of two interleaved signed half word elements of parameter **a** are summed together, and the results are placed into the word elements of parameter **d**. If A = 1, the result in parameter **d** is also placed into the accumulator.

Other registers altered: ACC (if A=1)

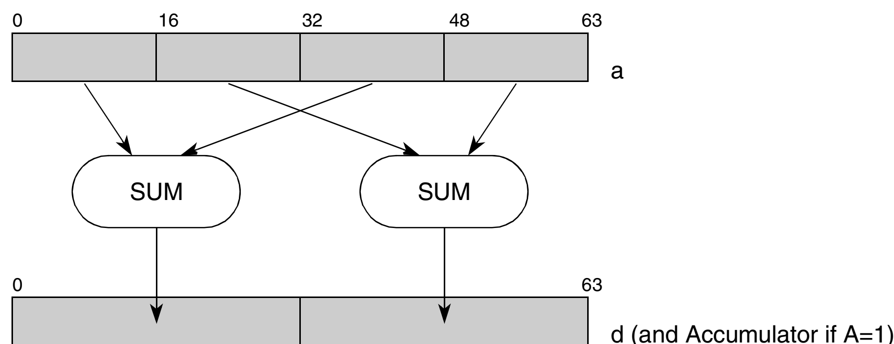


Figure 3-744. Vector Sum of 2 Half Words Interleaved Signed (to Accumulator) (__ev_sum2his[a])

A	d	a	Maps to
A = 0	__ev64_opaque__	__ev64_opaque__	**evsum2his d,a**
A = 1	__ev64_opaque__	__ev64_opaque__	**evsum2hisa d,a**

__ev_sum2hisaaw __ev_sum2hisaaw

Vector Sum of 2 Half Words Interleaved Signed and Accumulate into Words

d = __ev_sum2hisaaw (a)

$d_{0:31} \leftarrow ACC_{0:31} + EXTS(a_{0:15}) + EXTS(a_{32:47})$
$d_{32:63} \leftarrow ACC_{32:63} + EXTS(a_{16:31}) + EXTS(a_{48:63})$

// update accumulator
$ACC_{0:63} \leftarrow d_{0:63}$

Groups of two interleaved signed half word elements of parameter **a** are summed together, then added to the corresponding word elements in the accumulator, and the results are placed into the word elements of parameter **d** and the accumulator.

Other registers altered: ACC

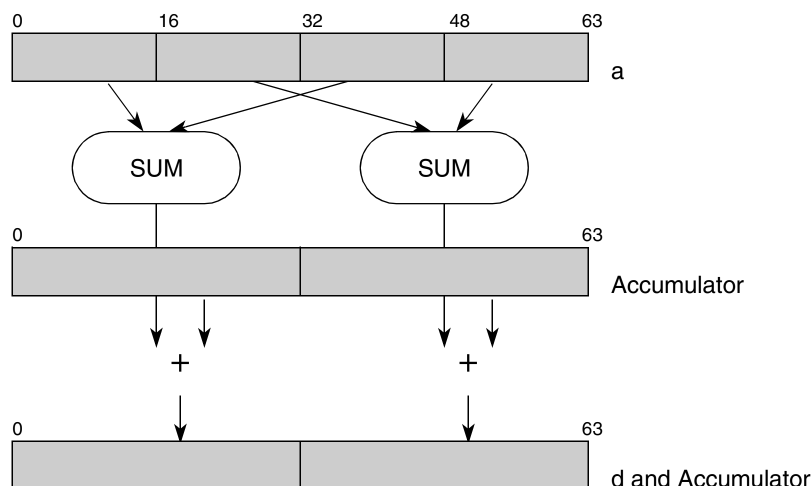


Figure 3-745. Vector Sum of 2 Half Words Interleaved Signed and Accumulate (__ev_sum2hisaaw)

d	a	Maps to
__ev64_opaque__	__ev64_opaque__	**evsum2hisaaw d,a**

__ev_sum2hs[a] __ev_sum2hs[a]

Vector Sum of 2 Half Words Signed (to Accumulator)

d = __ev_sum2hs (a) (A = 0)

d = __ev_sum2hsa (a) (A = 1)

```
d0:31 ←EXTS(a0:15) + EXTS(a16:315)
d32:63 ←EXTS(a32:47) + EXTS(a48:63)

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

Groups of two signed half word elements of parameter **a** are summed together, and the results are placed into the word elements of parameter **d**. If A = 1, the result in parameter **d** is also placed into the accumulator.

Other registers altered: ACC (if A=1)

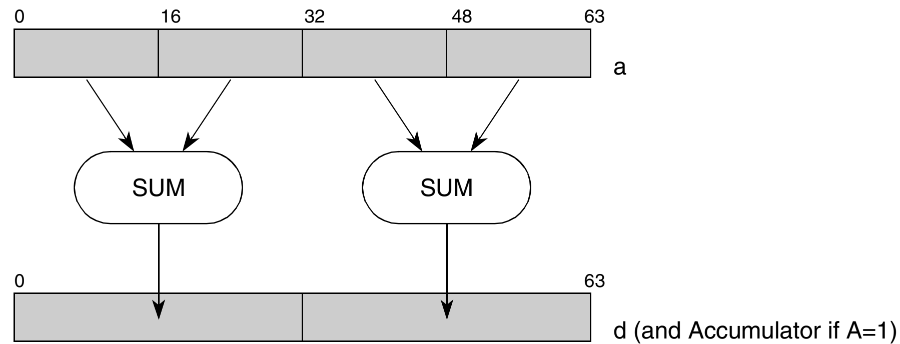


Figure 3-746. Vector Sum of 2 Half Words Signed (to Accumulator) (__ev_sum2hs[a])

A	d	a	Maps to
A = 0	__ev64_opaque__	__ev64_opaque__	**evsum2hs d,a**
A = 1	__ev64_opaque__	__ev64_opaque__	**evsum2hsa d,a**

__ev_sum2hsaaw __ev_sum2hsaaw

Vector Sum of 2 Half Words Signed and Accumulate into Words

d = __ev_sum2hsaaw (a)

```
d0:31 ←ACC0:31 +EXTS(a0:15) + EXTS(a16:315)
d32:63 ←ACC32:63 + EXTS(a32:47) + EXTS(a48:63)

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

Groups of two signed half word elements of parameter **a** are summed together, then added to the corresponding word elements in the accumulator, and the results are placed into the word elements of parameter **d** and the accumulator.

Other registers altered: ACC

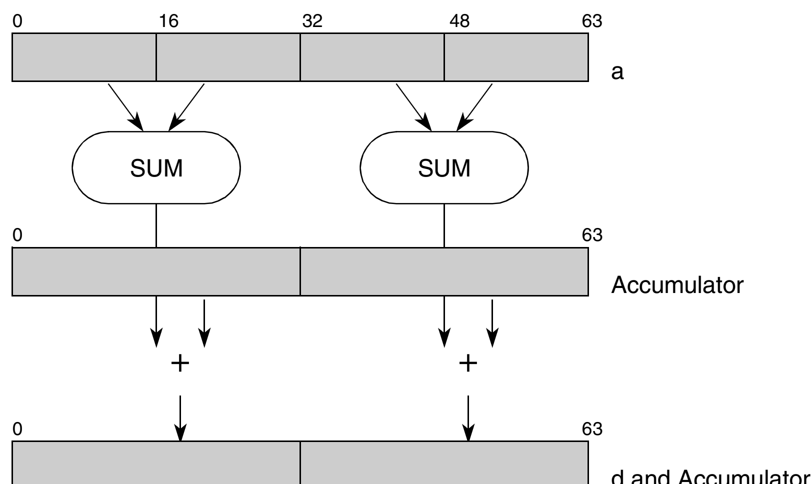


Figure 3-747. Vector Sum of 2 Half Words Signed and Accumulate (__ev_sum2hsaaw)

d	a	Maps to
__ev64_opaque__	__ev64_opaque__	**evsum2hsaaw d,a**

__ev_sum2hu[a] __ev_sum2hu[a]

Vector Sum of 2 Half Words Unsigned (to Accumulator)

d = __ev_sum2hu (a) (A = 0)

d = __ev_sum2hua (a) (A = 1)

```
d0:31 ←EXTZ(a0:15) + EXTZ(a16:315)
d32:63 ←EXTZ(a32:47) + EXTZ(a48:63)

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

Groups of two unsigned half word elements of parameter **a** are summed together, and the results are placed into the word elements of parameter **d**. If A = 1, the result in parameter **d** is also placed into the accumulator.

Other registers altered: ACC (if A=1)

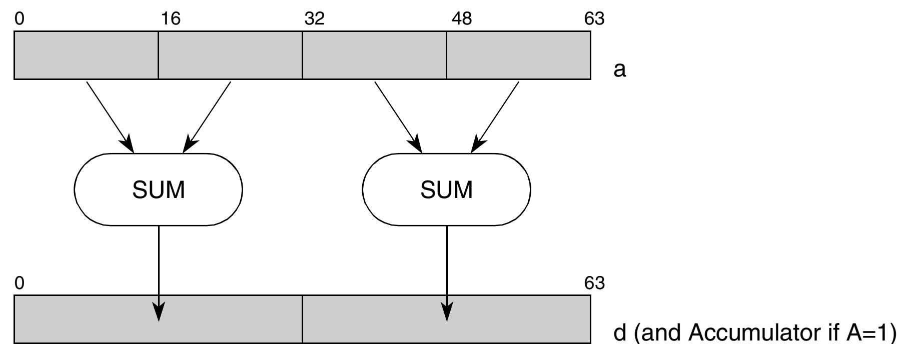


Figure 3-748. Vector Sum of 2 Half Words Unsigned (to Accumulator) (__ev_sum2hu[a])

A	d	a	Maps to
A = 0	__ev64_opaque__	__ev64_opaque__	**evsum2hu d,a**
A = 1	__ev64_opaque__	__ev64_opaque__	**evsum2hua d,a**

__ev_sum2huaaw   __ev_sum2huaaw

Vector Sum of 2 Half Words Unsigned and Accumulate into Words

d = __ev_sum2huaaw (a)

```
d0:31 ←ACC0:31 +EXTZ(a0:15) + EXTZ(a16:315)
d32:63 ←ACC32:63 + EXTZ(a32:47) + EXTZ(a48:63)

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

Groups of two unsigned half word elements of parameter **a** are summed together, then added to the corresponding word elements in the accumulator, and the results are placed into the word elements of parameter **d** and the accumulator.

Other registers altered: ACC

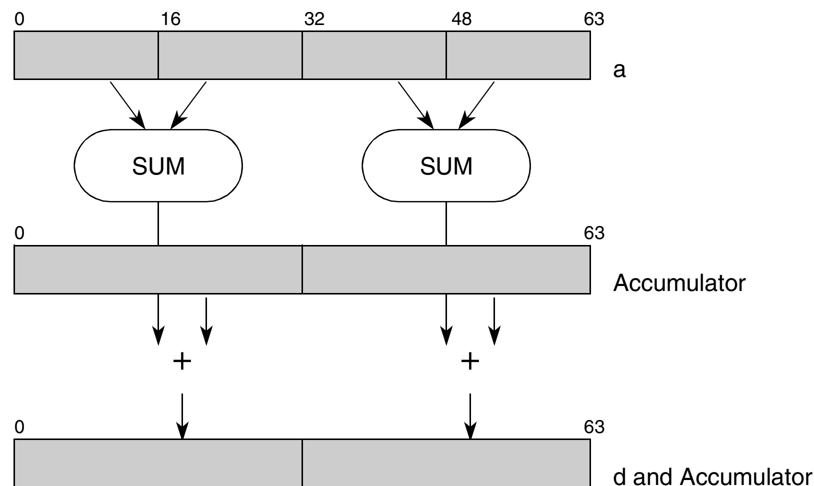


Figure 3-749. Vector Sum of 2 Half Words Signed and Accumulate (__ev_sum2huaaw)

d	a	Maps to
__ev64_opaque__	__ev64_opaque__	**evsum2huaaw d,a**

__ev_sum4bs[a] __ev_sum4bs[a]

Vector Sum of 4 Bytes SIgned (to Accumulator)

d = __ev_sum4bs (a) (A = 0)

d = __ev_sum4bsa (a) (A = 1)

```
d0:31 ←EXTS(a0:7) + EXTS(a8:15) + EXTS(a16:23) + EXTS(a24:31)
d32:63 ←EXTS(a32:39)+ EXTS(a40:47) + EXTS(a48:55) + EXTS(a56:63)

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

Groups of four signed byte elements of parameter **a** are summed together and the results are placed into the word elements of parameter **d**. If A = 1, the result in parameter **d** is also placed into the accumulator.

Other registers altered: ACC (if A=1)

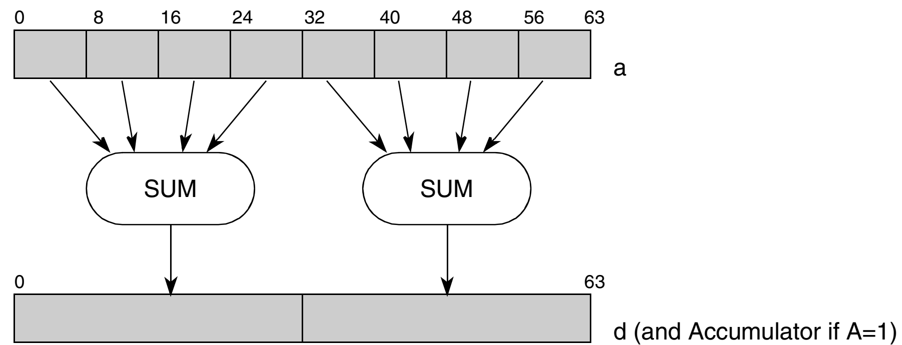


Figure 3-750. Vector Sum of 4 Bytes Signed (to Accumulator) (__ev_sum4bs[a])

A	d	a	Maps to
A = 0	__ev64_opaque__	__ev64_opaque__	**evsum4bs d,a**
A = 1	__ev64_opaque__	__ev64_opaque__	**evsum4bsa d,a**

__ev_sum4bsaaw __ev_sum4bsaaw

Vector Sum of 4 Bytes Signed and Accumulate into Words

d = __ev_sum4bsaaw (a)

```
d0:31 ←ACC0:31 + EXTS(a0:7) + EXTS(a8:15) + EXTS(a16:23) + EXTS(a24:31)
d32:63 ←ACC32:63 + EXTS(a32:39)+ EXTS(a40:47) + EXTS(a48:55) + EXTS(a56:63)
// update accumulator
ACC0:63 ←d0:63
```

Groups of four signed byte elements of parameter **a** are summed together, then added to the respective word elements of the accumulator, and the result is placed in parameter **d** and the accumulator.

Other registers altered: ACC

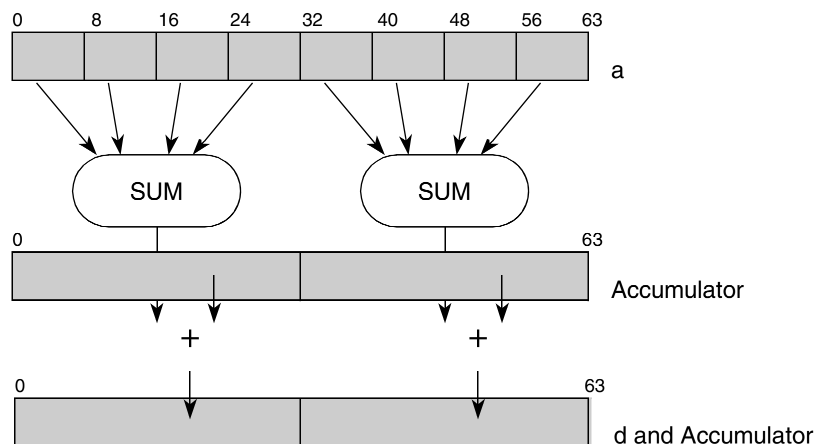


Figure 3-751. Vector Sum of 4 Bytes Signed and Accumulate into Words (__ev_sum4bsaaw)

d	a	Maps to
__ev64_opaque__	__ev64_opaque__	**evsum4bsaaw d,a**

__ev_sum4bu[a] __ev_sum4bu[a]

Vector Sum of 4 Bytes Unsigned (to Accumulator)

d = __ev_sum4bu (a) (A = 0)

d = __ev_sum4bua (a) (A = 1)

```
d0:31 ←EXTZ(a0:7) + EXTZ(a8:15) + EXTZ(a16:23) + EXTZ(a24:31)
d32:63 ←EXTZ(a32:39)+ EXTZ(a40:47) + EXTZ(a48:55) + EXTZ(a56:63)

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

Groups of four unsigned byte elements of parameter **a** are summed together and the results are placed into the word elements of parameter **d**. If A = 1, the result in parameter **d** is also placed into the accumulator.

Other registers altered: ACC (if A=1)

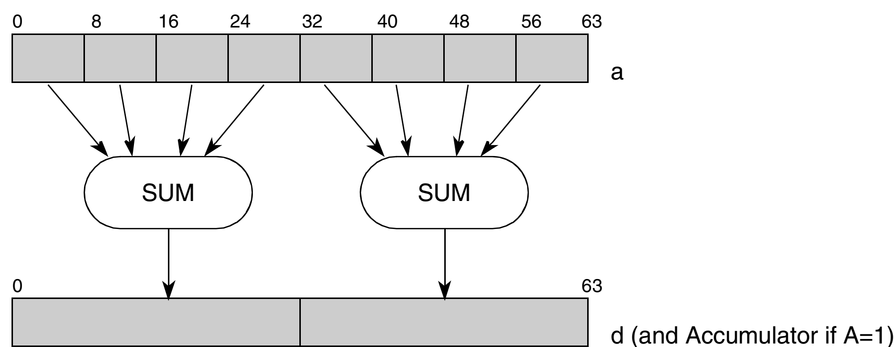


Figure 3-752. Vector Sum of 4 Bytes Unsigned (to Accumulator) (__ev_sum4bs[a])

A	d	a	Maps to
A = 0	__ev64_opaque__	__ev64_opaque__	**evsum4bu d,a**
A = 1	__ev64_opaque__	__ev64_opaque__	**evsum4bua d,a**

__ev_sum4buaaw __ev_sum4buaaw

Vector Sum of 4 Bytes Unsigned and Accumulate into Words

d = __ev_sum4buaaw (a)

$d_{0:31} \leftarrow ACC_{0:31} + EXTZ(a_{0:7}) + EXTZ(a_{8:15}) + EXTZ(a_{16:23}) + EXTZ(a_{24:31})$
$d_{32:63} \leftarrow ACC_{32:63} + EXTZ(a_{32:39}) + EXTZ(a_{40:47}) + EXTZ(a_{48:55}) + EXTZ(a_{56:63})$
// update accumulator
$ACC_{0:63} \leftarrow d_{0:63}$

Groups of four unsigned byte elements of parameter **a** are summed together, then added to the respective word elements of the accumulator, and the result is placed in parameter **d** and the accumulator.

Other registers altered: ACC

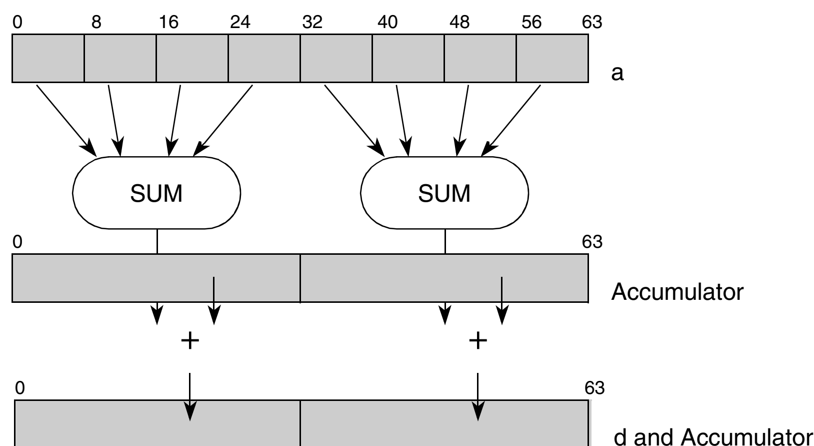


Figure 3-753. Vector Sum of 4 Bytes Unsigned and Accumulate into Words (__ev_sum4buaaw)

d	a	Maps to
__ev64_opaque__	__ev64_opaque__	**evsum4buaaw d,a**

__ev_sumws[a] __ev_sumws[a]

Vector Sum of Words Signed (to Accumulator)

d = __ev_sumws (a) (A = 0)

d = __ev_sumwsa (a) (A = 1)

```
d0:63 ←EXTS(a0:31) + EXTS(a32:63)

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

The signed word elements of parameter **a** are summed together, and the result is placed in parameter **d**. If A = 1, the result in parameter **d** is also placed into the accumulator.

Other registers altered: ACC (if A=1)

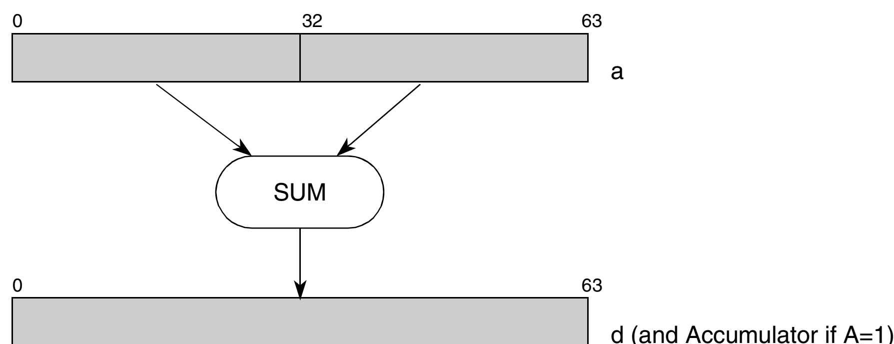


Figure 3-754. Vector Sum of Words Signed (to Accumulator) (__ev_sumws)

A	d	a	Maps to
A = 0	__ev64_opaque__	__ev64_opaque__	**evsumws d,a**
A = 1	__ev64_opaque__	__ev64_opaque__	**evsumwsa d,a**

__ev_sumwsaa __ev_sumwsaa

Vector Sum of Words Signed and Accumulate

d = __ev_sumwsaa (a)

$d_{0:63} \leftarrow \text{EXTS}(a_{0:31}) + \text{EXTS}(a_{32:63}) + ACC_{0:63}$

```
// update accumulator
```
$ACC_{0:63} \leftarrow d_{0:63}$

The signed word elements of parameter **a** are summed together then added to the contents of the accumulator, and the result is placed in parameter **d** and the accumulator.

Other registers altered: ACC

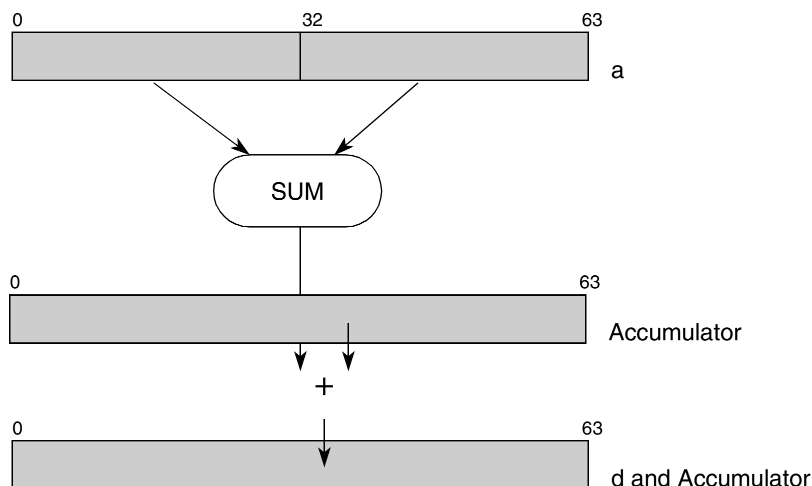


Figure 3-755. Vector Sum of Words Signed and Accumulate (__ev_sumwsaa)

d	a	Maps to
__ev64_opaque__	__ev64_opaque__	**evsumwsaa d,a**

__ev_sumwu[a]

__ev_sumwu[a]

Vector Sum of Words Unsigned (to Accumulator)

d = __ev_sumwu (a) (A = 0)

d = __ev_sumwua (a) (A = 1)

```
d0:63 ←EXTZ(a0:31) + EXTZ(a32:63)

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

The unsigned word elements of parameter **a** are summed together, and the result is placed in parameter **d**. If A = 1, the result in parameter **d** is also placed into the accumulator.

Other registers altered: ACC (if A=1)

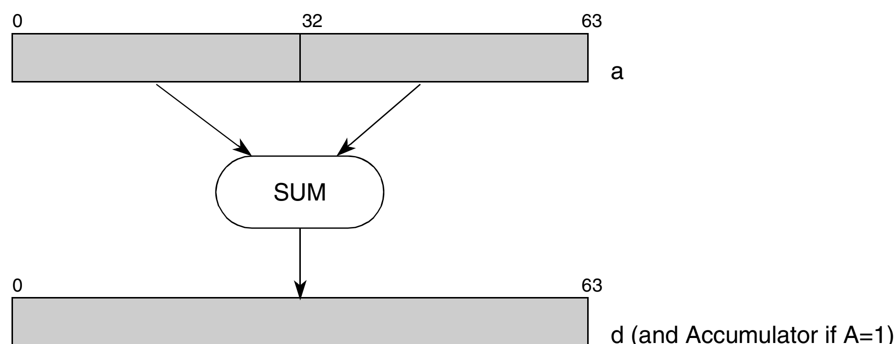


Figure 3-756. Vector Sum of Words Unsigned (to Accumulator) (__ev_sumwu)

A	d	a	Maps to
A = 0	__ev64_opaque__	__ev64_opaque__	**evsumwu d,a**
A = 1	__ev64_opaque__	__ev64_opaque__	**evsumwua d,a**

__ev_sumwuaa __ev_sumwuaa

Vector Sum of Words Unsigned and Accumulate

d = __ev_sumwuaa (a)

```
d0:63 ←EXTZ(a0:31) + EXTZ(a32:63) + ACC0:63

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

The unsigned word elements of parameter **a** are summed together then added to the contents of the accumulator, and the result is placed in parameter **d** and the accumulator.

Other registers altered: ACC

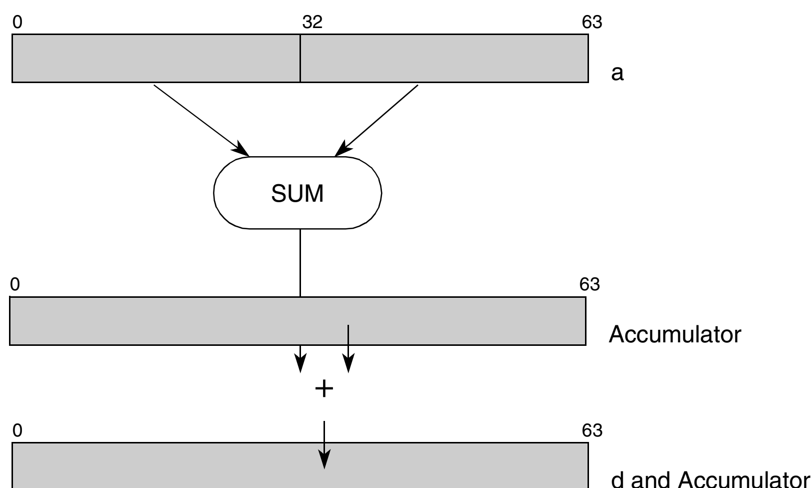


Figure 3-757. Vector Sum of Words Unsigned and Accumulate (__ev_sumwuaa)

d	a	Maps to
__ev64_opaque__	__ev64_opaque__	**evsumwuaa d,a**

__ev_swapbhilo

__ev_swapbhilo

Vector Swap Bytes High/Low

d = __ev_swapbhilo (a,b)

$d_{0:7} \leftarrow a_{8:15}$
$d_{8:15} \leftarrow a_{0:7}$
$d_{16:23} \leftarrow a_{24:31}$
$d_{24:31} \leftarrow a_{16:23}$
$d_{32:39} \leftarrow b_{40:47}$
$d_{40:47} \leftarrow b_{32:39}$
$d_{48:55} \leftarrow b_{56:63}$
$d_{56:63} \leftarrow b_{48:55}$

Pairs of byte elements within the high half words of parameter **a** and the low half words of parameter **b** are swapped and placed in parameter **d**.

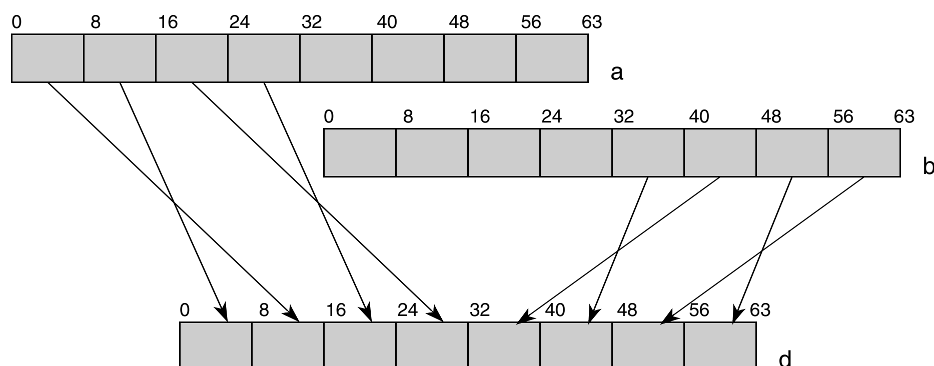


Figure 3-758. Vector Swap Bytes High/Low (__ev_swapbhilo)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evswapbhilo d,a,b**

__ev_swapblohi __ev_swapblohi

Vector Swap Bytes Low/High

d = __ev_swapblohi (a,b)

$d_{0:7} \leftarrow a_{40:47}$
$d_{8:15} \leftarrow a_{32:39}$
$d_{16:23} \leftarrow a_{56:63}$
$d_{24:31} \leftarrow a_{48:55}$
$d_{32:39} \leftarrow b_{8:15}$
$d_{40:47} \leftarrow b_{0:7}$
$d_{48:55} \leftarrow b_{24:31}$
$d_{56:63} \leftarrow b_{16:23}$

Pairs of byte elements within the low half words of parameter **a** and the high half words of parameter **b** are swapped and placed in parameter **d**.

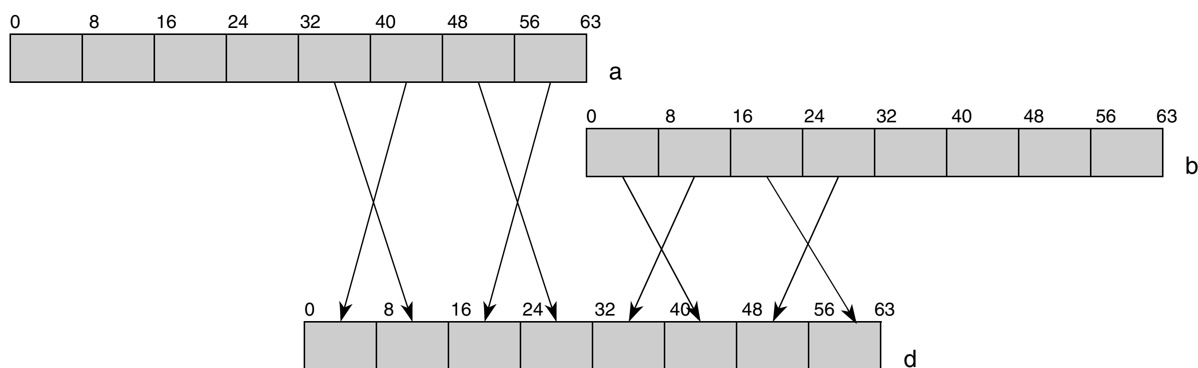


Figure 3-759. Vector Swap Bytes Low/High (__ev_swapblohi)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evswapblohi d,a,b**

__ev_swaphe

__ev_swaphe

Vector Swap Half Words Even

d = __ev_swaphe (a,b)

$d_{0:63} \leftarrow a_{32:47} \| b_{16:31} \| a_{0:15} \| b_{48:63}$

The even half word elements in parameter **a** are swapped and then merged with the odd half word elements in parameter **b** and placed into parameter **d**.

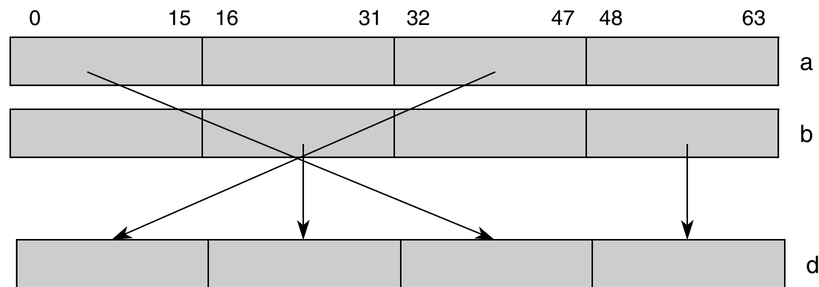


Figure 3-760. Vector Swap Half Words Even (__ev_swaphe)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evswaphe d,a,b**

__ev_swaphhi

__ev_swaphhi

Vector Swap Half Words High

d = __ev_swaphhi (a,b)

$d_{0:63} \leftarrow a_{16:31} \,||\, a_{0:15} \,||\, b_{32:47} \,||\, b_{48:63}$

The most significant two half word elements in parameter **a** are swapped and merged with the least significant two half word elements in parameter **b** and placed into parameter **d**.

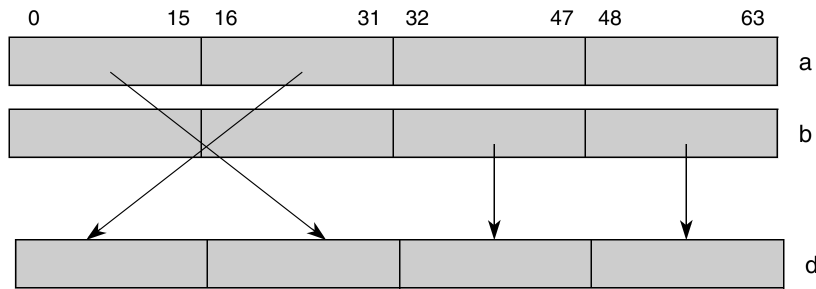


Figure 3-761. Vector Swap Half Words High (__ev_swaphhi)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evswaphhi d,a,b**

__ev_swaphhilo

__ev_swaphhilo

Vector Swap Half Words High/Low

d = __ev_swaphhilo (a,b)

$d_{0:63} \leftarrow a_{16:31} \; || \; a_{0:15} \; || \; b_{48:63} \; || \; b_{32:48}$

The most significant two half word elements in parameter **a** are swapped are merged with the swapped least significant two half word elements in parameter **b** and placed into parameter **d**.

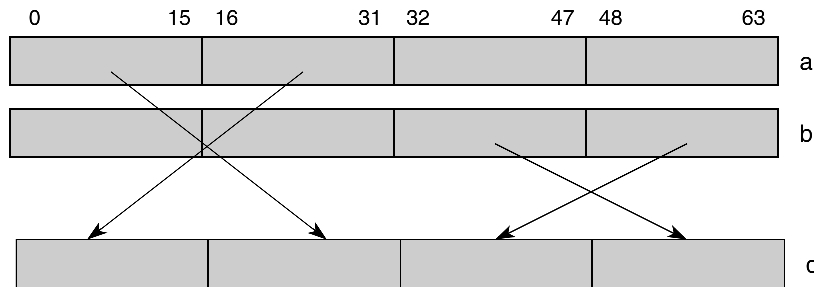


Figure 3-762. Vector Swap Half Words High/Low (__ev_swaphhilo)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evswaphhilo d,a,b**

__ev_swaphlo

Vector Swap Half Words Low

d = __ev_swaphlo (a,b)

$d_{0:63} \leftarrow b_{0:15} \,||\, b_{16:31} \,||\, a_{48:63} \,||\, a_{32:47}$

The least significant two half word elements in parameter **a** are swapped and then merged with the most significant two half word elements in parameter **b** and placed into parameter **d**.

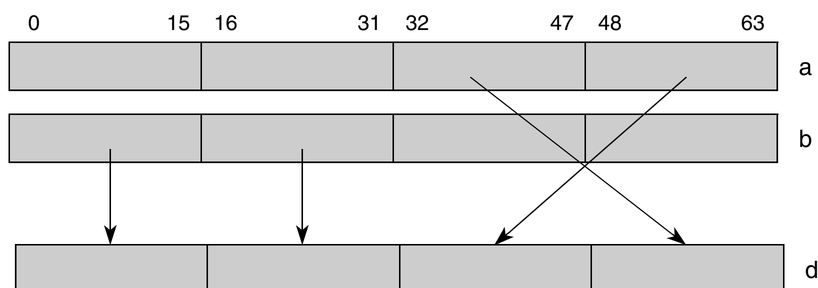


Figure 3-763. Vector Swap Half Words Low (__ev_swaphlo)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evswaphlo d,a,b**

__ev_swaphlohi __ev_swaphlohi

Vector Swap Half Words Low/High

d = __ev_swaphlohi (a,b)

$d_{0:63} \leftarrow a_{48:63} \, || \, a_{32:47} \, || \, b_{16:31} \, || \, b_{0:15}$

The least significant two half word elements in parameter **a** are swapped and merged with the swapped most significant two half word elements in parameter **b** and placed into parameter **d**.

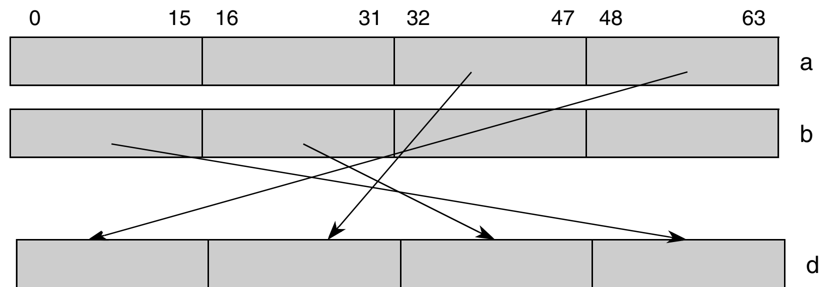


Figure 3-764. Vector Swap Half Words Low/High (__ev_swaphlohi)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evswaphlohi d,a,b**

__ev_swapho

__ev_swapho

Vector Swap Half Words Odd

d = __ev_swapho (a,b)

$d_{0:63} \leftarrow b_{0:15} \mid\mid a_{48:63} \mid\mid b_{32:47} \mid\mid a_{16:31}$

The odd half word elements in parameter **a** are swapped and then merged with the even half word elements in parameter **b** and placed into parameter **d**.

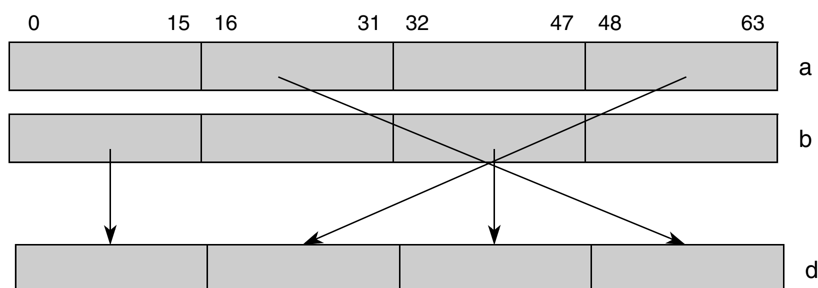


Figure 3-765. Vector Swap Half Words Odd (__ev_swapho)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evswapho d,a,b**

__ev_unpkhibsi

__ev_unpkhibsi

Vector Unpack High Bytes as Signed Integers

d = __ev_unpkhibsi (a)

