ARM C Language Extensions for SVE

Version 00bet1

Abstract

This document is a beta version of the ARM C language extensions (ACLE) for the ARM Scalable Vector Extension (SVE). 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 instructions.

Keywords

ACLE, SVE, C, C++

Confidentiality status

This document is non-confidential.

Proprietary notice

ARM Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

LES-PRE-20349

Table of Contents

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

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)
C++14	ISO/IEC 14882:2014	ISO	Standard C++ (based on draft N3797)

1.3. Terms and abbreviations

This document uses the following terms and abbreviations:

Term	Meaning
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-specific part of the ACLE; within the document, the term ACLE generally refers to the SVE-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 (Section 4.7, “First-faulting and non-faulting loads”).
SIMD	Single Instruction Multiple Data.

Term	Meaning
sizeless type	A C and C++ type that can be used to create objects, but that has no measurable size (Section 3.2, “Sizeless types”)
SVE	The ARMv8-A Scalable Vector Extension.
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). 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 instructions. Section 7.1, “List of instructions” lists each SVE 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 instructions.

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

2.1. Feature macro

The feature macro `__ARM_FEATURE_SVE` is defined to 1 if SVE is supported and if the extensions described in this document are available. Values higher than 1 are reserved.

2.2. Header file

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

```
#ifdef __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*×64. Predicates have one bit for each vector byte, so the number of usable bits in a predicate is *VG*×8.

Code that makes direct use of SVE instructions therefore needs access to vector types that have *VG*×64 bits and predicate types that have *VG*×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.[1] 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;

[1] 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[2].

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:

[2] 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`

3.4. Vector types

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

```
svint8_t                svuint8_t
svint16_t               svuint16_t              svfloat16_t
svint32_t               svuint32_t              svfloat32_t
svint64_t               svuint64_t              svfloat64_t
```

Each type `sv`*BASE*`_t` is a 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.

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 `svcvt` 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:

```
svint8x2_t              svuint8x2_t
svint16x2_t             svuint16x2_t            svfloat16x2_t
svint32x2_t             svuint32x2_t            svfloat32x2_t
svint64x2_t             svuint64x2_t            svfloat64x2_t

svint8x3_t              svuint8x3_t
svint16x3_t             svuint16x3_t            svfloat16x3_t
svint32x3_t             svuint32x3_t            svfloat32x3_t
svint64x3_t             svuint64x3_t            svfloat64x3_t

svint8x4_t              svuint8x4_t
svint16x4_t             svuint16x4_t            svfloat16x4_t
svint32x4_t             svuint32x4_t            svfloat32x4_t
svint64x4_t             svuint64x4_t            svfloat64x4_t
```

Each type `sv`*BASE*`x`*N*`_t` is sizeless and contains a sequence of *N* `sv`*BASE*`_t`s. The individual vectors are members with names `v0`, `v1`, and so on. For example, `svfloat64x4_t` contains four `svfloat64_t` vectors with the names `v0`, `v1`, `v2` and `v3`.

The ACLE does not provide a way of defining these tuples directly in C or C++, so the exact definition depends on the implementation. However, at a minimum, the implementation must support `.v`*N* as a means of extracting vector *N* from a tuple and both:

```
…(svBASExN_t) { x1, …, xN }…;
svBASExN_t xt = { x1, …, xN };
```

as means of creating tuples from *N* individual vectors `x1`, …, `x`*N*.

Note that the structure of these tuples is different from the corresponding Advanced SIMD types, where something like `float64x2x`*N*`_t` has a single array field `val` with *N* elements. This difference is necessary to avoid arrays of sizeless types.

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 `svbool_t` bit (from lsb)**									
`svint8_t`	0	1	2	3	4	5	6	7	8	…
`svint16_t`	0	-	1	-	2	-	3	-	4	…
`svint32_t`	0	-	-	-	1	-	-	-	2	…
`svint64_t`	0	-	-	-	-	-	-	-	1	…

4. Functions

4.1. Naming convention

The SVE ACLE functions have the form:

sv*base*[*_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_t`s while `svld1sb_u32` sign-extends 8-bit data to a vector of `uint32_t`s.

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 letter followed by an element size in bits[3]:

signed integers	`s8`	`s16`	`s32`	`s64`
unsigned integers	`u8`	`u16`	`u32`	`u64`
floating-point numbers		`f16`	`f32`	`f64`

[3] These suffixes are the same as for `arm_neon.h`.

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.

`x` “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 and predicated comparisons 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[4]. 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 `[_`*`xNN`*`base]`, where *`xNN`* is the type of the address vector elements.

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:

`_offset` The displacement is a 64-bit scalar byte count.

`_index` The displacement is a 64-bit scalar element count. For example, if a function loads 16-bit elements from memory, an `_index` displacement of 1 is equivalent to an `_offset` displacement of 2.

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

`_vnum` 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 `_vnum` displacement of 1 is equivalent to an `_offset` displacement of *VG*×8 (the number of bytes in an SVE vector). If a function loads 16-bit elements and extends them to 32 bits, a `_vnum` displacement of 1 is equivalent to an `_offset` displacement of *VG*×4 (half the number of bytes in an SVE vector).

This form corresponds to the `MUL VL` addressing mode. However, the displacement argument can be any scalar value; it does not need to be a constant in a particular range.

`_[`*xNN*`]offset` The displacement is a vector of type *xNN* and each element specifies a separate byte count. In other words, `_[s32]offset` specifies a vector of signed 32-bit byte counts and is equivalent to the `SXTW` addressing mode. `_[u32]offset` specifies a vector of unsigned 32-bit offsets and is equivalent to the `UXTW` addressing mode. `_[s64]offset` and `_[u64]offset` both specify vectors of 64-bit offsets; the sign is unimportant in this case.

`_[`*xNN*`]index` Similar, but each element specifies an element count rather than a byte count, in the same way as for `_index`.

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.

Some SVE instructions have “wide” forms in which a single 64-bit element is used for all operations on the overlapping elements of another vector. For example, in:

```
CMPEQ Pd.H, Pg/Z, Zn.H, Zm.D
```

there are four `Zn.H` elements for each `Zm.D` element. The instruction uses element 1 of `Zm.D` for the comparison of elements 4 to 7 of `Zn.H`. In these cases the associated scalar functions also take 64-bit values, regardless of the element size. For example:

```
svint16_t x;
int64_t y;
…
x = svcmpeq[_n_s16](pg, x, y);
```

compares every active element of `x` with the full 64-bit value of `y`.

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 Section 6, “List of functions”, 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 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 = svldff1[_f64](p0, x0);
z1 = svldff1[_f64](p1, x1);
z2 = svldff1[_f64](p2, x2);
p3 = svrdffr();
```


