

# Arm C Language Extensions for SVE

Version 00bet6

## Abstract

This document is a beta version of the Arm C language extensions (ACLE) for both the Arm Scalable Vector Extension (SVE) and SVE2. The language extensions have two main purposes: to provide a set of types and accessors for SVE vectors and predicates, and to provide a function interface for all relevant SVE and SVE2 instructions.

## Keywords

ACLE, SVE, SVE2, C, C++

## Non-Confidential Proprietary Notice

This document is protected by copyright and other related rights and the practice or implementation of the information contained in this document may be protected by one or more patents or pending patent applications. No part of this document may be reproduced in any form by any means without the express prior written permission of Arm. **No license, express or implied, by estoppel or otherwise to any intellectual property rights is granted by this document unless specifically stated.**

Your access to the information in this document is conditional upon your acceptance that you will not use or permit others to use the information for the purposes of determining whether implementations infringe any third party patents.

THIS DOCUMENT IS PROVIDED **AS IS**. ARM PROVIDES NO REPRESENTATIONS AND NO WARRANTIES, EXPRESS, IMPLIED OR STATUTORY, INCLUDING, WITHOUT LIMITATION, THE IMPLIED WARRANTIES OF MERCHANTABILITY, SATISFACTORY QUALITY, NON-INFRINGEMENT OR FITNESS FOR A PARTICULAR PURPOSE WITH RESPECT TO THE DOCUMENT. For the avoidance of doubt, Arm makes no representation with respect to, and has undertaken no analysis to identify or understand the scope and content of, third party patents, copyrights, trade secrets, or other rights.

This document may include technical inaccuracies or typographical errors.

TO THE EXTENT NOT PROHIBITED BY LAW, IN NO EVENT WILL ARM BE LIABLE FOR ANY DAMAGES, INCLUDING WITHOUT LIMITATION ANY DIRECT, INDIRECT, SPECIAL, INCIDENTAL, PUNITIVE, OR CONSEQUENTIAL DAMAGES, HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF ANY USE OF THIS DOCUMENT, EVEN IF ARM HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

This document consists solely of commercial items. You shall be responsible for ensuring that any use, duplication or disclosure of this document complies fully with any relevant export laws and regulations to assure that this document or any portion thereof is not exported, directly or indirectly, in violation of such export laws. Use of the word “partner” in reference to Arm's customers is not intended to create or refer to any partnership relationship with any other company. Arm may make changes to this document at any time and without notice.

If any of the provisions contained in these terms conflict with any of the provisions of any click through or signed written agreement covering this document with Arm, then the click through or

signed written agreement prevails over and supersedes the conflicting provisions of these terms. This document may be translated into other languages for convenience, and you agree that if there is any conflict between the English version of this document and any translation, the terms of the English version of the Agreement shall prevail.

The Arm corporate logo and words marked with ® or ™ are registered trademarks or trademarks of Arm Limited or its affiliates in the US and/or elsewhere. All rights reserved. Other brands and names mentioned in this document may be the trademarks of their respective owners. Please follow Arm's trademark usage guidelines at <http://www.arm.com/company/policies/trademarks>.

Copyright © 2015-2020. Arm Limited or its affiliates. All rights reserved.

Arm Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

LES-PRE-20349

## **Confidentiality Status**

This document is non-confidential. The right to use, copy and disclose this document may be subject to license restrictions in accordance with the terms of the agreement entered into by Arm and the party that Arm delivered this document to. Unrestricted Access is an Arm internal classification.

## **Product Status**

The information in this document is for a Beta product, that is a product under development.

## **Web Address**

<http://www.arm.com/>.

## **Contact Address**

Please contact [arm.acle@arm.com](mailto:arm.acle@arm.com) if you have any questions about the specification, or would like to report a defect in it.

# Table of Contents

1. About this document .....	13
1.1. Change control .....	13
1.1.1. Current status and expected changes .....	13
1.1.2. Change history .....	13
1.2. References .....	13
1.3. Terms and abbreviations .....	14
1.4. Conventions .....	14
2. Introduction .....	15
2.1. Feature macros .....	15
2.2. Header file .....	17
3. Types .....	18
3.1. Overview .....	18
3.2. Sizeless types .....	18
3.2.1. Informal definition .....	18
3.2.2. Formal definition .....	19
3.2.3. Interaction with other C and C++ extensions .....	19
3.3. Scalar types .....	19
3.4. Vector types .....	20
3.5. Predicate types .....	21
3.6. Non-member operator overloads in C++ .....	21
3.7. Fixed-length types .....	22
3.7.1. Introduction .....	22
3.7.2. The <code>__ARM_FEATURE_SVE_BITS</code> macro .....	23
3.7.3. The <code>arm_sve_vector_bits</code> attribute .....	23
4. Functions .....	30
4.1. Naming convention .....	30
4.2. Overloaded aliases .....	32
4.3. Addressing modes .....	32
4.4. Operations involving vectors and scalars .....	34
4.5. Immediate arguments .....	34
4.6. Faults and exceptions .....	34
4.7. First-faulting and non-faulting loads .....	35
5. Enum declarations .....	37
6. List of base SVE functions .....	38
6.1. Introduction .....	38
6.2. Loads .....	38
6.2.1. LD1: Unextended load .....	38
6.2.2. LD1SB: Load 8-bit data and sign-extend .....	40
6.2.3. LD1UB: Load 8-bit data and zero-extend .....	42
6.2.4. LD1SH: Load 16-bit data and sign-extend .....	43
6.2.5. LD1UH: Load 16-bit data and zero-extend .....	45
6.2.6. LD1SW: Load 32-bit data and sign-extend .....	46
6.2.7. LD1UW: Load 32-bit data and zero-extend .....	48
6.2.8. LD1RQ: Unextended load and replicate to quadword .....	49
6.2.9. LDFF1: Unextended load, first-faulting .....	49
6.2.10. LDFF1SB: Load 8-bit data and sign-extend, first-faulting .....	52
6.2.11. LDFF1UB: Load 8-bit data and zero-extend, first-faulting .....	54
6.2.12. LDFF1SH: Load 16-bit data and sign-extend, first-faulting .....	55
6.2.13. LDFF1UH: Load 16-bit data and zero-extend, first-faulting .....	57
6.2.14. LDFF1SW: Load 32-bit data and sign-extend, first-faulting .....	59
6.2.15. LDFF1UW: Load 32-bit data and zero-extend, first-faulting .....	61

6.2.16. LDNFI: Unextended load, non-faulting .....	62
6.2.17. LDNFISB: Load 8-bit data and sign-extend, non-faulting .....	63
6.2.18. LDNFUIB: Load 8-bit data and zero-extend, non-faulting .....	64
6.2.19. LDNFISH: Load 16-bit data and sign-extend, non-faulting .....	64
6.2.20. LDNFUIH: Load 16-bit data and zero-extend, non-faulting .....	65
6.2.21. LDNFISW: Load 32-bit data and sign-extend, non-faulting .....	65
6.2.22. LDNFUIW: Load 32-bit data and zero-extend, non-faulting .....	66
6.2.23. LDNT1: Unextended load, non-temporal .....	66
6.2.24. LD2: Load two-element structures into two vectors .....	67
6.2.25. LD3: Load three-element structures into three vectors .....	68
6.2.26. LD4: Load four-element structures into four vectors .....	69
6.3. Stores .....	70
6.3.1. ST1: Store one vector, with no truncation .....	70
6.3.2. ST1B: Store one vector, truncating to 8 bits .....	73
6.3.3. ST1H: Store one vector, truncating to 16 bits .....	74
6.3.4. ST1W: Store one vector, truncating to 32 bits .....	76
6.3.5. STNT1: Store one vector, with no truncation, non-temporal .....	77
6.3.6. ST2: Store two vectors into two-element structures .....	78
6.3.7. ST3: Store three vectors into three-element structures .....	79
6.3.8. ST4: Store four vectors into four-element structures .....	80
6.4. Prefetches .....	81
6.4.1. PRFB: Prefetch 8-bit data .....	81
6.4.2. PRFH: Prefetch 16-bit data .....	81
6.4.3. PRFW: Prefetch 32-bit data .....	82
6.4.4. PRFD: Prefetch 64-bit data .....	83
6.5. Address calculations .....	84
6.5.1. ADRB: Compute vector address for 8-bit data .....	84
6.5.2. ADRH: Compute vector address for 16-bit data .....	84
6.5.3. ADRW: Compute vector address for 32-bit data .....	84
6.5.4. ADRD: Compute vector address for 64-bit data .....	85
6.6. Scalar to vector operations .....	85
6.6.1. DUP: Duplicate scalar value .....	85
6.6.2. DUPQ: Duplicate scalars to every quadword of a vector .....	86
6.6.3. INDEX: Create index series .....	87
6.7. Integer arithmetic .....	88
6.7.1. ADD: Modular integer addition .....	88
6.7.2. QADD: Saturating integer addition .....	89
6.7.3. SUB: Modular integer subtraction .....	90
6.7.4. SUBR: Modular integer subtraction, reversed .....	91
6.7.5. QSUB: Saturating integer subtraction .....	93
6.7.6. ABD: Integer absolute difference .....	93
6.7.7. MUL: Integer multiplication, returning low half .....	95
6.7.8. MULH: Integer multiplication, returning high half .....	96
6.7.9. MAD: Integer addition of product (multiplicand first) .....	98
6.7.10. MLA: Integer addition of product (addend first) .....	100
6.7.11. MSB: Integer subtraction of product (multiplicand first) .....	102
6.7.12. MLS: Integer subtraction of product (minuend first) .....	104
6.7.13. DOT: Integer addition of dot product .....	107
6.7.14. DIV: Integer division .....	108
6.7.15. DIVR: Integer division, reversed .....	109
6.7.16. MAX: Integer maximum .....	110
6.7.17. MIN: Integer minimum .....	111
6.7.18. NEG: Integer negation .....	112
6.7.19. ABS: Integer absolute .....	113

6.8. Logical operations .....	114
6.8.1. AND: Bitwise AND .....	114
6.8.2. BIC: Bitwise AND NOT .....	115
6.8.3. ORR: Bitwise OR .....	116
6.8.4. EOR: Bitwise exclusive OR .....	118
6.8.5. NOT: Bitwise inverse .....	119
6.8.6. CNOT: Logical inverse .....	120
6.9. Shifts .....	121
6.9.1. LSL: Shift left .....	121
6.9.2. LSR: Logical shift right .....	123
6.9.3. ASR: Arithmetic shift right, rounding towards -Inf .....	125
6.9.4. ASRD: Arithmetic shift right, rounding towards zero .....	127
6.9.5. INSR: Shift vector and insert scalar .....	128
6.10. Integer reductions .....	128
6.10.1. ADDV: Integer addition reduction .....	128
6.10.2. MAXV: Integer maximum reduction .....	129
6.10.3. MINV: Integer minimum reduction .....	129
6.10.4. ANDV: Integer AND reduction .....	129
6.10.5. ORV: Integer OR reduction .....	130
6.10.6. EORV: Integer exclusive OR reduction .....	130
6.11. Integer comparisons .....	130
6.11.1. CMPEQ: Integer compare equal .....	130
6.11.2. CMPNE: Integer compare not equal .....	131
6.11.3. CMPLT: Integer compare less than .....	132
6.11.4. CMPLE: Integer compare less than or equal to .....	133
6.11.5. CMPGE: Integer compare greater than or equal to .....	134
6.11.6. CMPGT: Integer compare greater than .....	135
6.12. While comparisons .....	136
6.12.1. WHILELT: While incrementing variable is less than .....	136
6.12.2. WHILELE: While incrementing variable is less than or equal to .....	136
6.13. Counting bits .....	137
6.13.1. CLS: Count leading sign bits .....	137
6.13.2. CLZ: Count leading zero bits .....	138
6.13.3. CNT: Count nonzero bits .....	138
6.14. Conversion .....	139
6.14.1. EXTB: Extend from low 8 bits .....	139
6.14.2. EXTH: Extend from low 16 bits .....	140
6.14.3. EXTW: Extend from low 32 bits .....	141
6.15. Reversal .....	141
6.15.1. RBIT: Reverse bits within elements .....	141
6.15.2. REVB: Reverse bytes within elements .....	142
6.15.3. REVH: Reverse halfwords within elements .....	143
6.15.4. REVW: Reverse words within elements .....	143
6.16. Floating-point arithmetic .....	144
6.16.1. ADD: Floating-point addition .....	144
6.16.2. CADD: Floating-point complex addition with rotation .....	145
6.16.3. SUB: Floating-point subtraction .....	145
6.16.4. SUBR: Floating-point subtraction, reversed .....	146
6.16.5. ABD: Floating-point absolute difference .....	147
6.16.6. MUL: Floating-point multiplication .....	148
6.16.7. MULX: Floating-point multiplication extended .....	149
6.16.8. MAD: Fused floating-point addition of product (multiplicand first) .....	150
6.16.9. MLA: Fused floating-point addition of product (addend first) .....	152
6.16.10. CMLA: Fused floating-point complex addition of product with rotation .....	153

6.16.11. MSB: Fused floating-point subtraction of product (multiplicand first) .....	154
6.16.12. MLS: Fused floating-point subtraction of product (minuend first) .....	156
6.16.13. NMAD: Fused floating-point addition of product, negated (multiplicand first) .....	157
6.16.14. NMLA: Fused floating-point addition of product, negated (addend first) .....	159
6.16.15. NMSB: Fused floating-point subtraction of product, negated (multiplicand first) .....	160
6.16.16. NMLS: Fused floating-point subtraction of product, negated (minuend first) ...	161
6.16.17. DIV: Floating-point division .....	162
6.16.18. DIVR: Floating-point division, reversed .....	163
6.16.19. MAX: Floating-point maximum .....	164
6.16.20. MAXNM: Floating-point maximum number .....	165
6.16.21. MIN: Floating-point minimum .....	166
6.16.22. MINNM: Floating-point minimum number .....	167
6.16.23. SCALE: Floating-point adjust exponent .....	168
6.16.24. TSMUL: Floating-point trigonometric starting value .....	169
6.16.25. TMAD: Floating-point trigonometric multiply-add coefficient .....	170
6.16.26. TSSEL: Floating-point trigonometric select coefficient .....	170
6.16.27. ABS: Floating-point absolute .....	170
6.16.28. NEG: Floating-point negation .....	171
6.16.29. SQRT: Floating-point square root .....	171
6.16.30. EXPA: Floating-point exponent accelerator .....	172
6.16.31. RECPE: Floating-point reciprocal estimate .....	172
6.16.32. RECPS: Floating-point reciprocal step .....	172
6.16.33. RECPX: Floating-point reciprocal exponent .....	173
6.16.34. RSQRTE: Floating-point reciprocal square root estimate .....	173
6.16.35. RSQRTS: Floating-point reciprocal square root step .....	173
6.16.36. RINTA: Floating-point round to nearest, ties away from zero .....	173
6.16.37. RINTI: Floating-point round using current rounding mode (inexact) .....	174
6.16.38. RINTM: Floating-point round towards -Inf .....	175
6.16.39. RINTN: Floating-point round to nearest, ties to even .....	175
6.16.40. RINTP: Floating-point round towards +Inf .....	176
6.16.41. RINTX: Floating-point round using current rounding mode (exact) .....	176
6.16.42. RINTZ: Floating-point round towards zero .....	177
6.17. Floating-point reductions .....	178
6.17.1. ADDA: Left-to-right floating-point addition reduction .....	178
6.17.2. ADDV: Tree-based floating-point addition reduction .....	178
6.17.3. MAXV: Floating-point maximum reduction .....	178
6.17.4. MAXNMV: Floating-point maximum number reduction .....	178
6.17.5. MINV: Floating-point minimum reduction .....	179
6.17.6. MINNMV: Floating-point minimum number reduction .....	179
6.18. Floating-point comparisons .....	179
6.18.1. CMPEQ: Floating-point compare equal .....	179
6.18.2. CMPNE: Floating-point compare not equal .....	180
6.18.3. CMPLT: Floating-point compare less than .....	180
6.18.4. CMPLE: Floating-point compare less than or equal to .....	180
6.18.5. CMPGE: Floating-point compare greater than or equal to .....	181
6.18.6. CMPGT: Floating-point compare greater than .....	181
6.18.7. CMPUO: Floating-point compare unordered .....	182
6.18.8. ACLT: Floating-point absolute compare less than .....	182
6.18.9. ACLE: Floating-point absolute compare less than or equal to .....	182
6.18.10. ACGE: Floating-point absolute compare greater than or equal to .....	183
6.18.11. ACGT: Floating-point absolute compare greater than .....	183
6.19. Floating-point conversions .....	184

6.19.1. CVT: Convert floating-point value to integer .....	184
6.19.2. CVT: Convert integer value to floating-point .....	185
6.19.3. CVT: Convert floating-point value to wider type .....	186
6.19.4. CVT: Convert floating-point value to narrower type .....	187
6.20. Permutation and selection .....	188
6.20.1. LASTA: Extract element after last active .....	188
6.20.2. LASTB: Extract last active element .....	188
6.20.3. CLASTA: Extract element after last active with fallback .....	188
6.20.4. CLASTB: Extract last active element with fallback .....	189
6.20.5. COMPACT: Compact vector and fill with zero .....	190
6.20.6. SPLICE: Splice two vectors under predicate control .....	190
6.20.7. EXT: Extract vector from pair of vectors .....	191
6.20.8. SEL: Conditionally select elements from two inputs .....	191
6.20.9. DUP: Duplicate one element of a vector .....	192
6.20.10. DUPQ: Duplicate one quadword of a vector .....	192
6.20.11. TBL: Table lookup/permute using vector of indices .....	193
6.20.12. REV: Reverse the elements in a single input .....	193
6.20.13. TRN1: Interleave even elements from two inputs .....	194
6.20.14. TRN2: Interleave odd elements from two inputs .....	194
6.20.15. UNPKHI: Unpack and extend high half of an input .....	195
6.20.16. UNPKLO: Unpack and extend low half of an input .....	195
6.20.17. UZP1: Select even elements from two inputs .....	196
6.20.18. UZP2: Select odd elements from two inputs .....	196
6.20.19. ZIP1: Interleave elements from low halves of two inputs .....	197
6.20.20. ZIP2: Interleave elements from high halves of two inputs .....	197
6.21. Vector creation .....	198
6.21.1. CREATE2: Create a tuple of two vectors .....	198
6.21.2. CREATE3: Create a tuple of three vectors .....	198
6.21.3. CREATE4: Create a tuple of four vectors .....	199
6.21.4. UNDEF: Create an uninitialized vector .....	199
6.21.5. UNDEF2: Create an uninitialized tuple of two vectors .....	200
6.21.6. UNDEF3: Create an uninitialized tuple of three vectors .....	200
6.21.7. UNDEF4: Create an uninitialized tuple of four vectors .....	200
6.22. Vector insertion and extraction .....	201
6.22.1. SET2: Change one vector in a tuple of two vectors .....	201
6.22.2. SET3: Change one vector in a tuple of three vectors .....	201
6.22.3. SET4: Change one vector in a tuple of four vectors .....	202
6.22.4. GET2: Extract one vector from a tuple of two vectors .....	202
6.22.5. GET3: Extract one vector from a tuple of three vectors .....	203
6.22.6. GET4: Extract one vector from a tuple of four vectors .....	203
6.23. Predicate creation .....	204
6.23.1. PTRUE: Return an all-true predicate for a given pattern .....	204
6.23.2. PFALSE: Return an all-false predicate .....	204
6.23.3. DUP: Duplicate boolean value .....	204
6.23.4. DUPQ: Duplicate boolean values to fill a predicate .....	205
6.24. Predicate operations .....	205
6.24.1. MOV: Copy predicate .....	205
6.24.2. AND: Predicate AND .....	206
6.24.3. BIC: Predicate AND NOT .....	206
6.24.4. NAND: Predicate NAND .....	206
6.24.5. ORR: Predicate OR .....	206
6.24.6. ORN: Predicate OR NOT .....	206
6.24.7. NOR: Predicate NOR .....	206
6.24.8. EOR: Predicate exclusive OR .....	207

6.24.9. NOT: Predicate NOT .....	207
6.24.10. BRKA: Break after first true condition .....	207
6.24.11. BRKB: Break before first true condition .....	207
6.24.12. BRKN: Propagate break to next partition .....	208
6.24.13. BRKPA: Propagate and break after first true condition .....	208
6.24.14. BRKPB: Propagate and break before first true condition .....	208
6.24.15. PFIRST: Set first active predicate element to true .....	208
6.24.16. PNEXT: Set next active predicate element to true .....	208
6.25. Testing predicates .....	209
6.25.1. PTEST: Test active elements .....	209
6.26. FFR manipulation .....	209
6.26.1. RDIFFR: Read the first-fault register .....	209
6.26.2. SETFFR: Set the first-fault register .....	209
6.26.3. WRFFR: Write to the first-fault register .....	210
6.27. Counting elements .....	210
6.27.1. CNTP: Count active elements .....	210
6.27.2. CNTB: Count the number of 8-bit elements in a pattern .....	210
6.27.3. CNTH: Count the number of 16-bit elements in a pattern .....	210
6.27.4. CNTW: Count the number of 32-bit elements in a pattern .....	211
6.27.5. CNTD: Count the number of 64-bit elements in a pattern .....	211
6.27.6. LEN: Return the number of elements in a vector .....	211
6.28. Saturating scalar arithmetic .....	212
6.28.1. QINCB: Saturating increment by a multiple of <code>svcntb</code> .....	212
6.28.2. QINCH: Saturating increment by a multiple of <code>svcnth</code> .....	212
6.28.3. QINCW: Saturating increment by a multiple of <code>svcntw</code> .....	213
6.28.4. QINCD: Saturating increment by a multiple of <code>svcntd</code> .....	214
6.28.5. QINCP: Saturating increment by a multiple of <code>svcntp</code> .....	215
6.28.6. QDECB: Saturating decrement by a multiple of <code>svcntb</code> .....	216
6.28.7. QDECH: Saturating decrement by a multiple of <code>svcnth</code> .....	216
6.28.8. QDECW: Saturating decrement by a multiple of <code>svcntw</code> .....	217
6.28.9. QDECD: Saturating decrement by a multiple of <code>svcntd</code> .....	218
6.28.10. QDECP: Saturating decrement by a multiple of <code>svcntp</code> .....	219
6.29. Reinterpreting data .....	219
6.29.1. REINTERPRET: Reinterpret vector contents .....	219
7. List of optional SVE functions .....	223
7.1. Introduction .....	223
7.2. BFloat16 extensions .....	223
7.2.1. BFDOT: BFloat16 addition of dot product .....	223
7.2.2. BFMMLA: Accumulating multiplication of BFloat16 matrices .....	223
7.2.3. BFMLALB: BFloat16 addition of product long (bottom) .....	224
7.2.4. BFMLALT: BFloat16 addition of product long (top) .....	224
7.2.5. CVT: Convert single-precision floats to BFloat16 .....	225
7.2.6. CVTNT: Convert single-precision floats to BFloat16 (top) .....	225
7.3. INT8 matrix multiply extensions .....	226
7.3.1. MMLA: Accumulating widening multiplication of integer matrices .....	226
7.3.2. USMMLA: Accumulating widening multiplication of integer matrices (un- signed×signed) .....	226
7.3.3. USDOT: Integer addition of dot product (unsigned×signed) .....	226
7.3.4. SUDOT: Integer addition of dot product (signed×unsigned) .....	227
7.4. FP32 matrix multiply extensions .....	227
7.4.1. MMLA: Accumulating multiplication of 2×2 float matrices .....	227
7.5. FP64 matrix multiply extensions .....	228
7.5.1. MMLA: Accumulating multiplication of 2×2 double matrices .....	228
7.5.2. LD1RO: Unextended load and replicate to octoword .....	228



7.5.3. TRN1Q: Interleave even quadwords from two inputs .....	229
7.5.4. TRN2Q: Interleave odd quadwords from two inputs .....	229
7.5.5. UZP1Q: Select even quadwords from two inputs .....	230
7.5.6. UZP2Q: Select odd quadwords from two inputs .....	230
7.5.7. ZIP1Q: Interleave quadwords from low halves of two inputs .....	230
7.5.8. ZIP2Q: Interleave quadwords from high halves of two inputs .....	231
8. List of base SVE2 functions .....	232
8.1. Introduction .....	232
8.2. While greater comparisons .....	232
8.2.1. WHILEGT: While decrementing variable is greater than .....	232
8.2.2. WHILEGE: While decrementing variable is greater than or equal to .....	232
8.3. Uniform DSP operations .....	233
8.3.1. QADD: Saturating integer addition .....	233
8.3.2. SQADD: Saturating integer addition of signed value .....	234
8.3.3. UQADD: Saturating integer addition of unsigned value .....	235
8.3.4. HADD: Halving integer addition .....	236
8.3.5. RHADD: Rounding halving integer addition .....	238
8.3.6. QSUB: Saturating integer subtraction .....	239
8.3.7. QSUBR: Saturating integer subtraction, reversed .....	241
8.3.8. HSUB: Halving integer subtraction .....	242
8.3.9. HSUBR: Halving integer subtraction, reversed .....	244
8.3.10. ABA: Integer addition of absolute difference .....	245
8.3.11. QDMULH: Doubling integer multiplication, saturated high half .....	246
8.3.12. QRDMLH: Doubling integer multiplication, saturated rounded high half .....	246
8.3.13. QRDMLAH: Doubling integer multiplication, saturating addition of rounded high half .....	247
8.3.14. QRDMLSH: Doubling integer multiplication, saturating subtraction of round- ed high half .....	248
8.3.15. RSHL: Rounding shift left .....	248
8.3.16. RSHR: Rounding shift right .....	250
8.3.17. QSHL: Saturating shift left .....	251
8.3.18. QSHLU: Saturating shift left with unsigned result .....	252
8.3.19. QRSHL: Saturating rounding shift left .....	253
8.3.20. SRA: Shift right and accumulate .....	254
8.3.21. RSRA: Rounding shift right and accumulate .....	255
8.3.22. QNEG: Saturating integer negation .....	255
8.3.23. QABS: Saturating integer absolute value .....	256
8.3.24. RECPE: Integer reciprocal estimate .....	256
8.3.25. RSQRTE: Integer reciprocal square root estimate .....	257
8.3.26. SLI: Shift left and insert .....	257
8.3.27. SRI: Shift right and insert .....	257
8.4. Widening DSP operations .....	258
8.4.1. ADDLB: Integer addition long (bottom) .....	258
8.4.2. ADDLT: Integer addition long (top) .....	258
8.4.3. ADDWB: Integer addition wide (bottom) .....	259
8.4.4. ADDWT: Integer addition wide (top) .....	259
8.4.5. SUBLB: Integer subtraction long (bottom) .....	260
8.4.6. SUBLT: Integer subtraction long (top) .....	260
8.4.7. SUBWB: Integer subtraction wide (bottom) .....	261
8.4.8. SUBWT: Integer subtraction wide (top) .....	261
8.4.9. ABDLB: Integer absolute difference long (bottom) .....	262
8.4.10. ABDLT: Integer absolute difference long (top) .....	262
8.4.11. ABALB: Integer addition of absolute difference long (bottom) .....	263
8.4.12. ABALT: Integer addition of absolute difference long (top) .....	263

8.4.13. MULLB: Integer multiplication long (bottom) .....	264
8.4.14. MULLT: Integer multiplication long (top) .....	265
8.4.15. MLALB: Integer addition of product long (bottom) .....	265
8.4.16. MLALT: Integer addition of product long (top) .....	266
8.4.17. MLSLB: Integer subtraction of product long (bottom) .....	267
8.4.18. MLSLT: Integer subtraction of product long (top) .....	268
8.4.19. MOVLB: Move long (bottom) .....	269
8.4.20. MOVLT: Move long (top) .....	269
8.4.21. QDMULLB: Saturating integer doubled product long (bottom) .....	269
8.4.22. QDMULLT: Saturating integer doubled product long (top) .....	270
8.4.23. QDMLALB: Saturating integer addition of doubled product long (bottom) .....	271
8.4.24. QDMLALT: Saturating integer addition of doubled product long (top) .....	271
8.4.25. QDMLSLB: Saturating integer subtraction of doubled product long (bottom) ....	272
8.4.26. QDMLSLT: Saturating integer subtraction of doubled product long (top) .....	273
8.4.27. SHLLB: Shift left long by immediate (bottom) .....	273
8.4.28. SHLLT: Shift left long by immediate (top) .....	274
8.5. Narrowing DSP operations .....	274
8.5.1. ADDHNB: Integer addition, narrowing to high half (bottom) .....	274
8.5.2. ADDHNT: Integer addition, narrowing to high half (top) .....	275
8.5.3. RADDHNB: Integer addition, rounding narrowing to high half (bottom) .....	275
8.5.4. RADDHNT: Integer addition, rounding narrowing to high half (top) .....	276
8.5.5. SUBHNB: Integer subtraction, narrowing to high half (bottom) .....	276
8.5.6. SUBHNT: Integer subtraction, narrowing to high half (top) .....	277
8.5.7. RSUBHNB: Integer subtraction, rounding narrowing to high half (bottom) .....	278
8.5.8. RSUBHNT: Integer subtraction, rounding narrowing to high half (top) .....	278
8.5.9. SHRNB: Shift right, narrowing (bottom) .....	279
8.5.10. SHRNT: Shift right, narrowing (top) .....	279
8.5.11. RSHRNB: Rounding shift right, narrowing (bottom) .....	279
8.5.12. RSHRNT: Rounding shift right, narrowing (top) .....	280
8.5.13. QSHRNB: Shift right, saturating narrowing (bottom) .....	280
8.5.14. QSHRNT: Shift right, saturating narrowing (top) .....	281
8.5.15. QSHRUNB: Shift right, saturating narrowing to unsigned (bottom) .....	281
8.5.16. QSHRUNT: Shift right, saturating narrowing to unsigned (top) .....	281
8.5.17. QRSHRNB: Rounding shift right, saturating narrowing (bottom) .....	281
8.5.18. QRSHRNT: Rounding shift right, saturating narrowing (top) .....	282
8.5.19. QRSHRUNB: Rounding shift right, saturating narrowing to unsigned (bottom) .	282
8.5.20. QRSHRUNT: Rounding shift right, saturating narrowing to unsigned (top) .....	282
8.6. Unary narrowing operations .....	283
8.6.1. QXTNB: Saturating narrowing (bottom) .....	283
8.6.2. QXTNT: Saturating narrowing (top) .....	283
8.6.3. QXTUNB: Saturating narrowing to unsigned (bottom) .....	283
8.6.4. QXTUNT: Saturating narrowing to unsigned (top) .....	284
8.7. Non-widening pairwise arithmetic .....	284
8.7.1. ADDP: Integer pairwise addition .....	284
8.7.2. ADDP: Floating-point pairwise addition .....	284
8.7.3. MAXP: Integer pairwise maximum .....	285
8.7.4. MAXP: Floating-point pairwise maximum .....	285
8.7.5. MAXNMP: Floating-point pairwise maximum number .....	286
8.7.6. MINP: Integer pairwise minimum .....	286
8.7.7. MINP: Floating-point pairwise minimum .....	287
8.7.8. MINNMP: Floating-point pairwise minimum number .....	287
8.8. Widening pairwise arithmetic .....	288
8.8.1. ADALP: Integer pairwise addition and accumulate .....	288
8.9. Bitwise ternary logical instructions .....	289

8.9.1. BCAX: Bitwise clear and exclusive OR .....	289
8.9.2. BSL: Bitwise select .....	289
8.9.3. BSL1N: Bitwise select with first input inverted .....	290
8.9.4. BSL2N: Bitwise select with second input inverted .....	290
8.9.5. EOR3: Bitwise exclusive OR three vectors .....	291
8.9.6. NBSL: Bitwise inverted select .....	291
8.9.7. XAR: Bitwise exclusive OR and rotate right by immediate .....	292
8.10. Large integer arithmetic .....	292
8.10.1. ADCLB: Integer addition with carry (bottom) .....	292
8.10.2. ADCLT: Integer addition with carry (top) .....	293
8.10.3. SBCLB: Integer subtraction with carry (bottom) .....	293
8.10.4. SBCLT: Integer subtraction with carry (top) .....	293
8.11. Multiplication by indexed elements .....	294
8.11.1. MUL: Integer multiplication .....	294
8.11.2. MLA: Integer addition of product (addend first) .....	294
8.11.3. MLS: Integer subtraction of product (addend first) .....	295
8.12. Uniform complex integer arithmetic .....	295
8.12.1. CADD: Integer complex addition with rotation .....	295
8.12.2. QCADD: Integer complex saturating addition with rotation .....	296
8.12.3. CMLA: Integer complex addition of product with rotation .....	296
8.12.4. QRDCMLAH: Doubling complex integer multiplication with rotation, saturating addition of rounded high half .....	297
8.13. Widening complex integer arithmetic .....	298
8.13.1. ADDLBT: Integer addition long (bottom + top) .....	298
8.13.2. SUBLBT: Integer subtraction long (bottom - top) .....	298
8.13.3. SUBLTB: Integer subtraction long (top - bottom) .....	299
8.13.4. QDMLALBT: Saturating integer addition of doubled product long (bottom × top) .....	299
8.13.5. QDMLSLBT: Saturating integer subtraction of doubled product long (bottom × top) .....	299
8.14. Complex integer dot product .....	300
8.14.1. CDOT: Integer complex dot-product .....	300
8.15. Extra floating-point conversions .....	301
8.15.1. CVTLT: Convert floating-point value to wider type (top) .....	301
8.15.2. CVTNT: Convert floating-point value to narrower type (top) .....	301
8.15.3. CVTX: Convert floating-point value to narrower type, rounding to odd .....	302
8.15.4. CVTXNT: Convert floating-point value to narrower type, rounding to odd (top) .....	302
8.16. Floating-point widening multiply-accumulate .....	303
8.16.1. MLALB: Floating-point addition of product long (bottom) .....	303
8.16.2. MLALT: Floating-point addition of product long (top) .....	303
8.16.3. MSLB: Floating-point subtraction of product long (bottom) .....	304
8.16.4. MSLT: Floating-point subtraction of product long (top) .....	304
8.17. Floating-point integer binary logarithm .....	305
8.17.1. LOGB: Floating-point base 2 logarithm as integer .....	305
8.18. Vector histogram count .....	306
8.18.1. HISTCNT: Count matching elements in vector .....	306
8.18.2. HISTSEG: Count matching elements in vector segments .....	306
8.19. Character match .....	306
8.19.1. MATCH: Detect any matching elements .....	306
8.19.2. NMATCH: Detect no matching elements .....	306
8.20. Contiguous conflict detection .....	307
8.20.1. WHILERW: While free of read-after-write conflicts .....	307
8.20.2. WHILEWR: While free of write-after-read conflicts .....	307

8.21. Polynomial arithmetic .....	308
8.21.1. EORBT: Interleaving exclusive OR (bottom, top) .....	308
8.21.2. EORTB: Interleaving exclusive OR (top, bottom) .....	308
8.21.3. PMUL: Polynomial multiply .....	309
8.21.4. PMULLB: Polynomial multiply long (bottom) .....	309
8.21.5. PMULLT: Polynomial multiply long (top) .....	310
8.22. Extended table lookup/permute .....	310
8.22.1. TBL2: Table lookup/permute of two vectors using vector of indices .....	310
8.22.2. TBX: Table lookup/permute using vector of indices (merging) .....	311
8.23. Non-temporal gather/scatter .....	311
8.23.1. LDNT1: Unextended load, non-temporal .....	311
8.23.2. LDNT1SB: Load 8-bit data and sign-extend, non-temporal .....	313
8.23.3. LDNT1UB: Load 8-bit data and zero-extend, non-temporal .....	314
8.23.4. LDNT1SH: Load 16-bit data and sign-extend, non-temporal .....	315
8.23.5. LDNT1UH: Load 16-bit data and zero-extend, non-temporal .....	316
8.23.6. LDNT1SW: Load 32-bit data and sign-extend, non-temporal .....	318
8.23.7. LDNT1UW: Load 32-bit data and zero-extend, non-temporal .....	319
8.23.8. STNT1: Store one vector, with no truncation, non-temporal .....	320
8.23.9. STNT1B: Store one vector, truncating to 8 bits, non-temporal .....	322
8.23.10. STNT1H: Store one vector, truncating to 16 bits, non-temporal .....	323
8.23.11. STNT1W: Store one vector, truncating to 32 bits, non-temporal .....	324
9. List of optional SVE2 functions .....	326
9.1. Introduction .....	326
9.2. Bit permutation .....	326
9.2.1. BDEP: Bit deposit under mask control .....	326
9.2.2. BEXT: Bit extract under mask control .....	326
9.2.3. BGRP: Group bits to right or left as selected by bitmask .....	327
9.3. AES-128 functions .....	327
9.3.1. PMULLB: Polynomial multiply long (bottom) .....	327
9.3.2. PMULLT: Polynomial multiply long (top) .....	327
9.3.3. AESD: AES single round decryption .....	328
9.3.4. AESIMC: AES inverse mix columns .....	328
9.3.5. AESE: AES single round encryption .....	328
9.3.6. AESMC: AES mix columns .....	328
9.4. SHA-3 functions .....	329
9.4.1. RAX1: Bitwise rotate left by 1 and exclusive OR .....	329
9.5. SM4 functions .....	329
9.5.1. SM4E: SM4 encryption and decryption .....	329
9.5.2. SM4EKEY: SM4 key updates .....	329
10. Mapping of instructions to functions .....	330
10.1. List of instructions .....	330
10.2. CTERM EQ and CTERM NE .....	369
10.3. ADDPL, ADDVL, INC and DEC .....	369
A. Sizeless types in C .....	371
B. Sizeless types in C++ .....	373

# 1. About this document

## 1.1. Change control

### 1.1.1. Current status and expected changes

This document is a beta version of the Arm C language extensions for SVE.

Anticipated changes include:

- support for converting between Advanced SIMD and SVE vectors;
- stricter diagnostic requirements for invalid uses of sizeless types; and
- clarifications and corrections.

### 1.1.2. Change history

Issue	Date	By	Change
00bet1	06/04/17	RS	First public release
00bet2	14/06/19	RS	Turn tuple types into opaque types. Add <code>svundef...</code> , <code>svcreate...</code> , <code>svget...</code> , <code>svset...</code> and <code>svrdffr_z</code> . Add SVE2 functions.
00bet3	22/08/19	RS	Narrow the scalar arguments to the <code>_n</code> forms of the shift and integer comparison functions from 64 bits to the same width as the vector elements. Add <code>_n</code> forms to the corresponding <code>_wide</code> functions. Remove the unsigned forms of the <code>svcmpeq_wide</code> and <code>svcmpne_wide</code> functions. Move the 128-bit <code>svpmullb</code> and <code>svpmullt</code> functions to the AES section. Add support for the Armv8.6 Matrix Multiply and BFloat16 extensions.
00bet4	18/12/19	RS	Remove the zeroing forms of the <code>svaddp</code> , <code>svmaxp</code> , <code>sv-maxnmp</code> , <code>svminp</code> and <code>svminnmp</code> functions. Also remove the unsigned forms of the <code>svcdot</code> function.
00bet5	21/05/20	RS	Add support for non-member operator overloads in C++. Also add support for defining fixed-length versions of SVE vector and predicate types. Both features are optional.
00bet6	14/09/20	RS	Describe the C++ mangling for fixed-length versions of SVE vector and predicate types. Fix the names of enum types in <a href="#">Section 5, “Enum declarations”</a> . Add enum keywords to the prototypes.

## 1.2. References

This document refers to the following documents:

Reference	Document number	Author(s)	Title
Core ACLE	IHI 0053D	Arm	Arm C Language Extensions
C11	ISO/IEC 9899:2011	ISO	Standard C (based on draft N1570)

Reference	Document number	Author(s)	Title
C++14	ISO/IEC 14882:2014	ISO	Standard C++ (based on draft N3797)
AAPCS64	n/a	Arm	Procedure Call Standard for the Arm® 64-bit Architecture (AArch64)

## 1.3. Terms and abbreviations

This document uses the following terms and abbreviations:

Term	Meaning
AAPCS64	“Procedure Call Standard for the Arm® 64-bit Architecture (AArch64)”. This document describes the calling conventions for AArch64 code, as well as the layout of types. It includes rules for passing and returning SVE types in an efficient way.
ACLE	The Arm C language extensions. The extensions consist of several parts: a core ACLE defined in document IHI 0053D, plus supplemental documents for some architecture extensions. This document defines the SVE- and SVE2-specific part of the ACLE; within the document, the term ACLE generally refers to the SVE- and SVE2-specific part in particular.
Advanced SIMD	A 64-bit/128-bit SIMD instruction set defined as part of the Arm architecture.
Armv8-A	The base architecture for which SVE is specified.
FFR	The SVE first-fault register.
FFRT	The first-fault register token. This is a conceptual construct that forms part of the ACLE model of first-faulting and non-faulting loads ( <a href="#">Section 4.7, “First-faulting and non-faulting loads”</a> ).
SIMD	Single Instruction Multiple Data.
sizeless type	A C and C++ type that can be used to create objects, but that has no measurable size ( <a href="#">Section 3.2, “Sizeless types”</a> ).
SVE	The Armv8-A Scalable Vector Extension.
SVE2	A future architecture technology for the Arm A architecture profile, built on top of SVE.
VG	The number of 64-bit elements (“vector granules”) in an SVE vector.

## 1.4. Conventions

Most SVE ACLE functions have two names: a longer unique name and a shorter overloaded alias. The convention adopted in this document is to enclose characters in square brackets if they are only present in the longer name. For example:

```
svclz[_u16]_m
```

refers to a function whose full name is `svclz_u16_m` and whose overloaded alias is `svclz_m`.

Ranges in this document use square brackets if the bound is inclusive and round brackets if the bound is exclusive. For example, when describing the range of an integer parameter, the range `[1, 3]` specifies the set `{1, 2, 3}` and the range `[1, 5)` specifies the set `{1, 2, 3, 4}`.

## 2. Introduction

The aim of the Arm C language extensions (ACLE) is to make features of the Arm architecture directly available in C and C++ programs. The core ACLE is defined in a separate document (IHI 0053D), while the current document defines the part that is specific to the Arm Scalable Vector Extension (SVE) and to SVE2. The document does not assume prior knowledge of the core ACLE and the two are largely independent.

The SVE ACLE defines a set of C and C++ types and accessors for SVE vectors and predicates. It also defines C and C++ functions for almost all SVE and SVE2 instructions. [Section 10.1, “List of instructions”](#) lists each SVE and SVE2 instruction and links to the corresponding ACLE function (if one exists).

However, the function interface is more general than the underlying architecture, so not every function maps directly to an architectural instruction. The intention is to provide a regular interface and leave the compiler to pick the best mapping to SVE and SVE2 instructions.

In the rest of this document, the term ACLE usually refers to the SVE- and SVE2-specific part of the ACLE in particular.

### 2.1. Feature macros

The ACLE specifies a set of preprocessor macros that indicate which features (if any) are available in the current environment. Implementations of the ACLE define these macros to advertize the features they support. Code that wants to use a particular feature can then use preprocessor conditions to test whether the feature is available.

For example, the processor macro `__ARM_FEATURE_SVE` indicates that the ACLE is available in some form. Implementations that support the ACLE advertize that fact by defining the macro to a nonzero value (which must currently be 1). The most portable way to use the ACLE is therefore to guard its use with:

```
#ifdef __ARM_FEATURE_SVE
...
#endif

or:

#if __ARM_FEATURE_SVE
...
#endif
```

The full list of feature macros and their valid values is as follows:

`__ARM_FEATURE_SVE==1`

Indicates that the implementation is generating code for an SVE target and that all the functions in [Section 6, “List of base SVE functions”](#) are available.

`__ARM_FEATURE_SVE_BF16==1`

Indicates that all the functions in [Section 7.2, “BFloat16 extensions”](#) are available. Implementations that define this macro must also define `__ARM_FEATURE_BF16_SCALAR_ARITHMETIC`.

`__ARM_FEATURE_SVE_MATMUL_INT8==1`

Indicates that all the functions in [Section 7.3, “INT8 matrix multiply extensions”](#) are available.

`__ARM_FEATURE_SVE_MATMUL_FP32==1`

Indicates that all the functions in [Section 7.4, “FP32 matrix multiply extensions”](#) are available.

`__ARM_FEATURE_SVE_MATMUL_FP64==1`

Indicates that all the functions in [Section 7.5, “FP64 matrix multiply extensions”](#) are available.

`__ARM_FEATURE_SVE2==1`

Indicates that the implementation is generating code for an SVE2 target and that all the functions in [Section 8, “List of base SVE2 functions”](#) are available.

`__ARM_FEATURE_SVE2_BITPERM==1`

Indicates that all the functions in [Section 9.2, “Bit permutation”](#) are available. Implementations that define this macro must also define `__ARM_FEATURE_SVE2`.

`__ARM_FEATURE_SVE2_AES==1`

Indicates that all the functions in [Section 9.3, “AES-128 functions”](#) are available. Implementations that define this macro must also define `__ARM_FEATURE_SVE2` and `__ARM_FEATURE_AES`.

`__ARM_FEATURE_SVE2_SHA3==1`

Indicates that all the functions in [Section 9.4, “SHA-3 functions”](#) are available. Implementations that define this macro must also define `__ARM_FEATURE_SVE2` and `__ARM_FEATURE_SHA3`.

`__ARM_FEATURE_SVE2_SM4==1`

Indicates that all the functions in [Section 9.5, “SM4 functions”](#) are available. Implementations that define this macro must also define `__ARM_FEATURE_SVE2` and `__ARM_FEATURE_SM4`.

`__ARM_FEATURE_SVE_BITS==N`

When *N* is nonzero, indicates that the implementation is generating code for an *N*-bit SVE target and that the `arm_sve_vector_bits(N)` attribute is available. *N* may also be zero, but this carries the same meaning as not defining the macro. See [Section 3.7.2, “The `\_\_ARM\_FEATURE\_SVE\_BITS` macro”](#) for details.

`__ARM_FEATURE_SVE_VECTOR_OPERATORS==1`

Indicates that applying the `arm_sve_vector_bits` attribute to an SVE vector type creates a type that supports the GNU vector extensions. The state of this macro is only meaningful when `__ARM_FEATURE_SVE_BITS` is nonzero. See [Section 3.7.3.3, “Behavior specific to SVE vectors”](#) for details.

`__ARM_FEATURE_SVE_PREDICATE_OPERATORS==1`

Indicates that applying the `arm_sve_vector_bits` attribute to `svbool_t` creates a type that supports basic built-in vector operations. The state of this macro is only meaningful when `__ARM_FEATURE_SVE_BITS` is nonzero. See [Section 3.7.3.4, “Behavior specific to SVE predicates”](#) for details.

`__ARM_FEATURE_SVE_NONMEMBER_OPERATORS==1`

Indicates that C++ code can define non-member operator functions for SVE types. See [Section 3.6, “Non-member operator overloads in C++”](#) for details.



Except for `__ARM_FEATURE_SVE_BITS`, all macro values act as version numbers, with higher macro values being reserved for backward-compatible extensions to existing features. Other macro values are invalid.

An implementation that defines one of these macros must also define `__ARM_FEATURE_SVE`.

## 2.2. Header file

Translation units that use the ACLE must first include `arm_sve.h`, guarded by `__ARM_FEATURE_SVE`:

```
#ifndef __ARM_FEATURE_SVE
#include <arm_sve.h>
#endif /* __ARM_FEATURE_SVE */
```

It is safe to include the header file more than once.

All functions and types defined in the header file have the prefix `sv`, in order to reduce the chance of collisions with other extensions.

## 3. Types

### 3.1. Overview

SVE is a *vector-length agnostic* architecture. Instead of mandating a particular vector length, it allows implementations to choose any multiple of 128 bits up to the architectural maximum of 2048 bits. It also allows higher exception levels to constrain the vector lengths of lower exception levels. The effective vector length is therefore not a compile-time constant and can in principle change during execution.

Although the architectural increment is 128 bits, it is often more natural to think in terms of 64-bit quantities, since that is the widest element that the architecture supports. The term *VG* (“vector granules”) refers to the current number of 64-bit elements in an SVE vector, so that the number of usable bits in a vector is  $VG \times 64$ . Predicates have one bit for each vector byte, so the number of usable bits in a predicate is  $VG \times 8$ .

Code that makes direct use of SVE instructions therefore needs access to vector types that have  $VG \times 64$  bits and predicate types that have  $VG \times 8$  bits, but without the value of *VG* being fixed. Ideally it should be possible to pass and return these types by value, like it is for other (fixed-length) vector extensions. However, standard C and C++ do not provide variable-length types that are both self-contained (rather than dependent on separate storage) and suitable for passing and returning by value.<sup>1</sup> There is therefore no existing mechanism that maps directly to the concept of an SVE vector or predicate.

Any approach to defining the types would fall into one of three categories:

1. Limit the types in such a way that there is no concept of size.
2. Define the size of the types to be variable.
3. Define the size of the types to be constant, with the constant being large enough for all possible *VG* or with the types pointing to separate memory (as for classes like `std::string`).

The ACLE takes the first approach and classifies SVE vectors and predicates as belonging to a new category of type called *sizeless types*. The next section describes these types in more detail.

### 3.2. Sizeless types

For the reasons explained in the previous section, the ACLE defines a new category of type called *sizeless types*. They are less restrictive than the standard *incomplete types* but more restrictive than *complete types*. SVE vectors and predicates have sizeless type.

#### 3.2.1. Informal definition

Informally, sizeless types can be used in the following situations:

- as the type of an object with automatic storage duration;
- as a function parameter or return type;
- as a type name in a `_Generic` association;
- as the type in a `(type) {value}` compound literal;
- as the type in a C++ `type()` expression;

<sup>1</sup> For example, C's variable-length arrays are self-contained but are not valid return types. C++'s `std::string` can store variable-length data and is suitable for passing and returning by value, but it relies on separately-allocated storage.

- as the target of a pointer or reference type; and
- as a template type argument.

Sizeless types may not be used in the following situations:

- as the type of a variable with static or thread-local storage duration (regardless of whether the variable is being defined or just declared);
- as the type of an array element;
- as the operand to a new expression; and
- as the type of object being deleted by a `delete` expression.

In all other respects, sizeless types have the same restrictions as the standard-defined *incomplete types*. This specifically includes (but is not limited to) the following:

- The argument to `sizeof` and `_Alignof` cannot be a sizeless type, or an object of sizeless type.
- It is not possible to perform arithmetic on pointers to sizeless types. (This affects the `+`, `-`, `++` and `--` operators.)
- Members of unions, structures and classes cannot have sizeless type.
- `_Atomic` variables cannot have sizeless type.
- It is not possible to throw or catch objects of sizeless type.
- Lambda expressions cannot capture sizeless types by value, although they can capture them by reference. (This is a corollary of not allowing member variables to have sizeless type.)
- Standard library containers like `std::vector` cannot have a sizeless `value_type`.

### 3.2.2. Formal definition

[Appendix A, \*Sizeless types in C\*](#) gives a more formal definition of sizeless types for C, in the form of an edit to the standard. [Appendix B, \*Sizeless types in C++\*](#) gives the corresponding changes for C++. Implementations should correctly compile any code that follows these rules, but they do not need to report a diagnostic for invalid uses of sizeless types. Whether they report a diagnostic or not is a quality of implementation issue<sup>2</sup>.

### 3.2.3. Interaction with other C and C++ extensions

It is not feasible to reconcile the definition of sizeless types with each individual current and future extension to C and C++. The default position is therefore that extensions to C and C++ should treat sizeless types as incomplete types unless otherwise specified.

## 3.3. Scalar types

`arm_sve.h` includes `stdint.h` and so provides standard types such as `uint32_t`. When included from C code the header also includes `stdbool.h` and so provides the `bool` type.

In addition, the header file defines the following scalar data types:

<sup>2</sup> Note that future versions of the ACLE might have stricter diagnostic requirements.

`float16_t` equivalent to `__fp16`

`float32_t` equivalent to `float`

`float64_t` equivalent to `double`

If the feature macro `__ARM_FEATURE_BF16_SCALAR_ARITHMETIC` is defined, `arm_sve.h` also includes `arm_bf16.h`, which defines the scalar 16-bit brain floating-point type `__bf16`. Then, under the control of the same feature macro, `arm_sve.h` defines an alias of `__bf16` called `bfloat16_t`.

## 3.4. Vector types

The header file defines the following sizeless types for single vectors:

<code>svint8_t</code>	<code>svuint8_t</code>		
<code>svint16_t</code>	<code>svuint16_t</code>	<code>svfloat16_t</code>	<code>svbfloat16_t</code>
<code>svint32_t</code>	<code>svuint32_t</code>	<code>svfloat32_t</code>	
<code>svint64_t</code>	<code>svuint64_t</code>	<code>svfloat64_t</code>	

Each type `svBASE_t` is an opaque length-agnostic vector of `BASE_t` elements. The types in each row have the same number of elements and have twice as many elements as the types in the row below them.

`svbfloat16_t` is only available if the header file also provides a definition of `bfloat16_t` (see [Section 3.3, “Scalar types”](#)). The other types are available unconditionally.

The ACLE provides two sets of functions for converting between vector types. The `svreinterpret` functions simply reinterpret a vector of one type as a vector of another type, without changing any of the bits. The `svcv` functions instead perform numerical conversion from one type to another, such as converting integers to floating-point values. To avoid any ambiguity between the two operations, the ACLE does not allow C-style casting from one vector type to another.

The header file also defines tuples of two, three, and four vectors, as follows:

<code>svint8x2_t</code>	<code>svuint8x2_t</code>		
<code>svint16x2_t</code>	<code>svuint16x2_t</code>	<code>svfloat16x2_t</code>	<code>svbfloat16x2_t</code>
<code>svint32x2_t</code>	<code>svuint32x2_t</code>	<code>svfloat32x2_t</code>	
<code>svint64x2_t</code>	<code>svuint64x2_t</code>	<code>svfloat64x2_t</code>	
 <code>svint8x3_t</code>	 <code>svuint8x3_t</code>		
<code>svint16x3_t</code>	<code>svuint16x3_t</code>	<code>svfloat16x3_t</code>	<code>svbfloat16x3_t</code>
<code>svint32x3_t</code>	<code>svuint32x3_t</code>	<code>svfloat32x3_t</code>	
<code>svint64x3_t</code>	<code>svuint64x3_t</code>	<code>svfloat64x3_t</code>	
 <code>svint8x4_t</code>	 <code>svuint8x4_t</code>		
<code>svint16x4_t</code>	<code>svuint16x4_t</code>	<code>svfloat16x4_t</code>	<code>svbfloat16x4_t</code>
<code>svint32x4_t</code>	<code>svuint32x4_t</code>	<code>svfloat32x4_t</code>	
<code>svint64x4_t</code>	<code>svuint64x4_t</code>	<code>svfloat64x4_t</code>	

Each `svBASExN_t` is an opaque sizeless type that conceptually contains a sequence of  $N$  `svBASE_t`s, indexed starting at 0. It is available whenever the corresponding single-vector type is.

The ACLE provides the following functions for creating and accessing these tuple types:

```
svcreateN(svBASE_t x0, ...svBASE_t xN-1)
```

Return a `svBASExN_t` in which the vector at index  $i$  has the value given by  $x_i$ .

Section 6.21.1, “CREATE2: Create a tuple of two vectors”  
 Section 6.21.2, “CREATE3: Create a tuple of three vectors”  
 Section 6.21.3, “CREATE4: Create a tuple of four vectors”

```
svsetN(svBASExN_t tuple, uint64_t imm_index, svBASE_t x)
```

Return a copy of *tuple* in which the vector at index *imm\_index* has the value given by *x*. *imm\_index* must be an integer constant expression in the range [0, *N*-1].

Section 6.22.1, “SET2: Change one vector in a tuple of two vectors”  
 Section 6.22.2, “SET3: Change one vector in a tuple of three vectors”  
 Section 6.22.3, “SET4: Change one vector in a tuple of four vectors”

```
svgetN(svBASExN_t tuple, uint64_t imm_index)
```

Return the value of vector *imm\_index* in *tuple*. *imm\_index* must be an integer constant expression in the range [0, *N*-1].

Section 6.22.4, “GET2: Extract one vector from a tuple of two vectors”  
 Section 6.22.5, “GET3: Extract one vector from a tuple of three vectors”  
 Section 6.22.6, “GET4: Extract one vector from a tuple of four vectors”

It is also possible to create completely-undefined vectors and tuples of vectors using `svundef`:

Section 6.21.4, “UNDEF: Create an uninitialized vector”  
 Section 6.21.5, “UNDEF2: Create an uninitialized tuple of two vectors”  
 Section 6.21.6, “UNDEF3: Create an uninitialized tuple of three vectors”  
 Section 6.21.7, “UNDEF4: Create an uninitialized tuple of four vectors”

## 3.5. Predicate types

The header file defines a single sizeless predicate type `svbool_t`, which has enough bits to control an operation on a vector of bytes.

The main use of predicates is to select elements in a vector. When the elements in the vector have *N* bytes, only the low bit in each sequence of *N* predicate bits is significant. For example:

Vector type	Element selected by each <code>svbool_t</code> bit (from lsb)									
<code>svint8_t</code>	0	1	2	3	4	5	6	7	8	...
<code>svint16_t</code>	0	-	1	-	2	-	3	-	4	...
<code>svint32_t</code>	0	-	-	-	1	-	-	-	2	...
<code>svint64_t</code>	0	-	-	-	-	-	-	-	1	...

## 3.6. Non-member operator overloads in C++

C++ allows non-member operator overloads for class and enumeration types. For example:

```
enum E { E1 };
E& operator+=(E&, E) { ... }
```

overloads the `+=` operator for enumeration *E*. As an optional extension, an ACLE implementation may allow such overloads for the SVE vector, predicate and tuple types defined in [Section 3.4, “Vector types”](#) and [Section 3.5, “Predicate types”](#).

The effect of this feature is to modify the C++ standard as follows:

- Replace “enumeration” in sections `[over.match.oper]` and `[over.oper]` with wording that includes both enumerations and the SVE types listed above.
- Add an extra restriction to `[over.oper]`:

If a non-member operator function takes at least one parameter whose type is an SVE type or a reference to an SVE type, and if the function does not take a parameter whose type is a class, a reference to a class, an enumeration, or a reference to an enumeration, then the function must be declared static or within a namespace.

This restriction is intended to protect against accidental violations of the One Definition Rule, since such operators are not closely tied to a type defined by the source code.

- Extend the enumeration handling of `operator=` in `[over.built]` to include SVE types.

These edits are based on the 2020 edition of the standard but apply to earlier editions too.

The feature macro for this extension is:

```
__ARM_FEATURE_SVE_NONMEMBER_OPERATORS
```

An implementation that supports the extension for C++ may define the macro for other languages too; the macro does not indicate that the language being compiled is C++. See [Section 2.1, “Feature macros”](#) for further details about feature macros.

This extension makes it possible to define functions like:

```
#if __ARM_FEATURE_SVE_NONMEMBER_OPERATORS
static svint32_t operator+(svint32_t x, svint32_t y) { ... }
static svint32_t operator+(svint32_t x, int32_t y) { ... }
#endif
```

and use them in code like:

```
#if __ARM_FEATURE_SVE_NONMEMBER_OPERATORS
void f(svint32_t x, svint32_t y) { ... x + y ... }
void f(svint32_t x, int32_t y) { ... x + y ... }
#endif
```

## 3.7. Fixed-length types

### 3.7.1. Introduction

As described in [Section 3.1, “Overview”](#), SVE is a “length-agnostic” architecture that supports a range of different vector register sizes. Some SVE ACLE code might be written to work with all of these register sizes; such code is fully “length-agnostic”. Other SVE ACLE code might instead require the register size to belong to a restricted set. Possible restrictions include imposing a minimum register size, a maximum register size, a power-of-two register size, or a single exact register size. This is a choice for the programmer to make. For example, the program snippet:

```
svfloat32_t data = svld1(svptrue_pat_b32(SV_VL16), ptr);
```

loads no data when the vector registers are smaller than 512 bits, so in practice the code is likely to require a register size of 512 bits or more.

Similarly, for any given piece of source code (whether it uses the ACLE or not), a compiler might produce SVE object code that only correctly implements the source code for certain runtime register sizes. The

code is likely to abort or malfunction if the registers have a different size. For example, if the compiler assumes that the vector register size is exactly 256 bits, it might use:

```
add x1, x2, #32
```

instead of:

```
addvl x1, x2, #1
```

Let SourceVL be the set of vector register sizes supported by a piece of SVE ACLE source code and ObjectVL be the set of vector register sizes for which the object code correctly implements the source code. In both cases, the set might be the set of all possible register sizes (referred to as “unrestricted” below).

In general, object code created from SVE ACLE source code only works correctly when the runtime register size belongs to the intersection of SourceVL and ObjectVL. The behavior in other cases is undefined.

The compiler might impose a fixed ObjectVL set for all code or provide a level of user control. It might also allow ObjectVL to vary within the program, either within the same translation unit or between different translation units. Whether and how this happens is currently outside the scope of the ACLE.

Note that the combination of a singleton SourceVL and an unrestricted ObjectVL makes sense: ACLE source code written for a single register size can still be compiled correctly without knowing what that size is, just like an assembler can assemble SVE code without knowing what register sizes the assembly code supports. Similarly, the combination of a singleton ObjectVL and an unrestricted SourceVL makes sense: ACLE source code written for all register sizes can still be compiled for only a single register size, perhaps to try to improve the performance of surrounding code. Other combinations are also possible.

This section describes ACLE features related to singleton ObjectVLs. They are intended to help cases in which SourceVL is the same singleton.

All the features are optional: a conforming ACLE implementation does not need to provide them.

Note that the types defined in [Section 3.4, “Vector types”](#) and [Section 3.5, “Predicate types”](#) are always sizeless types and are subject to the same restrictions for all SourceVL and for all ObjectVL.

### 3.7.2. The `__ARM_FEATURE_SVE_BITS` macro

As an optional extension, an ACLE implementation may set the preprocessor macro:

```
__ARM_FEATURE_SVE_BITS
```

to a nonzero value  $N$  provided that:

- ObjectVL (see [Section 3.7.1, “Introduction”](#)) is a singleton vector register size of  $N$  bits; and
- it is possible to apply the `arm_sve_vector_bits( $N$ )` attribute to SVE vector types and SVE predicate types, as described in [Section 3.7.3, “The `arm\_sve\_vector\_bits` attribute”](#).

If the implementation allows ObjectVL to be changed within a translation unit, it must ensure that the value of the macro remains accurate after each change. It is therefore possible for `__ARM_FEATURE_SVE_BITS` to have different values at different points in a translation unit.

Defining the macro to zero has the same meaning as not defining it all.

### 3.7.3. The `arm_sve_vector_bits` attribute

#### 3.7.3.1. Syntax and requirements

An ACLE implementation may provide the following GNU-style type attribute:

```
__attribute__((arm_sve_vector_bits(...)))
```

The ACLE only defines the effect of the attribute if all of the following are true:

1. the attribute is attached to a single SVE vector type (such as `svint32_t`) or to the SVE predicate type `svbool_t`;
2. the arguments “...” consist of a single nonzero integer constant expression (referred to as  $N$  below); and
3.  $N == \text{__ARM\_FEATURE\_SVE\_BITS}$ .

In other cases the implementation must do one of the following:

- ignore the attribute; a warning would then be appropriate, but is not required
- reject the program with a diagnostic
- extend requirement (3) above to support other values of  $N$  besides `__ARM_FEATURE_SVE_BITS`
- process the attribute in accordance with a later revision of the ACLE

### 3.7.3.2. Behavior common to vectors and predicates

Let *VLST* be a valid type:

```
VLAT __attribute__((arm_sve_vector_bits(N)))
```

for some SVE vector type or SVE predicate type *VLAT*. *VLST* is then a fixed-length version of *VLAT* that is specialized for an  $N$ -bit SVE target. It is a normal sized (rather than sizeless) type, so unlike *VLAT*, it can be used as the type of a global variable, class member, `constexpr`, and so on. For example:

```
#if __ARM_FEATURE_SVE_BITS==512
typedef svint32_t vec __attribute__((arm_sve_vector_bits(512)));
typedef svbool_t pred __attribute__((arm_sve_vector_bits(512)));
#endif
```

creates a type called `vec` that acts a fixed-length version of `svint32_t` and a type called `pred` that acts as a fixed-length version of `svbool_t`. Both types are normal sized types; `vec` contains exactly 512 bits (64 bytes) and `pred` contains exactly 64 bits (8 bytes):

```
#if __ARM_FEATURE_SVE_BITS==512
typedef svint32_t vec __attribute__((arm_sve_vector_bits(512)));
typedef svbool_t pred __attribute__((arm_sve_vector_bits(512)));

svint32_t g1; // Invalid, svint32_t is sizeless
vec g2;      // OK
svbool_t g3; // Invalid, svbool_t is sizeless
pred g4;     // OK

struct wrap1 { svint32_t x; }; // Invalid, svint32_t is sizeless
struct wrap2 { vec x; };      // OK
struct wrap3 { svbool_t x; }; // Invalid, svbool_t is sizeless
struct wrap4 { pred x; };     // OK

size_t size1 = sizeof(svint32_t); // Invalid, svint32_t is sizeless
size_t size2 = sizeof(vec);        // OK, equals 64
size_t size3 = sizeof(svbool_t);   // Invalid, svbool_t is sizeless
size_t size4 = sizeof(pred);       // OK, equals 8
#endif
```



More specifically, the following rules apply to both vector and predicate *VLST* types:

- Whenever `__ARM_FEATURE_SVE_BITS==N`, *VLST* implicitly converts to *VLAT* and *VLAT* implicitly converts to *VLST*. In C++, these conversions have a rank just below derived-to-base conversion.

This behavior makes it possible to use *VLST* with ACLE functions that operate on *VLAT*. For example:

```
#if __ARM_FEATURE_SVE_BITS==512
typedef svbool_t pred __attribute__((arm_sve_vector_bits(512)));
pred f(pred x, pred y) { return svbrka_z(x, y); }
#endif
```

- A conditional expression of the form `C ? E1 : E2` is ill-formed if:
  - *E1* has type *VLAT* or cv-qualified *VLAT* and *E2* has type *VLST* or cv-qualified *VLST*; or
  - *E1* has type *VLST* or cv-qualified *VLST* and *E2* has type *VLAT* or cv-qualified *VLAT*.

However, the implementation does not need to diagnose these cases.

This rule avoids any ambiguity about whether the result of the conditional expression is sized or sizeless. For example:

```
#if __ARM_FEATURE_SVE_BITS==512
typedef svint32_t vec __attribute__((arm_sve_vector_bits(512)));
auto f(bool cond, vec x, svint32_t y) { return cond ? x : y; }
#endif
```

is ill-formed because it is not clear whether the type of the result should come from *x* or *y*.

- *VLST* maps to the same AAPCS64 ABI type as *VLAT*. For example:

```
svint32_t __attribute__((arm_sve_vector_bits(128)))
```

maps to the same AAPCS64 type as `svint32_t`, rather than to the same AAPCS64 type as the `arm_neon.h` type `int32x4_t`.

- An object type “requires size *N*” if:
  - it is an SVE vector type or SVE predicate type that has the attribute `arm_sve_vector_bits(N)`;
  - it is an array type whose element type requires size *N*; or
  - it is a class or struct and:
    - at least one non-static member has a type that requires size *N*; or
    - a base type requires size *N*

Note that this definition specifically excludes reference and pointer types.

- A function type “requires size *N*” if the return type requires size *N* or if at least one parameter type requires size *N*.
- A program is ill-formed if:
  - any function type requires two different sizes *N1* and *N2*.
  - `__ARM_FEATURE_SVE_BITS!=N` when defining a function whose type requires size *N*.

- `__ARM_FEATURE_SVE_BITS!=N` when calling a function whose type requires size *N*. (Note that this explicitly excludes parameters passed through “...”.)
- an object whose type requires some size *N* is passed to an unprototyped function.

However, the implementation does not need to diagnose these cases.

These rules only have an effect if an implementation allows the register size to vary within a translation unit. The aim is to forbid function definitions and function calls that have no mapping to the AAPCS64 calling conventions. For example, if an implementation supports some form of pragma that changes `__ARM_FEATURE_SVE_BITS` within a translation unit:

```
...pragma that changes __ARM_FEATURE_SVE_BITS to 256...

typedef svfloat32_t vec256 __attribute__((arm_sve_vector_bits(256)));

...pragma that changes __ARM_FEATURE_SVE_BITS to 512...

typedef svbool_t bool512 __attribute__((arm_sve_vector_bits(512)));
struct mix1 { bool512 a; vec256 b; }; // OK
struct mix2 { bool512 a; vec256 *b; }; // OK
union mix3 { bool512 a; vec256 b; }; // OK
typedef bool512 (*callback)(vec256); // Invalid, requires sizes 256, 512
bool512 f1(vec256); // Invalid, requires sizes 256, 512
vec256 f2(); // OK, just a declaration
vec256 f3() { ... } // Invalid, requires size 256
void f4(struct mix1 m) { ... } // Invalid, requires sizes 256, 512
void f5(struct mix2 m) { ... } // OK, just requires size 512
void f6(union mix3 m) { ... } // OK, no size requirements
bool512 f7() { ... } // OK, just requires size 512
void f8(bool512 *x, vec256 *y) { ... } // OK, no size requirements
```

See [Section 3.7.3.3, “Behavior specific to SVE vectors”](#) for additional rules that apply specifically to vectors and [Section 3.7.3.4, “Behavior specific to SVE predicates”](#) for additional rules that apply specifically to predicates.

### 3.7.3.3. Behavior specific to SVE vectors

If *VLAT* is an SVE vector type `svBASE_t` and *VLST* is a valid type:

```
VLAT __attribute__((arm_sve_vector_bits(N)))
```

then *VLST* is a sized type that has the following properties:

```
sizeof(VLST) == N/8
alignof(VLST) == 16
```

If in addition the implementation defines the preprocessor macro

```
__ARM_FEATURE_SVE_VECTOR_OPERATORS
```

and if *VLAT* is not `svbfloat16_t`, then:

- The GNU `__attribute__((vector_size))` extension is available.
- *VLST* supports the same forms of elementwise initialization as:

```
BASE_t __attribute__((vector_size(N/8)))
```

(referred to as *GNUT* below). For example:

```
#if __ARM_FEATURE_SVE_BITS==256 && __ARM_FEATURE_SVE_VECTOR_OPERATORS
svint64_t vec __attribute__((arm_sve_vector_bits(256))) = { 0, 1, 2, 3 };
#endif
```

- *VLST* supports the same built-in C and C++ operators as *GNUT*. Any result that has type *GNUT* for *GNUT* operators has type *VLST* for *VLST* operators. For example:

```
#if __ARM_FEATURE_SVE_BITS==512 && __ARM_FEATURE_SVE_VECTOR_OPERATORS
typedef svint32_t vec __attribute__((arm_sve_vector_bits(512)));
auto f(vec x, vec y) { return x + y; } // Returns a vec.
#endif
```

- Whenever `__ARM_FEATURE_SVE_BITS==N`, *GNUT* implicitly converts to *VLAT* and *VLAT* implicitly converts to *GNUT*. In C++, these conversions have a rank just below derived-to-base conversion.

This behavior makes it possible to use *GNUT* with ACLE functions that operate on *VLAT*. For example:

```
typedef int8_t vec __attribute__((vector_size(32)));
#if __ARM_FEATURE_SVE_BITS==256 && __ARM_FEATURE_SVE_VECTOR_OPERATORS
vec f(vec x) { return svasrd_x(svptrue_b8(), x, 1); }
#endif
```

- Irrespective of `__ARM_FEATURE_SVE_BITS`, *GNUT* implicitly converts to *VLST* and *VLST* implicitly converts to *GNUT*. In C++, these conversions again have a rank just below derived-to-base conversion.

This behavior makes it possible to use *VLST* with existing interfaces that operate on *GNUT*. For example:

```
typedef int8_t vec1 __attribute__((vector_size(32)));
void f(vec1);
#if __ARM_FEATURE_SVE_BITS==256 && __ARM_FEATURE_SVE_VECTOR_OPERATORS
typedef svint8_t vec2 __attribute__((arm_sve_vector_bits(256)));
void g(vec2 x) { f(x); } // OK
#endif
```

Also, if an implementation supports some form of pragma that changes `__ARM_FEATURE_SVE_BITS` within a translation unit:

```
#ifdef __ARM_FEATURE_SVE_VECTOR_OPERATORS
...pragma that changes __ARM_FEATURE_SVE_BITS to 256...

int8_t x __attribute__((vector_size(32)));
svint8_t y __attribute__((arm_sve_vector_bits(256)));

...pragma that changes __ARM_FEATURE_SVE_BITS to 512...

void f() { x = y; } // OK
#endif
```

- A conditional expression of the form `C ? E1 : E2` is ill-formed if:
  - *E1* has type *VLAT* or cv-qualified *VLAT* and *E2* has type *GNUT* or cv-qualified *GNUT*; or
  - *E1* has type *GNUT* or cv-qualified *GNUT* and *E2* has type *VLAT* or cv-qualified *VLAT*.

However, the implementation does not need to diagnose these cases.

This rule avoids any ambiguity about whether the result of the conditional expression is sized or sizeless. For example:

```
#if __ARM_FEATURE_SVE_BITS==512 && __ARM_FEATURE_SVE_VECTOR_OPERATORS
typedef int32_t vec __attribute__((vector_size(64)));
auto f(bool cond, vec x, svint32_t y) { return cond ? x : y; }
#endif
```

is ill-formed because it is not clear whether the type of the result should come from *x* or *y*. The choice would affect the ABI of the function.

- A binary expression of the form *E1 op E2* or a conditional expression of the form *C ? E1 : E2* is ill-formed if:
  - *E1* has type *VLST* or cv-qualified *VLST* and *E2* has type *GNUT* or cv-qualified *GNUT*; or
  - *E1* has type *GNUT* or cv-qualified *GNUT* and *E2* has type *VLST* or cv-qualified *VLST*.

However, the implementation does not need to diagnose these cases.

This rule avoids any ambiguity about whether the result of the expression is a GNU or SVE type. For example:

```
#if __ARM_FEATURE_SVE_BITS==512 && __ARM_FEATURE_SVE_VECTOR_OPERATORS
typedef int32_t vec1 __attribute__((vector_size(64)));
typedef svint32_t vec2 __attribute__((arm_sve_vector_bits(512)));
auto f(vec1 x, vec2 y) { return x + y; }
#endif
```

is ill-formed because it is not clear whether the type of the result should come from *x* or *y*. The choice would affect the ABI of the function.

### 3.7.3.4. Behavior specific to SVE predicates

Let *VLST* be a valid type:

```
svbool_t __attribute__((arm_sve_vector_bits(N)))
```

*VLST* is then a sized type that has the following properties:

```
sizeof(VLST) == N/64
alignof(VLST) == 2
```

If in addition the implementation defines the preprocessor macro

```
__ARM_FEATURE_SVE_PREDICATE_OPERATORS
```

then:

- The GNU `__attribute__((vector_size))` extension is available.
- *VLST* supports the same forms of elementwise initialization as the vector type:
 

```
uint8_t __attribute__((vector_size(N/8)))
```

 except that the elements have type `bool` instead of `uint8_t`.
- *VLST* supports the following operators, all of which take arguments of type *VLST* and return either a *VLST* or a reference to a *VLST*:
  - unary `~`
  - binary `&`, `|` and `^`

- assignments `&=`, `|=` and `^=`
- comparisons `<`, `<=`, `==`, `!=`, `>=` and `>`

All operators behave analogously to GNU vectors. For example:

```
#if __ARM_FEATURE_SVE_BITS==256 && __ARM_FEATURE_SVE_PREDICATE_OPERATORS
typedef svbool_t pred __attribute__((arm_sve_vector_bits(256)));
auto f(pred x, pred y) { return x & y; } // Returns a pred
#endif
```

- *VLST* supports the array subscript operator `[ ]` in a way that behaves analogously to GNU vectors, except that it is not possible to take the address of the result. For example:

```
#if __ARM_FEATURE_SVE_BITS==256 && __ARM_FEATURE_SVE_PREDICATE_OPERATORS
typedef svbool_t pred __attribute__((arm_sve_vector_bits(256)));
pred x;
bool f() { x[1] = true; return x[0]; }
#endif
```

- Within a byte, subscripts count from the least significant bit. For example:

```
#if __ARM_FEATURE_SVE_BITS==256 && __ARM_FEATURE_SVE_PREDICATE_OPERATORS
svbool_t p __attribute__((arm_sve_vector_bits(256))) = { 0, 1 };
#endif
```

sets the first byte of `p` to 2 for both big- and little-endian targets.

An implementation may provide other operators too, but it does not need to do so.

Note that, at the time of writing, the GNU `vector_size` extension is not defined for `bool` elements, nor for packed vectors of single bits. The definition above is intended to be a conservative subset of what such an extension might provide.

### 3.7.3.5. C++ mangling

Let *VLST* be a valid C++ type:

```
VLAT __attribute__((arm_sve_vector_bits(N)))
```

for some SVE vector type or SVE predicate type *VLAT*. *VLST* is mangled in the same way as a template:

```
template<typename, unsigned> struct __SVE_VLS;
```

with the arguments:

```
__SVE_VLS<VLAT, N>
```

For example:

```
#if __ARM_FEATURE_SVE_BITS==512
// Mangled as 9__SVE_VLSiu11__SVInt32_tLj512EE
typedef svint32_t vec __attribute__((arm_sve_vector_bits(512)));
// Mangled as 9__SVE_VLSiu10__SVBool_tLj512EE
typedef svbool_t pred __attribute__((arm_sve_vector_bits(512)));
#endif
```

## 4. Functions

### 4.1. Naming convention

The SVE ACLE functions have the form:

```
svbase[_disambiguator][_type0][_type1]...[_predication]
```

where the individual parts are as follows:

*base*

For most functions this is the lower-case name of an SVE instruction, but with some adjustments:

- The most common change is to drop F, S and U if they stand for “floating-point”, “signed” and “unsigned” respectively, in cases where this would duplicate information in the type suffixes below.
- Simple non-extending loads and non-truncating stores drop the size suffix (B, H, W or D), which would always duplicate information in the suffixes.
- Conversely, extending loads always specify an explicit extension type, since this information is not available in the suffixes. A sign-extending load has the same base as the architectural instruction (e.g. `ld1sb`) while a zero-extending load replaces the `s` with a `u` (e.g. `ld1ub` for a zero-extending LD1B). Thus `svld1ub_u32` zero-extends 8-bit data to a vector of `uint32_ts` while `svld1sb_u32` sign-extends 8-bit data to a vector of `uint32_ts`.

*disambiguator*

This field distinguishes between different forms of a function. There are several common uses:

- Load, store, prefetch, and ADR functions use this field to distinguish between different addressing modes. See [Section 4.3, “Addressing modes”](#) for a detailed description of these modes.
- Arithmetic operations use the disambiguator `_n` when the final operand is a scalar rather than a vector. See [Section 4.4, “Operations involving vectors and scalars”](#) for more information about these operations.
- Some predicate-based operations use the disambiguator `_pat` to show that they operate on an explicit constant predicate pattern like `MUL3` instead of either an all-true predicate or an `svbool_t`.

*type0*

*type1*

...

These fields list the types of vectors and predicates, starting with the return type and continuing with the argument types. They do not include the types of vector bases and displacements, which form part of the addressing mode disambiguator instead. They also do not include argument types that are uniquely determined by the previous argument types and return type.

For vectors the field is a type category followed by an element size in bits<sup>3</sup>:

<b>signed integers</b>	s8	s16	s32	s64
<b>unsigned integers</b>	u8	u16	u32	u64

<sup>3</sup> These suffixes are the same as for `arm_neon.h`.

floating-point numbers		f16	f32	f64
brain floating-point numbers		bf16		

For predicates the suffix is b followed by the size of the associated data elements in bits, or simply b if the operation does not assume a particular element size. For example, the function for

`PTRUE Pd.B`

is `svptrue_b8` while the function for

`PTRUE Pd.H`

is `svptrue_b16`. The function for:

`PFALSE Pd.B`

is `svpfalse_b` rather than `svpfalse_b8` since the result is suitable for all element sizes.

#### *predication*

This suffix describes the inactive elements in the result of a predicated operation. It can be one of the following:

- z** Zero predication: set all inactive elements of the result to zero.
- m** Merge predication: copy all inactive elements from the first vector argument.

Unary operations have a separate vector argument that comes before the general predicate; for example:

`CLZ Zd.H, Pg/M, Zs.H`

corresponds to:

`Zd = svclz_u16_m(Zd, Pg, Zs);`

Note that the argument does not need to be syntactically related to the result; calls such as:

`Zd = svclz_u16_m(svadd_u16_z(...), Pg, Zs);`

are also valid.

Binary and ternary operations reuse the first argument to the operation as the merge input; for example:

`ADD Zd.S, Pg/M, Zd.S, Zs2.S`

corresponds to:

`Zd = svadd_u32_m(Pg, Zd, Zs2);`

where the `Zd` argument supplies both the values of inactive elements and the first operand to the addition. Again, the merge input does not need to be syntactically related to the result.

- × “Don't-care” predication: set the inactive elements to **unknown** values. These values could be arbitrary register contents left around from a previous operation, so accidentally using the inactive elements could lead to data leakage.

This form of predication removes the need to choose between zeroing and merging in cases where the inactive elements are unimportant. The compiler can then pick whichever form of instruction seems to give the best code. This includes using unpredicated instructions, where available and suitable.

Predicated loads, predicated comparisons and predicated string match operations are always zeroing operations, so for brevity, the corresponding functions have no predication suffix.

## 4.2. Overloaded aliases

The function names that are described in [Section 4.1, “Naming convention”](#) often carry information that is obvious from the types of the arguments. For example, `svclz_u16_z` always takes a `svuint16_t` and no other form of `svclz_type_z` does. The ACLE therefore defines shorter, overloaded, aliases that are compatible with both C++ overloading and C `_Generic` associations<sup>4</sup>. At a minimum, this means that all overloaded forms of a function have the same number of arguments and that it is possible to select the correct overload by considering each argument from left to right.

The usual integer promotions mean that overloading based on scalar integer types can be non-obvious. For example, in the C++ code:

```
int f(unsigned char) { return 1; }
int f(int) { return 2; }
int g1(unsigned char c) { return f(c); }
int g2(unsigned char c) { return f(c + 1); }
```

`g1` returns 1 but `g2` returns 2. A cast would be required to make `g2` use the `unsigned char` version of `f` instead of the `int` version. For this reason, the ACLE does not use overloading of scalar arguments alone to determine the type of a vector result. Functions like `svdup` and `svindex` (whose only arguments are scalar) always require a suffix indicating the return type.

Overloaded aliases are always a full function name with parts of the suffix removed. The rest of this document refers to both the full function name and its overloaded alias by enclosing the elided suffix characters in square brackets. For example:

```
svclz[_u16]_m
```

says that the full name is `svclz_u16_m` and that its overloaded alias is `svclz_m`.

## 4.3. Addressing modes

Load, store, prefetch, and ADR functions have different forms for different addressing modes. The exact set of addressing modes depends on the particular operation but they always have a base component and may also have a displacement component.

The base may be either a single C pointer or a vector of address values. In the latter case, the address values may be 32 or 64 bits in size. Addressing modes with vector bases use the disambiguator `[_xNNbase]`, where `xNN` is the type of the address vector elements.

<sup>4</sup> Although the overloading is compatible with `_Generic`, the header file can use some other implementation-specific way of achieving the same effect.



The displacement, if present, may be a 64-bit scalar or a vector of 32-bit or 64-bit elements. There are five forms in total, with the following disambiguators:

<code>_offset</code>	The displacement is a 64-bit scalar byte count.
<code>_index</code>	The displacement is a 64-bit scalar element count. For example, if a function loads 16-bit elements from memory, an <code>_index</code> displacement of 1 is equivalent to an <code>_offset</code> displacement of 2.
<code>_vnum</code>	The displacement is a 64-bit scalar that counts a single vector's worth of elements. For example, if a function loads one or more full vectors from memory, a <code>_vnum</code> displacement of 1 is equivalent to an <code>_offset</code> displacement of $VG \times 8$ (the number of bytes in an SVE vector). If a function loads 16-bit elements and extends them to 32 bits, a <code>_vnum</code> displacement of 1 is equivalent to an <code>_offset</code> displacement of $VG \times 4$ (half the number of bytes in an SVE vector).  This form corresponds to the <code>MUL VL</code> addressing mode. However, the displacement argument can be any scalar value; it does not need to be a constant in a particular range.
<code>_[xNN]offset</code>	The displacement is a vector of type <code>xNN</code> and each element specifies a separate byte count. In other words, <code>_[s32]offset</code> specifies a vector of signed 32-bit byte counts and is equivalent to the <code>SXTW</code> addressing mode. <code>_[u32]offset</code> specifies a vector of unsigned 32-bit offsets and is equivalent to the <code>UXTW</code> addressing mode. <code>_[s64]offset</code> and <code>_[u64]offset</code> both specify vectors of 64-bit offsets; the sign is unimportant in this case.
<code>_[xNN]index</code>	Similar, but each element specifies an element count rather than a byte count, in the same way as for <code>_index</code> .

These displacements do not need to be constant and they do not need to be within a specific range.

For example, the simplest load addressing mode is:

```
svint16_t svld1[_s16](svbool_t pg, const int16_t *base)
```

which loads  $N$  elements from `base[0]` to `base[N-1]` inclusive.

It is possible to apply a displacement measured in whole vectors using:

```
svint16_t svld1_vnum[_s16](svbool_t pg, const int16_t *base, int64_t vnum)
```

which loads  $N$  elements from `base[N*vnum]` to `base[N*vnum+N-1]` inclusive.

The following function instead takes a vector of offsets, measured in bytes:

```
svuint32_t svld1_gather[_s32]offset[_u32](svbool_t pg, const uint32_t *base,
                                           svint32_t offsets)
```

In this case the address of element  $i$  is:

```
(int32_t *)((uintptr_t)base + offsets[i])
```

For:

```
svuint32_t svld1_gather[_s32]index[_u32](svbool_t pg, const uint32_t *base,
                                           svint32_t indices)
```

the address of element  $i$  is simply `&base[indices[i]]`.

The following function is an example of one that combines a vector base with a scalar index:

```
svint32_t svld1_gather[_u32base]_index_s32(svbool_t pg, svuint32_t bases,
                                           int64_t index)
```

The address of element *i* is:

```
((int32_t *) (uintptr_t) bases[i]) + index
```

## 4.4. Operations involving vectors and scalars

Some of the SVE instructions have immediate forms; for example:

```
ADD Zd.S, Zd.S, #1
```

adds 1 to every element of *Zd.S*. The ACLE extends this approach to all arithmetic operations and to all scalar inputs (not just immediates). The compiler can then use immediate forms where possible or duplicate the scalar into a vector otherwise.

For example:

```
svint32_t x;
...
x = svadd[_n_s32]_x(pg, x, 1);
```

adds 1 to every active element of *x* while:

```
svfloat64_t x;
double *ptr;
...
x = svadd[_n_f64]_x(pg, x, *ptr);
```

adds *\*ptr* to every active element of *x*. The first example is likely to use the immediate form of *ADD* while the latter is likely to use *LD1RD*.

In a vector operation, the disambiguator *\_n* indicates that the final operand is a scalar rather than a vector.

## 4.5. Immediate arguments

Some functions take enumeration values as arguments. These enumerations must always be integer constant expressions that specify a valid enumeration value.

A few functions take general integer immediates as arguments. In the function-specific documentation in [Section 6, “List of base SVE functions”](#) and following sections, these arguments always start with the prefix *imm*. Immediate arguments must be integer constant expressions within a certain range (which is the same as the range of the underlying SVE or SVE2 instruction).

## 4.6. Faults and exceptions

Loads and stores can trigger faults in the same way as normal pointer dereferences. Exactly what happens then depends on the host environment and is out of scope for this document. However, these faults should be symptoms of a program going wrong rather than something that the programmer deliberately planted. The compiler can therefore remove or reorder potentially faulting operations as long as:

1. doing so would not cause a previously non-faulting program to fault; and
2. faults do not move between C++ exception blocks, in cases where faults are reported as exceptions.

Also, many floating-point operations can raise IEEE exceptions. A similar set of rules apply there: the compiler can remove or reorder operations that might raise an IEEE exception if:

1. doing so would not cause a program to raise IEEE exceptions in cases where it would not previously;
2. IEEE exceptions do not move between C++ exception blocks, in cases where IEEE exceptions are reported as C++ exceptions; and
3. IEEE exceptions do not move across functions that manipulate the IEEE exception state.

For example, the compiler can remove a floating-point addition whose result is not used. It can also reorder independent floating-point operations, even if that would change the order that exceptions occur. It cannot however move floating-point addition across a direct or indirect call to `feclearexcept` unless it can prove that the addition would not raise an exception.

Many compilers have a mode that ignores IEEE exceptions. The floating-point restrictions above would not apply when such a mode is in effect.

## 4.7. First-faulting and non-faulting loads

SVE provides load instructions in which only the first active element can fault, and others in which no elements can fault. A special register called the first-fault register (FFR) records which elements were loaded successfully. The FFR describes all loads that have executed since the last write to the register.

If a first-faulting or non-faulting load does not load an active element due to a potential fault, it clears the FFR from that element onwards. Those elements of the returned vector then have an unknown value. Elements also have an unknown value if the associated FFR element was *already* clear before the instruction started.

For example, in:

```
SETFFR
LDFF1D Z0.D, P0/Z, [X0]
LDFF1D Z1.D, P1/Z, [X1]
LDFF1D Z2.D, P2/Z, [X2]
RDFFR P3.B
```

the load of `Z0.D` might suppress the load of `[X0, #16]` due to a potential fault. It would then clear bit 16 onwards of the FFR and leave elements 2 onwards of `Z0.D` in an unknown state. The same elements of `Z1.D` and `Z2.D` would then also be unknown, regardless of whether the loads based on `X1` or `X2` might fault. At the end of the sequence, `P3.B` indicates which elements of `Z0.D`, `Z1.D` and `Z2.D` are valid.

In this sequence, the leading inactive elements of `Z0.D` are guaranteed to be zero and the first active element is guaranteed to be valid (assuming that the first element did not trigger a fault). The same guarantees apply to `Z1.D` only if the first active element of `P1` is guaranteed to be earlier than the second active element of `P0`. In this sense, the contents of `Z0.D`, `Z1.D` and `Z2.D` do depend on the order of the instructions, since the guarantees would be different if the loads had a different order. However, the point of the loads is to try to access more than the first active element, so these relationships are not useful in practice.

The ACLE therefore divides first-faulting and non-faulting loads (but not normal loads) into “FFR groups”. Each group begins and ends with a function that explicitly reads from or writes to the FFR. More precisely, the ACLE introduces a global “first-fault register token” (FFRT) that identifies the current FFR group. This FFRT is a purely conceptual construct and contains three pieces of information:

*nwrite*      the number of explicit writes to the FFR

*lastwrite*   the last value written to the FFR

*nread*            the number of explicit reads from the FFR since the last write

There are only two possible ways of modifying this FFRT:

1. Increment *nwrite* by one, set *lastwrite* to a given value, and set *nread* to zero.
2. Increment *nread* by one.

Functions that explicitly write to the FFR do the first operation. Functions that explicitly read from the FFR do the second operation. First-faulting and non-faulting loads read from the FFRT and depend on its value, but they do not write to it.

One consequence of this arrangement is that a function that writes to the FFR is only dead if there are no further references to the FFRT in the program. However, the normal “as if” rules apply, so the compiler can generate an empty code sequence for the write if doing so would not affect the behavior of the program. Similarly, if a first-faulting or non-faulting load follows a function that explicitly reads from the FFR, without an intervening write, the load keeps the read function alive even if the result of that read is unused. Again, the normal “as if” rules mean that the compiler can generate an empty code sequence for the read if doing so would not affect the behavior of the program.

These FFRT dependencies are the only FFR-based ones that the compiler needs to consider when optimizing first-faulting and non-faulting loads; in all other respects the compiler can treat the loads like normal loads. This includes removing loads whose results are not used, suppressing loads of individual elements if their values do not matter, or reordering loads within a group (subject to the usual rules for normal loads). In practice the value of the FFR before a load does still affect which elements of the load result are defined, and in practice the loads do still write to the FFR, but the input program does not control these effects directly.

Assuming `float64_t` data, the C version of the code above would be:

```
svfloat64_t z0, z1, z2;
svbool_t p0, p1, p2, p3;
double *x0, *x1, *x2;
...
svsetffr();
z0 = svldffl[_f64](p0, x0);
z1 = svldffl[_f64](p1, x1);
z2 = svldffl[_f64](p2, x2);
p3 = svrdffr();
```

## 5. Enum declarations

The following enum enumerates all the possible patterns returned by a PTRUE:

```
enum svpattern
{
    SV_POW2 = 0,
    SV_VL1 = 1,
    SV_VL2 = 2,
    SV_VL3 = 3,
    SV_VL4 = 4,
    SV_VL5 = 5,
    SV_VL6 = 6,
    SV_VL7 = 7,
    SV_VL8 = 8,
    SV_VL16 = 9,
    SV_VL32 = 10,
    SV_VL64 = 11,
    SV_VL128 = 12,
    SV_VL256 = 13,
    SV_MUL4 = 29,
    SV_MUL3 = 30,
    SV_ALL = 31
};
```

The following enum lists the possible prefetch types:

```
enum svprfop
{
    SV_PLDL1KEEP = 0,
    SV_PLDL1STRM = 1,
    SV_PLDL2KEEP = 2,
    SV_PLDL2STRM = 3,
    SV_PLDL3KEEP = 4,
    SV_PLDL3STRM = 5,
    SV_PSTL1KEEP = 8,
    SV_PSTL1STRM = 9,
    SV_PSTL2KEEP = 10,
    SV_PSTL2STRM = 11,
    SV_PSTL3KEEP = 12,
    SV_PSTL3STRM = 13
};
```

## 6. List of base SVE functions

### 6.1. Introduction

This section contains a list of all base SVE functions, grouped into categories and then subdivided based on the first part of the name (up to the first underscore). All implementations of `arm_sve.h` provide these functions.

### 6.2. Loads

#### 6.2.1. LD1: Unextended load

These functions load values from memory and store the results in a vector. The vector elements have the same width as the loaded data; the functions do not perform any kind of extension.

##### 6.2.1.1. LD1 (scalar base)

Instances
<pre> svint8_t  svld1[_s8](svbool_t pg, const int8_t *base) svint16_t svld1[_s16](svbool_t pg, const int16_t *base) svint32_t svld1[_s32](svbool_t pg, const int32_t *base) svint64_t svld1[_s64](svbool_t pg, const int64_t *base) svuint8_t svld1[_u8](svbool_t pg, const uint8_t *base) svuint16_t svld1[_u16](svbool_t pg, const uint16_t *base) svuint32_t svld1[_u32](svbool_t pg, const uint32_t *base) svuint64_t svld1[_u64](svbool_t pg, const uint64_t *base) svfloat16_t svld1[_f16](svbool_t pg, const float16_t *base) svfloat32_t svld1[_f32](svbool_t pg, const float32_t *base) svfloat64_t svld1[_f64](svbool_t pg, const float64_t *base) svbfloat16_t svld1[_bf16](svbool_t pg, const bfloat16_t *base) </pre>

##### 6.2.1.2. LD1 (scalar base, VL displacement)

Instances
<pre> svint8_t  svld1_vnum[_s8](svbool_t pg, const int8_t *base, int64_t vnum) svint16_t svld1_vnum[_s16](svbool_t pg, const int16_t *base, int64_t vnum) svint32_t svld1_vnum[_s32](svbool_t pg, const int32_t *base, int64_t vnum) svint64_t svld1_vnum[_s64](svbool_t pg, const int64_t *base, int64_t vnum) svuint8_t svld1_vnum[_u8](svbool_t pg, const uint8_t *base, int64_t vnum) svuint16_t svld1_vnum[_u16](svbool_t pg, const uint16_t *base,                              int64_t vnum) svuint32_t svld1_vnum[_u32](svbool_t pg, const uint32_t *base,                              int64_t vnum) svuint64_t svld1_vnum[_u64](svbool_t pg, const uint64_t *base,                              int64_t vnum) svfloat16_t svld1_vnum[_f16](svbool_t pg, const float16_t *base,                               int64_t vnum) svfloat32_t svld1_vnum[_f32](svbool_t pg, const float32_t *base,                               int64_t vnum) svfloat64_t svld1_vnum[_f64](svbool_t pg, const float64_t *base,                               int64_t vnum) svbfloat16_t svld1_vnum[_bf16](svbool_t pg, const bfloat16_t *base,                                 int64_t vnum) </pre>

### 6.2.1.3. LD1 (vector base)

#### Instances

```
svint32_t svld1_gather[_u32base]_s32(svbool_t pg, svuint32_t bases)
svint64_t svld1_gather[_u64base]_s64(svbool_t pg, svuint64_t bases)
svuint32_t svld1_gather[_u32base]_u32(svbool_t pg, svuint32_t bases)
svuint64_t svld1_gather[_u64base]_u64(svbool_t pg, svuint64_t bases)
svfloat32_t svld1_gather[_u32base]_f32(svbool_t pg, svuint32_t bases)
svfloat64_t svld1_gather[_u64base]_f64(svbool_t pg, svuint64_t bases)
```

### 6.2.1.4. LD1 (scalar base, vector offset in bytes)

#### Instances

```
svint32_t svld1_gather[_s32]offset[_s32](svbool_t pg, const int32_t *base,
                                           svint32_t offsets)
svint64_t svld1_gather[_s64]offset[_s64](svbool_t pg, const int64_t *base,
                                           svint64_t offsets)
svuint32_t svld1_gather[_s32]offset[_u32](svbool_t pg, const uint32_t *base,
                                           svint32_t offsets)
svuint64_t svld1_gather[_s64]offset[_u64](svbool_t pg, const uint64_t *base,
                                           svint64_t offsets)
svfloat32_t svld1_gather[_s32]offset[_f32](svbool_t pg,
                                           const float32_t *base,
                                           svint32_t offsets)
svfloat64_t svld1_gather[_s64]offset[_f64](svbool_t pg,
                                           const float64_t *base,
                                           svint64_t offsets)
svint32_t svld1_gather[_u32]offset[_s32](svbool_t pg, const int32_t *base,
                                           svuint32_t offsets)
svint64_t svld1_gather[_u64]offset[_s64](svbool_t pg, const int64_t *base,
                                           svuint64_t offsets)
svuint32_t svld1_gather[_u32]offset[_u32](svbool_t pg, const uint32_t *base,
                                           svuint32_t offsets)
svuint64_t svld1_gather[_u64]offset[_u64](svbool_t pg, const uint64_t *base,
                                           svuint64_t offsets)
svfloat32_t svld1_gather[_u32]offset[_f32](svbool_t pg,
                                           const float32_t *base,
                                           svuint32_t offsets)
svfloat64_t svld1_gather[_u64]offset[_f64](svbool_t pg,
                                           const float64_t *base,
                                           svuint64_t offsets)
```

### 6.2.1.5. LD1 (vector base, scalar offset in bytes)

#### Instances

```
svint32_t svld1_gather[_u32base]_offset_s32(svbool_t pg, svuint32_t bases,
                                           int64_t offset)
svint64_t svld1_gather[_u64base]_offset_s64(svbool_t pg, svuint64_t bases,
                                           int64_t offset)
svuint32_t svld1_gather[_u32base]_offset_u32(svbool_t pg, svuint32_t bases,
                                           int64_t offset)
svuint64_t svld1_gather[_u64base]_offset_u64(svbool_t pg, svuint64_t bases,
                                           int64_t offset)
svfloat32_t svld1_gather[_u32base]_offset_f32(svbool_t pg, svuint32_t bases,
                                           int64_t offset)
svfloat64_t svld1_gather[_u64base]_offset_f64(svbool_t pg, svuint64_t bases,
                                           int64_t offset)
```

### 6.2.1.6. LD1 (scalar base, vector index)

Instances	
<code>svint32_t</code>	<code>svld1_gather_[s32]index[_s32](svbool_t pg, const int32_t *base, svint32_t indices)</code>
<code>svint64_t</code>	<code>svld1_gather_[s64]index[_s64](svbool_t pg, const int64_t *base, svint64_t indices)</code>
<code>svuint32_t</code>	<code>svld1_gather_[s32]index[_u32](svbool_t pg, const uint32_t *base, svint32_t indices)</code>
<code>svuint64_t</code>	<code>svld1_gather_[s64]index[_u64](svbool_t pg, const uint64_t *base, svint64_t indices)</code>
<code>svfloat32_t</code>	<code>svld1_gather_[s32]index[_f32](svbool_t pg, const float32_t *base, svint32_t indices)</code>
<code>svfloat64_t</code>	<code>svld1_gather_[s64]index[_f64](svbool_t pg, const float64_t *base, svint64_t indices)</code>
<code>svint32_t</code>	<code>svld1_gather_[u32]index[_s32](svbool_t pg, const int32_t *base, svuint32_t indices)</code>
<code>svint64_t</code>	<code>svld1_gather_[u64]index[_s64](svbool_t pg, const int64_t *base, svuint64_t indices)</code>
<code>svuint32_t</code>	<code>svld1_gather_[u32]index[_u32](svbool_t pg, const uint32_t *base, svuint32_t indices)</code>
<code>svuint64_t</code>	<code>svld1_gather_[u64]index[_u64](svbool_t pg, const uint64_t *base, svuint64_t indices)</code>
<code>svfloat32_t</code>	<code>svld1_gather_[u32]index[_f32](svbool_t pg, const float32_t *base, svuint32_t indices)</code>
<code>svfloat64_t</code>	<code>svld1_gather_[u64]index[_f64](svbool_t pg, const float64_t *base, svuint64_t indices)</code>

### 6.2.1.7. LD1 (vector base, scalar index)

Instances	
<code>svint32_t</code>	<code>svld1_gather[_u32base]_index_s32(svbool_t pg, svuint32_t bases, int64_t index)</code>
<code>svint64_t</code>	<code>svld1_gather[_u64base]_index_s64(svbool_t pg, svuint64_t bases, int64_t index)</code>
<code>svuint32_t</code>	<code>svld1_gather[_u32base]_index_u32(svbool_t pg, svuint32_t bases, int64_t index)</code>
<code>svuint64_t</code>	<code>svld1_gather[_u64base]_index_u64(svbool_t pg, svuint64_t bases, int64_t index)</code>
<code>svfloat32_t</code>	<code>svld1_gather[_u32base]_index_f32(svbool_t pg, svuint32_t bases, int64_t index)</code>
<code>svfloat64_t</code>	<code>svld1_gather[_u64base]_index_f64(svbool_t pg, svuint64_t bases, int64_t index)</code>

## 6.2.2. LD1SB: Load 8-bit data and sign-extend

These functions load 8-bit values from memory, sign-extend them, and store the results in a vector.

### 6.2.2.1. LD1SB (scalar base)

Instances	
<code>svint16_t</code>	<code>svld1sb_s16(svbool_t pg, const int8_t *base)</code>



**Instances**

```
svint32_t svldlsb_s32(svbool_t pg, const int8_t *base)
svint64_t svldlsb_s64(svbool_t pg, const int8_t *base)
svuint16_t svldlsb_u16(svbool_t pg, const int8_t *base)
svuint32_t svldlsb_u32(svbool_t pg, const int8_t *base)
svuint64_t svldlsb_u64(svbool_t pg, const int8_t *base)
```

**6.2.2.2. LD1SB (scalar base, VL displacement)****Instances**

```
svint16_t svldlsb_vnum_s16(svbool_t pg, const int8_t *base, int64_t vnum)
svint32_t svldlsb_vnum_s32(svbool_t pg, const int8_t *base, int64_t vnum)
svint64_t svldlsb_vnum_s64(svbool_t pg, const int8_t *base, int64_t vnum)
svuint16_t svldlsb_vnum_u16(svbool_t pg, const int8_t *base, int64_t vnum)
svuint32_t svldlsb_vnum_u32(svbool_t pg, const int8_t *base, int64_t vnum)
svuint64_t svldlsb_vnum_u64(svbool_t pg, const int8_t *base, int64_t vnum)
```

**6.2.2.3. LD1SB (vector base)****Instances**

```
svint32_t svldlsb_gather[_u32base]_s32(svbool_t pg, svuint32_t bases)
svint64_t svldlsb_gather[_u64base]_s64(svbool_t pg, svuint64_t bases)
svuint32_t svldlsb_gather[_u32base]_u32(svbool_t pg, svuint32_t bases)
svuint64_t svldlsb_gather[_u64base]_u64(svbool_t pg, svuint64_t bases)
```

**6.2.2.4. LD1SB (scalar base, vector offset in bytes)****Instances**

```
svint32_t svldlsb_gather[_s32]_offset_s32(svbool_t pg, const int8_t *base,
                                           svint32_t offsets)
svint64_t svldlsb_gather[_s64]_offset_s64(svbool_t pg, const int8_t *base,
                                           svint64_t offsets)
svuint32_t svldlsb_gather[_s32]_offset_u32(svbool_t pg, const int8_t *base,
                                           svint32_t offsets)
svuint64_t svldlsb_gather[_s64]_offset_u64(svbool_t pg, const int8_t *base,
                                           svint64_t offsets)
svint32_t svldlsb_gather[_u32]_offset_s32(svbool_t pg, const int8_t *base,
                                           svuint32_t offsets)
svint64_t svldlsb_gather[_u64]_offset_s64(svbool_t pg, const int8_t *base,
                                           svuint64_t offsets)
svuint32_t svldlsb_gather[_u32]_offset_u32(svbool_t pg, const int8_t *base,
                                           svuint32_t offsets)
svuint64_t svldlsb_gather[_u64]_offset_u64(svbool_t pg, const int8_t *base,
                                           svuint64_t offsets)
```

**6.2.2.5. LD1SB (vector base, scalar offset in bytes)****Instances**

```
svint32_t svldlsb_gather[_u32base]_offset_s32(svbool_t pg, svuint32_t bases,
                                              int64_t offset)
svint64_t svldlsb_gather[_u64base]_offset_s64(svbool_t pg, svuint64_t bases,
                                              int64_t offset)
svuint32_t svldlsb_gather[_u32base]_offset_u32(svbool_t pg,
                                              svuint32_t bases,
                                              int64_t offset)
```

**Instances**

```
svuint64_t svldlsb_gather[_u64base]_offset_u64(svbool_t pg,
                                                svuint64_t bases,
                                                int64_t offset)
```

**6.2.3. LD1UB: Load 8-bit data and zero-extend**

These functions load 8-bit values from memory, zero-extend them, and store the results in a vector.

**6.2.3.1. LD1UB (scalar base)****Instances**

```
svint16_t svldlub_s16(svbool_t pg, const uint8_t *base)
svint32_t svldlub_s32(svbool_t pg, const uint8_t *base)
svint64_t svldlub_s64(svbool_t pg, const uint8_t *base)
svuint16_t svldlub_u16(svbool_t pg, const uint8_t *base)
svuint32_t svldlub_u32(svbool_t pg, const uint8_t *base)
svuint64_t svldlub_u64(svbool_t pg, const uint8_t *base)
```

**6.2.3.2. LD1UB (scalar base, VL displacement)****Instances**

```
svint16_t svldlub_vnum_s16(svbool_t pg, const uint8_t *base, int64_t vnum)
svint32_t svldlub_vnum_s32(svbool_t pg, const uint8_t *base, int64_t vnum)
svint64_t svldlub_vnum_s64(svbool_t pg, const uint8_t *base, int64_t vnum)
svuint16_t svldlub_vnum_u16(svbool_t pg, const uint8_t *base, int64_t vnum)
svuint32_t svldlub_vnum_u32(svbool_t pg, const uint8_t *base, int64_t vnum)
svuint64_t svldlub_vnum_u64(svbool_t pg, const uint8_t *base, int64_t vnum)
```

**6.2.3.3. LD1UB (vector base)****Instances**

```
svint32_t svldlub_gather[_u32base]_s32(svbool_t pg, svuint32_t bases)
svint64_t svldlub_gather[_u64base]_s64(svbool_t pg, svuint64_t bases)
svuint32_t svldlub_gather[_u32base]_u32(svbool_t pg, svuint32_t bases)
svuint64_t svldlub_gather[_u64base]_u64(svbool_t pg, svuint64_t bases)
```

**6.2.3.4. LD1UB (scalar base, vector offset in bytes)****Instances**

```
svint32_t svldlub_gather_[s32]_offset_s32(svbool_t pg, const uint8_t *base,
                                           svint32_t offsets)
svint64_t svldlub_gather_[s64]_offset_s64(svbool_t pg, const uint8_t *base,
                                           svint64_t offsets)
svuint32_t svldlub_gather_[s32]_offset_u32(svbool_t pg, const uint8_t *base,
                                           svint32_t offsets)
svuint64_t svldlub_gather_[s64]_offset_u64(svbool_t pg, const uint8_t *base,
                                           svint64_t offsets)
svint32_t svldlub_gather_[u32]_offset_s32(svbool_t pg, const uint8_t *base,
                                           svuint32_t offsets)
svint64_t svldlub_gather_[u64]_offset_s64(svbool_t pg, const uint8_t *base,
                                           svuint64_t offsets)
svuint32_t svldlub_gather_[u32]_offset_u32(svbool_t pg, const uint8_t *base,
                                           svuint32_t offsets)
```

**Instances**

```
svuint64_t svldlub_gather[_u64]offset_u64(svbool_t pg, const uint8_t *base,
                                           svuint64_t offsets)
```

**6.2.3.5. LD1UB (vector base, scalar offset in bytes)****Instances**

```
svint32_t svldlub_gather[_u32base]offset_s32(svbool_t pg, svuint32_t bases,
                                              int64_t offset)
svint64_t svldlub_gather[_u64base]offset_s64(svbool_t pg, svuint64_t bases,
                                              int64_t offset)
svuint32_t svldlub_gather[_u32base]offset_u32(svbool_t pg,
                                              svuint32_t bases,
                                              int64_t offset)
svuint64_t svldlub_gather[_u64base]offset_u64(svbool_t pg,
                                              svuint64_t bases,
                                              int64_t offset)
```

**6.2.4. LD1SH: Load 16-bit data and sign-extend**

These functions load 16-bit values from memory, sign-extend them, and store the results in a vector.

**6.2.4.1. LD1SH (scalar base)****Instances**

```
svint32_t svld1sh_s32(svbool_t pg, const int16_t *base)
svint64_t svld1sh_s64(svbool_t pg, const int16_t *base)
svuint32_t svld1sh_u32(svbool_t pg, const int16_t *base)
svuint64_t svld1sh_u64(svbool_t pg, const int16_t *base)
```

**6.2.4.2. LD1SH (scalar base, VL displacement)****Instances**

```
svint32_t svld1sh_vnum_s32(svbool_t pg, const int16_t *base, int64_t vnum)
svint64_t svld1sh_vnum_s64(svbool_t pg, const int16_t *base, int64_t vnum)
svuint32_t svld1sh_vnum_u32(svbool_t pg, const int16_t *base, int64_t vnum)
svuint64_t svld1sh_vnum_u64(svbool_t pg, const int16_t *base, int64_t vnum)
```

**6.2.4.3. LD1SH (vector base)****Instances**

```
svint32_t svld1sh_gather[_u32base]_s32(svbool_t pg, svuint32_t bases)
svint64_t svld1sh_gather[_u64base]_s64(svbool_t pg, svuint64_t bases)
svuint32_t svld1sh_gather[_u32base]_u32(svbool_t pg, svuint32_t bases)
svuint64_t svld1sh_gather[_u64base]_u64(svbool_t pg, svuint64_t bases)
```

**6.2.4.4. LD1SH (scalar base, vector offset in bytes)****Instances**

```
svint32_t svld1sh_gather[_s32]offset_s32(svbool_t pg, const int16_t *base,
                                           svint32_t offsets)
svint64_t svld1sh_gather[_s64]offset_s64(svbool_t pg, const int16_t *base,
                                           svint64_t offsets)
```

Instances	
<code>svuint32_t svld1sh_gather_[s32]offset_u32</code>	<code>(svbool_t pg, const int16_t *base, svint32_t offsets)</code>
<code>svuint64_t svld1sh_gather_[s64]offset_u64</code>	<code>(svbool_t pg, const int16_t *base, svint64_t offsets)</code>
<code>svint32_t svld1sh_gather_[u32]offset_s32</code>	<code>(svbool_t pg, const int16_t *base, svuint32_t offsets)</code>
<code>svint64_t svld1sh_gather_[u64]offset_s64</code>	<code>(svbool_t pg, const int16_t *base, svuint64_t offsets)</code>
<code>svuint32_t svld1sh_gather_[u32]offset_u32</code>	<code>(svbool_t pg, const int16_t *base, svuint32_t offsets)</code>
<code>svuint64_t svld1sh_gather_[u64]offset_u64</code>	<code>(svbool_t pg, const int16_t *base, svuint64_t offsets)</code>

#### 6.2.4.5. LD1SH (vector base, scalar offset in bytes)

Instances	
<code>svint32_t svld1sh_gather[_u32base]_offset_s32</code>	<code>(svbool_t pg, svuint32_t bases, int64_t offset)</code>
<code>svint64_t svld1sh_gather[_u64base]_offset_s64</code>	<code>(svbool_t pg, svuint64_t bases, int64_t offset)</code>
<code>svuint32_t svld1sh_gather[_u32base]_offset_u32</code>	<code>(svbool_t pg, svuint32_t bases, int64_t offset)</code>
<code>svuint64_t svld1sh_gather[_u64base]_offset_u64</code>	<code>(svbool_t pg, svuint64_t bases, int64_t offset)</code>

#### 6.2.4.6. LD1SH (scalar base, vector index)

Instances	
<code>svint32_t svld1sh_gather_[s32]index_s32</code>	<code>(svbool_t pg, const int16_t *base, svint32_t indices)</code>
<code>svint64_t svld1sh_gather_[s64]index_s64</code>	<code>(svbool_t pg, const int16_t *base, svint64_t indices)</code>
<code>svuint32_t svld1sh_gather_[s32]index_u32</code>	<code>(svbool_t pg, const int16_t *base, svint32_t indices)</code>
<code>svuint64_t svld1sh_gather_[s64]index_u64</code>	<code>(svbool_t pg, const int16_t *base, svint64_t indices)</code>
<code>svint32_t svld1sh_gather_[u32]index_s32</code>	<code>(svbool_t pg, const int16_t *base, svuint32_t indices)</code>
<code>svint64_t svld1sh_gather_[u64]index_s64</code>	<code>(svbool_t pg, const int16_t *base, svuint64_t indices)</code>
<code>svuint32_t svld1sh_gather_[u32]index_u32</code>	<code>(svbool_t pg, const int16_t *base, svuint32_t indices)</code>
<code>svuint64_t svld1sh_gather_[u64]index_u64</code>	<code>(svbool_t pg, const int16_t *base, svuint64_t indices)</code>

#### 6.2.4.7. LD1SH (vector base, scalar index)

Instances	
<code>svint32_t svld1sh_gather[_u32base]_index_s32</code>	<code>(svbool_t pg, svuint32_t bases, int64_t index)</code>
<code>svint64_t svld1sh_gather[_u64base]_index_s64</code>	<code>(svbool_t pg, svuint64_t bases, int64_t index)</code>
<code>svuint32_t svld1sh_gather[_u32base]_index_u32</code>	<code>(svbool_t pg, svuint32_t bases, int64_t index)</code>

Instances	
<code>svuint64_t</code>	<code>svld1sh_gather[_u64base]_index_u64(svbool_t pg, svuint64_t bases, int64_t index)</code>

## 6.2.5. LD1UH: Load 16-bit data and zero-extend

These functions load 16-bit values from memory, zero-extend them, and store the results in a vector.

### 6.2.5.1. LD1UH (scalar base)

Instances	
<code>svint32_t</code>	<code>svld1uh_s32(svbool_t pg, const uint16_t *base)</code>
<code>svint64_t</code>	<code>svld1uh_s64(svbool_t pg, const uint16_t *base)</code>
<code>svuint32_t</code>	<code>svld1uh_u32(svbool_t pg, const uint16_t *base)</code>
<code>svuint64_t</code>	<code>svld1uh_u64(svbool_t pg, const uint16_t *base)</code>

### 6.2.5.2. LD1UH (scalar base, VL displacement)

Instances	
<code>svint32_t</code>	<code>svld1uh_vnum_s32(svbool_t pg, const uint16_t *base, int64_t vnum)</code>
<code>svint64_t</code>	<code>svld1uh_vnum_s64(svbool_t pg, const uint16_t *base, int64_t vnum)</code>
<code>svuint32_t</code>	<code>svld1uh_vnum_u32(svbool_t pg, const uint16_t *base, int64_t vnum)</code>
<code>svuint64_t</code>	<code>svld1uh_vnum_u64(svbool_t pg, const uint16_t *base, int64_t vnum)</code>

### 6.2.5.3. LD1UH (vector base)

Instances	
<code>svint32_t</code>	<code>svld1uh_gather[_u32base]_s32(svbool_t pg, svuint32_t bases)</code>
<code>svint64_t</code>	<code>svld1uh_gather[_u64base]_s64(svbool_t pg, svuint64_t bases)</code>
<code>svuint32_t</code>	<code>svld1uh_gather[_u32base]_u32(svbool_t pg, svuint32_t bases)</code>
<code>svuint64_t</code>	<code>svld1uh_gather[_u64base]_u64(svbool_t pg, svuint64_t bases)</code>

### 6.2.5.4. LD1UH (scalar base, vector offset in bytes)

Instances	
<code>svint32_t</code>	<code>svld1uh_gather[_s32]_offset_s32(svbool_t pg, const uint16_t *base, svint32_t offsets)</code>
<code>svint64_t</code>	<code>svld1uh_gather[_s64]_offset_s64(svbool_t pg, const uint16_t *base, svint64_t offsets)</code>
<code>svuint32_t</code>	<code>svld1uh_gather[_s32]_offset_u32(svbool_t pg, const uint16_t *base, svint32_t offsets)</code>
<code>svuint64_t</code>	<code>svld1uh_gather[_s64]_offset_u64(svbool_t pg, const uint16_t *base, svint64_t offsets)</code>
<code>svint32_t</code>	<code>svld1uh_gather[_u32]_offset_s32(svbool_t pg, const uint16_t *base, svuint32_t offsets)</code>
<code>svint64_t</code>	<code>svld1uh_gather[_u64]_offset_s64(svbool_t pg, const uint16_t *base, svuint64_t offsets)</code>
<code>svuint32_t</code>	<code>svld1uh_gather[_u32]_offset_u32(svbool_t pg, const uint16_t *base, svuint32_t offsets)</code>
<code>svuint64_t</code>	<code>svld1uh_gather[_u64]_offset_u64(svbool_t pg, const uint16_t *base, svuint64_t offsets)</code>

### 6.2.5.5. LD1UH (vector base, scalar offset in bytes)

Instances	
<code>svint32_t</code>	<code>svld1uh_gather[_u32base]_offset_s32(svbool_t pg, svuint32_t bases, int64_t offset)</code>
<code>svint64_t</code>	<code>svld1uh_gather[_u64base]_offset_s64(svbool_t pg, svuint64_t bases, int64_t offset)</code>
<code>svuint32_t</code>	<code>svld1uh_gather[_u32base]_offset_u32(svbool_t pg, svuint32_t bases, int64_t offset)</code>
<code>svuint64_t</code>	<code>svld1uh_gather[_u64base]_offset_u64(svbool_t pg, svuint64_t bases, int64_t offset)</code>

### 6.2.5.6. LD1UH (scalar base, vector index)

Instances	
<code>svint32_t</code>	<code>svld1uh_gather[_s32]_index_s32(svbool_t pg, const uint16_t *base, svint32_t indices)</code>
<code>svint64_t</code>	<code>svld1uh_gather[_s64]_index_s64(svbool_t pg, const uint16_t *base, svint64_t indices)</code>
<code>svuint32_t</code>	<code>svld1uh_gather[_s32]_index_u32(svbool_t pg, const uint16_t *base, svint32_t indices)</code>
<code>svuint64_t</code>	<code>svld1uh_gather[_s64]_index_u64(svbool_t pg, const uint16_t *base, svint64_t indices)</code>
<code>svint32_t</code>	<code>svld1uh_gather[_u32]_index_s32(svbool_t pg, const uint16_t *base, svuint32_t indices)</code>
<code>svint64_t</code>	<code>svld1uh_gather[_u64]_index_s64(svbool_t pg, const uint16_t *base, svuint64_t indices)</code>
<code>svuint32_t</code>	<code>svld1uh_gather[_u32]_index_u32(svbool_t pg, const uint16_t *base, svuint32_t indices)</code>
<code>svuint64_t</code>	<code>svld1uh_gather[_u64]_index_u64(svbool_t pg, const uint16_t *base, svuint64_t indices)</code>

### 6.2.5.7. LD1UH (vector base, scalar index)

Instances	
<code>svint32_t</code>	<code>svld1uh_gather[_u32base]_index_s32(svbool_t pg, svuint32_t bases, int64_t index)</code>
<code>svint64_t</code>	<code>svld1uh_gather[_u64base]_index_s64(svbool_t pg, svuint64_t bases, int64_t index)</code>
<code>svuint32_t</code>	<code>svld1uh_gather[_u32base]_index_u32(svbool_t pg, svuint32_t bases, int64_t index)</code>
<code>svuint64_t</code>	<code>svld1uh_gather[_u64base]_index_u64(svbool_t pg, svuint64_t bases, int64_t index)</code>

## 6.2.6. LD1SW: Load 32-bit data and sign-extend

These functions load 32-bit values from memory, sign-extend them, and store the results in a vector.

### 6.2.6.1. LD1SW (scalar base)

Instances	
<code>svint64_t</code>	<code>svld1sw_s64(svbool_t pg, const int32_t *base)</code>
<code>svuint64_t</code>	<code>svld1sw_u64(svbool_t pg, const int32_t *base)</code>

### 6.2.6.2. LD1SW (scalar base, VL displacement)

Instances
svint64_t <b>svld1sw_vnum_s64</b> (svbool_t <i>pg</i> , const int32_t * <i>base</i> , int64_t <i>vnum</i> )
svuint64_t <b>svld1sw_vnum_u64</b> (svbool_t <i>pg</i> , const int32_t * <i>base</i> , int64_t <i>vnum</i> )

### 6.2.6.3. LD1SW (vector base)

Instances
svint64_t <b>svld1sw_gather[_u64base]_s64</b> (svbool_t <i>pg</i> , svuint64_t <i>bases</i> )
svuint64_t <b>svld1sw_gather[_u64base]_u64</b> (svbool_t <i>pg</i> , svuint64_t <i>bases</i> )

### 6.2.6.4. LD1SW (scalar base, vector offset in bytes)

Instances
svint64_t <b>svld1sw_gather[_s64]_offset_s64</b> (svbool_t <i>pg</i> , const int32_t * <i>base</i> , svint64_t <i>offsets</i> )
svuint64_t <b>svld1sw_gather[_s64]_offset_u64</b> (svbool_t <i>pg</i> , const int32_t * <i>base</i> , svint64_t <i>offsets</i> )
svint64_t <b>svld1sw_gather[_u64]_offset_s64</b> (svbool_t <i>pg</i> , const int32_t * <i>base</i> , svuint64_t <i>offsets</i> )
svuint64_t <b>svld1sw_gather[_u64]_offset_u64</b> (svbool_t <i>pg</i> , const int32_t * <i>base</i> , svuint64_t <i>offsets</i> )

### 6.2.6.5. LD1SW (vector base, scalar offset in bytes)

Instances
svint64_t <b>svld1sw_gather[_u64base]_offset_s64</b> (svbool_t <i>pg</i> , svuint64_t <i>bases</i> , int64_t <i>offset</i> )
svuint64_t <b>svld1sw_gather[_u64base]_offset_u64</b> (svbool_t <i>pg</i> , svuint64_t <i>bases</i> , int64_t <i>offset</i> )

### 6.2.6.6. LD1SW (scalar base, vector index)

Instances
svint64_t <b>svld1sw_gather[_s64]_index_s64</b> (svbool_t <i>pg</i> , const int32_t * <i>base</i> , svint64_t <i>indices</i> )
svuint64_t <b>svld1sw_gather[_s64]_index_u64</b> (svbool_t <i>pg</i> , const int32_t * <i>base</i> , svint64_t <i>indices</i> )
svint64_t <b>svld1sw_gather[_u64]_index_s64</b> (svbool_t <i>pg</i> , const int32_t * <i>base</i> , svuint64_t <i>indices</i> )
svuint64_t <b>svld1sw_gather[_u64]_index_u64</b> (svbool_t <i>pg</i> , const int32_t * <i>base</i> , svuint64_t <i>indices</i> )

### 6.2.6.7. LD1SW (vector base, scalar index)

Instances
svint64_t <b>svld1sw_gather[_u64base]_index_s64</b> (svbool_t <i>pg</i> , svuint64_t <i>bases</i> , int64_t <i>index</i> )
svuint64_t <b>svld1sw_gather[_u64base]_index_u64</b> (svbool_t <i>pg</i> , svuint64_t <i>bases</i> , int64_t <i>index</i> )

## 6.2.7. LD1UW: Load 32-bit data and zero-extend

These functions load 32-bit values from memory, zero-extend them, and store the results in a vector.

### 6.2.7.1. LD1UW (scalar base)

Instances
<pre>svint64_t svldluw_s64(svbool_t pg, const uint32_t *base) svuint64_t svldluw_u64(svbool_t pg, const uint32_t *base)</pre>

### 6.2.7.2. LD1UW (scalar base, VL displacement)

Instances
<pre>svint64_t svldluw_vnum_s64(svbool_t pg, const uint32_t *base, int64_t vnum) svuint64_t svldluw_vnum_u64(svbool_t pg, const uint32_t *base,                              int64_t vnum)</pre>

### 6.2.7.3. LD1UW (vector base)

Instances
<pre>svint64_t svldluw_gather[_u64base]_s64(svbool_t pg, svuint64_t bases) svuint64_t svldluw_gather[_u64base]_u64(svbool_t pg, svuint64_t bases)</pre>

### 6.2.7.4. LD1UW (scalar base, vector offset in bytes)

Instances
<pre>svint64_t svldluw_gather_[s64]offset_s64(svbool_t pg, const uint32_t *base,  svint64_t offsets) svuint64_t svldluw_gather_[s64]offset_u64(svbool_t pg, const uint32_t *base,  svint64_t offsets) svint64_t svldluw_gather_[u64]offset_s64(svbool_t pg, const uint32_t *base,  svuint64_t offsets) svuint64_t svldluw_gather_[u64]offset_u64(svbool_t pg, const uint32_t *base,  svuint64_t offsets)</pre>

### 6.2.7.5. LD1UW (vector base, scalar offset in bytes)

Instances
<pre>svint64_t svldluw_gather[_u64base]_offset_s64(svbool_t pg, svuint64_t bases,   int64_t offset) svuint64_t svldluw_gather[_u64base]_offset_u64(svbool_t pg,   svuint64_t bases,   int64_t offset)</pre>

### 6.2.7.6. LD1UW (scalar base, vector index)

Instances
<pre>svint64_t svldluw_gather_[s64]index_s64(svbool_t pg, const uint32_t *base,  svint64_t indices) svuint64_t svldluw_gather_[s64]index_u64(svbool_t pg, const uint32_t *base,  svint64_t indices) svint64_t svldluw_gather_[u64]index_s64(svbool_t pg, const uint32_t *base,  svuint64_t indices)</pre>



**Instances**

```
svuint64_t svldluw_gather[_u64]index_u64(svbool_t pg, const uint32_t *base,
                                          svuint64_t indices)
```

**6.2.7.7. LD1UW (vector base, scalar index)****Instances**

```
svint64_t svldluw_gather[_u64base]_index_s64(svbool_t pg, svuint64_t bases,
                                              int64_t index)
svuint64_t svldluw_gather[_u64base]_index_u64(svbool_t pg, svuint64_t bases,
                                              int64_t index)
```

**6.2.8. LD1RQ: Unextended load and replicate to quadword**

These functions load 128 bits of data under predicate control, setting inactive elements in the 128 bits to zero. The functions then duplicate the data to every 128-bit quadword of the vector result.

**6.2.8.1. LD1RQ (scalar base)****Instances**

```
svint8_t svldlrq[_s8](svbool_t pg, const int8_t *base)
svint16_t svldlrq[_s16](svbool_t pg, const int16_t *base)
svint32_t svldlrq[_s32](svbool_t pg, const int32_t *base)
svint64_t svldlrq[_s64](svbool_t pg, const int64_t *base)
svuint8_t svldlrq[_u8](svbool_t pg, const uint8_t *base)
svuint16_t svldlrq[_u16](svbool_t pg, const uint16_t *base)
svuint32_t svldlrq[_u32](svbool_t pg, const uint32_t *base)
svuint64_t svldlrq[_u64](svbool_t pg, const uint64_t *base)
svfloat16_t svldlrq[_f16](svbool_t pg, const float16_t *base)
svfloat32_t svldlrq[_f32](svbool_t pg, const float32_t *base)
svfloat64_t svldlrq[_f64](svbool_t pg, const float64_t *base)
svbfloat16_t svldlrq[_bf16](svbool_t pg, const bfloat16_t *base)
```

**6.2.9. LDFF1: Unextended load, first-faulting**

These functions attempt to load values from memory and store the results in a vector, but they suppress faults for all elements except the first active one. The vector elements have the same width as the loaded data; the functions do not perform any kind of extension.

See [Section 4.7, “First-faulting and non-faulting loads”](#) for a description of the FFR handling. Code that uses these loads should call `svrdffr` to determine which elements of the result have defined values.

**6.2.9.1. LDFF1 (scalar base)****Instances**

```
svint8_t svldff1[_s8](svbool_t pg, const int8_t *base)
svint16_t svldff1[_s16](svbool_t pg, const int16_t *base)
svint32_t svldff1[_s32](svbool_t pg, const int32_t *base)
svint64_t svldff1[_s64](svbool_t pg, const int64_t *base)
svuint8_t svldff1[_u8](svbool_t pg, const uint8_t *base)
svuint16_t svldff1[_u16](svbool_t pg, const uint16_t *base)
svuint32_t svldff1[_u32](svbool_t pg, const uint32_t *base)
svuint64_t svldff1[_u64](svbool_t pg, const uint64_t *base)
svfloat16_t svldff1[_f16](svbool_t pg, const float16_t *base)
```

**Instances**

```
svfloat32_t svldff1[_f32](svbool_t pg, const float32_t *base)
svfloat64_t svldff1[_f64](svbool_t pg, const float64_t *base)
svbfloat16_t svldff1[_bf16](svbool_t pg, const bfloat16_t *base)
```

**6.2.9.2. LDFF1 (scalar base, VL displacement)****Instances**

```
svint8_t svldff1_vnum[_s8](svbool_t pg, const int8_t *base, int64_t vnum)
svint16_t svldff1_vnum[_s16](svbool_t pg, const int16_t *base,
                             int64_t vnum)
svint32_t svldff1_vnum[_s32](svbool_t pg, const int32_t *base,
                             int64_t vnum)
svint64_t svldff1_vnum[_s64](svbool_t pg, const int64_t *base,
                             int64_t vnum)
svuint8_t svldff1_vnum[_u8](svbool_t pg, const uint8_t *base, int64_t vnum)
svuint16_t svldff1_vnum[_u16](svbool_t pg, const uint16_t *base,
                              int64_t vnum)
svuint32_t svldff1_vnum[_u32](svbool_t pg, const uint32_t *base,
                              int64_t vnum)
svuint64_t svldff1_vnum[_u64](svbool_t pg, const uint64_t *base,
                              int64_t vnum)
svfloat16_t svldff1_vnum[_f16](svbool_t pg, const float16_t *base,
                              int64_t vnum)
svfloat32_t svldff1_vnum[_f32](svbool_t pg, const float32_t *base,
                              int64_t vnum)
svfloat64_t svldff1_vnum[_f64](svbool_t pg, const float64_t *base,
                              int64_t vnum)
svbfloat16_t svldff1_vnum[_bf16](svbool_t pg, const bfloat16_t *base,
                                 int64_t vnum)
```

**6.2.9.3. LDFF1 (vector base)****Instances**

```
svint32_t svldff1_gather[_u32base]_s32(svbool_t pg, svuint32_t bases)
svint64_t svldff1_gather[_u64base]_s64(svbool_t pg, svuint64_t bases)
svuint32_t svldff1_gather[_u32base]_u32(svbool_t pg, svuint32_t bases)
svuint64_t svldff1_gather[_u64base]_u64(svbool_t pg, svuint64_t bases)
svfloat32_t svldff1_gather[_u32base]_f32(svbool_t pg, svuint32_t bases)
svfloat64_t svldff1_gather[_u64base]_f64(svbool_t pg, svuint64_t bases)
```

**6.2.9.4. LDFF1 (scalar base, vector offset in bytes)****Instances**

```
svint32_t svldff1_gather[_s32]offset[_s32](svbool_t pg, const int32_t *base,
                                             svint32_t offsets)
svint64_t svldff1_gather[_s64]offset[_s64](svbool_t pg, const int64_t *base,
                                             svint64_t offsets)
svuint32_t svldff1_gather[_s32]offset[_u32](svbool_t pg,
                                             const uint32_t *base,
                                             svint32_t offsets)
svuint64_t svldff1_gather[_s64]offset[_u64](svbool_t pg,
                                             const uint64_t *base,
                                             svint64_t offsets)
svfloat32_t svldff1_gather[_s32]offset[_f32](svbool_t pg,
                                             const float32_t *base,
```

**Instances**

```

svfloat64_t svldff1_gather[_s64]offset[_f64](svbool_t pg,
                                              const float64_t *base,
                                              svint64_t offsets)
svint32_t svldff1_gather[_u32]offset[_s32](svbool_t pg, const int32_t *base,
                                              svuint32_t offsets)
svint64_t svldff1_gather[_u64]offset[_s64](svbool_t pg, const int64_t *base,
                                              svuint64_t offsets)
svuint32_t svldff1_gather[_u32]offset[_u32](svbool_t pg,
                                              const uint32_t *base,
                                              svuint32_t offsets)
svuint64_t svldff1_gather[_u64]offset[_u64](svbool_t pg,
                                              const uint64_t *base,
                                              svuint64_t offsets)
svfloat32_t svldff1_gather[_u32]offset[_f32](svbool_t pg,
                                              const float32_t *base,
                                              svuint32_t offsets)
svfloat64_t svldff1_gather[_u64]offset[_f64](svbool_t pg,
                                              const float64_t *base,
                                              svuint64_t offsets)

```

**6.2.9.5. LDFF1 (vector base, scalar offset in bytes)****Instances**

```

svint32_t svldff1_gather[_u32base]_offset_s32(svbool_t pg, svuint32_t bases,
                                              int64_t offset)
svint64_t svldff1_gather[_u64base]_offset_s64(svbool_t pg, svuint64_t bases,
                                              int64_t offset)
svuint32_t svldff1_gather[_u32base]_offset_u32(svbool_t pg,
                                              svuint32_t bases,
                                              int64_t offset)
svuint64_t svldff1_gather[_u64base]_offset_u64(svbool_t pg,
                                              svuint64_t bases,
                                              int64_t offset)
svfloat32_t svldff1_gather[_u32base]_offset_f32(svbool_t pg,
                                              svuint32_t bases,
                                              int64_t offset)
svfloat64_t svldff1_gather[_u64base]_offset_f64(svbool_t pg,
                                              svuint64_t bases,
                                              int64_t offset)

```

**6.2.9.6. LDFF1 (scalar base, vector index)****Instances**

```

svint32_t svldff1_gather[_s32]index[_s32](svbool_t pg, const int32_t *base,
                                              svint32_t indices)
svint64_t svldff1_gather[_s64]index[_s64](svbool_t pg, const int64_t *base,
                                              svint64_t indices)
svuint32_t svldff1_gather[_s32]index[_u32](svbool_t pg,
                                              const uint32_t *base,
                                              svint32_t indices)
svuint64_t svldff1_gather[_s64]index[_u64](svbool_t pg,
                                              const uint64_t *base,
                                              svint64_t indices)
svfloat32_t svldff1_gather[_s32]index[_f32](svbool_t pg,
                                              const float32_t *base,

```

Instances	
<code>svfloat64_t svldffl_gather[_s64]index[_f64]</code>	<code>(svbool_t pg, const float64_t *base, svint64_t indices)</code>
<code>svint32_t svldffl_gather[_u32]index[_s32]</code>	<code>(svbool_t pg, const int32_t *base, svuint32_t indices)</code>
<code>svint64_t svldffl_gather[_u64]index[_s64]</code>	<code>(svbool_t pg, const int64_t *base, svuint64_t indices)</code>
<code>svuint32_t svldffl_gather[_u32]index[_u32]</code>	<code>(svbool_t pg, const uint32_t *base, svuint32_t indices)</code>
<code>svuint64_t svldffl_gather[_u64]index[_u64]</code>	<code>(svbool_t pg, const uint64_t *base, svuint64_t indices)</code>
<code>svfloat32_t svldffl_gather[_u32]index[_f32]</code>	<code>(svbool_t pg, const float32_t *base, svuint32_t indices)</code>
<code>svfloat64_t svldffl_gather[_u64]index[_f64]</code>	<code>(svbool_t pg, const float64_t *base, svuint64_t indices)</code>

### 6.2.9.7. LDFF1 (vector base, scalar index)

Instances	
<code>svint32_t svldffl_gather[_u32base]_index_s32</code>	<code>(svbool_t pg, svuint32_t bases, int64_t index)</code>
<code>svint64_t svldffl_gather[_u64base]_index_s64</code>	<code>(svbool_t pg, svuint64_t bases, int64_t index)</code>
<code>svuint32_t svldffl_gather[_u32base]_index_u32</code>	<code>(svbool_t pg, svuint32_t bases, int64_t index)</code>
<code>svuint64_t svldffl_gather[_u64base]_index_u64</code>	<code>(svbool_t pg, svuint64_t bases, int64_t index)</code>
<code>svfloat32_t svldffl_gather[_u32base]_index_f32</code>	<code>(svbool_t pg, svuint32_t bases, int64_t index)</code>
<code>svfloat64_t svldffl_gather[_u64base]_index_f64</code>	<code>(svbool_t pg, svuint64_t bases, int64_t index)</code>

### 6.2.10. LDFF1SB: Load 8-bit data and sign-extend, first-faulting

These functions attempt to load 8-bit values from memory, sign-extend them, and store the results in a vector. Only the first active element can trigger a fault.