```
d0:15 ←EXTS(a0:7)
d16:31 ←EXTS(a8:15)
d32:47 ←EXTS(a16:23)
d48:63 ←EXTS(a24:31)
```

The high four signed byte elements in parameter **a** are sign-extended into half words and placed in the same order into parameter **d**.

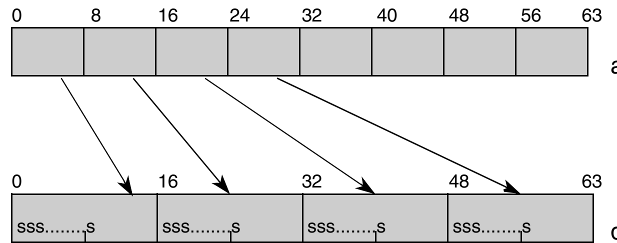


Figure 3-766. Vector Unpack High Bytes as Signed Integer (__ev_unpkhibsi)

d	a	Maps to
__ev64_opaque__	__ev64_opaque__	**evunpkhibsi d,a**

__ev_unpkhibui __ev_unpkhibui

Vector Unpack High Bytes as Unsigned Integers

d = __ev_unpkhibui (a)

$d_{0:15} \leftarrow EXTZ(a_{0:7})$
$d_{16:31} \leftarrow EXTZ(a_{8:15})$
$d_{32:47} \leftarrow EXTZ(a_{16:23})$
$d_{48:63} \leftarrow EXTZ(a_{24:31})$

The high four unsigned byte elements in parameter **a** are zero-extended into half words and placed in the same order into parameter **d**.

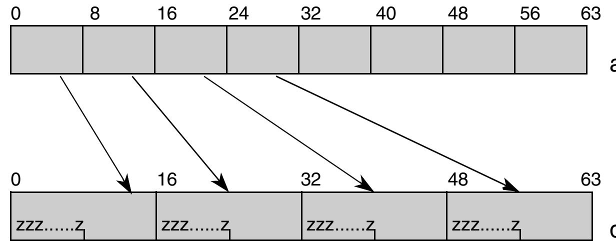


Figure 3-767. Vector Unpack High Bytes as Unsigned Integers (__ev_unpkhibui)

d	a	Maps to
__ev64_opaque__	__ev64_opaque__	**evunpkhibui d,a**

__ev_unpkhihf __ev_unpkhihf

Vector Unpack High Half Words as Fractional

d = __ev_unpkhihf (a)

$d_{0:31} \leftarrow a_{0:15} \; || \; {}^{16}0$
$d_{32:63} \leftarrow a_{16:31} \; || \; {}^{16}0$

The half word elements 0 and 1 in parameter **a** are padded with 16 zeros, and the results are placed into word elements 0 and 1 respectively of parameter **d**.

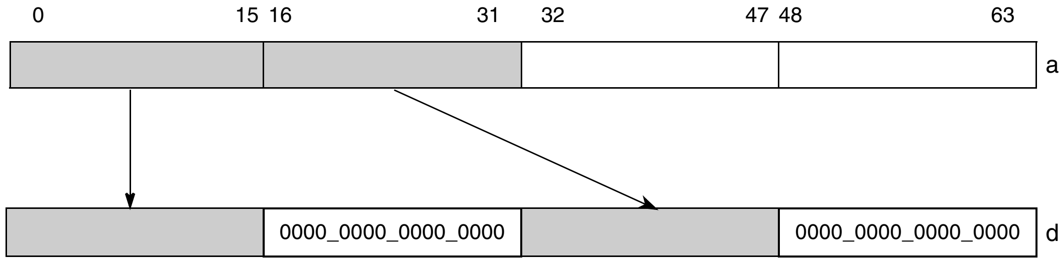


Figure 3-768. Vector Unpack High Half Words as Fractional (__ev_unpkhihf)

d	a	Maps to
__ev64_opaque__	__ev64_opaque__	**evunpkhihf d,a**

__ev_unpkhihsi __ev_unpkhihsi

Vector Extract High Half Words as Signed Integers

d = __ev_unpkhihsi (a)

$d_{0:31} \leftarrow EXTS(a_{0:15})$
$d_{32:63} \leftarrow EXTS(a_{16:31})$

The half word elements 0 and 1 in parameter **a** are sign-extended to 32 bits, and the results are placed into word elements 0 and 1 respectively of parameter **d**.

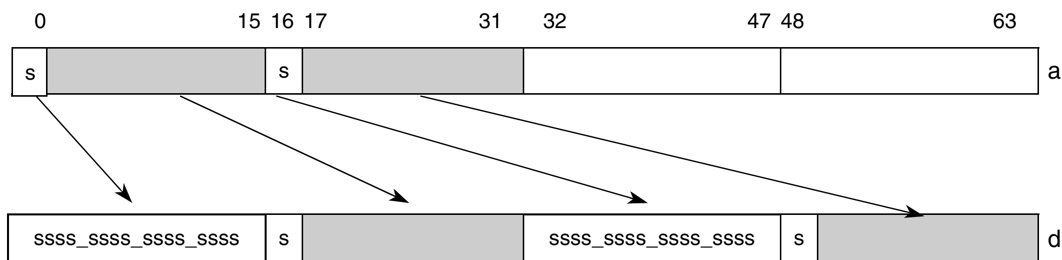


Figure 3-769. Vector Extract High Half Words as Signed Integers (__ev_unpkhihsi)

d	a	Maps to
__ev64_opaque__	__ev64_opaque__	**evunpkhihsi d,a**

__ev_unpkhihui __ev_unpkhihui

Vector Extract High Half Words as Signed Integers

d = __ev_unpkhihui (a)

$d_{0:31} \leftarrow EXTZ(a_{0:15})$
$d_{32:63} \leftarrow EXTZ(a_{16:31})$

The half word elements 0 and 1 in parameter **a** are zero-extended to 32 bits, and the results are placed into word elements 0 and 1 respectively of parameter **d**.

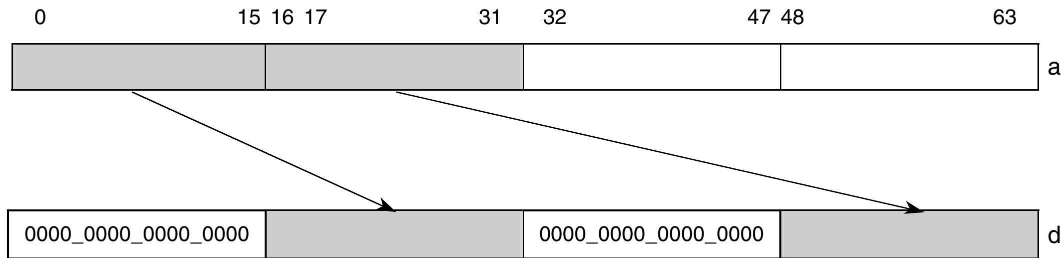


Figure 3-770. Vector Extract High Half Words as Unsigned Integers (__ev_unpkhihui)

d	a	Maps to
__ev64_opaque__	__ev64_opaque__	**evunpkhihui d,a**

__ev_unpkhiwgsf __ev_unpkhiwgsf

Vector Unpack High Word to Guarded Signed Fraction

d = __ev_unpkhiwgsf (a)

$d_{0:63} \leftarrow (EXTS_{48}(a_{0:31}) \; || \; {}^{16}0)$

The even word element of parameter **a** is sign-extended with 16 guard bits and padded with 16 0's, and the result is placed into parameter **d**.

Note: __ev_unpkhiwgsf is used to convert a 1.31 fractional word to a 17.47 fractional format.

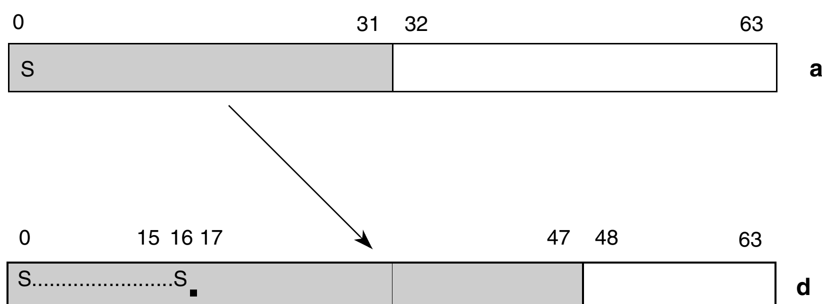


Figure 3-771. Vector Unpack High Word to Guarded Signed Fraction (__ev_unpkhiwgsf)

d	a	Maps to
__ev64_opaque__	__ev64_opaque__	**evunpkhiwgsf d,a**

__ev_unpklobsi __ev_unpklobsi

Vector Unpack Low Bytes as Signed Integers

d = __ev_unpklobsi (a)

$d_{0:15} \leftarrow EXTS(a_{32:39})$
$d_{16:31} \leftarrow EXTS(a_{40:47})$
$d_{32:47} \leftarrow EXTS(a_{48:55})$
$d_{48:63} \leftarrow EXTS(a_{56:63})$

The low four signed byte elements in parameter **a** are sign-extended into half words and placed in the same order into parameter **d**.

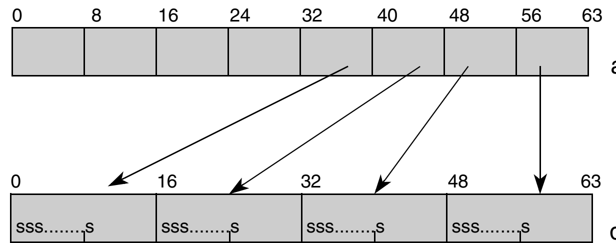


Figure 3-772. Vector Unpack Low Bytes as Signed Integers (__ev_unpklobsi)

d	a	Maps to
__ev64_opaque__	__ev64_opaque__	**evunpklobsi d,a**

__ev_unpklobui __ev_unpklobui

Vector Unpack Low Bytes as Unsigned Integers

d = __ev_unpklobui (a)

```
d0:15 ←EXTZ(a32:39)
d16:31 ←EXTZ(a40:47)
d32:47 ←EXTZ(a48:55)
d48:63 ←EXTZ(a56:63)
```

The low four unsigned byte elements in parameter **a** are zero-extended into half words and placed in the same order into parameter **d**.

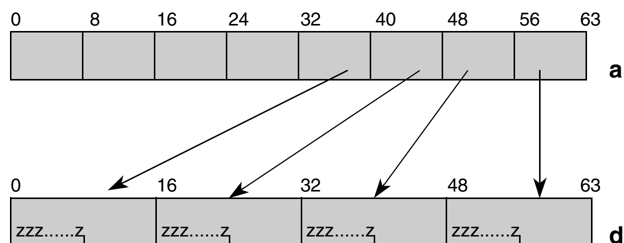


Figure 3-773. Vector Unpack Low Bytes as Unsigned Integers (__ev_unpklobui)

d	a	Maps to
__ev64_opaque__	__ev64_opaque__	**evunpklobui d,a**

__ev_unpklohf __ev_unpklohf

Vector Unpack Low Half Words as Fractional

d = __ev_unpklohf (a)

$d_{0:31} \leftarrow a_{32:47} \;||\; {}^{16}0$
$d_{32:63} \leftarrow a_{48:63} \;||\; {}^{16}0$

The half word elements 2 and 3 in parameter **a** are padded with 16 zeros, and the results are placed into word elements 0 and 1 respectively of parameter **d**.

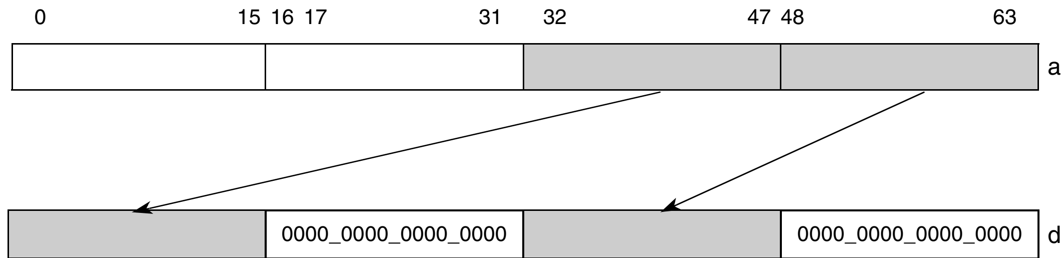


Figure 3-774. Vector Unpack Low Half Words as Fractional (__ev_unpklohf)

d	a	Maps to
__ev64_opaque__	__ev64_opaque__	**evunpklohf d,a**

__ev_unpklohsi

__ev_unpklohsi

Vector Unpack Low Half Words as Signed Integers

d = __ev_unpklohsi (a)

```
d0:31  ← EXTS(a32:47)
d32:63 ← EXTS(a48:63)
```

The half word elements 2 and 3 in parameter **a** are sign-extended to 32 bits, and the results are placed into word elements 0 and 1 respectively of parameter **d**.

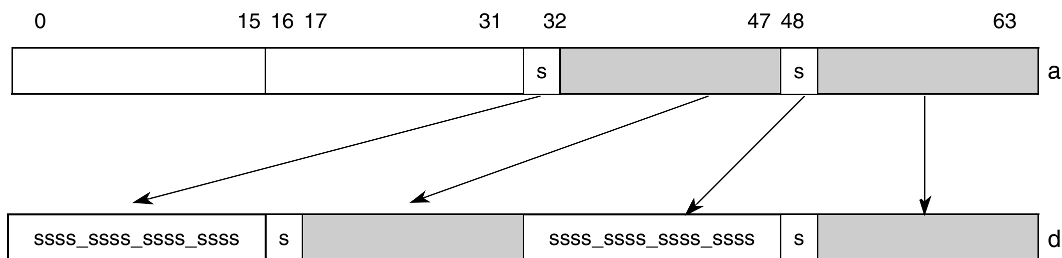


Figure 3-775. Vector Unpack Low Half Words as Signed Integers (__ev_unpklohsi)

d	a	Maps to
__ev64_opaque__	__ev64_opaque__	**evunpklohsi d,a**

__ev_unpklohui __ev_unpklohui

Vector Unpack Low Half Words as Unsigned Integers

d = __ev_unpklohui (a)

```
d0:31  ← EXTZ(a32:47)
d32:63 ← EXTZ(a48:63)
```

The half word elements 2 and 3 in parameter **a** are zero-extended to 32 bits, and the results are placed into word elements 0 and 1 respectively of parameter **d**.

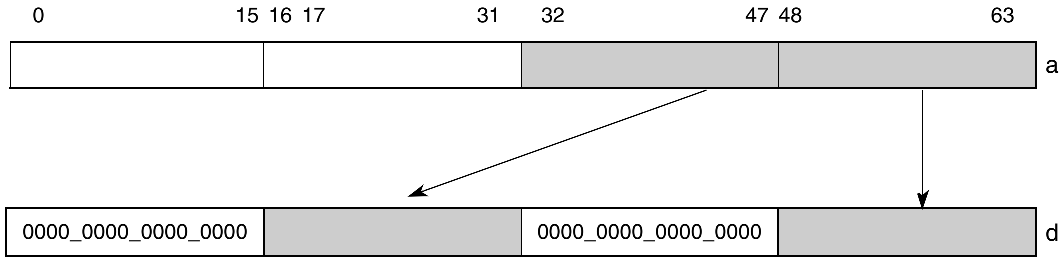


Figure 3-776. Vector Unpack Low Half Words as Unsigned Integers (__ev_unpklohui)

d	a	Maps to
__ev64_opaque__	__ev64_opaque__	**evunpklohui d,a**

__ev_unpklowgsf __ev_unpklowgsf

Vector Unpack Low Word to Guarded Signed Fraction

d = __ev_unpklowgsf (a)

$d_{0:63} \leftarrow (EXTS_{48}(a_{32:63}) \; || \; ^{16}0)$

The odd word element of parameter **a** is sign-extended with 16 guard bits and padded with 16 0's, and the result is placed into parameter **d**.

Note: __ev_unpklowgsf is used to convert a 1.31 fractional word to a 17.47 fractional format.

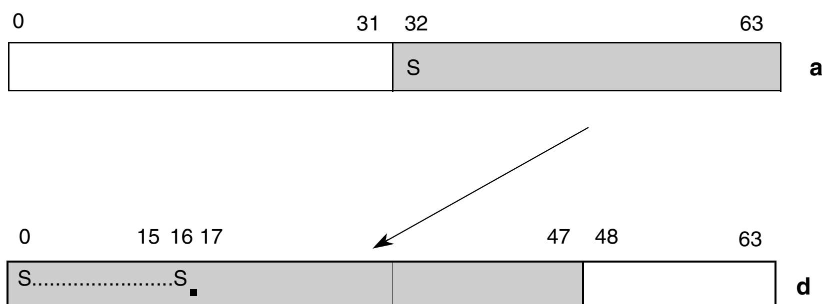


Figure 3-777. Vector Unpack Low Word to Guarded Signed Fraction(__ev_unpklowgsf)

d	a	Maps to
__ev64_opaque__	__ev64_opaque__	**evunpklowgsf d,a**

__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**.

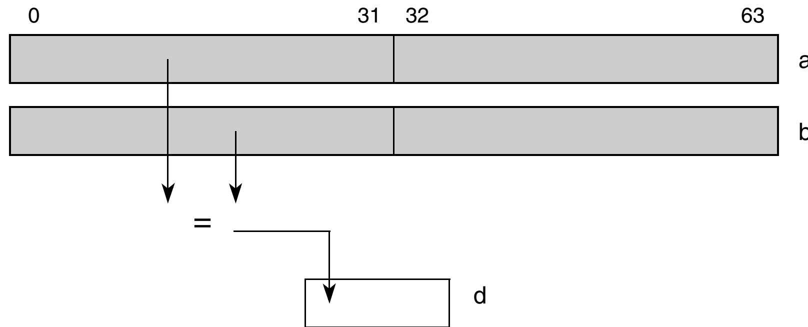


Figure 3-778. Vector Upper Equal(__ev_upper_eq)

d	a	b	Maps to
_Bool	__ev64_opaque__	__ev64_opaque__	**evcmpeq 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**.

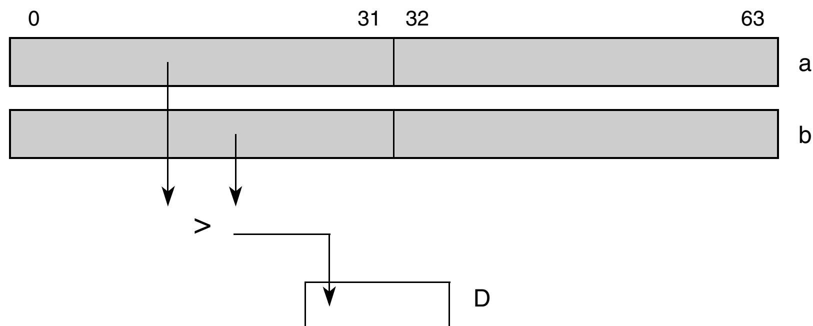


Figure 3-779. Vector Upper Greater Than Signed (__ev_upper_gts)

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**.

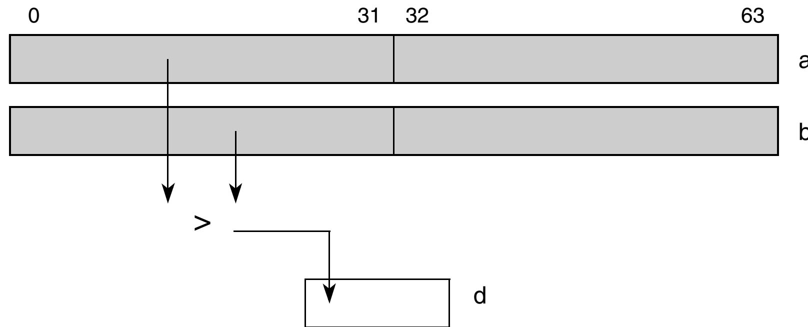


Figure 3-780. Vector Upper Greater Than Unsigned (__ev_upper_gtu)

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**.

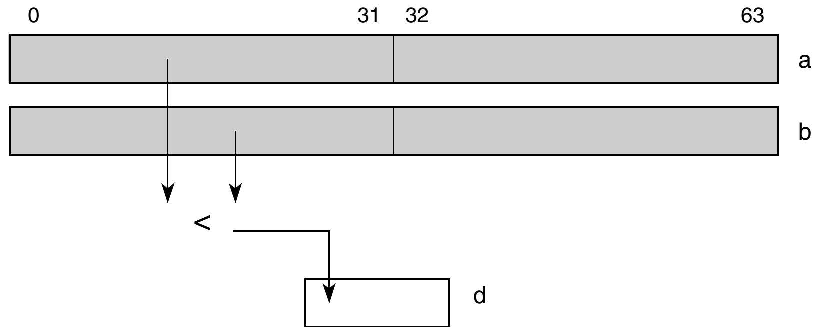


Figure 3-781. Vector Upper Less Than Signed (__ev_upper_lts)

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**.

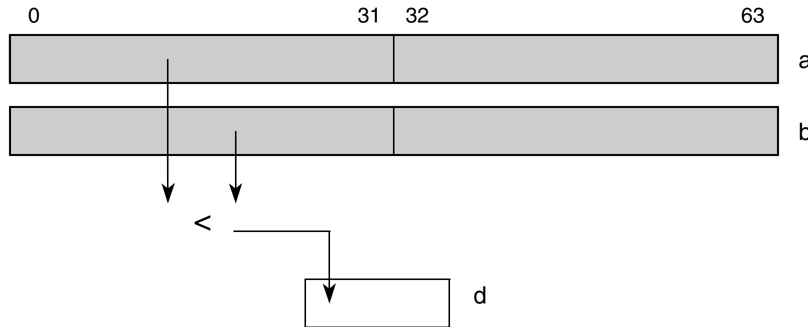


Figure 3-782. Vector Upper Less Than Unsigned (__ev_upper_ltu)

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**.

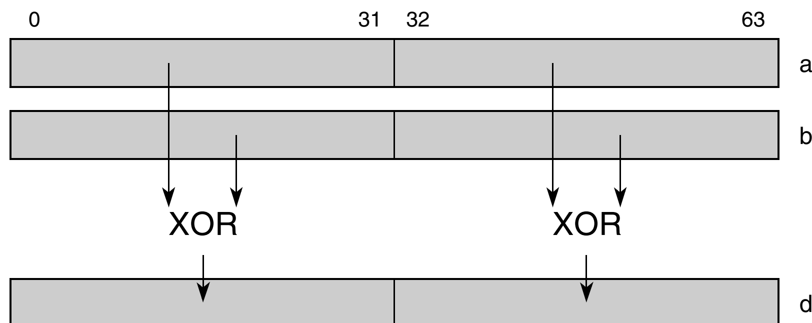


Figure 3-783. Vector XOR (__ev_xor)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evxor d,a,b**

__ev_xtrb

__ev_xtrb

Vector Extract Byte

d = __ev_xtrb (a,b,c)

```
temp0:7 ←a0+(c*8):7+(c*8)
if (b=0) then d0:7 ←temp0:7 else d0:7 ←0x00
if (b=1) then d8:15 ←temp0:7 else d8:15 ←0x00
if (b=2) then d16:23 ←temp0:7 else d16:23 ←0x00
if (b=3) then d24:31 ←temp0:7 else d24:31 ←0x00
if (b=4) then d32:39 ←temp0:7 else d32:39 ←0x00
if (b=5) then d40:47 ←temp0:7 else d40:47 ←0x00
if (b=6) then d48:55 ←temp0:7 else d48:55 ←0x00
if (b=7) then d56:63 ←temp0:7 else d56:63 ←0x00
```

The byte element of parameter **a** specified by the 3-bit unsigned literal in parameter **c** is placed into the byte element of parameter **d** specified by the 3-bit unsigned literal in parameter **b**, zero-filling the remaining bytes of parameter **d**. Byte 0 is the most-significant byte.

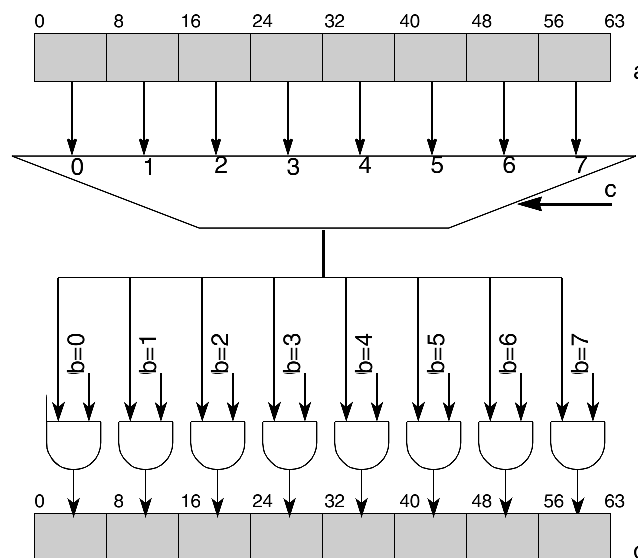


Figure 3-784. Vector Extract Byte (__ev_xtrb)

d	a	b	c	Maps to
__ev64_opaque__	__ev64_opaque__	3-bit unsigned literal	3-bit unsigned literal	**evxtrb d,a,b,c**

__ev_xtrd __ev_xtrd

Vector Extract Double Word

d = __ev_xtrd (a,b,c)

```
offset ←c
n ←(offset * 8)
temp0:127 ←a0:63 || b0:63
d0:63 ←temp0+n:63+n
```

A double word is extracted from the concatenation of parameter **a** and parameter **b** beginning at byte offset provided by the 3-bit unsigned literal in parameter **c** and placed into parameter **d**. “offset” must be in the range [1:7].

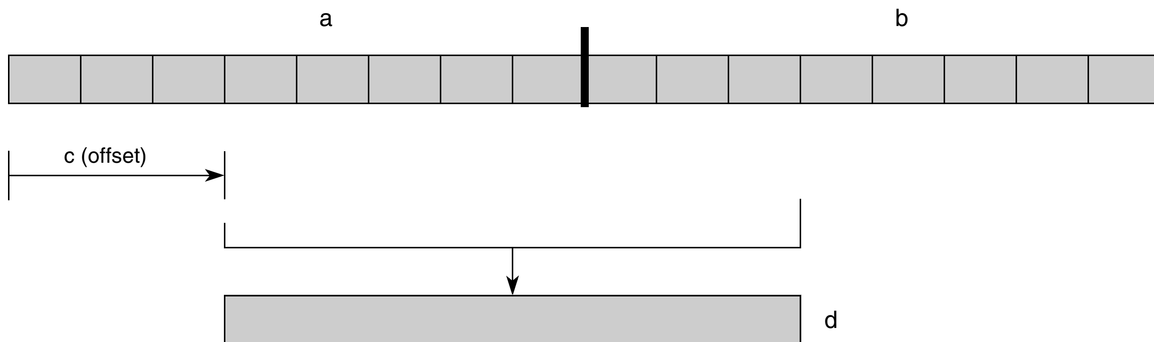


Figure 3-785. Vector Extract Double Word (__ev_xtrd)

d	a	b	c	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	3-bit unsigned literal	**evxtrd d,a,b,c**

__ev_xtrh

__ev_xtrh

Vector Extract Half Word

d = __ev_xtrh (a,b,c)

```
temp0:15 ←a0+(c*16):15+(c*16)
if (b=0) then d0:15 ←temp0:15 else d0:15 ←0x0000
if (b=1) then d16:31 ←temp0:15 else d16:31 ←0x0000
if (b=2) then d32:47 ←temp0:15 else d32:47 ←0x0000
if (b=3) then d48:63 ←temp0:15 else d48:63 ←0x0000
```

The half word element of parameter **a** specified by the 2-bit unsigned literal in parameter **c** is placed into the half word element of parameter **d** specified by the 2-bit unsigned literal in parameter **b**, zero-filling the remaining half words of parameter **d**. Half word 0 is the most-significant half word.

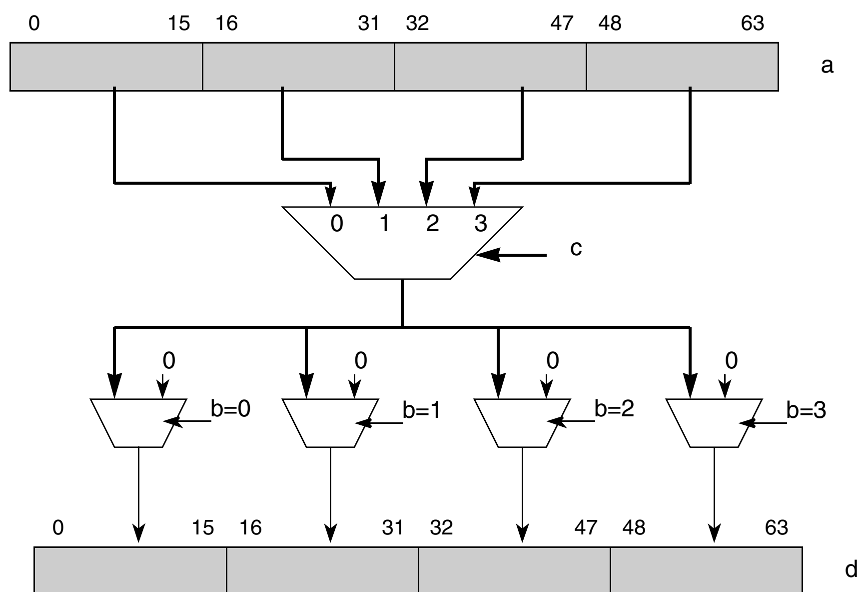


Figure 3-786. Vector Extract Half Word (__ev_xtrh)

d	a	b	c	Maps to
__ev64_opaque__	__ev64_opaque__	2-bit unsigned literal	2-bit unsigned literal	**evxtrh d,a,b,c**

3.7.2 EFP2 Intrinsic Definitions

This section provides the intrinsic definitions for the Embedded Floating-Point Version 2 (EFP2) APU.

__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**.

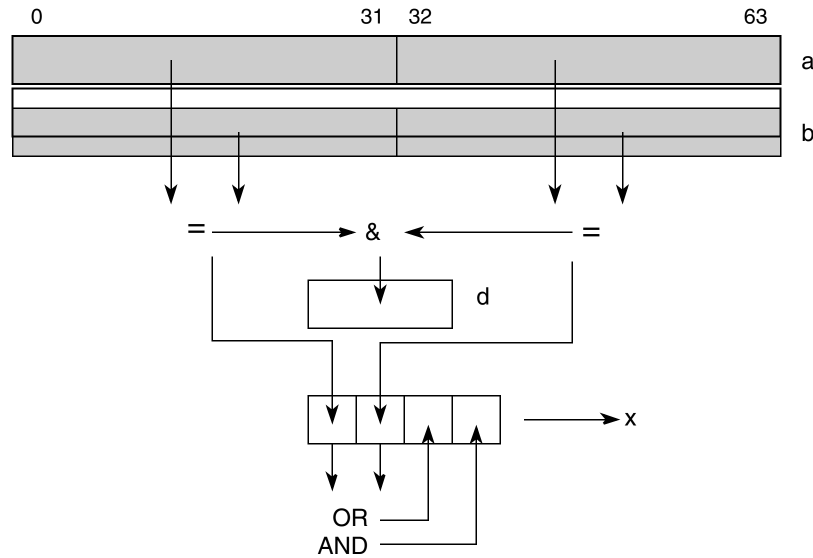


Figure 3-787. Vector All Floating-Point Equal (__ev_all_fs_eq)

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**.

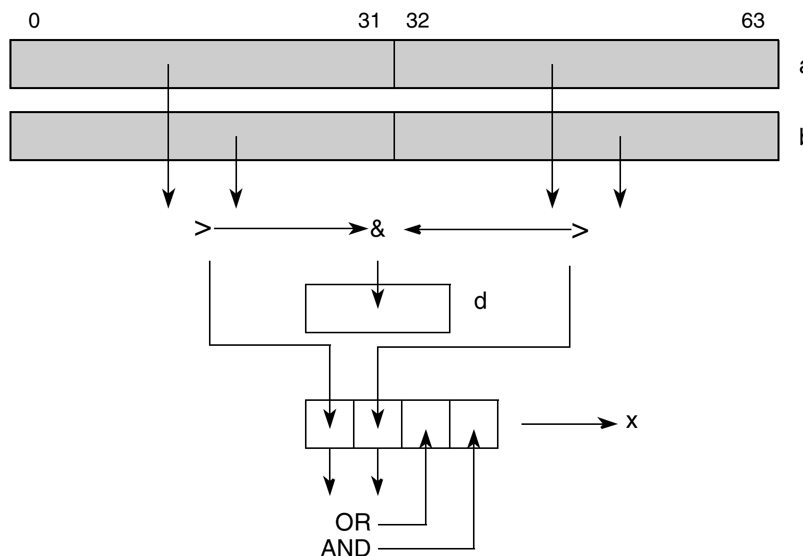


Figure 3-788. Vector All Floating-Point Greater Than (__ev_all_fs_gt)

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**.

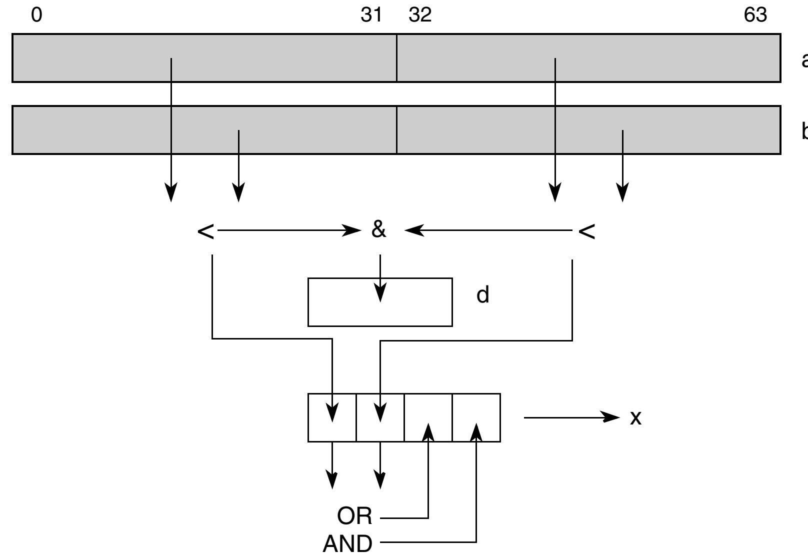


Figure 3-789. Vector All Floating-Point Less Than (__ev_all_fs_lt)

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.

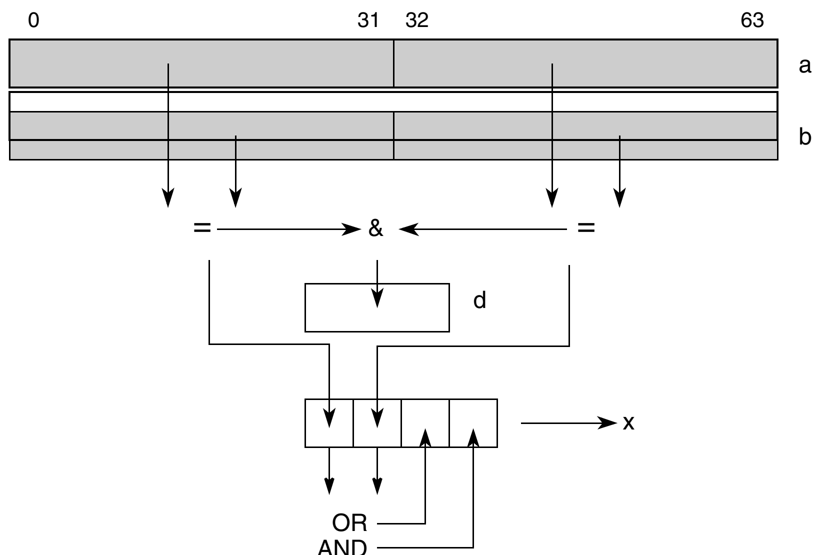


Figure 3-790. Vector All Floating-Point Test Equal (__ev_all_fs_tst_eq)

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.

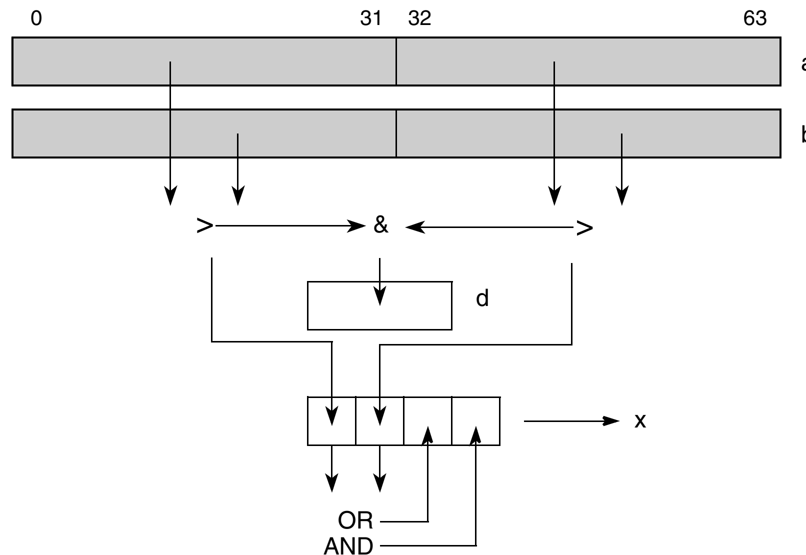


Figure 3-791. Vector All Floating-Point Test Greater Than (__ev_all_fs_tst_gt)

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.

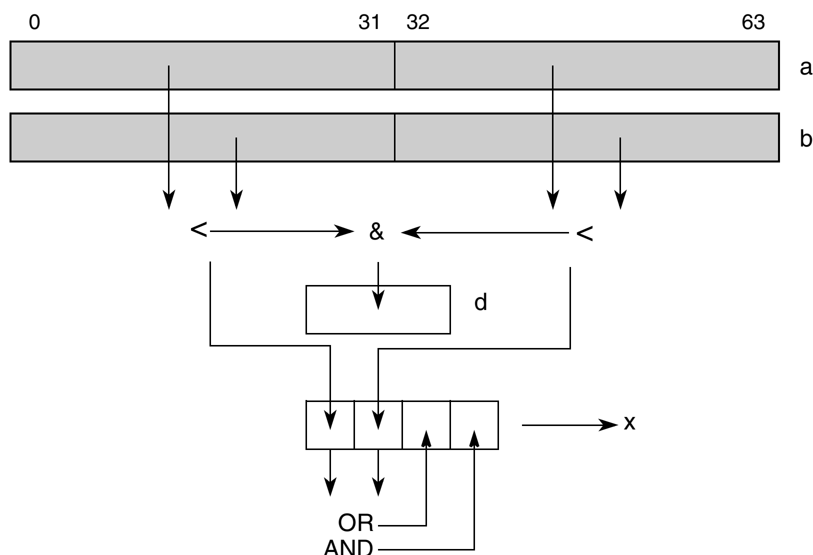


Figure 3-792. Vector All Floating-Point Test Less Than (__ev_all_fs_tst_lt)

d	a	b	Maps to
_Bool	__ev64_opaque__	__ev64_opaque__	**evfststlt x,a,b**

__ev_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**.

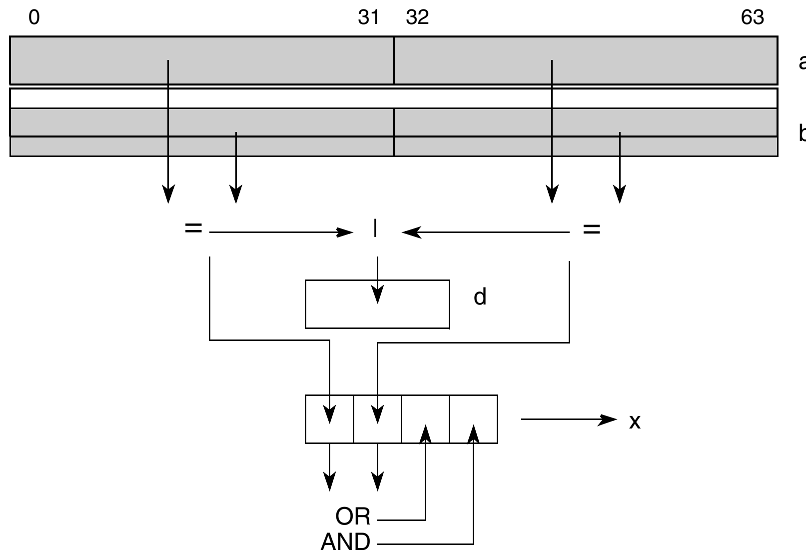


Figure 3-793. Vector Any Floating-Point Equal (__ev_any_fs_eq)

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**.

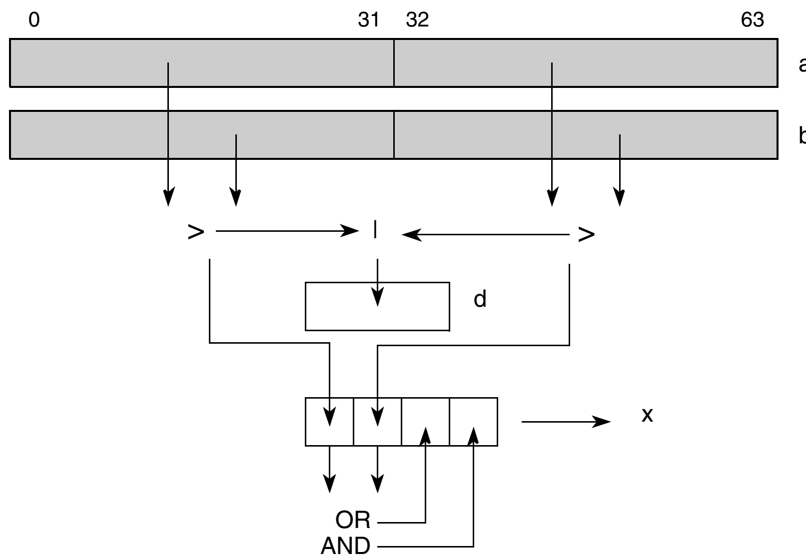


Figure 3-794. Vector Any Floating-Point Greater Than (__ev_any_fs_gt)

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**.

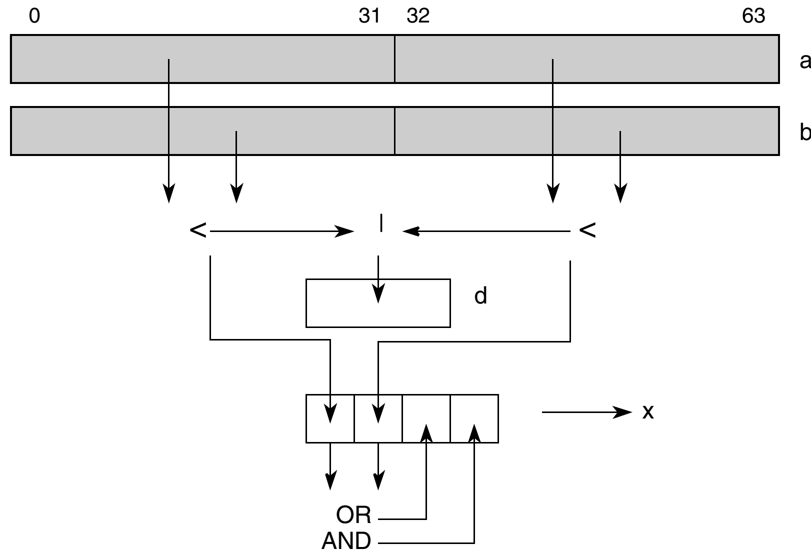


Figure 3-795. Vector Any Floating-Point Less Than (__ev_any_fs_lt)

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.

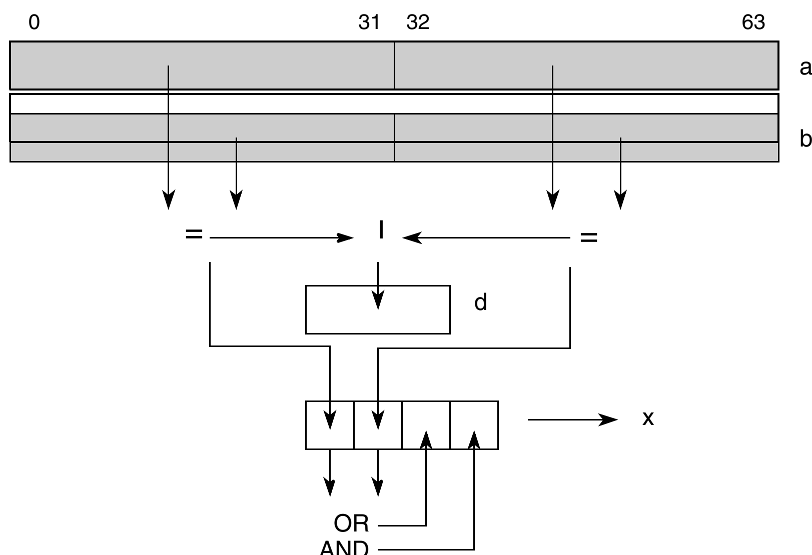


Figure 3-796. Vector Any Floating-Point Test Equal (__ev_any_fs_tst_eq)

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.

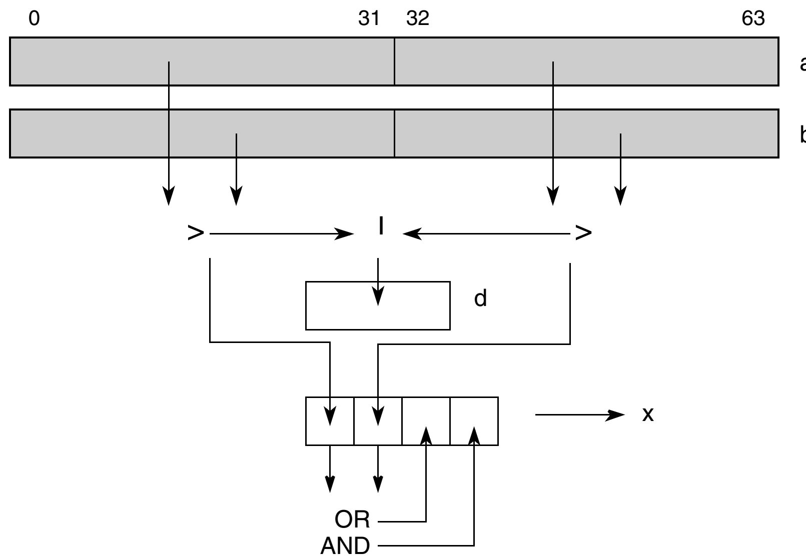


Figure 3-797. Vector Any Floating-Point Test Greater Than (__ev_any_fs_tst_gt)

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.

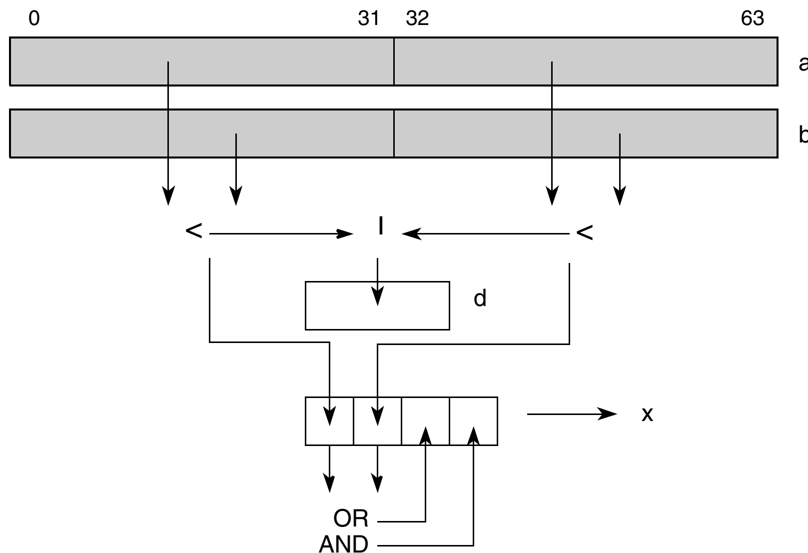


Figure 3-798. Vector Any Floating-Point Test Less Than (__ev_any_fs_tst_lt)

d	a	b	Maps to
_Bool	__ev64_opaque__	__ev64_opaque__	**evfststlt x,a,b**

__ev_fsabs __ev_fsabs

Vector Floating-Point Absolute Value

d = __ev_fsabs (a)