5. Enum declarations

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

```
enum sv_pattern
{
  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 sv_prfop
{
  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 functions

6.1. Introduction

This section contains a list of all functions, grouped into categories and then subdivided based on the first part of the name (up to the first underscore).

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
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*)

6.2.1.2. LD1 (scalar base, VL displacement)

Instances
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*)

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*)

Instances
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
svint32_t **svld1_gather**_[s32]**index**[_s32](svbool_t *pg*, const int32_t **base*,

Instances
svint32_t *indices*) svint64_t **svld1_gather**_[s64]**index**[_s64](svbool_t *pg*, const int64_t **base*, svint32_t *indices*) svuint32_t **svld1_gather**_[s32]**index**[_u32](svbool_t *pg*, const uint32_t **base*, svint32_t *indices*) svuint64_t **svld1_gather**_[s64]**index**[_u64](svbool_t *pg*, const uint64_t **base*, svint64_t *indices*) svfloat32_t **svld1_gather**_[s32]**index**[_f32](svbool_t *pg*, const float32_t **base*, svint32_t *indices*) svfloat64_t **svld1_gather**_[s64]**index**[_f64](svbool_t *pg*, const float64_t **base*, svint64_t *indices*) svint32_t **svld1_gather**_[u32]**index**[_s32](svbool_t *pg*, const int32_t **base*, svuint32_t *indices*) svint64_t **svld1_gather**_[u64]**index**[_s64](svbool_t *pg*, const int64_t **base*, svuint64_t *indices*) svuint32_t **svld1_gather**_[u32]**index**[_u32](svbool_t *pg*, const uint32_t **base*, svuint32_t *indices*) svuint64_t **svld1_gather**_[u64]**index**[_u64](svbool_t *pg*, const uint64_t **base*, svuint64_t *indices*) svfloat32_t **svld1_gather**_[u32]**index**[_f32](svbool_t *pg*, const float32_t **base*, svuint32_t *indices*) svfloat64_t **svld1_gather**_[u64]**index**[_f64](svbool_t *pg*, const float64_t **base*, svuint64_t *indices*)

6.2.1.7. LD1 (vector base, scalar index)

Instances
svint32_t **svld1_gather**[_u32base]_**index_s32**(svbool_t *pg*, svuint32_t *bases*, int64_t *index*) svint64_t **svld1_gather**[_u64base]_**index_s64**(svbool_t *pg*, svuint64_t *bases*, int64_t *index*) svuint32_t **svld1_gather**[_u32base]_**index_u32**(svbool_t *pg*, svuint32_t *bases*, int64_t *index*) svuint64_t **svld1_gather**[_u64base]_**index_u64**(svbool_t *pg*, svuint64_t *bases*, int64_t *index*) svfloat32_t **svld1_gather**[_u32base]_**index_f32**(svbool_t *pg*, svuint32_t *bases*, int64_t *index*) svfloat64_t **svld1_gather**[_u64base]_**index_f64**(svbool_t *pg*, svuint64_t *bases*, int64_t *index*)

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
svint16_t **svld1sb_s16**(svbool_t *pg*, const int8_t **base*) svint32_t **svld1sb_s32**(svbool_t *pg*, const int8_t **base*) svint64_t **svld1sb_s64**(svbool_t *pg*, const int8_t **base*) svuint16_t **svld1sb_u16**(svbool_t *pg*, const int8_t **base*)

Instances
svuint32_t **svld1sb_u32**(svbool_t *pg*, const int8_t **base*)
svuint64_t **svld1sb_u64**(svbool_t *pg*, const int8_t **base*)

6.2.2.2. LD1SB (scalar base, VL displacement)

Instances
svint16_t **svld1sb_vnum_s16**(svbool_t *pg*, const int8_t **base*, int64_t *vnum*)