See [Section 4.7, “First-faulting and non-faulting loads”](#) for a description of the FFR handling. Code that uses these loads should call `svrdffr` to determine which elements of the result have defined values.

#### 6.2.10.1. LDFF1SB (scalar base)

Instances	
<code>svint16_t svldff1sb_s16</code>	<code>(svbool_t pg, const int8_t *base)</code>
<code>svint32_t svldff1sb_s32</code>	<code>(svbool_t pg, const int8_t *base)</code>
<code>svint64_t svldff1sb_s64</code>	<code>(svbool_t pg, const int8_t *base)</code>
<code>svuint16_t svldff1sb_u16</code>	<code>(svbool_t pg, const int8_t *base)</code>
<code>svuint32_t svldff1sb_u32</code>	<code>(svbool_t pg, const int8_t *base)</code>

**Instances**

```
svuint64_t svldfflsb_u64(svbool_t pg, const int8_t *base)
```

**6.2.10.2. LDFF1SB (scalar base, VL displacement)****Instances**

```
svint16_t svldfflsb_vnum_s16(svbool_t pg, const int8_t *base, int64_t vnum)
svint32_t svldfflsb_vnum_s32(svbool_t pg, const int8_t *base, int64_t vnum)
svint64_t svldfflsb_vnum_s64(svbool_t pg, const int8_t *base, int64_t vnum)
svuint16_t svldfflsb_vnum_u16(svbool_t pg, const int8_t *base,
                             int64_t vnum)
svuint32_t svldfflsb_vnum_u32(svbool_t pg, const int8_t *base,
                             int64_t vnum)
svuint64_t svldfflsb_vnum_u64(svbool_t pg, const int8_t *base,
                             int64_t vnum)
```

**6.2.10.3. LDFF1SB (vector base)****Instances**

```
svint32_t svldfflsb_gather[_u32base]_s32(svbool_t pg, svuint32_t bases)
svint64_t svldfflsb_gather[_u64base]_s64(svbool_t pg, svuint64_t bases)
svuint32_t svldfflsb_gather[_u32base]_u32(svbool_t pg, svuint32_t bases)
svuint64_t svldfflsb_gather[_u64base]_u64(svbool_t pg, svuint64_t bases)
```

**6.2.10.4. LDFF1SB (scalar base, vector offset in bytes)****Instances**

```
svint32_t svldfflsb_gather[_s32]_offset_s32(svbool_t pg, const int8_t *base,
                                             svint32_t offsets)
svint64_t svldfflsb_gather[_s64]_offset_s64(svbool_t pg, const int8_t *base,
                                             svint64_t offsets)
svuint32_t svldfflsb_gather[_s32]_offset_u32(svbool_t pg, const int8_t *base,
                                              svint32_t offsets)
svuint64_t svldfflsb_gather[_s64]_offset_u64(svbool_t pg, const int8_t *base,
                                              svint64_t offsets)
svint32_t svldfflsb_gather[_u32]_offset_s32(svbool_t pg, const int8_t *base,
                                             svuint32_t offsets)
svint64_t svldfflsb_gather[_u64]_offset_s64(svbool_t pg, const int8_t *base,
                                             svuint64_t offsets)
svuint32_t svldfflsb_gather[_u32]_offset_u32(svbool_t pg, const int8_t *base,
                                              svuint32_t offsets)
svuint64_t svldfflsb_gather[_u64]_offset_u64(svbool_t pg, const int8_t *base,
                                              svuint64_t offsets)
```

**6.2.10.5. LDFF1SB (vector base, scalar offset in bytes)****Instances**

```
svint32_t svldfflsb_gather[_u32base]_offset_s32(svbool_t pg,
                                                svuint32_t bases,
                                                int64_t offset)
svint64_t svldfflsb_gather[_u64base]_offset_s64(svbool_t pg,
                                                svuint64_t bases,
                                                int64_t offset)
svuint32_t svldfflsb_gather[_u32base]_offset_u32(svbool_t pg,
                                                svuint32_t bases,
```

Instances	
<code>svuint64_t</code>	<code>svldfflsb_gather[_u64base]_offset_u64(svbool_t pg, svuint64_t bases, int64_t offset)</code>

## 6.2.11. LDFF1UB: Load 8-bit data and zero-extend, first-faulting

These functions attempt to load 8-bit values from memory, zero-extend them, and store the results in a vector. Only the first active element can trigger a fault.

See [Section 4.7, “First-faulting and non-faulting loads”](#) for a description of the FFR handling. Code that uses these loads should call `svrddfr` to determine which elements of the result have defined values.

### 6.2.11.1. LDFF1UB (scalar base)

Instances	
<code>svint16_t</code>	<code>svldfflub_s16(svbool_t pg, const uint8_t *base)</code>
<code>svint32_t</code>	<code>svldfflub_s32(svbool_t pg, const uint8_t *base)</code>
<code>svint64_t</code>	<code>svldfflub_s64(svbool_t pg, const uint8_t *base)</code>
<code>svuint16_t</code>	<code>svldfflub_u16(svbool_t pg, const uint8_t *base)</code>
<code>svuint32_t</code>	<code>svldfflub_u32(svbool_t pg, const uint8_t *base)</code>
<code>svuint64_t</code>	<code>svldfflub_u64(svbool_t pg, const uint8_t *base)</code>

### 6.2.11.2. LDFF1UB (scalar base, VL displacement)

Instances	
<code>svint16_t</code>	<code>svldfflub_vnum_s16(svbool_t pg, const uint8_t *base, int64_t vnum)</code>
<code>svint32_t</code>	<code>svldfflub_vnum_s32(svbool_t pg, const uint8_t *base, int64_t vnum)</code>
<code>svint64_t</code>	<code>svldfflub_vnum_s64(svbool_t pg, const uint8_t *base, int64_t vnum)</code>
<code>svuint16_t</code>	<code>svldfflub_vnum_u16(svbool_t pg, const uint8_t *base, int64_t vnum)</code>
<code>svuint32_t</code>	<code>svldfflub_vnum_u32(svbool_t pg, const uint8_t *base, int64_t vnum)</code>
<code>svuint64_t</code>	<code>svldfflub_vnum_u64(svbool_t pg, const uint8_t *base, int64_t vnum)</code>

### 6.2.11.3. LDFF1UB (vector base)

Instances	
<code>svint32_t</code>	<code>svldfflub_gather[_u32base]_s32(svbool_t pg, svuint32_t bases)</code>
<code>svint64_t</code>	<code>svldfflub_gather[_u64base]_s64(svbool_t pg, svuint64_t bases)</code>
<code>svuint32_t</code>	<code>svldfflub_gather[_u32base]_u32(svbool_t pg, svuint32_t bases)</code>
<code>svuint64_t</code>	<code>svldfflub_gather[_u64base]_u64(svbool_t pg, svuint64_t bases)</code>

### 6.2.11.4. LDFF1UB (scalar base, vector offset in bytes)

Instances	
<code>svint32_t</code>	<code>svldfflub_gather[_s32]_offset_s32(svbool_t pg, const uint8_t *base, svint32_t offsets)</code>
<code>svint64_t</code>	<code>svldfflub_gather[_s64]_offset_s64(svbool_t pg, const uint8_t *base,</code>

Instances	
<code>svuint32_t</code>	<code>svldfflub_gather_[s32]offset_u32(svbool_t pg, svint64_t offsets)</code>
	<code>const uint8_t *base,</code>
	<code>svint32_t offsets)</code>
<code>svuint64_t</code>	<code>svldfflub_gather_[s64]offset_u64(svbool_t pg, svint64_t offsets)</code>
	<code>const uint8_t *base,</code>
	<code>svint64_t offsets)</code>
<code>svint32_t</code>	<code>svldfflub_gather_[u32]offset_s32(svbool_t pg, const uint8_t *base, svuint32_t offsets)</code>
<code>svint64_t</code>	<code>svldfflub_gather_[u64]offset_s64(svbool_t pg, const uint8_t *base, svuint64_t offsets)</code>
<code>svuint32_t</code>	<code>svldfflub_gather_[u32]offset_u32(svbool_t pg, const uint8_t *base, svuint32_t offsets)</code>
<code>svuint64_t</code>	<code>svldfflub_gather_[u64]offset_u64(svbool_t pg, const uint8_t *base, svuint64_t offsets)</code>

### 6.2.11.5. LDFF1UB (vector base, scalar offset in bytes)

Instances	
<code>svint32_t</code>	<code>svldfflub_gather[_u32base]_offset_s32(svbool_t pg, svuint32_t bases, int64_t offset)</code>
<code>svint64_t</code>	<code>svldfflub_gather[_u64base]_offset_s64(svbool_t pg, svuint64_t bases, int64_t offset)</code>
<code>svuint32_t</code>	<code>svldfflub_gather[_u32base]_offset_u32(svbool_t pg, svuint32_t bases, int64_t offset)</code>
<code>svuint64_t</code>	<code>svldfflub_gather[_u64base]_offset_u64(svbool_t pg, svuint64_t bases, int64_t offset)</code>

### 6.2.12. LDFF1SH: Load 16-bit data and sign-extend, first-faulting

These functions attempt to load 16-bit values from memory, sign-extend them, and store the results in a vector. Only the first active element can trigger a fault.

See [Section 4.7, “First-faulting and non-faulting loads”](#) for a description of the FFR handling. Code that uses these loads should call `svrdffr` to determine which elements of the result have defined values.

#### 6.2.12.1. LDFF1SH (scalar base)

Instances	
<code>svint32_t</code>	<code>svldfflsh_s32(svbool_t pg, const int16_t *base)</code>
<code>svint64_t</code>	<code>svldfflsh_s64(svbool_t pg, const int16_t *base)</code>
<code>svuint32_t</code>	<code>svldfflsh_u32(svbool_t pg, const int16_t *base)</code>
<code>svuint64_t</code>	<code>svldfflsh_u64(svbool_t pg, const int16_t *base)</code>

#### 6.2.12.2. LDFF1SH (scalar base, VL displacement)

Instances	
<code>svint32_t</code>	<code>svldfflsh_vnum_s32(svbool_t pg, const int16_t *base,</code>

Instances	
	int64_t vnum)
svint64_t	svldff1sh_vnum_s64(svbool_t pg, const int16_t *base, int64_t vnum)
svuint32_t	svldff1sh_vnum_u32(svbool_t pg, const int16_t *base, int64_t vnum)
svuint64_t	svldff1sh_vnum_u64(svbool_t pg, const int16_t *base, int64_t vnum)

### 6.2.12.3. LDFF1SH (vector base)

Instances	
svint32_t	svldff1sh_gather[_u32base]_s32(svbool_t pg, svuint32_t bases)
svint64_t	svldff1sh_gather[_u64base]_s64(svbool_t pg, svuint64_t bases)
svuint32_t	svldff1sh_gather[_u32base]_u32(svbool_t pg, svuint32_t bases)
svuint64_t	svldff1sh_gather[_u64base]_u64(svbool_t pg, svuint64_t bases)

### 6.2.12.4. LDFF1SH (scalar base, vector offset in bytes)

Instances	
svint32_t	svldff1sh_gather_[s32]offset_s32(svbool_t pg, const int16_t *base, svint32_t offsets)
svint64_t	svldff1sh_gather_[s64]offset_s64(svbool_t pg, const int16_t *base, svint64_t offsets)
svuint32_t	svldff1sh_gather_[s32]offset_u32(svbool_t pg, const int16_t *base, svint32_t offsets)
svuint64_t	svldff1sh_gather_[s64]offset_u64(svbool_t pg, const int16_t *base, svint64_t offsets)
svint32_t	svldff1sh_gather_[u32]offset_s32(svbool_t pg, const int16_t *base, svuint32_t offsets)
svint64_t	svldff1sh_gather_[u64]offset_s64(svbool_t pg, const int16_t *base, svuint64_t offsets)
svuint32_t	svldff1sh_gather_[u32]offset_u32(svbool_t pg, const int16_t *base, svuint32_t offsets)
svuint64_t	svldff1sh_gather_[u64]offset_u64(svbool_t pg, const int16_t *base, svuint64_t offsets)

### 6.2.12.5. LDFF1SH (vector base, scalar offset in bytes)

Instances	
svint32_t	svldff1sh_gather[_u32base]_offset_s32(svbool_t pg, svuint32_t bases, int64_t offset)
svint64_t	svldff1sh_gather[_u64base]_offset_s64(svbool_t pg, svuint64_t bases, int64_t offset)
svuint32_t	svldff1sh_gather[_u32base]_offset_u32(svbool_t pg, svuint32_t bases, int64_t offset)
svuint64_t	svldff1sh_gather[_u64base]_offset_u64(svbool_t pg, svuint64_t bases, int64_t offset)



Instances
<code>int64_t offset)</code>

### 6.2.12.6. LDFF1SH (scalar base, vector index)

Instances
<code>svint32_t svldff1sh_gather_[s32]index_s32(svbool_t pg, const int16_t *base, svint32_t indices)</code>
<code>svint64_t svldff1sh_gather_[s64]index_s64(svbool_t pg, const int16_t *base, svint64_t indices)</code>
<code>svuint32_t svldff1sh_gather_[s32]index_u32(svbool_t pg, const int16_t *base, svint32_t indices)</code>
<code>svuint64_t svldff1sh_gather_[s64]index_u64(svbool_t pg, const int16_t *base, svint64_t indices)</code>
<code>svint32_t svldff1sh_gather_[u32]index_s32(svbool_t pg, const int16_t *base, svuint32_t indices)</code>
<code>svint64_t svldff1sh_gather_[u64]index_s64(svbool_t pg, const int16_t *base, svuint64_t indices)</code>
<code>svuint32_t svldff1sh_gather_[u32]index_u32(svbool_t pg, const int16_t *base, svuint32_t indices)</code>
<code>svuint64_t svldff1sh_gather_[u64]index_u64(svbool_t pg, const int16_t *base, svuint64_t indices)</code>

### 6.2.12.7. LDFF1SH (vector base, scalar index)

Instances
<code>svint32_t svldff1sh_gather[_u32base]_index_s32(svbool_t pg, svuint32_t bases, int64_t index)</code>
<code>svint64_t svldff1sh_gather[_u64base]_index_s64(svbool_t pg, svuint64_t bases, int64_t index)</code>
<code>svuint32_t svldff1sh_gather[_u32base]_index_u32(svbool_t pg, svuint32_t bases, int64_t index)</code>
<code>svuint64_t svldff1sh_gather[_u64base]_index_u64(svbool_t pg, svuint64_t bases, int64_t index)</code>

## 6.2.13. LDFF1UH: Load 16-bit data and zero-extend, first-faulting

These functions attempt to load 16-bit values from memory, zero-extend them, and store the results in a vector. Only the first active element can trigger a fault.

See [Section 4.7, “First-faulting and non-faulting loads”](#) for a description of the FFR handling. Code that uses these loads should call `svrdffr` to determine which elements of the result have defined values.

### 6.2.13.1. LDFF1UH (scalar base)

Instances
<code>svint32_t svldff1uh_s32(svbool_t pg, const uint16_t *base)</code>
<code>svint64_t svldff1uh_s64(svbool_t pg, const uint16_t *base)</code>
<code>svuint32_t svldff1uh_u32(svbool_t pg, const uint16_t *base)</code>
<code>svuint64_t svldff1uh_u64(svbool_t pg, const uint16_t *base)</code>

### 6.2.13.2. LDFF1UH (scalar base, VL displacement)

Instances	
svint32_t	<b>svldff1uh_vnum_s32</b> (svbool_t <i>pg</i> , const uint16_t * <i>base</i> , int64_t <i>vnum</i> )
svint64_t	<b>svldff1uh_vnum_s64</b> (svbool_t <i>pg</i> , const uint16_t * <i>base</i> , int64_t <i>vnum</i> )
svuint32_t	<b>svldff1uh_vnum_u32</b> (svbool_t <i>pg</i> , const uint16_t * <i>base</i> , int64_t <i>vnum</i> )
svuint64_t	<b>svldff1uh_vnum_u64</b> (svbool_t <i>pg</i> , const uint16_t * <i>base</i> , int64_t <i>vnum</i> )

### 6.2.13.3. LDFF1UH (vector base)

Instances	
svint32_t	<b>svldff1uh_gather[_u32base]_s32</b> (svbool_t <i>pg</i> , svuint32_t <i>bases</i> )
svint64_t	<b>svldff1uh_gather[_u64base]_s64</b> (svbool_t <i>pg</i> , svuint64_t <i>bases</i> )
svuint32_t	<b>svldff1uh_gather[_u32base]_u32</b> (svbool_t <i>pg</i> , svuint32_t <i>bases</i> )
svuint64_t	<b>svldff1uh_gather[_u64base]_u64</b> (svbool_t <i>pg</i> , svuint64_t <i>bases</i> )

### 6.2.13.4. LDFF1UH (scalar base, vector offset in bytes)

Instances	
svint32_t	<b>svldff1uh_gather[_s32]_offset_s32</b> (svbool_t <i>pg</i> , const uint16_t * <i>base</i> , svint32_t <i>offsets</i> )
svint64_t	<b>svldff1uh_gather[_s64]_offset_s64</b> (svbool_t <i>pg</i> , const uint16_t * <i>base</i> , svint64_t <i>offsets</i> )
svuint32_t	<b>svldff1uh_gather[_s32]_offset_u32</b> (svbool_t <i>pg</i> , const uint16_t * <i>base</i> , svint32_t <i>offsets</i> )
svuint64_t	<b>svldff1uh_gather[_s64]_offset_u64</b> (svbool_t <i>pg</i> , const uint16_t * <i>base</i> , svint64_t <i>offsets</i> )
svint32_t	<b>svldff1uh_gather[_u32]_offset_s32</b> (svbool_t <i>pg</i> , const uint16_t * <i>base</i> , svuint32_t <i>offsets</i> )
svint64_t	<b>svldff1uh_gather[_u64]_offset_s64</b> (svbool_t <i>pg</i> , const uint16_t * <i>base</i> , svuint64_t <i>offsets</i> )
svuint32_t	<b>svldff1uh_gather[_u32]_offset_u32</b> (svbool_t <i>pg</i> , const uint16_t * <i>base</i> , svuint32_t <i>offsets</i> )
svuint64_t	<b>svldff1uh_gather[_u64]_offset_u64</b> (svbool_t <i>pg</i> , const uint16_t * <i>base</i> , svuint64_t <i>offsets</i> )

### 6.2.13.5. LDFF1UH (vector base, scalar offset in bytes)

Instances	
svint32_t	<b>svldff1uh_gather[_u32base]_offset_s32</b> (svbool_t <i>pg</i> , svuint32_t <i>bases</i> , int64_t <i>offset</i> )
svint64_t	<b>svldff1uh_gather[_u64base]_offset_s64</b> (svbool_t <i>pg</i> , svuint64_t <i>bases</i> , int64_t <i>offset</i> )

Instances	
svuint32_t svldffluh_gather[_u32base]_offset_u32	(svbool_t pg, int64_t offset)
svuint64_t svldffluh_gather[_u64base]_offset_u64	(svbool_t pg, svuint32_t bases, int64_t offset)

### 6.2.13.6. LDFF1UH (scalar base, vector index)

Instances	
svint32_t svldffluh_gather[_s32]_index_s32	(svbool_t pg, const uint16_t *base, svint32_t indices)
svint64_t svldffluh_gather[_s64]_index_s64	(svbool_t pg, const uint16_t *base, svint64_t indices)
svuint32_t svldffluh_gather[_s32]_index_u32	(svbool_t pg, const uint16_t *base, svint32_t indices)
svuint64_t svldffluh_gather[_s64]_index_u64	(svbool_t pg, const uint16_t *base, svint64_t indices)
svint32_t svldffluh_gather[_u32]_index_s32	(svbool_t pg, const uint16_t *base, svuint32_t indices)
svint64_t svldffluh_gather[_u64]_index_s64	(svbool_t pg, const uint16_t *base, svuint64_t indices)
svuint32_t svldffluh_gather[_u32]_index_u32	(svbool_t pg, const uint16_t *base, svuint32_t indices)
svuint64_t svldffluh_gather[_u64]_index_u64	(svbool_t pg, const uint16_t *base, svuint64_t indices)

### 6.2.13.7. LDFF1UH (vector base, scalar index)

Instances	
svint32_t svldffluh_gather[_u32base]_index_s32	(svbool_t pg, svuint32_t bases, int64_t index)
svint64_t svldffluh_gather[_u64base]_index_s64	(svbool_t pg, svuint64_t bases, int64_t index)
svuint32_t svldffluh_gather[_u32base]_index_u32	(svbool_t pg, svuint32_t bases, int64_t index)
svuint64_t svldffluh_gather[_u64base]_index_u64	(svbool_t pg, svuint64_t bases, int64_t index)

## 6.2.14. LDFF1SW: Load 32-bit data and sign-extend, first-faulting

These functions attempt to load 32-bit values from memory, sign-extend them, and store the results in a vector. Only the first active element can trigger a fault.

See [Section 4.7, “First-faulting and non-faulting loads”](#) for a description of the FFR handling. Code that uses these loads should call `svrdffr` to determine which elements of the result have defined values.

### 6.2.14.1. LDFF1SW (scalar base)

Instances
<pre>svint64_t svldff1sw_s64(svbool_t pg, const int32_t *base) svuint64_t svldff1sw_u64(svbool_t pg, const int32_t *base)</pre>

### 6.2.14.2. LDFF1SW (scalar base, VL displacement)

Instances
<pre>svint64_t svldff1sw_vnum_s64(svbool_t pg, const int32_t *base,                              int64_t vnum) svuint64_t svldff1sw_vnum_u64(svbool_t pg, const int32_t *base,                               int64_t vnum)</pre>

### 6.2.14.3. LDFF1SW (vector base)

Instances
<pre>svint64_t svldff1sw_gather[_u64base]_s64(svbool_t pg, svuint64_t bases) svuint64_t svldff1sw_gather[_u64base]_u64(svbool_t pg, svuint64_t bases)</pre>

### 6.2.14.4. LDFF1SW (scalar base, vector offset in bytes)

Instances
<pre>svint64_t svldff1sw_gather[_s64]_offset_s64(svbool_t pg, const int32_t *base,   svint64_t offsets) svuint64_t svldff1sw_gather[_s64]_offset_u64(svbool_t pg,   const int32_t *base,   svint64_t offsets) svint64_t svldff1sw_gather[_u64]_offset_s64(svbool_t pg, const int32_t *base,   svuint64_t offsets) svuint64_t svldff1sw_gather[_u64]_offset_u64(svbool_t pg,   const int32_t *base,   svuint64_t offsets)</pre>

### 6.2.14.5. LDFF1SW (vector base, scalar offset in bytes)

Instances
<pre>svint64_t svldff1sw_gather[_u64base]_offset_s64(svbool_t pg,   svuint64_t bases,   int64_t offset) svuint64_t svldff1sw_gather[_u64base]_offset_u64(svbool_t pg,   svuint64_t bases,   int64_t offset)</pre>

### 6.2.14.6. LDFF1SW (scalar base, vector index)

Instances
<pre>svint64_t svldff1sw_gather[_s64]_index_s64(svbool_t pg, const int32_t *base,   svint64_t indices) svuint64_t svldff1sw_gather[_s64]_index_u64(svbool_t pg, const int32_t *base,   svint64_t indices) svint64_t svldff1sw_gather[_u64]_index_s64(svbool_t pg, const int32_t *base,   svuint64_t indices) svuint64_t svldff1sw_gather[_u64]_index_u64(svbool_t pg, const int32_t *base,</pre>

Instances
<code>svuint64_t indices)</code>

### 6.2.14.7. LDFF1SW (vector base, scalar index)

Instances
<pre>svint64_t svldff1sw_gather[_u64base]_index_s64(svbool_t pg,   svuint64_t bases,   int64_t index) svuint64_t svldff1sw_gather[_u64base]_index_u64(svbool_t pg,   svuint64_t bases,   int64_t index)</pre>

## 6.2.15. LDFF1UW: Load 32-bit data and zero-extend, first-faulting

These functions attempt to load 32-bit values from memory, zero-extend them, and store the results in a vector. Only the first active element can trigger a fault.

See [Section 4.7, “First-faulting and non-faulting loads”](#) for a description of the FFR handling. Code that uses these loads should call `svrddffr` to determine which elements of the result have defined values.

### 6.2.15.1. LDFF1UW (scalar base)

Instances
<pre>svint64_t svldff1uw_s64(svbool_t pg, const uint32_t *base) svuint64_t svldff1uw_u64(svbool_t pg, const uint32_t *base)</pre>

### 6.2.15.2. LDFF1UW (scalar base, VL displacement)

Instances
<pre>svint64_t svldff1uw_vnum_s64(svbool_t pg, const uint32_t *base,                              int64_t vnum) svuint64_t svldff1uw_vnum_u64(svbool_t pg, const uint32_t *base,                              int64_t vnum)</pre>

### 6.2.15.3. LDFF1UW (vector base)

Instances
<pre>svint64_t svldff1uw_gather[_u64base]_s64(svbool_t pg, svuint64_t bases) svuint64_t svldff1uw_gather[_u64base]_u64(svbool_t pg, svuint64_t bases)</pre>

### 6.2.15.4. LDFF1UW (scalar base, vector offset in bytes)

Instances
<pre>svint64_t svldff1uw_gather[_s64]_offset_s64(svbool_t pg,   const uint32_t *base,   svint64_t offsets) svuint64_t svldff1uw_gather[_s64]_offset_u64(svbool_t pg,   const uint32_t *base,   svint64_t offsets) svint64_t svldff1uw_gather[_u64]_offset_s64(svbool_t pg,   const uint32_t *base,   svuint64_t offsets) svuint64_t svldff1uw_gather[_u64]_offset_u64(svbool_t pg,</pre>

Instances
<pre>const uint32_t *base, svuint64_t offsets)</pre>

### 6.2.15.5. LDFF1UW (vector base, scalar offset in bytes)

Instances
<pre>svint64_t svldffluw_gather[_u64base]_offset_s64(svbool_t pg,   svuint64_t bases,   int64_t offset) svuint64_t svldffluw_gather[_u64base]_offset_u64(svbool_t pg,   svuint64_t bases,   int64_t offset)</pre>

### 6.2.15.6. LDFF1UW (scalar base, vector index)

Instances
<pre>svint64_t svldffluw_gather[_s64]_index_s64(svbool_t pg, const uint32_t *base,   svint64_t indices) svuint64_t svldffluw_gather[_s64]_index_u64(svbool_t pg,   const uint32_t *base,   svint64_t indices) svint64_t svldffluw_gather[_u64]_index_s64(svbool_t pg, const uint32_t *base,   svuint64_t indices) svuint64_t svldffluw_gather[_u64]_index_u64(svbool_t pg,   const uint32_t *base,   svuint64_t indices)</pre>

### 6.2.15.7. LDFF1UW (vector base, scalar index)

Instances
<pre>svint64_t svldffluw_gather[_u64base]_index_s64(svbool_t pg,   svuint64_t bases,   int64_t index) svuint64_t svldffluw_gather[_u64base]_index_u64(svbool_t pg,   svuint64_t bases,   int64_t index)</pre>

## 6.2.16. LDNF1: Unextended load, non-faulting

These functions attempt to load values from memory and store the results in a vector, but without dereferencing any pointers that would fault<sup>5</sup>. The vector elements have the same width as the loaded data; the functions do not perform any kind of extension.

See [Section 4.7, “First-faulting and non-faulting loads”](#) for a description of the FFR handling. Code that uses these loads should call `svrdffr` to determine which elements of the result have defined values.

### 6.2.16.1. LDNF1 (scalar base)

Instances
<pre>svint8_t svldnf1[_s8](svbool_t pg, const int8_t *base)</pre>

<sup>5</sup> In other words, the functions behave like `svldff1` ([Section 6.2.9, “LDFF1: Unextended load, first-faulting”](#)) but without the special treatment of the first active element.

**Instances**

```

svint16_t  svldnfl[_s16](svbool_t pg, const int16_t *base)
svint32_t  svldnfl[_s32](svbool_t pg, const int32_t *base)
svint64_t  svldnfl[_s64](svbool_t pg, const int64_t *base)
svuint8_t  svldnfl[_u8](svbool_t pg, const uint8_t *base)
svuint16_t svldnfl[_u16](svbool_t pg, const uint16_t *base)
svuint32_t svldnfl[_u32](svbool_t pg, const uint32_t *base)
svuint64_t svldnfl[_u64](svbool_t pg, const uint64_t *base)
svfloat16_t svldnfl[_f16](svbool_t pg, const float16_t *base)
svfloat32_t svldnfl[_f32](svbool_t pg, const float32_t *base)
svfloat64_t svldnfl[_f64](svbool_t pg, const float64_t *base)
svbfloat16_t svldnfl[_bf16](svbool_t pg, const bfloat16_t *base)

```

**6.2.16.2. LDNF1 (scalar base, VL displacement)****Instances**

```

svint8_t  svldnfl_vnum[_s8](svbool_t pg, const int8_t *base, int64_t vnum)
svint16_t svldnfl_vnum[_s16](svbool_t pg, const int16_t *base,
                               int64_t vnum)
svint32_t svldnfl_vnum[_s32](svbool_t pg, const int32_t *base,
                               int64_t vnum)
svint64_t svldnfl_vnum[_s64](svbool_t pg, const int64_t *base,
                               int64_t vnum)
svuint8_t  svldnfl_vnum[_u8](svbool_t pg, const uint8_t *base, int64_t vnum)
svuint16_t svldnfl_vnum[_u16](svbool_t pg, const uint16_t *base,
                               int64_t vnum)
svuint32_t svldnfl_vnum[_u32](svbool_t pg, const uint32_t *base,
                               int64_t vnum)
svuint64_t svldnfl_vnum[_u64](svbool_t pg, const uint64_t *base,
                               int64_t vnum)
svfloat16_t svldnfl_vnum[_f16](svbool_t pg, const float16_t *base,
                               int64_t vnum)
svfloat32_t svldnfl_vnum[_f32](svbool_t pg, const float32_t *base,
                               int64_t vnum)
svfloat64_t svldnfl_vnum[_f64](svbool_t pg, const float64_t *base,
                               int64_t vnum)
svbfloat16_t svldnfl_vnum[_bf16](svbool_t pg, const bfloat16_t *base,
                               int64_t vnum)

```

**6.2.17. LDNF1SB: Load 8-bit data and sign-extend, non-faulting**

These functions attempt to load 8-bit values from memory, sign-extend them, and store the results in a vector. None of the accesses trigger a fault.

See [Section 4.7, “First-faulting and non-faulting loads”](#) for a description of the FFR handling. Code that uses these loads should call `svrddfrr` to determine which elements of the result have defined values.

**6.2.17.1. LDNF1SB (scalar base)****Instances**

```

svint16_t  svldnflsb_s16(svbool_t pg, const int8_t *base)
svint32_t  svldnflsb_s32(svbool_t pg, const int8_t *base)
svint64_t  svldnflsb_s64(svbool_t pg, const int8_t *base)
svuint16_t svldnflsb_u16(svbool_t pg, const int8_t *base)
svuint32_t svldnflsb_u32(svbool_t pg, const int8_t *base)
svuint64_t svldnflsb_u64(svbool_t pg, const int8_t *base)

```

### 6.2.17.2. LDNF1SB (scalar base, VL displacement)

Instances	
<code>svint16_t</code>	<code>svldnf1sb_vnum_s16(svbool_t pg, const int8_t *base, int64_t vnum)</code>
<code>svint32_t</code>	<code>svldnf1sb_vnum_s32(svbool_t pg, const int8_t *base, int64_t vnum)</code>
<code>svint64_t</code>	<code>svldnf1sb_vnum_s64(svbool_t pg, const int8_t *base, int64_t vnum)</code>
<code>svuint16_t</code>	<code>svldnf1sb_vnum_u16(svbool_t pg, const int8_t *base, int64_t vnum)</code>
<code>svuint32_t</code>	<code>svldnf1sb_vnum_u32(svbool_t pg, const int8_t *base, int64_t vnum)</code>
<code>svuint64_t</code>	<code>svldnf1sb_vnum_u64(svbool_t pg, const int8_t *base, int64_t vnum)</code>

### 6.2.18. LDNF1UB: Load 8-bit data and zero-extend, non-faulting

These functions attempt to load 8-bit values from memory, zero-extend them, and store the results in a vector. None of the accesses trigger a fault.

See [Section 4.7, “First-faulting and non-faulting loads”](#) for a description of the FFR handling. Code that uses these loads should call `svrddfrr` to determine which elements of the result have defined values.

#### 6.2.18.1. LDNF1UB (scalar base)

Instances	
<code>svint16_t</code>	<code>svldnf1ub_s16(svbool_t pg, const uint8_t *base)</code>
<code>svint32_t</code>	<code>svldnf1ub_s32(svbool_t pg, const uint8_t *base)</code>
<code>svint64_t</code>	<code>svldnf1ub_s64(svbool_t pg, const uint8_t *base)</code>
<code>svuint16_t</code>	<code>svldnf1ub_u16(svbool_t pg, const uint8_t *base)</code>
<code>svuint32_t</code>	<code>svldnf1ub_u32(svbool_t pg, const uint8_t *base)</code>
<code>svuint64_t</code>	<code>svldnf1ub_u64(svbool_t pg, const uint8_t *base)</code>

#### 6.2.18.2. LDNF1UB (scalar base, VL displacement)

Instances	
<code>svint16_t</code>	<code>svldnf1ub_vnum_s16(svbool_t pg, const uint8_t *base, int64_t vnum)</code>
<code>svint32_t</code>	<code>svldnf1ub_vnum_s32(svbool_t pg, const uint8_t *base, int64_t vnum)</code>
<code>svint64_t</code>	<code>svldnf1ub_vnum_s64(svbool_t pg, const uint8_t *base, int64_t vnum)</code>
<code>svuint16_t</code>	<code>svldnf1ub_vnum_u16(svbool_t pg, const uint8_t *base, int64_t vnum)</code>
<code>svuint32_t</code>	<code>svldnf1ub_vnum_u32(svbool_t pg, const uint8_t *base, int64_t vnum)</code>
<code>svuint64_t</code>	<code>svldnf1ub_vnum_u64(svbool_t pg, const uint8_t *base, int64_t vnum)</code>

### 6.2.19. LDNF1SH: Load 16-bit data and sign-extend, non-faulting

These functions attempt to load 16-bit values from memory, sign-extend them, and store the results in a vector. None of the accesses trigger a fault.

See [Section 4.7, “First-faulting and non-faulting loads”](#) for a description of the FFR handling. Code that uses these loads should call `svrddfrr` to determine which elements of the result have defined values.



### 6.2.19.1. LDNF1SH (scalar base)

#### Instances

```
svint32_t svldnf1sh_s32(svbool_t pg, const int16_t *base)
svint64_t svldnf1sh_s64(svbool_t pg, const int16_t *base)
svuint32_t svldnf1sh_u32(svbool_t pg, const int16_t *base)
svuint64_t svldnf1sh_u64(svbool_t pg, const int16_t *base)
```

### 6.2.19.2. LDNF1SH (scalar base, VL displacement)

#### Instances

```
svint32_t svldnf1sh_vnum_s32(svbool_t pg, const int16_t *base,
                             int64_t vnum)
svint64_t svldnf1sh_vnum_s64(svbool_t pg, const int16_t *base,
                             int64_t vnum)
svuint32_t svldnf1sh_vnum_u32(svbool_t pg, const int16_t *base,
                              int64_t vnum)
svuint64_t svldnf1sh_vnum_u64(svbool_t pg, const int16_t *base,
                              int64_t vnum)
```

## 6.2.20. LDNF1UH: Load 16-bit data and zero-extend, non-faulting

These functions attempt to load 16-bit values from memory, zero-extend them, and store the results in a vector. None of the accesses trigger a fault.

See [Section 4.7, “First-faulting and non-faulting loads”](#) for a description of the FFR handling. Code that uses these loads should call `svrddfr` to determine which elements of the result have defined values.

### 6.2.20.1. LDNF1UH (scalar base)

#### Instances

```
svint32_t svldnf1uh_s32(svbool_t pg, const uint16_t *base)
svint64_t svldnf1uh_s64(svbool_t pg, const uint16_t *base)
svuint32_t svldnf1uh_u32(svbool_t pg, const uint16_t *base)
svuint64_t svldnf1uh_u64(svbool_t pg, const uint16_t *base)
```

### 6.2.20.2. LDNF1UH (scalar base, VL displacement)

#### Instances

```
svint32_t svldnf1uh_vnum_s32(svbool_t pg, const uint16_t *base,
                             int64_t vnum)
svint64_t svldnf1uh_vnum_s64(svbool_t pg, const uint16_t *base,
                             int64_t vnum)
svuint32_t svldnf1uh_vnum_u32(svbool_t pg, const uint16_t *base,
                              int64_t vnum)
svuint64_t svldnf1uh_vnum_u64(svbool_t pg, const uint16_t *base,
                              int64_t vnum)
```

## 6.2.21. LDNF1SW: Load 32-bit data and sign-extend, non-faulting

These functions attempt to load 32-bit values from memory, sign-extend them, and store the results in a vector. None of the accesses trigger a fault.

See [Section 4.7, “First-faulting and non-faulting loads”](#) for a description of the FFR handling. Code that uses these loads should call `svrddfr` to determine which elements of the result have defined values.

### 6.2.21.1. LDNF1SW (scalar base)

Instances
<code>svint64_t svldnflsw_s64(svbool_t pg, const int32_t *base)</code>
<code>svuint64_t svldnflsw_u64(svbool_t pg, const int32_t *base)</code>

### 6.2.21.2. LDNF1SW (scalar base, VL displacement)

Instances
<code>svint64_t svldnflsw_vnum_s64(svbool_t pg, const int32_t *base, int64_t vnum)</code>
<code>svuint64_t svldnflsw_vnum_u64(svbool_t pg, const int32_t *base, int64_t vnum)</code>

## 6.2.22. LDNF1UW: Load 32-bit data and zero-extend, non-faulting

These functions attempt to load 32-bit values from memory, zero-extend them, and store the results in a vector. None of the accesses trigger a fault.

See [Section 4.7, “First-faulting and non-faulting loads”](#) for a description of the FFR handling. Code that uses these loads should call `svrddfr` to determine which elements of the result have defined values.

### 6.2.22.1. LDNF1UW (scalar base)

Instances
<code>svint64_t svldnfluw_s64(svbool_t pg, const uint32_t *base)</code>
<code>svuint64_t svldnfluw_u64(svbool_t pg, const uint32_t *base)</code>

### 6.2.22.2. LDNF1UW (scalar base, VL displacement)

Instances
<code>svint64_t svldnfluw_vnum_s64(svbool_t pg, const uint32_t *base, int64_t vnum)</code>
<code>svuint64_t svldnfluw_vnum_u64(svbool_t pg, const uint32_t *base, int64_t vnum)</code>

## 6.2.23. LDNT1: Unextended load, non-temporal

These functions behave like the loads in [Section 6.2.1, “LD1: Unextended load”](#), but additionally provide a hint to the system that the loaded memory is unlikely to be accessed again soon.

### 6.2.23.1. LDNT1 (scalar base)

Instances
<code>svint8_t svldnt1[_s8](svbool_t pg, const int8_t *base)</code>
<code>svint16_t svldnt1[_s16](svbool_t pg, const int16_t *base)</code>
<code>svint32_t svldnt1[_s32](svbool_t pg, const int32_t *base)</code>
<code>svint64_t svldnt1[_s64](svbool_t pg, const int64_t *base)</code>

**Instances**

```

svuint8_t  svldnt1[_u8](svbool_t pg, const uint8_t *base)
svuint16_t svldnt1[_u16](svbool_t pg, const uint16_t *base)
svuint32_t svldnt1[_u32](svbool_t pg, const uint32_t *base)
svuint64_t svldnt1[_u64](svbool_t pg, const uint64_t *base)
svfloat16_t svldnt1[_f16](svbool_t pg, const float16_t *base)
svfloat32_t svldnt1[_f32](svbool_t pg, const float32_t *base)
svfloat64_t svldnt1[_f64](svbool_t pg, const float64_t *base)
svbfloat16_t svldnt1[_bf16](svbool_t pg, const bfloat16_t *base)

```

**6.2.23.2. LDNT1 (scalar base, VL displacement)****Instances**

```

svint8_t  svldnt1_vnum[_s8](svbool_t pg, const int8_t *base, int64_t vnum)
svint16_t svldnt1_vnum[_s16](svbool_t pg, const int16_t *base,
                               int64_t vnum)
svint32_t svldnt1_vnum[_s32](svbool_t pg, const int32_t *base,
                               int64_t vnum)
svint64_t svldnt1_vnum[_s64](svbool_t pg, const int64_t *base,
                               int64_t vnum)
svuint8_t  svldnt1_vnum[_u8](svbool_t pg, const uint8_t *base, int64_t vnum)
svuint16_t svldnt1_vnum[_u16](svbool_t pg, const uint16_t *base,
                               int64_t vnum)
svuint32_t svldnt1_vnum[_u32](svbool_t pg, const uint32_t *base,
                               int64_t vnum)
svuint64_t svldnt1_vnum[_u64](svbool_t pg, const uint64_t *base,
                               int64_t vnum)
svfloat16_t svldnt1_vnum[_f16](svbool_t pg, const float16_t *base,
                               int64_t vnum)
svfloat32_t svldnt1_vnum[_f32](svbool_t pg, const float32_t *base,
                               int64_t vnum)
svfloat64_t svldnt1_vnum[_f64](svbool_t pg, const float64_t *base,
                               int64_t vnum)
svbfloat16_t svldnt1_vnum[_bf16](svbool_t pg, const bfloat16_t *base,
                               int64_t vnum)

```

**6.2.24. LD2: Load two-element structures into two vectors**

These functions load an array of two-element structures into two vectors, with one vector per structure element. In other words, element *N* of vector *M* corresponds to element *M* of structure *N*.

**6.2.24.1. LD2 (scalar base)****Instances**

```

svint8x2_t  svld2[_s8](svbool_t pg, const int8_t *base)
svint16x2_t svld2[_s16](svbool_t pg, const int16_t *base)
svint32x2_t svld2[_s32](svbool_t pg, const int32_t *base)
svint64x2_t svld2[_s64](svbool_t pg, const int64_t *base)
svuint8x2_t  svld2[_u8](svbool_t pg, const uint8_t *base)
svuint16x2_t svld2[_u16](svbool_t pg, const uint16_t *base)
svuint32x2_t svld2[_u32](svbool_t pg, const uint32_t *base)
svuint64x2_t svld2[_u64](svbool_t pg, const uint64_t *base)
svfloat16x2_t svld2[_f16](svbool_t pg, const float16_t *base)
svfloat32x2_t svld2[_f32](svbool_t pg, const float32_t *base)
svfloat64x2_t svld2[_f64](svbool_t pg, const float64_t *base)
svbfloat16x2_t svld2[_bf16](svbool_t pg, const bfloat16_t *base)

```

### 6.2.24.2. LD2 (scalar base, VL displacement)

Instances
<pre> svint8x2_t  svld2_vnum[_s8](svbool_t pg, const int8_t *base, int64_t vnum) svint16x2_t svld2_vnum[_s16](svbool_t pg, const int16_t *base,                              int64_t vnum) svint32x2_t svld2_vnum[_s32](svbool_t pg, const int32_t *base,                              int64_t vnum) svint64x2_t svld2_vnum[_s64](svbool_t pg, const int64_t *base,                              int64_t vnum) svuint8x2_t  svld2_vnum[_u8](svbool_t pg, const uint8_t *base, int64_t vnum) svuint16x2_t svld2_vnum[_u16](svbool_t pg, const uint16_t *base,                               int64_t vnum) svuint32x2_t svld2_vnum[_u32](svbool_t pg, const uint32_t *base,                               int64_t vnum) svuint64x2_t svld2_vnum[_u64](svbool_t pg, const uint64_t *base,                               int64_t vnum) svfloat16x2_t svld2_vnum[_f16](svbool_t pg, const float16_t *base,                                 int64_t vnum) svfloat32x2_t svld2_vnum[_f32](svbool_t pg, const float32_t *base,                                 int64_t vnum) svfloat64x2_t svld2_vnum[_f64](svbool_t pg, const float64_t *base,                                 int64_t vnum) svbfloat16x2_t svld2_vnum[_bf16](svbool_t pg, const bfloat16_t *base,                                   int64_t vnum) </pre>

### 6.2.25. LD3: Load three-element structures into three vectors

These functions load an array of three-element structures into three vectors, with one vector per structure element. In other words, element *N* of vector *M* corresponds to element *M* of structure *N*.

#### 6.2.25.1. LD3 (scalar base)

Instances
<pre> svint8x3_t  svld3[_s8](svbool_t pg, const int8_t *base) svint16x3_t svld3[_s16](svbool_t pg, const int16_t *base) svint32x3_t svld3[_s32](svbool_t pg, const int32_t *base) svint64x3_t svld3[_s64](svbool_t pg, const int64_t *base) svuint8x3_t  svld3[_u8](svbool_t pg, const uint8_t *base) svuint16x3_t svld3[_u16](svbool_t pg, const uint16_t *base) svuint32x3_t svld3[_u32](svbool_t pg, const uint32_t *base) svuint64x3_t svld3[_u64](svbool_t pg, const uint64_t *base) svfloat16x3_t svld3[_f16](svbool_t pg, const float16_t *base) svfloat32x3_t svld3[_f32](svbool_t pg, const float32_t *base) svfloat64x3_t svld3[_f64](svbool_t pg, const float64_t *base) svbfloat16x3_t svld3[_bf16](svbool_t pg, const bfloat16_t *base) </pre>

#### 6.2.25.2. LD3 (scalar base, VL displacement)

Instances
<pre> svint8x3_t  svld3_vnum[_s8](svbool_t pg, const int8_t *base, int64_t vnum) svint16x3_t svld3_vnum[_s16](svbool_t pg, const int16_t *base,                              int64_t vnum) svint32x3_t svld3_vnum[_s32](svbool_t pg, const int32_t *base,                              int64_t vnum) </pre>

**Instances**

```

svint64x3_t svld3_vnum[_s64](svbool_t pg, const int64_t *base,
                             int64_t vnum)
svuint8x3_t svld3_vnum[_u8](svbool_t pg, const uint8_t *base, int64_t vnum)
svuint16x3_t svld3_vnum[_u16](svbool_t pg, const uint16_t *base,
                              int64_t vnum)
svuint32x3_t svld3_vnum[_u32](svbool_t pg, const uint32_t *base,
                              int64_t vnum)
svuint64x3_t svld3_vnum[_u64](svbool_t pg, const uint64_t *base,
                              int64_t vnum)
svfloat16x3_t svld3_vnum[_f16](svbool_t pg, const float16_t *base,
                               int64_t vnum)
svfloat32x3_t svld3_vnum[_f32](svbool_t pg, const float32_t *base,
                               int64_t vnum)
svfloat64x3_t svld3_vnum[_f64](svbool_t pg, const float64_t *base,
                               int64_t vnum)
svbfloat16x3_t svld3_vnum[_bf16](svbool_t pg, const bfloat16_t *base,
                                 int64_t vnum)

```

**6.2.26. LD4: Load four-element structures into four vectors**

These functions load an array of four-element structures into four vectors, with one vector per structure element. In other words, element *N* of vector *M* corresponds to element *M* of structure *N*.

**6.2.26.1. LD4 (scalar base)****Instances**

```

svint8x4_t svld4[_s8](svbool_t pg, const int8_t *base)
svint16x4_t svld4[_s16](svbool_t pg, const int16_t *base)
svint32x4_t svld4[_s32](svbool_t pg, const int32_t *base)
svint64x4_t svld4[_s64](svbool_t pg, const int64_t *base)
svuint8x4_t svld4[_u8](svbool_t pg, const uint8_t *base)
svuint16x4_t svld4[_u16](svbool_t pg, const uint16_t *base)
svuint32x4_t svld4[_u32](svbool_t pg, const uint32_t *base)
svuint64x4_t svld4[_u64](svbool_t pg, const uint64_t *base)
svfloat16x4_t svld4[_f16](svbool_t pg, const float16_t *base)
svfloat32x4_t svld4[_f32](svbool_t pg, const float32_t *base)
svfloat64x4_t svld4[_f64](svbool_t pg, const float64_t *base)
svbfloat16x4_t svld4[_bf16](svbool_t pg, const bfloat16_t *base)

```

**6.2.26.2. LD4 (scalar base, VL displacement)****Instances**

```

svint8x4_t svld4_vnum[_s8](svbool_t pg, const int8_t *base, int64_t vnum)
svint16x4_t svld4_vnum[_s16](svbool_t pg, const int16_t *base,
                             int64_t vnum)
svint32x4_t svld4_vnum[_s32](svbool_t pg, const int32_t *base,
                              int64_t vnum)
svint64x4_t svld4_vnum[_s64](svbool_t pg, const int64_t *base,
                              int64_t vnum)
svuint8x4_t svld4_vnum[_u8](svbool_t pg, const uint8_t *base, int64_t vnum)
svuint16x4_t svld4_vnum[_u16](svbool_t pg, const uint16_t *base,
                              int64_t vnum)
svuint32x4_t svld4_vnum[_u32](svbool_t pg, const uint32_t *base,
                              int64_t vnum)
svuint64x4_t svld4_vnum[_u64](svbool_t pg, const uint64_t *base,
                              int64_t vnum)

```

**Instances**

```

                                int64_t vnum)
svfloat16x4_t svld4_vnum[_f16](svbool_t pg, const float16_t *base,
                                int64_t vnum)
svfloat32x4_t svld4_vnum[_f32](svbool_t pg, const float32_t *base,
                                int64_t vnum)
svfloat64x4_t svld4_vnum[_f64](svbool_t pg, const float64_t *base,
                                int64_t vnum)
svbfloat16x4_t svld4_vnum[_bf16](svbool_t pg, const bfloat16_t *base,
                                int64_t vnum)

```

## 6.3. Stores

### 6.3.1. ST1: Store one vector, with no truncation

These functions read elements from a vector and store them to memory. The vector elements have the same width as the stored data; the functions do not perform any kind of truncation.

#### 6.3.1.1. ST1 (scalar base)

**Instances**

```

void svst1[_s8](svbool_t pg, int8_t *base, svint8_t data)
void svst1[_s16](svbool_t pg, int16_t *base, svint16_t data)
void svst1[_s32](svbool_t pg, int32_t *base, svint32_t data)
void svst1[_s64](svbool_t pg, int64_t *base, svint64_t data)
void svst1[_u8](svbool_t pg, uint8_t *base, svuint8_t data)
void svst1[_u16](svbool_t pg, uint16_t *base, svuint16_t data)
void svst1[_u32](svbool_t pg, uint32_t *base, svuint32_t data)
void svst1[_u64](svbool_t pg, uint64_t *base, svuint64_t data)
void svst1[_f16](svbool_t pg, float16_t *base, svfloat16_t data)
void svst1[_f32](svbool_t pg, float32_t *base, svfloat32_t data)
void svst1[_f64](svbool_t pg, float64_t *base, svfloat64_t data)
void svst1[_bf16](svbool_t pg, bfloat16_t *base, svbfloat16_t data)

```

#### 6.3.1.2. ST1 (scalar base, VL displacement)

**Instances**

```

void svst1_vnum[_s8](svbool_t pg, int8_t *base, int64_t vnum,
                    svint8_t data)
void svst1_vnum[_s16](svbool_t pg, int16_t *base, int64_t vnum,
                    svint16_t data)
void svst1_vnum[_s32](svbool_t pg, int32_t *base, int64_t vnum,
                    svint32_t data)
void svst1_vnum[_s64](svbool_t pg, int64_t *base, int64_t vnum,
                    svint64_t data)
void svst1_vnum[_u8](svbool_t pg, uint8_t *base, int64_t vnum,
                    svuint8_t data)
void svst1_vnum[_u16](svbool_t pg, uint16_t *base, int64_t vnum,
                    svuint16_t data)
void svst1_vnum[_u32](svbool_t pg, uint32_t *base, int64_t vnum,
                    svuint32_t data)
void svst1_vnum[_u64](svbool_t pg, uint64_t *base, int64_t vnum,
                    svuint64_t data)
void svst1_vnum[_f16](svbool_t pg, float16_t *base, int64_t vnum,
                    svfloat16_t data)

```

**Instances**

```
void svstl_vnum[_f32](svbool_t pg, float32_t *base, int64_t vnum,
                     svfloat32_t data)
void svstl_vnum[_f64](svbool_t pg, float64_t *base, int64_t vnum,
                     svfloat64_t data)
void svstl_vnum[_bf16](svbool_t pg, bfloat16_t *base, int64_t vnum,
                     svbfloat16_t data)
```

**6.3.1.3. ST1 (vector base)****Instances**

```
void svstl_scatter[_u32base_s32](svbool_t pg, svuint32_t bases,
                                svint32_t data)
void svstl_scatter[_u64base_s64](svbool_t pg, svuint64_t bases,
                                svint64_t data)
void svstl_scatter[_u32base_u32](svbool_t pg, svuint32_t bases,
                                svuint32_t data)
void svstl_scatter[_u64base_u64](svbool_t pg, svuint64_t bases,
                                svuint64_t data)
void svstl_scatter[_u32base_f32](svbool_t pg, svuint32_t bases,
                                svfloat32_t data)
void svstl_scatter[_u64base_f64](svbool_t pg, svuint64_t bases,
                                svfloat64_t data)
```

**6.3.1.4. ST1 (scalar base, vector offset in bytes)****Instances**

```
void svstl_scatter[_s32]offset[_s32](svbool_t pg, int32_t *base,
                                     svint32_t offsets, svint32_t data)
void svstl_scatter[_s64]offset[_s64](svbool_t pg, int64_t *base,
                                     svint64_t offsets, svint64_t data)
void svstl_scatter[_s32]offset[_u32](svbool_t pg, uint32_t *base,
                                     svint32_t offsets, svuint32_t data)
void svstl_scatter[_s64]offset[_u64](svbool_t pg, uint64_t *base,
                                     svint64_t offsets, svuint64_t data)
void svstl_scatter[_s32]offset[_f32](svbool_t pg, float32_t *base,
                                     svint32_t offsets, svfloat32_t data)
void svstl_scatter[_s64]offset[_f64](svbool_t pg, float64_t *base,
                                     svint64_t offsets, svfloat64_t data)
void svstl_scatter[_u32]offset[_s32](svbool_t pg, int32_t *base,
                                     svuint32_t offsets, svint32_t data)
void svstl_scatter[_u64]offset[_s64](svbool_t pg, int64_t *base,
                                     svuint64_t offsets, svint64_t data)
void svstl_scatter[_u32]offset[_u32](svbool_t pg, uint32_t *base,
                                     svuint32_t offsets, svuint32_t data)
void svstl_scatter[_u64]offset[_u64](svbool_t pg, uint64_t *base,
                                     svuint64_t offsets, svuint64_t data)
void svstl_scatter[_u32]offset[_f32](svbool_t pg, float32_t *base,
                                     svuint32_t offsets, svfloat32_t data)
void svstl_scatter[_u64]offset[_f64](svbool_t pg, float64_t *base,
                                     svuint64_t offsets, svfloat64_t data)
```

**6.3.1.5. ST1 (vector base, scalar offset in bytes)****Instances**

```
void svstl_scatter[_u32base]_offset[_s32](svbool_t pg, svuint32_t bases,
```

Instances	
<code>void svstl_scatter[_u64base]_offset[_s64]</code>	<code>(svbool_t pg, svuint64_t bases, int64_t offset, svint32_t data)</code>
<code>void svstl_scatter[_u32base]_offset[_u32]</code>	<code>(svbool_t pg, svuint32_t bases, int64_t offset, svuint32_t data)</code>
<code>void svstl_scatter[_u64base]_offset[_u64]</code>	<code>(svbool_t pg, svuint64_t bases, int64_t offset, svuint64_t data)</code>
<code>void svstl_scatter[_u32base]_offset[_f32]</code>	<code>(svbool_t pg, svuint32_t bases, int64_t offset, svfloat32_t data)</code>
<code>void svstl_scatter[_u64base]_offset[_f64]</code>	<code>(svbool_t pg, svuint64_t bases, int64_t offset, svfloat64_t data)</code>

### 6.3.1.6. ST1 (scalar base, vector index)

Instances	
<code>void svstl_scatter[_s32]_index[_s32]</code>	<code>(svbool_t pg, int32_t *base, svint32_t indices, svint32_t data)</code>
<code>void svstl_scatter[_s64]_index[_s64]</code>	<code>(svbool_t pg, int64_t *base, svint64_t indices, svint64_t data)</code>
<code>void svstl_scatter[_s32]_index[_u32]</code>	<code>(svbool_t pg, uint32_t *base, svint32_t indices, svuint32_t data)</code>
<code>void svstl_scatter[_s64]_index[_u64]</code>	<code>(svbool_t pg, uint64_t *base, svint64_t indices, svuint64_t data)</code>
<code>void svstl_scatter[_s32]_index[_f32]</code>	<code>(svbool_t pg, float32_t *base, svint32_t indices, svfloat32_t data)</code>
<code>void svstl_scatter[_s64]_index[_f64]</code>	<code>(svbool_t pg, float64_t *base, svint64_t indices, svfloat64_t data)</code>
<code>void svstl_scatter[_u32]_index[_s32]</code>	<code>(svbool_t pg, int32_t *base, svuint32_t indices, svint32_t data)</code>
<code>void svstl_scatter[_u64]_index[_s64]</code>	<code>(svbool_t pg, int64_t *base, svuint64_t indices, svint64_t data)</code>
<code>void svstl_scatter[_u32]_index[_u32]</code>	<code>(svbool_t pg, uint32_t *base, svuint32_t indices, svuint32_t data)</code>
<code>void svstl_scatter[_u64]_index[_u64]</code>	<code>(svbool_t pg, uint64_t *base, svuint64_t indices, svuint64_t data)</code>
<code>void svstl_scatter[_u32]_index[_f32]</code>	<code>(svbool_t pg, float32_t *base, svuint32_t indices, svfloat32_t data)</code>
<code>void svstl_scatter[_u64]_index[_f64]</code>	<code>(svbool_t pg, float64_t *base, svuint64_t indices, svfloat64_t data)</code>

### 6.3.1.7. ST1 (vector base, scalar index)

Instances	
<code>void svstl_scatter[_u32base]_index[_s32]</code>	<code>(svbool_t pg, svuint32_t bases, int64_t index, svint32_t data)</code>
<code>void svstl_scatter[_u64base]_index[_s64]</code>	<code>(svbool_t pg, svuint64_t bases, int64_t index, svint64_t data)</code>
<code>void svstl_scatter[_u32base]_index[_u32]</code>	<code>(svbool_t pg, svuint32_t bases, int64_t index, svuint32_t data)</code>
<code>void svstl_scatter[_u64base]_index[_u64]</code>	<code>(svbool_t pg, svuint64_t bases, int64_t index, svuint64_t data)</code>
<code>void svstl_scatter[_u32base]_index[_f32]</code>	<code>(svbool_t pg, svuint32_t bases, int64_t index, svfloat32_t data)</code>
<code>void svstl_scatter[_u64base]_index[_f64]</code>	<code>(svbool_t pg, svuint64_t bases, int64_t index, svfloat64_t data)</code>



### 6.3.2. ST1B: Store one vector, truncating to 8 bits

These functions read elements from a vector, truncate them to 8 bits, then store them to memory.

#### 6.3.2.1. ST1B (scalar base)

##### Instances

```
void svst1b[_s16](svbool_t pg, int8_t *base, svint16_t data)
void svst1b[_s32](svbool_t pg, int8_t *base, svint32_t data)
void svst1b[_s64](svbool_t pg, int8_t *base, svint64_t data)
void svst1b[_u16](svbool_t pg, uint8_t *base, svuint16_t data)
void svst1b[_u32](svbool_t pg, uint8_t *base, svuint32_t data)
void svst1b[_u64](svbool_t pg, uint8_t *base, svuint64_t data)
```

#### 6.3.2.2. ST1B (scalar base, VL displacement)

##### Instances

```
void svst1b_vnum[_s16](svbool_t pg, int8_t *base, int64_t vnum,
                        svint16_t data)
void svst1b_vnum[_s32](svbool_t pg, int8_t *base, int64_t vnum,
                        svint32_t data)
void svst1b_vnum[_s64](svbool_t pg, int8_t *base, int64_t vnum,
                        svint64_t data)
void svst1b_vnum[_u16](svbool_t pg, uint8_t *base, int64_t vnum,
                        svuint16_t data)
void svst1b_vnum[_u32](svbool_t pg, uint8_t *base, int64_t vnum,
                        svuint32_t data)
void svst1b_vnum[_u64](svbool_t pg, uint8_t *base, int64_t vnum,
                        svuint64_t data)
```

#### 6.3.2.3. ST1B (vector base)

##### Instances

```
void svst1b_scatter[_u32base_s32](svbool_t pg, svuint32_t bases,
                                   svint32_t data)
void svst1b_scatter[_u64base_s64](svbool_t pg, svuint64_t bases,
                                   svint64_t data)
void svst1b_scatter[_u32base_u32](svbool_t pg, svuint32_t bases,
                                   svuint32_t data)
void svst1b_scatter[_u64base_u64](svbool_t pg, svuint64_t bases,
                                   svuint64_t data)
```

#### 6.3.2.4. ST1B (scalar base, vector offset in bytes)

##### Instances

```
void svst1b_scatter[_s32]offset[_s32](svbool_t pg, int8_t *base,
                                         svint32_t offsets, svint32_t data)
void svst1b_scatter[_s64]offset[_s64](svbool_t pg, int8_t *base,
                                         svint64_t offsets, svint64_t data)
void svst1b_scatter[_s32]offset[_u32](svbool_t pg, uint8_t *base,
                                         svint32_t offsets, svuint32_t data)
void svst1b_scatter[_s64]offset[_u64](svbool_t pg, uint8_t *base,
                                         svint64_t offsets, svuint64_t data)
void svst1b_scatter[_u32]offset[_s32](svbool_t pg, int8_t *base,
                                         svuint32_t offsets, svint32_t data)
```

**Instances**

```
void svst1b_scatter[_u64]offset[_s64](svbool_t pg, int8_t *base,
                                       svuint64_t offsets, svint64_t data)
void svst1b_scatter[_u32]offset[_u32](svbool_t pg, uint8_t *base,
                                       svuint32_t offsets, svuint32_t data)
void svst1b_scatter[_u64]offset[_u64](svbool_t pg, uint8_t *base,
                                       svuint64_t offsets, svuint64_t data)
```

**6.3.2.5. ST1B (vector base, scalar offset in bytes)****Instances**

```
void svst1b_scatter[_u32base]offset[_s32](svbool_t pg, svuint32_t bases,
                                           int64_t offset, svint32_t data)
void svst1b_scatter[_u64base]offset[_s64](svbool_t pg, svuint64_t bases,
                                           int64_t offset, svint64_t data)
void svst1b_scatter[_u32base]offset[_u32](svbool_t pg, svuint32_t bases,
                                           int64_t offset, svuint32_t data)
void svst1b_scatter[_u64base]offset[_u64](svbool_t pg, svuint64_t bases,
                                           int64_t offset, svuint64_t data)
```

**6.3.3. ST1H: Store one vector, truncating to 16 bits**

These functions read elements from a vector, truncate them to 16 bits, then store them to memory.

**6.3.3.1. ST1H (scalar base)****Instances**

```
void svst1h[_s32](svbool_t pg, int16_t *base, svint32_t data)
void svst1h[_s64](svbool_t pg, int16_t *base, svint64_t data)
void svst1h[_u32](svbool_t pg, uint16_t *base, svuint32_t data)
void svst1h[_u64](svbool_t pg, uint16_t *base, svuint64_t data)
```

**6.3.3.2. ST1H (scalar base, VL displacement)****Instances**

```
void svst1h_vnum[_s32](svbool_t pg, int16_t *base, int64_t vnum,
                      svint32_t data)
void svst1h_vnum[_s64](svbool_t pg, int16_t *base, int64_t vnum,
                      svint64_t data)
void svst1h_vnum[_u32](svbool_t pg, uint16_t *base, int64_t vnum,
                      svuint32_t data)
void svst1h_vnum[_u64](svbool_t pg, uint16_t *base, int64_t vnum,
                      svuint64_t data)
```

**6.3.3.3. ST1H (vector base)****Instances**

```
void svst1h_scatter[_u32base_s32](svbool_t pg, svuint32_t bases,
                                   svint32_t data)
void svst1h_scatter[_u64base_s64](svbool_t pg, svuint64_t bases,
                                   svint64_t data)
void svst1h_scatter[_u32base_u32](svbool_t pg, svuint32_t bases,
                                   svuint32_t data)
void svst1h_scatter[_u64base_u64](svbool_t pg, svuint64_t bases,
                                   svuint64_t data)
```

### 6.3.3.4. ST1H (scalar base, vector offset in bytes)

Instances
void <b>svst1h_scatter</b> [_s32] <b>offset</b> [_s32](svbool_t <i>pg</i> , int16_t * <i>base</i> , svint32_t <i>offsets</i> , svint32_t <i>data</i> )
void <b>svst1h_scatter</b> [_s64] <b>offset</b> [_s64](svbool_t <i>pg</i> , int16_t * <i>base</i> , svint64_t <i>offsets</i> , svint64_t <i>data</i> )
void <b>svst1h_scatter</b> [_s32] <b>offset</b> [_u32](svbool_t <i>pg</i> , uint16_t * <i>base</i> , svint32_t <i>offsets</i> , svuint32_t <i>data</i> )
void <b>svst1h_scatter</b> [_s64] <b>offset</b> [_u64](svbool_t <i>pg</i> , uint16_t * <i>base</i> , svint64_t <i>offsets</i> , svuint64_t <i>data</i> )
void <b>svst1h_scatter</b> [_u32] <b>offset</b> [_s32](svbool_t <i>pg</i> , int16_t * <i>base</i> , svuint32_t <i>offsets</i> , svint32_t <i>data</i> )
void <b>svst1h_scatter</b> [_u64] <b>offset</b> [_s64](svbool_t <i>pg</i> , int16_t * <i>base</i> , svuint64_t <i>offsets</i> , svint64_t <i>data</i> )
void <b>svst1h_scatter</b> [_u32] <b>offset</b> [_u32](svbool_t <i>pg</i> , uint16_t * <i>base</i> , svuint32_t <i>offsets</i> , svuint32_t <i>data</i> )
void <b>svst1h_scatter</b> [_u64] <b>offset</b> [_u64](svbool_t <i>pg</i> , uint16_t * <i>base</i> , svuint64_t <i>offsets</i> , svuint64_t <i>data</i> )

### 6.3.3.5. ST1H (vector base, scalar offset in bytes)

Instances
void <b>svst1h_scatter</b> [_u32base] <b>offset</b> [_s32](svbool_t <i>pg</i> , svuint32_t <i>bases</i> , int64_t <i>offset</i> , svint32_t <i>data</i> )
void <b>svst1h_scatter</b> [_u64base] <b>offset</b> [_s64](svbool_t <i>pg</i> , svuint64_t <i>bases</i> , int64_t <i>offset</i> , svint64_t <i>data</i> )
void <b>svst1h_scatter</b> [_u32base] <b>offset</b> [_u32](svbool_t <i>pg</i> , svuint32_t <i>bases</i> , int64_t <i>offset</i> , svuint32_t <i>data</i> )
void <b>svst1h_scatter</b> [_u64base] <b>offset</b> [_u64](svbool_t <i>pg</i> , svuint64_t <i>bases</i> , int64_t <i>offset</i> , svuint64_t <i>data</i> )

### 6.3.3.6. ST1H (scalar base, vector index)

Instances
void <b>svst1h_scatter</b> [_s32] <b>index</b> [_s32](svbool_t <i>pg</i> , int16_t * <i>base</i> , svint32_t <i>indices</i> , svint32_t <i>data</i> )
void <b>svst1h_scatter</b> [_s64] <b>index</b> [_s64](svbool_t <i>pg</i> , int16_t * <i>base</i> , svint64_t <i>indices</i> , svint64_t <i>data</i> )
void <b>svst1h_scatter</b> [_s32] <b>index</b> [_u32](svbool_t <i>pg</i> , uint16_t * <i>base</i> , svint32_t <i>indices</i> , svuint32_t <i>data</i> )
void <b>svst1h_scatter</b> [_s64] <b>index</b> [_u64](svbool_t <i>pg</i> , uint16_t * <i>base</i> , svint64_t <i>indices</i> , svuint64_t <i>data</i> )
void <b>svst1h_scatter</b> [_u32] <b>index</b> [_s32](svbool_t <i>pg</i> , int16_t * <i>base</i> , svuint32_t <i>indices</i> , svint32_t <i>data</i> )
void <b>svst1h_scatter</b> [_u64] <b>index</b> [_s64](svbool_t <i>pg</i> , int16_t * <i>base</i> , svuint64_t <i>indices</i> , svint64_t <i>data</i> )
void <b>svst1h_scatter</b> [_u32] <b>index</b> [_u32](svbool_t <i>pg</i> , uint16_t * <i>base</i> , svuint32_t <i>indices</i> , svuint32_t <i>data</i> )
void <b>svst1h_scatter</b> [_u64] <b>index</b> [_u64](svbool_t <i>pg</i> , uint16_t * <i>base</i> , svuint64_t <i>indices</i> , svuint64_t <i>data</i> )

### 6.3.3.7. ST1H (vector base, scalar index)

Instances
void <b>svst1h_scatter</b> [_u32base] <b>index</b> [_s32](svbool_t <i>pg</i> , svuint32_t <i>bases</i> ,

Instances
<pre>                                 int64_t index, svint32_t data) void svst1h_scatter[_u64base]_index[_s64](svbool_t pg, svuint64_t bases,                                 int64_t index, svint64_t data) void svst1h_scatter[_u32base]_index[_u32](svbool_t pg, svuint32_t bases,                                 int64_t index, svuint32_t data) void svst1h_scatter[_u64base]_index[_u64](svbool_t pg, svuint64_t bases,                                 int64_t index, svuint64_t data) </pre>

### 6.3.4. ST1W: Store one vector, truncating to 32 bits

These functions read elements from a vector, truncate them to 32 bits, then store them to memory.

#### 6.3.4.1. ST1W (scalar base)

Instances
<pre> void svst1w[_s64](svbool_t pg, int32_t *base, svint64_t data) void svst1w[_u64](svbool_t pg, uint32_t *base, svuint64_t data) </pre>

#### 6.3.4.2. ST1W (scalar base, VL displacement)

Instances
<pre> void svst1w_vnum[_s64](svbool_t pg, int32_t *base, int64_t vnum,                         svint64_t data) void svst1w_vnum[_u64](svbool_t pg, uint32_t *base, int64_t vnum,                         svuint64_t data) </pre>

#### 6.3.4.3. ST1W (vector base)

Instances
<pre> void svst1w_scatter[_u64base_s64](svbool_t pg, svuint64_t bases,                                 svint64_t data) void svst1w_scatter[_u64base_u64](svbool_t pg, svuint64_t bases,                                 svuint64_t data) </pre>

#### 6.3.4.4. ST1W (scalar base, vector offset in bytes)

Instances
<pre> void svst1w_scatter[_s64]_offset[_s64](svbool_t pg, int32_t *base,                                 svint64_t offsets, svint64_t data) void svst1w_scatter[_s64]_offset[_u64](svbool_t pg, uint32_t *base,                                 svint64_t offsets, svuint64_t data) void svst1w_scatter[_u64]_offset[_s64](svbool_t pg, int32_t *base,                                 svuint64_t offsets, svint64_t data) void svst1w_scatter[_u64]_offset[_u64](svbool_t pg, uint32_t *base,                                 svuint64_t offsets, svuint64_t data) </pre>

#### 6.3.4.5. ST1W (vector base, scalar offset in bytes)

Instances
<pre> void svst1w_scatter[_u64base]_offset[_s64](svbool_t pg, svuint64_t bases,                                 int64_t offset, svint64_t data) </pre>

**Instances**

```
void svstlw_scatter[_u64base]_offset[_u64](svbool_t pg, svuint64_t bases,
                                             int64_t offset, svuint64_t data)
```

**6.3.4.6. ST1W (scalar base, vector index)****Instances**

```
void svstlw_scatter[_s64]_index[_s64](svbool_t pg, int32_t *base,
                                         svint64_t indices, svint64_t data)
void svstlw_scatter[_s64]_index[_u64](svbool_t pg, uint32_t *base,
                                         svint64_t indices, svuint64_t data)
void svstlw_scatter[_u64]_index[_s64](svbool_t pg, int32_t *base,
                                         svuint64_t indices, svint64_t data)
void svstlw_scatter[_u64]_index[_u64](svbool_t pg, uint32_t *base,
                                         svuint64_t indices, svuint64_t data)
```

**6.3.4.7. ST1W (vector base, scalar index)****Instances**

```
void svstlw_scatter[_u64base]_index[_s64](svbool_t pg, svuint64_t bases,
                                             int64_t index, svint64_t data)
void svstlw_scatter[_u64base]_index[_u64](svbool_t pg, svuint64_t bases,
                                             int64_t index, svuint64_t data)
```

**6.3.5. STNT1: Store one vector, with no truncation, non-temporal**

These functions behave like the stores in [Section 6.3.1, “ST1: Store one vector, with no truncation”](#), but additionally provide a hint to the system that the stored memory is unlikely to be accessed again soon.

**6.3.5.1. STNT1 (scalar base)****Instances**

```
void svstnt1[_s8](svbool_t pg, int8_t *base, svint8_t data)
void svstnt1[_s16](svbool_t pg, int16_t *base, svint16_t data)
void svstnt1[_s32](svbool_t pg, int32_t *base, svint32_t data)
void svstnt1[_s64](svbool_t pg, int64_t *base, svint64_t data)
void svstnt1[_u8](svbool_t pg, uint8_t *base, svuint8_t data)
void svstnt1[_u16](svbool_t pg, uint16_t *base, svuint16_t data)
void svstnt1[_u32](svbool_t pg, uint32_t *base, svuint32_t data)
void svstnt1[_u64](svbool_t pg, uint64_t *base, svuint64_t data)
void svstnt1[_f16](svbool_t pg, float16_t *base, svfloat16_t data)
void svstnt1[_f32](svbool_t pg, float32_t *base, svfloat32_t data)
void svstnt1[_f64](svbool_t pg, float64_t *base, svfloat64_t data)
void svstnt1[_bf16](svbool_t pg, bfloat16_t *base, svbfloat16_t data)
```

**6.3.5.2. STNT1 (scalar base, VL displacement)****Instances**

```
void svstnt1_vnum[_s8](svbool_t pg, int8_t *base, int64_t vnum,
                        svint8_t data)
void svstnt1_vnum[_s16](svbool_t pg, int16_t *base, int64_t vnum,
                        svint16_t data)
void svstnt1_vnum[_s32](svbool_t pg, int32_t *base, int64_t vnum,
                        svint32_t data)
```

**Instances**

```

void svstnt1_vnum[_s64](svbool_t pg, int64_t *base, int64_t vnum,
                        svint64_t data)
void svstnt1_vnum[_u8](svbool_t pg, uint8_t *base, int64_t vnum,
                      svuint8_t data)
void svstnt1_vnum[_u16](svbool_t pg, uint16_t *base, int64_t vnum,
                       svuint16_t data)
void svstnt1_vnum[_u32](svbool_t pg, uint32_t *base, int64_t vnum,
                       svuint32_t data)
void svstnt1_vnum[_u64](svbool_t pg, uint64_t *base, int64_t vnum,
                       svuint64_t data)
void svstnt1_vnum[_f16](svbool_t pg, float16_t *base, int64_t vnum,
                       svfloat16_t data)
void svstnt1_vnum[_f32](svbool_t pg, float32_t *base, int64_t vnum,
                       svfloat32_t data)
void svstnt1_vnum[_f64](svbool_t pg, float64_t *base, int64_t vnum,
                       svfloat64_t data)
void svstnt1_vnum[_bf16](svbool_t pg, bfloat16_t *base, int64_t vnum,
                        svbfloat16_t data)

```

### 6.3.6. ST2: Store two vectors into two-element structures

These functions store a pair of vectors to an array of two-element structures, with one vector per structure element. In other words, element *N* of vector *M* corresponds to element *M* of structure *N*.

#### 6.3.6.1. ST2 (scalar base)

**Instances**

```

void svst2[_s8](svbool_t pg, int8_t *base, svint8x2_t data)
void svst2[_s16](svbool_t pg, int16_t *base, svint16x2_t data)
void svst2[_s32](svbool_t pg, int32_t *base, svint32x2_t data)
void svst2[_s64](svbool_t pg, int64_t *base, svint64x2_t data)
void svst2[_u8](svbool_t pg, uint8_t *base, svuint8x2_t data)
void svst2[_u16](svbool_t pg, uint16_t *base, svuint16x2_t data)
void svst2[_u32](svbool_t pg, uint32_t *base, svuint32x2_t data)
void svst2[_u64](svbool_t pg, uint64_t *base, svuint64x2_t data)
void svst2[_f16](svbool_t pg, float16_t *base, svfloat16x2_t data)
void svst2[_f32](svbool_t pg, float32_t *base, svfloat32x2_t data)
void svst2[_f64](svbool_t pg, float64_t *base, svfloat64x2_t data)
void svst2[_bf16](svbool_t pg, bfloat16_t *base, svbfloat16x2_t data)

```

#### 6.3.6.2. ST2 (scalar base, VL displacement)

**Instances**

```

void svst2_vnum[_s8](svbool_t pg, int8_t *base, int64_t vnum,
                     svint8x2_t data)
void svst2_vnum[_s16](svbool_t pg, int16_t *base, int64_t vnum,
                      svint16x2_t data)
void svst2_vnum[_s32](svbool_t pg, int32_t *base, int64_t vnum,
                      svint32x2_t data)
void svst2_vnum[_s64](svbool_t pg, int64_t *base, int64_t vnum,
                      svint64x2_t data)
void svst2_vnum[_u8](svbool_t pg, uint8_t *base, int64_t vnum,
                     svuint8x2_t data)
void svst2_vnum[_u16](svbool_t pg, uint16_t *base, int64_t vnum,
                     svuint16x2_t data)

```

**Instances**

```

void svst2_vnum[_u32](svbool_t pg, uint32_t *base, int64_t vnum,
                     svuint32x2_t data)
void svst2_vnum[_u64](svbool_t pg, uint64_t *base, int64_t vnum,
                     svuint64x2_t data)
void svst2_vnum[_f16](svbool_t pg, float16_t *base, int64_t vnum,
                     svfloat16x2_t data)
void svst2_vnum[_f32](svbool_t pg, float32_t *base, int64_t vnum,
                     svfloat32x2_t data)
void svst2_vnum[_f64](svbool_t pg, float64_t *base, int64_t vnum,
                     svfloat64x2_t data)
void svst2_vnum[_bf16](svbool_t pg, bfloat16_t *base, int64_t vnum,
                     svbfloat16x2_t data)

```

**6.3.7. ST3: Store three vectors into three-element structures**

These functions store three vectors to an array of three-element structures, with one vector per structure element. In other words, element *N* of vector *M* corresponds to element *M* of structure *N*.

**6.3.7.1. ST3 (scalar base)****Instances**

```

void svst3[_s8](svbool_t pg, int8_t *base, svint8x3_t data)
void svst3[_s16](svbool_t pg, int16_t *base, svint16x3_t data)
void svst3[_s32](svbool_t pg, int32_t *base, svint32x3_t data)
void svst3[_s64](svbool_t pg, int64_t *base, svint64x3_t data)
void svst3[_u8](svbool_t pg, uint8_t *base, svuint8x3_t data)
void svst3[_u16](svbool_t pg, uint16_t *base, svuint16x3_t data)
void svst3[_u32](svbool_t pg, uint32_t *base, svuint32x3_t data)
void svst3[_u64](svbool_t pg, uint64_t *base, svuint64x3_t data)
void svst3[_f16](svbool_t pg, float16_t *base, svfloat16x3_t data)
void svst3[_f32](svbool_t pg, float32_t *base, svfloat32x3_t data)
void svst3[_f64](svbool_t pg, float64_t *base, svfloat64x3_t data)
void svst3[_bf16](svbool_t pg, bfloat16_t *base, svbfloat16x3_t data)

```

**6.3.7.2. ST3 (scalar base, VL displacement)****Instances**

```

void svst3_vnum[_s8](svbool_t pg, int8_t *base, int64_t vnum,
                     svint8x3_t data)
void svst3_vnum[_s16](svbool_t pg, int16_t *base, int64_t vnum,
                     svint16x3_t data)
void svst3_vnum[_s32](svbool_t pg, int32_t *base, int64_t vnum,
                     svint32x3_t data)
void svst3_vnum[_s64](svbool_t pg, int64_t *base, int64_t vnum,
                     svint64x3_t data)
void svst3_vnum[_u8](svbool_t pg, uint8_t *base, int64_t vnum,
                     svuint8x3_t data)
void svst3_vnum[_u16](svbool_t pg, uint16_t *base, int64_t vnum,
                     svuint16x3_t data)
void svst3_vnum[_u32](svbool_t pg, uint32_t *base, int64_t vnum,
                     svuint32x3_t data)
void svst3_vnum[_u64](svbool_t pg, uint64_t *base, int64_t vnum,
                     svuint64x3_t data)
void svst3_vnum[_f16](svbool_t pg, float16_t *base, int64_t vnum,
                     svfloat16x3_t data)

```

**Instances**

```
void svst3_vnum[_f32](svbool_t pg, float32_t *base, int64_t vnum,
                     svfloat32x3_t data)
void svst3_vnum[_f64](svbool_t pg, float64_t *base, int64_t vnum,
                     svfloat64x3_t data)
void svst3_vnum[_bf16](svbool_t pg, bfloat16_t *base, int64_t vnum,
                      svbfloat16x3_t data)
```

### 6.3.8. ST4: Store four vectors into four-element structures

These functions store four vectors to an array of four-element structures, with one vector per structure element. In other words, element *N* of vector *M* corresponds to element *M* of structure *N*.

#### 6.3.8.1. ST4 (scalar base)

**Instances**

```
void svst4[_s8](svbool_t pg, int8_t *base, svint8x4_t data)
void svst4[_s16](svbool_t pg, int16_t *base, svint16x4_t data)
void svst4[_s32](svbool_t pg, int32_t *base, svint32x4_t data)
void svst4[_s64](svbool_t pg, int64_t *base, svint64x4_t data)
void svst4[_u8](svbool_t pg, uint8_t *base, svuint8x4_t data)
void svst4[_u16](svbool_t pg, uint16_t *base, svuint16x4_t data)
void svst4[_u32](svbool_t pg, uint32_t *base, svuint32x4_t data)
void svst4[_u64](svbool_t pg, uint64_t *base, svuint64x4_t data)
void svst4[_f16](svbool_t pg, float16_t *base, svfloat16x4_t data)
void svst4[_f32](svbool_t pg, float32_t *base, svfloat32x4_t data)
void svst4[_f64](svbool_t pg, float64_t *base, svfloat64x4_t data)
void svst4[_bf16](svbool_t pg, bfloat16_t *base, svbfloat16x4_t data)
```

#### 6.3.8.2. ST4 (scalar base, VL displacement)

**Instances**

```
void svst4_vnum[_s8](svbool_t pg, int8_t *base, int64_t vnum,
                     svint8x4_t data)
void svst4_vnum[_s16](svbool_t pg, int16_t *base, int64_t vnum,
                      svint16x4_t data)
void svst4_vnum[_s32](svbool_t pg, int32_t *base, int64_t vnum,
                      svint32x4_t data)
void svst4_vnum[_s64](svbool_t pg, int64_t *base, int64_t vnum,
                      svint64x4_t data)
void svst4_vnum[_u8](svbool_t pg, uint8_t *base, int64_t vnum,
                     svuint8x4_t data)
void svst4_vnum[_u16](svbool_t pg, uint16_t *base, int64_t vnum,
                     svuint16x4_t data)
void svst4_vnum[_u32](svbool_t pg, uint32_t *base, int64_t vnum,
                     svuint32x4_t data)
void svst4_vnum[_u64](svbool_t pg, uint64_t *base, int64_t vnum,
                     svuint64x4_t data)
void svst4_vnum[_f16](svbool_t pg, float16_t *base, int64_t vnum,
                      svfloat16x4_t data)
void svst4_vnum[_f32](svbool_t pg, float32_t *base, int64_t vnum,
                      svfloat32x4_t data)
void svst4_vnum[_f64](svbool_t pg, float64_t *base, int64_t vnum,
                      svfloat64x4_t data)
void svst4_vnum[_bf16](svbool_t pg, bfloat16_t *base, int64_t vnum,
                       svbfloat16x4_t data)
```



## 6.4. Prefetches

### 6.4.1. PRFB: Prefetch 8-bit data

#### 6.4.1.1. PRFB (scalar base)

##### Instances

```
void svprfb(svbool_t pg, const void *base, enum svprfop op)
```

#### 6.4.1.2. PRFB (scalar base, VL displacement)

##### Instances

```
void svprfb_vnum(svbool_t pg, const void *base, int64_t vnum,  
enum svprfop op)
```

#### 6.4.1.3. PRFB (vector base)

##### Instances

```
void svprfb_gather[_u32base](svbool_t pg, svuint32_t bases,  
enum svprfop op)  
void svprfb_gather[_u64base](svbool_t pg, svuint64_t bases,  
enum svprfop op)
```

#### 6.4.1.4. PRFB (scalar base, vector offset in bytes)

##### Instances

```
void svprfb_gather[_s32]offset(svbool_t pg, const void *base,  
svint32_t offsets, enum svprfop op)  
void svprfb_gather[_s64]offset(svbool_t pg, const void *base,  
svint64_t offsets, enum svprfop op)  
void svprfb_gather[_u32]offset(svbool_t pg, const void *base,  
svuint32_t offsets, enum svprfop op)  
void svprfb_gather[_u64]offset(svbool_t pg, const void *base,  
svuint64_t offsets, enum svprfop op)
```

#### 6.4.1.5. PRFB (vector base, scalar offset in bytes)

##### Instances

```
void svprfb_gather[_u32base]_offset(svbool_t pg, svuint32_t bases,  
int64_t offset, enum svprfop op)  
void svprfb_gather[_u64base]_offset(svbool_t pg, svuint64_t bases,  
int64_t offset, enum svprfop op)
```

### 6.4.2. PRFH: Prefetch 16-bit data

#### 6.4.2.1. PRFH (scalar base)

##### Instances

```
void svprfh(svbool_t pg, const void *base, enum svprfop op)
```

### 6.4.2.2. PRFH (scalar base, VL displacement)

Instances
void <b>svprfh_vnum</b> (svbool_t <i>pg</i> , const void * <i>base</i> , int64_t <i>vnum</i> , enum svprfop <i>op</i> )

### 6.4.2.3. PRFH (vector base)

Instances
void <b>svprfh_gather</b> [_u32base](svbool_t <i>pg</i> , svuint32_t <i>bases</i> , enum svprfop <i>op</i> )
void <b>svprfh_gather</b> [_u64base](svbool_t <i>pg</i> , svuint64_t <i>bases</i> , enum svprfop <i>op</i> )

### 6.4.2.4. PRFH (scalar base, vector index)

Instances
void <b>svprfh_gather</b> [_s32]index(svbool_t <i>pg</i> , const void * <i>base</i> , svint32_t <i>indices</i> , enum svprfop <i>op</i> )
void <b>svprfh_gather</b> [_s64]index(svbool_t <i>pg</i> , const void * <i>base</i> , svint64_t <i>indices</i> , enum svprfop <i>op</i> )
void <b>svprfh_gather</b> [_u32]index(svbool_t <i>pg</i> , const void * <i>base</i> , svuint32_t <i>indices</i> , enum svprfop <i>op</i> )
void <b>svprfh_gather</b> [_u64]index(svbool_t <i>pg</i> , const void * <i>base</i> , svuint64_t <i>indices</i> , enum svprfop <i>op</i> )

### 6.4.2.5. PRFH (vector base, scalar index)

Instances
void <b>svprfh_gather</b> [_u32base]_index(svbool_t <i>pg</i> , svuint32_t <i>bases</i> , int64_t <i>index</i> , enum svprfop <i>op</i> )
void <b>svprfh_gather</b> [_u64base]_index(svbool_t <i>pg</i> , svuint64_t <i>bases</i> , int64_t <i>index</i> , enum svprfop <i>op</i> )

## 6.4.3. PRFW: Prefetch 32-bit data

### 6.4.3.1. PRFW (scalar base)

Instances
void <b>svprfw</b> (svbool_t <i>pg</i> , const void * <i>base</i> , enum svprfop <i>op</i> )

### 6.4.3.2. PRFW (scalar base, VL displacement)

Instances
void <b>svprfw_vnum</b> (svbool_t <i>pg</i> , const void * <i>base</i> , int64_t <i>vnum</i> , enum svprfop <i>op</i> )

### 6.4.3.3. PRFW (vector base)

Instances
void <b>svprfw_gather</b> [_u32base](svbool_t <i>pg</i> , svuint32_t <i>bases</i> , enum svprfop <i>op</i> )

**Instances**

```
void svprfw_gather[_u64base](svbool_t pg, svuint64_t bases,
                             enum svprfop op)
```

**6.4.3.4. PRFW (scalar base, vector index)****Instances**

```
void svprfw_gather[_s32]index(svbool_t pg, const void *base,
                              svint32_t indices, enum svprfop op)
void svprfw_gather[_s64]index(svbool_t pg, const void *base,
                              svint64_t indices, enum svprfop op)
void svprfw_gather[_u32]index(svbool_t pg, const void *base,
                              svuint32_t indices, enum svprfop op)
void svprfw_gather[_u64]index(svbool_t pg, const void *base,
                              svuint64_t indices, enum svprfop op)
```

**6.4.3.5. PRFW (vector base, scalar index)****Instances**

```
void svprfw_gather[_u32base]_index(svbool_t pg, svuint32_t bases,
                                   int64_t index, enum svprfop op)
void svprfw_gather[_u64base]_index(svbool_t pg, svuint64_t bases,
                                   int64_t index, enum svprfop op)
```

**6.4.4. PRFD: Prefetch 64-bit data****6.4.4.1. PRFD (scalar base)****Instances**

```
void svprfd(svbool_t pg, const void *base, enum svprfop op)
```

**6.4.4.2. PRFD (scalar base, VL displacement)****Instances**

```
void svprfd_vnum(svbool_t pg, const void *base, int64_t vnum,
                  enum svprfop op)
```

**6.4.4.3. PRFD (vector base)****Instances**

```
void svprfd_gather[_u32base](svbool_t pg, svuint32_t bases,
                              enum svprfop op)
void svprfd_gather[_u64base](svbool_t pg, svuint64_t bases,
                              enum svprfop op)
```

**6.4.4.4. PRFD (scalar base, vector index)****Instances**

```
void svprfd_gather[_s32]index(svbool_t pg, const void *base,
                              svint32_t indices, enum svprfop op)
void svprfd_gather[_s64]index(svbool_t pg, const void *base,
                              svint64_t indices, enum svprfop op)
void svprfd_gather[_u32]index(svbool_t pg, const void *base,
```

**Instances**

```

                                svuint32_t indices, enum svprfop op)
void svprfd_gather[_u64]index(svbool_t pg, const void *base,
                                svuint64_t indices, enum svprfop op)

```

**6.4.4.5. PRFD (vector base, scalar index)****Instances**

```

void svprfd_gather[_u32base]_index(svbool_t pg, svuint32_t bases,
                                   int64_t index, enum svprfop op)
void svprfd_gather[_u64base]_index(svbool_t pg, svuint64_t bases,
                                   int64_t index, enum svprfop op)

```

**6.5. Address calculations****6.5.1. ADRB: Compute vector address for 8-bit data**

These functions add a vector of offsets to a vector of bases, without applying a scaling factor to the offsets.

**6.5.1.1. ADRB (vector base, vector offset in bytes)****Instances**

```

svuint32_t svadrb[_u32base][_s32]offset(svuint32_t bases,
                                         svint32_t offsets)
svuint64_t svadrb[_u64base][_s64]offset(svuint64_t bases,
                                         svint64_t offsets)
svuint32_t svadrb[_u32base][_u32]offset(svuint32_t bases,
                                         svuint32_t offsets)
svuint64_t svadrb[_u64base][_u64]offset(svuint64_t bases,
                                         svuint64_t offsets)

```

**6.5.2. ADRH: Compute vector address for 16-bit data**

These functions multiply a vector of indices by 2 (bytes) and add them to a vector of bases.

**6.5.2.1. ADRH (vector base, vector index)****Instances**

```

svuint32_t svadrh[_u32base][_s32]index(svuint32_t bases, svint32_t indices)
svuint64_t svadrh[_u64base][_s64]index(svuint64_t bases, svint64_t indices)
svuint32_t svadrh[_u32base][_u32]index(svuint32_t bases,
                                         svuint32_t indices)
svuint64_t svadrh[_u64base][_u64]index(svuint64_t bases,
                                         svuint64_t indices)

```

**6.5.3. ADRW: Compute vector address for 32-bit data**

These functions multiply a vector of indices by 4 (bytes) and add them to a vector of bases.

**6.5.3.1. ADRW (vector base, vector index)****Instances**

```

svuint32_t svadrw[_u32base][_s32]index(svuint32_t bases, svint32_t indices)

```

Instances	
svuint64_t	<b>svadrw</b> [_u64base][_s64] <b>index</b> (svuint64_t <i>bases</i> , svint64_t <i>indices</i> )
svuint32_t	<b>svadrw</b> [_u32base][_u32] <b>index</b> (svuint32_t <i>bases</i> , svuint32_t <i>indices</i> )
svuint64_t	<b>svadrw</b> [_u64base][_u64] <b>index</b> (svuint64_t <i>bases</i> , svuint64_t <i>indices</i> )

## 6.5.4. ADRD: Compute vector address for 64-bit data

These functions multiply a vector of indices by 8 (bytes) and add them to a vector of bases.

### 6.5.4.1. ADRD (vector base, vector index)

Instances	
svuint32_t	<b>svadrd</b> [_u32base][_s32] <b>index</b> (svuint32_t <i>bases</i> , svint32_t <i>indices</i> )
svuint64_t	<b>svadrd</b> [_u64base][_s64] <b>index</b> (svuint64_t <i>bases</i> , svint64_t <i>indices</i> )
svuint32_t	<b>svadrd</b> [_u32base][_u32] <b>index</b> (svuint32_t <i>bases</i> , svuint32_t <i>indices</i> )
svuint64_t	<b>svadrd</b> [_u64base][_u64] <b>index</b> (svuint64_t <i>bases</i> , svuint64_t <i>indices</i> )

## 6.6. Scalar to vector operations

### 6.6.1. DUP: Duplicate scalar value

These functions copy a scalar value into selected elements of a vector.

Although the functions are named after the DUP instruction, the implementation can use any instruction sequence that achieves the same effect (just as it can with other ACLE functions). For example, the predicated forms map more exactly to CPY than to DUP. If the scalar value is a constant, it may be more efficient to use DUPM or FCPY. If the scalar value is in memory, it may be more efficient to use LD1Rx.

#### 6.6.1.1. DUP (scalar)

Instances	
svint8_t	<b>svdup</b> [_n]_s8(int8_t <i>op</i> )
svint16_t	<b>svdup</b> [_n]_s16(int16_t <i>op</i> )
svint32_t	<b>svdup</b> [_n]_s32(int32_t <i>op</i> )
svint64_t	<b>svdup</b> [_n]_s64(int64_t <i>op</i> )
svuint8_t	<b>svdup</b> [_n]_u8(uint8_t <i>op</i> )
svuint16_t	<b>svdup</b> [_n]_u16(uint16_t <i>op</i> )
svuint32_t	<b>svdup</b> [_n]_u32(uint32_t <i>op</i> )
svuint64_t	<b>svdup</b> [_n]_u64(uint64_t <i>op</i> )
svfloat16_t	<b>svdup</b> [_n]_f16(float16_t <i>op</i> )
svfloat32_t	<b>svdup</b> [_n]_f32(float32_t <i>op</i> )
svfloat64_t	<b>svdup</b> [_n]_f64(float64_t <i>op</i> )
svbfloat16_t	<b>svdup</b> [_n]_bf16(bfloat16_t <i>op</i> )

#### 6.6.1.2. DUP (scalar), setting inactive to zero

Instances	
svint8_t	<b>svdup</b> [_n]_s8_z(svbool_t <i>pg</i> , int8_t <i>op</i> )
svint16_t	<b>svdup</b> [_n]_s16_z(svbool_t <i>pg</i> , int16_t <i>op</i> )

**Instances**

```

svint32_t svdup[_n]_s32_z(svbool_t pg, int32_t op)
svint64_t svdup[_n]_s64_z(svbool_t pg, int64_t op)
svuint8_t svdup[_n]_u8_z(svbool_t pg, uint8_t op)
svuint16_t svdup[_n]_u16_z(svbool_t pg, uint16_t op)
svuint32_t svdup[_n]_u32_z(svbool_t pg, uint32_t op)
svuint64_t svdup[_n]_u64_z(svbool_t pg, uint64_t op)
svfloat16_t svdup[_n]_f16_z(svbool_t pg, float16_t op)
svfloat32_t svdup[_n]_f32_z(svbool_t pg, float32_t op)
svfloat64_t svdup[_n]_f64_z(svbool_t pg, float64_t op)
svbfloat16_t svdup[_n]_bf16_z(svbool_t pg, bfloat16_t op)

```

**6.6.1.3. DUP (scalar), merging with separate vector****Instances**

```

svint8_t svdup[_n]_s8_m(svint8_t inactive, svbool_t pg, int8_t op)
svint16_t svdup[_n]_s16_m(svint16_t inactive, svbool_t pg, int16_t op)
svint32_t svdup[_n]_s32_m(svint32_t inactive, svbool_t pg, int32_t op)
svint64_t svdup[_n]_s64_m(svint64_t inactive, svbool_t pg, int64_t op)
svuint8_t svdup[_n]_u8_m(svuint8_t inactive, svbool_t pg, uint8_t op)
svuint16_t svdup[_n]_u16_m(svuint16_t inactive, svbool_t pg, uint16_t op)
svuint32_t svdup[_n]_u32_m(svuint32_t inactive, svbool_t pg, uint32_t op)
svuint64_t svdup[_n]_u64_m(svuint64_t inactive, svbool_t pg, uint64_t op)
svfloat16_t svdup[_n]_f16_m(svfloat16_t inactive, svbool_t pg,
                           float16_t op)
svfloat32_t svdup[_n]_f32_m(svfloat32_t inactive, svbool_t pg,
                           float32_t op)
svfloat64_t svdup[_n]_f64_m(svfloat64_t inactive, svbool_t pg,
                           float64_t op)
svbfloat16_t svdup[_n]_bf16_m(svbfloat16_t inactive, svbool_t pg,
                             bfloat16_t op)

```

**6.6.1.4. DUP (scalar), setting inactive to unknown****Instances**

```

svint8_t svdup[_n]_s8_x(svbool_t pg, int8_t op)
svint16_t svdup[_n]_s16_x(svbool_t pg, int16_t op)
svint32_t svdup[_n]_s32_x(svbool_t pg, int32_t op)
svint64_t svdup[_n]_s64_x(svbool_t pg, int64_t op)
svuint8_t svdup[_n]_u8_x(svbool_t pg, uint8_t op)
svuint16_t svdup[_n]_u16_x(svbool_t pg, uint16_t op)
svuint32_t svdup[_n]_u32_x(svbool_t pg, uint32_t op)
svuint64_t svdup[_n]_u64_x(svbool_t pg, uint64_t op)
svfloat16_t svdup[_n]_f16_x(svbool_t pg, float16_t op)
svfloat32_t svdup[_n]_f32_x(svbool_t pg, float32_t op)
svfloat64_t svdup[_n]_f64_x(svbool_t pg, float64_t op)
svbfloat16_t svdup[_n]_bf16_x(svbool_t pg, bfloat16_t op)

```

**6.6.2. DUPQ: Duplicate scalars to every quadword of a vector**

These functions take enough scalar values to fill one 128-bit quadword of a vector, in element index order. They replicate this sequence to fill an entire vector and return the result.

If the implementation chooses to assemble the 128-bit sequence in the first elements of a vector, the final step of duplicating it to every quadword is compatible with the .Q form of the DUP instruction. However, as with other ACLE functions, the implementation can use any sequence that achieves the same effect.

For example, if the scalar values are constants, it may be more efficient to put them in a constant pool and load them using LD1RQ.

### 6.6.2.1. DUPQ (16 scalars)

Instances
<pre>svint8_t  svdupq[_n]_s8(int8_t x0, int8_t x1, int8_t x2, int8_t x3,                         int8_t x4, int8_t x5, int8_t x6, int8_t x7,                         int8_t x8, int8_t x9, int8_t x10, int8_t x11,                         int8_t x12, int8_t x13, int8_t x14, int8_t x15) svuint8_t svdupq[_n]_u8(uint8_t x0, uint8_t x1, uint8_t x2, uint8_t x3,                         uint8_t x4, uint8_t x5, uint8_t x6, uint8_t x7,                         uint8_t x8, uint8_t x9, uint8_t x10, uint8_t x11,                         uint8_t x12, uint8_t x13, uint8_t x14, uint8_t x15)</pre>

### 6.6.2.2. DUPQ (8 scalars)

Instances
<pre>svint16_t svdupq[_n]_s16(int16_t x0, int16_t x1, int16_t x2, int16_t x3,                         int16_t x4, int16_t x5, int16_t x6, int16_t x7) svuint16_t svdupq[_n]_u16(uint16_t x0, uint16_t x1, uint16_t x2,                         uint16_t x3, uint16_t x4, uint16_t x5,                         uint16_t x6, uint16_t x7) svfloat16_t svdupq[_n]_f16(float16_t x0, float16_t x1, float16_t x2,                         float16_t x3, float16_t x4, float16_t x5,                         float16_t x6, float16_t x7) svbfloat16_t svdupq[_n]_bf16(bfloat16_t x0, bfloat16_t x1, bfloat16_t x2,                         bfloat16_t x3, bfloat16_t x4, bfloat16_t x5,                         bfloat16_t x6, bfloat16_t x7)</pre>

### 6.6.2.3. DUPQ (4 scalars)

Instances
<pre>svint32_t svdupq[_n]_s32(int32_t x0, int32_t x1, int32_t x2, int32_t x3) svuint32_t svdupq[_n]_u32(uint32_t x0, uint32_t x1, uint32_t x2,                         uint32_t x3) svfloat32_t svdupq[_n]_f32(float32_t x0, float32_t x1, float32_t x2,                         float32_t x3)</pre>

### 6.6.2.4. DUPQ (2 scalars)

Instances
<pre>svint64_t svdupq[_n]_s64(int64_t x0, int64_t x1) svuint64_t svdupq[_n]_u64(uint64_t x0, uint64_t x1) svfloat64_t svdupq[_n]_f64(float64_t x0, float64_t x1)</pre>

## 6.6.3. INDEX: Create index series

These functions create a vector series of the form:

*{base, base+step, base+step\*2, base+step\*3, ...}*

The multiplications and additions use modular arithmetic and the result is well-defined for all inputs. There is no undefined behavior for signed overflow.

### 6.6.3.1. INDEX (scalar, scalar)

Instances
<code>svint8_t svindex_s8(int8_t base, int8_t step)</code>
<code>svint16_t svindex_s16(int16_t base, int16_t step)</code>
<code>svint32_t svindex_s32(int32_t base, int32_t step)</code>
<code>svint64_t svindex_s64(int64_t base, int64_t step)</code>
<code>svuint8_t svindex_u8(uint8_t base, uint8_t step)</code>
<code>svuint16_t svindex_u16(uint16_t base, uint16_t step)</code>
<code>svuint32_t svindex_u32(uint32_t base, uint32_t step)</code>
<code>svuint64_t svindex_u64(uint64_t base, uint64_t step)</code>

## 6.7. Integer arithmetic

### 6.7.1. ADD: Modular integer addition

These functions perform modular addition on two integer inputs; that is, if the input elements have  $N$  bits, the result is the low  $N$  bits of the sum.

The result is well-defined for all inputs; unlike C, there is no undefined behavior for signed overflow.

#### 6.7.1.1. ADD (vector, vector), setting inactive to zero

Instances
<code>svint8_t svadd[_s8]_z(svbool_t pg, svint8_t op1, svint8_t op2)</code>
<code>svint16_t svadd[_s16]_z(svbool_t pg, svint16_t op1, svint16_t op2)</code>
<code>svint32_t svadd[_s32]_z(svbool_t pg, svint32_t op1, svint32_t op2)</code>
<code>svint64_t svadd[_s64]_z(svbool_t pg, svint64_t op1, svint64_t op2)</code>
<code>svuint8_t svadd[_u8]_z(svbool_t pg, svuint8_t op1, svuint8_t op2)</code>
<code>svuint16_t svadd[_u16]_z(svbool_t pg, svuint16_t op1, svuint16_t op2)</code>
<code>svuint32_t svadd[_u32]_z(svbool_t pg, svuint32_t op1, svuint32_t op2)</code>
<code>svuint64_t svadd[_u64]_z(svbool_t pg, svuint64_t op1, svuint64_t op2)</code>

#### 6.7.1.2. ADD (vector, vector), merging with first input

Instances
<code>svint8_t svadd[_s8]_m(svbool_t pg, svint8_t op1, svint8_t op2)</code>
<code>svint16_t svadd[_s16]_m(svbool_t pg, svint16_t op1, svint16_t op2)</code>
<code>svint32_t svadd[_s32]_m(svbool_t pg, svint32_t op1, svint32_t op2)</code>
<code>svint64_t svadd[_s64]_m(svbool_t pg, svint64_t op1, svint64_t op2)</code>
<code>svuint8_t svadd[_u8]_m(svbool_t pg, svuint8_t op1, svuint8_t op2)</code>
<code>svuint16_t svadd[_u16]_m(svbool_t pg, svuint16_t op1, svuint16_t op2)</code>
<code>svuint32_t svadd[_u32]_m(svbool_t pg, svuint32_t op1, svuint32_t op2)</code>
<code>svuint64_t svadd[_u64]_m(svbool_t pg, svuint64_t op1, svuint64_t op2)</code>

#### 6.7.1.3. ADD (vector, vector), setting inactive to unknown

Instances
<code>svint8_t svadd[_s8]_x(svbool_t pg, svint8_t op1, svint8_t op2)</code>
<code>svint16_t svadd[_s16]_x(svbool_t pg, svint16_t op1, svint16_t op2)</code>
<code>svint32_t svadd[_s32]_x(svbool_t pg, svint32_t op1, svint32_t op2)</code>
<code>svint64_t svadd[_s64]_x(svbool_t pg, svint64_t op1, svint64_t op2)</code>
<code>svuint8_t svadd[_u8]_x(svbool_t pg, svuint8_t op1, svuint8_t op2)</code>
<code>svuint16_t svadd[_u16]_x(svbool_t pg, svuint16_t op1, svuint16_t op2)</code>
<code>svuint32_t svadd[_u32]_x(svbool_t pg, svuint32_t op1, svuint32_t op2)</code>



**Instances**

```
svuint64_t svadd[_u64]_x(svbool_t pg, svuint64_t op1, svuint64_t op2)
```

**6.7.1.4. ADD (vector, scalar), setting inactive to zero****Instances**

```
svint8_t svadd[_n_s8]_z(svbool_t pg, svint8_t op1, int8_t op2)
svint16_t svadd[_n_s16]_z(svbool_t pg, svint16_t op1, int16_t op2)
svint32_t svadd[_n_s32]_z(svbool_t pg, svint32_t op1, int32_t op2)
svint64_t svadd[_n_s64]_z(svbool_t pg, svint64_t op1, int64_t op2)
svuint8_t svadd[_n_u8]_z(svbool_t pg, svuint8_t op1, uint8_t op2)
svuint16_t svadd[_n_u16]_z(svbool_t pg, svuint16_t op1, uint16_t op2)
svuint32_t svadd[_n_u32]_z(svbool_t pg, svuint32_t op1, uint32_t op2)
svuint64_t svadd[_n_u64]_z(svbool_t pg, svuint64_t op1, uint64_t op2)
```

**6.7.1.5. ADD (vector, scalar), merging with first input****Instances**

```
svint8_t svadd[_n_s8]_m(svbool_t pg, svint8_t op1, int8_t op2)
svint16_t svadd[_n_s16]_m(svbool_t pg, svint16_t op1, int16_t op2)
svint32_t svadd[_n_s32]_m(svbool_t pg, svint32_t op1, int32_t op2)
svint64_t svadd[_n_s64]_m(svbool_t pg, svint64_t op1, int64_t op2)
svuint8_t svadd[_n_u8]_m(svbool_t pg, svuint8_t op1, uint8_t op2)
svuint16_t svadd[_n_u16]_m(svbool_t pg, svuint16_t op1, uint16_t op2)
svuint32_t svadd[_n_u32]_m(svbool_t pg, svuint32_t op1, uint32_t op2)
svuint64_t svadd[_n_u64]_m(svbool_t pg, svuint64_t op1, uint64_t op2)
```

**6.7.1.6. ADD (vector, scalar), setting inactive to unknown****Instances**

```
svint8_t svadd[_n_s8]_x(svbool_t pg, svint8_t op1, int8_t op2)
svint16_t svadd[_n_s16]_x(svbool_t pg, svint16_t op1, int16_t op2)
svint32_t svadd[_n_s32]_x(svbool_t pg, svint32_t op1, int32_t op2)
svint64_t svadd[_n_s64]_x(svbool_t pg, svint64_t op1, int64_t op2)
svuint8_t svadd[_n_u8]_x(svbool_t pg, svuint8_t op1, uint8_t op2)
svuint16_t svadd[_n_u16]_x(svbool_t pg, svuint16_t op1, uint16_t op2)
svuint32_t svadd[_n_u32]_x(svbool_t pg, svuint32_t op1, uint32_t op2)
svuint64_t svadd[_n_u64]_x(svbool_t pg, svuint64_t op1, uint64_t op2)
```

**6.7.2. QADD: Saturating integer addition**

These functions perform saturating addition on two integer inputs; that is, if the sum is outside the range of the type, the result is the nearest in-range value.

**6.7.2.1. QADD (vector, vector)****Instances**

```
svint8_t svqadd[_s8](svint8_t op1, svint8_t op2)
svint16_t svqadd[_s16](svint16_t op1, svint16_t op2)
svint32_t svqadd[_s32](svint32_t op1, svint32_t op2)
svint64_t svqadd[_s64](svint64_t op1, svint64_t op2)
svuint8_t svqadd[_u8](svuint8_t op1, svuint8_t op2)
svuint16_t svqadd[_u16](svuint16_t op1, svuint16_t op2)
svuint32_t svqadd[_u32](svuint32_t op1, svuint32_t op2)
svuint64_t svqadd[_u64](svuint64_t op1, svuint64_t op2)
```

### 6.7.2.2. QADD (vector, scalar)

Instances
svint8_t <b>svqadd</b> [_n_s8](svint8_t op1, int8_t op2)
svint16_t <b>svqadd</b> [_n_s16](svint16_t op1, int16_t op2)
svint32_t <b>svqadd</b> [_n_s32](svint32_t op1, int32_t op2)
svint64_t <b>svqadd</b> [_n_s64](svint64_t op1, int64_t op2)
svuint8_t <b>svqadd</b> [_n_u8](svuint8_t op1, uint8_t op2)
svuint16_t <b>svqadd</b> [_n_u16](svuint16_t op1, uint16_t op2)
svuint32_t <b>svqadd</b> [_n_u32](svuint32_t op1, uint32_t op2)
svuint64_t <b>svqadd</b> [_n_u64](svuint64_t op1, uint64_t op2)

### 6.7.3. SUB: Modular integer subtraction

These functions subtract the second integer input from the first input using modular arithmetic; that is, if the input elements have  $N$  bits, the result is the low  $N$  bits of the difference.

The result is well-defined for all inputs; unlike C, there is no undefined behavior for signed overflow.

#### 6.7.3.1. SUB (vector, vector), setting inactive to zero

Instances
svint8_t <b>svsub</b> [_s8]_z(svbool_t pg, svint8_t op1, svint8_t op2)
svint16_t <b>svsub</b> [_s16]_z(svbool_t pg, svint16_t op1, svint16_t op2)
svint32_t <b>svsub</b> [_s32]_z(svbool_t pg, svint32_t op1, svint32_t op2)
svint64_t <b>svsub</b> [_s64]_z(svbool_t pg, svint64_t op1, svint64_t op2)
svuint8_t <b>svsub</b> [_u8]_z(svbool_t pg, svuint8_t op1, svuint8_t op2)
svuint16_t <b>svsub</b> [_u16]_z(svbool_t pg, svuint16_t op1, svuint16_t op2)
svuint32_t <b>svsub</b> [_u32]_z(svbool_t pg, svuint32_t op1, svuint32_t op2)
svuint64_t <b>svsub</b> [_u64]_z(svbool_t pg, svuint64_t op1, svuint64_t op2)

#### 6.7.3.2. SUB (vector, vector), merging with first input

Instances
svint8_t <b>svsub</b> [_s8]_m(svbool_t pg, svint8_t op1, svint8_t op2)
svint16_t <b>svsub</b> [_s16]_m(svbool_t pg, svint16_t op1, svint16_t op2)
svint32_t <b>svsub</b> [_s32]_m(svbool_t pg, svint32_t op1, svint32_t op2)
svint64_t <b>svsub</b> [_s64]_m(svbool_t pg, svint64_t op1, svint64_t op2)
svuint8_t <b>svsub</b> [_u8]_m(svbool_t pg, svuint8_t op1, svuint8_t op2)
svuint16_t <b>svsub</b> [_u16]_m(svbool_t pg, svuint16_t op1, svuint16_t op2)
svuint32_t <b>svsub</b> [_u32]_m(svbool_t pg, svuint32_t op1, svuint32_t op2)
svuint64_t <b>svsub</b> [_u64]_m(svbool_t pg, svuint64_t op1, svuint64_t op2)

#### 6.7.3.3. SUB (vector, vector), setting inactive to unknown

Instances
svint8_t <b>svsub</b> [_s8]_x(svbool_t pg, svint8_t op1, svint8_t op2)
svint16_t <b>svsub</b> [_s16]_x(svbool_t pg, svint16_t op1, svint16_t op2)
svint32_t <b>svsub</b> [_s32]_x(svbool_t pg, svint32_t op1, svint32_t op2)
svint64_t <b>svsub</b> [_s64]_x(svbool_t pg, svint64_t op1, svint64_t op2)
svuint8_t <b>svsub</b> [_u8]_x(svbool_t pg, svuint8_t op1, svuint8_t op2)
svuint16_t <b>svsub</b> [_u16]_x(svbool_t pg, svuint16_t op1, svuint16_t op2)
svuint32_t <b>svsub</b> [_u32]_x(svbool_t pg, svuint32_t op1, svuint32_t op2)
svuint64_t <b>svsub</b> [_u64]_x(svbool_t pg, svuint64_t op1, svuint64_t op2)

### 6.7.3.4. SUB (vector, scalar), setting inactive to zero

Instances
<code>svint8_t svsub[_n_s8]_z(svbool_t pg, svint8_t op1, int8_t op2)</code>
<code>svint16_t svsub[_n_s16]_z(svbool_t pg, svint16_t op1, int16_t op2)</code>
<code>svint32_t svsub[_n_s32]_z(svbool_t pg, svint32_t op1, int32_t op2)</code>
<code>svint64_t svsub[_n_s64]_z(svbool_t pg, svint64_t op1, int64_t op2)</code>
<code>svuint8_t svsub[_n_u8]_z(svbool_t pg, svuint8_t op1, uint8_t op2)</code>
<code>svuint16_t svsub[_n_u16]_z(svbool_t pg, svuint16_t op1, uint16_t op2)</code>
<code>svuint32_t svsub[_n_u32]_z(svbool_t pg, svuint32_t op1, uint32_t op2)</code>
<code>svuint64_t svsub[_n_u64]_z(svbool_t pg, svuint64_t op1, uint64_t op2)</code>

### 6.7.3.5. SUB (vector, scalar), merging with first input

Instances
<code>svint8_t svsub[_n_s8]_m(svbool_t pg, svint8_t op1, int8_t op2)</code>
<code>svint16_t svsub[_n_s16]_m(svbool_t pg, svint16_t op1, int16_t op2)</code>
<code>svint32_t svsub[_n_s32]_m(svbool_t pg, svint32_t op1, int32_t op2)</code>
<code>svint64_t svsub[_n_s64]_m(svbool_t pg, svint64_t op1, int64_t op2)</code>
<code>svuint8_t svsub[_n_u8]_m(svbool_t pg, svuint8_t op1, uint8_t op2)</code>
<code>svuint16_t svsub[_n_u16]_m(svbool_t pg, svuint16_t op1, uint16_t op2)</code>
<code>svuint32_t svsub[_n_u32]_m(svbool_t pg, svuint32_t op1, uint32_t op2)</code>
<code>svuint64_t svsub[_n_u64]_m(svbool_t pg, svuint64_t op1, uint64_t op2)</code>

### 6.7.3.6. SUB (vector, scalar), setting inactive to unknown

Instances
<code>svint8_t svsub[_n_s8]_x(svbool_t pg, svint8_t op1, int8_t op2)</code>
<code>svint16_t svsub[_n_s16]_x(svbool_t pg, svint16_t op1, int16_t op2)</code>
<code>svint32_t svsub[_n_s32]_x(svbool_t pg, svint32_t op1, int32_t op2)</code>
<code>svint64_t svsub[_n_s64]_x(svbool_t pg, svint64_t op1, int64_t op2)</code>
<code>svuint8_t svsub[_n_u8]_x(svbool_t pg, svuint8_t op1, uint8_t op2)</code>
<code>svuint16_t svsub[_n_u16]_x(svbool_t pg, svuint16_t op1, uint16_t op2)</code>
<code>svuint32_t svsub[_n_u32]_x(svbool_t pg, svuint32_t op1, uint32_t op2)</code>
<code>svuint64_t svsub[_n_u64]_x(svbool_t pg, svuint64_t op1, uint64_t op2)</code>

## 6.7.4. SUBR: Modular integer subtraction, reversed

These functions subtract the first integer input from the second input; that is, if the input elements have  $N$  bits, the result is the low  $N$  bits of the difference.

The result is well-defined for all inputs; unlike C, there is no undefined behavior for signed overflow.

### 6.7.4.1. SUBR (vector, vector), setting inactive to zero

Instances
<code>svint8_t svsubr[_s8]_z(svbool_t pg, svint8_t op1, svint8_t op2)</code>
<code>svint16_t svsubr[_s16]_z(svbool_t pg, svint16_t op1, svint16_t op2)</code>
<code>svint32_t svsubr[_s32]_z(svbool_t pg, svint32_t op1, svint32_t op2)</code>
<code>svint64_t svsubr[_s64]_z(svbool_t pg, svint64_t op1, svint64_t op2)</code>
<code>svuint8_t svsubr[_u8]_z(svbool_t pg, svuint8_t op1, svuint8_t op2)</code>
<code>svuint16_t svsubr[_u16]_z(svbool_t pg, svuint16_t op1, svuint16_t op2)</code>
<code>svuint32_t svsubr[_u32]_z(svbool_t pg, svuint32_t op1, svuint32_t op2)</code>
<code>svuint64_t svsubr[_u64]_z(svbool_t pg, svuint64_t op1, svuint64_t op2)</code>

#### 6.7.4.2. SUBR (vector, vector), merging with first input

Instances
<pre> svint8_t  svsubr[_s8]_m(svbool_t pg, svint8_t op1, svint8_t op2) svint16_t svsubr[_s16]_m(svbool_t pg, svint16_t op1, svint16_t op2) svint32_t svsubr[_s32]_m(svbool_t pg, svint32_t op1, svint32_t op2) svint64_t svsubr[_s64]_m(svbool_t pg, svint64_t op1, svint64_t op2) svuint8_t svsubr[_u8]_m(svbool_t pg, svuint8_t op1, svuint8_t op2) svuint16_t svsubr[_u16]_m(svbool_t pg, svuint16_t op1, svuint16_t op2) svuint32_t svsubr[_u32]_m(svbool_t pg, svuint32_t op1, svuint32_t op2) svuint64_t svsubr[_u64]_m(svbool_t pg, svuint64_t op1, svuint64_t op2) </pre>

#### 6.7.4.3. SUBR (vector, vector), setting inactive to unknown

Instances
<pre> svint8_t  svsubr[_s8]_x(svbool_t pg, svint8_t op1, svint8_t op2) svint16_t svsubr[_s16]_x(svbool_t pg, svint16_t op1, svint16_t op2) svint32_t svsubr[_s32]_x(svbool_t pg, svint32_t op1, svint32_t op2) svint64_t svsubr[_s64]_x(svbool_t pg, svint64_t op1, svint64_t op2) svuint8_t svsubr[_u8]_x(svbool_t pg, svuint8_t op1, svuint8_t op2) svuint16_t svsubr[_u16]_x(svbool_t pg, svuint16_t op1, svuint16_t op2) svuint32_t svsubr[_u32]_x(svbool_t pg, svuint32_t op1, svuint32_t op2) svuint64_t svsubr[_u64]_x(svbool_t pg, svuint64_t op1, svuint64_t op2) </pre>

#### 6.7.4.4. SUBR (vector, scalar), setting inactive to zero

Instances
<pre> svint8_t  svsubr[_n_s8]_z(svbool_t pg, svint8_t op1, int8_t op2) svint16_t svsubr[_n_s16]_z(svbool_t pg, svint16_t op1, int16_t op2) svint32_t svsubr[_n_s32]_z(svbool_t pg, svint32_t op1, int32_t op2) svint64_t svsubr[_n_s64]_z(svbool_t pg, svint64_t op1, int64_t op2) svuint8_t svsubr[_n_u8]_z(svbool_t pg, svuint8_t op1, uint8_t op2) svuint16_t svsubr[_n_u16]_z(svbool_t pg, svuint16_t op1, uint16_t op2) svuint32_t svsubr[_n_u32]_z(svbool_t pg, svuint32_t op1, uint32_t op2) svuint64_t svsubr[_n_u64]_z(svbool_t pg, svuint64_t op1, uint64_t op2) </pre>

#### 6.7.4.5. SUBR (vector, scalar), merging with first input

Instances
<pre> svint8_t  svsubr[_n_s8]_m(svbool_t pg, svint8_t op1, int8_t op2) svint16_t svsubr[_n_s16]_m(svbool_t pg, svint16_t op1, int16_t op2) svint32_t svsubr[_n_s32]_m(svbool_t pg, svint32_t op1, int32_t op2) svint64_t svsubr[_n_s64]_m(svbool_t pg, svint64_t op1, int64_t op2) svuint8_t svsubr[_n_u8]_m(svbool_t pg, svuint8_t op1, uint8_t op2) svuint16_t svsubr[_n_u16]_m(svbool_t pg, svuint16_t op1, uint16_t op2) svuint32_t svsubr[_n_u32]_m(svbool_t pg, svuint32_t op1, uint32_t op2) svuint64_t svsubr[_n_u64]_m(svbool_t pg, svuint64_t op1, uint64_t op2) </pre>

#### 6.7.4.6. SUBR (vector, scalar), setting inactive to unknown

Instances
<pre> svint8_t  svsubr[_n_s8]_x(svbool_t pg, svint8_t op1, int8_t op2) svint16_t svsubr[_n_s16]_x(svbool_t pg, svint16_t op1, int16_t op2) </pre>

**Instances**

```
svint32_t svsubr[_n_s32]_x(svbool_t pg, svint32_t op1, int32_t op2)
svint64_t svsubr[_n_s64]_x(svbool_t pg, svint64_t op1, int64_t op2)
svuint8_t svsubr[_n_u8]_x(svbool_t pg, svuint8_t op1, uint8_t op2)
svuint16_t svsubr[_n_u16]_x(svbool_t pg, svuint16_t op1, uint16_t op2)
svuint32_t svsubr[_n_u32]_x(svbool_t pg, svuint32_t op1, uint32_t op2)
svuint64_t svsubr[_n_u64]_x(svbool_t pg, svuint64_t op1, uint64_t op2)
```

## 6.7.5. QSUB: Saturating integer subtraction

These functions perform saturating subtraction on two integer inputs; that is, if the difference is outside the range of the type, the result is the nearest in-range value.

### 6.7.5.1. QSUB (vector, vector)

**Instances**

```
svint8_t svqsub[_s8](svint8_t op1, svint8_t op2)
svint16_t svqsub[_s16](svint16_t op1, svint16_t op2)
svint32_t svqsub[_s32](svint32_t op1, svint32_t op2)
svint64_t svqsub[_s64](svint64_t op1, svint64_t op2)
svuint8_t svqsub[_u8](svuint8_t op1, svuint8_t op2)
svuint16_t svqsub[_u16](svuint16_t op1, svuint16_t op2)
svuint32_t svqsub[_u32](svuint32_t op1, svuint32_t op2)
svuint64_t svqsub[_u64](svuint64_t op1, svuint64_t op2)
```

### 6.7.5.2. QSUB (vector, scalar)

**Instances**

```
svint8_t svqsub[_n_s8](svint8_t op1, int8_t op2)
svint16_t svqsub[_n_s16](svint16_t op1, int16_t op2)
svint32_t svqsub[_n_s32](svint32_t op1, int32_t op2)
svint64_t svqsub[_n_s64](svint64_t op1, int64_t op2)
svuint8_t svqsub[_n_u8](svuint8_t op1, uint8_t op2)
svuint16_t svqsub[_n_u16](svuint16_t op1, uint16_t op2)
svuint32_t svqsub[_n_u32](svuint32_t op1, uint32_t op2)
svuint64_t svqsub[_n_u64](svuint64_t op1, uint64_t op2)
```

## 6.7.6. ABD: Integer absolute difference

These functions compute the absolute difference of two integer inputs. This operation is well-defined for all input values.

### 6.7.6.1. ABD (vector, vector), setting inactive to zero

**Instances**

```
svint8_t svabd[_s8]_z(svbool_t pg, svint8_t op1, svint8_t op2)
svint16_t svabd[_s16]_z(svbool_t pg, svint16_t op1, svint16_t op2)
svint32_t svabd[_s32]_z(svbool_t pg, svint32_t op1, svint32_t op2)
svint64_t svabd[_s64]_z(svbool_t pg, svint64_t op1, svint64_t op2)
svuint8_t svabd[_u8]_z(svbool_t pg, svuint8_t op1, svuint8_t op2)
svuint16_t svabd[_u16]_z(svbool_t pg, svuint16_t op1, svuint16_t op2)
svuint32_t svabd[_u32]_z(svbool_t pg, svuint32_t op1, svuint32_t op2)
svuint64_t svabd[_u64]_z(svbool_t pg, svuint64_t op1, svuint64_t op2)
```

### 6.7.6.2. ABD (vector, vector), merging with first input

Instances
svint8_t <b>svabd</b> [_s8]_m(svbool_t pg, svint8_t op1, svint8_t op2)
svint16_t <b>svabd</b> [_s16]_m(svbool_t pg, svint16_t op1, svint16_t op2)
svint32_t <b>svabd</b> [_s32]_m(svbool_t pg, svint32_t op1, svint32_t op2)
svint64_t <b>svabd</b> [_s64]_m(svbool_t pg, svint64_t op1, svint64_t op2)
svuint8_t <b>svabd</b> [_u8]_m(svbool_t pg, svuint8_t op1, svuint8_t op2)
svuint16_t <b>svabd</b> [_u16]_m(svbool_t pg, svuint16_t op1, svuint16_t op2)
svuint32_t <b>svabd</b> [_u32]_m(svbool_t pg, svuint32_t op1, svuint32_t op2)
svuint64_t <b>svabd</b> [_u64]_m(svbool_t pg, svuint64_t op1, svuint64_t op2)

### 6.7.6.3. ABD (vector, vector), setting inactive to unknown

Instances
svint8_t <b>svabd</b> [_s8]_x(svbool_t pg, svint8_t op1, svint8_t op2)
svint16_t <b>svabd</b> [_s16]_x(svbool_t pg, svint16_t op1, svint16_t op2)
svint32_t <b>svabd</b> [_s32]_x(svbool_t pg, svint32_t op1, svint32_t op2)
svint64_t <b>svabd</b> [_s64]_x(svbool_t pg, svint64_t op1, svint64_t op2)
svuint8_t <b>svabd</b> [_u8]_x(svbool_t pg, svuint8_t op1, svuint8_t op2)
svuint16_t <b>svabd</b> [_u16]_x(svbool_t pg, svuint16_t op1, svuint16_t op2)
svuint32_t <b>svabd</b> [_u32]_x(svbool_t pg, svuint32_t op1, svuint32_t op2)
svuint64_t <b>svabd</b> [_u64]_x(svbool_t pg, svuint64_t op1, svuint64_t op2)

### 6.7.6.4. ABD (vector, scalar), setting inactive to zero

Instances
svint8_t <b>svabd</b> [_n_s8]_z(svbool_t pg, svint8_t op1, int8_t op2)
svint16_t <b>svabd</b> [_n_s16]_z(svbool_t pg, svint16_t op1, int16_t op2)
svint32_t <b>svabd</b> [_n_s32]_z(svbool_t pg, svint32_t op1, int32_t op2)
svint64_t <b>svabd</b> [_n_s64]_z(svbool_t pg, svint64_t op1, int64_t op2)
svuint8_t <b>svabd</b> [_n_u8]_z(svbool_t pg, svuint8_t op1, uint8_t op2)
svuint16_t <b>svabd</b> [_n_u16]_z(svbool_t pg, svuint16_t op1, uint16_t op2)
svuint32_t <b>svabd</b> [_n_u32]_z(svbool_t pg, svuint32_t op1, uint32_t op2)
svuint64_t <b>svabd</b> [_n_u64]_z(svbool_t pg, svuint64_t op1, uint64_t op2)

### 6.7.6.5. ABD (vector, scalar), merging with first input

Instances
svint8_t <b>svabd</b> [_n_s8]_m(svbool_t pg, svint8_t op1, int8_t op2)
svint16_t <b>svabd</b> [_n_s16]_m(svbool_t pg, svint16_t op1, int16_t op2)
svint32_t <b>svabd</b> [_n_s32]_m(svbool_t pg, svint32_t op1, int32_t op2)
svint64_t <b>svabd</b> [_n_s64]_m(svbool_t pg, svint64_t op1, int64_t op2)
svuint8_t <b>svabd</b> [_n_u8]_m(svbool_t pg, svuint8_t op1, uint8_t op2)
svuint16_t <b>svabd</b> [_n_u16]_m(svbool_t pg, svuint16_t op1, uint16_t op2)
svuint32_t <b>svabd</b> [_n_u32]_m(svbool_t pg, svuint32_t op1, uint32_t op2)
svuint64_t <b>svabd</b> [_n_u64]_m(svbool_t pg, svuint64_t op1, uint64_t op2)

### 6.7.6.6. ABD (vector, scalar), setting inactive to unknown

Instances
svint8_t <b>svabd</b> [_n_s8]_x(svbool_t pg, svint8_t op1, int8_t op2)
svint16_t <b>svabd</b> [_n_s16]_x(svbool_t pg, svint16_t op1, int16_t op2)
svint32_t <b>svabd</b> [_n_s32]_x(svbool_t pg, svint32_t op1, int32_t op2)
svint64_t <b>svabd</b> [_n_s64]_x(svbool_t pg, svint64_t op1, int64_t op2)

**Instances**

```
svuint8_t  svabd[_n_u8]_x(svbool_t pg, svuint8_t op1, uint8_t op2)
svuint16_t svabd[_n_u16]_x(svbool_t pg, svuint16_t op1, uint16_t op2)
svuint32_t svabd[_n_u32]_x(svbool_t pg, svuint32_t op1, uint32_t op2)
svuint64_t svabd[_n_u64]_x(svbool_t pg, svuint64_t op1, uint64_t op2)
```

**6.7.7. MUL: Integer multiplication, returning low half**

These functions multiply two integer inputs and return the low half of the result. That is, if the input elements have  $N$  bits, the result is the low  $N$  bits of their product.