```
d0:31 ← 0b0 || a1:31
d32:63 ← 0b0 || a33:63
```

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.

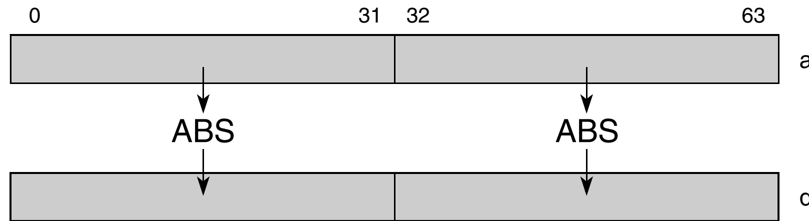


Figure 3-799. Vector Floating-Point Absolute Value (__ev_fsabs)

d	a	Maps to
__ev64_opaque__	__ev64_opaque__	**evfsabs 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 parameter **a** or parameter **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

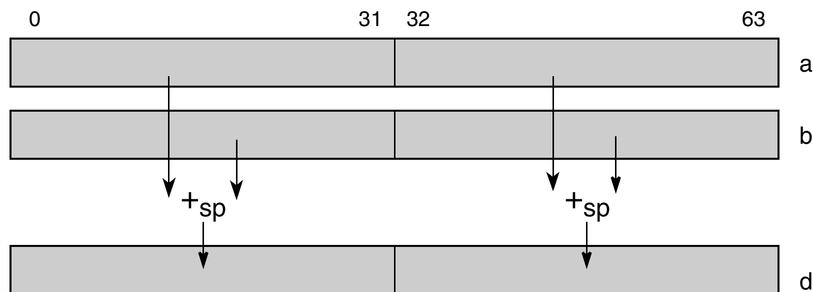


Figure 3-800. Vector Floating-Point Add (__ev_fsadd)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evfsadd d,a,b**

__ev_fsaddsub __ev_fsaddsub

Vector Floating-Point Single-Precision Add / Subtract

d = __ev_fsaddsub (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 high order single-precision floating-point element of parameter **a** is added to the corresponding element of parameter **b**, the low order single-precision floating-point element of parameter **b** is subtracted from the corresponding element of parameter **a**, and the results are stored in parameter **d**. If an element of parameter **a** is NaN or infinity, the corresponding result is either *pmax* or *nmax* (as appropriate). Otherwise, if an element of parameter **b** is NaN or infinity, the corresponding result is either *pmax* or *nmax* (as appropriate). Otherwise, if an overflow occurs, *pmax* or *nmax* (as appropriate) is stored in the corresponding element of parameter **d**. If an underflow occurs, +0 (for rounding modes RN, RZ, RP) or -0 (for rounding mode RM) is stored in the corresponding element of parameter **d**.

Exceptions:

If the contents of either element of parameter **a** or parameter **b** are Infinity, Denorm, or NaN, $SPEFSCR_{FINV,FINVH}$ are set appropriately, and $SPEFSCR_{FGH,FXH,FG,FX}$ are cleared appropriately. If $SPEFSCR_{FINVE}$ is set, an interrupt is taken and parameter **d** is not updated. Otherwise, if an overflow occurs, $SPEFSCR_{FOVF,FOVFH}$ are set appropriately, or if an underflow occurs, $SPEFSCR_{FUNF,FUNFH}$ are set appropriately. If either underflow or overflow exceptions are enabled and a corresponding status bit is set, an interrupt is taken. If any of these interrupts are taken, parameter **d** is not updated.

If either result element of this instruction is inexact, or overflows but overflow exceptions are disabled, and no other interrupt is taken, or underflows but underflow exceptions are disabled, and no other interrupt is taken, $SPEFSCR_{FINXS,FINXSH}$ is set. If the floating-point inexact exception is enabled, an interrupt is taken using the floating-point round interrupt vector. In this case, parameter **d** is updated with the truncated result(s). The FG and FX bits are properly updated to allow rounding to be performed in the interrupt handler.

FG and FX (FGH and FXH) are cleared if an overflow or underflow interrupt is taken, or if an invalid operation/input error is signaled for the low (high) element (regardless of FINVE).

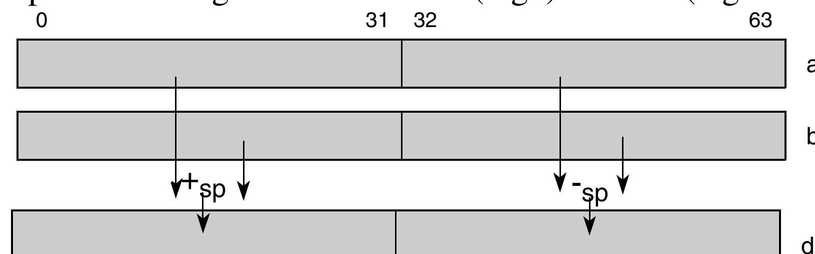


Figure 3-801. Vector Floating-Point Single-Precision Add / Subtract (__ev_fsaddsub)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evfsaddsub d,a,b**

__ev_fsaddsubx __ev_fsaddsubx

Vector Floating-Point Single-Precision Add / Subtract Exchanged

d = __ev_fsaddsubx (a,b)

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

The high-order single-precision floating-point element of parameter **b** is added to the low-order element of parameter **a**, the low-order single-precision floating-point element of parameter **b** is subtracted from the high-order element of parameter **a**, and the results are stored in parameter **d**. If an element of parameter **a** is NaN or infinity, the corresponding result is either *pmax* or *nmax* (as appropriate). Otherwise, if an element of parameter **b** is NaN or infinity, the corresponding result is either *pmax* or *nmax* (as appropriate). Otherwise, if an overflow occurs, *pmax* or *nmax* (as appropriate) is stored in the corresponding element of parameter d. If an underflow occurs, +0 (for rounding modes RN, RZ, RP) or -0 (for rounding mode RM) is stored in the corresponding element of parameter **d**.

Exceptions:

If the contents of either element of parameter **a** or **b** are Infinity, Denorm, or NaN, $SPEFSCR_{FINV,FINVH}$ are set appropriately, and $SPEFSCR_{FGH,FXH,FG,FX}$ are cleared appropriately. If $SPEFSCR_{FINVE}$ is set, an interrupt is taken and parameter **d** is not updated. Otherwise, if an overflow occurs, $SPEFSCR_{FOVF,FOVFH}$ are set appropriately, or if an underflow occurs, $SPEFSCR_{FUNF,FUNFH}$ are set appropriately. If either underflow or overflow exceptions are enabled and a corresponding status bit is set, an interrupt is taken. If any of these interrupts are taken, parameter **d** is not updated.

If either result element of this instruction is inexact, or overflows but overflow exceptions are disabled, and no other interrupt is taken, or underflows but underflow exceptions are disabled, and no other interrupt is taken, $SPEFSCR_{FINXS,FINXSH}$ is set. If the floating-point inexact exception is enabled, an interrupt is taken using the floating-point round interrupt vector. In this case, parameter **d** is updated with the truncated result(s). The FG and FX bits are properly updated to allow rounding to be performed in the interrupt handler.

FG and FX (FGH and FXH) are cleared if an overflow or underflow interrupt is taken, or if an invalid operation/input error is signaled for the low (high) element (regardless of FINVE).

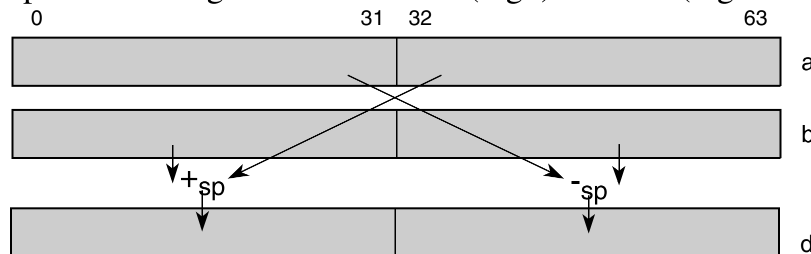


Figure 3-802. Vector Floating-Point Add / Subtract Exchanged (__ev_fsaddsubx)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evfsaddsubx d,a,b**

__ev_fsaddx __ev_fsaddx

Vector Floating-Point Single-Precision Add Exchanged

d = __ev_fsaddx (a,b)

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

The high-order single-precision floating-point element of parameter **b** is added to the low-order element of parameter **a**, the low-order single-precision floating-point element of parameter **b** is added to the high-order element of parameter **a**, and the results are stored in parameter **d**. If an element of parameter **a** is NaN or infinity, the corresponding result is either *pmax* or *nmax* (as appropriate). Otherwise, if an element of parameter **b** is NaN or infinity, the corresponding result is either *pmax* or *nmax* (as appropriate). Otherwise, if an overflow occurs, *pmax* or *nmax* (as appropriate) is stored in the corresponding element of parameter **d**. If an underflow occurs, +0 (for rounding modes RN, RZ, RP) or -0 (for rounding mode RM) is stored in the corresponding element of parameter **d**.

Exceptions:

If the contents of either element of parameter **a** or **b** are Infinity, Denorm, or NaN, $SPEFSCR_{FINV,FINVH}$ are set appropriately, and $SPEFSCR_{FGH,FXH,FG,FX}$ are cleared appropriately. If $SPEFSCR_{FINVE}$ is set, an interrupt is taken and parameter **d** is not updated. Otherwise, if an overflow occurs, $SPEFSCR_{FOVF,FOVFH}$ are set appropriately, or if an underflow occurs, $SPEFSCR_{FUNF,FUNFH}$ are set appropriately. If either underflow or overflow exceptions are enabled and a corresponding status bit is set, an interrupt is taken. If any of these interrupts are taken, parameter **d** is not updated.

If either result element of this instruction is inexact, or overflows but overflow exceptions are disabled, and no other interrupt is taken, or underflows but underflow exceptions are disabled, and no other interrupt is taken, $SPEFSCR_{FINXS,FINXSH}$ is set. If the floating-point inexact exception is enabled, an interrupt is taken using the floating-point round interrupt vector. In this case, parameter **d** is updated with the truncated result(s). The FG and FX bits are properly updated to allow rounding to be performed in the interrupt handler.

FG and FX (FGH and FXH) are cleared if an overflow or underflow interrupt is taken, or if an invalid operation/input error is signaled for the low (high) element (regardless of FINVE).

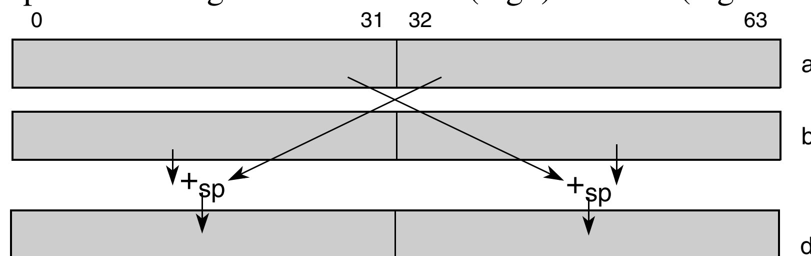


Figure 3-803. Vector Floating-Point Add Exchanged (__ev_fsaddx)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evfsaddx d,a,b**

__ev_fscfsf __ev_fscfsf

Vector Convert Floating-Point from Signed Fraction

d = __ev_fscfsf (a)

```
d0:31 ←CnvtI32ToFP32Sat(a0:31, SIGN, UPPER, F)
d32:63 ←CnvtI32ToFP32Sat(a32:63, 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

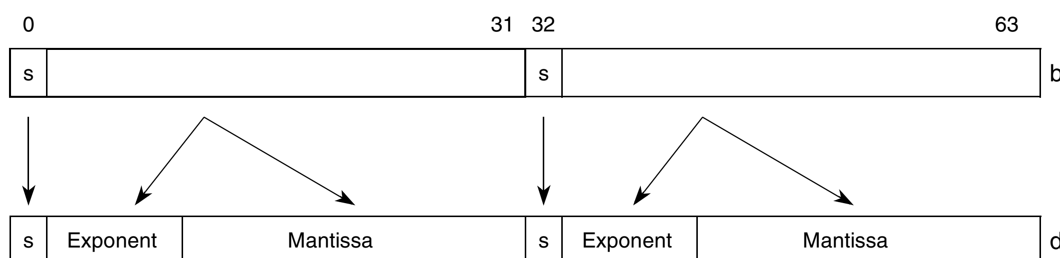


Figure 3-804. Vector Convert Floating-Point from Signed Fraction (__ev_fscfsf)

d	a	Maps to
__ev64_opaque__	__ev64_opaque__	**evfscfsf 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

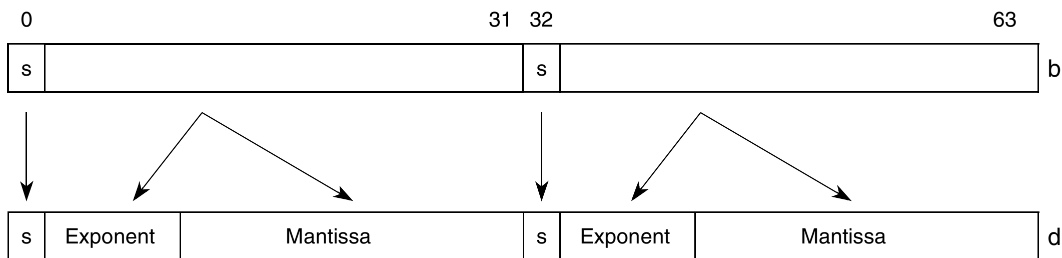


Figure 3-805. Vector Convert Floating-Point from Signed Integer (__ev_fscfsi)

d	a	Maps to
__ev64_opaque__	__ev64_opaque__	**evfscfsi d,a**

__ev_fscfuf __ev_fscfuf

Vector Convert Floating-Point from Unsigned Fraction

d = __ev_fscfuf (a)

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

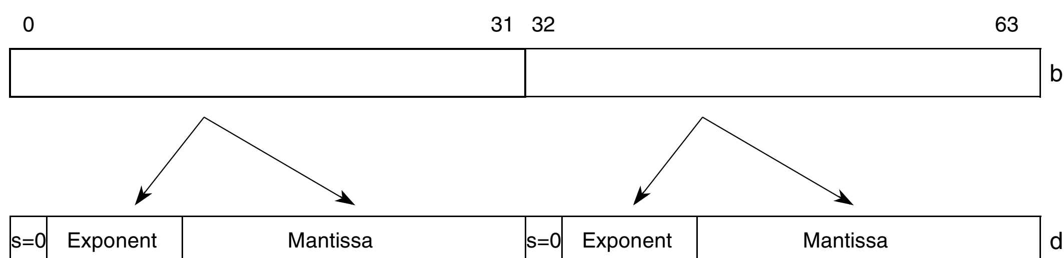


Figure 3-806. Vector Convert Floating-Point from Unsigned Fraction (__ev_fscfuf)

d	a	Maps to
__ev64_opaque__	__ev64_opaque__	**evfscfuf 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

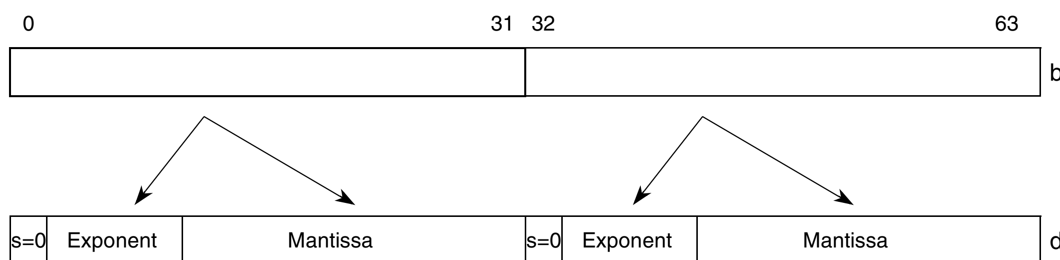


Figure 3-807. Vector Convert Floating-Point from Unsigned Integer (__ev_fscfui)

d	a	Maps to
__ev64_opaque__	__ev64_opaque__	**evfscfui 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

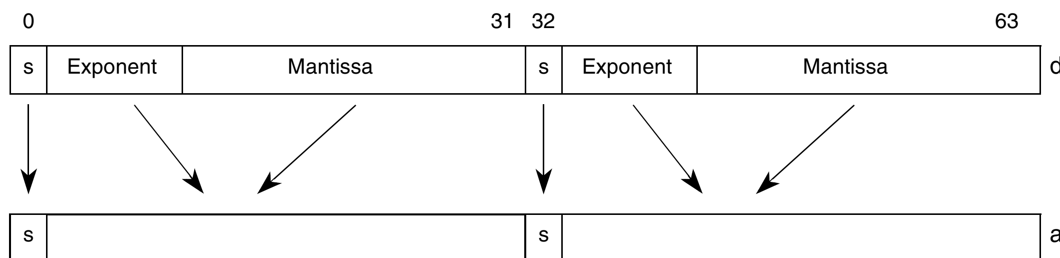


Figure 3-808. Vector Convert Floating-Point to Signed Fraction (__ev_x)

d	a	Maps to
__ev64_opaque__	__ev64_opaque__	**evfsctsf 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

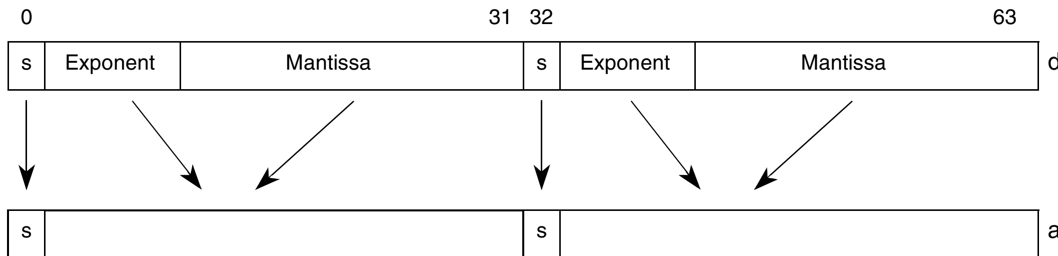


Figure 3-809. Vector Convert Floating-Point to Signed Integer (__ev_fsctsi)

d	a	Maps to
__ev64_opaque__	__ev64_opaque__	**evfsctsi d,a**

__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

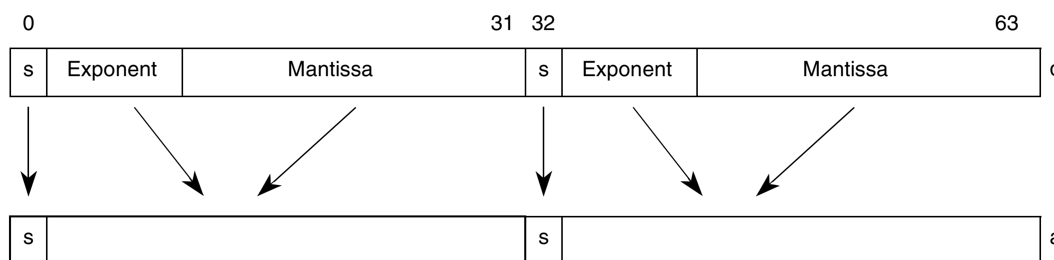


Figure 3-810. Vector Convert Floating-Point to Signed Integer with Round Toward Zero (__ev_fsctsiz)

d	a	Maps to
__ev64_opaque__	__ev64_opaque__	**evfsctsiz 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

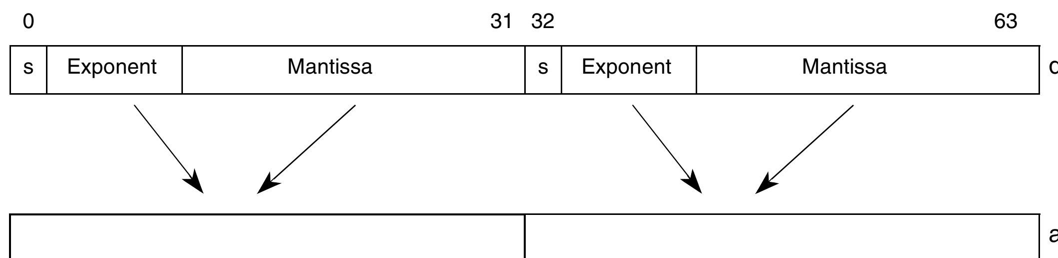


Figure 3-811. Vector Convert Floating-Point to Unsigned Fraction (__ev_fsctuf)

d	a	Maps to
__ev64_opaque__	__ev64_opaque__	**evfsctuf 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

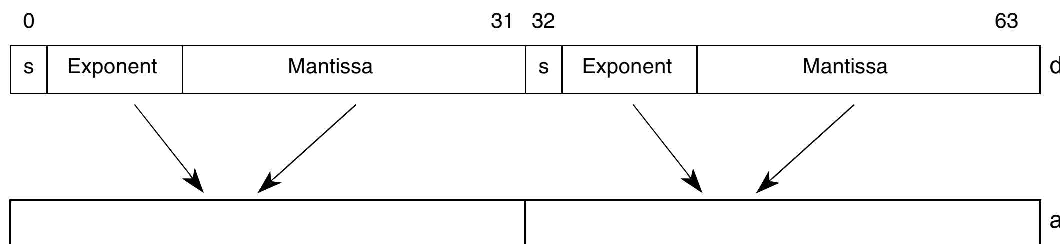


Figure 3-812. Vector Convert Floating-Point to Unsigned Integer (__ev_fsctui)

d	a	Maps to
__ev64_opaque__	__ev64_opaque__	**evfsctui 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

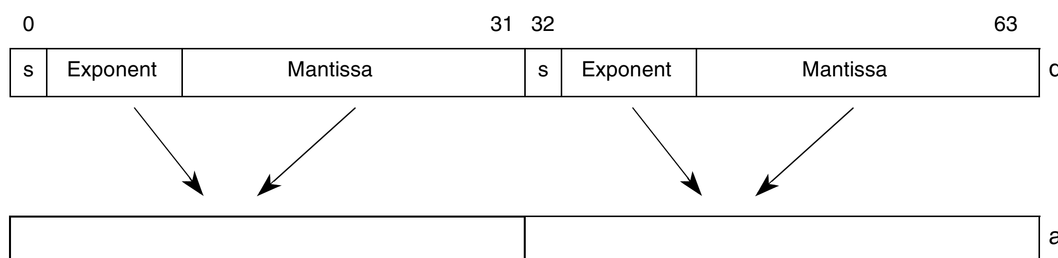


Figure 3-813. Vector Convert Floating-Point to Unsigned Integer with Round Toward Zero (__ev_fsctuiz)

d	a	Maps to
__ev64_opaque__	__ev64_opaque__	**evfsctuiz d,a**

__ev_fsdiff __ev_fsdiff

Vector Floating-Point Single-Precision Differences

d = __ev_fsdiff (a,b)

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

The low-order single-precision floating-point element of parameter **a** is subtracted from the high-order element of parameter **a**, the low-order single-precision floating-point element of parameter **b** is subtracted from the high-order element of parameter **b**, and the results are stored in parameter **d**. If the high-order element of parameter **a** or **b** is NaN or infinity, the corresponding result is either *pmax* or *nmax* (as appropriate). Otherwise, if the low order element of parameter **a** or **b** is NaN or infinity, the corresponding result is either *pmax* or *nmax* (as appropriate). Otherwise, if an overflow occurs, *pmax* or *nmax* (as appropriate) is stored in the corresponding element of parameter **d**. If an underflow occurs, +0 (for rounding modes RN, RZ, RP) or -0 (for rounding mode RM) is stored in the corresponding element of parameter **d**.

Exceptions:

If the contents of either element of parameter **a** or **b** are Infinity, Denorm, or NaN, $SPEFSCR_{FINV,FINVH}$ are set appropriately, and $SPEFSCR_{FGH,FXH,FG,FX}$ are cleared appropriately. If $SPEFSCR_{FINVE}$ is set, an interrupt is taken and parameter **d** is not updated. Otherwise, if an overflow occurs, $SPEFSCR_{FOVF,FOVFH}$ are set appropriately, or if an underflow occurs, $SPEFSCR_{FUNF,FUNFH}$ are set appropriately. If either underflow or overflow exceptions are enabled and a corresponding status bit is set, an interrupt is taken. If any of these interrupts are taken, parameter **d** is not updated.

If either result element of this instruction is inexact, or overflows but overflow exceptions are disabled, and no other interrupt is taken, or underflows but underflow exceptions are disabled, and no other interrupt is taken, $SPEFSCR_{FINXS,FINXSH}$ is set. If the floating-point inexact exception is enabled, an interrupt is taken using the floating-point round interrupt vector. In this case, parameter **d** is updated with the truncated result(s). The FG and FX bits are properly updated to allow rounding to be performed in the interrupt handler.

FG and FX (FGH and FXH) are cleared if an overflow or underflow interrupt is taken, or if an invalid operation/input error is signaled for the low (high) element (regardless of FINVE).

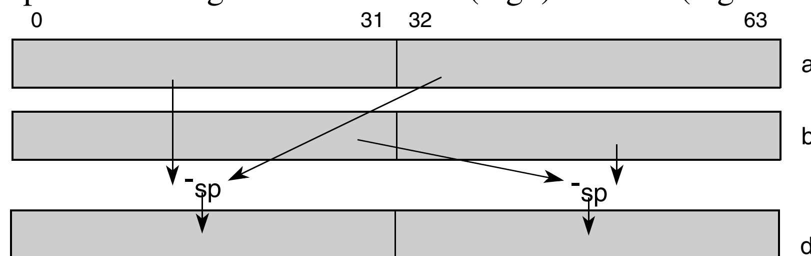


Figure 3-814. Vector Floating-Point Single-Precision Differences (__ev_fsdiff)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evfsdiff d,a,b**

__ev_fsdiffsum __ev_fsdiffsum

Vector Floating-Point Single-Precision Difference / Sum

d = __ev_fsdiffsum (a,b)

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

The low-order single-precision floating-point element of parameter **a** is subtracted from the high-order element of parameter **a**, the low-order single-precision floating-point element of parameter **b** is added to the high-order element of parameter **b**, and the results are stored in parameter **d**. If the high-order element of parameter **a** or **b** is NaN or infinity, the corresponding result is either *pmax* or *nmax* (as appropriate). Otherwise, if the low order element of parameter **a** or **b** is NaN or infinity, the corresponding result is either *pmax* or *nmax* (as appropriate). Otherwise, if an overflow occurs, *pmax* or *nmax* (as appropriate) is stored in the corresponding element of parameter **d**. If an underflow occurs, +0 (for rounding modes RN, RZ, RP) or -0 (for rounding mode RM) is stored in the corresponding element of parameter **d**.

Exceptions:

If the contents of either element of parameter **a** or **b** are Infinity, Denorm, or NaN, $SPEFSCR_{FINV,FINVH}$ are set appropriately, and $SPEFSCR_{FGH,FXH,FG,FX}$ are cleared appropriately. If $SPEFSCR_{FINVE}$ is set, an interrupt is taken and parameter **d** is not updated. Otherwise, if an overflow occurs, $SPEFSCR_{FOVF,FOVFH}$ are set appropriately, or if an underflow occurs, $SPEFSCR_{FUNF,FUNFH}$ are set appropriately. If either underflow or overflow exceptions are enabled and a corresponding status bit is set, an interrupt is taken. If any of these interrupts are taken, parameter **d** is not updated.

If either result element of this instruction is inexact, or overflows but overflow exceptions are disabled, and no other interrupt is taken, or underflows but underflow exceptions are disabled, and no other interrupt is taken, $SPEFSCR_{FINXS,FINXSH}$ is set. If the floating-point inexact exception is enabled, an interrupt is taken using the floating-point round interrupt vector. In this case, parameter **d** is updated with the truncated result(s). The FG and FX bits are properly updated to allow rounding to be performed in the interrupt handler.

FG and FX (FGH and FXH) are cleared if an overflow or underflow interrupt is taken, or if an invalid operation/input error is signaled for the low (high) element (regardless of FINVE).