svint32_t **svld1sb_vnum_s32**(svbool_t *pg*, const int8_t **base*, int64_t *vnum*)
svint64_t **svld1sb_vnum_s64**(svbool_t *pg*, const int8_t **base*, int64_t *vnum*)
svuint16_t **svld1sb_vnum_u16**(svbool_t *pg*, const int8_t **base*, int64_t *vnum*)
svuint32_t **svld1sb_vnum_u32**(svbool_t *pg*, const int8_t **base*, int64_t *vnum*)
svuint64_t **svld1sb_vnum_u64**(svbool_t *pg*, const int8_t **base*, int64_t *vnum*)

6.2.2.3. LD1SB (vector base)

Instances
svint32_t **svld1sb_gather**[_u32base]_**s32**(svbool_t *pg*, svuint32_t *bases*)
svint64_t **svld1sb_gather**[_u64base]_**s64**(svbool_t *pg*, svuint64_t *bases*)
svuint32_t **svld1sb_gather**[_u32base]_**u32**(svbool_t *pg*, svuint32_t *bases*)
svuint64_t **svld1sb_gather**[_u64base]_**u64**(svbool_t *pg*, svuint64_t *bases*)

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

Instances
svint32_t **svld1sb_gather**_[s32]**offset_s32**(svbool_t *pg*, const int8_t **base*, svint32_t *offsets*)
svint64_t **svld1sb_gather**_[s64]**offset_s64**(svbool_t *pg*, const int8_t **base*, svint64_t *offsets*)
svuint32_t **svld1sb_gather**_[s32]**offset_u32**(svbool_t *pg*, const int8_t **base*, svint32_t *offsets*)
svuint64_t **svld1sb_gather**_[s64]**offset_u64**(svbool_t *pg*, const int8_t **base*, svint64_t *offsets*)
svint32_t **svld1sb_gather**_[u32]**offset_s32**(svbool_t *pg*, const int8_t **base*, svuint32_t *offsets*)
svint64_t **svld1sb_gather**_[u64]**offset_s64**(svbool_t *pg*, const int8_t **base*, svuint64_t *offsets*)
svuint32_t **svld1sb_gather**_[u32]**offset_u32**(svbool_t *pg*, const int8_t **base*, svuint32_t *offsets*)
svuint64_t **svld1sb_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 **svld1sb_gather**[_u32base]_**offset_s32**(svbool_t *pg*, svuint32_t *bases*, int64_t *offset*)
svint64_t **svld1sb_gather**[_u64base]_**offset_s64**(svbool_t *pg*, svuint64_t *bases*, int64_t *offset*)
svuint32_t **svld1sb_gather**[_u32base]_**offset_u32**(svbool_t *pg*, svuint32_t *bases*, int64_t *offset*)
svuint64_t **svld1sb_gather**[_u64base]_**offset_u64**(svbool_t *pg*, svuint64_t *bases*,

Instances
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 **svld1ub_s16**(svbool_t *pg*, const uint8_t **base*)
svint32_t **svld1ub_s32**(svbool_t *pg*, const uint8_t **base*)
svint64_t **svld1ub_s64**(svbool_t *pg*, const uint8_t **base*)
svuint16_t **svld1ub_u16**(svbool_t *pg*, const uint8_t **base*)
svuint32_t **svld1ub_u32**(svbool_t *pg*, const uint8_t **base*)
svuint64_t **svld1ub_u64**(svbool_t *pg*, const uint8_t **base*)

6.2.3.2. LD1UB (scalar base, VL displacement)

Instances
svint16_t **svld1ub_vnum_s16**(svbool_t *pg*, const uint8_t **base*, int64_t *vnum*)
svint32_t **svld1ub_vnum_s32**(svbool_t *pg*, const uint8_t **base*, int64_t *vnum*)
svint64_t **svld1ub_vnum_s64**(svbool_t *pg*, const uint8_t **base*, int64_t *vnum*)
svuint16_t **svld1ub_vnum_u16**(svbool_t *pg*, const uint8_t **base*, int64_t *vnum*)
svuint32_t **svld1ub_vnum_u32**(svbool_t *pg*, const uint8_t **base*, int64_t *vnum*)
svuint64_t **svld1ub_vnum_u64**(svbool_t *pg*, const uint8_t **base*, int64_t *vnum*)

6.2.3.3. LD1UB (vector base)

Instances
svint32_t **svld1ub_gather**[_u32base]_**s32**(svbool_t *pg*, svuint32_t *bases*)
svint64_t **svld1ub_gather**[_u64base]_**s64**(svbool_t *pg*, svuint64_t *bases*)