The result is well-defined for all inputs; unlike C, there is no undefined behavior for signed overflow.

**6.7.7.1. MUL (vector, vector), setting inactive to zero****Instances**

```
svint8_t  svmul[_s8]_z(svbool_t pg, svint8_t op1, svint8_t op2)
svint16_t svmul[_s16]_z(svbool_t pg, svint16_t op1, svint16_t op2)
svint32_t svmul[_s32]_z(svbool_t pg, svint32_t op1, svint32_t op2)
svint64_t svmul[_s64]_z(svbool_t pg, svint64_t op1, svint64_t op2)
svuint8_t svmul[_u8]_z(svbool_t pg, svuint8_t op1, svuint8_t op2)
svuint16_t svmul[_u16]_z(svbool_t pg, svuint16_t op1, svuint16_t op2)
svuint32_t svmul[_u32]_z(svbool_t pg, svuint32_t op1, svuint32_t op2)
svuint64_t svmul[_u64]_z(svbool_t pg, svuint64_t op1, svuint64_t op2)
```

**6.7.7.2. MUL (vector, vector), merging with first input****Instances**

```
svint8_t  svmul[_s8]_m(svbool_t pg, svint8_t op1, svint8_t op2)
svint16_t svmul[_s16]_m(svbool_t pg, svint16_t op1, svint16_t op2)
svint32_t svmul[_s32]_m(svbool_t pg, svint32_t op1, svint32_t op2)
svint64_t svmul[_s64]_m(svbool_t pg, svint64_t op1, svint64_t op2)
svuint8_t svmul[_u8]_m(svbool_t pg, svuint8_t op1, svuint8_t op2)
svuint16_t svmul[_u16]_m(svbool_t pg, svuint16_t op1, svuint16_t op2)
svuint32_t svmul[_u32]_m(svbool_t pg, svuint32_t op1, svuint32_t op2)
svuint64_t svmul[_u64]_m(svbool_t pg, svuint64_t op1, svuint64_t op2)
```

**6.7.7.3. MUL (vector, vector), setting inactive to unknown****Instances**

```
svint8_t  svmul[_s8]_x(svbool_t pg, svint8_t op1, svint8_t op2)
svint16_t svmul[_s16]_x(svbool_t pg, svint16_t op1, svint16_t op2)
svint32_t svmul[_s32]_x(svbool_t pg, svint32_t op1, svint32_t op2)
svint64_t svmul[_s64]_x(svbool_t pg, svint64_t op1, svint64_t op2)
svuint8_t svmul[_u8]_x(svbool_t pg, svuint8_t op1, svuint8_t op2)
svuint16_t svmul[_u16]_x(svbool_t pg, svuint16_t op1, svuint16_t op2)
svuint32_t svmul[_u32]_x(svbool_t pg, svuint32_t op1, svuint32_t op2)
svuint64_t svmul[_u64]_x(svbool_t pg, svuint64_t op1, svuint64_t op2)
```

**6.7.7.4. MUL (vector, scalar), setting inactive to zero****Instances**

```
svint8_t  svmul[_n_s8]_z(svbool_t pg, svint8_t op1, int8_t op2)
svint16_t svmul[_n_s16]_z(svbool_t pg, svint16_t op1, int16_t op2)
```

Instances
svint32_t <b>svmul</b> [_n_s32]_z(svbool_t pg, svint32_t op1, int32_t op2)
svint64_t <b>svmul</b> [_n_s64]_z(svbool_t pg, svint64_t op1, int64_t op2)
svuint8_t <b>svmul</b> [_n_u8]_z(svbool_t pg, svuint8_t op1, uint8_t op2)
svuint16_t <b>svmul</b> [_n_u16]_z(svbool_t pg, svuint16_t op1, uint16_t op2)
svuint32_t <b>svmul</b> [_n_u32]_z(svbool_t pg, svuint32_t op1, uint32_t op2)
svuint64_t <b>svmul</b> [_n_u64]_z(svbool_t pg, svuint64_t op1, uint64_t op2)

#### 6.7.7.5. MUL (vector, scalar), merging with first input

Instances
svint8_t <b>svmul</b> [_n_s8]_m(svbool_t pg, svint8_t op1, int8_t op2)
svint16_t <b>svmul</b> [_n_s16]_m(svbool_t pg, svint16_t op1, int16_t op2)
svint32_t <b>svmul</b> [_n_s32]_m(svbool_t pg, svint32_t op1, int32_t op2)
svint64_t <b>svmul</b> [_n_s64]_m(svbool_t pg, svint64_t op1, int64_t op2)
svuint8_t <b>svmul</b> [_n_u8]_m(svbool_t pg, svuint8_t op1, uint8_t op2)
svuint16_t <b>svmul</b> [_n_u16]_m(svbool_t pg, svuint16_t op1, uint16_t op2)
svuint32_t <b>svmul</b> [_n_u32]_m(svbool_t pg, svuint32_t op1, uint32_t op2)
svuint64_t <b>svmul</b> [_n_u64]_m(svbool_t pg, svuint64_t op1, uint64_t op2)

#### 6.7.7.6. MUL (vector, scalar), setting inactive to unknown

Instances
svint8_t <b>svmul</b> [_n_s8]_x(svbool_t pg, svint8_t op1, int8_t op2)
svint16_t <b>svmul</b> [_n_s16]_x(svbool_t pg, svint16_t op1, int16_t op2)
svint32_t <b>svmul</b> [_n_s32]_x(svbool_t pg, svint32_t op1, int32_t op2)
svint64_t <b>svmul</b> [_n_s64]_x(svbool_t pg, svint64_t op1, int64_t op2)
svuint8_t <b>svmul</b> [_n_u8]_x(svbool_t pg, svuint8_t op1, uint8_t op2)
svuint16_t <b>svmul</b> [_n_u16]_x(svbool_t pg, svuint16_t op1, uint16_t op2)
svuint32_t <b>svmul</b> [_n_u32]_x(svbool_t pg, svuint32_t op1, uint32_t op2)
svuint64_t <b>svmul</b> [_n_u64]_x(svbool_t pg, svuint64_t op1, uint64_t op2)

### 6.7.8. MULH: Integer multiplication, returning high half

These functions multiply two integer inputs and return the high half of the result. That is, if the input elements have  $N$  bits, the result contains bits  $[N, 2 \times N)$  of their product.

#### 6.7.8.1. MULH (vector, vector), setting inactive to zero

Instances
svint8_t <b>svmulh</b> [_s8]_z(svbool_t pg, svint8_t op1, svint8_t op2)
svint16_t <b>svmulh</b> [_s16]_z(svbool_t pg, svint16_t op1, svint16_t op2)
svint32_t <b>svmulh</b> [_s32]_z(svbool_t pg, svint32_t op1, svint32_t op2)
svint64_t <b>svmulh</b> [_s64]_z(svbool_t pg, svint64_t op1, svint64_t op2)
svuint8_t <b>svmulh</b> [_u8]_z(svbool_t pg, svuint8_t op1, svuint8_t op2)
svuint16_t <b>svmulh</b> [_u16]_z(svbool_t pg, svuint16_t op1, svuint16_t op2)
svuint32_t <b>svmulh</b> [_u32]_z(svbool_t pg, svuint32_t op1, svuint32_t op2)
svuint64_t <b>svmulh</b> [_u64]_z(svbool_t pg, svuint64_t op1, svuint64_t op2)

#### 6.7.8.2. MULH (vector, vector), merging with first input

Instances
svint8_t <b>svmulh</b> [_s8]_m(svbool_t pg, svint8_t op1, svint8_t op2)
svint16_t <b>svmulh</b> [_s16]_m(svbool_t pg, svint16_t op1, svint16_t op2)



**Instances**

```
svint32_t svmulh[_s32]_m(svbool_t pg, svint32_t op1, svint32_t op2)
svint64_t svmulh[_s64]_m(svbool_t pg, svint64_t op1, svint64_t op2)
svuint8_t svmulh[_u8]_m(svbool_t pg, svuint8_t op1, svuint8_t op2)
svuint16_t svmulh[_u16]_m(svbool_t pg, svuint16_t op1, svuint16_t op2)
svuint32_t svmulh[_u32]_m(svbool_t pg, svuint32_t op1, svuint32_t op2)
svuint64_t svmulh[_u64]_m(svbool_t pg, svuint64_t op1, svuint64_t op2)
```

**6.7.8.3. MULH (vector, vector), setting inactive to unknown****Instances**

```
svint8_t svmulh[_s8]_x(svbool_t pg, svint8_t op1, svint8_t op2)
svint16_t svmulh[_s16]_x(svbool_t pg, svint16_t op1, svint16_t op2)
svint32_t svmulh[_s32]_x(svbool_t pg, svint32_t op1, svint32_t op2)
svint64_t svmulh[_s64]_x(svbool_t pg, svint64_t op1, svint64_t op2)
svuint8_t svmulh[_u8]_x(svbool_t pg, svuint8_t op1, svuint8_t op2)
svuint16_t svmulh[_u16]_x(svbool_t pg, svuint16_t op1, svuint16_t op2)
svuint32_t svmulh[_u32]_x(svbool_t pg, svuint32_t op1, svuint32_t op2)
svuint64_t svmulh[_u64]_x(svbool_t pg, svuint64_t op1, svuint64_t op2)
```

**6.7.8.4. MULH (vector, scalar), setting inactive to zero****Instances**

```
svint8_t svmulh[_n_s8]_z(svbool_t pg, svint8_t op1, int8_t op2)
svint16_t svmulh[_n_s16]_z(svbool_t pg, svint16_t op1, int16_t op2)
svint32_t svmulh[_n_s32]_z(svbool_t pg, svint32_t op1, int32_t op2)
svint64_t svmulh[_n_s64]_z(svbool_t pg, svint64_t op1, int64_t op2)
svuint8_t svmulh[_n_u8]_z(svbool_t pg, svuint8_t op1, uint8_t op2)
svuint16_t svmulh[_n_u16]_z(svbool_t pg, svuint16_t op1, uint16_t op2)
svuint32_t svmulh[_n_u32]_z(svbool_t pg, svuint32_t op1, uint32_t op2)
svuint64_t svmulh[_n_u64]_z(svbool_t pg, svuint64_t op1, uint64_t op2)
```

**6.7.8.5. MULH (vector, scalar), merging with first input****Instances**

```
svint8_t svmulh[_n_s8]_m(svbool_t pg, svint8_t op1, int8_t op2)
svint16_t svmulh[_n_s16]_m(svbool_t pg, svint16_t op1, int16_t op2)
svint32_t svmulh[_n_s32]_m(svbool_t pg, svint32_t op1, int32_t op2)
svint64_t svmulh[_n_s64]_m(svbool_t pg, svint64_t op1, int64_t op2)
svuint8_t svmulh[_n_u8]_m(svbool_t pg, svuint8_t op1, uint8_t op2)
svuint16_t svmulh[_n_u16]_m(svbool_t pg, svuint16_t op1, uint16_t op2)
svuint32_t svmulh[_n_u32]_m(svbool_t pg, svuint32_t op1, uint32_t op2)
svuint64_t svmulh[_n_u64]_m(svbool_t pg, svuint64_t op1, uint64_t op2)
```

**6.7.8.6. MULH (vector, scalar), setting inactive to unknown****Instances**

```
svint8_t svmulh[_n_s8]_x(svbool_t pg, svint8_t op1, int8_t op2)
svint16_t svmulh[_n_s16]_x(svbool_t pg, svint16_t op1, int16_t op2)
svint32_t svmulh[_n_s32]_x(svbool_t pg, svint32_t op1, int32_t op2)
svint64_t svmulh[_n_s64]_x(svbool_t pg, svint64_t op1, int64_t op2)
svuint8_t svmulh[_n_u8]_x(svbool_t pg, svuint8_t op1, uint8_t op2)
svuint16_t svmulh[_n_u16]_x(svbool_t pg, svuint16_t op1, uint16_t op2)
svuint32_t svmulh[_n_u32]_x(svbool_t pg, svuint32_t op1, uint32_t op2)
svuint64_t svmulh[_n_u64]_x(svbool_t pg, svuint64_t op1, uint64_t op2)
```

### 6.7.9. MAD: Integer addition of product (multiplicand first)

These functions multiply the first two integer inputs and add the result to the third input. Both the multiplication and addition use modular arithmetic.

The result is well-defined for all inputs; there is no undefined behavior for signed overflow.

#### 6.7.9.1. MAD (vector, vector, vector), setting inactive to zero

Instances	
<code>svint8_t</code>	<code>svmad[_s8]_z(svbool_t pg, svint8_t op1, svint8_t op2, svint8_t op3)</code>
<code>svint16_t</code>	<code>svmad[_s16]_z(svbool_t pg, svint16_t op1, svint16_t op2, svint16_t op3)</code>
<code>svint32_t</code>	<code>svmad[_s32]_z(svbool_t pg, svint32_t op1, svint32_t op2, svint32_t op3)</code>
<code>svint64_t</code>	<code>svmad[_s64]_z(svbool_t pg, svint64_t op1, svint64_t op2, svint64_t op3)</code>
<code>svuint8_t</code>	<code>svmad[_u8]_z(svbool_t pg, svuint8_t op1, svuint8_t op2, svuint8_t op3)</code>
<code>svuint16_t</code>	<code>svmad[_u16]_z(svbool_t pg, svuint16_t op1, svuint16_t op2, svuint16_t op3)</code>
<code>svuint32_t</code>	<code>svmad[_u32]_z(svbool_t pg, svuint32_t op1, svuint32_t op2, svuint32_t op3)</code>
<code>svuint64_t</code>	<code>svmad[_u64]_z(svbool_t pg, svuint64_t op1, svuint64_t op2, svuint64_t op3)</code>

#### 6.7.9.2. MAD (vector, vector, vector), merging with first input

Instances	
<code>svint8_t</code>	<code>svmad[_s8]_m(svbool_t pg, svint8_t op1, svint8_t op2, svint8_t op3)</code>
<code>svint16_t</code>	<code>svmad[_s16]_m(svbool_t pg, svint16_t op1, svint16_t op2, svint16_t op3)</code>
<code>svint32_t</code>	<code>svmad[_s32]_m(svbool_t pg, svint32_t op1, svint32_t op2, svint32_t op3)</code>
<code>svint64_t</code>	<code>svmad[_s64]_m(svbool_t pg, svint64_t op1, svint64_t op2, svint64_t op3)</code>
<code>svuint8_t</code>	<code>svmad[_u8]_m(svbool_t pg, svuint8_t op1, svuint8_t op2, svuint8_t op3)</code>
<code>svuint16_t</code>	<code>svmad[_u16]_m(svbool_t pg, svuint16_t op1, svuint16_t op2, svuint16_t op3)</code>
<code>svuint32_t</code>	<code>svmad[_u32]_m(svbool_t pg, svuint32_t op1, svuint32_t op2, svuint32_t op3)</code>
<code>svuint64_t</code>	<code>svmad[_u64]_m(svbool_t pg, svuint64_t op1, svuint64_t op2, svuint64_t op3)</code>

#### 6.7.9.3. MAD (vector, vector, vector), setting inactive to unknown

Instances	
<code>svint8_t</code>	<code>svmad[_s8]_x(svbool_t pg, svint8_t op1, svint8_t op2, svint8_t op3)</code>
<code>svint16_t</code>	<code>svmad[_s16]_x(svbool_t pg, svint16_t op1, svint16_t op2, svint16_t op3)</code>
<code>svint32_t</code>	<code>svmad[_s32]_x(svbool_t pg, svint32_t op1, svint32_t op2, svint32_t op3)</code>

**Instances**

```
svint64_t svmad[_s64]_x(svbool_t pg, svint64_t op1, svint64_t op2,
                        svint64_t op3)
svuint8_t svmad[_u8]_x(svbool_t pg, svuint8_t op1, svuint8_t op2,
                       svuint8_t op3)
svuint16_t svmad[_u16]_x(svbool_t pg, svuint16_t op1, svuint16_t op2,
                         svuint16_t op3)
svuint32_t svmad[_u32]_x(svbool_t pg, svuint32_t op1, svuint32_t op2,
                         svuint32_t op3)
svuint64_t svmad[_u64]_x(svbool_t pg, svuint64_t op1, svuint64_t op2,
                         svuint64_t op3)
```

**6.7.9.4. MAD (vector, vector, scalar), setting inactive to zero****Instances**

```
svint8_t svmad[_n_s8]_z(svbool_t pg, svint8_t op1, svint8_t op2,
                        int8_t op3)
svint16_t svmad[_n_s16]_z(svbool_t pg, svint16_t op1, svint16_t op2,
                          int16_t op3)
svint32_t svmad[_n_s32]_z(svbool_t pg, svint32_t op1, svint32_t op2,
                          int32_t op3)
svint64_t svmad[_n_s64]_z(svbool_t pg, svint64_t op1, svint64_t op2,
                          int64_t op3)
svuint8_t svmad[_n_u8]_z(svbool_t pg, svuint8_t op1, svuint8_t op2,
                         uint8_t op3)
svuint16_t svmad[_n_u16]_z(svbool_t pg, svuint16_t op1, svuint16_t op2,
                          uint16_t op3)
svuint32_t svmad[_n_u32]_z(svbool_t pg, svuint32_t op1, svuint32_t op2,
                          uint32_t op3)
svuint64_t svmad[_n_u64]_z(svbool_t pg, svuint64_t op1, svuint64_t op2,
                          uint64_t op3)
```

**6.7.9.5. MAD (vector, vector, scalar), merging with first input****Instances**

```
svint8_t svmad[_n_s8]_m(svbool_t pg, svint8_t op1, svint8_t op2,
                        int8_t op3)
svint16_t svmad[_n_s16]_m(svbool_t pg, svint16_t op1, svint16_t op2,
                          int16_t op3)
svint32_t svmad[_n_s32]_m(svbool_t pg, svint32_t op1, svint32_t op2,
                          int32_t op3)
svint64_t svmad[_n_s64]_m(svbool_t pg, svint64_t op1, svint64_t op2,
                          int64_t op3)
svuint8_t svmad[_n_u8]_m(svbool_t pg, svuint8_t op1, svuint8_t op2,
                         uint8_t op3)
svuint16_t svmad[_n_u16]_m(svbool_t pg, svuint16_t op1, svuint16_t op2,
                          uint16_t op3)
svuint32_t svmad[_n_u32]_m(svbool_t pg, svuint32_t op1, svuint32_t op2,
                          uint32_t op3)
svuint64_t svmad[_n_u64]_m(svbool_t pg, svuint64_t op1, svuint64_t op2,
                          uint64_t op3)
```

**6.7.9.6. MAD (vector, vector, scalar), setting inactive to unknown****Instances**

```
svint8_t svmad[_n_s8]_x(svbool_t pg, svint8_t op1, svint8_t op2,
```

Instances	
<code>svint16_t</code>	<code>svmad[_n_s16]_x(svbool_t pg, svint16_t op1, svint16_t op2, int8_t op3)</code>
<code>svint32_t</code>	<code>svmad[_n_s32]_x(svbool_t pg, svint32_t op1, svint32_t op2, int16_t op3)</code>
<code>svint64_t</code>	<code>svmad[_n_s64]_x(svbool_t pg, svint64_t op1, svint64_t op2, int32_t op3)</code>
<code>svuint8_t</code>	<code>svmad[_n_u8]_x(svbool_t pg, svuint8_t op1, svuint8_t op2, int64_t op3)</code>
<code>svuint16_t</code>	<code>svmad[_n_u16]_x(svbool_t pg, svuint16_t op1, svuint16_t op2, uint8_t op3)</code>
<code>svuint32_t</code>	<code>svmad[_n_u32]_x(svbool_t pg, svuint32_t op1, svuint32_t op2, uint16_t op3)</code>
<code>svuint64_t</code>	<code>svmad[_n_u64]_x(svbool_t pg, svuint64_t op1, svuint64_t op2, uint32_t op3)</code>

## 6.7.10. MLA: Integer addition of product (addend first)

These functions multiply the second and third integer inputs and add the result to the first input. Both the multiplication and addition use modular arithmetic.

The result is well-defined for all inputs; there is no undefined behavior for signed overflow.

### 6.7.10.1. MLA (vector, vector, vector), setting inactive to zero

Instances	
<code>svint8_t</code>	<code>svmla[_s8]_z(svbool_t pg, svint8_t op1, svint8_t op2, svint8_t op3)</code>
<code>svint16_t</code>	<code>svmla[_s16]_z(svbool_t pg, svint16_t op1, svint16_t op2, svint16_t op3)</code>
<code>svint32_t</code>	<code>svmla[_s32]_z(svbool_t pg, svint32_t op1, svint32_t op2, svint32_t op3)</code>
<code>svint64_t</code>	<code>svmla[_s64]_z(svbool_t pg, svint64_t op1, svint64_t op2, svint64_t op3)</code>
<code>svuint8_t</code>	<code>svmla[_u8]_z(svbool_t pg, svuint8_t op1, svuint8_t op2, svuint8_t op3)</code>
<code>svuint16_t</code>	<code>svmla[_u16]_z(svbool_t pg, svuint16_t op1, svuint16_t op2, svuint16_t op3)</code>
<code>svuint32_t</code>	<code>svmla[_u32]_z(svbool_t pg, svuint32_t op1, svuint32_t op2, svuint32_t op3)</code>
<code>svuint64_t</code>	<code>svmla[_u64]_z(svbool_t pg, svuint64_t op1, svuint64_t op2, svuint64_t op3)</code>

### 6.7.10.2. MLA (vector, vector, vector), merging with first input

Instances	
<code>svint8_t</code>	<code>svmla[_s8]_m(svbool_t pg, svint8_t op1, svint8_t op2, svint8_t op3)</code>
<code>svint16_t</code>	<code>svmla[_s16]_m(svbool_t pg, svint16_t op1, svint16_t op2, svint16_t op3)</code>
<code>svint32_t</code>	<code>svmla[_s32]_m(svbool_t pg, svint32_t op1, svint32_t op2, svint32_t op3)</code>
<code>svint64_t</code>	<code>svmla[_s64]_m(svbool_t pg, svint64_t op1, svint64_t op2, svint64_t op3)</code>
<code>svuint8_t</code>	<code>svmla[_u8]_m(svbool_t pg, svuint8_t op1, svuint8_t op2, svuint8_t op3)</code>

**Instances**

```

svuint8_t svmla[_u8]_m(svbool_t pg, svuint8_t op1, svuint8_t op2,
                        svuint8_t op3)
svuint16_t svmla[_u16]_m(svbool_t pg, svuint16_t op1, svuint16_t op2,
                        svuint16_t op3)
svuint32_t svmla[_u32]_m(svbool_t pg, svuint32_t op1, svuint32_t op2,
                        svuint32_t op3)
svuint64_t svmla[_u64]_m(svbool_t pg, svuint64_t op1, svuint64_t op2,
                        svuint64_t op3)

```

**6.7.10.3. MLA (vector, vector, vector), setting inactive to unknown****Instances**

```

svint8_t svmla[_s8]_x(svbool_t pg, svint8_t op1, svint8_t op2,
                     svint8_t op3)
svint16_t svmla[_s16]_x(svbool_t pg, svint16_t op1, svint16_t op2,
                     svint16_t op3)
svint32_t svmla[_s32]_x(svbool_t pg, svint32_t op1, svint32_t op2,
                     svint32_t op3)
svint64_t svmla[_s64]_x(svbool_t pg, svint64_t op1, svint64_t op2,
                     svint64_t op3)
svuint8_t svmla[_u8]_x(svbool_t pg, svuint8_t op1, svuint8_t op2,
                     svuint8_t op3)
svuint16_t svmla[_u16]_x(svbool_t pg, svuint16_t op1, svuint16_t op2,
                     svuint16_t op3)
svuint32_t svmla[_u32]_x(svbool_t pg, svuint32_t op1, svuint32_t op2,
                     svuint32_t op3)
svuint64_t svmla[_u64]_x(svbool_t pg, svuint64_t op1, svuint64_t op2,
                     svuint64_t op3)

```

**6.7.10.4. MLA (vector, vector, scalar), setting inactive to zero****Instances**

```

svint8_t svmla[_n_s8]_z(svbool_t pg, svint8_t op1, svint8_t op2,
                       int8_t op3)
svint16_t svmla[_n_s16]_z(svbool_t pg, svint16_t op1, svint16_t op2,
                       int16_t op3)
svint32_t svmla[_n_s32]_z(svbool_t pg, svint32_t op1, svint32_t op2,
                       int32_t op3)
svint64_t svmla[_n_s64]_z(svbool_t pg, svint64_t op1, svint64_t op2,
                       int64_t op3)
svuint8_t svmla[_n_u8]_z(svbool_t pg, svuint8_t op1, svuint8_t op2,
                       uint8_t op3)
svuint16_t svmla[_n_u16]_z(svbool_t pg, svuint16_t op1, svuint16_t op2,
                       uint16_t op3)
svuint32_t svmla[_n_u32]_z(svbool_t pg, svuint32_t op1, svuint32_t op2,
                       uint32_t op3)
svuint64_t svmla[_n_u64]_z(svbool_t pg, svuint64_t op1, svuint64_t op2,
                       uint64_t op3)

```

**6.7.10.5. MLA (vector, vector, scalar), merging with first input****Instances**

```

svint8_t svmla[_n_s8]_m(svbool_t pg, svint8_t op1, svint8_t op2,
                       int8_t op3)
svint16_t svmla[_n_s16]_m(svbool_t pg, svint16_t op1, svint16_t op2,

```

Instances	
<code>svint32_t</code>	<code>svmla[_n_s32]_m(svbool_t pg, svint32_t op1, svint32_t op2, int16_t op3)</code>
<code>svint64_t</code>	<code>svmla[_n_s64]_m(svbool_t pg, svint64_t op1, svint64_t op2, int32_t op3)</code>
<code>svuint8_t</code>	<code>svmla[_n_u8]_m(svbool_t pg, svuint8_t op1, svuint8_t op2, int64_t op3)</code>
<code>svuint16_t</code>	<code>svmla[_n_u16]_m(svbool_t pg, svuint16_t op1, svuint16_t op2, uint8_t op3)</code>
<code>svuint32_t</code>	<code>svmla[_n_u32]_m(svbool_t pg, svuint32_t op1, svuint32_t op2, uint16_t op3)</code>
<code>svuint64_t</code>	<code>svmla[_n_u64]_m(svbool_t pg, svuint64_t op1, svuint64_t op2, uint32_t op3)</code>

#### 6.7.10.6. MLA (vector, vector, scalar), setting inactive to unknown

Instances	
<code>svint8_t</code>	<code>svmla[_n_s8]_x(svbool_t pg, svint8_t op1, svint8_t op2, int8_t op3)</code>
<code>svint16_t</code>	<code>svmla[_n_s16]_x(svbool_t pg, svint16_t op1, svint16_t op2, int16_t op3)</code>
<code>svint32_t</code>	<code>svmla[_n_s32]_x(svbool_t pg, svint32_t op1, svint32_t op2, int32_t op3)</code>
<code>svint64_t</code>	<code>svmla[_n_s64]_x(svbool_t pg, svint64_t op1, svint64_t op2, int64_t op3)</code>
<code>svuint8_t</code>	<code>svmla[_n_u8]_x(svbool_t pg, svuint8_t op1, svuint8_t op2, uint8_t op3)</code>
<code>svuint16_t</code>	<code>svmla[_n_u16]_x(svbool_t pg, svuint16_t op1, svuint16_t op2, uint16_t op3)</code>
<code>svuint32_t</code>	<code>svmla[_n_u32]_x(svbool_t pg, svuint32_t op1, svuint32_t op2, uint32_t op3)</code>
<code>svuint64_t</code>	<code>svmla[_n_u64]_x(svbool_t pg, svuint64_t op1, svuint64_t op2, uint64_t op3)</code>

#### 6.7.11. MSB: Integer subtraction of product (multiplicand first)

These functions multiply the first two integer inputs and subtract the result from the third input. Both the multiplication and subtraction use modular arithmetic.

The result is well-defined for all inputs; there is no undefined behavior for signed overflow.

##### 6.7.11.1. MSB (vector, vector, vector), setting inactive to zero

Instances	
<code>svint8_t</code>	<code>svmsb[_s8]_z(svbool_t pg, svint8_t op1, svint8_t op2, svint8_t op3)</code>
<code>svint16_t</code>	<code>svmsb[_s16]_z(svbool_t pg, svint16_t op1, svint16_t op2, svint16_t op3)</code>
<code>svint32_t</code>	<code>svmsb[_s32]_z(svbool_t pg, svint32_t op1, svint32_t op2, svint32_t op3)</code>
<code>svint64_t</code>	<code>svmsb[_s64]_z(svbool_t pg, svint64_t op1, svint64_t op2, svint64_t op3)</code>
<code>svuint8_t</code>	<code>svmsb[_u8]_z(svbool_t pg, svuint8_t op1, svuint8_t op2, svuint8_t op3)</code>
<code>svuint16_t</code>	<code>svmsb[_u16]_z(svbool_t pg, svuint16_t op1, svuint16_t op2, svuint16_t op3)</code>

**Instances**

```

svuint16_t op3)
svuint32_t svmsb[_u32]_z(svbool_t pg, svuint32_t op1, svuint32_t op2,
svuint32_t op3)
svuint64_t svmsb[_u64]_z(svbool_t pg, svuint64_t op1, svuint64_t op2,
svuint64_t op3)

```

**6.7.11.2. MSB (vector, vector, vector), merging with first input****Instances**

```

svint8_t svmsb[_s8]_m(svbool_t pg, svint8_t op1, svint8_t op2,
svint8_t op3)
svint16_t svmsb[_s16]_m(svbool_t pg, svint16_t op1, svint16_t op2,
svint16_t op3)
svint32_t svmsb[_s32]_m(svbool_t pg, svint32_t op1, svint32_t op2,
svint32_t op3)
svint64_t svmsb[_s64]_m(svbool_t pg, svint64_t op1, svint64_t op2,
svint64_t op3)
svuint8_t svmsb[_u8]_m(svbool_t pg, svuint8_t op1, svuint8_t op2,
svuint8_t op3)
svuint16_t svmsb[_u16]_m(svbool_t pg, svuint16_t op1, svuint16_t op2,
svuint16_t op3)
svuint32_t svmsb[_u32]_m(svbool_t pg, svuint32_t op1, svuint32_t op2,
svuint32_t op3)
svuint64_t svmsb[_u64]_m(svbool_t pg, svuint64_t op1, svuint64_t op2,
svuint64_t op3)

```

**6.7.11.3. MSB (vector, vector, vector), setting inactive to unknown****Instances**

```

svint8_t svmsb[_s8]_x(svbool_t pg, svint8_t op1, svint8_t op2,
svint8_t op3)
svint16_t svmsb[_s16]_x(svbool_t pg, svint16_t op1, svint16_t op2,
svint16_t op3)
svint32_t svmsb[_s32]_x(svbool_t pg, svint32_t op1, svint32_t op2,
svint32_t op3)
svint64_t svmsb[_s64]_x(svbool_t pg, svint64_t op1, svint64_t op2,
svint64_t op3)
svuint8_t svmsb[_u8]_x(svbool_t pg, svuint8_t op1, svuint8_t op2,
svuint8_t op3)
svuint16_t svmsb[_u16]_x(svbool_t pg, svuint16_t op1, svuint16_t op2,
svuint16_t op3)
svuint32_t svmsb[_u32]_x(svbool_t pg, svuint32_t op1, svuint32_t op2,
svuint32_t op3)
svuint64_t svmsb[_u64]_x(svbool_t pg, svuint64_t op1, svuint64_t op2,
svuint64_t op3)

```

**6.7.11.4. MSB (vector, vector, scalar), setting inactive to zero****Instances**

```

svint8_t svmsb[_n_s8]_z(svbool_t pg, svint8_t op1, svint8_t op2,
int8_t op3)
svint16_t svmsb[_n_s16]_z(svbool_t pg, svint16_t op1, svint16_t op2,
int16_t op3)
svint32_t svmsb[_n_s32]_z(svbool_t pg, svint32_t op1, svint32_t op2,
int32_t op3)

```

**Instances**

```
svint64_t svmsb[_n_s64]_z(svbool_t pg, svint64_t op1, svint64_t op2,
                           int64_t op3)
svuint8_t svmsb[_n_u8]_z(svbool_t pg, svuint8_t op1, svuint8_t op2,
                          uint8_t op3)
svuint16_t svmsb[_n_u16]_z(svbool_t pg, svuint16_t op1, svuint16_t op2,
                            uint16_t op3)
svuint32_t svmsb[_n_u32]_z(svbool_t pg, svuint32_t op1, svuint32_t op2,
                            uint32_t op3)
svuint64_t svmsb[_n_u64]_z(svbool_t pg, svuint64_t op1, svuint64_t op2,
                            uint64_t op3)
```

**6.7.11.5. MSB (vector, vector, scalar), merging with first input****Instances**

```
svint8_t svmsb[_n_s8]_m(svbool_t pg, svint8_t op1, svint8_t op2,
                         int8_t op3)
svint16_t svmsb[_n_s16]_m(svbool_t pg, svint16_t op1, svint16_t op2,
                           int16_t op3)
svint32_t svmsb[_n_s32]_m(svbool_t pg, svint32_t op1, svint32_t op2,
                           int32_t op3)
svint64_t svmsb[_n_s64]_m(svbool_t pg, svint64_t op1, svint64_t op2,
                           int64_t op3)
svuint8_t svmsb[_n_u8]_m(svbool_t pg, svuint8_t op1, svuint8_t op2,
                          uint8_t op3)
svuint16_t svmsb[_n_u16]_m(svbool_t pg, svuint16_t op1, svuint16_t op2,
                            uint16_t op3)
svuint32_t svmsb[_n_u32]_m(svbool_t pg, svuint32_t op1, svuint32_t op2,
                            uint32_t op3)
svuint64_t svmsb[_n_u64]_m(svbool_t pg, svuint64_t op1, svuint64_t op2,
                            uint64_t op3)
```

**6.7.11.6. MSB (vector, vector, scalar), setting inactive to unknown****Instances**

```
svint8_t svmsb[_n_s8]_x(svbool_t pg, svint8_t op1, svint8_t op2,
                         int8_t op3)
svint16_t svmsb[_n_s16]_x(svbool_t pg, svint16_t op1, svint16_t op2,
                           int16_t op3)
svint32_t svmsb[_n_s32]_x(svbool_t pg, svint32_t op1, svint32_t op2,
                           int32_t op3)
svint64_t svmsb[_n_s64]_x(svbool_t pg, svint64_t op1, svint64_t op2,
                           int64_t op3)
svuint8_t svmsb[_n_u8]_x(svbool_t pg, svuint8_t op1, svuint8_t op2,
                          uint8_t op3)
svuint16_t svmsb[_n_u16]_x(svbool_t pg, svuint16_t op1, svuint16_t op2,
                            uint16_t op3)
svuint32_t svmsb[_n_u32]_x(svbool_t pg, svuint32_t op1, svuint32_t op2,
                            uint32_t op3)
svuint64_t svmsb[_n_u64]_x(svbool_t pg, svuint64_t op1, svuint64_t op2,
                            uint64_t op3)
```

**6.7.12. MLS: Integer subtraction of product (minuend first)**

These functions multiply the second and third integer inputs and subtract the result from the first input. Both the multiplication and subtraction use modular arithmetic.



The result is well-defined for all inputs; there is no undefined behavior for signed overflow.

### 6.7.12.1. MLS (vector, vector, vector), setting inactive to zero

Instances
svint8_t <b>svmls</b> [_s8]_z(svbool_t pg, svint8_t op1, svint8_t op2, svint8_t op3)
svint16_t <b>svmls</b> [_s16]_z(svbool_t pg, svint16_t op1, svint16_t op2, svint16_t op3)
svint32_t <b>svmls</b> [_s32]_z(svbool_t pg, svint32_t op1, svint32_t op2, svint32_t op3)
svint64_t <b>svmls</b> [_s64]_z(svbool_t pg, svint64_t op1, svint64_t op2, svint64_t op3)
svuint8_t <b>svmls</b> [_u8]_z(svbool_t pg, svuint8_t op1, svuint8_t op2, svuint8_t op3)
svuint16_t <b>svmls</b> [_u16]_z(svbool_t pg, svuint16_t op1, svuint16_t op2, svuint16_t op3)
svuint32_t <b>svmls</b> [_u32]_z(svbool_t pg, svuint32_t op1, svuint32_t op2, svuint32_t op3)
svuint64_t <b>svmls</b> [_u64]_z(svbool_t pg, svuint64_t op1, svuint64_t op2, svuint64_t op3)

### 6.7.12.2. MLS (vector, vector, vector), merging with first input

Instances
svint8_t <b>svmls</b> [_s8]_m(svbool_t pg, svint8_t op1, svint8_t op2, svint8_t op3)
svint16_t <b>svmls</b> [_s16]_m(svbool_t pg, svint16_t op1, svint16_t op2, svint16_t op3)
svint32_t <b>svmls</b> [_s32]_m(svbool_t pg, svint32_t op1, svint32_t op2, svint32_t op3)
svint64_t <b>svmls</b> [_s64]_m(svbool_t pg, svint64_t op1, svint64_t op2, svint64_t op3)
svuint8_t <b>svmls</b> [_u8]_m(svbool_t pg, svuint8_t op1, svuint8_t op2, svuint8_t op3)
svuint16_t <b>svmls</b> [_u16]_m(svbool_t pg, svuint16_t op1, svuint16_t op2, svuint16_t op3)
svuint32_t <b>svmls</b> [_u32]_m(svbool_t pg, svuint32_t op1, svuint32_t op2, svuint32_t op3)
svuint64_t <b>svmls</b> [_u64]_m(svbool_t pg, svuint64_t op1, svuint64_t op2, svuint64_t op3)

### 6.7.12.3. MLS (vector, vector, vector), setting inactive to unknown

Instances
svint8_t <b>svmls</b> [_s8]_x(svbool_t pg, svint8_t op1, svint8_t op2, svint8_t op3)
svint16_t <b>svmls</b> [_s16]_x(svbool_t pg, svint16_t op1, svint16_t op2, svint16_t op3)
svint32_t <b>svmls</b> [_s32]_x(svbool_t pg, svint32_t op1, svint32_t op2, svint32_t op3)
svint64_t <b>svmls</b> [_s64]_x(svbool_t pg, svint64_t op1, svint64_t op2, svint64_t op3)
svuint8_t <b>svmls</b> [_u8]_x(svbool_t pg, svuint8_t op1, svuint8_t op2, svuint8_t op3)
svuint16_t <b>svmls</b> [_u16]_x(svbool_t pg, svuint16_t op1, svuint16_t op2, svuint16_t op3)

Instances
<pre> svuint16_t op3) svuint32_t svmls[_u32]_x(svbool_t pg, svuint32_t op1, svuint32_t op2, svuint32_t op3) svuint64_t svmls[_u64]_x(svbool_t pg, svuint64_t op1, svuint64_t op2, svuint64_t op3) </pre>

#### 6.7.12.4. MLS (vector, vector, scalar), setting inactive to zero

Instances
<pre> svint8_t svmls[_n_s8]_z(svbool_t pg, svint8_t op1, svint8_t op2, int8_t op3) svint16_t svmls[_n_s16]_z(svbool_t pg, svint16_t op1, svint16_t op2, int16_t op3) svint32_t svmls[_n_s32]_z(svbool_t pg, svint32_t op1, svint32_t op2, int32_t op3) svint64_t svmls[_n_s64]_z(svbool_t pg, svint64_t op1, svint64_t op2, int64_t op3) svuint8_t svmls[_n_u8]_z(svbool_t pg, svuint8_t op1, svuint8_t op2, uint8_t op3) svuint16_t svmls[_n_u16]_z(svbool_t pg, svuint16_t op1, svuint16_t op2, uint16_t op3) svuint32_t svmls[_n_u32]_z(svbool_t pg, svuint32_t op1, svuint32_t op2, uint32_t op3) svuint64_t svmls[_n_u64]_z(svbool_t pg, svuint64_t op1, svuint64_t op2, uint64_t op3) </pre>

#### 6.7.12.5. MLS (vector, vector, scalar), merging with first input

Instances
<pre> svint8_t svmls[_n_s8]_m(svbool_t pg, svint8_t op1, svint8_t op2, int8_t op3) svint16_t svmls[_n_s16]_m(svbool_t pg, svint16_t op1, svint16_t op2, int16_t op3) svint32_t svmls[_n_s32]_m(svbool_t pg, svint32_t op1, svint32_t op2, int32_t op3) svint64_t svmls[_n_s64]_m(svbool_t pg, svint64_t op1, svint64_t op2, int64_t op3) svuint8_t svmls[_n_u8]_m(svbool_t pg, svuint8_t op1, svuint8_t op2, uint8_t op3) svuint16_t svmls[_n_u16]_m(svbool_t pg, svuint16_t op1, svuint16_t op2, uint16_t op3) svuint32_t svmls[_n_u32]_m(svbool_t pg, svuint32_t op1, svuint32_t op2, uint32_t op3) svuint64_t svmls[_n_u64]_m(svbool_t pg, svuint64_t op1, svuint64_t op2, uint64_t op3) </pre>

#### 6.7.12.6. MLS (vector, vector, scalar), setting inactive to unknown

Instances
<pre> svint8_t svmls[_n_s8]_x(svbool_t pg, svint8_t op1, svint8_t op2, int8_t op3) svint16_t svmls[_n_s16]_x(svbool_t pg, svint16_t op1, svint16_t op2, int16_t op3) svint32_t svmls[_n_s32]_x(svbool_t pg, svint32_t op1, svint32_t op2, </pre>

**Instances**

```

                                int32_t op3)
svint64_t svmls[_n_s64]_x(svbool_t pg, svint64_t op1, svint64_t op2,
                                int64_t op3)
svuint8_t svmls[_n_u8]_x(svbool_t pg, svuint8_t op1, svuint8_t op2,
                                uint8_t op3)
svuint16_t svmls[_n_u16]_x(svbool_t pg, svuint16_t op1, svuint16_t op2,
                                uint16_t op3)
svuint32_t svmls[_n_u32]_x(svbool_t pg, svuint32_t op1, svuint32_t op2,
                                uint32_t op3)
svuint64_t svmls[_n_u64]_x(svbool_t pg, svuint64_t op1, svuint64_t op2,
                                uint64_t op3)

```

**6.7.13. DOT: Integer addition of dot product**

These functions partition the second and third vector inputs into groups of four elements. They calculate the dot product of each group (without loss of precision) and then add each result to the overlapping element of the first vector input.

The `_lane` forms of the functions take one group of four elements in each 128-bit quadword of the third input and replicate it to fill the rest of the quadword. The `imm_index` parameter specifies the group of four elements within each quadword that the functions should replicate. This index must be an integer constant expression in the range  $[0, 128/N]$ , where  $N$  is the number of bits in each group.

The result is well-defined for all inputs; there is no undefined behavior if a signed addition overflows.

**6.7.13.1. DOT (vector, vector, vector)****Instances**

```

svint32_t svdot[_s32](svint32_t op1, svint8_t op2, svint8_t op3)
svint64_t svdot[_s64](svint64_t op1, svint16_t op2, svint16_t op3)
svuint32_t svdot[_u32](svuint32_t op1, svuint8_t op2, svuint8_t op3)
svuint64_t svdot[_u64](svuint64_t op1, svuint16_t op2, svuint16_t op3)

```

**6.7.13.2. DOT (vector, vector, scalar)****Instances**

```

svint32_t svdot[_n_s32](svint32_t op1, svint8_t op2, int8_t op3)
svint64_t svdot[_n_s64](svint64_t op1, svint16_t op2, int16_t op3)
svuint32_t svdot[_n_u32](svuint32_t op1, svuint8_t op2, uint8_t op3)
svuint64_t svdot[_n_u64](svuint64_t op1, svuint16_t op2, uint16_t op3)

```

**6.7.13.3. DOT (vector, vector, vector, lane)****Instances**

```

svint32_t svdot_lane[_s32](svint32_t op1, svint8_t op2, svint8_t op3,
                                uint64_t imm_index)
svint64_t svdot_lane[_s64](svint64_t op1, svint16_t op2, svint16_t op3,
                                uint64_t imm_index)
svuint32_t svdot_lane[_u32](svuint32_t op1, svuint8_t op2, svuint8_t op3,
                                uint64_t imm_index)
svuint64_t svdot_lane[_u64](svuint64_t op1, svuint16_t op2, svuint16_t op3,
                                uint64_t imm_index)

```

## 6.7.14. DIV: Integer division

These functions divide the first integer input by the second input, rounding the result towards zero. Dividing a value by zero gives a zero result. Dividing the minimum signed value by -1 gives the minimum signed value.

The result is well-defined for all inputs and the functions do not raise any exceptions.

### 6.7.14.1. DIV (vector, vector), setting inactive to zero

Instances
<pre>svint32_t svdiv[_s32]_z(svbool_t pg, svint32_t op1, svint32_t op2) svint64_t svdiv[_s64]_z(svbool_t pg, svint64_t op1, svint64_t op2) svuint32_t svdiv[_u32]_z(svbool_t pg, svuint32_t op1, svuint32_t op2) svuint64_t svdiv[_u64]_z(svbool_t pg, svuint64_t op1, svuint64_t op2)</pre>

### 6.7.14.2. DIV (vector, vector), merging with first input

Instances
<pre>svint32_t svdiv[_s32]_m(svbool_t pg, svint32_t op1, svint32_t op2) svint64_t svdiv[_s64]_m(svbool_t pg, svint64_t op1, svint64_t op2) svuint32_t svdiv[_u32]_m(svbool_t pg, svuint32_t op1, svuint32_t op2) svuint64_t svdiv[_u64]_m(svbool_t pg, svuint64_t op1, svuint64_t op2)</pre>

### 6.7.14.3. DIV (vector, vector), setting inactive to unknown

Instances
<pre>svint32_t svdiv[_s32]_x(svbool_t pg, svint32_t op1, svint32_t op2) svint64_t svdiv[_s64]_x(svbool_t pg, svint64_t op1, svint64_t op2) svuint32_t svdiv[_u32]_x(svbool_t pg, svuint32_t op1, svuint32_t op2) svuint64_t svdiv[_u64]_x(svbool_t pg, svuint64_t op1, svuint64_t op2)</pre>

### 6.7.14.4. DIV (vector, scalar), setting inactive to zero

Instances
<pre>svint32_t svdiv[_n_s32]_z(svbool_t pg, svint32_t op1, int32_t op2) svint64_t svdiv[_n_s64]_z(svbool_t pg, svint64_t op1, int64_t op2) svuint32_t svdiv[_n_u32]_z(svbool_t pg, svuint32_t op1, uint32_t op2) svuint64_t svdiv[_n_u64]_z(svbool_t pg, svuint64_t op1, uint64_t op2)</pre>

### 6.7.14.5. DIV (vector, scalar), merging with first input

Instances
<pre>svint32_t svdiv[_n_s32]_m(svbool_t pg, svint32_t op1, int32_t op2) svint64_t svdiv[_n_s64]_m(svbool_t pg, svint64_t op1, int64_t op2) svuint32_t svdiv[_n_u32]_m(svbool_t pg, svuint32_t op1, uint32_t op2) svuint64_t svdiv[_n_u64]_m(svbool_t pg, svuint64_t op1, uint64_t op2)</pre>

### 6.7.14.6. DIV (vector, scalar), setting inactive to unknown

Instances
<pre>svint32_t svdiv[_n_s32]_x(svbool_t pg, svint32_t op1, int32_t op2)</pre>

**Instances**

```
svint64_t sdiv[_n_s64]_x(svbool_t pg, svint64_t op1, int64_t op2)
svuint32_t sdiv[_n_u32]_x(svbool_t pg, svuint32_t op1, uint32_t op2)
svuint64_t sdiv[_n_u64]_x(svbool_t pg, svuint64_t op1, uint64_t op2)
```

**6.7.15. DIVR: Integer division, reversed**

These functions divide the second integer input by the first input, rounding the result towards zero. Dividing a value by zero gives a zero result. Dividing the minimum signed value by -1 gives the minimum signed value.

The result is well-defined for all inputs and the functions do not raise any exceptions.

**6.7.15.1. DIVR (vector, vector), setting inactive to zero****Instances**

```
svint32_t sdivr[_s32]_z(svbool_t pg, svint32_t op1, svint32_t op2)
svint64_t sdivr[_s64]_z(svbool_t pg, svint64_t op1, svint64_t op2)
svuint32_t sdivr[_u32]_z(svbool_t pg, svuint32_t op1, svuint32_t op2)
svuint64_t sdivr[_u64]_z(svbool_t pg, svuint64_t op1, svuint64_t op2)
```

**6.7.15.2. DIVR (vector, vector), merging with first input****Instances**

```
svint32_t sdivr[_s32]_m(svbool_t pg, svint32_t op1, svint32_t op2)
svint64_t sdivr[_s64]_m(svbool_t pg, svint64_t op1, svint64_t op2)
svuint32_t sdivr[_u32]_m(svbool_t pg, svuint32_t op1, svuint32_t op2)
svuint64_t sdivr[_u64]_m(svbool_t pg, svuint64_t op1, svuint64_t op2)
```

**6.7.15.3. DIVR (vector, vector), setting inactive to unknown****Instances**

```
svint32_t sdivr[_s32]_x(svbool_t pg, svint32_t op1, svint32_t op2)
svint64_t sdivr[_s64]_x(svbool_t pg, svint64_t op1, svint64_t op2)
svuint32_t sdivr[_u32]_x(svbool_t pg, svuint32_t op1, svuint32_t op2)
svuint64_t sdivr[_u64]_x(svbool_t pg, svuint64_t op1, svuint64_t op2)
```

**6.7.15.4. DIVR (vector, scalar), setting inactive to zero****Instances**

```
svint32_t sdivr[_n_s32]_z(svbool_t pg, svint32_t op1, int32_t op2)
svint64_t sdivr[_n_s64]_z(svbool_t pg, svint64_t op1, int64_t op2)
svuint32_t sdivr[_n_u32]_z(svbool_t pg, svuint32_t op1, uint32_t op2)
svuint64_t sdivr[_n_u64]_z(svbool_t pg, svuint64_t op1, uint64_t op2)
```

**6.7.15.5. DIVR (vector, scalar), merging with first input****Instances**

```
svint32_t sdivr[_n_s32]_m(svbool_t pg, svint32_t op1, int32_t op2)
svint64_t sdivr[_n_s64]_m(svbool_t pg, svint64_t op1, int64_t op2)
svuint32_t sdivr[_n_u32]_m(svbool_t pg, svuint32_t op1, uint32_t op2)
svuint64_t sdivr[_n_u64]_m(svbool_t pg, svuint64_t op1, uint64_t op2)
```

### 6.7.15.6. DIVR (vector, scalar), setting inactive to unknown

Instances
svint32_t <b>svdivr</b> [_n_s32]_x(svbool_t pg, svint32_t op1, int32_t op2)
svint64_t <b>svdivr</b> [_n_s64]_x(svbool_t pg, svint64_t op1, int64_t op2)
svuint32_t <b>svdivr</b> [_n_u32]_x(svbool_t pg, svuint32_t op1, uint32_t op2)
svuint64_t <b>svdivr</b> [_n_u64]_x(svbool_t pg, svuint64_t op1, uint64_t op2)

## 6.7.16. MAX: Integer maximum

These functions return the maximum of two integer inputs.

### 6.7.16.1. MAX (vector, vector), setting inactive to zero

Instances
svint8_t <b>svmax</b> [_s8]_z(svbool_t pg, svint8_t op1, svint8_t op2)
svint16_t <b>svmax</b> [_s16]_z(svbool_t pg, svint16_t op1, svint16_t op2)
svint32_t <b>svmax</b> [_s32]_z(svbool_t pg, svint32_t op1, svint32_t op2)
svint64_t <b>svmax</b> [_s64]_z(svbool_t pg, svint64_t op1, svint64_t op2)
svuint8_t <b>svmax</b> [_u8]_z(svbool_t pg, svuint8_t op1, svuint8_t op2)
svuint16_t <b>svmax</b> [_u16]_z(svbool_t pg, svuint16_t op1, svuint16_t op2)
svuint32_t <b>svmax</b> [_u32]_z(svbool_t pg, svuint32_t op1, svuint32_t op2)
svuint64_t <b>svmax</b> [_u64]_z(svbool_t pg, svuint64_t op1, svuint64_t op2)

### 6.7.16.2. MAX (vector, vector), merging with first input

Instances
svint8_t <b>svmax</b> [_s8]_m(svbool_t pg, svint8_t op1, svint8_t op2)
svint16_t <b>svmax</b> [_s16]_m(svbool_t pg, svint16_t op1, svint16_t op2)
svint32_t <b>svmax</b> [_s32]_m(svbool_t pg, svint32_t op1, svint32_t op2)
svint64_t <b>svmax</b> [_s64]_m(svbool_t pg, svint64_t op1, svint64_t op2)
svuint8_t <b>svmax</b> [_u8]_m(svbool_t pg, svuint8_t op1, svuint8_t op2)
svuint16_t <b>svmax</b> [_u16]_m(svbool_t pg, svuint16_t op1, svuint16_t op2)
svuint32_t <b>svmax</b> [_u32]_m(svbool_t pg, svuint32_t op1, svuint32_t op2)
svuint64_t <b>svmax</b> [_u64]_m(svbool_t pg, svuint64_t op1, svuint64_t op2)

### 6.7.16.3. MAX (vector, vector), setting inactive to unknown

Instances
svint8_t <b>svmax</b> [_s8]_x(svbool_t pg, svint8_t op1, svint8_t op2)
svint16_t <b>svmax</b> [_s16]_x(svbool_t pg, svint16_t op1, svint16_t op2)
svint32_t <b>svmax</b> [_s32]_x(svbool_t pg, svint32_t op1, svint32_t op2)
svint64_t <b>svmax</b> [_s64]_x(svbool_t pg, svint64_t op1, svint64_t op2)
svuint8_t <b>svmax</b> [_u8]_x(svbool_t pg, svuint8_t op1, svuint8_t op2)
svuint16_t <b>svmax</b> [_u16]_x(svbool_t pg, svuint16_t op1, svuint16_t op2)
svuint32_t <b>svmax</b> [_u32]_x(svbool_t pg, svuint32_t op1, svuint32_t op2)
svuint64_t <b>svmax</b> [_u64]_x(svbool_t pg, svuint64_t op1, svuint64_t op2)

### 6.7.16.4. MAX (vector, scalar), setting inactive to zero

Instances
svint8_t <b>svmax</b> [_n_s8]_z(svbool_t pg, svint8_t op1, int8_t op2)
svint16_t <b>svmax</b> [_n_s16]_z(svbool_t pg, svint16_t op1, int16_t op2)
svint32_t <b>svmax</b> [_n_s32]_z(svbool_t pg, svint32_t op1, int32_t op2)
svint64_t <b>svmax</b> [_n_s64]_z(svbool_t pg, svint64_t op1, int64_t op2)

**Instances**

```
svuint8_t svmax[_n_u8]_z(svbool_t pg, svuint8_t op1, uint8_t op2)
svuint16_t svmax[_n_u16]_z(svbool_t pg, svuint16_t op1, uint16_t op2)
svuint32_t svmax[_n_u32]_z(svbool_t pg, svuint32_t op1, uint32_t op2)
svuint64_t svmax[_n_u64]_z(svbool_t pg, svuint64_t op1, uint64_t op2)
```

**6.7.16.5. MAX (vector, scalar), merging with first input****Instances**

```
svint8_t svmax[_n_s8]_m(svbool_t pg, svint8_t op1, int8_t op2)
svint16_t svmax[_n_s16]_m(svbool_t pg, svint16_t op1, int16_t op2)
svint32_t svmax[_n_s32]_m(svbool_t pg, svint32_t op1, int32_t op2)
svint64_t svmax[_n_s64]_m(svbool_t pg, svint64_t op1, int64_t op2)
svuint8_t svmax[_n_u8]_m(svbool_t pg, svuint8_t op1, uint8_t op2)
svuint16_t svmax[_n_u16]_m(svbool_t pg, svuint16_t op1, uint16_t op2)
svuint32_t svmax[_n_u32]_m(svbool_t pg, svuint32_t op1, uint32_t op2)
svuint64_t svmax[_n_u64]_m(svbool_t pg, svuint64_t op1, uint64_t op2)
```

**6.7.16.6. MAX (vector, scalar), setting inactive to unknown****Instances**

```
svint8_t svmax[_n_s8]_x(svbool_t pg, svint8_t op1, int8_t op2)
svint16_t svmax[_n_s16]_x(svbool_t pg, svint16_t op1, int16_t op2)
svint32_t svmax[_n_s32]_x(svbool_t pg, svint32_t op1, int32_t op2)
svint64_t svmax[_n_s64]_x(svbool_t pg, svint64_t op1, int64_t op2)
svuint8_t svmax[_n_u8]_x(svbool_t pg, svuint8_t op1, uint8_t op2)
svuint16_t svmax[_n_u16]_x(svbool_t pg, svuint16_t op1, uint16_t op2)
svuint32_t svmax[_n_u32]_x(svbool_t pg, svuint32_t op1, uint32_t op2)
svuint64_t svmax[_n_u64]_x(svbool_t pg, svuint64_t op1, uint64_t op2)
```

**6.7.17. MIN: Integer minimum**

These functions return the minimum of two integer inputs.

**6.7.17.1. MIN (vector, vector), setting inactive to zero****Instances**

```
svint8_t svmin[_s8]_z(svbool_t pg, svint8_t op1, svint8_t op2)
svint16_t svmin[_s16]_z(svbool_t pg, svint16_t op1, svint16_t op2)
svint32_t svmin[_s32]_z(svbool_t pg, svint32_t op1, svint32_t op2)
svint64_t svmin[_s64]_z(svbool_t pg, svint64_t op1, svint64_t op2)
svuint8_t svmin[_u8]_z(svbool_t pg, svuint8_t op1, svuint8_t op2)
svuint16_t svmin[_u16]_z(svbool_t pg, svuint16_t op1, svuint16_t op2)
svuint32_t svmin[_u32]_z(svbool_t pg, svuint32_t op1, svuint32_t op2)
svuint64_t svmin[_u64]_z(svbool_t pg, svuint64_t op1, svuint64_t op2)
```

**6.7.17.2. MIN (vector, vector), merging with first input****Instances**

```
svint8_t svmin[_s8]_m(svbool_t pg, svint8_t op1, svint8_t op2)
svint16_t svmin[_s16]_m(svbool_t pg, svint16_t op1, svint16_t op2)
svint32_t svmin[_s32]_m(svbool_t pg, svint32_t op1, svint32_t op2)
svint64_t svmin[_s64]_m(svbool_t pg, svint64_t op1, svint64_t op2)
svuint8_t svmin[_u8]_m(svbool_t pg, svuint8_t op1, svuint8_t op2)
svuint16_t svmin[_u16]_m(svbool_t pg, svuint16_t op1, svuint16_t op2)
```

Instances
svuint32_t <b>svmin</b> [_u32]_m(svbool_t pg, svuint32_t op1, svuint32_t op2)
svuint64_t <b>svmin</b> [_u64]_m(svbool_t pg, svuint64_t op1, svuint64_t op2)

### 6.7.17.3. MIN (vector, vector), setting inactive to unknown

Instances
svint8_t <b>svmin</b> [_s8]_x(svbool_t pg, svint8_t op1, svint8_t op2)
svint16_t <b>svmin</b> [_s16]_x(svbool_t pg, svint16_t op1, svint16_t op2)
svint32_t <b>svmin</b> [_s32]_x(svbool_t pg, svint32_t op1, svint32_t op2)
svint64_t <b>svmin</b> [_s64]_x(svbool_t pg, svint64_t op1, svint64_t op2)
svuint8_t <b>svmin</b> [_u8]_x(svbool_t pg, svuint8_t op1, svuint8_t op2)
svuint16_t <b>svmin</b> [_u16]_x(svbool_t pg, svuint16_t op1, svuint16_t op2)
svuint32_t <b>svmin</b> [_u32]_x(svbool_t pg, svuint32_t op1, svuint32_t op2)
svuint64_t <b>svmin</b> [_u64]_x(svbool_t pg, svuint64_t op1, svuint64_t op2)

### 6.7.17.4. MIN (vector, scalar), setting inactive to zero

Instances
svint8_t <b>svmin</b> [_n_s8]_z(svbool_t pg, svint8_t op1, int8_t op2)
svint16_t <b>svmin</b> [_n_s16]_z(svbool_t pg, svint16_t op1, int16_t op2)
svint32_t <b>svmin</b> [_n_s32]_z(svbool_t pg, svint32_t op1, int32_t op2)
svint64_t <b>svmin</b> [_n_s64]_z(svbool_t pg, svint64_t op1, int64_t op2)
svuint8_t <b>svmin</b> [_n_u8]_z(svbool_t pg, svuint8_t op1, uint8_t op2)
svuint16_t <b>svmin</b> [_n_u16]_z(svbool_t pg, svuint16_t op1, uint16_t op2)
svuint32_t <b>svmin</b> [_n_u32]_z(svbool_t pg, svuint32_t op1, uint32_t op2)
svuint64_t <b>svmin</b> [_n_u64]_z(svbool_t pg, svuint64_t op1, uint64_t op2)

### 6.7.17.5. MIN (vector, scalar), merging with first input

Instances
svint8_t <b>svmin</b> [_n_s8]_m(svbool_t pg, svint8_t op1, int8_t op2)
svint16_t <b>svmin</b> [_n_s16]_m(svbool_t pg, svint16_t op1, int16_t op2)
svint32_t <b>svmin</b> [_n_s32]_m(svbool_t pg, svint32_t op1, int32_t op2)
svint64_t <b>svmin</b> [_n_s64]_m(svbool_t pg, svint64_t op1, int64_t op2)
svuint8_t <b>svmin</b> [_n_u8]_m(svbool_t pg, svuint8_t op1, uint8_t op2)
svuint16_t <b>svmin</b> [_n_u16]_m(svbool_t pg, svuint16_t op1, uint16_t op2)
svuint32_t <b>svmin</b> [_n_u32]_m(svbool_t pg, svuint32_t op1, uint32_t op2)
svuint64_t <b>svmin</b> [_n_u64]_m(svbool_t pg, svuint64_t op1, uint64_t op2)

### 6.7.17.6. MIN (vector, scalar), setting inactive to unknown

Instances
svint8_t <b>svmin</b> [_n_s8]_x(svbool_t pg, svint8_t op1, int8_t op2)
svint16_t <b>svmin</b> [_n_s16]_x(svbool_t pg, svint16_t op1, int16_t op2)
svint32_t <b>svmin</b> [_n_s32]_x(svbool_t pg, svint32_t op1, int32_t op2)
svint64_t <b>svmin</b> [_n_s64]_x(svbool_t pg, svint64_t op1, int64_t op2)
svuint8_t <b>svmin</b> [_n_u8]_x(svbool_t pg, svuint8_t op1, uint8_t op2)
svuint16_t <b>svmin</b> [_n_u16]_x(svbool_t pg, svuint16_t op1, uint16_t op2)
svuint32_t <b>svmin</b> [_n_u32]_x(svbool_t pg, svuint32_t op1, uint32_t op2)
svuint64_t <b>svmin</b> [_n_u64]_x(svbool_t pg, svuint64_t op1, uint64_t op2)

## 6.7.18. NEG: Integer negation

These functions negate an integer input. The negative of the minimum (signed) value is itself.



The result is well-defined for all inputs; unlike C, there is no undefined behavior for signed overflow.

### 6.7.18.1. NEG (vector), setting inactive to zero

Instances
svint8_t <b>svneg</b> [_s8]_z(svbool_t <i>pg</i> , svint8_t <i>op</i> )
svint16_t <b>svneg</b> [_s16]_z(svbool_t <i>pg</i> , svint16_t <i>op</i> )
svint32_t <b>svneg</b> [_s32]_z(svbool_t <i>pg</i> , svint32_t <i>op</i> )
svint64_t <b>svneg</b> [_s64]_z(svbool_t <i>pg</i> , svint64_t <i>op</i> )

### 6.7.18.2. NEG (vector), merging with separate vector

Instances
svint8_t <b>svneg</b> [_s8]_m(svint8_t <i>inactive</i> , svbool_t <i>pg</i> , svint8_t <i>op</i> )
svint16_t <b>svneg</b> [_s16]_m(svint16_t <i>inactive</i> , svbool_t <i>pg</i> , svint16_t <i>op</i> )
svint32_t <b>svneg</b> [_s32]_m(svint32_t <i>inactive</i> , svbool_t <i>pg</i> , svint32_t <i>op</i> )
svint64_t <b>svneg</b> [_s64]_m(svint64_t <i>inactive</i> , svbool_t <i>pg</i> , svint64_t <i>op</i> )

### 6.7.18.3. NEG (vector), setting inactive to unknown

Instances
svint8_t <b>svneg</b> [_s8]_x(svbool_t <i>pg</i> , svint8_t <i>op</i> )
svint16_t <b>svneg</b> [_s16]_x(svbool_t <i>pg</i> , svint16_t <i>op</i> )
svint32_t <b>svneg</b> [_s32]_x(svbool_t <i>pg</i> , svint32_t <i>op</i> )
svint64_t <b>svneg</b> [_s64]_x(svbool_t <i>pg</i> , svint64_t <i>op</i> )

## 6.7.19. ABS: Integer absolute

These functions compute the absolute value of a signed integer input. The absolute value of the minimum (signed) value is itself.

This operation is well-defined for all inputs.

### 6.7.19.1. ABS (vector), setting inactive to zero

Instances
svint8_t <b>svabs</b> [_s8]_z(svbool_t <i>pg</i> , svint8_t <i>op</i> )
svint16_t <b>svabs</b> [_s16]_z(svbool_t <i>pg</i> , svint16_t <i>op</i> )
svint32_t <b>svabs</b> [_s32]_z(svbool_t <i>pg</i> , svint32_t <i>op</i> )
svint64_t <b>svabs</b> [_s64]_z(svbool_t <i>pg</i> , svint64_t <i>op</i> )

### 6.7.19.2. ABS (vector), merging with separate vector

Instances
svint8_t <b>svabs</b> [_s8]_m(svint8_t <i>inactive</i> , svbool_t <i>pg</i> , svint8_t <i>op</i> )
svint16_t <b>svabs</b> [_s16]_m(svint16_t <i>inactive</i> , svbool_t <i>pg</i> , svint16_t <i>op</i> )
svint32_t <b>svabs</b> [_s32]_m(svint32_t <i>inactive</i> , svbool_t <i>pg</i> , svint32_t <i>op</i> )
svint64_t <b>svabs</b> [_s64]_m(svint64_t <i>inactive</i> , svbool_t <i>pg</i> , svint64_t <i>op</i> )

### 6.7.19.3. ABS (vector), setting inactive to unknown

Instances
svint8_t <b>svabs</b> [_s8]_x(svbool_t <i>pg</i> , svint8_t <i>op</i> )
svint16_t <b>svabs</b> [_s16]_x(svbool_t <i>pg</i> , svint16_t <i>op</i> )

**Instances**

```
svint32_t svabs[_s32]_x(svbool_t pg, svint32_t op)
svint64_t svabs[_s64]_x(svbool_t pg, svint64_t op)
```

## 6.8. Logical operations

### 6.8.1. AND: Bitwise AND

These functions perform a bitwise AND of two integer inputs.

#### 6.8.1.1. AND (vector, vector), setting inactive to zero

**Instances**

```
svint8_t svand[_s8]_z(svbool_t pg, svint8_t op1, svint8_t op2)
svint16_t svand[_s16]_z(svbool_t pg, svint16_t op1, svint16_t op2)
svint32_t svand[_s32]_z(svbool_t pg, svint32_t op1, svint32_t op2)
svint64_t svand[_s64]_z(svbool_t pg, svint64_t op1, svint64_t op2)
svuint8_t svand[_u8]_z(svbool_t pg, svuint8_t op1, svuint8_t op2)
svuint16_t svand[_u16]_z(svbool_t pg, svuint16_t op1, svuint16_t op2)
svuint32_t svand[_u32]_z(svbool_t pg, svuint32_t op1, svuint32_t op2)
svuint64_t svand[_u64]_z(svbool_t pg, svuint64_t op1, svuint64_t op2)
```

#### 6.8.1.2. AND (vector, vector), merging with first input

**Instances**

```
svint8_t svand[_s8]_m(svbool_t pg, svint8_t op1, svint8_t op2)
svint16_t svand[_s16]_m(svbool_t pg, svint16_t op1, svint16_t op2)
svint32_t svand[_s32]_m(svbool_t pg, svint32_t op1, svint32_t op2)
svint64_t svand[_s64]_m(svbool_t pg, svint64_t op1, svint64_t op2)
svuint8_t svand[_u8]_m(svbool_t pg, svuint8_t op1, svuint8_t op2)
svuint16_t svand[_u16]_m(svbool_t pg, svuint16_t op1, svuint16_t op2)
svuint32_t svand[_u32]_m(svbool_t pg, svuint32_t op1, svuint32_t op2)
svuint64_t svand[_u64]_m(svbool_t pg, svuint64_t op1, svuint64_t op2)
```

#### 6.8.1.3. AND (vector, vector), setting inactive to unknown

**Instances**

```
svint8_t svand[_s8]_x(svbool_t pg, svint8_t op1, svint8_t op2)
svint16_t svand[_s16]_x(svbool_t pg, svint16_t op1, svint16_t op2)
svint32_t svand[_s32]_x(svbool_t pg, svint32_t op1, svint32_t op2)
svint64_t svand[_s64]_x(svbool_t pg, svint64_t op1, svint64_t op2)
svuint8_t svand[_u8]_x(svbool_t pg, svuint8_t op1, svuint8_t op2)
svuint16_t svand[_u16]_x(svbool_t pg, svuint16_t op1, svuint16_t op2)
svuint32_t svand[_u32]_x(svbool_t pg, svuint32_t op1, svuint32_t op2)
svuint64_t svand[_u64]_x(svbool_t pg, svuint64_t op1, svuint64_t op2)
```

#### 6.8.1.4. AND (vector, scalar), setting inactive to zero

**Instances**

```
svint8_t svand[_n_s8]_z(svbool_t pg, svint8_t op1, int8_t op2)
svint16_t svand[_n_s16]_z(svbool_t pg, svint16_t op1, int16_t op2)
svint32_t svand[_n_s32]_z(svbool_t pg, svint32_t op1, int32_t op2)
svint64_t svand[_n_s64]_z(svbool_t pg, svint64_t op1, int64_t op2)
svuint8_t svand[_n_u8]_z(svbool_t pg, svuint8_t op1, uint8_t op2)
```

**Instances**

```
svuint16_t svand[_n_u16]_z(svbool_t pg, svuint16_t op1, uint16_t op2)
svuint32_t svand[_n_u32]_z(svbool_t pg, svuint32_t op1, uint32_t op2)
svuint64_t svand[_n_u64]_z(svbool_t pg, svuint64_t op1, uint64_t op2)
```

**6.8.1.5. AND (vector, scalar), merging with first input****Instances**

```
svint8_t svand[_n_s8]_m(svbool_t pg, svint8_t op1, int8_t op2)
svint16_t svand[_n_s16]_m(svbool_t pg, svint16_t op1, int16_t op2)
svint32_t svand[_n_s32]_m(svbool_t pg, svint32_t op1, int32_t op2)
svint64_t svand[_n_s64]_m(svbool_t pg, svint64_t op1, int64_t op2)
svuint8_t svand[_n_u8]_m(svbool_t pg, svuint8_t op1, uint8_t op2)
svuint16_t svand[_n_u16]_m(svbool_t pg, svuint16_t op1, uint16_t op2)
svuint32_t svand[_n_u32]_m(svbool_t pg, svuint32_t op1, uint32_t op2)
svuint64_t svand[_n_u64]_m(svbool_t pg, svuint64_t op1, uint64_t op2)
```

**6.8.1.6. AND (vector, scalar), setting inactive to unknown****Instances**

```
svint8_t svand[_n_s8]_x(svbool_t pg, svint8_t op1, int8_t op2)
svint16_t svand[_n_s16]_x(svbool_t pg, svint16_t op1, int16_t op2)
svint32_t svand[_n_s32]_x(svbool_t pg, svint32_t op1, int32_t op2)
svint64_t svand[_n_s64]_x(svbool_t pg, svint64_t op1, int64_t op2)
svuint8_t svand[_n_u8]_x(svbool_t pg, svuint8_t op1, uint8_t op2)
svuint16_t svand[_n_u16]_x(svbool_t pg, svuint16_t op1, uint16_t op2)
svuint32_t svand[_n_u32]_x(svbool_t pg, svuint32_t op1, uint32_t op2)
svuint64_t svand[_n_u64]_x(svbool_t pg, svuint64_t op1, uint64_t op2)
```

**6.8.2. BIC: Bitwise AND NOT**

These functions perform a bitwise AND NOT of two integer inputs; that is, they invert the second input and then AND it with the first.

**6.8.2.1. BIC (vector, vector), setting inactive to zero****Instances**

```
svint8_t svbic[_s8]_z(svbool_t pg, svint8_t op1, svint8_t op2)
svint16_t svbic[_s16]_z(svbool_t pg, svint16_t op1, svint16_t op2)
svint32_t svbic[_s32]_z(svbool_t pg, svint32_t op1, svint32_t op2)
svint64_t svbic[_s64]_z(svbool_t pg, svint64_t op1, svint64_t op2)
svuint8_t svbic[_u8]_z(svbool_t pg, svuint8_t op1, svuint8_t op2)
svuint16_t svbic[_u16]_z(svbool_t pg, svuint16_t op1, svuint16_t op2)
svuint32_t svbic[_u32]_z(svbool_t pg, svuint32_t op1, svuint32_t op2)
svuint64_t svbic[_u64]_z(svbool_t pg, svuint64_t op1, svuint64_t op2)
```

**6.8.2.2. BIC (vector, vector), merging with first input****Instances**

```
svint8_t svbic[_s8]_m(svbool_t pg, svint8_t op1, svint8_t op2)
svint16_t svbic[_s16]_m(svbool_t pg, svint16_t op1, svint16_t op2)
svint32_t svbic[_s32]_m(svbool_t pg, svint32_t op1, svint32_t op2)
svint64_t svbic[_s64]_m(svbool_t pg, svint64_t op1, svint64_t op2)
svuint8_t svbic[_u8]_m(svbool_t pg, svuint8_t op1, svuint8_t op2)
svuint16_t svbic[_u16]_m(svbool_t pg, svuint16_t op1, svuint16_t op2)
```

**Instances**

```
svuint32_t svbic[_u32]_m(svbool_t pg, svuint32_t op1, svuint32_t op2)
svuint64_t svbic[_u64]_m(svbool_t pg, svuint64_t op1, svuint64_t op2)
```

**6.8.2.3. BIC (vector, vector), setting inactive to unknown****Instances**

```
svint8_t svbic[_s8]_x(svbool_t pg, svint8_t op1, svint8_t op2)
svint16_t svbic[_s16]_x(svbool_t pg, svint16_t op1, svint16_t op2)
svint32_t svbic[_s32]_x(svbool_t pg, svint32_t op1, svint32_t op2)
svint64_t svbic[_s64]_x(svbool_t pg, svint64_t op1, svint64_t op2)
svuint8_t svbic[_u8]_x(svbool_t pg, svuint8_t op1, svuint8_t op2)
svuint16_t svbic[_u16]_x(svbool_t pg, svuint16_t op1, svuint16_t op2)
svuint32_t svbic[_u32]_x(svbool_t pg, svuint32_t op1, svuint32_t op2)
svuint64_t svbic[_u64]_x(svbool_t pg, svuint64_t op1, svuint64_t op2)
```

**6.8.2.4. BIC (vector, scalar), setting inactive to zero****Instances**

```
svint8_t svbic[_n_s8]_z(svbool_t pg, svint8_t op1, int8_t op2)
svint16_t svbic[_n_s16]_z(svbool_t pg, svint16_t op1, int16_t op2)
svint32_t svbic[_n_s32]_z(svbool_t pg, svint32_t op1, int32_t op2)
svint64_t svbic[_n_s64]_z(svbool_t pg, svint64_t op1, int64_t op2)
svuint8_t svbic[_n_u8]_z(svbool_t pg, svuint8_t op1, uint8_t op2)
svuint16_t svbic[_n_u16]_z(svbool_t pg, svuint16_t op1, uint16_t op2)
svuint32_t svbic[_n_u32]_z(svbool_t pg, svuint32_t op1, uint32_t op2)
svuint64_t svbic[_n_u64]_z(svbool_t pg, svuint64_t op1, uint64_t op2)
```

**6.8.2.5. BIC (vector, scalar), merging with first input****Instances**

```
svint8_t svbic[_n_s8]_m(svbool_t pg, svint8_t op1, int8_t op2)
svint16_t svbic[_n_s16]_m(svbool_t pg, svint16_t op1, int16_t op2)
svint32_t svbic[_n_s32]_m(svbool_t pg, svint32_t op1, int32_t op2)
svint64_t svbic[_n_s64]_m(svbool_t pg, svint64_t op1, int64_t op2)
svuint8_t svbic[_n_u8]_m(svbool_t pg, svuint8_t op1, uint8_t op2)
svuint16_t svbic[_n_u16]_m(svbool_t pg, svuint16_t op1, uint16_t op2)
svuint32_t svbic[_n_u32]_m(svbool_t pg, svuint32_t op1, uint32_t op2)
svuint64_t svbic[_n_u64]_m(svbool_t pg, svuint64_t op1, uint64_t op2)
```

**6.8.2.6. BIC (vector, scalar), setting inactive to unknown****Instances**

```
svint8_t svbic[_n_s8]_x(svbool_t pg, svint8_t op1, int8_t op2)
svint16_t svbic[_n_s16]_x(svbool_t pg, svint16_t op1, int16_t op2)
svint32_t svbic[_n_s32]_x(svbool_t pg, svint32_t op1, int32_t op2)
svint64_t svbic[_n_s64]_x(svbool_t pg, svint64_t op1, int64_t op2)
svuint8_t svbic[_n_u8]_x(svbool_t pg, svuint8_t op1, uint8_t op2)
svuint16_t svbic[_n_u16]_x(svbool_t pg, svuint16_t op1, uint16_t op2)
svuint32_t svbic[_n_u32]_x(svbool_t pg, svuint32_t op1, uint32_t op2)
svuint64_t svbic[_n_u64]_x(svbool_t pg, svuint64_t op1, uint64_t op2)
```

**6.8.3. ORR: Bitwise OR**

These functions perform a bitwise OR of two integer inputs.

### 6.8.3.1. ORR (vector, vector), setting inactive to zero

#### Instances

```
svint8_t  svorr[_s8]_z(svbool_t pg, svint8_t op1, svint8_t op2)
svint16_t svorr[_s16]_z(svbool_t pg, svint16_t op1, svint16_t op2)
svint32_t svorr[_s32]_z(svbool_t pg, svint32_t op1, svint32_t op2)
svint64_t svorr[_s64]_z(svbool_t pg, svint64_t op1, svint64_t op2)
svuint8_t svorr[_u8]_z(svbool_t pg, svuint8_t op1, svuint8_t op2)
svuint16_t svorr[_u16]_z(svbool_t pg, svuint16_t op1, svuint16_t op2)
svuint32_t svorr[_u32]_z(svbool_t pg, svuint32_t op1, svuint32_t op2)
svuint64_t svorr[_u64]_z(svbool_t pg, svuint64_t op1, svuint64_t op2)
```

### 6.8.3.2. ORR (vector, vector), merging with first input

#### Instances

```
svint8_t  svorr[_s8]_m(svbool_t pg, svint8_t op1, svint8_t op2)
svint16_t svorr[_s16]_m(svbool_t pg, svint16_t op1, svint16_t op2)
svint32_t svorr[_s32]_m(svbool_t pg, svint32_t op1, svint32_t op2)
svint64_t svorr[_s64]_m(svbool_t pg, svint64_t op1, svint64_t op2)
svuint8_t svorr[_u8]_m(svbool_t pg, svuint8_t op1, svuint8_t op2)
svuint16_t svorr[_u16]_m(svbool_t pg, svuint16_t op1, svuint16_t op2)
svuint32_t svorr[_u32]_m(svbool_t pg, svuint32_t op1, svuint32_t op2)
svuint64_t svorr[_u64]_m(svbool_t pg, svuint64_t op1, svuint64_t op2)
```

### 6.8.3.3. ORR (vector, vector), setting inactive to unknown

#### Instances

```
svint8_t  svorr[_s8]_x(svbool_t pg, svint8_t op1, svint8_t op2)
svint16_t svorr[_s16]_x(svbool_t pg, svint16_t op1, svint16_t op2)
svint32_t svorr[_s32]_x(svbool_t pg, svint32_t op1, svint32_t op2)
svint64_t svorr[_s64]_x(svbool_t pg, svint64_t op1, svint64_t op2)
svuint8_t svorr[_u8]_x(svbool_t pg, svuint8_t op1, svuint8_t op2)
svuint16_t svorr[_u16]_x(svbool_t pg, svuint16_t op1, svuint16_t op2)
svuint32_t svorr[_u32]_x(svbool_t pg, svuint32_t op1, svuint32_t op2)
svuint64_t svorr[_u64]_x(svbool_t pg, svuint64_t op1, svuint64_t op2)
```

### 6.8.3.4. ORR (vector, scalar), setting inactive to zero

#### Instances

```
svint8_t  svorr[_n_s8]_z(svbool_t pg, svint8_t op1, int8_t op2)
svint16_t svorr[_n_s16]_z(svbool_t pg, svint16_t op1, int16_t op2)
svint32_t svorr[_n_s32]_z(svbool_t pg, svint32_t op1, int32_t op2)
svint64_t svorr[_n_s64]_z(svbool_t pg, svint64_t op1, int64_t op2)
svuint8_t svorr[_n_u8]_z(svbool_t pg, svuint8_t op1, uint8_t op2)
svuint16_t svorr[_n_u16]_z(svbool_t pg, svuint16_t op1, uint16_t op2)
svuint32_t svorr[_n_u32]_z(svbool_t pg, svuint32_t op1, uint32_t op2)
svuint64_t svorr[_n_u64]_z(svbool_t pg, svuint64_t op1, uint64_t op2)
```

### 6.8.3.5. ORR (vector, scalar), merging with first input

#### Instances

```
svint8_t  svorr[_n_s8]_m(svbool_t pg, svint8_t op1, int8_t op2)
svint16_t svorr[_n_s16]_m(svbool_t pg, svint16_t op1, int16_t op2)
svint32_t svorr[_n_s32]_m(svbool_t pg, svint32_t op1, int32_t op2)
svint64_t svorr[_n_s64]_m(svbool_t pg, svint64_t op1, int64_t op2)
```

**Instances**

```
svuint8_t svorr[_n_u8]_m(svbool_t pg, svuint8_t op1, uint8_t op2)
svuint16_t svorr[_n_u16]_m(svbool_t pg, svuint16_t op1, uint16_t op2)
svuint32_t svorr[_n_u32]_m(svbool_t pg, svuint32_t op1, uint32_t op2)
svuint64_t svorr[_n_u64]_m(svbool_t pg, svuint64_t op1, uint64_t op2)
```

**6.8.3.6. ORR (vector, scalar), setting inactive to unknown****Instances**

```
svint8_t svorr[_n_s8]_x(svbool_t pg, svint8_t op1, int8_t op2)
svint16_t svorr[_n_s16]_x(svbool_t pg, svint16_t op1, int16_t op2)
svint32_t svorr[_n_s32]_x(svbool_t pg, svint32_t op1, int32_t op2)
svint64_t svorr[_n_s64]_x(svbool_t pg, svint64_t op1, int64_t op2)
svuint8_t svorr[_n_u8]_x(svbool_t pg, svuint8_t op1, uint8_t op2)
svuint16_t svorr[_n_u16]_x(svbool_t pg, svuint16_t op1, uint16_t op2)
svuint32_t svorr[_n_u32]_x(svbool_t pg, svuint32_t op1, uint32_t op2)
svuint64_t svorr[_n_u64]_x(svbool_t pg, svuint64_t op1, uint64_t op2)
```

**6.8.4. EOR: Bitwise exclusive OR**

These functions perform a bitwise exclusive OR of two integer inputs.

**6.8.4.1. EOR (vector, vector), setting inactive to zero****Instances**

```
svint8_t sveor[_s8]_z(svbool_t pg, svint8_t op1, svint8_t op2)
svint16_t sveor[_s16]_z(svbool_t pg, svint16_t op1, svint16_t op2)
svint32_t sveor[_s32]_z(svbool_t pg, svint32_t op1, svint32_t op2)
svint64_t sveor[_s64]_z(svbool_t pg, svint64_t op1, svint64_t op2)
svuint8_t sveor[_u8]_z(svbool_t pg, svuint8_t op1, svuint8_t op2)
svuint16_t sveor[_u16]_z(svbool_t pg, svuint16_t op1, svuint16_t op2)
svuint32_t sveor[_u32]_z(svbool_t pg, svuint32_t op1, svuint32_t op2)
svuint64_t sveor[_u64]_z(svbool_t pg, svuint64_t op1, svuint64_t op2)
```

**6.8.4.2. EOR (vector, vector), merging with first input****Instances**

```
svint8_t sveor[_s8]_m(svbool_t pg, svint8_t op1, svint8_t op2)
svint16_t sveor[_s16]_m(svbool_t pg, svint16_t op1, svint16_t op2)
svint32_t sveor[_s32]_m(svbool_t pg, svint32_t op1, svint32_t op2)
svint64_t sveor[_s64]_m(svbool_t pg, svint64_t op1, svint64_t op2)
svuint8_t sveor[_u8]_m(svbool_t pg, svuint8_t op1, svuint8_t op2)
svuint16_t sveor[_u16]_m(svbool_t pg, svuint16_t op1, svuint16_t op2)
svuint32_t sveor[_u32]_m(svbool_t pg, svuint32_t op1, svuint32_t op2)
svuint64_t sveor[_u64]_m(svbool_t pg, svuint64_t op1, svuint64_t op2)
```

**6.8.4.3. EOR (vector, vector), setting inactive to unknown****Instances**

```
svint8_t sveor[_s8]_x(svbool_t pg, svint8_t op1, svint8_t op2)
svint16_t sveor[_s16]_x(svbool_t pg, svint16_t op1, svint16_t op2)
svint32_t sveor[_s32]_x(svbool_t pg, svint32_t op1, svint32_t op2)
svint64_t sveor[_s64]_x(svbool_t pg, svint64_t op1, svint64_t op2)
svuint8_t sveor[_u8]_x(svbool_t pg, svuint8_t op1, svuint8_t op2)
```

**Instances**

```
svuint16_t sveor[_u16]_x(svbool_t pg, svuint16_t op1, svuint16_t op2)
svuint32_t sveor[_u32]_x(svbool_t pg, svuint32_t op1, svuint32_t op2)
svuint64_t sveor[_u64]_x(svbool_t pg, svuint64_t op1, svuint64_t op2)
```

**6.8.4.4. EOR (vector, scalar), setting inactive to zero****Instances**

```
svint8_t sveor[_n_s8]_z(svbool_t pg, svint8_t op1, int8_t op2)
svint16_t sveor[_n_s16]_z(svbool_t pg, svint16_t op1, int16_t op2)
svint32_t sveor[_n_s32]_z(svbool_t pg, svint32_t op1, int32_t op2)
svint64_t sveor[_n_s64]_z(svbool_t pg, svint64_t op1, int64_t op2)
svuint8_t sveor[_n_u8]_z(svbool_t pg, svuint8_t op1, uint8_t op2)
svuint16_t sveor[_n_u16]_z(svbool_t pg, svuint16_t op1, uint16_t op2)
svuint32_t sveor[_n_u32]_z(svbool_t pg, svuint32_t op1, uint32_t op2)
svuint64_t sveor[_n_u64]_z(svbool_t pg, svuint64_t op1, uint64_t op2)
```

**6.8.4.5. EOR (vector, scalar), merging with first input****Instances**

```
svint8_t sveor[_n_s8]_m(svbool_t pg, svint8_t op1, int8_t op2)
svint16_t sveor[_n_s16]_m(svbool_t pg, svint16_t op1, int16_t op2)
svint32_t sveor[_n_s32]_m(svbool_t pg, svint32_t op1, int32_t op2)
svint64_t sveor[_n_s64]_m(svbool_t pg, svint64_t op1, int64_t op2)
svuint8_t sveor[_n_u8]_m(svbool_t pg, svuint8_t op1, uint8_t op2)
svuint16_t sveor[_n_u16]_m(svbool_t pg, svuint16_t op1, uint16_t op2)
svuint32_t sveor[_n_u32]_m(svbool_t pg, svuint32_t op1, uint32_t op2)
svuint64_t sveor[_n_u64]_m(svbool_t pg, svuint64_t op1, uint64_t op2)
```

**6.8.4.6. EOR (vector, scalar), setting inactive to unknown****Instances**

```
svint8_t sveor[_n_s8]_x(svbool_t pg, svint8_t op1, int8_t op2)
svint16_t sveor[_n_s16]_x(svbool_t pg, svint16_t op1, int16_t op2)
svint32_t sveor[_n_s32]_x(svbool_t pg, svint32_t op1, int32_t op2)
svint64_t sveor[_n_s64]_x(svbool_t pg, svint64_t op1, int64_t op2)
svuint8_t sveor[_n_u8]_x(svbool_t pg, svuint8_t op1, uint8_t op2)
svuint16_t sveor[_n_u16]_x(svbool_t pg, svuint16_t op1, uint16_t op2)
svuint32_t sveor[_n_u32]_x(svbool_t pg, svuint32_t op1, uint32_t op2)
svuint64_t sveor[_n_u64]_x(svbool_t pg, svuint64_t op1, uint64_t op2)
```

**6.8.5. NOT: Bitwise inverse**

These functions compute the bitwise inverse of an integer input; that is, they invert each bit individually.

**6.8.5.1. NOT (vector), setting inactive to zero****Instances**

```
svint8_t svnot[_s8]_z(svbool_t pg, svint8_t op)
svint16_t svnot[_s16]_z(svbool_t pg, svint16_t op)
svint32_t svnot[_s32]_z(svbool_t pg, svint32_t op)
svint64_t svnot[_s64]_z(svbool_t pg, svint64_t op)
svuint8_t svnot[_u8]_z(svbool_t pg, svuint8_t op)
svuint16_t svnot[_u16]_z(svbool_t pg, svuint16_t op)
svuint32_t svnot[_u32]_z(svbool_t pg, svuint32_t op)
```

Instances
<code>svuint64_t svnot[_u64]_z(svbool_t pg, svuint64_t op)</code>

### 6.8.5.2. NOT (vector), merging with separate vector

Instances
<code>svint8_t svnot[_s8]_m(svint8_t inactive, svbool_t pg, svint8_t op)</code>
<code>svint16_t svnot[_s16]_m(svint16_t inactive, svbool_t pg, svint16_t op)</code>
<code>svint32_t svnot[_s32]_m(svint32_t inactive, svbool_t pg, svint32_t op)</code>
<code>svint64_t svnot[_s64]_m(svint64_t inactive, svbool_t pg, svint64_t op)</code>
<code>svuint8_t svnot[_u8]_m(svuint8_t inactive, svbool_t pg, svuint8_t op)</code>
<code>svuint16_t svnot[_u16]_m(svuint16_t inactive, svbool_t pg, svuint16_t op)</code>
<code>svuint32_t svnot[_u32]_m(svuint32_t inactive, svbool_t pg, svuint32_t op)</code>
<code>svuint64_t svnot[_u64]_m(svuint64_t inactive, svbool_t pg, svuint64_t op)</code>

### 6.8.5.3. NOT (vector), setting inactive to unknown

Instances
<code>svint8_t svnot[_s8]_x(svbool_t pg, svint8_t op)</code>
<code>svint16_t svnot[_s16]_x(svbool_t pg, svint16_t op)</code>
<code>svint32_t svnot[_s32]_x(svbool_t pg, svint32_t op)</code>
<code>svint64_t svnot[_s64]_x(svbool_t pg, svint64_t op)</code>
<code>svuint8_t svnot[_u8]_x(svbool_t pg, svuint8_t op)</code>
<code>svuint16_t svnot[_u16]_x(svbool_t pg, svuint16_t op)</code>
<code>svuint32_t svnot[_u32]_x(svbool_t pg, svuint32_t op)</code>
<code>svuint64_t svnot[_u64]_x(svbool_t pg, svuint64_t op)</code>

## 6.8.6. CNOT: Logical inverse

These functions compute the logical inverse of an integer input, so that a zero input gives a result of one and all other inputs give a result of zero.

### 6.8.6.1. CNOT (vector), setting inactive to zero

Instances
<code>svint8_t svcnot[_s8]_z(svbool_t pg, svint8_t op)</code>
<code>svint16_t svcnot[_s16]_z(svbool_t pg, svint16_t op)</code>
<code>svint32_t svcnot[_s32]_z(svbool_t pg, svint32_t op)</code>
<code>svint64_t svcnot[_s64]_z(svbool_t pg, svint64_t op)</code>
<code>svuint8_t svcnot[_u8]_z(svbool_t pg, svuint8_t op)</code>
<code>svuint16_t svcnot[_u16]_z(svbool_t pg, svuint16_t op)</code>
<code>svuint32_t svcnot[_u32]_z(svbool_t pg, svuint32_t op)</code>
<code>svuint64_t svcnot[_u64]_z(svbool_t pg, svuint64_t op)</code>

### 6.8.6.2. CNOT (vector), merging with separate vector

Instances
<code>svint8_t svcnot[_s8]_m(svint8_t inactive, svbool_t pg, svint8_t op)</code>
<code>svint16_t svcnot[_s16]_m(svint16_t inactive, svbool_t pg, svint16_t op)</code>
<code>svint32_t svcnot[_s32]_m(svint32_t inactive, svbool_t pg, svint32_t op)</code>
<code>svint64_t svcnot[_s64]_m(svint64_t inactive, svbool_t pg, svint64_t op)</code>
<code>svuint8_t svcnot[_u8]_m(svuint8_t inactive, svbool_t pg, svuint8_t op)</code>
<code>svuint16_t svcnot[_u16]_m(svuint16_t inactive, svbool_t pg, svuint16_t op)</code>
<code>svuint32_t svcnot[_u32]_m(svuint32_t inactive, svbool_t pg, svuint32_t op)</code>
<code>svuint64_t svcnot[_u64]_m(svuint64_t inactive, svbool_t pg, svuint64_t op)</code>



### 6.8.6.3. CNOT (vector), setting inactive to unknown

#### Instances

```
svint8_t  svcnot[_s8]_x(svbool_t pg, svint8_t op)
svint16_t svcnot[_s16]_x(svbool_t pg, svint16_t op)
svint32_t svcnot[_s32]_x(svbool_t pg, svint32_t op)
svint64_t svcnot[_s64]_x(svbool_t pg, svint64_t op)
svuint8_t svcnot[_u8]_x(svbool_t pg, svuint8_t op)
svuint16_t svcnot[_u16]_x(svbool_t pg, svuint16_t op)
svuint32_t svcnot[_u32]_x(svbool_t pg, svuint32_t op)
svuint64_t svcnot[_u64]_x(svbool_t pg, svuint64_t op)
```

## 6.9. Shifts

### 6.9.1. LSL: Shift left

These functions multiply the first integer input by two to the power of the second integer input and return the low bits of the result. Every bit of the second input is significant.

These functions are well-defined for all inputs.

#### 6.9.1.1. LSL (vector, vector), setting inactive to zero

#### Instances

```
svint8_t  svlsl[_s8]_z(svbool_t pg, svint8_t op1, svuint8_t op2)
svint16_t svlsl[_s16]_z(svbool_t pg, svint16_t op1, svuint16_t op2)
svint32_t svlsl[_s32]_z(svbool_t pg, svint32_t op1, svuint32_t op2)
svint64_t svlsl[_s64]_z(svbool_t pg, svint64_t op1, svuint64_t op2)
svuint8_t svlsl[_u8]_z(svbool_t pg, svuint8_t op1, svuint8_t op2)
svuint16_t svlsl[_u16]_z(svbool_t pg, svuint16_t op1, svuint16_t op2)
svuint32_t svlsl[_u32]_z(svbool_t pg, svuint32_t op1, svuint32_t op2)
svuint64_t svlsl[_u64]_z(svbool_t pg, svuint64_t op1, svuint64_t op2)
```

#### 6.9.1.2. LSL (vector, vector), merging with first input

#### Instances

```
svint8_t  svlsl[_s8]_m(svbool_t pg, svint8_t op1, svuint8_t op2)
svint16_t svlsl[_s16]_m(svbool_t pg, svint16_t op1, svuint16_t op2)
svint32_t svlsl[_s32]_m(svbool_t pg, svint32_t op1, svuint32_t op2)
svint64_t svlsl[_s64]_m(svbool_t pg, svint64_t op1, svuint64_t op2)
svuint8_t svlsl[_u8]_m(svbool_t pg, svuint8_t op1, svuint8_t op2)
svuint16_t svlsl[_u16]_m(svbool_t pg, svuint16_t op1, svuint16_t op2)
svuint32_t svlsl[_u32]_m(svbool_t pg, svuint32_t op1, svuint32_t op2)
svuint64_t svlsl[_u64]_m(svbool_t pg, svuint64_t op1, svuint64_t op2)
```

#### 6.9.1.3. LSL (vector, vector), setting inactive to unknown

#### Instances

```
svint8_t  svlsl[_s8]_x(svbool_t pg, svint8_t op1, svuint8_t op2)
svint16_t svlsl[_s16]_x(svbool_t pg, svint16_t op1, svuint16_t op2)
svint32_t svlsl[_s32]_x(svbool_t pg, svint32_t op1, svuint32_t op2)
svint64_t svlsl[_s64]_x(svbool_t pg, svint64_t op1, svuint64_t op2)
svuint8_t svlsl[_u8]_x(svbool_t pg, svuint8_t op1, svuint8_t op2)
svuint16_t svlsl[_u16]_x(svbool_t pg, svuint16_t op1, svuint16_t op2)
```

Instances
svuint32_t <b>svlsl</b> [_u32]_x(svbool_t pg, svuint32_t op1, svuint32_t op2)
svuint64_t <b>svlsl</b> [_u64]_x(svbool_t pg, svuint64_t op1, svuint64_t op2)

#### 6.9.1.4. LSL (vector, scalar), setting inactive to zero

Instances
svint8_t <b>svlsl</b> [_n_s8]_z(svbool_t pg, svint8_t op1, uint8_t op2)
svint16_t <b>svlsl</b> [_n_s16]_z(svbool_t pg, svint16_t op1, uint16_t op2)
svint32_t <b>svlsl</b> [_n_s32]_z(svbool_t pg, svint32_t op1, uint32_t op2)
svint64_t <b>svlsl</b> [_n_s64]_z(svbool_t pg, svint64_t op1, uint64_t op2)
svuint8_t <b>svlsl</b> [_n_u8]_z(svbool_t pg, svuint8_t op1, uint8_t op2)
svuint16_t <b>svlsl</b> [_n_u16]_z(svbool_t pg, svuint16_t op1, uint16_t op2)
svuint32_t <b>svlsl</b> [_n_u32]_z(svbool_t pg, svuint32_t op1, uint32_t op2)
svuint64_t <b>svlsl</b> [_n_u64]_z(svbool_t pg, svuint64_t op1, uint64_t op2)

#### 6.9.1.5. LSL (vector, scalar), merging with first input

Instances
svint8_t <b>svlsl</b> [_n_s8]_m(svbool_t pg, svint8_t op1, uint8_t op2)
svint16_t <b>svlsl</b> [_n_s16]_m(svbool_t pg, svint16_t op1, uint16_t op2)
svint32_t <b>svlsl</b> [_n_s32]_m(svbool_t pg, svint32_t op1, uint32_t op2)
svint64_t <b>svlsl</b> [_n_s64]_m(svbool_t pg, svint64_t op1, uint64_t op2)
svuint8_t <b>svlsl</b> [_n_u8]_m(svbool_t pg, svuint8_t op1, uint8_t op2)
svuint16_t <b>svlsl</b> [_n_u16]_m(svbool_t pg, svuint16_t op1, uint16_t op2)
svuint32_t <b>svlsl</b> [_n_u32]_m(svbool_t pg, svuint32_t op1, uint32_t op2)
svuint64_t <b>svlsl</b> [_n_u64]_m(svbool_t pg, svuint64_t op1, uint64_t op2)

#### 6.9.1.6. LSL (vector, scalar), setting inactive to unknown

Instances
svint8_t <b>svlsl</b> [_n_s8]_x(svbool_t pg, svint8_t op1, uint8_t op2)
svint16_t <b>svlsl</b> [_n_s16]_x(svbool_t pg, svint16_t op1, uint16_t op2)
svint32_t <b>svlsl</b> [_n_s32]_x(svbool_t pg, svint32_t op1, uint32_t op2)
svint64_t <b>svlsl</b> [_n_s64]_x(svbool_t pg, svint64_t op1, uint64_t op2)
svuint8_t <b>svlsl</b> [_n_u8]_x(svbool_t pg, svuint8_t op1, uint8_t op2)
svuint16_t <b>svlsl</b> [_n_u16]_x(svbool_t pg, svuint16_t op1, uint16_t op2)
svuint32_t <b>svlsl</b> [_n_u32]_x(svbool_t pg, svuint32_t op1, uint32_t op2)
svuint64_t <b>svlsl</b> [_n_u64]_x(svbool_t pg, svuint64_t op1, uint64_t op2)

#### 6.9.1.7. LSL (vector, wide vector), setting inactive to zero

Instances
svint8_t <b>svlsl_wide</b> [_s8]_z(svbool_t pg, svint8_t op1, svuint64_t op2)
svint16_t <b>svlsl_wide</b> [_s16]_z(svbool_t pg, svint16_t op1, svuint64_t op2)
svint32_t <b>svlsl_wide</b> [_s32]_z(svbool_t pg, svint32_t op1, svuint64_t op2)
svuint8_t <b>svlsl_wide</b> [_u8]_z(svbool_t pg, svuint8_t op1, svuint64_t op2)
svuint16_t <b>svlsl_wide</b> [_u16]_z(svbool_t pg, svuint16_t op1, svuint64_t op2)
svuint32_t <b>svlsl_wide</b> [_u32]_z(svbool_t pg, svuint32_t op1, svuint64_t op2)

#### 6.9.1.8. LSL (vector, wide vector), merging with first input

Instances
svint8_t <b>svlsl_wide</b> [_s8]_m(svbool_t pg, svint8_t op1, svuint64_t op2)

**Instances**

```
svint16_t svls1_wide[_s16]_m(svbool_t pg, svint16_t op1, svuint64_t op2)
svint32_t svls1_wide[_s32]_m(svbool_t pg, svint32_t op1, svuint64_t op2)
svuint8_t svls1_wide[_u8]_m(svbool_t pg, svuint8_t op1, svuint64_t op2)
svuint16_t svls1_wide[_u16]_m(svbool_t pg, svuint16_t op1, svuint64_t op2)
svuint32_t svls1_wide[_u32]_m(svbool_t pg, svuint32_t op1, svuint64_t op2)
```

**6.9.1.9. LSL (vector, wide vector), setting inactive to unknown****Instances**

```
svint8_t svls1_wide[_s8]_x(svbool_t pg, svint8_t op1, svuint64_t op2)
svint16_t svls1_wide[_s16]_x(svbool_t pg, svint16_t op1, svuint64_t op2)
svint32_t svls1_wide[_s32]_x(svbool_t pg, svint32_t op1, svuint64_t op2)
svuint8_t svls1_wide[_u8]_x(svbool_t pg, svuint8_t op1, svuint64_t op2)
svuint16_t svls1_wide[_u16]_x(svbool_t pg, svuint16_t op1, svuint64_t op2)
svuint32_t svls1_wide[_u32]_x(svbool_t pg, svuint32_t op1, svuint64_t op2)
```

**6.9.1.10. LSL (vector, wide scalar), setting inactive to zero****Instances**

```
svint8_t svls1_wide[_n_s8]_z(svbool_t pg, svint8_t op1, uint64_t op2)
svint16_t svls1_wide[_n_s16]_z(svbool_t pg, svint16_t op1, uint64_t op2)
svint32_t svls1_wide[_n_s32]_z(svbool_t pg, svint32_t op1, uint64_t op2)
svuint8_t svls1_wide[_n_u8]_z(svbool_t pg, svuint8_t op1, uint64_t op2)
svuint16_t svls1_wide[_n_u16]_z(svbool_t pg, svuint16_t op1, uint64_t op2)
svuint32_t svls1_wide[_n_u32]_z(svbool_t pg, svuint32_t op1, uint64_t op2)
```

**6.9.1.11. LSL (vector, wide scalar), merging with first input****Instances**

```
svint8_t svls1_wide[_n_s8]_m(svbool_t pg, svint8_t op1, uint64_t op2)
svint16_t svls1_wide[_n_s16]_m(svbool_t pg, svint16_t op1, uint64_t op2)
svint32_t svls1_wide[_n_s32]_m(svbool_t pg, svint32_t op1, uint64_t op2)
svuint8_t svls1_wide[_n_u8]_m(svbool_t pg, svuint8_t op1, uint64_t op2)
svuint16_t svls1_wide[_n_u16]_m(svbool_t pg, svuint16_t op1, uint64_t op2)
svuint32_t svls1_wide[_n_u32]_m(svbool_t pg, svuint32_t op1, uint64_t op2)
```

**6.9.1.12. LSL (vector, wide scalar), setting inactive to unknown****Instances**

```
svint8_t svls1_wide[_n_s8]_x(svbool_t pg, svint8_t op1, uint64_t op2)
svint16_t svls1_wide[_n_s16]_x(svbool_t pg, svint16_t op1, uint64_t op2)
svint32_t svls1_wide[_n_s32]_x(svbool_t pg, svint32_t op1, uint64_t op2)
svuint8_t svls1_wide[_n_u8]_x(svbool_t pg, svuint8_t op1, uint64_t op2)
svuint16_t svls1_wide[_n_u16]_x(svbool_t pg, svuint16_t op1, uint64_t op2)
svuint32_t svls1_wide[_n_u32]_x(svbool_t pg, svuint32_t op1, uint64_t op2)
```

**6.9.2. LSR: Logical shift right**

These functions divide the first integer input by two to the power of the second integer input and round the result towards zero (or, equivalently, towards -Inf). Every bit of the second input is significant.

These functions are well-defined for all inputs.

### 6.9.2.1. LSR (vector, vector), setting inactive to zero

Instances
svuint8_t <b>svlsr</b> [_u8]_z(svbool_t pg, svuint8_t op1, svuint8_t op2)
svuint16_t <b>svlsr</b> [_u16]_z(svbool_t pg, svuint16_t op1, svuint16_t op2)
svuint32_t <b>svlsr</b> [_u32]_z(svbool_t pg, svuint32_t op1, svuint32_t op2)
svuint64_t <b>svlsr</b> [_u64]_z(svbool_t pg, svuint64_t op1, svuint64_t op2)

### 6.9.2.2. LSR (vector, vector), merging with first input

Instances
svuint8_t <b>svlsr</b> [_u8]_m(svbool_t pg, svuint8_t op1, svuint8_t op2)
svuint16_t <b>svlsr</b> [_u16]_m(svbool_t pg, svuint16_t op1, svuint16_t op2)
svuint32_t <b>svlsr</b> [_u32]_m(svbool_t pg, svuint32_t op1, svuint32_t op2)
svuint64_t <b>svlsr</b> [_u64]_m(svbool_t pg, svuint64_t op1, svuint64_t op2)

### 6.9.2.3. LSR (vector, vector), setting inactive to unknown

Instances
svuint8_t <b>svlsr</b> [_u8]_x(svbool_t pg, svuint8_t op1, svuint8_t op2)
svuint16_t <b>svlsr</b> [_u16]_x(svbool_t pg, svuint16_t op1, svuint16_t op2)
svuint32_t <b>svlsr</b> [_u32]_x(svbool_t pg, svuint32_t op1, svuint32_t op2)
svuint64_t <b>svlsr</b> [_u64]_x(svbool_t pg, svuint64_t op1, svuint64_t op2)

### 6.9.2.4. LSR (vector, scalar), setting inactive to zero

Instances
svuint8_t <b>svlsr</b> [_n_u8]_z(svbool_t pg, svuint8_t op1, uint8_t op2)
svuint16_t <b>svlsr</b> [_n_u16]_z(svbool_t pg, svuint16_t op1, uint16_t op2)
svuint32_t <b>svlsr</b> [_n_u32]_z(svbool_t pg, svuint32_t op1, uint32_t op2)
svuint64_t <b>svlsr</b> [_n_u64]_z(svbool_t pg, svuint64_t op1, uint64_t op2)

### 6.9.2.5. LSR (vector, scalar), merging with first input

Instances
svuint8_t <b>svlsr</b> [_n_u8]_m(svbool_t pg, svuint8_t op1, uint8_t op2)
svuint16_t <b>svlsr</b> [_n_u16]_m(svbool_t pg, svuint16_t op1, uint16_t op2)
svuint32_t <b>svlsr</b> [_n_u32]_m(svbool_t pg, svuint32_t op1, uint32_t op2)
svuint64_t <b>svlsr</b> [_n_u64]_m(svbool_t pg, svuint64_t op1, uint64_t op2)

### 6.9.2.6. LSR (vector, scalar), setting inactive to unknown

Instances
svuint8_t <b>svlsr</b> [_n_u8]_x(svbool_t pg, svuint8_t op1, uint8_t op2)
svuint16_t <b>svlsr</b> [_n_u16]_x(svbool_t pg, svuint16_t op1, uint16_t op2)
svuint32_t <b>svlsr</b> [_n_u32]_x(svbool_t pg, svuint32_t op1, uint32_t op2)
svuint64_t <b>svlsr</b> [_n_u64]_x(svbool_t pg, svuint64_t op1, uint64_t op2)

### 6.9.2.7. LSR (vector, wide vector), setting inactive to zero

Instances
svuint8_t <b>svlsr_wide</b> [_u8]_z(svbool_t pg, svuint8_t op1, svuint64_t op2)

**Instances**

```
svuint16_t svlsr_wide[_u16]_z(svbool_t pg, svuint16_t op1, svuint64_t op2)
svuint32_t svlsr_wide[_u32]_z(svbool_t pg, svuint32_t op1, svuint64_t op2)
```

**6.9.2.8. LSR (vector, wide vector), merging with first input****Instances**

```
svuint8_t svlsr_wide[_u8]_m(svbool_t pg, svuint8_t op1, svuint64_t op2)
svuint16_t svlsr_wide[_u16]_m(svbool_t pg, svuint16_t op1, svuint64_t op2)
svuint32_t svlsr_wide[_u32]_m(svbool_t pg, svuint32_t op1, svuint64_t op2)
```

**6.9.2.9. LSR (vector, wide vector), setting inactive to unknown****Instances**

```
svuint8_t svlsr_wide[_u8]_x(svbool_t pg, svuint8_t op1, svuint64_t op2)
svuint16_t svlsr_wide[_u16]_x(svbool_t pg, svuint16_t op1, svuint64_t op2)
svuint32_t svlsr_wide[_u32]_x(svbool_t pg, svuint32_t op1, svuint64_t op2)
```

**6.9.2.10. LSR (vector, wide scalar), setting inactive to zero****Instances**

```
svuint8_t svlsr_wide[_n_u8]_z(svbool_t pg, svuint8_t op1, uint64_t op2)
svuint16_t svlsr_wide[_n_u16]_z(svbool_t pg, svuint16_t op1, uint64_t op2)
svuint32_t svlsr_wide[_n_u32]_z(svbool_t pg, svuint32_t op1, uint64_t op2)
```

**6.9.2.11. LSR (vector, wide scalar), merging with first input****Instances**

```
svuint8_t svlsr_wide[_n_u8]_m(svbool_t pg, svuint8_t op1, uint64_t op2)
svuint16_t svlsr_wide[_n_u16]_m(svbool_t pg, svuint16_t op1, uint64_t op2)
svuint32_t svlsr_wide[_n_u32]_m(svbool_t pg, svuint32_t op1, uint64_t op2)
```

**6.9.2.12. LSR (vector, wide scalar), setting inactive to unknown****Instances**

```
svuint8_t svlsr_wide[_n_u8]_x(svbool_t pg, svuint8_t op1, uint64_t op2)
svuint16_t svlsr_wide[_n_u16]_x(svbool_t pg, svuint16_t op1, uint64_t op2)
svuint32_t svlsr_wide[_n_u32]_x(svbool_t pg, svuint32_t op1, uint64_t op2)
```

**6.9.3. ASR: Arithmetic shift right, rounding towards -Inf**

These functions divide the first integer input by two to the power of the second integer input and round the result towards -Inf. Every bit of the second input is significant.

These functions are well-defined for all inputs.

**6.9.3.1. ASR (vector, vector), setting inactive to zero****Instances**

```
svint8_t svasr[_s8]_z(svbool_t pg, svint8_t op1, svuint8_t op2)
```

Instances	
svint16_t	<b>svasr</b> [_s16]_z(svbool_t pg, svint16_t op1, svuint16_t op2)
svint32_t	<b>svasr</b> [_s32]_z(svbool_t pg, svint32_t op1, svuint32_t op2)
svint64_t	<b>svasr</b> [_s64]_z(svbool_t pg, svint64_t op1, svuint64_t op2)

### 6.9.3.2. ASR (vector, vector), merging with first input

Instances	
svint8_t	<b>svasr</b> [_s8]_m(svbool_t pg, svint8_t op1, svuint8_t op2)
svint16_t	<b>svasr</b> [_s16]_m(svbool_t pg, svint16_t op1, svuint16_t op2)
svint32_t	<b>svasr</b> [_s32]_m(svbool_t pg, svint32_t op1, svuint32_t op2)
svint64_t	<b>svasr</b> [_s64]_m(svbool_t pg, svint64_t op1, svuint64_t op2)

### 6.9.3.3. ASR (vector, vector), setting inactive to unknown

Instances	
svint8_t	<b>svasr</b> [_s8]_x(svbool_t pg, svint8_t op1, svuint8_t op2)
svint16_t	<b>svasr</b> [_s16]_x(svbool_t pg, svint16_t op1, svuint16_t op2)
svint32_t	<b>svasr</b> [_s32]_x(svbool_t pg, svint32_t op1, svuint32_t op2)
svint64_t	<b>svasr</b> [_s64]_x(svbool_t pg, svint64_t op1, svuint64_t op2)

### 6.9.3.4. ASR (vector, scalar), setting inactive to zero

Instances	
svint8_t	<b>svasr</b> [_n_s8]_z(svbool_t pg, svint8_t op1, uint8_t op2)
svint16_t	<b>svasr</b> [_n_s16]_z(svbool_t pg, svint16_t op1, uint16_t op2)
svint32_t	<b>svasr</b> [_n_s32]_z(svbool_t pg, svint32_t op1, uint32_t op2)
svint64_t	<b>svasr</b> [_n_s64]_z(svbool_t pg, svint64_t op1, uint64_t op2)

### 6.9.3.5. ASR (vector, scalar), merging with first input

Instances	
svint8_t	<b>svasr</b> [_n_s8]_m(svbool_t pg, svint8_t op1, uint8_t op2)
svint16_t	<b>svasr</b> [_n_s16]_m(svbool_t pg, svint16_t op1, uint16_t op2)
svint32_t	<b>svasr</b> [_n_s32]_m(svbool_t pg, svint32_t op1, uint32_t op2)
svint64_t	<b>svasr</b> [_n_s64]_m(svbool_t pg, svint64_t op1, uint64_t op2)

### 6.9.3.6. ASR (vector, scalar), setting inactive to unknown

Instances	
svint8_t	<b>svasr</b> [_n_s8]_x(svbool_t pg, svint8_t op1, uint8_t op2)
svint16_t	<b>svasr</b> [_n_s16]_x(svbool_t pg, svint16_t op1, uint16_t op2)
svint32_t	<b>svasr</b> [_n_s32]_x(svbool_t pg, svint32_t op1, uint32_t op2)
svint64_t	<b>svasr</b> [_n_s64]_x(svbool_t pg, svint64_t op1, uint64_t op2)

### 6.9.3.7. ASR (vector, wide vector), setting inactive to zero

Instances	
svint8_t	<b>svasr_wide</b> [_s8]_z(svbool_t pg, svint8_t op1, svuint64_t op2)
svint16_t	<b>svasr_wide</b> [_s16]_z(svbool_t pg, svint16_t op1, svuint64_t op2)
svint32_t	<b>svasr_wide</b> [_s32]_z(svbool_t pg, svint32_t op1, svuint64_t op2)

### 6.9.3.8. ASR (vector, wide vector), merging with first input

Instances
svint8_t <b>svasr_wide</b> [_s8]_m(svbool_t pg, svint8_t op1, svuint64_t op2)
svint16_t <b>svasr_wide</b> [_s16]_m(svbool_t pg, svint16_t op1, svuint64_t op2)
svint32_t <b>svasr_wide</b> [_s32]_m(svbool_t pg, svint32_t op1, svuint64_t op2)

### 6.9.3.9. ASR (vector, wide vector), setting inactive to unknown

Instances
svint8_t <b>svasr_wide</b> [_s8]_x(svbool_t pg, svint8_t op1, svuint64_t op2)
svint16_t <b>svasr_wide</b> [_s16]_x(svbool_t pg, svint16_t op1, svuint64_t op2)
svint32_t <b>svasr_wide</b> [_s32]_x(svbool_t pg, svint32_t op1, svuint64_t op2)

### 6.9.3.10. ASR (vector, wide scalar), setting inactive to zero

Instances
svint8_t <b>svasr_wide</b> [_n_s8]_z(svbool_t pg, svint8_t op1, uint64_t op2)
svint16_t <b>svasr_wide</b> [_n_s16]_z(svbool_t pg, svint16_t op1, uint64_t op2)
svint32_t <b>svasr_wide</b> [_n_s32]_z(svbool_t pg, svint32_t op1, uint64_t op2)

### 6.9.3.11. ASR (vector, wide scalar), merging with first input

Instances
svint8_t <b>svasr_wide</b> [_n_s8]_m(svbool_t pg, svint8_t op1, uint64_t op2)
svint16_t <b>svasr_wide</b> [_n_s16]_m(svbool_t pg, svint16_t op1, uint64_t op2)
svint32_t <b>svasr_wide</b> [_n_s32]_m(svbool_t pg, svint32_t op1, uint64_t op2)

### 6.9.3.12. ASR (vector, wide scalar), setting inactive to unknown

Instances
svint8_t <b>svasr_wide</b> [_n_s8]_x(svbool_t pg, svint8_t op1, uint64_t op2)
svint16_t <b>svasr_wide</b> [_n_s16]_x(svbool_t pg, svint16_t op1, uint64_t op2)
svint32_t <b>svasr_wide</b> [_n_s32]_x(svbool_t pg, svint32_t op1, uint64_t op2)

## 6.9.4. ASRD: Arithmetic shift right, rounding towards zero

These functions divide the first integer input by two to the power of the second integer input and round the result towards zero. If the first input has  $N$  bits, the second input must be an integer constant expression in the range  $[1, N]$ .

These functions are well-defined for all inputs.

### 6.9.4.1. ASRD (vector, immediate), setting inactive to zero

Instances
svint8_t <b>svasrd</b> [_n_s8]_z(svbool_t pg, svint8_t op1, uint64_t imm2)
svint16_t <b>svasrd</b> [_n_s16]_z(svbool_t pg, svint16_t op1, uint64_t imm2)
svint32_t <b>svasrd</b> [_n_s32]_z(svbool_t pg, svint32_t op1, uint64_t imm2)
svint64_t <b>svasrd</b> [_n_s64]_z(svbool_t pg, svint64_t op1, uint64_t imm2)

### 6.9.4.2. ASRD (vector, immediate), merging with first input

Instances
svint8_t <b>svasrd</b> [_n_s8]_m(svbool_t pg, svint8_t op1, uint64_t imm2)
svint16_t <b>svasrd</b> [_n_s16]_m(svbool_t pg, svint16_t op1, uint64_t imm2)
svint32_t <b>svasrd</b> [_n_s32]_m(svbool_t pg, svint32_t op1, uint64_t imm2)
svint64_t <b>svasrd</b> [_n_s64]_m(svbool_t pg, svint64_t op1, uint64_t imm2)

### 6.9.4.3. ASRD (vector, immediate), setting inactive to unknown

Instances
svint8_t <b>svasrd</b> [_n_s8]_x(svbool_t pg, svint8_t op1, uint64_t imm2)
svint16_t <b>svasrd</b> [_n_s16]_x(svbool_t pg, svint16_t op1, uint64_t imm2)
svint32_t <b>svasrd</b> [_n_s32]_x(svbool_t pg, svint32_t op1, uint64_t imm2)
svint64_t <b>svasrd</b> [_n_s64]_x(svbool_t pg, svint64_t op1, uint64_t imm2)

## 6.9.5. INSR: Shift vector and insert scalar

These functions shift the first input left by one element and set the lowest element to the second input.

### 6.9.5.1. INSR (vector, scalar)

Instances
svint8_t <b>svinsr</b> [_n_s8](svint8_t op1, int8_t op2)
svint16_t <b>svinsr</b> [_n_s16](svint16_t op1, int16_t op2)
svint32_t <b>svinsr</b> [_n_s32](svint32_t op1, int32_t op2)
svint64_t <b>svinsr</b> [_n_s64](svint64_t op1, int64_t op2)
svuint8_t <b>svinsr</b> [_n_u8](svuint8_t op1, uint8_t op2)
svuint16_t <b>svinsr</b> [_n_u16](svuint16_t op1, uint16_t op2)
svuint32_t <b>svinsr</b> [_n_u32](svuint32_t op1, uint32_t op2)
svuint64_t <b>svinsr</b> [_n_u64](svuint64_t op1, uint64_t op2)
svfloat16_t <b>svinsr</b> [_n_f16](svfloat16_t op1, float16_t op2)
svfloat32_t <b>svinsr</b> [_n_f32](svfloat32_t op1, float32_t op2)
svfloat64_t <b>svinsr</b> [_n_f64](svfloat64_t op1, float64_t op2)
svbfloat16_t <b>svinsr</b> [_n_bf16](svbfloat16_t op1, bfloat16_t op2)

## 6.10. Integer reductions

### 6.10.1. ADDV: Integer addition reduction

These functions sum all active elements of an integer vector, without loss of precision, and return the low 64 bits of the result. The result is zero if no elements are active.

These functions are well-defined even if the addition overflows (which is only possible for 64-bit inputs).

#### 6.10.1.1. ADDV (vector)

Instances
int64_t <b>svaddv</b> [_s8](svbool_t pg, svint8_t op)
int64_t <b>svaddv</b> [_s16](svbool_t pg, svint16_t op)
int64_t <b>svaddv</b> [_s32](svbool_t pg, svint32_t op)
int64_t <b>svaddv</b> [_s64](svbool_t pg, svint64_t op)
uint64_t <b>svaddv</b> [_u8](svbool_t pg, svuint8_t op)



**Instances**

```
uint64_t svaddv[_u16](svbool_t pg, svuint16_t op)
uint64_t svaddv[_u32](svbool_t pg, svuint32_t op)
uint64_t svaddv[_u64](svbool_t pg, svuint64_t op)
```

## 6.10.2. MAXV: Integer maximum reduction

These functions return the maximum active element of an integer vector, or the minimum representable value if no elements are active.

### 6.10.2.1. MAXV (vector)

**Instances**

```
int8_t svmaxv[_s8](svbool_t pg, svint8_t op)
int16_t svmaxv[_s16](svbool_t pg, svint16_t op)
int32_t svmaxv[_s32](svbool_t pg, svint32_t op)
int64_t svmaxv[_s64](svbool_t pg, svint64_t op)
uint8_t svmaxv[_u8](svbool_t pg, svuint8_t op)
uint16_t svmaxv[_u16](svbool_t pg, svuint16_t op)
uint32_t svmaxv[_u32](svbool_t pg, svuint32_t op)
uint64_t svmaxv[_u64](svbool_t pg, svuint64_t op)
```

## 6.10.3. MINV: Integer minimum reduction

These functions return the minimum active element of an integer vector, or the maximum representable value if no elements are active.

### 6.10.3.1. MINV (vector)

**Instances**

```
int8_t svminv[_s8](svbool_t pg, svint8_t op)
int16_t svminv[_s16](svbool_t pg, svint16_t op)
int32_t svminv[_s32](svbool_t pg, svint32_t op)
int64_t svminv[_s64](svbool_t pg, svint64_t op)
uint8_t svminv[_u8](svbool_t pg, svuint8_t op)
uint16_t svminv[_u16](svbool_t pg, svuint16_t op)
uint32_t svminv[_u32](svbool_t pg, svuint32_t op)
uint64_t svminv[_u64](svbool_t pg, svuint64_t op)
```

## 6.10.4. ANDV: Integer AND reduction

These functions perform a bitwise AND of all active elements of an integer vector. The result is all-ones if no elements are active.

### 6.10.4.1. ANDV (vector)

**Instances**

```
int8_t svandv[_s8](svbool_t pg, svint8_t op)
int16_t svandv[_s16](svbool_t pg, svint16_t op)
int32_t svandv[_s32](svbool_t pg, svint32_t op)
int64_t svandv[_s64](svbool_t pg, svint64_t op)
uint8_t svandv[_u8](svbool_t pg, svuint8_t op)
uint16_t svandv[_u16](svbool_t pg, svuint16_t op)
uint32_t svandv[_u32](svbool_t pg, svuint32_t op)
```

**Instances**

```
uint64_t svandv[_u64](svbool_t pg, svuint64_t op)
```

## 6.10.5. ORV: Integer OR reduction

These functions perform a bitwise OR of all active elements of an integer vector. The result is zero if no elements are active.

### 6.10.5.1. ORV (vector)

**Instances**

```
int8_t svorv[_s8](svbool_t pg, svint8_t op)
int16_t svorv[_s16](svbool_t pg, svint16_t op)
int32_t svorv[_s32](svbool_t pg, svint32_t op)
int64_t svorv[_s64](svbool_t pg, svint64_t op)
uint8_t svorv[_u8](svbool_t pg, svuint8_t op)
uint16_t svorv[_u16](svbool_t pg, svuint16_t op)
uint32_t svorv[_u32](svbool_t pg, svuint32_t op)
uint64_t svorv[_u64](svbool_t pg, svuint64_t op)
```

## 6.10.6. EORV: Integer exclusive OR reduction

These functions perform a bitwise exclusive OR of all active elements of an integer vector. The result is zero if no elements are active.

### 6.10.6.1. EORV (vector)

**Instances**

```
int8_t sveorv[_s8](svbool_t pg, svint8_t op)
int16_t sveorv[_s16](svbool_t pg, svint16_t op)
int32_t sveorv[_s32](svbool_t pg, svint32_t op)
int64_t sveorv[_s64](svbool_t pg, svint64_t op)
uint8_t sveorv[_u8](svbool_t pg, svuint8_t op)
uint16_t sveorv[_u16](svbool_t pg, svuint16_t op)
uint32_t sveorv[_u32](svbool_t pg, svuint32_t op)
uint64_t sveorv[_u64](svbool_t pg, svuint64_t op)
```

## 6.11. Integer comparisons

### 6.11.1. CMPEQ: Integer compare equal

These functions compare two integer inputs and return a predicate bit that indicates whether the inputs are equal.

#### 6.11.1.1. CMPEQ (vector, vector)

**Instances**

```
svbool_t svcmpeq[_s8](svbool_t pg, svint8_t op1, svint8_t op2)
svbool_t svcmpeq[_s16](svbool_t pg, svint16_t op1, svint16_t op2)
svbool_t svcmpeq[_s32](svbool_t pg, svint32_t op1, svint32_t op2)
svbool_t svcmpeq[_s64](svbool_t pg, svint64_t op1, svint64_t op2)
svbool_t svcmpeq[_u8](svbool_t pg, svuint8_t op1, svuint8_t op2)
svbool_t svcmpeq[_u16](svbool_t pg, svuint16_t op1, svuint16_t op2)
svbool_t svcmpeq[_u32](svbool_t pg, svuint32_t op1, svuint32_t op2)
```

Instances
svbool_t <b>svcmpeq</b> [_u64](svbool_t <i>pg</i> , svuint64_t <i>op1</i> , svuint64_t <i>op2</i> )

### 6.11.1.2. CMPEQ (vector, scalar)

Instances
svbool_t <b>svcmpeq</b> [_n_s8](svbool_t <i>pg</i> , svint8_t <i>op1</i> , int8_t <i>op2</i> )
svbool_t <b>svcmpeq</b> [_n_s16](svbool_t <i>pg</i> , svint16_t <i>op1</i> , int16_t <i>op2</i> )
svbool_t <b>svcmpeq</b> [_n_s32](svbool_t <i>pg</i> , svint32_t <i>op1</i> , int32_t <i>op2</i> )
svbool_t <b>svcmpeq</b> [_n_s64](svbool_t <i>pg</i> , svint64_t <i>op1</i> , int64_t <i>op2</i> )
svbool_t <b>svcmpeq</b> [_n_u8](svbool_t <i>pg</i> , svuint8_t <i>op1</i> , uint8_t <i>op2</i> )
svbool_t <b>svcmpeq</b> [_n_u16](svbool_t <i>pg</i> , svuint16_t <i>op1</i> , uint16_t <i>op2</i> )
svbool_t <b>svcmpeq</b> [_n_u32](svbool_t <i>pg</i> , svuint32_t <i>op1</i> , uint32_t <i>op2</i> )
svbool_t <b>svcmpeq</b> [_n_u64](svbool_t <i>pg</i> , svuint64_t <i>op1</i> , uint64_t <i>op2</i> )

### 6.11.1.3. CMPEQ (vector, wide vector)

Instances
svbool_t <b>svcmpeq_wide</b> [_s8](svbool_t <i>pg</i> , svint8_t <i>op1</i> , svint64_t <i>op2</i> )
svbool_t <b>svcmpeq_wide</b> [_s16](svbool_t <i>pg</i> , svint16_t <i>op1</i> , svint64_t <i>op2</i> )
svbool_t <b>svcmpeq_wide</b> [_s32](svbool_t <i>pg</i> , svint32_t <i>op1</i> , svint64_t <i>op2</i> )

### 6.11.1.4. CMPEQ (vector, wide scalar)

Instances
svbool_t <b>svcmpeq_wide</b> [_n_s8](svbool_t <i>pg</i> , svint8_t <i>op1</i> , int64_t <i>op2</i> )
svbool_t <b>svcmpeq_wide</b> [_n_s16](svbool_t <i>pg</i> , svint16_t <i>op1</i> , int64_t <i>op2</i> )
svbool_t <b>svcmpeq_wide</b> [_n_s32](svbool_t <i>pg</i> , svint32_t <i>op1</i> , int64_t <i>op2</i> )

## 6.11.2. CMPNE: Integer compare not equal

These functions compare two integer inputs and return a predicate bit that indicates whether the inputs are not equal.

### 6.11.2.1. CMPNE (vector, vector)

Instances
svbool_t <b>svcmpne</b> [_s8](svbool_t <i>pg</i> , svint8_t <i>op1</i> , svint8_t <i>op2</i> )
svbool_t <b>svcmpne</b> [_s16](svbool_t <i>pg</i> , svint16_t <i>op1</i> , svint16_t <i>op2</i> )
svbool_t <b>svcmpne</b> [_s32](svbool_t <i>pg</i> , svint32_t <i>op1</i> , svint32_t <i>op2</i> )
svbool_t <b>svcmpne</b> [_s64](svbool_t <i>pg</i> , svint64_t <i>op1</i> , svint64_t <i>op2</i> )
svbool_t <b>svcmpne</b> [_u8](svbool_t <i>pg</i> , svuint8_t <i>op1</i> , svuint8_t <i>op2</i> )
svbool_t <b>svcmpne</b> [_u16](svbool_t <i>pg</i> , svuint16_t <i>op1</i> , svuint16_t <i>op2</i> )
svbool_t <b>svcmpne</b> [_u32](svbool_t <i>pg</i> , svuint32_t <i>op1</i> , svuint32_t <i>op2</i> )
svbool_t <b>svcmpne</b> [_u64](svbool_t <i>pg</i> , svuint64_t <i>op1</i> , svuint64_t <i>op2</i> )

### 6.11.2.2. CMPNE (vector, scalar)

Instances
svbool_t <b>svcmpne</b> [_n_s8](svbool_t <i>pg</i> , svint8_t <i>op1</i> , int8_t <i>op2</i> )
svbool_t <b>svcmpne</b> [_n_s16](svbool_t <i>pg</i> , svint16_t <i>op1</i> , int16_t <i>op2</i> )
svbool_t <b>svcmpne</b> [_n_s32](svbool_t <i>pg</i> , svint32_t <i>op1</i> , int32_t <i>op2</i> )
svbool_t <b>svcmpne</b> [_n_s64](svbool_t <i>pg</i> , svint64_t <i>op1</i> , int64_t <i>op2</i> )
svbool_t <b>svcmpne</b> [_n_u8](svbool_t <i>pg</i> , svuint8_t <i>op1</i> , uint8_t <i>op2</i> )

Instances	
svbool_t	<b>svcmpne</b> [_n_u16](svbool_t <i>pg</i> , svuint16_t <i>op1</i> , uint16_t <i>op2</i> )
svbool_t	<b>svcmpne</b> [_n_u32](svbool_t <i>pg</i> , svuint32_t <i>op1</i> , uint32_t <i>op2</i> )
svbool_t	<b>svcmpne</b> [_n_u64](svbool_t <i>pg</i> , svuint64_t <i>op1</i> , uint64_t <i>op2</i> )

### 6.11.2.3. CMPNE (vector, wide vector)

Instances	
svbool_t	<b>svcmpne_wide</b> [_s8](svbool_t <i>pg</i> , svint8_t <i>op1</i> , svint64_t <i>op2</i> )
svbool_t	<b>svcmpne_wide</b> [_s16](svbool_t <i>pg</i> , svint16_t <i>op1</i> , svint64_t <i>op2</i> )
svbool_t	<b>svcmpne_wide</b> [_s32](svbool_t <i>pg</i> , svint32_t <i>op1</i> , svint64_t <i>op2</i> )

### 6.11.2.4. CMPNE (vector, wide scalar)

Instances	
svbool_t	<b>svcmpne_wide</b> [_n_s8](svbool_t <i>pg</i> , svint8_t <i>op1</i> , int64_t <i>op2</i> )
svbool_t	<b>svcmpne_wide</b> [_n_s16](svbool_t <i>pg</i> , svint16_t <i>op1</i> , int64_t <i>op2</i> )
svbool_t	<b>svcmpne_wide</b> [_n_s32](svbool_t <i>pg</i> , svint32_t <i>op1</i> , int64_t <i>op2</i> )

## 6.11.3. CMPLT: Integer compare less than

These functions compare two integer inputs and return a predicate bit that indicates whether the first input is less than the second.

These functions handle both signed and unsigned inputs; there are no separate functions for CMPLO.