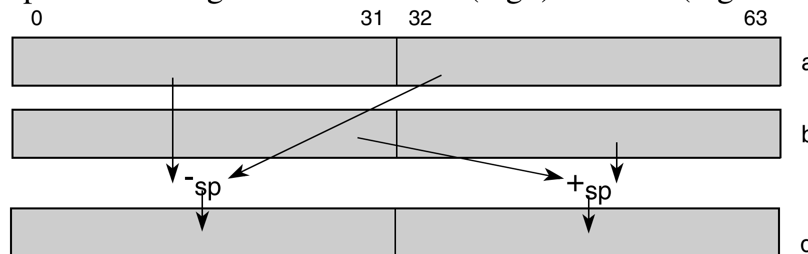


Figure 3-815. Vector Floating-Point Single-Precision Difference / Sum (__ev_fsdiffsum)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evfsdiffsum d,a,b**

__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

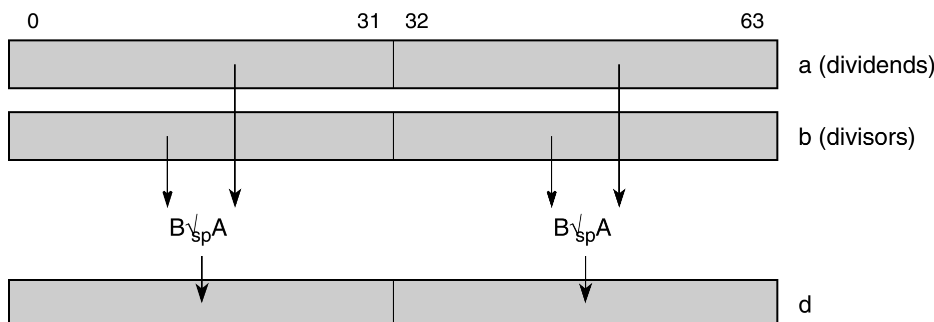


Figure 3-816. Vector Floating-Point Divide (__ev_fsdiv)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evfsdiv d,a,b**

__ev_fsmax

__ev_fsmax

Vector Floating-Point Single-Precision Maximum

d = __ev_fsmax (a,b)

```
ah ←a0:31
bh ←b0:31
if (ah < bh) then temph ←bh
else temph ←ah
d0:31 ←temph

al ←a32:63
bl ←b32:63
if (al < bl) then templ ←bl
else templ ←al
d32:63 ←templ
```

Each single-precison floating-point element of parameter **a** is compared against the corresponding elements of parameter **b**. The larger element is selected and placed into the corresponding element of parameter **d**. The maximum of +0 and -0 is +0.

Exceptions:

If the contents of either element of parameter **a** or **b** are Infinity, Denorm, or NaN, $SPEFSCR_{FINV,FINVH}$ are set appropriately, and $SPEFSCR_{FGH,FXH,FG,FX}$ are cleared appropriately. If $SPEFSCR_{FINVE}$ is set, an interrupt is taken, and parameter **d** is not updated. Otherwise, the comparison proceeds after treating NaNs, Infinities, and Denorms as normalized numbers, using their values of '*e*' and '*f*' directly. If the selected element is denorm, the result is a same signed zero. If the selected element is +NaN or +infinity, the corresponding result is *pmax*. Otherwise, if the selected element is -NaN or -infinity, the corresponding result is *nmax*.

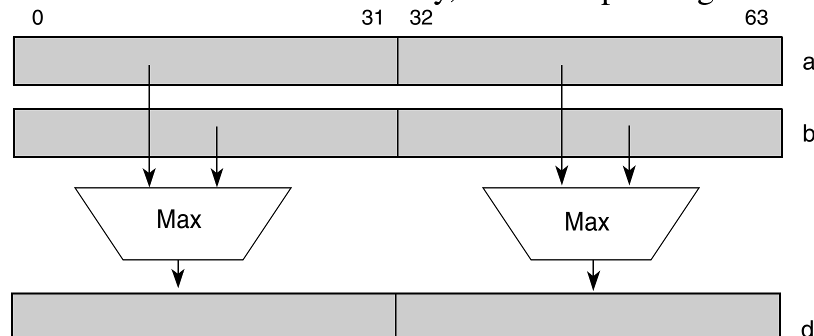


Figure 3-817. Vector Floating-Point Single-Precision Maximum (__ev_fsmax)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evfsmax d,a,b**

__ev_fsmin __ev_fsmin

Vector Floating-Point Single-Precision Minimum

d = __ev_fsmin (a,b)

```
ah ←a0:31
bh ←b0:31
if (ah < bh) then temph ←ah
else temph ←bh
d0:31 ←temph

al ←a32:63
bl ←b32:63
if (al < bl) then templ ←al
else templ ←bl
d32:63 ←templ
```

Each single-precison floating-point element of parameter **a** is compared against the corresponding element of parameter **b**. The smaller element is selected and placed into the corresponding element of parameter **d**. The minimum of +0 and -0 is -0.

Exceptions:

If the contents of either element of parameter **a** or **b** are Infinity, Denorm, or NaN, $SPEFSCR_{FINV,FINVH}$ are set appropriately, and $SPEFSCR_{FGH,FXH,FG,FX}$ are cleared appropriately. If $SPEFSCR_{FINVE}$ is set, an interrupt is taken, and parameter **d** is not updated. Otherwise, the comparison proceeds after treating NaNs, Infinities, and Denorms as normalized numbers, using their values of ‘*e*’ and ‘*f*’ directly. If the selected element is denorm, the result is a same signed zero. If the selected element is +NaN or +infinity, the corresponding result is *pmax*. Otherwise, if the selected element is -NaN or -infinity, the corresponding result is *nmax*.

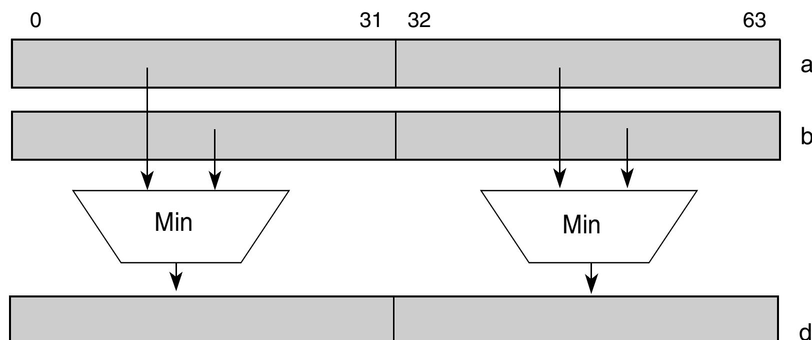


Figure 3-818. Vector Floating-Point Single-Precision Maximum (__ev_fsmin)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evfsmin 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.

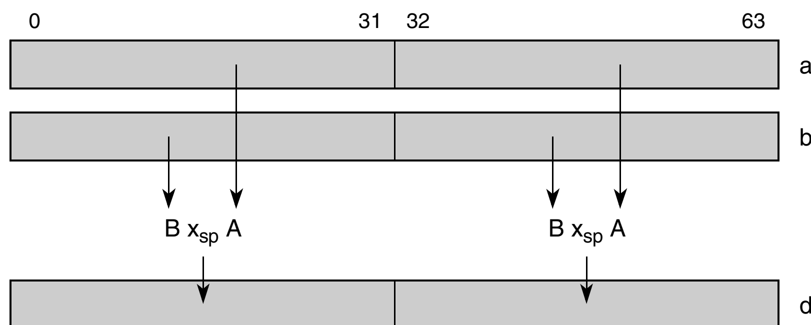


Figure 3-819. Vector Floating-Point Multiply (__ev_fsmul)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evfsmul d,a,b**

__ev_fsmule __ev_fsmule

Vector Floating-Point Single-Precision Multiply By Even Element

d = __ev_fsmule (a,b)

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

The single-precision floating-point elements of parameter **b** are multiplied by the even (high-order) element of parameter **a**, and the results are stored in parameter **d**. If an element of parameter **b** or the even element of parameter **a** is either zero (or a denormalized number optionally transformed to zero by the implementation), the corresponding result is a properly signed zero. Otherwise, if an element of parameter **b** or the even element of parameter **a** is either NaN or infinity, the corresponding result is either *pmax* (a_{sign}==b_{sign}), or *nmax* (a_{sign}!=b_{sign}). Otherwise, if an overflow occurs, *pmax* or *nmax* (as appropriate) is stored in the corresponding element of parameter **d**. If an underflow occurs, +0 or -0 (as appropriate) is stored in the corresponding element of parameter **d**.

Exceptions:

If the contents of either element of parameter **b** or the even element of parameter **a** is Infinity, Denorm, or NaN, $SPEFSCR_{FINV,FINVH}$ are set appropriately, and $SPEFSCR_{FGH,FXH,FG,FX}$ are cleared appropriately. If $SPEFSCR_{FINVE}$ is set, an interrupt is taken and parameter **d** is not updated. Otherwise, if an overflow occurs, $SPEFSCR_{FOVF,FOVFH}$ are set appropriately, or if an underflow occurs, $SPEFSCR_{FUNF,FUNFH}$ are set appropriately. If either underflow or overflow exceptions are enabled and a corresponding status bit is set, an interrupt is taken. If any of these interrupts are taken, parameter **d** is not updated.

If either result element of this instruction is inexact, or overflows but overflow exceptions are disabled, and no other interrupt is taken, or underflows but underflow exceptions are disabled, and no other interrupt is taken, $SPEFSCR_{FINXS}$ is set. If the floating-point inexact exception is enabled, an interrupt is taken using the floating-point round interrupt vector. In this case, parameter **d** is updated with the truncated result(s). The FG and FX bits are properly updated to allow rounding to be performed in the interrupt handler.

FG and FX (FGH and FXH) are cleared if an overflow or underflow exception is taken, or if an invalid operation/input error is signaled for the low (high) element (regardless of FINVE).

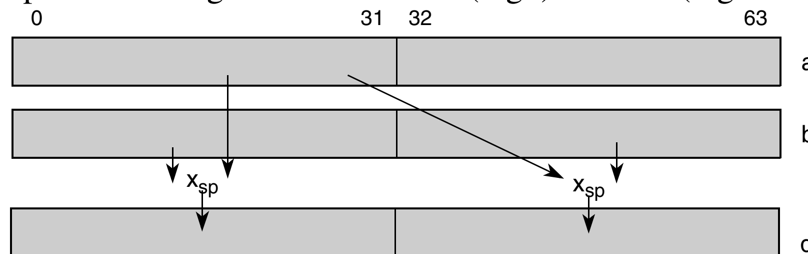


Figure 3-820. Vector Floating-Point Multiply By Even Element (__ev_fsmule)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evfsmule d,a,b**

__ev_fsmulo __ev_fsmulo

Vector Floating-Point Single-Precision Multiply By Odd Element

d = __ev_fsmulo (a,b)

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

The single-precision floating-point elements of parameter **b** are multiplied by the odd (low-order) element of parameter **a**, and the results are stored in parameter **d**. If an element of parameter **b** or the odd element of parameter **a** is either zero (or a denormalized number optionally transformed to zero by the implementation), the corresponding result is a properly signed zero. Otherwise, if an element of parameter **b** or the odd element of parameter **a** is either NaN or infinity, the corresponding result is either *pmax* (a_{sign}==b_{sign}), or *nmax* (a_{sign}!=b_{sign}). Otherwise, if an overflow occurs, *pmax* or *nmax* (as appropriate) is stored in the corresponding element of parameter **d**. If an underflow occurs, +0 or -0 (as appropriate) is stored in the corresponding element of parameter **d**.

Exceptions:

If the contents of either element of parameter **b** or the odd element of parameter **a** is Infinity, Denorm, or NaN, $SPEFSCR_{FINV,FINVH}$ are set appropriately, and $SPEFSCR_{FGH,FXH,FG,FX}$ are cleared appropriately. If $SPEFSCR_{FINVE}$ is set, an interrupt is taken and parameter **d** is not updated. Otherwise, if an overflow occurs, $SPEFSCR_{FOVF,FOVFH}$ are set appropriately, or if an underflow occurs, $SPEFSCR_{FUNF,FUNFH}$ are set appropriately. If either underflow or overflow exceptions are enabled and a corresponding status bit is set, an interrupt is taken. If any of these interrupts are taken, parameter **d** is not updated.

If either result element of this instruction is inexact, or overflows but overflow exceptions are disabled, and no other interrupt is taken, or underflows but underflow exceptions are disabled, and no other interrupt is taken, $SPEFSCR_{FINXS}$ is set. If the floating-point inexact exception is enabled, an interrupt is taken using the floating-point round interrupt vector. In this case, parameter **d** is updated with the truncated result(s). The FG and FX bits are properly updated to allow rounding to be performed in the interrupt handler.

FG and FX (FGH and FXH) are cleared if an overflow or underflow exception is taken, or if an invalid operation/input error is signaled for the low (high) element (regardless of FINVE).

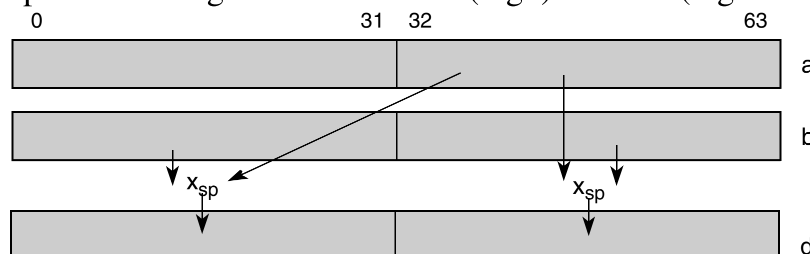


Figure 3-821. Vector Floating-Point Multiply By Even Element (__ev_fsmulo)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evfsmulo d,a,b**

__ev_fsmulx __ev_fsmulx

Vector Floating-Point Single-Precision Multiply Exchanged

d = __ev_fsmulx (a,b)

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

The high-order single-precision floating-point element of parameter **b** is multiplied by the low-order element of parameter **a**, the low-order single-precision floating-point element of parameter **b** is multiplied by the high-order element of parameter **a**, and the results are stored in parameter **d**. If an element of parameter **a** or **b** is either zero (or a denormalized number optionally transformed to zero by the implementation), the corresponding result is a properly signed zero. Otherwise, if an element of parameter **a** or **b** is either NaN or infinity, the corresponding result is either *pmax* (a_{sign}==b_{sign}), or *nmax* (a_{sign}!=b_{sign}). Otherwise, if an overflow occurs, *pmax* or *nmax* (as appropriate) is stored in the corresponding element of parameter **d**. If an underflow occurs, +0 or -0 (as appropriate) is stored in the corresponding element of parameter **d**.

Exceptions:

If the contents of either element of parameter **a** or **b** are Infinity, Denorm, or NaN, $SPEFSCR_{FINV,FINVH}$ are set appropriately, and $SPEFSCR_{FGH,FXH,FG,FX}$ are cleared appropriately. If $SPEFSCR_{FINVE}$ is set, an interrupt is taken and parameter **d** is not updated. Otherwise, if an overflow occurs, $SPEFSCR_{FOVF,FOVFH}$ are set appropriately, or if an underflow occurs, $SPEFSCR_{FUNF,FUNFH}$ are set appropriately. If either underflow or overflow exceptions are enabled and a corresponding status bit is set, an interrupt is taken. If any of these interrupts are taken, parameter **d** is not updated.

If either result element of this instruction is inexact, or overflows but overflow exceptions are disabled, and no other interrupt is taken, or underflows but underflow exceptions are disabled, and no other interrupt is taken, $SPEFSCR_{FINXS}$ is set. If the floating-point inexact exception is enabled, an interrupt is taken using the floating-point round interrupt vector. In this case, parameter **d** is updated with the truncated result(s). The FG and FX bits are properly updated to allow rounding to be performed in the interrupt handler.

FG and FX (FGH and FXH) are cleared if an overflow or underflow exception is taken, or if an invalid operation/input error is signaled for the low (high) element (regardless of FINVE).

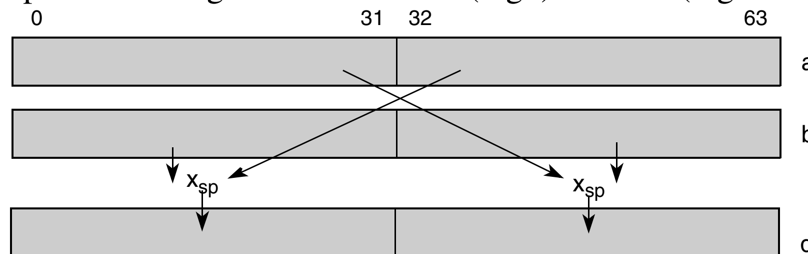


Figure 3-822. Vector Floating-Point Multiply By Even Element (__ev_fsmulo)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evfsmulx d,a,b**

__ev_fsnabs __ev_fsnabs

Vector Floating-Point Negative Absolute Value

d = __ev_fsnabs (a)

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

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.

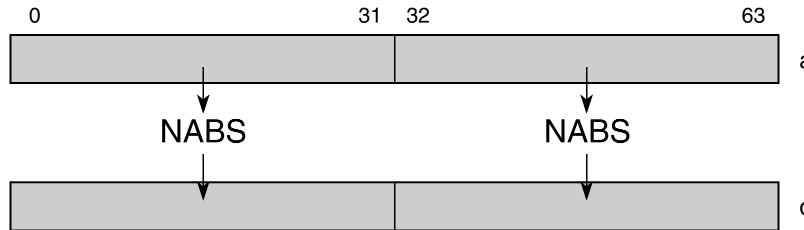


Figure 3-823. Vector Floating-Point Negative Absolute Value (__ev_fsnabs)

d	a	Maps to
__ev64_opaque__	__ev64_opaque__	**evfsnabs 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.

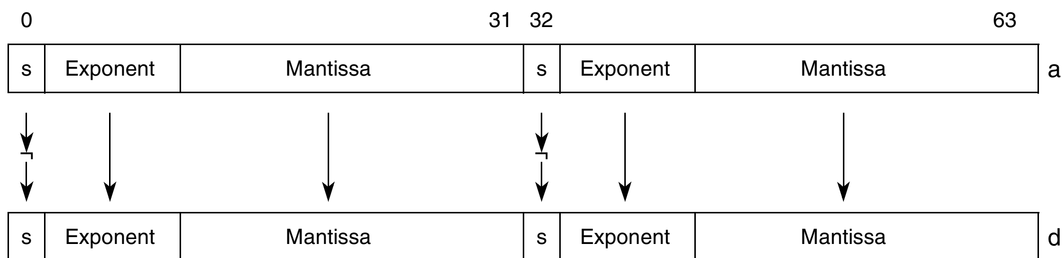


Figure 3-824. Vector Floating-Point Negate (__ev_fsneg)

d	a	Maps to
__ev64_opaque__	__ev64_opaque__	**evfsneg d,a**

__ev_fssqrt

__ev_fssqrt

Vector Floating-Point Single-Precision Square Root

d = __ev_fssqrt (a)

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

The square root of each single-precision floating-point element of parameter **a** is calculated, and the results are stored in parameter **d**. If an element of parameter **a** is zero or denorm, the result is a same signed zero. If an element of parameter **a** is +NaN or +infinity, the corresponding result is *pmax*. Otherwise, if an element of parameter **a** is non-zero and has a negative sign, including -NaN or -infinity, the corresponding result is TBD. Otherwise, if an underflow occurs, +0 (for rounding modes RN, RZ, RP) or -0 (for rounding mode RM) is stored in the corresponding element of parameter **d**.

Exceptions:

If the contents of either element of parameter **a** are non-zero and have a negative sign, or are Infinity, Denorm, or NaN, $SPEFSCR_{FINV,FINVH}$ are set appropriately, and $SPEFSCR_{FGH,FXH,FG,FX}$ are cleared appropriately. If $SPEFSCR_{FINVE}$ is set, an interrupt is taken and parameter **d** is not updated. Otherwise, if an underflow occurs, $SPEFSCR_{FUNF,FUNFH}$ are set appropriately. If underflow exceptions are enabled and a corresponding status bit is set, an interrupt is taken. If any of these interrupts are taken, parameter **d** is not updated.

If either result element of this instruction is inexact, or underflows but underflow exceptions are disabled, and no other interrupt is taken, $SPEFSCR_{FINXS,FINXSH}$ is set. If the floating-point inexact exception is enabled, an interrupt is taken using the floating-point round interrupt vector. In this case, parameter **d** is updated with the truncated result(s). The FG and FX bits are properly updated to allow rounding to be performed in the interrupt handler.

FG and FX (FGH and FXH) are cleared if an underflow interrupt is taken, or if an invalid operation/input error is signaled for the low (high) element (regardless of FINVE).

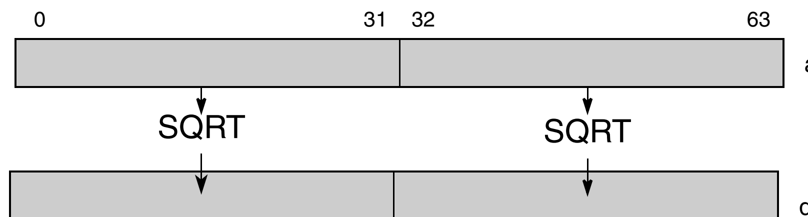


Figure 3-825. Vector Floating-Point Multiply By Even Element (__ev_fssqrt)

d	a	Maps to
__ev64_opaque__	__ev64_opaque__	**evfssqrt 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.

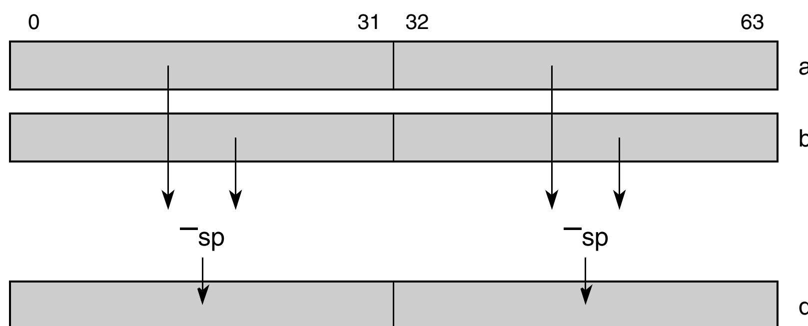


Figure 3-826. Vector Floating-Point Subtract (__ev_fssub)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evfssub d,a,b**

__ev_fssubadd __ev_fssubadd

Vector Floating-Point Single-Precision Subtract / Add

d = __ev_fssubadd (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 high-order floating-point element of parameter **b** is subtracted from the corresponding element of parameter **a**, the low-order floating-point element of parameter **b** is subtracted from the corresponding element of parameter **a**, and the results are stored in parameter **d**. If an element of parameter **a** is NaN or infinity, the corresponding result is either *pmax* or *nmax* (as appropriate). Otherwise, if an element of parameter **b** is NaN or infinity, the corresponding result is either *nmax* or *pmax* (as appropriate). Otherwise, if an overflow occurs, *pmax* or *nmax* (as appropriate) is stored in the corresponding element of parameter **d**. If an underflow occurs, +0 (for rounding modes RN, RZ, RP) or -0 (for rounding mode RM) is stored in the corresponding element of parameter **d**.

Exceptions:

If the contents of either element of parameter **a** or **b** are Infinity, Denorm, or NaN, $SPEFSCR_{FINV,FINVH}$ are set appropriately, and $SPEFSCR_{FGH,FXH,FG,FX}$ are cleared appropriately. If $SPEFSCR_{FINVE}$ is set, an interrupt is taken and parameter **d** is not updated. Otherwise, if an overflow occurs, $SPEFSCR_{FOVF,FOVFH}$ are set appropriately, or if an underflow occurs, $SPEFSCR_{FUNF,FUNFH}$ are set appropriately. If either underflow or overflow exceptions are enabled and a corresponding status bit is set, an interrupt is taken. If any of these interrupts are taken, parameter **d** is not updated.

If either result element of this instruction is inexact, or overflows but overflow exceptions are disabled, and no other interrupt is taken, or underflows but underflow exceptions are disabled, and no other interrupt is taken, $SPEFSCR_{FINXS}$ is set. If the floating-point inexact exception is enabled, an interrupt is taken using the floating-point round interrupt vector. In this case, parameter **d** is updated with the truncated result(s). The FG and FX bits are properly updated to allow rounding to be performed in the interrupt handler.

FG and FX (FGH and FXH) are cleared if an overflow or underflow interrupt is taken, or if an invalid operation/input error is signaled for the low (high) element (regardless of FINVE).

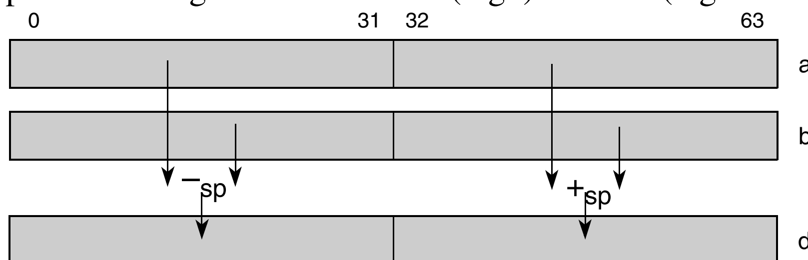


Figure 3-827. Vector Floating-Point Subtract / Add(__ev_fssubadd)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evfssubadd d,a,b**

__ev_fssubaddx __ev_fssubaddx

Vector Floating-Point Single-Precision Subtract / Add Exchanged

d = __ev_fssubaddx (a,b)

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

The high-order floating-point element of parameter **b** is subtracted from the low-order element of parameter **a**, the low-order floating-point element of parameter **b** is added to the high-order from the corresponding element of parameter **a**, and the results are stored in parameter **d**. If an element of parameter **a** is NaN or infinity, the corresponding result is either *pmax* or *nmax* (as appropriate). Otherwise, if an element of parameter **b** is NaN or infinity, the corresponding result is either *nmax* or *pmax* (as appropriate). Otherwise, if an overflow occurs, *pmax* or *nmax* (as appropriate) is stored in the corresponding element of parameter **d**. If an underflow occurs, +0 (for rounding modes RN, RZ, RP) or -0 (for rounding mode RM) is stored in the corresponding element of parameter **d**.

Exceptions:

If the contents of either element of parameter **a** or **b** are Infinity, Denorm, or NaN, $SPEFSCR_{FINV,FINVH}$ are set appropriately, and $SPEFSCR_{FGH,FXH,FG,FX}$ are cleared appropriately. If $SPEFSCR_{FINVE}$ is set, an interrupt is taken and parameter **d** is not updated. Otherwise, if an overflow occurs, $SPEFSCR_{FOVF,FOVFH}$ are set appropriately, or if an underflow occurs, $SPEFSCR_{FUNF,FUNFH}$ are set appropriately. If either underflow or overflow exceptions are enabled and a corresponding status bit is set, an interrupt is taken. If any of these interrupts are taken, parameter **d** is not updated.

If either result element of this instruction is inexact, or overflows but overflow exceptions are disabled, and no other interrupt is taken, or underflows but underflow exceptions are disabled, and no other interrupt is taken, $SPEFSCR_{FINXS}$ is set. If the floating-point inexact exception is enabled, an interrupt is taken using the floating-point round interrupt vector. In this case, parameter **d** is updated with the truncated result(s). The FG and FX bits are properly updated to allow rounding to be performed in the interrupt handler.

FG and FX (FGH and FXH) are cleared if an overflow or underflow interrupt is taken, or if an invalid operation/input error is signaled for the low (high) element (regardless of FINVE).

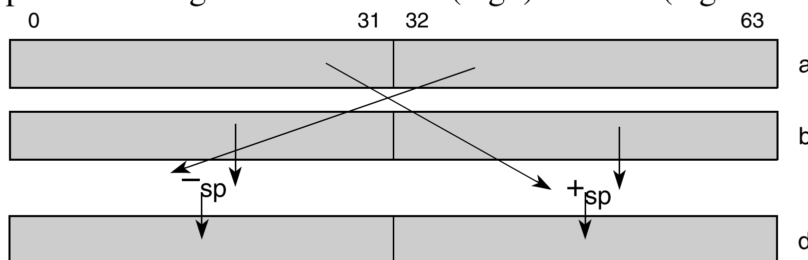


Figure 3-828. Vector Floating-Point Subtract / Add Exchanged (__ev_fssubaddx)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evfssubaddx d,a,b**

__ev_fssubx __ev_fssubx

Vector Floating-Point Single-Precision Subtract Exchanged

d = __ev_fssubx (a,b)

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

The high-order floating-point element of parameter **b** is subtracted from the low-order element of parameter **a**, the low-order floating-point element of parameter **b** is subtracted from the high-order element of parameter **a**, and the results are stored in parameter **d**. If an element of parameter **a** is NaN or infinity, the corresponding result is either *pmax* or *nmax* (as appropriate). Otherwise, if an element of parameter **b** is NaN or infinity, the corresponding result is either *nmax* or *pmax* (as appropriate). Otherwise, if an overflow occurs, *pmax* or *nmax* (as appropriate) is stored in the corresponding element of parameter **d**. If an underflow occurs, +0 (for rounding modes RN, RZ, RP) or -0 (for rounding mode RM) is stored in the corresponding element of parameter **d**.

Exceptions:

If the contents of either element of parameter **a** or **b** are Infinity, Denorm, or NaN, $SPEFSCR_{FINV,FINVH}$ are set appropriately, and $SPEFSCR_{FGH,FXH,FG,FX}$ are cleared appropriately. If $SPEFSCR_{FINVE}$ is set, an interrupt is taken and parameter **d** is not updated. Otherwise, if an overflow occurs, $SPEFSCR_{FOVF,FOVFH}$ are set appropriately, or if an underflow occurs, $SPEFSCR_{FUNF,FUNFH}$ are set appropriately. If either underflow or overflow exceptions are enabled and a corresponding status bit is set, an interrupt is taken. If any of these interrupts are taken, parameter **d** is not updated.

If either result element of this instruction is inexact, or overflows but overflow exceptions are disabled, and no other interrupt is taken, or underflows but underflow exceptions are disabled, and no other interrupt is taken, $SPEFSCR_{FINXS}$ is set. If the floating-point inexact exception is enabled, an interrupt is taken using the floating-point round interrupt vector. In this case, parameter **d** is updated with the truncated result(s). The FG and FX bits are properly updated to allow rounding to be performed in the interrupt handler.

FG and FX (FGH and FXH) are cleared if an overflow or underflow interrupt is taken, or if an invalid operation/input error is signaled for the low (high) element (regardless of FINVE).

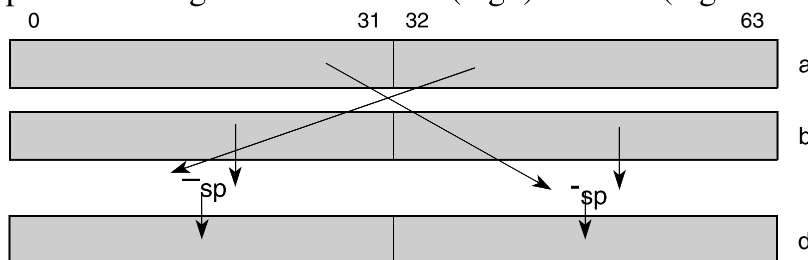


Figure 3-829. Vector Floating-Point Subtract Exchanged (__ev_fssubx)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evfssubx d,a,b**

__ev_fssum __ev_fssum

Vector Floating-Point Single-Precision Sums

d = __ev_fssum (a,b)

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

The high-order floating-point element of parameter **a** is added to the low-order element of parameter **a**, the high-order floating-point element of parameter **b** is added to the low-order element of parameter **b**, and the results are stored in parameter **d**. If the high-order element of parameter **a** or **b** is NaN or infinity, the corresponding result is either *pmax* or *nmax* (as appropriate). Otherwise, if the low order element of parameter **a** or **b** is NaN or infinity, the corresponding result is either *pmax* or *nmax* (as appropriate). Otherwise, if an overflow occurs, *pmax* or *nmax* (as appropriate) is stored in the corresponding element of parameter **d**. If an underflow occurs, +0 (for rounding modes RN, RZ, RP) or -0 (for rounding mode RM) is stored in the corresponding element of parameter **d**.

Exceptions:

If the contents of either element of parameter **a** or **b** are Infinity, Denorm, or NaN, $SPEFSCR_{FINV,FINVH}$ are set appropriately, and $SPEFSCR_{FGH,FXH,FG,FX}$ are cleared appropriately. If $SPEFSCR_{FINVE}$ is set, an interrupt is taken and parameter **d** is not updated. Otherwise, if an overflow occurs, $SPEFSCR_{FOVF,FOVFH}$ are set appropriately, or if an underflow occurs, $SPEFSCR_{FUNF,FUNFH}$ are set appropriately. If either underflow or overflow exceptions are enabled and a corresponding status bit is set, an interrupt is taken. If any of these interrupts are taken, parameter **d** is not updated.

If either result element of this instruction is inexact, or overflows but overflow exceptions are disabled, and no other interrupt is taken, or underflows but underflow exceptions are disabled, and no other interrupt is taken, $SPEFSCR_{FINXS,FINXSH}$ is set. If the floating-point inexact exception is enabled, an interrupt is taken using the floating-point round interrupt vector. In this case, parameter **d** is updated with the truncated result(s). The FG and FX bits are properly updated to allow rounding to be performed in the interrupt handler.

FG and FX (FGH and FXH) are cleared if an overflow or underflow interrupt is taken, or if an invalid operation/input error is signaled for the low (high) element (regardless of FINVE).

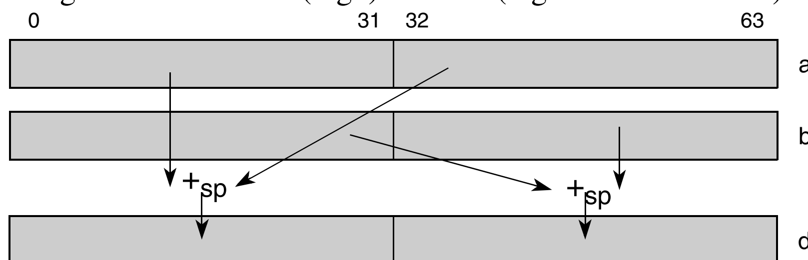


Figure 3-830. Vector Floating-Point Sums (__ev_fssum)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evfssum d,a,b**

__ev_fssumdiff __ev_fssumdiff

Vector Floating-Point Single-Precision Sum / Difference

d = __ev_fssumdiff (a,b)

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

The high-order floating-point element of parameter **a** is added to the low-order element of parameter **a**, the low-order floating-point element of parameter **b** is subtracted from the high-order element of parameter **b**, and the results are stored in parameter **d**. If the high-order element of parameter **a** or **b** is NaN or infinity, the corresponding result is either *pmax* or *nmax* (as appropriate). Otherwise, if the low order element of parameter **a** or **b** is NaN or infinity, the corresponding result is either *pmax* or *nmax* (as appropriate). Otherwise, if an overflow occurs, *pmax* or *nmax* (as appropriate) is stored in the corresponding element of parameter **d**. If an underflow occurs, +0 (for rounding modes RN, RZ, RP) or -0 (for rounding mode RM) is stored in the corresponding element of parameter **d**.

Exceptions:

If the contents of either element of parameter **a** or **b** are Infinity, Denorm, or NaN, $SPEFSCR_{FINV,FINVH}$ are set appropriately, and $SPEFSCR_{FGH,FXH,FG,FX}$ are cleared appropriately. If $SPEFSCR_{FINVE}$ is set, an interrupt is taken and parameter **d** is not updated. Otherwise, if an overflow occurs, $SPEFSCR_{FOVF,FOVFH}$ are set appropriately, or if an underflow occurs, $SPEFSCR_{FUNF,FUNFH}$ are set appropriately. If either underflow or overflow exceptions are enabled and a corresponding status bit is set, an interrupt is taken. If any of these interrupts are taken, parameter **d** is not updated.

If either result element of this instruction is inexact, or overflows but overflow exceptions are disabled, and no other interrupt is taken, or underflows but underflow exceptions are disabled, and no other interrupt is taken, $SPEFSCR_{FINXS,FINXSH}$ is set. If the floating-point inexact exception is enabled, an interrupt is taken using the floating-point round interrupt vector. In this case, parameter **d** is updated with the truncated result(s). The FG and FX bits are properly updated to allow rounding to be performed in the interrupt handler.

FG and FX (FGH and FXH) are cleared if an overflow or underflow interrupt is taken, or if an invalid operation/input error is signaled for the low (high) element (regardless of FINVE).

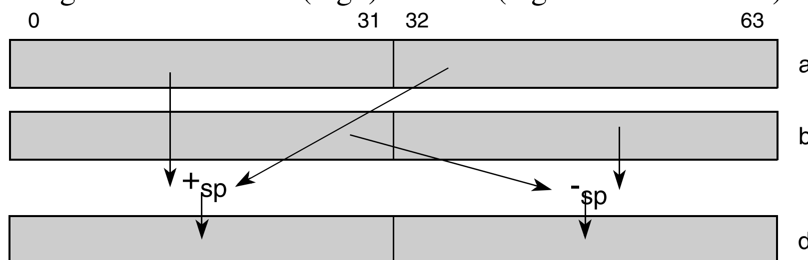


Figure 3-831. Vector Floating-Point Sum / Difference (__ev_fssumdiff)

d	a	b	Maps to
__ev64_opaque__	__ev64_opaque__	__ev64_opaque__	**evfssumdiff d,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**.

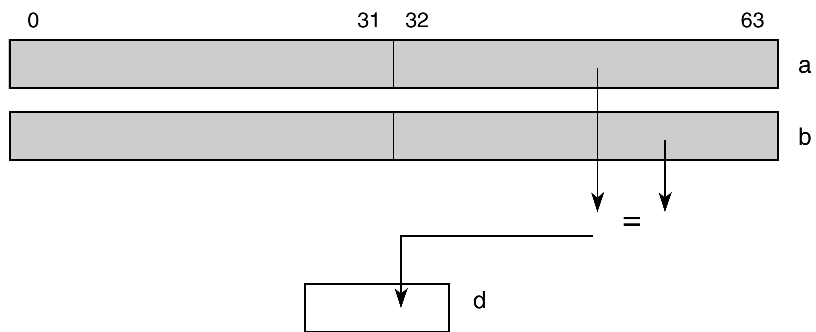


Figure 3-832. Vector Lower Floating-Point Equal (__ev_lower_fs_eq)

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**.

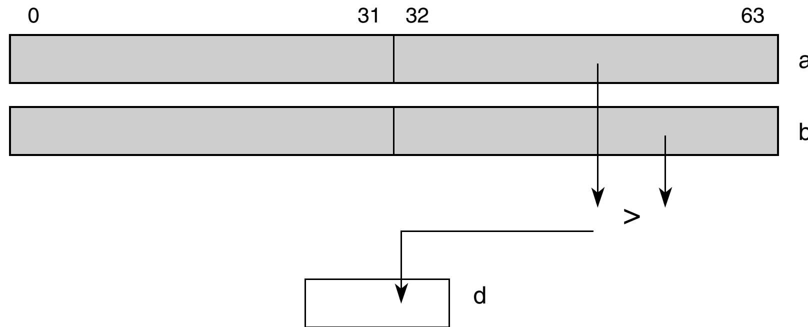


Figure 3-833. Vector Lower Floating-Point Greater Than (__ev_lower_fs_gt)

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**.

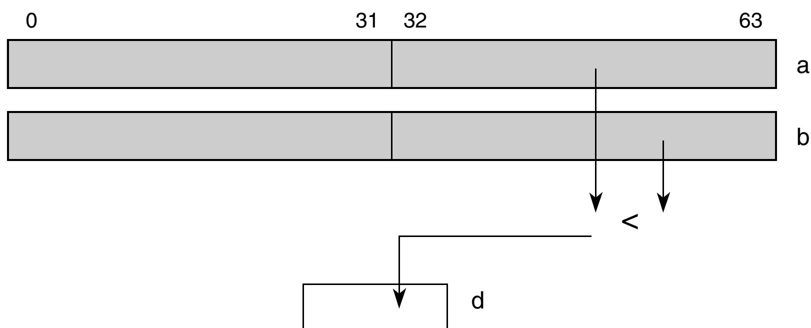


Figure 3-834. Vector Lower Floating-Point Less Than (__ev_lower_fs_lt)

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.

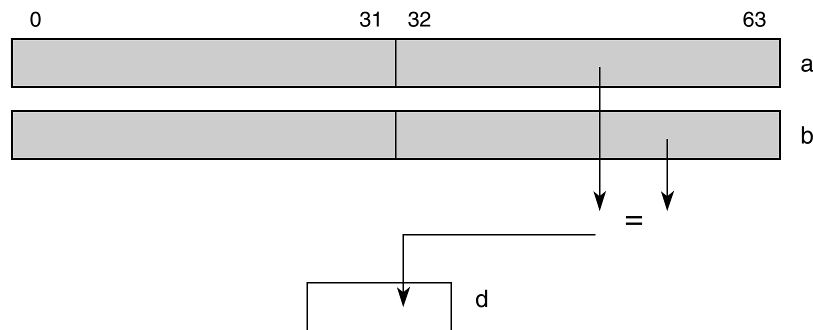


Figure 3-835. Vector Lower Floating-Point Test Equal (__ev_lower_fs_tst_eq)

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.

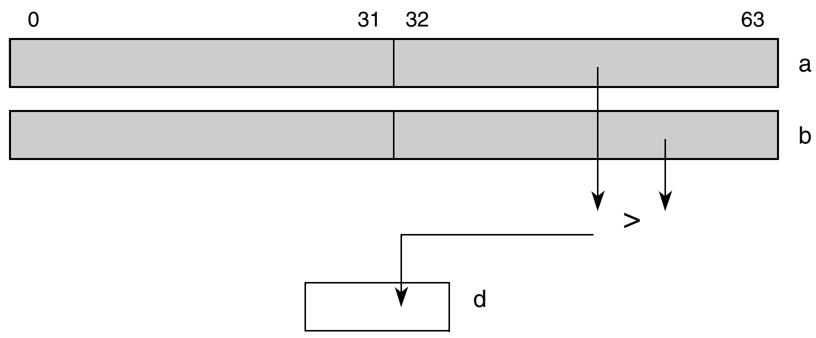


Figure 3-836. Vector Lower Floating-Point Test Greater Than (__ev_lower_fs_tst_gt)

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.

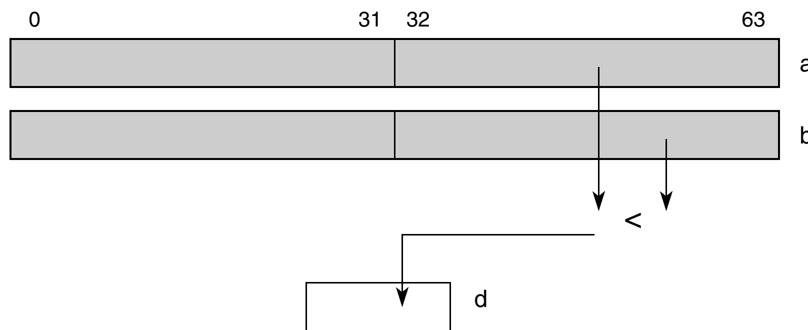


Figure 3-837. Vector Lower Floating-Point Test Less Than (__ev_lower_fs_tst_lt)

d	a	b	Maps to
_Bool	__ev64_opaque__	__ev64_opaque__	**evfststlt x,a,b**

__ev_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.

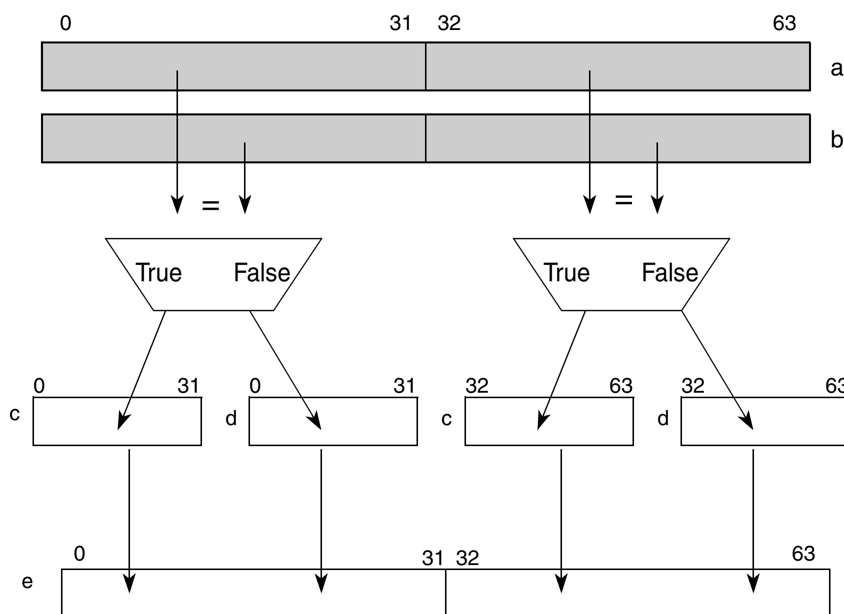


Figure 3-838. Vector Select Floating-Point Equal (__ev_select_fs_eq)

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.

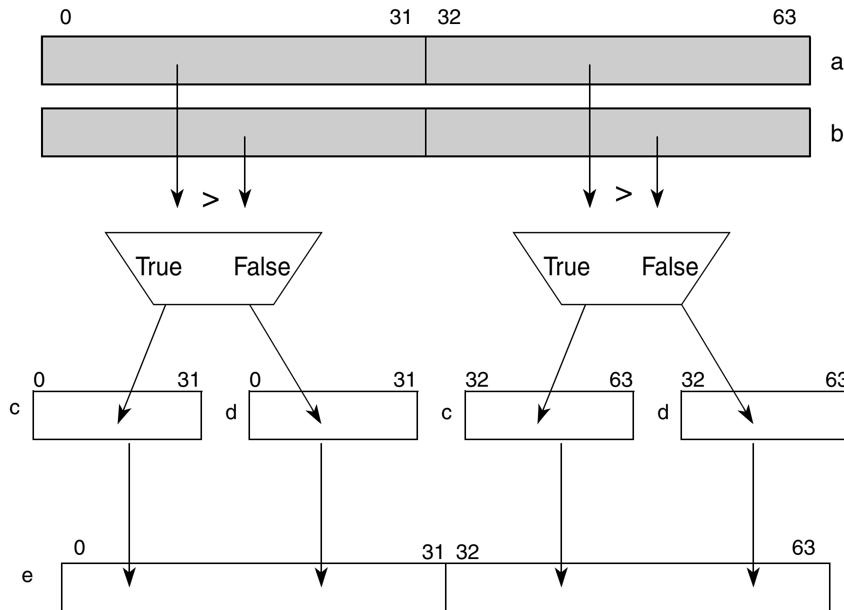


Figure 3-839. Vector Select Floating-Point Greater Than (__ev_select_fs_gt)

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.

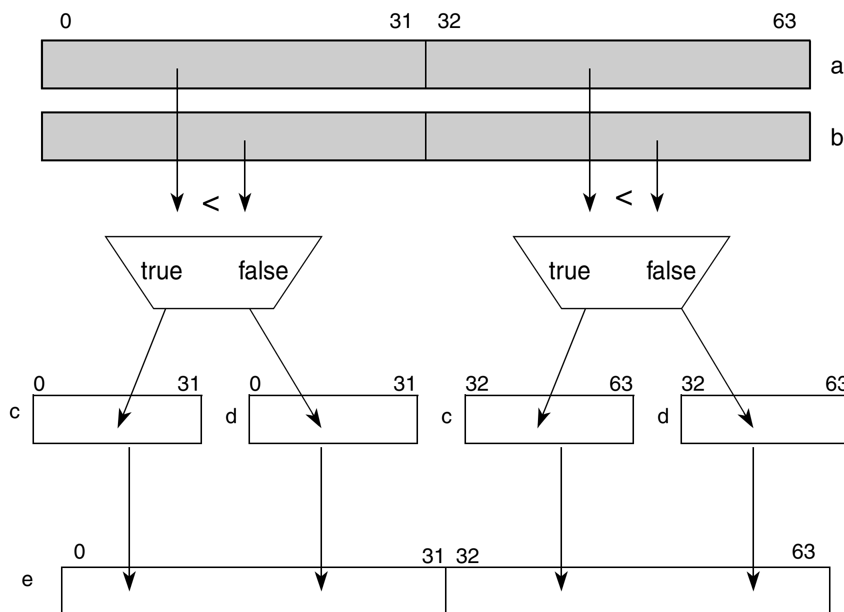


Figure 3-840. Vector Select Floating-Point Less Than (__ev_select_fs_lt)

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)

$$\text{if } (a_{0:31} = b_{0:31}) \text{ then } e_{0:31} \leftarrow c_{0:31}$$
$$\text{else } e_{0:31} \leftarrow d_{0:31}$$

$$\text{if } (a_{32:63} = b_{32:63}) \text{ then } e_{32:63} \leftarrow c_{32:63}$$
$$\text{else } e_{32:63} \leftarrow d_{32:63}$$

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.

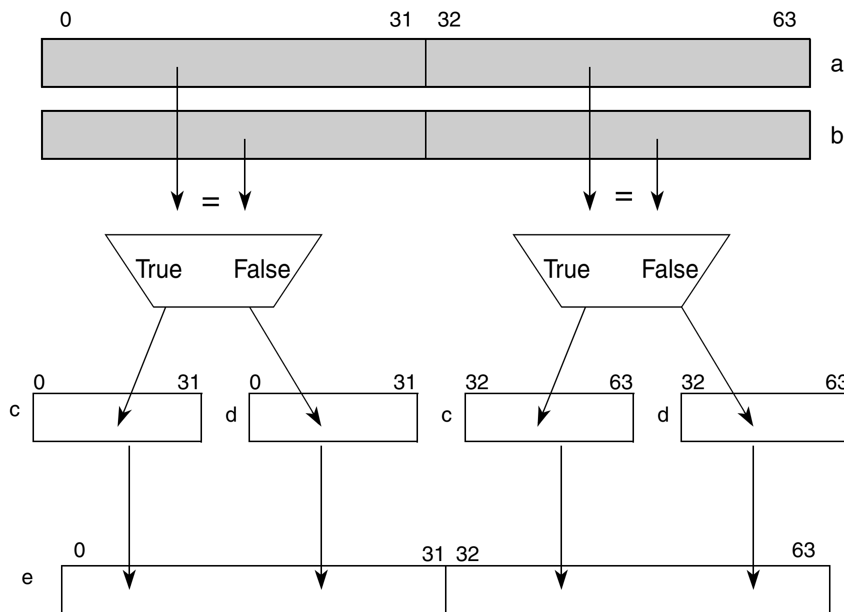


Figure 3-841. Vector Select Floating-Point Test Equal (__ev_select_fs_tst_eq)

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.

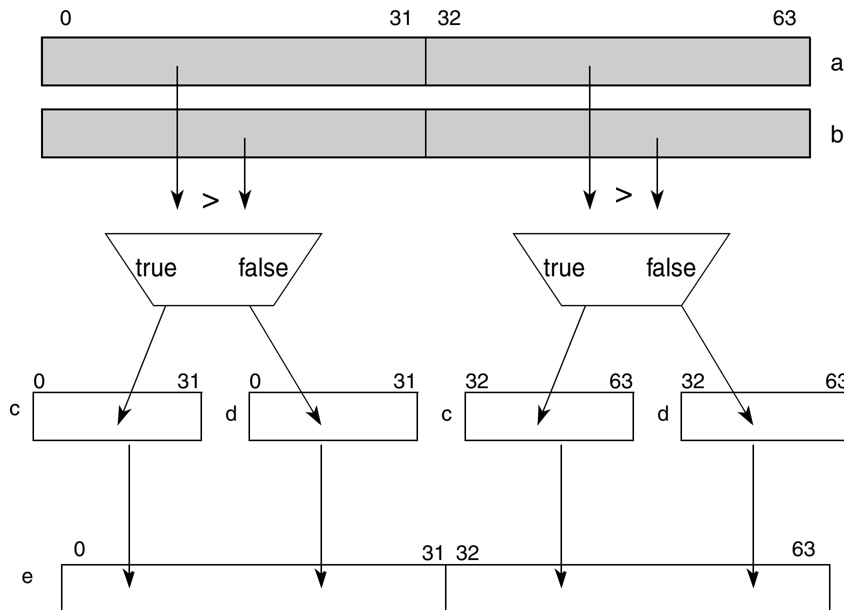


Figure 3-842. Vector Select Floating-Point Test Greater Than (__ev_select_fs_tst_gt)

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.

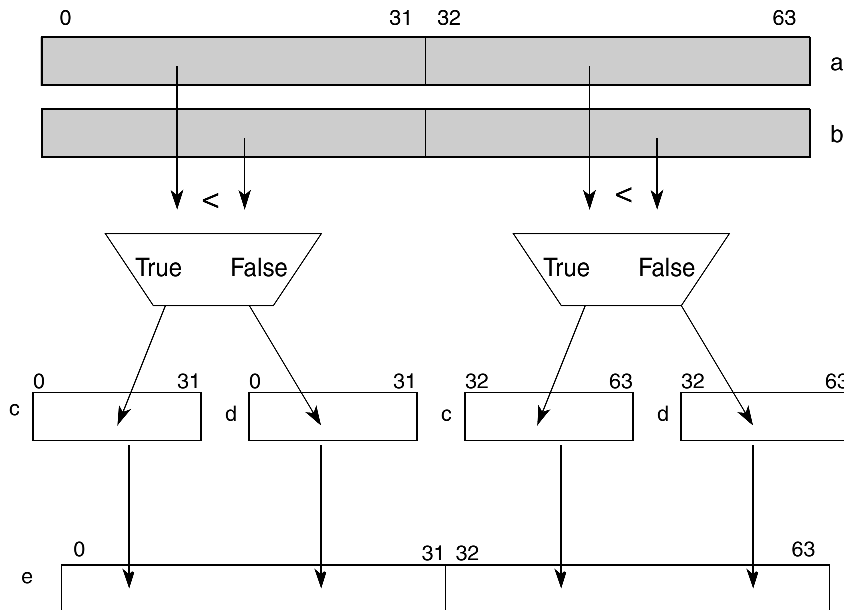


Figure 3-843. Vector Select Floating-Point Test Less Than (__ev_select_fs_tst_lt)

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_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**.

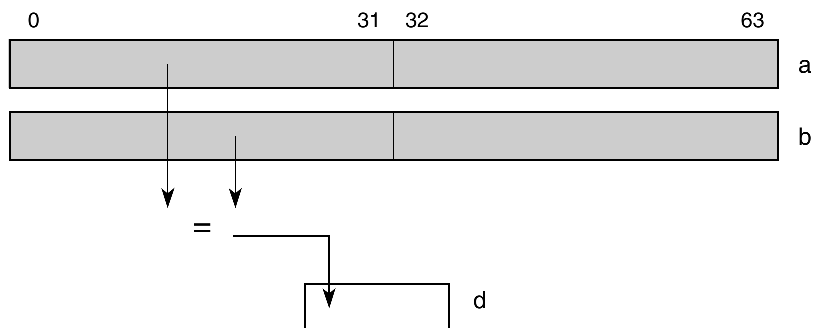


Figure 3-844. Vector Upper Floating-Point Equal(__ev_upper_fs_eq)

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**.

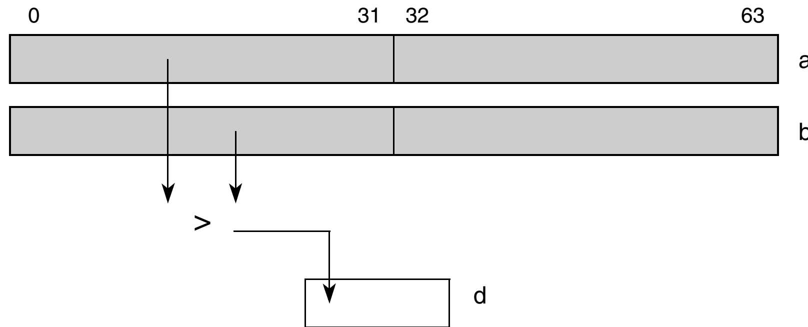


Figure 3-845. Vector Upper Floating-Point Greater Than (__ev_upper_fs_gt)

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**.

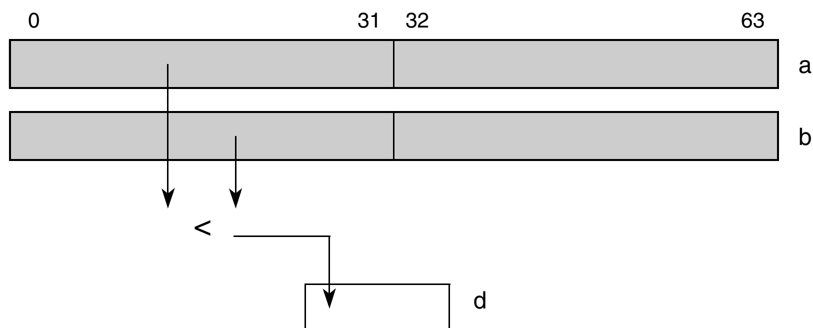


Figure 3-846. Vector Upper Floating-Point Less Than (__ev_upper_fs_lt)

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.

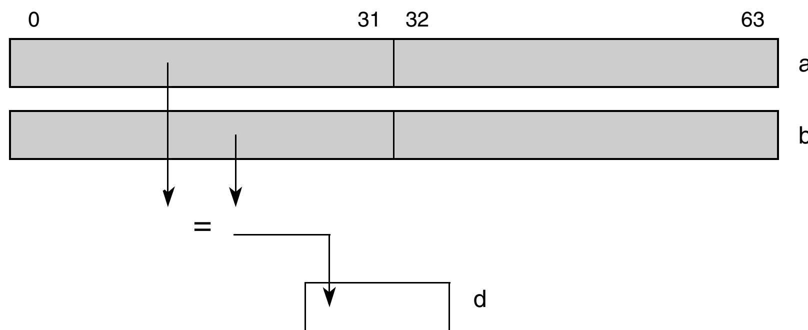


Figure 3-847. Vector Upper Floating-Point Test Equal (__ev_upper_fs_tst_eq)

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.

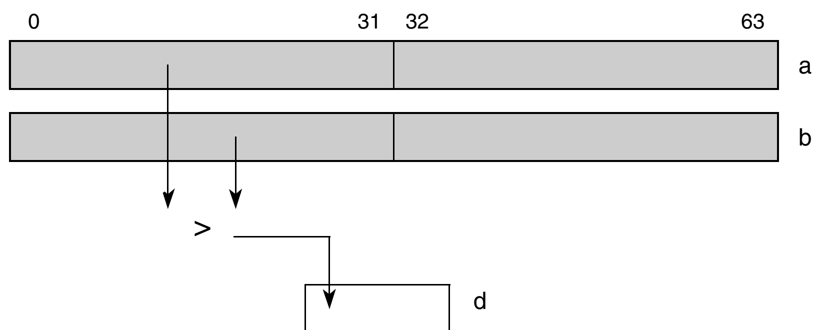


Figure 3-848. Vector Upper Floating-Point Test Greater Than (__ev_upper_fs_tst_gt)

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.

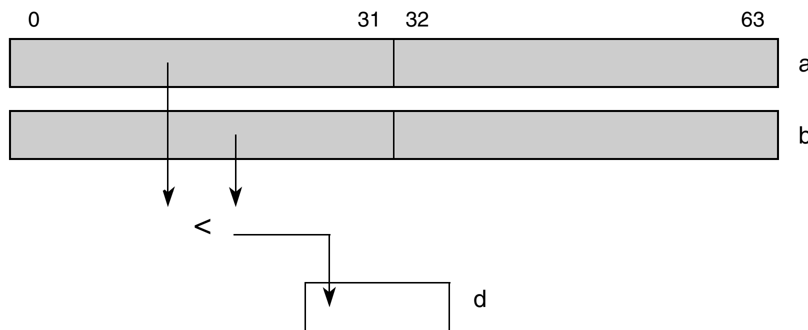


Figure 3-849. Vector Upper Floating-Point Test Less Than (__ev_upper_fs_tst_lt)