### 6.11.3.1. CMPLT (vector, vector)

Instances	
svbool_t	<b>svcmplt</b> [_s8](svbool_t <i>pg</i> , svint8_t <i>op1</i> , svint8_t <i>op2</i> )
svbool_t	<b>svcmplt</b> [_s16](svbool_t <i>pg</i> , svint16_t <i>op1</i> , svint16_t <i>op2</i> )
svbool_t	<b>svcmplt</b> [_s32](svbool_t <i>pg</i> , svint32_t <i>op1</i> , svint32_t <i>op2</i> )
svbool_t	<b>svcmplt</b> [_s64](svbool_t <i>pg</i> , svint64_t <i>op1</i> , svint64_t <i>op2</i> )
svbool_t	<b>svcmplt</b> [_u8](svbool_t <i>pg</i> , svuint8_t <i>op1</i> , svuint8_t <i>op2</i> )
svbool_t	<b>svcmplt</b> [_u16](svbool_t <i>pg</i> , svuint16_t <i>op1</i> , svuint16_t <i>op2</i> )
svbool_t	<b>svcmplt</b> [_u32](svbool_t <i>pg</i> , svuint32_t <i>op1</i> , svuint32_t <i>op2</i> )
svbool_t	<b>svcmplt</b> [_u64](svbool_t <i>pg</i> , svuint64_t <i>op1</i> , svuint64_t <i>op2</i> )

### 6.11.3.2. CMPLT (vector, scalar)

Instances	
svbool_t	<b>svcmplt</b> [_n_s8](svbool_t <i>pg</i> , svint8_t <i>op1</i> , int8_t <i>op2</i> )
svbool_t	<b>svcmplt</b> [_n_s16](svbool_t <i>pg</i> , svint16_t <i>op1</i> , int16_t <i>op2</i> )
svbool_t	<b>svcmplt</b> [_n_s32](svbool_t <i>pg</i> , svint32_t <i>op1</i> , int32_t <i>op2</i> )
svbool_t	<b>svcmplt</b> [_n_s64](svbool_t <i>pg</i> , svint64_t <i>op1</i> , int64_t <i>op2</i> )
svbool_t	<b>svcmplt</b> [_n_u8](svbool_t <i>pg</i> , svuint8_t <i>op1</i> , uint8_t <i>op2</i> )
svbool_t	<b>svcmplt</b> [_n_u16](svbool_t <i>pg</i> , svuint16_t <i>op1</i> , uint16_t <i>op2</i> )
svbool_t	<b>svcmplt</b> [_n_u32](svbool_t <i>pg</i> , svuint32_t <i>op1</i> , uint32_t <i>op2</i> )
svbool_t	<b>svcmplt</b> [_n_u64](svbool_t <i>pg</i> , svuint64_t <i>op1</i> , uint64_t <i>op2</i> )

### 6.11.3.3. CMPLT (vector, wide vector)

Instances	
svbool_t	<b>svcmplt_wide</b> [_s8](svbool_t <i>pg</i> , svint8_t <i>op1</i> , svint64_t <i>op2</i> )

Instances	
svbool_t	<b>svcmplt_wide</b> [_s16](svbool_t <i>pg</i> , svint16_t <i>op1</i> , svint64_t <i>op2</i> )
svbool_t	<b>svcmplt_wide</b> [_s32](svbool_t <i>pg</i> , svint32_t <i>op1</i> , svint64_t <i>op2</i> )
svbool_t	<b>svcmplt_wide</b> [_u8](svbool_t <i>pg</i> , svuint8_t <i>op1</i> , svuint64_t <i>op2</i> )
svbool_t	<b>svcmplt_wide</b> [_u16](svbool_t <i>pg</i> , svuint16_t <i>op1</i> , svuint64_t <i>op2</i> )
svbool_t	<b>svcmplt_wide</b> [_u32](svbool_t <i>pg</i> , svuint32_t <i>op1</i> , svuint64_t <i>op2</i> )

#### 6.11.3.4. CMPLT (vector, wide scalar)

Instances	
svbool_t	<b>svcmplt_wide</b> [_n_s8](svbool_t <i>pg</i> , svint8_t <i>op1</i> , int64_t <i>op2</i> )
svbool_t	<b>svcmplt_wide</b> [_n_s16](svbool_t <i>pg</i> , svint16_t <i>op1</i> , int64_t <i>op2</i> )
svbool_t	<b>svcmplt_wide</b> [_n_s32](svbool_t <i>pg</i> , svint32_t <i>op1</i> , int64_t <i>op2</i> )
svbool_t	<b>svcmplt_wide</b> [_n_u8](svbool_t <i>pg</i> , svuint8_t <i>op1</i> , uint64_t <i>op2</i> )
svbool_t	<b>svcmplt_wide</b> [_n_u16](svbool_t <i>pg</i> , svuint16_t <i>op1</i> , uint64_t <i>op2</i> )
svbool_t	<b>svcmplt_wide</b> [_n_u32](svbool_t <i>pg</i> , svuint32_t <i>op1</i> , uint64_t <i>op2</i> )

#### 6.11.4. CMPLE: Integer compare less than or equal to

These functions compare two integer inputs and return a predicate bit that indicates whether the first input is less than or equal to the second.

These functions handle both signed and unsigned inputs; there are no separate functions for CMPLS.

##### 6.11.4.1. CMPLE (vector, vector)

Instances	
svbool_t	<b>svcmple</b> [_s8](svbool_t <i>pg</i> , svint8_t <i>op1</i> , svint8_t <i>op2</i> )
svbool_t	<b>svcmple</b> [_s16](svbool_t <i>pg</i> , svint16_t <i>op1</i> , svint16_t <i>op2</i> )
svbool_t	<b>svcmple</b> [_s32](svbool_t <i>pg</i> , svint32_t <i>op1</i> , svint32_t <i>op2</i> )
svbool_t	<b>svcmple</b> [_s64](svbool_t <i>pg</i> , svint64_t <i>op1</i> , svint64_t <i>op2</i> )
svbool_t	<b>svcmple</b> [_u8](svbool_t <i>pg</i> , svuint8_t <i>op1</i> , svuint8_t <i>op2</i> )
svbool_t	<b>svcmple</b> [_u16](svbool_t <i>pg</i> , svuint16_t <i>op1</i> , svuint16_t <i>op2</i> )
svbool_t	<b>svcmple</b> [_u32](svbool_t <i>pg</i> , svuint32_t <i>op1</i> , svuint32_t <i>op2</i> )
svbool_t	<b>svcmple</b> [_u64](svbool_t <i>pg</i> , svuint64_t <i>op1</i> , svuint64_t <i>op2</i> )

##### 6.11.4.2. CMPLE (vector, scalar)

Instances	
svbool_t	<b>svcmple</b> [_n_s8](svbool_t <i>pg</i> , svint8_t <i>op1</i> , int8_t <i>op2</i> )
svbool_t	<b>svcmple</b> [_n_s16](svbool_t <i>pg</i> , svint16_t <i>op1</i> , int16_t <i>op2</i> )
svbool_t	<b>svcmple</b> [_n_s32](svbool_t <i>pg</i> , svint32_t <i>op1</i> , int32_t <i>op2</i> )
svbool_t	<b>svcmple</b> [_n_s64](svbool_t <i>pg</i> , svint64_t <i>op1</i> , int64_t <i>op2</i> )
svbool_t	<b>svcmple</b> [_n_u8](svbool_t <i>pg</i> , svuint8_t <i>op1</i> , uint8_t <i>op2</i> )
svbool_t	<b>svcmple</b> [_n_u16](svbool_t <i>pg</i> , svuint16_t <i>op1</i> , uint16_t <i>op2</i> )
svbool_t	<b>svcmple</b> [_n_u32](svbool_t <i>pg</i> , svuint32_t <i>op1</i> , uint32_t <i>op2</i> )
svbool_t	<b>svcmple</b> [_n_u64](svbool_t <i>pg</i> , svuint64_t <i>op1</i> , uint64_t <i>op2</i> )

##### 6.11.4.3. CMPLE (vector, wide vector)

Instances	
svbool_t	<b>svcmple_wide</b> [_s8](svbool_t <i>pg</i> , svint8_t <i>op1</i> , svint64_t <i>op2</i> )
svbool_t	<b>svcmple_wide</b> [_s16](svbool_t <i>pg</i> , svint16_t <i>op1</i> , svint64_t <i>op2</i> )
svbool_t	<b>svcmple_wide</b> [_s32](svbool_t <i>pg</i> , svint32_t <i>op1</i> , svint64_t <i>op2</i> )

Instances	
svbool_t	<b>svcmple_wide</b> [_u8](svbool_t <i>pg</i> , svuint8_t <i>op1</i> , svuint64_t <i>op2</i> )
svbool_t	<b>svcmple_wide</b> [_u16](svbool_t <i>pg</i> , svuint16_t <i>op1</i> , svuint64_t <i>op2</i> )
svbool_t	<b>svcmple_wide</b> [_u32](svbool_t <i>pg</i> , svuint32_t <i>op1</i> , svuint64_t <i>op2</i> )

#### 6.11.4.4. CMPLE (vector, wide scalar)

Instances	
svbool_t	<b>svcmple_wide</b> [_n_s8](svbool_t <i>pg</i> , svint8_t <i>op1</i> , int64_t <i>op2</i> )
svbool_t	<b>svcmple_wide</b> [_n_s16](svbool_t <i>pg</i> , svint16_t <i>op1</i> , int64_t <i>op2</i> )
svbool_t	<b>svcmple_wide</b> [_n_s32](svbool_t <i>pg</i> , svint32_t <i>op1</i> , int64_t <i>op2</i> )
svbool_t	<b>svcmple_wide</b> [_n_u8](svbool_t <i>pg</i> , svuint8_t <i>op1</i> , uint64_t <i>op2</i> )
svbool_t	<b>svcmple_wide</b> [_n_u16](svbool_t <i>pg</i> , svuint16_t <i>op1</i> , uint64_t <i>op2</i> )
svbool_t	<b>svcmple_wide</b> [_n_u32](svbool_t <i>pg</i> , svuint32_t <i>op1</i> , uint64_t <i>op2</i> )

### 6.11.5. CMPGE: Integer compare greater than or equal to

These functions compare two integer inputs and return a predicate bit that indicates whether the first input is greater than or equal to the second.

These functions handle both signed and unsigned inputs; there are no separate functions for CMPHS.

#### 6.11.5.1. CMPGE (vector, vector)

Instances	
svbool_t	<b>svcmpge</b> [_s8](svbool_t <i>pg</i> , svint8_t <i>op1</i> , svint8_t <i>op2</i> )
svbool_t	<b>svcmpge</b> [_s16](svbool_t <i>pg</i> , svint16_t <i>op1</i> , svint16_t <i>op2</i> )
svbool_t	<b>svcmpge</b> [_s32](svbool_t <i>pg</i> , svint32_t <i>op1</i> , svint32_t <i>op2</i> )
svbool_t	<b>svcmpge</b> [_s64](svbool_t <i>pg</i> , svint64_t <i>op1</i> , svint64_t <i>op2</i> )
svbool_t	<b>svcmpge</b> [_u8](svbool_t <i>pg</i> , svuint8_t <i>op1</i> , svuint8_t <i>op2</i> )
svbool_t	<b>svcmpge</b> [_u16](svbool_t <i>pg</i> , svuint16_t <i>op1</i> , svuint16_t <i>op2</i> )
svbool_t	<b>svcmpge</b> [_u32](svbool_t <i>pg</i> , svuint32_t <i>op1</i> , svuint32_t <i>op2</i> )
svbool_t	<b>svcmpge</b> [_u64](svbool_t <i>pg</i> , svuint64_t <i>op1</i> , svuint64_t <i>op2</i> )

#### 6.11.5.2. CMPGE (vector, scalar)

Instances	
svbool_t	<b>svcmpge</b> [_n_s8](svbool_t <i>pg</i> , svint8_t <i>op1</i> , int8_t <i>op2</i> )
svbool_t	<b>svcmpge</b> [_n_s16](svbool_t <i>pg</i> , svint16_t <i>op1</i> , int16_t <i>op2</i> )
svbool_t	<b>svcmpge</b> [_n_s32](svbool_t <i>pg</i> , svint32_t <i>op1</i> , int32_t <i>op2</i> )
svbool_t	<b>svcmpge</b> [_n_s64](svbool_t <i>pg</i> , svint64_t <i>op1</i> , int64_t <i>op2</i> )
svbool_t	<b>svcmpge</b> [_n_u8](svbool_t <i>pg</i> , svuint8_t <i>op1</i> , uint8_t <i>op2</i> )
svbool_t	<b>svcmpge</b> [_n_u16](svbool_t <i>pg</i> , svuint16_t <i>op1</i> , uint16_t <i>op2</i> )
svbool_t	<b>svcmpge</b> [_n_u32](svbool_t <i>pg</i> , svuint32_t <i>op1</i> , uint32_t <i>op2</i> )
svbool_t	<b>svcmpge</b> [_n_u64](svbool_t <i>pg</i> , svuint64_t <i>op1</i> , uint64_t <i>op2</i> )

#### 6.11.5.3. CMPGE (vector, wide vector)

Instances	
svbool_t	<b>svcmpge_wide</b> [_s8](svbool_t <i>pg</i> , svint8_t <i>op1</i> , svint64_t <i>op2</i> )
svbool_t	<b>svcmpge_wide</b> [_s16](svbool_t <i>pg</i> , svint16_t <i>op1</i> , svint64_t <i>op2</i> )
svbool_t	<b>svcmpge_wide</b> [_s32](svbool_t <i>pg</i> , svint32_t <i>op1</i> , svint64_t <i>op2</i> )
svbool_t	<b>svcmpge_wide</b> [_u8](svbool_t <i>pg</i> , svuint8_t <i>op1</i> , svuint64_t <i>op2</i> )
svbool_t	<b>svcmpge_wide</b> [_u16](svbool_t <i>pg</i> , svuint16_t <i>op1</i> , svuint64_t <i>op2</i> )

**Instances**

```
svbool_t svcmpge_wide[_u32](svbool_t pg, svuint32_t op1, svuint64_t op2)
```

**6.11.5.4. CMPGE (vector, wide scalar)****Instances**

```
svbool_t svcmpge_wide[_n_s8](svbool_t pg, svint8_t op1, int64_t op2)
svbool_t svcmpge_wide[_n_s16](svbool_t pg, svint16_t op1, int64_t op2)
svbool_t svcmpge_wide[_n_s32](svbool_t pg, svint32_t op1, int64_t op2)
svbool_t svcmpge_wide[_n_u8](svbool_t pg, svuint8_t op1, uint64_t op2)
svbool_t svcmpge_wide[_n_u16](svbool_t pg, svuint16_t op1, uint64_t op2)
svbool_t svcmpge_wide[_n_u32](svbool_t pg, svuint32_t op1, uint64_t op2)
```

**6.11.6. CMPGT: Integer compare greater than**

These functions compare two integer inputs and return a predicate bit that indicates whether the first input is greater than the second.

These functions handle both signed and unsigned inputs; there are no separate functions for CMPHI.

**6.11.6.1. CMPGT (vector, vector)****Instances**

```
svbool_t svcmpgt[_s8](svbool_t pg, svint8_t op1, svint8_t op2)
svbool_t svcmpgt[_s16](svbool_t pg, svint16_t op1, svint16_t op2)
svbool_t svcmpgt[_s32](svbool_t pg, svint32_t op1, svint32_t op2)
svbool_t svcmpgt[_s64](svbool_t pg, svint64_t op1, svint64_t op2)
svbool_t svcmpgt[_u8](svbool_t pg, svuint8_t op1, svuint8_t op2)
svbool_t svcmpgt[_u16](svbool_t pg, svuint16_t op1, svuint16_t op2)
svbool_t svcmpgt[_u32](svbool_t pg, svuint32_t op1, svuint32_t op2)
svbool_t svcmpgt[_u64](svbool_t pg, svuint64_t op1, svuint64_t op2)
```

**6.11.6.2. CMPGT (vector, scalar)****Instances**

```
svbool_t svcmpgt[_n_s8](svbool_t pg, svint8_t op1, int8_t op2)
svbool_t svcmpgt[_n_s16](svbool_t pg, svint16_t op1, int16_t op2)
svbool_t svcmpgt[_n_s32](svbool_t pg, svint32_t op1, int32_t op2)
svbool_t svcmpgt[_n_s64](svbool_t pg, svint64_t op1, int64_t op2)
svbool_t svcmpgt[_n_u8](svbool_t pg, svuint8_t op1, uint8_t op2)
svbool_t svcmpgt[_n_u16](svbool_t pg, svuint16_t op1, uint16_t op2)
svbool_t svcmpgt[_n_u32](svbool_t pg, svuint32_t op1, uint32_t op2)
svbool_t svcmpgt[_n_u64](svbool_t pg, svuint64_t op1, uint64_t op2)
```

**6.11.6.3. CMPGT (vector, wide vector)****Instances**

```
svbool_t svcmpgt_wide[_s8](svbool_t pg, svint8_t op1, svint64_t op2)
svbool_t svcmpgt_wide[_s16](svbool_t pg, svint16_t op1, svint64_t op2)
svbool_t svcmpgt_wide[_s32](svbool_t pg, svint32_t op1, svint64_t op2)
svbool_t svcmpgt_wide[_u8](svbool_t pg, svuint8_t op1, svuint64_t op2)
svbool_t svcmpgt_wide[_u16](svbool_t pg, svuint16_t op1, svuint64_t op2)
svbool_t svcmpgt_wide[_u32](svbool_t pg, svuint32_t op1, svuint64_t op2)
```

### 6.11.6.4. CMPGT (vector, wide scalar)

Instances	
svbool_t	<b>svcmpgt_wide</b> [_n_s8](svbool_t <i>pg</i> , svint8_t <i>op1</i> , int64_t <i>op2</i> )
svbool_t	<b>svcmpgt_wide</b> [_n_s16](svbool_t <i>pg</i> , svint16_t <i>op1</i> , int64_t <i>op2</i> )
svbool_t	<b>svcmpgt_wide</b> [_n_s32](svbool_t <i>pg</i> , svint32_t <i>op1</i> , int64_t <i>op2</i> )
svbool_t	<b>svcmpgt_wide</b> [_n_u8](svbool_t <i>pg</i> , svuint8_t <i>op1</i> , uint64_t <i>op2</i> )
svbool_t	<b>svcmpgt_wide</b> [_n_u16](svbool_t <i>pg</i> , svuint16_t <i>op1</i> , uint64_t <i>op2</i> )
svbool_t	<b>svcmpgt_wide</b> [_n_u32](svbool_t <i>pg</i> , svuint32_t <i>op1</i> , uint64_t <i>op2</i> )

## 6.12. While comparisons

### 6.12.1. WHILELT: While incrementing variable is less than

These functions return a predicate in which element *N* is active if, for all values *M* in the range  $[0, N]$ , adding *M* to the first input gives a value that is less than the second. (Note that the behavior is the same regardless of whether the addition uses modular, saturating or natural arithmetic.)

A suffix starting with **\_b** indicates the number of bits in an element. For example, a suffix of **\_b8** indicates that the predicate controls 8-bit data, so element *N* corresponds to bit *N* of the predicate. A suffix of **\_b16** indicates that the predicate controls 16-bit data, so element *N* corresponds to bit  $N \times 2$  of the predicate. When an element has more than one predicate bit associated with it, only the lowest of those bits is ever true.

These functions handle both signed and unsigned inputs; there are no separate functions for WHILELO.

#### 6.12.1.1. WHILELT (scalar, scalar)

Instances	
svbool_t	<b>svwhilelt_b8</b> [_s32](int32_t <i>op1</i> , int32_t <i>op2</i> )
svbool_t	<b>svwhilelt_b16</b> [_s32](int32_t <i>op1</i> , int32_t <i>op2</i> )
svbool_t	<b>svwhilelt_b32</b> [_s32](int32_t <i>op1</i> , int32_t <i>op2</i> )
svbool_t	<b>svwhilelt_b64</b> [_s32](int32_t <i>op1</i> , int32_t <i>op2</i> )
svbool_t	<b>svwhilelt_b8</b> [_u32](uint32_t <i>op1</i> , uint32_t <i>op2</i> )
svbool_t	<b>svwhilelt_b16</b> [_u32](uint32_t <i>op1</i> , uint32_t <i>op2</i> )
svbool_t	<b>svwhilelt_b32</b> [_u32](uint32_t <i>op1</i> , uint32_t <i>op2</i> )
svbool_t	<b>svwhilelt_b64</b> [_u32](uint32_t <i>op1</i> , uint32_t <i>op2</i> )
svbool_t	<b>svwhilelt_b8</b> [_s64](int64_t <i>op1</i> , int64_t <i>op2</i> )
svbool_t	<b>svwhilelt_b16</b> [_s64](int64_t <i>op1</i> , int64_t <i>op2</i> )
svbool_t	<b>svwhilelt_b32</b> [_s64](int64_t <i>op1</i> , int64_t <i>op2</i> )
svbool_t	<b>svwhilelt_b64</b> [_s64](int64_t <i>op1</i> , int64_t <i>op2</i> )
svbool_t	<b>svwhilelt_b8</b> [_u64](uint64_t <i>op1</i> , uint64_t <i>op2</i> )
svbool_t	<b>svwhilelt_b16</b> [_u64](uint64_t <i>op1</i> , uint64_t <i>op2</i> )
svbool_t	<b>svwhilelt_b32</b> [_u64](uint64_t <i>op1</i> , uint64_t <i>op2</i> )
svbool_t	<b>svwhilelt_b64</b> [_u64](uint64_t <i>op1</i> , uint64_t <i>op2</i> )

### 6.12.2. WHILELE: While incrementing variable is less than or equal to

These functions return a predicate in which element *N* is active if, for all values *M* in the range  $[0, N]$ , adding *M* to the first input gives a value that is less than or equal to the second. The addition uses modular arithmetic in the type of the first operand, so one plus the maximum value gives the minimum value. This wrapping is well-defined even for signed types.

See [Section 6.12.1, “WHILELT: While incrementing variable is less than”](#) for a description of the suffix.



These functions handle both signed and unsigned inputs; there are no separate functions for WHILELS. Note that when the second operand is the maximum value, every element in the returned predicate will be active.

### 6.12.2.1. WHILELE (scalar, scalar)

Instances	
svbool_t	<b>svwhilele_b8</b> [_s32](int32_t op1, int32_t op2)
svbool_t	<b>svwhilele_b16</b> [_s32](int32_t op1, int32_t op2)
svbool_t	<b>svwhilele_b32</b> [_s32](int32_t op1, int32_t op2)
svbool_t	<b>svwhilele_b64</b> [_s32](int32_t op1, int32_t op2)
svbool_t	<b>svwhilele_b8</b> [_u32](uint32_t op1, uint32_t op2)
svbool_t	<b>svwhilele_b16</b> [_u32](uint32_t op1, uint32_t op2)
svbool_t	<b>svwhilele_b32</b> [_u32](uint32_t op1, uint32_t op2)
svbool_t	<b>svwhilele_b64</b> [_u32](uint32_t op1, uint32_t op2)
svbool_t	<b>svwhilele_b8</b> [_s64](int64_t op1, int64_t op2)
svbool_t	<b>svwhilele_b16</b> [_s64](int64_t op1, int64_t op2)
svbool_t	<b>svwhilele_b32</b> [_s64](int64_t op1, int64_t op2)
svbool_t	<b>svwhilele_b64</b> [_s64](int64_t op1, int64_t op2)
svbool_t	<b>svwhilele_b8</b> [_u64](uint64_t op1, uint64_t op2)
svbool_t	<b>svwhilele_b16</b> [_u64](uint64_t op1, uint64_t op2)
svbool_t	<b>svwhilele_b32</b> [_u64](uint64_t op1, uint64_t op2)
svbool_t	<b>svwhilele_b64</b> [_u64](uint64_t op1, uint64_t op2)

## 6.13. Counting bits

### 6.13.1. CLS: Count leading sign bits

These functions count the number of leading sign bits in an integer input, returning the result as an unsigned value.

#### 6.13.1.1. CLS (vector), setting inactive to zero

Instances	
svuint8_t	<b>svcls</b> [_s8]_z(svbool_t pg, svint8_t op)
svuint16_t	<b>svcls</b> [_s16]_z(svbool_t pg, svint16_t op)
svuint32_t	<b>svcls</b> [_s32]_z(svbool_t pg, svint32_t op)
svuint64_t	<b>svcls</b> [_s64]_z(svbool_t pg, svint64_t op)

#### 6.13.1.2. CLS (vector), merging with separate vector

Instances	
svuint8_t	<b>svcls</b> [_s8]_m(svuint8_t inactive, svbool_t pg, svint8_t op)
svuint16_t	<b>svcls</b> [_s16]_m(svuint16_t inactive, svbool_t pg, svint16_t op)
svuint32_t	<b>svcls</b> [_s32]_m(svuint32_t inactive, svbool_t pg, svint32_t op)
svuint64_t	<b>svcls</b> [_s64]_m(svuint64_t inactive, svbool_t pg, svint64_t op)

#### 6.13.1.3. CLS (vector), setting inactive to unknown

Instances	
svuint8_t	<b>svcls</b> [_s8]_x(svbool_t pg, svint8_t op)
svuint16_t	<b>svcls</b> [_s16]_x(svbool_t pg, svint16_t op)
svuint32_t	<b>svcls</b> [_s32]_x(svbool_t pg, svint32_t op)
svuint64_t	<b>svcls</b> [_s64]_x(svbool_t pg, svint64_t op)

## 6.13.2. CLZ: Count leading zero bits

These functions count the number of leading (high-order) zero bits in an integer input, returning the result as an unsigned value.

### 6.13.2.1. CLZ (vector), setting inactive to zero

Instances
<pre> svuint8_t  svclz[_s8]_z(svbool_t pg, svint8_t op) svuint16_t svclz[_s16]_z(svbool_t pg, svint16_t op) svuint32_t svclz[_s32]_z(svbool_t pg, svint32_t op) svuint64_t svclz[_s64]_z(svbool_t pg, svint64_t op) svuint8_t  svclz[_u8]_z(svbool_t pg, svuint8_t op) svuint16_t svclz[_u16]_z(svbool_t pg, svuint16_t op) svuint32_t svclz[_u32]_z(svbool_t pg, svuint32_t op) svuint64_t svclz[_u64]_z(svbool_t pg, svuint64_t op) </pre>

### 6.13.2.2. CLZ (vector), merging with separate vector

Instances
<pre> svuint8_t  svclz[_s8]_m(svuint8_t inactive, svbool_t pg, svint8_t op) svuint16_t svclz[_s16]_m(svuint16_t inactive, svbool_t pg, svint16_t op) svuint32_t svclz[_s32]_m(svuint32_t inactive, svbool_t pg, svint32_t op) svuint64_t svclz[_s64]_m(svuint64_t inactive, svbool_t pg, svint64_t op) svuint8_t  svclz[_u8]_m(svuint8_t inactive, svbool_t pg, svuint8_t op) svuint16_t svclz[_u16]_m(svuint16_t inactive, svbool_t pg, svuint16_t op) svuint32_t svclz[_u32]_m(svuint32_t inactive, svbool_t pg, svuint32_t op) svuint64_t svclz[_u64]_m(svuint64_t inactive, svbool_t pg, svuint64_t op) </pre>

### 6.13.2.3. CLZ (vector), setting inactive to unknown

Instances
<pre> svuint8_t  svclz[_s8]_x(svbool_t pg, svint8_t op) svuint16_t svclz[_s16]_x(svbool_t pg, svint16_t op) svuint32_t svclz[_s32]_x(svbool_t pg, svint32_t op) svuint64_t svclz[_s64]_x(svbool_t pg, svint64_t op) svuint8_t  svclz[_u8]_x(svbool_t pg, svuint8_t op) svuint16_t svclz[_u16]_x(svbool_t pg, svuint16_t op) svuint32_t svclz[_u32]_x(svbool_t pg, svuint32_t op) svuint64_t svclz[_u64]_x(svbool_t pg, svuint64_t op) </pre>

## 6.13.3. CNT: Count nonzero bits

These functions count the number of nonzero bits in an integer input, returning the result as an unsigned value.

### 6.13.3.1. CNT (vector), setting inactive to zero

Instances
<pre> svuint8_t  svcnt[_s8]_z(svbool_t pg, svint8_t op) svuint16_t svcnt[_s16]_z(svbool_t pg, svint16_t op) svuint32_t svcnt[_s32]_z(svbool_t pg, svint32_t op) svuint64_t svcnt[_s64]_z(svbool_t pg, svint64_t op) svuint8_t  svcnt[_u8]_z(svbool_t pg, svuint8_t op) </pre>

**Instances**

```

svuint16_t svcnt[_u16]_z(svbool_t pg, svuint16_t op)
svuint32_t svcnt[_u32]_z(svbool_t pg, svuint32_t op)
svuint64_t svcnt[_u64]_z(svbool_t pg, svuint64_t op)
svuint16_t svcnt[_f16]_z(svbool_t pg, svfloat16_t op)
svuint32_t svcnt[_f32]_z(svbool_t pg, svfloat32_t op)
svuint64_t svcnt[_f64]_z(svbool_t pg, svfloat64_t op)
svuint16_t svcnt[_bf16]_z(svbool_t pg, svbfloat16_t op)

```

**6.13.3.2. CNT (vector), merging with separate vector****Instances**

```

svuint8_t svcnt[_s8]_m(svuint8_t inactive, svbool_t pg, svint8_t op)
svuint16_t svcnt[_s16]_m(svuint16_t inactive, svbool_t pg, svint16_t op)
svuint32_t svcnt[_s32]_m(svuint32_t inactive, svbool_t pg, svint32_t op)
svuint64_t svcnt[_s64]_m(svuint64_t inactive, svbool_t pg, svint64_t op)
svuint8_t svcnt[_u8]_m(svuint8_t inactive, svbool_t pg, svuint8_t op)
svuint16_t svcnt[_u16]_m(svuint16_t inactive, svbool_t pg, svuint16_t op)
svuint32_t svcnt[_u32]_m(svuint32_t inactive, svbool_t pg, svuint32_t op)
svuint64_t svcnt[_u64]_m(svuint64_t inactive, svbool_t pg, svuint64_t op)
svuint16_t svcnt[_f16]_m(svuint16_t inactive, svbool_t pg, svfloat16_t op)
svuint32_t svcnt[_f32]_m(svuint32_t inactive, svbool_t pg, svfloat32_t op)
svuint64_t svcnt[_f64]_m(svuint64_t inactive, svbool_t pg, svfloat64_t op)
svuint16_t svcnt[_bf16]_m(svuint16_t inactive, svbool_t pg,
                        svbfloat16_t op)

```

**6.13.3.3. CNT (vector), setting inactive to unknown****Instances**

```

svuint8_t svcnt[_s8]_x(svbool_t pg, svint8_t op)
svuint16_t svcnt[_s16]_x(svbool_t pg, svint16_t op)
svuint32_t svcnt[_s32]_x(svbool_t pg, svint32_t op)
svuint64_t svcnt[_s64]_x(svbool_t pg, svint64_t op)
svuint8_t svcnt[_u8]_x(svbool_t pg, svuint8_t op)
svuint16_t svcnt[_u16]_x(svbool_t pg, svuint16_t op)
svuint32_t svcnt[_u32]_x(svbool_t pg, svuint32_t op)
svuint64_t svcnt[_u64]_x(svbool_t pg, svuint64_t op)
svuint16_t svcnt[_f16]_x(svbool_t pg, svfloat16_t op)
svuint32_t svcnt[_f32]_x(svbool_t pg, svfloat32_t op)
svuint64_t svcnt[_f64]_x(svbool_t pg, svfloat64_t op)
svuint16_t svcnt[_bf16]_x(svbool_t pg, svbfloat16_t op)

```

**6.14. Conversion****6.14.1. EXTB: Extend from low 8 bits**

These functions extend the low 8 bits of an integer input to the width of the result. They use sign extension if the result is signed and zero extension if the result is unsigned.

**6.14.1.1. EXTB (vector), setting inactive to zero****Instances**

```

svint16_t svextb[_s16]_z(svbool_t pg, svint16_t op)
svint32_t svextb[_s32]_z(svbool_t pg, svint32_t op)

```

Instances
svint64_t <b>svextb</b> [_s64]_z(svbool_t pg, svint64_t op)
svuint16_t <b>svextb</b> [_u16]_z(svbool_t pg, svuint16_t op)
svuint32_t <b>svextb</b> [_u32]_z(svbool_t pg, svuint32_t op)
svuint64_t <b>svextb</b> [_u64]_z(svbool_t pg, svuint64_t op)

### 6.14.1.2. EXTB (vector), merging with separate vector

Instances
svint16_t <b>svextb</b> [_s16]_m(svint16_t inactive, svbool_t pg, svint16_t op)
svint32_t <b>svextb</b> [_s32]_m(svint32_t inactive, svbool_t pg, svint32_t op)
svint64_t <b>svextb</b> [_s64]_m(svint64_t inactive, svbool_t pg, svint64_t op)
svuint16_t <b>svextb</b> [_u16]_m(svuint16_t inactive, svbool_t pg, svuint16_t op)
svuint32_t <b>svextb</b> [_u32]_m(svuint32_t inactive, svbool_t pg, svuint32_t op)
svuint64_t <b>svextb</b> [_u64]_m(svuint64_t inactive, svbool_t pg, svuint64_t op)

### 6.14.1.3. EXTB (vector), setting inactive to unknown

Instances
svint16_t <b>svextb</b> [_s16]_x(svbool_t pg, svint16_t op)
svint32_t <b>svextb</b> [_s32]_x(svbool_t pg, svint32_t op)
svint64_t <b>svextb</b> [_s64]_x(svbool_t pg, svint64_t op)
svuint16_t <b>svextb</b> [_u16]_x(svbool_t pg, svuint16_t op)
svuint32_t <b>svextb</b> [_u32]_x(svbool_t pg, svuint32_t op)
svuint64_t <b>svextb</b> [_u64]_x(svbool_t pg, svuint64_t op)

## 6.14.2. EXTH: Extend from low 16 bits

These functions extend the low 16 bits of an integer input to the width of the result. They use sign extension if the result is signed and zero extension if the result is unsigned.

### 6.14.2.1. EXTH (vector), setting inactive to zero

Instances
svint32_t <b>svexth</b> [_s32]_z(svbool_t pg, svint32_t op)
svint64_t <b>svexth</b> [_s64]_z(svbool_t pg, svint64_t op)
svuint32_t <b>svexth</b> [_u32]_z(svbool_t pg, svuint32_t op)
svuint64_t <b>svexth</b> [_u64]_z(svbool_t pg, svuint64_t op)

### 6.14.2.2. EXTH (vector), merging with separate vector

Instances
svint32_t <b>svexth</b> [_s32]_m(svint32_t inactive, svbool_t pg, svint32_t op)
svint64_t <b>svexth</b> [_s64]_m(svint64_t inactive, svbool_t pg, svint64_t op)
svuint32_t <b>svexth</b> [_u32]_m(svuint32_t inactive, svbool_t pg, svuint32_t op)
svuint64_t <b>svexth</b> [_u64]_m(svuint64_t inactive, svbool_t pg, svuint64_t op)

### 6.14.2.3. EXTH (vector), setting inactive to unknown

Instances
svint32_t <b>svexth</b> [_s32]_x(svbool_t pg, svint32_t op)
svint64_t <b>svexth</b> [_s64]_x(svbool_t pg, svint64_t op)

**Instances**

```
svuint32_t svexth[_u32]_x(svbool_t pg, svuint32_t op)
svuint64_t svexth[_u64]_x(svbool_t pg, svuint64_t op)
```

**6.14.3. EXTW: Extend from low 32 bits**

These functions extend the low 32 bits of an integer input to the width of the result. They use sign extension if the result is signed and zero extension if the result is unsigned.

**6.14.3.1. EXTW (vector), setting inactive to zero****Instances**

```
svint64_t svextw[_s64]_z(svbool_t pg, svint64_t op)
svuint64_t svextw[_u64]_z(svbool_t pg, svuint64_t op)
```

**6.14.3.2. EXTW (vector), merging with separate vector****Instances**

```
svint64_t svextw[_s64]_m(svint64_t inactive, svbool_t pg, svint64_t op)
svuint64_t svextw[_u64]_m(svuint64_t inactive, svbool_t pg, svuint64_t op)
```

**6.14.3.3. EXTW (vector), setting inactive to unknown****Instances**

```
svint64_t svextw[_s64]_x(svbool_t pg, svint64_t op)
svuint64_t svextw[_u64]_x(svbool_t pg, svuint64_t op)
```

**6.15. Reversal****6.15.1. RBIT: Reverse bits within elements**

These functions reverse the bits of each active input element. The order of the elements does not change.

**6.15.1.1. RBIT (vector), setting inactive to zero****Instances**

```
svint8_t svrbit[_s8]_z(svbool_t pg, svint8_t op)
svint16_t svrbit[_s16]_z(svbool_t pg, svint16_t op)
svint32_t svrbit[_s32]_z(svbool_t pg, svint32_t op)
svint64_t svrbit[_s64]_z(svbool_t pg, svint64_t op)
svuint8_t svrbit[_u8]_z(svbool_t pg, svuint8_t op)
svuint16_t svrbit[_u16]_z(svbool_t pg, svuint16_t op)
svuint32_t svrbit[_u32]_z(svbool_t pg, svuint32_t op)
svuint64_t svrbit[_u64]_z(svbool_t pg, svuint64_t op)
```

**6.15.1.2. RBIT (vector), merging with separate vector****Instances**

```
svint8_t svrbit[_s8]_m(svint8_t inactive, svbool_t pg, svint8_t op)
svint16_t svrbit[_s16]_m(svint16_t inactive, svbool_t pg, svint16_t op)
svint32_t svrbit[_s32]_m(svint32_t inactive, svbool_t pg, svint32_t op)
```

Instances
svint64_t <b>svrbit</b> [_s64] <b>_m</b> (svint64_t <i>inactive</i> , svbool_t <i>pg</i> , svint64_t <i>op</i> )
svuint8_t <b>svrbit</b> [_u8] <b>_m</b> (svuint8_t <i>inactive</i> , svbool_t <i>pg</i> , svuint8_t <i>op</i> )
svuint16_t <b>svrbit</b> [_u16] <b>_m</b> (svuint16_t <i>inactive</i> , svbool_t <i>pg</i> , svuint16_t <i>op</i> )
svuint32_t <b>svrbit</b> [_u32] <b>_m</b> (svuint32_t <i>inactive</i> , svbool_t <i>pg</i> , svuint32_t <i>op</i> )
svuint64_t <b>svrbit</b> [_u64] <b>_m</b> (svuint64_t <i>inactive</i> , svbool_t <i>pg</i> , svuint64_t <i>op</i> )

### 6.15.1.3. RBIT (vector), setting inactive to unknown

Instances
svint8_t <b>svrbit</b> [_s8] <b>_x</b> (svbool_t <i>pg</i> , svint8_t <i>op</i> )
svint16_t <b>svrbit</b> [_s16] <b>_x</b> (svbool_t <i>pg</i> , svint16_t <i>op</i> )
svint32_t <b>svrbit</b> [_s32] <b>_x</b> (svbool_t <i>pg</i> , svint32_t <i>op</i> )
svint64_t <b>svrbit</b> [_s64] <b>_x</b> (svbool_t <i>pg</i> , svint64_t <i>op</i> )
svuint8_t <b>svrbit</b> [_u8] <b>_x</b> (svbool_t <i>pg</i> , svuint8_t <i>op</i> )
svuint16_t <b>svrbit</b> [_u16] <b>_x</b> (svbool_t <i>pg</i> , svuint16_t <i>op</i> )
svuint32_t <b>svrbit</b> [_u32] <b>_x</b> (svbool_t <i>pg</i> , svuint32_t <i>op</i> )
svuint64_t <b>svrbit</b> [_u64] <b>_x</b> (svbool_t <i>pg</i> , svuint64_t <i>op</i> )

## 6.15.2. REVB: Reverse bytes within elements

These functions reverse the 8-bit bytes of each active input element. The order of the elements does not change.

### 6.15.2.1. REVB (vector), setting inactive to zero

Instances
svint16_t <b>svrevb</b> [_s16] <b>_z</b> (svbool_t <i>pg</i> , svint16_t <i>op</i> )
svint32_t <b>svrevb</b> [_s32] <b>_z</b> (svbool_t <i>pg</i> , svint32_t <i>op</i> )
svint64_t <b>svrevb</b> [_s64] <b>_z</b> (svbool_t <i>pg</i> , svint64_t <i>op</i> )
svuint16_t <b>svrevb</b> [_u16] <b>_z</b> (svbool_t <i>pg</i> , svuint16_t <i>op</i> )
svuint32_t <b>svrevb</b> [_u32] <b>_z</b> (svbool_t <i>pg</i> , svuint32_t <i>op</i> )
svuint64_t <b>svrevb</b> [_u64] <b>_z</b> (svbool_t <i>pg</i> , svuint64_t <i>op</i> )

### 6.15.2.2. REVB (vector), merging with separate vector

Instances
svint16_t <b>svrevb</b> [_s16] <b>_m</b> (svint16_t <i>inactive</i> , svbool_t <i>pg</i> , svint16_t <i>op</i> )
svint32_t <b>svrevb</b> [_s32] <b>_m</b> (svint32_t <i>inactive</i> , svbool_t <i>pg</i> , svint32_t <i>op</i> )
svint64_t <b>svrevb</b> [_s64] <b>_m</b> (svint64_t <i>inactive</i> , svbool_t <i>pg</i> , svint64_t <i>op</i> )
svuint16_t <b>svrevb</b> [_u16] <b>_m</b> (svuint16_t <i>inactive</i> , svbool_t <i>pg</i> , svuint16_t <i>op</i> )
svuint32_t <b>svrevb</b> [_u32] <b>_m</b> (svuint32_t <i>inactive</i> , svbool_t <i>pg</i> , svuint32_t <i>op</i> )
svuint64_t <b>svrevb</b> [_u64] <b>_m</b> (svuint64_t <i>inactive</i> , svbool_t <i>pg</i> , svuint64_t <i>op</i> )

### 6.15.2.3. REVB (vector), setting inactive to unknown

Instances
svint16_t <b>svrevb</b> [_s16] <b>_x</b> (svbool_t <i>pg</i> , svint16_t <i>op</i> )
svint32_t <b>svrevb</b> [_s32] <b>_x</b> (svbool_t <i>pg</i> , svint32_t <i>op</i> )
svint64_t <b>svrevb</b> [_s64] <b>_x</b> (svbool_t <i>pg</i> , svint64_t <i>op</i> )
svuint16_t <b>svrevb</b> [_u16] <b>_x</b> (svbool_t <i>pg</i> , svuint16_t <i>op</i> )
svuint32_t <b>svrevb</b> [_u32] <b>_x</b> (svbool_t <i>pg</i> , svuint32_t <i>op</i> )
svuint64_t <b>svrevb</b> [_u64] <b>_x</b> (svbool_t <i>pg</i> , svuint64_t <i>op</i> )

### 6.15.3. REVH: Reverse halfwords within elements

These functions reverse the 16-bit halfwords of each active input element. The order of the elements does not change.

#### 6.15.3.1. REVH (vector), setting inactive to zero

##### Instances

```
svint32_t svrevh[_s32]_z(svbool_t pg, svint32_t op)
svint64_t svrevh[_s64]_z(svbool_t pg, svint64_t op)
svuint32_t svrevh[_u32]_z(svbool_t pg, svuint32_t op)
svuint64_t svrevh[_u64]_z(svbool_t pg, svuint64_t op)
```

#### 6.15.3.2. REVH (vector), merging with separate vector

##### Instances

```
svint32_t svrevh[_s32]_m(svint32_t inactive, svbool_t pg, svint32_t op)
svint64_t svrevh[_s64]_m(svint64_t inactive, svbool_t pg, svint64_t op)
svuint32_t svrevh[_u32]_m(svuint32_t inactive, svbool_t pg, svuint32_t op)
svuint64_t svrevh[_u64]_m(svuint64_t inactive, svbool_t pg, svuint64_t op)
```

#### 6.15.3.3. REVH (vector), setting inactive to unknown

##### Instances

```
svint32_t svrevh[_s32]_x(svbool_t pg, svint32_t op)
svint64_t svrevh[_s64]_x(svbool_t pg, svint64_t op)
svuint32_t svrevh[_u32]_x(svbool_t pg, svuint32_t op)
svuint64_t svrevh[_u64]_x(svbool_t pg, svuint64_t op)
```

### 6.15.4. REVW: Reverse words within elements

These functions reverse the 32-bit words of each active input element. The order of the elements does not change.

#### 6.15.4.1. REVW (vector), setting inactive to zero

##### Instances

```
svint64_t svrevw[_s64]_z(svbool_t pg, svint64_t op)
svuint64_t svrevw[_u64]_z(svbool_t pg, svuint64_t op)
```

#### 6.15.4.2. REVW (vector), merging with separate vector

##### Instances

```
svint64_t svrevw[_s64]_m(svint64_t inactive, svbool_t pg, svint64_t op)
svuint64_t svrevw[_u64]_m(svuint64_t inactive, svbool_t pg, svuint64_t op)
```

#### 6.15.4.3. REVW (vector), setting inactive to unknown

##### Instances

```
svint64_t svrevw[_s64]_x(svbool_t pg, svint64_t op)
```

Instances
svuint64_t <b>svrevw</b> [_u64]_x(svbool_t pg, svuint64_t op)

## 6.16. Floating-point arithmetic

### 6.16.1. ADD: Floating-point addition

These functions perform addition on two floating-point inputs.

Signaling NaNs trigger an IEEE Invalid exception but quiet NaNs do not. Adding +Inf and -Inf together also triggers an IEEE Invalid exception.

#### 6.16.1.1. ADD (vector, vector), setting inactive to zero

Instances
svfloat16_t <b>svadd</b> [_f16]_z(svbool_t pg, svfloat16_t op1, svfloat16_t op2)
svfloat32_t <b>svadd</b> [_f32]_z(svbool_t pg, svfloat32_t op1, svfloat32_t op2)
svfloat64_t <b>svadd</b> [_f64]_z(svbool_t pg, svfloat64_t op1, svfloat64_t op2)

#### 6.16.1.2. ADD (vector, vector), merging with first input

Instances
svfloat16_t <b>svadd</b> [_f16]_m(svbool_t pg, svfloat16_t op1, svfloat16_t op2)
svfloat32_t <b>svadd</b> [_f32]_m(svbool_t pg, svfloat32_t op1, svfloat32_t op2)
svfloat64_t <b>svadd</b> [_f64]_m(svbool_t pg, svfloat64_t op1, svfloat64_t op2)

#### 6.16.1.3. ADD (vector, vector), setting inactive to unknown

Instances
svfloat16_t <b>svadd</b> [_f16]_x(svbool_t pg, svfloat16_t op1, svfloat16_t op2)
svfloat32_t <b>svadd</b> [_f32]_x(svbool_t pg, svfloat32_t op1, svfloat32_t op2)
svfloat64_t <b>svadd</b> [_f64]_x(svbool_t pg, svfloat64_t op1, svfloat64_t op2)

#### 6.16.1.4. ADD (vector, scalar), setting inactive to zero

Instances
svfloat16_t <b>svadd</b> [_n_f16]_z(svbool_t pg, svfloat16_t op1, float16_t op2)
svfloat32_t <b>svadd</b> [_n_f32]_z(svbool_t pg, svfloat32_t op1, float32_t op2)
svfloat64_t <b>svadd</b> [_n_f64]_z(svbool_t pg, svfloat64_t op1, float64_t op2)

#### 6.16.1.5. ADD (vector, scalar), merging with first input

Instances
svfloat16_t <b>svadd</b> [_n_f16]_m(svbool_t pg, svfloat16_t op1, float16_t op2)
svfloat32_t <b>svadd</b> [_n_f32]_m(svbool_t pg, svfloat32_t op1, float32_t op2)
svfloat64_t <b>svadd</b> [_n_f64]_m(svbool_t pg, svfloat64_t op1, float64_t op2)

#### 6.16.1.6. ADD (vector, scalar), setting inactive to unknown

Instances
svfloat16_t <b>svadd</b> [_n_f16]_x(svbool_t pg, svfloat16_t op1, float16_t op2)
svfloat32_t <b>svadd</b> [_n_f32]_x(svbool_t pg, svfloat32_t op1, float32_t op2)



**Instances**

```
svfloat64_t svadd[_n_f64]_x(svbool_t pg, svfloat64_t op1, float64_t op2)
```

## 6.16.2. CADD: Floating-point complex addition with rotation

These functions take and return complex floating-point values, with the real components in even elements and the imaginary components in odd elements. The functions rotate the second complex input by the number of degrees specified by the final (rotation) input and then add the result to the first complex input. The rotation input must be an integer constant expression with the value 90 or 270.

Signaling NaNs trigger an IEEE Invalid exception but quiet NaNs do not. Adding +Inf and -Inf together also triggers an IEEE Invalid exception.

### 6.16.2.1. CADD (vector, vector, immediate), setting inactive to zero

**Instances**

```
svfloat16_t svcadd[_f16]_z(svbool_t pg, svfloat16_t op1, svfloat16_t op2,
                           uint64_t imm_rotation)
svfloat32_t svcadd[_f32]_z(svbool_t pg, svfloat32_t op1, svfloat32_t op2,
                           uint64_t imm_rotation)
svfloat64_t svcadd[_f64]_z(svbool_t pg, svfloat64_t op1, svfloat64_t op2,
                           uint64_t imm_rotation)
```

### 6.16.2.2. CADD (vector, vector, immediate), merging with first input

**Instances**

```
svfloat16_t svcadd[_f16]_m(svbool_t pg, svfloat16_t op1, svfloat16_t op2,
                           uint64_t imm_rotation)
svfloat32_t svcadd[_f32]_m(svbool_t pg, svfloat32_t op1, svfloat32_t op2,
                           uint64_t imm_rotation)
svfloat64_t svcadd[_f64]_m(svbool_t pg, svfloat64_t op1, svfloat64_t op2,
                           uint64_t imm_rotation)
```

### 6.16.2.3. CADD (vector, vector, immediate), setting inactive to unknown

**Instances**

```
svfloat16_t svcadd[_f16]_x(svbool_t pg, svfloat16_t op1, svfloat16_t op2,
                           uint64_t imm_rotation)
svfloat32_t svcadd[_f32]_x(svbool_t pg, svfloat32_t op1, svfloat32_t op2,
                           uint64_t imm_rotation)
svfloat64_t svcadd[_f64]_x(svbool_t pg, svfloat64_t op1, svfloat64_t op2,
                           uint64_t imm_rotation)
```

## 6.16.3. SUB: Floating-point subtraction

These functions subtract the second floating-point input from the first input.

Signaling NaNs trigger an IEEE Invalid exception but quiet NaNs do not. Subtracting an infinity from itself also triggers an IEEE Invalid exception.

### 6.16.3.1. SUB (vector, vector), setting inactive to zero

**Instances**

```
svfloat16_t svsub[_f16]_z(svbool_t pg, svfloat16_t op1, svfloat16_t op2)
```

Instances
svfloat32_t <b>svsub</b> [_f32]_z(svbool_t pg, svfloat32_t op1, svfloat32_t op2)
svfloat64_t <b>svsub</b> [_f64]_z(svbool_t pg, svfloat64_t op1, svfloat64_t op2)

### 6.16.3.2. SUB (vector, vector), merging with first input

Instances
svfloat16_t <b>svsub</b> [_f16]_m(svbool_t pg, svfloat16_t op1, svfloat16_t op2)
svfloat32_t <b>svsub</b> [_f32]_m(svbool_t pg, svfloat32_t op1, svfloat32_t op2)
svfloat64_t <b>svsub</b> [_f64]_m(svbool_t pg, svfloat64_t op1, svfloat64_t op2)

### 6.16.3.3. SUB (vector, vector), setting inactive to unknown

Instances
svfloat16_t <b>svsub</b> [_f16]_x(svbool_t pg, svfloat16_t op1, svfloat16_t op2)
svfloat32_t <b>svsub</b> [_f32]_x(svbool_t pg, svfloat32_t op1, svfloat32_t op2)
svfloat64_t <b>svsub</b> [_f64]_x(svbool_t pg, svfloat64_t op1, svfloat64_t op2)

### 6.16.3.4. SUB (vector, scalar), setting inactive to zero

Instances
svfloat16_t <b>svsub</b> [_n_f16]_z(svbool_t pg, svfloat16_t op1, float16_t op2)
svfloat32_t <b>svsub</b> [_n_f32]_z(svbool_t pg, svfloat32_t op1, float32_t op2)
svfloat64_t <b>svsub</b> [_n_f64]_z(svbool_t pg, svfloat64_t op1, float64_t op2)

### 6.16.3.5. SUB (vector, scalar), merging with first input

Instances
svfloat16_t <b>svsub</b> [_n_f16]_m(svbool_t pg, svfloat16_t op1, float16_t op2)
svfloat32_t <b>svsub</b> [_n_f32]_m(svbool_t pg, svfloat32_t op1, float32_t op2)
svfloat64_t <b>svsub</b> [_n_f64]_m(svbool_t pg, svfloat64_t op1, float64_t op2)

### 6.16.3.6. SUB (vector, scalar), setting inactive to unknown

Instances
svfloat16_t <b>svsub</b> [_n_f16]_x(svbool_t pg, svfloat16_t op1, float16_t op2)
svfloat32_t <b>svsub</b> [_n_f32]_x(svbool_t pg, svfloat32_t op1, float32_t op2)
svfloat64_t <b>svsub</b> [_n_f64]_x(svbool_t pg, svfloat64_t op1, float64_t op2)

## 6.16.4. SUBR: Floating-point subtraction, reversed

These functions subtract the first floating-point input from the second input.

Signaling NaNs trigger an IEEE Invalid exception but quiet NaNs do not. Subtracting an infinity from itself also triggers an IEEE Invalid exception.

### 6.16.4.1. SUBR (vector, vector), setting inactive to zero

Instances
svfloat16_t <b>svsubr</b> [_f16]_z(svbool_t pg, svfloat16_t op1, svfloat16_t op2)
svfloat32_t <b>svsubr</b> [_f32]_z(svbool_t pg, svfloat32_t op1, svfloat32_t op2)

Instances
svfloat64_t <b>svsubr</b> [_f64]_z(svbool_t <i>pg</i> , svfloat64_t <i>op1</i> , svfloat64_t <i>op2</i> )

#### 6.16.4.2. SUBR (vector, vector), merging with first input

Instances
svfloat16_t <b>svsubr</b> [_f16]_m(svbool_t <i>pg</i> , svfloat16_t <i>op1</i> , svfloat16_t <i>op2</i> )
svfloat32_t <b>svsubr</b> [_f32]_m(svbool_t <i>pg</i> , svfloat32_t <i>op1</i> , svfloat32_t <i>op2</i> )
svfloat64_t <b>svsubr</b> [_f64]_m(svbool_t <i>pg</i> , svfloat64_t <i>op1</i> , svfloat64_t <i>op2</i> )

#### 6.16.4.3. SUBR (vector, vector), setting inactive to unknown

Instances
svfloat16_t <b>svsubr</b> [_f16]_x(svbool_t <i>pg</i> , svfloat16_t <i>op1</i> , svfloat16_t <i>op2</i> )
svfloat32_t <b>svsubr</b> [_f32]_x(svbool_t <i>pg</i> , svfloat32_t <i>op1</i> , svfloat32_t <i>op2</i> )
svfloat64_t <b>svsubr</b> [_f64]_x(svbool_t <i>pg</i> , svfloat64_t <i>op1</i> , svfloat64_t <i>op2</i> )

#### 6.16.4.4. SUBR (vector, scalar), setting inactive to zero

Instances
svfloat16_t <b>svsubr</b> [_n_f16]_z(svbool_t <i>pg</i> , svfloat16_t <i>op1</i> , float16_t <i>op2</i> )
svfloat32_t <b>svsubr</b> [_n_f32]_z(svbool_t <i>pg</i> , svfloat32_t <i>op1</i> , float32_t <i>op2</i> )
svfloat64_t <b>svsubr</b> [_n_f64]_z(svbool_t <i>pg</i> , svfloat64_t <i>op1</i> , float64_t <i>op2</i> )

#### 6.16.4.5. SUBR (vector, scalar), merging with first input

Instances
svfloat16_t <b>svsubr</b> [_n_f16]_m(svbool_t <i>pg</i> , svfloat16_t <i>op1</i> , float16_t <i>op2</i> )
svfloat32_t <b>svsubr</b> [_n_f32]_m(svbool_t <i>pg</i> , svfloat32_t <i>op1</i> , float32_t <i>op2</i> )
svfloat64_t <b>svsubr</b> [_n_f64]_m(svbool_t <i>pg</i> , svfloat64_t <i>op1</i> , float64_t <i>op2</i> )

#### 6.16.4.6. SUBR (vector, scalar), setting inactive to unknown

Instances
svfloat16_t <b>svsubr</b> [_n_f16]_x(svbool_t <i>pg</i> , svfloat16_t <i>op1</i> , float16_t <i>op2</i> )
svfloat32_t <b>svsubr</b> [_n_f32]_x(svbool_t <i>pg</i> , svfloat32_t <i>op1</i> , float32_t <i>op2</i> )
svfloat64_t <b>svsubr</b> [_n_f64]_x(svbool_t <i>pg</i> , svfloat64_t <i>op1</i> , float64_t <i>op2</i> )

### 6.16.5. ABD: Floating-point absolute difference

These functions compute the absolute difference of two floating-point inputs.

Signaling NaNs trigger an IEEE Invalid exception but quiet NaNs do not. Subtracting an infinity from itself also triggers an IEEE Invalid exception.

#### 6.16.5.1. ABD (vector, vector), setting inactive to zero

Instances
svfloat16_t <b>svabd</b> [_f16]_z(svbool_t <i>pg</i> , svfloat16_t <i>op1</i> , svfloat16_t <i>op2</i> )
svfloat32_t <b>svabd</b> [_f32]_z(svbool_t <i>pg</i> , svfloat32_t <i>op1</i> , svfloat32_t <i>op2</i> )
svfloat64_t <b>svabd</b> [_f64]_z(svbool_t <i>pg</i> , svfloat64_t <i>op1</i> , svfloat64_t <i>op2</i> )

### 6.16.5.2. ABD (vector, vector), merging with first input

Instances	
svfloat16_t	<b>svabd</b> [_f16]_m(svbool_t pg, svfloat16_t op1, svfloat16_t op2)
svfloat32_t	<b>svabd</b> [_f32]_m(svbool_t pg, svfloat32_t op1, svfloat32_t op2)
svfloat64_t	<b>svabd</b> [_f64]_m(svbool_t pg, svfloat64_t op1, svfloat64_t op2)

### 6.16.5.3. ABD (vector, vector), setting inactive to unknown

Instances	
svfloat16_t	<b>svabd</b> [_f16]_x(svbool_t pg, svfloat16_t op1, svfloat16_t op2)
svfloat32_t	<b>svabd</b> [_f32]_x(svbool_t pg, svfloat32_t op1, svfloat32_t op2)
svfloat64_t	<b>svabd</b> [_f64]_x(svbool_t pg, svfloat64_t op1, svfloat64_t op2)

### 6.16.5.4. ABD (vector, scalar), setting inactive to zero

Instances	
svfloat16_t	<b>svabd</b> [_n_f16]_z(svbool_t pg, svfloat16_t op1, float16_t op2)
svfloat32_t	<b>svabd</b> [_n_f32]_z(svbool_t pg, svfloat32_t op1, float32_t op2)
svfloat64_t	<b>svabd</b> [_n_f64]_z(svbool_t pg, svfloat64_t op1, float64_t op2)

### 6.16.5.5. ABD (vector, scalar), merging with first input

Instances	
svfloat16_t	<b>svabd</b> [_n_f16]_m(svbool_t pg, svfloat16_t op1, float16_t op2)
svfloat32_t	<b>svabd</b> [_n_f32]_m(svbool_t pg, svfloat32_t op1, float32_t op2)
svfloat64_t	<b>svabd</b> [_n_f64]_m(svbool_t pg, svfloat64_t op1, float64_t op2)

### 6.16.5.6. ABD (vector, scalar), setting inactive to unknown

Instances	
svfloat16_t	<b>svabd</b> [_n_f16]_x(svbool_t pg, svfloat16_t op1, float16_t op2)
svfloat32_t	<b>svabd</b> [_n_f32]_x(svbool_t pg, svfloat32_t op1, float32_t op2)
svfloat64_t	<b>svabd</b> [_n_f64]_x(svbool_t pg, svfloat64_t op1, float64_t op2)

## 6.16.6. MUL: Floating-point multiplication

These functions multiply two floating-point inputs.

The `_lane` forms of the functions take one element in each 128-bit quadword of the second input and replicate it to fill the rest of the quadword. The `imm_index` parameter specifies the element within each quadword that the functions should replicate. This index must be an integer constant expression in the range  $[0, 128/N)$ , where  $N$  is the number of bits in each element.

Signaling NaNs trigger an IEEE Invalid exception but quiet NaNs do not. Multiplying an infinity by zero also triggers an IEEE Invalid exception.

### 6.16.6.1. MUL (vector, vector), setting inactive to zero

Instances	
svfloat16_t	<b>svmul</b> [_f16]_z(svbool_t pg, svfloat16_t op1, svfloat16_t op2)
svfloat32_t	<b>svmul</b> [_f32]_z(svbool_t pg, svfloat32_t op1, svfloat32_t op2)
svfloat64_t	<b>svmul</b> [_f64]_z(svbool_t pg, svfloat64_t op1, svfloat64_t op2)

### 6.16.6.2. MUL (vector, vector), merging with first input

#### Instances

```
svfloat16_t svmul[_f16]_m(svbool_t pg, svfloat16_t op1, svfloat16_t op2)
svfloat32_t svmul[_f32]_m(svbool_t pg, svfloat32_t op1, svfloat32_t op2)
svfloat64_t svmul[_f64]_m(svbool_t pg, svfloat64_t op1, svfloat64_t op2)
```

### 6.16.6.3. MUL (vector, vector), setting inactive to unknown

#### Instances

```
svfloat16_t svmul[_f16]_x(svbool_t pg, svfloat16_t op1, svfloat16_t op2)
svfloat32_t svmul[_f32]_x(svbool_t pg, svfloat32_t op1, svfloat32_t op2)
svfloat64_t svmul[_f64]_x(svbool_t pg, svfloat64_t op1, svfloat64_t op2)
```

### 6.16.6.4. MUL (vector, scalar), setting inactive to zero

#### Instances

```
svfloat16_t svmul[_n_f16]_z(svbool_t pg, svfloat16_t op1, float16_t op2)
svfloat32_t svmul[_n_f32]_z(svbool_t pg, svfloat32_t op1, float32_t op2)
svfloat64_t svmul[_n_f64]_z(svbool_t pg, svfloat64_t op1, float64_t op2)
```

### 6.16.6.5. MUL (vector, scalar), merging with first input

#### Instances

```
svfloat16_t svmul[_n_f16]_m(svbool_t pg, svfloat16_t op1, float16_t op2)
svfloat32_t svmul[_n_f32]_m(svbool_t pg, svfloat32_t op1, float32_t op2)
svfloat64_t svmul[_n_f64]_m(svbool_t pg, svfloat64_t op1, float64_t op2)
```

### 6.16.6.6. MUL (vector, scalar), setting inactive to unknown

#### Instances

```
svfloat16_t svmul[_n_f16]_x(svbool_t pg, svfloat16_t op1, float16_t op2)
svfloat32_t svmul[_n_f32]_x(svbool_t pg, svfloat32_t op1, float32_t op2)
svfloat64_t svmul[_n_f64]_x(svbool_t pg, svfloat64_t op1, float64_t op2)
```

### 6.16.6.7. MUL (vector, vector, lane)

#### Instances

```
svfloat16_t svmul_lane[_f16](svfloat16_t op1, svfloat16_t op2,
                             uint64_t imm_index)
svfloat32_t svmul_lane[_f32](svfloat32_t op1, svfloat32_t op2,
                             uint64_t imm_index)
svfloat64_t svmul_lane[_f64](svfloat64_t op1, svfloat64_t op2,
                             uint64_t imm_index)
```

## 6.16.7. MULX: Floating-point multiplication extended

### 6.16.7.1. MULX (vector, vector), setting inactive to zero

#### Instances

```
svfloat16_t svmulx[_f16]_z(svbool_t pg, svfloat16_t op1, svfloat16_t op2)
svfloat32_t svmulx[_f32]_z(svbool_t pg, svfloat32_t op1, svfloat32_t op2)
```

Instances
svfloat64_t <b>svmulx</b> [_f64]_z(svbool_t <i>pg</i> , svfloat64_t <i>op1</i> , svfloat64_t <i>op2</i> )

### 6.16.7.2. MULX (vector, vector), merging with first input

Instances
svfloat16_t <b>svmulx</b> [_f16]_m(svbool_t <i>pg</i> , svfloat16_t <i>op1</i> , svfloat16_t <i>op2</i> )
svfloat32_t <b>svmulx</b> [_f32]_m(svbool_t <i>pg</i> , svfloat32_t <i>op1</i> , svfloat32_t <i>op2</i> )
svfloat64_t <b>svmulx</b> [_f64]_m(svbool_t <i>pg</i> , svfloat64_t <i>op1</i> , svfloat64_t <i>op2</i> )

### 6.16.7.3. MULX (vector, vector), setting inactive to unknown

Instances
svfloat16_t <b>svmulx</b> [_f16]_x(svbool_t <i>pg</i> , svfloat16_t <i>op1</i> , svfloat16_t <i>op2</i> )
svfloat32_t <b>svmulx</b> [_f32]_x(svbool_t <i>pg</i> , svfloat32_t <i>op1</i> , svfloat32_t <i>op2</i> )
svfloat64_t <b>svmulx</b> [_f64]_x(svbool_t <i>pg</i> , svfloat64_t <i>op1</i> , svfloat64_t <i>op2</i> )

### 6.16.7.4. MULX (vector, scalar), setting inactive to zero

Instances
svfloat16_t <b>svmulx</b> [_n_f16]_z(svbool_t <i>pg</i> , svfloat16_t <i>op1</i> , float16_t <i>op2</i> )
svfloat32_t <b>svmulx</b> [_n_f32]_z(svbool_t <i>pg</i> , svfloat32_t <i>op1</i> , float32_t <i>op2</i> )
svfloat64_t <b>svmulx</b> [_n_f64]_z(svbool_t <i>pg</i> , svfloat64_t <i>op1</i> , float64_t <i>op2</i> )

### 6.16.7.5. MULX (vector, scalar), merging with first input

Instances
svfloat16_t <b>svmulx</b> [_n_f16]_m(svbool_t <i>pg</i> , svfloat16_t <i>op1</i> , float16_t <i>op2</i> )
svfloat32_t <b>svmulx</b> [_n_f32]_m(svbool_t <i>pg</i> , svfloat32_t <i>op1</i> , float32_t <i>op2</i> )
svfloat64_t <b>svmulx</b> [_n_f64]_m(svbool_t <i>pg</i> , svfloat64_t <i>op1</i> , float64_t <i>op2</i> )

### 6.16.7.6. MULX (vector, scalar), setting inactive to unknown

Instances
svfloat16_t <b>svmulx</b> [_n_f16]_x(svbool_t <i>pg</i> , svfloat16_t <i>op1</i> , float16_t <i>op2</i> )
svfloat32_t <b>svmulx</b> [_n_f32]_x(svbool_t <i>pg</i> , svfloat32_t <i>op1</i> , float32_t <i>op2</i> )
svfloat64_t <b>svmulx</b> [_n_f64]_x(svbool_t <i>pg</i> , svfloat64_t <i>op1</i> , float64_t <i>op2</i> )

## 6.16.8. MAD: Fused floating-point addition of product (multiplicand first)

These functions multiply the first two floating-point inputs and add the result to the third input. There is no intermediate rounding step after the multiplication.

Signaling NaNs trigger an IEEE Invalid exception but quiet NaNs do not. Multiplying an infinity by zero also triggers an IEEE Invalid exception, as does adding +Inf and -Inf together.

### 6.16.8.1. MAD (vector, vector, vector), setting inactive to zero

Instances
svfloat16_t <b>svmad</b> [_f16]_z(svbool_t <i>pg</i> , svfloat16_t <i>op1</i> , svfloat16_t <i>op2</i> ,

**Instances**

```

                                svfloat16_t op3)
svfloat32_t svmad[_f32]_z(svbool_t pg, svfloat32_t op1, svfloat32_t op2,
                                svfloat32_t op3)
svfloat64_t svmad[_f64]_z(svbool_t pg, svfloat64_t op1, svfloat64_t op2,
                                svfloat64_t op3)

```

**6.16.8.2. MAD (vector, vector, vector), merging with first input****Instances**

```

svfloat16_t svmad[_f16]_m(svbool_t pg, svfloat16_t op1, svfloat16_t op2,
                                svfloat16_t op3)
svfloat32_t svmad[_f32]_m(svbool_t pg, svfloat32_t op1, svfloat32_t op2,
                                svfloat32_t op3)
svfloat64_t svmad[_f64]_m(svbool_t pg, svfloat64_t op1, svfloat64_t op2,
                                svfloat64_t op3)

```

**6.16.8.3. MAD (vector, vector, vector), setting inactive to unknown****Instances**

```

svfloat16_t svmad[_f16]_x(svbool_t pg, svfloat16_t op1, svfloat16_t op2,
                                svfloat16_t op3)
svfloat32_t svmad[_f32]_x(svbool_t pg, svfloat32_t op1, svfloat32_t op2,
                                svfloat32_t op3)
svfloat64_t svmad[_f64]_x(svbool_t pg, svfloat64_t op1, svfloat64_t op2,
                                svfloat64_t op3)

```

**6.16.8.4. MAD (vector, vector, scalar), setting inactive to zero****Instances**

```

svfloat16_t svmad[_n_f16]_z(svbool_t pg, svfloat16_t op1, svfloat16_t op2,
                                float16_t op3)
svfloat32_t svmad[_n_f32]_z(svbool_t pg, svfloat32_t op1, svfloat32_t op2,
                                float32_t op3)
svfloat64_t svmad[_n_f64]_z(svbool_t pg, svfloat64_t op1, svfloat64_t op2,
                                float64_t op3)

```

**6.16.8.5. MAD (vector, vector, scalar), merging with first input****Instances**

```

svfloat16_t svmad[_n_f16]_m(svbool_t pg, svfloat16_t op1, svfloat16_t op2,
                                float16_t op3)
svfloat32_t svmad[_n_f32]_m(svbool_t pg, svfloat32_t op1, svfloat32_t op2,
                                float32_t op3)
svfloat64_t svmad[_n_f64]_m(svbool_t pg, svfloat64_t op1, svfloat64_t op2,
                                float64_t op3)

```

**6.16.8.6. MAD (vector, vector, scalar), setting inactive to unknown****Instances**

```

svfloat16_t svmad[_n_f16]_x(svbool_t pg, svfloat16_t op1, svfloat16_t op2,
                                float16_t op3)
svfloat32_t svmad[_n_f32]_x(svbool_t pg, svfloat32_t op1, svfloat32_t op2,
                                float32_t op3)

```

**Instances**

```
svfloat64_t svmad[_n_f64]_x(svbool_t pg, svfloat64_t op1, svfloat64_t op2,
                             float64_t op3)
```

### 6.16.9. MLA: Fused floating-point addition of product (addend first)

These functions multiply the second and third floating-point inputs and add the result to the first input. There is no intermediate rounding step after the multiplication.

The `_lane` forms of the functions take one element in each 128-bit quadword of the third input and replicate it to fill the rest of the quadword. The `imm_index` parameter specifies the element within each quadword that the functions should replicate. This index must be an integer constant expression in the range  $[0, 128/N]$ , where  $N$  is the number of bits in each element.

Signaling NaNs trigger an IEEE Invalid exception but quiet NaNs do not. Multiplying an infinity by zero also triggers an IEEE Invalid exception, as does adding `+Inf` and `-Inf` together.

#### 6.16.9.1. MLA (vector, vector, vector), setting inactive to zero

**Instances**

```
svfloat16_t svmla[_f16]_z(svbool_t pg, svfloat16_t op1, svfloat16_t op2,
                           svfloat16_t op3)
svfloat32_t svmla[_f32]_z(svbool_t pg, svfloat32_t op1, svfloat32_t op2,
                           svfloat32_t op3)
svfloat64_t svmla[_f64]_z(svbool_t pg, svfloat64_t op1, svfloat64_t op2,
                           svfloat64_t op3)
```

#### 6.16.9.2. MLA (vector, vector, vector), merging with first input

**Instances**

```
svfloat16_t svmla[_f16]_m(svbool_t pg, svfloat16_t op1, svfloat16_t op2,
                           svfloat16_t op3)
svfloat32_t svmla[_f32]_m(svbool_t pg, svfloat32_t op1, svfloat32_t op2,
                           svfloat32_t op3)
svfloat64_t svmla[_f64]_m(svbool_t pg, svfloat64_t op1, svfloat64_t op2,
                           svfloat64_t op3)
```

#### 6.16.9.3. MLA (vector, vector, vector), setting inactive to unknown

**Instances**

```
svfloat16_t svmla[_f16]_x(svbool_t pg, svfloat16_t op1, svfloat16_t op2,
                           svfloat16_t op3)
svfloat32_t svmla[_f32]_x(svbool_t pg, svfloat32_t op1, svfloat32_t op2,
                           svfloat32_t op3)
svfloat64_t svmla[_f64]_x(svbool_t pg, svfloat64_t op1, svfloat64_t op2,
                           svfloat64_t op3)
```

#### 6.16.9.4. MLA (vector, vector, scalar), setting inactive to zero

**Instances**

```
svfloat16_t svmla[_n_f16]_z(svbool_t pg, svfloat16_t op1, svfloat16_t op2,
                             float16_t op3)
svfloat32_t svmla[_n_f32]_z(svbool_t pg, svfloat32_t op1, svfloat32_t op2,
                             float32_t op3)
```



**Instances**

```
svfloat64_t svmla[_n_f64]_z(svbool_t pg, svfloat64_t op1, svfloat64_t op2,
                             float64_t op3)
```

**6.16.9.5. MLA (vector, vector, scalar), merging with first input****Instances**

```
svfloat16_t svmla[_n_f16]_m(svbool_t pg, svfloat16_t op1, svfloat16_t op2,
                             float16_t op3)
svfloat32_t svmla[_n_f32]_m(svbool_t pg, svfloat32_t op1, svfloat32_t op2,
                             float32_t op3)
svfloat64_t svmla[_n_f64]_m(svbool_t pg, svfloat64_t op1, svfloat64_t op2,
                             float64_t op3)
```

**6.16.9.6. MLA (vector, vector, scalar), setting inactive to unknown****Instances**

```
svfloat16_t svmla[_n_f16]_x(svbool_t pg, svfloat16_t op1, svfloat16_t op2,
                             float16_t op3)
svfloat32_t svmla[_n_f32]_x(svbool_t pg, svfloat32_t op1, svfloat32_t op2,
                             float32_t op3)
svfloat64_t svmla[_n_f64]_x(svbool_t pg, svfloat64_t op1, svfloat64_t op2,
                             float64_t op3)
```

**6.16.9.7. MLA (vector, vector, vector, lane)****Instances**

```
svfloat16_t svmla_lane[_f16](svfloat16_t op1, svfloat16_t op2,
                             svfloat16_t op3, uint64_t imm_index)
svfloat32_t svmla_lane[_f32](svfloat32_t op1, svfloat32_t op2,
                             svfloat32_t op3, uint64_t imm_index)
svfloat64_t svmla_lane[_f64](svfloat64_t op1, svfloat64_t op2,
                             svfloat64_t op3, uint64_t imm_index)
```

**6.16.10. CMLA: Fused floating-point complex addition of product with rotation**

These functions take and return complex floating-point values, with the real components in even elements and the imaginary components in odd elements. The functions rotate the third complex input by the number of degrees specified by the final (rotation) input, multiply the result by one component of the second complex input, then add the result to the first complex input.

The rotation input must be an integer constant expression with the value 0, 90, 180 or 270. When the rotation value is 0 or 180, the multiplication selects the real components of the second input, otherwise it selects the imaginary components. There is no intermediate rounding step after the multiplication.

The `_lane` forms of the functions take one complex value in each 128-bit quadword of the third input and replicate it to fill the rest of the quadword. The `imm_index` parameter specifies the complex value within each quadword that the functions should replicate. This index must be an integer constant expression in the range  $[0, 128/N)$ , where  $N$  is the number of bits in each complex value.

Signaling NaNs trigger an IEEE Invalid exception but quiet NaNs do not. Multiplying an infinity by zero also triggers an IEEE Invalid exception, as does adding `+Inf` and `-Inf` together.

### 6.16.10.1. CMLA (vector, vector, vector, immediate), setting inactive to zero

Instances
svfloat16_t <b>svcmla</b> [_f16]_z(svbool_t pg, svfloat16_t op1, svfloat16_t op2, svfloat16_t op3, uint64_t imm_rotation)
svfloat32_t <b>svcmla</b> [_f32]_z(svbool_t pg, svfloat32_t op1, svfloat32_t op2, svfloat32_t op3, uint64_t imm_rotation)
svfloat64_t <b>svcmla</b> [_f64]_z(svbool_t pg, svfloat64_t op1, svfloat64_t op2, svfloat64_t op3, uint64_t imm_rotation)

### 6.16.10.2. CMLA (vector, vector, vector, immediate), merging with first input

Instances
svfloat16_t <b>svcmla</b> [_f16]_m(svbool_t pg, svfloat16_t op1, svfloat16_t op2, svfloat16_t op3, uint64_t imm_rotation)
svfloat32_t <b>svcmla</b> [_f32]_m(svbool_t pg, svfloat32_t op1, svfloat32_t op2, svfloat32_t op3, uint64_t imm_rotation)
svfloat64_t <b>svcmla</b> [_f64]_m(svbool_t pg, svfloat64_t op1, svfloat64_t op2, svfloat64_t op3, uint64_t imm_rotation)

### 6.16.10.3. CMLA (vector, vector, vector, immediate), setting inactive to unknown

Instances
svfloat16_t <b>svcmla</b> [_f16]_x(svbool_t pg, svfloat16_t op1, svfloat16_t op2, svfloat16_t op3, uint64_t imm_rotation)
svfloat32_t <b>svcmla</b> [_f32]_x(svbool_t pg, svfloat32_t op1, svfloat32_t op2, svfloat32_t op3, uint64_t imm_rotation)
svfloat64_t <b>svcmla</b> [_f64]_x(svbool_t pg, svfloat64_t op1, svfloat64_t op2, svfloat64_t op3, uint64_t imm_rotation)

### 6.16.10.4. CMLA (vector, vector, vector, lane, immediate)

Instances
svfloat16_t <b>svcmla_lane</b> [_f16](svfloat16_t op1, svfloat16_t op2, svfloat16_t op3, uint64_t imm_index, uint64_t imm_rotation)
svfloat32_t <b>svcmla_lane</b> [_f32](svfloat32_t op1, svfloat32_t op2, svfloat32_t op3, uint64_t imm_index, uint64_t imm_rotation)

## 6.16.11. MSB: Fused floating-point subtraction of product (multiplied first)

These functions multiply the first two floating-point inputs and subtract the result from the third input. There is no intermediate rounding step after the multiplication.

Signaling NaNs trigger an IEEE Invalid exception but quiet NaNs do not. Multiplying an infinity by zero also triggers an IEEE Invalid exception, as does subtracting an infinity from itself.

### 6.16.11.1. MSB (vector, vector, vector), setting inactive to zero

Instances
svfloat16_t <b>svmsb</b> [_f16]_z(svbool_t pg, svfloat16_t op1, svfloat16_t op2,

**Instances**

```

svfloat16_t op3)
svfloat32_t svmsb[_f32]_z(svbool_t pg, svfloat32_t op1, svfloat32_t op2,
svfloat32_t op3)
svfloat64_t svmsb[_f64]_z(svbool_t pg, svfloat64_t op1, svfloat64_t op2,
svfloat64_t op3)

```

**6.16.11.2. MSB (vector, vector, vector), merging with first input****Instances**

```

svfloat16_t svmsb[_f16]_m(svbool_t pg, svfloat16_t op1, svfloat16_t op2,
svfloat16_t op3)
svfloat32_t svmsb[_f32]_m(svbool_t pg, svfloat32_t op1, svfloat32_t op2,
svfloat32_t op3)
svfloat64_t svmsb[_f64]_m(svbool_t pg, svfloat64_t op1, svfloat64_t op2,
svfloat64_t op3)

```

**6.16.11.3. MSB (vector, vector, vector), setting inactive to unknown****Instances**

```

svfloat16_t svmsb[_f16]_x(svbool_t pg, svfloat16_t op1, svfloat16_t op2,
svfloat16_t op3)
svfloat32_t svmsb[_f32]_x(svbool_t pg, svfloat32_t op1, svfloat32_t op2,
svfloat32_t op3)
svfloat64_t svmsb[_f64]_x(svbool_t pg, svfloat64_t op1, svfloat64_t op2,
svfloat64_t op3)

```

**6.16.11.4. MSB (vector, vector, scalar), setting inactive to zero****Instances**

```

svfloat16_t svmsb[_n_f16]_z(svbool_t pg, svfloat16_t op1, svfloat16_t op2,
float16_t op3)
svfloat32_t svmsb[_n_f32]_z(svbool_t pg, svfloat32_t op1, svfloat32_t op2,
float32_t op3)
svfloat64_t svmsb[_n_f64]_z(svbool_t pg, svfloat64_t op1, svfloat64_t op2,
float64_t op3)

```

**6.16.11.5. MSB (vector, vector, scalar), merging with first input****Instances**

```

svfloat16_t svmsb[_n_f16]_m(svbool_t pg, svfloat16_t op1, svfloat16_t op2,
float16_t op3)
svfloat32_t svmsb[_n_f32]_m(svbool_t pg, svfloat32_t op1, svfloat32_t op2,
float32_t op3)
svfloat64_t svmsb[_n_f64]_m(svbool_t pg, svfloat64_t op1, svfloat64_t op2,
float64_t op3)

```

**6.16.11.6. MSB (vector, vector, scalar), setting inactive to unknown****Instances**

```

svfloat16_t svmsb[_n_f16]_x(svbool_t pg, svfloat16_t op1, svfloat16_t op2,
float16_t op3)
svfloat32_t svmsb[_n_f32]_x(svbool_t pg, svfloat32_t op1, svfloat32_t op2,

```

Instances
<pre> float32_t op3) svfloat64_t svmsb[_n_f64]_x(svbool_t pg, svfloat64_t op1, svfloat64_t op2, float64_t op3) </pre>

### 6.16.12. MLS: Fused floating-point subtraction of product (minuend first)

These functions multiply the second and third floating-point inputs and subtract the result from the first input. There is no intermediate rounding step after the multiplication.

The `_lane` forms of the functions take one element in each 128-bit quadword of the third input and replicate it to fill the rest of the quadword. The `imm_index` parameter specifies the element within each quadword that the functions should replicate. This index must be an integer constant expression in the range  $[0, 128/N]$ , where  $N$  is the number of bits in each element.

Signaling NaNs trigger an IEEE Invalid exception but quiet NaNs do not. Multiplying an infinity by zero also triggers an IEEE Invalid exception, as does subtracting an infinity from itself.

#### 6.16.12.1. MLS (vector, vector, vector), setting inactive to zero

Instances
<pre> svfloat16_t svmls[_f16]_z(svbool_t pg, svfloat16_t op1, svfloat16_t op2, svfloat16_t op3) svfloat32_t svmls[_f32]_z(svbool_t pg, svfloat32_t op1, svfloat32_t op2, svfloat32_t op3) svfloat64_t svmls[_f64]_z(svbool_t pg, svfloat64_t op1, svfloat64_t op2, svfloat64_t op3) </pre>

#### 6.16.12.2. MLS (vector, vector, vector), merging with first input

Instances
<pre> svfloat16_t svmls[_f16]_m(svbool_t pg, svfloat16_t op1, svfloat16_t op2, svfloat16_t op3) svfloat32_t svmls[_f32]_m(svbool_t pg, svfloat32_t op1, svfloat32_t op2, svfloat32_t op3) svfloat64_t svmls[_f64]_m(svbool_t pg, svfloat64_t op1, svfloat64_t op2, svfloat64_t op3) </pre>

#### 6.16.12.3. MLS (vector, vector, vector), setting inactive to unknown

Instances
<pre> svfloat16_t svmls[_f16]_x(svbool_t pg, svfloat16_t op1, svfloat16_t op2, svfloat16_t op3) svfloat32_t svmls[_f32]_x(svbool_t pg, svfloat32_t op1, svfloat32_t op2, svfloat32_t op3) svfloat64_t svmls[_f64]_x(svbool_t pg, svfloat64_t op1, svfloat64_t op2, svfloat64_t op3) </pre>

#### 6.16.12.4. MLS (vector, vector, scalar), setting inactive to zero

Instances
<pre> svfloat16_t svmls[_n_f16]_z(svbool_t pg, svfloat16_t op1, svfloat16_t op2, </pre>

Instances
<pre> float16_t op3) svfloat32_t svmls[_n_f32]_z(svbool_t pg, svfloat32_t op1, svfloat32_t op2, float32_t op3) svfloat64_t svmls[_n_f64]_z(svbool_t pg, svfloat64_t op1, svfloat64_t op2, float64_t op3) </pre>

#### 6.16.12.5. MLS (vector, vector, scalar), merging with first input

Instances
<pre> svfloat16_t svmls[_n_f16]_m(svbool_t pg, svfloat16_t op1, svfloat16_t op2, float16_t op3) svfloat32_t svmls[_n_f32]_m(svbool_t pg, svfloat32_t op1, svfloat32_t op2, float32_t op3) svfloat64_t svmls[_n_f64]_m(svbool_t pg, svfloat64_t op1, svfloat64_t op2, float64_t op3) </pre>

#### 6.16.12.6. MLS (vector, vector, scalar), setting inactive to unknown

Instances
<pre> svfloat16_t svmls[_n_f16]_x(svbool_t pg, svfloat16_t op1, svfloat16_t op2, float16_t op3) svfloat32_t svmls[_n_f32]_x(svbool_t pg, svfloat32_t op1, svfloat32_t op2, float32_t op3) svfloat64_t svmls[_n_f64]_x(svbool_t pg, svfloat64_t op1, svfloat64_t op2, float64_t op3) </pre>

#### 6.16.12.7. MLS (vector, vector, vector, lane)

Instances
<pre> svfloat16_t svmls_lane[_f16](svfloat16_t op1, svfloat16_t op2, svfloat16_t op3, uint64_t imm_index) svfloat32_t svmls_lane[_f32](svfloat32_t op1, svfloat32_t op2, svfloat32_t op3, uint64_t imm_index) svfloat64_t svmls_lane[_f64](svfloat64_t op1, svfloat64_t op2, svfloat64_t op3, uint64_t imm_index) </pre>

### 6.16.13. NMAD: Fused floating-point addition of product, negated (multiplicand first)

These functions multiply the first two floating-point inputs, add the product to the third input, and then negate the result. There is no intermediate rounding step after the multiplication.

Signaling NaNs trigger an IEEE Invalid exception but quiet NaNs do not. Multiplying an infinity by zero also triggers an IEEE Invalid exception, as does adding +Inf and -Inf together.

#### 6.16.13.1. NMAD (vector, vector, vector), setting inactive to zero

Instances
<pre> svfloat16_t svnmad[_f16]_z(svbool_t pg, svfloat16_t op1, svfloat16_t op2, svfloat16_t op3) svfloat32_t svnmad[_f32]_z(svbool_t pg, svfloat32_t op1, svfloat32_t op2, svfloat32_t op3) </pre>

Instances
svfloat64_t <b>svnmad</b> [_f64]_z(svbool_t pg, svfloat64_t op1, svfloat64_t op2, svfloat64_t op3)

### 6.16.13.2. NMAD (vector, vector, vector), merging with first input

Instances
svfloat16_t <b>svnmad</b> [_f16]_m(svbool_t pg, svfloat16_t op1, svfloat16_t op2, svfloat16_t op3)
svfloat32_t <b>svnmad</b> [_f32]_m(svbool_t pg, svfloat32_t op1, svfloat32_t op2, svfloat32_t op3)
svfloat64_t <b>svnmad</b> [_f64]_m(svbool_t pg, svfloat64_t op1, svfloat64_t op2, svfloat64_t op3)

### 6.16.13.3. NMAD (vector, vector, vector), setting inactive to unknown

Instances
svfloat16_t <b>svnmad</b> [_f16]_x(svbool_t pg, svfloat16_t op1, svfloat16_t op2, svfloat16_t op3)
svfloat32_t <b>svnmad</b> [_f32]_x(svbool_t pg, svfloat32_t op1, svfloat32_t op2, svfloat32_t op3)
svfloat64_t <b>svnmad</b> [_f64]_x(svbool_t pg, svfloat64_t op1, svfloat64_t op2, svfloat64_t op3)

### 6.16.13.4. NMAD (vector, vector, scalar), setting inactive to zero

Instances
svfloat16_t <b>svnmad</b> [_n_f16]_z(svbool_t pg, svfloat16_t op1, svfloat16_t op2, float16_t op3)
svfloat32_t <b>svnmad</b> [_n_f32]_z(svbool_t pg, svfloat32_t op1, svfloat32_t op2, float32_t op3)
svfloat64_t <b>svnmad</b> [_n_f64]_z(svbool_t pg, svfloat64_t op1, svfloat64_t op2, float64_t op3)

### 6.16.13.5. NMAD (vector, vector, scalar), merging with first input

Instances
svfloat16_t <b>svnmad</b> [_n_f16]_m(svbool_t pg, svfloat16_t op1, svfloat16_t op2, float16_t op3)
svfloat32_t <b>svnmad</b> [_n_f32]_m(svbool_t pg, svfloat32_t op1, svfloat32_t op2, float32_t op3)
svfloat64_t <b>svnmad</b> [_n_f64]_m(svbool_t pg, svfloat64_t op1, svfloat64_t op2, float64_t op3)

### 6.16.13.6. NMAD (vector, vector, scalar), setting inactive to unknown

Instances
svfloat16_t <b>svnmad</b> [_n_f16]_x(svbool_t pg, svfloat16_t op1, svfloat16_t op2, float16_t op3)
svfloat32_t <b>svnmad</b> [_n_f32]_x(svbool_t pg, svfloat32_t op1, svfloat32_t op2, float32_t op3)
svfloat64_t <b>svnmad</b> [_n_f64]_x(svbool_t pg, svfloat64_t op1, svfloat64_t op2, float64_t op3)

## 6.16.14. NMLA: Fused floating-point addition of product, negated (addend first)

These functions multiply the second and third floating-point inputs, add the product to the first input, then negate the result. There is no intermediate rounding step after the multiplication.

Signaling NaNs trigger an IEEE Invalid exception but quiet NaNs do not. Multiplying an infinity by zero also triggers an IEEE Invalid exception, as does adding +Inf and -Inf together.

### 6.16.14.1. NMLA (vector, vector, vector), setting inactive to zero

Instances
svfloat16_t <b>svnmla</b> [_f16]_z(svbool_t pg, svfloat16_t op1, svfloat16_t op2, svfloat16_t op3)
svfloat32_t <b>svnmla</b> [_f32]_z(svbool_t pg, svfloat32_t op1, svfloat32_t op2, svfloat32_t op3)
svfloat64_t <b>svnmla</b> [_f64]_z(svbool_t pg, svfloat64_t op1, svfloat64_t op2, svfloat64_t op3)

### 6.16.14.2. NMLA (vector, vector, vector), merging with first input

Instances
svfloat16_t <b>svnmla</b> [_f16]_m(svbool_t pg, svfloat16_t op1, svfloat16_t op2, svfloat16_t op3)
svfloat32_t <b>svnmla</b> [_f32]_m(svbool_t pg, svfloat32_t op1, svfloat32_t op2, svfloat32_t op3)
svfloat64_t <b>svnmla</b> [_f64]_m(svbool_t pg, svfloat64_t op1, svfloat64_t op2, svfloat64_t op3)

### 6.16.14.3. NMLA (vector, vector, vector), setting inactive to unknown

Instances
svfloat16_t <b>svnmla</b> [_f16]_x(svbool_t pg, svfloat16_t op1, svfloat16_t op2, svfloat16_t op3)
svfloat32_t <b>svnmla</b> [_f32]_x(svbool_t pg, svfloat32_t op1, svfloat32_t op2, svfloat32_t op3)
svfloat64_t <b>svnmla</b> [_f64]_x(svbool_t pg, svfloat64_t op1, svfloat64_t op2, svfloat64_t op3)

### 6.16.14.4. NMLA (vector, vector, scalar), setting inactive to zero

Instances
svfloat16_t <b>svnmla</b> [_n_f16]_z(svbool_t pg, svfloat16_t op1, svfloat16_t op2, float16_t op3)
svfloat32_t <b>svnmla</b> [_n_f32]_z(svbool_t pg, svfloat32_t op1, svfloat32_t op2, float32_t op3)
svfloat64_t <b>svnmla</b> [_n_f64]_z(svbool_t pg, svfloat64_t op1, svfloat64_t op2, float64_t op3)

### 6.16.14.5. NMLA (vector, vector, scalar), merging with first input

Instances
svfloat16_t <b>svnmla</b> [_n_f16]_m(svbool_t pg, svfloat16_t op1, svfloat16_t op2, float16_t op3)

Instances	
	float16_t op3)
svfloat32_t	<b>svnmmla</b> [_n_f32]_m(svbool_t pg, svfloat32_t op1, svfloat32_t op2, float32_t op3)
svfloat64_t	<b>svnmmla</b> [_n_f64]_m(svbool_t pg, svfloat64_t op1, svfloat64_t op2, float64_t op3)

#### 6.16.14.6. NMLA (vector, vector, scalar), setting inactive to unknown

Instances	
svfloat16_t	<b>svnmmla</b> [_n_f16]_x(svbool_t pg, svfloat16_t op1, svfloat16_t op2, float16_t op3)
svfloat32_t	<b>svnmmla</b> [_n_f32]_x(svbool_t pg, svfloat32_t op1, svfloat32_t op2, float32_t op3)
svfloat64_t	<b>svnmmla</b> [_n_f64]_x(svbool_t pg, svfloat64_t op1, svfloat64_t op2, float64_t op3)

### 6.16.15. NMSB: Fused floating-point subtraction of product, negated (multiplicand first)

These functions multiply the first two floating-point inputs, subtract the product from the third input, then negate the result. There is no intermediate rounding step after the multiplication.

Signaling NaNs trigger an IEEE Invalid exception but quiet NaNs do not. Multiplying an infinity by zero also triggers an IEEE Invalid exception, as does subtracting an infinity from itself.

#### 6.16.15.1. NMSB (vector, vector, vector), setting inactive to zero

Instances	
svfloat16_t	<b>svnmsb</b> [_f16]_z(svbool_t pg, svfloat16_t op1, svfloat16_t op2, svfloat16_t op3)
svfloat32_t	<b>svnmsb</b> [_f32]_z(svbool_t pg, svfloat32_t op1, svfloat32_t op2, svfloat32_t op3)
svfloat64_t	<b>svnmsb</b> [_f64]_z(svbool_t pg, svfloat64_t op1, svfloat64_t op2, svfloat64_t op3)

#### 6.16.15.2. NMSB (vector, vector, vector), merging with first input

Instances	
svfloat16_t	<b>svnmsb</b> [_f16]_m(svbool_t pg, svfloat16_t op1, svfloat16_t op2, svfloat16_t op3)
svfloat32_t	<b>svnmsb</b> [_f32]_m(svbool_t pg, svfloat32_t op1, svfloat32_t op2, svfloat32_t op3)
svfloat64_t	<b>svnmsb</b> [_f64]_m(svbool_t pg, svfloat64_t op1, svfloat64_t op2, svfloat64_t op3)

#### 6.16.15.3. NMSB (vector, vector, vector), setting inactive to unknown

Instances	
svfloat16_t	<b>svnmsb</b> [_f16]_x(svbool_t pg, svfloat16_t op1, svfloat16_t op2, svfloat16_t op3)
svfloat32_t	<b>svnmsb</b> [_f32]_x(svbool_t pg, svfloat32_t op1, svfloat32_t op2, svfloat32_t op3)



**Instances**

```
svfloat64_t svnmsb[_f64]_x(svbool_t pg, svfloat64_t op1, svfloat64_t op2,
                             svfloat64_t op3)
```

**6.16.15.4. NMSB (vector, vector, scalar), setting inactive to zero****Instances**

```
svfloat16_t svnmsb[_n_f16]_z(svbool_t pg, svfloat16_t op1, svfloat16_t op2,
                               float16_t op3)
svfloat32_t svnmsb[_n_f32]_z(svbool_t pg, svfloat32_t op1, svfloat32_t op2,
                               float32_t op3)
svfloat64_t svnmsb[_n_f64]_z(svbool_t pg, svfloat64_t op1, svfloat64_t op2,
                               float64_t op3)
```

**6.16.15.5. NMSB (vector, vector, scalar), merging with first input****Instances**

```
svfloat16_t svnmsb[_n_f16]_m(svbool_t pg, svfloat16_t op1, svfloat16_t op2,
                               float16_t op3)
svfloat32_t svnmsb[_n_f32]_m(svbool_t pg, svfloat32_t op1, svfloat32_t op2,
                               float32_t op3)
svfloat64_t svnmsb[_n_f64]_m(svbool_t pg, svfloat64_t op1, svfloat64_t op2,
                               float64_t op3)
```

**6.16.15.6. NMSB (vector, vector, scalar), setting inactive to unknown****Instances**

```
svfloat16_t svnmsb[_n_f16]_x(svbool_t pg, svfloat16_t op1, svfloat16_t op2,
                               float16_t op3)
svfloat32_t svnmsb[_n_f32]_x(svbool_t pg, svfloat32_t op1, svfloat32_t op2,
                               float32_t op3)
svfloat64_t svnmsb[_n_f64]_x(svbool_t pg, svfloat64_t op1, svfloat64_t op2,
                               float64_t op3)
```

**6.16.16. NMLS: Fused floating-point subtraction of product, negated (minuend first)**

These functions multiply the second and third floating-point inputs, subtract the product from the first input, then negate the result. There is no intermediate rounding step after the multiplication.

Signaling NaNs trigger an IEEE Invalid exception but quiet NaNs do not. Multiplying an infinity by zero also triggers an IEEE Invalid exception, as does subtracting an infinity from itself.

**6.16.16.1. NMLS (vector, vector, vector), setting inactive to zero****Instances**

```
svfloat16_t svnmmls[_f16]_z(svbool_t pg, svfloat16_t op1, svfloat16_t op2,
                              svfloat16_t op3)
svfloat32_t svnmmls[_f32]_z(svbool_t pg, svfloat32_t op1, svfloat32_t op2,
                              svfloat32_t op3)
svfloat64_t svnmmls[_f64]_z(svbool_t pg, svfloat64_t op1, svfloat64_t op2,
                              svfloat64_t op3)
```

### 6.16.16.2. NMLS (vector, vector, vector), merging with first input

Instances
svfloat16_t <b>svnmls</b> [_f16]_m(svbool_t pg, svfloat16_t op1, svfloat16_t op2, svfloat16_t op3)
svfloat32_t <b>svnmls</b> [_f32]_m(svbool_t pg, svfloat32_t op1, svfloat32_t op2, svfloat32_t op3)
svfloat64_t <b>svnmls</b> [_f64]_m(svbool_t pg, svfloat64_t op1, svfloat64_t op2, svfloat64_t op3)

### 6.16.16.3. NMLS (vector, vector, vector), setting inactive to unknown

Instances
svfloat16_t <b>svnmls</b> [_f16]_x(svbool_t pg, svfloat16_t op1, svfloat16_t op2, svfloat16_t op3)
svfloat32_t <b>svnmls</b> [_f32]_x(svbool_t pg, svfloat32_t op1, svfloat32_t op2, svfloat32_t op3)
svfloat64_t <b>svnmls</b> [_f64]_x(svbool_t pg, svfloat64_t op1, svfloat64_t op2, svfloat64_t op3)

### 6.16.16.4. NMLS (vector, vector, scalar), setting inactive to zero

Instances
svfloat16_t <b>svnmls</b> [_n_f16]_z(svbool_t pg, svfloat16_t op1, svfloat16_t op2, float16_t op3)
svfloat32_t <b>svnmls</b> [_n_f32]_z(svbool_t pg, svfloat32_t op1, svfloat32_t op2, float32_t op3)
svfloat64_t <b>svnmls</b> [_n_f64]_z(svbool_t pg, svfloat64_t op1, svfloat64_t op2, float64_t op3)

### 6.16.16.5. NMLS (vector, vector, scalar), merging with first input

Instances
svfloat16_t <b>svnmls</b> [_n_f16]_m(svbool_t pg, svfloat16_t op1, svfloat16_t op2, float16_t op3)
svfloat32_t <b>svnmls</b> [_n_f32]_m(svbool_t pg, svfloat32_t op1, svfloat32_t op2, float32_t op3)
svfloat64_t <b>svnmls</b> [_n_f64]_m(svbool_t pg, svfloat64_t op1, svfloat64_t op2, float64_t op3)

### 6.16.16.6. NMLS (vector, vector, scalar), setting inactive to unknown

Instances
svfloat16_t <b>svnmls</b> [_n_f16]_x(svbool_t pg, svfloat16_t op1, svfloat16_t op2, float16_t op3)
svfloat32_t <b>svnmls</b> [_n_f32]_x(svbool_t pg, svfloat32_t op1, svfloat32_t op2, float32_t op3)
svfloat64_t <b>svnmls</b> [_n_f64]_x(svbool_t pg, svfloat64_t op1, svfloat64_t op2, float64_t op3)

## 6.16.17. DIV: Floating-point division

These functions divide the first floating-point value by the second input.

Signaling NaNs trigger an IEEE Invalid exception but quiet NaNs do not. Dividing an infinity by an infinity or a zero by a zero also triggers an IEEE Invalid exception. Other divisions by zero trigger an IEEE DivideByZero exception.

#### 6.16.17.1. DIV (vector, vector), setting inactive to zero

Instances	
svfloat16_t	<b>svdiv</b> [_f16]_z(svbool_t <i>pg</i> , svfloat16_t <i>op1</i> , svfloat16_t <i>op2</i> )
svfloat32_t	<b>svdiv</b> [_f32]_z(svbool_t <i>pg</i> , svfloat32_t <i>op1</i> , svfloat32_t <i>op2</i> )
svfloat64_t	<b>svdiv</b> [_f64]_z(svbool_t <i>pg</i> , svfloat64_t <i>op1</i> , svfloat64_t <i>op2</i> )

#### 6.16.17.2. DIV (vector, vector), merging with first input

Instances	
svfloat16_t	<b>svdiv</b> [_f16]_m(svbool_t <i>pg</i> , svfloat16_t <i>op1</i> , svfloat16_t <i>op2</i> )
svfloat32_t	<b>svdiv</b> [_f32]_m(svbool_t <i>pg</i> , svfloat32_t <i>op1</i> , svfloat32_t <i>op2</i> )
svfloat64_t	<b>svdiv</b> [_f64]_m(svbool_t <i>pg</i> , svfloat64_t <i>op1</i> , svfloat64_t <i>op2</i> )

#### 6.16.17.3. DIV (vector, vector), setting inactive to unknown

Instances	
svfloat16_t	<b>svdiv</b> [_f16]_x(svbool_t <i>pg</i> , svfloat16_t <i>op1</i> , svfloat16_t <i>op2</i> )
svfloat32_t	<b>svdiv</b> [_f32]_x(svbool_t <i>pg</i> , svfloat32_t <i>op1</i> , svfloat32_t <i>op2</i> )
svfloat64_t	<b>svdiv</b> [_f64]_x(svbool_t <i>pg</i> , svfloat64_t <i>op1</i> , svfloat64_t <i>op2</i> )

#### 6.16.17.4. DIV (vector, scalar), setting inactive to zero

Instances	
svfloat16_t	<b>svdiv</b> [_n_f16]_z(svbool_t <i>pg</i> , svfloat16_t <i>op1</i> , float16_t <i>op2</i> )
svfloat32_t	<b>svdiv</b> [_n_f32]_z(svbool_t <i>pg</i> , svfloat32_t <i>op1</i> , float32_t <i>op2</i> )
svfloat64_t	<b>svdiv</b> [_n_f64]_z(svbool_t <i>pg</i> , svfloat64_t <i>op1</i> , float64_t <i>op2</i> )

#### 6.16.17.5. DIV (vector, scalar), merging with first input

Instances	
svfloat16_t	<b>svdiv</b> [_n_f16]_m(svbool_t <i>pg</i> , svfloat16_t <i>op1</i> , float16_t <i>op2</i> )
svfloat32_t	<b>svdiv</b> [_n_f32]_m(svbool_t <i>pg</i> , svfloat32_t <i>op1</i> , float32_t <i>op2</i> )
svfloat64_t	<b>svdiv</b> [_n_f64]_m(svbool_t <i>pg</i> , svfloat64_t <i>op1</i> , float64_t <i>op2</i> )

#### 6.16.17.6. DIV (vector, scalar), setting inactive to unknown

Instances	
svfloat16_t	<b>svdiv</b> [_n_f16]_x(svbool_t <i>pg</i> , svfloat16_t <i>op1</i> , float16_t <i>op2</i> )
svfloat32_t	<b>svdiv</b> [_n_f32]_x(svbool_t <i>pg</i> , svfloat32_t <i>op1</i> , float32_t <i>op2</i> )
svfloat64_t	<b>svdiv</b> [_n_f64]_x(svbool_t <i>pg</i> , svfloat64_t <i>op1</i> , float64_t <i>op2</i> )

### 6.16.18. DIVR: Floating-point division, reversed

These functions divide the second floating-point value by the first input.

Signaling NaNs trigger an IEEE Invalid exception but quiet NaNs do not. Dividing an infinity by an infinity or a zero by a zero also triggers an IEEE Invalid exception. Other divisions by zero trigger an IEEE DivideByZero exception.

### 6.16.18.1. DIVR (vector, vector), setting inactive to zero

Instances	
svfloat16_t	<b>svdivr</b> [_f16]_z(svbool_t <i>pg</i> , svfloat16_t <i>op1</i> , svfloat16_t <i>op2</i> )
svfloat32_t	<b>svdivr</b> [_f32]_z(svbool_t <i>pg</i> , svfloat32_t <i>op1</i> , svfloat32_t <i>op2</i> )
svfloat64_t	<b>svdivr</b> [_f64]_z(svbool_t <i>pg</i> , svfloat64_t <i>op1</i> , svfloat64_t <i>op2</i> )

### 6.16.18.2. DIVR (vector, vector), merging with first input

Instances	
svfloat16_t	<b>svdivr</b> [_f16]_m(svbool_t <i>pg</i> , svfloat16_t <i>op1</i> , svfloat16_t <i>op2</i> )
svfloat32_t	<b>svdivr</b> [_f32]_m(svbool_t <i>pg</i> , svfloat32_t <i>op1</i> , svfloat32_t <i>op2</i> )
svfloat64_t	<b>svdivr</b> [_f64]_m(svbool_t <i>pg</i> , svfloat64_t <i>op1</i> , svfloat64_t <i>op2</i> )

### 6.16.18.3. DIVR (vector, vector), setting inactive to unknown

Instances	
svfloat16_t	<b>svdivr</b> [_f16]_x(svbool_t <i>pg</i> , svfloat16_t <i>op1</i> , svfloat16_t <i>op2</i> )
svfloat32_t	<b>svdivr</b> [_f32]_x(svbool_t <i>pg</i> , svfloat32_t <i>op1</i> , svfloat32_t <i>op2</i> )
svfloat64_t	<b>svdivr</b> [_f64]_x(svbool_t <i>pg</i> , svfloat64_t <i>op1</i> , svfloat64_t <i>op2</i> )

### 6.16.18.4. DIVR (vector, scalar), setting inactive to zero

Instances	
svfloat16_t	<b>svdivr</b> [_n_f16]_z(svbool_t <i>pg</i> , svfloat16_t <i>op1</i> , float16_t <i>op2</i> )
svfloat32_t	<b>svdivr</b> [_n_f32]_z(svbool_t <i>pg</i> , svfloat32_t <i>op1</i> , float32_t <i>op2</i> )
svfloat64_t	<b>svdivr</b> [_n_f64]_z(svbool_t <i>pg</i> , svfloat64_t <i>op1</i> , float64_t <i>op2</i> )

### 6.16.18.5. DIVR (vector, scalar), merging with first input

Instances	
svfloat16_t	<b>svdivr</b> [_n_f16]_m(svbool_t <i>pg</i> , svfloat16_t <i>op1</i> , float16_t <i>op2</i> )
svfloat32_t	<b>svdivr</b> [_n_f32]_m(svbool_t <i>pg</i> , svfloat32_t <i>op1</i> , float32_t <i>op2</i> )
svfloat64_t	<b>svdivr</b> [_n_f64]_m(svbool_t <i>pg</i> , svfloat64_t <i>op1</i> , float64_t <i>op2</i> )

### 6.16.18.6. DIVR (vector, scalar), setting inactive to unknown

Instances	
svfloat16_t	<b>svdivr</b> [_n_f16]_x(svbool_t <i>pg</i> , svfloat16_t <i>op1</i> , float16_t <i>op2</i> )
svfloat32_t	<b>svdivr</b> [_n_f32]_x(svbool_t <i>pg</i> , svfloat32_t <i>op1</i> , float32_t <i>op2</i> )
svfloat64_t	<b>svdivr</b> [_n_f64]_x(svbool_t <i>pg</i> , svfloat64_t <i>op1</i> , float64_t <i>op2</i> )

## 6.16.19. MAX: Floating-point maximum

These functions compute the maximum of two floating-point inputs. If one input is a NaN, the result is also a NaN.

Signaling NaNs trigger an IEEE Invalid exception but quiet NaNs do not.

### 6.16.19.1. MAX (vector, vector), setting inactive to zero

Instances	
svfloat16_t	<b>svmax</b> [_f16]_z(svbool_t <i>pg</i> , svfloat16_t <i>op1</i> , svfloat16_t <i>op2</i> )

Instances
svfloat32_t <b>svmax</b> [_f32]_z(svbool_t pg, svfloat32_t op1, svfloat32_t op2)
svfloat64_t <b>svmax</b> [_f64]_z(svbool_t pg, svfloat64_t op1, svfloat64_t op2)

### 6.16.19.2. MAX (vector, vector), merging with first input

Instances
svfloat16_t <b>svmax</b> [_f16]_m(svbool_t pg, svfloat16_t op1, svfloat16_t op2)
svfloat32_t <b>svmax</b> [_f32]_m(svbool_t pg, svfloat32_t op1, svfloat32_t op2)
svfloat64_t <b>svmax</b> [_f64]_m(svbool_t pg, svfloat64_t op1, svfloat64_t op2)

### 6.16.19.3. MAX (vector, vector), setting inactive to unknown

Instances
svfloat16_t <b>svmax</b> [_f16]_x(svbool_t pg, svfloat16_t op1, svfloat16_t op2)
svfloat32_t <b>svmax</b> [_f32]_x(svbool_t pg, svfloat32_t op1, svfloat32_t op2)
svfloat64_t <b>svmax</b> [_f64]_x(svbool_t pg, svfloat64_t op1, svfloat64_t op2)

### 6.16.19.4. MAX (vector, scalar), setting inactive to zero

Instances
svfloat16_t <b>svmax</b> [_n_f16]_z(svbool_t pg, svfloat16_t op1, float16_t op2)
svfloat32_t <b>svmax</b> [_n_f32]_z(svbool_t pg, svfloat32_t op1, float32_t op2)
svfloat64_t <b>svmax</b> [_n_f64]_z(svbool_t pg, svfloat64_t op1, float64_t op2)

### 6.16.19.5. MAX (vector, scalar), merging with first input

Instances
svfloat16_t <b>svmax</b> [_n_f16]_m(svbool_t pg, svfloat16_t op1, float16_t op2)
svfloat32_t <b>svmax</b> [_n_f32]_m(svbool_t pg, svfloat32_t op1, float32_t op2)
svfloat64_t <b>svmax</b> [_n_f64]_m(svbool_t pg, svfloat64_t op1, float64_t op2)

### 6.16.19.6. MAX (vector, scalar), setting inactive to unknown

Instances
svfloat16_t <b>svmax</b> [_n_f16]_x(svbool_t pg, svfloat16_t op1, float16_t op2)
svfloat32_t <b>svmax</b> [_n_f32]_x(svbool_t pg, svfloat32_t op1, float32_t op2)
svfloat64_t <b>svmax</b> [_n_f64]_x(svbool_t pg, svfloat64_t op1, float64_t op2)

## 6.16.20. MAXNM: Floating-point maximum number

These functions compute the maximum of two floating-point inputs. If both inputs are NaNs, the result is also a NaN. If only one input is a NaN, the result is the other input.

Signaling NaNs trigger an IEEE Invalid exception but quiet NaNs do not.

### 6.16.20.1. MAXNM (vector, vector), setting inactive to zero

Instances
svfloat16_t <b>svmaxnm</b> [_f16]_z(svbool_t pg, svfloat16_t op1, svfloat16_t op2)

Instances
svfloat32_t <b>svmaxnm</b> [_f32]_z(svbool_t pg, svfloat32_t op1, svfloat32_t op2)
svfloat64_t <b>svmaxnm</b> [_f64]_z(svbool_t pg, svfloat64_t op1, svfloat64_t op2)

#### 6.16.20.2. MAXNM (vector, vector), merging with first input

Instances
svfloat16_t <b>svmaxnm</b> [_f16]_m(svbool_t pg, svfloat16_t op1, svfloat16_t op2)
svfloat32_t <b>svmaxnm</b> [_f32]_m(svbool_t pg, svfloat32_t op1, svfloat32_t op2)
svfloat64_t <b>svmaxnm</b> [_f64]_m(svbool_t pg, svfloat64_t op1, svfloat64_t op2)

#### 6.16.20.3. MAXNM (vector, vector), setting inactive to unknown

Instances
svfloat16_t <b>svmaxnm</b> [_f16]_x(svbool_t pg, svfloat16_t op1, svfloat16_t op2)
svfloat32_t <b>svmaxnm</b> [_f32]_x(svbool_t pg, svfloat32_t op1, svfloat32_t op2)
svfloat64_t <b>svmaxnm</b> [_f64]_x(svbool_t pg, svfloat64_t op1, svfloat64_t op2)

#### 6.16.20.4. MAXNM (vector, scalar), setting inactive to zero

Instances
svfloat16_t <b>svmaxnm</b> [_n_f16]_z(svbool_t pg, svfloat16_t op1, float16_t op2)
svfloat32_t <b>svmaxnm</b> [_n_f32]_z(svbool_t pg, svfloat32_t op1, float32_t op2)
svfloat64_t <b>svmaxnm</b> [_n_f64]_z(svbool_t pg, svfloat64_t op1, float64_t op2)

#### 6.16.20.5. MAXNM (vector, scalar), merging with first input

Instances
svfloat16_t <b>svmaxnm</b> [_n_f16]_m(svbool_t pg, svfloat16_t op1, float16_t op2)
svfloat32_t <b>svmaxnm</b> [_n_f32]_m(svbool_t pg, svfloat32_t op1, float32_t op2)
svfloat64_t <b>svmaxnm</b> [_n_f64]_m(svbool_t pg, svfloat64_t op1, float64_t op2)

#### 6.16.20.6. MAXNM (vector, scalar), setting inactive to unknown

Instances
svfloat16_t <b>svmaxnm</b> [_n_f16]_x(svbool_t pg, svfloat16_t op1, float16_t op2)
svfloat32_t <b>svmaxnm</b> [_n_f32]_x(svbool_t pg, svfloat32_t op1, float32_t op2)
svfloat64_t <b>svmaxnm</b> [_n_f64]_x(svbool_t pg, svfloat64_t op1, float64_t op2)

### 6.16.21. MIN: Floating-point minimum

These functions compute the minimum of two floating-point inputs. If one input is a NaN, the result is also a NaN.

Signaling NaNs trigger an IEEE Invalid exception but quiet NaNs do not.

#### 6.16.21.1. MIN (vector, vector), setting inactive to zero

Instances
svfloat16_t <b>svmin</b> [_f16]_z(svbool_t pg, svfloat16_t op1, svfloat16_t op2)

**Instances**

```
svfloat32_t svmin[_f32]_z(svbool_t pg, svfloat32_t op1, svfloat32_t op2)
svfloat64_t svmin[_f64]_z(svbool_t pg, svfloat64_t op1, svfloat64_t op2)
```

**6.16.21.2. MIN (vector, vector), merging with first input****Instances**

```
svfloat16_t svmin[_f16]_m(svbool_t pg, svfloat16_t op1, svfloat16_t op2)
svfloat32_t svmin[_f32]_m(svbool_t pg, svfloat32_t op1, svfloat32_t op2)
svfloat64_t svmin[_f64]_m(svbool_t pg, svfloat64_t op1, svfloat64_t op2)
```

**6.16.21.3. MIN (vector, vector), setting inactive to unknown****Instances**

```
svfloat16_t svmin[_f16]_x(svbool_t pg, svfloat16_t op1, svfloat16_t op2)
svfloat32_t svmin[_f32]_x(svbool_t pg, svfloat32_t op1, svfloat32_t op2)
svfloat64_t svmin[_f64]_x(svbool_t pg, svfloat64_t op1, svfloat64_t op2)
```

**6.16.21.4. MIN (vector, scalar), setting inactive to zero****Instances**

```
svfloat16_t svmin[_n_f16]_z(svbool_t pg, svfloat16_t op1, float16_t op2)
svfloat32_t svmin[_n_f32]_z(svbool_t pg, svfloat32_t op1, float32_t op2)
svfloat64_t svmin[_n_f64]_z(svbool_t pg, svfloat64_t op1, float64_t op2)
```

**6.16.21.5. MIN (vector, scalar), merging with first input****Instances**

```
svfloat16_t svmin[_n_f16]_m(svbool_t pg, svfloat16_t op1, float16_t op2)
svfloat32_t svmin[_n_f32]_m(svbool_t pg, svfloat32_t op1, float32_t op2)
svfloat64_t svmin[_n_f64]_m(svbool_t pg, svfloat64_t op1, float64_t op2)
```

**6.16.21.6. MIN (vector, scalar), setting inactive to unknown****Instances**

```
svfloat16_t svmin[_n_f16]_x(svbool_t pg, svfloat16_t op1, float16_t op2)
svfloat32_t svmin[_n_f32]_x(svbool_t pg, svfloat32_t op1, float32_t op2)
svfloat64_t svmin[_n_f64]_x(svbool_t pg, svfloat64_t op1, float64_t op2)
```

**6.16.22. MINNM: Floating-point minimum number**

These functions compute the minimum of two floating-point inputs. If both inputs are NaNs, the result is also a NaN. If only one input is a NaN, the result is the other input.

Signaling NaNs trigger an IEEE Invalid exception but quiet NaNs do not.

**6.16.22.1. MINNM (vector, vector), setting inactive to zero****Instances**

```
svfloat16_t svminnm[_f16]_z(svbool_t pg, svfloat16_t op1, svfloat16_t op2)
```

Instances
svfloat32_t <b>svminnm</b> [_f32]_z(svbool_t pg, svfloat32_t op1, svfloat32_t op2)
svfloat64_t <b>svminnm</b> [_f64]_z(svbool_t pg, svfloat64_t op1, svfloat64_t op2)

#### 6.16.22.2. MINNM (vector, vector), merging with first input

Instances
svfloat16_t <b>svminnm</b> [_f16]_m(svbool_t pg, svfloat16_t op1, svfloat16_t op2)
svfloat32_t <b>svminnm</b> [_f32]_m(svbool_t pg, svfloat32_t op1, svfloat32_t op2)
svfloat64_t <b>svminnm</b> [_f64]_m(svbool_t pg, svfloat64_t op1, svfloat64_t op2)

#### 6.16.22.3. MINNM (vector, vector), setting inactive to unknown

Instances
svfloat16_t <b>svminnm</b> [_f16]_x(svbool_t pg, svfloat16_t op1, svfloat16_t op2)
svfloat32_t <b>svminnm</b> [_f32]_x(svbool_t pg, svfloat32_t op1, svfloat32_t op2)
svfloat64_t <b>svminnm</b> [_f64]_x(svbool_t pg, svfloat64_t op1, svfloat64_t op2)

#### 6.16.22.4. MINNM (vector, scalar), setting inactive to zero

Instances
svfloat16_t <b>svminnm</b> [_n_f16]_z(svbool_t pg, svfloat16_t op1, float16_t op2)
svfloat32_t <b>svminnm</b> [_n_f32]_z(svbool_t pg, svfloat32_t op1, float32_t op2)
svfloat64_t <b>svminnm</b> [_n_f64]_z(svbool_t pg, svfloat64_t op1, float64_t op2)

#### 6.16.22.5. MINNM (vector, scalar), merging with first input

Instances
svfloat16_t <b>svminnm</b> [_n_f16]_m(svbool_t pg, svfloat16_t op1, float16_t op2)
svfloat32_t <b>svminnm</b> [_n_f32]_m(svbool_t pg, svfloat32_t op1, float32_t op2)
svfloat64_t <b>svminnm</b> [_n_f64]_m(svbool_t pg, svfloat64_t op1, float64_t op2)

#### 6.16.22.6. MINNM (vector, scalar), setting inactive to unknown

Instances
svfloat16_t <b>svminnm</b> [_n_f16]_x(svbool_t pg, svfloat16_t op1, float16_t op2)
svfloat32_t <b>svminnm</b> [_n_f32]_x(svbool_t pg, svfloat32_t op1, float32_t op2)
svfloat64_t <b>svminnm</b> [_n_f64]_x(svbool_t pg, svfloat64_t op1, float64_t op2)

### 6.16.23. SCALE: Floating-point adjust exponent

These functions take a floating-point input and an integer scale. They multiply the floating-point input by 2 to the power of the integer scale and return the result.

Signaling NaNs trigger an IEEE Invalid exception but quiet NaNs do not.

#### 6.16.23.1. SCALE (vector, vector), setting inactive to zero

Instances
svfloat16_t <b>svscale</b> [_f16]_z(svbool_t pg, svfloat16_t op1, svint16_t op2)
svfloat32_t <b>svscale</b> [_f32]_z(svbool_t pg, svfloat32_t op1, svint32_t op2)



**Instances**

```
svfloat64_t svscale[_f64]_z(svbool_t pg, svfloat64_t op1, svint64_t op2)
```

**6.16.23.2. SCALE (vector, vector), merging with first input****Instances**

```
svfloat16_t svscale[_f16]_m(svbool_t pg, svfloat16_t op1, svint16_t op2)
svfloat32_t svscale[_f32]_m(svbool_t pg, svfloat32_t op1, svint32_t op2)
svfloat64_t svscale[_f64]_m(svbool_t pg, svfloat64_t op1, svint64_t op2)
```

**6.16.23.3. SCALE (vector, vector), setting inactive to unknown****Instances**

```
svfloat16_t svscale[_f16]_x(svbool_t pg, svfloat16_t op1, svint16_t op2)
svfloat32_t svscale[_f32]_x(svbool_t pg, svfloat32_t op1, svint32_t op2)
svfloat64_t svscale[_f64]_x(svbool_t pg, svfloat64_t op1, svint64_t op2)
```

**6.16.23.4. SCALE (vector, scalar), setting inactive to zero****Instances**

```
svfloat16_t svscale[_n_f16]_z(svbool_t pg, svfloat16_t op1, int16_t op2)
svfloat32_t svscale[_n_f32]_z(svbool_t pg, svfloat32_t op1, int32_t op2)
svfloat64_t svscale[_n_f64]_z(svbool_t pg, svfloat64_t op1, int64_t op2)
```

**6.16.23.5. SCALE (vector, scalar), merging with first input****Instances**

```
svfloat16_t svscale[_n_f16]_m(svbool_t pg, svfloat16_t op1, int16_t op2)
svfloat32_t svscale[_n_f32]_m(svbool_t pg, svfloat32_t op1, int32_t op2)
svfloat64_t svscale[_n_f64]_m(svbool_t pg, svfloat64_t op1, int64_t op2)
```

**6.16.23.6. SCALE (vector, scalar), setting inactive to unknown****Instances**

```
svfloat16_t svscale[_n_f16]_x(svbool_t pg, svfloat16_t op1, int16_t op2)
svfloat32_t svscale[_n_f32]_x(svbool_t pg, svfloat32_t op1, int32_t op2)
svfloat64_t svscale[_n_f64]_x(svbool_t pg, svfloat64_t op1, int64_t op2)
```

**6.16.24. TSMUL: Floating-point trigonometric starting value**

These functions calculate an initial value for use by the `svtmad` functions. They take a floating-point input and an integer input and return a floating-point result. The result is the square of the floating-point input with the sign bit taken from bit 0 of the integer input.

Signaling NaNs trigger an IEEE Invalid exception but quiet NaNs do not.

**6.16.24.1. TSMUL (vector, vector)****Instances**

```
svfloat16_t svtasmul[_f16](svfloat16_t op1, svuint16_t op2)
svfloat32_t svtasmul[_f32](svfloat32_t op1, svuint32_t op2)
```

Instances
svfloat64_t <b>svtsmul</b> [_f64](svfloat64_t op1, svuint64_t op2)

## 6.16.25. TMAD: Floating-point trigonometric multiply-add coefficient

These functions take two floating-point inputs and an integer constant expression  $I$  in the range [0, 7]. The sign of the second floating-point input selects a table of 8 coefficients, which the integer  $I$  then indexes.

The functions multiply the first input by the absolute value of the second input and add the selected coefficient. The multiplication and addition are a single fused operation; there is no intermediate rounding step.

Signaling NaNs trigger an IEEE Invalid exception but quiet NaNs do not. Multiplying an infinity by zero also triggers an IEEE Invalid exception.

### 6.16.25.1. TMAD (vector, vector, immediate)

Instances
svfloat16_t <b>svtmad</b> [_f16](svfloat16_t op1, svfloat16_t op2, uint64_t imm3)
svfloat32_t <b>svtmad</b> [_f32](svfloat32_t op1, svfloat32_t op2, uint64_t imm3)
svfloat64_t <b>svtmad</b> [_f64](svfloat64_t op1, svfloat64_t op2, uint64_t imm3)

## 6.16.26. TSSEL: Floating-point trigonometric select coefficient

These functions take a floating-point input and an integer input. They use bit 0 of the integer input to select between the floating-point input and 1.0, with a set bit selecting the latter. They then flip the sign bit if bit 1 of the integer input is set.

These functions never raise an IEEE exception.

### 6.16.26.1. TSSEL (vector, vector)

Instances
svfloat16_t <b>svtsel</b> [_f16](svfloat16_t op1, svuint16_t op2)
svfloat32_t <b>svtsel</b> [_f32](svfloat32_t op1, svuint32_t op2)
svfloat64_t <b>svtsel</b> [_f64](svfloat64_t op1, svuint64_t op2)

## 6.16.27. ABS: Floating-point absolute

These functions return a copy of a floating-point input in which the sign bit is clear. They do not raise an exception for any input.

### 6.16.27.1. ABS (vector), setting inactive to zero

Instances
svfloat16_t <b>svabs</b> [_f16]_z(svbool_t pg, svfloat16_t op)
svfloat32_t <b>svabs</b> [_f32]_z(svbool_t pg, svfloat32_t op)
svfloat64_t <b>svabs</b> [_f64]_z(svbool_t pg, svfloat64_t op)

### 6.16.27.2. ABS (vector), merging with separate vector

Instances
svfloat16_t <b>svabs</b> [_f16]_m(svfloat16_t inactive, svbool_t pg, svfloat16_t op)

**Instances**

```
svfloat32_t svabs[_f32]_m(svfloat32_t inactive, svbool_t pg,
                          svfloat32_t op)
svfloat64_t svabs[_f64]_m(svfloat64_t inactive, svbool_t pg,
                          svfloat64_t op)
```

**6.16.27.3. ABS (vector), setting inactive to unknown****Instances**

```
svfloat16_t svabs[_f16]_x(svbool_t pg, svfloat16_t op)
svfloat32_t svabs[_f32]_x(svbool_t pg, svfloat32_t op)
svfloat64_t svabs[_f64]_x(svbool_t pg, svfloat64_t op)
```

**6.16.28. NEG: Floating-point negation**

These functions return a copy of a floating-point input in which the sign bit is inverted. They do not raise an exception for any input.

**6.16.28.1. NEG (vector), setting inactive to zero****Instances**

```
svfloat16_t svneg[_f16]_z(svbool_t pg, svfloat16_t op)
svfloat32_t svneg[_f32]_z(svbool_t pg, svfloat32_t op)
svfloat64_t svneg[_f64]_z(svbool_t pg, svfloat64_t op)
```

**6.16.28.2. NEG (vector), merging with separate vector****Instances**

```
svfloat16_t svneg[_f16]_m(svfloat16_t inactive, svbool_t pg,
                          svfloat16_t op)
svfloat32_t svneg[_f32]_m(svfloat32_t inactive, svbool_t pg,
                          svfloat32_t op)
svfloat64_t svneg[_f64]_m(svfloat64_t inactive, svbool_t pg,
                          svfloat64_t op)
```

**6.16.28.3. NEG (vector), setting inactive to unknown****Instances**

```
svfloat16_t svneg[_f16]_x(svbool_t pg, svfloat16_t op)
svfloat32_t svneg[_f32]_x(svbool_t pg, svfloat32_t op)
svfloat64_t svneg[_f64]_x(svbool_t pg, svfloat64_t op)
```

**6.16.29. SQRT: Floating-point square root**

These functions compute the square root of a floating-point input.

Signaling NaNs trigger an IEEE Invalid exception but quiet NaNs do not. Taking the square root of a negative nonzero value also triggers an IEEE Invalid exception.

**6.16.29.1. SQRT (vector), setting inactive to zero****Instances**

```
svfloat16_t svsqrt[_f16]_z(svbool_t pg, svfloat16_t op)
```

**Instances**

```
svfloat32_t svsqr[_f32]_z(svbool_t pg, svfloat32_t op)
svfloat64_t svsqr[_f64]_z(svbool_t pg, svfloat64_t op)
```

**6.16.29.2. SQRT (vector), merging with separate vector****Instances**

```
svfloat16_t svsqr[_f16]_m(svfloat16_t inactive, svbool_t pg,
                           svfloat16_t op)
svfloat32_t svsqr[_f32]_m(svfloat32_t inactive, svbool_t pg,
                           svfloat32_t op)
svfloat64_t svsqr[_f64]_m(svfloat64_t inactive, svbool_t pg,
                           svfloat64_t op)
```

**6.16.29.3. SQRT (vector), setting inactive to unknown****Instances**

```
svfloat16_t svsqr[_f16]_x(svbool_t pg, svfloat16_t op)
svfloat32_t svsqr[_f32]_x(svbool_t pg, svfloat32_t op)
svfloat64_t svsqr[_f64]_x(svbool_t pg, svfloat64_t op)
```

**6.16.30. EXPA: Floating-point exponent accelerator**

These functions take an unsigned integer input in which the low 6 bits  $I$  specify a coefficient and in which the next 8 bits (for single precision) or 11 bits (for double precision) specify an exponent  $E$ . The functions return a copy of  $2^{(I/64)}$  with the exponent field replaced by  $E$ .

**6.16.30.1. EXPA (vector)****Instances**

```
svfloat16_t svexpa[_f16](svuint16_t op)
svfloat32_t svexpa[_f32](svuint32_t op)
svfloat64_t svexpa[_f64](svuint64_t op)
```

**6.16.31. RECPE: Floating-point reciprocal estimate****6.16.31.1. RECPE (vector)****Instances**

```
svfloat16_t svrecpe[_f16](svfloat16_t op)
svfloat32_t svrecpe[_f32](svfloat32_t op)
svfloat64_t svrecpe[_f64](svfloat64_t op)
```

**6.16.32. RECPS: Floating-point reciprocal step****6.16.32.1. RECPS (vector, vector)****Instances**

```
svfloat16_t svrecps[_f16](svfloat16_t op1, svfloat16_t op2)
svfloat32_t svrecps[_f32](svfloat32_t op1, svfloat32_t op2)
svfloat64_t svrecps[_f64](svfloat64_t op1, svfloat64_t op2)
```

## 6.16.33. RECPX: Floating-point reciprocal exponent

### 6.16.33.1. RECPX (vector), setting inactive to zero

Instances
svfloat16_t <b>svrecpx</b> [_f16]_z(svbool_t pg, svfloat16_t op)
svfloat32_t <b>svrecpx</b> [_f32]_z(svbool_t pg, svfloat32_t op)
svfloat64_t <b>svrecpx</b> [_f64]_z(svbool_t pg, svfloat64_t op)

### 6.16.33.2. RECPX (vector), merging with separate vector

Instances
svfloat16_t <b>svrecpx</b> [_f16]_m(svfloat16_t inactive, svbool_t pg, svfloat16_t op)
svfloat32_t <b>svrecpx</b> [_f32]_m(svfloat32_t inactive, svbool_t pg, svfloat32_t op)
svfloat64_t <b>svrecpx</b> [_f64]_m(svfloat64_t inactive, svbool_t pg, svfloat64_t op)

### 6.16.33.3. RECPX (vector), setting inactive to unknown

Instances
svfloat16_t <b>svrecpx</b> [_f16]_x(svbool_t pg, svfloat16_t op)
svfloat32_t <b>svrecpx</b> [_f32]_x(svbool_t pg, svfloat32_t op)
svfloat64_t <b>svrecpx</b> [_f64]_x(svbool_t pg, svfloat64_t op)

## 6.16.34. RSQRTE: Floating-point reciprocal square root estimate

### 6.16.34.1. RSQRTE (vector)

Instances
svfloat16_t <b>svrsqrte</b> [_f16](svfloat16_t op)
svfloat32_t <b>svrsqrte</b> [_f32](svfloat32_t op)
svfloat64_t <b>svrsqrte</b> [_f64](svfloat64_t op)

## 6.16.35. RSQRTS: Floating-point reciprocal square root step

### 6.16.35.1. RSQRTS (vector, vector)

Instances
svfloat16_t <b>svrsqrts</b> [_f16](svfloat16_t op1, svfloat16_t op2)
svfloat32_t <b>svrsqrts</b> [_f32](svfloat32_t op1, svfloat32_t op2)
svfloat64_t <b>svrsqrts</b> [_f64](svfloat64_t op1, svfloat64_t op2)

## 6.16.36. RINTA: Floating-point round to nearest, ties away from zero

These functions round a floating-point input to the nearest integral value, rounding away from zero in the event of a tie. They never raise an IEEE Inexact exception.

Signaling NaNs trigger an IEEE Invalid exception but quiet NaNs do not.