d	a	b	Maps to
_Bool	__ev64_opaque__	__ev64_opaque__	**evfststlt x,a,b**

3.8 Basic Instruction Mapping

```
//
uint32_t __brinc( uint32_t a, uint32_t b );
__ev64_opaque__ __ev_circinc( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_abs( __ev64_opaque__ a );
__ev64_opaque__ __ev_absb( __ev64_opaque__ a );
__ev64_opaque__ __ev_absbs( __ev64_opaque__ a );
__ev64_opaque__ __ev_absd( __ev64_opaque__ a );
__ev64_opaque__ __ev_absdifsb( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_absdifsh( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_absdifsw( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_absdifub( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_absdifuh( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_absdifuw( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_absds( __ev64_opaque__ a );
__ev64_opaque__ __ev_absh( __ev64_opaque__ a );
__ev64_opaque__ __ev_abshs( __ev64_opaque__ a );
__ev64_opaque__ __ev_abss( __ev64_opaque__ a );
```

```
__ev64_opaque__ __ev_add2subf2h( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_add2subf2hss( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_addsmiaa( __ev64_opaque__ a );
__ev64_opaque__ __ev_addb( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_addbss( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_addbus( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_addd( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_adddss( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_adddus( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_addh( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_addhhisw( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_addhhiuw( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_addhlosw( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_addhlouw( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_addhss( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_addhus( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_addhx( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_addhxss( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_addhxus( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_addib( __ev64_opaque__ a, 5-bit unsigned literal );
__ev64_opaque__ __ev_addih( __ev64_opaque__ a, 5-bit unsigned literal );
__ev64_opaque__ __ev_addiw( __ev64_opaque__ a, 5-bit unsigned literal );
__ev64_opaque__ __ev_addsmiaaw( __ev64_opaque__ a );
__ev64_opaque__ __ev_addssiaa( __ev64_opaque__ a );
__ev64_opaque__ __ev_addssiaaw( __ev64_opaque__ a );
__ev64_opaque__ __ev_addsubfh( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_addsubfhss( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_addsubfhx( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_addsubfhxss( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_addsubfw( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_addsubfwss( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_addsubfwx( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_addsubfwxss( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_addumiaaw( __ev64_opaque__ a );
__ev64_opaque__ __ev_addusiaa( __ev64_opaque__ a );
__ev64_opaque__ __ev_addusiaaw( __ev64_opaque__ a );
__ev64_opaque__ __ev_addw( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_addwegsf( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_addwegsi( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_addwogsf( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_addwogsi( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_addwss( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_addwus( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_addwx( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_addwxss( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_addwxus( __ev64_opaque__ a, __ev64_opaque__ b );

__ev64_opaque__ __ev_and( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_andc( __ev64_opaque__ a, __ev64_opaque__ b );

__ev64_opaque__ __ev_avgbs( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_avgbsr( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_avgbu( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_avgbur( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_avgds( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_avgdsr( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_avgdu( __ev64_opaque__ a, __ev64_opaque__ b );
```

```
__ev64_opaque__ __ev_avgdur( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_avghs( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_avghsr( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_avghu( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_avghur( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_avgws( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_avgwsr( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_avgwu( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_avgwur( __ev64_opaque__ a, __ev64_opaque__ b );

__ev64_opaque__ __ev_clrbe( __ev64_opaque__ a, 4-bit unsigned literal );
__ev64_opaque__ __ev_clrbo( __ev64_opaque__ a, 4-bit unsigned literal );
__ev64_opaque__ __ev_clrh( __ev64_opaque__ a, 4-bit unsigned literal );

__ev64_opaque__ __ev_cntlsh( __ev64_opaque__ a );
__ev64_opaque__ __ev_cntlsw( __ev64_opaque__ a );
__ev64_opaque__ __ev_cntlzh( __ev64_opaque__ a );
__ev64_opaque__ __ev_cntlzw( __ev64_opaque__ a );

__ev64_opaque__ __ev_diff2his( __ev64_opaque__ a );
__ev64_opaque__ __ev_diff2hisa( __ev64_opaque__ a );
__ev64_opaque__ __ev_diff2hisaaw( __ev64_opaque__ a );

__ev64_opaque__ __ev_divs( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_divu( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_divws( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_divwsf( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_divwu( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_divwuf( __ev64_opaque__ a, __ev64_opaque__ b );

__ev64_opaque__ __ev_dlveb( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_dlveh( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_dlveob( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_dlveoh( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_dlvob( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_dlvoeb( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_dlvoeh( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_dlvoh( __ev64_opaque__ a, __ev64_opaque__ b );

__ev64_opaque__ __ev_dotp4hgasmf(__ev64_opaque__ a,__ev64_opaque__ b);
__ev64_opaque__ __ev_dotp4hgasmfa(__ev64_opaque__ a,__ev64_opaque__ b);
__ev64_opaque__ __ev_dotp4hgasmfaa(__ev64_opaque__ a,__ev64_opaque__ b);
__ev64_opaque__ __ev_dotp4hgasmfaa3(__ev64_opaque__ a,__ev64_opaque__ b,__ev64_opaque__ c);
__ev64_opaque__ __ev_dotp4hgasmi(__ev64_opaque__ a,__ev64_opaque__ b);
__ev64_opaque__ __ev_dotp4hgasmia(__ev64_opaque__ a,__ev64_opaque__ b);
__ev64_opaque__ __ev_dotp4hgasmiaa(__ev64_opaque__ a,__ev64_opaque__ b);
__ev64_opaque__ __ev_dotp4hgasmiaa3(__ev64_opaque__ a,__ev64_opaque__ b,__ev64_opaque__ c);
__ev64_opaque__ __ev_dotp4hgasumi(__ev64_opaque__ a,__ev64_opaque__ b);
__ev64_opaque__ __ev_dotp4hgasumia(__ev64_opaque__ a,__ev64_opaque__ b);
__ev64_opaque__ __ev_dotp4hgasumiaa(__ev64_opaque__ a,__ev64_opaque__ b);
__ev64_opaque__ __ev_dotp4hgasumiaa3(__ev64_opaque__ a,__ev64_opaque__ b,__ev64_opaque__
c);
__ev64_opaque__ __ev_dotp4hgaumi(__ev64_opaque__ a,__ev64_opaque__ b);
__ev64_opaque__ __ev_dotp4hgaumia(__ev64_opaque__ a,__ev64_opaque__ b);
__ev64_opaque__ __ev_dotp4hgaumiaa(__ev64_opaque__ a,__ev64_opaque__ b);
__ev64_opaque__ __ev_dotp4hgaumiaa3(__ev64_opaque__ a,__ev64_opaque__ b,__ev64_opaque__ c);
__ev64_opaque__ __ev_dotp4hgssmf(__ev64_opaque__ a,__ev64_opaque__ b);
```

```
__ev64_opaque__ __ev_dotp4hgssmfa(__ev64_opaque__ a,__ev64_opaque__ b);
__ev64_opaque__ __ev_dotp4hgssmfaa(__ev64_opaque__ a,__ev64_opaque__ b);
__ev64_opaque__ __ev_dotp4hgssmfaa3(__ev64_opaque__ a,__ev64_opaque__ b,__ev64_opaque__ c);
__ev64_opaque__ __ev_dotp4hgssmi(__ev64_opaque__ a,__ev64_opaque__ b);
__ev64_opaque__ __ev_dotp4hgssmia(__ev64_opaque__ a,__ev64_opaque__ b);
__ev64_opaque__ __ev_dotp4hgssmiaa(__ev64_opaque__ a,__ev64_opaque__ b);
__ev64_opaque__ __ev_dotp4hgssmiaa3(__ev64_opaque__ a,__ev64_opaque__ b,__ev64_opaque__ c);
__ev64_opaque__ __ev_dotp4hxgasmf(__ev64_opaque__ a,__ev64_opaque__ b);
__ev64_opaque__ __ev_dotp4hxgasmfa(__ev64_opaque__ a,__ev64_opaque__ b);
__ev64_opaque__ __ev_dotp4hxgasmfaa(__ev64_opaque__ a,__ev64_opaque__ b);
__ev64_opaque__ __ev_dotp4hxgasmfaa3(__ev64_opaque__ a,__ev64_opaque__ b,__ev64_opaque__
c);
__ev64_opaque__ __ev_dotp4hxgasmi(__ev64_opaque__ a,__ev64_opaque__ b);
__ev64_opaque__ __ev_dotp4hxgasmia(__ev64_opaque__ a,__ev64_opaque__ b);
__ev64_opaque__ __ev_dotp4hxgasmiaa(__ev64_opaque__ a,__ev64_opaque__ b);
__ev64_opaque__ __ev_dotp4hxgasmiaa3(__ev64_opaque__ a,__ev64_opaque__ b,__ev64_opaque__
c);
__ev64_opaque__ __ev_dotp4hxgssmf(__ev64_opaque__ a,__ev64_opaque__ b);
__ev64_opaque__ __ev_dotp4hxgssmfa(__ev64_opaque__ a,__ev64_opaque__ b);
__ev64_opaque__ __ev_dotp4hxgssmfaa(__ev64_opaque__ a,__ev64_opaque__ b);
__ev64_opaque__ __ev_dotp4hxgssmfaa3(__ev64_opaque__ a,__ev64_opaque__ b,__ev64_opaque__
c);
__ev64_opaque__ __ev_dotp4hxgssmi(__ev64_opaque__ a,__ev64_opaque__ b);
__ev64_opaque__ __ev_dotp4hxgssmia(__ev64_opaque__ a,__ev64_opaque__ b);
__ev64_opaque__ __ev_dotp4hxgssmiaa(__ev64_opaque__ a,__ev64_opaque__ b);
__ev64_opaque__ __ev_dotp4hxgssmiaa3(__ev64_opaque__ a,__ev64_opaque__ b,__ev64_opaque__
c);
__ev64_opaque__ __ev_dotpbasmi(__ev64_opaque__ a,__ev64_opaque__ b);
__ev64_opaque__ __ev_dotpbasmia(__ev64_opaque__ a,__ev64_opaque__ b);
__ev64_opaque__ __ev_dotpbasmiaaw(__ev64_opaque__ a,__ev64_opaque__ b);
__ev64_opaque__ __ev_dotpbasmiaaw3(__ev64_opaque__ a,__ev64_opaque__ b,__ev64_opaque__ c);
__ev64_opaque__ __ev_dotpbasumi(__ev64_opaque__ a,__ev64_opaque__ b);
__ev64_opaque__ __ev_dotpbasumia(__ev64_opaque__ a,__ev64_opaque__ b);
__ev64_opaque__ __ev_dotpbasumiaaw(__ev64_opaque__ a,__ev64_opaque__ b);
__ev64_opaque__ __ev_dotpbasumiaaw3(__ev64_opaque__ a,__ev64_opaque__ b,__ev64_opaque__ c);
__ev64_opaque__ __ev_dotpbaumi(__ev64_opaque__ a,__ev64_opaque__ b);
__ev64_opaque__ __ev_dotpbaumia(__ev64_opaque__ a,__ev64_opaque__ b);
__ev64_opaque__ __ev_dotpbaumiaaw(__ev64_opaque__ a,__ev64_opaque__ b);
__ev64_opaque__ __ev_dotpbaumiaaw3(__ev64_opaque__ a,__ev64_opaque__ b,__ev64_opaque__ c);
__ev64_opaque__ __ev_dotphasmi(__ev64_opaque__ a,__ev64_opaque__ b);
__ev64_opaque__ __ev_dotphasmia(__ev64_opaque__ a,__ev64_opaque__ b);
__ev64_opaque__ __ev_dotphasmiaaw(__ev64_opaque__ a,__ev64_opaque__ b);
__ev64_opaque__ __ev_dotphasmiaaw3(__ev64_opaque__ a,__ev64_opaque__ b,__ev64_opaque__ c);
__ev64_opaque__ __ev_dotphassf(__ev64_opaque__ a,__ev64_opaque__ b);
__ev64_opaque__ __ev_dotphassfa(__ev64_opaque__ a,__ev64_opaque__ b);
__ev64_opaque__ __ev_dotphassfaaw(__ev64_opaque__ a,__ev64_opaque__ b);
__ev64_opaque__ __ev_dotphassfaaw3(__ev64_opaque__ a,__ev64_opaque__ b,__ev64_opaque__ c);
__ev64_opaque__ __ev_dotphassfr(__ev64_opaque__ a,__ev64_opaque__ b);
__ev64_opaque__ __ev_dotphassfra(__ev64_opaque__ a,__ev64_opaque__ b);
__ev64_opaque__ __ev_dotphassfraaw(__ev64_opaque__ a,__ev64_opaque__ b);
__ev64_opaque__ __ev_dotphassfraaw3(__ev64_opaque__ a,__ev64_opaque__ b,__ev64_opaque__ c);
__ev64_opaque__ __ev_dotphassi(__ev64_opaque__ a,__ev64_opaque__ b);
__ev64_opaque__ __ev_dotphassia(__ev64_opaque__ a,__ev64_opaque__ b);
__ev64_opaque__ __ev_dotphassiaaw(__ev64_opaque__ a,__ev64_opaque__ b);
__ev64_opaque__ __ev_dotphassiaaw3(__ev64_opaque__ a,__ev64_opaque__ b,__ev64_opaque__ c);
__ev64_opaque__ __ev_dotphasumi(__ev64_opaque__ a,__ev64_opaque__ b);
__ev64_opaque__ __ev_dotphasumia(__ev64_opaque__ a,__ev64_opaque__ b);
```

```
__ev64_opaque__ __ev_dotphasumiaaw(__ev64_opaque__ a,__ev64_opaque__ b);
__ev64_opaque__ __ev_dotphasumiaaw3(__ev64_opaque__ a,__ev64_opaque__ b,__ev64_opaque__ c);
__ev64_opaque__ __ev_dotphasusi(__ev64_opaque__ a,__ev64_opaque__ b);
__ev64_opaque__ __ev_dotphasusia(__ev64_opaque__ a,__ev64_opaque__ b);
__ev64_opaque__ __ev_dotphasusiaaw(__ev64_opaque__ a,__ev64_opaque__ b);
__ev64_opaque__ __ev_dotphasusiaaw3(__ev64_opaque__ a,__ev64_opaque__ b,__ev64_opaque__ c);
__ev64_opaque__ __ev_dotphaumi(__ev64_opaque__ a,__ev64_opaque__ b);
__ev64_opaque__ __ev_dotphaumia(__ev64_opaque__ a,__ev64_opaque__ b);
__ev64_opaque__ __ev_dotphaumiaaw(__ev64_opaque__ a,__ev64_opaque__ b);
__ev64_opaque__ __ev_dotphaumiaaw3(__ev64_opaque__ a,__ev64_opaque__ b,__ev64_opaque__ c);
__ev64_opaque__ __ev_dotphausi(__ev64_opaque__ a,__ev64_opaque__ b);
__ev64_opaque__ __ev_dotphausia(__ev64_opaque__ a,__ev64_opaque__ b);
__ev64_opaque__ __ev_dotphausiaaw(__ev64_opaque__ a,__ev64_opaque__ b);
__ev64_opaque__ __ev_dotphausiaaw3(__ev64_opaque__ a,__ev64_opaque__ b,__ev64_opaque__ c);
__ev64_opaque__ __ev_dotphihcsmi(__ev64_opaque__ a,__ev64_opaque__ b);
__ev64_opaque__ __ev_dotphihcsmia(__ev64_opaque__ a,__ev64_opaque__ b);
__ev64_opaque__ __ev_dotphihcsmiaaw(__ev64_opaque__ a,__ev64_opaque__ b);
__ev64_opaque__ __ev_dotphihcsmiaaw3(__ev64_opaque__ a,__ev64_opaque__ b,__ev64_opaque__
c);
__ev64_opaque__ __ev_dotphihcssf(__ev64_opaque__ a,__ev64_opaque__ b);
__ev64_opaque__ __ev_dotphihcssfa(__ev64_opaque__ a,__ev64_opaque__ b);
__ev64_opaque__ __ev_dotphihcssfaaw(__ev64_opaque__ a,__ev64_opaque__ b);
__ev64_opaque__ __ev_dotphihcssfaaw3(__ev64_opaque__ a,__ev64_opaque__ b,__ev64_opaque__
c);
__ev64_opaque__ __ev_dotphihcssfr(__ev64_opaque__ a,__ev64_opaque__ b);
__ev64_opaque__ __ev_dotphihcssfra(__ev64_opaque__ a,__ev64_opaque__ b);
__ev64_opaque__ __ev_dotphihcssfraaw(__ev64_opaque__ a,__ev64_opaque__ b);
__ev64_opaque__ __ev_dotphihcssfraaw3(__ev64_opaque__ a,__ev64_opaque__ b,__ev64_opaque__
c);
__ev64_opaque__ __ev_dotphihcssi(__ev64_opaque__ a,__ev64_opaque__ b);
__ev64_opaque__ __ev_dotphihcssia(__ev64_opaque__ a,__ev64_opaque__ b);
__ev64_opaque__ __ev_dotphihcssiaaw(__ev64_opaque__ a,__ev64_opaque__ b);
__ev64_opaque__ __ev_dotphihcssiaaw3(__ev64_opaque__ a,__ev64_opaque__ b,__ev64_opaque__
c);
__ev64_opaque__ __ev_dotphssmi(__ev64_opaque__ a,__ev64_opaque__ b);
__ev64_opaque__ __ev_dotphssmia(__ev64_opaque__ a,__ev64_opaque__ b);
__ev64_opaque__ __ev_dotphssmiaaw(__ev64_opaque__ a,__ev64_opaque__ b);
__ev64_opaque__ __ev_dotphssmiaaw3(__ev64_opaque__ a,__ev64_opaque__ b,__ev64_opaque__ c);
__ev64_opaque__ __ev_dotphsssf(__ev64_opaque__ a,__ev64_opaque__ b);
__ev64_opaque__ __ev_dotphsssfa(__ev64_opaque__ a,__ev64_opaque__ b);
__ev64_opaque__ __ev_dotphsssfaaw(__ev64_opaque__ a,__ev64_opaque__ b);
__ev64_opaque__ __ev_dotphsssfaaw3(__ev64_opaque__ a,__ev64_opaque__ b,__ev64_opaque__ c);
__ev64_opaque__ __ev_dotphsssfr(__ev64_opaque__ a,__ev64_opaque__ b);
__ev64_opaque__ __ev_dotphsssfra(__ev64_opaque__ a,__ev64_opaque__ b);
__ev64_opaque__ __ev_dotphsssfraaw(__ev64_opaque__ a,__ev64_opaque__ b);
__ev64_opaque__ __ev_dotphsssfraaw3(__ev64_opaque__ a,__ev64_opaque__ b,__ev64_opaque__ c);
__ev64_opaque__ __ev_dotphsssi(__ev64_opaque__ a,__ev64_opaque__ b);
__ev64_opaque__ __ev_dotphsssia(__ev64_opaque__ a,__ev64_opaque__ b);
__ev64_opaque__ __ev_dotphsssiaaw(__ev64_opaque__ a,__ev64_opaque__ b);
__ev64_opaque__ __ev_dotphsssiaaw3(__ev64_opaque__ a,__ev64_opaque__ b,__ev64_opaque__ c);
__ev64_opaque__ __ev_dotplohcsmi(__ev64_opaque__ a,__ev64_opaque__ b);
__ev64_opaque__ __ev_dotplohcsmia(__ev64_opaque__ a,__ev64_opaque__ b);
__ev64_opaque__ __ev_dotplohcsmiaaw(__ev64_opaque__ a,__ev64_opaque__ b);
__ev64_opaque__ __ev_dotplohcsmiaaw3(__ev64_opaque__ a,__ev64_opaque__ b,__ev64_opaque__
c);
__ev64_opaque__ __ev_dotplohcssf(__ev64_opaque__ a,__ev64_opaque__ b);
__ev64_opaque__ __ev_dotplohcssfa(__ev64_opaque__ a,__ev64_opaque__ b);
```

```
__ev64_opaque__ __ev_dotplohcssfaaw(__ev64_opaque__ a,__ev64_opaque__ b);
__ev64_opaque__ __ev_dotplohcssfaaw3(__ev64_opaque__ a,__ev64_opaque__ b,__ev64_opaque__
c);
__ev64_opaque__ __ev_dotplohcssfr(__ev64_opaque__ a,__ev64_opaque__ b);
__ev64_opaque__ __ev_dotplohcssfra(__ev64_opaque__ a,__ev64_opaque__ b);
__ev64_opaque__ __ev_dotplohcssfraaw(__ev64_opaque__ a,__ev64_opaque__ b);
__ev64_opaque__ __ev_dotplohcssfraaw3(__ev64_opaque__ a,__ev64_opaque__ b,__ev64_opaque__
c);
__ev64_opaque__ __ev_dotplohcssi(__ev64_opaque__ a,__ev64_opaque__ b);
__ev64_opaque__ __ev_dotplohcssia(__ev64_opaque__ a,__ev64_opaque__ b);
__ev64_opaque__ __ev_dotplohcssiaaw(__ev64_opaque__ a,__ev64_opaque__ b);
__ev64_opaque__ __ev_dotplohcssiaaw3(__ev64_opaque__ a,__ev64_opaque__ b,__ev64_opaque__
c);
__ev64_opaque__ __ev_dotpwasmi(__ev64_opaque__ a,__ev64_opaque__ b);
__ev64_opaque__ __ev_dotpwasmia(__ev64_opaque__ a,__ev64_opaque__ b);
__ev64_opaque__ __ev_dotpwasmiaa(__ev64_opaque__ a,__ev64_opaque__ b);
__ev64_opaque__ __ev_dotpwasmiaa3(__ev64_opaque__ a,__ev64_opaque__ b,__ev64_opaque__ c);
__ev64_opaque__ __ev_dotpwassi(__ev64_opaque__ a,__ev64_opaque__ b);
__ev64_opaque__ __ev_dotpwassia(__ev64_opaque__ a,__ev64_opaque__ b);
__ev64_opaque__ __ev_dotpwassiaa(__ev64_opaque__ a,__ev64_opaque__ b);
__ev64_opaque__ __ev_dotpwassiaa3(__ev64_opaque__ a,__ev64_opaque__ b,__ev64_opaque__ c);
__ev64_opaque__ __ev_dotpwasumi(__ev64_opaque__ a,__ev64_opaque__ b);
__ev64_opaque__ __ev_dotpwasumia(__ev64_opaque__ a,__ev64_opaque__ b);
__ev64_opaque__ __ev_dotpwasumiaa(__ev64_opaque__ a,__ev64_opaque__ b);
__ev64_opaque__ __ev_dotpwasumiaa3(__ev64_opaque__ a,__ev64_opaque__ b,__ev64_opaque__ c);
__ev64_opaque__ __ev_dotpwasusi(__ev64_opaque__ a,__ev64_opaque__ b);
__ev64_opaque__ __ev_dotpwasusia(__ev64_opaque__ a,__ev64_opaque__ b);
__ev64_opaque__ __ev_dotpwasusiaa(__ev64_opaque__ a,__ev64_opaque__ b);
__ev64_opaque__ __ev_dotpwasusiaa3(__ev64_opaque__ a,__ev64_opaque__ b,__ev64_opaque__ c);
__ev64_opaque__ __ev_dotpwaumi(__ev64_opaque__ a,__ev64_opaque__ b);
__ev64_opaque__ __ev_dotpwaumia(__ev64_opaque__ a,__ev64_opaque__ b);
__ev64_opaque__ __ev_dotpwaumiaa(__ev64_opaque__ a,__ev64_opaque__ b);
__ev64_opaque__ __ev_dotpwaumiaa3(__ev64_opaque__ a,__ev64_opaque__ b,__ev64_opaque__ c);
__ev64_opaque__ __ev_dotpwausi(__ev64_opaque__ a,__ev64_opaque__ b);
__ev64_opaque__ __ev_dotpwausia(__ev64_opaque__ a,__ev64_opaque__ b);
__ev64_opaque__ __ev_dotpwausiaa(__ev64_opaque__ a,__ev64_opaque__ b);
__ev64_opaque__ __ev_dotpwausiaa3(__ev64_opaque__ a,__ev64_opaque__ b,__ev64_opaque__ c);
__ev64_opaque__ __ev_dotpwcsmi(__ev64_opaque__ a,__ev64_opaque__ b);
__ev64_opaque__ __ev_dotpwcsmia(__ev64_opaque__ a,__ev64_opaque__ b);
__ev64_opaque__ __ev_dotpwcsmiaaw(__ev64_opaque__ a,__ev64_opaque__ b);
__ev64_opaque__ __ev_dotpwcsmiaaw3(__ev64_opaque__ a,__ev64_opaque__ b,__ev64_opaque__ c);
__ev64_opaque__ __ev_dotpwcssf(__ev64_opaque__ a,__ev64_opaque__ b);
__ev64_opaque__ __ev_dotpwcssfa(__ev64_opaque__ a,__ev64_opaque__ b);
__ev64_opaque__ __ev_dotpwcssfaaw(__ev64_opaque__ a,__ev64_opaque__ b);
__ev64_opaque__ __ev_dotpwcssfaaw3(__ev64_opaque__ a,__ev64_opaque__ b,__ev64_opaque__ c);
__ev64_opaque__ __ev_dotpwcssfr(__ev64_opaque__ a,__ev64_opaque__ b);
__ev64_opaque__ __ev_dotpwcssfra(__ev64_opaque__ a,__ev64_opaque__ b);
__ev64_opaque__ __ev_dotpwcssfraaw(__ev64_opaque__ a,__ev64_opaque__ b);
__ev64_opaque__ __ev_dotpwcssfraaw3(__ev64_opaque__ a,__ev64_opaque__ b,__ev64_opaque__ c);
__ev64_opaque__ __ev_dotpwcssi(__ev64_opaque__ a,__ev64_opaque__ b);
__ev64_opaque__ __ev_dotpwcssia(__ev64_opaque__ a,__ev64_opaque__ b);
__ev64_opaque__ __ev_dotpwcssiaaw(__ev64_opaque__ a,__ev64_opaque__ b);
__ev64_opaque__ __ev_dotpwcssiaaw3(__ev64_opaque__ a,__ev64_opaque__ b,__ev64_opaque__ c);
__ev64_opaque__ __ev_dotpwgasmf(__ev64_opaque__ a,__ev64_opaque__ b);
__ev64_opaque__ __ev_dotpwgasmfa(__ev64_opaque__ a,__ev64_opaque__ b);
__ev64_opaque__ __ev_dotpwgasmfaa(__ev64_opaque__ a,__ev64_opaque__ b);
__ev64_opaque__ __ev_dotpwgasmfaa3(__ev64_opaque__ a,__ev64_opaque__ b,__ev64_opaque__ c);
```

```
__ev64_opaque__ __ev_dotpwgasmfr(__ev64_opaque__ a,__ev64_opaque__ b);
__ev64_opaque__ __ev_dotpwgasmfra(__ev64_opaque__ a,__ev64_opaque__ b);
__ev64_opaque__ __ev_dotpwgasmfraa(__ev64_opaque__ a,__ev64_opaque__ b);
__ev64_opaque__ __ev_dotpwgasmfraa3(__ev64_opaque__ a,__ev64_opaque__ b,__ev64_opaque__ c);
__ev64_opaque__ __ev_dotpwgssmf(__ev64_opaque__ a,__ev64_opaque__ b);
__ev64_opaque__ __ev_dotpwgssmfa(__ev64_opaque__ a,__ev64_opaque__ b);
__ev64_opaque__ __ev_dotpwgssmfaa(__ev64_opaque__ a,__ev64_opaque__ b);
__ev64_opaque__ __ev_dotpwgssmfaa3(__ev64_opaque__ a,__ev64_opaque__ b,__ev64_opaque__ c);
__ev64_opaque__ __ev_dotpwgssmfr(__ev64_opaque__ a,__ev64_opaque__ b);
__ev64_opaque__ __ev_dotpwgssmfra(__ev64_opaque__ a,__ev64_opaque__ b);
__ev64_opaque__ __ev_dotpwgssmfraa(__ev64_opaque__ a,__ev64_opaque__ b);
__ev64_opaque__ __ev_dotpwgssmfraa3(__ev64_opaque__ a,__ev64_opaque__ b,__ev64_opaque__ c);
__ev64_opaque__ __ev_dotpwssmi(__ev64_opaque__ a,__ev64_opaque__ b);
__ev64_opaque__ __ev_dotpwssmia(__ev64_opaque__ a,__ev64_opaque__ b);
__ev64_opaque__ __ev_dotpwssmiaa(__ev64_opaque__ a,__ev64_opaque__ b);
__ev64_opaque__ __ev_dotpwssmiaa3(__ev64_opaque__ a,__ev64_opaque__ b,__ev64_opaque__ c);
__ev64_opaque__ __ev_dotpwsssi(__ev64_opaque__ a,__ev64_opaque__ b);
__ev64_opaque__ __ev_dotpwsssia(__ev64_opaque__ a,__ev64_opaque__ b);
__ev64_opaque__ __ev_dotpwsssiaa(__ev64_opaque__ a,__ev64_opaque__ b);
__ev64_opaque__ __ev_dotpwsssiaa3(__ev64_opaque__ a,__ev64_opaque__ b,__ev64_opaque__ c);
__ev64_opaque__ __ev_dotpwxgasmf(__ev64_opaque__ a,__ev64_opaque__ b);
__ev64_opaque__ __ev_dotpwxgasmfa(__ev64_opaque__ a,__ev64_opaque__ b);
__ev64_opaque__ __ev_dotpwxgasmfaa(__ev64_opaque__ a,__ev64_opaque__ b);
__ev64_opaque__ __ev_dotpwxgasmfaa3(__ev64_opaque__ a,__ev64_opaque__ b,__ev64_opaque__ c);
__ev64_opaque__ __ev_dotpwxgasmfr(__ev64_opaque__ a,__ev64_opaque__ b);
__ev64_opaque__ __ev_dotpwxgasmfra(__ev64_opaque__ a,__ev64_opaque__ b);
__ev64_opaque__ __ev_dotpwxgasmfraa(__ev64_opaque__ a,__ev64_opaque__ b);
__ev64_opaque__ __ev_dotpwxgasmfraa3(__ev64_opaque__ a,__ev64_opaque__ b,__ev64_opaque__
c);
__ev64_opaque__ __ev_dotpwxgssmf(__ev64_opaque__ a,__ev64_opaque__ b);
__ev64_opaque__ __ev_dotpwxgssmfa(__ev64_opaque__ a,__ev64_opaque__ b);
__ev64_opaque__ __ev_dotpwxgssmfaa(__ev64_opaque__ a,__ev64_opaque__ b);
__ev64_opaque__ __ev_dotpwxgssmfaa3(__ev64_opaque__ a,__ev64_opaque__ b,__ev64_opaque__ c);
__ev64_opaque__ __ev_dotpwxgssmfr(__ev64_opaque__ a,__ev64_opaque__ b);
__ev64_opaque__ __ev_dotpwxgssmfra(__ev64_opaque__ a,__ev64_opaque__ b);
__ev64_opaque__ __ev_dotpwxgssmfraa(__ev64_opaque__ a,__ev64_opaque__ b);
__ev64_opaque__ __ev_dotpwxgssmfraa3(__ev64_opaque__ a,__ev64_opaque__ b,__ev64_opaque__
c);
__ev64_opaque__ __ev_eqv( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_extsb( __ev64_opaque__ a );
__ev64_opaque__ __ev_extsbh( __ev64_opaque__ a );
__ev64_opaque__ __ev_extsh( __ev64_opaque__ a );
__ev64_opaque__ __ev_extsw( __ev64_opaque__ a );
__ev64_opaque__ __ev_extzb( __ev64_opaque__ a );
__ev64_opaque__ __ev_insb( __ev64_opaque__ a, __ev64_opaque__ b, 3-bit unsigned literal,
                        3-bit unsigned literal );
__ev64_opaque__ __ev_insh( __ev64_opaque__ a, __ev64_opaque__ b, 2-bit unsigned literal,
                        2-bit unsigned literal );
__ev64_opaque__ __ev_ilveh( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_ilveoh( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_ilvhih( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_ilvhiloh( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_ilvloh( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_ilvlohih( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_ilvoeh( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_ilvoh( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_lvsl( __ev64_opaque__ a, __ev64_opaque__ b );
```

```
__ev64_opaque__ __ev_lvsr( __ev64_opaque__ a, __ev64_opaque__ b );

__ev64_opaque__ __ev_mar( );

__ev64_opaque__ __ev_maxbpsh( __ev64_opaque__ a );
__ev64_opaque__ __ev_maxbpuh( __ev64_opaque__ a );
__ev64_opaque__ __ev_maxbs( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_maxbu( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_maxds( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_maxdu( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_maxhpsw( __ev64_opaque__ a );
__ev64_opaque__ __ev_maxhpuw( __ev64_opaque__ a );
__ev64_opaque__ __ev_maxhs( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_maxhu( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_maxmagws( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_maxwpsd( __ev64_opaque__ a );
__ev64_opaque__ __ev_maxwpud( __ev64_opaque__ a );
__ev64_opaque__ __ev_maxws( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_maxwu( __ev64_opaque__ a, __ev64_opaque__ b );

__ev64_opaque__ __ev_mbesmi( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mbesmia( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mbesmiaah( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mbesmianh( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mbessiaah( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mbessianh( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mbesumi( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mbesumia( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mbesumiaah( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mbesumianh( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mbesusiaah( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mbesusianh( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mbeumi( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mbeumia( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mbeumiaah( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mbeumianh( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mbeusiaah( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mbeusianh( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mbosmi( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mbosmia( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mbosmiaah( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mbosmianh( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mbossiaah( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mbossianh( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mbosumi( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mbosumia( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mbosumiaah( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mbosumianh( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mbosusiaah( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mbosusianh( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mboumi( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mboumia( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mboumiaah( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mboumianh( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mbousiaah( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mbousianh( __ev64_opaque__ a, __ev64_opaque__ b );
```

```
__ev64_opaque__ __ev_mergehi( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mergehilo( __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_mhegsmfaa( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mhegsmfan( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mhegsmiaa( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mhegsmian( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mhegumiaa( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mhegumian( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mhesmf( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mhesmfa( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mhesmfaaw( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mhesmfanw( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mhesmi( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mhesmia( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mhesmiaaw( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mhesmianw( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mhessf( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mhessfa( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mhessfaaw( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mhessfanw( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mhessiaaw( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mhessianw( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mhesumi( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mhesumia( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mhesumiaaw( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mhesumianw( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mhesusiaaw( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mhesusianw( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mheumi( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mheumia( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mheumiaaw( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mheumianw( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mheusiaaw( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mheusianw( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mhogsmfaa( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mhogsmfan( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mhogsmiaa( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mhogsmian( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mhogumiaa( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mhogumian( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mhosmf( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mhosmfa( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mhosmfaaw( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mhosmfanw( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mhosmi( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mhosmia( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mhosmiaaw( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mhosmianw( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mhossf( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mhossfa( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mhossfaaw( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mhossfanw( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mhossiaaw( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mhossianw( __ev64_opaque__ a, __ev64_opaque__ b );
```

```
__ev64_opaque__ __ev_mhosumi( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mhosumia( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mhosumiaaw( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mhosumianw( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mhosusiaaw( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mhosusianw( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mhoumi( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mhoumia( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mhoumiaaw( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mhoumianw( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mhousiaaw( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mhousianw( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mhssf( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mhssfr( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mhssi( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mhsusi( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mhumi( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mhusi( __ev64_opaque__ a, __ev64_opaque__ b );

__ev64_opaque__ __ev_minbpsh( __ev64_opaque__ a );
__ev64_opaque__ __ev_minbpuh( __ev64_opaque__ a );
__ev64_opaque__ __ev_minbs( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_minbu( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_minds( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mindu( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_minhpsw( __ev64_opaque__ a );
__ev64_opaque__ __ev_minhpuw( __ev64_opaque__ a );
__ev64_opaque__ __ev_minhs( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_minhu( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_minwpsd( __ev64_opaque__ a );
__ev64_opaque__ __ev_minwpud( __ev64_opaque__ a );
__ev64_opaque__ __ev_minws( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_minwu( __ev64_opaque__ a, __ev64_opaque__ b );

__ev64_opaque__ __ev_mra( __ev64_opaque__ a );

__ev64_opaque__ __ev_mwhsmf( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mwhsmfa( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mwehgsmf( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mwehgsmfa( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mwehgsmfaa( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mwehgsmfan( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mwehgsmfr( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mwehgsmfra( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mwehgsmfraa( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mwehgsmfran( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mwhsmi( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mwhsmia( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mwhssf( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mwhssfa( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mwhssfaaw( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mwhssfaaw3( __ev64_opaque__ a, __ev64_opaque__ b, __ev64_opaque__ c );
__ev64_opaque__ __ev_mwhssfanw( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mwhssfanw3( __ev64_opaque__ a, __ev64_opaque__ b, __ev64_opaque__ c);
__ev64_opaque__ __ev_mwhssfr( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mwhssfra( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mwhssfraaw( __ev64_opaque__ a, __ev64_opaque__ b );
```

```
__ev64_opaque__ __ev_mwhssfraaw3( __ev64_opaque__ a, __ev64_opaque__ b, __ev64_opaque__ c);
__ev64_opaque__ __ev_mwhssfranw( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mwhssfranw3( __ev64_opaque__ a, __ev64_opaque__ b, __ev64_opaque__ c);
__ev64_opaque__ __ev_mwhumi( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mwhumia( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mwlsmiaaw( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mwlsmiaaw3( __ev64_opaque__ a, __ev64_opaque__ b, __ev64_opaque__ c );
__ev64_opaque__ __ev_mwlsmianw( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mwlsmianw3( __ev64_opaque__ a, __ev64_opaque__ b, __ev64_opaque__ c );
__ev64_opaque__ __ev_mwlssiaaw( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mwlssiaaw3( __ev64_opaque__ a, __ev64_opaque__ b, __ev64_opaque__ c );
__ev64_opaque__ __ev_mwlssianw( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mwlssianw3( __ev64_opaque__ a, __ev64_opaque__ b, __ev64_opaque__ c );
__ev64_opaque__ __ev_mwlumi( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mwlumia( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mwlumiaaw( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mwlumiaaw3( __ev64_opaque__ a, __ev64_opaque__ b, __ev64_opaque__ c );
__ev64_opaque__ __ev_mwlumianw( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mwlumianw3( __ev64_opaque__ a, __ev64_opaque__ b, __ev64_opaque__ c );
__ev64_opaque__ __ev_mwlusiaaw( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mwlusiaaw3( __ev64_opaque__ a, __ev64_opaque__ b, __ev64_opaque__ c );
__ev64_opaque__ __ev_mwlusianw( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mwlusianw3( __ev64_opaque__ a, __ev64_opaque__ b, __ev64_opaque__ c );
__ev64_opaque__ __ev_mwohgsmf( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mwohgsmfa( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mwohgsmfaa( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mwohgsmfan( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mwohgsmfr( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mwohgsmfra( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mwohgsmfraa( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mwohgsmfran( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mwsmf( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mwsmfa( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mwsmfaa( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mwsmfan( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mwsmi( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mwsmia( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mwsmiaa( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mwsmian( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mwssf( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mwssfa( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mwssfaa( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mwssfan( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mwssiaa( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mwssian( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mwssiw( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mwumi( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mwumia( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mwumiaa( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mwumian( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mwusiaa( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mwusian( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_mwusiw( __ev64_opaque__ a, __ev64_opaque__ b );

__ev64_opaque__ __ev_nand( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_neg( __ev64_opaque__ a );
__ev64_opaque__ __ev_negb( __ev64_opaque__ a );
```

```
__ev64_opaque__ __ev_negbo( __ev64_opaque__ a );
__ev64_opaque__ __ev_negbos( __ev64_opaque__ a );
__ev64_opaque__ __ev_negbs( __ev64_opaque__ a );
__ev64_opaque__ __ev_negd( __ev64_opaque__ a );
__ev64_opaque__ __ev_negds( __ev64_opaque__ a );
__ev64_opaque__ __ev_negh( __ev64_opaque__ a );
__ev64_opaque__ __ev_negho( __ev64_opaque__ a );
__ev64_opaque__ __ev_neghos( __ev64_opaque__ a );
__ev64_opaque__ __ev_neghs( __ev64_opaque__ a );
__ev64_opaque__ __ev_negs( __ev64_opaque__ a );
__ev64_opaque__ __ev_negwo( __ev64_opaque__ a );
__ev64_opaque__ __ev_negwos( __ev64_opaque__ a );
__ev64_opaque__ __ev_nor( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_or( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_orc( __ev64_opaque__ a, __ev64_opaque__ b );

__ev64_opaque__ __ev_perm( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_perm2( __ev64_opaque__ a, __ev64_opaque__ b, __ev64_opaque__ c );
__ev64_opaque__ __ev_perm3( __ev64_opaque__ a, __ev64_opaque__ b, __ev64_opaque__ c );

__ev64_opaque__ __ev_pksdshefrs( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_pksdswfrs( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_pksdsws( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_pkshsbs( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_pkshubs( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_pkswgshefrs( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_pkswgswfrs( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_pkswshfrs( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_pkswshilvfrs( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_pkswshilvs( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_pkswshs( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_pkswuhs( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_pkuduws( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_pkuhubs( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_pkuwuhs( __ev64_opaque__ a, __ev64_opaque__ b );

__ev64_opaque__ __ev_popcntb( __ev64_opaque__ a );

__ev64_opaque__ __ev_rlb( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_rlbi( __ev64_opaque__ a, 5-bit unsigned literal );
__ev64_opaque__ __ev_rlh( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_rlhi( __ev64_opaque__ a, 5-bit unsigned literal );
__ev64_opaque__ __ev_rlw( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_rlwi( __ev64_opaque__ a, 5-bit unsigned literal );

__ev64_opaque__ __ev_rnddnw( __ev64_opaque__ a );
__ev64_opaque__ __ev_rnddnwss( __ev64_opaque__ a );
__ev64_opaque__ __ev_rnddnwus( __ev64_opaque__ a );
__ev64_opaque__ __ev_rnddw( __ev64_opaque__ a );
__ev64_opaque__ __ev_rnddwss( __ev64_opaque__ a );
__ev64_opaque__ __ev_rnddwus( __ev64_opaque__ a );
__ev64_opaque__ __ev_rndhb( __ev64_opaque__ a );
__ev64_opaque__ __ev_rndhbss( __ev64_opaque__ a );
__ev64_opaque__ __ev_rndhbus( __ev64_opaque__ a );
__ev64_opaque__ __ev_rndhnb( __ev64_opaque__ a );
__ev64_opaque__ __ev_rndhnbss( __ev64_opaque__ a );
__ev64_opaque__ __ev_rndhnbus( __ev64_opaque__ a );
```

```
__ev64_opaque__ __ev_rndw( __ev64_opaque__ a );
__ev64_opaque__ __ev_rndwh( __ev64_opaque__ a );
__ev64_opaque__ __ev_rndwhss( __ev64_opaque__ a );
__ev64_opaque__ __ev_rndwhus( __ev64_opaque__ a );
__ev64_opaque__ __ev_rndwnh( __ev64_opaque__ a );
__ev64_opaque__ __ev_rndwnhss( __ev64_opaque__ a );
__ev64_opaque__ __ev_rndwnhus( __ev64_opaque__ a );
__ev64_opaque__ __ev_sad2sh( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_sad2sha( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_sad2shaaw( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_sad2uh( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_sad2uha( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_sad2uhaaw( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_sad4sb( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_sad4sba( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_sad4sbaaw( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_sad4ub( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_sad4uba( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_sad4ubaaw( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_sadsw( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_sadswa( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_sadswaa( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_saduw( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_saduwa( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_saduwaa( __ev64_opaque__ a, __ev64_opaque__ b );

__ev64_opaque__ __ev_satsbub( __ev64_opaque__ a );
__ev64_opaque__ __ev_satsdsw( __ev64_opaque__ a );
__ev64_opaque__ __ev_satsduw( __ev64_opaque__ a );
__ev64_opaque__ __ev_satshsb( __ev64_opaque__ a );
__ev64_opaque__ __ev_satshub( __ev64_opaque__ a );
__ev64_opaque__ __ev_satshuh( __ev64_opaque__ a );
__ev64_opaque__ __ev_satswgsdf( __ev64_opaque__ a );
__ev64_opaque__ __ev_satswsh( __ev64_opaque__ a );
__ev64_opaque__ __ev_satswuh( __ev64_opaque__ a );
__ev64_opaque__ __ev_satswuw( __ev64_opaque__ a );
__ev64_opaque__ __ev_satuduw( __ev64_opaque__ a );
__ev64_opaque__ __ev_satubsb( __ev64_opaque__ a );
__ev64_opaque__ __ev_satuhsh( __ev64_opaque__ a );
__ev64_opaque__ __ev_satuhub( __ev64_opaque__ a );
__ev64_opaque__ __ev_satuwsw( __ev64_opaque__ a );
__ev64_opaque__ __ev_satuwuh( __ev64_opaque__ a );

__ev64_opaque__ __ev_selbit( __ev64_opaque__ a, __ev64_opaque__ b, __ev64_opaque__ c );
__ev64_opaque__ __ev_selbitm0( __ev64_opaque__ a, __ev64_opaque__ b, __ev64_opaque__ c );
__ev64_opaque__ __ev_selbitm1( __ev64_opaque__ a, __ev64_opaque__ b, __ev64_opaque__ c );

__ev64_opaque__ __ev_seteqb( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_seteqb_rc( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_seteqh( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_seteqh_rc( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_seteqw( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_seteqw_rc( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_setgtbs( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_setgtbs_rc( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_setgtbu( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_setgtbu_rc( __ev64_opaque__ a, __ev64_opaque__ b );
```

```
__ev64_opaque__ __ev_setgths( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_setgths_rc( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_setgthu( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_setgthu_rc( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_setgtws( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_setgtws_rc( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_setgtwu( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_setgtwu_rc( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_setltbs( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_setltbs_rc( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_setltbu( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_setltbu_rc( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_setlths( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_setlths_rc( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_setlthu( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_setlthu_rc( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_setltws( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_setltws_rc( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_setltwu( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_setltwu_rc( __ev64_opaque__ a, __ev64_opaque__ b );

__ev64_opaque__ __ev_sl( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_sli( __ev64_opaque__ a, 5-bit unsigned literal );
__ev64_opaque__ __ev_slb( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_slbi( __ev64_opaque__ a, 5-bit unsigned literal );
__ev64_opaque__ __ev_slh( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_slhi( __ev64_opaque__ a, 5-bit unsigned literal );
__ev64_opaque__ __ev_sloi( __ev64_opaque__ a, 3-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_splatb( __ev64_opaque__ a, 3-bit unsigned literal );
__ev64_opaque__ __ev_splatfi( 5-bit signed literal );
__ev64_opaque__ __ev_splatfia( 5-bit signed literal );
__ev64_opaque__ __ev_splatfib( 5-bit signed literal );
__ev64_opaque__ __ev_splatfiba( 5-bit signed literal );
__ev64_opaque__ __ev_splatfibo( 5-bit signed literal );
__ev64_opaque__ __ev_splatfiboa( 5-bit signed literal );
__ev64_opaque__ __ev_splatfid( 5-bit signed literal );
__ev64_opaque__ __ev_splatfida( 5-bit signed literal );
__ev64_opaque__ __ev_splatfih( 5-bit signed literal );
__ev64_opaque__ __ev_splatfiha( 5-bit signed literal );
__ev64_opaque__ __ev_splatfiho( 5-bit signed literal );
__ev64_opaque__ __ev_splatfihoa( 5-bit signed literal );
__ev64_opaque__ __ev_splatfio( 5-bit signed literal );
__ev64_opaque__ __ev_splatfioa( 5-bit signed literal );
__ev64_opaque__ __ev_splath( __ev64_opaque__ a, 2-bit unsigned literal );
__ev64_opaque__ __ev_splati( 5-bit signed literal );
__ev64_opaque__ __ev_splatia( 5-bit signed literal );
__ev64_opaque__ __ev_splatib( 5-bit signed literal );
__ev64_opaque__ __ev_splatiba( 5-bit signed literal );
__ev64_opaque__ __ev_splatibe( 5-bit signed literal );
__ev64_opaque__ __ev_splatibea( 5-bit signed literal );
__ev64_opaque__ __ev_splatid( 5-bit signed literal );
__ev64_opaque__ __ev_splatida( 5-bit signed literal );
__ev64_opaque__ __ev_splatie( 5-bit signed literal );
__ev64_opaque__ __ev_splatiea( 5-bit signed literal );
```

```
__ev64_opaque__ __ev_splatih( 5-bit signed literal );
__ev64_opaque__ __ev_splatiha( 5-bit signed literal );
__ev64_opaque__ __ev_splatihe( 5-bit signed literal );
__ev64_opaque__ __ev_splatihea( 5-bit signed literal );

__ev64_opaque__ __ev_srbis( __ev64_opaque__ a, 5-bit unsigned literal );
__ev64_opaque__ __ev_srbiu( __ev64_opaque__ a, 5-bit unsigned literal );
__ev64_opaque__ __ev_srbs( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_srbu( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_srhis( __ev64_opaque__ a, 5-bit unsigned literal );
__ev64_opaque__ __ev_srhiu( __ev64_opaque__ a, 5-bit unsigned literal );
__ev64_opaque__ __ev_srhs( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_srhu( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_sris( __ev64_opaque__ a, 5-bit unsigned literal );
__ev64_opaque__ __ev_sriu( __ev64_opaque__ a, 5-bit unsigned literal );
__ev64_opaque__ __ev_srois( __ev64_opaque__ a, 3-bit unsigned literal );
__ev64_opaque__ __ev_sroiu( __ev64_opaque__ a, 3-bit unsigned literal );
__ev64_opaque__ __ev_srs( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_sru( __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_srws( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_srwu( __ev64_opaque__ a, __ev64_opaque__ b );

__ev64_opaque__ __ev_subf2add2h( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_subf2add2hss( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_subfaddh( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_subfaddhss( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_subfaddhx( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_subfaddhxss( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_subfaddw( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_subfaddwss( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_subfaddwx( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_subfaddwxss( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_subfb( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_subfbss( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_subfbus( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_subfd( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_subfdss( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_subfdus( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_subfh( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_subfhhisw( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_subfhhiuw( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_subfhlosw( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_subfhlouw( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_subfhss( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_subfhus( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_subfhx( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_subfhxss( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_subfhxus( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_subfsmiaa( __ev64_opaque__ a );
__ev64_opaque__ __ev_subfsmiaaw( __ev64_opaque__ a );
__ev64_opaque__ __ev_subfssiaa( __ev64_opaque__ a );
__ev64_opaque__ __ev_subfssiaaw( __ev64_opaque__ a );
__ev64_opaque__ __ev_subfumiaaw( __ev64_opaque__ a );
__ev64_opaque__ __ev_subfusiaa( __ev64_opaque__ a );
__ev64_opaque__ __ev_subfusiaaw( __ev64_opaque__ a );
```

```
__ev64_opaque__ __ev_subfw( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_subfwegsf( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_subfwegsi( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_subfwogsf( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_subfwogsi( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_subfwss( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_subfwus( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_subfwx( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_subfwxss( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_subfwxus( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_subifb( 5-bit unsigned literal, __ev64_opaque__ b );
__ev64_opaque__ __ev_subifh( 5-bit unsigned literal, __ev64_opaque__ b );
__ev64_opaque__ __ev_subifw( 5-bit unsigned literal, __ev64_opaque__ b );

__ev64_opaque__ __ev_sum2his( __ev64_opaque__ a );
__ev64_opaque__ __ev_sum2hisa( __ev64_opaque__ a );
__ev64_opaque__ __ev_sum2hisaaw( __ev64_opaque__ a );
__ev64_opaque__ __ev_sum2hs( __ev64_opaque__ a );
__ev64_opaque__ __ev_sum2hsa( __ev64_opaque__ a );
__ev64_opaque__ __ev_sum2hsaaw( __ev64_opaque__ a );
__ev64_opaque__ __ev_sum2hu( __ev64_opaque__ a );
__ev64_opaque__ __ev_sum2hua( __ev64_opaque__ a );
__ev64_opaque__ __ev_sum2huaaw( __ev64_opaque__ a );
__ev64_opaque__ __ev_sum4bs( __ev64_opaque__ a );
__ev64_opaque__ __ev_sum4bsa( __ev64_opaque__ a );
__ev64_opaque__ __ev_sum4bsaaw( __ev64_opaque__ a );
__ev64_opaque__ __ev_sum4bu( __ev64_opaque__ a );
__ev64_opaque__ __ev_sum4bua( __ev64_opaque__ a );
__ev64_opaque__ __ev_sum4buaaw( __ev64_opaque__ a );
__ev64_opaque__ __ev_sumws( __ev64_opaque__ a );
__ev64_opaque__ __ev_sumwsa( __ev64_opaque__ a );
__ev64_opaque__ __ev_sumwsaa( __ev64_opaque__ a );
__ev64_opaque__ __ev_sumwu( __ev64_opaque__ a );
__ev64_opaque__ __ev_sumwua( __ev64_opaque__ a );
__ev64_opaque__ __ev_sumwuaa( __ev64_opaque__ a );

__ev64_opaque__ __ev_swapbhilo( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_swapblohi( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_swaphe( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_swaphhi( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_swaphhilo( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_swaphlo( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_swaphlohi( __ev64_opaque__ a, __ev64_opaque__ b );
__ev64_opaque__ __ev_swapho( __ev64_opaque__ a, __ev64_opaque__ b );

__ev64_opaque__ __ev_unpkhibsi( __ev64_opaque__ a );
__ev64_opaque__ __ev_unpkhibui( __ev64_opaque__ a );
__ev64_opaque__ __ev_unpkhihf( __ev64_opaque__ a );
__ev64_opaque__ __ev_unpkhihsi( __ev64_opaque__ a );
__ev64_opaque__ __ev_unpkhihui( __ev64_opaque__ a );
__ev64_opaque__ __ev_unpkhiwgsf( __ev64_opaque__ a );
__ev64_opaque__ __ev_unpklobsi( __ev64_opaque__ a );
__ev64_opaque__ __ev_unpklobui( __ev64_opaque__ a );
__ev64_opaque__ __ev_unpklohf( __ev64_opaque__ a );
__ev64_opaque__ __ev_unpklohsi( __ev64_opaque__ a );
__ev64_opaque__ __ev_unpklohui( __ev64_opaque__ a );
__ev64_opaque__ __ev_unpklowgsf( __ev64_opaque__ a );
```

```
__ev64_opaque__ __ev_xor( __ev64_opaque__ a, __ev64_opaque__ b );

__ev64_opaque__ __ev_xtrb(__ev64_opaque__ a,3-bit unsigned literal,3-bit unsigned literal);
__ev64_opaque__ __ev_xtrd( __ev64_opaque__ a, __ev64_opaque__ b, 3-bit unsigned literal);
__ev64_opaque__ __ev_xtrh(__ev64_opaque__ a,2-bit unsigned literal,2-bit unsigned literal);


# 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);

# 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_lbbsplatb( uint8_t * p, 5-bit unsigned literal );
__ev64_opaque__ __ev_lbbsplatbu( uint8_t * &p, 5-bit unsigned literal );
__ev64_opaque__ __ev_lbbsplatbx( uint8_t * p, int32_t offset );
__ev64_opaque__ __ev_lbbsplatbmx( uint8_t * &p, int32_t offset );
__ev64_opaque__ __ev_ldb( __ev64_opaque__ * p, 5-bit unsigned literal );
__ev64_opaque__ __ev_ldbu( __ev64_opaque__ * &p, 5-bit unsigned literal );
__ev64_opaque__ __ev_ldbx( __ev64_opaque__ * p, int32_t offset );
__ev64_opaque__ __ev_ldbmx( __ev64_opaque__ * &p, int32_t offset );
__ev64_opaque__ __ev_ldd( __ev64_opaque__ * p, 5-bit unsigned literal );
__ev64_opaque__ __ev_lddu( __ev64_opaque__ * &p, 5-bit unsigned literal );
__ev64_opaque__ __ev_lddx( __ev64_opaque__ * p, int32_t offset );
__ev64_opaque__ __ev_lddmx( __ev64_opaque__ * &p, int32_t offset );
__ev64_opaque__ __ev_ldh( __ev64_opaque__ * p, 5-bit unsigned literal );
__ev64_opaque__ __ev_ldhu( __ev64_opaque__ * &p, 5-bit unsigned literal );
__ev64_opaque__ __ev_ldhx( __ev64_opaque__ * p, int32_t offset );
__ev64_opaque__ __ev_ldhmx( __ev64_opaque__ * &p, int32_t offset );
__ev64_opaque__ __ev_ldw( __ev64_opaque__ * p, 5-bit unsigned literal );
__ev64_opaque__ __ev_ldwu( __ev64_opaque__ * &p, 5-bit unsigned literal );
__ev64_opaque__ __ev_ldwx( __ev64_opaque__ * p, int32_t offset );
__ev64_opaque__ __ev_ldwmx( __ev64_opaque__ * &p, int32_t offset );
__ev64_opaque__ __ev_lhhesplat( uint16_t * p, 5-bit unsigned literal );
__ev64_opaque__ __ev_lhhesplatu( uint16_t * &p, 5-bit unsigned literal );
__ev64_opaque__ __ev_lhhesplatx( uint16_t * p, int32_t offset );
__ev64_opaque__ __ev_lhhesplatmx( uint16_t * &p, int32_t offset );
__ev64_opaque__ __ev_lhhossplat( uint16_t * p, 5-bit unsigned literal );
__ev64_opaque__ __ev_lhhossplatu( uint16_t * &p, 5-bit unsigned literal );
__ev64_opaque__ __ev_lhhossplatx( uint16_t * p, int32_t offset );
__ev64_opaque__ __ev_lhhossplatmx( uint16_t * &p, int32_t offset );
__ev64_opaque__ __ev_lhhousplat( uint16_t * p, 5-bit unsigned literal );
__ev64_opaque__ __ev_lhhousplatu( uint16_t * &p, 5-bit unsigned literal );
__ev64_opaque__ __ev_lhhousplatx( uint16_t * p, int32_t offset );
__ev64_opaque__ __ev_lhhousplatmx( uint16_t * &p, int32_t offset );
__ev64_opaque__ __ev_lhhsplath( uint16_t * p, 5-bit unsigned literal );
__ev64_opaque__ __ev_lhhsplathu( uint16_t * &p, 5-bit unsigned literal );
__ev64_opaque__ __ev_lhhsplathx( uint16_t * p, int32_t offset );
__ev64_opaque__ __ev_lhhsplathmx( uint16_t * &p, int32_t offset );
__ev64_opaque__ __ev_lwbe( uint32_t * p, 5-bit unsigned literal );
__ev64_opaque__ __ev_lwbeu( uint32_t * &p, 5-bit unsigned literal );
__ev64_opaque__ __ev_lwbex( uint32_t * p, int32_t offset );
__ev64_opaque__ __ev_lwbemx( uint32_t * &p, int32_t offset );
__ev64_opaque__ __ev_lwbos( uint32_t * p, 5-bit unsigned literal );
__ev64_opaque__ __ev_lwbosu( uint32_t * &p, 5-bit unsigned literal );
__ev64_opaque__ __ev_lwbosx( uint32_t * p, int32_t offset );
__ev64_opaque__ __ev_lwbosmx( uint32_t * &p, int32_t offset );
__ev64_opaque__ __ev_lwbou( uint32_t * p, 5-bit unsigned literal );
__ev64_opaque__ __ev_lwbouu( uint32_t * &p, 5-bit unsigned literal );
```

```
__ev64_opaque__ __ev_lwboux( uint32_t * p, int32_t offset );
__ev64_opaque__ __ev_lwboumx( uint32_t * &p, int32_t offset );
__ev64_opaque__ __ev_lwbsplatw( uint32_t * p, 5-bit unsigned literal );
__ev64_opaque__ __ev_lwbsplatwu( uint32_t * &p, 5-bit unsigned literal );
__ev64_opaque__ __ev_lwbsplatwx( uint32_t * p, int32_t offset );
__ev64_opaque__ __ev_lwbsplatwmx( uint32_t * &p, int32_t offset );
__ev64_opaque__ __ev_lwhe( uint32_t * p, 5-bit unsigned literal );
__ev64_opaque__ __ev_lwheu( uint32_t * &p, 5-bit unsigned literal );
__ev64_opaque__ __ev_lwhex( uint32_t * p, int32_t offset );
__ev64_opaque__ __ev_lwhemx( uint32_t * &p, int32_t offset );
__ev64_opaque__ __ev_lwhos( uint32_t * p, 5-bit unsigned literal );
__ev64_opaque__ __ev_lwhosu( uint32_t * &p, 5-bit unsigned literal );
__ev64_opaque__ __ev_lwhosx( uint32_t * p, int32_t offset );
__ev64_opaque__ __ev_lwhosmx( uint32_t * &p, int32_t offset );
__ev64_opaque__ __ev_lwhou( uint32_t * p, 5-bit unsigned literal );
__ev64_opaque__ __ev_lwhouu( uint32_t * &p, 5-bit unsigned literal );
__ev64_opaque__ __ev_lwhoux( uint32_t * p, int32_t offset );
__ev64_opaque__ __ev_lwhoumx( uint32_t * &p, int32_t offset );
__ev64_opaque__ __ev_lwhsplat( uint32_t * p, 5-bit unsigned literal );
__ev64_opaque__ __ev_lwhsplatu( uint32_t * &p, 5-bit unsigned literal );
__ev64_opaque__ __ev_lwhsplatw( uint32_t * p, 5-bit unsigned literal );
__ev64_opaque__ __ev_lwhsplatwu( uint32_t * &p, 5-bit unsigned literal );
__ev64_opaque__ __ev_lwhsplatwx( uint32_t * p, int32_t offset );
__ev64_opaque__ __ev_lwhsplatwmx( uint32_t * &p, int32_t offset );
__ev64_opaque__ __ev_lwhsplatx( uint32_t * p, int32_t offset );
__ev64_opaque__ __ev_lwhsplatmx( uint32_t * &p, int32_t offset );
__ev64_opaque__ __ev_lwwsplat( uint32_t * p, 5-bit unsigned literal );
__ev64_opaque__ __ev_lwwsplatu( uint32_t * &p, 5-bit unsigned literal );
__ev64_opaque__ __ev_lwwsplatx( uint32_t * p, int32_t offset );
__ev64_opaque__ __ev_lwwsplatmx( uint32_t * &p, int32_t offset );

void __ev_stdb( __ev64_opaque__ a, __ev64_opaque__ * p, 5-bit unsigned literal );
void __ev_stdbu( __ev64_opaque__ a, __ev64_opaque__ * &p, 5-bit unsigned literal );
void __ev_stdbx( __ev64_opaque__ a, __ev64_opaque__ * p, int32_t offset );
void __ev_stdbmx( __ev64_opaque__ a, __ev64_opaque__ * &p, int32_t offset );
void __ev_stdd( __ev64_opaque__ a, __ev64_opaque__ * p, 5-bit unsigned literal );
void __ev_stddu( __ev64_opaque__ a, __ev64_opaque__ * &p, 5-bit unsigned literal );
void __ev_stddx( __ev64_opaque__ a, __ev64_opaque__ * p, int32_t offset );
void __ev_stddmx( __ev64_opaque__ a, __ev64_opaque__ * &p, int32_t offset );
void __ev_stdh( __ev64_opaque__ a, __ev64_opaque__ * p, 5-bit unsigned literal );
void __ev_stdhu( __ev64_opaque__ a, __ev64_opaque__ * &p, 5-bit unsigned literal );
void __ev_stdhx( __ev64_opaque__ a, __ev64_opaque__ * p, int32_t offset );
void __ev_stdhmx( __ev64_opaque__ a, __ev64_opaque__ * &p, int32_t offset );
void __ev_stdw( __ev64_opaque__ a, __ev64_opaque__ * p, 5-bit unsigned literal );
void __ev_stdwu( __ev64_opaque__ a, __ev64_opaque__ * &p, 5-bit unsigned literal );
void __ev_stdwx( __ev64_opaque__ a, __ev64_opaque__ * p, int32_t offset );
void __ev_stdwmx( __ev64_opaque__ a, __ev64_opaque__ * &p, int32_t offset );
void __ev_sthb( __ev64_opaque__ a, uint16_t * p, 5-bit unsigned literal );
void __ev_sthbu( __ev64_opaque__ a, uint16_t * &p, 5-bit unsigned literal );
void __ev_sthbx( __ev64_opaque__ a, uint16_t * p, int32_t offset );
void __ev_sthbmx( __ev64_opaque__ a, uint16_t * &p, int32_t offset );
void __ev_stwb( __ev64_opaque__ a, uint32_t * p, 5-bit unsigned literal );
void __ev_stwbu( __ev64_opaque__ a, uint32_t * &p, 5-bit unsigned literal );
void __ev_stwbx( __ev64_opaque__ a, uint32_t * p, int32_t offset );
void __ev_stwbmx( __ev64_opaque__ a, uint32_t * &p, int32_t offset );
void __ev_stwbe( __ev64_opaque__ a, uint32_t * p, 5-bit unsigned literal );
void __ev_stwbeu( __ev64_opaque__ a, uint32_t * &p, 5-bit unsigned literal );
```

```
void __ev_stwbex( __ev64_opaque__ a, uint32_t * p, int32_t offset );
void __ev_stwbemx( __ev64_opaque__ a, uint32_t * &p, int32_t offset );
void __ev_stwbo( __ev64_opaque__ a, uint32_t * p, 5-bit unsigned literal );
void __ev_stwbou( __ev64_opaque__ a, uint32_t * &p, 5-bit unsigned literal );
void __ev_stwbox( __ev64_opaque__ a, uint32_t * p, int32_t offset );
void __ev_stwbomx( __ev64_opaque__ a, uint32_t * &p, int32_t offset );
void __ev_stwhe( __ev64_opaque__ a, uint32_t * p, 5-bit unsigned literal );
void __ev_stwheu( __ev64_opaque__ a, uint32_t * &p, 5-bit unsigned literal );
void __ev_stwhex( __ev64_opaque__ a, uint32_t * p, int32_t offset );
void __ev_stwhemx( __ev64_opaque__ a, uint32_t * &p, int32_t offset );
void __ev_stwho( __ev64_opaque__ a, uint32_t * p, 5-bit unsigned literal );
void __ev_stwhou( __ev64_opaque__ a, uint32_t * &p, 5-bit unsigned literal );
void __ev_stwhox( __ev64_opaque__ a, uint32_t * p, int32_t offset );
void __ev_stwhomx( __ev64_opaque__ a, uint32_t * &p, int32_t offset );
void __ev_stwwe( __ev64_opaque__ a, uint32_t * p, 5-bit unsigned literal );
void __ev_stwweu( __ev64_opaque__ a, uint32_t * &p, 5-bit unsigned literal );
void __ev_stwwex( __ev64_opaque__ a, uint32_t * p, int32_t offset );
void __ev_stwwemx( __ev64_opaque__ a, uint32_t * &p, int32_t offset );
void __ev_stwwo( __ev64_opaque__ a, uint32_t * p, 5-bit unsigned literal );
void __ev_stwwou( __ev64_opaque__ a, uint32_t * &p, 5-bit unsigned literal );
void __ev_stwwox( __ev64_opaque__ a, uint32_t * p, int32_t offset );
void __ev_stwwomx( __ev64_opaque__ a, uint32_t * &p, int32_t offset );

//***************************

// 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 );

// 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 );

// 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 );

// 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 );

// 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 );

// 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_mwhssiaaw( __ev64_opaque__ a, __ev64_opaque__ b ) {
 __ev64_opaque__ temp = __ev_mwhsmi(a,b);
 __ev_addssiaaw(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_mwhssianw( __ev64_opaque__ a, __ev64_opaque__ b ) {
 __ev64_opaque__ temp = __ev_mwhsmi(a,b);
 __ev_subfssiaaw(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.


// 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 );

// maps to __ev_mwumiaa
 __ev64_opaque__ __ev_mwumfaa( __ev64_opaque__ a, __ev64_opaque__ b );

// maps to __ev_mwumian
 __ev64_opaque__ __ev_mwumfan( __ev64_opaque__ a, __ev64_opaque__ b );

// 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 );

// 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_fsadd( __ev64_opaque__ a, __ev64_opaque__ b);
__ev64_opaque__ __ev_fsaddsub( __ev64_opaque__ a, __ev64_opaque__ b);
__ev64_opaque__ __ev_fsaddsubx( __ev64_opaque__ a, __ev64_opaque__ b);
__ev64_opaque__ __ev_fsaddx( __ev64_opaque__ a, __ev64_opaque__ b);
__ev64_opaque__ __ev_fscfsf( __ev64_opaque__ a);
__ev64_opaque__ __ev_fscfsi( __ev64_opaque__ a);
__ev64_opaque__ __ev_fscfuf( __ev64_opaque__ a);
__ev64_opaque__ __ev_fscfui( __ev64_opaque__ a);
__ev64_opaque__ __ev_fsctsf( __ev64_opaque__ a);
__ev64_opaque__ __ev_fsctsi( __ev64_opaque__ a);
__ev64_opaque__ __ev_fsctsiz( __ev64_opaque__ a);
__ev64_opaque__ __ev_fsctuf( __ev64_opaque__ a);
__ev64_opaque__ __ev_fsctui( __ev64_opaque__ a);
__ev64_opaque__ __ev_fsctuiz( __ev64_opaque__ a);
__ev64_opaque__ __ev_fsdiff( __ev64_opaque__ a, __ev64_opaque__ b);
__ev64_opaque__ __ev_fsdiffsum( __ev64_opaque__ a, __ev64_opaque__ b);
__ev64_opaque__ __ev_fsdiv( __ev64_opaque__ a, __ev64_opaque__ b);
__ev64_opaque__ __ev_fsmax( __ev64_opaque__ a, __ev64_opaque__ b);
__ev64_opaque__ __ev_fsmin( __ev64_opaque__ a, __ev64_opaque__ b);
__ev64_opaque__ __ev_fsmul( __ev64_opaque__ a, __ev64_opaque__ b);
__ev64_opaque__ __ev_fsmule( __ev64_opaque__ a, __ev64_opaque__ b);
__ev64_opaque__ __ev_fsmulo( __ev64_opaque__ a, __ev64_opaque__ b);
__ev64_opaque__ __ev_fsmulx( __ev64_opaque__ a, __ev64_opaque__ b);
__ev64_opaque__ __ev_fsnabs( __ev64_opaque__ a)
__ev64_opaque__ __ev_fsneg( __ev64_opaque__ a);
__ev64_opaque__ __ev_fssqrt( __ev64_opaque__ a);
__ev64_opaque__ __ev_fssub( __ev64_opaque__ a, __ev64_opaque__ b);
__ev64_opaque__ __ev_fssubadd( __ev64_opaque__ a, __ev64_opaque__ b);
__ev64_opaque__ __ev_fssubaddx( __ev64_opaque__ a, __ev64_opaque__ b);
__ev64_opaque__ __ev_fssubx( __ev64_opaque__ a, __ev64_opaque__ b);
__ev64_opaque__ __ev_fssum( __ev64_opaque__ a, __ev64_opaque__ b);
__ev64_opaque__ __ev_fssumdiff( __ev64_opaque__ a, __ev64_opaque__ b);
```

```
# COMPARE PREDICATES
```