### 6.16.36.1. RINTA (vector), setting inactive to zero

Instances
svfloat16_t <b>svrinta</b> [_f16]_z(svbool_t <i>pg</i> , svfloat16_t <i>op</i> )
svfloat32_t <b>svrinta</b> [_f32]_z(svbool_t <i>pg</i> , svfloat32_t <i>op</i> )
svfloat64_t <b>svrinta</b> [_f64]_z(svbool_t <i>pg</i> , svfloat64_t <i>op</i> )

### 6.16.36.2. RINTA (vector), merging with separate vector

Instances
svfloat16_t <b>svrinta</b> [_f16]_m(svfloat16_t <i>inactive</i> , svbool_t <i>pg</i> , svfloat16_t <i>op</i> )
svfloat32_t <b>svrinta</b> [_f32]_m(svfloat32_t <i>inactive</i> , svbool_t <i>pg</i> , svfloat32_t <i>op</i> )
svfloat64_t <b>svrinta</b> [_f64]_m(svfloat64_t <i>inactive</i> , svbool_t <i>pg</i> , svfloat64_t <i>op</i> )

### 6.16.36.3. RINTA (vector), setting inactive to unknown

Instances
svfloat16_t <b>svrinta</b> [_f16]_x(svbool_t <i>pg</i> , svfloat16_t <i>op</i> )
svfloat32_t <b>svrinta</b> [_f32]_x(svbool_t <i>pg</i> , svfloat32_t <i>op</i> )
svfloat64_t <b>svrinta</b> [_f64]_x(svbool_t <i>pg</i> , svfloat64_t <i>op</i> )

## 6.16.37. RINTI: Floating-point round using current rounding mode (inexact)

These functions round a floating-point input to an integral value using the current rounding mode. They never raise an IEEE Inexact exception.

Signaling NaNs trigger an IEEE Invalid exception but quiet NaNs do not.

### 6.16.37.1. RINTI (vector), setting inactive to zero

Instances
svfloat16_t <b>svrinti</b> [_f16]_z(svbool_t <i>pg</i> , svfloat16_t <i>op</i> )
svfloat32_t <b>svrinti</b> [_f32]_z(svbool_t <i>pg</i> , svfloat32_t <i>op</i> )
svfloat64_t <b>svrinti</b> [_f64]_z(svbool_t <i>pg</i> , svfloat64_t <i>op</i> )

### 6.16.37.2. RINTI (vector), merging with separate vector

Instances
svfloat16_t <b>svrinti</b> [_f16]_m(svfloat16_t <i>inactive</i> , svbool_t <i>pg</i> , svfloat16_t <i>op</i> )
svfloat32_t <b>svrinti</b> [_f32]_m(svfloat32_t <i>inactive</i> , svbool_t <i>pg</i> , svfloat32_t <i>op</i> )
svfloat64_t <b>svrinti</b> [_f64]_m(svfloat64_t <i>inactive</i> , svbool_t <i>pg</i> , svfloat64_t <i>op</i> )

### 6.16.37.3. RINTI (vector), setting inactive to unknown

Instances
svfloat16_t <b>svrinti</b> [_f16]_x(svbool_t <i>pg</i> , svfloat16_t <i>op</i> )

**Instances**

```
svfloat32_t svrinti[_f32]_x(svbool_t pg, svfloat32_t op)
svfloat64_t svrinti[_f64]_x(svbool_t pg, svfloat64_t op)
```

**6.16.38. RINTM: Floating-point round towards -Inf**

These functions round a floating-point input to an integral value, in the direction of -Inf. They never raise an IEEE Inexact exception.

Signaling NaNs trigger an IEEE Invalid exception but quiet NaNs do not.

**6.16.38.1. RINTM (vector), setting inactive to zero****Instances**

```
svfloat16_t svrintm[_f16]_z(svbool_t pg, svfloat16_t op)
svfloat32_t svrintm[_f32]_z(svbool_t pg, svfloat32_t op)
svfloat64_t svrintm[_f64]_z(svbool_t pg, svfloat64_t op)
```

**6.16.38.2. RINTM (vector), merging with separate vector****Instances**

```
svfloat16_t svrintm[_f16]_m(svfloat16_t inactive, svbool_t pg,
                           svfloat16_t op)
svfloat32_t svrintm[_f32]_m(svfloat32_t inactive, svbool_t pg,
                           svfloat32_t op)
svfloat64_t svrintm[_f64]_m(svfloat64_t inactive, svbool_t pg,
                           svfloat64_t op)
```

**6.16.38.3. RINTM (vector), setting inactive to unknown****Instances**

```
svfloat16_t svrintm[_f16]_x(svbool_t pg, svfloat16_t op)
svfloat32_t svrintm[_f32]_x(svbool_t pg, svfloat32_t op)
svfloat64_t svrintm[_f64]_x(svbool_t pg, svfloat64_t op)
```

**6.16.39. RINTN: Floating-point round to nearest, ties to even**

These functions round a floating-point input to the nearest integral value, rounding to even in the event of a tie. They never raise an IEEE Inexact exception.

Signaling NaNs trigger an IEEE Invalid exception but quiet NaNs do not.

**6.16.39.1. RINTN (vector), setting inactive to zero****Instances**

```
svfloat16_t svrintn[_f16]_z(svbool_t pg, svfloat16_t op)
svfloat32_t svrintn[_f32]_z(svbool_t pg, svfloat32_t op)
svfloat64_t svrintn[_f64]_z(svbool_t pg, svfloat64_t op)
```

**6.16.39.2. RINTN (vector), merging with separate vector****Instances**

```
svfloat16_t svrintn[_f16]_m(svfloat16_t inactive, svbool_t pg,
```

Instances
svfloat16_t <i>op</i> )
svfloat32_t <b>svrintn</b> [_f32]_m(svfloat32_t <i>inactive</i> , svbool_t <i>pg</i> , svfloat32_t <i>op</i> )
svfloat64_t <b>svrintn</b> [_f64]_m(svfloat64_t <i>inactive</i> , svbool_t <i>pg</i> , svfloat64_t <i>op</i> )

### 6.16.39.3. RINTN (vector), setting inactive to unknown

Instances
svfloat16_t <b>svrintn</b> [_f16]_x(svbool_t <i>pg</i> , svfloat16_t <i>op</i> )
svfloat32_t <b>svrintn</b> [_f32]_x(svbool_t <i>pg</i> , svfloat32_t <i>op</i> )
svfloat64_t <b>svrintn</b> [_f64]_x(svbool_t <i>pg</i> , svfloat64_t <i>op</i> )

## 6.16.40. RINTP: Floating-point round towards +Inf

These functions round a floating-point input to an integral value, in the direction of +Inf. They never raise an IEEE Inexact exception.

Signaling NaNs trigger an IEEE Invalid exception but quiet NaNs do not.

### 6.16.40.1. RINTP (vector), setting inactive to zero

Instances
svfloat16_t <b>svrintp</b> [_f16]_z(svbool_t <i>pg</i> , svfloat16_t <i>op</i> )
svfloat32_t <b>svrintp</b> [_f32]_z(svbool_t <i>pg</i> , svfloat32_t <i>op</i> )
svfloat64_t <b>svrintp</b> [_f64]_z(svbool_t <i>pg</i> , svfloat64_t <i>op</i> )

### 6.16.40.2. RINTP (vector), merging with separate vector

Instances
svfloat16_t <b>svrintp</b> [_f16]_m(svfloat16_t <i>inactive</i> , svbool_t <i>pg</i> , svfloat16_t <i>op</i> )
svfloat32_t <b>svrintp</b> [_f32]_m(svfloat32_t <i>inactive</i> , svbool_t <i>pg</i> , svfloat32_t <i>op</i> )
svfloat64_t <b>svrintp</b> [_f64]_m(svfloat64_t <i>inactive</i> , svbool_t <i>pg</i> , svfloat64_t <i>op</i> )

### 6.16.40.3. RINTP (vector), setting inactive to unknown

Instances
svfloat16_t <b>svrintp</b> [_f16]_x(svbool_t <i>pg</i> , svfloat16_t <i>op</i> )
svfloat32_t <b>svrintp</b> [_f32]_x(svbool_t <i>pg</i> , svfloat32_t <i>op</i> )
svfloat64_t <b>svrintp</b> [_f64]_x(svbool_t <i>pg</i> , svfloat64_t <i>op</i> )

## 6.16.41. RINTX: Floating-point round using current rounding mode (exact)

These functions round a floating-point input to an integral value using the current rounding mode. They raise an IEEE Inexact exception if the result is different from the input.

Signaling NaNs trigger an IEEE Invalid exception but quiet NaNs do not.



### 6.16.41.1. RINTX (vector), setting inactive to zero

#### Instances

```
svfloat16_t svrintx[_f16]_z(svbool_t pg, svfloat16_t op)
svfloat32_t svrintx[_f32]_z(svbool_t pg, svfloat32_t op)
svfloat64_t svrintx[_f64]_z(svbool_t pg, svfloat64_t op)
```

### 6.16.41.2. RINTX (vector), merging with separate vector

#### Instances

```
svfloat16_t svrintx[_f16]_m(svfloat16_t inactive, svbool_t pg,
                             svfloat16_t op)
svfloat32_t svrintx[_f32]_m(svfloat32_t inactive, svbool_t pg,
                             svfloat32_t op)
svfloat64_t svrintx[_f64]_m(svfloat64_t inactive, svbool_t pg,
                             svfloat64_t op)
```

### 6.16.41.3. RINTX (vector), setting inactive to unknown

#### Instances

```
svfloat16_t svrintx[_f16]_x(svbool_t pg, svfloat16_t op)
svfloat32_t svrintx[_f32]_x(svbool_t pg, svfloat32_t op)
svfloat64_t svrintx[_f64]_x(svbool_t pg, svfloat64_t op)
```

## 6.16.42. RINTZ: Floating-point round towards zero

These functions round a floating-point input to an integral value, in the direction of zero. They never raise an IEEE Inexact exception.

Signaling NaNs trigger an IEEE Invalid exception but quiet NaNs do not.

### 6.16.42.1. RINTZ (vector), setting inactive to zero

#### Instances

```
svfloat16_t svrintz[_f16]_z(svbool_t pg, svfloat16_t op)
svfloat32_t svrintz[_f32]_z(svbool_t pg, svfloat32_t op)
svfloat64_t svrintz[_f64]_z(svbool_t pg, svfloat64_t op)
```

### 6.16.42.2. RINTZ (vector), merging with separate vector

#### Instances

```
svfloat16_t svrintz[_f16]_m(svfloat16_t inactive, svbool_t pg,
                             svfloat16_t op)
svfloat32_t svrintz[_f32]_m(svfloat32_t inactive, svbool_t pg,
                             svfloat32_t op)
svfloat64_t svrintz[_f64]_m(svfloat64_t inactive, svbool_t pg,
                             svfloat64_t op)
```

### 6.16.42.3. RINTZ (vector), setting inactive to unknown

#### Instances

```
svfloat16_t svrintz[_f16]_x(svbool_t pg, svfloat16_t op)
svfloat32_t svrintz[_f32]_x(svbool_t pg, svfloat32_t op)
```

**Instances**

```
svfloat64_t svrintz[_f64]_x(svbool_t pg, svfloat64_t op)
```

## 6.17. Floating-point reductions

### 6.17.1. ADDA: Left-to-right floating-point addition reduction

These functions start with a floating-point seed value and successively add each active element of a vector, proceeding in index order. The result is the seed value if no elements are active.

#### 6.17.1.1. ADDA (scalar, vector)

**Instances**

```
float16_t svadda[_f16](svbool_t pg, float16_t initial, svfloat16_t op)
float32_t svadda[_f32](svbool_t pg, float32_t initial, svfloat32_t op)
float64_t svadda[_f64](svbool_t pg, float64_t initial, svfloat64_t op)
```

### 6.17.2. ADDV: Tree-based floating-point addition reduction

These functions sum all active elements of a floating-point vector. They use a tree-based rather than left-to-right reduction, so the result might not be the same as that produced by ADDA ([Section 6.17.1, “ADDA: Left-to-right floating-point addition reduction”](#)). The result is zero if no elements are active.

#### 6.17.2.1. ADDV (vector)

**Instances**

```
float16_t svaddv[_f16](svbool_t pg, svfloat16_t op)
float32_t svaddv[_f32](svbool_t pg, svfloat32_t op)
float64_t svaddv[_f64](svbool_t pg, svfloat64_t op)
```

### 6.17.3. MAXV: Floating-point maximum reduction

These functions return the maximum active element of a floating-point vector. The result is a NaN if any active element is a NaN. It is -Inf if no elements are active.

#### 6.17.3.1. MAXV (vector)

**Instances**

```
float16_t svmaxv[_f16](svbool_t pg, svfloat16_t op)
float32_t svmaxv[_f32](svbool_t pg, svfloat32_t op)
float64_t svmaxv[_f64](svbool_t pg, svfloat64_t op)
```

### 6.17.4. MAXNMV: Floating-point maximum number reduction

These functions return the maximum active element of a floating-point vector. The result is a NaN if all active elements are NaNs (including when no elements are active).

#### 6.17.4.1. MAXNMV (vector)

**Instances**

```
float16_t svmaxnmv[_f16](svbool_t pg, svfloat16_t op)
```

**Instances**

```
float32_t svmaxnmv[_f32](svbool_t pg, svfloat32_t op)
float64_t svmaxnmv[_f64](svbool_t pg, svfloat64_t op)
```

## 6.17.5. MINV: Floating-point minimum reduction

These functions return the minimum active element of a floating-point vector. The result is a NaN if any active input element is a NaN. It is +Inf if no elements are active.

### 6.17.5.1. MINV (vector)

**Instances**

```
float16_t svminv[_f16](svbool_t pg, svfloat16_t op)
float32_t svminv[_f32](svbool_t pg, svfloat32_t op)
float64_t svminv[_f64](svbool_t pg, svfloat64_t op)
```

## 6.17.6. MINNMV: Floating-point minimum number reduction

These functions return the minimum active element of a floating-point vector. The result is a NaN if all active elements are NaNs (including when no elements are active).

### 6.17.6.1. MINNMV (vector)

**Instances**

```
float16_t svminnmv[_f16](svbool_t pg, svfloat16_t op)
float32_t svminnmv[_f32](svbool_t pg, svfloat32_t op)
float64_t svminnmv[_f64](svbool_t pg, svfloat64_t op)
```

## 6.18. Floating-point comparisons

### 6.18.1. CMPEQ: Floating-point compare equal

These functions compare two floating-point inputs and return a predicate bit that indicates whether the inputs are equal.

Signaling NaNs trigger an IEEE Invalid exception but quiet NaNs do not.

#### 6.18.1.1. CMPEQ (vector, vector)

**Instances**

```
svbool_t svcmpcq[_f16](svbool_t pg, svfloat16_t op1, svfloat16_t op2)
svbool_t svcmpcq[_f32](svbool_t pg, svfloat32_t op1, svfloat32_t op2)
svbool_t svcmpcq[_f64](svbool_t pg, svfloat64_t op1, svfloat64_t op2)
```

#### 6.18.1.2. CMPEQ (vector, scalar)

**Instances**

```
svbool_t svcmpcq[_n_f16](svbool_t pg, svfloat16_t op1, float16_t op2)
svbool_t svcmpcq[_n_f32](svbool_t pg, svfloat32_t op1, float32_t op2)
svbool_t svcmpcq[_n_f64](svbool_t pg, svfloat64_t op1, float64_t op2)
```

## 6.18.2. CMPNE: Floating-point compare not equal

These functions compare two floating-point inputs and return a predicate bit that indicates whether the inputs are not equal.

Signaling NaNs trigger an IEEE Invalid exception but quiet NaNs do not.

### 6.18.2.1. CMPNE (vector, vector)

Instances	
svbool_t	<b>svcmpne</b> [_f16](svbool_t <i>pg</i> , svfloat16_t <i>op1</i> , svfloat16_t <i>op2</i> )
svbool_t	<b>svcmpne</b> [_f32](svbool_t <i>pg</i> , svfloat32_t <i>op1</i> , svfloat32_t <i>op2</i> )
svbool_t	<b>svcmpne</b> [_f64](svbool_t <i>pg</i> , svfloat64_t <i>op1</i> , svfloat64_t <i>op2</i> )

### 6.18.2.2. CMPNE (vector, scalar)

Instances	
svbool_t	<b>svcmpne</b> [_n_f16](svbool_t <i>pg</i> , svfloat16_t <i>op1</i> , float16_t <i>op2</i> )
svbool_t	<b>svcmpne</b> [_n_f32](svbool_t <i>pg</i> , svfloat32_t <i>op1</i> , float32_t <i>op2</i> )
svbool_t	<b>svcmpne</b> [_n_f64](svbool_t <i>pg</i> , svfloat64_t <i>op1</i> , float64_t <i>op2</i> )

## 6.18.3. CMPLT: Floating-point compare less than

These functions compare two floating-point inputs and return a predicate bit that indicates whether the first input is less than the second.

Both signaling and quiet NaNs trigger an IEEE Invalid exception.

### 6.18.3.1. CMPLT (vector, vector)

Instances	
svbool_t	<b>svcmplt</b> [_f16](svbool_t <i>pg</i> , svfloat16_t <i>op1</i> , svfloat16_t <i>op2</i> )
svbool_t	<b>svcmplt</b> [_f32](svbool_t <i>pg</i> , svfloat32_t <i>op1</i> , svfloat32_t <i>op2</i> )
svbool_t	<b>svcmplt</b> [_f64](svbool_t <i>pg</i> , svfloat64_t <i>op1</i> , svfloat64_t <i>op2</i> )

### 6.18.3.2. CMPLT (vector, scalar)

Instances	
svbool_t	<b>svcmplt</b> [_n_f16](svbool_t <i>pg</i> , svfloat16_t <i>op1</i> , float16_t <i>op2</i> )
svbool_t	<b>svcmplt</b> [_n_f32](svbool_t <i>pg</i> , svfloat32_t <i>op1</i> , float32_t <i>op2</i> )
svbool_t	<b>svcmplt</b> [_n_f64](svbool_t <i>pg</i> , svfloat64_t <i>op1</i> , float64_t <i>op2</i> )

## 6.18.4. CMPLE: Floating-point compare less than or equal to

These functions compare two floating-point inputs and return a predicate bit that indicates whether the first input is less than or equal to the second.

Both signaling and quiet NaNs trigger an IEEE Invalid exception.

### 6.18.4.1. CMPLE (vector, vector)

Instances	
svbool_t	<b>svcmple</b> [_f16](svbool_t <i>pg</i> , svfloat16_t <i>op1</i> , svfloat16_t <i>op2</i> )

Instances
svbool_t <b>svcmpeq</b> [_f32](svbool_t pg, svfloat32_t op1, svfloat32_t op2)
svbool_t <b>svcmpeq</b> [_f64](svbool_t pg, svfloat64_t op1, svfloat64_t op2)

#### 6.18.4.2. CMPLE (vector, scalar)

Instances
svbool_t <b>svcmple</b> [_n_f16](svbool_t pg, svfloat16_t op1, float16_t op2)
svbool_t <b>svcmple</b> [_n_f32](svbool_t pg, svfloat32_t op1, float32_t op2)
svbool_t <b>svcmple</b> [_n_f64](svbool_t pg, svfloat64_t op1, float64_t op2)

### 6.18.5. CMPGE: Floating-point compare greater than or equal to

These functions compare two floating-point inputs and return a predicate bit that indicates whether the first input is greater than or equal the second.

Both signaling and quiet NaNs trigger an IEEE Invalid exception.

#### 6.18.5.1. CMPGE (vector, vector)

Instances
svbool_t <b>svcmpge</b> [_f16](svbool_t pg, svfloat16_t op1, svfloat16_t op2)
svbool_t <b>svcmpge</b> [_f32](svbool_t pg, svfloat32_t op1, svfloat32_t op2)
svbool_t <b>svcmpge</b> [_f64](svbool_t pg, svfloat64_t op1, svfloat64_t op2)

#### 6.18.5.2. CMPGE (vector, scalar)

Instances
svbool_t <b>svcmpge</b> [_n_f16](svbool_t pg, svfloat16_t op1, float16_t op2)
svbool_t <b>svcmpge</b> [_n_f32](svbool_t pg, svfloat32_t op1, float32_t op2)
svbool_t <b>svcmpge</b> [_n_f64](svbool_t pg, svfloat64_t op1, float64_t op2)

### 6.18.6. CMPGT: Floating-point compare greater than

These functions compare two floating-point inputs and return a predicate bit that indicates whether the first input is greater than the second.

Both signaling and quiet NaNs trigger an IEEE Invalid exception.

#### 6.18.6.1. CMPGT (vector, vector)

Instances
svbool_t <b>svcmpgt</b> [_f16](svbool_t pg, svfloat16_t op1, svfloat16_t op2)
svbool_t <b>svcmpgt</b> [_f32](svbool_t pg, svfloat32_t op1, svfloat32_t op2)
svbool_t <b>svcmpgt</b> [_f64](svbool_t pg, svfloat64_t op1, svfloat64_t op2)

#### 6.18.6.2. CMPGT (vector, scalar)

Instances
svbool_t <b>svcmpgt</b> [_n_f16](svbool_t pg, svfloat16_t op1, float16_t op2)
svbool_t <b>svcmpgt</b> [_n_f32](svbool_t pg, svfloat32_t op1, float32_t op2)
svbool_t <b>svcmpgt</b> [_n_f64](svbool_t pg, svfloat64_t op1, float64_t op2)

## 6.18.7. CMPUO: Floating-point compare unordered

These functions compare two floating-point inputs and return a predicate bit that indicates whether they are unordered (in other words, whether at least one of them is a NaN).

These functions never raise an IEEE exception.

### 6.18.7.1. CMPUO (vector, vector)

Instances	
svbool_t	<b>svcmpuo</b> [_f16](svbool_t <i>pg</i> , svfloat16_t <i>op1</i> , svfloat16_t <i>op2</i> )
svbool_t	<b>svcmpuo</b> [_f32](svbool_t <i>pg</i> , svfloat32_t <i>op1</i> , svfloat32_t <i>op2</i> )
svbool_t	<b>svcmpuo</b> [_f64](svbool_t <i>pg</i> , svfloat64_t <i>op1</i> , svfloat64_t <i>op2</i> )

### 6.18.7.2. CMPUO (vector, scalar)

Instances	
svbool_t	<b>svcmpuo</b> [_n_f16](svbool_t <i>pg</i> , svfloat16_t <i>op1</i> , float16_t <i>op2</i> )
svbool_t	<b>svcmpuo</b> [_n_f32](svbool_t <i>pg</i> , svfloat32_t <i>op1</i> , float32_t <i>op2</i> )
svbool_t	<b>svcmpuo</b> [_n_f64](svbool_t <i>pg</i> , svfloat64_t <i>op1</i> , float64_t <i>op2</i> )

## 6.18.8. ACLT: Floating-point absolute compare less than

These functions take the absolute values of two floating-point inputs and return a predicate bit that indicates whether the absolute value of the first input is less than the absolute value of the second.

Both signaling and quiet NaNs trigger an IEEE Invalid exception.

### 6.18.8.1. ACLT (vector, vector)

Instances	
svbool_t	<b>svaclt</b> [_f16](svbool_t <i>pg</i> , svfloat16_t <i>op1</i> , svfloat16_t <i>op2</i> )
svbool_t	<b>svaclt</b> [_f32](svbool_t <i>pg</i> , svfloat32_t <i>op1</i> , svfloat32_t <i>op2</i> )
svbool_t	<b>svaclt</b> [_f64](svbool_t <i>pg</i> , svfloat64_t <i>op1</i> , svfloat64_t <i>op2</i> )

### 6.18.8.2. ACLT (vector, scalar)

Instances	
svbool_t	<b>svaclt</b> [_n_f16](svbool_t <i>pg</i> , svfloat16_t <i>op1</i> , float16_t <i>op2</i> )
svbool_t	<b>svaclt</b> [_n_f32](svbool_t <i>pg</i> , svfloat32_t <i>op1</i> , float32_t <i>op2</i> )
svbool_t	<b>svaclt</b> [_n_f64](svbool_t <i>pg</i> , svfloat64_t <i>op1</i> , float64_t <i>op2</i> )

## 6.18.9. ACLE: Floating-point absolute compare less than or equal to

These functions take the absolute values of two floating-point inputs and return a predicate bit that indicates whether the absolute value of the first input is less than or equal to the absolute value of the second.

Both signaling and quiet NaNs trigger an IEEE Invalid exception.

### 6.18.9.1. ACLE (vector, vector)

Instances	
svbool_t	<b>svacle</b> [_f16](svbool_t <i>pg</i> , svfloat16_t <i>op1</i> , svfloat16_t <i>op2</i> )

**Instances**

```
svbool_t svacle[_f32](svbool_t pg, svfloat32_t op1, svfloat32_t op2)
svbool_t svacle[_f64](svbool_t pg, svfloat64_t op1, svfloat64_t op2)
```

**6.18.9.2. ACLE (vector, scalar)****Instances**

```
svbool_t svacle[_n_f16](svbool_t pg, svfloat16_t op1, float16_t op2)
svbool_t svacle[_n_f32](svbool_t pg, svfloat32_t op1, float32_t op2)
svbool_t svacle[_n_f64](svbool_t pg, svfloat64_t op1, float64_t op2)
```

**6.18.10. ACGE: Floating-point absolute compare greater than or equal to**

These functions take the absolute values of two floating-point inputs and return a predicate bit that indicates whether the absolute value of the first input is greater than or equal to the absolute value of the second.

Both signaling and quiet NaNs trigger an IEEE Invalid exception.

**6.18.10.1. ACGE (vector, vector)****Instances**

```
svbool_t svacge[_f16](svbool_t pg, svfloat16_t op1, svfloat16_t op2)
svbool_t svacge[_f32](svbool_t pg, svfloat32_t op1, svfloat32_t op2)
svbool_t svacge[_f64](svbool_t pg, svfloat64_t op1, svfloat64_t op2)
```

**6.18.10.2. ACGE (vector, scalar)****Instances**

```
svbool_t svacge[_n_f16](svbool_t pg, svfloat16_t op1, float16_t op2)
svbool_t svacge[_n_f32](svbool_t pg, svfloat32_t op1, float32_t op2)
svbool_t svacge[_n_f64](svbool_t pg, svfloat64_t op1, float64_t op2)
```

**6.18.11. ACGT: Floating-point absolute compare greater than**

These functions take the absolute values of two floating-point inputs and return a predicate bit that indicates whether the absolute value of the first input is greater than the absolute value of the second.

Both signaling and quiet NaNs trigger an IEEE Invalid exception.

**6.18.11.1. ACGT (vector, vector)****Instances**

```
svbool_t svacgt[_f16](svbool_t pg, svfloat16_t op1, svfloat16_t op2)
svbool_t svacgt[_f32](svbool_t pg, svfloat32_t op1, svfloat32_t op2)
svbool_t svacgt[_f64](svbool_t pg, svfloat64_t op1, svfloat64_t op2)
```

**6.18.11.2. ACGT (vector, scalar)****Instances**

```
svbool_t svacgt[_n_f16](svbool_t pg, svfloat16_t op1, float16_t op2)
```

**Instances**

```
svbool_t svacgt[_n_f32](svbool_t pg, svfloat32_t op1, float32_t op2)
svbool_t svacgt[_n_f64](svbool_t pg, svfloat64_t op1, float64_t op2)
```

## 6.19. Floating-point conversions

### 6.19.1. CVT: Convert floating-point value to integer

These functions convert a floating-point value to an integer type, rounding towards zero. If the input value is too large for the integer type, the functions return the nearest in-range value.

Both signaling and quiet NaNs trigger an IEEE Invalid exception.

#### 6.19.1.1. CVT (vector), setting inactive to zero

**Instances**

```
svint16_t svcvt_s16[_f16]_z(svbool_t pg, svfloat16_t op)
svuint16_t svcvt_u16[_f16]_z(svbool_t pg, svfloat16_t op)
svint32_t svcvt_s32[_f16]_z(svbool_t pg, svfloat16_t op)
svint32_t svcvt_s32[_f32]_z(svbool_t pg, svfloat32_t op)
svint32_t svcvt_s32[_f64]_z(svbool_t pg, svfloat64_t op)
svint64_t svcvt_s64[_f16]_z(svbool_t pg, svfloat16_t op)
svint64_t svcvt_s64[_f32]_z(svbool_t pg, svfloat32_t op)
svint64_t svcvt_s64[_f64]_z(svbool_t pg, svfloat64_t op)
svuint32_t svcvt_u32[_f16]_z(svbool_t pg, svfloat16_t op)
svuint32_t svcvt_u32[_f32]_z(svbool_t pg, svfloat32_t op)
svuint32_t svcvt_u32[_f64]_z(svbool_t pg, svfloat64_t op)
svuint64_t svcvt_u64[_f16]_z(svbool_t pg, svfloat16_t op)
svuint64_t svcvt_u64[_f32]_z(svbool_t pg, svfloat32_t op)
svuint64_t svcvt_u64[_f64]_z(svbool_t pg, svfloat64_t op)
```

#### 6.19.1.2. CVT (vector), merging with separate vector

**Instances**

```
svint16_t svcvt_s16[_f16]_m(svint16_t inactive, svbool_t pg,
                             svfloat16_t op)
svuint16_t svcvt_u16[_f16]_m(svuint16_t inactive, svbool_t pg,
                              svfloat16_t op)
svint32_t svcvt_s32[_f16]_m(svint32_t inactive, svbool_t pg,
                             svfloat16_t op)
svint32_t svcvt_s32[_f32]_m(svint32_t inactive, svbool_t pg,
                             svfloat32_t op)
svint32_t svcvt_s32[_f64]_m(svint32_t inactive, svbool_t pg,
                             svfloat64_t op)
svint64_t svcvt_s64[_f16]_m(svint64_t inactive, svbool_t pg,
                             svfloat16_t op)
svint64_t svcvt_s64[_f32]_m(svint64_t inactive, svbool_t pg,
                             svfloat32_t op)
svint64_t svcvt_s64[_f64]_m(svint64_t inactive, svbool_t pg,
                             svfloat64_t op)
svuint32_t svcvt_u32[_f16]_m(svuint32_t inactive, svbool_t pg,
                              svfloat16_t op)
svuint32_t svcvt_u32[_f32]_m(svuint32_t inactive, svbool_t pg,
                              svfloat32_t op)
svuint32_t svcvt_u32[_f64]_m(svuint32_t inactive, svbool_t pg,
                              svfloat64_t op)
```



**Instances**

```
svuint64_t svcvt_u64[_f16]_m(svuint64_t inactive, svbool_t pg,
                               svfloat16_t op)
svuint64_t svcvt_u64[_f32]_m(svuint64_t inactive, svbool_t pg,
                               svfloat32_t op)
svuint64_t svcvt_u64[_f64]_m(svuint64_t inactive, svbool_t pg,
                               svfloat64_t op)
```

**6.19.1.3. CVT (vector), setting inactive to unknown****Instances**

```
svint16_t svcvt_s16[_f16]_x(svbool_t pg, svfloat16_t op)
svuint16_t svcvt_u16[_f16]_x(svbool_t pg, svfloat16_t op)
svint32_t svcvt_s32[_f16]_x(svbool_t pg, svfloat16_t op)
svint32_t svcvt_s32[_f32]_x(svbool_t pg, svfloat32_t op)
svint32_t svcvt_s32[_f64]_x(svbool_t pg, svfloat64_t op)
svint64_t svcvt_s64[_f16]_x(svbool_t pg, svfloat16_t op)
svint64_t svcvt_s64[_f32]_x(svbool_t pg, svfloat32_t op)
svint64_t svcvt_s64[_f64]_x(svbool_t pg, svfloat64_t op)
svuint32_t svcvt_u32[_f16]_x(svbool_t pg, svfloat16_t op)
svuint32_t svcvt_u32[_f32]_x(svbool_t pg, svfloat32_t op)
svuint32_t svcvt_u32[_f64]_x(svbool_t pg, svfloat64_t op)
svuint64_t svcvt_u64[_f16]_x(svbool_t pg, svfloat16_t op)
svuint64_t svcvt_u64[_f32]_x(svbool_t pg, svfloat32_t op)
svuint64_t svcvt_u64[_f64]_x(svbool_t pg, svfloat64_t op)
```

**6.19.2. CVT: Convert integer value to floating-point**

These functions convert an integer value to a floating-point type, rounding according to the current rounding mode.

**6.19.2.1. CVT (vector), setting inactive to zero****Instances**

```
svfloat16_t svcvt_f16[_s16]_z(svbool_t pg, svint16_t op)
svfloat16_t svcvt_f16[_u16]_z(svbool_t pg, svuint16_t op)
svfloat16_t svcvt_f16[_s32]_z(svbool_t pg, svint32_t op)
svfloat32_t svcvt_f32[_s32]_z(svbool_t pg, svint32_t op)
svfloat64_t svcvt_f64[_s32]_z(svbool_t pg, svint32_t op)
svfloat16_t svcvt_f16[_s64]_z(svbool_t pg, svint64_t op)
svfloat32_t svcvt_f32[_s64]_z(svbool_t pg, svint64_t op)
svfloat64_t svcvt_f64[_s64]_z(svbool_t pg, svint64_t op)
svfloat16_t svcvt_f16[_u32]_z(svbool_t pg, svuint32_t op)
svfloat32_t svcvt_f32[_u32]_z(svbool_t pg, svuint32_t op)
svfloat64_t svcvt_f64[_u32]_z(svbool_t pg, svuint32_t op)
svfloat16_t svcvt_f16[_u64]_z(svbool_t pg, svuint64_t op)
svfloat32_t svcvt_f32[_u64]_z(svbool_t pg, svuint64_t op)
svfloat64_t svcvt_f64[_u64]_z(svbool_t pg, svuint64_t op)
```

**6.19.2.2. CVT (vector), merging with separate vector****Instances**

```
svfloat16_t svcvt_f16[_s16]_m(svfloat16_t inactive, svbool_t pg,
                               svint16_t op)
svfloat16_t svcvt_f16[_u16]_m(svfloat16_t inactive, svbool_t pg,
```

Instances	
svfloat16_t	<b>svcvt_f16</b> [_s32]_m(svfloat16_t inactive, svbool_t pg, svint32_t op)
svfloat32_t	<b>svcvt_f32</b> [_s32]_m(svfloat32_t inactive, svbool_t pg, svint32_t op)
svfloat64_t	<b>svcvt_f64</b> [_s32]_m(svfloat64_t inactive, svbool_t pg, svint32_t op)
svfloat16_t	<b>svcvt_f16</b> [_s64]_m(svfloat16_t inactive, svbool_t pg, svint64_t op)
svfloat32_t	<b>svcvt_f32</b> [_s64]_m(svfloat32_t inactive, svbool_t pg, svint64_t op)
svfloat64_t	<b>svcvt_f64</b> [_s64]_m(svfloat64_t inactive, svbool_t pg, svint64_t op)
svfloat16_t	<b>svcvt_f16</b> [_u32]_m(svfloat16_t inactive, svbool_t pg, svuint32_t op)
svfloat32_t	<b>svcvt_f32</b> [_u32]_m(svfloat32_t inactive, svbool_t pg, svuint32_t op)
svfloat64_t	<b>svcvt_f64</b> [_u32]_m(svfloat64_t inactive, svbool_t pg, svuint32_t op)
svfloat16_t	<b>svcvt_f16</b> [_u64]_m(svfloat16_t inactive, svbool_t pg, svuint64_t op)
svfloat32_t	<b>svcvt_f32</b> [_u64]_m(svfloat32_t inactive, svbool_t pg, svuint64_t op)
svfloat64_t	<b>svcvt_f64</b> [_u64]_m(svfloat64_t inactive, svbool_t pg, svuint64_t op)

### 6.19.2.3. CVT (vector), setting inactive to unknown

Instances	
svfloat16_t	<b>svcvt_f16</b> [_s16]_x(svbool_t pg, svint16_t op)
svfloat16_t	<b>svcvt_f16</b> [_u16]_x(svbool_t pg, svuint16_t op)
svfloat16_t	<b>svcvt_f16</b> [_s32]_x(svbool_t pg, svint32_t op)
svfloat32_t	<b>svcvt_f32</b> [_s32]_x(svbool_t pg, svint32_t op)
svfloat64_t	<b>svcvt_f64</b> [_s32]_x(svbool_t pg, svint32_t op)
svfloat16_t	<b>svcvt_f16</b> [_s64]_x(svbool_t pg, svint64_t op)
svfloat32_t	<b>svcvt_f32</b> [_s64]_x(svbool_t pg, svint64_t op)
svfloat64_t	<b>svcvt_f64</b> [_s64]_x(svbool_t pg, svint64_t op)
svfloat16_t	<b>svcvt_f16</b> [_u32]_x(svbool_t pg, svuint32_t op)
svfloat32_t	<b>svcvt_f32</b> [_u32]_x(svbool_t pg, svuint32_t op)
svfloat64_t	<b>svcvt_f64</b> [_u32]_x(svbool_t pg, svuint32_t op)
svfloat16_t	<b>svcvt_f16</b> [_u64]_x(svbool_t pg, svuint64_t op)
svfloat32_t	<b>svcvt_f32</b> [_u64]_x(svbool_t pg, svuint64_t op)
svfloat64_t	<b>svcvt_f64</b> [_u64]_x(svbool_t pg, svuint64_t op)

## 6.19.3. CVT: Convert floating-point value to wider type

These functions extend a floating-point value to a wider type.

Signaling NaNs trigger an IEEE Invalid exception but quiet NaNs do not.

### 6.19.3.1. CVT (vector), setting inactive to zero

Instances	
svfloat32_t	<b>svcvt_f32</b> [_f16]_z(svbool_t pg, svfloat16_t op)
svfloat64_t	<b>svcvt_f64</b> [_f16]_z(svbool_t pg, svfloat16_t op)

**Instances**

```
svfloat64_t svcvt_f64[_f32]_z(svbool_t pg, svfloat32_t op)
```

**6.19.3.2. CVT (vector), merging with separate vector****Instances**

```
svfloat32_t svcvt_f32[_f16]_m(svfloat32_t inactive, svbool_t pg,
                                svfloat16_t op)
svfloat64_t svcvt_f64[_f16]_m(svfloat64_t inactive, svbool_t pg,
                                svfloat16_t op)
svfloat64_t svcvt_f64[_f32]_m(svfloat64_t inactive, svbool_t pg,
                                svfloat32_t op)
```

**6.19.3.3. CVT (vector), setting inactive to unknown****Instances**

```
svfloat32_t svcvt_f32[_f16]_x(svbool_t pg, svfloat16_t op)
svfloat64_t svcvt_f64[_f16]_x(svbool_t pg, svfloat16_t op)
svfloat64_t svcvt_f64[_f32]_x(svbool_t pg, svfloat32_t op)
```

**6.19.4. CVT: Convert floating-point value to narrower type**

These functions convert a floating-point value to a narrower type, rounding according to the current rounding mode.

Signaling NaNs trigger an IEEE Invalid exception. Quiet NaNs and infinities trigger an IEEE Invalid exception only if the result type cannot represent them (which is true for float16\_t when the AHP bit is set).

**6.19.4.1. CVT (vector), setting inactive to zero****Instances**

```
svfloat16_t svcvt_f16[_f32]_z(svbool_t pg, svfloat32_t op)
svfloat16_t svcvt_f16[_f64]_z(svbool_t pg, svfloat64_t op)
svfloat32_t svcvt_f32[_f64]_z(svbool_t pg, svfloat64_t op)
```

**6.19.4.2. CVT (vector), merging with separate vector****Instances**

```
svfloat16_t svcvt_f16[_f32]_m(svfloat16_t inactive, svbool_t pg,
                                svfloat32_t op)
svfloat16_t svcvt_f16[_f64]_m(svfloat16_t inactive, svbool_t pg,
                                svfloat64_t op)
svfloat32_t svcvt_f32[_f64]_m(svfloat32_t inactive, svbool_t pg,
                                svfloat64_t op)
```

**6.19.4.3. CVT (vector), setting inactive to unknown****Instances**

```
svfloat16_t svcvt_f16[_f32]_x(svbool_t pg, svfloat32_t op)
svfloat16_t svcvt_f16[_f64]_x(svbool_t pg, svfloat64_t op)
svfloat32_t svcvt_f32[_f64]_x(svbool_t pg, svfloat64_t op)
```

## 6.20. Permutation and selection

### 6.20.1. LASTA: Extract element after last active

These functions return the element that follows the last active element. They return the first element if the last element is active or if no elements are active.

#### 6.20.1.1. LASTA (vector)

Instances
<pre> int8_t  svlasta[_s8](svbool_t pg, svint8_t op) int16_t svlasta[_s16](svbool_t pg, svint16_t op) int32_t svlasta[_s32](svbool_t pg, svint32_t op) int64_t svlasta[_s64](svbool_t pg, svint64_t op) uint8_t svlasta[_u8](svbool_t pg, svuint8_t op) uint16_t svlasta[_u16](svbool_t pg, svuint16_t op) uint32_t svlasta[_u32](svbool_t pg, svuint32_t op) uint64_t svlasta[_u64](svbool_t pg, svuint64_t op) float16_t svlasta[_f16](svbool_t pg, svfloat16_t op) float32_t svlasta[_f32](svbool_t pg, svfloat32_t op) float64_t svlasta[_f64](svbool_t pg, svfloat64_t op) bfloat16_t svlasta[_bf16](svbool_t pg, svbfloat16_t op) </pre>

### 6.20.2. LASTB: Extract last active element

These functions return the last active element of a vector, or the last element if no elements are active.

#### 6.20.2.1. LASTB (vector)

Instances
<pre> int8_t  svlastb[_s8](svbool_t pg, svint8_t op) int16_t svlastb[_s16](svbool_t pg, svint16_t op) int32_t svlastb[_s32](svbool_t pg, svint32_t op) int64_t svlastb[_s64](svbool_t pg, svint64_t op) uint8_t svlastb[_u8](svbool_t pg, svuint8_t op) uint16_t svlastb[_u16](svbool_t pg, svuint16_t op) uint32_t svlastb[_u32](svbool_t pg, svuint32_t op) uint64_t svlastb[_u64](svbool_t pg, svuint64_t op) float16_t svlastb[_f16](svbool_t pg, svfloat16_t op) float32_t svlastb[_f32](svbool_t pg, svfloat32_t op) float64_t svlastb[_f64](svbool_t pg, svfloat64_t op) bfloat16_t svlastb[_bf16](svbool_t pg, svbfloat16_t op) </pre>

### 6.20.3. CLASTA: Extract element after last active with fallback

These functions return the element that follows the last active element of a vector. They return the first element if the last element is active and a supplied fallback value if no elements are active.

#### 6.20.3.1. CLASTA (vector, vector)

Instances
<pre> svint8_t  svclasta[_s8](svbool_t pg, svint8_t fallback, svint8_t data) svint16_t svclasta[_s16](svbool_t pg, svint16_t fallback, svint16_t data) svint32_t svclasta[_s32](svbool_t pg, svint32_t fallback, svint32_t data) svint64_t svclasta[_s64](svbool_t pg, svint64_t fallback, svint64_t data) </pre>

**Instances**

```

svuint8_t  svclasta[_u8](svbool_t pg, svuint8_t fallback, svuint8_t data)
svuint16_t svclasta[_u16](svbool_t pg, svuint16_t fallback,
                           svuint16_t data)
svuint32_t svclasta[_u32](svbool_t pg, svuint32_t fallback,
                           svuint32_t data)
svuint64_t svclasta[_u64](svbool_t pg, svuint64_t fallback,
                           svuint64_t data)
svfloat16_t svclasta[_f16](svbool_t pg, svfloat16_t fallback,
                           svfloat16_t data)
svfloat32_t svclasta[_f32](svbool_t pg, svfloat32_t fallback,
                           svfloat32_t data)
svfloat64_t svclasta[_f64](svbool_t pg, svfloat64_t fallback,
                           svfloat64_t data)
svbfloat16_t svclasta[_bf16](svbool_t pg, svbfloat16_t fallback,
                              svbfloat16_t data)

```

**6.20.3.2. CLASTA (scalar, vector)****Instances**

```

int8_t  svclasta[_n_s8](svbool_t pg, int8_t fallback, svint8_t data)
int16_t svclasta[_n_s16](svbool_t pg, int16_t fallback, svint16_t data)
int32_t svclasta[_n_s32](svbool_t pg, int32_t fallback, svint32_t data)
int64_t svclasta[_n_s64](svbool_t pg, int64_t fallback, svint64_t data)
uint8_t svclasta[_n_u8](svbool_t pg, uint8_t fallback, svuint8_t data)
uint16_t svclasta[_n_u16](svbool_t pg, uint16_t fallback, svuint16_t data)
uint32_t svclasta[_n_u32](svbool_t pg, uint32_t fallback, svuint32_t data)
uint64_t svclasta[_n_u64](svbool_t pg, uint64_t fallback, svuint64_t data)
float16_t svclasta[_n_f16](svbool_t pg, float16_t fallback,
                           svfloat16_t data)
float32_t svclasta[_n_f32](svbool_t pg, float32_t fallback,
                           svfloat32_t data)
float64_t svclasta[_n_f64](svbool_t pg, float64_t fallback,
                           svfloat64_t data)
bfloat16_t svclasta[_n_bf16](svbool_t pg, bfloat16_t fallback,
                              svbfloat16_t data)

```

**6.20.4. CLASTB: Extract last active element with fallback**

These functions return the last active element of a vector, or a supplied fallback value if no elements are active.

**6.20.4.1. CLASTB (vector, vector)****Instances**

```

svint8_t  svclastb[_s8](svbool_t pg, svint8_t fallback, svint8_t data)
svint16_t svclastb[_s16](svbool_t pg, svint16_t fallback, svint16_t data)
svint32_t svclastb[_s32](svbool_t pg, svint32_t fallback, svint32_t data)
svint64_t svclastb[_s64](svbool_t pg, svint64_t fallback, svint64_t data)
svuint8_t svclastb[_u8](svbool_t pg, svuint8_t fallback, svuint8_t data)
svuint16_t svclastb[_u16](svbool_t pg, svuint16_t fallback,
                           svuint16_t data)
svuint32_t svclastb[_u32](svbool_t pg, svuint32_t fallback,
                           svuint32_t data)
svuint64_t svclastb[_u64](svbool_t pg, svuint64_t fallback,
                           svuint64_t data)
svfloat16_t svclastb[_f16](svbool_t pg, svfloat16_t fallback,

```

Instances
<pre> svfloat16_t data) svfloat32_t <b>svclastb</b>[_f32](svbool_t pg, svfloat32_t fallback, svfloat32_t data) svfloat64_t <b>svclastb</b>[_f64](svbool_t pg, svfloat64_t fallback, svfloat64_t data) svbfloat16_t <b>svclastb</b>[_bf16](svbool_t pg, svbfloat16_t fallback, svbfloat16_t data) </pre>

#### 6.20.4.2. CLASTB (scalar, vector)

Instances
<pre> int8_t <b>svclastb</b>[_n_s8](svbool_t pg, int8_t fallback, svint8_t data) int16_t <b>svclastb</b>[_n_s16](svbool_t pg, int16_t fallback, svint16_t data) int32_t <b>svclastb</b>[_n_s32](svbool_t pg, int32_t fallback, svint32_t data) int64_t <b>svclastb</b>[_n_s64](svbool_t pg, int64_t fallback, svint64_t data) uint8_t <b>svclastb</b>[_n_u8](svbool_t pg, uint8_t fallback, svuint8_t data) uint16_t <b>svclastb</b>[_n_u16](svbool_t pg, uint16_t fallback, svuint16_t data) uint32_t <b>svclastb</b>[_n_u32](svbool_t pg, uint32_t fallback, svuint32_t data) uint64_t <b>svclastb</b>[_n_u64](svbool_t pg, uint64_t fallback, svuint64_t data) float16_t <b>svclastb</b>[_n_f16](svbool_t pg, float16_t fallback, svfloat16_t data) float32_t <b>svclastb</b>[_n_f32](svbool_t pg, float32_t fallback, svfloat32_t data) float64_t <b>svclastb</b>[_n_f64](svbool_t pg, float64_t fallback, svfloat64_t data) bfloat16_t <b>svclastb</b>[_n_bf16](svbool_t pg, bfloat16_t fallback, svbfloat16_t data) </pre>

#### 6.20.5. COMPACT: Compact vector and fill with zero

These functions concatenate the active elements of the input vector, filling any remaining elements with zero.

##### 6.20.5.1. COMPACT (vector)

Instances
<pre> svint32_t <b>svcompact</b>[_s32](svbool_t pg, svint32_t op) svint64_t <b>svcompact</b>[_s64](svbool_t pg, svint64_t op) svuint32_t <b>svcompact</b>[_u32](svbool_t pg, svuint32_t op) svuint64_t <b>svcompact</b>[_u64](svbool_t pg, svuint64_t op) svfloat32_t <b>svcompact</b>[_f32](svbool_t pg, svfloat32_t op) svfloat64_t <b>svcompact</b>[_f64](svbool_t pg, svfloat64_t op) </pre>

#### 6.20.6. SPLICE: Splice two vectors under predicate control

These functions take the first active element onwards of the first input vector and append elements from the start of the second input vector onwards until they have a full vector result.

##### 6.20.6.1. SPLICE (vector, vector)

Instances
<pre> svint8_t <b>svsplice</b>[_s8](svbool_t pg, svint8_t op1, svint8_t op2) svint16_t <b>svsplice</b>[_s16](svbool_t pg, svint16_t op1, svint16_t op2) svint32_t <b>svsplice</b>[_s32](svbool_t pg, svint32_t op1, svint32_t op2) </pre>

**Instances**

```

svint64_t svsplice[_s64](svbool_t pg, svint64_t op1, svint64_t op2)
svuint8_t svsplice[_u8](svbool_t pg, svuint8_t op1, svuint8_t op2)
svuint16_t svsplice[_u16](svbool_t pg, svuint16_t op1, svuint16_t op2)
svuint32_t svsplice[_u32](svbool_t pg, svuint32_t op1, svuint32_t op2)
svuint64_t svsplice[_u64](svbool_t pg, svuint64_t op1, svuint64_t op2)
svfloat16_t svsplice[_f16](svbool_t pg, svfloat16_t op1, svfloat16_t op2)
svfloat32_t svsplice[_f32](svbool_t pg, svfloat32_t op1, svfloat32_t op2)
svfloat64_t svsplice[_f64](svbool_t pg, svfloat64_t op1, svfloat64_t op2)
svbfloat16_t svsplice[_bf16](svbool_t pg, svbfloat16_t op1,
svbfloat16_t op2)

```

**6.20.7. EXT: Extract vector from pair of vectors**

These functions take three inputs: two vectors and an immediate index. If the index is in range of the first vector, they take the elements from that point onwards of the first vector and append elements from the start of the second vector onwards until they have a full vector result. If the index is out of range, the functions return the first vector unmodified.

If the elements have  $N$  bits, the index must be an integer constant expression in the range  $[0, 2048/N)$ .

**6.20.7.1. EXT (vector, vector, immediate)****Instances**

```

svint8_t svext[_s8](svint8_t op1, svint8_t op2, uint64_t imm3)
svint16_t svext[_s16](svint16_t op1, svint16_t op2, uint64_t imm3)
svint32_t svext[_s32](svint32_t op1, svint32_t op2, uint64_t imm3)
svint64_t svext[_s64](svint64_t op1, svint64_t op2, uint64_t imm3)
svuint8_t svext[_u8](svuint8_t op1, svuint8_t op2, uint64_t imm3)
svuint16_t svext[_u16](svuint16_t op1, svuint16_t op2, uint64_t imm3)
svuint32_t svext[_u32](svuint32_t op1, svuint32_t op2, uint64_t imm3)
svuint64_t svext[_u64](svuint64_t op1, svuint64_t op2, uint64_t imm3)
svfloat16_t svext[_f16](svfloat16_t op1, svfloat16_t op2, uint64_t imm3)
svfloat32_t svext[_f32](svfloat32_t op1, svfloat32_t op2, uint64_t imm3)
svfloat64_t svext[_f64](svfloat64_t op1, svfloat64_t op2, uint64_t imm3)
svbfloat16_t svext[_bf16](svbfloat16_t op1, svbfloat16_t op2,
uint64_t imm3)

```

**6.20.8. SEL: Conditionally select elements from two inputs**

These functions return a vector or predicate in which the active elements come from the first input and the inactive elements come from the second input.

**6.20.8.1. SEL (vector, vector)****Instances**

```

svint8_t svsel[_s8](svbool_t pg, svint8_t op1, svint8_t op2)
svint16_t svsel[_s16](svbool_t pg, svint16_t op1, svint16_t op2)
svint32_t svsel[_s32](svbool_t pg, svint32_t op1, svint32_t op2)
svint64_t svsel[_s64](svbool_t pg, svint64_t op1, svint64_t op2)
svuint8_t svsel[_u8](svbool_t pg, svuint8_t op1, svuint8_t op2)
svuint16_t svsel[_u16](svbool_t pg, svuint16_t op1, svuint16_t op2)
svuint32_t svsel[_u32](svbool_t pg, svuint32_t op1, svuint32_t op2)
svuint64_t svsel[_u64](svbool_t pg, svuint64_t op1, svuint64_t op2)
svfloat16_t svsel[_f16](svbool_t pg, svfloat16_t op1, svfloat16_t op2)
svfloat32_t svsel[_f32](svbool_t pg, svfloat32_t op1, svfloat32_t op2)

```

**Instances**

```
svfloat64_t svsel[_f64](svbool_t pg, svfloat64_t op1, svfloat64_t op2)
svbfloat16_t svsel[_bf16](svbool_t pg, svbfloat16_t op1, svbfloat16_t op2)
```

**6.20.8.2. SEL (predicate, predicate)****Instances**

```
svbool_t svsel[_b](svbool_t pg, svbool_t op1, svbool_t op2)
```

**6.20.9. DUP: Duplicate one element of a vector**

These functions use the second input to select one element of the first input and then duplicate that element to fill an entire vector. They return a zero vector if the second input is out of range.

These semantics match the DUP (indexed) instruction if the second input is a constant in the range of that instruction. They are also consistent with duplicating the second input to fill a vector and then using a TBL instruction.

**6.20.9.1. DUP (vector, scalar)****Instances**

```
svint8_t svdup_lane[_s8](svint8_t data, uint8_t index)
svint16_t svdup_lane[_s16](svint16_t data, uint16_t index)
svint32_t svdup_lane[_s32](svint32_t data, uint32_t index)
svint64_t svdup_lane[_s64](svint64_t data, uint64_t index)
svuint8_t svdup_lane[_u8](svuint8_t data, uint8_t index)
svuint16_t svdup_lane[_u16](svuint16_t data, uint16_t index)
svuint32_t svdup_lane[_u32](svuint32_t data, uint32_t index)
svuint64_t svdup_lane[_u64](svuint64_t data, uint64_t index)
svfloat16_t svdup_lane[_f16](svfloat16_t data, uint16_t index)
svfloat32_t svdup_lane[_f32](svfloat32_t data, uint32_t index)
svfloat64_t svdup_lane[_f64](svfloat64_t data, uint64_t index)
svbfloat16_t svdup_lane[_bf16](svbfloat16_t data, uint16_t index)
```

**6.20.10. DUPQ: Duplicate one quadword of a vector**

These functions take one 128-bit quadword from the vector input and duplicate it to fill an entire vector. The second input specifies the index of the quadword to duplicate, with quadword 0 containing element 0. Although there is one instance of this operation for every vector type, the results are bitwise identical.

More precisely, the behavior of:

```
svdupq_lane_u64(data, index)
```

is identical to:

```
svtbl(data, svadd_x(svptrue_b64(),
                    svand_x(svptrue_b64(), svindex_u64(0, 1), 1),
                    index * 2))
```

although the implementation can use any instruction sequence that achieves this effect. The other functions effectively reinterpret the input as `svuint64_t`, perform the operation above, then reinterpret the result as the original input type.

When *index* is a constant in the range [0, 3], the operation is equivalent to a single `.Q DUP (indexed)` instruction.



### 6.20.10.1. DUPQ (vector, scalar)

Instances
<pre> svint8_t  svdupq_lane[_s8](svint8_t data, uint64_t index) svint16_t svdupq_lane[_s16](svint16_t data, uint64_t index) svint32_t svdupq_lane[_s32](svint32_t data, uint64_t index) svint64_t svdupq_lane[_s64](svint64_t data, uint64_t index) svuint8_t svdupq_lane[_u8](svuint8_t data, uint64_t index) svuint16_t svdupq_lane[_u16](svuint16_t data, uint64_t index) svuint32_t svdupq_lane[_u32](svuint32_t data, uint64_t index) svuint64_t svdupq_lane[_u64](svuint64_t data, uint64_t index) svfloat16_t svdupq_lane[_f16](svfloat16_t data, uint64_t index) svfloat32_t svdupq_lane[_f32](svfloat32_t data, uint64_t index) svfloat64_t svdupq_lane[_f64](svfloat64_t data, uint64_t index) svbfloat16_t svdupq_lane[_bf16](svbfloat16_t data, uint64_t index) </pre>

## 6.20.11. TBL: Table lookup/permute using vector of indices

These functions return a vector in which element *N* contains the element of the first input vector specified by the index value in element *N* of the second input vector. Out-of-range indices select the value zero.

### 6.20.11.1. TBL (vector, vector)

Instances
<pre> svint8_t  svtbl[_s8](svint8_t data, svuint8_t indices) svint16_t svtbl[_s16](svint16_t data, svuint16_t indices) svint32_t svtbl[_s32](svint32_t data, svuint32_t indices) svint64_t svtbl[_s64](svint64_t data, svuint64_t indices) svuint8_t svtbl[_u8](svuint8_t data, svuint8_t indices) svuint16_t svtbl[_u16](svuint16_t data, svuint16_t indices) svuint32_t svtbl[_u32](svuint32_t data, svuint32_t indices) svuint64_t svtbl[_u64](svuint64_t data, svuint64_t indices) svfloat16_t svtbl[_f16](svfloat16_t data, svuint16_t indices) svfloat32_t svtbl[_f32](svfloat32_t data, svuint32_t indices) svfloat64_t svtbl[_f64](svfloat64_t data, svuint64_t indices) svbfloat16_t svtbl[_bf16](svbfloat16_t data, svuint16_t indices) </pre>

## 6.20.12. REV: Reverse the elements in a single input

These functions reverse the order of the elements in a single predicate or vector.

### 6.20.12.1. REV (vector)

Instances
<pre> svint8_t  svrev[_s8](svint8_t op) svint16_t svrev[_s16](svint16_t op) svint32_t svrev[_s32](svint32_t op) svint64_t svrev[_s64](svint64_t op) svuint8_t svrev[_u8](svuint8_t op) svuint16_t svrev[_u16](svuint16_t op) svuint32_t svrev[_u32](svuint32_t op) svuint64_t svrev[_u64](svuint64_t op) svfloat16_t svrev[_f16](svfloat16_t op) svfloat32_t svrev[_f32](svfloat32_t op) svfloat64_t svrev[_f64](svfloat64_t op) svbfloat16_t svrev[_bf16](svbfloat16_t op) </pre>

### 6.20.12.2. REV (predicate)

Instances
svbool_t <b>svrev_b8</b> (svbool_t op)
svbool_t <b>svrev_b16</b> (svbool_t op)
svbool_t <b>svrev_b32</b> (svbool_t op)
svbool_t <b>svrev_b64</b> (svbool_t op)

### 6.20.13. TRN1: Interleave even elements from two inputs

These functions extract the even-indexed elements from two vectors or predicates and interleave them, so that each even element of the first input is followed by the corresponding element of the second input.

#### 6.20.13.1. TRN1 (vector, vector)

Instances
svint8_t <b>svtrn1[_s8]</b> (svint8_t op1, svint8_t op2)
svint16_t <b>svtrn1[_s16]</b> (svint16_t op1, svint16_t op2)
svint32_t <b>svtrn1[_s32]</b> (svint32_t op1, svint32_t op2)
svint64_t <b>svtrn1[_s64]</b> (svint64_t op1, svint64_t op2)
svuint8_t <b>svtrn1[_u8]</b> (svuint8_t op1, svuint8_t op2)
svuint16_t <b>svtrn1[_u16]</b> (svuint16_t op1, svuint16_t op2)
svuint32_t <b>svtrn1[_u32]</b> (svuint32_t op1, svuint32_t op2)
svuint64_t <b>svtrn1[_u64]</b> (svuint64_t op1, svuint64_t op2)
svfloat16_t <b>svtrn1[_f16]</b> (svfloat16_t op1, svfloat16_t op2)
svfloat32_t <b>svtrn1[_f32]</b> (svfloat32_t op1, svfloat32_t op2)
svfloat64_t <b>svtrn1[_f64]</b> (svfloat64_t op1, svfloat64_t op2)
svbfloat16_t <b>svtrn1[_bf16]</b> (svbfloat16_t op1, svbfloat16_t op2)

#### 6.20.13.2. TRN1 (predicate, predicate)

Instances
svbool_t <b>svtrn1_b8</b> (svbool_t op1, svbool_t op2)
svbool_t <b>svtrn1_b16</b> (svbool_t op1, svbool_t op2)
svbool_t <b>svtrn1_b32</b> (svbool_t op1, svbool_t op2)
svbool_t <b>svtrn1_b64</b> (svbool_t op1, svbool_t op2)

### 6.20.14. TRN2: Interleave odd elements from two inputs

These functions extract the odd-indexed elements from two vectors or predicates and interleave them, so that each odd element of the first input is followed by the corresponding element of the second input.

#### 6.20.14.1. TRN2 (vector, vector)

Instances
svint8_t <b>svtrn2[_s8]</b> (svint8_t op1, svint8_t op2)
svint16_t <b>svtrn2[_s16]</b> (svint16_t op1, svint16_t op2)
svint32_t <b>svtrn2[_s32]</b> (svint32_t op1, svint32_t op2)
svint64_t <b>svtrn2[_s64]</b> (svint64_t op1, svint64_t op2)
svuint8_t <b>svtrn2[_u8]</b> (svuint8_t op1, svuint8_t op2)
svuint16_t <b>svtrn2[_u16]</b> (svuint16_t op1, svuint16_t op2)
svuint32_t <b>svtrn2[_u32]</b> (svuint32_t op1, svuint32_t op2)
svuint64_t <b>svtrn2[_u64]</b> (svuint64_t op1, svuint64_t op2)
svfloat16_t <b>svtrn2[_f16]</b> (svfloat16_t op1, svfloat16_t op2)
svfloat32_t <b>svtrn2[_f32]</b> (svfloat32_t op1, svfloat32_t op2)

**Instances**

```
svfloat64_t svtrn2[_f64](svfloat64_t op1, svfloat64_t op2)
svbfloat16_t svtrn2[_bf16](svbfloat16_t op1, svbfloat16_t op2)
```

**6.20.14.2. TRN2 (predicate, predicate)****Instances**

```
svbool_t svtrn2_b8(svbool_t op1, svbool_t op2)
svbool_t svtrn2_b16(svbool_t op1, svbool_t op2)
svbool_t svtrn2_b32(svbool_t op1, svbool_t op2)
svbool_t svtrn2_b64(svbool_t op1, svbool_t op2)
```

**6.20.15. UNPKHI: Unpack and extend high half of an input**

These functions extract the highest-indexed half of a vector or predicate and extend each element to double the width. The vector forms use sign extension if the results are signed and zero extension if the results are unsigned. The predicate form extends with false bits.

**6.20.15.1. UNPKHI (vector)****Instances**

```
svint16_t svunpkhi[_s16](svint8_t op)
svint32_t svunpkhi[_s32](svint16_t op)
svint64_t svunpkhi[_s64](svint32_t op)
svuint16_t svunpkhi[_u16](svuint8_t op)
svuint32_t svunpkhi[_u32](svuint16_t op)
svuint64_t svunpkhi[_u64](svuint32_t op)
```

**6.20.15.2. UNPKHI (predicate)****Instances**

```
svbool_t svunpkhi[_b](svbool_t op)
```

**6.20.16. UNPKLO: Unpack and extend low half of an input**

These functions extract the lowest-indexed half of a vector or predicate and extend each element to double the width. The vector forms use sign extension if the results are signed and zero extension if the results are unsigned. The predicate form extends with false bits.

**6.20.16.1. UNPKLO (vector)****Instances**

```
svint16_t svunpklo[_s16](svint8_t op)
svint32_t svunpklo[_s32](svint16_t op)
svint64_t svunpklo[_s64](svint32_t op)
svuint16_t svunpklo[_u16](svuint8_t op)
svuint32_t svunpklo[_u32](svuint16_t op)
svuint64_t svunpklo[_u64](svuint32_t op)
```

**6.20.16.2. UNPKLO (predicate)****Instances**

```
svbool_t svunpklo[_b](svbool_t op)
```

## 6.20.17. UZP1: Select even elements from two inputs

These functions extract the even-indexed elements from two vectors or predicates and concatenate them together.

### 6.20.17.1. UZP1 (vector, vector)

Instances
<pre> svint8_t  svuzp1[_s8](svint8_t op1, svint8_t op2) svint16_t svuzp1[_s16](svint16_t op1, svint16_t op2) svint32_t svuzp1[_s32](svint32_t op1, svint32_t op2) svint64_t svuzp1[_s64](svint64_t op1, svint64_t op2) svuint8_t svuzp1[_u8](svuint8_t op1, svuint8_t op2) svuint16_t svuzp1[_u16](svuint16_t op1, svuint16_t op2) svuint32_t svuzp1[_u32](svuint32_t op1, svuint32_t op2) svuint64_t svuzp1[_u64](svuint64_t op1, svuint64_t op2) svfloat16_t svuzp1[_f16](svfloat16_t op1, svfloat16_t op2) svfloat32_t svuzp1[_f32](svfloat32_t op1, svfloat32_t op2) svfloat64_t svuzp1[_f64](svfloat64_t op1, svfloat64_t op2) svbfloat16_t svuzp1[_bf16](svbfloat16_t op1, svbfloat16_t op2) </pre>

### 6.20.17.2. UZP1 (predicate, predicate)

Instances
<pre> svbool_t svuzp1_b8(svbool_t op1, svbool_t op2) svbool_t svuzp1_b16(svbool_t op1, svbool_t op2) svbool_t svuzp1_b32(svbool_t op1, svbool_t op2) svbool_t svuzp1_b64(svbool_t op1, svbool_t op2) </pre>

## 6.20.18. UZP2: Select odd elements from two inputs

These functions extract the odd-indexed elements from two vectors or predicates and concatenate them together.

### 6.20.18.1. UZP2 (vector, vector)

Instances
<pre> svint8_t  svuzp2[_s8](svint8_t op1, svint8_t op2) svint16_t svuzp2[_s16](svint16_t op1, svint16_t op2) svint32_t svuzp2[_s32](svint32_t op1, svint32_t op2) svint64_t svuzp2[_s64](svint64_t op1, svint64_t op2) svuint8_t svuzp2[_u8](svuint8_t op1, svuint8_t op2) svuint16_t svuzp2[_u16](svuint16_t op1, svuint16_t op2) svuint32_t svuzp2[_u32](svuint32_t op1, svuint32_t op2) svuint64_t svuzp2[_u64](svuint64_t op1, svuint64_t op2) svfloat16_t svuzp2[_f16](svfloat16_t op1, svfloat16_t op2) svfloat32_t svuzp2[_f32](svfloat32_t op1, svfloat32_t op2) svfloat64_t svuzp2[_f64](svfloat64_t op1, svfloat64_t op2) svbfloat16_t svuzp2[_bf16](svbfloat16_t op1, svbfloat16_t op2) </pre>

### 6.20.18.2. UZP2 (predicate, predicate)

Instances
<pre> svbool_t svuzp2_b8(svbool_t op1, svbool_t op2) </pre>

**Instances**

```
svbool_t svuzp2_b16(svbool_t op1, svbool_t op2)
svbool_t svuzp2_b32(svbool_t op1, svbool_t op2)
svbool_t svuzp2_b64(svbool_t op1, svbool_t op2)
```

**6.20.19. ZIP1: Interleave elements from low halves of two inputs**

These functions extract elements from the lowest-indexed halves of two vectors or predicates and interleave them, so that each low element of the first input is followed by the corresponding element of the second input.

**6.20.19.1. ZIP1 (vector, vector)****Instances**

```
svint8_t svzip1[_s8](svint8_t op1, svint8_t op2)
svint16_t svzip1[_s16](svint16_t op1, svint16_t op2)
svint32_t svzip1[_s32](svint32_t op1, svint32_t op2)
svint64_t svzip1[_s64](svint64_t op1, svint64_t op2)
svuint8_t svzip1[_u8](svuint8_t op1, svuint8_t op2)
svuint16_t svzip1[_u16](svuint16_t op1, svuint16_t op2)
svuint32_t svzip1[_u32](svuint32_t op1, svuint32_t op2)
svuint64_t svzip1[_u64](svuint64_t op1, svuint64_t op2)
svfloat16_t svzip1[_f16](svfloat16_t op1, svfloat16_t op2)
svfloat32_t svzip1[_f32](svfloat32_t op1, svfloat32_t op2)
svfloat64_t svzip1[_f64](svfloat64_t op1, svfloat64_t op2)
svbfloat16_t svzip1[_bf16](svbfloat16_t op1, svbfloat16_t op2)
```

**6.20.19.2. ZIP1 (predicate, predicate)****Instances**

```
svbool_t svzip1_b8(svbool_t op1, svbool_t op2)
svbool_t svzip1_b16(svbool_t op1, svbool_t op2)
svbool_t svzip1_b32(svbool_t op1, svbool_t op2)
svbool_t svzip1_b64(svbool_t op1, svbool_t op2)
```

**6.20.20. ZIP2: Interleave elements from high halves of two inputs**

These functions extract elements from the highest-indexed halves of two vectors or predicates and interleave them, so that each high element of the first input is followed by the corresponding element of the second input.

**6.20.20.1. ZIP2 (vector, vector)****Instances**

```
svint8_t svzip2[_s8](svint8_t op1, svint8_t op2)
svint16_t svzip2[_s16](svint16_t op1, svint16_t op2)
svint32_t svzip2[_s32](svint32_t op1, svint32_t op2)
svint64_t svzip2[_s64](svint64_t op1, svint64_t op2)
svuint8_t svzip2[_u8](svuint8_t op1, svuint8_t op2)
svuint16_t svzip2[_u16](svuint16_t op1, svuint16_t op2)
svuint32_t svzip2[_u32](svuint32_t op1, svuint32_t op2)
svuint64_t svzip2[_u64](svuint64_t op1, svuint64_t op2)
svfloat16_t svzip2[_f16](svfloat16_t op1, svfloat16_t op2)
svfloat32_t svzip2[_f32](svfloat32_t op1, svfloat32_t op2)
```

**Instances**

```
svfloat64_t svzip2[_f64](svfloat64_t op1, svfloat64_t op2)
svbfloat16_t svzip2[_bf16](svbfloat16_t op1, svbfloat16_t op2)
```

**6.20.20.2. ZIP2 (predicate, predicate)****Instances**

```
svbool_t svzip2_b8(svbool_t op1, svbool_t op2)
svbool_t svzip2_b16(svbool_t op1, svbool_t op2)
svbool_t svzip2_b32(svbool_t op1, svbool_t op2)
svbool_t svzip2_b64(svbool_t op1, svbool_t op2)
```

**6.21. Vector creation****6.21.1. CREATE2: Create a tuple of two vectors**

These functions combine two individual vectors of the same type into a tuple of two vectors.

**6.21.1.1. CREATE2****Instances**

```
svint8x2_t svcreate2[_s8](svint8_t x0, svint8_t x1)
svint16x2_t svcreate2[_s16](svint16_t x0, svint16_t x1)
svint32x2_t svcreate2[_s32](svint32_t x0, svint32_t x1)
svint64x2_t svcreate2[_s64](svint64_t x0, svint64_t x1)
svuint8x2_t svcreate2[_u8](svuint8_t x0, svuint8_t x1)
svuint16x2_t svcreate2[_u16](svuint16_t x0, svuint16_t x1)
svuint32x2_t svcreate2[_u32](svuint32_t x0, svuint32_t x1)
svuint64x2_t svcreate2[_u64](svuint64_t x0, svuint64_t x1)
svfloat16x2_t svcreate2[_f16](svfloat16_t x0, svfloat16_t x1)
svfloat32x2_t svcreate2[_f32](svfloat32_t x0, svfloat32_t x1)
svfloat64x2_t svcreate2[_f64](svfloat64_t x0, svfloat64_t x1)
svbfloat16x2_t svcreate2[_bf16](svbfloat16_t x0, svbfloat16_t x1)
```

**6.21.2. CREATE3: Create a tuple of three vectors**

These functions combine three individual vectors of the same type into a tuple of three vectors.

**6.21.2.1. CREATE3****Instances**

```
svint8x3_t svcreate3[_s8](svint8_t x0, svint8_t x1, svint8_t x2)
svint16x3_t svcreate3[_s16](svint16_t x0, svint16_t x1, svint16_t x2)
svint32x3_t svcreate3[_s32](svint32_t x0, svint32_t x1, svint32_t x2)
svint64x3_t svcreate3[_s64](svint64_t x0, svint64_t x1, svint64_t x2)
svuint8x3_t svcreate3[_u8](svuint8_t x0, svuint8_t x1, svuint8_t x2)
svuint16x3_t svcreate3[_u16](svuint16_t x0, svuint16_t x1, svuint16_t x2)
svuint32x3_t svcreate3[_u32](svuint32_t x0, svuint32_t x1, svuint32_t x2)
svuint64x3_t svcreate3[_u64](svuint64_t x0, svuint64_t x1, svuint64_t x2)
svfloat16x3_t svcreate3[_f16](svfloat16_t x0, svfloat16_t x1,
                               svfloat16_t x2)
svfloat32x3_t svcreate3[_f32](svfloat32_t x0, svfloat32_t x1,
                               svfloat32_t x2)
svfloat64x3_t svcreate3[_f64](svfloat64_t x0, svfloat64_t x1,
```

**Instances**

```

svfloat64_t x2)
svbfloat16x3_t svcreate3[_bf16](svbfloat16_t x0, svbfloat16_t x1,
svbfloat16_t x2)

```

**6.21.3. CREATE4: Create a tuple of four vectors**

These functions combine four individual vectors of the same type into a tuple of four vectors.

**6.21.3.1. CREATE4****Instances**

```

svint8x4_t svcreate4[_s8](svint8_t x0, svint8_t x1, svint8_t x2,
svint8_t x3)
svint16x4_t svcreate4[_s16](svint16_t x0, svint16_t x1, svint16_t x2,
svint16_t x3)
svint32x4_t svcreate4[_s32](svint32_t x0, svint32_t x1, svint32_t x2,
svint32_t x3)
svint64x4_t svcreate4[_s64](svint64_t x0, svint64_t x1, svint64_t x2,
svint64_t x3)
svuint8x4_t svcreate4[_u8](svuint8_t x0, svuint8_t x1, svuint8_t x2,
svuint8_t x3)
svuint16x4_t svcreate4[_u16](svuint16_t x0, svuint16_t x1, svuint16_t x2,
svuint16_t x3)
svuint32x4_t svcreate4[_u32](svuint32_t x0, svuint32_t x1, svuint32_t x2,
svuint32_t x3)
svuint64x4_t svcreate4[_u64](svuint64_t x0, svuint64_t x1, svuint64_t x2,
svuint64_t x3)
svfloat16x4_t svcreate4[_f16](svfloat16_t x0, svfloat16_t x1,
svfloat16_t x2, svfloat16_t x3)
svfloat32x4_t svcreate4[_f32](svfloat32_t x0, svfloat32_t x1,
svfloat32_t x2, svfloat32_t x3)
svfloat64x4_t svcreate4[_f64](svfloat64_t x0, svfloat64_t x1,
svfloat64_t x2, svfloat64_t x3)
svbfloat16x4_t svcreate4[_bf16](svbfloat16_t x0, svbfloat16_t x1,
svbfloat16_t x2, svbfloat16_t x3)

```

**6.21.4. UNDEF: Create an uninitialized vector**

These functions return a vector with arbitrary, undefined contents. The functions can be useful for “seeding” a sequence of operations that cumulatively leaves all active elements in a defined state.

**6.21.4.1. UNDEF (inherent)****Instances**

```

svint8_t svundef_s8()
svint16_t svundef_s16()
svint32_t svundef_s32()
svint64_t svundef_s64()
svuint8_t svundef_u8()
svuint16_t svundef_u16()
svuint32_t svundef_u32()
svuint64_t svundef_u64()
svfloat16_t svundef_f16()
svfloat32_t svundef_f32()
svfloat64_t svundef_f64()

```

Instances
svbfloat16_t <b>svundef_bf16()</b>

## 6.21.5. UNDEF2: Create an uninitialized tuple of two vectors

These functions return a tuple of two vectors with arbitrary, undefined contents. The functions can be useful for “seeding” a sequence of operations that cumulatively leaves all active elements in a defined state.

### 6.21.5.1. UNDEF2 (inherent)

Instances
svint8x2_t <b>svundef2_s8()</b> svint16x2_t <b>svundef2_s16()</b> svint32x2_t <b>svundef2_s32()</b> svint64x2_t <b>svundef2_s64()</b> svuint8x2_t <b>svundef2_u8()</b> svuint16x2_t <b>svundef2_u16()</b> svuint32x2_t <b>svundef2_u32()</b> svuint64x2_t <b>svundef2_u64()</b> svfloat16x2_t <b>svundef2_f16()</b> svfloat32x2_t <b>svundef2_f32()</b> svfloat64x2_t <b>svundef2_f64()</b> svbfloat16x2_t <b>svundef2_bf16()</b>

## 6.21.6. UNDEF3: Create an uninitialized tuple of three vectors

These functions return a tuple of three vectors with arbitrary, undefined contents. The functions can be useful for “seeding” a sequence of operations that cumulatively leaves all active elements in a defined state.

### 6.21.6.1. UNDEF3 (inherent)

Instances
svint8x3_t <b>svundef3_s8()</b> svint16x3_t <b>svundef3_s16()</b> svint32x3_t <b>svundef3_s32()</b> svint64x3_t <b>svundef3_s64()</b> svuint8x3_t <b>svundef3_u8()</b> svuint16x3_t <b>svundef3_u16()</b> svuint32x3_t <b>svundef3_u32()</b> svuint64x3_t <b>svundef3_u64()</b> svfloat16x3_t <b>svundef3_f16()</b> svfloat32x3_t <b>svundef3_f32()</b> svfloat64x3_t <b>svundef3_f64()</b> svbfloat16x3_t <b>svundef3_bf16()</b>

## 6.21.7. UNDEF4: Create an uninitialized tuple of four vectors

These functions return a tuple of four vectors with arbitrary, undefined contents. The functions can be useful for “seeding” a sequence of operations that cumulatively leaves all active elements in a defined state.

### 6.21.7.1. UNDEF4 (inherent)

Instances
svint8x4_t <b>svundef4_s8()</b> svint16x4_t <b>svundef4_s16()</b>



**Instances**

```

svint32x4_t  svundef4_s32()
svint64x4_t  svundef4_s64()
svuint8x4_t  svundef4_u8()
svuint16x4_t svundef4_u16()
svuint32x4_t svundef4_u32()
svuint64x4_t svundef4_u64()
svfloat16x4_t svundef4_f16()
svfloat32x4_t svundef4_f32()
svfloat64x4_t svundef4_f64()
svbfloat16x4_t svundef4_bf16()

```

## 6.22. Vector insertion and extraction

### 6.22.1. SET2: Change one vector in a tuple of two vectors

These functions return a tuple of two vectors like *tuple* except that vector *imm\_index* has value *x*. *imm\_index* must be an integer constant expression in the range [0, 1].

#### 6.22.1.1. SET2

**Instances**

```

svint8x2_t  svset2[_s8](svint8x2_t tuple, uint64_t imm_index, svint8_t x)
svint16x2_t svset2[_s16](svint16x2_t tuple, uint64_t imm_index,
                          svint16_t x)
svint32x2_t svset2[_s32](svint32x2_t tuple, uint64_t imm_index,
                          svint32_t x)
svint64x2_t svset2[_s64](svint64x2_t tuple, uint64_t imm_index,
                          svint64_t x)
svuint8x2_t svset2[_u8](svuint8x2_t tuple, uint64_t imm_index, svuint8_t x)
svuint16x2_t svset2[_u16](svuint16x2_t tuple, uint64_t imm_index,
                          svuint16_t x)
svuint32x2_t svset2[_u32](svuint32x2_t tuple, uint64_t imm_index,
                          svuint32_t x)
svuint64x2_t svset2[_u64](svuint64x2_t tuple, uint64_t imm_index,
                          svuint64_t x)
svfloat16x2_t svset2[_f16](svfloat16x2_t tuple, uint64_t imm_index,
                          svfloat16_t x)
svfloat32x2_t svset2[_f32](svfloat32x2_t tuple, uint64_t imm_index,
                          svfloat32_t x)
svfloat64x2_t svset2[_f64](svfloat64x2_t tuple, uint64_t imm_index,
                          svfloat64_t x)
svbfloat16x2_t svset2[_bf16](svbfloat16x2_t tuple, uint64_t imm_index,
                          svbfloat16_t x)

```

### 6.22.2. SET3: Change one vector in a tuple of three vectors

These functions return a tuple of three vectors like *tuple* except that vector *imm\_index* has value *x*. *imm\_index* must be an integer constant expression in the range [0, 2].

#### 6.22.2.1. SET3

**Instances**

```

svint8x3_t  svset3[_s8](svint8x3_t tuple, uint64_t imm_index, svint8_t x)
svint16x3_t svset3[_s16](svint16x3_t tuple, uint64_t imm_index,

```

Instances	
<code>svint16_t x)</code>	
<code>svint32x3_t svset3[_s32](svint32x3_t tuple, uint64_t imm_index,</code>	
<code>svint32_t x)</code>	
<code>svint64x3_t svset3[_s64](svint64x3_t tuple, uint64_t imm_index,</code>	
<code>svint64_t x)</code>	
<code>svuint8x3_t svset3[_u8](svuint8x3_t tuple, uint64_t imm_index, svuint8_t x)</code>	
<code>svuint16x3_t svset3[_u16](svuint16x3_t tuple, uint64_t imm_index,</code>	
<code>svuint16_t x)</code>	
<code>svuint32x3_t svset3[_u32](svuint32x3_t tuple, uint64_t imm_index,</code>	
<code>svuint32_t x)</code>	
<code>svuint64x3_t svset3[_u64](svuint64x3_t tuple, uint64_t imm_index,</code>	
<code>svuint64_t x)</code>	
<code>svfloat16x3_t svset3[_f16](svfloat16x3_t tuple, uint64_t imm_index,</code>	
<code>svfloat16_t x)</code>	
<code>svfloat32x3_t svset3[_f32](svfloat32x3_t tuple, uint64_t imm_index,</code>	
<code>svfloat32_t x)</code>	
<code>svfloat64x3_t svset3[_f64](svfloat64x3_t tuple, uint64_t imm_index,</code>	
<code>svfloat64_t x)</code>	
<code>svbfloat16x3_t svset3[_bf16](svbfloat16x3_t tuple, uint64_t imm_index,</code>	
<code>svbfloat16_t x)</code>	

### 6.22.3. SET4: Change one vector in a tuple of four vectors

These functions return a tuple of four vectors like *tuple* except that vector *imm\_index* has value *x*. *imm\_index* must be an integer constant expression in the range [0, 3].

#### 6.22.3.1. SET4

Instances	
<code>svint8x4_t svset4[_s8](svint8x4_t tuple, uint64_t imm_index, svint8_t x)</code>	
<code>svint16x4_t svset4[_s16](svint16x4_t tuple, uint64_t imm_index,</code>	
<code>svint16_t x)</code>	
<code>svint32x4_t svset4[_s32](svint32x4_t tuple, uint64_t imm_index,</code>	
<code>svint32_t x)</code>	
<code>svint64x4_t svset4[_s64](svint64x4_t tuple, uint64_t imm_index,</code>	
<code>svint64_t x)</code>	
<code>svuint8x4_t svset4[_u8](svuint8x4_t tuple, uint64_t imm_index, svuint8_t x)</code>	
<code>svuint16x4_t svset4[_u16](svuint16x4_t tuple, uint64_t imm_index,</code>	
<code>svuint16_t x)</code>	
<code>svuint32x4_t svset4[_u32](svuint32x4_t tuple, uint64_t imm_index,</code>	
<code>svuint32_t x)</code>	
<code>svuint64x4_t svset4[_u64](svuint64x4_t tuple, uint64_t imm_index,</code>	
<code>svuint64_t x)</code>	
<code>svfloat16x4_t svset4[_f16](svfloat16x4_t tuple, uint64_t imm_index,</code>	
<code>svfloat16_t x)</code>	
<code>svfloat32x4_t svset4[_f32](svfloat32x4_t tuple, uint64_t imm_index,</code>	
<code>svfloat32_t x)</code>	
<code>svfloat64x4_t svset4[_f64](svfloat64x4_t tuple, uint64_t imm_index,</code>	
<code>svfloat64_t x)</code>	
<code>svbfloat16x4_t svset4[_bf16](svbfloat16x4_t tuple, uint64_t imm_index,</code>	
<code>svbfloat16_t x)</code>	

### 6.22.4. GET2: Extract one vector from a tuple of two vectors

These functions extract vector *imm\_index* from the tuple of two vectors given by *tuple*. *imm\_index* must be an integer constant expression in the range [0, 1].

### 6.22.4.1. GET2

#### Instances

```
svint8_t  svget2[_s8](svint8x2_t tuple, uint64_t imm_index)
svint16_t svget2[_s16](svint16x2_t tuple, uint64_t imm_index)
svint32_t svget2[_s32](svint32x2_t tuple, uint64_t imm_index)
svint64_t svget2[_s64](svint64x2_t tuple, uint64_t imm_index)
svuint8_t svget2[_u8](svuint8x2_t tuple, uint64_t imm_index)
svuint16_t svget2[_u16](svuint16x2_t tuple, uint64_t imm_index)
svuint32_t svget2[_u32](svuint32x2_t tuple, uint64_t imm_index)
svuint64_t svget2[_u64](svuint64x2_t tuple, uint64_t imm_index)
svfloat16_t svget2[_f16](svfloat16x2_t tuple, uint64_t imm_index)
svfloat32_t svget2[_f32](svfloat32x2_t tuple, uint64_t imm_index)
svfloat64_t svget2[_f64](svfloat64x2_t tuple, uint64_t imm_index)
svbfloat16_t svget2[_bf16](svbfloat16x2_t tuple, uint64_t imm_index)
```

### 6.22.5. GET3: Extract one vector from a tuple of three vectors

These functions extract vector *imm\_index* from the tuple of three vectors given by *tuple*. *imm\_index* must be an integer constant expression in the range [0, 2].

#### 6.22.5.1. GET3

#### Instances

```
svint8_t  svget3[_s8](svint8x3_t tuple, uint64_t imm_index)
svint16_t svget3[_s16](svint16x3_t tuple, uint64_t imm_index)
svint32_t svget3[_s32](svint32x3_t tuple, uint64_t imm_index)
svint64_t svget3[_s64](svint64x3_t tuple, uint64_t imm_index)
svuint8_t svget3[_u8](svuint8x3_t tuple, uint64_t imm_index)
svuint16_t svget3[_u16](svuint16x3_t tuple, uint64_t imm_index)
svuint32_t svget3[_u32](svuint32x3_t tuple, uint64_t imm_index)
svuint64_t svget3[_u64](svuint64x3_t tuple, uint64_t imm_index)
svfloat16_t svget3[_f16](svfloat16x3_t tuple, uint64_t imm_index)
svfloat32_t svget3[_f32](svfloat32x3_t tuple, uint64_t imm_index)
svfloat64_t svget3[_f64](svfloat64x3_t tuple, uint64_t imm_index)
svbfloat16_t svget3[_bf16](svbfloat16x3_t tuple, uint64_t imm_index)
```

### 6.22.6. GET4: Extract one vector from a tuple of four vectors

These functions extract vector *imm\_index* from the tuple of four vectors given by *tuple*. *imm\_index* must be an integer constant expression in the range [0, 3].

#### 6.22.6.1. GET4

#### Instances

```
svint8_t  svget4[_s8](svint8x4_t tuple, uint64_t imm_index)
svint16_t svget4[_s16](svint16x4_t tuple, uint64_t imm_index)
svint32_t svget4[_s32](svint32x4_t tuple, uint64_t imm_index)
svint64_t svget4[_s64](svint64x4_t tuple, uint64_t imm_index)
svuint8_t svget4[_u8](svuint8x4_t tuple, uint64_t imm_index)
svuint16_t svget4[_u16](svuint16x4_t tuple, uint64_t imm_index)
svuint32_t svget4[_u32](svuint32x4_t tuple, uint64_t imm_index)
svuint64_t svget4[_u64](svuint64x4_t tuple, uint64_t imm_index)
svfloat16_t svget4[_f16](svfloat16x4_t tuple, uint64_t imm_index)
svfloat32_t svget4[_f32](svfloat32x4_t tuple, uint64_t imm_index)
svfloat64_t svget4[_f64](svfloat64x4_t tuple, uint64_t imm_index)
```

**Instances**

```
svbfloat16_t svget4[_bf16](svbfloat16x4_t tuple, uint64_t imm_index)
```

## 6.23. Predicate creation

### 6.23.1. PTRUE: Return an all-true predicate for a given pattern

These functions return an all-true predicate for a particular vector pattern and element size. When an element has more than one predicate bit associated with it, only the lowest of those bits is ever true.

There are two forms: one with a `_pat` suffix that takes an explicit vector pattern and one without a `_pat` suffix in which the pattern is implicitly `SV_ALL`.

#### 6.23.1.1. PTRUE (inherent)

**Instances**

```
svbool_t svptrue_b8()
svbool_t svptrue_b16()
svbool_t svptrue_b32()
svbool_t svptrue_b64()
```

#### 6.23.1.2. PTRUE (pattern)

**Instances**

```
svbool_t svptrue_pat_b8(enum svpattern pattern)
svbool_t svptrue_pat_b16(enum svpattern pattern)
svbool_t svptrue_pat_b32(enum svpattern pattern)
svbool_t svptrue_pat_b64(enum svpattern pattern)
```

### 6.23.2. PFALSE: Return an all-false predicate

This function returns a predicate in which every bit is false.

#### 6.23.2.1. PFALSE (inherent)

**Instances**

```
svbool_t svpfalse[_b]()
```

### 6.23.3. DUP: Duplicate boolean value

These functions copy a boolean value into every element of a predicate. More precisely, the behavior of:

```
svdup_n_bNN(op)
```

is identical to:

```
op ? svptrue_bNN() : svpfalse_b()
```

although the implementation can use any instruction sequence that achieves the same effect.

#### 6.23.3.1. DUP (scalar)

**Instances**

```
svbool_t svdup[_n]_b8(bool op)
```

**Instances**

```
svbool_t svdup[_n]_b16(bool op)
svbool_t svdup[_n]_b32(bool op)
svbool_t svdup[_n]_b64(bool op)
```

**6.23.4. DUPQ: Duplicate boolean values to fill a predicate**

These functions take enough booleans to control 128 bits of data, in element order. They replicate this sequence to fill an entire predicate and return the result. For example:

```
svdupq_n_b64(x0, x1)
```

is identical to:

```
svcmpne_n_u64(svptrue_b64(), svdupq_n_u64((bool)x0, (bool)x1), 0)
```

although the implementation can use any instruction sequence that achieves the same effect.

**6.23.4.1. DUPQ (16 scalars)****Instances**

```
svbool_t svdupq[_n]_b8(bool x0, bool x1, bool x2, bool x3, bool x4, bool x5,
                        bool x6, bool x7, bool x8, bool x9, bool x10,
                        bool x11, bool x12, bool x13, bool x14, bool x15)
```

**6.23.4.2. DUPQ (8 scalars)****Instances**

```
svbool_t svdupq[_n]_b16(bool x0, bool x1, bool x2, bool x3, bool x4,
                        bool x5, bool x6, bool x7)
```

**6.23.4.3. DUPQ (4 scalars)****Instances**

```
svbool_t svdupq[_n]_b32(bool x0, bool x1, bool x2, bool x3)
```

**6.23.4.4. DUPQ (2 scalars)****Instances**

```
svbool_t svdupq[_n]_b64(bool x0, bool x1)
```

**6.24. Predicate operations****6.24.1. MOV: Copy predicate**

These functions take a copy of the active elements of a predicate.

**6.24.1.1. MOV (predicate), setting inactive to zero****Instances**

```
svbool_t svmov[_b]_z(svbool_t pg, svbool_t op)
```

## 6.24.2. AND: Predicate AND

These functions perform an AND of two predicate inputs.

### 6.24.2.1. AND (predicate, predicate), setting inactive to zero

Instances
svbool_t <b>svand</b> [_b]_z(svbool_t <i>pg</i> , svbool_t <i>op1</i> , svbool_t <i>op2</i> )

## 6.24.3. BIC: Predicate AND NOT

These functions invert the second predicate input and AND it with the first predicate input.

### 6.24.3.1. BIC (predicate, predicate), setting inactive to zero

Instances
svbool_t <b>svbic</b> [_b]_z(svbool_t <i>pg</i> , svbool_t <i>op1</i> , svbool_t <i>op2</i> )

## 6.24.4. NAND: Predicate NAND

These functions perform an AND of two predicate inputs and invert the result.

### 6.24.4.1. NAND (predicate, predicate), setting inactive to zero

Instances
svbool_t <b>svnand</b> [_b]_z(svbool_t <i>pg</i> , svbool_t <i>op1</i> , svbool_t <i>op2</i> )

## 6.24.5. ORR: Predicate OR

These functions perform an OR of two predicate inputs.

### 6.24.5.1. ORR (predicate, predicate), setting inactive to zero

Instances
svbool_t <b>svorr</b> [_b]_z(svbool_t <i>pg</i> , svbool_t <i>op1</i> , svbool_t <i>op2</i> )

## 6.24.6. ORN: Predicate OR NOT

These functions invert the second predicate input and OR it with the first predicate input.

### 6.24.6.1. ORN (predicate, predicate), setting inactive to zero

Instances
svbool_t <b>svorn</b> [_b]_z(svbool_t <i>pg</i> , svbool_t <i>op1</i> , svbool_t <i>op2</i> )

## 6.24.7. NOR: Predicate NOR

These functions perform an OR of two predicate inputs and invert the result.

### 6.24.7.1. NOR (predicate, predicate), setting inactive to zero

Instances
svbool_t <b>svnor</b> [_b]_z(svbool_t <i>pg</i> , svbool_t <i>op1</i> , svbool_t <i>op2</i> )

## 6.24.8. EOR: Predicate exclusive OR

These functions perform an exclusive OR of two predicate inputs.

### 6.24.8.1. EOR (predicate, predicate), setting inactive to zero

Instances
svbool_t <b>sveor</b> [_b]_z(svbool_t <i>pg</i> , svbool_t <i>op1</i> , svbool_t <i>op2</i> )

## 6.24.9. NOT: Predicate NOT

These functions invert a predicate input.

### 6.24.9.1. NOT (predicate), setting inactive to zero

Instances
svbool_t <b>svnot</b> [_b]_z(svbool_t <i>pg</i> , svbool_t <i>op</i> )

## 6.24.10. BRKA: Break after first true condition

These functions return a predicate in which each active element is true if all previous active elements of the input predicate are false.

### 6.24.10.1. BRKA (predicate), setting inactive to zero

Instances
svbool_t <b>svbrka</b> [_b]_z(svbool_t <i>pg</i> , svbool_t <i>op</i> )

### 6.24.10.2. BRKA (predicate), merging with separate predicate

Instances
svbool_t <b>svbrka</b> [_b]_m(svbool_t <i>inactive</i> , svbool_t <i>pg</i> , svbool_t <i>op</i> )

## 6.24.11. BRKB: Break before first true condition

These functions return a predicate in which each active element is true if the corresponding element of the input predicate, and all previous active elements of the input predicate, are false.

### 6.24.11.1. BRKB (predicate), setting inactive to zero

Instances
svbool_t <b>svbrkb</b> [_b]_z(svbool_t <i>pg</i> , svbool_t <i>op</i> )

### 6.24.11.2. BRKB (predicate), merging with separate predicate

Instances
svbool_t <b>svbrkb</b> [_b]_m(svbool_t <i>inactive</i> , svbool_t <i>pg</i> , svbool_t <i>op</i> )

## 6.24.12. BRKN: Propagate break to next partition

These functions return the second predicate input if the last active element of the first predicate input is true, otherwise they return an all-false predicate.

### 6.24.12.1. BRKN (predicate, predicate), setting inactive to zero

Instances
<code>svbool_t svbrkn[_b]_z(svbool_t pg, svbool_t op1, svbool_t op2)</code>

## 6.24.13. BRKPA: Propagate and break after first true condition

If the last active element of the first predicate input is false, these functions return an all-false predicate. Otherwise they return a predicate in which each active element is true if all previous active elements of the second input predicate are false. Inactive elements of the result are always false.

### 6.24.13.1. BRKPA (predicate, predicate), setting inactive to zero

Instances
<code>svbool_t svbrkpa[_b]_z(svbool_t pg, svbool_t op1, svbool_t op2)</code>

## 6.24.14. BRKPB: Propagate and break before first true condition

If the last active element of the first predicate input is false, these functions return an all-false predicate. Otherwise they return a predicate in which each active element is true if the corresponding element of the second input predicate, and all previous active elements of the second input predicate, are false. Inactive elements of the result are always false.

### 6.24.14.1. BRKPB (predicate, predicate), setting inactive to zero

Instances
<code>svbool_t svbrkpb[_b]_z(svbool_t pg, svbool_t op1, svbool_t op2)</code>

## 6.24.15. PFIRST: Set first active predicate element to true

These functions return a copy of the input predicate in which the first active element is set to true.

### 6.24.15.1. PFIRST (predicate)

Instances
<code>svbool_t svpfirst[_b](svbool_t pg, svbool_t op)</code>

## 6.24.16. PNEXT: Set next active predicate element to true

These functions find the first active element for which that element and all subsequent active and inactive elements of the predicate input are false. If such an element exists, the functions return a predicate in which only that element is true, otherwise they return an all-false predicate.

### 6.24.16.1. PNEXT (predicate)

Instances
<code>svbool_t svpnex_b8(svbool_t pg, svbool_t op)</code>



**Instances**

```
svbool_t svpnex_b16(svbool_t pg, svbool_t op)
svbool_t svpnex_b32(svbool_t pg, svbool_t op)
svbool_t svpnex_b64(svbool_t pg, svbool_t op)
```

## 6.25. Testing predicates

### 6.25.1. PTEST: Test active elements

These functions test which active elements of the input predicate are true and return a boolean based on the result. There are three conditions:

any      Return true if any active element is true.  
first  
last      Return true if the first active element is true.  
  
Return true if the last active element is true.

The compiler may be able to avoid an explicit PTEST instruction by reusing flags from a previous instruction.

#### 6.25.1.1. PTEST (predicate)

**Instances**

```
bool svptest_any(svbool_t pg, svbool_t op)
bool svptest_first(svbool_t pg, svbool_t op)
bool svptest_last(svbool_t pg, svbool_t op)
```

## 6.26. FFR manipulation

### 6.26.1. RDFFR: Read the first-fault register

This function returns the contents of the first-fault register (FFR). Conceptually, it also updates the first-fault register token (FFRT) and starts a new FFR group. See [Section 4.7, “First-faulting and non-faulting loads”](#) for a description of FFR groups.

#### 6.26.1.1. RDFFR (inherent)

**Instances**

```
svbool_t svrdffr()
```

#### 6.26.1.2. RDFFR, setting inactive to zero

**Instances**

```
svbool_t svrdffr_z(svbool_t pg)
```

### 6.26.2. SETFFR: Set the first-fault register

This function sets the first-fault register (FFR) to all-true. Conceptually, it also updates the first-fault register token (FFRT) and starts a new FFR group. See [Section 4.7, “First-faulting and non-faulting loads”](#) for a description of FFR groups.

### 6.26.2.1. SETFFR (inherent)

Instances
<code>void svsetffr()</code>

### 6.26.3. WRFFR: Write to the first-fault register

This function sets the first-fault register (FFR) to the given value. Conceptually, it also updates the first-fault register token (FFRT) and starts a new FFR group. See [Section 4.7, “First-faulting and non-faulting loads”](#) a description of FFR groups.

#### 6.26.3.1. WRFFR (predicate)

Instances
<code>void svwrffr(svbool_t op)</code>

## 6.27. Counting elements

### 6.27.1. CNTP: Count active elements

These functions count the number of active elements in a predicate input. A numerical suffix indicates the size of data that the predicate controls.

#### 6.27.1.1. CNTP (predicate)

Instances
<code>uint64_t svcntp_b8(svbool_t pg, svbool_t op)</code>
<code>uint64_t svcntp_b16(svbool_t pg, svbool_t op)</code>
<code>uint64_t svcntp_b32(svbool_t pg, svbool_t op)</code>
<code>uint64_t svcntp_b64(svbool_t pg, svbool_t op)</code>

### 6.27.2. CNTB: Count the number of 8-bit elements in a pattern

There are two forms: one with a `_pat` suffix that takes an explicit vector pattern and one without a `_pat` suffix in which the pattern is implicitly `SV_ALL`.

#### 6.27.2.1. CNTB (inherent)

Instances
<code>uint64_t svcntb()</code>

#### 6.27.2.2. CNTB (pattern)

Instances
<code>uint64_t svcntb_pat(enum svpattern pattern)</code>

### 6.27.3. CNTH: Count the number of 16-bit elements in a pattern

There are two forms: one with a `_pat` suffix that takes an explicit vector pattern and one without a `_pat` suffix in which the pattern is implicitly `SV_ALL`.

### 6.27.3.1. CNTH (inherent)

Instances
<code>uint64_t svcnth()</code>

### 6.27.3.2. CNTH (pattern)

Instances
<code>uint64_t svcnth_pat(enum svpattern pattern)</code>

## 6.27.4. CNTW: Count the number of 32-bit elements in a pattern

There are two forms: one with a `_pat` suffix that takes an explicit vector pattern and one without a `_pat` suffix in which the pattern is implicitly `SV_ALL`.

### 6.27.4.1. CNTW (inherent)

Instances
<code>uint64_t svcntw()</code>

### 6.27.4.2. CNTW (pattern)

Instances
<code>uint64_t svcntw_pat(enum svpattern pattern)</code>

## 6.27.5. CNTD: Count the number of 64-bit elements in a pattern

There are two forms: one with a `_pat` suffix that takes an explicit vector pattern and one without a `_pat` suffix in which the pattern is implicitly `SV_ALL`.

### 6.27.5.1. CNTD (inherent)

Instances
<code>uint64_t svcntd()</code>

### 6.27.5.2. CNTD (pattern)

Instances
<code>uint64_t svcntd_pat(enum svpattern pattern)</code>

## 6.27.6. LEN: Return the number of elements in a vector

These functions return the number of elements in a full vector. The only purpose of the input is to specify a type; its contents do not matter. (However, unlike `sizeof`, the input is still evaluated.)

In practice only the overloaded versions are useful.

### 6.27.6.1. LEN (vector)

Instances
<code>uint64_t svlen[_s8](svint8_t op)</code>

**Instances**

```

uint64_t svlen[_s16](svint16_t op)
uint64_t svlen[_s32](svint32_t op)
uint64_t svlen[_s64](svint64_t op)
uint64_t svlen[_u8](svuint8_t op)
uint64_t svlen[_u16](svuint16_t op)
uint64_t svlen[_u32](svuint32_t op)
uint64_t svlen[_u64](svuint64_t op)
uint64_t svlen[_f16](svfloat16_t op)
uint64_t svlen[_f32](svfloat32_t op)
uint64_t svlen[_f64](svfloat64_t op)
uint64_t svlen[_bf16](svbfloat16_t op)

```

## 6.28. Saturating scalar arithmetic

### 6.28.1. QINCB: Saturating increment by a multiple of svcntb

These functions calculate the number of vectors in a pattern, as for `svcntb`, multiply the result by an immediate factor, and then add the result to the first input. The addition is saturating, so if the sum is outside the range of the return type, the result is the nearest in-range value.

There are two forms: one with a `_pat` suffix that takes an explicit vector pattern and one without a `_pat` suffix in which the pattern is implicitly `SV_ALL`.

The immediate multiplication factor must be an integer constant expression in the range [1,16].

#### 6.28.1.1. QINCB (scalar, multiplier)

**Instances**

```

int32_t svqincb[_n_s32](int32_t op, uint64_t imm_factor)
int64_t svqincb[_n_s64](int64_t op, uint64_t imm_factor)
uint32_t svqincb[_n_u32](uint32_t op, uint64_t imm_factor)
uint64_t svqincb[_n_u64](uint64_t op, uint64_t imm_factor)

```

#### 6.28.1.2. QINCB (scalar, pattern, multiplier)

**Instances**

```

int32_t svqincb_pat[_n_s32](int32_t op, enum svpattern pattern,
                             uint64_t imm_factor)
int64_t svqincb_pat[_n_s64](int64_t op, enum svpattern pattern,
                             uint64_t imm_factor)
uint32_t svqincb_pat[_n_u32](uint32_t op, enum svpattern pattern,
                             uint64_t imm_factor)
uint64_t svqincb_pat[_n_u64](uint64_t op, enum svpattern pattern,
                             uint64_t imm_factor)

```

### 6.28.2. QINCH: Saturating increment by a multiple of svcnth

These functions calculate the number of vectors in a pattern, as for `svcnth`, multiply the result by an immediate factor, and then add the result to the first input. The addition is saturating, so if the sum is outside the range of the return type, the result is the nearest in-range value.

There are two forms: one with a `_pat` suffix that takes an explicit vector pattern and one without a `_pat` suffix in which the pattern is implicitly `SV_ALL`.

The immediate multiplication factor must be an integer constant expression in the range [1,16].

### 6.28.2.1. QINCH (scalar, multiplier)

Instances
<code>int32_t svqinch[_n_s32](int32_t op, uint64_t imm_factor)</code>
<code>int64_t svqinch[_n_s64](int64_t op, uint64_t imm_factor)</code>
<code>uint32_t svqinch[_n_u32](uint32_t op, uint64_t imm_factor)</code>
<code>uint64_t svqinch[_n_u64](uint64_t op, uint64_t imm_factor)</code>

### 6.28.2.2. QINCH (scalar, pattern, multiplier)

Instances
<code>int32_t svqinch_pat[_n_s32](int32_t op, enum svpattern pattern, uint64_t imm_factor)</code>
<code>int64_t svqinch_pat[_n_s64](int64_t op, enum svpattern pattern, uint64_t imm_factor)</code>
<code>uint32_t svqinch_pat[_n_u32](uint32_t op, enum svpattern pattern, uint64_t imm_factor)</code>
<code>uint64_t svqinch_pat[_n_u64](uint64_t op, enum svpattern pattern, uint64_t imm_factor)</code>

### 6.28.2.3. QINCH (vector, multiplier)

Instances
<code>svint16_t svqinch[_s16](svint16_t op, uint64_t imm_factor)</code>
<code>svuint16_t svqinch[_u16](svuint16_t op, uint64_t imm_factor)</code>

### 6.28.2.4. QINCH (vector, pattern, multiplier)

Instances
<code>svint16_t svqinch_pat[_s16](svint16_t op, enum svpattern pattern, uint64_t imm_factor)</code>
<code>svuint16_t svqinch_pat[_u16](svuint16_t op, enum svpattern pattern, uint64_t imm_factor)</code>

## 6.28.3. QINCW: Saturating increment by a multiple of svcntw

These functions calculate the number of vectors in a pattern, as for `svcntw`, multiply the result by an immediate factor, and then add the result to the first input. The addition is saturating, so if the sum is outside the range of the return type, the result is the nearest in-range value.

There are two forms: one with a `_pat` suffix that takes an explicit vector pattern and one without a `_pat` suffix in which the pattern is implicitly `SV_ALL`.

The immediate multiplication factor must be an integer constant expression in the range [1,16].

### 6.28.3.1. QINCW (scalar, multiplier)

Instances
<code>int32_t svqincw[_n_s32](int32_t op, uint64_t imm_factor)</code>
<code>int64_t svqincw[_n_s64](int64_t op, uint64_t imm_factor)</code>
<code>uint32_t svqincw[_n_u32](uint32_t op, uint64_t imm_factor)</code>

Instances
<code>uint64_t svqincw[_n_u64](uint64_t op, uint64_t imm_factor)</code>

### 6.28.3.2. QINCW (scalar, pattern, multiplier)

Instances
<code>int32_t svqincw_pat[_n_s32](int32_t op, enum svpattern pattern, uint64_t imm_factor)</code>
<code>int64_t svqincw_pat[_n_s64](int64_t op, enum svpattern pattern, uint64_t imm_factor)</code>
<code>uint32_t svqincw_pat[_n_u32](uint32_t op, enum svpattern pattern, uint64_t imm_factor)</code>
<code>uint64_t svqincw_pat[_n_u64](uint64_t op, enum svpattern pattern, uint64_t imm_factor)</code>

### 6.28.3.3. QINCW (vector, multiplier)

Instances
<code>svint32_t svqincw[_s32](svint32_t op, uint64_t imm_factor)</code>
<code>svuint32_t svqincw[_u32](svuint32_t op, uint64_t imm_factor)</code>

### 6.28.3.4. QINCW (vector, pattern, multiplier)

Instances
<code>svint32_t svqincw_pat[_s32](svint32_t op, enum svpattern pattern, uint64_t imm_factor)</code>
<code>svuint32_t svqincw_pat[_u32](svuint32_t op, enum svpattern pattern, uint64_t imm_factor)</code>

## 6.28.4. QINCD: Saturating increment by a multiple of svcntd

These functions calculate the number of vectors in a pattern, as for `svcntd`, multiply the result by an immediate factor, and then add the result to the first input. The addition is saturating, so if the sum is outside the range of the return type, the result is the nearest in-range value.

There are two forms: one with a `_pat` suffix that takes an explicit vector pattern and one without a `_pat` suffix in which the pattern is implicitly `SV_ALL`.

The immediate multiplication factor must be an integer constant expression in the range [1,16].

### 6.28.4.1. QINCD (scalar, multiplier)

Instances
<code>int32_t svqincd[_n_s32](int32_t op, uint64_t imm_factor)</code>
<code>int64_t svqincd[_n_s64](int64_t op, uint64_t imm_factor)</code>
<code>uint32_t svqincd[_n_u32](uint32_t op, uint64_t imm_factor)</code>
<code>uint64_t svqincd[_n_u64](uint64_t op, uint64_t imm_factor)</code>

### 6.28.4.2. QINCD (scalar, pattern, multiplier)

Instances
<code>int32_t svqincd_pat[_n_s32](int32_t op, enum svpattern pattern,</code>

**Instances**

```

uint64_t imm_factor)
int64_t svqincd_pat[_n_s64](int64_t op, enum svpattern pattern,
uint64_t imm_factor)
uint32_t svqincd_pat[_n_u32](uint32_t op, enum svpattern pattern,
uint64_t imm_factor)
uint64_t svqincd_pat[_n_u64](uint64_t op, enum svpattern pattern,
uint64_t imm_factor)

```

**6.28.4.3. QINCD (vector, multiplier)****Instances**

```

svint64_t svqincd[_s64](svint64_t op, uint64_t imm_factor)
svuint64_t svqincd[_u64](svuint64_t op, uint64_t imm_factor)

```

**6.28.4.4. QINCD (vector, pattern, multiplier)****Instances**

```

svint64_t svqincd_pat[_s64](svint64_t op, enum svpattern pattern,
uint64_t imm_factor)
svuint64_t svqincd_pat[_u64](svuint64_t op, enum svpattern pattern,
uint64_t imm_factor)

```

**6.28.5. QINCP: Saturating increment by a multiple of svcntp**

These functions pass the second input to `svcntp` and add the result to the first input. The addition is saturating, so if the sum is outside the range of the return type, the result is the nearest in-range value.

**6.28.5.1. QINCP (scalar)****Instances**

```

int32_t svqincp[_n_s32]_b8(int32_t op, svbool_t pg)
int32_t svqincp[_n_s32]_b16(int32_t op, svbool_t pg)
int32_t svqincp[_n_s32]_b32(int32_t op, svbool_t pg)
int32_t svqincp[_n_s32]_b64(int32_t op, svbool_t pg)
int64_t svqincp[_n_s64]_b8(int64_t op, svbool_t pg)
int64_t svqincp[_n_s64]_b16(int64_t op, svbool_t pg)
int64_t svqincp[_n_s64]_b32(int64_t op, svbool_t pg)
int64_t svqincp[_n_s64]_b64(int64_t op, svbool_t pg)
uint32_t svqincp[_n_u32]_b8(uint32_t op, svbool_t pg)
uint32_t svqincp[_n_u32]_b16(uint32_t op, svbool_t pg)
uint32_t svqincp[_n_u32]_b32(uint32_t op, svbool_t pg)
uint32_t svqincp[_n_u32]_b64(uint32_t op, svbool_t pg)
uint64_t svqincp[_n_u64]_b8(uint64_t op, svbool_t pg)
uint64_t svqincp[_n_u64]_b16(uint64_t op, svbool_t pg)
uint64_t svqincp[_n_u64]_b32(uint64_t op, svbool_t pg)
uint64_t svqincp[_n_u64]_b64(uint64_t op, svbool_t pg)

```

**6.28.5.2. QINCP (vector)****Instances**

```

svint16_t svqincp[_s16](svint16_t op, svbool_t pg)
svint32_t svqincp[_s32](svint32_t op, svbool_t pg)
svint64_t svqincp[_s64](svint64_t op, svbool_t pg)

```

**Instances**

```
svuint16_t svqincp[_u16](svuint16_t op, svbool_t pg)
svuint32_t svqincp[_u32](svuint32_t op, svbool_t pg)
svuint64_t svqincp[_u64](svuint64_t op, svbool_t pg)
```

**6.28.6. QDECB: Saturating decrement by a multiple of `svcntb`**

These functions calculate the number of vectors in a pattern, as for `svcntb`, multiply the result by an immediate factor, and then subtract the result from the first input. The subtraction is saturating, so if the difference is outside the range of the return type, the result is the nearest in-range value.

There are two forms: one with a `_pat` suffix that takes an explicit vector pattern and one without a `_pat` suffix in which the pattern is implicitly `SV_ALL`.

The immediate multiplication factor must be an integer constant expression in the range [1,16].

**6.28.6.1. QDECB (scalar, multiplier)****Instances**

```
int32_t svqdec[_n_s32](int32_t op, uint64_t imm_factor)
int64_t svqdec[_n_s64](int64_t op, uint64_t imm_factor)
uint32_t svqdec[_n_u32](uint32_t op, uint64_t imm_factor)
uint64_t svqdec[_n_u64](uint64_t op, uint64_t imm_factor)
```

**6.28.6.2. QDECB (scalar, pattern, multiplier)****Instances**

```
int32_t svqdec_pat[_n_s32](int32_t op, enum svpattern pattern,
                           uint64_t imm_factor)
int64_t svqdec_pat[_n_s64](int64_t op, enum svpattern pattern,
                           uint64_t imm_factor)
uint32_t svqdec_pat[_n_u32](uint32_t op, enum svpattern pattern,
                           uint64_t imm_factor)
uint64_t svqdec_pat[_n_u64](uint64_t op, enum svpattern pattern,
                           uint64_t imm_factor)
```

**6.28.7. QDECH: Saturating decrement by a multiple of `svcnth`**

These functions calculate the number of vectors in a pattern, as for `svcnth`, multiply the result by an immediate factor, and then subtract the result from the first input. The subtraction is saturating, so if the difference is outside the range of the return type, the result is the nearest in-range value.

There are two forms: one with a `_pat` suffix that takes an explicit vector pattern and one without a `_pat` suffix in which the pattern is implicitly `SV_ALL`.

The immediate multiplication factor must be an integer constant expression in the range [1,16].

**6.28.7.1. QDECH (scalar, multiplier)****Instances**

```
int32_t svqdech[_n_s32](int32_t op, uint64_t imm_factor)
int64_t svqdech[_n_s64](int64_t op, uint64_t imm_factor)
uint32_t svqdech[_n_u32](uint32_t op, uint64_t imm_factor)
uint64_t svqdech[_n_u64](uint64_t op, uint64_t imm_factor)
```



### 6.28.7.2. QDECH (scalar, pattern, multiplier)

Instances
<code>int32_t svqdech_pat[_n_s32](int32_t op, enum svpattern pattern, uint64_t imm_factor)</code>
<code>int64_t svqdech_pat[_n_s64](int64_t op, enum svpattern pattern, uint64_t imm_factor)</code>
<code>uint32_t svqdech_pat[_n_u32](uint32_t op, enum svpattern pattern, uint64_t imm_factor)</code>
<code>uint64_t svqdech_pat[_n_u64](uint64_t op, enum svpattern pattern, uint64_t imm_factor)</code>

### 6.28.7.3. QDECH (vector, multiplier)

Instances
<code>svint16_t svqdech[_s16](svint16_t op, uint64_t imm_factor)</code>
<code>svuint16_t svqdech[_u16](svuint16_t op, uint64_t imm_factor)</code>

### 6.28.7.4. QDECH (vector, pattern, multiplier)

Instances
<code>svint16_t svqdech_pat[_s16](svint16_t op, enum svpattern pattern, uint64_t imm_factor)</code>
<code>svuint16_t svqdech_pat[_u16](svuint16_t op, enum svpattern pattern, uint64_t imm_factor)</code>

## 6.28.8. QDECW: Saturating decrement by a multiple of svcntw

These functions calculate the number of vectors in a pattern, as for `svcntw`, multiply the result by an immediate factor, and then subtract the result from the first input. The subtraction is saturating, so if the difference is outside the range of the return type, the result is the nearest in-range value.

There are two forms: one with a `_pat` suffix that takes an explicit vector pattern and one without a `_pat` suffix in which the pattern is implicitly `SV_ALL`.

The immediate multiplication factor must be an integer constant expression in the range [1,16].

### 6.28.8.1. QDECW (scalar, multiplier)

Instances
<code>int32_t svqdecw[_n_s32](int32_t op, uint64_t imm_factor)</code>
<code>int64_t svqdecw[_n_s64](int64_t op, uint64_t imm_factor)</code>
<code>uint32_t svqdecw[_n_u32](uint32_t op, uint64_t imm_factor)</code>
<code>uint64_t svqdecw[_n_u64](uint64_t op, uint64_t imm_factor)</code>

### 6.28.8.2. QDECW (scalar, pattern, multiplier)

Instances
<code>int32_t svqdecw_pat[_n_s32](int32_t op, enum svpattern pattern, uint64_t imm_factor)</code>
<code>int64_t svqdecw_pat[_n_s64](int64_t op, enum svpattern pattern, uint64_t imm_factor)</code>
<code>uint32_t svqdecw_pat[_n_u32](uint32_t op, enum svpattern pattern, uint64_t imm_factor)</code>

**Instances**

```
uint64_t svqdecw_pat[_n_u64](uint64_t op, enum svpattern pattern,
                             uint64_t imm_factor)
```

**6.28.8.3. QDECW (vector, multiplier)****Instances**

```
svint32_t svqdecw[_s32](svint32_t op, uint64_t imm_factor)
svuint32_t svqdecw[_u32](svuint32_t op, uint64_t imm_factor)
```

**6.28.8.4. QDECW (vector, pattern, multiplier)****Instances**

```
svint32_t svqdecw_pat[_s32](svint32_t op, enum svpattern pattern,
                             uint64_t imm_factor)
svuint32_t svqdecw_pat[_u32](svuint32_t op, enum svpattern pattern,
                             uint64_t imm_factor)
```

**6.28.9. QDECD: Saturating decrement by a multiple of svcntd**

These functions calculate the number of vectors in a pattern, as for `svcntd`, multiply the result by an immediate factor, and then subtract the result from the first input. The subtraction is saturating, so if the difference is outside the range of the return type, the result is the nearest in-range value.

There are two forms: one with a `_pat` suffix that takes an explicit vector pattern and one without a `_pat` suffix in which the pattern is implicitly `SV_ALL`.

The immediate multiplication factor must be an integer constant expression in the range [1,16].

**6.28.9.1. QDECD (scalar, multiplier)****Instances**

```
int32_t svqdecd[_n_s32](int32_t op, uint64_t imm_factor)
int64_t svqdecd[_n_s64](int64_t op, uint64_t imm_factor)
uint32_t svqdecd[_n_u32](uint32_t op, uint64_t imm_factor)
uint64_t svqdecd[_n_u64](uint64_t op, uint64_t imm_factor)
```

**6.28.9.2. QDECD (scalar, pattern, multiplier)****Instances**

```
int32_t svqdecd_pat[_n_s32](int32_t op, enum svpattern pattern,
                             uint64_t imm_factor)
int64_t svqdecd_pat[_n_s64](int64_t op, enum svpattern pattern,
                             uint64_t imm_factor)
uint32_t svqdecd_pat[_n_u32](uint32_t op, enum svpattern pattern,
                             uint64_t imm_factor)
uint64_t svqdecd_pat[_n_u64](uint64_t op, enum svpattern pattern,
                             uint64_t imm_factor)
```

**6.28.9.3. QDECD (vector, multiplier)****Instances**

```
svint64_t svqdecd[_s64](svint64_t op, uint64_t imm_factor)
svuint64_t svqdecd[_u64](svuint64_t op, uint64_t imm_factor)
```

#### 6.28.9.4. QDECD (vector, pattern, multiplier)

##### Instances

```
svint64_t svqdec_d_pat[_s64](svint64_t op, enum svpattern pattern,
                             uint64_t imm_factor)
svuint64_t svqdec_d_pat[_u64](svuint64_t op, enum svpattern pattern,
                              uint64_t imm_factor)
```

### 6.28.10. QDECP: Saturating decrement by a multiple of `svcntp`

These functions pass the second input to `svcntp` and subtract the result from the first input. The subtraction is saturating, so if the difference is outside the range of the return type, the result is the nearest in-range value.

#### 6.28.10.1. QDECP (scalar)

##### Instances

```
int32_t svqdec_p[_n_s32]_b8(int32_t op, svbool_t pg)
int32_t svqdec_p[_n_s32]_b16(int32_t op, svbool_t pg)
int32_t svqdec_p[_n_s32]_b32(int32_t op, svbool_t pg)
int32_t svqdec_p[_n_s32]_b64(int32_t op, svbool_t pg)
int64_t svqdec_p[_n_s64]_b8(int64_t op, svbool_t pg)
int64_t svqdec_p[_n_s64]_b16(int64_t op, svbool_t pg)
int64_t svqdec_p[_n_s64]_b32(int64_t op, svbool_t pg)
int64_t svqdec_p[_n_s64]_b64(int64_t op, svbool_t pg)
uint32_t svqdec_p[_n_u32]_b8(uint32_t op, svbool_t pg)
uint32_t svqdec_p[_n_u32]_b16(uint32_t op, svbool_t pg)
uint32_t svqdec_p[_n_u32]_b32(uint32_t op, svbool_t pg)
uint32_t svqdec_p[_n_u32]_b64(uint32_t op, svbool_t pg)
uint64_t svqdec_p[_n_u64]_b8(uint64_t op, svbool_t pg)
uint64_t svqdec_p[_n_u64]_b16(uint64_t op, svbool_t pg)
uint64_t svqdec_p[_n_u64]_b32(uint64_t op, svbool_t pg)
uint64_t svqdec_p[_n_u64]_b64(uint64_t op, svbool_t pg)
```

#### 6.28.10.2. QDECP (vector)

##### Instances

```
svint16_t svqdec_p[_s16](svint16_t op, svbool_t pg)
svint32_t svqdec_p[_s32](svint32_t op, svbool_t pg)
svint64_t svqdec_p[_s64](svint64_t op, svbool_t pg)
svuint16_t svqdec_p[_u16](svuint16_t op, svbool_t pg)
svuint32_t svqdec_p[_u32](svuint32_t op, svbool_t pg)
svuint64_t svqdec_p[_u64](svuint64_t op, svbool_t pg)
```

## 6.29. Reinterpreting data

### 6.29.1. REINTERPRET: Reinterpret vector contents

These functions reinterpret the contents of a vector as a different type, without changing any bits. As [Section 3.4, “Vector types”](#) explains, such conversions need to be explicit function calls rather than C-style casts.

There is one function for every possible pair of vector types, including functions that “reinterpret” each type as itself.

### 6.29.1.1. REINTERPRET (vector)

#### Instances

```

svint8_t  svreinterpret_s8[_s8](svint8_t op)
svint8_t  svreinterpret_s8[_s16](svint16_t op)
svint8_t  svreinterpret_s8[_s32](svint32_t op)
svint8_t  svreinterpret_s8[_s64](svint64_t op)
svint8_t  svreinterpret_s8[_u8](svuint8_t op)
svint8_t  svreinterpret_s8[_u16](svuint16_t op)
svint8_t  svreinterpret_s8[_u32](svuint32_t op)
svint8_t  svreinterpret_s8[_u64](svuint64_t op)
svint8_t  svreinterpret_s8[_f16](svfloat16_t op)
svint8_t  svreinterpret_s8[_f32](svfloat32_t op)
svint8_t  svreinterpret_s8[_f64](svfloat64_t op)
svint8_t  svreinterpret_s8[_bf16](svbfloat16_t op)
svint16_t svreinterpret_s16[_s8](svint8_t op)
svint16_t svreinterpret_s16[_s16](svint16_t op)
svint16_t svreinterpret_s16[_s32](svint32_t op)
svint16_t svreinterpret_s16[_s64](svint64_t op)
svint16_t svreinterpret_s16[_u8](svuint8_t op)
svint16_t svreinterpret_s16[_u16](svuint16_t op)
svint16_t svreinterpret_s16[_u32](svuint32_t op)
svint16_t svreinterpret_s16[_u64](svuint64_t op)
svint16_t svreinterpret_s16[_f16](svfloat16_t op)
svint16_t svreinterpret_s16[_f32](svfloat32_t op)
svint16_t svreinterpret_s16[_f64](svfloat64_t op)
svint16_t svreinterpret_s16[_bf16](svbfloat16_t op)
svint32_t svreinterpret_s32[_s8](svint8_t op)
svint32_t svreinterpret_s32[_s16](svint16_t op)
svint32_t svreinterpret_s32[_s32](svint32_t op)
svint32_t svreinterpret_s32[_s64](svint64_t op)
svint32_t svreinterpret_s32[_u8](svuint8_t op)
svint32_t svreinterpret_s32[_u16](svuint16_t op)
svint32_t svreinterpret_s32[_u32](svuint32_t op)
svint32_t svreinterpret_s32[_u64](svuint64_t op)
svint32_t svreinterpret_s32[_f16](svfloat16_t op)
svint32_t svreinterpret_s32[_f32](svfloat32_t op)
svint32_t svreinterpret_s32[_f64](svfloat64_t op)
svint32_t svreinterpret_s32[_bf16](svbfloat16_t op)
svint64_t svreinterpret_s64[_s8](svint8_t op)
svint64_t svreinterpret_s64[_s16](svint16_t op)
svint64_t svreinterpret_s64[_s32](svint32_t op)
svint64_t svreinterpret_s64[_s64](svint64_t op)
svint64_t svreinterpret_s64[_u8](svuint8_t op)
svint64_t svreinterpret_s64[_u16](svuint16_t op)
svint64_t svreinterpret_s64[_u32](svuint32_t op)
svint64_t svreinterpret_s64[_u64](svuint64_t op)
svint64_t svreinterpret_s64[_f16](svfloat16_t op)
svint64_t svreinterpret_s64[_f32](svfloat32_t op)
svint64_t svreinterpret_s64[_f64](svfloat64_t op)
svint64_t svreinterpret_s64[_bf16](svbfloat16_t op)
svuint8_t svreinterpret_u8[_s8](svint8_t op)
svuint8_t svreinterpret_u8[_s16](svint16_t op)
svuint8_t svreinterpret_u8[_s32](svint32_t op)
svuint8_t svreinterpret_u8[_s64](svint64_t op)
svuint8_t svreinterpret_u8[_u8](svuint8_t op)
svuint8_t svreinterpret_u8[_u16](svuint16_t op)
svuint8_t svreinterpret_u8[_u32](svuint32_t op)
svuint8_t svreinterpret_u8[_u64](svuint64_t op)

```

**Instances**

```

svuint8_t svreinterpret_u8[_f16](svfloat16_t op)
svuint8_t svreinterpret_u8[_f32](svfloat32_t op)
svuint8_t svreinterpret_u8[_f64](svfloat64_t op)
svuint8_t svreinterpret_u8[_bf16](svbfloat16_t op)
svuint16_t svreinterpret_u16[_s8](svint8_t op)
svuint16_t svreinterpret_u16[_s16](svint16_t op)
svuint16_t svreinterpret_u16[_s32](svint32_t op)
svuint16_t svreinterpret_u16[_s64](svint64_t op)
svuint16_t svreinterpret_u16[_u8](svuint8_t op)
svuint16_t svreinterpret_u16[_u16](svuint16_t op)
svuint16_t svreinterpret_u16[_u32](svuint32_t op)
svuint16_t svreinterpret_u16[_u64](svuint64_t op)
svuint16_t svreinterpret_u16[_f16](svfloat16_t op)
svuint16_t svreinterpret_u16[_f32](svfloat32_t op)
svuint16_t svreinterpret_u16[_f64](svfloat64_t op)
svuint16_t svreinterpret_u16[_bf16](svbfloat16_t op)
svuint32_t svreinterpret_u32[_s8](svint8_t op)
svuint32_t svreinterpret_u32[_s16](svint16_t op)
svuint32_t svreinterpret_u32[_s32](svint32_t op)
svuint32_t svreinterpret_u32[_s64](svint64_t op)
svuint32_t svreinterpret_u32[_u8](svuint8_t op)
svuint32_t svreinterpret_u32[_u16](svuint16_t op)
svuint32_t svreinterpret_u32[_u32](svuint32_t op)
svuint32_t svreinterpret_u32[_u64](svuint64_t op)
svuint32_t svreinterpret_u32[_f16](svfloat16_t op)
svuint32_t svreinterpret_u32[_f32](svfloat32_t op)
svuint32_t svreinterpret_u32[_f64](svfloat64_t op)
svuint32_t svreinterpret_u32[_bf16](svbfloat16_t op)
svuint64_t svreinterpret_u64[_s8](svint8_t op)
svuint64_t svreinterpret_u64[_s16](svint16_t op)
svuint64_t svreinterpret_u64[_s32](svint32_t op)
svuint64_t svreinterpret_u64[_s64](svint64_t op)
svuint64_t svreinterpret_u64[_u8](svuint8_t op)
svuint64_t svreinterpret_u64[_u16](svuint16_t op)
svuint64_t svreinterpret_u64[_u32](svuint32_t op)
svuint64_t svreinterpret_u64[_u64](svuint64_t op)
svuint64_t svreinterpret_u64[_f16](svfloat16_t op)
svuint64_t svreinterpret_u64[_f32](svfloat32_t op)
svuint64_t svreinterpret_u64[_f64](svfloat64_t op)
svuint64_t svreinterpret_u64[_bf16](svbfloat16_t op)
svfloat16_t svreinterpret_f16[_s8](svint8_t op)
svfloat16_t svreinterpret_f16[_s16](svint16_t op)
svfloat16_t svreinterpret_f16[_s32](svint32_t op)
svfloat16_t svreinterpret_f16[_s64](svint64_t op)
svfloat16_t svreinterpret_f16[_u8](svuint8_t op)
svfloat16_t svreinterpret_f16[_u16](svuint16_t op)
svfloat16_t svreinterpret_f16[_u32](svuint32_t op)
svfloat16_t svreinterpret_f16[_u64](svuint64_t op)
svfloat16_t svreinterpret_f16[_f16](svfloat16_t op)
svfloat16_t svreinterpret_f16[_f32](svfloat32_t op)
svfloat16_t svreinterpret_f16[_f64](svfloat64_t op)
svfloat16_t svreinterpret_f16[_bf16](svbfloat16_t op)
svfloat32_t svreinterpret_f32[_s8](svint8_t op)
svfloat32_t svreinterpret_f32[_s16](svint16_t op)
svfloat32_t svreinterpret_f32[_s32](svint32_t op)
svfloat32_t svreinterpret_f32[_s64](svint64_t op)
svfloat32_t svreinterpret_f32[_u8](svuint8_t op)
svfloat32_t svreinterpret_f32[_u16](svuint16_t op)

```

**Instances**

```

svfloat32_t svreinterpret_f32[_u32](svuint32_t op)
svfloat32_t svreinterpret_f32[_u64](svuint64_t op)
svfloat32_t svreinterpret_f32[_f16](svfloat16_t op)
svfloat32_t svreinterpret_f32[_f32](svfloat32_t op)
svfloat32_t svreinterpret_f32[_f64](svfloat64_t op)
svfloat32_t svreinterpret_f32[_bf16](svbfloat16_t op)
svfloat64_t svreinterpret_f64[_s8](svint8_t op)
svfloat64_t svreinterpret_f64[_s16](svint16_t op)
svfloat64_t svreinterpret_f64[_s32](svint32_t op)
svfloat64_t svreinterpret_f64[_s64](svint64_t op)
svfloat64_t svreinterpret_f64[_u8](svuint8_t op)
svfloat64_t svreinterpret_f64[_u16](svuint16_t op)
svfloat64_t svreinterpret_f64[_u32](svuint32_t op)
svfloat64_t svreinterpret_f64[_u64](svuint64_t op)
svfloat64_t svreinterpret_f64[_f16](svfloat16_t op)
svfloat64_t svreinterpret_f64[_f32](svfloat32_t op)
svfloat64_t svreinterpret_f64[_f64](svfloat64_t op)
svfloat64_t svreinterpret_f64[_bf16](svbfloat16_t op)
svbfloat16_t svreinterpret_bf16[_s8](svint8_t op)
svbfloat16_t svreinterpret_bf16[_s16](svint16_t op)
svbfloat16_t svreinterpret_bf16[_s32](svint32_t op)
svbfloat16_t svreinterpret_bf16[_s64](svint64_t op)
svbfloat16_t svreinterpret_bf16[_u8](svuint8_t op)
svbfloat16_t svreinterpret_bf16[_u16](svuint16_t op)
svbfloat16_t svreinterpret_bf16[_u32](svuint32_t op)
svbfloat16_t svreinterpret_bf16[_u64](svuint64_t op)
svbfloat16_t svreinterpret_bf16[_f16](svfloat16_t op)
svbfloat16_t svreinterpret_bf16[_f32](svfloat32_t op)
svbfloat16_t svreinterpret_bf16[_f64](svfloat64_t op)
svbfloat16_t svreinterpret_bf16[_bf16](svbfloat16_t op)

```

## 7. List of optional SVE functions

### 7.1. Introduction

This section contains a list of optional SVE functions, grouped into categories and then subdivided based on the first part of the name (up to the first underscore). Each group has its own feature macro.

### 7.2. BFloat16 extensions

These functions are only available when the feature macro `__ARM_FEATURE_SVE_BF16` is defined.

#### 7.2.1. BFDOT: BFloat16 addition of dot product

These functions map to the BFDOT instruction. They partition the second and third vector inputs into sequences of two elements, calculate the dot product of each sequence of the second input and the corresponding sequence of the third input, and then add each result to the overlapping element of the first vector input.

The `_lane` forms of the functions take one sequence of two elements in each 128-bit quadword of the third input and replicate it to fill the rest of the quadword. The `imm_index` parameter specifies the sequence of two elements within each quadword that the functions should replicate. This index must be an integer constant expression in the range  $[0, 128/N)$ , where  $N$  is the number of bits in each sequence.

The functions ignore the FPCR and do not raise any exceptions.

##### 7.2.1.1. BFDOT (vector, vector, vector)

Instances
<code>svfloat32_t svbfdot[_f32](svfloat32_t op1, svbfloat16_t op2, svbfloat16_t op3)</code>

##### 7.2.1.2. BFDOT (vector, vector, scalar)

Instances
<code>svfloat32_t svbfdot[_n_f32](svfloat32_t op1, svbfloat16_t op2, bfloat16_t op3)</code>

##### 7.2.1.3. BFDOT (vector, vector, vector, lane)

Instances
<code>svfloat32_t svbfdot_lane[_f32](svfloat32_t op1, svbfloat16_t op2, svbfloat16_t op3, uint64_t imm_index)</code>

#### 7.2.2. BFMMLA: Accumulating multiplication of BFloat16 matrices

This function maps to the BFMMLA instruction. It partitions the inputs into 128-bit quadwords, with each quadword of the first input containing a row-by-row  $2 \times 2$  matrix, the second input containing a row-by-row  $2 \times 4$  matrix and the third input containing a column-by-column  $4 \times 2$  matrix. For each quadword the function then multiplies the second input matrix by the third input matrix and adds the result to the first input matrix.

The functions ignore the FPCR and do not raise any exceptions.

### 7.2.2.1. BFMMMLA (vector, vector, vector)

Instances
svfloat32_t <b>svbfmmla</b> [_f32](svfloat32_t op1, svbfloat16_t op2, svbfloat16_t op3)

### 7.2.3. BFMLALB: BFloat16 addition of product long (bottom)

These functions map to the BFMLALB instruction. They extract the even-indexed elements of the second and third vectors, giving the same number of elements as the first. They then widen the second and third inputs to single-precision floats, multiply the results, and add the product to the first input. There is no intermediate rounding step after the multiplication.

The `_lane` forms of the functions take one element in each 128-bit quadword of the third input and replicate it to fill the rest of the quadword. The `imm_index` parameter specifies the element within each quadword that the functions should replicate. This index must be an integer constant expression in the range  $[0, 128/N]$ , where  $N$  is the number of bits in each element.

Signaling NaNs trigger an IEEE Invalid exception but quiet NaNs do not. Multiplying an infinity by zero also triggers an IEEE Invalid exception, as does adding `+Inf` and `-Inf` together.

#### 7.2.3.1. BFMLALB (vector, vector, vector)

Instances
svfloat32_t <b>svbfmlalb</b> [_f32](svfloat32_t op1, svbfloat16_t op2, svbfloat16_t op3)

#### 7.2.3.2. BFMLALB (vector, vector, scalar)

Instances
svfloat32_t <b>svbfmlalb</b> [_n_f32](svfloat32_t op1, svbfloat16_t op2, bfloat16_t op3)

#### 7.2.3.3. BFMLALB (vector, vector, vector, lane)

Instances
svfloat32_t <b>svbfmlalb_lane</b> [_f32](svfloat32_t op1, svbfloat16_t op2, svbfloat16_t op3, uint64_t imm_index)

### 7.2.4. BFMLALT: BFloat16 addition of product long (top)

These functions map to the BFMLALT instruction. They extract the odd-indexed elements of the second and third vectors, giving the same number of elements as the first. They then widen the second and third inputs to single-precision floats, multiply the results, and add the product to the first input. There is no intermediate rounding step after the multiplication.

The `_lane` forms of the functions take one element in each 128-bit quadword of the third input and replicate it to fill the rest of the quadword. The `imm_index` parameter specifies the element within each quadword that the functions should replicate. This index must be an integer constant expression in the range  $[0, 128/N]$ , where  $N$  is the number of bits in each element.

Signaling NaNs trigger an IEEE Invalid exception but quiet NaNs do not. Multiplying an infinity by zero also triggers an IEEE Invalid exception, as does adding `+Inf` and `-Inf` together.



### 7.2.4.1. BFMLALT (vector, vector, vector)

#### Instances

```
svfloat32_t svbfmlalt[_f32](svfloat32_t op1, svbfloat16_t op2,
                             svbfloat16_t op3)
```

### 7.2.4.2. BFMLALT (vector, vector, scalar)

#### Instances

```
svfloat32_t svbfmlalt[_n_f32](svfloat32_t op1, svbfloat16_t op2,
                                bfloat16_t op3)
```

### 7.2.4.3. BFMLALT (vector, vector, vector, lane)

#### Instances

```
svfloat32_t svbfmlalt_lane[_f32](svfloat32_t op1, svbfloat16_t op2,
                                    svbfloat16_t op3, uint64_t imm_index)
```

## 7.2.5. CVT: Convert single-precision floats to BFloat16

These functions map to the BFCVT instruction. They convert single-precision floating-point values to BFloat16 values, rounding according to the current rounding mode.

Signaling NaNs trigger an IEEE Invalid exception but quiet NaNs do not.

### 7.2.5.1. CVT (vector), setting inactive to zero

#### Instances

```
svbfloat16_t svcvb_bf16[_f32]_z(svbool_t pg, svfloat32_t op)
```

### 7.2.5.2. CVT (vector), merging with separate vector

#### Instances

```
svbfloat16_t svcvb_bf16[_f32]_m(svbfloat16_t inactive, svbool_t pg,
                                    svfloat32_t op)
```

### 7.2.5.3. CVT (vector), setting inactive to unknown

#### Instances

```
svbfloat16_t svcvb_bf16[_f32]_x(svbool_t pg, svfloat32_t op)
```

## 7.2.6. CVTNT: Convert single-precision floats to BFloat16 (top)

These functions map to the BFCVTNT instruction. They convert the second floating-point input to BFloat16 values, rounding according to the current rounding mode, and store the converted values in the odd-indexed elements of the result. The even-indexed elements of the first input supply the corresponding elements of the result.

The functions described in [Section 7.2.5, “CVT: Convert single-precision floats to BFloat16”](#) provide the equivalent “bottom” operation.

Signaling NaNs trigger an IEEE Invalid exception but quiet NaNs do not.

### 7.2.6.1. CVTNT (vector), merging with even vector

#### Instances

```
svbfloat16_t svcvtnt_bf16[_f32]_m(svbfloat16_t even, svbool_t pg,
                                     svfloat32_t op)
```

### 7.2.6.2. CVTNT (vector), setting inactive to unknown

#### Instances

```
svbfloat16_t svcvtnt_bf16[_f32]_x(svbfloat16_t even, svbool_t pg,
                                     svfloat32_t op)
```

## 7.3. INT8 matrix multiply extensions

These functions are only available when the feature macro `__ARM_FEATURE_SVE_MATMUL_INT8` is defined.

### 7.3.1. MMLA: Accumulating widening multiplication of integer matrices

These functions map to the SMMLA and UMMLA instructions. They partition the inputs into 128-bit quadwords, with the first input containing a row-by-row 2×2 matrix of 32-bit integers, the second input containing a row-by-row 2×8 matrix of 8-bit integers, and the third input containing a column-by-column 8×2 matrix of 8-bit integers. For each quadword, they multiply the second input matrix by the third input matrix using natural arithmetic and then add the result to the first input using modular arithmetic.

The result is well-defined for all inputs; there is no undefined behavior for signed overflow.

#### 7.3.1.1. MMLA (vector, vector, vector)

#### Instances

```
svint32_t svmmla[_s32](svint32_t op1, svint8_t op2, svint8_t op3)
svuint32_t svmmla[_u32](svuint32_t op1, svuint8_t op2, svuint8_t op3)
```

### 7.3.2. USMMLA: Accumulating widening multiplication of integer matrices (unsigned×signed)

This function maps to the USMMLA instruction. It behaves in the same way as the functions described in [Section 7.3.1, “MMLA: Accumulating widening multiplication of integer matrices”](#) except that the second input is unsigned while the third input is signed.

#### 7.3.2.1. USMMLA (vector, vector, vector)

#### Instances

```
svint32_t svusmmla[_s32](svint32_t op1, svuint8_t op2, svint8_t op3)
```

### 7.3.3. USDOT: Integer addition of dot product (unsigned×signed)

These functions map to the USDOT instruction. They behave in the same way as the functions described in [Section 6.7.13, “DOT: Integer addition of dot product”](#) except that the second input is unsigned while the third input is signed.

### 7.3.3.1. USDOT (vector, vector, vector)

Instances
svint32_t <b>svusdot</b> [_s32](svint32_t op1, svuint8_t op2, svint8_t op3)

### 7.3.3.2. USDOT (vector, vector, scalar)

Instances
svint32_t <b>svusdot</b> [_n_s32](svint32_t op1, svuint8_t op2, int8_t op3)

### 7.3.3.3. USDOT (vector, vector, vector, lane)

Instances
svint32_t <b>svusdot_lane</b> [_s32](svint32_t op1, svuint8_t op2, svint8_t op3, uint64_t imm_index)

## 7.3.4. SUDOT: Integer addition of dot product (signedxunsigned)

These functions behave in the same way as the functions described in [Section 6.7.13, “DOT: Integer addition of dot product”](#) except that the second input is signed while the third input is unsigned.

The `_lane` function maps to the SUDOT instruction. The other functions map to the USDOT instruction with the second and third inputs swapped.

### 7.3.4.1. SUDOT (vector, vector, vector)

Instances
svint32_t <b>svsudot</b> [_s32](svint32_t op1, svint8_t op2, svuint8_t op3)

### 7.3.4.2. SUDOT (vector, vector, scalar)

Instances
svint32_t <b>svsudot</b> [_n_s32](svint32_t op1, svint8_t op2, uint8_t op3)

### 7.3.4.3. SUDOT (vector, vector, vector, lane)

Instances
svint32_t <b>svsudot_lane</b> [_s32](svint32_t op1, svint8_t op2, svuint8_t op3, uint64_t imm_index)

## 7.4. FP32 matrix multiply extensions

These functions are only available when the feature macro `__ARM_FEATURE_SVE_MATMUL_FP32` is defined.

### 7.4.1. MMLA: Accumulating multiplication of 2x2 float matrices

This function maps to the `.S` form of the FMMLA. instruction. It partitions the inputs into 128-bit quadwords, with each quadword of the first and second inputs containing a row-by-row 2x2 matrix and the

third input containing a column-by-column 2×2 matrix. For each quadword the function then multiplies the second input matrix by the third input matrix and adds the result to the first input matrix. There is a rounding step after each scalar operation; the constituent multiplications and additions are not fused.

Signaling NaNs trigger an IEEE Invalid exception but quiet NaNs do not. Multiplying an infinity by zero also triggers an IEEE Invalid exception, as does adding +Inf and -Inf together.

#### 7.4.1.1. MMLA (vector, vector, vector)

Instances
svfloat32_t <b>svmmmla</b> [_f32](svfloat32_t op1, svfloat32_t op2, svfloat32_t op3)

## 7.5. FP64 matrix multiply extensions

These functions are only available when the feature macro `__ARM_FEATURE_SVE_MATMUL_FP64` is defined.

### 7.5.1. MMLA: Accumulating multiplication of 2×2 double matrices

This function maps to the .D form of the FMMLA instruction. If the vector length is a multiple of 256 bits, the function partitions the inputs into 256-bit octowords, with each octoword of the first and second inputs containing a row-by-row 2×2 matrix and the third input containing a column-by-column 2×2 matrix. For each octoword the function then multiplies the second input matrix by the third input matrix and adds the result to the first input matrix. There is a rounding step after each scalar operation; the constituent multiplications and additions are not fused. See the architecture documentation for a description of the behavior for other vector lengths.

Signaling NaNs trigger an IEEE Invalid exception but quiet NaNs do not. Multiplying an infinity by zero also triggers an IEEE Invalid exception, as does adding +Inf and -Inf together.

#### 7.5.1.1. MMLA (vector, vector, vector)

Instances
svfloat64_t <b>svmmmla</b> [_f64](svfloat64_t op1, svfloat64_t op2, svfloat64_t op3)

### 7.5.2. LD1RO: Unextended load and replicate to octoword

These functions map to the LD1ROB, LD1ROH, LD1ROW and LD1ROD instructions. If the vector length is a multiple of 256 bits, the functions load 256 bits of data under predicate control, setting inactive elements in the 256 bits to zero. The functions then duplicate the data to every 256-bit octoword of the vector result. See the architecture documentation for a description of the behavior for other vector lengths.

#### 7.5.2.1. LD1RO (scalar base)

Instances
svint8_t <b>svld1ro</b> [_s8](svbool_t pg, const int8_t *base)
svint16_t <b>svld1ro</b> [_s16](svbool_t pg, const int16_t *base)
svint32_t <b>svld1ro</b> [_s32](svbool_t pg, const int32_t *base)
svint64_t <b>svld1ro</b> [_s64](svbool_t pg, const int64_t *base)
svuint8_t <b>svld1ro</b> [_u8](svbool_t pg, const uint8_t *base)
svuint16_t <b>svld1ro</b> [_u16](svbool_t pg, const uint16_t *base)
svuint32_t <b>svld1ro</b> [_u32](svbool_t pg, const uint32_t *base)

**Instances**

```

svuint64_t svldlro[_u64](svbool_t pg, const uint64_t *base)
svfloat16_t svldlro[_f16](svbool_t pg, const float16_t *base)
svfloat32_t svldlro[_f32](svbool_t pg, const float32_t *base)
svfloat64_t svldlro[_f64](svbool_t pg, const float64_t *base)
svbfloat16_t svldlro[_bf16](svbool_t pg, const bfloat16_t *base)

```

### 7.5.3. TRN1Q: Interleave even quadwords from two inputs

These functions map to the .Q form of the TRN1 instruction. If the vector length is a multiple of 256 bits, the functions extract the even-indexed 128-bit quadwords from two vectors and interleave them, so that each even quadword of the first input is followed by the corresponding quadword of the second input. See the architecture documentation for a description of the behavior for other vector lengths.

Although there is one instance of this operation for every vector type, the results are bitwise identical.

#### 7.5.3.1. TRN1Q (vector, vector)

**Instances**

```

svint8_t svtrn1q[_s8](svint8_t op1, svint8_t op2)
svint16_t svtrn1q[_s16](svint16_t op1, svint16_t op2)
svint32_t svtrn1q[_s32](svint32_t op1, svint32_t op2)
svint64_t svtrn1q[_s64](svint64_t op1, svint64_t op2)
svuint8_t svtrn1q[_u8](svuint8_t op1, svuint8_t op2)
svuint16_t svtrn1q[_u16](svuint16_t op1, svuint16_t op2)
svuint32_t svtrn1q[_u32](svuint32_t op1, svuint32_t op2)
svuint64_t svtrn1q[_u64](svuint64_t op1, svuint64_t op2)
svfloat16_t svtrn1q[_f16](svfloat16_t op1, svfloat16_t op2)
svfloat32_t svtrn1q[_f32](svfloat32_t op1, svfloat32_t op2)
svfloat64_t svtrn1q[_f64](svfloat64_t op1, svfloat64_t op2)
svbfloat16_t svtrn1q[_bf16](svbfloat16_t op1, svbfloat16_t op2)

```

### 7.5.4. TRN2Q: Interleave odd quadwords from two inputs

These functions map to the .Q form of the TRN2 instruction. If the vector length is a multiple of 256 bits, the functions extract the odd-indexed 128-bit quadwords from two vectors and interleave them, so that each odd quadword of the first input is followed by the corresponding quadword of the second input. See the architecture documentation for a description of the behavior for other vector lengths.

Although there is one instance of this operation for every vector type, the results are bitwise identical.

#### 7.5.4.1. TRN2Q (vector, vector)

**Instances**

```

svint8_t svtrn2q[_s8](svint8_t op1, svint8_t op2)
svint16_t svtrn2q[_s16](svint16_t op1, svint16_t op2)
svint32_t svtrn2q[_s32](svint32_t op1, svint32_t op2)
svint64_t svtrn2q[_s64](svint64_t op1, svint64_t op2)
svuint8_t svtrn2q[_u8](svuint8_t op1, svuint8_t op2)
svuint16_t svtrn2q[_u16](svuint16_t op1, svuint16_t op2)
svuint32_t svtrn2q[_u32](svuint32_t op1, svuint32_t op2)
svuint64_t svtrn2q[_u64](svuint64_t op1, svuint64_t op2)
svfloat16_t svtrn2q[_f16](svfloat16_t op1, svfloat16_t op2)
svfloat32_t svtrn2q[_f32](svfloat32_t op1, svfloat32_t op2)
svfloat64_t svtrn2q[_f64](svfloat64_t op1, svfloat64_t op2)

```

**Instances**

```
svbfloat16_t svtrn2q[_bf16](svbfloat16_t op1, svbfloat16_t op2)
```

## 7.5.5. UZP1Q: Select even quadwords from two inputs

These functions map to the .Q form of the UZP1 instruction. If the vector length is a multiple of 256 bits, the functions extract the even-indexed 128-bit quadwords from two vectors and concatenate them together. See the architecture documentation for a description of the behavior for other vector lengths.

Although there is one instance of this operation for every vector type, the results are bitwise identical.

### 7.5.5.1. UZP1Q (vector, vector)

**Instances**

```
svint8_t svuzp1q[_s8](svint8_t op1, svint8_t op2)
svint16_t svuzp1q[_s16](svint16_t op1, svint16_t op2)
svint32_t svuzp1q[_s32](svint32_t op1, svint32_t op2)
svint64_t svuzp1q[_s64](svint64_t op1, svint64_t op2)
svuint8_t svuzp1q[_u8](svuint8_t op1, svuint8_t op2)
svuint16_t svuzp1q[_u16](svuint16_t op1, svuint16_t op2)
svuint32_t svuzp1q[_u32](svuint32_t op1, svuint32_t op2)
svuint64_t svuzp1q[_u64](svuint64_t op1, svuint64_t op2)
svfloat16_t svuzp1q[_f16](svfloat16_t op1, svfloat16_t op2)
svfloat32_t svuzp1q[_f32](svfloat32_t op1, svfloat32_t op2)
svfloat64_t svuzp1q[_f64](svfloat64_t op1, svfloat64_t op2)
svbfloat16_t svuzp1q[_bf16](svbfloat16_t op1, svbfloat16_t op2)
```

## 7.5.6. UZP2Q: Select odd quadwords from two inputs

These functions map to the .Q form of the UZP2 instruction. If the vector length is a multiple of 256 bits, the functions extract the odd-indexed 128-bit quadwords from two vectors and concatenate them together. See the architecture documentation for a description of the behavior for other vector lengths.

Although there is one instance of this operation for every vector type, the results are bitwise identical.

### 7.5.6.1. UZP2Q (vector, vector)

**Instances**

```
svint8_t svuzp2q[_s8](svint8_t op1, svint8_t op2)
svint16_t svuzp2q[_s16](svint16_t op1, svint16_t op2)
svint32_t svuzp2q[_s32](svint32_t op1, svint32_t op2)
svint64_t svuzp2q[_s64](svint64_t op1, svint64_t op2)
svuint8_t svuzp2q[_u8](svuint8_t op1, svuint8_t op2)
svuint16_t svuzp2q[_u16](svuint16_t op1, svuint16_t op2)
svuint32_t svuzp2q[_u32](svuint32_t op1, svuint32_t op2)
svuint64_t svuzp2q[_u64](svuint64_t op1, svuint64_t op2)
svfloat16_t svuzp2q[_f16](svfloat16_t op1, svfloat16_t op2)
svfloat32_t svuzp2q[_f32](svfloat32_t op1, svfloat32_t op2)
svfloat64_t svuzp2q[_f64](svfloat64_t op1, svfloat64_t op2)
svbfloat16_t svuzp2q[_bf16](svbfloat16_t op1, svbfloat16_t op2)
```

## 7.5.7. ZIP1Q: Interleave quadwords from low halves of two inputs

These functions map to the .Q form of the ZIP1 instruction. If the vector length is a multiple of 256 bits, the functions extract 128-bit quadwords from the lowest-indexed halves of two vectors and interleave them,

so that each low quadword of the first input is followed by the corresponding quadword of the second input. See the architecture documentation for a description of the behavior for other vector lengths.

Although there is one instance of this operation for every vector type, the results are bitwise identical.

### 7.5.7.1. ZIP1Q (vector, vector)

Instances
svint8_t <b>svzip1q</b> [_s8](svint8_t op1, svint8_t op2)
svint16_t <b>svzip1q</b> [_s16](svint16_t op1, svint16_t op2)
svint32_t <b>svzip1q</b> [_s32](svint32_t op1, svint32_t op2)
svint64_t <b>svzip1q</b> [_s64](svint64_t op1, svint64_t op2)
svuint8_t <b>svzip1q</b> [_u8](svuint8_t op1, svuint8_t op2)
svuint16_t <b>svzip1q</b> [_u16](svuint16_t op1, svuint16_t op2)
svuint32_t <b>svzip1q</b> [_u32](svuint32_t op1, svuint32_t op2)
svuint64_t <b>svzip1q</b> [_u64](svuint64_t op1, svuint64_t op2)
svfloat16_t <b>svzip1q</b> [_f16](svfloat16_t op1, svfloat16_t op2)
svfloat32_t <b>svzip1q</b> [_f32](svfloat32_t op1, svfloat32_t op2)
svfloat64_t <b>svzip1q</b> [_f64](svfloat64_t op1, svfloat64_t op2)
svbfloat16_t <b>svzip1q</b> [_bf16](svbfloat16_t op1, svbfloat16_t op2)

## 7.5.8. ZIP2Q: Interleave quadwords from high halves of two inputs

These functions map to the .Q form of the ZIP2 instruction. If the vector length is a multiple of 256 bits, the functions extract 128-bit quadwords from the highest-indexed halves of two vectors and interleave them, so that each high quadword of the first input is followed by the corresponding quadword of the second input. See the architecture documentation for a description of the behavior for other vector lengths.

Although there is one instance of this operation for every vector type, the results are bitwise identical.

### 7.5.8.1. ZIP2Q (vector, vector)

Instances
svint8_t <b>svzip2q</b> [_s8](svint8_t op1, svint8_t op2)
svint16_t <b>svzip2q</b> [_s16](svint16_t op1, svint16_t op2)
svint32_t <b>svzip2q</b> [_s32](svint32_t op1, svint32_t op2)
svint64_t <b>svzip2q</b> [_s64](svint64_t op1, svint64_t op2)
svuint8_t <b>svzip2q</b> [_u8](svuint8_t op1, svuint8_t op2)
svuint16_t <b>svzip2q</b> [_u16](svuint16_t op1, svuint16_t op2)
svuint32_t <b>svzip2q</b> [_u32](svuint32_t op1, svuint32_t op2)
svuint64_t <b>svzip2q</b> [_u64](svuint64_t op1, svuint64_t op2)
svfloat16_t <b>svzip2q</b> [_f16](svfloat16_t op1, svfloat16_t op2)
svfloat32_t <b>svzip2q</b> [_f32](svfloat32_t op1, svfloat32_t op2)
svfloat64_t <b>svzip2q</b> [_f64](svfloat64_t op1, svfloat64_t op2)
svbfloat16_t <b>svzip2q</b> [_bf16](svbfloat16_t op1, svbfloat16_t op2)

## 8. List of base SVE2 functions

### 8.1. Introduction

This section contains a list of base SVE2 functions, grouped into categories and then subdivided based on the first part of the name (up to the first underscore). The functions are only available when the feature macro `__ARM_FEATURE_SVE2` is defined.

### 8.2. While greater comparisons

#### 8.2.1. WHILEGT: While decrementing variable is greater than

These functions return the reverse of a predicate in which element  $N$  is active if, for all values  $M$  in the range  $[0, N]$ , subtracting  $M$  from the first input gives a value that is greater than the second. (Note that the behavior is the same regardless of whether the subtraction uses modular, saturating or natural arithmetic.)

See [Section 6.12.1, “WHILELT: While incrementing variable is less than”](#) for a description of the suffix.

These functions handle both signed and unsigned inputs; there are no separate functions for WHILEHI.

##### 8.2.1.1. WHILEGT (scalar, scalar)

Instances	
<code>svbool_t</code>	<code>svwhilegt_b8[_s32](int32_t op1, int32_t op2)</code>
<code>svbool_t</code>	<code>svwhilegt_b16[_s32](int32_t op1, int32_t op2)</code>
<code>svbool_t</code>	<code>svwhilegt_b32[_s32](int32_t op1, int32_t op2)</code>
<code>svbool_t</code>	<code>svwhilegt_b64[_s32](int32_t op1, int32_t op2)</code>
<code>svbool_t</code>	<code>svwhilegt_b8[_u32](uint32_t op1, uint32_t op2)</code>
<code>svbool_t</code>	<code>svwhilegt_b16[_u32](uint32_t op1, uint32_t op2)</code>
<code>svbool_t</code>	<code>svwhilegt_b32[_u32](uint32_t op1, uint32_t op2)</code>
<code>svbool_t</code>	<code>svwhilegt_b64[_u32](uint32_t op1, uint32_t op2)</code>
<code>svbool_t</code>	<code>svwhilegt_b8[_s64](int64_t op1, int64_t op2)</code>
<code>svbool_t</code>	<code>svwhilegt_b16[_s64](int64_t op1, int64_t op2)</code>
<code>svbool_t</code>	<code>svwhilegt_b32[_s64](int64_t op1, int64_t op2)</code>
<code>svbool_t</code>	<code>svwhilegt_b64[_s64](int64_t op1, int64_t op2)</code>
<code>svbool_t</code>	<code>svwhilegt_b8[_u64](uint64_t op1, uint64_t op2)</code>
<code>svbool_t</code>	<code>svwhilegt_b16[_u64](uint64_t op1, uint64_t op2)</code>
<code>svbool_t</code>	<code>svwhilegt_b32[_u64](uint64_t op1, uint64_t op2)</code>
<code>svbool_t</code>	<code>svwhilegt_b64[_u64](uint64_t op1, uint64_t op2)</code>

#### 8.2.2. WHILEGE: While decrementing variable is greater than or equal to

These functions return the reverse of a predicate in which element  $N$  is active if, for all values  $M$  in the range  $[0, N]$ , subtracting  $M$  from the first input gives a value that is greater than or equal to the second. The subtraction uses modular arithmetic in the type of the first operand, so subtracting one from the minimum value gives the maximum value. This wrapping is well-defined even for signed types.

See [Section 6.12.1, “WHILELT: While incrementing variable is less than”](#) for a description of the suffix.

These functions handle both signed and unsigned inputs; there are no separate functions for WHILEHS. Note that when the second operand is the minimum value, every element in the returned predicate will be active.



### 8.2.2.1. WHILEGE (scalar, scalar)

Instances	
svbool_t	<b>svwhilege_b8</b> [_s32](int32_t op1, int32_t op2)
svbool_t	<b>svwhilege_b16</b> [_s32](int32_t op1, int32_t op2)
svbool_t	<b>svwhilege_b32</b> [_s32](int32_t op1, int32_t op2)
svbool_t	<b>svwhilege_b64</b> [_s32](int32_t op1, int32_t op2)
svbool_t	<b>svwhilege_b8</b> [_u32](uint32_t op1, uint32_t op2)
svbool_t	<b>svwhilege_b16</b> [_u32](uint32_t op1, uint32_t op2)
svbool_t	<b>svwhilege_b32</b> [_u32](uint32_t op1, uint32_t op2)
svbool_t	<b>svwhilege_b64</b> [_u32](uint32_t op1, uint32_t op2)
svbool_t	<b>svwhilege_b8</b> [_s64](int64_t op1, int64_t op2)
svbool_t	<b>svwhilege_b16</b> [_s64](int64_t op1, int64_t op2)
svbool_t	<b>svwhilege_b32</b> [_s64](int64_t op1, int64_t op2)
svbool_t	<b>svwhilege_b64</b> [_s64](int64_t op1, int64_t op2)
svbool_t	<b>svwhilege_b8</b> [_u64](uint64_t op1, uint64_t op2)
svbool_t	<b>svwhilege_b16</b> [_u64](uint64_t op1, uint64_t op2)
svbool_t	<b>svwhilege_b32</b> [_u64](uint64_t op1, uint64_t op2)
svbool_t	<b>svwhilege_b64</b> [_u64](uint64_t op1, uint64_t op2)

## 8.3. Uniform DSP operations

### 8.3.1. QADD: Saturating integer addition

These functions provide additional forms of saturating integer addition; see [Section 6.7.2, “QADD: Saturating integer addition”](#) for the original SVE instructions. Also see [Section 8.3.2, “SQADD: Saturating integer addition of signed value”](#) and [Section 8.3.3, “UQADD: Saturating integer addition of unsigned value”](#) for forms of saturating addition that have mixed input types.

#### 8.3.1.1. QADD (vector, vector), setting inactive to zero

Instances	
svint8_t	<b>svqadd</b> [_s8]_z(svbool_t pg, svint8_t op1, svint8_t op2)
svint16_t	<b>svqadd</b> [_s16]_z(svbool_t pg, svint16_t op1, svint16_t op2)
svint32_t	<b>svqadd</b> [_s32]_z(svbool_t pg, svint32_t op1, svint32_t op2)
svint64_t	<b>svqadd</b> [_s64]_z(svbool_t pg, svint64_t op1, svint64_t op2)
svuint8_t	<b>svqadd</b> [_u8]_z(svbool_t pg, svuint8_t op1, svuint8_t op2)
svuint16_t	<b>svqadd</b> [_u16]_z(svbool_t pg, svuint16_t op1, svuint16_t op2)
svuint32_t	<b>svqadd</b> [_u32]_z(svbool_t pg, svuint32_t op1, svuint32_t op2)
svuint64_t	<b>svqadd</b> [_u64]_z(svbool_t pg, svuint64_t op1, svuint64_t op2)

#### 8.3.1.2. QADD (vector, vector), merging with first input

Instances	
svint8_t	<b>svqadd</b> [_s8]_m(svbool_t pg, svint8_t op1, svint8_t op2)
svint16_t	<b>svqadd</b> [_s16]_m(svbool_t pg, svint16_t op1, svint16_t op2)
svint32_t	<b>svqadd</b> [_s32]_m(svbool_t pg, svint32_t op1, svint32_t op2)
svint64_t	<b>svqadd</b> [_s64]_m(svbool_t pg, svint64_t op1, svint64_t op2)
svuint8_t	<b>svqadd</b> [_u8]_m(svbool_t pg, svuint8_t op1, svuint8_t op2)
svuint16_t	<b>svqadd</b> [_u16]_m(svbool_t pg, svuint16_t op1, svuint16_t op2)
svuint32_t	<b>svqadd</b> [_u32]_m(svbool_t pg, svuint32_t op1, svuint32_t op2)
svuint64_t	<b>svqadd</b> [_u64]_m(svbool_t pg, svuint64_t op1, svuint64_t op2)

### 8.3.1.3. QADD (vector, vector), setting inactive to unknown

Instances
<pre> svint8_t  svqadd[_s8]_x(svbool_t pg, svint8_t op1, svint8_t op2) svint16_t svqadd[_s16]_x(svbool_t pg, svint16_t op1, svint16_t op2) svint32_t svqadd[_s32]_x(svbool_t pg, svint32_t op1, svint32_t op2) svint64_t svqadd[_s64]_x(svbool_t pg, svint64_t op1, svint64_t op2) svuint8_t svqadd[_u8]_x(svbool_t pg, svuint8_t op1, svuint8_t op2) svuint16_t svqadd[_u16]_x(svbool_t pg, svuint16_t op1, svuint16_t op2) svuint32_t svqadd[_u32]_x(svbool_t pg, svuint32_t op1, svuint32_t op2) svuint64_t svqadd[_u64]_x(svbool_t pg, svuint64_t op1, svuint64_t op2) </pre>

### 8.3.1.4. QADD (vector, scalar), setting inactive to zero

Instances
<pre> svint8_t  svqadd[_n_s8]_z(svbool_t pg, svint8_t op1, int8_t op2) svint16_t svqadd[_n_s16]_z(svbool_t pg, svint16_t op1, int16_t op2) svint32_t svqadd[_n_s32]_z(svbool_t pg, svint32_t op1, int32_t op2) svint64_t svqadd[_n_s64]_z(svbool_t pg, svint64_t op1, int64_t op2) svuint8_t svqadd[_n_u8]_z(svbool_t pg, svuint8_t op1, uint8_t op2) svuint16_t svqadd[_n_u16]_z(svbool_t pg, svuint16_t op1, uint16_t op2) svuint32_t svqadd[_n_u32]_z(svbool_t pg, svuint32_t op1, uint32_t op2) svuint64_t svqadd[_n_u64]_z(svbool_t pg, svuint64_t op1, uint64_t op2) </pre>

### 8.3.1.5. QADD (vector, scalar), merging with first input

Instances
<pre> svint8_t  svqadd[_n_s8]_m(svbool_t pg, svint8_t op1, int8_t op2) svint16_t svqadd[_n_s16]_m(svbool_t pg, svint16_t op1, int16_t op2) svint32_t svqadd[_n_s32]_m(svbool_t pg, svint32_t op1, int32_t op2) svint64_t svqadd[_n_s64]_m(svbool_t pg, svint64_t op1, int64_t op2) svuint8_t svqadd[_n_u8]_m(svbool_t pg, svuint8_t op1, uint8_t op2) svuint16_t svqadd[_n_u16]_m(svbool_t pg, svuint16_t op1, uint16_t op2) svuint32_t svqadd[_n_u32]_m(svbool_t pg, svuint32_t op1, uint32_t op2) svuint64_t svqadd[_n_u64]_m(svbool_t pg, svuint64_t op1, uint64_t op2) </pre>

### 8.3.1.6. QADD (vector, scalar), setting inactive to unknown

Instances
<pre> svint8_t  svqadd[_n_s8]_x(svbool_t pg, svint8_t op1, int8_t op2) svint16_t svqadd[_n_s16]_x(svbool_t pg, svint16_t op1, int16_t op2) svint32_t svqadd[_n_s32]_x(svbool_t pg, svint32_t op1, int32_t op2) svint64_t svqadd[_n_s64]_x(svbool_t pg, svint64_t op1, int64_t op2) svuint8_t svqadd[_n_u8]_x(svbool_t pg, svuint8_t op1, uint8_t op2) svuint16_t svqadd[_n_u16]_x(svbool_t pg, svuint16_t op1, uint16_t op2) svuint32_t svqadd[_n_u32]_x(svbool_t pg, svuint32_t op1, uint32_t op2) svuint64_t svqadd[_n_u64]_x(svbool_t pg, svuint64_t op1, uint64_t op2) </pre>

## 8.3.2. SQADD: Saturating integer addition of signed value

These functions act like [Section 8.3.1, “QADD: Saturating integer addition”](#), except that the first input and the result are unsigned, while the second input is signed.

### 8.3.2.1. SQADD (vector, vector), setting inactive to zero

#### Instances

```
svuint8_t svsqadd[_u8]_z(svbool_t pg, svuint8_t op1, svint8_t op2)
svuint16_t svsqadd[_u16]_z(svbool_t pg, svuint16_t op1, svint16_t op2)
svuint32_t svsqadd[_u32]_z(svbool_t pg, svuint32_t op1, svint32_t op2)
svuint64_t svsqadd[_u64]_z(svbool_t pg, svuint64_t op1, svint64_t op2)
```

### 8.3.2.2. SQADD (vector, vector), merging with first input

#### Instances

```
svuint8_t svsqadd[_u8]_m(svbool_t pg, svuint8_t op1, svint8_t op2)
svuint16_t svsqadd[_u16]_m(svbool_t pg, svuint16_t op1, svint16_t op2)
svuint32_t svsqadd[_u32]_m(svbool_t pg, svuint32_t op1, svint32_t op2)
svuint64_t svsqadd[_u64]_m(svbool_t pg, svuint64_t op1, svint64_t op2)
```

### 8.3.2.3. SQADD (vector, vector), setting inactive to unknown

#### Instances

```
svuint8_t svsqadd[_u8]_x(svbool_t pg, svuint8_t op1, svint8_t op2)
svuint16_t svsqadd[_u16]_x(svbool_t pg, svuint16_t op1, svint16_t op2)
svuint32_t svsqadd[_u32]_x(svbool_t pg, svuint32_t op1, svint32_t op2)
svuint64_t svsqadd[_u64]_x(svbool_t pg, svuint64_t op1, svint64_t op2)
```

### 8.3.2.4. SQADD (vector, scalar), setting inactive to zero

#### Instances

```
svuint8_t svsqadd[_n_u8]_z(svbool_t pg, svuint8_t op1, int8_t op2)
svuint16_t svsqadd[_n_u16]_z(svbool_t pg, svuint16_t op1, int16_t op2)
svuint32_t svsqadd[_n_u32]_z(svbool_t pg, svuint32_t op1, int32_t op2)
svuint64_t svsqadd[_n_u64]_z(svbool_t pg, svuint64_t op1, int64_t op2)
```

### 8.3.2.5. SQADD (vector, scalar), merging with first input

#### Instances

```
svuint8_t svsqadd[_n_u8]_m(svbool_t pg, svuint8_t op1, int8_t op2)
svuint16_t svsqadd[_n_u16]_m(svbool_t pg, svuint16_t op1, int16_t op2)
svuint32_t svsqadd[_n_u32]_m(svbool_t pg, svuint32_t op1, int32_t op2)
svuint64_t svsqadd[_n_u64]_m(svbool_t pg, svuint64_t op1, int64_t op2)
```

### 8.3.2.6. SQADD (vector, scalar), setting inactive to unknown

#### Instances

```
svuint8_t svsqadd[_n_u8]_x(svbool_t pg, svuint8_t op1, int8_t op2)
svuint16_t svsqadd[_n_u16]_x(svbool_t pg, svuint16_t op1, int16_t op2)
svuint32_t svsqadd[_n_u32]_x(svbool_t pg, svuint32_t op1, int32_t op2)
svuint64_t svsqadd[_n_u64]_x(svbool_t pg, svuint64_t op1, int64_t op2)
```

## 8.3.3. UQADD: Saturating integer addition of unsigned value

These functions act like [Section 8.3.1, “QADD: Saturating integer addition”](#), except that the first input and the result are signed, while the second input is unsigned.

### 8.3.3.1. UQADD (vector, vector), setting inactive to zero

Instances
svint8_t <b>svuqadd</b> [_s8]_z(svbool_t pg, svint8_t op1, svuint8_t op2)
svint16_t <b>svuqadd</b> [_s16]_z(svbool_t pg, svint16_t op1, svuint16_t op2)
svint32_t <b>svuqadd</b> [_s32]_z(svbool_t pg, svint32_t op1, svuint32_t op2)
svint64_t <b>svuqadd</b> [_s64]_z(svbool_t pg, svint64_t op1, svuint64_t op2)

### 8.3.3.2. UQADD (vector, vector), merging with first input

Instances
svint8_t <b>svuqadd</b> [_s8]_m(svbool_t pg, svint8_t op1, svuint8_t op2)
svint16_t <b>svuqadd</b> [_s16]_m(svbool_t pg, svint16_t op1, svuint16_t op2)
svint32_t <b>svuqadd</b> [_s32]_m(svbool_t pg, svint32_t op1, svuint32_t op2)
svint64_t <b>svuqadd</b> [_s64]_m(svbool_t pg, svint64_t op1, svuint64_t op2)

### 8.3.3.3. UQADD (vector, vector), setting inactive to unknown

Instances
svint8_t <b>svuqadd</b> [_s8]_x(svbool_t pg, svint8_t op1, svuint8_t op2)
svint16_t <b>svuqadd</b> [_s16]_x(svbool_t pg, svint16_t op1, svuint16_t op2)
svint32_t <b>svuqadd</b> [_s32]_x(svbool_t pg, svint32_t op1, svuint32_t op2)
svint64_t <b>svuqadd</b> [_s64]_x(svbool_t pg, svint64_t op1, svuint64_t op2)

### 8.3.3.4. UQADD (vector, scalar), setting inactive to zero

Instances
svint8_t <b>svuqadd</b> [_n_s8]_z(svbool_t pg, svint8_t op1, uint8_t op2)
svint16_t <b>svuqadd</b> [_n_s16]_z(svbool_t pg, svint16_t op1, uint16_t op2)
svint32_t <b>svuqadd</b> [_n_s32]_z(svbool_t pg, svint32_t op1, uint32_t op2)
svint64_t <b>svuqadd</b> [_n_s64]_z(svbool_t pg, svint64_t op1, uint64_t op2)

### 8.3.3.5. UQADD (vector, scalar), merging with first input

Instances
svint8_t <b>svuqadd</b> [_n_s8]_m(svbool_t pg, svint8_t op1, uint8_t op2)
svint16_t <b>svuqadd</b> [_n_s16]_m(svbool_t pg, svint16_t op1, uint16_t op2)
svint32_t <b>svuqadd</b> [_n_s32]_m(svbool_t pg, svint32_t op1, uint32_t op2)
svint64_t <b>svuqadd</b> [_n_s64]_m(svbool_t pg, svint64_t op1, uint64_t op2)

### 8.3.3.6. UQADD (vector, scalar), setting inactive to unknown

Instances
svint8_t <b>svuqadd</b> [_n_s8]_x(svbool_t pg, svint8_t op1, uint8_t op2)
svint16_t <b>svuqadd</b> [_n_s16]_x(svbool_t pg, svint16_t op1, uint16_t op2)
svint32_t <b>svuqadd</b> [_n_s32]_x(svbool_t pg, svint32_t op1, uint32_t op2)
svint64_t <b>svuqadd</b> [_n_s64]_x(svbool_t pg, svint64_t op1, uint64_t op2)

## 8.3.4. HADD: Halving integer addition

These functions add the two integer inputs together using natural arithmetic, divide the result by 2, and round the quotient towards -Inf.

### 8.3.4.1. HADD (vector, vector), setting inactive to zero

Instances
svint8_t <b>svhadd</b> [_s8]_z(svbool_t pg, svint8_t op1, svint8_t op2)
svint16_t <b>svhadd</b> [_s16]_z(svbool_t pg, svint16_t op1, svint16_t op2)
svint32_t <b>svhadd</b> [_s32]_z(svbool_t pg, svint32_t op1, svint32_t op2)
svint64_t <b>svhadd</b> [_s64]_z(svbool_t pg, svint64_t op1, svint64_t op2)
svuint8_t <b>svhadd</b> [_u8]_z(svbool_t pg, svuint8_t op1, svuint8_t op2)
svuint16_t <b>svhadd</b> [_u16]_z(svbool_t pg, svuint16_t op1, svuint16_t op2)
svuint32_t <b>svhadd</b> [_u32]_z(svbool_t pg, svuint32_t op1, svuint32_t op2)
svuint64_t <b>svhadd</b> [_u64]_z(svbool_t pg, svuint64_t op1, svuint64_t op2)

### 8.3.4.2. HADD (vector, vector), merging with first input

Instances
svint8_t <b>svhadd</b> [_s8]_m(svbool_t pg, svint8_t op1, svint8_t op2)
svint16_t <b>svhadd</b> [_s16]_m(svbool_t pg, svint16_t op1, svint16_t op2)
svint32_t <b>svhadd</b> [_s32]_m(svbool_t pg, svint32_t op1, svint32_t op2)
svint64_t <b>svhadd</b> [_s64]_m(svbool_t pg, svint64_t op1, svint64_t op2)
svuint8_t <b>svhadd</b> [_u8]_m(svbool_t pg, svuint8_t op1, svuint8_t op2)
svuint16_t <b>svhadd</b> [_u16]_m(svbool_t pg, svuint16_t op1, svuint16_t op2)
svuint32_t <b>svhadd</b> [_u32]_m(svbool_t pg, svuint32_t op1, svuint32_t op2)
svuint64_t <b>svhadd</b> [_u64]_m(svbool_t pg, svuint64_t op1, svuint64_t op2)

### 8.3.4.3. HADD (vector, vector), setting inactive to unknown

Instances
svint8_t <b>svhadd</b> [_s8]_x(svbool_t pg, svint8_t op1, svint8_t op2)
svint16_t <b>svhadd</b> [_s16]_x(svbool_t pg, svint16_t op1, svint16_t op2)
svint32_t <b>svhadd</b> [_s32]_x(svbool_t pg, svint32_t op1, svint32_t op2)
svint64_t <b>svhadd</b> [_s64]_x(svbool_t pg, svint64_t op1, svint64_t op2)
svuint8_t <b>svhadd</b> [_u8]_x(svbool_t pg, svuint8_t op1, svuint8_t op2)
svuint16_t <b>svhadd</b> [_u16]_x(svbool_t pg, svuint16_t op1, svuint16_t op2)
svuint32_t <b>svhadd</b> [_u32]_x(svbool_t pg, svuint32_t op1, svuint32_t op2)
svuint64_t <b>svhadd</b> [_u64]_x(svbool_t pg, svuint64_t op1, svuint64_t op2)

### 8.3.4.4. HADD (vector, scalar), setting inactive to zero

Instances
svint8_t <b>svhadd</b> [_n_s8]_z(svbool_t pg, svint8_t op1, int8_t op2)
svint16_t <b>svhadd</b> [_n_s16]_z(svbool_t pg, svint16_t op1, int16_t op2)
svint32_t <b>svhadd</b> [_n_s32]_z(svbool_t pg, svint32_t op1, int32_t op2)
svint64_t <b>svhadd</b> [_n_s64]_z(svbool_t pg, svint64_t op1, int64_t op2)
svuint8_t <b>svhadd</b> [_n_u8]_z(svbool_t pg, svuint8_t op1, uint8_t op2)
svuint16_t <b>svhadd</b> [_n_u16]_z(svbool_t pg, svuint16_t op1, uint16_t op2)
svuint32_t <b>svhadd</b> [_n_u32]_z(svbool_t pg, svuint32_t op1, uint32_t op2)
svuint64_t <b>svhadd</b> [_n_u64]_z(svbool_t pg, svuint64_t op1, uint64_t op2)

### 8.3.4.5. HADD (vector, scalar), merging with first input

Instances
svint8_t <b>svhadd</b> [_n_s8]_m(svbool_t pg, svint8_t op1, int8_t op2)
svint16_t <b>svhadd</b> [_n_s16]_m(svbool_t pg, svint16_t op1, int16_t op2)
svint32_t <b>svhadd</b> [_n_s32]_m(svbool_t pg, svint32_t op1, int32_t op2)

Instances
svint64_t <b>svhadd</b> [_n_s64]_m(svbool_t pg, svint64_t op1, int64_t op2)
svuint8_t <b>svhadd</b> [_n_u8]_m(svbool_t pg, svuint8_t op1, uint8_t op2)
svuint16_t <b>svhadd</b> [_n_u16]_m(svbool_t pg, svuint16_t op1, uint16_t op2)
svuint32_t <b>svhadd</b> [_n_u32]_m(svbool_t pg, svuint32_t op1, uint32_t op2)
svuint64_t <b>svhadd</b> [_n_u64]_m(svbool_t pg, svuint64_t op1, uint64_t op2)

### 8.3.4.6. HADD (vector, scalar), setting inactive to unknown

Instances
svint8_t <b>svhadd</b> [_n_s8]_x(svbool_t pg, svint8_t op1, int8_t op2)
svint16_t <b>svhadd</b> [_n_s16]_x(svbool_t pg, svint16_t op1, int16_t op2)
svint32_t <b>svhadd</b> [_n_s32]_x(svbool_t pg, svint32_t op1, int32_t op2)
svint64_t <b>svhadd</b> [_n_s64]_x(svbool_t pg, svint64_t op1, int64_t op2)
svuint8_t <b>svhadd</b> [_n_u8]_x(svbool_t pg, svuint8_t op1, uint8_t op2)
svuint16_t <b>svhadd</b> [_n_u16]_x(svbool_t pg, svuint16_t op1, uint16_t op2)
svuint32_t <b>svhadd</b> [_n_u32]_x(svbool_t pg, svuint32_t op1, uint32_t op2)
svuint64_t <b>svhadd</b> [_n_u64]_x(svbool_t pg, svuint64_t op1, uint64_t op2)

### 8.3.5. RHADD: Rounding halving integer addition

These functions add the two integer inputs together using natural arithmetic, divide the result by 2, and round the quotient towards +Inf.

#### 8.3.5.1. RHADD (vector, vector), setting inactive to zero

Instances
svint8_t <b>svrhadd</b> [_s8]_z(svbool_t pg, svint8_t op1, svint8_t op2)
svint16_t <b>svrhadd</b> [_s16]_z(svbool_t pg, svint16_t op1, svint16_t op2)
svint32_t <b>svrhadd</b> [_s32]_z(svbool_t pg, svint32_t op1, svint32_t op2)
svint64_t <b>svrhadd</b> [_s64]_z(svbool_t pg, svint64_t op1, svint64_t op2)
svuint8_t <b>svrhadd</b> [_u8]_z(svbool_t pg, svuint8_t op1, svuint8_t op2)
svuint16_t <b>svrhadd</b> [_u16]_z(svbool_t pg, svuint16_t op1, svuint16_t op2)
svuint32_t <b>svrhadd</b> [_u32]_z(svbool_t pg, svuint32_t op1, svuint32_t op2)
svuint64_t <b>svrhadd</b> [_u64]_z(svbool_t pg, svuint64_t op1, svuint64_t op2)

#### 8.3.5.2. RHADD (vector, vector), merging with first input

Instances
svint8_t <b>svrhadd</b> [_s8]_m(svbool_t pg, svint8_t op1, svint8_t op2)
svint16_t <b>svrhadd</b> [_s16]_m(svbool_t pg, svint16_t op1, svint16_t op2)
svint32_t <b>svrhadd</b> [_s32]_m(svbool_t pg, svint32_t op1, svint32_t op2)
svint64_t <b>svrhadd</b> [_s64]_m(svbool_t pg, svint64_t op1, svint64_t op2)
svuint8_t <b>svrhadd</b> [_u8]_m(svbool_t pg, svuint8_t op1, svuint8_t op2)
svuint16_t <b>svrhadd</b> [_u16]_m(svbool_t pg, svuint16_t op1, svuint16_t op2)
svuint32_t <b>svrhadd</b> [_u32]_m(svbool_t pg, svuint32_t op1, svuint32_t op2)
svuint64_t <b>svrhadd</b> [_u64]_m(svbool_t pg, svuint64_t op1, svuint64_t op2)

#### 8.3.5.3. RHADD (vector, vector), setting inactive to unknown

Instances
svint8_t <b>svrhadd</b> [_s8]_x(svbool_t pg, svint8_t op1, svint8_t op2)
svint16_t <b>svrhadd</b> [_s16]_x(svbool_t pg, svint16_t op1, svint16_t op2)

Instances	
svint32_t	<b>svrhadd</b> [_s32]_x(svbool_t pg, svint32_t op1, svint32_t op2)
svint64_t	<b>svrhadd</b> [_s64]_x(svbool_t pg, svint64_t op1, svint64_t op2)
svuint8_t	<b>svrhadd</b> [_u8]_x(svbool_t pg, svuint8_t op1, svuint8_t op2)
svuint16_t	<b>svrhadd</b> [_u16]_x(svbool_t pg, svuint16_t op1, svuint16_t op2)
svuint32_t	<b>svrhadd</b> [_u32]_x(svbool_t pg, svuint32_t op1, svuint32_t op2)
svuint64_t	<b>svrhadd</b> [_u64]_x(svbool_t pg, svuint64_t op1, svuint64_t op2)

#### 8.3.5.4. RHADD (vector, scalar), setting inactive to zero

Instances	
svint8_t	<b>svrhadd</b> [_n_s8]_z(svbool_t pg, svint8_t op1, int8_t op2)
svint16_t	<b>svrhadd</b> [_n_s16]_z(svbool_t pg, svint16_t op1, int16_t op2)
svint32_t	<b>svrhadd</b> [_n_s32]_z(svbool_t pg, svint32_t op1, int32_t op2)
svint64_t	<b>svrhadd</b> [_n_s64]_z(svbool_t pg, svint64_t op1, int64_t op2)
svuint8_t	<b>svrhadd</b> [_n_u8]_z(svbool_t pg, svuint8_t op1, uint8_t op2)
svuint16_t	<b>svrhadd</b> [_n_u16]_z(svbool_t pg, svuint16_t op1, uint16_t op2)
svuint32_t	<b>svrhadd</b> [_n_u32]_z(svbool_t pg, svuint32_t op1, uint32_t op2)
svuint64_t	<b>svrhadd</b> [_n_u64]_z(svbool_t pg, svuint64_t op1, uint64_t op2)

#### 8.3.5.5. RHADD (vector, scalar), merging with first input

Instances	
svint8_t	<b>svrhadd</b> [_n_s8]_m(svbool_t pg, svint8_t op1, int8_t op2)
svint16_t	<b>svrhadd</b> [_n_s16]_m(svbool_t pg, svint16_t op1, int16_t op2)
svint32_t	<b>svrhadd</b> [_n_s32]_m(svbool_t pg, svint32_t op1, int32_t op2)
svint64_t	<b>svrhadd</b> [_n_s64]_m(svbool_t pg, svint64_t op1, int64_t op2)
svuint8_t	<b>svrhadd</b> [_n_u8]_m(svbool_t pg, svuint8_t op1, uint8_t op2)
svuint16_t	<b>svrhadd</b> [_n_u16]_m(svbool_t pg, svuint16_t op1, uint16_t op2)
svuint32_t	<b>svrhadd</b> [_n_u32]_m(svbool_t pg, svuint32_t op1, uint32_t op2)
svuint64_t	<b>svrhadd</b> [_n_u64]_m(svbool_t pg, svuint64_t op1, uint64_t op2)

#### 8.3.5.6. RHADD (vector, scalar), setting inactive to unknown

Instances	
svint8_t	<b>svrhadd</b> [_n_s8]_x(svbool_t pg, svint8_t op1, int8_t op2)
svint16_t	<b>svrhadd</b> [_n_s16]_x(svbool_t pg, svint16_t op1, int16_t op2)
svint32_t	<b>svrhadd</b> [_n_s32]_x(svbool_t pg, svint32_t op1, int32_t op2)
svint64_t	<b>svrhadd</b> [_n_s64]_x(svbool_t pg, svint64_t op1, int64_t op2)
svuint8_t	<b>svrhadd</b> [_n_u8]_x(svbool_t pg, svuint8_t op1, uint8_t op2)
svuint16_t	<b>svrhadd</b> [_n_u16]_x(svbool_t pg, svuint16_t op1, uint16_t op2)
svuint32_t	<b>svrhadd</b> [_n_u32]_x(svbool_t pg, svuint32_t op1, uint32_t op2)
svuint64_t	<b>svrhadd</b> [_n_u64]_x(svbool_t pg, svuint64_t op1, uint64_t op2)

### 8.3.6. QSUB: Saturating integer subtraction

These functions provide additional forms of saturating integer subtraction; see [Section 6.7.5, “QSUB: Saturating integer subtraction”](#) for the original SVE instructions.

#### 8.3.6.1. QSUB (vector, vector), setting inactive to zero

Instances	
svint8_t	<b>svqsub</b> [_s8]_z(svbool_t pg, svint8_t op1, svint8_t op2)
svint16_t	<b>svqsub</b> [_s16]_z(svbool_t pg, svint16_t op1, svint16_t op2)

**Instances**

```
svint32_t svqsub[_s32]_z(svbool_t pg, svint32_t op1, svint32_t op2)
svint64_t svqsub[_s64]_z(svbool_t pg, svint64_t op1, svint64_t op2)
svuint8_t svqsub[_u8]_z(svbool_t pg, svuint8_t op1, svuint8_t op2)
svuint16_t svqsub[_u16]_z(svbool_t pg, svuint16_t op1, svuint16_t op2)
svuint32_t svqsub[_u32]_z(svbool_t pg, svuint32_t op1, svuint32_t op2)
svuint64_t svqsub[_u64]_z(svbool_t pg, svuint64_t op1, svuint64_t op2)
```

**8.3.6.2. QSUB (vector, vector), merging with first input****Instances**

```
svint8_t svqsub[_s8]_m(svbool_t pg, svint8_t op1, svint8_t op2)
svint16_t svqsub[_s16]_m(svbool_t pg, svint16_t op1, svint16_t op2)
svint32_t svqsub[_s32]_m(svbool_t pg, svint32_t op1, svint32_t op2)
svint64_t svqsub[_s64]_m(svbool_t pg, svint64_t op1, svint64_t op2)
svuint8_t svqsub[_u8]_m(svbool_t pg, svuint8_t op1, svuint8_t op2)
svuint16_t svqsub[_u16]_m(svbool_t pg, svuint16_t op1, svuint16_t op2)
svuint32_t svqsub[_u32]_m(svbool_t pg, svuint32_t op1, svuint32_t op2)
svuint64_t svqsub[_u64]_m(svbool_t pg, svuint64_t op1, svuint64_t op2)
```

**8.3.6.3. QSUB (vector, vector), setting inactive to unknown****Instances**

```
svint8_t svqsub[_s8]_x(svbool_t pg, svint8_t op1, svint8_t op2)
svint16_t svqsub[_s16]_x(svbool_t pg, svint16_t op1, svint16_t op2)
svint32_t svqsub[_s32]_x(svbool_t pg, svint32_t op1, svint32_t op2)
svint64_t svqsub[_s64]_x(svbool_t pg, svint64_t op1, svint64_t op2)
svuint8_t svqsub[_u8]_x(svbool_t pg, svuint8_t op1, svuint8_t op2)
svuint16_t svqsub[_u16]_x(svbool_t pg, svuint16_t op1, svuint16_t op2)
svuint32_t svqsub[_u32]_x(svbool_t pg, svuint32_t op1, svuint32_t op2)
svuint64_t svqsub[_u64]_x(svbool_t pg, svuint64_t op1, svuint64_t op2)
```

**8.3.6.4. QSUB (vector, scalar), setting inactive to zero****Instances**

```
svint8_t svqsub[_n_s8]_z(svbool_t pg, svint8_t op1, int8_t op2)
svint16_t svqsub[_n_s16]_z(svbool_t pg, svint16_t op1, int16_t op2)
svint32_t svqsub[_n_s32]_z(svbool_t pg, svint32_t op1, int32_t op2)
svint64_t svqsub[_n_s64]_z(svbool_t pg, svint64_t op1, int64_t op2)
svuint8_t svqsub[_n_u8]_z(svbool_t pg, svuint8_t op1, uint8_t op2)
svuint16_t svqsub[_n_u16]_z(svbool_t pg, svuint16_t op1, uint16_t op2)
svuint32_t svqsub[_n_u32]_z(svbool_t pg, svuint32_t op1, uint32_t op2)
svuint64_t svqsub[_n_u64]_z(svbool_t pg, svuint64_t op1, uint64_t op2)
```

**8.3.6.5. QSUB (vector, scalar), merging with first input****Instances**

```
svint8_t svqsub[_n_s8]_m(svbool_t pg, svint8_t op1, int8_t op2)
svint16_t svqsub[_n_s16]_m(svbool_t pg, svint16_t op1, int16_t op2)
svint32_t svqsub[_n_s32]_m(svbool_t pg, svint32_t op1, int32_t op2)
svint64_t svqsub[_n_s64]_m(svbool_t pg, svint64_t op1, int64_t op2)
svuint8_t svqsub[_n_u8]_m(svbool_t pg, svuint8_t op1, uint8_t op2)
svuint16_t svqsub[_n_u16]_m(svbool_t pg, svuint16_t op1, uint16_t op2)
svuint32_t svqsub[_n_u32]_m(svbool_t pg, svuint32_t op1, uint32_t op2)
```



**Instances**

```
svuint64_t svqsub[_n_u64]_m(svbool_t pg, svuint64_t op1, uint64_t op2)
```

**8.3.6.6. QSUB (vector, scalar), setting inactive to unknown****Instances**

```
svint8_t svqsub[_n_s8]_x(svbool_t pg, svint8_t op1, int8_t op2)
svint16_t svqsub[_n_s16]_x(svbool_t pg, svint16_t op1, int16_t op2)
svint32_t svqsub[_n_s32]_x(svbool_t pg, svint32_t op1, int32_t op2)
svint64_t svqsub[_n_s64]_x(svbool_t pg, svint64_t op1, int64_t op2)
svuint8_t svqsub[_n_u8]_x(svbool_t pg, svuint8_t op1, uint8_t op2)
svuint16_t svqsub[_n_u16]_x(svbool_t pg, svuint16_t op1, uint16_t op2)
svuint32_t svqsub[_n_u32]_x(svbool_t pg, svuint32_t op1, uint32_t op2)
svuint64_t svqsub[_n_u64]_x(svbool_t pg, svuint64_t op1, uint64_t op2)
```

**8.3.7. QSUBR: Saturating integer subtraction, reversed**

These functions subtract the first integer input from the second integer input. The subtraction uses saturating arithmetic; if the difference is outside the range of the type, the result is the nearest in-range value.

**8.3.7.1. QSUBR (vector, vector), setting inactive to zero****Instances**

```
svint8_t svqsubr[_s8]_z(svbool_t pg, svint8_t op1, svint8_t op2)
svint16_t svqsubr[_s16]_z(svbool_t pg, svint16_t op1, svint16_t op2)
svint32_t svqsubr[_s32]_z(svbool_t pg, svint32_t op1, svint32_t op2)
svint64_t svqsubr[_s64]_z(svbool_t pg, svint64_t op1, svint64_t op2)
svuint8_t svqsubr[_u8]_z(svbool_t pg, svuint8_t op1, svuint8_t op2)
svuint16_t svqsubr[_u16]_z(svbool_t pg, svuint16_t op1, svuint16_t op2)
svuint32_t svqsubr[_u32]_z(svbool_t pg, svuint32_t op1, svuint32_t op2)
svuint64_t svqsubr[_u64]_z(svbool_t pg, svuint64_t op1, svuint64_t op2)
```

**8.3.7.2. QSUBR (vector, vector), merging with first input****Instances**

```
svint8_t svqsubr[_s8]_m(svbool_t pg, svint8_t op1, svint8_t op2)
svint16_t svqsubr[_s16]_m(svbool_t pg, svint16_t op1, svint16_t op2)
svint32_t svqsubr[_s32]_m(svbool_t pg, svint32_t op1, svint32_t op2)
svint64_t svqsubr[_s64]_m(svbool_t pg, svint64_t op1, svint64_t op2)
svuint8_t svqsubr[_u8]_m(svbool_t pg, svuint8_t op1, svuint8_t op2)
svuint16_t svqsubr[_u16]_m(svbool_t pg, svuint16_t op1, svuint16_t op2)
svuint32_t svqsubr[_u32]_m(svbool_t pg, svuint32_t op1, svuint32_t op2)
svuint64_t svqsubr[_u64]_m(svbool_t pg, svuint64_t op1, svuint64_t op2)
```

**8.3.7.3. QSUBR (vector, vector), setting inactive to unknown****Instances**

```
svint8_t svqsubr[_s8]_x(svbool_t pg, svint8_t op1, svint8_t op2)
svint16_t svqsubr[_s16]_x(svbool_t pg, svint16_t op1, svint16_t op2)
svint32_t svqsubr[_s32]_x(svbool_t pg, svint32_t op1, svint32_t op2)
svint64_t svqsubr[_s64]_x(svbool_t pg, svint64_t op1, svint64_t op2)
svuint8_t svqsubr[_u8]_x(svbool_t pg, svuint8_t op1, svuint8_t op2)
svuint16_t svqsubr[_u16]_x(svbool_t pg, svuint16_t op1, svuint16_t op2)
svuint32_t svqsubr[_u32]_x(svbool_t pg, svuint32_t op1, svuint32_t op2)
```

Instances
<code>svuint64_t svqsubr[_u64]_x(svbool_t pg, svuint64_t op1, svuint64_t op2)</code>

### 8.3.7.4. QSUBR (vector, scalar), setting inactive to zero

Instances
<code>svint8_t svqsubr[_n_s8]_z(svbool_t pg, svint8_t op1, int8_t op2)</code>
<code>svint16_t svqsubr[_n_s16]_z(svbool_t pg, svint16_t op1, int16_t op2)</code>
<code>svint32_t svqsubr[_n_s32]_z(svbool_t pg, svint32_t op1, int32_t op2)</code>
<code>svint64_t svqsubr[_n_s64]_z(svbool_t pg, svint64_t op1, int64_t op2)</code>
<code>svuint8_t svqsubr[_n_u8]_z(svbool_t pg, svuint8_t op1, uint8_t op2)</code>
<code>svuint16_t svqsubr[_n_u16]_z(svbool_t pg, svuint16_t op1, uint16_t op2)</code>
<code>svuint32_t svqsubr[_n_u32]_z(svbool_t pg, svuint32_t op1, uint32_t op2)</code>
<code>svuint64_t svqsubr[_n_u64]_z(svbool_t pg, svuint64_t op1, uint64_t op2)</code>

### 8.3.7.5. QSUBR (vector, scalar), merging with first input

Instances
<code>svint8_t svqsubr[_n_s8]_m(svbool_t pg, svint8_t op1, int8_t op2)</code>
<code>svint16_t svqsubr[_n_s16]_m(svbool_t pg, svint16_t op1, int16_t op2)</code>
<code>svint32_t svqsubr[_n_s32]_m(svbool_t pg, svint32_t op1, int32_t op2)</code>
<code>svint64_t svqsubr[_n_s64]_m(svbool_t pg, svint64_t op1, int64_t op2)</code>
<code>svuint8_t svqsubr[_n_u8]_m(svbool_t pg, svuint8_t op1, uint8_t op2)</code>
<code>svuint16_t svqsubr[_n_u16]_m(svbool_t pg, svuint16_t op1, uint16_t op2)</code>
<code>svuint32_t svqsubr[_n_u32]_m(svbool_t pg, svuint32_t op1, uint32_t op2)</code>
<code>svuint64_t svqsubr[_n_u64]_m(svbool_t pg, svuint64_t op1, uint64_t op2)</code>

### 8.3.7.6. QSUBR (vector, scalar), setting inactive to unknown

Instances
<code>svint8_t svqsubr[_n_s8]_x(svbool_t pg, svint8_t op1, int8_t op2)</code>
<code>svint16_t svqsubr[_n_s16]_x(svbool_t pg, svint16_t op1, int16_t op2)</code>
<code>svint32_t svqsubr[_n_s32]_x(svbool_t pg, svint32_t op1, int32_t op2)</code>
<code>svint64_t svqsubr[_n_s64]_x(svbool_t pg, svint64_t op1, int64_t op2)</code>
<code>svuint8_t svqsubr[_n_u8]_x(svbool_t pg, svuint8_t op1, uint8_t op2)</code>
<code>svuint16_t svqsubr[_n_u16]_x(svbool_t pg, svuint16_t op1, uint16_t op2)</code>
<code>svuint32_t svqsubr[_n_u32]_x(svbool_t pg, svuint32_t op1, uint32_t op2)</code>
<code>svuint64_t svqsubr[_n_u64]_x(svbool_t pg, svuint64_t op1, uint64_t op2)</code>

## 8.3.8. HSUB: Halving integer subtraction

These functions subtract the second integer input from the first integer input using natural arithmetic, divide the result by 2, and round the quotient towards -Inf.

### 8.3.8.1. HSUB (vector, vector), setting inactive to zero

Instances
<code>svint8_t svhsub[_s8]_z(svbool_t pg, svint8_t op1, svint8_t op2)</code>
<code>svint16_t svhsub[_s16]_z(svbool_t pg, svint16_t op1, svint16_t op2)</code>
<code>svint32_t svhsub[_s32]_z(svbool_t pg, svint32_t op1, svint32_t op2)</code>
<code>svint64_t svhsub[_s64]_z(svbool_t pg, svint64_t op1, svint64_t op2)</code>
<code>svuint8_t svhsub[_u8]_z(svbool_t pg, svuint8_t op1, svuint8_t op2)</code>
<code>svuint16_t svhsub[_u16]_z(svbool_t pg, svuint16_t op1, svuint16_t op2)</code>
<code>svuint32_t svhsub[_u32]_z(svbool_t pg, svuint32_t op1, svuint32_t op2)</code>
<code>svuint64_t svhsub[_u64]_z(svbool_t pg, svuint64_t op1, svuint64_t op2)</code>

### 8.3.8.2. HSUB (vector, vector), merging with first input

Instances
svint8_t <b>svhsub</b> [_s8]_m(svbool_t pg, svint8_t op1, svint8_t op2)
svint16_t <b>svhsub</b> [_s16]_m(svbool_t pg, svint16_t op1, svint16_t op2)
svint32_t <b>svhsub</b> [_s32]_m(svbool_t pg, svint32_t op1, svint32_t op2)
svint64_t <b>svhsub</b> [_s64]_m(svbool_t pg, svint64_t op1, svint64_t op2)
svuint8_t <b>svhsub</b> [_u8]_m(svbool_t pg, svuint8_t op1, svuint8_t op2)
svuint16_t <b>svhsub</b> [_u16]_m(svbool_t pg, svuint16_t op1, svuint16_t op2)
svuint32_t <b>svhsub</b> [_u32]_m(svbool_t pg, svuint32_t op1, svuint32_t op2)
svuint64_t <b>svhsub</b> [_u64]_m(svbool_t pg, svuint64_t op1, svuint64_t op2)

### 8.3.8.3. HSUB (vector, vector), setting inactive to unknown

Instances
svint8_t <b>svhsub</b> [_s8]_x(svbool_t pg, svint8_t op1, svint8_t op2)
svint16_t <b>svhsub</b> [_s16]_x(svbool_t pg, svint16_t op1, svint16_t op2)
svint32_t <b>svhsub</b> [_s32]_x(svbool_t pg, svint32_t op1, svint32_t op2)
svint64_t <b>svhsub</b> [_s64]_x(svbool_t pg, svint64_t op1, svint64_t op2)
svuint8_t <b>svhsub</b> [_u8]_x(svbool_t pg, svuint8_t op1, svuint8_t op2)
svuint16_t <b>svhsub</b> [_u16]_x(svbool_t pg, svuint16_t op1, svuint16_t op2)
svuint32_t <b>svhsub</b> [_u32]_x(svbool_t pg, svuint32_t op1, svuint32_t op2)
svuint64_t <b>svhsub</b> [_u64]_x(svbool_t pg, svuint64_t op1, svuint64_t op2)

### 8.3.8.4. HSUB (vector, scalar), setting inactive to zero

Instances
svint8_t <b>svhsub</b> [_n_s8]_z(svbool_t pg, svint8_t op1, int8_t op2)
svint16_t <b>svhsub</b> [_n_s16]_z(svbool_t pg, svint16_t op1, int16_t op2)
svint32_t <b>svhsub</b> [_n_s32]_z(svbool_t pg, svint32_t op1, int32_t op2)
svint64_t <b>svhsub</b> [_n_s64]_z(svbool_t pg, svint64_t op1, int64_t op2)
svuint8_t <b>svhsub</b> [_n_u8]_z(svbool_t pg, svuint8_t op1, uint8_t op2)
svuint16_t <b>svhsub</b> [_n_u16]_z(svbool_t pg, svuint16_t op1, uint16_t op2)
svuint32_t <b>svhsub</b> [_n_u32]_z(svbool_t pg, svuint32_t op1, uint32_t op2)
svuint64_t <b>svhsub</b> [_n_u64]_z(svbool_t pg, svuint64_t op1, uint64_t op2)

### 8.3.8.5. HSUB (vector, scalar), merging with first input

Instances
svint8_t <b>svhsub</b> [_n_s8]_m(svbool_t pg, svint8_t op1, int8_t op2)
svint16_t <b>svhsub</b> [_n_s16]_m(svbool_t pg, svint16_t op1, int16_t op2)
svint32_t <b>svhsub</b> [_n_s32]_m(svbool_t pg, svint32_t op1, int32_t op2)
svint64_t <b>svhsub</b> [_n_s64]_m(svbool_t pg, svint64_t op1, int64_t op2)
svuint8_t <b>svhsub</b> [_n_u8]_m(svbool_t pg, svuint8_t op1, uint8_t op2)
svuint16_t <b>svhsub</b> [_n_u16]_m(svbool_t pg, svuint16_t op1, uint16_t op2)
svuint32_t <b>svhsub</b> [_n_u32]_m(svbool_t pg, svuint32_t op1, uint32_t op2)
svuint64_t <b>svhsub</b> [_n_u64]_m(svbool_t pg, svuint64_t op1, uint64_t op2)

### 8.3.8.6. HSUB (vector, scalar), setting inactive to unknown

Instances
svint8_t <b>svhsub</b> [_n_s8]_x(svbool_t pg, svint8_t op1, int8_t op2)
svint16_t <b>svhsub</b> [_n_s16]_x(svbool_t pg, svint16_t op1, int16_t op2)
svint32_t <b>svhsub</b> [_n_s32]_x(svbool_t pg, svint32_t op1, int32_t op2)
svint64_t <b>svhsub</b> [_n_s64]_x(svbool_t pg, svint64_t op1, int64_t op2)

Instances
<code>svuint8_t svhsub[_n_u8]_x(svbool_t pg, svuint8_t op1, uint8_t op2)</code>
<code>svuint16_t svhsub[_n_u16]_x(svbool_t pg, svuint16_t op1, uint16_t op2)</code>
<code>svuint32_t svhsub[_n_u32]_x(svbool_t pg, svuint32_t op1, uint32_t op2)</code>
<code>svuint64_t svhsub[_n_u64]_x(svbool_t pg, svuint64_t op1, uint64_t op2)</code>

### 8.3.9. HSUBR: Halving integer subtraction, reversed

These functions subtract the first integer input from the second integer input using natural arithmetic, divide the result by 2, and round the quotient towards -Inf.

#### 8.3.9.1. HSUBR (vector, vector), setting inactive to zero

Instances
<code>svint8_t svhsubr[_s8]_z(svbool_t pg, svint8_t op1, svint8_t op2)</code>
<code>svint16_t svhsubr[_s16]_z(svbool_t pg, svint16_t op1, svint16_t op2)</code>
<code>svint32_t svhsubr[_s32]_z(svbool_t pg, svint32_t op1, svint32_t op2)</code>
<code>svint64_t svhsubr[_s64]_z(svbool_t pg, svint64_t op1, svint64_t op2)</code>
<code>svuint8_t svhsubr[_u8]_z(svbool_t pg, svuint8_t op1, svuint8_t op2)</code>
<code>svuint16_t svhsubr[_u16]_z(svbool_t pg, svuint16_t op1, svuint16_t op2)</code>
<code>svuint32_t svhsubr[_u32]_z(svbool_t pg, svuint32_t op1, svuint32_t op2)</code>
<code>svuint64_t svhsubr[_u64]_z(svbool_t pg, svuint64_t op1, svuint64_t op2)</code>

#### 8.3.9.2. HSUBR (vector, vector), merging with first input

Instances
<code>svint8_t svhsubr[_s8]_m(svbool_t pg, svint8_t op1, svint8_t op2)</code>
<code>svint16_t svhsubr[_s16]_m(svbool_t pg, svint16_t op1, svint16_t op2)</code>
<code>svint32_t svhsubr[_s32]_m(svbool_t pg, svint32_t op1, svint32_t op2)</code>
<code>svint64_t svhsubr[_s64]_m(svbool_t pg, svint64_t op1, svint64_t op2)</code>
<code>svuint8_t svhsubr[_u8]_m(svbool_t pg, svuint8_t op1, svuint8_t op2)</code>
<code>svuint16_t svhsubr[_u16]_m(svbool_t pg, svuint16_t op1, svuint16_t op2)</code>
<code>svuint32_t svhsubr[_u32]_m(svbool_t pg, svuint32_t op1, svuint32_t op2)</code>
<code>svuint64_t svhsubr[_u64]_m(svbool_t pg, svuint64_t op1, svuint64_t op2)</code>

#### 8.3.9.3. HSUBR (vector, vector), setting inactive to unknown

Instances
<code>svint8_t svhsubr[_s8]_x(svbool_t pg, svint8_t op1, svint8_t op2)</code>
<code>svint16_t svhsubr[_s16]_x(svbool_t pg, svint16_t op1, svint16_t op2)</code>
<code>svint32_t svhsubr[_s32]_x(svbool_t pg, svint32_t op1, svint32_t op2)</code>
<code>svint64_t svhsubr[_s64]_x(svbool_t pg, svint64_t op1, svint64_t op2)</code>
<code>svuint8_t svhsubr[_u8]_x(svbool_t pg, svuint8_t op1, svuint8_t op2)</code>
<code>svuint16_t svhsubr[_u16]_x(svbool_t pg, svuint16_t op1, svuint16_t op2)</code>
<code>svuint32_t svhsubr[_u32]_x(svbool_t pg, svuint32_t op1, svuint32_t op2)</code>
<code>svuint64_t svhsubr[_u64]_x(svbool_t pg, svuint64_t op1, svuint64_t op2)</code>

#### 8.3.9.4. HSUBR (vector, scalar), setting inactive to zero

Instances
<code>svint8_t svhsubr[_n_s8]_z(svbool_t pg, svint8_t op1, int8_t op2)</code>
<code>svint16_t svhsubr[_n_s16]_z(svbool_t pg, svint16_t op1, int16_t op2)</code>
<code>svint32_t svhsubr[_n_s32]_z(svbool_t pg, svint32_t op1, int32_t op2)</code>
<code>svint64_t svhsubr[_n_s64]_z(svbool_t pg, svint64_t op1, int64_t op2)</code>
<code>svuint8_t svhsubr[_n_u8]_z(svbool_t pg, svuint8_t op1, uint8_t op2)</code>

**Instances**

```
svuint16_t svhsubr[_n_u16]_z(svbool_t pg, svuint16_t op1, uint16_t op2)
svuint32_t svhsubr[_n_u32]_z(svbool_t pg, svuint32_t op1, uint32_t op2)
svuint64_t svhsubr[_n_u64]_z(svbool_t pg, svuint64_t op1, uint64_t op2)
```

**8.3.9.5. HSUBR (vector, scalar), merging with first input****Instances**

```
svint8_t svhsubr[_n_s8]_m(svbool_t pg, svint8_t op1, int8_t op2)
svint16_t svhsubr[_n_s16]_m(svbool_t pg, svint16_t op1, int16_t op2)
svint32_t svhsubr[_n_s32]_m(svbool_t pg, svint32_t op1, int32_t op2)
svint64_t svhsubr[_n_s64]_m(svbool_t pg, svint64_t op1, int64_t op2)
svuint8_t svhsubr[_n_u8]_m(svbool_t pg, svuint8_t op1, uint8_t op2)
svuint16_t svhsubr[_n_u16]_m(svbool_t pg, svuint16_t op1, uint16_t op2)
svuint32_t svhsubr[_n_u32]_m(svbool_t pg, svuint32_t op1, uint32_t op2)
svuint64_t svhsubr[_n_u64]_m(svbool_t pg, svuint64_t op1, uint64_t op2)
```

**8.3.9.6. HSUBR (vector, scalar), setting inactive to unknown****Instances**

```
svint8_t svhsubr[_n_s8]_x(svbool_t pg, svint8_t op1, int8_t op2)
svint16_t svhsubr[_n_s16]_x(svbool_t pg, svint16_t op1, int16_t op2)
svint32_t svhsubr[_n_s32]_x(svbool_t pg, svint32_t op1, int32_t op2)
svint64_t svhsubr[_n_s64]_x(svbool_t pg, svint64_t op1, int64_t op2)
svuint8_t svhsubr[_n_u8]_x(svbool_t pg, svuint8_t op1, uint8_t op2)
svuint16_t svhsubr[_n_u16]_x(svbool_t pg, svuint16_t op1, uint16_t op2)
svuint32_t svhsubr[_n_u32]_x(svbool_t pg, svuint32_t op1, uint32_t op2)
svuint64_t svhsubr[_n_u64]_x(svbool_t pg, svuint64_t op1, uint64_t op2)
```

**8.3.10. ABA: Integer addition of absolute difference**

These functions compute the absolute difference between the second and third inputs and add the result to the first input.

The addition uses modular arithmetic and the result is well-defined for all inputs; unlike C, there is no undefined behavior for signed overflow.

**8.3.10.1. ABA (vector, vector, vector)****Instances**

```
svint8_t svaba[_s8](svint8_t op1, svint8_t op2, svint8_t op3)
svint16_t svaba[_s16](svint16_t op1, svint16_t op2, svint16_t op3)
svint32_t svaba[_s32](svint32_t op1, svint32_t op2, svint32_t op3)
svint64_t svaba[_s64](svint64_t op1, svint64_t op2, svint64_t op3)
svuint8_t svaba[_u8](svuint8_t op1, svuint8_t op2, svuint8_t op3)
svuint16_t svaba[_u16](svuint16_t op1, svuint16_t op2, svuint16_t op3)
svuint32_t svaba[_u32](svuint32_t op1, svuint32_t op2, svuint32_t op3)
svuint64_t svaba[_u64](svuint64_t op1, svuint64_t op2, svuint64_t op3)
```

**8.3.10.2. ABA (vector, vector, scalar)****Instances**

```
svint8_t svaba[_n_s8](svint8_t op1, svint8_t op2, int8_t op3)
svint16_t svaba[_n_s16](svint16_t op1, svint16_t op2, int16_t op3)
```

Instances	
svint32_t	<b>svaba</b> [_n_s32](svint32_t op1, svint32_t op2, int32_t op3)
svint64_t	<b>svaba</b> [_n_s64](svint64_t op1, svint64_t op2, int64_t op3)
svuint8_t	<b>svaba</b> [_n_u8](svuint8_t op1, svuint8_t op2, uint8_t op3)
svuint16_t	<b>svaba</b> [_n_u16](svuint16_t op1, svuint16_t op2, uint16_t op3)
svuint32_t	<b>svaba</b> [_n_u32](svuint32_t op1, svuint32_t op2, uint32_t op3)
svuint64_t	<b>svaba</b> [_n_u64](svuint64_t op1, svuint64_t op2, uint64_t op3)

### 8.3.11. QDMULH: Doubling integer multiplication, saturated high half

If the elements have  $N$  bits, these functions multiply the two integer inputs, divide their product by  $2^{N-1}$ , round the quotient towards  $-\text{Inf}$ , and then saturate the result to be in range. All operations prior to the saturation use natural arithmetic.

The `_lane` forms of the functions take one element in each 128-bit quadword of the second input and replicate it to fill the rest of the quadword. The `imm_index` parameter specifies the element within each quadword that the functions should replicate. This index must be an integer constant expression in the range  $[0, 128/N]$ , where  $N$  is the number of bits in each element.

#### 8.3.11.1. QDMULH (vector, vector)

Instances	
svint8_t	<b>svqdmulh</b> [_s8](svint8_t op1, svint8_t op2)
svint16_t	<b>svqdmulh</b> [_s16](svint16_t op1, svint16_t op2)
svint32_t	<b>svqdmulh</b> [_s32](svint32_t op1, svint32_t op2)
svint64_t	<b>svqdmulh</b> [_s64](svint64_t op1, svint64_t op2)

#### 8.3.11.2. QDMULH (vector, scalar)

Instances	
svint8_t	<b>svqdmulh</b> [_n_s8](svint8_t op1, int8_t op2)
svint16_t	<b>svqdmulh</b> [_n_s16](svint16_t op1, int16_t op2)
svint32_t	<b>svqdmulh</b> [_n_s32](svint32_t op1, int32_t op2)
svint64_t	<b>svqdmulh</b> [_n_s64](svint64_t op1, int64_t op2)

#### 8.3.11.3. QDMULH (vector, vector, lane)

Instances	
svint16_t	<b>svqdmulh_lane</b> [_s16](svint16_t op1, svint16_t op2, uint64_t imm_index)
svint32_t	<b>svqdmulh_lane</b> [_s32](svint32_t op1, svint32_t op2, uint64_t imm_index)
svint64_t	<b>svqdmulh_lane</b> [_s64](svint64_t op1, svint64_t op2, uint64_t imm_index)

### 8.3.12. QRDMULH: Doubling integer multiplication, saturated rounded high half

If the elements have  $N$  bits, these functions multiply the two integer inputs, divide their product by  $2^{N-1}$ , round the quotient to nearest with ties towards  $+\text{Inf}$ , and then saturate the result to be in range. All operations prior to the saturation use natural arithmetic.

### 8.3.12.1. QRDMULH (vector, vector)

Instances
svint8_t <b>svqrdmulh</b> [_s8](svint8_t op1, svint8_t op2)
svint16_t <b>svqrdmulh</b> [_s16](svint16_t op1, svint16_t op2)
svint32_t <b>svqrdmulh</b> [_s32](svint32_t op1, svint32_t op2)
svint64_t <b>svqrdmulh</b> [_s64](svint64_t op1, svint64_t op2)

### 8.3.12.2. QRDMULH (vector, scalar)

Instances
svint8_t <b>svqrdmulh</b> [_n_s8](svint8_t op1, int8_t op2)
svint16_t <b>svqrdmulh</b> [_n_s16](svint16_t op1, int16_t op2)
svint32_t <b>svqrdmulh</b> [_n_s32](svint32_t op1, int32_t op2)
svint64_t <b>svqrdmulh</b> [_n_s64](svint64_t op1, int64_t op2)

### 8.3.12.3. QRDMULH (vector, vector, lane)

Instances
svint16_t <b>svqrdmulh_lane</b> [_s16](svint16_t op1, svint16_t op2, uint64_t imm_index)
svint32_t <b>svqrdmulh_lane</b> [_s32](svint32_t op1, svint32_t op2, uint64_t imm_index)
svint64_t <b>svqrdmulh_lane</b> [_s64](svint64_t op1, svint64_t op2, uint64_t imm_index)

## 8.3.13. QRDMLAH: Doubling integer multiplication, saturating addition of rounded high half

If the elements have  $N$  bits, these functions multiply the second and third integer inputs, divide their product by  $2^{N-1}$ , round the quotient to nearest with ties towards +Inf, add the rounded quotient to the first integer input, and then saturate the result to be in range. All operations prior to the saturation use natural arithmetic.

The `_lane` forms of the functions take one element in each 128-bit quadword of the third input and replicate it to fill the rest of the quadword. The `imm_index` parameter specifies the element within each quadword that the functions should replicate. This index must be an integer constant expression in the range  $[0, 128/N)$ , where  $N$  is the number of bits in each element.

### 8.3.13.1. QRDMLAH (vector, vector, vector)

Instances
svint8_t <b>svqrdmlah</b> [_s8](svint8_t op1, svint8_t op2, svint8_t op3)
svint16_t <b>svqrdmlah</b> [_s16](svint16_t op1, svint16_t op2, svint16_t op3)
svint32_t <b>svqrdmlah</b> [_s32](svint32_t op1, svint32_t op2, svint32_t op3)
svint64_t <b>svqrdmlah</b> [_s64](svint64_t op1, svint64_t op2, svint64_t op3)

### 8.3.13.2. QRDMLAH (vector, vector, scalar)

Instances
svint8_t <b>svqrdmlah</b> [_n_s8](svint8_t op1, svint8_t op2, int8_t op3)
svint16_t <b>svqrdmlah</b> [_n_s16](svint16_t op1, svint16_t op2, int16_t op3)
svint32_t <b>svqrdmlah</b> [_n_s32](svint32_t op1, svint32_t op2, int32_t op3)
svint64_t <b>svqrdmlah</b> [_n_s64](svint64_t op1, svint64_t op2, int64_t op3)

### 8.3.13.3. QRDMLAH (vector, vector, vector, lane)

Instances	
svint16_t	<b>svqrdmlah_lane</b> [_s16](svint16_t op1, svint16_t op2, svint16_t op3, uint64_t imm_index)
svint32_t	<b>svqrdmlah_lane</b> [_s32](svint32_t op1, svint32_t op2, svint32_t op3, uint64_t imm_index)
svint64_t	<b>svqrdmlah_lane</b> [_s64](svint64_t op1, svint64_t op2, svint64_t op3, uint64_t imm_index)

## 8.3.14. QRDMLSH: Doubling integer multiplication, saturating subtraction of rounded high half

If the elements have  $N$  bits, these functions multiply the second and third integer inputs, divide their product by  $2^{N-1}$ , round the quotient to nearest with ties towards +Inf, subtract the rounded quotient from the first integer input, and then saturate the result to be in range. All operations prior to the saturation use natural arithmetic.

### 8.3.14.1. QRDMLSH (vector, vector, vector)

Instances	
svint8_t	<b>svqrdmlsh</b> [_s8](svint8_t op1, svint8_t op2, svint8_t op3)
svint16_t	<b>svqrdmlsh</b> [_s16](svint16_t op1, svint16_t op2, svint16_t op3)
svint32_t	<b>svqrdmlsh</b> [_s32](svint32_t op1, svint32_t op2, svint32_t op3)
svint64_t	<b>svqrdmlsh</b> [_s64](svint64_t op1, svint64_t op2, svint64_t op3)

### 8.3.14.2. QRDMLSH (vector, vector, scalar)

Instances	
svint8_t	<b>svqrdmlsh</b> [_n_s8](svint8_t op1, svint8_t op2, int8_t op3)
svint16_t	<b>svqrdmlsh</b> [_n_s16](svint16_t op1, svint16_t op2, int16_t op3)
svint32_t	<b>svqrdmlsh</b> [_n_s32](svint32_t op1, svint32_t op2, int32_t op3)
svint64_t	<b>svqrdmlsh</b> [_n_s64](svint64_t op1, svint64_t op2, int64_t op3)

### 8.3.14.3. QRDMLSH (vector, vector, vector, lane)

Instances	
svint16_t	<b>svqrdmlsh_lane</b> [_s16](svint16_t op1, svint16_t op2, svint16_t op3, uint64_t imm_index)
svint32_t	<b>svqrdmlsh_lane</b> [_s32](svint32_t op1, svint32_t op2, svint32_t op3, uint64_t imm_index)
svint64_t	<b>svqrdmlsh_lane</b> [_s64](svint64_t op1, svint64_t op2, svint64_t op3, uint64_t imm_index)

## 8.3.15. RSHL: Rounding shift left

These functions multiply the first integer input by 2 to the power of the second integer input, round the result to nearest with ties towards +Inf, and then truncate it to be in range. The multiplication uses natural arithmetic.

When the second input is nonnegative, the functions are equivalent to [Section 6.9.1, “LSL: Shift left”](#), since the rounding has no effect in that case.



### 8.3.15.1. RSHL (vector, vector), setting inactive to zero

Instances
svint8_t <b>svrshl</b> [_s8]_z(svbool_t pg, svint8_t op1, svint8_t op2)
svint16_t <b>svrshl</b> [_s16]_z(svbool_t pg, svint16_t op1, svint16_t op2)
svint32_t <b>svrshl</b> [_s32]_z(svbool_t pg, svint32_t op1, svint32_t op2)
svint64_t <b>svrshl</b> [_s64]_z(svbool_t pg, svint64_t op1, svint64_t op2)
svuint8_t <b>svrshl</b> [_u8]_z(svbool_t pg, svuint8_t op1, svint8_t op2)
svuint16_t <b>svrshl</b> [_u16]_z(svbool_t pg, svuint16_t op1, svint16_t op2)
svuint32_t <b>svrshl</b> [_u32]_z(svbool_t pg, svuint32_t op1, svint32_t op2)
svuint64_t <b>svrshl</b> [_u64]_z(svbool_t pg, svuint64_t op1, svint64_t op2)

### 8.3.15.2. RSHL (vector, vector), merging with first input

Instances
svint8_t <b>svrshl</b> [_s8]_m(svbool_t pg, svint8_t op1, svint8_t op2)
svint16_t <b>svrshl</b> [_s16]_m(svbool_t pg, svint16_t op1, svint16_t op2)
svint32_t <b>svrshl</b> [_s32]_m(svbool_t pg, svint32_t op1, svint32_t op2)
svint64_t <b>svrshl</b> [_s64]_m(svbool_t pg, svint64_t op1, svint64_t op2)
svuint8_t <b>svrshl</b> [_u8]_m(svbool_t pg, svuint8_t op1, svint8_t op2)
svuint16_t <b>svrshl</b> [_u16]_m(svbool_t pg, svuint16_t op1, svint16_t op2)
svuint32_t <b>svrshl</b> [_u32]_m(svbool_t pg, svuint32_t op1, svint32_t op2)
svuint64_t <b>svrshl</b> [_u64]_m(svbool_t pg, svuint64_t op1, svint64_t op2)

### 8.3.15.3. RSHL (vector, vector), setting inactive to unknown

Instances
svint8_t <b>svrshl</b> [_s8]_x(svbool_t pg, svint8_t op1, svint8_t op2)
svint16_t <b>svrshl</b> [_s16]_x(svbool_t pg, svint16_t op1, svint16_t op2)
svint32_t <b>svrshl</b> [_s32]_x(svbool_t pg, svint32_t op1, svint32_t op2)
svint64_t <b>svrshl</b> [_s64]_x(svbool_t pg, svint64_t op1, svint64_t op2)
svuint8_t <b>svrshl</b> [_u8]_x(svbool_t pg, svuint8_t op1, svint8_t op2)
svuint16_t <b>svrshl</b> [_u16]_x(svbool_t pg, svuint16_t op1, svint16_t op2)
svuint32_t <b>svrshl</b> [_u32]_x(svbool_t pg, svuint32_t op1, svint32_t op2)
svuint64_t <b>svrshl</b> [_u64]_x(svbool_t pg, svuint64_t op1, svint64_t op2)

### 8.3.15.4. RSHL (vector, scalar), setting inactive to zero

Instances
svint8_t <b>svrshl</b> [_n_s8]_z(svbool_t pg, svint8_t op1, int8_t op2)
svint16_t <b>svrshl</b> [_n_s16]_z(svbool_t pg, svint16_t op1, int16_t op2)
svint32_t <b>svrshl</b> [_n_s32]_z(svbool_t pg, svint32_t op1, int32_t op2)
svint64_t <b>svrshl</b> [_n_s64]_z(svbool_t pg, svint64_t op1, int64_t op2)
svuint8_t <b>svrshl</b> [_n_u8]_z(svbool_t pg, svuint8_t op1, int8_t op2)
svuint16_t <b>svrshl</b> [_n_u16]_z(svbool_t pg, svuint16_t op1, int16_t op2)
svuint32_t <b>svrshl</b> [_n_u32]_z(svbool_t pg, svuint32_t op1, int32_t op2)
svuint64_t <b>svrshl</b> [_n_u64]_z(svbool_t pg, svuint64_t op1, int64_t op2)

### 8.3.15.5. RSHL (vector, scalar), merging with first input

Instances
svint8_t <b>svrshl</b> [_n_s8]_m(svbool_t pg, svint8_t op1, int8_t op2)
svint16_t <b>svrshl</b> [_n_s16]_m(svbool_t pg, svint16_t op1, int16_t op2)
svint32_t <b>svrshl</b> [_n_s32]_m(svbool_t pg, svint32_t op1, int32_t op2)

**Instances**

```
svint64_t svrshl[_n_s64]_m(svbool_t pg, svint64_t op1, int64_t op2)
svuint8_t svrshl[_n_u8]_m(svbool_t pg, svuint8_t op1, int8_t op2)
svuint16_t svrshl[_n_u16]_m(svbool_t pg, svuint16_t op1, int16_t op2)
svuint32_t svrshl[_n_u32]_m(svbool_t pg, svuint32_t op1, int32_t op2)
svuint64_t svrshl[_n_u64]_m(svbool_t pg, svuint64_t op1, int64_t op2)
```

**8.3.15.6. RSHL (vector, scalar), setting inactive to unknown****Instances**

```
svint8_t svrshl[_n_s8]_x(svbool_t pg, svint8_t op1, int8_t op2)
svint16_t svrshl[_n_s16]_x(svbool_t pg, svint16_t op1, int16_t op2)
svint32_t svrshl[_n_s32]_x(svbool_t pg, svint32_t op1, int32_t op2)
svint64_t svrshl[_n_s64]_x(svbool_t pg, svint64_t op1, int64_t op2)
svuint8_t svrshl[_n_u8]_x(svbool_t pg, svuint8_t op1, int8_t op2)
svuint16_t svrshl[_n_u16]_x(svbool_t pg, svuint16_t op1, int16_t op2)
svuint32_t svrshl[_n_u32]_x(svbool_t pg, svuint32_t op1, int32_t op2)
svuint64_t svrshl[_n_u64]_x(svbool_t pg, svuint64_t op1, int64_t op2)
```

**8.3.16. RSHR: Rounding shift right**

These functions divide the first integer input by 2 to the power of the second integer input and round the result to nearest with ties towards +Inf. If the first input has  $N$  bits, the second input must be an integer constant expression in the range  $[1, N]$ .

**8.3.16.1. RSHR (vector, immediate), setting inactive to zero****Instances**

```
svint8_t svrshr[_n_s8]_z(svbool_t pg, svint8_t op1, uint64_t imm2)
svint16_t svrshr[_n_s16]_z(svbool_t pg, svint16_t op1, uint64_t imm2)
svint32_t svrshr[_n_s32]_z(svbool_t pg, svint32_t op1, uint64_t imm2)
svint64_t svrshr[_n_s64]_z(svbool_t pg, svint64_t op1, uint64_t imm2)
svuint8_t svrshr[_n_u8]_z(svbool_t pg, svuint8_t op1, uint64_t imm2)
svuint16_t svrshr[_n_u16]_z(svbool_t pg, svuint16_t op1, uint64_t imm2)
svuint32_t svrshr[_n_u32]_z(svbool_t pg, svuint32_t op1, uint64_t imm2)
svuint64_t svrshr[_n_u64]_z(svbool_t pg, svuint64_t op1, uint64_t imm2)
```

**8.3.16.2. RSHR (vector, immediate), merging with first input****Instances**

```
svint8_t svrshr[_n_s8]_m(svbool_t pg, svint8_t op1, uint64_t imm2)
svint16_t svrshr[_n_s16]_m(svbool_t pg, svint16_t op1, uint64_t imm2)
svint32_t svrshr[_n_s32]_m(svbool_t pg, svint32_t op1, uint64_t imm2)
svint64_t svrshr[_n_s64]_m(svbool_t pg, svint64_t op1, uint64_t imm2)
svuint8_t svrshr[_n_u8]_m(svbool_t pg, svuint8_t op1, uint64_t imm2)
svuint16_t svrshr[_n_u16]_m(svbool_t pg, svuint16_t op1, uint64_t imm2)
svuint32_t svrshr[_n_u32]_m(svbool_t pg, svuint32_t op1, uint64_t imm2)
svuint64_t svrshr[_n_u64]_m(svbool_t pg, svuint64_t op1, uint64_t imm2)
```

**8.3.16.3. RSHR (vector, immediate), setting inactive to unknown****Instances**

```
svint8_t svrshr[_n_s8]_x(svbool_t pg, svint8_t op1, uint64_t imm2)
```

**Instances**

```

svint16_t svrshr[_n_s16]_x(svbool_t pg, svint16_t op1, uint64_t imm2)
svint32_t svrshr[_n_s32]_x(svbool_t pg, svint32_t op1, uint64_t imm2)
svint64_t svrshr[_n_s64]_x(svbool_t pg, svint64_t op1, uint64_t imm2)
svuint8_t svrshr[_n_u8]_x(svbool_t pg, svuint8_t op1, uint64_t imm2)
svuint16_t svrshr[_n_u16]_x(svbool_t pg, svuint16_t op1, uint64_t imm2)
svuint32_t svrshr[_n_u32]_x(svbool_t pg, svuint32_t op1, uint64_t imm2)
svuint64_t svrshr[_n_u64]_x(svbool_t pg, svuint64_t op1, uint64_t imm2)

```

### 8.3.17. QSHL: Saturating shift left

These functions multiply the first integer input by 2 to the power of the second integer input, round the result towards -Inf and then saturate it to be in range. The multiplication uses natural arithmetic.

When the second input is negative, these functions are equivalent to using [Section 6.9.2, “LSR: Logical shift right”](#) or [Section 6.9.3, “ASR: Arithmetic shift right, rounding towards -Inf”](#) with the negated value, since the saturation has no effect in that case.

#### 8.3.17.1. QSHL (vector, vector), setting inactive to zero

**Instances**

```

svint8_t svqshl[_s8]_z(svbool_t pg, svint8_t op1, svint8_t op2)
svint16_t svqshl[_s16]_z(svbool_t pg, svint16_t op1, svint16_t op2)
svint32_t svqshl[_s32]_z(svbool_t pg, svint32_t op1, svint32_t op2)
svint64_t svqshl[_s64]_z(svbool_t pg, svint64_t op1, svint64_t op2)
svuint8_t svqshl[_u8]_z(svbool_t pg, svuint8_t op1, svint8_t op2)
svuint16_t svqshl[_u16]_z(svbool_t pg, svuint16_t op1, svint16_t op2)
svuint32_t svqshl[_u32]_z(svbool_t pg, svuint32_t op1, svint32_t op2)
svuint64_t svqshl[_u64]_z(svbool_t pg, svuint64_t op1, svint64_t op2)

```

#### 8.3.17.2. QSHL (vector, vector), merging with first input

**Instances**

```

svint8_t svqshl[_s8]_m(svbool_t pg, svint8_t op1, svint8_t op2)
svint16_t svqshl[_s16]_m(svbool_t pg, svint16_t op1, svint16_t op2)
svint32_t svqshl[_s32]_m(svbool_t pg, svint32_t op1, svint32_t op2)
svint64_t svqshl[_s64]_m(svbool_t pg, svint64_t op1, svint64_t op2)
svuint8_t svqshl[_u8]_m(svbool_t pg, svuint8_t op1, svint8_t op2)
svuint16_t svqshl[_u16]_m(svbool_t pg, svuint16_t op1, svint16_t op2)
svuint32_t svqshl[_u32]_m(svbool_t pg, svuint32_t op1, svint32_t op2)
svuint64_t svqshl[_u64]_m(svbool_t pg, svuint64_t op1, svint64_t op2)

```

#### 8.3.17.3. QSHL (vector, vector), setting inactive to unknown

**Instances**

```

svint8_t svqshl[_s8]_x(svbool_t pg, svint8_t op1, svint8_t op2)
svint16_t svqshl[_s16]_x(svbool_t pg, svint16_t op1, svint16_t op2)
svint32_t svqshl[_s32]_x(svbool_t pg, svint32_t op1, svint32_t op2)
svint64_t svqshl[_s64]_x(svbool_t pg, svint64_t op1, svint64_t op2)
svuint8_t svqshl[_u8]_x(svbool_t pg, svuint8_t op1, svint8_t op2)
svuint16_t svqshl[_u16]_x(svbool_t pg, svuint16_t op1, svint16_t op2)
svuint32_t svqshl[_u32]_x(svbool_t pg, svuint32_t op1, svint32_t op2)
svuint64_t svqshl[_u64]_x(svbool_t pg, svuint64_t op1, svint64_t op2)

```

### 8.3.17.4. QSHL (vector, scalar), setting inactive to zero

Instances
svint8_t <b>svqshl</b> [_n_s8]_z(svbool_t pg, svint8_t op1, int8_t op2)
svint16_t <b>svqshl</b> [_n_s16]_z(svbool_t pg, svint16_t op1, int16_t op2)
svint32_t <b>svqshl</b> [_n_s32]_z(svbool_t pg, svint32_t op1, int32_t op2)
svint64_t <b>svqshl</b> [_n_s64]_z(svbool_t pg, svint64_t op1, int64_t op2)
svuint8_t <b>svqshl</b> [_n_u8]_z(svbool_t pg, svuint8_t op1, int8_t op2)
svuint16_t <b>svqshl</b> [_n_u16]_z(svbool_t pg, svuint16_t op1, int16_t op2)
svuint32_t <b>svqshl</b> [_n_u32]_z(svbool_t pg, svuint32_t op1, int32_t op2)
svuint64_t <b>svqshl</b> [_n_u64]_z(svbool_t pg, svuint64_t op1, int64_t op2)

### 8.3.17.5. QSHL (vector, scalar), merging with first input

Instances
svint8_t <b>svqshl</b> [_n_s8]_m(svbool_t pg, svint8_t op1, int8_t op2)
svint16_t <b>svqshl</b> [_n_s16]_m(svbool_t pg, svint16_t op1, int16_t op2)
svint32_t <b>svqshl</b> [_n_s32]_m(svbool_t pg, svint32_t op1, int32_t op2)
svint64_t <b>svqshl</b> [_n_s64]_m(svbool_t pg, svint64_t op1, int64_t op2)
svuint8_t <b>svqshl</b> [_n_u8]_m(svbool_t pg, svuint8_t op1, int8_t op2)
svuint16_t <b>svqshl</b> [_n_u16]_m(svbool_t pg, svuint16_t op1, int16_t op2)
svuint32_t <b>svqshl</b> [_n_u32]_m(svbool_t pg, svuint32_t op1, int32_t op2)
svuint64_t <b>svqshl</b> [_n_u64]_m(svbool_t pg, svuint64_t op1, int64_t op2)

### 8.3.17.6. QSHL (vector, scalar), setting inactive to unknown

Instances
svint8_t <b>svqshl</b> [_n_s8]_x(svbool_t pg, svint8_t op1, int8_t op2)
svint16_t <b>svqshl</b> [_n_s16]_x(svbool_t pg, svint16_t op1, int16_t op2)
svint32_t <b>svqshl</b> [_n_s32]_x(svbool_t pg, svint32_t op1, int32_t op2)
svint64_t <b>svqshl</b> [_n_s64]_x(svbool_t pg, svint64_t op1, int64_t op2)
svuint8_t <b>svqshl</b> [_n_u8]_x(svbool_t pg, svuint8_t op1, int8_t op2)
svuint16_t <b>svqshl</b> [_n_u16]_x(svbool_t pg, svuint16_t op1, int16_t op2)
svuint32_t <b>svqshl</b> [_n_u32]_x(svbool_t pg, svuint32_t op1, int32_t op2)
svuint64_t <b>svqshl</b> [_n_u64]_x(svbool_t pg, svuint64_t op1, int64_t op2)

## 8.3.18. QSHLU: Saturating shift left with unsigned result

These functions act like [Section 8.3.17, “QSHL: Saturating shift left”](#), except that the first input is signed and the result is unsigned. If the first input has  $N$  bits, the second input must be an integer constant expression in the range  $[0, N)$ .

### 8.3.18.1. QSHLU (vector, immediate), setting inactive to zero

Instances
svuint8_t <b>svqshlu</b> [_n_s8]_z(svbool_t pg, svint8_t op1, uint64_t imm2)
svuint16_t <b>svqshlu</b> [_n_s16]_z(svbool_t pg, svint16_t op1, uint64_t imm2)
svuint32_t <b>svqshlu</b> [_n_s32]_z(svbool_t pg, svint32_t op1, uint64_t imm2)
svuint64_t <b>svqshlu</b> [_n_s64]_z(svbool_t pg, svint64_t op1, uint64_t imm2)

### 8.3.18.2. QSHLU (vector, immediate), merging with first input

Instances
svuint8_t <b>svqshlu</b> [_n_s8]_m(svbool_t pg, svint8_t op1, uint64_t imm2)

**Instances**

```
svuint16_t svqshlu[_n_s16]_m(svbool_t pg, svint16_t op1, uint64_t imm2)
svuint32_t svqshlu[_n_s32]_m(svbool_t pg, svint32_t op1, uint64_t imm2)
svuint64_t svqshlu[_n_s64]_m(svbool_t pg, svint64_t op1, uint64_t imm2)
```

**8.3.18.3. QSHLU (vector, immediate), setting inactive to unknown****Instances**

```
svuint8_t svqshlu[_n_s8]_x(svbool_t pg, svint8_t op1, uint64_t imm2)
svuint16_t svqshlu[_n_s16]_x(svbool_t pg, svint16_t op1, uint64_t imm2)
svuint32_t svqshlu[_n_s32]_x(svbool_t pg, svint32_t op1, uint64_t imm2)
svuint64_t svqshlu[_n_s64]_x(svbool_t pg, svint64_t op1, uint64_t imm2)
```

**8.3.19. QRSHL: Saturating rounding shift left**

These functions multiply the first integer input by 2 to the power of the second integer input, round the quotient to nearest with ties towards +Inf, and then saturate it to be in range. The multiplication uses natural arithmetic.

When the second input is nonnegative, these functions are equivalent to [Section 8.3.17, “QSHL: Saturating shift left”](#), since the rounding does not matter in that case. Similarly, when the second input is negative, the functions are equivalent to [Section 8.3.15, “RSHL: Rounding shift left”](#), since the saturation has no effect in that case.

**8.3.19.1. QRSHL (vector, vector), setting inactive to zero****Instances**

```
svint8_t svqqrshl[_s8]_z(svbool_t pg, svint8_t op1, svint8_t op2)
svint16_t svqqrshl[_s16]_z(svbool_t pg, svint16_t op1, svint16_t op2)
svint32_t svqqrshl[_s32]_z(svbool_t pg, svint32_t op1, svint32_t op2)
svint64_t svqqrshl[_s64]_z(svbool_t pg, svint64_t op1, svint64_t op2)
svuint8_t svqqrshl[_u8]_z(svbool_t pg, svuint8_t op1, svint8_t op2)
svuint16_t svqqrshl[_u16]_z(svbool_t pg, svuint16_t op1, svint16_t op2)
svuint32_t svqqrshl[_u32]_z(svbool_t pg, svuint32_t op1, svint32_t op2)
svuint64_t svqqrshl[_u64]_z(svbool_t pg, svuint64_t op1, svint64_t op2)
```

**8.3.19.2. QRSHL (vector, vector), merging with first input****Instances**

```
svint8_t svqqrshl[_s8]_m(svbool_t pg, svint8_t op1, svint8_t op2)
svint16_t svqqrshl[_s16]_m(svbool_t pg, svint16_t op1, svint16_t op2)
svint32_t svqqrshl[_s32]_m(svbool_t pg, svint32_t op1, svint32_t op2)
svint64_t svqqrshl[_s64]_m(svbool_t pg, svint64_t op1, svint64_t op2)
svuint8_t svqqrshl[_u8]_m(svbool_t pg, svuint8_t op1, svint8_t op2)
svuint16_t svqqrshl[_u16]_m(svbool_t pg, svuint16_t op1, svint16_t op2)
svuint32_t svqqrshl[_u32]_m(svbool_t pg, svuint32_t op1, svint32_t op2)
svuint64_t svqqrshl[_u64]_m(svbool_t pg, svuint64_t op1, svint64_t op2)
```

**8.3.19.3. QRSHL (vector, vector), setting inactive to unknown****Instances**

```
svint8_t svqqrshl[_s8]_x(svbool_t pg, svint8_t op1, svint8_t op2)
```

Instances	
svint16_t	<b>svqrrshl</b> [_s16]_x(svbool_t pg, svint16_t op1, svint16_t op2)
svint32_t	<b>svqrrshl</b> [_s32]_x(svbool_t pg, svint32_t op1, svint32_t op2)
svint64_t	<b>svqrrshl</b> [_s64]_x(svbool_t pg, svint64_t op1, svint64_t op2)
svuint8_t	<b>svqrrshl</b> [_u8]_x(svbool_t pg, svuint8_t op1, svint8_t op2)
svuint16_t	<b>svqrrshl</b> [_u16]_x(svbool_t pg, svuint16_t op1, svint16_t op2)
svuint32_t	<b>svqrrshl</b> [_u32]_x(svbool_t pg, svuint32_t op1, svint32_t op2)
svuint64_t	<b>svqrrshl</b> [_u64]_x(svbool_t pg, svuint64_t op1, svint64_t op2)

#### 8.3.19.4. QRSHL (vector, scalar), setting inactive to zero

Instances	
svint8_t	<b>svqrrshl</b> [_n_s8]_z(svbool_t pg, svint8_t op1, int8_t op2)
svint16_t	<b>svqrrshl</b> [_n_s16]_z(svbool_t pg, svint16_t op1, int16_t op2)
svint32_t	<b>svqrrshl</b> [_n_s32]_z(svbool_t pg, svint32_t op1, int32_t op2)
svint64_t	<b>svqrrshl</b> [_n_s64]_z(svbool_t pg, svint64_t op1, int64_t op2)
svuint8_t	<b>svqrrshl</b> [_n_u8]_z(svbool_t pg, svuint8_t op1, int8_t op2)
svuint16_t	<b>svqrrshl</b> [_n_u16]_z(svbool_t pg, svuint16_t op1, int16_t op2)
svuint32_t	<b>svqrrshl</b> [_n_u32]_z(svbool_t pg, svuint32_t op1, int32_t op2)
svuint64_t	<b>svqrrshl</b> [_n_u64]_z(svbool_t pg, svuint64_t op1, int64_t op2)

#### 8.3.19.5. QRSHL (vector, scalar), merging with first input

Instances	
svint8_t	<b>svqrrshl</b> [_n_s8]_m(svbool_t pg, svint8_t op1, int8_t op2)
svint16_t	<b>svqrrshl</b> [_n_s16]_m(svbool_t pg, svint16_t op1, int16_t op2)
svint32_t	<b>svqrrshl</b> [_n_s32]_m(svbool_t pg, svint32_t op1, int32_t op2)
svint64_t	<b>svqrrshl</b> [_n_s64]_m(svbool_t pg, svint64_t op1, int64_t op2)
svuint8_t	<b>svqrrshl</b> [_n_u8]_m(svbool_t pg, svuint8_t op1, int8_t op2)
svuint16_t	<b>svqrrshl</b> [_n_u16]_m(svbool_t pg, svuint16_t op1, int16_t op2)
svuint32_t	<b>svqrrshl</b> [_n_u32]_m(svbool_t pg, svuint32_t op1, int32_t op2)
svuint64_t	<b>svqrrshl</b> [_n_u64]_m(svbool_t pg, svuint64_t op1, int64_t op2)

#### 8.3.19.6. QRSHL (vector, scalar), setting inactive to unknown

Instances	
svint8_t	<b>svqrrshl</b> [_n_s8]_x(svbool_t pg, svint8_t op1, int8_t op2)
svint16_t	<b>svqrrshl</b> [_n_s16]_x(svbool_t pg, svint16_t op1, int16_t op2)
svint32_t	<b>svqrrshl</b> [_n_s32]_x(svbool_t pg, svint32_t op1, int32_t op2)
svint64_t	<b>svqrrshl</b> [_n_s64]_x(svbool_t pg, svint64_t op1, int64_t op2)
svuint8_t	<b>svqrrshl</b> [_n_u8]_x(svbool_t pg, svuint8_t op1, int8_t op2)
svuint16_t	<b>svqrrshl</b> [_n_u16]_x(svbool_t pg, svuint16_t op1, int16_t op2)
svuint32_t	<b>svqrrshl</b> [_n_u32]_x(svbool_t pg, svuint32_t op1, int32_t op2)
svuint64_t	<b>svqrrshl</b> [_n_u64]_x(svbool_t pg, svuint64_t op1, int64_t op2)

### 8.3.20. SRA: Shift right and accumulate

These functions divide the second integer input by 2 to the power of the third integer input, round the quotient towards -Inf, and then add it to the first integer input.

The addition uses modular arithmetic and is well-defined for all inputs; unlike C, there is no undefined behavior for signed overflow.

If the second input has  $N$  bits, the third input must be an integer constant expression in the range  $[1, N]$ .

### 8.3.20.1. SRA (vector, vector, immediate)

#### Instances

```
svint8_t  svsra[_n_s8](svint8_t op1, svint8_t op2, uint64_t imm3)
svint16_t svsra[_n_s16](svint16_t op1, svint16_t op2, uint64_t imm3)
svint32_t svsra[_n_s32](svint32_t op1, svint32_t op2, uint64_t imm3)
svint64_t svsra[_n_s64](svint64_t op1, svint64_t op2, uint64_t imm3)
svuint8_t svsra[_n_u8](svuint8_t op1, svuint8_t op2, uint64_t imm3)
svuint16_t svsra[_n_u16](svuint16_t op1, svuint16_t op2, uint64_t imm3)
svuint32_t svsra[_n_u32](svuint32_t op1, svuint32_t op2, uint64_t imm3)
svuint64_t svsra[_n_u64](svuint64_t op1, svuint64_t op2, uint64_t imm3)
```

### 8.3.21. RSRA: Rounding shift right and accumulate

These functions divide the second integer input by 2 to the power of the third integer input, round the quotient to nearest with ties towards +Inf, and then add the rounded quotient to the first integer input.

The addition uses modular arithmetic and is well-defined for all inputs; unlike C, there is no undefined behavior for signed overflow.

If the second input has  $N$  bits, the third input must be an integer constant expression in the range  $[1, N]$ .

#### 8.3.21.1. RSRA (vector, vector, immediate)

#### Instances

```
svint8_t  svrsra[_n_s8](svint8_t op1, svint8_t op2, uint64_t imm3)
svint16_t svrsra[_n_s16](svint16_t op1, svint16_t op2, uint64_t imm3)
svint32_t svrsra[_n_s32](svint32_t op1, svint32_t op2, uint64_t imm3)
svint64_t svrsra[_n_s64](svint64_t op1, svint64_t op2, uint64_t imm3)
svuint8_t svrsra[_n_u8](svuint8_t op1, svuint8_t op2, uint64_t imm3)
svuint16_t svrsra[_n_u16](svuint16_t op1, svuint16_t op2, uint64_t imm3)
svuint32_t svrsra[_n_u32](svuint32_t op1, svuint32_t op2, uint64_t imm3)
svuint64_t svrsra[_n_u64](svuint64_t op1, svuint64_t op2, uint64_t imm3)
```

### 8.3.22. QNEG: Saturating integer negation

These functions negate the integer input using natural arithmetic and then saturate the result; that is, if the negated value is outside the range of the type, the result is the nearest in-range value.

#### 8.3.22.1. QNEG (vector), setting inactive to zero

#### Instances

```
svint8_t  svqneg[_s8]_z(svbool_t pg, svint8_t op)
svint16_t svqneg[_s16]_z(svbool_t pg, svint16_t op)
svint32_t svqneg[_s32]_z(svbool_t pg, svint32_t op)
svint64_t svqneg[_s64]_z(svbool_t pg, svint64_t op)
```

#### 8.3.22.2. QNEG (vector), merging with separate vector

#### Instances

```
svint8_t  svqneg[_s8]_m(svint8_t inactive, svbool_t pg, svint8_t op)
svint16_t svqneg[_s16]_m(svint16_t inactive, svbool_t pg, svint16_t op)
svint32_t svqneg[_s32]_m(svint32_t inactive, svbool_t pg, svint32_t op)
svint64_t svqneg[_s64]_m(svint64_t inactive, svbool_t pg, svint64_t op)
```

### 8.3.22.3. QNEG (vector), setting inactive to unknown

Instances
svint8_t <b>svqneg</b> [_s8]_x(svbool_t pg, svint8_t op)
svint16_t <b>svqneg</b> [_s16]_x(svbool_t pg, svint16_t op)
svint32_t <b>svqneg</b> [_s32]_x(svbool_t pg, svint32_t op)
svint64_t <b>svqneg</b> [_s64]_x(svbool_t pg, svint64_t op)

## 8.3.23. QABS: Saturating integer absolute value

These functions compute the absolute value of the integer input using natural arithmetic and then saturate the result; that is, if the absolute value is outside the range of the type, the result is the nearest in-range value.

### 8.3.23.1. QABS (vector), setting inactive to zero

Instances
svint8_t <b>svqabs</b> [_s8]_z(svbool_t pg, svint8_t op)
svint16_t <b>svqabs</b> [_s16]_z(svbool_t pg, svint16_t op)
svint32_t <b>svqabs</b> [_s32]_z(svbool_t pg, svint32_t op)
svint64_t <b>svqabs</b> [_s64]_z(svbool_t pg, svint64_t op)

### 8.3.23.2. QABS (vector), merging with separate vector

Instances
svint8_t <b>svqabs</b> [_s8]_m(svint8_t inactive, svbool_t pg, svint8_t op)
svint16_t <b>svqabs</b> [_s16]_m(svint16_t inactive, svbool_t pg, svint16_t op)
svint32_t <b>svqabs</b> [_s32]_m(svint32_t inactive, svbool_t pg, svint32_t op)
svint64_t <b>svqabs</b> [_s64]_m(svint64_t inactive, svbool_t pg, svint64_t op)

### 8.3.23.3. QABS (vector), setting inactive to unknown

Instances
svint8_t <b>svqabs</b> [_s8]_x(svbool_t pg, svint8_t op)
svint16_t <b>svqabs</b> [_s16]_x(svbool_t pg, svint16_t op)
svint32_t <b>svqabs</b> [_s32]_x(svbool_t pg, svint32_t op)
svint64_t <b>svqabs</b> [_s64]_x(svbool_t pg, svint64_t op)

## 8.3.24. RECPE: Integer reciprocal estimate

These functions divide the integer input by  $2^{32}$ , calculate the approximate reciprocal, and multiply the result by  $2^{31}$ .

### 8.3.24.1. RECPE (vector), setting inactive to zero

Instances
svuint32_t <b>svrecpe</b> [_u32]_z(svbool_t pg, svuint32_t op)

### 8.3.24.2. RECPE (vector), merging with separate vector

Instances
svuint32_t <b>svrecpe</b> [_u32]_m(svuint32_t inactive, svbool_t pg, svuint32_t op)



### 8.3.24.3. RECPE (vector), setting inactive to unknown

Instances
<code>svuint32_t svrecpe[_u32]_x(svbool_t pg, svuint32_t op)</code>

## 8.3.25. RSQRTE: Integer reciprocal square root estimate

These functions divide the integer input by  $2^{32}$ , calculate the approximate reciprocal square root, and multiply the result by  $2^{31}$ .

### 8.3.25.1. RSQRTE (vector), setting inactive to zero

Instances
<code>svuint32_t svrsqrte[_u32]_z(svbool_t pg, svuint32_t op)</code>

### 8.3.25.2. RSQRTE (vector), merging with separate vector

Instances
<code>svuint32_t svrsqrte[_u32]_m(svuint32_t inactive, svbool_t pg, svuint32_t op)</code>

### 8.3.25.3. RSQRTE (vector), setting inactive to unknown

Instances
<code>svuint32_t svrsqrte[_u32]_x(svbool_t pg, svuint32_t op)</code>

## 8.3.26. SLI: Shift left and insert

If the third integer input has the value *imm3*, these functions shift the second integer input left by *imm3* bits and fill the vacated bits with the low *imm3* bits of the first integer input.

If the second input has *N* bits, the third input must be an integer constant expression in the range  $[0, N]$ .

### 8.3.26.1. SLI (vector, vector, immediate)

Instances
<code>svint8_t svsl_i[_n_s8](svint8_t op1, svint8_t op2, uint64_t imm3)</code>
<code>svint16_t svsl_i[_n_s16](svint16_t op1, svint16_t op2, uint64_t imm3)</code>
<code>svint32_t svsl_i[_n_s32](svint32_t op1, svint32_t op2, uint64_t imm3)</code>
<code>svint64_t svsl_i[_n_s64](svint64_t op1, svint64_t op2, uint64_t imm3)</code>
<code>svuint8_t svsl_i[_n_u8](svuint8_t op1, svuint8_t op2, uint64_t imm3)</code>
<code>svuint16_t svsl_i[_n_u16](svuint16_t op1, svuint16_t op2, uint64_t imm3)</code>
<code>svuint32_t svsl_i[_n_u32](svuint32_t op1, svuint32_t op2, uint64_t imm3)</code>
<code>svuint64_t svsl_i[_n_u64](svuint64_t op1, svuint64_t op2, uint64_t imm3)</code>

## 8.3.27. SRI: Shift right and insert

If the third integer input has the value *imm3*, these functions shift the second integer input right by *imm3* bits and fill the vacated bits with the high *imm3* bits of the first integer input.

If the second input has *N* bits, the third input must be an integer constant expression in the range  $[1, N]$ .

### 8.3.27.1. SRI (vector, vector, immediate)

Instances
svint8_t <b>svsri</b> [_n_s8](svint8_t op1, svint8_t op2, uint64_t imm3)
svint16_t <b>svsri</b> [_n_s16](svint16_t op1, svint16_t op2, uint64_t imm3)
svint32_t <b>svsri</b> [_n_s32](svint32_t op1, svint32_t op2, uint64_t imm3)
svint64_t <b>svsri</b> [_n_s64](svint64_t op1, svint64_t op2, uint64_t imm3)
svuint8_t <b>svsri</b> [_n_u8](svuint8_t op1, svuint8_t op2, uint64_t imm3)
svuint16_t <b>svsri</b> [_n_u16](svuint16_t op1, svuint16_t op2, uint64_t imm3)
svuint32_t <b>svsri</b> [_n_u32](svuint32_t op1, svuint32_t op2, uint64_t imm3)
svuint64_t <b>svsri</b> [_n_u64](svuint64_t op1, svuint64_t op2, uint64_t imm3)

## 8.4. Widening DSP operations

### 8.4.1. ADDLB: Integer addition long (bottom)

These functions extract the even-indexed elements of the integer inputs and add them together, giving a result with twice the width.

#### 8.4.1.1. ADDLB (vector, vector)

Instances
svint16_t <b>svaddlb</b> [_s16](svint8_t op1, svint8_t op2)
svint32_t <b>svaddlb</b> [_s32](svint16_t op1, svint16_t op2)
svint64_t <b>svaddlb</b> [_s64](svint32_t op1, svint32_t op2)
svuint16_t <b>svaddlb</b> [_u16](svuint8_t op1, svuint8_t op2)
svuint32_t <b>svaddlb</b> [_u32](svuint16_t op1, svuint16_t op2)
svuint64_t <b>svaddlb</b> [_u64](svuint32_t op1, svuint32_t op2)

#### 8.4.1.2. ADDLB (vector, scalar)

Instances
svint16_t <b>svaddlb</b> [_n_s16](svint8_t op1, int8_t op2)
svint32_t <b>svaddlb</b> [_n_s32](svint16_t op1, int16_t op2)
svint64_t <b>svaddlb</b> [_n_s64](svint32_t op1, int32_t op2)
svuint16_t <b>svaddlb</b> [_n_u16](svuint8_t op1, uint8_t op2)
svuint32_t <b>svaddlb</b> [_n_u32](svuint16_t op1, uint16_t op2)
svuint64_t <b>svaddlb</b> [_n_u64](svuint32_t op1, uint32_t op2)

### 8.4.2. ADDLT: Integer addition long (top)

These functions extract the odd-indexed elements of the integer inputs and add them together, giving a result with twice the width.

#### 8.4.2.1. ADDLT (vector, vector)

Instances
svint16_t <b>svaddlt</b> [_s16](svint8_t op1, svint8_t op2)
svint32_t <b>svaddlt</b> [_s32](svint16_t op1, svint16_t op2)
svint64_t <b>svaddlt</b> [_s64](svint32_t op1, svint32_t op2)
svuint16_t <b>svaddlt</b> [_u16](svuint8_t op1, svuint8_t op2)
svuint32_t <b>svaddlt</b> [_u32](svuint16_t op1, svuint16_t op2)
svuint64_t <b>svaddlt</b> [_u64](svuint32_t op1, svuint32_t op2)

### 8.4.2.2. ADDLT (vector, scalar)

Instances	
svint16_t	<b>svaddlt</b> [_n_s16](svint8_t op1, int8_t op2)
svint32_t	<b>svaddlt</b> [_n_s32](svint16_t op1, int16_t op2)
svint64_t	<b>svaddlt</b> [_n_s64](svint32_t op1, int32_t op2)
svuint16_t	<b>svaddlt</b> [_n_u16](svuint8_t op1, uint8_t op2)
svuint32_t	<b>svaddlt</b> [_n_u32](svuint16_t op1, uint16_t op2)
svuint64_t	<b>svaddlt</b> [_n_u64](svuint32_t op1, uint32_t op2)

### 8.4.3. ADDWB: Integer addition wide (bottom)

These functions extract the even-indexed elements of the second vector, giving the same number of elements as the first vector. They then add the two inputs together.

The addition uses modular arithmetic and the result is well-defined for all inputs; unlike C, there is no undefined behavior for signed overflow.

#### 8.4.3.1. ADDWB (vector, vector)

Instances	
svint16_t	<b>svaddwb</b> [_s16](svint16_t op1, svint8_t op2)
svint32_t	<b>svaddwb</b> [_s32](svint32_t op1, svint16_t op2)
svint64_t	<b>svaddwb</b> [_s64](svint64_t op1, svint32_t op2)
svuint16_t	<b>svaddwb</b> [_u16](svuint16_t op1, svuint8_t op2)
svuint32_t	<b>svaddwb</b> [_u32](svuint32_t op1, svuint16_t op2)
svuint64_t	<b>svaddwb</b> [_u64](svuint64_t op1, svuint32_t op2)

#### 8.4.3.2. ADDWB (vector, scalar)

Instances	
svint16_t	<b>svaddwb</b> [_n_s16](svint16_t op1, int8_t op2)
svint32_t	<b>svaddwb</b> [_n_s32](svint32_t op1, int16_t op2)
svint64_t	<b>svaddwb</b> [_n_s64](svint64_t op1, int32_t op2)
svuint16_t	<b>svaddwb</b> [_n_u16](svuint16_t op1, uint8_t op2)
svuint32_t	<b>svaddwb</b> [_n_u32](svuint32_t op1, uint16_t op2)
svuint64_t	<b>svaddwb</b> [_n_u64](svuint64_t op1, uint32_t op2)

### 8.4.4. ADDWT: Integer addition wide (top)

These functions extract the odd-indexed elements of the second vector, giving the same number of elements as the first vector. They then add the two inputs together.

The addition uses modular arithmetic and the result is well-defined for all inputs; unlike C, there is no undefined behavior for signed overflow.

#### 8.4.4.1. ADDWT (vector, vector)

Instances	
svint16_t	<b>svaddwt</b> [_s16](svint16_t op1, svint8_t op2)
svint32_t	<b>svaddwt</b> [_s32](svint32_t op1, svint16_t op2)
svint64_t	<b>svaddwt</b> [_s64](svint64_t op1, svint32_t op2)
svuint16_t	<b>svaddwt</b> [_u16](svuint16_t op1, svuint8_t op2)

**Instances**

```
svuint32_t svaddwt[_u32](svuint32_t op1, svuint16_t op2)
svuint64_t svaddwt[_u64](svuint64_t op1, svuint32_t op2)
```

**8.4.4.2. ADDWT (vector, scalar)****Instances**

```
svint16_t svaddwt[_n_s16](svint16_t op1, int8_t op2)
svint32_t svaddwt[_n_s32](svint32_t op1, int16_t op2)
svint64_t svaddwt[_n_s64](svint64_t op1, int32_t op2)
svuint16_t svaddwt[_n_u16](svuint16_t op1, uint8_t op2)
svuint32_t svaddwt[_n_u32](svuint32_t op1, uint16_t op2)
svuint64_t svaddwt[_n_u64](svuint64_t op1, uint32_t op2)
```

**8.4.5. SUBLB: Integer subtraction long (bottom)**

These functions extract the even-indexed elements of the integer inputs and subtract the second from the first, giving a result with twice the width.

**8.4.5.1. SUBLB (vector, vector)****Instances**

```
svint16_t svsublb[_s16](svint8_t op1, svint8_t op2)
svint32_t svsublb[_s32](svint16_t op1, svint16_t op2)
svint64_t svsublb[_s64](svint32_t op1, svint32_t op2)
svuint16_t svsublb[_u16](svuint8_t op1, svuint8_t op2)
svuint32_t svsublb[_u32](svuint16_t op1, svuint16_t op2)
svuint64_t svsublb[_u64](svuint32_t op1, svuint32_t op2)
```

**8.4.5.2. SUBLB (vector, scalar)****Instances**

```
svint16_t svsublb[_n_s16](svint8_t op1, int8_t op2)
svint32_t svsublb[_n_s32](svint16_t op1, int16_t op2)
svint64_t svsublb[_n_s64](svint32_t op1, int32_t op2)
svuint16_t svsublb[_n_u16](svuint8_t op1, uint8_t op2)
svuint32_t svsublb[_n_u32](svuint16_t op1, uint16_t op2)
svuint64_t svsublb[_n_u64](svuint32_t op1, uint32_t op2)
```

**8.4.6. SUBLT: Integer subtraction long (top)**

These functions extract the odd-indexed elements of the integer inputs and subtract the second from the first, giving a result with twice the width.

**8.4.6.1. SUBLT (vector, vector)****Instances**

```
svint16_t svsublt[_s16](svint8_t op1, svint8_t op2)
svint32_t svsublt[_s32](svint16_t op1, svint16_t op2)
svint64_t svsublt[_s64](svint32_t op1, svint32_t op2)
svuint16_t svsublt[_u16](svuint8_t op1, svuint8_t op2)
svuint32_t svsublt[_u32](svuint16_t op1, svuint16_t op2)
svuint64_t svsublt[_u64](svuint32_t op1, svuint32_t op2)
```

### 8.4.6.2. SUBLT (vector, scalar)

Instances	
svint16_t	<b>svsublt</b> [_n_s16](svint8_t op1, int8_t op2)
svint32_t	<b>svsublt</b> [_n_s32](svint16_t op1, int16_t op2)
svint64_t	<b>svsublt</b> [_n_s64](svint32_t op1, int32_t op2)
svuint16_t	<b>svsublt</b> [_n_u16](svuint8_t op1, uint8_t op2)
svuint32_t	<b>svsublt</b> [_n_u32](svuint16_t op1, uint16_t op2)
svuint64_t	<b>svsublt</b> [_n_u64](svuint32_t op1, uint32_t op2)

### 8.4.7. SUBWB: Integer subtraction wide (bottom)

These functions extract the even-indexed elements of the second vector, giving the same number of elements as the first vector. They then subtract the second input from the first.

The subtraction uses modular arithmetic and the result is well-defined for all inputs; unlike C, there is no undefined behavior for signed overflow.

#### 8.4.7.1. SUBWB (vector, vector)

Instances	
svint16_t	<b>svsubwb</b> [_s16](svint16_t op1, svint8_t op2)
svint32_t	<b>svsubwb</b> [_s32](svint32_t op1, svint16_t op2)
svint64_t	<b>svsubwb</b> [_s64](svint64_t op1, svint32_t op2)
svuint16_t	<b>svsubwb</b> [_u16](svuint16_t op1, svuint8_t op2)
svuint32_t	<b>svsubwb</b> [_u32](svuint32_t op1, svuint16_t op2)
svuint64_t	<b>svsubwb</b> [_u64](svuint64_t op1, svuint32_t op2)

#### 8.4.7.2. SUBWB (vector, scalar)

Instances	
svint16_t	<b>svsubwb</b> [_n_s16](svint16_t op1, int8_t op2)
svint32_t	<b>svsubwb</b> [_n_s32](svint32_t op1, int16_t op2)
svint64_t	<b>svsubwb</b> [_n_s64](svint64_t op1, int32_t op2)
svuint16_t	<b>svsubwb</b> [_n_u16](svuint16_t op1, uint8_t op2)
svuint32_t	<b>svsubwb</b> [_n_u32](svuint32_t op1, uint16_t op2)
svuint64_t	<b>svsubwb</b> [_n_u64](svuint64_t op1, uint32_t op2)

### 8.4.8. SUBWT: Integer subtraction wide (top)

These functions extract the odd-indexed elements of the second vector, giving the same number of elements as the first vector. They then subtract the second input from the first.

The subtraction uses modular arithmetic and the result is well-defined for all inputs; unlike C, there is no undefined behavior for signed overflow.