NOTE

The __ev_select_* operations work much like the ? : operator does in C. For example:

```
__ev_select_fs_gts(a,b,c,d) maps to the logical expression a > b ? c : 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);
```

Chapter 4
Additional Operations

4.1 Data Manipulation

The intrinsics in Section Chapter 3, "SPE2 Operations" act like functions with parameters that are passed by value (except for the update forms of load and store intrinsics which are noted). 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.

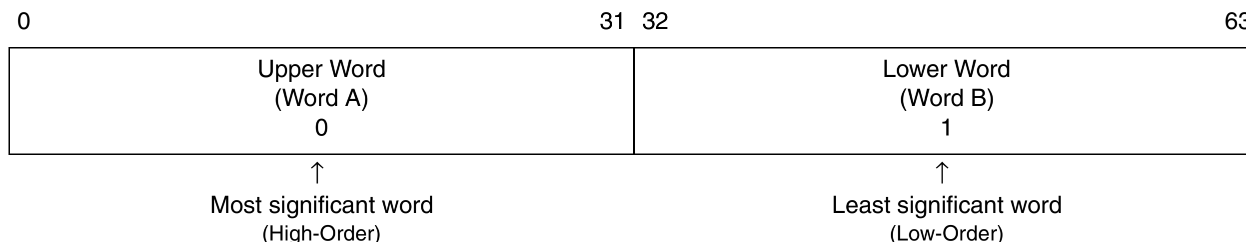


Figure 4-1. Big-Endian Word Ordering

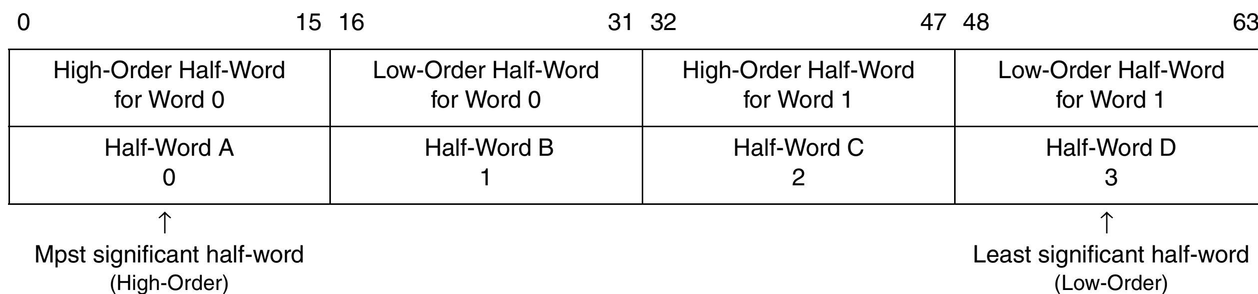


Figure 4-2. Big-Endian Half-Word Ordering

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_u8( uint8_t a, uint8_t b, uint8_t c, uint8_t d, \
                     uint8_t e, uint8_t f, uint8_t g, uint8_t h);
__ev64_opaque__ __ev_create_s8( int8_t a, int8_t b, int8_t c, int8_t d, \
                     int8_t e, int8_t f, int8_t g, int8_t h);
__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, either 0, 1, 2, or 3 for half-words, and either 0, 1, 2, 3, 4, 5, 6, or 7 for bytes.

```
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 );
uint8_t __ev_get_u8( __ev64_opaque__ a, uint32_t pos );
int8_t  __ev_get_s8( __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 or 0, 1, 2, 3, 4, 5, 6, or 7 for bytes.

```
__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 );
__ev64_opaque__ __ev_set_u8( __ev64_opaque__ a, uint8_t b, uint32_t pos );
__ev64_opaque__ __ev_set_s8( __ev64_opaque__ a, int8_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.

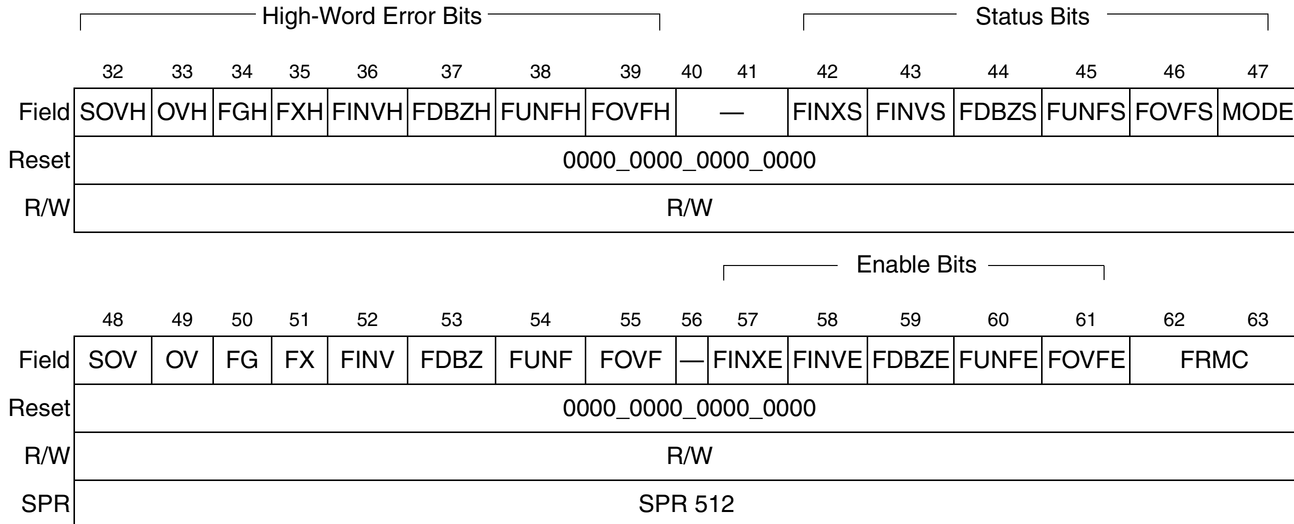


Figure 4-3. Signal Processing and Embedded Floating-Point Status and Control Register (SPEFSCR)

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 I FGH I FXH I FG I 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 I FDBZH I 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>
int8_t atosfix8(const char *str);
int16_t atosfix16(const char *str);
int32_t atosfix32(const char *str);
int64_t atosfix64(const char *str);

uint8_t atoufix8(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:

- 8-bit signed fixed-point number
- 16-bit signed fixed-point number
- 32-bit signed fixed-point number
- 64-bit signed fixed-point number
- 8-bit unsigned 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 int8_t, int16_t, int32_t, int64_t, uint8_t, uint16_t, uint32_t, and uint64_t, respectively.