#### 8.4.8.1. SUBWT (vector, vector)

Instances	
svint16_t	<b>svsubwt</b> [_s16](svint16_t op1, svint8_t op2)
svint32_t	<b>svsubwt</b> [_s32](svint32_t op1, svint16_t op2)
svint64_t	<b>svsubwt</b> [_s64](svint64_t op1, svint32_t op2)
svuint16_t	<b>svsubwt</b> [_u16](svuint16_t op1, svuint8_t op2)

Instances
svuint32_t <b>svsubwt</b> [_u32](svuint32_t op1, svuint16_t op2)
svuint64_t <b>svsubwt</b> [_u64](svuint64_t op1, svuint32_t op2)

#### 8.4.8.2. SUBWT (vector, scalar)

Instances
svint16_t <b>svsubwt</b> [_n_s16](svint16_t op1, int8_t op2)
svint32_t <b>svsubwt</b> [_n_s32](svint32_t op1, int16_t op2)
svint64_t <b>svsubwt</b> [_n_s64](svint64_t op1, int32_t op2)
svuint16_t <b>svsubwt</b> [_n_u16](svuint16_t op1, uint8_t op2)
svuint32_t <b>svsubwt</b> [_n_u32](svuint32_t op1, uint16_t op2)
svuint64_t <b>svsubwt</b> [_n_u64](svuint64_t op1, uint32_t op2)

### 8.4.9. ABDLB: Integer absolute difference long (bottom)

These functions extract the even-indexed elements of the integer inputs and compute their absolute difference, giving a result with twice the width.

#### 8.4.9.1. ABDLB (vector, vector)

Instances
svint16_t <b>svabdlb</b> [_s16](svint8_t op1, svint8_t op2)
svint32_t <b>svabdlb</b> [_s32](svint16_t op1, svint16_t op2)
svint64_t <b>svabdlb</b> [_s64](svint32_t op1, svint32_t op2)
svuint16_t <b>svabdlb</b> [_u16](svuint8_t op1, svuint8_t op2)
svuint32_t <b>svabdlb</b> [_u32](svuint16_t op1, svuint16_t op2)
svuint64_t <b>svabdlb</b> [_u64](svuint32_t op1, svuint32_t op2)

#### 8.4.9.2. ABDLB (vector, scalar)

Instances
svint16_t <b>svabdlb</b> [_n_s16](svint8_t op1, int8_t op2)
svint32_t <b>svabdlb</b> [_n_s32](svint16_t op1, int16_t op2)
svint64_t <b>svabdlb</b> [_n_s64](svint32_t op1, int32_t op2)
svuint16_t <b>svabdlb</b> [_n_u16](svuint8_t op1, uint8_t op2)
svuint32_t <b>svabdlb</b> [_n_u32](svuint16_t op1, uint16_t op2)
svuint64_t <b>svabdlb</b> [_n_u64](svuint32_t op1, uint32_t op2)

### 8.4.10. ABDLT: Integer absolute difference long (top)

These functions extract the odd-indexed elements of the integer inputs and compute their absolute difference, giving a result with twice the width.

#### 8.4.10.1. ABDLT (vector, vector)

Instances
svint16_t <b>svabdlb</b> [_s16](svint8_t op1, svint8_t op2)
svint32_t <b>svabdlb</b> [_s32](svint16_t op1, svint16_t op2)
svint64_t <b>svabdlb</b> [_s64](svint32_t op1, svint32_t op2)
svuint16_t <b>svabdlb</b> [_u16](svuint8_t op1, svuint8_t op2)
svuint32_t <b>svabdlb</b> [_u32](svuint16_t op1, svuint16_t op2)
svuint64_t <b>svabdlb</b> [_u64](svuint32_t op1, svuint32_t op2)

### 8.4.10.2. ABDLT (vector, scalar)

Instances
svint16_t <b>svabdl</b> [_n_s16](svint8_t op1, int8_t op2)
svint32_t <b>svabdl</b> [_n_s32](svint16_t op1, int16_t op2)
svint64_t <b>svabdl</b> [_n_s64](svint32_t op1, int32_t op2)
svuint16_t <b>svabdl</b> [_n_u16](svuint8_t op1, uint8_t op2)
svuint32_t <b>svabdl</b> [_n_u32](svuint16_t op1, uint16_t op2)
svuint64_t <b>svabdl</b> [_n_u64](svuint32_t op1, uint32_t op2)

### 8.4.11. ABALB: Integer addition of absolute difference long (bottom)

These functions extract the even-indexed elements of the second and third vectors, giving the same number of elements as the first. They then compute the absolute difference between the second and third inputs and add the result to the first input.

The addition uses modular arithmetic and the result is well-defined for all inputs; unlike C, there is no undefined behavior for signed overflow.

#### 8.4.11.1. ABALB (vector, vector, vector)

Instances
svint16_t <b>svabalb</b> [_s16](svint16_t op1, svint8_t op2, svint8_t op3)
svint32_t <b>svabalb</b> [_s32](svint32_t op1, svint16_t op2, svint16_t op3)
svint64_t <b>svabalb</b> [_s64](svint64_t op1, svint32_t op2, svint32_t op3)
svuint16_t <b>svabalb</b> [_u16](svuint16_t op1, svuint8_t op2, svuint8_t op3)
svuint32_t <b>svabalb</b> [_u32](svuint32_t op1, svuint16_t op2, svuint16_t op3)
svuint64_t <b>svabalb</b> [_u64](svuint64_t op1, svuint32_t op2, svuint32_t op3)

#### 8.4.11.2. ABALB (vector, vector, scalar)

Instances
svint16_t <b>svabalb</b> [_n_s16](svint16_t op1, svint8_t op2, int8_t op3)
svint32_t <b>svabalb</b> [_n_s32](svint32_t op1, svint16_t op2, int16_t op3)
svint64_t <b>svabalb</b> [_n_s64](svint64_t op1, svint32_t op2, int32_t op3)
svuint16_t <b>svabalb</b> [_n_u16](svuint16_t op1, svuint8_t op2, uint8_t op3)
svuint32_t <b>svabalb</b> [_n_u32](svuint32_t op1, svuint16_t op2, uint16_t op3)
svuint64_t <b>svabalb</b> [_n_u64](svuint64_t op1, svuint32_t op2, uint32_t op3)

### 8.4.12. ABALT: Integer addition of absolute difference long (top)

These functions extract the odd-indexed elements of the second and third inputs, giving the same number of elements as the first. They then compute the absolute difference between the second and third inputs and add the result to the first input.

The addition uses modular arithmetic and the result is well-defined for all inputs; unlike C, there is no undefined behavior for signed overflow.

#### 8.4.12.1. ABALT (vector, vector, vector)

Instances
svint16_t <b>svabalt</b> [_s16](svint16_t op1, svint8_t op2, svint8_t op3)
svint32_t <b>svabalt</b> [_s32](svint32_t op1, svint16_t op2, svint16_t op3)

Instances
svint64_t <b>svabalt</b> [_s64](svint64_t op1, svint32_t op2, svint32_t op3)
svuint16_t <b>svabalt</b> [_u16](svuint16_t op1, svuint8_t op2, svuint8_t op3)
svuint32_t <b>svabalt</b> [_u32](svuint32_t op1, svuint16_t op2, svuint16_t op3)
svuint64_t <b>svabalt</b> [_u64](svuint64_t op1, svuint32_t op2, svuint32_t op3)

#### 8.4.12.2. ABALT (vector, vector, scalar)

Instances
svint16_t <b>svabalt</b> [_n_s16](svint16_t op1, svint8_t op2, int8_t op3)
svint32_t <b>svabalt</b> [_n_s32](svint32_t op1, svint16_t op2, int16_t op3)
svint64_t <b>svabalt</b> [_n_s64](svint64_t op1, svint32_t op2, int32_t op3)
svuint16_t <b>svabalt</b> [_n_u16](svuint16_t op1, svuint8_t op2, uint8_t op3)
svuint32_t <b>svabalt</b> [_n_u32](svuint32_t op1, svuint16_t op2, uint16_t op3)
svuint64_t <b>svabalt</b> [_n_u64](svuint64_t op1, svuint32_t op2, uint32_t op3)

### 8.4.13. MULLB: Integer multiplication long (bottom)

These functions extract the even-indexed elements of the integer inputs and multiply them, giving a result with twice the width.

The `_lane` forms of the functions take one element in each 128-bit quadword of the second input and replicate it to fill the rest of the quadword. The `imm_index` parameter specifies the element within each quadword that the functions should replicate. This index must be an integer constant expression in the range [0, 128/*N*], where *N* is the number of bits in each element.

#### 8.4.13.1. MULLB (vector, vector)

Instances
svint16_t <b>svmullb</b> [_s16](svint8_t op1, svint8_t op2)
svint32_t <b>svmullb</b> [_s32](svint16_t op1, svint16_t op2)
svint64_t <b>svmullb</b> [_s64](svint32_t op1, svint32_t op2)
svuint16_t <b>svmullb</b> [_u16](svuint8_t op1, svuint8_t op2)
svuint32_t <b>svmullb</b> [_u32](svuint16_t op1, svuint16_t op2)
svuint64_t <b>svmullb</b> [_u64](svuint32_t op1, svuint32_t op2)

#### 8.4.13.2. MULLB (vector, scalar)

Instances
svint16_t <b>svmullb</b> [_n_s16](svint8_t op1, int8_t op2)
svint32_t <b>svmullb</b> [_n_s32](svint16_t op1, int16_t op2)
svint64_t <b>svmullb</b> [_n_s64](svint32_t op1, int32_t op2)
svuint16_t <b>svmullb</b> [_n_u16](svuint8_t op1, uint8_t op2)
svuint32_t <b>svmullb</b> [_n_u32](svuint16_t op1, uint16_t op2)
svuint64_t <b>svmullb</b> [_n_u64](svuint32_t op1, uint32_t op2)

#### 8.4.13.3. MULLB (vector, vector, lane)

Instances
svint32_t <b>svmullb_lane</b> [_s32](svint16_t op1, svint16_t op2, uint64_t imm_index)
svint64_t <b>svmullb_lane</b> [_s64](svint32_t op1, svint32_t op2, uint64_t imm_index)



**Instances**

```
svuint32_t svmultlb_lane[_u32](svuint16_t op1, svuint16_t op2,
                                uint64_t imm_index)
svuint64_t svmultlb_lane[_u64](svuint32_t op1, svuint32_t op2,
                                uint64_t imm_index)
```

**8.4.14. MULLT: Integer multiplication long (top)**

These functions extract the odd-indexed elements of the integer inputs and multiply them, giving a result with twice the width.

The `_lane` forms of the functions take one element in each 128-bit quadword of the second input and replicate it to fill the rest of the quadword. The `imm_index` parameter specifies the element within each quadword that the functions should replicate. This index must be an integer constant expression in the range  $[0, 128/N]$ , where  $N$  is the number of bits in each element.

**8.4.14.1. MULLT (vector, vector)****Instances**

```
svint16_t svmultlt[_s16](svint8_t op1, svint8_t op2)
svint32_t svmultlt[_s32](svint16_t op1, svint16_t op2)
svint64_t svmultlt[_s64](svint32_t op1, svint32_t op2)
svuint16_t svmultlt[_u16](svuint8_t op1, svuint8_t op2)
svuint32_t svmultlt[_u32](svuint16_t op1, svuint16_t op2)
svuint64_t svmultlt[_u64](svuint32_t op1, svuint32_t op2)
```

**8.4.14.2. MULLT (vector, scalar)****Instances**

```
svint16_t svmultlt[_n_s16](svint8_t op1, int8_t op2)
svint32_t svmultlt[_n_s32](svint16_t op1, int16_t op2)
svint64_t svmultlt[_n_s64](svint32_t op1, int32_t op2)
svuint16_t svmultlt[_n_u16](svuint8_t op1, uint8_t op2)
svuint32_t svmultlt[_n_u32](svuint16_t op1, uint16_t op2)
svuint64_t svmultlt[_n_u64](svuint32_t op1, uint32_t op2)
```

**8.4.14.3. MULLT (vector, vector, lane)****Instances**

```
svint32_t svmultlt_lane[_s32](svint16_t op1, svint16_t op2,
                                uint64_t imm_index)
svint64_t svmultlt_lane[_s64](svint32_t op1, svint32_t op2,
                                uint64_t imm_index)
svuint32_t svmultlt_lane[_u32](svuint16_t op1, svuint16_t op2,
                                uint64_t imm_index)
svuint64_t svmultlt_lane[_u64](svuint32_t op1, svuint32_t op2,
                                uint64_t imm_index)
```

**8.4.15. MLALB: Integer addition of product long (bottom)**

These functions extract the even-indexed elements of the second and third vectors, giving the same number of elements as the first. They then multiply the second and third inputs, giving a result with twice the width, and add the product to the first input.

The addition uses modular arithmetic and the result is well-defined for all inputs; unlike C, there is no undefined behavior for signed overflow.

The `_lane` forms of the functions take one element in each 128-bit quadword of the third input and replicate it to fill the rest of the quadword. The `imm_index` parameter specifies the element within each quadword that the functions should replicate. This index must be an integer constant expression in the range  $[0, 128/N)$ , where  $N$  is the number of bits in each element.

#### 8.4.15.1. MLALB (vector, vector, vector)

Instances
<code>svint16_t svmlalb[_s16](svint16_t op1, svint8_t op2, svint8_t op3)</code> <code>svint32_t svmlalb[_s32](svint32_t op1, svint16_t op2, svint16_t op3)</code> <code>svint64_t svmlalb[_s64](svint64_t op1, svint32_t op2, svint32_t op3)</code> <code>svuint16_t svmlalb[_u16](svuint16_t op1, svuint8_t op2, svuint8_t op3)</code> <code>svuint32_t svmlalb[_u32](svuint32_t op1, svuint16_t op2, svuint16_t op3)</code> <code>svuint64_t svmlalb[_u64](svuint64_t op1, svuint32_t op2, svuint32_t op3)</code>

#### 8.4.15.2. MLALB (vector, vector, scalar)

Instances
<code>svint16_t svmlalb[_n_s16](svint16_t op1, svint8_t op2, int8_t op3)</code> <code>svint32_t svmlalb[_n_s32](svint32_t op1, svint16_t op2, int16_t op3)</code> <code>svint64_t svmlalb[_n_s64](svint64_t op1, svint32_t op2, int32_t op3)</code> <code>svuint16_t svmlalb[_n_u16](svuint16_t op1, svuint8_t op2, uint8_t op3)</code> <code>svuint32_t svmlalb[_n_u32](svuint32_t op1, svuint16_t op2, uint16_t op3)</code> <code>svuint64_t svmlalb[_n_u64](svuint64_t op1, svuint32_t op2, uint32_t op3)</code>

#### 8.4.15.3. MLALB (vector, vector, vector, lane)

Instances
<code>svint32_t svmlalb_lane[_s32](svint32_t op1, svint16_t op2, svint16_t op3, uint64_t imm_index)</code> <code>svint64_t svmlalb_lane[_s64](svint64_t op1, svint32_t op2, svint32_t op3, uint64_t imm_index)</code> <code>svuint32_t svmlalb_lane[_u32](svuint32_t op1, svuint16_t op2, svuint16_t op3, uint64_t imm_index)</code> <code>svuint64_t svmlalb_lane[_u64](svuint64_t op1, svuint32_t op2, svuint32_t op3, uint64_t imm_index)</code>

### 8.4.16. MLALT: Integer addition of product long (top)

These functions extract the odd-indexed elements of the second and third vectors, giving the same number of elements as the first. They then multiply the second and third inputs, giving a result with twice the width, and add the product to the first input.

The addition uses modular arithmetic and the result is well-defined for all inputs; unlike C, there is no undefined behavior for signed overflow.

The `_lane` forms of the functions take one element in each 128-bit quadword of the third input and replicate it to fill the rest of the quadword. The `imm_index` parameter specifies the element within each quadword that the functions should replicate. This index must be an integer constant expression in the range  $[0, 128/N)$ , where  $N$  is the number of bits in each element.

### 8.4.16.1. MLALT (vector, vector, vector)

Instances	
svint16_t	<b>svmlalt</b> [_s16](svint16_t op1, svint8_t op2, svint8_t op3)
svint32_t	<b>svmlalt</b> [_s32](svint32_t op1, svint16_t op2, svint16_t op3)
svint64_t	<b>svmlalt</b> [_s64](svint64_t op1, svint32_t op2, svint32_t op3)
svuint16_t	<b>svmlalt</b> [_u16](svuint16_t op1, svuint8_t op2, svuint8_t op3)
svuint32_t	<b>svmlalt</b> [_u32](svuint32_t op1, svuint16_t op2, svuint16_t op3)
svuint64_t	<b>svmlalt</b> [_u64](svuint64_t op1, svuint32_t op2, svuint32_t op3)

### 8.4.16.2. MLALT (vector, vector, scalar)

Instances	
svint16_t	<b>svmlalt</b> [_n_s16](svint16_t op1, svint8_t op2, int8_t op3)
svint32_t	<b>svmlalt</b> [_n_s32](svint32_t op1, svint16_t op2, int16_t op3)
svint64_t	<b>svmlalt</b> [_n_s64](svint64_t op1, svint32_t op2, int32_t op3)
svuint16_t	<b>svmlalt</b> [_n_u16](svuint16_t op1, svuint8_t op2, uint8_t op3)
svuint32_t	<b>svmlalt</b> [_n_u32](svuint32_t op1, svuint16_t op2, uint16_t op3)
svuint64_t	<b>svmlalt</b> [_n_u64](svuint64_t op1, svuint32_t op2, uint32_t op3)

### 8.4.16.3. MLALT (vector, vector, vector, lane)

Instances	
svint32_t	<b>svmlalt_lane</b> [_s32](svint32_t op1, svint16_t op2, svint16_t op3, uint64_t imm_index)
svint64_t	<b>svmlalt_lane</b> [_s64](svint64_t op1, svint32_t op2, svint32_t op3, uint64_t imm_index)
svuint32_t	<b>svmlalt_lane</b> [_u32](svuint32_t op1, svuint16_t op2, svuint16_t op3, uint64_t imm_index)
svuint64_t	<b>svmlalt_lane</b> [_u64](svuint64_t op1, svuint32_t op2, svuint32_t op3, uint64_t imm_index)

## 8.4.17. MLSLB: Integer subtraction of product long (bottom)

These functions extract the even-indexed elements of the second and third vectors, giving the same number of elements as the first. They then multiply the second and third inputs, giving a result with twice the width, and subtract the product from the first input.

The subtraction uses modular arithmetic and the result is well-defined for all inputs; unlike C, there is no undefined behavior for signed overflow.

The `_lane` forms of the functions take one element in each 128-bit quadword of the third input and replicate it to fill the rest of the quadword. The `imm_index` parameter specifies the element within each quadword that the functions should replicate. This index must be an integer constant expression in the range  $[0, 128/N]$ , where  $N$  is the number of bits in each element.

### 8.4.17.1. MLSLB (vector, vector, vector)

Instances	
svint16_t	<b>svmlslb</b> [_s16](svint16_t op1, svint8_t op2, svint8_t op3)
svint32_t	<b>svmlslb</b> [_s32](svint32_t op1, svint16_t op2, svint16_t op3)
svint64_t	<b>svmlslb</b> [_s64](svint64_t op1, svint32_t op2, svint32_t op3)
svuint16_t	<b>svmlslb</b> [_u16](svuint16_t op1, svuint8_t op2, svuint8_t op3)
svuint32_t	<b>svmlslb</b> [_u32](svuint32_t op1, svuint16_t op2, svuint16_t op3)
svuint64_t	<b>svmlslb</b> [_u64](svuint64_t op1, svuint32_t op2, svuint32_t op3)

### 8.4.17.2. MLSLB (vector, vector, scalar)

Instances	
<code>svint16_t</code>	<code>svmlslb[_n_s16](svint16_t op1, svint8_t op2, int8_t op3)</code>
<code>svint32_t</code>	<code>svmlslb[_n_s32](svint32_t op1, svint16_t op2, int16_t op3)</code>
<code>svint64_t</code>	<code>svmlslb[_n_s64](svint64_t op1, svint32_t op2, int32_t op3)</code>
<code>svuint16_t</code>	<code>svmlslb[_n_u16](svuint16_t op1, svuint8_t op2, uint8_t op3)</code>
<code>svuint32_t</code>	<code>svmlslb[_n_u32](svuint32_t op1, svuint16_t op2, uint16_t op3)</code>
<code>svuint64_t</code>	<code>svmlslb[_n_u64](svuint64_t op1, svuint32_t op2, uint32_t op3)</code>

### 8.4.17.3. MLSLB (vector, vector, vector, lane)

Instances	
<code>svint32_t</code>	<code>svmlslb_lane[_s32](svint32_t op1, svint16_t op2, svint16_t op3, uint64_t imm_index)</code>
<code>svint64_t</code>	<code>svmlslb_lane[_s64](svint64_t op1, svint32_t op2, svint32_t op3, uint64_t imm_index)</code>
<code>svuint32_t</code>	<code>svmlslb_lane[_u32](svuint32_t op1, svuint16_t op2, svuint16_t op3, uint64_t imm_index)</code>
<code>svuint64_t</code>	<code>svmlslb_lane[_u64](svuint64_t op1, svuint32_t op2, svuint32_t op3, uint64_t imm_index)</code>

## 8.4.18. MLSLT: Integer subtraction of product long (top)

These functions extract the odd-indexed elements of the second and third vectors, giving the same number of elements as the first. They then multiply the second and third inputs, giving a result with twice the width, and subtract the product from the first input.

The subtraction uses modular arithmetic and the result is well-defined for all inputs; unlike C, there is no undefined behavior for signed overflow.

The `_lane` forms of the functions take one element in each 128-bit quadword of the third input and replicate it to fill the rest of the quadword. The `imm_index` parameter specifies the element within each quadword that the functions should replicate. This index must be an integer constant expression in the range  $[0, 128/N)$ , where  $N$  is the number of bits in each element.

### 8.4.18.1. MLSLT (vector, vector, vector)

Instances	
<code>svint16_t</code>	<code>svmlslt[_s16](svint16_t op1, svint8_t op2, svint8_t op3)</code>
<code>svint32_t</code>	<code>svmlslt[_s32](svint32_t op1, svint16_t op2, svint16_t op3)</code>
<code>svint64_t</code>	<code>svmlslt[_s64](svint64_t op1, svint32_t op2, svint32_t op3)</code>
<code>svuint16_t</code>	<code>svmlslt[_u16](svuint16_t op1, svuint8_t op2, svuint8_t op3)</code>
<code>svuint32_t</code>	<code>svmlslt[_u32](svuint32_t op1, svuint16_t op2, svuint16_t op3)</code>
<code>svuint64_t</code>	<code>svmlslt[_u64](svuint64_t op1, svuint32_t op2, svuint32_t op3)</code>

### 8.4.18.2. MLSLT (vector, vector, scalar)

Instances	
<code>svint16_t</code>	<code>svmlslt[_n_s16](svint16_t op1, svint8_t op2, int8_t op3)</code>
<code>svint32_t</code>	<code>svmlslt[_n_s32](svint32_t op1, svint16_t op2, int16_t op3)</code>
<code>svint64_t</code>	<code>svmlslt[_n_s64](svint64_t op1, svint32_t op2, int32_t op3)</code>
<code>svuint16_t</code>	<code>svmlslt[_n_u16](svuint16_t op1, svuint8_t op2, uint8_t op3)</code>
<code>svuint32_t</code>	<code>svmlslt[_n_u32](svuint32_t op1, svuint16_t op2, uint16_t op3)</code>

**Instances**

```
svuint64_t svmlslt[_n_u64](svuint64_t op1, svuint32_t op2, uint32_t op3)
```

**8.4.18.3. MSLT (vector, vector, vector, lane)****Instances**

```
svint32_t svmlslt_lane[_s32](svint32_t op1, svint16_t op2, svint16_t op3,
                               uint64_t imm_index)
svint64_t svmlslt_lane[_s64](svint64_t op1, svint32_t op2, svint32_t op3,
                               uint64_t imm_index)
svuint32_t svmlslt_lane[_u32](svuint32_t op1, svuint16_t op2,
                               svuint16_t op3, uint64_t imm_index)
svuint64_t svmlslt_lane[_u64](svuint64_t op1, svuint32_t op2,
                               svuint32_t op3, uint64_t imm_index)
```

**8.4.19. MOVLB: Move long (bottom)**

These functions extract the even-indexed elements of the integer input and extend them, giving a result with twice the width. This is equivalent to using [Section 8.4.27, “SHLLB: Shift left long by immediate \(bottom\)”](#) with a shift of zero.

**8.4.19.1. MOVLB (vector)****Instances**

```
svint16_t svmovlb[_s16](svint8_t op)
svint32_t svmovlb[_s32](svint16_t op)
svint64_t svmovlb[_s64](svint32_t op)
svuint16_t svmovlb[_u16](svuint8_t op)
svuint32_t svmovlb[_u32](svuint16_t op)
svuint64_t svmovlb[_u64](svuint32_t op)
```

**8.4.20. MOVLT: Move long (top)**

These functions extract the odd-indexed elements of the integer input and extend them, giving a result with twice the width. This is equivalent to using [Section 8.4.28, “SHLLT: Shift left long by immediate \(top\)”](#) with a shift of zero.

**8.4.20.1. MOVLT (vector)****Instances**

```
svint16_t svmovlt[_s16](svint8_t op)
svint32_t svmovlt[_s32](svint16_t op)
svint64_t svmovlt[_s64](svint32_t op)
svuint16_t svmovlt[_u16](svuint8_t op)
svuint32_t svmovlt[_u32](svuint16_t op)
svuint64_t svmovlt[_u64](svuint32_t op)
```

**8.4.21. QDMULLB: Saturating integer doubled product long (bottom)**

These functions extract the even-indexed elements of the integer inputs and multiply them, giving a result with twice the width. They then double the product and saturate the result to the range of the return type.

The `_lane` forms of the functions take one element in each 128-bit quadword of the second input and replicate it to fill the rest of the quadword. The `imm_index` parameter specifies the element within each

quadword that the functions should replicate. This index must be an integer constant expression in the range  $[0, 128/N]$ , where  $N$  is the number of bits in each element.

#### 8.4.21.1. QDMULLB (vector, vector)

Instances
svint16_t <b>svqdmullb</b> [_s16](svint8_t op1, svint8_t op2)
svint32_t <b>svqdmullb</b> [_s32](svint16_t op1, svint16_t op2)
svint64_t <b>svqdmullb</b> [_s64](svint32_t op1, svint32_t op2)

#### 8.4.21.2. QDMULLB (vector, scalar)

Instances
svint16_t <b>svqdmullb</b> [_n_s16](svint8_t op1, int8_t op2)
svint32_t <b>svqdmullb</b> [_n_s32](svint16_t op1, int16_t op2)
svint64_t <b>svqdmullb</b> [_n_s64](svint32_t op1, int32_t op2)

#### 8.4.21.3. QDMULLB (vector, vector, lane)

Instances
svint32_t <b>svqdmullb_lane</b> [_s32](svint16_t op1, svint16_t op2, uint64_t imm_index)
svint64_t <b>svqdmullb_lane</b> [_s64](svint32_t op1, svint32_t op2, uint64_t imm_index)

### 8.4.22. QDMULLT: Saturating integer doubled product long (top)

These functions extract the odd-indexed elements of the integer inputs and multiply them, giving a result with twice the width. They then double the product and saturate the result to the range of the return type.

The `_lane` forms of the functions take one element in each 128-bit quadword of the second input and replicate it to fill the rest of the quadword. The `imm_index` parameter specifies the element within each quadword that the functions should replicate. This index must be an integer constant expression in the range  $[0, 128/N]$ , where  $N$  is the number of bits in each element.

#### 8.4.22.1. QDMULLT (vector, vector)

Instances
svint16_t <b>svqdmullt</b> [_s16](svint8_t op1, svint8_t op2)
svint32_t <b>svqdmullt</b> [_s32](svint16_t op1, svint16_t op2)
svint64_t <b>svqdmullt</b> [_s64](svint32_t op1, svint32_t op2)

#### 8.4.22.2. QDMULLT (vector, scalar)

Instances
svint16_t <b>svqdmullt</b> [_n_s16](svint8_t op1, int8_t op2)
svint32_t <b>svqdmullt</b> [_n_s32](svint16_t op1, int16_t op2)
svint64_t <b>svqdmullt</b> [_n_s64](svint32_t op1, int32_t op2)

#### 8.4.22.3. QDMULLT (vector, vector, lane)

Instances
svint32_t <b>svqdmullt_lane</b> [_s32](svint16_t op1, svint16_t op2,

**Instances**

```

uint64_t imm_index)
svint64_t svqdmullt_lane[_s64](svint32_t op1, svint32_t op2,
                                uint64_t imm_index)

```

## 8.4.23. QDMLALB: Saturating integer addition of doubled product long (bottom)

These functions extract the even-indexed elements of the second and third vectors, giving the same number of elements as the first. They then multiply the second and third inputs, giving a result that is twice the width, then double the product and add the doubled product to the first input. The doubling and addition use saturating arithmetic; if the result of the operation is outside the range of the first input's type, they clamp it to the nearest in-range value.

The `_lane` forms of the functions take one element in each 128-bit quadword of the third input and replicate it to fill the rest of the quadword. The `imm_index` parameter specifies the element within each quadword that the functions should replicate. This index must be an integer constant expression in the range  $[0, 128/N]$ , where  $N$  is the number of bits in each element.

### 8.4.23.1. QDMLALB (vector, vector, vector)

**Instances**

```

svint16_t svqdmalalb[_s16](svint16_t op1, svint8_t op2, svint8_t op3)
svint32_t svqdmalalb[_s32](svint32_t op1, svint16_t op2, svint16_t op3)
svint64_t svqdmalalb[_s64](svint64_t op1, svint32_t op2, svint32_t op3)

```

### 8.4.23.2. QDMLALB (vector, vector, scalar)

**Instances**

```

svint16_t svqdmalalb[_n_s16](svint16_t op1, svint8_t op2, int8_t op3)
svint32_t svqdmalalb[_n_s32](svint32_t op1, svint16_t op2, int16_t op3)
svint64_t svqdmalalb[_n_s64](svint64_t op1, svint32_t op2, int32_t op3)

```

### 8.4.23.3. QDMLALB (vector, vector, vector, lane)

**Instances**

```

svint32_t svqdmalalb_lane[_s32](svint32_t op1, svint16_t op2, svint16_t op3,
                                uint64_t imm_index)
svint64_t svqdmalalb_lane[_s64](svint64_t op1, svint32_t op2, svint32_t op3,
                                uint64_t imm_index)

```

## 8.4.24. QDMLALT: Saturating integer addition of doubled product long (top)

These functions extract the odd-indexed elements of the second and third vectors, giving the same number of elements as the first. They then multiply the second and third inputs, giving a result that is twice the width, then double the product and add the doubled product to the first input. The doubling and addition use saturating arithmetic; if the result of the operation is outside the range of the first input's type, they clamp it to the nearest in-range value.

The `_lane` forms of the functions take one element in each 128-bit quadword of the third input and replicate it to fill the rest of the quadword. The `imm_index` parameter specifies the element within each

quadword that the functions should replicate. This index must be an integer constant expression in the range  $[0, 128/N]$ , where  $N$  is the number of bits in each element.

#### 8.4.24.1. QDMLALT (vector, vector, vector)

Instances	
<code>svint16_t</code>	<code>svqdmlalt[_s16](svint16_t op1, svint8_t op2, svint8_t op3)</code>
<code>svint32_t</code>	<code>svqdmlalt[_s32](svint32_t op1, svint16_t op2, svint16_t op3)</code>
<code>svint64_t</code>	<code>svqdmlalt[_s64](svint64_t op1, svint32_t op2, svint32_t op3)</code>

#### 8.4.24.2. QDMLALT (vector, vector, scalar)

Instances	
<code>svint16_t</code>	<code>svqdmlalt[_n_s16](svint16_t op1, svint8_t op2, int8_t op3)</code>
<code>svint32_t</code>	<code>svqdmlalt[_n_s32](svint32_t op1, svint16_t op2, int16_t op3)</code>
<code>svint64_t</code>	<code>svqdmlalt[_n_s64](svint64_t op1, svint32_t op2, int32_t op3)</code>

#### 8.4.24.3. QDMLALT (vector, vector, vector, lane)

Instances	
<code>svint32_t</code>	<code>svqdmlalt_lane[_s32](svint32_t op1, svint16_t op2, svint16_t op3, uint64_t imm_index)</code>
<code>svint64_t</code>	<code>svqdmlalt_lane[_s64](svint64_t op1, svint32_t op2, svint32_t op3, uint64_t imm_index)</code>

### 8.4.25. QDMLSLB: Saturating integer subtraction of doubled product long (bottom)

These functions extract the even-indexed elements of the second and third vectors, giving the same number of elements as the first. They then multiply the second and third inputs, giving a result that is twice the width, then double the product and subtract the doubled product from the first input. The doubling and subtraction use saturating arithmetic; if the result of the operation is outside the range of the first input's type, they clamp it to the nearest in-range value.

The `_lane` forms of the functions take one element in each 128-bit quadword of the third input and replicate it to fill the rest of the quadword. The `imm_index` parameter specifies the element within each quadword that the functions should replicate. This index must be an integer constant expression in the range  $[0, 128/N]$ , where  $N$  is the number of bits in each element.

#### 8.4.25.1. QDMLSLB (vector, vector, vector)

Instances	
<code>svint16_t</code>	<code>svqdmlslb[_s16](svint16_t op1, svint8_t op2, svint8_t op3)</code>
<code>svint32_t</code>	<code>svqdmlslb[_s32](svint32_t op1, svint16_t op2, svint16_t op3)</code>
<code>svint64_t</code>	<code>svqdmlslb[_s64](svint64_t op1, svint32_t op2, svint32_t op3)</code>

#### 8.4.25.2. QDMLSLB (vector, vector, scalar)

Instances	
<code>svint16_t</code>	<code>svqdmlslb[_n_s16](svint16_t op1, svint8_t op2, int8_t op3)</code>
<code>svint32_t</code>	<code>svqdmlslb[_n_s32](svint32_t op1, svint16_t op2, int16_t op3)</code>



**Instances**

```
svint64_t svqdmllslb[_n_s64](svint64_t op1, svint32_t op2, int32_t op3)
```

**8.4.25.3. QDMLSLB (vector, vector, vector, lane)****Instances**

```
svint32_t svqdmllslb_lane[_s32](svint32_t op1, svint16_t op2, svint16_t op3,
                                uint64_t imm_index)
svint64_t svqdmllslb_lane[_s64](svint64_t op1, svint32_t op2, svint32_t op3,
                                uint64_t imm_index)
```

**8.4.26. QDMLSLT: Saturating integer subtraction of doubled product long (top)**

These functions extract the odd-indexed elements of the second and third vectors, giving the same number of elements as the first. They then multiply the second and third inputs, giving a result that is twice the width, then double the product and subtract the doubled product from the first input. The doubling and subtraction use saturating arithmetic; if the result of the operation is outside the range of the first input's type, they clamp it to the nearest in-range value.

The `_lane` forms of the functions take one element in each 128-bit quadword of the third input and replicate it to fill the rest of the quadword. The `imm_index` parameter specifies the element within each quadword that the functions should replicate. This index must be an integer constant expression in the range  $[0, 128/N]$ , where  $N$  is the number of bits in each element.

**8.4.26.1. QDMLSLT (vector, vector, vector)****Instances**

```
svint16_t svqdmllslt[_s16](svint16_t op1, svint8_t op2, svint8_t op3)
svint32_t svqdmllslt[_s32](svint32_t op1, svint16_t op2, svint16_t op3)
svint64_t svqdmllslt[_s64](svint64_t op1, svint32_t op2, svint32_t op3)
```

**8.4.26.2. QDMLSLT (vector, vector, scalar)****Instances**

```
svint16_t svqdmllslt[_n_s16](svint16_t op1, svint8_t op2, int8_t op3)
svint32_t svqdmllslt[_n_s32](svint32_t op1, svint16_t op2, int16_t op3)
svint64_t svqdmllslt[_n_s64](svint64_t op1, svint32_t op2, int32_t op3)
```

**8.4.26.3. QDMLSLT (vector, vector, vector, lane)****Instances**

```
svint32_t svqdmllslt_lane[_s32](svint32_t op1, svint16_t op2, svint16_t op3,
                                uint64_t imm_index)
svint64_t svqdmllslt_lane[_s64](svint64_t op1, svint32_t op2, svint32_t op3,
                                uint64_t imm_index)
```

**8.4.27. SHLLB: Shift left long by immediate (bottom)**

These functions extract the even-indexed elements of the first integer input and multiply them by two to the power of the second integer input, giving a result with twice the width. If the first input has  $N$  bits, the second input must be an integer constant expression in the range  $[0, N]$ .

### 8.4.27.1. SHLLB (vector, immediate)

Instances
svint16_t <b>svshllb</b> [_n_s16](svint8_t op1, uint64_t imm2)
svint32_t <b>svshllb</b> [_n_s32](svint16_t op1, uint64_t imm2)
svint64_t <b>svshllb</b> [_n_s64](svint32_t op1, uint64_t imm2)
svuint16_t <b>svshllb</b> [_n_u16](svuint8_t op1, uint64_t imm2)
svuint32_t <b>svshllb</b> [_n_u32](svuint16_t op1, uint64_t imm2)
svuint64_t <b>svshllb</b> [_n_u64](svuint32_t op1, uint64_t imm2)

### 8.4.28. SHLLT: Shift left long by immediate (top)

These functions extract the odd-indexed elements of the first integer input and multiply them by two to the power of the second integer input, giving a result with twice the width. If the first input has  $N$  bits, the second input must be an integer constant expression in the range  $[0, N)$ .

#### 8.4.28.1. SHLLT (vector, immediate)

Instances
svint16_t <b>svshllt</b> [_n_s16](svint8_t op1, uint64_t imm2)
svint32_t <b>svshllt</b> [_n_s32](svint16_t op1, uint64_t imm2)
svint64_t <b>svshllt</b> [_n_s64](svint32_t op1, uint64_t imm2)
svuint16_t <b>svshllt</b> [_n_u16](svuint8_t op1, uint64_t imm2)
svuint32_t <b>svshllt</b> [_n_u32](svuint16_t op1, uint64_t imm2)
svuint64_t <b>svshllt</b> [_n_u64](svuint32_t op1, uint64_t imm2)

## 8.5. Narrowing DSP operations

### 8.5.1. ADDHNB: Integer addition, narrowing to high half (bottom)

If the elements of the return value have  $N$  bits, these functions add the two integer inputs, divide the result by  $2^N$ , rounding towards  $-\infty$ , and place the quotient in the even-indexed elements of the return value. The odd-indexed elements of the return value are zero.

The addition uses modular arithmetic and the result is well-defined for all inputs; unlike C, there is no undefined behavior for signed overflow.

#### 8.5.1.1. ADDHNB (vector, vector)

Instances
svint8_t <b>svaddhnb</b> [_s16](svint16_t op1, svint16_t op2)
svint16_t <b>svaddhnb</b> [_s32](svint32_t op1, svint32_t op2)
svint32_t <b>svaddhnb</b> [_s64](svint64_t op1, svint64_t op2)
svuint8_t <b>svaddhnb</b> [_u16](svuint16_t op1, svuint16_t op2)
svuint16_t <b>svaddhnb</b> [_u32](svuint32_t op1, svuint32_t op2)
svuint32_t <b>svaddhnb</b> [_u64](svuint64_t op1, svuint64_t op2)

#### 8.5.1.2. ADDHNB (vector, scalar)

Instances
svint8_t <b>svaddhnb</b> [_n_s16](svint16_t op1, int16_t op2)

**Instances**

```
svint16_t svaddhnb[_n_s32](svint32_t op1, int32_t op2)
svint32_t svaddhnb[_n_s64](svint64_t op1, int64_t op2)
svuint8_t svaddhnb[_n_u16](svuint16_t op1, uint16_t op2)
svuint16_t svaddhnb[_n_u32](svuint32_t op1, uint32_t op2)
svuint32_t svaddhnb[_n_u64](svuint64_t op1, uint64_t op2)
```

**8.5.2. ADDHNT: Integer addition, narrowing to high half (top)**

If the elements of the return value have  $N$  bits, these functions add the second and third integer inputs, divide the result by  $2^N$ , rounding towards  $-\text{Inf}$ , and place the quotient in the odd-indexed elements of the return value. The even-indexed elements of the first input supply the corresponding elements of the result.

The addition uses modular arithmetic and the result is well-defined for all inputs; unlike C, there is no undefined behavior for signed overflow.

**8.5.2.1. ADDHNT (vector, vector)****Instances**

```
svint8_t svaddhnt[_s16](svint8_t even, svint16_t op1, svint16_t op2)
svint16_t svaddhnt[_s32](svint16_t even, svint32_t op1, svint32_t op2)
svint32_t svaddhnt[_s64](svint32_t even, svint64_t op1, svint64_t op2)
svuint8_t svaddhnt[_u16](svuint8_t even, svuint16_t op1, svuint16_t op2)
svuint16_t svaddhnt[_u32](svuint16_t even, svuint32_t op1, svuint32_t op2)
svuint32_t svaddhnt[_u64](svuint32_t even, svuint64_t op1, svuint64_t op2)
```

**8.5.2.2. ADDHNT (vector, scalar)****Instances**

```
svint8_t svaddhnt[_n_s16](svint8_t even, svint16_t op1, int16_t op2)
svint16_t svaddhnt[_n_s32](svint16_t even, svint32_t op1, int32_t op2)
svint32_t svaddhnt[_n_s64](svint32_t even, svint64_t op1, int64_t op2)
svuint8_t svaddhnt[_n_u16](svuint8_t even, svuint16_t op1, uint16_t op2)
svuint16_t svaddhnt[_n_u32](svuint16_t even, svuint32_t op1, uint32_t op2)
svuint32_t svaddhnt[_n_u64](svuint32_t even, svuint64_t op1, uint64_t op2)
```

**8.5.3. RADDHNB: Integer addition, rounding narrowing to high half (bottom)**

If the elements of the return value have  $N$  bits, these functions add the two integer inputs, divide the result by  $2^N$ , round the quotient to nearest with ties towards  $+\text{Inf}$ , and place the rounded quotient in the even-indexed elements of the return value. The odd-indexed elements of the return value are zero.

The addition uses modular arithmetic and the result is well-defined for all inputs; unlike C, there is no undefined behavior for signed overflow.

**8.5.3.1. RADDHNB (vector, vector)****Instances**

```
svint8_t svraddhnb[_s16](svint16_t op1, svint16_t op2)
svint16_t svraddhnb[_s32](svint32_t op1, svint32_t op2)
svint32_t svraddhnb[_s64](svint64_t op1, svint64_t op2)
```

**Instances**

```
svuint8_t svraddhnb[_u16](svuint16_t op1, svuint16_t op2)
svuint16_t svraddhnb[_u32](svuint32_t op1, svuint32_t op2)
svuint32_t svraddhnb[_u64](svuint64_t op1, svuint64_t op2)
```

**8.5.3.2. RADDHNB (vector, scalar)****Instances**

```
svint8_t svraddhnb[_n_s16](svint16_t op1, int16_t op2)
svint16_t svraddhnb[_n_s32](svint32_t op1, int32_t op2)
svint32_t svraddhnb[_n_s64](svint64_t op1, int64_t op2)
svuint8_t svraddhnb[_n_u16](svuint16_t op1, uint16_t op2)
svuint16_t svraddhnb[_n_u32](svuint32_t op1, uint32_t op2)
svuint32_t svraddhnb[_n_u64](svuint64_t op1, uint64_t op2)
```

**8.5.4. RADDHNT: Integer addition, rounding narrowing to high half (top)**

If the elements of the return value have  $N$  bits, these functions add the second and third integer inputs, divide the result by  $2^N$ , round the quotient to nearest with ties towards +Inf, and place the rounded quotient in the odd-indexed elements of the return value. The even-indexed elements of the first input supply the corresponding elements of the result.

The addition uses modular arithmetic and the result is well-defined for all inputs; unlike C, there is no undefined behavior for signed overflow.

**8.5.4.1. RADDHNT (vector, vector)****Instances**

```
svint8_t svraddhnt[_s16](svint8_t even, svint16_t op1, svint16_t op2)
svint16_t svraddhnt[_s32](svint16_t even, svint32_t op1, svint32_t op2)
svint32_t svraddhnt[_s64](svint32_t even, svint64_t op1, svint64_t op2)
svuint8_t svraddhnt[_u16](svuint8_t even, svuint16_t op1, svuint16_t op2)
svuint16_t svraddhnt[_u32](svuint16_t even, svuint32_t op1, svuint32_t op2)
svuint32_t svraddhnt[_u64](svuint32_t even, svuint64_t op1, svuint64_t op2)
```

**8.5.4.2. RADDHNT (vector, scalar)****Instances**

```
svint8_t svraddhnt[_n_s16](svint8_t even, svint16_t op1, int16_t op2)
svint16_t svraddhnt[_n_s32](svint16_t even, svint32_t op1, int32_t op2)
svint32_t svraddhnt[_n_s64](svint32_t even, svint64_t op1, int64_t op2)
svuint8_t svraddhnt[_n_u16](svuint8_t even, svuint16_t op1, uint16_t op2)
svuint16_t svraddhnt[_n_u32](svuint16_t even, svuint32_t op1, uint32_t op2)
svuint32_t svraddhnt[_n_u64](svuint32_t even, svuint64_t op1, uint64_t op2)
```

**8.5.5. SUBHNB: Integer subtraction, narrowing to high half (bottom)**

If the elements of the return value have  $N$  bits, these functions subtract the second integer input from the first integer input, divide the result by  $2^N$ , rounding towards -Inf, and place the quotient in the even-indexed elements of the return value. The odd-indexed elements of the return value are zero.

The subtraction uses modular arithmetic and the result is well-defined for all inputs; unlike C, there is no undefined behavior for signed overflow.

### 8.5.5.1. SUBHNB (vector, vector)

Instances
svint8_t <b>svsubhnb</b> [_s16](svint16_t op1, svint16_t op2)
svint16_t <b>svsubhnb</b> [_s32](svint32_t op1, svint32_t op2)
svint32_t <b>svsubhnb</b> [_s64](svint64_t op1, svint64_t op2)
svuint8_t <b>svsubhnb</b> [_u16](svuint16_t op1, svuint16_t op2)
svuint16_t <b>svsubhnb</b> [_u32](svuint32_t op1, svuint32_t op2)
svuint32_t <b>svsubhnb</b> [_u64](svuint64_t op1, svuint64_t op2)

### 8.5.5.2. SUBHNB (vector, scalar)

Instances
svint8_t <b>svsubhnb</b> [_n_s16](svint16_t op1, int16_t op2)
svint16_t <b>svsubhnb</b> [_n_s32](svint32_t op1, int32_t op2)
svint32_t <b>svsubhnb</b> [_n_s64](svint64_t op1, int64_t op2)
svuint8_t <b>svsubhnb</b> [_n_u16](svuint16_t op1, uint16_t op2)
svuint16_t <b>svsubhnb</b> [_n_u32](svuint32_t op1, uint32_t op2)
svuint32_t <b>svsubhnb</b> [_n_u64](svuint64_t op1, uint64_t op2)

## 8.5.6. SUBHNT: Integer subtraction, narrowing to high half (top)

If the elements of the return value have  $N$  bits, these functions subtract the third integer input from the second integer input, divide the result by  $2^N$ , rounding towards  $-\infty$ , and place the quotient in the odd-indexed elements of the return value. The even-indexed elements of the first input supply the corresponding elements of the result.

The subtraction uses modular arithmetic and the result is well-defined for all inputs; unlike C, there is no undefined behavior for signed overflow.

### 8.5.6.1. SUBHNT (vector, vector)

Instances
svint8_t <b>svsubhnt</b> [_s16](svint8_t even, svint16_t op1, svint16_t op2)
svint16_t <b>svsubhnt</b> [_s32](svint16_t even, svint32_t op1, svint32_t op2)
svint32_t <b>svsubhnt</b> [_s64](svint32_t even, svint64_t op1, svint64_t op2)
svuint8_t <b>svsubhnt</b> [_u16](svuint8_t even, svuint16_t op1, svuint16_t op2)
svuint16_t <b>svsubhnt</b> [_u32](svuint16_t even, svuint32_t op1, svuint32_t op2)
svuint32_t <b>svsubhnt</b> [_u64](svuint32_t even, svuint64_t op1, svuint64_t op2)

### 8.5.6.2. SUBHNT (vector, scalar)

Instances
svint8_t <b>svsubhnt</b> [_n_s16](svint8_t even, svint16_t op1, int16_t op2)
svint16_t <b>svsubhnt</b> [_n_s32](svint16_t even, svint32_t op1, int32_t op2)
svint32_t <b>svsubhnt</b> [_n_s64](svint32_t even, svint64_t op1, int64_t op2)
svuint8_t <b>svsubhnt</b> [_n_u16](svuint8_t even, svuint16_t op1, uint16_t op2)
svuint16_t <b>svsubhnt</b> [_n_u32](svuint16_t even, svuint32_t op1, uint32_t op2)
svuint32_t <b>svsubhnt</b> [_n_u64](svuint32_t even, svuint64_t op1, uint64_t op2)

## 8.5.7. RSUBHNB: Integer subtraction, rounding narrowing to high half (bottom)

If the elements of the return value have  $N$  bits, these functions subtract the second integer input from the first integer input, divide the result by  $2^N$ , round the quotient to nearest with ties towards +Inf, and place the rounded quotient in the even-indexed elements of the return value. The odd-indexed elements of the return value are zero.

The subtraction uses modular arithmetic and the result is well-defined for all inputs; unlike C, there is no undefined behavior for signed overflow.

### 8.5.7.1. RSUBHNB (vector, vector)

Instances
svint8_t <b>svrsubhnb</b> [_s16](svint16_t op1, svint16_t op2)
svint16_t <b>svrsubhnb</b> [_s32](svint32_t op1, svint32_t op2)
svint32_t <b>svrsubhnb</b> [_s64](svint64_t op1, svint64_t op2)
svuint8_t <b>svrsubhnb</b> [_u16](svuint16_t op1, svuint16_t op2)
svuint16_t <b>svrsubhnb</b> [_u32](svuint32_t op1, svuint32_t op2)
svuint32_t <b>svrsubhnb</b> [_u64](svuint64_t op1, svuint64_t op2)

### 8.5.7.2. RSUBHNB (vector, scalar)

Instances
svint8_t <b>svrsubhnb</b> [_n_s16](svint16_t op1, int16_t op2)
svint16_t <b>svrsubhnb</b> [_n_s32](svint32_t op1, int32_t op2)
svint32_t <b>svrsubhnb</b> [_n_s64](svint64_t op1, int64_t op2)
svuint8_t <b>svrsubhnb</b> [_n_u16](svuint16_t op1, uint16_t op2)
svuint16_t <b>svrsubhnb</b> [_n_u32](svuint32_t op1, uint32_t op2)
svuint32_t <b>svrsubhnb</b> [_n_u64](svuint64_t op1, uint64_t op2)

## 8.5.8. RSUBHNT: Integer subtraction, rounding narrowing to high half (top)

If the elements of the return value have  $N$  bits, these functions subtract the third integer input from the second integer input, divide the result by  $2^N$ , round the quotient to nearest with ties towards +Inf, and place the rounded quotient in the odd-indexed elements of the return value. The even-indexed elements of the first input supply the corresponding elements of the result.

The subtraction uses modular arithmetic and the result is well-defined for all inputs; unlike C, there is no undefined behavior for signed overflow.

### 8.5.8.1. RSUBHNT (vector, vector)

Instances
svint8_t <b>svrsubhnt</b> [_s16](svint8_t even, svint16_t op1, svint16_t op2)
svint16_t <b>svrsubhnt</b> [_s32](svint16_t even, svint32_t op1, svint32_t op2)
svint32_t <b>svrsubhnt</b> [_s64](svint32_t even, svint64_t op1, svint64_t op2)
svuint8_t <b>svrsubhnt</b> [_u16](svuint8_t even, svuint16_t op1, svuint16_t op2)
svuint16_t <b>svrsubhnt</b> [_u32](svuint16_t even, svuint32_t op1, svuint32_t op2)
svuint32_t <b>svrsubhnt</b> [_u64](svuint32_t even, svuint64_t op1, svuint64_t op2)

### 8.5.8.2. RSUBHNT (vector, scalar)

Instances
svint8_t <b>svrsubhnt</b> [_n_s16](svint8_t <i>even</i> , svint16_t <i>op1</i> , int16_t <i>op2</i> )
svint16_t <b>svrsubhnt</b> [_n_s32](svint16_t <i>even</i> , svint32_t <i>op1</i> , int32_t <i>op2</i> )
svint32_t <b>svrsubhnt</b> [_n_s64](svint32_t <i>even</i> , svint64_t <i>op1</i> , int64_t <i>op2</i> )
svuint8_t <b>svrsubhnt</b> [_n_u16](svuint8_t <i>even</i> , svuint16_t <i>op1</i> , uint16_t <i>op2</i> )
svuint16_t <b>svrsubhnt</b> [_n_u32](svuint16_t <i>even</i> , svuint32_t <i>op1</i> , uint32_t <i>op2</i> )
svuint32_t <b>svrsubhnt</b> [_n_u64](svuint32_t <i>even</i> , svuint64_t <i>op1</i> , uint64_t <i>op2</i> )

### 8.5.9. SHRNB: Shift right, narrowing (bottom)

These functions divide the first integer input by two to the power of the second integer input, rounding towards -Inf. They then store the quotient in the even-indexed elements of the return value, truncating the quotient to make it fit. The odd-indexed elements of the return value are zero.

If the elements of the return type have  $N$  bits, the second input must be an integer constant expression in the range  $[1, N]$ .

#### 8.5.9.1. SHRNB (vector, immediate)

Instances
svint8_t <b>svshrn</b> [_n_s16](svint16_t <i>op1</i> , uint64_t <i>imm2</i> )
svint16_t <b>svshrn</b> [_n_s32](svint32_t <i>op1</i> , uint64_t <i>imm2</i> )
svint32_t <b>svshrn</b> [_n_s64](svint64_t <i>op1</i> , uint64_t <i>imm2</i> )
svuint8_t <b>svshrn</b> [_n_u16](svuint16_t <i>op1</i> , uint64_t <i>imm2</i> )
svuint16_t <b>svshrn</b> [_n_u32](svuint32_t <i>op1</i> , uint64_t <i>imm2</i> )
svuint32_t <b>svshrn</b> [_n_u64](svuint64_t <i>op1</i> , uint64_t <i>imm2</i> )

### 8.5.10. SHRNT: Shift right, narrowing (top)

These functions divide the second integer input by two to the power of the third integer input, rounding towards -Inf. They then store the quotient in the odd-indexed elements of the return value, truncating the quotient to make it fit. The even-indexed elements of the first input supply the corresponding elements of the result.

If the elements of the return type have  $N$  bits, the second input must be an integer constant expression in the range  $[1, N]$ .

#### 8.5.10.1. SHRNT (vector, immediate)

Instances
svint8_t <b>svshrnt</b> [_n_s16](svint8_t <i>even</i> , svint16_t <i>op1</i> , uint64_t <i>imm2</i> )
svint16_t <b>svshrnt</b> [_n_s32](svint16_t <i>even</i> , svint32_t <i>op1</i> , uint64_t <i>imm2</i> )
svint32_t <b>svshrnt</b> [_n_s64](svint32_t <i>even</i> , svint64_t <i>op1</i> , uint64_t <i>imm2</i> )
svuint8_t <b>svshrnt</b> [_n_u16](svuint8_t <i>even</i> , svuint16_t <i>op1</i> , uint64_t <i>imm2</i> )
svuint16_t <b>svshrnt</b> [_n_u32](svuint16_t <i>even</i> , svuint32_t <i>op1</i> , uint64_t <i>imm2</i> )
svuint32_t <b>svshrnt</b> [_n_u64](svuint32_t <i>even</i> , svuint64_t <i>op1</i> , uint64_t <i>imm2</i> )

### 8.5.11. RSHRNB: Rounding shift right, narrowing (bottom)

These functions divide the first integer input by two to the power of the second integer input and round the quotient to nearest with ties towards +Inf. They then store the rounded quotient in the even-indexed

elements of the return value, truncating it to make it fit. The odd-indexed elements of the return value are zero.

If the elements of the return type have  $N$  bits, the second input must be an integer constant expression in the range  $[1, N]$ .

### 8.5.11.1. RSHRNB (vector, immediate)

Instances	
<code>svint8_t</code>	<code>svrshrn timerb[_n_s16](svint16_t op1, uint64_t imm2)</code>
<code>svint16_t</code>	<code>svrshrn timerb[_n_s32](svint32_t op1, uint64_t imm2)</code>
<code>svint32_t</code>	<code>svrshrn timerb[_n_s64](svint64_t op1, uint64_t imm2)</code>
<code>svuint8_t</code>	<code>svrshrn timerb[_n_u16](svuint16_t op1, uint64_t imm2)</code>
<code>svuint16_t</code>	<code>svrshrn timerb[_n_u32](svuint32_t op1, uint64_t imm2)</code>
<code>svuint32_t</code>	<code>svrshrn timerb[_n_u64](svuint64_t op1, uint64_t imm2)</code>

## 8.5.12. RSHRNT: Rounding shift right, narrowing (top)

These functions divide the second integer input by two to the power of the third integer input and round the quotient to nearest with ties towards +Inf. They then store the rounded quotient in the odd-indexed elements of the return value, truncating it to make it fit. The even-indexed elements of the first input supply the corresponding elements of the result.

If the elements of the return type have  $N$  bits, the second input must be an integer constant expression in the range  $[1, N]$ .

### 8.5.12.1. RSHRNT (vector, immediate)

Instances	
<code>svint8_t</code>	<code>svrshrnt timerb[_n_s16](svint8_t even, svint16_t op1, uint64_t imm2)</code>
<code>svint16_t</code>	<code>svrshrnt timerb[_n_s32](svint16_t even, svint32_t op1, uint64_t imm2)</code>
<code>svint32_t</code>	<code>svrshrnt timerb[_n_s64](svint32_t even, svint64_t op1, uint64_t imm2)</code>
<code>svuint8_t</code>	<code>svrshrnt timerb[_n_u16](svuint8_t even, svuint16_t op1, uint64_t imm2)</code>
<code>svuint16_t</code>	<code>svrshrnt timerb[_n_u32](svuint16_t even, svuint32_t op1, uint64_t imm2)</code>
<code>svuint32_t</code>	<code>svrshrnt timerb[_n_u64](svuint32_t even, svuint64_t op1, uint64_t imm2)</code>

## 8.5.13. QSHRNB: Shift right, saturating narrowing (bottom)

These functions divide the first integer input by two to the power of the second integer input, rounding towards -Inf. They then store the quotient in the even-indexed elements of the return value, saturating the quotient to make it fit. The odd-indexed elements of the return value are zero.

If the elements of the return type have  $N$  bits, the second input must be an integer constant expression in the range  $[1, N]$ .

### 8.5.13.1. QSHRNB (vector, immediate)

Instances	
<code>svint8_t</code>	<code>svqshrnb timerb[_n_s16](svint16_t op1, uint64_t imm2)</code>
<code>svint16_t</code>	<code>svqshrnb timerb[_n_s32](svint32_t op1, uint64_t imm2)</code>
<code>svint32_t</code>	<code>svqshrnb timerb[_n_s64](svint64_t op1, uint64_t imm2)</code>
<code>svuint8_t</code>	<code>svqshrnb timerb[_n_u16](svuint16_t op1, uint64_t imm2)</code>
<code>svuint16_t</code>	<code>svqshrnb timerb[_n_u32](svuint32_t op1, uint64_t imm2)</code>
<code>svuint32_t</code>	<code>svqshrnb timerb[_n_u64](svuint64_t op1, uint64_t imm2)</code>



## 8.5.14. QSHRNT: Shift right, saturating narrowing (top)

These functions divide the second integer input by two to the power of the third integer input, rounding towards -Inf. They then store the quotient in the odd-indexed elements of the return value, saturating the quotient to make it fit. The even-indexed elements of the first input supply the corresponding elements of the result.

If the elements of the return type have  $N$  bits, the second input must be an integer constant expression in the range  $[1, N]$ .

### 8.5.14.1. QSHRNT (vector, immediate)

Instances
<code>svint8_t svqshrnt[_n_s16](svint8_t even, svint16_t op1, uint64_t imm2)</code>
<code>svint16_t svqshrnt[_n_s32](svint16_t even, svint32_t op1, uint64_t imm2)</code>
<code>svint32_t svqshrnt[_n_s64](svint32_t even, svint64_t op1, uint64_t imm2)</code>
<code>svuint8_t svqshrnt[_n_u16](svuint8_t even, svuint16_t op1, uint64_t imm2)</code>
<code>svuint16_t svqshrnt[_n_u32](svuint16_t even, svuint32_t op1, uint64_t imm2)</code>
<code>svuint32_t svqshrnt[_n_u64](svuint32_t even, svuint64_t op1, uint64_t imm2)</code>

## 8.5.15. QSHRUNB: Shift right, saturating narrowing to unsigned (bottom)

These functions act like [Section 8.5.13, “QSHRNB: Shift right, saturating narrowing \(bottom\)”](#), except that the first input is signed and the return value is unsigned.

### 8.5.15.1. QSHRUNB (vector, immediate)

Instances
<code>svuint8_t svqshrnb[_n_s16](svint16_t op1, uint64_t imm2)</code>
<code>svuint16_t svqshrnb[_n_s32](svint32_t op1, uint64_t imm2)</code>
<code>svuint32_t svqshrnb[_n_s64](svint64_t op1, uint64_t imm2)</code>

## 8.5.16. QSHRUNT: Shift right, saturating narrowing to unsigned (top)

These functions act like [Section 8.5.14, “QSHRNT: Shift right, saturating narrowing \(top\)”](#), except that the second input is signed and the return value is unsigned.

### 8.5.16.1. QSHRUNT (vector, immediate)

Instances
<code>svuint8_t svqshrunt[_n_s16](svuint8_t even, svint16_t op1, uint64_t imm2)</code>
<code>svuint16_t svqshrunt[_n_s32](svuint16_t even, svint32_t op1, uint64_t imm2)</code>
<code>svuint32_t svqshrunt[_n_s64](svuint32_t even, svint64_t op1, uint64_t imm2)</code>

## 8.5.17. QRSHRNB: Rounding shift right, saturating narrowing (bottom)

These functions divide the first integer input by two to the power of the second integer input and round the result to nearest with ties towards +Inf. They then store the rounded quotient in the even-indexed elements of the return value, saturating it to make it fit. The odd-indexed elements of the return value are zero.

If the elements of the return type have  $N$  bits, the second input must be an integer constant expression in the range  $[1, N]$ .

### 8.5.17.1. QRSHRNB (vector, immediate)

Instances	
<code>svint8_t</code>	<code>svqqrshrn timerb[_n_s16](svint16_t op1, uint64_t imm2)</code>
<code>svint16_t</code>	<code>svqqrshrn timerb[_n_s32](svint32_t op1, uint64_t imm2)</code>
<code>svint32_t</code>	<code>svqqrshrn timerb[_n_s64](svint64_t op1, uint64_t imm2)</code>
<code>svuint8_t</code>	<code>svqqrshrn timerb[_n_u16](svuint16_t op1, uint64_t imm2)</code>
<code>svuint16_t</code>	<code>svqqrshrn timerb[_n_u32](svuint32_t op1, uint64_t imm2)</code>
<code>svuint32_t</code>	<code>svqqrshrn timerb[_n_u64](svuint64_t op1, uint64_t imm2)</code>

### 8.5.18. QRSHRNT: Rounding shift right, saturating narrowing (top)

These functions divide the second integer input by two to the power of the third integer input and round the result to nearest with ties towards +Inf. They then store the rounded quotient in the odd-indexed elements of the return value, saturating it to make it fit. The even-indexed elements of the first input supply the corresponding elements of the result.

If the elements of the return type have  $N$  bits, the second input must be an integer constant expression in the range  $[1, N]$ .

#### 8.5.18.1. QRSHRNT (vector, immediate)

Instances	
<code>svint8_t</code>	<code>svqqrshrnt[_n_s16](svint8_t even, svint16_t op1, uint64_t imm2)</code>
<code>svint16_t</code>	<code>svqqrshrnt[_n_s32](svint16_t even, svint32_t op1, uint64_t imm2)</code>
<code>svint32_t</code>	<code>svqqrshrnt[_n_s64](svint32_t even, svint64_t op1, uint64_t imm2)</code>
<code>svuint8_t</code>	<code>svqqrshrnt[_n_u16](svuint8_t even, svuint16_t op1, uint64_t imm2)</code>
<code>svuint16_t</code>	<code>svqqrshrnt[_n_u32](svuint16_t even, svuint32_t op1, uint64_t imm2)</code>
<code>svuint32_t</code>	<code>svqqrshrnt[_n_u64](svuint32_t even, svuint64_t op1, uint64_t imm2)</code>

### 8.5.19. QRSHRUNB: Rounding shift right, saturating narrowing to unsigned (bottom)

These functions act like [Section 8.5.17, “QRSHRNB: Rounding shift right, saturating narrowing \(bottom\)”](#), except that the first input is signed and the return value is unsigned.

#### 8.5.19.1. QRSHRUNB (vector, immediate)

Instances	
<code>svuint8_t</code>	<code>svqqrshrnb[_n_s16](svint16_t op1, uint64_t imm2)</code>
<code>svuint16_t</code>	<code>svqqrshrnb[_n_s32](svint32_t op1, uint64_t imm2)</code>
<code>svuint32_t</code>	<code>svqqrshrnb[_n_s64](svint64_t op1, uint64_t imm2)</code>

### 8.5.20. QRSHRUNT: Rounding shift right, saturating narrowing to unsigned (top)

These functions act like [Section 8.5.18, “QRSHRNT: Rounding shift right, saturating narrowing \(top\)”](#), except that the second input is signed and the return value is unsigned.

### 8.5.20.1. QRSHRUNT (vector, immediate)

Instances	
<code>svuint8_t</code>	<code>svqqrshrun</code> <code>[_n_s16](svuint8_t even, svint16_t op1, uint64_t imm2)</code>
<code>svuint16_t</code>	<code>svqqrshrun</code> <code>[_n_s32](svuint16_t even, svint32_t op1, uint64_t imm2)</code>
<code>svuint32_t</code>	<code>svqqrshrun</code> <code>[_n_s64](svuint32_t even, svint64_t op1, uint64_t imm2)</code>

## 8.6. Unary narrowing operations

### 8.6.1. QXTNB: Saturating narrowing (bottom)

These functions saturate the integer input to half the original width and store the result in the even-indexed elements of the return value. The odd-indexed elements of the return value are zero.

#### 8.6.1.1. QXTNB (vector)

Instances	
<code>svint8_t</code>	<code>svqxtnb</code> <code>[_s16](svint16_t op)</code>
<code>svint16_t</code>	<code>svqxtnb</code> <code>[_s32](svint32_t op)</code>
<code>svint32_t</code>	<code>svqxtnb</code> <code>[_s64](svint64_t op)</code>
<code>svuint8_t</code>	<code>svqxtnb</code> <code>[_u16](svuint16_t op)</code>
<code>svuint16_t</code>	<code>svqxtnb</code> <code>[_u32](svuint32_t op)</code>
<code>svuint32_t</code>	<code>svqxtnb</code> <code>[_u64](svuint64_t op)</code>

### 8.6.2. QXTNT: Saturating narrowing (top)

These functions saturate the second integer input to half the original width and store the result in the odd-indexed elements of the return value. The even-indexed elements of the first input supply the corresponding elements of the result.

#### 8.6.2.1. QXTNT (vector)

Instances	
<code>svint8_t</code>	<code>svqxtn</code> <code>[_s16](svint8_t even, svint16_t op)</code>
<code>svint16_t</code>	<code>svqxtn</code> <code>[_s32](svint16_t even, svint32_t op)</code>
<code>svint32_t</code>	<code>svqxtn</code> <code>[_s64](svint32_t even, svint64_t op)</code>
<code>svuint8_t</code>	<code>svqxtn</code> <code>[_u16](svuint8_t even, svuint16_t op)</code>
<code>svuint16_t</code>	<code>svqxtn</code> <code>[_u32](svuint16_t even, svuint32_t op)</code>
<code>svuint32_t</code>	<code>svqxtn</code> <code>[_u64](svuint32_t even, svuint64_t op)</code>

### 8.6.3. QXTUNB: Saturating narrowing to unsigned (bottom)

These functions act like [Section 8.6.1, “QXTNB: Saturating narrowing \(bottom\)”](#) except that the input is signed and the return value is unsigned.

#### 8.6.3.1. QXTUNB (vector)

Instances	
<code>svuint8_t</code>	<code>svqxtnb</code> <code>[_s16](svint16_t op)</code>
<code>svuint16_t</code>	<code>svqxtnb</code> <code>[_s32](svint32_t op)</code>
<code>svuint32_t</code>	<code>svqxtnb</code> <code>[_s64](svint64_t op)</code>

## 8.6.4. QXTUNT: Saturating narrowing to unsigned (top)

These functions act like [Section 8.6.2, “QXTNT: Saturating narrowing \(top\)”](#) except that the second input is signed and the return value is unsigned.

### 8.6.4.1. QXTUNT (vector)

Instances
<code>svuint8_t svqxtunt[_s16](svuint8_t even, svint16_t op)</code>
<code>svuint16_t svqxtunt[_s32](svuint16_t even, svint32_t op)</code>
<code>svuint32_t svqxtunt[_s64](svuint32_t even, svint64_t op)</code>

## 8.7. Non-widening pairwise arithmetic

### 8.7.1. ADDP: Integer pairwise addition

These functions add pairs of adjacent elements from each individual integer input, then interleave the results so that one result from the first input is followed by the corresponding result from the second input.

The addition uses modular arithmetic and the result is well-defined for all inputs; unlike C, there is no undefined behavior for signed overflow.

#### 8.7.1.1. ADDP (vector, vector), merging with first input

Instances
<code>svint8_t svaddp[_s8]_m(svbool_t pg, svint8_t op1, svint8_t op2)</code>
<code>svint16_t svaddp[_s16]_m(svbool_t pg, svint16_t op1, svint16_t op2)</code>
<code>svint32_t svaddp[_s32]_m(svbool_t pg, svint32_t op1, svint32_t op2)</code>
<code>svint64_t svaddp[_s64]_m(svbool_t pg, svint64_t op1, svint64_t op2)</code>
<code>svuint8_t svaddp[_u8]_m(svbool_t pg, svuint8_t op1, svuint8_t op2)</code>
<code>svuint16_t svaddp[_u16]_m(svbool_t pg, svuint16_t op1, svuint16_t op2)</code>
<code>svuint32_t svaddp[_u32]_m(svbool_t pg, svuint32_t op1, svuint32_t op2)</code>
<code>svuint64_t svaddp[_u64]_m(svbool_t pg, svuint64_t op1, svuint64_t op2)</code>

#### 8.7.1.2. ADDP (vector, vector), setting inactive to unknown

Instances
<code>svint8_t svaddp[_s8]_x(svbool_t pg, svint8_t op1, svint8_t op2)</code>
<code>svint16_t svaddp[_s16]_x(svbool_t pg, svint16_t op1, svint16_t op2)</code>
<code>svint32_t svaddp[_s32]_x(svbool_t pg, svint32_t op1, svint32_t op2)</code>
<code>svint64_t svaddp[_s64]_x(svbool_t pg, svint64_t op1, svint64_t op2)</code>
<code>svuint8_t svaddp[_u8]_x(svbool_t pg, svuint8_t op1, svuint8_t op2)</code>
<code>svuint16_t svaddp[_u16]_x(svbool_t pg, svuint16_t op1, svuint16_t op2)</code>
<code>svuint32_t svaddp[_u32]_x(svbool_t pg, svuint32_t op1, svuint32_t op2)</code>
<code>svuint64_t svaddp[_u64]_x(svbool_t pg, svuint64_t op1, svuint64_t op2)</code>

### 8.7.2. ADDP: Floating-point pairwise addition

These functions add pairs of adjacent elements from each individual floating-point input, then interleave the results so that one result from the first input is followed by the corresponding result from the second input.

Signaling NaNs trigger an IEEE Invalid exception but quiet NaNs do not. Adding +Inf and -Inf together also triggers an IEEE Invalid exception.

### 8.7.2.1. ADDP (vector, vector), merging with first input

Instances	
svfloat16_t	<b>svaddp</b> [_f16]_m(svbool_t pg, svfloat16_t op1, svfloat16_t op2)
svfloat32_t	<b>svaddp</b> [_f32]_m(svbool_t pg, svfloat32_t op1, svfloat32_t op2)
svfloat64_t	<b>svaddp</b> [_f64]_m(svbool_t pg, svfloat64_t op1, svfloat64_t op2)

### 8.7.2.2. ADDP (vector, vector), setting inactive to unknown

Instances	
svfloat16_t	<b>svaddp</b> [_f16]_x(svbool_t pg, svfloat16_t op1, svfloat16_t op2)
svfloat32_t	<b>svaddp</b> [_f32]_x(svbool_t pg, svfloat32_t op1, svfloat32_t op2)
svfloat64_t	<b>svaddp</b> [_f64]_x(svbool_t pg, svfloat64_t op1, svfloat64_t op2)

## 8.7.3. MAXP: Integer pairwise maximum

These functions partition each individual integer input into pairs of adjacent elements and calculate the maximum value in each pair. They then interleave the results so that one result from the first input is followed by the corresponding result from the second input.

### 8.7.3.1. MAXP (vector, vector), merging with first input

Instances	
svint8_t	<b>svmaxp</b> [_s8]_m(svbool_t pg, svint8_t op1, svint8_t op2)
svint16_t	<b>svmaxp</b> [_s16]_m(svbool_t pg, svint16_t op1, svint16_t op2)
svint32_t	<b>svmaxp</b> [_s32]_m(svbool_t pg, svint32_t op1, svint32_t op2)
svint64_t	<b>svmaxp</b> [_s64]_m(svbool_t pg, svint64_t op1, svint64_t op2)
svuint8_t	<b>svmaxp</b> [_u8]_m(svbool_t pg, svuint8_t op1, svuint8_t op2)
svuint16_t	<b>svmaxp</b> [_u16]_m(svbool_t pg, svuint16_t op1, svuint16_t op2)
svuint32_t	<b>svmaxp</b> [_u32]_m(svbool_t pg, svuint32_t op1, svuint32_t op2)
svuint64_t	<b>svmaxp</b> [_u64]_m(svbool_t pg, svuint64_t op1, svuint64_t op2)

### 8.7.3.2. MAXP (vector, vector), setting inactive to unknown

Instances	
svint8_t	<b>svmaxp</b> [_s8]_x(svbool_t pg, svint8_t op1, svint8_t op2)
svint16_t	<b>svmaxp</b> [_s16]_x(svbool_t pg, svint16_t op1, svint16_t op2)
svint32_t	<b>svmaxp</b> [_s32]_x(svbool_t pg, svint32_t op1, svint32_t op2)
svint64_t	<b>svmaxp</b> [_s64]_x(svbool_t pg, svint64_t op1, svint64_t op2)
svuint8_t	<b>svmaxp</b> [_u8]_x(svbool_t pg, svuint8_t op1, svuint8_t op2)
svuint16_t	<b>svmaxp</b> [_u16]_x(svbool_t pg, svuint16_t op1, svuint16_t op2)
svuint32_t	<b>svmaxp</b> [_u32]_x(svbool_t pg, svuint32_t op1, svuint32_t op2)
svuint64_t	<b>svmaxp</b> [_u64]_x(svbool_t pg, svuint64_t op1, svuint64_t op2)

## 8.7.4. MAXP: Floating-point pairwise maximum

These functions partition each individual floating-point input into pairs of adjacent elements and calculate the maximum value in each pair. They then interleave the results so that one result from the first input is followed by the corresponding result from the second input.

If either element in a pair is NaN, the result for that pair is also a NaN. Signaling NaNs trigger an IEEE Invalid exception but quiet NaNs do not.

### 8.7.4.1. MAXP (vector, vector), merging with first input

Instances	
svfloat16_t	<b>svmaxp</b> [_f16]_m(svbool_t pg, svfloat16_t op1, svfloat16_t op2)
svfloat32_t	<b>svmaxp</b> [_f32]_m(svbool_t pg, svfloat32_t op1, svfloat32_t op2)
svfloat64_t	<b>svmaxp</b> [_f64]_m(svbool_t pg, svfloat64_t op1, svfloat64_t op2)

### 8.7.4.2. MAXP (vector, vector), setting inactive to unknown

Instances	
svfloat16_t	<b>svmaxp</b> [_f16]_x(svbool_t pg, svfloat16_t op1, svfloat16_t op2)
svfloat32_t	<b>svmaxp</b> [_f32]_x(svbool_t pg, svfloat32_t op1, svfloat32_t op2)
svfloat64_t	<b>svmaxp</b> [_f64]_x(svbool_t pg, svfloat64_t op1, svfloat64_t op2)

## 8.7.5. MAXNMP: Floating-point pairwise maximum number

These functions partition each individual floating-point input into pairs of adjacent elements and calculate the maximum value in each pair. They then interleave the results so that one result from the first input is followed by the corresponding result from the second input.

If both elements in a pair are NaNs, the result is also a NaN. If only one element is a NaN, the result is the other element. Signaling NaNs trigger an IEEE Invalid exception but quiet NaNs do not.

### 8.7.5.1. MAXNMP (vector, vector), merging with first input

Instances	
svfloat16_t	<b>svmaxnmp</b> [_f16]_m(svbool_t pg, svfloat16_t op1, svfloat16_t op2)
svfloat32_t	<b>svmaxnmp</b> [_f32]_m(svbool_t pg, svfloat32_t op1, svfloat32_t op2)
svfloat64_t	<b>svmaxnmp</b> [_f64]_m(svbool_t pg, svfloat64_t op1, svfloat64_t op2)

### 8.7.5.2. MAXNMP (vector, vector), setting inactive to unknown

Instances	
svfloat16_t	<b>svmaxnmp</b> [_f16]_x(svbool_t pg, svfloat16_t op1, svfloat16_t op2)
svfloat32_t	<b>svmaxnmp</b> [_f32]_x(svbool_t pg, svfloat32_t op1, svfloat32_t op2)
svfloat64_t	<b>svmaxnmp</b> [_f64]_x(svbool_t pg, svfloat64_t op1, svfloat64_t op2)

## 8.7.6. MINP: Integer pairwise minimum

These functions partition each individual integer input into pairs of adjacent elements and calculate the minimum value in each pair. They then interleave the results so that one result from the first input is followed by the corresponding result from the second input.

### 8.7.6.1. MINP (vector, vector), merging with first input

Instances	
svint8_t	<b>svminp</b> [_s8]_m(svbool_t pg, svint8_t op1, svint8_t op2)
svint16_t	<b>svminp</b> [_s16]_m(svbool_t pg, svint16_t op1, svint16_t op2)
svint32_t	<b>svminp</b> [_s32]_m(svbool_t pg, svint32_t op1, svint32_t op2)
svint64_t	<b>svminp</b> [_s64]_m(svbool_t pg, svint64_t op1, svint64_t op2)
svuint8_t	<b>svminp</b> [_u8]_m(svbool_t pg, svuint8_t op1, svuint8_t op2)
svuint16_t	<b>svminp</b> [_u16]_m(svbool_t pg, svuint16_t op1, svuint16_t op2)

**Instances**

```
svuint32_t svminp[_u32]_m(svbool_t pg, svuint32_t op1, svuint32_t op2)
svuint64_t svminp[_u64]_m(svbool_t pg, svuint64_t op1, svuint64_t op2)
```

**8.7.6.2. MINP (vector, vector), setting inactive to unknown****Instances**

```
svint8_t svminp[_s8]_x(svbool_t pg, svint8_t op1, svint8_t op2)
svint16_t svminp[_s16]_x(svbool_t pg, svint16_t op1, svint16_t op2)
svint32_t svminp[_s32]_x(svbool_t pg, svint32_t op1, svint32_t op2)
svint64_t svminp[_s64]_x(svbool_t pg, svint64_t op1, svint64_t op2)
svuint8_t svminp[_u8]_x(svbool_t pg, svuint8_t op1, svuint8_t op2)
svuint16_t svminp[_u16]_x(svbool_t pg, svuint16_t op1, svuint16_t op2)
svuint32_t svminp[_u32]_x(svbool_t pg, svuint32_t op1, svuint32_t op2)
svuint64_t svminp[_u64]_x(svbool_t pg, svuint64_t op1, svuint64_t op2)
```

**8.7.7. MINP: Floating-point pairwise minimum**

These functions partition each individual floating-point input into pairs of adjacent elements and calculate the minimum value in each pair. They then interleave the results so that one result from the first input is followed by the corresponding result from the second input.

If either element in a pair is NaN, the result for that pair is also a NaN. Signaling NaNs trigger an IEEE Invalid exception but quiet NaNs do not.

**8.7.7.1. MINP (vector, vector), merging with first input****Instances**

```
svfloat16_t svminp[_f16]_m(svbool_t pg, svfloat16_t op1, svfloat16_t op2)
svfloat32_t svminp[_f32]_m(svbool_t pg, svfloat32_t op1, svfloat32_t op2)
svfloat64_t svminp[_f64]_m(svbool_t pg, svfloat64_t op1, svfloat64_t op2)
```

**8.7.7.2. MINP (vector, vector), setting inactive to unknown****Instances**

```
svfloat16_t svminp[_f16]_x(svbool_t pg, svfloat16_t op1, svfloat16_t op2)
svfloat32_t svminp[_f32]_x(svbool_t pg, svfloat32_t op1, svfloat32_t op2)
svfloat64_t svminp[_f64]_x(svbool_t pg, svfloat64_t op1, svfloat64_t op2)
```

**8.7.8. MINNMP: Floating-point pairwise minimum number**

These functions partition each individual floating-point input into pairs of adjacent elements and calculate the minimum value in each pair. They then interleave the results so that one result from the first input is followed by the corresponding result from the second input.

If both elements in a pair are NaNs, the result is also a NaN. If only one element is a NaN, the result is the other element. Signaling NaNs trigger an IEEE Invalid exception but quiet NaNs do not.

**8.7.8.1. MINNMP (vector, vector), merging with first input****Instances**

```
svfloat16_t svminnmp[_f16]_m(svbool_t pg, svfloat16_t op1, svfloat16_t op2)
```

Instances
svfloat32_t <b>svminnmp</b> [_f32]_m(svbool_t pg, svfloat32_t op1, svfloat32_t op2)
svfloat64_t <b>svminnmp</b> [_f64]_m(svbool_t pg, svfloat64_t op1, svfloat64_t op2)

### 8.7.8.2. MINNMP (vector, vector), setting inactive to unknown

Instances
svfloat16_t <b>svminnmp</b> [_f16]_x(svbool_t pg, svfloat16_t op1, svfloat16_t op2)
svfloat32_t <b>svminnmp</b> [_f32]_x(svbool_t pg, svfloat32_t op1, svfloat32_t op2)
svfloat64_t <b>svminnmp</b> [_f64]_x(svbool_t pg, svfloat64_t op1, svfloat64_t op2)

## 8.8. Widening pairwise arithmetic

### 8.8.1. ADALP: Integer pairwise addition and accumulate

These functions add pairs of adjacent elements from the second integer input, giving one double-width result for each element of the first input. They then add each element of the first input to the corresponding addition result.

The second addition uses modular arithmetic and the result is well-defined for all inputs; unlike C, there is no undefined behavior for signed overflow.

#### 8.8.1.1. ADALP (vector, vector), setting inactive to zero

Instances
svint16_t <b>svadalp</b> [_s16]_z(svbool_t pg, svint16_t op1, svint8_t op2)
svint32_t <b>svadalp</b> [_s32]_z(svbool_t pg, svint32_t op1, svint16_t op2)
svint64_t <b>svadalp</b> [_s64]_z(svbool_t pg, svint64_t op1, svint32_t op2)
svuint16_t <b>svadalp</b> [_u16]_z(svbool_t pg, svuint16_t op1, svuint8_t op2)
svuint32_t <b>svadalp</b> [_u32]_z(svbool_t pg, svuint32_t op1, svuint16_t op2)
svuint64_t <b>svadalp</b> [_u64]_z(svbool_t pg, svuint64_t op1, svuint32_t op2)

#### 8.8.1.2. ADALP (vector, vector), merging with first input

Instances
svint16_t <b>svadalp</b> [_s16]_m(svbool_t pg, svint16_t op1, svint8_t op2)
svint32_t <b>svadalp</b> [_s32]_m(svbool_t pg, svint32_t op1, svint16_t op2)
svint64_t <b>svadalp</b> [_s64]_m(svbool_t pg, svint64_t op1, svint32_t op2)
svuint16_t <b>svadalp</b> [_u16]_m(svbool_t pg, svuint16_t op1, svuint8_t op2)
svuint32_t <b>svadalp</b> [_u32]_m(svbool_t pg, svuint32_t op1, svuint16_t op2)
svuint64_t <b>svadalp</b> [_u64]_m(svbool_t pg, svuint64_t op1, svuint32_t op2)

#### 8.8.1.3. ADALP (vector, vector), setting inactive to unknown

Instances
svint16_t <b>svadalp</b> [_s16]_x(svbool_t pg, svint16_t op1, svint8_t op2)
svint32_t <b>svadalp</b> [_s32]_x(svbool_t pg, svint32_t op1, svint16_t op2)
svint64_t <b>svadalp</b> [_s64]_x(svbool_t pg, svint64_t op1, svint32_t op2)
svuint16_t <b>svadalp</b> [_u16]_x(svbool_t pg, svuint16_t op1, svuint8_t op2)
svuint32_t <b>svadalp</b> [_u32]_x(svbool_t pg, svuint32_t op1, svuint16_t op2)
svuint64_t <b>svadalp</b> [_u64]_x(svbool_t pg, svuint64_t op1, svuint32_t op2)



## 8.9. Bitwise ternary logical instructions

### 8.9.1. BCAX: Bitwise clear and exclusive OR

These functions AND the second input with the inverse of the third and return the exclusive OR of this value and the first input.

#### 8.9.1.1. BCAX (vector, vector, vector)

##### Instances

```
svint8_t  svbcax[_s8](svint8_t op1, svint8_t op2, svint8_t op3)
svint16_t svbcax[_s16](svint16_t op1, svint16_t op2, svint16_t op3)
svint32_t svbcax[_s32](svint32_t op1, svint32_t op2, svint32_t op3)
svint64_t svbcax[_s64](svint64_t op1, svint64_t op2, svint64_t op3)
svuint8_t svbcax[_u8](svuint8_t op1, svuint8_t op2, svuint8_t op3)
svuint16_t svbcax[_u16](svuint16_t op1, svuint16_t op2, svuint16_t op3)
svuint32_t svbcax[_u32](svuint32_t op1, svuint32_t op2, svuint32_t op3)
svuint64_t svbcax[_u64](svuint64_t op1, svuint64_t op2, svuint64_t op3)
```

#### 8.9.1.2. BCAX (vector, vector, scalar)

##### Instances

```
svint8_t  svbcax[_n_s8](svint8_t op1, svint8_t op2, int8_t op3)
svint16_t svbcax[_n_s16](svint16_t op1, svint16_t op2, int16_t op3)
svint32_t svbcax[_n_s32](svint32_t op1, svint32_t op2, int32_t op3)
svint64_t svbcax[_n_s64](svint64_t op1, svint64_t op2, int64_t op3)
svuint8_t svbcax[_n_u8](svuint8_t op1, svuint8_t op2, uint8_t op3)
svuint16_t svbcax[_n_u16](svuint16_t op1, svuint16_t op2, uint16_t op3)
svuint32_t svbcax[_n_u32](svuint32_t op1, svuint32_t op2, uint32_t op3)
svuint64_t svbcax[_n_u64](svuint64_t op1, svuint64_t op2, uint64_t op3)
```

### 8.9.2. BSL: Bitwise select

These functions return a vector in which bit *N* contains bit *N* of the first input when bit *N* of the third input is 1 and which contains bit *N* of the second input otherwise.

#### 8.9.2.1. BSL (vector, vector, vector)

##### Instances

```
svint8_t  svbsl[_s8](svint8_t op1, svint8_t op2, svint8_t op3)
svint16_t svbsl[_s16](svint16_t op1, svint16_t op2, svint16_t op3)
svint32_t svbsl[_s32](svint32_t op1, svint32_t op2, svint32_t op3)
svint64_t svbsl[_s64](svint64_t op1, svint64_t op2, svint64_t op3)
svuint8_t svbsl[_u8](svuint8_t op1, svuint8_t op2, svuint8_t op3)
svuint16_t svbsl[_u16](svuint16_t op1, svuint16_t op2, svuint16_t op3)
svuint32_t svbsl[_u32](svuint32_t op1, svuint32_t op2, svuint32_t op3)
svuint64_t svbsl[_u64](svuint64_t op1, svuint64_t op2, svuint64_t op3)
```

#### 8.9.2.2. BSL (vector, vector, scalar)

##### Instances

```
svint8_t  svbsl[_n_s8](svint8_t op1, svint8_t op2, int8_t op3)
```

**Instances**

```

svint16_t svbs1[_n_s16](svint16_t op1, svint16_t op2, int16_t op3)
svint32_t svbs1[_n_s32](svint32_t op1, svint32_t op2, int32_t op3)
svint64_t svbs1[_n_s64](svint64_t op1, svint64_t op2, int64_t op3)
svuint8_t svbs1[_n_u8](svuint8_t op1, svuint8_t op2, uint8_t op3)
svuint16_t svbs1[_n_u16](svuint16_t op1, svuint16_t op2, uint16_t op3)
svuint32_t svbs1[_n_u32](svuint32_t op1, svuint32_t op2, uint32_t op3)
svuint64_t svbs1[_n_u64](svuint64_t op1, svuint64_t op2, uint64_t op3)

```

### 8.9.3. BSL1N: Bitwise select with first input inverted

These functions return a vector in which bit *N* contains the inverse of bit *N* of the first input when bit *N* of the third input is 1 and which contains bit *N* of the second input otherwise.

#### 8.9.3.1. BSL1N (vector, vector, vector)

**Instances**

```

svint8_t svbs11n[_s8](svint8_t op1, svint8_t op2, svint8_t op3)
svint16_t svbs11n[_s16](svint16_t op1, svint16_t op2, svint16_t op3)
svint32_t svbs11n[_s32](svint32_t op1, svint32_t op2, svint32_t op3)
svint64_t svbs11n[_s64](svint64_t op1, svint64_t op2, svint64_t op3)
svuint8_t svbs11n[_u8](svuint8_t op1, svuint8_t op2, svuint8_t op3)
svuint16_t svbs11n[_u16](svuint16_t op1, svuint16_t op2, svuint16_t op3)
svuint32_t svbs11n[_u32](svuint32_t op1, svuint32_t op2, svuint32_t op3)
svuint64_t svbs11n[_u64](svuint64_t op1, svuint64_t op2, svuint64_t op3)

```

#### 8.9.3.2. BSL1N (vector, vector, scalar)

**Instances**

```

svint8_t svbs11n[_n_s8](svint8_t op1, svint8_t op2, int8_t op3)
svint16_t svbs11n[_n_s16](svint16_t op1, svint16_t op2, int16_t op3)
svint32_t svbs11n[_n_s32](svint32_t op1, svint32_t op2, int32_t op3)
svint64_t svbs11n[_n_s64](svint64_t op1, svint64_t op2, int64_t op3)
svuint8_t svbs11n[_n_u8](svuint8_t op1, svuint8_t op2, uint8_t op3)
svuint16_t svbs11n[_n_u16](svuint16_t op1, svuint16_t op2, uint16_t op3)
svuint32_t svbs11n[_n_u32](svuint32_t op1, svuint32_t op2, uint32_t op3)
svuint64_t svbs11n[_n_u64](svuint64_t op1, svuint64_t op2, uint64_t op3)

```

### 8.9.4. BSL2N: Bitwise select with second input inverted

These functions return a vector in which bit *N* contains bit *N* of the first input when bit *N* of the third input is 1 and which contains the inverse of bit *N* of the second input otherwise.

#### 8.9.4.1. BSL2N (vector, vector, vector)

**Instances**

```

svint8_t svbs12n[_s8](svint8_t op1, svint8_t op2, svint8_t op3)
svint16_t svbs12n[_s16](svint16_t op1, svint16_t op2, svint16_t op3)
svint32_t svbs12n[_s32](svint32_t op1, svint32_t op2, svint32_t op3)
svint64_t svbs12n[_s64](svint64_t op1, svint64_t op2, svint64_t op3)
svuint8_t svbs12n[_u8](svuint8_t op1, svuint8_t op2, svuint8_t op3)
svuint16_t svbs12n[_u16](svuint16_t op1, svuint16_t op2, svuint16_t op3)
svuint32_t svbs12n[_u32](svuint32_t op1, svuint32_t op2, svuint32_t op3)
svuint64_t svbs12n[_u64](svuint64_t op1, svuint64_t op2, svuint64_t op3)

```

### 8.9.4.2. BSL2N (vector, vector, scalar)

#### Instances

```
svint8_t  svbsl2n[_n_s8](svint8_t op1, svint8_t op2, int8_t op3)
svint16_t svbsl2n[_n_s16](svint16_t op1, svint16_t op2, int16_t op3)
svint32_t svbsl2n[_n_s32](svint32_t op1, svint32_t op2, int32_t op3)
svint64_t svbsl2n[_n_s64](svint64_t op1, svint64_t op2, int64_t op3)
svuint8_t svbsl2n[_n_u8](svuint8_t op1, svuint8_t op2, uint8_t op3)
svuint16_t svbsl2n[_n_u16](svuint16_t op1, svuint16_t op2, uint16_t op3)
svuint32_t svbsl2n[_n_u32](svuint32_t op1, svuint32_t op2, uint32_t op3)
svuint64_t svbsl2n[_n_u64](svuint64_t op1, svuint64_t op2, uint64_t op3)
```

### 8.9.5. EOR3: Bitwise exclusive OR three vectors

These functions return the exclusive OR of three inputs.

#### 8.9.5.1. EOR3 (vector, vector, vector)

#### Instances

```
svint8_t  sveor3[_s8](svint8_t op1, svint8_t op2, svint8_t op3)
svint16_t sveor3[_s16](svint16_t op1, svint16_t op2, svint16_t op3)
svint32_t sveor3[_s32](svint32_t op1, svint32_t op2, svint32_t op3)
svint64_t sveor3[_s64](svint64_t op1, svint64_t op2, svint64_t op3)
svuint8_t sveor3[_u8](svuint8_t op1, svuint8_t op2, svuint8_t op3)
svuint16_t sveor3[_u16](svuint16_t op1, svuint16_t op2, svuint16_t op3)
svuint32_t sveor3[_u32](svuint32_t op1, svuint32_t op2, svuint32_t op3)
svuint64_t sveor3[_u64](svuint64_t op1, svuint64_t op2, svuint64_t op3)
```

#### 8.9.5.2. EOR3 (vector, vector, scalar)

#### Instances

```
svint8_t  sveor3[_n_s8](svint8_t op1, svint8_t op2, int8_t op3)
svint16_t sveor3[_n_s16](svint16_t op1, svint16_t op2, int16_t op3)
svint32_t sveor3[_n_s32](svint32_t op1, svint32_t op2, int32_t op3)
svint64_t sveor3[_n_s64](svint64_t op1, svint64_t op2, int64_t op3)
svuint8_t sveor3[_n_u8](svuint8_t op1, svuint8_t op2, uint8_t op3)
svuint16_t sveor3[_n_u16](svuint16_t op1, svuint16_t op2, uint16_t op3)
svuint32_t sveor3[_n_u32](svuint32_t op1, svuint32_t op2, uint32_t op3)
svuint64_t sveor3[_n_u64](svuint64_t op1, svuint64_t op2, uint64_t op3)
```

### 8.9.6. NBSL: Bitwise inverted select

These functions return the inverse of [Section 8.9.2, “BSL: Bitwise select”](#).

#### 8.9.6.1. NBSL (vector, vector, vector)

#### Instances

```
svint8_t  svnbsl[_s8](svint8_t op1, svint8_t op2, svint8_t op3)
svint16_t svnbsl[_s16](svint16_t op1, svint16_t op2, svint16_t op3)
svint32_t svnbsl[_s32](svint32_t op1, svint32_t op2, svint32_t op3)
svint64_t svnbsl[_s64](svint64_t op1, svint64_t op2, svint64_t op3)
svuint8_t svnbsl[_u8](svuint8_t op1, svuint8_t op2, svuint8_t op3)
svuint16_t svnbsl[_u16](svuint16_t op1, svuint16_t op2, svuint16_t op3)
svuint32_t svnbsl[_u32](svuint32_t op1, svuint32_t op2, svuint32_t op3)
svuint64_t svnbsl[_u64](svuint64_t op1, svuint64_t op2, svuint64_t op3)
```

### 8.9.6.2. NBSL (vector, vector, scalar)

Instances
svint8_t <b>svnbsl</b> [_n_s8](svint8_t op1, svint8_t op2, int8_t op3)
svint16_t <b>svnbsl</b> [_n_s16](svint16_t op1, svint16_t op2, int16_t op3)
svint32_t <b>svnbsl</b> [_n_s32](svint32_t op1, svint32_t op2, int32_t op3)
svint64_t <b>svnbsl</b> [_n_s64](svint64_t op1, svint64_t op2, int64_t op3)
svuint8_t <b>svnbsl</b> [_n_u8](svuint8_t op1, svuint8_t op2, uint8_t op3)
svuint16_t <b>svnbsl</b> [_n_u16](svuint16_t op1, svuint16_t op2, uint16_t op3)
svuint32_t <b>svnbsl</b> [_n_u32](svuint32_t op1, svuint32_t op2, uint32_t op3)
svuint64_t <b>svnbsl</b> [_n_u64](svuint64_t op1, svuint64_t op2, uint64_t op3)

### 8.9.7. XAR: Bitwise exclusive OR and rotate right by immediate

These functions compute the exclusive OR of the first two inputs and then rotate the result right by the number of bits specified by the third input. If the elements have  $N$  bits, the third input must be an integer constant expression in the range  $[1, N]$ .

#### 8.9.7.1. XAR (vector, vector, immediate)

Instances
svint8_t <b>svxar</b> [_n_s8](svint8_t op1, svint8_t op2, uint64_t imm3)
svint16_t <b>svxar</b> [_n_s16](svint16_t op1, svint16_t op2, uint64_t imm3)
svint32_t <b>svxar</b> [_n_s32](svint32_t op1, svint32_t op2, uint64_t imm3)
svint64_t <b>svxar</b> [_n_s64](svint64_t op1, svint64_t op2, uint64_t imm3)
svuint8_t <b>svxar</b> [_n_u8](svuint8_t op1, svuint8_t op2, uint64_t imm3)
svuint16_t <b>svxar</b> [_n_u16](svuint16_t op1, svuint16_t op2, uint64_t imm3)
svuint32_t <b>svxar</b> [_n_u32](svuint32_t op1, svuint32_t op2, uint64_t imm3)
svuint64_t <b>svxar</b> [_n_u64](svuint64_t op1, svuint64_t op2, uint64_t imm3)

## 8.10. Large integer arithmetic

### 8.10.1. ADCLB: Integer addition with carry (bottom)

These functions take and return pairs of values: an even-indexed element followed by an odd-indexed element. For each pair of the result, the functions take the even-indexed first input, the even-indexed second input, and the low bit of the odd-indexed third input. They add the three values together using natural arithmetic, then store the low part of the result in the even-indexed element and a carry value (0 or 1) in the odd-indexed element.

#### 8.10.1.1. ADCLB (vector, vector, vector)

Instances
svuint32_t <b>svadclb</b> [_u32](svuint32_t op1, svuint32_t op2, svuint32_t op3)
svuint64_t <b>svadclb</b> [_u64](svuint64_t op1, svuint64_t op2, svuint64_t op3)

#### 8.10.1.2. ADCLB (vector, vector, scalar)

Instances
svuint32_t <b>svadclb</b> [_n_u32](svuint32_t op1, svuint32_t op2, uint32_t op3)
svuint64_t <b>svadclb</b> [_n_u64](svuint64_t op1, svuint64_t op2, uint64_t op3)

## 8.10.2. ADCLT: Integer addition with carry (top)

These functions take and return pairs of values: an even-indexed element followed by an odd-indexed element. For each pair of the result, the functions take the even-indexed first input, the odd-indexed second input, and the low bit of the odd-indexed third input. They add the three values together using natural arithmetic, then store the low part of the result in the even-indexed element and a carry value (0 or 1) in the odd-indexed element.

### 8.10.2.1. ADCLT (vector, vector, vector)

Instances	
svuint32_t	<b>svadclt</b> [_u32](svuint32_t op1, svuint32_t op2, svuint32_t op3)
svuint64_t	<b>svadclt</b> [_u64](svuint64_t op1, svuint64_t op2, svuint64_t op3)

### 8.10.2.2. ADCLT (vector, vector, scalar)

Instances	
svuint32_t	<b>svadclt</b> [_n_u32](svuint32_t op1, svuint32_t op2, uint32_t op3)
svuint64_t	<b>svadclt</b> [_n_u64](svuint64_t op1, svuint64_t op2, uint64_t op3)

## 8.10.3. SBCLB: Integer subtraction with carry (bottom)

These functions take and return pairs of values: an even-indexed element followed by an odd-indexed element. For each pair of the result, the functions add the even-indexed second input and the inverse of the low bit of the odd-indexed third input and subtract the result from the even-indexed first input, all using natural arithmetic. They then store the low part of the result in the even-indexed element and a carry value (0 or 1) in the odd-indexed element.

### 8.10.3.1. SBCLB (vector, vector, vector)

Instances	
svuint32_t	<b>svsbc1b</b> [_u32](svuint32_t op1, svuint32_t op2, svuint32_t op3)
svuint64_t	<b>svsbc1b</b> [_u64](svuint64_t op1, svuint64_t op2, svuint64_t op3)

### 8.10.3.2. SBCLB (vector, vector, scalar)

Instances	
svuint32_t	<b>svsbc1b</b> [_n_u32](svuint32_t op1, svuint32_t op2, uint32_t op3)
svuint64_t	<b>svsbc1b</b> [_n_u64](svuint64_t op1, svuint64_t op2, uint64_t op3)

## 8.10.4. SBCLT: Integer subtraction with carry (top)

These functions take and return pairs of values: an even-indexed element followed by an odd-indexed element. For each pair of the result, the functions add the odd-indexed second input and the inverse of the low bit of the odd-indexed third input and subtract the result from the even-indexed first input, all using natural arithmetic. They then store the low part of the result in the even-indexed element and a carry value (0 or 1) in the odd-indexed element.

### 8.10.4.1. SBCLT (vector, vector, vector)

Instances	
svuint32_t	<b>svsbclt</b> [_u32](svuint32_t op1, svuint32_t op2, svuint32_t op3)

Instances
svuint64_t <b>svsbclt</b> [_u64](svuint64_t op1, svuint64_t op2, svuint64_t op3)

#### 8.10.4.2. SBCLT (vector, vector, scalar)

Instances
svuint32_t <b>svsbclt</b> [_n_u32](svuint32_t op1, svuint32_t op2, uint32_t op3)
svuint64_t <b>svsbclt</b> [_n_u64](svuint64_t op1, svuint64_t op2, uint64_t op3)

## 8.11. Multiplication by indexed elements

### 8.11.1. MUL: Integer multiplication

These functions provide additional forms of integer multiplication; see [Section 6.7.7, “MUL: Integer multiplication, returning low half”](#) for the original SVE instructions.

The `_lane` forms of the functions take one element in each 128-bit quadword of the second input and replicate it to fill the rest of the quadword. The `imm_index` parameter specifies the element within each quadword that the functions should replicate. This index must be an integer constant expression in the range  $[0, 128/N]$ , where  $N$  is the number of bits in each element.

#### 8.11.1.1. MUL (vector, vector, lane)

Instances
svint16_t <b>svmul_lane</b> [_s16](svint16_t op1, svint16_t op2, uint64_t imm_index)
svint32_t <b>svmul_lane</b> [_s32](svint32_t op1, svint32_t op2, uint64_t imm_index)
svint64_t <b>svmul_lane</b> [_s64](svint64_t op1, svint64_t op2, uint64_t imm_index)
svuint16_t <b>svmul_lane</b> [_u16](svuint16_t op1, svuint16_t op2, uint64_t imm_index)
svuint32_t <b>svmul_lane</b> [_u32](svuint32_t op1, svuint32_t op2, uint64_t imm_index)
svuint64_t <b>svmul_lane</b> [_u64](svuint64_t op1, svuint64_t op2, uint64_t imm_index)

### 8.11.2. MLA: Integer addition of product (addend first)

These functions provide additional forms of the functions described in [Section 6.7.10, “MLA: Integer addition of product \(addend first\)”](#).

The `_lane` forms of the functions take one element in each 128-bit quadword of the third input and replicate it to fill the rest of the quadword. The `imm_index` parameter specifies the element within each quadword that the functions should replicate. This index must be an integer constant expression in the range  $[0, 128/N]$ , where  $N$  is the number of bits in each element.

#### 8.11.2.1. MLA (vector, vector, vector, lane)

Instances
svint16_t <b>svmla_lane</b> [_s16](svint16_t op1, svint16_t op2, svint16_t op3, uint64_t imm_index)
svint32_t <b>svmla_lane</b> [_s32](svint32_t op1, svint32_t op2, svint32_t op3, uint64_t imm_index)

**Instances**

```

uint64_t imm_index)
svint64_t svmla_lane[_s64](svint64_t op1, svint64_t op2, svint64_t op3,
uint64_t imm_index)
svuint16_t svmla_lane[_u16](svuint16_t op1, svuint16_t op2, svuint16_t op3,
uint64_t imm_index)
svuint32_t svmla_lane[_u32](svuint32_t op1, svuint32_t op2, svuint32_t op3,
uint64_t imm_index)
svuint64_t svmla_lane[_u64](svuint64_t op1, svuint64_t op2, svuint64_t op3,
uint64_t imm_index)

```

**8.11.3. MLS: Integer subtraction of product (addend first)**

These functions provide additional forms of the functions described in [Section 6.7.12, “MLS: Integer subtraction of product \(minuend first\)”](#).

The `_lane` forms of the functions take one element in each 128-bit quadword of the third input and replicate it to fill the rest of the quadword. The `imm_index` parameter specifies the element within each quadword that the functions should replicate. This index must be an integer constant expression in the range  $[0, 128/N]$ , where  $N$  is the number of bits in each element.

**8.11.3.1. MLS (vector, vector, vector, lane)****Instances**

```

svint16_t svmls_lane[_s16](svint16_t op1, svint16_t op2, svint16_t op3,
uint64_t imm_index)
svint32_t svmls_lane[_s32](svint32_t op1, svint32_t op2, svint32_t op3,
uint64_t imm_index)
svint64_t svmls_lane[_s64](svint64_t op1, svint64_t op2, svint64_t op3,
uint64_t imm_index)
svuint16_t svmls_lane[_u16](svuint16_t op1, svuint16_t op2, svuint16_t op3,
uint64_t imm_index)
svuint32_t svmls_lane[_u32](svuint32_t op1, svuint32_t op2, svuint32_t op3,
uint64_t imm_index)
svuint64_t svmls_lane[_u64](svuint64_t op1, svuint64_t op2, svuint64_t op3,
uint64_t imm_index)

```

**8.12. Uniform complex integer arithmetic****8.12.1. CADD: Integer complex addition with rotation**

These functions take and return complex integer values, with the real components in even elements and the imaginary components in odd elements. The functions rotate the second complex input by the number of degrees specified by the final (rotation) input and then add the result to the first complex input. The rotation input must be an integer constant expression with the value 90 or 270.

The addition uses modular arithmetic and the result is well-defined for all inputs; unlike C, there is no undefined behavior for signed overflow.

**8.12.1.1. CADD (vector, vector, immediate)****Instances**

```

svint8_t svcadd[_s8](svint8_t op1, svint8_t op2, uint64_t imm_rotation)
svint16_t svcadd[_s16](svint16_t op1, svint16_t op2, uint64_t imm_rotation)

```

Instances	
<code>svint32_t</code>	<code>svcadd[_s32](svint32_t op1, svint32_t op2, uint64_t imm_rotation)</code>
<code>svint64_t</code>	<code>svcadd[_s64](svint64_t op1, svint64_t op2, uint64_t imm_rotation)</code>
<code>svuint8_t</code>	<code>svcadd[_u8](svuint8_t op1, svuint8_t op2, uint64_t imm_rotation)</code>
<code>svuint16_t</code>	<code>svcadd[_u16](svuint16_t op1, svuint16_t op2, uint64_t imm_rotation)</code>
<code>svuint32_t</code>	<code>svcadd[_u32](svuint32_t op1, svuint32_t op2, uint64_t imm_rotation)</code>
<code>svuint64_t</code>	<code>svcadd[_u64](svuint64_t op1, svuint64_t op2, uint64_t imm_rotation)</code>

## 8.12.2. QCADD: Integer complex saturating addition with rotation

These functions take and return complex integer values, with the real components in even elements and the imaginary components in odd elements. The functions rotate the second complex input by the number of degrees specified by the final (rotation) input, add the result to the first complex input, and then saturate the result to be in range. The rotation input must be an integer constant expression with the value 90 or 270.

### 8.12.2.1. QCADD (vector, vector, immediate)

Instances	
<code>svint8_t</code>	<code>svqcadd[_s8](svint8_t op1, svint8_t op2, uint64_t imm_rotation)</code>
<code>svint16_t</code>	<code>svqcadd[_s16](svint16_t op1, svint16_t op2, uint64_t imm_rotation)</code>
<code>svint32_t</code>	<code>svqcadd[_s32](svint32_t op1, svint32_t op2, uint64_t imm_rotation)</code>
<code>svint64_t</code>	<code>svqcadd[_s64](svint64_t op1, svint64_t op2, uint64_t imm_rotation)</code>

## 8.12.3. CMLA: Integer complex addition of product with rotation

These functions take and return complex integer values, with the real components in even elements and the imaginary components in odd elements. The functions rotate the third complex input by the number of degrees specified by the final (rotation) input, multiply the result by one component of the second complex input, then add the result to the first complex input.

The rotation input must be an integer constant expression with the value 0, 90, 180 or 270. When the rotation value is 0 or 180, the multiplication selects the real components of the second input, otherwise it selects the imaginary components.

The multiplication and addition use modular arithmetic and the result is well-defined for all inputs; unlike C, there is no undefined behavior for signed overflow.

The `_lane` forms of the functions take one complex value in each 128-bit quadword of the third input and replicate it to fill the rest of the quadword. The `imm_index` parameter specifies the complex value within each quadword that the functions should replicate. This index must be an integer constant expression in the range  $[0, 128/N)$ , where  $N$  is the number of bits in each complex value.

### 8.12.3.1. CMLA (vector, vector, vector, immediate)

Instances	
<code>svint8_t</code>	<code>svcmla[_s8](svint8_t op1, svint8_t op2, svint8_t op3, uint64_t imm_rotation)</code>
<code>svint16_t</code>	<code>svcmla[_s16](svint16_t op1, svint16_t op2, svint16_t op3, uint64_t imm_rotation)</code>



**Instances**

```
svint32_t svcmcla[_s32](svint32_t op1, svint32_t op2, svint32_t op3,
                        uint64_t imm_rotation)
svint64_t svcmcla[_s64](svint64_t op1, svint64_t op2, svint64_t op3,
                        uint64_t imm_rotation)
svuint8_t svcmcla[_u8](svuint8_t op1, svuint8_t op2, svuint8_t op3,
                       uint64_t imm_rotation)
svuint16_t svcmcla[_u16](svuint16_t op1, svuint16_t op2, svuint16_t op3,
                        uint64_t imm_rotation)
svuint32_t svcmcla[_u32](svuint32_t op1, svuint32_t op2, svuint32_t op3,
                        uint64_t imm_rotation)
svuint64_t svcmcla[_u64](svuint64_t op1, svuint64_t op2, svuint64_t op3,
                        uint64_t imm_rotation)
```

**8.12.3.2. CMLA (vector, vector, vector, lane, immediate)****Instances**

```
svint16_t svcmcla_lane[_s16](svint16_t op1, svint16_t op2, svint16_t op3,
                             uint64_t imm_index, uint64_t imm_rotation)
svint32_t svcmcla_lane[_s32](svint32_t op1, svint32_t op2, svint32_t op3,
                             uint64_t imm_index, uint64_t imm_rotation)
svuint16_t svcmcla_lane[_u16](svuint16_t op1, svuint16_t op2, svuint16_t op3,
                              uint64_t imm_index, uint64_t imm_rotation)
svuint32_t svcmcla_lane[_u32](svuint32_t op1, svuint32_t op2, svuint32_t op3,
                              uint64_t imm_index, uint64_t imm_rotation)
```

**8.12.4. QRDCMLAH: Doubling complex integer multiplication with rotation, saturating addition of rounded high half**

These functions take and return complex integer values, with the real components in even elements and the imaginary components in odd elements. The functions rotate the third complex input by the number of degrees specified by the final (rotation) input, multiply the result by one component of the second complex input, divide their product by  $2^{N-1}$ , round the quotient to nearest with ties towards +Inf, add the rounded quotient to the first complex input, and then saturate the result to be in range. All operations prior to the saturation use natural arithmetic.

The rotation input must be an integer constant expression with the value 0, 90, 180 or 270. When the rotation value is 0 or 180, the multiplication selects the real components of the second input, otherwise it selects the imaginary components.

The multiplication and addition use modular arithmetic and the result is well-defined for all inputs; unlike C, there is no undefined behavior for signed overflow.

The `_lane` forms of the functions take one complex value in each 128-bit quadword of the third input and replicate it to fill the rest of the quadword. The `imm_index` parameter specifies the complex value within each quadword that the functions should replicate. This index must be an integer constant expression in the range  $[0, 128/N)$ , where  $N$  is the number of bits in each complex value.

**8.12.4.1. QRDCMLAH (vector, vector, vector, immediate)****Instances**

```
svint8_t svqrdcmclah[_s8](svint8_t op1, svint8_t op2, svint8_t op3,
                           uint64_t imm_rotation)
svint16_t svqrdcmclah[_s16](svint16_t op1, svint16_t op2, svint16_t op3,
```

Instances	
<code>svint32_t</code>	<code>svqrdclah[_s32](svint32_t op1, svint32_t op2, svint32_t op3, uint64_t imm_rotation)</code>
<code>svint64_t</code>	<code>svqrdclah[_s64](svint64_t op1, svint64_t op2, svint64_t op3, uint64_t imm_rotation)</code>

#### 8.12.4.2. QRDCMLAH (vector, vector, vector, lane, immediate)

Instances	
<code>svint16_t</code>	<code>svqrdclah_lane[_s16](svint16_t op1, svint16_t op2, svint16_t op3, uint64_t imm_index, uint64_t imm_rotation)</code>
<code>svint32_t</code>	<code>svqrdclah_lane[_s32](svint32_t op1, svint32_t op2, svint32_t op3, uint64_t imm_index, uint64_t imm_rotation)</code>

## 8.13. Widening complex integer arithmetic

### 8.13.1. ADDLBT: Integer addition long (bottom + top)

These functions add the even-indexed elements of the first integer input to the odd-indexed elements of the second integer input, giving a result with twice the width.

#### 8.13.1.1. ADDLBT (vector, vector)

Instances	
<code>svint16_t</code>	<code>svaddlbt[_s16](svint8_t op1, svint8_t op2)</code>
<code>svint32_t</code>	<code>svaddlbt[_s32](svint16_t op1, svint16_t op2)</code>
<code>svint64_t</code>	<code>svaddlbt[_s64](svint32_t op1, svint32_t op2)</code>

#### 8.13.1.2. ADDLBT (vector, scalar)

Instances	
<code>svint16_t</code>	<code>svaddlbt[_n_s16](svint8_t op1, int8_t op2)</code>
<code>svint32_t</code>	<code>svaddlbt[_n_s32](svint16_t op1, int16_t op2)</code>
<code>svint64_t</code>	<code>svaddlbt[_n_s64](svint32_t op1, int32_t op2)</code>

### 8.13.2. SUBLBT: Integer subtraction long (bottom - top)

These functions subtract the odd-indexed elements of the second integer input from the even-indexed elements of the first integer input, giving a result with twice the width.

#### 8.13.2.1. SUBLBT (vector, vector)

Instances	
<code>svint16_t</code>	<code>svsublbt[_s16](svint8_t op1, svint8_t op2)</code>
<code>svint32_t</code>	<code>svsublbt[_s32](svint16_t op1, svint16_t op2)</code>
<code>svint64_t</code>	<code>svsublbt[_s64](svint32_t op1, svint32_t op2)</code>

#### 8.13.2.2. SUBLBT (vector, scalar)

Instances	
<code>svint16_t</code>	<code>svsublbt[_n_s16](svint8_t op1, int8_t op2)</code>

**Instances**

```
svint32_t svsublbt[_n_s32](svint16_t op1, int16_t op2)
svint64_t svsublbt[_n_s64](svint32_t op1, int32_t op2)
```

**8.13.3. SUBLTB: Integer subtraction long (top - bottom)**

These functions subtract the even-indexed elements of the second integer input from the odd-indexed elements of the first integer input, giving a result with twice the width.

**8.13.3.1. SUBLTB (vector, vector)****Instances**

```
svint16_t svsubltb[_s16](svint8_t op1, svint8_t op2)
svint32_t svsubltb[_s32](svint16_t op1, svint16_t op2)
svint64_t svsubltb[_s64](svint32_t op1, svint32_t op2)
```

**8.13.3.2. SUBLTB (vector, scalar)****Instances**

```
svint16_t svsubltb[_n_s16](svint8_t op1, int8_t op2)
svint32_t svsubltb[_n_s32](svint16_t op1, int16_t op2)
svint64_t svsubltb[_n_s64](svint32_t op1, int32_t op2)
```

**8.13.4. QDMLALBT: Saturating integer addition of doubled product long (bottom × top)**

These functions multiply the even-indexed elements of the second input by the odd-indexed elements of the third input, giving one double-width product for each element of the first input. They then double this product and add the doubled product to the first input. The doubling and addition use saturating arithmetic; if the result of the operation is outside the range of the first input's type, they clamp it to the nearest in-range value.

**8.13.4.1. QDMLALBT (vector, vector, vector)****Instances**

```
svint16_t svqdmlalbt[_s16](svint16_t op1, svint8_t op2, svint8_t op3)
svint32_t svqdmlalbt[_s32](svint32_t op1, svint16_t op2, svint16_t op3)
svint64_t svqdmlalbt[_s64](svint64_t op1, svint32_t op2, svint32_t op3)
```

**8.13.4.2. QDMLALBT (vector, vector, scalar)****Instances**

```
svint16_t svqdmlalbt[_n_s16](svint16_t op1, svint8_t op2, int8_t op3)
svint32_t svqdmlalbt[_n_s32](svint32_t op1, svint16_t op2, int16_t op3)
svint64_t svqdmlalbt[_n_s64](svint64_t op1, svint32_t op2, int32_t op3)
```

**8.13.5. QDMLSLBT: Saturating integer subtraction of doubled product long (bottom × top)**

These functions multiply the even-indexed elements of the second input by the odd-indexed elements of the third input, giving one double-width product for each element of the first input. They then double this

product and subtract the doubled product from the first input. The doubling and subtraction use saturating arithmetic; if the result of the operation is outside the range of the first input's type, they clamp it to the nearest in-range value.

### 8.13.5.1. QDMLSLBT (vector, vector, vector)

Instances
svint16_t <b>svqdmmlslbt</b> [_s16](svint16_t op1, svint8_t op2, svint8_t op3)
svint32_t <b>svqdmmlslbt</b> [_s32](svint32_t op1, svint16_t op2, svint16_t op3)
svint64_t <b>svqdmmlslbt</b> [_s64](svint64_t op1, svint32_t op2, svint32_t op3)

### 8.13.5.2. QDMLSLBT (vector, vector, scalar)

Instances
svint16_t <b>svqdmmlslbt</b> [_n_s16](svint16_t op1, svint8_t op2, int8_t op3)
svint32_t <b>svqdmmlslbt</b> [_n_s32](svint32_t op1, svint16_t op2, int16_t op3)
svint64_t <b>svqdmmlslbt</b> [_n_s64](svint64_t op1, svint32_t op2, int32_t op3)

## 8.14. Complex integer dot product

### 8.14.1. CDOT: Integer complex dot-product

The second and third inputs represent complex integer values, with the real components in even elements and the imaginary components in odd elements. The first input and the return value contain scalar integers. There are two complex values for each element of the first input, and for each element of the result.

The functions rotate the third input by the number of degrees specified by the final (rotation) input, multiply the result by one component of the second input, giving a result that is four times wider. There are two such results for each element of the first input. The functions then add these two results and the first input together.

The rotation input must be an integer constant expression with the value 0, 90, 180 or 270. When the rotation value is 0 or 180, the multiplication selects the real components of the second input, otherwise it selects the imaginary components.

The additions use modular arithmetic and the result is well-defined for all inputs; unlike C, there is no undefined behavior for signed overflow.

#### 8.14.1.1. CDOT (vector, vector, vector, immediate)

Instances
svint32_t <b>svcdot</b> [_s32](svint32_t op1, svint8_t op2, svint8_t op3, uint64_t imm_rotation)
svint64_t <b>svcdot</b> [_s64](svint64_t op1, svint16_t op2, svint16_t op3, uint64_t imm_rotation)

#### 8.14.1.2. CDOT (vector, vector, vector, lane, immediate)

Instances
svint32_t <b>svcdot_lane</b> [_s32](svint32_t op1, svint8_t op2, svint8_t op3, uint64_t imm_index, uint64_t imm_rotation)
svint64_t <b>svcdot_lane</b> [_s64](svint64_t op1, svint16_t op2, svint16_t op3, uint64_t imm_index, uint64_t imm_rotation)

Instances
<code>uint64_t imm_index, uint64_t imm_rotation)</code>

## 8.15. Extra floating-point conversions

### 8.15.1. CVTLT: Convert floating-point value to wider type (top)

These functions extend the odd-indexed values of the input to a wider type. See [Section 6.19.3, “CVT: Convert floating-point value to wider type”](#) for the corresponding operation on even-indexed values.

Signaling NaNs trigger an IEEE Invalid exception but quiet NaNs do not.

#### 8.15.1.1. CVTLT (vector), merging with separate vector

Instances
<code>svfloat32_t svcvltt_f32[_f16]_m(svfloat32_t inactive, svbool_t pg, svfloat16_t op)</code>
<code>svfloat64_t svcvltt_f64[_f32]_m(svfloat64_t inactive, svbool_t pg, svfloat32_t op)</code>

#### 8.15.1.2. CVTLT (vector), setting inactive to unknown

Instances
<code>svfloat32_t svcvltt_f32[_f16]_x(svbool_t pg, svfloat16_t op)</code>
<code>svfloat64_t svcvltt_f64[_f32]_x(svbool_t pg, svfloat32_t op)</code>

### 8.15.2. CVTNT: Convert floating-point value to narrower type (top)

These functions convert the second floating-point input to a narrower type, rounding according to the current rounding mode, and store the converted values in the odd-indexed elements of the result. The even-indexed elements of the first input supply the corresponding elements of the result.

The functions described in [Section 6.19.4, “CVT: Convert floating-point value to narrower type”](#) provide the equivalent “bottom” operation.

Signaling NaNs trigger an IEEE Invalid exception but quiet NaNs do not.

#### 8.15.2.1. CVTNT (vector), merging with even vector

Instances
<code>svfloat16_t svcvntt_f16[_f32]_m(svfloat16_t even, svbool_t pg, svfloat32_t op)</code>
<code>svfloat32_t svcvntt_f32[_f64]_m(svfloat32_t even, svbool_t pg, svfloat64_t op)</code>

#### 8.15.2.2. CVTNT (vector), setting inactive to unknown

Instances
<code>svfloat16_t svcvntt_f16[_f32]_x(svfloat16_t even, svbool_t pg, svfloat32_t op)</code>
<code>svfloat32_t svcvntt_f32[_f64]_x(svfloat32_t even, svbool_t pg,</code>

Instances
<code>svfloat64_t op)</code>

### 8.15.3. CVTX: Convert floating-point value to narrower type, rounding to odd

These functions convert a floating-point value to a narrower type, rounding to odd.

Signaling NaNs trigger an IEEE Invalid exception but quiet NaNs do not.

#### 8.15.3.1. CVTX (vector), setting inactive to zero

Instances
<code>svfloat32_t <b>svcvtx_f32</b>[_f64]_z(svbool_t pg, svfloat64_t op)</code>

#### 8.15.3.2. CVTX (vector), merging with separate vector

Instances
<code>svfloat32_t <b>svcvtx_f32</b>[_f64]_m(svfloat32_t inactive, svbool_t pg, svfloat64_t op)</code>

#### 8.15.3.3. CVTX (vector), setting inactive to unknown

Instances
<code>svfloat32_t <b>svcvtx_f32</b>[_f64]_x(svbool_t pg, svfloat64_t op)</code>

### 8.15.4. CVTXNT: Convert floating-point value to narrower type, rounding to odd (top)

These functions convert the second floating-point input to a narrower type, rounding to odd, and store the results in the odd-indexed elements of the result. The even-indexed elements of the first input supply the corresponding elements of the result.

The functions described in [Section 8.15.3, “CVTX: Convert floating-point value to narrower type, rounding to odd”](#) provide the equivalent “bottom” operation.

Signaling NaNs trigger an IEEE Invalid exception but quiet NaNs do not.

#### 8.15.4.1. CVTXNT (vector), merging with even vector

Instances
<code>svfloat32_t <b>svcvtxnt_f32</b>[_f64]_m(svfloat32_t even, svbool_t pg, svfloat64_t op)</code>

#### 8.15.4.2. CVTXNT (vector), setting inactive to unknown

Instances
<code>svfloat32_t <b>svcvtxnt_f32</b>[_f64]_x(svfloat32_t even, svbool_t pg, svfloat64_t op)</code>

## 8.16. Floating-point widening multiply-accumulate

### 8.16.1. MLALB: Floating-point addition of product long (bottom)

These functions extract the even-indexed elements of the second and third vectors, giving the same number of elements as the first. They then extend the second and third inputs, multiply the results, and add the product to the first input. There is no intermediate rounding step after the multiplication.

The `_lane` forms of the functions take one element in each 128-bit quadword of the third input and replicate it to fill the rest of the quadword. The `imm_index` parameter specifies the element within each quadword that the functions should replicate. This index must be an integer constant expression in the range  $[0, 128/N)$ , where  $N$  is the number of bits in each element.

Signaling NaNs trigger an IEEE Invalid exception but quiet NaNs do not. Multiplying an infinity by zero also triggers an IEEE Invalid exception, as does adding `+Inf` and `-Inf` together.

#### 8.16.1.1. MLALB (vector, vector, vector)

Instances
<pre>svfloat32_t svmlalb[_f32](svfloat32_t op1, svfloat16_t op2,                           svfloat16_t op3)</pre>

#### 8.16.1.2. MLALB (vector, vector, scalar)

Instances
<pre>svfloat32_t svmlalb[_n_f32](svfloat32_t op1, svfloat16_t op2,                              float16_t op3)</pre>

#### 8.16.1.3. MLALB (vector, vector, vector, lane)

Instances
<pre>svfloat32_t svmlalb_lane[_f32](svfloat32_t op1, svfloat16_t op2,                                 svfloat16_t op3, uint64_t imm_index)</pre>

### 8.16.2. MLALT: Floating-point addition of product long (top)

These functions extract the odd-indexed elements of the second and third vectors, giving the same number of elements as the first. They then extend the second and third inputs, multiply the results, and add the product to the first input. There is no intermediate rounding step after the multiplication.

The `_lane` forms of the functions take one element in each 128-bit quadword of the third input and replicate it to fill the rest of the quadword. The `imm_index` parameter specifies the element within each quadword that the functions should replicate. This index must be an integer constant expression in the range  $[0, 128/N)$ , where  $N$  is the number of bits in each element.

Signaling NaNs trigger an IEEE Invalid exception but quiet NaNs do not. Multiplying an infinity by zero also triggers an IEEE Invalid exception, as does adding `+Inf` and `-Inf` together.

#### 8.16.2.1. MLALT (vector, vector, vector)

Instances
<pre>svfloat32_t svmlalt[_f32](svfloat32_t op1, svfloat16_t op2,                           svfloat16_t op3)</pre>

### 8.16.2.2. MLALT (vector, vector, scalar)

Instances
svfloat32_t <b>svmlalt</b> [_n_f32](svfloat32_t op1, svfloat16_t op2, float16_t op3)

### 8.16.2.3. MLALT (vector, vector, vector, lane)

Instances
svfloat32_t <b>svmlalt_lane</b> [_f32](svfloat32_t op1, svfloat16_t op2, svfloat16_t op3, uint64_t imm_index)

## 8.16.3. MLSLB: Floating-point subtraction of product long (bottom)

These functions extract the even-indexed elements of the second and third vectors, giving the same number of elements as the first. They then extend the second and third inputs, multiply the results, and subtract the product from the first input. There is no intermediate rounding step after the multiplication.

The `_lane` forms of the functions take one element in each 128-bit quadword of the third input and replicate it to fill the rest of the quadword. The `imm_index` parameter specifies the element within each quadword that the functions should replicate. This index must be an integer constant expression in the range  $[0, 128/N]$ , where  $N$  is the number of bits in each element.

Signaling NaNs trigger an IEEE Invalid exception but quiet NaNs do not. Multiplying an infinity by zero also triggers an IEEE Invalid exception, as does adding `+Inf` and `-Inf` together.

### 8.16.3.1. MLSLB (vector, vector, vector)

Instances
svfloat32_t <b>svmlslb</b> [_f32](svfloat32_t op1, svfloat16_t op2, svfloat16_t op3)

### 8.16.3.2. MLSLB (vector, vector, scalar)

Instances
svfloat32_t <b>svmlslb</b> [_n_f32](svfloat32_t op1, svfloat16_t op2, float16_t op3)

### 8.16.3.3. MLSLB (vector, vector, vector, lane)

Instances
svfloat32_t <b>svmlslb_lane</b> [_f32](svfloat32_t op1, svfloat16_t op2, svfloat16_t op3, uint64_t imm_index)

## 8.16.4. MLSLT: Floating-point subtraction of product long (top)

These functions extract the odd-indexed elements of the second and third vectors, giving the same number of elements as the first. They then extend the second and third inputs, multiply the results, and subtract the product from the first input. There is no intermediate rounding step after the multiplication.

The `_lane` forms of the functions take one element in each 128-bit quadword of the third input and replicate it to fill the rest of the quadword. The `imm_index` parameter specifies the element within each quadword that the functions should replicate. This index must be an integer constant expression in the range  $[0, 128/N]$ , where  $N$  is the number of bits in each element.



Signaling NaNs trigger an IEEE Invalid exception but quiet NaNs do not. Multiplying an infinity by zero also triggers an IEEE Invalid exception, as does adding +Inf and -Inf together.

#### 8.16.4.1. MLSLT (vector, vector, vector)

Instances
svfloat32_t <b>svmlslt</b> [_f32](svfloat32_t op1, svfloat16_t op2, svfloat16_t op3)

#### 8.16.4.2. MLSLT (vector, vector, scalar)

Instances
svfloat32_t <b>svmlslt</b> [_n_f32](svfloat32_t op1, svfloat16_t op2, float16_t op3)

#### 8.16.4.3. MLSLT (vector, vector, vector, lane)

Instances
svfloat32_t <b>svmlslt_lane</b> [_f32](svfloat32_t op1, svfloat16_t op2, svfloat16_t op3, uint64_t imm_index)

## 8.17. Floating-point integer binary logarithm

### 8.17.1. LOGB: Floating-point base 2 logarithm as integer

These functions return the base 2 logarithm of the absolute value of the input. If the input is a NaN or zero, the result is the return type's minimum value. If the input is infinite, the result is the return type's maximum value.

NaNs and zeros trigger an IEEE Invalid exception.

#### 8.17.1.1. LOGB (vector), setting inactive to zero

Instances
svint16_t <b>svlogb</b> [_f16]_z(svbool_t pg, svfloat16_t op)
svint32_t <b>svlogb</b> [_f32]_z(svbool_t pg, svfloat32_t op)
svint64_t <b>svlogb</b> [_f64]_z(svbool_t pg, svfloat64_t op)

#### 8.17.1.2. LOGB (vector), merging with separate vector

Instances
svint16_t <b>svlogb</b> [_f16]_m(svint16_t inactive, svbool_t pg, svfloat16_t op)
svint32_t <b>svlogb</b> [_f32]_m(svint32_t inactive, svbool_t pg, svfloat32_t op)
svint64_t <b>svlogb</b> [_f64]_m(svint64_t inactive, svbool_t pg, svfloat64_t op)

#### 8.17.1.3. LOGB (vector), setting inactive to unknown

Instances
svint16_t <b>svlogb</b> [_f16]_x(svbool_t pg, svfloat16_t op)
svint32_t <b>svlogb</b> [_f32]_x(svbool_t pg, svfloat32_t op)
svint64_t <b>svlogb</b> [_f64]_x(svbool_t pg, svfloat64_t op)

## 8.18. Vector histogram count

### 8.18.1. HISTCNT: Count matching elements in vector

These functions compare each active element of the first input with all active elements up to the same position in the second input and return the number of matches.

#### 8.18.1.1. HISTCNT (vector, vector), setting inactive to zero

Instances	
<code>svuint32_t</code>	<code>svhistcnt[_s32]_z(svbool_t pg, svint32_t op1, svint32_t op2)</code>
<code>svuint64_t</code>	<code>svhistcnt[_s64]_z(svbool_t pg, svint64_t op1, svint64_t op2)</code>
<code>svuint32_t</code>	<code>svhistcnt[_u32]_z(svbool_t pg, svuint32_t op1, svuint32_t op2)</code>
<code>svuint64_t</code>	<code>svhistcnt[_u64]_z(svbool_t pg, svuint64_t op1, svuint64_t op2)</code>

### 8.18.2. HISTSEG: Count matching elements in vector segments

These functions compare each element of the first input with all elements of the same quadword in the second input and return the number of matches.

#### 8.18.2.1. HISTSEG (vector, vector)

Instances	
<code>svuint8_t</code>	<code>svhistseg[_s8](svint8_t op1, svint8_t op2)</code>
<code>svuint8_t</code>	<code>svhistseg[_u8](svuint8_t op1, svuint8_t op2)</code>

## 8.19. Character match

### 8.19.1. MATCH: Detect any matching elements

These functions compare each active element of the first input with all elements of the same quadword in the second input and return a predicate bit that indicates whether there was at least one match.

#### 8.19.1.1. MATCH (vector, vector)

Instances	
<code>svbool_t</code>	<code>svmatch[_s8](svbool_t pg, svint8_t op1, svint8_t op2)</code>
<code>svbool_t</code>	<code>svmatch[_s16](svbool_t pg, svint16_t op1, svint16_t op2)</code>
<code>svbool_t</code>	<code>svmatch[_u8](svbool_t pg, svuint8_t op1, svuint8_t op2)</code>
<code>svbool_t</code>	<code>svmatch[_u16](svbool_t pg, svuint16_t op1, svuint16_t op2)</code>

### 8.19.2. NMATCH: Detect no matching elements

These functions compare each active element of the first input with all elements of the same quadword in the second input and return a predicate bit that indicates whether there were no matches.