Except for the behavior on error, they are equivalent to the following:

```
atosfix8: strtosfix8(str, (char **)NULL)
atosfix16: strtosfix16(str, (char **)NULL)
atosfix32: strtosfix32(str, (char **)NULL)
atosfix64: strtosfix64(str, (char **)NULL)
atoufix8: strtoufix8(str, (char **)NULL)
atoufix16: strtoufix16(str, (char **)NULL)
atoufix32: strtoufix32(str, (char **)NULL)
atoufix64: strtoufix64(str, (char **)NULL)
```

```
#include <spe.h>
int8_t strtosfix8(const char *str, char **endptr);
int16_t strtosfix16(const char *str, char **endptr);
int32_t strtosfix32(const char *str, char **endptr);
int64_t strtosfix64(const char *str, char **endptr);

uint8_t strtoufix8(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 strtosfix8, strtosfix16, strtosfix32, strtosfix64, strtoufix8, strtoufix16, strtoufix32, strtoufix64 functions convert the initial portion of the string to which str points to the following numbers:

- 8-bit signed fixed-point number
- 16-bit signed fixed-point number
- 32-bit signed fixed-point number
- 64-bit signed fixed-point number
- 8-bit unsigned 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 int8_t, int16_t, int32_t, int64_t, uint8_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 SPE2 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 SPE2 data types except __ev64_opaque__ can be used to qualify the array initialization.

5.1.2 Array Initialization of SPE2 Data Types

The following examples show array initialization of SPE2 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 SPE2 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 SPE2 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 SPE2 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 SPE2 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 SPE2 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 SPE2 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 *Enhanced Signal Processing Engine Auxiliary Processing Unit Programming Interface Manual*.

Revision	Date	Description
0.904-1	8/25/06	• Initial version -- Based on SPEPIM Rev 0 and Hardware Specification SPE2 APU Rev 0.904
0.904-2	8/30/06	• added confidential markings (FCP).
0.93-1	10/2007	• Changes to match Hardware Specification SPE2 APU Rev 0.93 • Added Embedded Floating Point-2 support corresponding to Hardware Specification EFP2 Rev 1.3
0.94-1	10/2008	• Corrected section 3.7 Basic Instruction Mapping to match number of parameters described in section 3.6 • Changes to match Hardware Specification SPE2 APU Rev 0.94 and EFP2 Rev 1.4 • Added vectors containing 8-bit data types.
1.0-1	10/2011	• Added new intrinsics to match SPE2 APU Rev 1.0 and EFP2 Rev 1.4 in section 3.7.1 • Modified section 3.2.2 • Added section 3.2.3.

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

Symbols

__ev_dotpwgasmfra, 3-348

A

AA instruction field, 3-8
ABI, 1-1, 2-1, 4-10
Accumulator (ACC), 3-3
Address operator, 2-3
Alignment, 2-2
APU
  accumulator, 3-3
  registers, 3-1
Array initialization, 5-2
Assembly language interface, 1-1
Assignment, 2-3

B

BA instruction field, 3-8
BD instruction field, 3-9
BFA instruction field, 3-9
BI instruction field, 3-9
Big-endian, 4-1, 1-3
BO instruction field, 3-9
brinc, 3-17
BT instruction field, 3-9

C

C or C++ languages, 2-1
CIA, 3-12
circinc, 3-18
Convert intrinsics, 4-2
CT instruction field, 3-9

D

D instruction field, 3-9
Data types, 2-1, 5-1
DE instruction field, 3-9
DES instruction field, 3-9

E

E instruction field, 3-9
Embedded floating-point status and control register, 4-7
Endian mode, 1-1
Enhanced Signal processing engine
  APU registers, 3-1
EnhancedSignal processing engine, APU registers, 3-1
evabs, 3-20
evabsb, 3-21
evabsbs, 3-22
evabsd, 3-23
evabsdifsb, 3-24
evabsdifsh, 3-25
evabsdifsw, 3-26
evabsdifub, 3-27
evabsdifuh, 3-28
evabsdifuw, 3-29
evabsds, 3-30
evabsh, 3-31
evabshs, 3-32
evabss, 3-33
evadd2subf2h, 3-34
evadd2subf2hss, 3-35
evaddb, 3-37
evaddbss, 3-38
evaddbus, 3-39
evaddd, 3-40
evadddss, 3-41
evadddus, 3-42
evaddh, 3-43
evaddhhisw, 3-44
evaddhhiuw, 3-45
evaddhlosw, 3-46
evaddhlouw, 3-47
evaddhss, 3-48
evaddhus, 3-49
evaddhx, 3-50
evaddhxss, 3-51
evaddhxus, 3-52
evaddib, 3-53
evaddih, 3-54
evaddiw, 3-55
evaddsmiaa, 3-36
evaddsmiaaw, 3-14, 3-56
evaddssiaa, 3-57
evaddssiaaw, 3-58
evaddsubfh, 3-59
evaddsubfhss, 3-60
evaddsubfhx, 3-61
evaddsubfhxss, 3-62

evaddsubfw, 3-63
evaddsubfwss, 3-64
evaddsubfwx, 3-65
evaddsubfwxss, 3-66
evaddumiaaw, 3-67
evaddusiaa, 3-68
evaddusiaaw, 3-69
evaddw, 3-70, 3-72
evaddwogsf, 3-73
evaddwogsi, 3-74
evaddwss, 3-75
evaddwus, 3-76
evaddwwegsf, 3-71
evaddwx, 3-77
evaddwxss, 3-78
evaddwxus, 3-79
evand, 3-85
evandc, 3-86
evavgbs, 3-87
evavgbsr, 3-88
evavgbu, 3-89
evavgbur, 3-90
evavgds, 3-91
evavgdsr, 3-92
evavgdu, 3-93
evavgdur, 3-94
evavghs, 3-95
evavghsr, 3-96
evavghu, 3-97
evavghur, 3-98
evavgws, 3-99
evavgwsr, 3-100
evavgwu, 3-101
evavgwur, 3-102
evclrbe, 3-103
evclrbo, 3-104
evclrh, 3-105
evcmpeq, 3-80, 3-106, 3-779, 3-949
evcmpgts, 3-81, 3-107, 3-780, 3-950, 3-951
evcmpgtu, 3-82, 3-108, 3-781, 3-951
evcmplts, 3-83, 3-109, 3-782, 3-952
evcmpltu, 3-84, 3-110, 3-783, 3-953
evcntlsh, 3-111
evcntlsw, 3-112
evcntlzh, 3-113
evcntlzw, 3-114
evdiff2hisa, 3-115
evdiff2hisaaw, 3-116
evdivs, 3-117
evdivu, 3-118
evdivws, 3-119
evdivwsf, 3-120
evdivwu, 3-121, 3-122
evdlveb, 3-123
evdlveh, 3-124
evdlveob, 3-125
evdlveoh, 3-126
evdlvob, 3-127
evdlvoeb, 3-128
evdlvoeh, 3-129
evdlvoh, 3-130
evdotp4hgasmf, 3-131
evdotp4hgasmfaa, 3-132
evdotp4hgasmfaa3, 3-133
evdotp4hgasmi, 3-135
evdotp4hgasmiaa, 3-136
evdotp4hgasmiaa3, 3-137
evdotp4hgasumi, 3-138
evdotp4hgasumiaa3, 3-140
evdotp4hgaumi, 3-141
evdotp4hgaumiaa, 3-139, 3-142
evdotp4hgaumiaa3, 3-143
evdotp4hgssmfa, 3-144
evdotp4hgssmfaa, 3-146
evdotp4hgssmfaa3, 3-148
evdotp4hgssmia, 3-150
evdotp4hgssmiaa, 3-152
evdotp4hgssmiaa3, 3-154
evdotp4hxgasmfa, 3-156
evdotp4hxgasmfaa, 3-158
evdotp4hxgasmfaa3, 3-160
evdotp4hxgasmia, 3-162
evdotp4hxgasmiaa, 3-164
evdotp4hxgasmiaa3, 3-166
evdotp4hxgssmfa, 3-168
evdotp4hxgssmfaa, 3-170
evdotp4hxgssmfaa3, 3-172
evdotp4hxgssmia, 3-174
evdotp4hxgssmiaa, 3-176
evdotp4hxgssmiaa3, 3-178
evdotpbasmia, 3-180
evdotpbasmiaaw, 3-181
evdotpbasmiaaw3, 3-182
evdotpbasumi, 3-184
evdotpbasumiaaw, 3-185
evdotpbasumiaaw3, 3-186
evdotpbaumi, 3-188
evdotpbaumiaaw, 3-189
evdotpbaumiaaw3, 3-190
evdotphasmi, 3-192
evdotphasmiaaw, 3-193, 3-194
evdotphassf, 3-196
evdotphassfaaw, 3-198
evdotphassfaaw3, 3-200

evdotphassfra, 3-202
evdotphassfraaw, 3-204
evdotphassfraaw3, 3-206
evdotphassi, 3-208
evdotphassiaaw, 3-209
evdotphassiaaw3, 3-210
evdotphasumia, 3-212
evdotphasumiaaw, 3-213
evdotphasumiaaw3, 3-214
evdotphasusia, 3-216
evdotphasusiaaw, 3-217
evdotphasusiaaw3, 3-218
evdotphaumi, 3-220
evdotphaumiaaw, 3-221
evdotphaumiaaw3, 3-222
evdotphausia, 3-224
evdotphausiaaw, 3-225, 3-226
evdotphcssfa, 3-234
evdotphihcsmia, 3-228
evdotphihcsmiaaw, 3-230
evdotphihcsmiaaw3, 3-232
evdotphihcssfaaw, 3-236
evdotphihcssfaaw3, 3-238
evdotphihcssfra, 3-240
evdotphihcssfraaw3, 3-244
evdotphihcssia, 3-246
evdotphihcssiaaw, 3-248
evdotphihcssiaaw3, 3-250
evdotphssmia, 3-252
evdotphssmiaaw, 3-254
evdotphssmiaaw3, 3-256
evdotphsssfa, 3-258
evdotphsssfaaw, 3-242, 3-260
evdotphsssfaaw3, 3-262
evdotphsssfr, 3-264
evdotphsssfraaw, 3-266
evdotphsssfraaw3, 3-268
evdotphsssi, 3-270
evdotphsssiaaw, 3-271
evdotphsssiaaw3, 3-272
evdotplohcsmia, 3-274
evdotplohcsmiaaw, 3-276
evdotplohcsmiaaw3, 3-278
evdotplohcssf, 3-280
evdotplohcssfaaw, 3-282
evdotplohcssfaaw3, 3-284
evdotplohcssfra, 3-286
evdotplohcssfraaw, 3-288
evdotplohcssfraaw3, 3-290
evdotplohcssia, 3-292
evdotplohcssiaaw, 3-294
evdotplohcssiaaw3, 3-296
evdotpwasmi, 3-298
evdotpwasmiaa, 3-299
evdotpwasmiaa3, 3-300
evdotpwassi, 3-301
evdotpwassiaa, 3-302
evdotpwassiaa3, 3-303
evdotpwasumi, 3-305
evdotpwasumiaa, 3-306
evdotpwasumiaa3, 3-307
evdotpwasusia, 3-308, 3-309
evdotpwasusiaa3, 3-310
evdotpwaumi, 3-312
evdotpwaumiaa, 3-313
evdotpwaumiaa3, 3-314
evdotpwausi, 3-315
evdotpwausiaa, 3-316
evdotpwausiaa3, 3-317
evdotpwcsmi, 3-318
evdotpwcsmiaa3, 3-322
evdotpwcsmiaaw, 3-320
evdotpwcssfa, 3-324
evdotpwcssfaaw, 3-326
evdotpwcssfaaw3, 3-328
evdotpwcssfra, 3-330
evdotpwcssfraaw, 3-332
evdotpwcssfraaw3, 3-334
evdotpwcssia, 3-336
evdotpwcssiaaw, 3-338
evdotpwcssiaaw3, 3-340
evdotpwgasmf, 3-342
evdotpwgasmfaa, 3-344
evdotpwgasmfaa3, 3-346
evdotpwgasmfraa, 3-350
evdotpwgasmfraa3, 3-352
evdotpwgssmfa, 3-354
evdotpwgssmfaa, 3-356
evdotpwgssmfaa3, 3-358
evdotpwgssmfra, 3-360
evdotpwgssmfraa, 3-362
evdotpwgssmfraa3, 3-364
evdotpwssmi, 3-366
evdotpwssmiaa, 3-367
evdotpwssmiaa3, 3-368
evdotpwsssi, 3-369
evdotpwsssiaa, 3-370
evdotpwsssiaa3, 3-371
evdotpwxgasmfa, 3-373
evdotpwxgasmfaa, 3-375
evdotpwxgasmfaa3, 3-377
evdotpwxgasmfra, 3-379
evdotpwxgasmfraa, 3-381
evdotpwxgasmfraa3, 3-383

evdotpwxgssmfa, 3-385
evdotpwxgssmfaa, 3-387
evdotpwxgssmfaa3, 3-389
evdotpwxgssmfra, 3-391
evdotpwxgssmfraa, 3-393
evdotpwxgssmfraa3, 3-395
eveqv, 3-397
evextsb, 3-398
evextsbh, 3-399
evextsh, 3-400, 3-401, 3-402
evextsw, 3-401
evextzb, 3-402
evfsabs, 3-971
evfsadd, 3-972
evfsaddsub, 3-973
evfsaddsubx, 3-974
evfsaddx, 3-975
evfscfsf, 3-976, 3-977
evfscfsi, 3-977
evfscfuf, 3-978
evfscfui, 3-979
evfscmpeq, 3-965, 3-1004, 3-1010, 3-1016
evfscmpgt, 3-960, 3-966, 3-1005, 3-1011, 3-1017
evfscmplt, 3-959, 3-961, 3-967, 3-1006, 3-1012, 3-1018
evfsctsf, 3-980
evfsctsi, 3-981
evfsctsiz, 3-982
evfsctuf, 3-983
evfsctui, 3-984
evfsctuiz, 3-985
evfsdiff, 3-986
evfsdiffsum, 3-987
evfsdiv, 3-988
evfsmax, 3-989
evfsmin, 3-990
evfsmul, 3-991
evfsmule, 3-992
evfsmulo, 3-993
evfsmulx, 3-994
evfsnabs, 3-995
evfsneg, 3-996
evfssqrt, 3-997
evfssub, 3-998
evfssubadd, 3-999
evfssubaddx, 3-1000
evfssubx, 3-1001
evfssum, 3-1002
evfssumdiff, 3-1003
evfststeq, 3-962, 3-968, 3-1007, 3-1013, 3-1019
evfststgt, 3-963, 3-969, 3-1008, 3-1014, 3-1020
evfststlt, 3-964, 3-970, 3-1009, 3-1015, 3-1021
evilveh, 3-405
evilveoh, 3-406
evilvhih, 3-407
evilvhiloh, 3-408
evilvloh, 3-409
evilvlohih, 3-410
evilvoeh, 3-411
evilvoh, 3-412
evinsb, 3-403
evinsh, 3-404
evlbbsplatbmx, 3-414
evlbbsplatbu, 3-413
evldbmx, 3-416
evldbu, 3-415
evlddmx, 3-418
evlddu, 3-417
evldh, 3-419
evldhx, 3-420
evldw, 3-421
evldwx, 3-422
evlhhesplat, 3-423
evlhhesplatx, 3-424
evlhhossplat, 3-425
evlhhossplatx, 3-426
evlhhousplat, 3-427
evlhhousplatx, 3-428
evlhhsplath, 3-429
evlhhsplathx, 3-430
evlvsl, 3-436
evlvsr, 3-437
evlwbe, 3-438
evlwbex, 3-439
evlwbos, 3-441
evlwbosx, 3-442
evlwbou, 3-443, 3-445
evlwboux, 3-444, 3-446
evlwhe, 3-447
evlwhex, 3-448
evlwhos, 3-449
evlwhosx, 3-450
evlwhou, 3-451
evlwhoux, 3-452
evlwhsplat, 3-453
evlwhsplatw, 3-454
evlwhsplatwx, 3-455
evlwhsplatx, 3-456
evlwwsplat, 3-457
evlwwsplatx, 3-458
evmaxbpsh, 3-459, 3-460
evmaxbpuh, 3-461
evmaxbs, 3-462, 3-464, 3-465
evmaxbu, 3-463
evmaxhpsw, 3-466

evmaxhpuw, 3-467
evmaxhs, 3-468
evmaxhu, 3-469, 3-470
evmaxwpsd, 3-471
evmaxwpud, 3-472
evmaxws, 3-473
evmaxwu, 3-474
evmbesmi, 3-475
evmbesmiaah, 3-476
evmbesmianh, 3-477
evmbessiaah, 3-478
evmbessianh, 3-480
evmbesumi, 3-482
evmbesumiaah, 3-483
evmbesumianh, 3-484
evmbesusiaah, 3-485
evmbesusianh, 3-487
evmbeumi, 3-489
evmbeumiaah, 3-490
evmbeumianh, 3-491
evmbeusiaah, 3-492
evmbeusianh, 3-494
evmbosmi, 3-496
evmbosmiaah, 3-497
evmbosmianh, 3-498
evmbossiaah, 3-499
evmbossianh, 3-501
evmbosumi, 3-503
evmbosumiaah, 3-504
evmbosumianh, 3-505
evmbosusiaah, 3-506
evmbosusianh, 3-508
evmboumi, 3-510
evmboumiaah, 3-511
evmboumianh, 3-512
evmbousiaah, 3-513
evmbousianh, 3-515
evmergelohi, 3-520
evmhegsmfan, 3-522
evmhegsmiaa, 3-523
evmhegsmian, 3-524
evmhegumiaa, 3-525
evmhegumian, 3-526
evmhesmf, 3-527
evmhesmfaaw, 3-528
evmhesmfanw, 3-529
evmhesmi, 3-530
evmhesmiaaw, 3-531
evmhesmianw, 3-532
evmhessf, 3-533
evmhessfaaw, 3-535
evmhessfanw, 3-537
evmhessiaaw, 3-539
evmhessianw, 3-540
evmhesumi, 3-541
evmhesumiaaw, 3-542
evmhesumianw, 3-543
evmhesusiaaw, 3-544
evmhesusianw, 3-545
evmheumi, 3-546
evmheumiaaw, 3-547
evmheumianw, 3-548
evmheusiaaw, 3-549
evmheusianw, 3-550
evmhogsmfaa, 3-551
evmhogsmfan, 3-552
evmhogsmiaa, 3-553
evmhogsmian, 3-554
evmhogumiaa, 3-555
evmhogumian, 3-556
evmhosmf, 3-557
evmhosmfaaw, 3-558
evmhosmfanw, 3-559
evmhosmi, 3-560
evmhosmiaaw, 3-561
evmhosmianw, 3-562
evmhossf, 3-563
evmhossfaaw, 3-565
evmhossfanw, 3-566
evmhossiaaw, 3-567
evmhossianw, 3-568
evmhosumi, 3-569
evmhosumiaaw, 3-570
evmhosumianw, 3-571
evmhosusiaaw, 3-572
evmhosusianw, 3-573
evmhoumi, 3-574
evmhoumiaaw, 3-575
evmhoumianw, 3-576
evmhousiaaw, 3-577
evmhousianw, 3-578
evmhssf, 3-579
evmhssfr, 3-580
evmhssi, 3-581, 3-582, 3-583, 3-584
evminbpsh, 3-585
evminbpuh, 3-586
evminbs, 3-587
evminbu, 3-588, 3-589, 3-590
evminhpsw, 3-591
evminhpuw, 3-592
evminhs, 3-593
evminhu, 3-594
evminwpsd, 3-595
evminwpud, 3-596

evminws, 3-597
evminwu, 3-598
evmra, 3-599
evmwehgsmfr, 3-605
evmwehgsmfraa, 3-607
evmwehgsmfran, 3-609
evmwhsmf, 3-600, 3-611
evmwhsmfaaw, 3-601
evmwhsmfanw, 3-603
evmwhsmi, 3-612
evmwhssf, 3-613
evmwhssfaaw, 3-614
evmwhssfaaw3, 3-616
evmwhssfanw, 3-618
evmwhssfanw3, 3-620
evmwhssfr, 3-622
evmwhssfraaw, 3-623
evmwhssfraaw3, 3-625
evmwhssfranw, 3-627
evmwhssfranw3, 3-629
evmwhumi, 3-631
evmwlsmiaaw, 3-632
evmwlsmiaaw3, 3-633, 3-637
evmwlsmianw, 3-634, 3-638
evmwlsmianw3, 3-635, 3-639
evmwlssiaaw, 3-636
evmwlumainw, 3-643
evmwlumainw3, 3-644
evmwlumia, 3-640
evmwlumiaaw, 3-641
evmwlumiaaw3, 3-642
evmwlusiaaw, 3-645
evmwlusiaaw3, 3-646
evmwlusianw, 3-647
evmwlusianw3, 3-648
evmwohgsmfa, 3-650
evmwohgsmfaa, 3-651
evmwohgsmfan, 3-653
evmwohgsmfr, 3-655
evmwohgsmfraa, 3-657
evmwohgsmfran, 3-659
evmwsmf, 3-661
evmwsmfaa, 3-662
evmwsmfan, 3-663
evmwsmi, 3-664
evmwsmiaa, 3-665
evmwsmian, 3-666
evmwssf, 3-667
evmwssfa, 3-668
evmwssfan, 3-669
evmwssiaa, 3-670
evmwssian, 3-671
evmwssiw, 3-672
evmwumia, 3-673
evmwumiaa, 3-674
evmwumian, 3-675
evmwusiaa, 3-676
evmwusian, 3-677
evmwusiw, 3-678
evnand, 3-679
evneg, 3-680
evnegb, 3-681
evnegbo, 3-682
evnegbos, 3-683
evnegbs, 3-685
evnegd, 3-686
evnegds, 3-687
evnegh, 3-688, 3-690
evnegho, 3-689
evneghs, 3-691
evnegs, 3-692
evnegwo, 3-693
evnegwos, 3-694
evnor, 3-695
evor, 3-696
evorc, 3-697
evperm, 3-698
evperm2, 3-699
evperm3, 3-700
evpksdshefrs, 3-701
evpksdswfrs, 3-703
evpkshsbs, 3-705
evpkshsdsws, 3-704
evpkshubs, 3-706
evpkswgshefrs, 3-707
evpkswgswfrs, 3-708
evpkswshfrs, 3-710
evpkswshilvfrs, 3-712
evpkswshilvs, 3-714
evpkswshs, 3-715
evpkswuhs, 3-716
evpkuduws, 3-717
evpkuhubs, 3-718
evpkuwuhs, 3-719
evpopcntb, 3-720
evrlb, 3-721
evrlbi, 3-722
evrlh, 3-723
evrlhi, 3-724
evrlw, 3-725
evrlwi, 3-726
evrndbhss, 3-734
evrndbhus, 3-736
evrnddnw, 3-727

evrnddnwss, 3-728
evrnddw, 3-730
evrnddwss, 3-731
evrnddwus, 3-732
evrndhb, 3-733
evrndhnb, 3-738
evrndhnbss, 3-739
evrndhnbus, 3-741
evrndnwus, 3-729
evrndwh, 3-743
evrndwhss, 3-744
evrndwhus, 3-745
evrndwnh, 3-746
evrndwnhss, 3-747
evrndwnhus, 3-748
evsad2sha, 3-749
evsad2shaaw, 3-751
evsad2uha, 3-752
evsad2uhaaw, 3-753
evsad4sb, 3-755
evsad4sbaaw, 3-756
evsad4uba, 3-757
evsad4ubaaw, 3-758
evsadswa, 3-759
evsadswaa, 3-760
evsaduw, 3-761
evsaduwaa, 3-762
evsatsbub, 3-763
evsatsdsw, 3-764
evsatsduw, 3-765
evsatshsb, 3-766
evsatshub, 3-767
evsatshuh, 3-768
evsatswgsdf, 3-769
evsatswsh, 3-770
evsatswuh, 3-771
evsatswuw, 3-772
evsatubsb, 3-774
evsatuduw, 3-773
evsatuhsh, 3-775
evsatuhub, 3-776
evsatuwsw, 3-777
evsatuwuh, 3-778
evselbit, 3-784
evselbitm0, 3-785
evselbitm1, 3-786
evseteqb, 3-787
evseteqh, 3-788
evseteqw, 3-789
evsetgtbs, 3-790
evsetgtbu, 3-791
evsetgths, 3-792
evsetgthu, 3-793
evsetgtws, 3-794
evsetgtwu, 3-795
evsetltbs, 3-796
evsetltbu, 3-797
evsetlths, 3-798
evsetlthu, 3-799
evsetltws, 3-800
evsetltwu, 3-801
evsl, 3-802
evslb, 3-804
evslbi, 3-805
evslh, 3-806
evslhi, 3-807
evsli, 3-803
evsloi, 3-808
evslw, 3-809
evslwi, 3-810
evsplatb, 3-811
evsplatfi, 3-812
evsplatfiba, 3-813
evsplatfiboa, 3-814
evsplatfida, 3-815
evsplatfih, 3-816
evsplatfihoa, 3-817
evsplatfioa, 3-818
evsplath, 3-819
evsplati, 3-820
evsplatib, 3-821
evsplatibea, 3-822
evsplatid, 3-823
evsplatiea, 3-824
evsplatih, 3-825
evsplatihea, 3-826
evsrbis, 3-827
evsrbiu, 3-828
evsrbs, 3-829
evsrbu, 3-830
evsrhis, 3-831
evsrhiu, 3-832
evsrhs, 3-833
evsrhu, 3-834
evsris, 3-835
evsriu, 3-836
evsrois, 3-837
evsroiu, 3-838
evsrs, 3-839
evsru, 3-840
evsrwis, 3-841
evsrwiu, 3-842
evsrws, 3-843
evsrwu, 3-844

evstbmx, 3-856
evstdb, 3-845
evstdbx, 3-846
evstdd, 3-847
evstddx, 3-848
evstdh, 3-849
evstdhx, 3-850
evstdw, 3-851
evstdwx, 3-852
evsthb, 3-853
evsthbmx, 3-854
evstwbe, 3-857
evstwbex, 3-858
evstwbo, 3-859
evstwbox, 3-860
evstwbu, 3-855
evstwhe, 3-861
evstwhemx, 3-862
evstwho, 3-863
evstwhomx, 3-864
evstwweu, 3-865
evstwwex, 3-866
evstwwomx, 3-868
evstwwou, 3-867
evsubf2add2h, 3-869
evsubf2add2hss, 3-870
evsubfaddh, 3-871
evsubfaddhss, 3-872
evsubfaddhx, 3-873
evsubfaddhxss, 3-874
evsubfaddw, 3-875
evsubfaddwss, 3-876
evsubfaddwx, 3-877
evsubfaddwxss, 3-878
evsubfb, 3-879
evsubfbss, 3-880
evsubfbus, 3-881
evsubfd, 3-882
evsubfdss, 3-883
evsubfdus, 3-884
evsubfh, 3-885
evsubfhhisw, 3-886
evsubfhhiuw, 3-887
evsubfhlosw, 3-888
evsubfhlouw, 3-889
evsubfhss, 3-890
evsubfhus, 3-891
evsubfhx, 3-892
evsubfhxss, 3-893
evsubfhxus, 3-894
evsubfsmiaa, 3-895
evsubfsmiaaw, 3-896
evsubfssiaa, 3-897
evsubfssiaaw, 3-898
evsubfumiaaw, 3-899
evsubfusiaa, 3-900
evsubfusiaaw, 3-901
evsubfw, 3-902
evsubfwegsf, 3-903
evsubfwegsi, 3-904
evsubfwogsf, 3-905
evsubfwogsi, 3-906
evsubfwss, 3-907
evsubfwus, 3-908
evsubfwx, 3-909
evsubfwxss, 3-910
evsubfwxus, 3-911
evsubifb, 3-912
evsubifh, 3-913
evsubifw, 3-914
evsum2his, 3-915
evsum2hisaaw, 3-916
evsum2hs, 3-917
evsum2hsaaw, 3-918
evsum2hu, 3-919
evsum2huaaw, 3-920
evsum4bs, 3-921
evsum4bsaaw, 3-922
evsum4bu, 3-923
evsum4buaaw, 3-924
evsumws, 3-925
evsumwsaa, 3-926
evsumwu, 3-927
evsumwuaa, 3-928
evswapbhilo, 3-929
evswapblohi, 3-930
evswaphe, 3-931
evswaphhi, 3-932
evswaphhilo, 3-933
evswaphlo, 3-934
evswaphlohi, 3-935
evswapho, 3-936
evunpkhibsi, 3-937
evunpkhibui, 3-938
evunpkhihf, 3-939
evunpkhihsi, 3-940
evunpkhihui, 3-941, 3-942
evunpklobsi, 3-943
evunpklobui, 3-944
evunpklohf, 3-945
evunpklohsi, 3-946
evunpklohui, 3-947
evunpklowgsf, 3-948
evxor, 3-954

evxtrb, 3-955
evxtrd, 3-956
evxtrh, 3-957
Examples, programming interface, 5-1

F

Fixed-point accessors, 5-3
FXM instruction field, 3-9

G

Get intrinsics, 4-2
Get_Upper/Lower, 4-2

H

High-level language interface, 1-1, 2-1

I

Initialization, 2-4, 5-1
Instruction fields
AA, 3-8
BA, 3-8
BD, 3-9
BFA, 3-9
BI, 3-9
BO, 3-9
BT, 3-9
CT, 3-9
D, 3-9
DE, 3-9
DES, 3-9
descriptions, 3-8
E, 3-9
FXM, 3-9
LI, 3-9
LK, 3-9
MB, 3-9
mb, 3-9
ME, 3-9
me, 3-9
NB, 3-9
RA, 3-9
RB, 3-9
Rc, 3-9
RS, 3-9
RT, 3-9
SH, 3-10
sh, 3-10
SI, 3-10
SPRF, 3-10
TO, 3-10
UI, 3-10
WS, 3-10
Instruction mapping, 3-1021
Instructions
brinc, 3-17
circinc, 3-18
evabs, 3-20
evabsb, 3-21
evabsbs, 3-22
evabsd, 3-23
evabsdifsb, 3-24
evabsdifsh, 3-25
evabsdifsw, 3-26
evabsdifub, 3-27
evabsdifuh, 3-28
evabsdifuw, 3-29
evabsds, 3-30
evabsh, 3-31
evabshs, 3-32
evabss, 3-33
evadd2subf2h, 3-34
evadd2subf2hss, 3-35
evaddb, 3-37
evaddbss, 3-38
evaddbus, 3-39
evaddd, 3-40
evadddss, 3-41
evadddus, 3-42
evaddh, 3-43
evaddhhisw, 3-44
evaddhhiuw, 3-45
evaddhlosw, 3-46
evaddhlouw, 3-47
evaddhss, 3-48
evaddhus, 3-49
evaddhx, 3-50
evaddhxss, 3-51
evaddhxus, 3-52
evaddib, 3-53
evaddih, 3-54
evaddiw, 3-55
evaddsmiaa, 3-36
evaddsmiaaw, 3-14, 3-56
evaddssiaa, 3-57
evaddssiaaw, 3-58
evaddsubfh, 3-59
evaddsubfhss, 3-60
evaddsubfhx, 3-61
evaddsubfhxss, 3-62
evaddsubfw, 3-63
evaddsubfwss, 3-64
evaddsubfwx, 3-65

evaddsubfwxss, 3-66
evaddumiaaw, 3-67
evaddusiaa, 3-68
evaddusiaaw, 3-69
evaddw, 3-70
evaddwegsf, 3-71
evaddwegsi, 3-72
evaddwogsf, 3-73
evaddwogsi, 3-74
evaddwss, 3-75
evaddwus, 3-76
evaddwx, 3-77
evaddwxss, 3-78
evaddwxus, 3-79
evand, 3-85
evandc, 3-86
evavgbs, 3-87
evavgbsr, 3-88
evavgbu, 3-89
evavgbur, 3-90
evavgds, 3-91
evavgdsr, 3-92
evavgdu, 3-93
evavgdur, 3-94
evavghs, 3-95
evavghsr, 3-96
evavghu, 3-97
evavghur, 3-98
evavgws, 3-99
evavgwsr, 3-100
evavgwu, 3-101
evavgwur, 3-102
evclrbe, 3-103
evclrbo, 3-104
evclrh, 3-105
evcmpep, 3-80
evcmpeq, 3-106, 3-779, 3-949
evcmpgts, 3-81, 3-107, 3-780, 3-950, 3-951
evcmpgtu, 3-82, 3-108, 3-781, 3-951
evcmplts, 3-83, 3-109, 3-782, 3-952
evcmpltu, 3-84, 3-110, 3-783, 3-953
evcntlsh, 3-111
evcntlsw, 3-112
evcntlzh, 3-113
evcntlzw, 3-114
evdiff2his, 3-115
evdiff2hisaaw, 3-116
evdivs, 3-117
evdivu, 3-118
evdivws, 3-119
evdivwsf, 3-120
evdivwu, 3-121, 3-122
evdlveb, 3-123
evdlveh, 3-124
evdlveob, 3-125
evdlveoh, 3-126
evdlvob, 3-127
evdlvoeb, 3-128
evdlvoeh, 3-129
evdlvoh, 3-130
evdotp4hgasmf, 3-131
evdotp4hgasmfaa, 3-132
evdotp4hgasmfaa3, 3-133
evdotp4hgasmi, 3-135
evdotp4hgasmiaa, 3-136
evdotp4hgasmiaa3, 3-137
evdotp4hgasumi, 3-138
evdotp4hgasumiaa3, 3-140
evdotp4hgaumi, 3-141
evdotp4hgaumiaa, 3-139, 3-142
evdotp4hgaumiaa3, 3-143
evdotp4hgssmf, 3-144
evdotp4hgssmfaa, 3-146
evdotp4hgssmfaa3, 3-148
evdotp4hgssmi, 3-150
evdotp4hgssmiaa, 3-152
evdotp4hgssmiaa3, 3-154
evdotp4hxgasmf, 3-156
evdotp4hxgasmfaa, 3-158
evdotp4hxgasmfaa3, 3-160
evdotp4hxgasmi, 3-162
evdotp4hxgasmiaa, 3-164
evdotp4hxgasmiaa3, 3-166
evdotp4hxgssmf, 3-168
evdotp4hxgssmfaa, 3-170
evdotp4hxgssmfaa3, 3-172
evdotp4hxgssmi, 3-174
evdotp4hxgssmiaa, 3-176
evdotp4hxgssmiaa3, 3-178
evdotpbasmi, 3-180
evdotpbasmiaaw, 3-181
evdotpbasmiaaw3, 3-182
evdotpbasumi, 3-184
evdotpbasumiaaw, 3-185
evdotpbasumiaaw3, 3-186
evdotpbaumi, 3-188
evdotpbaumiaaw, 3-189
evdotpbaumiaaw3, 3-190
evdotphasmi, 3-192
evdotphasmiaaw, 3-193, 3-194
evdotphassf, 3-196
evdotphassfaaw, 3-198
evdotphassfaaw3, 3-200
evdotphassfr, 3-202

evdotphassfraaw, 3-204
evdotphassfraaw3, 3-206
evdotphassi, 3-208
evdotphassiaaw, 3-209
evdotphassiaaw3, 3-210
evdotphasumi, 3-212
evdotphasumiaaw, 3-213
evdotphasumiaaw3, 3-214
evdotphasusi, 3-216
evdotphasusiaaw, 3-217
evdotphasusiaaw3, 3-218
evdotphaumi, 3-220
evdotphaumiaaw, 3-221
evdotphaumiaaw3, 3-222
evdotphausi, 3-224
evdotphausiaaw, 3-225, 3-226
evdotphcssf, 3-234
evdotphihcsmi, 3-228
evdotphihcsmiaaw, 3-230
evdotphihcsmiaaw3, 3-232
evdotphihcssfaaw, 3-236
evdotphihcssfaaw3, 3-238
evdotphihcssfr, 3-240
evdotphihcssfraaw3, 3-244
evdotphihcssi, 3-246
evdotphihcssiaaw, 3-248
evdotphihcssiaaw3, 3-250
evdotphssmi, 3-252
evdotphssmiaaw, 3-254
evdotphssmiaaw3, 3-256
evdotphsssf, 3-258
evdotphsssfaaw, 3-242, 3-260
evdotphsssfaaw3, 3-262
evdotphsssfr, 3-264
evdotphsssfraaw, 3-266
evdotphsssfraaw3, 3-268
evdotphsssi, 3-270
evdotphsssiaaw, 3-271
evdotphsssiaaw3, 3-272
evdotplohcsmi, 3-274
evdotplohcsmiaaw, 3-276
evdotplohcsmiaaw3, 3-278
evdotplohcssf, 3-280
evdotplohcssfaaw, 3-282
evdotplohcssfaaw3, 3-284
evdotplohcssfr, 3-286
evdotplohcssfraaw, 3-288
evdotplohcssfraaw3, 3-290
evdotplohcssi, 3-292
evdotplohcssiaaw, 3-294
evdotplohcssiaaw3, 3-296
evdotpwasmi, 3-298
evdotpwasmiaa, 3-299
evdotpwasmiaa3, 3-300
evdotpwassi, 3-301
evdotpwassiaa, 3-302
evdotpwassiaa3, 3-303
evdotpwasumi, 3-305
evdotpwasumiaa, 3-306
evdotpwasumiaa3, 3-307
evdotpwasusi, 3-308
evdotpwasusia, 3-309
evdotpwasusiaa3, 3-310
evdotpwaumi, 3-312
evdotpwaumiaa, 3-313
evdotpwaumiaa3, 3-314
evdotpwausi, 3-315
evdotpwausiaa, 3-316
evdotpwausiaa3, 3-317
evdotpwcsmi, 3-318
evdotpwcsmiaa3, 3-322
evdotpwcsmiaaw, 3-320
evdotpwcssf, 3-324
evdotpwcssfaaw, 3-326
evdotpwcssfaaw3, 3-328
evdotpwcssfr, 3-330
evdotpwcssfraaw, 3-332
evdotpwcssfraaw3, 3-334
evdotpwcssi, 3-336
evdotpwcssiaaw, 3-338
evdotpwcssiaaw3, 3-340
evdotpwgasmf, 3-342
evdotpwgasmfaa, 3-344
evdotpwgasmfaa3, 3-346
evdotpwgasmfr, 3-348
evdotpwgasmfraa, 3-350
evdotpwgasmfraa3, 3-352
evdotpwgssmf, 3-354
evdotpwgssmfaa, 3-356
evdotpwgssmfaa3, 3-358
evdotpwgssmfr, 3-360
evdotpwgssmfraa, 3-362
evdotpwgssmfraa3, 3-364
evdotpwssmi, 3-366
evdotpwssmiaa, 3-367
evdotpwssmiaa3, 3-368
evdotpwsssi, 3-369
evdotpwsssiaa, 3-370
evdotpwsssiaa3, 3-371
evdotpwxgasmf, 3-373
evdotpwxgasmfaa, 3-375
evdotpwxgasmfaa3, 3-377
evdotpwxgasmfr, 3-379
evdotpwxgasmfraa, 3-381

evdotpwxgasmfraa3, 3-383
evdotpwxgssmf, 3-385
evdotpwxgssmfaa, 3-387
evdotpwxgssmfaa3, 3-389
evdotpwxgssmfr, 3-391
evdotpwxgssmfraa, 3-393
evdotpwxgssmfraa3, 3-395
eveqv, 3-397
evextsb, 3-398
evextsbh, 3-399
evextsh, 3-400, 3-401, 3-402
evextsw, 3-401
evextzb, 3-402
evfsabs, 3-971
evfsadd, 3-972
evfsaddsub, 3-973
evfsaddsubx, 3-974
evfsaddx, 3-975
evfscfsf, 3-976, 3-977
evfscfsi, 3-977
evfscfuf, 3-978
evfscfui, 3-979
evfscmpeq, 3-965, 3-1004, 3-1010, 3-1012, 3-1016
evfscmpgt, 3-960, 3-966, 3-1005, 3-1011, 3-1017
evfscmplt, 3-959, 3-961, 3-967, 3-1006, 3-1018
evfsctsf, 3-980
evfsctsi, 3-981
evfsctsiz, 3-982
evfsctuf, 3-983
evfsctui, 3-984
evfsctuiz, 3-985
evfsdiff, 3-986
evfsdiffsum, 3-987
evfsdiv, 3-988
evfsmax, 3-989
evfsmin, 3-990
evfsmul, 3-991
evfsmule, 3-992
evfsmulo, 3-993
evfsmulx, 3-994
evfsnabs, 3-995
evfsneg, 3-996
evfssqrt, 3-997
evfssub, 3-998
evfssubadd, 3-999
evfssubaddx, 3-1000
evfssubx, 3-1001
evfssum, 3-1002
evfssumdiff, 3-1003
evfststeq, 3-962, 3-968, 3-1007, 3-1013, 3-1019
evfststgt, 3-963, 3-969, 3-1008, 3-1014, 3-1020
evfststlt, 3-964, 3-970, 3-1009, 3-1015, 3-1021
evilveh, 3-405
evilveoh, 3-406
evilvhih, 3-407
evilvhiloh, 3-408
evilvloh, 3-409
evilvlohih, 3-410
evilvoeh, 3-411
evilvoh, 3-412
evinsb, 3-403
evinsh, 3-404
evlbbsplatb, 3-413
evlbbsplatbx, 3-414
evldb, 3-415
evldbx, 3-416
evldd, 3-417
evlddx, 3-418
evldh, 3-419
evldhx, 3-420
evldw, 3-421
evldwx, 3-422
evlhhesplat, 3-423
evlhhesplatx, 3-424
evlhhossplat, 3-425
evlhhossplatx, 3-426
evlhhousplat, 3-427
evlhhousplatx, 3-428
evlhhsplath, 3-429
evlhhsplathx, 3-430
evlvsl, 3-436
evlvsr, 3-437
evlwbe, 3-438
evlwbex, 3-439
evlwbos, 3-441
evlwbosx, 3-442
evlwbou, 3-443, 3-445
evlwboux, 3-444, 3-446
evlwhe, 3-447
evlwhex, 3-448
evlwhos, 3-449
evlwhosx, 3-450
evlwhou, 3-451
evlwhoux, 3-452
evlwhsplat, 3-453
evlwhsplatw, 3-454
evlwhsplatwx, 3-455
evlwhsplatx, 3-456
evlwwsplat, 3-457
evlwwsplatx, 3-458
evmaxbpsh, 3-459, 3-460
evmaxbpuh, 3-461
evmaxbs, 3-462, 3-464, 3-465
evmaxbu, 3-463

evmaxhpsw, 3-466
evmaxhpuw, 3-467
evmaxhs, 3-468
evmaxhu, 3-469, 3-470
evmaxwpsd, 3-471
evmaxwpud, 3-472
evmaxws, 3-473
evmaxwu, 3-474
evmbesmi, 3-475
evmbesmiaah, 3-476
evmbesmianh, 3-477
evmbessiaah, 3-478
evmbessianh, 3-480
evmbesumi, 3-482
evmbesumiaah, 3-483
evmbesumianh, 3-484
evmbesusiaah, 3-485
evmbesusianh, 3-487
evmbeumi, 3-489
evmbeumiaah, 3-490
evmbeumianh, 3-491
evmbeusiaah, 3-492
evmbeusianh, 3-494
evmbosmi, 3-496
evmbosmiaah, 3-497
evmbosmianh, 3-498
evmbossiaah, 3-499
evmbossianh, 3-501
evmbosumi, 3-503
evmbosumiaah, 3-504
evmbosumianh, 3-505
evmbosusiaah, 3-506
evmbosusianh, 3-508
evmboumi, 3-510
evmboumiaah, 3-511
evmboumianh, 3-512
evmbousiaah, 3-513
evmbousianh, 3-515
evmergelohi, 3-520
evmhegsmfan, 3-522
evmhegsmiaa, 3-523
evmhegsmiam, 3-524
evmhegumiaa, 3-525
evmhegumian, 3-526
evmhesmf, 3-527
evmhesmfaaw, 3-528
evmhesmfanw, 3-529
evmhesmi, 3-530
evmhesmiaaw, 3-531
evmhesmianw, 3-532
evmhessf, 3-533
evmhessfaaw, 3-535
evmhessfanw, 3-537
evmhessiaaw, 3-539
evmhessianw, 3-540
evmhesumi, 3-541
evmhesumiaaw, 3-542
evmhesumianw, 3-543
evmhesusiaaw, 3-544
evmhesusianw, 3-545
evmheumi, 3-546
evmheumiaaw, 3-547
evmheumianw, 3-548
evmheusiaaw, 3-549
evmheusianw, 3-550
evmhogsmfaa, 3-551
evmhogsmfan, 3-552
evmhogsmiaa, 3-553
evmhogsmian, 3-554
evmhogumiaa, 3-555
evmhogumian, 3-556
evmhosmf, 3-557
evmhosmfaaw, 3-558
evmhosmfanw, 3-559
evmhosmi, 3-560
evmhosmiaaw, 3-561
evmhosmianw, 3-562
evmhossf, 3-563
evmhossfaaw, 3-565
evmhossfanw, 3-566
evmhossiaaw, 3-567
evmhossianw, 3-568
evmhosumi, 3-569
evmhosumiaaw, 3-570
evmhosumianw, 3-571
evmhosusiaaw, 3-572
evmhosusianw, 3-573
evmhoumi, 3-574
evmhoumiaaw, 3-575
evmhoumianw, 3-576
evmhousiaaw, 3-577
evmhousianw, 3-578
evmhssf, 3-579
evmhssfr, 3-580
evmhssi, 3-581, 3-582, 3-583, 3-584
evminbpsh, 3-585
evminbpuh, 3-586
evminbs, 3-587
evminbu, 3-588, 3-589, 3-590
evminhpsw, 3-591
evminhpuw, 3-592
evminhs, 3-593
evminhu, 3-594
evminwpsd, 3-595

evminwpud, 3-596
evminws, 3-597
evminwu, 3-598
evmra, 3-599
evmwehgsmfr, 3-605
evmwehgsmfraa, 3-607
evmwehgsmfran, 3-609
evmwhsmf, 3-600, 3-611
evmwhsmfaaw, 3-601
evmwhsmfanw, 3-603
evmwhsmi, 3-612
evmwhssf, 3-613
evmwhssfaaw, 3-614
evmwhssfaaw3, 3-616
evmwhssfanw, 3-618
evmwhssfanw3, 3-620
evmwhssfr, 3-622
evmwhssfraaw, 3-623
evmwhssfraaw3, 3-625
evmwhssfranw, 3-627
evmwhssfranw3, 3-629
evmwhumi, 3-631
evmwlsmiaaw, 3-632
evmwlsmiaaw3, 3-633, 3-637
evmwlsmianw, 3-634, 3-638
evmwlsmianw3, 3-635, 3-639
evmwlssiaaw, 3-636
evmwlumi, 3-640
evmwlumiaaw, 3-641
evmwlumiaaw3, 3-642
evmwlumianw, 3-643
evmwlumianw3, 3-644
evmwlusiaaw, 3-645
evmwlusiaaw3, 3-646
evmwlusianw, 3-647
evmwlusianw3, 3-648
evmwohgsmf, 3-650
evmwohgsmfaa, 3-651
evmwohgsmfan, 3-653
evmwohgsmfr, 3-655
evmwohgsmfraa, 3-657
evmwohgsmfran, 3-659
evmwsmf, 3-661
evmwsmfaa, 3-662
evmwsmfan, 3-663
evmwsmi, 3-664
evmwsmiaa, 3-665
evmwsmian, 3-666
evmwssf, 3-667
evmwssfa, 3-668
evmwssfan, 3-669
evmwssiaa, 3-670
evmwssian, 3-671
evmwssiw, 3-672
evmwumi, 3-673
evmwumiaa, 3-674
evmwumian, 3-675
evmwusiaa, 3-676
evmwusian, 3-677
evmwusiw, 3-678
evnand, 3-679
evneg, 3-680
evnegb, 3-681
evnegbo, 3-682
evnegbos, 3-683
evnegbs, 3-685
evnegd, 3-686
evnegds, 3-687
evnegh, 3-688, 3-690
evnegho, 3-689
evneghs, 3-691
evnegs, 3-692
evnegwo, 3-693
evnegwos, 3-694
evnor, 3-695
evor, 3-696
evorc, 3-697
evperm, 3-698
evperm2, 3-699
evperm3, 3-700
evpksdshefrs, 3-701
evpksdswfrs, 3-703
evpkshsbs, 3-705
evpkshsdsws, 3-704
evpkshubs, 3-706
evpkspkswshfrs, 3-710
evpkswgshefrs, 3-707
evpkswgswfrs, 3-708
evpkswshilvfrs, 3-712
evpkswshilvs, 3-714
evpkswshs, 3-715
evpkswuhs, 3-716
evpkuduws, 3-717
evpkuhubs, 3-718
evpkuwuhs, 3-719
evpopcntb, 3-720
evrlb, 3-721
evrlbi, 3-722
evrlh, 3-723
evrlhi, 3-724
evrlw, 3-725
evrlwi, 3-726
evrndbhus, 3-736
evrnddnw, 3-727

evrnddnwss, 3-728
evrnddw, 3-730
evrnddwss, 3-731
evrnddwus, 3-732
evrndhb, 3-733
evrndhbss, 3-734
evrndhnb, 3-738
evrndhnbss, 3-739
evrndhnbus, 3-741
evrndnwus, 3-729
evrndwh, 3-743
evrndwhss, 3-744
evrndwhus, 3-745
evrndwnh, 3-746
evrndwnhss, 3-747
evrndwnhus, 3-748
evsad2sh, 3-749
evsad2shaaw, 3-751
evsad2uh, 3-752
evsad2uhaaw, 3-753
evsad4sb, 3-755
evsad4sbaaw, 3-756
evsad4ub, 3-757
evsad4ubaaw, 3-758
evsadsw, 3-759
evsadswaa, 3-760
evsaduw, 3-761
evsaduwaa, 3-762
evsatsbub, 3-763
evsatsdsw, 3-764
evsatsduw, 3-765
evsatshsb, 3-766
evsatshub, 3-767
evsatshuh, 3-768
evsatswgsdf, 3-769
evsatswsh, 3-770
evsatswuh, 3-771
evsatswuw, 3-772
evsatubsb, 3-774
evsatuduw, 3-773
evsatuhsh, 3-775
evsatuhub, 3-776
evsatuwsw, 3-777
evsatuwuh, 3-778
evselbit, 3-784
evselbitm0, 3-785
evselbitm1, 3-786
evseteqb, 3-787
evseteqh, 3-788
evseteqw, 3-789
evsetgtbs, 3-790
evsetgtbu, 3-791
evsetgths, 3-792
evsetgthu, 3-793
evsetgtws, 3-794
evsetgtwu, 3-795
evsetltbs, 3-796
evsetltbu, 3-797
evsetlths, 3-798
evsetlthu, 3-799
evsetltws, 3-800
evsetltwu, 3-801
evsl, 3-802
evslb, 3-804
evslbi, 3-805
evslh, 3-806
evslhi, 3-807
evsli, 3-803
evsloi, 3-808
evslw, 3-809
evslwi, 3-810
evsplatb, 3-811
evsplatfi, 3-812
evsplatfib, 3-813
evsplatfibo, 3-814
evsplatfid, 3-815
evsplatfih, 3-816
evsplatfiho, 3-817
evsplatfio, 3-818
evsplath, 3-819
evsplati, 3-820
evsplatib, 3-821
evsplatibe, 3-822
evsplatid, 3-823
evsplatie, 3-824
evsplatih, 3-825
evsplatihe, 3-826
evsrbis, 3-827
evsrbiu, 3-828
evsrbs, 3-829
evsrbu, 3-830
evsrhis, 3-831
evsrhiu, 3-832
evsrhs, 3-833
evsrhu, 3-834
evsris, 3-835
evsriu, 3-836
evsrois, 3-837
evsroiu, 3-838
evsrs, 3-839
evsru, 3-840
evsrwis, 3-841
evsrwiu, 3-842
evsrws, 3-843

evsrwu, 3-844
evstbx, 3-856
evstdb, 3-845
evstdbx, 3-846
evstdd, 3-847
evstddx, 3-848
evstdh, 3-849
evstdhx, 3-850
evstdw, 3-851
evstdwx, 3-852
evsthb, 3-853
evsthbx, 3-854
evstwb, 3-855
evstwbe, 3-857
evstwbex, 3-858
evstwbo, 3-859
evstwbox, 3-860
evstwhe, 3-861
evstwhex, 3-862
evstwho, 3-863
evstwhox, 3-864
evstwwe, 3-865
evstwwex, 3-866
evstwwo, 3-867
evstwwomx, 3-868
evsubf2add2h, 3-869
evsubf2add2hss, 3-870
evsubfaddh, 3-871
evsubfaddhss, 3-872
evsubfaddhx, 3-873
evsubfaddhxss, 3-874
evsubfaddw, 3-875
evsubfaddwss, 3-876
evsubfaddwx, 3-877
evsubfaddwxss, 3-878
evsubfb, 3-879
evsubfbss, 3-880
evsubfbus, 3-881
evsubfd, 3-882
evsubfdss, 3-883
evsubfdus, 3-884
evsubfh, 3-885
evsubfhhisw, 3-886
evsubfhhiuw, 3-887
evsubfhlosw, 3-888
evsubfhlouw, 3-889
evsubfhss, 3-890
evsubfhus, 3-891
evsubfhx, 3-892
evsubfhxss, 3-893
evsubfhxus, 3-894
evsubfsmiaa, 3-895
evsubfsmiaaw, 3-896
evsubfssiaa, 3-897
evsubfssiaaw, 3-898
evsubfumiaaw, 3-899
evsubfusiaa, 3-900
evsubfusiaaw, 3-901
evsubfw, 3-902
evsubfwegsf, 3-903
evsubfwegsi, 3-904
evsubfwogsf, 3-905
evsubfwogsi, 3-906
evsubfwss, 3-907
evsubfwus, 3-908
evsubfwx, 3-909
evsubfwxss, 3-910
evsubfwxus, 3-911
evsubifb, 3-912
evsubifh, 3-913
evsubifw, 3-914
evsum2his, 3-915
evsum2hisaaw, 3-916
evsum2hs, 3-917
evsum2hsaaw, 3-918
evsum2hu, 3-919
evsum2huaaw, 3-920
evsum4bs, 3-921
evsum4bsaaw, 3-922
evsum4bu, 3-923
evsum4buaaw, 3-924
evsumws, 3-925
evsumwsaa, 3-926
evsumwu, 3-927
evsumwuaa, 3-928
evswapbhilo, 3-929
evswapblohi, 3-930
evswaphe, 3-931
evswaphhi, 3-932
evswaphhilo, 3-933
evswaphlo, 3-934
evswaphlohi, 3-935
evswapho, 3-936
evunpkhibsi, 3-937
evunpkhibui, 3-938
evunpkhihf, 3-939
evunpkhihsi, 3-940
evunpkhihui, 3-941
evunpkhiwgsf, 3-942
evunpklobsi, 3-943
evunpklobui, 3-944
evunpklohf, 3-945
evunpklohsi, 3-946
evunpklohui, 3-947

evunpklowgsf, 3-948
evxor, 3-954
evxtrb, 3-955
evxtrd, 3-956
evxtrh, 3-957
Interface
application binary, 1-1
assembly language, 1-1
high-level language, 1-1

L

LI instruction field, 3-9
Library routines, 4-11
LK instruction field, 3-9
Load and Store Instructions, 3-4
Loads, 5-6

M

MB instruction field, 3-9
mb instruction field, 3-9
ME instruction field, 3-9
me instruction field, 3-9
Mnemonic, 3-13

N

NB instruction field, 3-9
NIA, 3-13

O

Operators, new, 2-2, 2-4

P

Pointer
arithmetic, 2-3
dereferencing, 2-3
type casting, 2-3
PowerPC, 1-i, 2-1
Prototypes for additional library routines, 2-5

R

RA instruction field, 3-9
RB instruction field, 3-9
Rc instruction field, 3-9
Reading list, 1-ii
References, 1-ii
Register allocation, 2-1
RISC, 1-i
RS instruction field, 3-9
RT instruction field, 3-9

S

Set Intrinsics, 4-4
SH instruction field, 3-10
sh instruction field, 3-10
SI instruction field, 3-10
SPRF instruction field, 3-10
Stack frame, 1-1

T

TBR instruction field, 3-10
TO instruction field, 3-10
Type casting, 2-3

U

UI instruction field, 3-10
Undefined, 3-12

V

Vector register, 1-1

W

Website, 1-i
WS instruction field, 3-10