#### 8.19.2.1. NMATCH (vector, vector)

Instances	
<code>svbool_t</code>	<code>svnmatch[_s8](svbool_t pg, svint8_t op1, svint8_t op2)</code>
<code>svbool_t</code>	<code>svnmatch[_s16](svbool_t pg, svint16_t op1, svint16_t op2)</code>

**Instances**

```
svbool_t svnmacth[_u8](svbool_t pg, svuint8_t op1, svuint8_t op2)
svbool_t svnmacth[_u16](svbool_t pg, svuint16_t op1, svuint16_t op2)
```

## 8.20. Contiguous conflict detection

### 8.20.1. WHILERW: While free of read-after-write conflicts

These functions consider a hypothetical loop:

```
for (int i = 0; i < n; ++i) {
    op1[i] = ...;
    ... = op2[i];
}
```

and return a predicate in which lane *j* is active if *j*+1 iterations of the loop could be performed in parallel without changing the dependencies between *op1* and *op2*. Lane 0 of the result is always active.

#### 8.20.1.1. WHILERW (pointer, pointer)

**Instances**

```
svbool_t svwhilerw[_s8](const int8_t *op1, const int8_t *op2)
svbool_t svwhilerw[_s16](const int16_t *op1, const int16_t *op2)
svbool_t svwhilerw[_s32](const int32_t *op1, const int32_t *op2)
svbool_t svwhilerw[_s64](const int64_t *op1, const int64_t *op2)
svbool_t svwhilerw[_u8](const uint8_t *op1, const uint8_t *op2)
svbool_t svwhilerw[_u16](const uint16_t *op1, const uint16_t *op2)
svbool_t svwhilerw[_u32](const uint32_t *op1, const uint32_t *op2)
svbool_t svwhilerw[_u64](const uint64_t *op1, const uint64_t *op2)
svbool_t svwhilerw[_f16](const float16_t *op1, const float16_t *op2)
svbool_t svwhilerw[_f32](const float32_t *op1, const float32_t *op2)
svbool_t svwhilerw[_f64](const float64_t *op1, const float64_t *op2)
svbool_t svwhilerw[_bf16](const bfloat16_t *op1, const bfloat16_t *op2)
```

### 8.20.2. WHILEWR: While free of write-after-read conflicts

These functions consider a hypothetical loop:

```
for (int i = 0; i < n; ++i) {
    ... = op1[i];
    op2[i] = ...;
}
```

and return a predicate in which lane *j* is active if *j*+1 iterations of the loop could be performed in parallel without changing the dependencies between *op1* and *op2*. Lane 0 of the result is always active.

#### 8.20.2.1. WHILEWR (pointer, pointer)

**Instances**

```
svbool_t svwhilewr[_s8](const int8_t *op1, const int8_t *op2)
svbool_t svwhilewr[_s16](const int16_t *op1, const int16_t *op2)
svbool_t svwhilewr[_s32](const int32_t *op1, const int32_t *op2)
svbool_t svwhilewr[_s64](const int64_t *op1, const int64_t *op2)
svbool_t svwhilewr[_u8](const uint8_t *op1, const uint8_t *op2)
svbool_t svwhilewr[_u16](const uint16_t *op1, const uint16_t *op2)
```

**Instances**

```
svbool_t svwhilewr[_u32](const uint32_t *op1, const uint32_t *op2)
svbool_t svwhilewr[_u64](const uint64_t *op1, const uint64_t *op2)
svbool_t svwhilewr[_f16](const float16_t *op1, const float16_t *op2)
svbool_t svwhilewr[_f32](const float32_t *op1, const float32_t *op2)
svbool_t svwhilewr[_f64](const float64_t *op1, const float64_t *op2)
svbool_t svwhilewr[_bf16](const bfloat16_t *op1, const bfloat16_t *op2)
```

## 8.21. Polynomial arithmetic

### 8.21.1. EORBT: Interleaving exclusive OR (bottom, top)

These functions take the even-indexed elements of the second input and the odd-indexed elements of the third input and store their exclusive OR in the even-indexed elements of the result. They fill the odd-indexed elements of the result from the corresponding elements of the first input.

#### 8.21.1.1. EORBT (vector, vector)

**Instances**

```
svint8_t sveorbt[_s8](svint8_t odd, svint8_t op1, svint8_t op2)
svint16_t sveorbt[_s16](svint16_t odd, svint16_t op1, svint16_t op2)
svint32_t sveorbt[_s32](svint32_t odd, svint32_t op1, svint32_t op2)
svint64_t sveorbt[_s64](svint64_t odd, svint64_t op1, svint64_t op2)
svuint8_t sveorbt[_u8](svuint8_t odd, svuint8_t op1, svuint8_t op2)
svuint16_t sveorbt[_u16](svuint16_t odd, svuint16_t op1, svuint16_t op2)
svuint32_t sveorbt[_u32](svuint32_t odd, svuint32_t op1, svuint32_t op2)
svuint64_t sveorbt[_u64](svuint64_t odd, svuint64_t op1, svuint64_t op2)
```

#### 8.21.1.2. EORBT (vector, scalar)

**Instances**

```
svint8_t sveorbt[_n_s8](svint8_t odd, svint8_t op1, int8_t op2)
svint16_t sveorbt[_n_s16](svint16_t odd, svint16_t op1, int16_t op2)
svint32_t sveorbt[_n_s32](svint32_t odd, svint32_t op1, int32_t op2)
svint64_t sveorbt[_n_s64](svint64_t odd, svint64_t op1, int64_t op2)
svuint8_t sveorbt[_n_u8](svuint8_t odd, svuint8_t op1, uint8_t op2)
svuint16_t sveorbt[_n_u16](svuint16_t odd, svuint16_t op1, uint16_t op2)
svuint32_t sveorbt[_n_u32](svuint32_t odd, svuint32_t op1, uint32_t op2)
svuint64_t sveorbt[_n_u64](svuint64_t odd, svuint64_t op1, uint64_t op2)
```

### 8.21.2. EORTB: Interleaving exclusive OR (top, bottom)

These functions take the odd-indexed elements of the second input and the even-indexed elements of the third input and store their exclusive OR in the odd-indexed elements of the result. They fill the even-indexed elements of the result from the corresponding elements of the first input.

#### 8.21.2.1. EORTB (vector, vector)

**Instances**

```
svint8_t sveortb[_s8](svint8_t even, svint8_t op1, svint8_t op2)
svint16_t sveortb[_s16](svint16_t even, svint16_t op1, svint16_t op2)
svint32_t sveortb[_s32](svint32_t even, svint32_t op1, svint32_t op2)
svint64_t sveortb[_s64](svint64_t even, svint64_t op1, svint64_t op2)
svuint8_t sveortb[_u8](svuint8_t even, svuint8_t op1, svuint8_t op2)
```

**Instances**

```
svuint16_t sveortb[_u16](svuint16_t even, svuint16_t op1, svuint16_t op2)
svuint32_t sveortb[_u32](svuint32_t even, svuint32_t op1, svuint32_t op2)
svuint64_t sveortb[_u64](svuint64_t even, svuint64_t op1, svuint64_t op2)
```

**8.21.2.2. EORTB (vector, scalar)****Instances**

```
svint8_t sveortb[_n_s8](svint8_t even, svint8_t op1, int8_t op2)
svint16_t sveortb[_n_s16](svint16_t even, svint16_t op1, int16_t op2)
svint32_t sveortb[_n_s32](svint32_t even, svint32_t op1, int32_t op2)
svint64_t sveortb[_n_s64](svint64_t even, svint64_t op1, int64_t op2)
svuint8_t sveortb[_n_u8](svuint8_t even, svuint8_t op1, uint8_t op2)
svuint16_t sveortb[_n_u16](svuint16_t even, svuint16_t op1, uint16_t op2)
svuint32_t sveortb[_n_u32](svuint32_t even, svuint32_t op1, uint32_t op2)
svuint64_t sveortb[_n_u64](svuint64_t even, svuint64_t op1, uint64_t op2)
```

**8.21.3. PMUL: Polynomial multiply**

These functions treat the two integer inputs as polynomials over GF(2) and return the low half of their product.

**8.21.3.1. PMUL (vector, vector)****Instances**

```
svuint8_t svpmul[_u8](svuint8_t op1, svuint8_t op2)
```

**8.21.3.2. PMUL (vector, scalar)****Instances**

```
svuint8_t svpmul[_n_u8](svuint8_t op1, uint8_t op2)
```

**8.21.4. PMULLB: Polynomial multiply long (bottom)**

These functions treat the two integer inputs as polynomials over GF(2). They extract the even-indexed elements of the inputs and multiply them, giving a result with twice the width.

The `_pair` forms of the functions split the double-width result into high and low halves, putting the low halves in even-indexed elements and the high halves in odd-indexed elements.

**8.21.4.1. PMULLB (vector, vector)****Instances**

```
svuint16_t svpmullb[_u16](svuint8_t op1, svuint8_t op2)
svuint64_t svpmullb[_u64](svuint32_t op1, svuint32_t op2)
svuint8_t svpmullb_pair[_u8](svuint8_t op1, svuint8_t op2)
svuint32_t svpmullb_pair[_u32](svuint32_t op1, svuint32_t op2)
```

**8.21.4.2. PMULLB (vector, scalar)****Instances**

```
svuint16_t svpmullb[_n_u16](svuint8_t op1, uint8_t op2)
```

**Instances**

```
svuint64_t svpmullb[_n_u64](svuint32_t op1, uint32_t op2)
svuint8_t svpmullb_pair[_n_u8](svuint8_t op1, uint8_t op2)
svuint32_t svpmullb_pair[_n_u32](svuint32_t op1, uint32_t op2)
```

## 8.21.5. PMULLT: Polynomial multiply long (top)

These functions treat the two integer inputs as polynomials over GF(2). They extract the odd-indexed elements of the inputs and multiply them, giving a result with twice the width.

The `_pair` forms of the functions split the double-width result into high and low halves, putting the low halves in even-indexed elements and the high halves in odd-indexed elements.

### 8.21.5.1. PMULLT (vector, vector)

**Instances**

```
svuint16_t svpmullt[_u16](svuint8_t op1, svuint8_t op2)
svuint64_t svpmullt[_u64](svuint32_t op1, svuint32_t op2)
svuint8_t svpmullt_pair[_u8](svuint8_t op1, svuint8_t op2)
svuint32_t svpmullt_pair[_u32](svuint32_t op1, svuint32_t op2)
```

### 8.21.5.2. PMULLT (vector, scalar)

**Instances**

```
svuint16_t svpmullt[_n_u16](svuint8_t op1, uint8_t op2)
svuint64_t svpmullt[_n_u64](svuint32_t op1, uint32_t op2)
svuint8_t svpmullt_pair[_n_u8](svuint8_t op1, uint8_t op2)
svuint32_t svpmullt_pair[_n_u32](svuint32_t op1, uint32_t op2)
```

## 8.22. Extended table lookup/permute

### 8.22.1. TBL2: Table lookup/permute of two vectors using vector of indices

These functions concatenate the two vectors in the first argument and return a vector in which element *i* contains the element of the concatenated vector specified by the index value in element *i* of the second input vector. Out-of-range indices select the value zero.

#### 8.22.1.1. TBL2 (vector pair, vector)

**Instances**

```
svint8_t svtbl2[_s8](svint8x2_t data, svuint8_t indices)
svint16_t svtbl2[_s16](svint16x2_t data, svuint16_t indices)
svint32_t svtbl2[_s32](svint32x2_t data, svuint32_t indices)
svint64_t svtbl2[_s64](svint64x2_t data, svuint64_t indices)
svuint8_t svtbl2[_u8](svuint8x2_t data, svuint8_t indices)
svuint16_t svtbl2[_u16](svuint16x2_t data, svuint16_t indices)
svuint32_t svtbl2[_u32](svuint32x2_t data, svuint32_t indices)
svuint64_t svtbl2[_u64](svuint64x2_t data, svuint64_t indices)
svfloat16_t svtbl2[_f16](svfloat16x2_t data, svuint16_t indices)
svfloat32_t svtbl2[_f32](svfloat32x2_t data, svuint32_t indices)
svfloat64_t svtbl2[_f64](svfloat64x2_t data, svuint64_t indices)
svbfloat16_t svtbl2[_bf16](svbfloat16x2_t data, svuint16_t indices)
```

## 8.22.2. TBX: Table lookup/permute using vector of indices (merging)

These functions return a vector in which element *i* contains the element of the second input vector specified by the index value in element *i* of the third input vector, or element *i* of the first input vector if the index is out of range.

### 8.22.2.1. TBX (vector, vector, vector)

Instances
<pre> svint8_t  <b>svtbx</b>[_s8](svint8_t fallback, svint8_t data, svuint8_t indices) svint16_t <b>svtbx</b>[_s16](svint16_t fallback, svint16_t data,                         svuint16_t indices) svint32_t <b>svtbx</b>[_s32](svint32_t fallback, svint32_t data,                         svuint32_t indices) svint64_t <b>svtbx</b>[_s64](svint64_t fallback, svint64_t data,                         svuint64_t indices) svuint8_t <b>svtbx</b>[_u8](svuint8_t fallback, svuint8_t data, svuint8_t indices) svuint16_t <b>svtbx</b>[_u16](svuint16_t fallback, svuint16_t data,                         svuint16_t indices) svuint32_t <b>svtbx</b>[_u32](svuint32_t fallback, svuint32_t data,                         svuint32_t indices) svuint64_t <b>svtbx</b>[_u64](svuint64_t fallback, svuint64_t data,                         svuint64_t indices) svfloat16_t <b>svtbx</b>[_f16](svfloat16_t fallback, svfloat16_t data,                         svuint16_t indices) svfloat32_t <b>svtbx</b>[_f32](svfloat32_t fallback, svfloat32_t data,                         svuint32_t indices) svfloat64_t <b>svtbx</b>[_f64](svfloat64_t fallback, svfloat64_t data,                         svuint64_t indices) svbfloat16_t <b>svtbx</b>[_bf16](svbfloat16_t fallback, svbfloat16_t data,                         svuint16_t indices) </pre>

## 8.23. Non-temporal gather/scatter

### 8.23.1. LDNT1: Unextended load, non-temporal

These functions behave like the loads in [Section 6.2.1, “LD1: Unextended load”](#), but additionally provide a hint to the system that the loaded memory is unlikely to be accessed again soon.

#### 8.23.1.1. LDNT1 (vector base)

Instances
<pre> svint32_t <b>svldnt1_gather</b>[_u32base]_s32(svbool_t pg, svuint32_t bases) svint64_t <b>svldnt1_gather</b>[_u64base]_s64(svbool_t pg, svuint64_t bases) svuint32_t <b>svldnt1_gather</b>[_u32base]_u32(svbool_t pg, svuint32_t bases) svuint64_t <b>svldnt1_gather</b>[_u64base]_u64(svbool_t pg, svuint64_t bases) svfloat32_t <b>svldnt1_gather</b>[_u32base]_f32(svbool_t pg, svuint32_t bases) svfloat64_t <b>svldnt1_gather</b>[_u64base]_f64(svbool_t pg, svuint64_t bases) </pre>

#### 8.23.1.2. LDNT1 (scalar base, vector offset in bytes)

Instances
<pre> svint64_t <b>svldnt1_gather</b>[_s64]_offset[_s64](svbool_t pg, const int64_t *base,   svint64_t offsets) </pre>

Instances	
<code>svuint64_t svldnt1_gather_[s64]offset[_u64]</code>	<code>(svbool_t pg, const uint64_t *base, svint64_t offsets)</code>
<code>svfloat64_t svldnt1_gather_[s64]offset[_f64]</code>	<code>(svbool_t pg, const float64_t *base, svint64_t offsets)</code>
<code>svint32_t svldnt1_gather_[u32]offset[_s32]</code>	<code>(svbool_t pg, const int32_t *base, svuint32_t offsets)</code>
<code>svint64_t svldnt1_gather_[u64]offset[_s64]</code>	<code>(svbool_t pg, const int64_t *base, svuint64_t offsets)</code>
<code>svuint32_t svldnt1_gather_[u32]offset[_u32]</code>	<code>(svbool_t pg, const uint32_t *base, svuint32_t offsets)</code>
<code>svuint64_t svldnt1_gather_[u64]offset[_u64]</code>	<code>(svbool_t pg, const uint64_t *base, svuint64_t offsets)</code>
<code>svfloat32_t svldnt1_gather_[u32]offset[_f32]</code>	<code>(svbool_t pg, const float32_t *base, svuint32_t offsets)</code>
<code>svfloat64_t svldnt1_gather_[u64]offset[_f64]</code>	<code>(svbool_t pg, const float64_t *base, svuint64_t offsets)</code>

### 8.23.1.3. LDNT1 (vector base, scalar offset in bytes)

Instances	
<code>svint32_t svldnt1_gather[_u32base]_offset_s32</code>	<code>(svbool_t pg, svuint32_t bases, int64_t offset)</code>
<code>svint64_t svldnt1_gather[_u64base]_offset_s64</code>	<code>(svbool_t pg, svuint64_t bases, int64_t offset)</code>
<code>svuint32_t svldnt1_gather[_u32base]_offset_u32</code>	<code>(svbool_t pg, svuint32_t bases, int64_t offset)</code>
<code>svuint64_t svldnt1_gather[_u64base]_offset_u64</code>	<code>(svbool_t pg, svuint64_t bases, int64_t offset)</code>
<code>svfloat32_t svldnt1_gather[_u32base]_offset_f32</code>	<code>(svbool_t pg, svuint32_t bases, int64_t offset)</code>
<code>svfloat64_t svldnt1_gather[_u64base]_offset_f64</code>	<code>(svbool_t pg, svuint64_t bases, int64_t offset)</code>

### 8.23.1.4. LDNT1 (scalar base, vector index)

Instances	
<code>svint64_t svldnt1_gather_[s64]index[_s64]</code>	<code>(svbool_t pg, const int64_t *base, svint64_t indices)</code>
<code>svuint64_t svldnt1_gather_[s64]index[_u64]</code>	<code>(svbool_t pg, const uint64_t *base, svint64_t indices)</code>
<code>svfloat64_t svldnt1_gather_[s64]index[_f64]</code>	<code>(svbool_t pg, const float64_t *base, svint64_t indices)</code>
<code>svint64_t svldnt1_gather_[u64]index[_s64]</code>	<code>(svbool_t pg, const int64_t *base, svuint64_t indices)</code>



Instances	
<code>svuint64_t</code>	<code>svldnt1_gather[_u64]index[_u64](svbool_t pg, const uint64_t *base, svuint64_t indices)</code>
<code>svfloat64_t</code>	<code>svldnt1_gather[_u64]index[_f64](svbool_t pg, const float64_t *base, svuint64_t indices)</code>

### 8.23.1.5. LDNT1 (vector base, scalar index)

Instances	
<code>svint32_t</code>	<code>svldnt1_gather[_u32base]_index_s32(svbool_t pg, svuint32_t bases, int64_t index)</code>
<code>svint64_t</code>	<code>svldnt1_gather[_u64base]_index_s64(svbool_t pg, svuint64_t bases, int64_t index)</code>
<code>svuint32_t</code>	<code>svldnt1_gather[_u32base]_index_u32(svbool_t pg, svuint32_t bases, int64_t index)</code>
<code>svuint64_t</code>	<code>svldnt1_gather[_u64base]_index_u64(svbool_t pg, svuint64_t bases, int64_t index)</code>
<code>svfloat32_t</code>	<code>svldnt1_gather[_u32base]_index_f32(svbool_t pg, svuint32_t bases, int64_t index)</code>
<code>svfloat64_t</code>	<code>svldnt1_gather[_u64base]_index_f64(svbool_t pg, svuint64_t bases, int64_t index)</code>

## 8.23.2. LDNT1SB: Load 8-bit data and sign-extend, non-temporal

These functions behave like the loads in [Section 6.2.2, “LD1SB: Load 8-bit data and sign-extend”](#), but additionally provide a hint to the system that the loaded memory is unlikely to be accessed again soon.

### 8.23.2.1. LDNT1SB (vector base)

Instances	
<code>svint32_t</code>	<code>svldnt1sb_gather[_u32base]_s32(svbool_t pg, svuint32_t bases)</code>
<code>svint64_t</code>	<code>svldnt1sb_gather[_u64base]_s64(svbool_t pg, svuint64_t bases)</code>
<code>svuint32_t</code>	<code>svldnt1sb_gather[_u32base]_u32(svbool_t pg, svuint32_t bases)</code>
<code>svuint64_t</code>	<code>svldnt1sb_gather[_u64base]_u64(svbool_t pg, svuint64_t bases)</code>

### 8.23.2.2. LDNT1SB (scalar base, vector offset in bytes)

Instances	
<code>svint64_t</code>	<code>svldnt1sb_gather[_s64]offset_s64(svbool_t pg, const int8_t *base, svint64_t offsets)</code>
<code>svuint64_t</code>	<code>svldnt1sb_gather[_s64]offset_u64(svbool_t pg, const int8_t *base, svint64_t offsets)</code>
<code>svint32_t</code>	<code>svldnt1sb_gather[_u32]offset_s32(svbool_t pg, const int8_t *base, svuint32_t offsets)</code>
<code>svint64_t</code>	<code>svldnt1sb_gather[_u64]offset_s64(svbool_t pg, const int8_t *base, svuint64_t offsets)</code>
<code>svuint32_t</code>	<code>svldnt1sb_gather[_u32]offset_u32(svbool_t pg, const int8_t *base, svuint32_t offsets)</code>
<code>svuint64_t</code>	<code>svldnt1sb_gather[_u64]offset_u64(svbool_t pg, const int8_t *base, svuint64_t offsets)</code>

### 8.23.2.3. LDNT1SB (vector base, scalar offset in bytes)

Instances	
<code>svint32_t</code>	<code>svldnt1sb_gather[_u32base]_offset_s32(svbool_t pg, svuint32_t bases, int64_t offset)</code>
<code>svint64_t</code>	<code>svldnt1sb_gather[_u64base]_offset_s64(svbool_t pg, svuint64_t bases, int64_t offset)</code>
<code>svuint32_t</code>	<code>svldnt1sb_gather[_u32base]_offset_u32(svbool_t pg, svuint32_t bases, int64_t offset)</code>
<code>svuint64_t</code>	<code>svldnt1sb_gather[_u64base]_offset_u64(svbool_t pg, svuint64_t bases, int64_t offset)</code>

### 8.23.3. LDNT1UB: Load 8-bit data and zero-extend, non-temporal

These functions behave like the loads in [Section 6.2.3, “LD1UB: Load 8-bit data and zero-extend”](#), but additionally provide a hint to the system that the loaded memory is unlikely to be accessed again soon.

#### 8.23.3.1. LDNT1UB (vector base)

Instances	
<code>svint32_t</code>	<code>svldnt1ub_gather[_u32base]_s32(svbool_t pg, svuint32_t bases)</code>
<code>svint64_t</code>	<code>svldnt1ub_gather[_u64base]_s64(svbool_t pg, svuint64_t bases)</code>
<code>svuint32_t</code>	<code>svldnt1ub_gather[_u32base]_u32(svbool_t pg, svuint32_t bases)</code>
<code>svuint64_t</code>	<code>svldnt1ub_gather[_u64base]_u64(svbool_t pg, svuint64_t bases)</code>

#### 8.23.3.2. LDNT1UB (scalar base, vector offset in bytes)

Instances	
<code>svint64_t</code>	<code>svldnt1ub_gather[_s64]_offset_s64(svbool_t pg, const uint8_t *base, svint64_t offsets)</code>
<code>svuint64_t</code>	<code>svldnt1ub_gather[_s64]_offset_u64(svbool_t pg, const uint8_t *base, svint64_t offsets)</code>
<code>svint32_t</code>	<code>svldnt1ub_gather[_u32]_offset_s32(svbool_t pg, const uint8_t *base, svuint32_t offsets)</code>
<code>svint64_t</code>	<code>svldnt1ub_gather[_u64]_offset_s64(svbool_t pg, const uint8_t *base, svuint64_t offsets)</code>
<code>svuint32_t</code>	<code>svldnt1ub_gather[_u32]_offset_u32(svbool_t pg, const uint8_t *base, svuint32_t offsets)</code>
<code>svuint64_t</code>	<code>svldnt1ub_gather[_u64]_offset_u64(svbool_t pg, const uint8_t *base, svuint64_t offsets)</code>

#### 8.23.3.3. LDNT1UB (vector base, scalar offset in bytes)

Instances	
<code>svint32_t</code>	<code>svldnt1ub_gather[_u32base]_offset_s32(svbool_t pg, svuint32_t bases, int64_t offset)</code>

**Instances**

```
svint64_t svldntlub_gather[_u64base]_offset_s64(svbool_t pg,
                                                svuint64_t bases,
                                                int64_t offset)
svuint32_t svldntlub_gather[_u32base]_offset_u32(svbool_t pg,
                                                  svuint32_t bases,
                                                  int64_t offset)
svuint64_t svldntlub_gather[_u64base]_offset_u64(svbool_t pg,
                                                  svuint64_t bases,
                                                  int64_t offset)
```

**8.23.4. LDNT1SH: Load 16-bit data and sign-extend, non-temporal**

These functions behave like the loads in [Section 6.2.4, “LD1SH: Load 16-bit data and sign-extend”](#), but additionally provide a hint to the system that the loaded memory is unlikely to be accessed again soon.

**8.23.4.1. LDNT1SH (vector base)****Instances**

```
svint32_t svldnt1sh_gather[_u32base]_s32(svbool_t pg, svuint32_t bases)
svint64_t svldnt1sh_gather[_u64base]_s64(svbool_t pg, svuint64_t bases)
svuint32_t svldnt1sh_gather[_u32base]_u32(svbool_t pg, svuint32_t bases)
svuint64_t svldnt1sh_gather[_u64base]_u64(svbool_t pg, svuint64_t bases)
```

**8.23.4.2. LDNT1SH (scalar base, vector offset in bytes)****Instances**

```
svint64_t svldnt1sh_gather[_s64]_offset_s64(svbool_t pg, const int16_t *base,
                                              svint64_t offsets)
svuint64_t svldnt1sh_gather[_s64]_offset_u64(svbool_t pg,
                                              const int16_t *base,
                                              svint64_t offsets)
svint32_t svldnt1sh_gather[_u32]_offset_s32(svbool_t pg, const int16_t *base,
                                              svuint32_t offsets)
svint64_t svldnt1sh_gather[_u64]_offset_s64(svbool_t pg, const int16_t *base,
                                              svuint64_t offsets)
svuint32_t svldnt1sh_gather[_u32]_offset_u32(svbool_t pg,
                                              const int16_t *base,
                                              svuint32_t offsets)
svuint64_t svldnt1sh_gather[_u64]_offset_u64(svbool_t pg,
                                              const int16_t *base,
                                              svuint64_t offsets)
```

**8.23.4.3. LDNT1SH (vector base, scalar offset in bytes)****Instances**

```
svint32_t svldnt1sh_gather[_u32base]_offset_s32(svbool_t pg,
                                                  svuint32_t bases,
                                                  int64_t offset)
svint64_t svldnt1sh_gather[_u64base]_offset_s64(svbool_t pg,
                                                  svuint64_t bases,
                                                  int64_t offset)
svuint32_t svldnt1sh_gather[_u32base]_offset_u32(svbool_t pg,
                                                  svuint32_t bases,
                                                  int64_t offset)
```

Instances
<pre>svuint64_t svldnt1sh_gather[_u64base]_offset_u64(svbool_t pg,   svuint64_t bases,   int64_t offset)</pre>

#### 8.23.4.4. LDNT1SH (scalar base, vector index)

Instances
<pre>svint64_t svldnt1sh_gather_[s64]index_s64(svbool_t pg, const int16_t *base,   svint64_t indices)</pre>
<pre>svuint64_t svldnt1sh_gather_[s64]index_u64(svbool_t pg, const int16_t *base,   svint64_t indices)</pre>
<pre>svint64_t svldnt1sh_gather_[u64]index_s64(svbool_t pg, const int16_t *base,   svuint64_t indices)</pre>
<pre>svuint64_t svldnt1sh_gather_[u64]index_u64(svbool_t pg, const int16_t *base,   svuint64_t indices)</pre>

#### 8.23.4.5. LDNT1SH (vector base, scalar index)

Instances
<pre>svint32_t svldnt1sh_gather[_u32base]_index_s32(svbool_t pg,   svuint32_t bases,   int64_t index)</pre>
<pre>svint64_t svldnt1sh_gather[_u64base]_index_s64(svbool_t pg,   svuint64_t bases,   int64_t index)</pre>
<pre>svuint32_t svldnt1sh_gather[_u32base]_index_u32(svbool_t pg,   svuint32_t bases,   int64_t index)</pre>
<pre>svuint64_t svldnt1sh_gather[_u64base]_index_u64(svbool_t pg,   svuint64_t bases,   int64_t index)</pre>

### 8.23.5. LDNT1UH: Load 16-bit data and zero-extend, non-temporal

These functions behave like the loads in [Section 6.2.5, “LD1UH: Load 16-bit data and zero-extend”](#), but additionally provide a hint to the system that the loaded memory is unlikely to be accessed again soon.

#### 8.23.5.1. LDNT1UH (vector base)

Instances
<pre>svint32_t svldnt1uh_gather[_u32base]_s32(svbool_t pg, svuint32_t bases)</pre>
<pre>svint64_t svldnt1uh_gather[_u64base]_s64(svbool_t pg, svuint64_t bases)</pre>
<pre>svuint32_t svldnt1uh_gather[_u32base]_u32(svbool_t pg, svuint32_t bases)</pre>
<pre>svuint64_t svldnt1uh_gather[_u64base]_u64(svbool_t pg, svuint64_t bases)</pre>

#### 8.23.5.2. LDNT1UH (scalar base, vector offset in bytes)

Instances
<pre>svint64_t svldnt1uh_gather_[s64]offset_s64(svbool_t pg,   const uint16_t *base,   svint64_t offsets)</pre>
<pre>svuint64_t svldnt1uh_gather_[s64]offset_u64(svbool_t pg,</pre>

**Instances**

```

const uint16_t *base,
svint64_t offsets)
svint32_t svldnt1uh_gather[_u32]offset_s32(svbool_t pg,
const uint16_t *base,
svuint32_t offsets)
svint64_t svldnt1uh_gather[_u64]offset_s64(svbool_t pg,
const uint16_t *base,
svuint64_t offsets)
svuint32_t svldnt1uh_gather[_u32]offset_u32(svbool_t pg,
const uint16_t *base,
svuint32_t offsets)
svuint64_t svldnt1uh_gather[_u64]offset_u64(svbool_t pg,
const uint16_t *base,
svuint64_t offsets)

```

**8.23.5.3. LDNT1UH (vector base, scalar offset in bytes)****Instances**

```

svint32_t svldnt1uh_gather[_u32base]_offset_s32(svbool_t pg,
svuint32_t bases,
int64_t offset)
svint64_t svldnt1uh_gather[_u64base]_offset_s64(svbool_t pg,
svuint64_t bases,
int64_t offset)
svuint32_t svldnt1uh_gather[_u32base]_offset_u32(svbool_t pg,
svuint32_t bases,
int64_t offset)
svuint64_t svldnt1uh_gather[_u64base]_offset_u64(svbool_t pg,
svuint64_t bases,
int64_t offset)

```

**8.23.5.4. LDNT1UH (scalar base, vector index)****Instances**

```

svint64_t svldnt1uh_gather[_s64]index_s64(svbool_t pg, const uint16_t *base,
svint64_t indices)
svuint64_t svldnt1uh_gather[_s64]index_u64(svbool_t pg,
const uint16_t *base,
svint64_t indices)
svint64_t svldnt1uh_gather[_u64]index_s64(svbool_t pg, const uint16_t *base,
svuint64_t indices)
svuint64_t svldnt1uh_gather[_u64]index_u64(svbool_t pg,
const uint16_t *base,
svuint64_t indices)

```

**8.23.5.5. LDNT1UH (vector base, scalar index)****Instances**

```

svint32_t svldnt1uh_gather[_u32base]_index_s32(svbool_t pg,
svuint32_t bases,
int64_t index)
svint64_t svldnt1uh_gather[_u64base]_index_s64(svbool_t pg,
svuint64_t bases,
int64_t index)
svuint32_t svldnt1uh_gather[_u32base]_index_u32(svbool_t pg,

```

Instances	
<code>svuint64_t</code>	<code>svldnt1uh_gather[_u64base]_index_u64(svbool_t pg, svuint64_t bases, int64_t index)</code>

### 8.23.6. LDNT1SW: Load 32-bit data and sign-extend, non-temporal

These functions behave like the loads in [Section 6.2.6, “LD1SW: Load 32-bit data and sign-extend”](#), but additionally provide a hint to the system that the loaded memory is unlikely to be accessed again soon.

#### 8.23.6.1. LDNT1SW (vector base)

Instances	
<code>svint64_t</code>	<code>svldnt1sw_gather[_u64base]_s64(svbool_t pg, svuint64_t bases)</code>
<code>svuint64_t</code>	<code>svldnt1sw_gather[_u64base]_u64(svbool_t pg, svuint64_t bases)</code>

#### 8.23.6.2. LDNT1SW (scalar base, vector offset in bytes)

Instances	
<code>svint64_t</code>	<code>svldnt1sw_gather[_s64]_offset_s64(svbool_t pg, const int32_t *base, svint64_t offsets)</code>
<code>svuint64_t</code>	<code>svldnt1sw_gather[_s64]_offset_u64(svbool_t pg, const int32_t *base, svint64_t offsets)</code>
<code>svint64_t</code>	<code>svldnt1sw_gather[_u64]_offset_s64(svbool_t pg, const int32_t *base, svuint64_t offsets)</code>
<code>svuint64_t</code>	<code>svldnt1sw_gather[_u64]_offset_u64(svbool_t pg, const int32_t *base, svuint64_t offsets)</code>

#### 8.23.6.3. LDNT1SW (vector base, scalar offset in bytes)

Instances	
<code>svint64_t</code>	<code>svldnt1sw_gather[_u64base]_offset_s64(svbool_t pg, svuint64_t bases, int64_t offset)</code>
<code>svuint64_t</code>	<code>svldnt1sw_gather[_u64base]_offset_u64(svbool_t pg, svuint64_t bases, int64_t offset)</code>

#### 8.23.6.4. LDNT1SW (scalar base, vector index)

Instances	
<code>svint64_t</code>	<code>svldnt1sw_gather[_s64]_index_s64(svbool_t pg, const int32_t *base, svint64_t indices)</code>
<code>svuint64_t</code>	<code>svldnt1sw_gather[_s64]_index_u64(svbool_t pg, const int32_t *base, svint64_t indices)</code>
<code>svint64_t</code>	<code>svldnt1sw_gather[_u64]_index_s64(svbool_t pg, const int32_t *base, svuint64_t indices)</code>
<code>svuint64_t</code>	<code>svldnt1sw_gather[_u64]_index_u64(svbool_t pg, const int32_t *base, svuint64_t indices)</code>

### 8.23.6.5. LDNT1SW (vector base, scalar index)

Instances
<pre>svint64_t svldnt1sw_gather[_u64base]_index_s64(svbool_t pg,   svuint64_t bases,   int64_t index) svuint64_t svldnt1sw_gather[_u64base]_index_u64(svbool_t pg,   svuint64_t bases,   int64_t index)</pre>

## 8.23.7. LDNT1UW: Load 32-bit data and zero-extend, non-temporal

These functions behave like the loads in [Section 6.2.7, “LD1UW: Load 32-bit data and zero-extend”](#), but additionally provide a hint to the system that the loaded memory is unlikely to be accessed again soon.

### 8.23.7.1. LDNT1UW (vector base)

Instances
<pre>svint64_t svldnt1uw_gather[_u64base]_s64(svbool_t pg, svuint64_t bases) svuint64_t svldnt1uw_gather[_u64base]_u64(svbool_t pg, svuint64_t bases)</pre>

### 8.23.7.2. LDNT1UW (scalar base, vector offset in bytes)

Instances
<pre>svint64_t svldnt1uw_gather[_s64]_offset_s64(svbool_t pg,   const uint32_t *base,   svint64_t offsets) svuint64_t svldnt1uw_gather[_s64]_offset_u64(svbool_t pg,   const uint32_t *base,   svint64_t offsets) svint64_t svldnt1uw_gather[_u64]_offset_s64(svbool_t pg,   const uint32_t *base,   svuint64_t offsets) svuint64_t svldnt1uw_gather[_u64]_offset_u64(svbool_t pg,   const uint32_t *base,   svuint64_t offsets)</pre>

### 8.23.7.3. LDNT1UW (vector base, scalar offset in bytes)

Instances
<pre>svint64_t svldnt1uw_gather[_u64base]_offset_s64(svbool_t pg,   svuint64_t bases,   int64_t offset) svuint64_t svldnt1uw_gather[_u64base]_offset_u64(svbool_t pg,   svuint64_t bases,   int64_t offset)</pre>

### 8.23.7.4. LDNT1UW (scalar base, vector index)

Instances
<pre>svint64_t svldnt1uw_gather[_s64]_index_s64(svbool_t pg, const uint32_t *base,   svint64_t indices) svuint64_t svldnt1uw_gather[_s64]_index_u64(svbool_t pg,</pre>

Instances	
<code>svint64_t</code>	<code>svldntlw_gather[_u64]index_s64(svbool_t pg, const uint32_t *base, svint64_t indices)</code>
<code>svuint64_t</code>	<code>svldntlw_gather[_u64]index_u64(svbool_t pg, const uint32_t *base, svuint64_t indices)</code>

### 8.23.7.5. LDNT1UW (vector base, scalar index)

Instances	
<code>svint64_t</code>	<code>svldntlw_gather[_u64base]_index_s64(svbool_t pg, svuint64_t bases, int64_t index)</code>
<code>svuint64_t</code>	<code>svldntlw_gather[_u64base]_index_u64(svbool_t pg, svuint64_t bases, int64_t index)</code>

### 8.23.8. STNT1: Store one vector, with no truncation, non-temporal

These functions behave like the stores in [Section 6.3.1, “ST1: Store one vector, with no truncation”](#), but additionally provide a hint to the system that the stored memory is unlikely to be accessed again soon.

#### 8.23.8.1. STNT1 (vector base)

Instances	
<code>void</code>	<code>svstntl_scatter[_u32base_s32](svbool_t pg, svuint32_t bases, svint32_t data)</code>
<code>void</code>	<code>svstntl_scatter[_u64base_s64](svbool_t pg, svuint64_t bases, svint64_t data)</code>
<code>void</code>	<code>svstntl_scatter[_u32base_u32](svbool_t pg, svuint32_t bases, svuint32_t data)</code>
<code>void</code>	<code>svstntl_scatter[_u64base_u64](svbool_t pg, svuint64_t bases, svuint64_t data)</code>
<code>void</code>	<code>svstntl_scatter[_u32base_f32](svbool_t pg, svuint32_t bases, svfloat32_t data)</code>
<code>void</code>	<code>svstntl_scatter[_u64base_f64](svbool_t pg, svuint64_t bases, svfloat64_t data)</code>

#### 8.23.8.2. STNT1 (scalar base, vector offset in bytes)

Instances	
<code>void</code>	<code>svstntl_scatter[_s64]offset[_s64](svbool_t pg, int64_t *base, svint64_t offsets, svint64_t data)</code>
<code>void</code>	<code>svstntl_scatter[_s64]offset[_u64](svbool_t pg, uint64_t *base, svint64_t offsets, svuint64_t data)</code>
<code>void</code>	<code>svstntl_scatter[_s64]offset[_f64](svbool_t pg, float64_t *base, svint64_t offsets, svfloat64_t data)</code>
<code>void</code>	<code>svstntl_scatter[_u32]offset[_s32](svbool_t pg, int32_t *base, svuint32_t offsets, svint32_t data)</code>
<code>void</code>	<code>svstntl_scatter[_u64]offset[_s64](svbool_t pg, int64_t *base, svuint64_t offsets, svint64_t data)</code>
<code>void</code>	<code>svstntl_scatter[_u32]offset[_u32](svbool_t pg, uint32_t *base, svuint32_t offsets, svuint32_t data)</code>



**Instances**

```
void svstnt1_scatter[_u64]offset[_u64](svbool_t pg, uint64_t *base,
                                       svuint64_t offsets, svuint64_t data)
void svstnt1_scatter[_u32]offset[_f32](svbool_t pg, float32_t *base,
                                       svuint32_t offsets,
                                       svfloat32_t data)
void svstnt1_scatter[_u64]offset[_f64](svbool_t pg, float64_t *base,
                                       svuint64_t offsets,
                                       svfloat64_t data)
```

**8.23.8.3. STNT1 (vector base, scalar offset in bytes)****Instances**

```
void svstnt1_scatter[_u32base]_offset[_s32](svbool_t pg, svuint32_t bases,
                                             int64_t offset, svint32_t data)
void svstnt1_scatter[_u64base]_offset[_s64](svbool_t pg, svuint64_t bases,
                                             int64_t offset, svint64_t data)
void svstnt1_scatter[_u32base]_offset[_u32](svbool_t pg, svuint32_t bases,
                                             int64_t offset,
                                             svuint32_t data)
void svstnt1_scatter[_u64base]_offset[_u64](svbool_t pg, svuint64_t bases,
                                             int64_t offset,
                                             svuint64_t data)
void svstnt1_scatter[_u32base]_offset[_f32](svbool_t pg, svuint32_t bases,
                                             int64_t offset,
                                             svfloat32_t data)
void svstnt1_scatter[_u64base]_offset[_f64](svbool_t pg, svuint64_t bases,
                                             int64_t offset,
                                             svfloat64_t data)
```

**8.23.8.4. STNT1 (scalar base, vector index)****Instances**

```
void svstnt1_scatter[_s64]index[_s64](svbool_t pg, int64_t *base,
                                       svint64_t indices, svint64_t data)
void svstnt1_scatter[_s64]index[_u64](svbool_t pg, uint64_t *base,
                                       svint64_t indices, svuint64_t data)
void svstnt1_scatter[_s64]index[_f64](svbool_t pg, float64_t *base,
                                       svint64_t indices, svfloat64_t data)
void svstnt1_scatter[_u64]index[_s64](svbool_t pg, int64_t *base,
                                       svuint64_t indices, svint64_t data)
void svstnt1_scatter[_u64]index[_u64](svbool_t pg, uint64_t *base,
                                       svuint64_t indices, svuint64_t data)
void svstnt1_scatter[_u64]index[_f64](svbool_t pg, float64_t *base,
                                       svuint64_t indices, svfloat64_t data)
```

**8.23.8.5. STNT1 (vector base, scalar index)****Instances**

```
void svstnt1_scatter[_u32base]_index[_s32](svbool_t pg, svuint32_t bases,
                                             int64_t index, svint32_t data)
void svstnt1_scatter[_u64base]_index[_s64](svbool_t pg, svuint64_t bases,
                                             int64_t index, svint64_t data)
void svstnt1_scatter[_u32base]_index[_u32](svbool_t pg, svuint32_t bases,
                                             int64_t index, svuint32_t data)
void svstnt1_scatter[_u64base]_index[_u64](svbool_t pg, svuint64_t bases,
```

Instances	
	<code>int64_t index, svuint64_t data)</code>
<code>void svstnt1b_scatter[_u32base]_index[_f32](svbool_t pg, svuint32_t bases,</code>	<code>int64_t index, svfloat32_t data)</code>
<code>void svstnt1b_scatter[_u64base]_index[_f64](svbool_t pg, svuint64_t bases,</code>	<code>int64_t index, svfloat64_t data)</code>

### 8.23.9. STNT1B: Store one vector, truncating to 8 bits, non-temporal

These functions behave like the stores in [Section 6.3.2, “ST1B: Store one vector, truncating to 8 bits”](#), but additionally provide a hint to the system that the stored memory is unlikely to be accessed again soon.

#### 8.23.9.1. STNT1B (vector base)

Instances	
<code>void svstnt1b_scatter[_u32base_s32](svbool_t pg, svuint32_t bases,</code>	<code>svint32_t data)</code>
<code>void svstnt1b_scatter[_u64base_s64](svbool_t pg, svuint64_t bases,</code>	<code>svint64_t data)</code>
<code>void svstnt1b_scatter[_u32base_u32](svbool_t pg, svuint32_t bases,</code>	<code>svuint32_t data)</code>
<code>void svstnt1b_scatter[_u64base_u64](svbool_t pg, svuint64_t bases,</code>	<code>svuint64_t data)</code>

#### 8.23.9.2. STNT1B (scalar base, vector offset in bytes)

Instances	
<code>void svstnt1b_scatter[_s64]_offset[_s64](svbool_t pg, int8_t *base,</code>	<code>svint64_t offsets, svint64_t data)</code>
<code>void svstnt1b_scatter[_s64]_offset[_u64](svbool_t pg, uint8_t *base,</code>	<code>svint64_t offsets, svuint64_t data)</code>
<code>void svstnt1b_scatter[_u32]_offset[_s32](svbool_t pg, int8_t *base,</code>	<code>svuint32_t offsets, svint32_t data)</code>
<code>void svstnt1b_scatter[_u64]_offset[_s64](svbool_t pg, int8_t *base,</code>	<code>svuint64_t offsets, svint64_t data)</code>
<code>void svstnt1b_scatter[_u32]_offset[_u32](svbool_t pg, uint8_t *base,</code>	<code>svuint32_t offsets,</code> <code>svuint32_t data)</code>
<code>void svstnt1b_scatter[_u64]_offset[_u64](svbool_t pg, uint8_t *base,</code>	<code>svuint64_t offsets,</code> <code>svuint64_t data)</code>

#### 8.23.9.3. STNT1B (vector base, scalar offset in bytes)

Instances	
<code>void svstnt1b_scatter[_u32base]_offset[_s32](svbool_t pg, svuint32_t bases,</code>	<code>int64_t offset,</code> <code>svint32_t data)</code>
<code>void svstnt1b_scatter[_u64base]_offset[_s64](svbool_t pg, svuint64_t bases,</code>	<code>int64_t offset,</code> <code>svint64_t data)</code>
<code>void svstnt1b_scatter[_u32base]_offset[_u32](svbool_t pg, svuint32_t bases,</code>	<code>int64_t offset,</code> <code>svuint32_t data)</code>
<code>void svstnt1b_scatter[_u64base]_offset[_u64](svbool_t pg, svuint64_t bases,</code>	

**Instances**

```
int64_t offset,
svuint64_t data)
```

## 8.23.10. STNT1H: Store one vector, truncating to 16 bits, non-temporal

These functions behave like the stores in [Section 6.3.3, “ST1H: Store one vector, truncating to 16 bits”](#), but additionally provide a hint to the system that the stored memory is unlikely to be accessed again soon.

### 8.23.10.1. STNT1H (vector base)

**Instances**

```
void svstnt1h_scatter[_u32base_s32](svbool_t pg, svuint32_t bases,
svint32_t data)
void svstnt1h_scatter[_u64base_s64](svbool_t pg, svuint64_t bases,
svint64_t data)
void svstnt1h_scatter[_u32base_u32](svbool_t pg, svuint32_t bases,
svuint32_t data)
void svstnt1h_scatter[_u64base_u64](svbool_t pg, svuint64_t bases,
svuint64_t data)
```

### 8.23.10.2. STNT1H (scalar base, vector offset in bytes)

**Instances**

```
void svstnt1h_scatter[_s64]offset[_s64](svbool_t pg, int16_t *base,
svint64_t offsets, svint64_t data)
void svstnt1h_scatter[_s64]offset[_u64](svbool_t pg, uint16_t *base,
svint64_t offsets, svuint64_t data)
void svstnt1h_scatter[_u32]offset[_s32](svbool_t pg, int16_t *base,
svuint32_t offsets, svint32_t data)
void svstnt1h_scatter[_u64]offset[_s64](svbool_t pg, int16_t *base,
svuint64_t offsets, svint64_t data)
void svstnt1h_scatter[_u32]offset[_u32](svbool_t pg, uint16_t *base,
svuint32_t offsets,
svuint32_t data)
void svstnt1h_scatter[_u64]offset[_u64](svbool_t pg, uint16_t *base,
svuint64_t offsets,
svuint64_t data)
```

### 8.23.10.3. STNT1H (vector base, scalar offset in bytes)

**Instances**

```
void svstnt1h_scatter[_u32base]offset[_s32](svbool_t pg, svuint32_t bases,
int64_t offset,
svint32_t data)
void svstnt1h_scatter[_u64base]offset[_s64](svbool_t pg, svuint64_t bases,
int64_t offset,
svint64_t data)
void svstnt1h_scatter[_u32base]offset[_u32](svbool_t pg, svuint32_t bases,
int64_t offset,
svuint32_t data)
void svstnt1h_scatter[_u64base]offset[_u64](svbool_t pg, svuint64_t bases,
int64_t offset,
```

Instances
<code>svuint64_t data)</code>

#### 8.23.10.4. STNT1H (scalar base, vector index)

Instances
<code>void svstnt1h_scatter[_s64]index[_s64](svbool_t pg, int16_t *base, svint64_t indices, svint64_t data)</code>
<code>void svstnt1h_scatter[_s64]index[_u64](svbool_t pg, uint16_t *base, svint64_t indices, svuint64_t data)</code>
<code>void svstnt1h_scatter[_u64]index[_s64](svbool_t pg, int16_t *base, svuint64_t indices, svint64_t data)</code>
<code>void svstnt1h_scatter[_u64]index[_u64](svbool_t pg, uint16_t *base, svuint64_t indices, svuint64_t data)</code>

#### 8.23.10.5. STNT1H (vector base, scalar index)

Instances
<code>void svstnt1h_scatter[_u32base]_index[_s32](svbool_t pg, svuint32_t bases, int64_t index, svint32_t data)</code>
<code>void svstnt1h_scatter[_u64base]_index[_s64](svbool_t pg, svuint64_t bases, int64_t index, svint64_t data)</code>
<code>void svstnt1h_scatter[_u32base]_index[_u32](svbool_t pg, svuint32_t bases, int64_t index, svuint32_t data)</code>
<code>void svstnt1h_scatter[_u64base]_index[_u64](svbool_t pg, svuint64_t bases, int64_t index, svuint64_t data)</code>

### 8.23.11. STNT1W: Store one vector, truncating to 32 bits, non-temporal

These functions behave like the stores in [Section 6.3.4, “ST1W: Store one vector, truncating to 32 bits”](#), but additionally provide a hint to the system that the stored memory is unlikely to be accessed again soon.

#### 8.23.11.1. STNT1W (vector base)

Instances
<code>void svstnt1w_scatter[_u64base_s64](svbool_t pg, svuint64_t bases, svint64_t data)</code>
<code>void svstnt1w_scatter[_u64base_u64](svbool_t pg, svuint64_t bases, svuint64_t data)</code>

#### 8.23.11.2. STNT1W (scalar base, vector offset in bytes)

Instances
<code>void svstnt1w_scatter[_s64]offset[_s64](svbool_t pg, int32_t *base, svint64_t offsets, svint64_t data)</code>
<code>void svstnt1w_scatter[_s64]offset[_u64](svbool_t pg, uint32_t *base, svint64_t offsets, svuint64_t data)</code>
<code>void svstnt1w_scatter[_u64]offset[_s64](svbool_t pg, int32_t *base, svuint64_t offsets, svint64_t data)</code>
<code>void svstnt1w_scatter[_u64]offset[_u64](svbool_t pg, uint32_t *base, svuint64_t offsets, svuint64_t data)</code>

**8.23.11.3. STNT1W (vector base, scalar offset in bytes)****Instances**

```
void svstntlw_scatter[_u64base]_offset[_s64](svbool_t pg, svuint64_t bases,
                                              int64_t offset,
                                              svint64_t data)
void svstntlw_scatter[_u64base]_offset[_u64](svbool_t pg, svuint64_t bases,
                                              int64_t offset,
                                              svuint64_t data)
```

**8.23.11.4. STNT1W (scalar base, vector index)****Instances**

```
void svstntlw_scatter[_s64]index[_s64](svbool_t pg, int32_t *base,
                                         svint64_t indices, svint64_t data)
void svstntlw_scatter[_s64]index[_u64](svbool_t pg, uint32_t *base,
                                         svint64_t indices, svuint64_t data)
void svstntlw_scatter[_u64]index[_s64](svbool_t pg, int32_t *base,
                                         svuint64_t indices, svint64_t data)
void svstntlw_scatter[_u64]index[_u64](svbool_t pg, uint32_t *base,
                                         svuint64_t indices, svuint64_t data)
```

**8.23.11.5. STNT1W (vector base, scalar index)****Instances**

```
void svstntlw_scatter[_u64base]_index[_s64](svbool_t pg, svuint64_t bases,
                                              int64_t index, svint64_t data)
void svstntlw_scatter[_u64base]_index[_u64](svbool_t pg, svuint64_t bases,
                                              int64_t index, svuint64_t data)
```

## 9. List of optional SVE2 functions

### 9.1. Introduction

This section contains a list of optional SVE2 functions, grouped into categories and then subdivided based on the first part of the name (up to the first underscore). Each group has its own feature macro.

### 9.2. Bit permutation

These functions are only available when the feature macro `__ARM_FEATURE_SVE2_BITPERM` is defined.

#### 9.2.1. BDEP: Bit deposit under mask control

These functions scatter the low bits of the first integer input to the bit positions indicated by non-zero bits in the second integer input, preserving their order, and set the remaining bits to zero.

##### 9.2.1.1. BDEP (vector, vector)

Instances
<code>svuint8_t svbdep[_u8](svuint8_t op1, svuint8_t op2)</code>
<code>svuint16_t svbdep[_u16](svuint16_t op1, svuint16_t op2)</code>
<code>svuint32_t svbdep[_u32](svuint32_t op1, svuint32_t op2)</code>
<code>svuint64_t svbdep[_u64](svuint64_t op1, svuint64_t op2)</code>

##### 9.2.1.2. BDEP (vector, scalar)

Instances
<code>svuint8_t svbdep[_n_u8](svuint8_t op1, uint8_t op2)</code>
<code>svuint16_t svbdep[_n_u16](svuint16_t op1, uint16_t op2)</code>
<code>svuint32_t svbdep[_n_u32](svuint32_t op1, uint32_t op2)</code>
<code>svuint64_t svbdep[_n_u64](svuint64_t op1, uint64_t op2)</code>

#### 9.2.2. BEXT: Bit extract under mask control

These functions gather bits of the first integer input from the bit positions indicated by non-zero bits in the second integer input into the low bits of the result, preserving their order, and set the remaining high-order bits to zero.

##### 9.2.2.1. BEXT (vector, vector)

Instances
<code>svuint8_t svbext[_u8](svuint8_t op1, svuint8_t op2)</code>
<code>svuint16_t svbext[_u16](svuint16_t op1, svuint16_t op2)</code>
<code>svuint32_t svbext[_u32](svuint32_t op1, svuint32_t op2)</code>
<code>svuint64_t svbext[_u64](svuint64_t op1, svuint64_t op2)</code>

##### 9.2.2.2. BEXT (vector, scalar)

Instances
<code>svuint8_t svbext[_n_u8](svuint8_t op1, uint8_t op2)</code>

**Instances**

```

svuint16_t svbext[_n_u16](svuint16_t op1, uint16_t op2)
svuint32_t svbext[_n_u32](svuint32_t op1, uint32_t op2)
svuint64_t svbext[_n_u64](svuint64_t op1, uint64_t op2)

```

### 9.2.3. BGRP: Group bits to right or left as selected by bitmask

These functions gather bits of the first integer input from the bit positions indicated by non-zero bits in the second integer input into the low bits of the result and gather the remaining bits of the first input to the high bits of the result, preserving the order of the bits in each group.

#### 9.2.3.1. BGRP (vector, vector)

**Instances**

```

svuint8_t svbgrp[_u8](svuint8_t op1, svuint8_t op2)
svuint16_t svbgrp[_u16](svuint16_t op1, svuint16_t op2)
svuint32_t svbgrp[_u32](svuint32_t op1, svuint32_t op2)
svuint64_t svbgrp[_u64](svuint64_t op1, svuint64_t op2)

```

#### 9.2.3.2. BGRP (vector, scalar)

**Instances**

```

svuint8_t svbgrp[_n_u8](svuint8_t op1, uint8_t op2)
svuint16_t svbgrp[_n_u16](svuint16_t op1, uint16_t op2)
svuint32_t svbgrp[_n_u32](svuint32_t op1, uint32_t op2)
svuint64_t svbgrp[_n_u64](svuint64_t op1, uint64_t op2)

```

## 9.3. AES-128 functions

These functions are only available when the feature macro `__ARM_FEATURE_SVE2_AES` is defined.

### 9.3.1. PMULLB: Polynomial multiply long (bottom)

These functions provide additional forms of the functions described in [Section 8.21.4, “PMULLB: Polynomial multiply long \(bottom\)”](#).

#### 9.3.1.1. PMULLB (vector, vector)

**Instances**

```

svuint64_t svpmullb_pair[_u64](svuint64_t op1, svuint64_t op2)

```

#### 9.3.1.2. PMULLB (vector, scalar)

**Instances**

```

svuint64_t svpmullb_pair[_n_u64](svuint64_t op1, uint64_t op2)

```

### 9.3.2. PMULLT: Polynomial multiply long (top)

These functions provide additional forms of the functions described in [Section 8.21.4, “PMULLB: Polynomial multiply long \(bottom\)”](#).

### 9.3.2.1. PMULLT (vector, vector)

Instances
<code>svuint64_t svpmullt_pair[_u64](svuint64_t op1, svuint64_t op2)</code>

### 9.3.2.2. PMULLT (vector, scalar)

Instances
<code>svuint64_t svpmullt_pair[_n_u64](svuint64_t op1, uint64_t op2)</code>

## 9.3.3. AESD: AES single round decryption

This function reads a 16-byte state array from each 128-bit segment of the first input, together with a round key from the corresponding 128-bit segment of the second input, and applies a single round of the AddRoundKey(), InvSubBytes() and InvShiftRows() transformations in accordance with the AES standard.

### 9.3.3.1. AESD (vector, vector)

Instances
<code>svuint8_t svaesd[_u8](svuint8_t op1, svuint8_t op2)</code>

## 9.3.4. AESIMC: AES inverse mix columns

This function reads a 16-byte state array from each 128-bit segment of the input applies a single round of the InvMixColumns() transformation in accordance with the AES standard.

### 9.3.4.1. AESIMC (vector)

Instances
<code>svuint8_t svaesimc[_u8](svuint8_t op)</code>

## 9.3.5. AESE: AES single round encryption

This function reads a 16-byte state array from each 128-bit segment of the first input, together with a round key from the corresponding 128-bit segment of the second input, and applies a single round of the AddRoundKey(), SubBytes() and ShiftRows() transformations in accordance with the AES standard.

### 9.3.5.1. AESE (vector, vector)

Instances
<code>svuint8_t svaese[_u8](svuint8_t op1, svuint8_t op2)</code>

## 9.3.6. AESMC: AES mix columns

This function reads a 16-byte state array from each 128-bit segment of the input applies a single round of the MixColumns() transformation in accordance with the AES standard.

### 9.3.6.1. AESMC (vector)

Instances
<code>svuint8_t svaesmc[_u8](svuint8_t op)</code>



## 9.4. SHA-3 functions

These functions are only available when the feature macro `__ARM_FEATURE_SVE2_SHA3` is defined.

### 9.4.1. RAX1: Bitwise rotate left by 1 and exclusive OR

These functions rotate the second input left by one bit and return the exclusive OR of the rotated value and the first input.

#### 9.4.1.1. RAX1 (vector, vector)

Instances
<code>svint64_t svrax1[_s64](svint64_t op1, svint64_t op2)</code>
<code>svuint64_t svrax1[_u64](svuint64_t op1, svuint64_t op2)</code>

## 9.5. SM4 functions

These functions are only available when the feature macro `__ARM_FEATURE_SVE2_SM4` is defined.

### 9.5.1. SM4E: SM4 encryption and decryption

This function reads 16 bytes of data from each 128-bit segment of the first input, together with four iterations of 32-bit round keys from the corresponding 128-bit segment of the second input, and encrypts each block by four rounds in accordance with the SM4 standard.

#### 9.5.1.1. SM4E (vector, vector)

Instances
<code>svuint32_t svsm4e[_u32](svuint32_t op1, svuint32_t op2)</code>

### 9.5.2. SM4EKEY: SM4 key updates

This function reads four rounds of 32-bit input key values from each 128-bit segment of the first input, along with four rounds of 32-bit constants from the corresponding 128-bit segment of the second input, and derives four rounds of output key values in accordance with the SM4 standard.

#### 9.5.2.1. SM4EKEY (vector, vector)

Instances
<code>svuint32_t svsm4ekey[_u32](svuint32_t op1, svuint32_t op2)</code>

# 10. Mapping of instructions to functions

## 10.1. List of instructions

This section contains a list of all SVE and SVE2 instructions. For each one it gives a reference to the associated ACLE function or explains why no such function exists.

The list is generally in the same order as the architecture specification, although some entries with the same mnemonic are grouped differently to avoid repeating the explanation.

ABS

[Section 6.7.19, “ABS: Integer absolute”](#)

ADCLB

[Section 8.10.1, “ADCLB: Integer addition with carry \(bottom\)”](#)

ADCLT

[Section 8.10.1, “ADCLB: Integer addition with carry \(bottom\)”](#)

ADD (immediate)

ADD (vectors, predicated)

ADD (vectors, unpredicated)

[Section 6.7.1, “ADD: Modular integer addition”](#)

ADDHNB

[Section 8.5.1, “ADDHNB: Integer addition, narrowing to high half \(bottom\)”](#)

ADDHNT

[Section 8.5.2, “ADDHNT: Integer addition, narrowing to high half \(top\)”](#)

ADDP

[Section 8.7.1, “ADDP: Integer pairwise addition”](#)

ADDPL

ADDVL

No direct support; see [Section 10.3, “ADDPL, ADDVL, INC and DEC”](#)

ADR

[Section 6.5.1, “ADRB: Compute vector address for 8-bit data”](#)

[Section 6.5.2, “ADRH: Compute vector address for 16-bit data”](#)

[Section 6.5.3, “ADRW: Compute vector address for 32-bit data”](#)

[Section 6.5.4, “ARD: Compute vector address for 64-bit data”](#)

AESD

[Section 9.3.3, “AESD: AES single round decryption”](#)

AESE

[Section 9.3.5, “AESE: AES single round encryption”](#)

## AESIMC

[Section 9.3.4, “AESIMC: AES inverse mix columns”](#)

## AESMC

[Section 9.3.6, “AESMC: AES mix columns”](#)

AND (immediate)

AND (vectors, predicated)

AND (vectors, unpredicated)

[Section 6.8.1, “AND: Bitwise AND”](#)

AND (predicates)

ANDS

[Section 6.24.2, “AND: Predicate AND”](#)

## ANDV

[Section 6.10.4, “ANDV: Integer AND reduction”](#)

ASR (immediate, predicated)

ASR (immediate, unpredicated)

ASR (vectors)

ASR (wide elements, predicated)

ASR (wide elements, unpredicated)

[Section 6.9.3, “ASR: Arithmetic shift right, rounding towards -Inf”](#)

## ASRD

[Section 6.9.4, “ASRD: Arithmetic shift right, rounding towards zero”](#)

## ASRR

[Section 6.9.3, “ASR: Arithmetic shift right, rounding towards -Inf”](#)

## BCAX

[Section 8.9.1, “BCAX: Bitwise clear and exclusive OR”](#)

## BDEP

[Section 9.2.1, “BDEP: Bit deposit under mask control”](#)

## BEXT

[Section 9.2.2, “BEXT: Bit extract under mask control”](#)

## BGRP

[Section 9.2.3, “BGRP: Group bits to right or left as selected by bitmask”](#)

## BFCVT

[Section 7.2.5, “CVT: Convert single-precision floats to BFloat16”](#)

**BFCVTNT**

[Section 7.2.6, “CVTNT: Convert single-precision floats to BFloat16 \(top\)”](#)

**BFDOT**

[Section 7.2.1, “BFDOT: BFloat16 addition of dot product”](#)

**BFMLALB**

[Section 7.2.3, “BFMLALB: BFloat16 addition of product long \(bottom\)”](#)

**BFMLALT**

[Section 7.2.4, “BFMLALT: BFloat16 addition of product long \(top\)”](#)

**BFMMLA**

[Section 7.2.2, “BFMMLA: Accumulating multiplication of BFloat16 matrices”](#)

**BIC (immediate)****BIC (vectors, predicated)****BIC (vectors, unpredicated)**

[Section 6.8.2, “BIC: Bitwise AND NOT”](#)

**BIC (predicates)****BICS**

[Section 6.24.3, “BIC: Predicate AND NOT”](#)

**BRKA****BRKAS**

[Section 6.24.10, “BRKA: Break after first true condition”](#)

**BRKB****BRKBS**

[Section 6.24.11, “BRKB: Break before first true condition”](#)

**BRKN****BRKNS**

[Section 6.24.12, “BRKN: Propagate break to next partition”](#)

**BRKPA****BRKPAS**

[Section 6.24.13, “BRKPA: Propagate and break after first true condition”](#)

**BRKPB****BRKPBS**

[Section 6.24.14, “BRKPB: Propagate and break before first true condition”](#)

**BSL**

[Section 8.9.2, “BSL: Bitwise select”](#)

**BSL1N**

[Section 8.9.3, “BSL1N: Bitwise select with first input inverted”](#)

**BSL2N**

[Section 8.9.4, “BSL2N: Bitwise select with second input inverted”](#)

**CADD**

[Section 8.12.1, “CADD: Integer complex addition with rotation”](#)

**CDOT (indexed)****CDOT (vectors)**

[Section 8.14.1, “CDOT: Integer complex dot-product”](#)

**CLASTA (scalar)****CLASTA (SIMD&FP scalar)****CLASTA (vectors)**

[Section 6.20.3, “CLASTA: Extract element after last active with fallback”](#)

**CLASTB (scalar)****CLASTB (SIMD&FP scalar)****CLASTB (vectors)**

[Section 6.20.4, “CLASTB: Extract last active element with fallback”](#)

**CLS**

[Section 6.13.1, “CLS: Count leading sign bits”](#)

**CLZ**

[Section 6.13.2, “CLZ: Count leading zero bits”](#)

**CMLA (indexed)****CMLA (vectors)**

[Section 8.12.3, “CMLA: Integer complex addition of product with rotation”](#)

**CMPEQ (immediate)****CMPEQ (vectors)****CMPEQ (wide elements)**

[Section 6.11.1, “CMPEQ: Integer compare equal”](#)

**CMUGE (immediate)****CMUGE (vectors)****CMUGE (wide elements)**

[Section 6.11.5, “CMUGE: Integer compare greater than or equal to”](#)

**CMPGT (immediate)****CMPGT (vectors)****CMPGT (wide elements)**

[Section 6.11.6, “CMPGT: Integer compare greater than”](#)

CMPHI (immediate)  
CMPHI (vectors)  
CMPHI (wide elements)

[Section 6.11.6, “CMPGT: Integer compare greater than”](#)

CMPHS (immediate)  
CMPHS (vectors)  
CMPHS (wide elements)

[Section 6.11.5, “CMPGE: Integer compare greater than or equal to”](#)

CMPL (immediate)  
CMPL (vectors)  
CMPL (wide elements)

[Section 6.11.4, “CMPL: Integer compare less than or equal to”](#)

CMPLO (immediate)  
CMPLO (vectors)  
CMPLO (wide elements)

[Section 6.11.3, “CMPLT: Integer compare less than”](#)

CMPLS (immediate)  
CMPLS (vectors)  
CMPLS (wide elements)

[Section 6.11.4, “CMPL: Integer compare less than or equal to”](#)

CMPLT (immediate)  
CMPLT (vectors)  
CMPLT (wide elements)

[Section 6.11.3, “CMPLT: Integer compare less than”](#)

CMPNE (immediate)  
CMPNE (vectors)  
CMPNE (wide elements)

[Section 6.11.2, “CMPNE: Integer compare not equal”](#)

CNOT

[Section 6.8.6, “CNOT: Logical inverse”](#)

CNT

[Section 6.13.3, “CNT: Count nonzero bits”](#)

CNTB

[Section 6.27.2, “CNTB: Count the number of 8-bit elements in a pattern”](#)

CNTD

[Section 6.27.5, “CNTD: Count the number of 64-bit elements in a pattern”](#)

**CNTH**

[Section 6.27.3, “CNTH: Count the number of 16-bit elements in a pattern”](#)

**CNTP**

[Section 6.27.1, “CNTP: Count active elements”](#)

**CNTW**

[Section 6.27.4, “CNTW: Count the number of 32-bit elements in a pattern”](#)

**COMPACT**

[Section 6.20.5, “COMPACT: Compact vector and fill with zero”](#)

**CPY (immediate)****CPY (scalar)****CPY (SIMD&FP scalar)**

[Section 6.6.1, “DUP: Duplicate scalar value”](#)

**CTERMEQ****CTERMNE**

No direct support; see [Section 10.2, “CTERMEQ and CTERMNE”](#)

**DECB****DECD (scalar)****DECD (vector)****DECH (scalar)****DECH (vector)****DECP (scalar)****DECP (vector)****DECW (scalar)****DECW (vector)**

No direct support; see [Section 10.3, “ADDPL, ADDVL, INC and DEC”](#)

**DUP (indexed)**

[Section 6.20.9, “DUP: Duplicate one element of a vector”](#)

[Section 6.20.10, “DUPQ: Duplicate one quadword of a vector”](#)

**DUP (immediate)****DUP (scalar)****DUPM**

[Section 6.6.1, “DUP: Duplicate scalar value”](#)

[Section 6.6.2, “DUPQ: Duplicate scalars to every quadword of a vector”](#)

[Section 6.23.3, “DUP: Duplicate boolean value”](#)

[Section 6.23.4, “DUPQ: Duplicate boolean values to fill a predicate”](#)

**EON****EOR (immediate)****EOR (vectors, predicated)****EOR (vectors, unpredicated)**

[Section 6.8.4, “EOR: Bitwise exclusive OR”](#)

EOR (predicates)  
EORS

[Section 6.24.8, “EOR: Predicate exclusive OR”](#)

EOR3

[Section 8.9.5, “EOR3: Bitwise exclusive OR three vectors”](#)

EORBT

[Section 8.21.1, “EORBT: Interleaving exclusive OR \(bottom, top\)”](#)

EORTB

[Section 8.21.2, “EORTB: Interleaving exclusive OR \(top, bottom\)”](#)

EORV

[Section 6.10.6, “EORV: Integer exclusive OR reduction”](#)

EXT

[Section 6.20.7, “EXT: Extract vector from pair of vectors”](#)

FABD

[Section 6.16.5, “ABD: Floating-point absolute difference”](#)

FABS

[Section 6.16.27, “ABS: Floating-point absolute”](#)

FACGE

[Section 6.18.10, “ACGE: Floating-point absolute compare greater than or equal to”](#)

FACGT

[Section 6.18.11, “ACGT: Floating-point absolute compare greater than”](#)

FACLE

[Section 6.18.9, “ACLE: Floating-point absolute compare less than or equal to”](#)

FACLT

[Section 6.18.8, “ACLT: Floating-point absolute compare less than”](#)

FADD (immediate)  
FADD (vectors, predicated)  
FADD (vectors, unpredicated)

[Section 6.16.1, “ADD: Floating-point addition”](#)

FADDA

[Section 6.17.1, “ADDA: Left-to-right floating-point addition reduction”](#)

FADDP

[Section 8.7.2, “ADDP: Floating-point pairwise addition”](#)



**FADDV**

[Section 6.17.2, “ADDV: Tree-based floating-point addition reduction”](#)

**FCADD**

[Section 6.16.2, “CADD: Floating-point complex addition with rotation”](#)

**FCMEQ (vectors)****FCMEQ (zero)**

[Section 6.18.1, “CMPEQ: Floating-point compare equal”](#)

**FCMGE (vectors)****FCMGE (zero)**

[Section 6.18.5, “CMPGE: Floating-point compare greater than or equal to”](#)

**FCMGT (vectors)****FCMGT (zero)**

[Section 6.18.6, “CMPGT: Floating-point compare greater than”](#)

**FCMLA (indexed)****FCMLA (vectors)**

[Section 6.16.10, “CMLA: Fused floating-point complex addition of product with rotation”](#)

**FCMLE (vectors)****FCMLE (zero)**

[Section 6.18.4, “CMPLE: Floating-point compare less than or equal to”](#)

**FCMLT (vectors)****FCMLT (zero)**

[Section 6.18.3, “CMPLT: Floating-point compare less than”](#)

**FCMNE (vectors)****FCMNE (zero)**

[Section 6.18.2, “CMPNE: Floating-point compare not equal”](#)

**FCMUO**

[Section 6.18.7, “CMPUO: Floating-point compare unordered”](#)

**FCPY**

[Section 6.6.1, “DUP: Duplicate scalar value”](#)

**FCVT**

[Section 6.19.3, “CVT: Convert floating-point value to wider type”](#)

[Section 6.19.4, “CVT: Convert floating-point value to narrower type”](#)

**FCVTLT**

[Section 8.15.1, “CVTLT: Convert floating-point value to wider type \(top\)”](#)

**FCVTNT**

[Section 8.15.2, “CVTNT: Convert floating-point value to narrower type \(top\)”](#)

**FCVTX**

[Section 8.15.3, “CVTX: Convert floating-point value to narrower type, rounding to odd”](#)

**FCVTXNT**

[Section 8.15.4, “CVTXNT: Convert floating-point value to narrower type, rounding to odd \(top\)”](#)

**FCVTZS****FCVTZU**

[Section 6.19.1, “CVT: Convert floating-point value to integer”](#)

**FDIV**

[Section 6.16.17, “DIV: Floating-point division”](#)

**FDIVR**

[Section 6.16.18, “DIVR: Floating-point division, reversed”](#)

**FDUP**

[Section 6.6.1, “DUP: Duplicate scalar value”](#)

**FEXPA**

[Section 6.16.30, “EXPA: Floating-point exponent accelerator”](#)

**FLOGB**

[Section 8.17.1, “LOGB: Floating-point base 2 logarithm as integer”](#)

**FMAD**

[Section 6.16.8, “MAD: Fused floating-point addition of product \(multiplicand first\)”](#)

**FMAX (immediate)****FMAX (vectors)**

[Section 6.16.19, “MAX: Floating-point maximum”](#)

**FMAXNM (immediate)****FMAXNM (vectors)**

[Section 6.16.20, “MAXNM: Floating-point maximum number”](#)

**FMAXNMP**

[Section 8.7.5, “MAXNMP: Floating-point pairwise maximum number”](#)

**FMAXNMV**

[Section 6.17.4, “MAXNMV: Floating-point maximum number reduction”](#)

**FMAXP**

[Section 8.7.4, “MAXP: Floating-point pairwise maximum”](#)

**FMAXV**

[Section 6.17.3, “MAXV: Floating-point maximum reduction”](#)

**FMIN (immediate)**

**FMIN (vectors)**

[Section 6.16.21, “MIN: Floating-point minimum”](#)

**FMINNM (immediate)**

**FMINNM (vectors)**

[Section 6.16.22, “MINNM: Floating-point minimum number”](#)

**FMINNMP**

[Section 8.7.8, “MINNMP: Floating-point pairwise minimum number”](#)

**FMINNMV**

[Section 6.17.6, “MINNMV: Floating-point minimum number reduction”](#)

**FMINP**

[Section 8.7.7, “MINP: Floating-point pairwise minimum”](#)

**FMINV**

[Section 6.17.5, “MINV: Floating-point minimum reduction”](#)

**FMLA (indexed)**

**FMLA (vectors)**

[Section 6.16.9, “MLA: Fused floating-point addition of product \(addend first\)”](#)

**FMLALB (indexed)**

**FMLALB (vectors)**

[Section 8.16.1, “MLALB: Floating-point addition of product long \(bottom\)”](#)

**FMLALT (indexed)**

**FMLALT (vectors)**

[Section 8.16.2, “MLALT: Floating-point addition of product long \(top\)”](#)

**FMLS (indexed)**

**FMLS (vectors)**

[Section 6.16.12, “MLS: Fused floating-point subtraction of product \(minuend first\)”](#)

**FMLS LB (indexed)**

**FMLS LB (vectors)**

[Section 8.16.3, “MLS LB: Floating-point subtraction of product long \(bottom\)”](#)

**FMLS LT (indexed)**

**FMLS LT (vectors)**

[Section 8.16.4, “MLS LT: Floating-point subtraction of product long \(top\)”](#)

**FMMLA**

[Section 7.4.1, “MMLA: Accumulating multiplication of 2×2 float matrices”](#)

[Section 7.5.1, “MMLA: Accumulating multiplication of 2×2 double matrices”](#)

FMOV (immediate, predicated)

FMOV (immediate, unpredicated)

FMOV (zero, predicated)

FMOV (zero, unpredicated)

[Section 6.6.1, “DUP: Duplicate scalar value”](#)

**FMSB**

[Section 6.16.11, “MSB: Fused floating-point subtraction of product \(multiplicand first\)”](#)

FMUL (immediate)

FMUL (indexed)

FMUL (vectors, predicated)

FMUL (vectors, unpredicated)

[Section 6.16.6, “MUL: Floating-point multiplication”](#)

**FMULX**

[Section 6.16.7, “MULX: Floating-point multiplication extended”](#)

**FNEG**

[Section 6.16.28, “NEG: Floating-point negation”](#)

**FNMAD**

[Section 6.16.13, “NMAD: Fused floating-point addition of product, negated \(multiplicand first\)”](#)

**FNMLA**

[Section 6.16.14, “NMLA: Fused floating-point addition of product, negated \(addend first\)”](#)

**FNMLS**

[Section 6.16.16, “NMLS: Fused floating-point subtraction of product, negated \(minuend first\)”](#)

**FNMSB**

[Section 6.16.15, “NMSB: Fused floating-point subtraction of product, negated \(multiplicand first\)”](#)

**FRECPE**

[Section 6.16.31, “RECPE: Floating-point reciprocal estimate”](#)

**FRECPS**

[Section 6.16.32, “RECPS: Floating-point reciprocal step”](#)

**FRECPX**

[Section 6.16.33, “RECPX: Floating-point reciprocal exponent”](#)

**FRINTA**

Section 6.16.36, “RINTA: Floating-point round to nearest, ties away from zero”

**FRINTI**

Section 6.16.37, “RINTI: Floating-point round using current rounding mode (inexact)”

**FRINTM**

Section 6.16.38, “RINTM: Floating-point round towards -Inf”

**FRINTN**

Section 6.16.39, “RINTN: Floating-point round to nearest, ties to even”

**FRINTP**

Section 6.16.40, “RINTP: Floating-point round towards +Inf”

**FRINTX**

Section 6.16.41, “RINTX: Floating-point round using current rounding mode (exact)”

**FRINTZ**

Section 6.16.42, “RINTZ: Floating-point round towards zero”

**FRSQ RTE**

Section 6.16.34, “RSQ RTE: Floating-point reciprocal square root estimate”

**FRSQ RTS**

Section 6.16.35, “RSQ RTS: Floating-point reciprocal square root step”

**FSCALE**

Section 6.16.23, “SCALE: Floating-point adjust exponent”

**FSQRT**

Section 6.16.29, “SQRT: Floating-point square root”

**FSUB (immediate)****FSUB (vectors, predicated)****FSUB (vectors, unpredicated)**

Section 6.16.3, “SUB: Floating-point subtraction”

**FSUBR (immediate)****FSUBR (vectors)**

Section 6.16.4, “SUBR: Floating-point subtraction, reversed”

**FTMAD**

Section 6.16.25, “TMAD: Floating-point trigonometric multiply-add coefficient”

**FTSMUL**

[Section 6.16.24, “TSMUL: Floating-point trigonometric starting value”](#)

**FTSSEL**

[Section 6.16.26, “TSSEL: Floating-point trigonometric select coefficient”](#)

**HISTCNT**

[Section 8.18.1, “HISTCNT: Count matching elements in vector”](#)

**HISTSEG**

[Section 8.18.2, “HISTSEG: Count matching elements in vector segments”](#)

**INCB**

INCD (scalar)

INCD (vector)

INCH (scalar)

INCH (vector)

INCP (scalar)

INCP (vector)

INCW (scalar)

INCW (vector)

No direct support; see [Section 10.3, “ADDPL, ADDVL, INC and DEC”](#)

INDEX (immediate, scalar)

INDEX (immediates)

INDEX (scalar, immediate)

INDEX (scalars)

[Section 6.6.3, “INDEX: Create index series”](#)

INSR (scalar)

INSR (SIMD&FP scalar)

[Section 6.9.5, “INSR: Shift vector and insert scalar”](#)

LASTA (scalar)

LASTA (SIMD&FP scalar)

[Section 6.20.1, “LASTA: Extract element after last active”](#)

LASTB (scalar)

LASTB (SIMD&FP scalar)

[Section 6.20.2, “LASTB: Extract last active element”](#)

LD1B (scalar plus immediate)

LD1B (scalar plus scalar)

LD1B (scalar plus vector)

LD1B (vector plus immediate)

[Section 6.2.1, “LD1: Unextended load”](#)

[Section 6.2.3, “LD1UB: Load 8-bit data and zero-extend ”](#)

LD1D (scalar plus immediate)  
 LD1D (scalar plus scalar)  
 LD1D (scalar plus vector)  
 LD1D (vector plus immediate)

[Section 6.2.1, “LD1: Unextended load”](#)

LD1H (scalar plus immediate)  
 LD1H (scalar plus scalar)  
 LD1H (scalar plus vector)  
 LD1H (vector plus immediate)

[Section 6.2.1, “LD1: Unextended load”](#)

[Section 6.2.5, “LD1UH: Load 16-bit data and zero-extend ”](#)

LD1RB  
 LD1RD  
 LD1RH  
 LD1RSB  
 LD1RSH  
 LD1RSW  
 LD1RW

No direct support, but the compiler can use these instructions to optimize `svdups` in which the scalar value comes from memory; see [Section 6.6.1, “DUP: Duplicate scalar value”](#).

LD1ROB (scalar plus immediate)  
 LD1ROB (scalar plus scalar)  
 LD1ROD (scalar plus immediate)  
 LD1ROD (scalar plus scalar)  
 LD1ROH (scalar plus immediate)  
 LD1ROH (scalar plus scalar)  
 LD1ROW (scalar plus immediate)  
 LD1ROW (scalar plus scalar)

[Section 7.5.2, “LD1RO: Unextended load and replicate to octoword”](#)

LD1RQB (scalar plus immediate)  
 LD1RQB (scalar plus scalar)  
 LD1RQD (scalar plus immediate)  
 LD1RQD (scalar plus scalar)  
 LD1RQH (scalar plus immediate)  
 LD1RQH (scalar plus scalar)  
 LD1RQW (scalar plus immediate)  
 LD1RQW (scalar plus scalar)

[Section 6.2.8, “LD1RQ: Unextended load and replicate to quadword”](#)

[Section 6.6.2, “DUPQ: Duplicate scalars to every quadword of a vector”](#)

[Section 6.23.4, “DUPQ: Duplicate boolean values to fill a predicate”](#)

LD1SB (scalar plus immediate)  
 LD1SB (scalar plus scalar)  
 LD1SB (scalar plus vector)  
 LD1SB (vector plus immediate)

[Section 6.2.2, “LD1SB: Load 8-bit data and sign-extend ”](#)

LD1SH (scalar plus immediate)  
LD1SH (scalar plus scalar)  
LD1SH (scalar plus vector)  
LD1SH (vector plus immediate)

[Section 6.2.4, “LD1SH: Load 16-bit data and sign-extend”](#)

LD1SW (scalar plus immediate)  
LD1SW (scalar plus scalar)  
LD1SW (scalar plus vector)  
LD1SW (vector plus immediate)

[Section 6.2.6, “LD1SW: Load 32-bit data and sign-extend”](#)

LD1W (scalar plus immediate)  
LD1W (scalar plus scalar)  
LD1W (scalar plus vector)  
LD1W (vector plus immediate)

[Section 6.2.1, “LD1: Unextended load”](#)

[Section 6.2.7, “LD1UW: Load 32-bit data and zero-extend”](#)

LD2B (scalar plus immediate)  
LD2B (scalar plus scalar)  
LD2D (scalar plus immediate)  
LD2D (scalar plus scalar)  
LD2H (scalar plus immediate)  
LD2H (scalar plus scalar)  
LD2W (scalar plus immediate)  
LD2W (scalar plus scalar)

[Section 6.2.24, “LD2: Load two-element structures into two vectors”](#)

LD3B (scalar plus immediate)  
LD3B (scalar plus scalar)  
LD3D (scalar plus immediate)  
LD3D (scalar plus scalar)  
LD3H (scalar plus immediate)  
LD3H (scalar plus scalar)  
LD3W (scalar plus immediate)  
LD3W (scalar plus scalar)

[Section 6.2.25, “LD3: Load three-element structures into three vectors”](#)

LD4B (scalar plus immediate)  
LD4B (scalar plus scalar)  
LD4D (scalar plus immediate)  
LD4D (scalar plus scalar)  
LD4H (scalar plus immediate)  
LD4H (scalar plus scalar)  
LD4W (scalar plus immediate)  
LD4W (scalar plus scalar)

[Section 6.2.26, “LD4: Load four-element structures into four vectors”](#)



LDFF1B (scalar plus scalar)  
 LDFF1B (scalar plus vector)  
 LDFF1B (vector plus immediate)

[Section 6.2.9, “LDFF1: Unextended load, first-faulting”](#)

[Section 6.2.11, “LDFF1UB: Load 8-bit data and zero-extend, first-faulting ”](#)

LDFF1D (scalar plus scalar)  
 LDFF1D (scalar plus vector)  
 LDFF1D (vector plus immediate)

[Section 6.2.9, “LDFF1: Unextended load, first-faulting”](#)

LDFF1H (scalar plus scalar)  
 LDFF1H (scalar plus vector)  
 LDFF1H (vector plus immediate)

[Section 6.2.9, “LDFF1: Unextended load, first-faulting”](#)

[Section 6.2.13, “LDFF1UH: Load 16-bit data and zero-extend, first-faulting ”](#)

LDFF1SB (scalar plus scalar)  
 LDFF1SB (scalar plus vector)  
 LDFF1SB (vector plus immediate)

[Section 6.2.10, “LDFF1SB: Load 8-bit data and sign-extend, first-faulting ”](#)

LDFF1SH (scalar plus scalar)  
 LDFF1SH (scalar plus vector)  
 LDFF1SH (vector plus immediate)

[Section 6.2.12, “LDFF1SH: Load 16-bit data and sign-extend, first-faulting ”](#)

LDFF1SW (scalar plus scalar)  
 LDFF1SW (scalar plus vector)  
 LDFF1SW (vector plus immediate)

[Section 6.2.14, “LDFF1SW: Load 32-bit data and sign-extend, first-faulting ”](#)

LDFF1W (scalar plus scalar)  
 LDFF1W (scalar plus vector)  
 LDFF1W (vector plus immediate)

[Section 6.2.9, “LDFF1: Unextended load, first-faulting”](#)

[Section 6.2.15, “LDFF1UW: Load 32-bit data and zero-extend, first-faulting ”](#)

LDNF1B

[Section 6.2.16, “LDNF1: Unextended load, non-faulting”](#)

[Section 6.2.18, “LDNF1UB: Load 8-bit data and zero-extend, non-faulting”](#)

LDNF1D

[Section 6.2.16, “LDNF1: Unextended load, non-faulting”](#)

LDNF1H

[Section 6.2.16, “LDNF1: Unextended load, non-faulting”](#)

[Section 6.2.20, “LDNF1UH: Load 16-bit data and zero-extend, non-faulting”](#)

## LDNF1SB

[Section 6.2.17, “LDNF1SB: Load 8-bit data and sign-extend, non-faulting”](#)

## LDNF1SH

[Section 6.2.19, “LDNF1SH: Load 16-bit data and sign-extend, non-faulting”](#)

## LDNF1SW

[Section 6.2.21, “LDNF1SW: Load 32-bit data and sign-extend, non-faulting”](#)

## LDNF1W

[Section 6.2.16, “LDNF1: Unextended load, non-faulting”](#)

[Section 6.2.22, “LDNF1UW: Load 32-bit data and zero-extend, non-faulting”](#)

LDNT1B (scalar plus immediate)

LDNT1B (scalar plus scalar)

LDNT1D (scalar plus immediate)

LDNT1D (scalar plus scalar)

LDNT1H (scalar plus immediate)

LDNT1H (scalar plus scalar)

LDNT1W (scalar plus immediate)

LDNT1W (scalar plus scalar)

[Section 6.2.23, “LDNT1: Unextended load, non-temporal”](#)

LDNT1B (vector plus scalar)

[Section 8.23.1, “LDNT1: Unextended load, non-temporal”](#)

[Section 8.23.3, “LDNT1UB: Load 8-bit data and zero-extend, non-temporal ”](#)

LDNT1D (vector plus scalar)

[Section 8.23.1, “LDNT1: Unextended load, non-temporal”](#)

LDNT1H (vector plus scalar)

[Section 8.23.1, “LDNT1: Unextended load, non-temporal”](#)

[Section 8.23.5, “LDNT1UH: Load 16-bit data and zero-extend, non-temporal ”](#)

## LDNT1SB

[Section 8.23.2, “LDNT1SB: Load 8-bit data and sign-extend, non-temporal ”](#)

## LDNT1SH

[Section 8.23.4, “LDNT1SH: Load 16-bit data and sign-extend, non-temporal ”](#)

## LDNT1SW

[Section 8.23.6, “LDNT1SW: Load 32-bit data and sign-extend, non-temporal ”](#)

LDNT1W (vector plus scalar)

[Section 8.23.1, “LDNT1: Unextended load, non-temporal”](#)

[Section 8.23.7, “LDNT1UW: Load 32-bit data and zero-extend, non-temporal ”](#)

LDR (predicate)

LDR (vector)

No direct support, although the compiler can use these instructions to refill registers from the stack.

LSL (immediate, predicated)

LSL (immediate, unpredicated)

LSL (vectors)

LSL (wide elements, predicated)

LSL (wide elements, unpredicated)

LSLR

[Section 6.9.1, “LSL: Shift left”](#)

LSR (immediate, predicated)

LSR (immediate, unpredicated)

LSR (vectors)

LSR (wide elements, predicated)

LSR (wide elements, unpredicated)

LSRR

[Section 6.9.2, “LSR: Logical shift right”](#)

MAD

[Section 6.7.9, “MAD: Integer addition of product \(multiplicand first\)”](#)

MATCH

[Section 8.19.1, “MATCH: Detect any matching elements”](#)

MLA (indexed)

[Section 8.11.2, “MLA: Integer addition of product \(addend first\)”](#)

MLA (vectors)

[Section 6.7.10, “MLA: Integer addition of product \(addend first\)”](#)

MLS (indexed)

[Section 8.11.3, “MLS: Integer subtraction of product \(addend first\)”](#)

MLS (vectors)

[Section 6.7.12, “MLS: Integer subtraction of product \(minuend first\)”](#)

MOV (bitmask immediate)

MOV (immediate, predicated)

MOV (immediate, unpredicated)

[Section 6.6.1, “DUP: Duplicate scalar value”](#)

MOVPRFX (predicated)  
MOVPRFX (unpredicated)

No direct support, but the compiler can use these instructions to provide forms of predication that do not exist as a single instruction.

MOV (scalar, predicated)  
MOV (scalar, unpredicated)  
MOV (SIMD&FP scalar, predicated)  
MOV (SIMD&FP scalar, unpredicated)

[Section 6.6.1, “DUP: Duplicate scalar value”](#)

MOV (predicate, predicated, merging)  
MOV (vector, predicated)

[Section 6.20.8, “SEL: Conditionally select elements from two inputs”](#)

MOV (predicate, predicated, zeroing)  
MOVS (predicated)

[Section 6.24.1, “MOV: Copy predicate”](#)

MOV (predicate, unpredicated)  
MOV (vector, unpredicated)  
MOVS (unpredicated)

No direct support, although the compiler can use these instructions to copy vectors and predicates around.

MSB

[Section 6.7.11, “MSB: Integer subtraction of product \(multiplicand first\)”](#)

MUL (immediate)  
MUL (vectors, predicated)  
MUL (vectors, unpredicated)

[Section 6.7.7, “MUL: Integer multiplication, returning low half”](#)

MUL (indexed)

[Section 8.11.1, “MUL: Integer multiplication”](#)

NAND  
NANDS

[Section 6.24.4, “NAND: Predicate NAND”](#)

NBSL

[Section 8.9.6, “NBSL: Bitwise inverted select”](#)

NEG

[Section 6.7.18, “NEG: Integer negation”](#)

NMATCH

[Section 8.19.2, “NMATCH: Detect no matching elements”](#)

NOR

NORS

[Section 6.24.7, “NOR: Predicate NOR”](#)

NOT (vector)

[Section 6.8.5, “NOT: Bitwise inverse”](#)

NOT (predicate)

NOTS

[Section 6.24.9, “NOT: Predicate NOT”](#)

ORN (predicates)

ORNS

[Section 6.24.6, “ORN: Predicate OR NOT”](#)

ORN (immediate)

ORR (immediate)

ORR (vectors, predicated)

ORR (vectors, unpredicated)

[Section 6.8.3, “ORR: Bitwise OR”](#)

ORR (predicates)

ORRS

[Section 6.24.5, “ORR: Predicate OR”](#)

ORV

[Section 6.10.5, “ORV: Integer OR reduction”](#)

PFALSE

[Section 6.23.2, “PFALSE: Return an all-false predicate”](#)

PFIRST

[Section 6.24.15, “PFIRST: Set first active predicate element to true”](#)

PMUL

[Section 8.21.3, “PMUL: Polynomial multiply”](#)

PMULLB

[Section 8.21.4, “PMULLB: Polynomial multiply long \(bottom\)”](#)

[Section 9.3.1, “PMULLB: Polynomial multiply long \(bottom\)”](#)

PMULLT

[Section 8.21.5, “PMULLT: Polynomial multiply long \(top\)”](#)

[Section 9.3.2, “PMULLT: Polynomial multiply long \(top\)”](#)**PNEXT**[Section 6.24.16, “PNEXT: Set next active predicate element to true”](#)

PRFB (scalar plus immediate)  
PRFB (scalar plus scalar)  
PRFB (scalar plus vector)  
PRFB (vector plus immediate)

[Section 6.4.1, “PRFB: Prefetch 8-bit data”](#)

PRFD (scalar plus immediate)  
PRFD (scalar plus scalar)  
PRFD (scalar plus vector)  
PRFD (vector plus immediate)

[Section 6.4.4, “PRFD: Prefetch 64-bit data”](#)

PRFH (scalar plus immediate)  
PRFH (scalar plus scalar)  
PRFH (scalar plus vector)  
PRFH (vector plus immediate)

[Section 6.4.2, “PRFH: Prefetch 16-bit data”](#)

PRFW (scalar plus immediate)  
PRFW (scalar plus scalar)  
PRFW (scalar plus vector)  
PRFW (vector plus immediate)

[Section 6.4.3, “PRFW: Prefetch 32-bit data”](#)**PTEST**[Section 6.25.1, “PTEST: Test active elements”](#)

PTRUE  
PTRUES

[Section 6.23.1, “PTRUE: Return an all-true predicate for a given pattern”](#)**PUNPKHI**[Section 6.20.15, “UNPKHI: Unpack and extend high half of an input”](#)**PUNPKLO**[Section 6.20.16, “UNPKLO: Unpack and extend low half of an input”](#)**RADDHNB**[Section 8.5.3, “RADDHNB: Integer addition, rounding narrowing to high half \(bottom\)”](#)**RADDHNT**[Section 8.5.4, “RADDHNT: Integer addition, rounding narrowing to high half \(top\)”](#)

**RAX1**

[Section 9.4.1, “RAX1: Bitwise rotate left by 1 and exclusive OR”](#)

**RBIT**

[Section 6.15.1, “RBIT: Reverse bits within elements”](#)

**RDFFR (predicated)****RDFFR (unpredicated)****RDFFRS**

[Section 6.26.1, “RDFFR: Read the first-fault register”](#)

**RDVL**

No direct support. In an ACLE context the `svcntb`, `svcnth`, `svcntw` and `svcntd` functions should be more appropriate.

[Section 6.27.2, “CNTB: Count the number of 8-bit elements in a pattern”](#)

[Section 6.27.3, “CNTH: Count the number of 16-bit elements in a pattern”](#)

[Section 6.27.4, “CNTW: Count the number of 32-bit elements in a pattern”](#)

[Section 6.27.5, “CNTD: Count the number of 64-bit elements in a pattern”](#)

**REV (predicate)****REV (vector)**

[Section 6.20.12, “REV: Reverse the elements in a single input”](#)

**REVB**

[Section 6.15.2, “REVB: Reverse bytes within elements”](#)

**REVB**

[Section 6.15.3, “REVB: Reverse halfwords within elements”](#)

**REVB**

[Section 6.15.4, “REVB: Reverse words within elements”](#)

**RSHRNB**

[Section 8.5.11, “RSHRNB: Rounding shift right, narrowing \(bottom\)”](#)

**RSHRNT**

[Section 8.5.12, “RSHRNT: Rounding shift right, narrowing \(top\)”](#)

**RSUBHNB**

[Section 8.5.7, “RSUBHNB: Integer subtraction, rounding narrowing to high half \(bottom\)”](#)

**RSUBHNT**

[Section 8.5.8, “RSUBHNT: Integer subtraction, rounding narrowing to high half \(top\)”](#)

**SABA**

[Section 8.3.10, “ABA: Integer addition of absolute difference”](#)

**SABALB**

Section 8.4.11, “ABALB: Integer addition of absolute difference long (bottom)”

**SABALT**

Section 8.4.12, “ABALT: Integer addition of absolute difference long (top)”

**SABD**

Section 6.7.6, “ABD: Integer absolute difference”

**SABDLB**

Section 8.4.9, “ABDLB: Integer absolute difference long (bottom)”

**SABDLT**

Section 8.4.10, “ABDLT: Integer absolute difference long (top)”

**SADALP**

Section 8.8.1, “ADALP: Integer pairwise addition and accumulate”

**SADDLB**

Section 8.4.1, “ADDLB: Integer addition long (bottom)”

**SADDLBT**

Section 8.13.1, “ADDLBT: Integer addition long (bottom + top)”

**SADDLT**

Section 8.4.2, “ADDLT: Integer addition long (top)”

**SADDV**

Section 6.10.1, “ADDV: Integer addition reduction”

**SADDWB**

Section 8.4.3, “ADDWB: Integer addition wide (bottom)”

**SADDWT**

Section 8.4.4, “ADDWT: Integer addition wide (top)”

**SBCLB**

Section 8.10.3, “SBCLB: Integer subtraction with carry (bottom)”

**SBCLT**

Section 8.10.4, “SBCLT: Integer subtraction with carry (top)”

**SCVTF**

Section 6.19.2, “CVT: Convert integer value to floating-point”



**SDIV**

[Section 6.7.14, “DIV: Integer division”](#)

**SDIVR**

[Section 6.7.15, “DIVR: Integer division, reversed”](#)

**SDOT (indexed)****SDOT (vectors)**

[Section 6.7.13, “DOT: Integer addition of dot product”](#)

**SEL (predicates)****SEL (vectors)**

[Section 6.20.8, “SEL: Conditionally select elements from two inputs”](#)

**SETFFR**

[Section 6.26.2, “SETFFR: Set the first-fault register”](#)

**SHADD**

[Section 8.3.4, “HADD: Halving integer addition”](#)

**SHRNB**

[Section 8.5.9, “SHRNB: Shift right, narrowing \(bottom\)”](#)

**SHRNT**

[Section 8.5.10, “SHRNT: Shift right, narrowing \(top\)”](#)

**SHSUB**

[Section 8.3.8, “HSUB: Halving integer subtraction”](#)

**SHSUBR**

[Section 8.3.9, “HSUBR: Halving integer subtraction, reversed”](#)

**SLI**

[Section 8.3.26, “SLI: Shift left and insert”](#)

**SM4E**

[Section 9.5.1, “SM4E: SM4 encryption and decryption”](#)

**SM4EKEY**

[Section 9.5.2, “SM4EKEY: SM4 key updates”](#)

**SMAx (immediate)****SMAx (vectors)**

[Section 6.7.16, “MAX: Integer maximum”](#)

**SMAXP**

[Section 8.7.3, “MAXP: Integer pairwise maximum”](#)

**SMA XV**

[Section 6.10.2, “MAXV: Integer maximum reduction”](#)

S MIN (immediate)

S MIN (vectors)

[Section 6.7.17, “MIN: Integer minimum”](#)

**S MINP**

[Section 8.7.6, “MINP: Integer pairwise minimum”](#)

**S MINV**

[Section 6.10.3, “MINV: Integer minimum reduction”](#)

S MLALB (indexed)

S MLALB (vectors)

[Section 8.4.15, “MLALB: Integer addition of product long \(bottom\)”](#)

S MLALT (indexed)

S MLALT (vectors)

[Section 8.4.16, “MLALT: Integer addition of product long \(top\)”](#)

S MLSLB (indexed)

S MLSLB (vectors)

[Section 8.4.17, “MLSLB: Integer subtraction of product long \(bottom\)”](#)

S MLSLT (indexed)

S MLSLT (vectors)

[Section 8.4.18, “MLSLT: Integer subtraction of product long \(top\)”](#)

**S MMLA**

[Section 7.3.1, “MMLA: Accumulating widening multiplication of integer matrices”](#)

S MULH (predicated)

S MULH (unpredicated)

[Section 6.7.8, “MULH: Integer multiplication, returning high half”](#)

S MULLB (indexed)

S MULLB (vectors)

[Section 8.4.13, “MULLB: Integer multiplication long \(bottom\)”](#)

S MULLT (indexed)

S MULLT (vectors)

[Section 8.4.14, “MULLT: Integer multiplication long \(top\)”](#)

**SPLICE**

[Section 6.20.6, “SPLICE: Splice two vectors under predicate control”](#)

**SQABS**

[Section 8.3.23, “QABS: Saturating integer absolute value”](#)

**SQADD (immediate)****SQADD (vectors, unpredicated)**

[Section 6.7.2, “QADD: Saturating integer addition”](#)

[Section 8.3.1, “QADD: Saturating integer addition”](#)

**SQADD (vectors, predicated)**

[Section 8.3.1, “QADD: Saturating integer addition”](#)

**SQCADD**

[Section 8.12.2, “QCADD: Integer complex saturating addition with rotation”](#)

**SQDECB**

[Section 6.28.6, “QDECB: Saturating decrement by a multiple of `svcntb`”](#)

**SQDECD (scalar)****SQDECD (vector)**

[Section 6.28.9, “QDECD: Saturating decrement by a multiple of `svcntd`”](#)

**SQDECH (scalar)****SQDECH (vector)**

[Section 6.28.7, “QDECH: Saturating decrement by a multiple of `svcnth`”](#)

**SQDECP (scalar)****SQDECP (vector)**

[Section 6.28.10, “QDECP: Saturating decrement by a multiple of `svcntp`”](#)

**SQDECW (scalar)****SQDECW (vector)**

[Section 6.28.8, “QDECW: Saturating decrement by a multiple of `svcntw`”](#)

**SQINCB**

[Section 6.28.1, “QINCB: Saturating increment by a multiple of `svcntb`”](#)

**SQINCD (scalar)****SQINCD (vector)**

[Section 6.28.4, “QINCD: Saturating increment by a multiple of `svcntd`”](#)

**SQINCH (scalar)****SQINCH (vector)**

[Section 6.28.2, “QINCH: Saturating increment by a multiple of `svcnth`”](#)

SQINCP (scalar)

SQINCP (vector)

Section 6.28.5, “QINCP: Saturating increment by a multiple of `svcntp`”

SQINCW (scalar)

SQINCW (vector)

Section 6.28.3, “QINCW: Saturating increment by a multiple of `svcntw`”

SQDMLALB (indexed)

SQDMLALB (vectors)

Section 8.4.23, “QDMLALB: Saturating integer addition of doubled product long (bottom)”

SQDMLALBT

Section 8.13.4, “QDMLALBT: Saturating integer addition of doubled product long (bottom  $\times$  top)”

SQDMLALT (indexed)

SQDMLALT (vectors)

Section 8.4.24, “QDMLALT: Saturating integer addition of doubled product long (top)”

SQDMLSLB (indexed)

SQDMLSLB (vectors)

Section 8.4.25, “QDMLSLB: Saturating integer subtraction of doubled product long (bottom)”

SQDMLSLBT

Section 8.13.5, “QDMLSLBT: Saturating integer subtraction of doubled product long (bottom  $\times$  top)”

SQDMLSLT (indexed)

SQDMLSLT (vectors)

Section 8.4.26, “QDMLSLT: Saturating integer subtraction of doubled product long (top)”

SQDMULH (indexed)

SQDMULH (vectors)

Section 8.3.11, “QDMULH: Doubling integer multiplication, saturated high half”

SQDMULLB (indexed)

SQDMULLB (vectors)

Section 8.4.21, “QDMULLB: Saturating integer doubled product long (bottom)”

SQDMULLT (indexed)

SQDMULLT (vectors)

Section 8.4.22, “QDMULLT: Saturating integer doubled product long (top)”

SQNEG

Section 8.3.22, “QNEG: Saturating integer negation”

SQRDCMLAH (indexed)  
SQRDCMLAH (vectors)

Section 8.12.4, “QRDCMLAH: Doubling complex integer multiplication with rotation, saturating addition of rounded high half”

SQRDMLAH (indexed)  
SQRDMLAH (vectors)

Section 8.3.13, “QRDMLAH: Doubling integer multiplication, saturating addition of rounded high half”

SQRDMLSH (indexed)  
SQRDMLSH (vectors)

Section 8.3.14, “QRDMLSH: Doubling integer multiplication, saturating subtraction of rounded high half”

SQRDMULH (indexed)  
SQRDMULH (vectors)

Section 8.3.12, “QRDMULH: Doubling integer multiplication, saturated rounded high half”

SQRSHL  
SQRSHLR

Section 8.3.19, “QRSHL: Saturating rounding shift left”

SQRSHRNB

Section 8.5.17, “QRSHRNB: Rounding shift right, saturating narrowing (bottom)”

SQRSHRNT

Section 8.5.18, “QRSHRNT: Rounding shift right, saturating narrowing (top)”

SQRSHRUNB

Section 8.5.19, “QRSHRUNB: Rounding shift right, saturating narrowing to unsigned (bottom)”

SQRSHRUNT

Section 8.5.20, “QRSHRUNT: Rounding shift right, saturating narrowing to unsigned (top)”

SQSHL (immediate)  
SQSHL (vectors)  
SQSHLR

Section 8.3.17, “QSHL: Saturating shift left”

SQSHLU

Section 8.3.18, “QSHLU: Saturating shift left with unsigned result”

SQSHRNB

Section 8.5.13, “QSHRNB: Shift right, saturating narrowing (bottom)”

**SQSHRNT**

[Section 8.5.14, “QSHRNT: Shift right, saturating narrowing \(top\)”](#)

**SQSHRUNB**

[Section 8.5.15, “QSHRUNB: Shift right, saturating narrowing to unsigned \(bottom\)”](#)

**SQSHRUNT**

[Section 8.5.16, “QSHRUNT: Shift right, saturating narrowing to unsigned \(top\)”](#)

**SQSUB (immediate)****SQSUB (vectors, unpredicated)**

[Section 6.7.5, “QSUB: Saturating integer subtraction”](#)

[Section 8.3.6, “QSUB: Saturating integer subtraction”](#)

**SQSUB (vectors, predicated)****SQSUBR**

[Section 8.3.6, “QSUB: Saturating integer subtraction”](#)

**SQXTNB**

[Section 8.6.1, “QXTNB: Saturating narrowing \(bottom\)”](#)

**SQXTNT**

[Section 8.6.2, “QXTNT: Saturating narrowing \(top\)”](#)

**SQXTUNB**

[Section 8.6.3, “QXTUNB: Saturating narrowing to unsigned \(bottom\)”](#)

**SQXTUNT**

[Section 8.6.4, “QXTUNT: Saturating narrowing to unsigned \(top\)”](#)

**SRHADD**

[Section 8.3.5, “RHADD: Rounding halving integer addition”](#)

**SRI**

[Section 8.3.27, “SRI: Shift right and insert”](#)

**SRSHL****SRSHLR**

[Section 8.3.15, “RSHL: Rounding shift left”](#)

**SRSHR**

[Section 8.3.16, “RSHR: Rounding shift right”](#)

**SRSRA**

[Section 8.3.21, “RSRA: Rounding shift right and accumulate”](#)

**SSHLLB**

Section 8.4.27, “SHLLB: Shift left long by immediate (bottom)”  
Section 8.4.19, “MOVLB: Move long (bottom)”

**SSHLLT**

Section 8.4.28, “SHLLT: Shift left long by immediate (top)”  
Section 8.4.20, “MOVLT: Move long (top)”

**SSRA**

Section 8.3.20, “SRA: Shift right and accumulate”

**SSUBLB**

Section 8.4.5, “SUBLB: Integer subtraction long (bottom)”

**SSUBLBT**

Section 8.13.2, “SUBLBT: Integer subtraction long (bottom - top)”

**SSUBLT**

Section 8.4.6, “SUBLT: Integer subtraction long (top)”

**SSUBLTB**

Section 8.13.3, “SUBLTB: Integer subtraction long (top - bottom)”

**SSUBWB**

Section 8.4.7, “SUBWB: Integer subtraction wide (bottom)”

**SSUBWT**

Section 8.4.8, “SUBWT: Integer subtraction wide (top)”

ST1B (scalar plus immediate)  
ST1B (scalar plus scalar)  
ST1B (scalar plus vector)  
ST1B (vector plus immediate)

Section 6.3.1, “ST1: Store one vector, with no truncation”  
Section 6.3.2, “ST1B: Store one vector, truncating to 8 bits”

ST1D (scalar plus immediate)  
ST1D (scalar plus scalar)  
ST1D (scalar plus vector)  
ST1D (vector plus immediate)

Section 6.3.1, “ST1: Store one vector, with no truncation”

ST1H (scalar plus immediate)  
ST1H (scalar plus scalar)  
ST1H (scalar plus vector)  
ST1H (vector plus immediate)

Section 6.3.1, “ST1: Store one vector, with no truncation”

[Section 6.3.3, “ST1H: Store one vector, truncating to 16 bits”](#)

ST1W (scalar plus immediate)  
ST1W (scalar plus scalar)  
ST1W (scalar plus vector)  
ST1W (vector plus immediate)

[Section 6.3.1, “ST1: Store one vector, with no truncation”](#)[Section 6.3.4, “ST1W: Store one vector, truncating to 32 bits”](#)

ST2B (scalar plus immediate)  
ST2B (scalar plus scalar)  
ST2D (scalar plus immediate)  
ST2D (scalar plus scalar)  
ST2H (scalar plus immediate)  
ST2H (scalar plus scalar)  
ST2W (scalar plus immediate)  
ST2W (scalar plus scalar)

[Section 6.3.6, “ST2: Store two vectors into two-element structures”](#)

ST3B (scalar plus immediate)  
ST3B (scalar plus scalar)  
ST3D (scalar plus immediate)  
ST3D (scalar plus scalar)  
ST3H (scalar plus immediate)  
ST3H (scalar plus scalar)  
ST3W (scalar plus immediate)  
ST3W (scalar plus scalar)

[Section 6.3.7, “ST3: Store three vectors into three-element structures”](#)

ST4B (scalar plus immediate)  
ST4B (scalar plus scalar)  
ST4D (scalar plus immediate)  
ST4D (scalar plus scalar)  
ST4H (scalar plus immediate)  
ST4H (scalar plus scalar)  
ST4W (scalar plus immediate)  
ST4W (scalar plus scalar)

[Section 6.3.8, “ST4: Store four vectors into four-element structures”](#)

STNT1B (scalar plus immediate)  
STNT1B (scalar plus scalar)  
STNT1D (scalar plus immediate)  
STNT1D (scalar plus scalar)  
STNT1H (scalar plus immediate)  
STNT1H (scalar plus scalar)  
STNT1W (scalar plus immediate)  
STNT1W (scalar plus scalar)

[Section 6.3.5, “STNT1: Store one vector, with no truncation, non-temporal”](#)

STNT1B (vector plus scalar)

[Section 8.23.8, “STNT1: Store one vector, with no truncation, non-temporal”](#)



Section 8.23.9, “STNT1B: Store one vector, truncating to 8 bits, non-temporal”

STNT1D (vector plus scalar)

Section 8.23.8, “STNT1: Store one vector, with no truncation, non-temporal”

STNT1H (vector plus scalar)

Section 8.23.8, “STNT1: Store one vector, with no truncation, non-temporal”

Section 8.23.10, “STNT1H: Store one vector, truncating to 16 bits, non-temporal”

STNT1W (vector plus scalar)

Section 8.23.8, “STNT1: Store one vector, with no truncation, non-temporal”

Section 8.23.11, “STNT1W: Store one vector, truncating to 32 bits, non-temporal”

STR (predicate)

STR (vector)

No direct support, although the compiler can use these instructions to spill registers to the stack.

SUB (immediate)

SUB (vectors, predicated)

SUB (vectors, unpredicated)

Section 6.7.3, “SUB: Modular integer subtraction”

SUBHNB

Section 8.5.5, “SUBHNB: Integer subtraction, narrowing to high half (bottom)”

SUBHNT

Section 8.5.6, “SUBHNT: Integer subtraction, narrowing to high half (top)”

SUBR (immediate)

SUBR (vectors)

Section 6.7.4, “SUBR: Modular integer subtraction, reversed”

SUDOT

Section 7.3.4, “SUDOT: Integer addition of dot product (signed×unsigned)”

SUQADD

Section 8.3.3, “UQADD: Saturating integer addition of unsigned value”

SUNPKHI

Section 6.20.15, “UNPKHI: Unpack and extend high half of an input”

SUNPKLO

Section 6.20.16, “UNPKLO: Unpack and extend low half of an input”

SXTB

Section 6.14.1, “EXTB: Extend from low 8 bits”

**SXTH**

Section 6.14.2, “EXTH: Extend from low 16 bits”

**SXTW**

Section 6.14.3, “EXTW: Extend from low 32 bits”

**TBL**

Section 6.20.11, “TBL: Table lookup/permute using vector of indices”

Section 6.20.9, “DUP: Duplicate one element of a vector”

Section 8.22.1, “TBL2: Table lookup/permute of two vectors using vector of indices”

**TBX**

Section 8.22.2, “TBX: Table lookup/permute using vector of indices (merging)”

**TRN1 (predicates)****TRN1 (vectors)**

Section 6.20.13, “TRN1: Interleave even elements from two inputs”

Section 7.5.3, “TRN1Q: Interleave even quadwords from two inputs”

**TRN2 (predicates)****TRN2 (vectors)**

Section 6.20.14, “TRN2: Interleave odd elements from two inputs”

Section 7.5.4, “TRN2Q: Interleave odd quadwords from two inputs”

**UABA**

Section 8.3.10, “ABA: Integer addition of absolute difference”

**UABALB**

Section 8.4.11, “ABALB: Integer addition of absolute difference long (bottom)”

**UABALT**

Section 8.4.12, “ABALT: Integer addition of absolute difference long (top)”

**UABD**

Section 6.7.6, “ABD: Integer absolute difference”

**UABDLB**

Section 8.4.9, “ABDLB: Integer absolute difference long (bottom)”

**UABDLT**

Section 8.4.10, “ABDLT: Integer absolute difference long (top)”

**UADALP**

Section 8.8.1, “ADALP: Integer pairwise addition and accumulate”

**UADDLB**

[Section 8.4.1, “ADDLB: Integer addition long \(bottom\)”](#)

**UADDLT**

[Section 8.4.2, “ADDLT: Integer addition long \(top\)”](#)

**UADDV**

[Section 6.10.1, “ADDV: Integer addition reduction”](#)

**UADDWB**

[Section 8.4.3, “ADDWB: Integer addition wide \(bottom\)”](#)

**UADDWT**

[Section 8.4.4, “ADDWT: Integer addition wide \(top\)”](#)

**UCVTF**

[Section 6.19.2, “CVT: Convert integer value to floating-point”](#)

**UDIV**

[Section 6.7.14, “DIV: Integer division”](#)

**UDIVR**

[Section 6.7.15, “DIVR: Integer division, reversed”](#)

**UDOT (indexed)****UDOT (vectors)**

[Section 6.7.13, “DOT: Integer addition of dot product”](#)

**UHADD**

[Section 8.3.4, “HADD: Halving integer addition”](#)

**UHSUB**

[Section 8.3.8, “HSUB: Halving integer subtraction”](#)

**UHSUBR**

[Section 8.3.9, “HSUBR: Halving integer subtraction, reversed”](#)

**UMAX (immediate)****UMAX (vectors)**

[Section 6.7.16, “MAX: Integer maximum”](#)

**UMAXP**

[Section 8.7.3, “MAXP: Integer pairwise maximum”](#)

## UMAXV

[Section 6.10.2, “MAXV: Integer maximum reduction”](#)

UMIN (immediate)

UMIN (vectors)

[Section 6.7.17, “MIN: Integer minimum”](#)

## UMINP

[Section 8.7.6, “MINP: Integer pairwise minimum”](#)

## UMINV

[Section 6.10.3, “MINV: Integer minimum reduction”](#)

UMLALB (indexed)

UMLALB (vectors)

[Section 8.4.15, “MLALB: Integer addition of product long \(bottom\)”](#)

UMLALT (indexed)

UMLALT (vectors)

[Section 8.4.16, “MLALT: Integer addition of product long \(top\)”](#)

UMLSLB (indexed)

UMLSLB (vectors)

[Section 8.4.17, “MLSLB: Integer subtraction of product long \(bottom\)”](#)

UMLSLT (indexed)

UMLSLT (vectors)

[Section 8.4.18, “MLSLT: Integer subtraction of product long \(top\)”](#)

## UMMLA

[Section 7.3.1, “MMLA: Accumulating widening multiplication of integer matrices”](#)

UMULH (predicated)

UMULH (unpredicated)

[Section 6.7.8, “MULH: Integer multiplication, returning high half”](#)

UMULLB (indexed)

UMULLB (vectors)

[Section 8.4.13, “MULLB: Integer multiplication long \(bottom\)”](#)

UMULLT (indexed)

UMULLT (vectors)

[Section 8.4.14, “MULLT: Integer multiplication long \(top\)”](#)

UQADD (immediate)

UQADD (vectors, unpredicated)

[Section 6.7.2, “QADD: Saturating integer addition”](#)

### Section 8.3.1, “QADD: Saturating integer addition”

UQADD (vectors, predicated)

### Section 8.3.1, “QADD: Saturating integer addition”

UQDECB

### Section 6.28.6, “QDECB: Saturating decrement by a multiple of `svcntb`”

UQDECD (scalar)

UQDECD (vector)

### Section 6.28.9, “QDECD: Saturating decrement by a multiple of `svcntd`”

UQDECH (scalar)

UQDECH (vector)

### Section 6.28.7, “QDECH: Saturating decrement by a multiple of `svcnth`”

UQDECP (scalar)

UQDECP (vector)

### Section 6.28.10, “QDECP: Saturating decrement by a multiple of `svcntp`”

UQDECW (scalar)

UQDECW (vector)

### Section 6.28.8, “QDECW: Saturating decrement by a multiple of `svcntw`”

UQINCB

### Section 6.28.1, “QINCB: Saturating increment by a multiple of `svcntb`”

UQINCD (scalar)

UQINCD (vector)

### Section 6.28.4, “QINCD: Saturating increment by a multiple of `svcntd`”

UQINCH (scalar)

UQINCH (vector)

### Section 6.28.2, “QINCH: Saturating increment by a multiple of `svcnth`”

UQINCP (scalar)

UQINCP (vector)

### Section 6.28.5, “QINCP: Saturating increment by a multiple of `svcntp`”

UQINCW (scalar)

UQINCW (vector)

### Section 6.28.3, “QINCW: Saturating increment by a multiple of `svcntw`”

UQRSHL

UQRSHLR

### Section 8.3.19, “QRSHL: Saturating rounding shift left”

**UQRSHRNB**

[Section 8.5.17, “QRSHRNB: Rounding shift right, saturating narrowing \(bottom\)”](#)

**UQRSHRNT**

[Section 8.5.18, “QRSHRNT: Rounding shift right, saturating narrowing \(top\)”](#)

**UQSHL (immediate)****UQSHL (vectors)****UQSHLR**

[Section 8.3.17, “QSHL: Saturating shift left”](#)

**UQSHRNB**

[Section 8.5.13, “QSHRNB: Shift right, saturating narrowing \(bottom\)”](#)

**UQSHRNT**

[Section 8.5.14, “QSHRNT: Shift right, saturating narrowing \(top\)”](#)

**UQSUB (immediate)****UQSUB (vectors, unpredicated)**

[Section 6.7.5, “QSUB: Saturating integer subtraction”](#)

[Section 8.3.6, “QSUB: Saturating integer subtraction”](#)

**UQSUB (vectors, predicated)****UQSUBR**

[Section 8.3.6, “QSUB: Saturating integer subtraction”](#)

**UQXTNB**

[Section 8.6.1, “QXTNB: Saturating narrowing \(bottom\)”](#)

**UQXTNT**

[Section 8.6.2, “QXTNT: Saturating narrowing \(top\)”](#)

**URECPE**

[Section 8.3.24, “RECPE: Integer reciprocal estimate”](#)

**URHADD**

[Section 8.3.5, “RHADD: Rounding halving integer addition”](#)

**URSHL****URSHLR**

[Section 8.3.15, “RSHL: Rounding shift left”](#)

**URSHR**

[Section 8.3.16, “RSHR: Rounding shift right”](#)

**URSQRTE**

Section 8.3.25, “RSQRTE: Integer reciprocal square root estimate”

**URSRA**

Section 8.3.21, “RSRA: Rounding shift right and accumulate”

**USDOT**

Section 7.3.3, “USDOT: Integer addition of dot product (unsigned×signed)”

Section 7.3.4, “SUDOT: Integer addition of dot product (signed×unsigned)”

**USHLLB**

Section 8.4.27, “SHLLB: Shift left long by immediate (bottom)”

Section 8.4.19, “MOVLB: Move long (bottom)”

**USHLLT**

Section 8.4.28, “SHLLT: Shift left long by immediate (top)”

Section 8.4.20, “MOVLT: Move long (top)”

**USMMLA**

Section 7.3.2, “USMMLA: Accumulating widening multiplication of integer matrices (unsigned×signed)”

**USQADD**

Section 8.3.2, “SQADD: Saturating integer addition of signed value”

**USRA**

Section 8.3.20, “SRA: Shift right and accumulate”

**USUBLB**

Section 8.4.5, “SUBLB: Integer subtraction long (bottom)”

**USUBLT**

Section 8.4.6, “SUBLT: Integer subtraction long (top)”

**USUBWB**

Section 8.4.7, “SUBWB: Integer subtraction wide (bottom)”

**USUBWT**

Section 8.4.8, “SUBWT: Integer subtraction wide (top)”

**UUNPKHI**

Section 6.20.15, “UNPKHI: Unpack and extend high half of an input”

**UUNPKLO**

Section 6.20.16, “UNPKLO: Unpack and extend low half of an input”

**UXTB**

[Section 6.14.1, “EXTB: Extend from low 8 bits”](#)

**UXTH**

[Section 6.14.2, “EXTH: Extend from low 16 bits”](#)

**UXTW**

[Section 6.14.3, “EXTW: Extend from low 32 bits”](#)

**UZP1 (predicates)****UZP1 (vectors)**

[Section 6.20.17, “UZP1: Select even elements from two inputs”](#)

[Section 7.5.5, “UZP1Q: Select even quadwords from two inputs”](#)

**UZP2 (predicates)****UZP2 (vectors)**

[Section 6.20.18, “UZP2: Select odd elements from two inputs”](#)

[Section 7.5.6, “UZP2Q: Select odd quadwords from two inputs”](#)

**WHILEGE**

[Section 8.2.2, “WHILEGE: While decrementing variable is greater than or equal to”](#)

**WHILEGT**

[Section 8.2.1, “WHILEGT: While decrementing variable is greater than”](#)

**WHILEHI**

[Section 8.2.1, “WHILEGT: While decrementing variable is greater than”](#)

**WHILEHS**

[Section 8.2.2, “WHILEGE: While decrementing variable is greater than or equal to”](#)

**WHILELE**

[Section 6.12.2, “WHILELE: While incrementing variable is less than or equal to”](#)

**WHILELO**

[Section 6.12.1, “WHILELT: While incrementing variable is less than”](#)

**WHILELS**

[Section 6.12.2, “WHILELE: While incrementing variable is less than or equal to”](#)

**WHILELT**

[Section 6.12.1, “WHILELT: While incrementing variable is less than”](#)

**WHILERW**

[Section 8.20.1, “WHILERW: While free of read-after-write conflicts”](#)



**WHILEWR**

[Section 8.20.2, “WHILEWR: While free of write-after-read conflicts”](#)

**WRFFR**

[Section 6.26.3, “WRFFR: Write to the first-fault register”](#)

**XAR**

[Section 8.9.7, “XAR: Bitwise exclusive OR and rotate right by immediate”](#)

**ZIP1 (predicates)****ZIP1 (vectors)**

[Section 6.20.19, “ZIP1: Interleave elements from low halves of two inputs”](#)

[Section 7.5.7, “ZIP1Q: Interleave quadwords from low halves of two inputs”](#)

**ZIP2 (predicates)****ZIP2 (vectors)**

[Section 6.20.20, “ZIP2: Interleave elements from high halves of two inputs”](#)

[Section 7.5.8, “ZIP2Q: Interleave quadwords from high halves of two inputs”](#)

## 10.2. CTERMEQ and CTERMNE

CTERMEQ and CTERMNE provide a way of optimizing conditions such as:

```
svbool_t p1, p2;
uint64_t x, y;
...
if (svptest_last(p1, p2) && x == y)
    ...stmts...
```

Here the compiler could use PTEST to test whether the last active element of p2 is active and follow it by CTERMNE to test whether x is not equal to y. The code should then execute *stmts* if the LT condition holds (i.e. if the last element is active and the “termination condition”  $x \neq y$  does not hold).

There are no ACLE functions that perform a combined predicate and scalar test since the separate tests should be more readable.

## 10.3. ADDPL, ADDVL, INC and DEC

The architecture provides instructions for adding or subtracting an element count, where the count might come from a pattern (such as [Section 6.27.2, “CNTB: Count the number of 8-bit elements in a pattern”](#)) or from a predicate ([Section 6.27.1, “CNTP: Count active elements”](#)). For example, INCH adds the number of halfwords in a pattern to a scalar register or to every element of a vector register.

At the C and C++ level it is usually simpler to write the addition out normally. Also, having functions that map directly to architectural instructions like INCH could cause confusion for pointers, since the instructions would operate on the raw integer value of the pointer. For example, applying INCH to a pointer to halfwords would effectively convert the pointer to an integer, add the number of halfwords, and then convert the result back to a pointer. The pointer would then only advance by half a vector.

For these reasons there are no dedicated functions for:

ADDPL

DECB

DECP

INCD

ADDVL	DECH	INCB	INCP
	DECW	INCH	
	DECD	INCW	

Instead the compiler should use these instructions to optimize things like:

```
svbool_t p1;
...
x += svcntp_b16(p1, p1);
```

The same concerns do not apply to saturating scalar arithmetic, since using saturating arithmetic only makes sense for displacements rather than pointers. There is also no standard way of doing a saturating addition or subtraction of arbitrary integers. The ACLE does therefore provide functions for:

SQDECB	SQINCB	UQDECB	UQINCB
SQDECH	SQINCH	UQDECH	UQINCH
SQDECW	SQINCW	UQDECW	UQINCW
SQDECD	SQINCD	UQDECD	UQINCD
SQDECP	SQINCP	UQDECP	UQINCP

## A. Sizeless types in C

This section specifies the behavior for sizeless types as an edit to the N1570 version of the C standard.

### 6.2.5 Types

In 6.2.5/1, replace:

At various points within a translation unit an object type may be *incomplete* ...

onwards with:

Object types are further partitioned into *sized* and *sizeless*; all basic and derived types defined in this standard are sized, but an implementation may provide additional sizeless types.

and add two additional clauses:

- At various points within a translation unit an object type may be *indefinite* (lacking sufficient information to construct an object of that type) or *definite* (having sufficient information). An object type is said to be *complete* if it is both sized and definite; all other object types are said to be *incomplete*. Complete types have sufficient information to determine the size of an object of that type while incomplete types do not.
- Arrays, structures, unions and enumerated types are always sized, so for them the term *incomplete* is equivalent to (and used interchangeably with) the term *indefinite*.

Change 6.2.5/19 to:

The `void` type comprises an empty set of values; it is a sized indefinite object type that cannot be completed (made definite).

Replace *incomplete* with *indefinite* and *complete* with *definite* in 6.2.5/37, which describes how a type's state can change throughout a translation unit.

#### 6.3.2.1 Lvalues, arrays, and function designators

Replace *incomplete* with *indefinite* in 6.3.2.1/1, so that sizeless definite types are modifiable lvalues.

Make the same replacement in 6.3.2.1/2, to prevent undefined behavior when lvalues have sizeless definite type.

#### 6.5.1.1 Generic selection

Replace *complete object type* with *definite object type* in 6.5.1.1/2, so that the type name in a generic association can be a sizeless definite type.

#### 6.5.2.2 Function calls

Replace *complete object type* with *definite object type* in 6.5.2.2/1, so that functions can return sizeless definite types.

Make the same change in 6.5.2.2/4, so that arguments can also have sizeless definite type.

#### 6.5.2.5 Compound literals

Replace *complete object type* with *definite object type* in 6.5.2.5/1, so that compound literals can have sizeless definite type.

#### 6.7 Declarations

Insert the following new clause after 6.7/4:

- If an identifier for an object does not have automatic storage duration, its type must be sized rather than sizeless.

Replace *complete* with *definite* in 6.7/7, which describes when the type of an object becomes definite.

#### 6.7.6.3 Function declarators (including prototypes)

Replace *incomplete type* with *indefinite type* in 6.7.6.3/4, so that parameters can also have sizeless definite type.

Make the same change in 6.7.6.3/12, which allows even indefinite types to be function parameters if no function definition is present.

#### 6.7.9 Initialization

Replace *complete object type* with *definite object type* in 6.7.9/3, to allow initialization of identifiers with sizeless definite type.

#### 6.9.1 Function definitions

Replace *complete object type* with *definite object type* in 6.9.1/3, so that functions can return sizeless definite types.

Make the same change in 6.9.1/7, so that adjusted parameter types can be sizeless definite types.

#### J.2 Undefined behavior

Update the entries that refer to the clauses above.

## B. Sizeless types in C++

This section specifies the behavior for sizeless types as an edit to the N3797 version of the C++ standard.

### 3.1 Declarations and definitions [basic.def]

Replace *incomplete type* with *indefinite type* in 3.1/5, so that objects can have sizeless definite type in some situations. Add a new clause immediately afterwards:

- A program is ill-formed if any declaration of an object gives it both a sizeless type and either static or thread-local storage duration.

### 3.9 Types [basic.types]

Replace 3.9/5 with these clauses:

- A class that has been declared but not defined, an enumeration type in certain contexts, or an array of unknown size or of indefinite element type, is an *indefinite object type*. Indefinite object types and the void types are *indefinite types*. Objects shall not be defined to have an indefinite type.
- Object and void types are further partitioned into *sized* and *sizeless*; all basic and derived types defined in this standard are sized, but an implementation may provide additional sizeless types.
- An object or void type is said to be *complete* if it is both sized and definite; all other object and void types are said to be *incomplete*. The term *completely-defined object type* is synonymous with *complete object type*.
- Arrays, class types and enumeration types are always sized, so for them the term *incomplete* is equivalent to (and used interchangeably with) the term *indefinite*.

Replace *incomplete types* with *indefinite types* in 3.9/7, which is simply a cross-reference note.

#### 3.9.1 Fundamental Types [basic.fundamental]

Replace the second sentence of 3.9.1/9 with:

The `void` type is a sized indefinite type that cannot be completed (made definite).

#### 3.9.2. Compound Types [basic.compound]

Add “(including indefinite types)” after “Pointers to incomplete types” in 3.9.2/3, to emphasize that pointers to incomplete types are more restricted than pointers to complete types, even if the types are definite.

### 3.10 Lvalues and rvalues [basic.lval]

Replace *complete types* with *definite types* and *incomplete types* with *indefinite types* in 3.10/4, so that prvalues can be sizeless definite types.

Replace *incomplete* with *indefinite* and *complete* with *definite* in 3.10/7, which describes the modifiability of a pointer and the object to which it points.

#### 4.1 Lvalue-to-rvalue conversion [conv.lval]

Replace *incomplete type* with *indefinite type* in 4.1/1, so that glvalues of sizeless definite type can be converted to prvalues.

### 5.2.2 Function call [expr.call]

Replace *completely-defined object type* with *definite object type* and *incomplete class type* with *indefinite class type*, so that parameters can have sizeless definite type.

Replace *incomplete* with *indefinite* and *complete* with *definite* in 5.2.2/11, so that functions can return sizeless definite types.

### 5.2.3 Explicit type conversion (function notation) [expr.type.conv]

Replace *complete object type* with *definite object type* in 5.2.3/2, so that the  $T()$  notation extends to sizeless definite types.

### 5.3.1 Unary operators [expr.unary.op]

Replace *incomplete type* with *indefinite type* in 5.3.1/1. This simply updates a cross-reference to the 4.1 change above.

### 5.3.5 Delete [expr.delete]

Insert the following after the first sentence of 5.3.5/2, (which describes the implicit conversion of an object type to a pointer type):

The type of the operand must now be a pointer to a sized type, otherwise the program is ill-formed.

### 8.3.4 Arrays [dcl.array]

In 8.3.4/1, add *a sizeless type* to the list of types that the element type  $T$  cannot have.

### 9.4.2 Static data members [class.static.data]

Replace *an incomplete type* with *a sized indefinite type* in 9.4.2/2, so that static data members cannot be declared with sizeless type.

Add a new final clause:

- A static data member shall not have sizeless type.

### 14.3.1 Template type parameters [temp.arg.type]

Add “(including an indefinite type)” after “an incomplete type” in the note describing template type arguments, to emphasize that the arguments can be sizeless.

### 20.10.4.3 Type properties [meta.unary.prop]

Replace *complete* with *definite* in 20.10.4.3/3, so that the situations in which an implementation may implicitly instantiate template arguments remain the same as before.

### 20.10.6 Relationships between types [meta.rel]

Replace *complete* with *definite* in 20.10.6/2, so that these metaprogramming facilities extend to sized definite types.

### 20.10.7.5 Pointer modifications [meta.trans.ptr]

Likewise replace *complete* with *definite* in the initial table, so that these metaprogramming facilities also extend to sized definite types.