svuint32_t **svld1ub_gather**[_u32base]_**u32**(svbool_t *pg*, svuint32_t *bases*)
svuint64_t **svld1ub_gather**[_u64base]_**u64**(svbool_t *pg*, svuint64_t *bases*)

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

Instances
svint32_t **svld1ub_gather**_[s32]**offset_s32**(svbool_t *pg*, const uint8_t **base*, svint32_t *offsets*)
svint64_t **svld1ub_gather**_[s64]**offset_s64**(svbool_t *pg*, const uint8_t **base*, svint64_t *offsets*)
svuint32_t **svld1ub_gather**_[s32]**offset_u32**(svbool_t *pg*, const uint8_t **base*, svint32_t *offsets*)
svuint64_t **svld1ub_gather**_[s64]**offset_u64**(svbool_t *pg*, const uint8_t **base*, svint64_t *offsets*)
svint32_t **svld1ub_gather**_[u32]**offset_s32**(svbool_t *pg*, const uint8_t **base*, svuint32_t *offsets*)
svint64_t **svld1ub_gather**_[u64]**offset_s64**(svbool_t *pg*, const uint8_t **base*, svuint64_t *offsets*)
svuint32_t **svld1ub_gather**_[u32]**offset_u32**(svbool_t *pg*, const uint8_t **base*, svuint32_t *offsets*)
svuint64_t **svld1ub_gather**_[u64]**offset_u64**(svbool_t *pg*, const uint8_t **base*,

Instances
svuint64_t *offsets*)

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

Instances
svint32_t **svld1ub_gather**[_u32base]_**offset_s32**(svbool_t *pg*, svuint32_t *bases*, int64_t *offset*) svint64_t **svld1ub_gather**[_u64base]_**offset_s64**(svbool_t *pg*, svuint64_t *bases*, int64_t *offset*) svuint32_t **svld1ub_gather**[_u32base]_**offset_u32**(svbool_t *pg*, svuint32_t *bases*, int64_t *offset*) svuint64_t **svld1ub_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
svuint32_t **svld1sh_gather**_[s32]**offset_u32**(svbool_t *pg*, const int16_t **base*, svint32_t *offsets*) svuint64_t **svld1sh_gather**_[s64]**offset_u64**(svbool_t *pg*, const int16_t **base*, svint64_t *offsets*) svint32_t **svld1sh_gather**_[u32]**offset_s32**(svbool_t *pg*, const int16_t **base*, svuint32_t *offsets*) svint64_t **svld1sh_gather**_[u64]**offset_s64**(svbool_t *pg*, const int16_t **base*, svuint64_t *offsets*) svuint32_t **svld1sh_gather**_[u32]**offset_u32**(svbool_t *pg*, const int16_t **base*, svuint32_t *offsets*) svuint64_t **svld1sh_gather**_[u64]**offset_u64**(svbool_t *pg*, const int16_t **base*, svuint64_t *offsets*)

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

Instances
svint32_t **svld1sh_gather**[_u32base]_**offset_s32**(svbool_t *pg*, svuint32_t *bases*, int64_t *offset*) svint64_t **svld1sh_gather**[_u64base]_**offset_s64**(svbool_t *pg*, svuint64_t *bases*, int64_t *offset*) svuint32_t **svld1sh_gather**[_u32base]_**offset_u32**(svbool_t *pg*, svuint32_t *bases*, int64_t *offset*) svuint64_t **svld1sh_gather**[_u64base]_**offset_u64**(svbool_t *pg*, svuint64_t *bases*, int64_t *offset*)

6.2.4.6. LD1SH (scalar base, vector index)

Instances
svint32_t **svld1sh_gather**_[s32]**index_s32**(svbool_t *pg*, const int16_t **base*, svint32_t *indices*) svint64_t **svld1sh_gather**_[s64]**index_s64**(svbool_t *pg*, const int16_t **base*, svint64_t *indices*) svuint32_t **svld1sh_gather**_[s32]**index_u32**(svbool_t *pg*, const int16_t **base*, svint32_t *indices*) svuint64_t **svld1sh_gather**_[s64]**index_u64**(svbool_t *pg*, const int16_t **base*, svint64_t *indices*) svint32_t **svld1sh_gather**_[u32]**index_s32**(svbool_t *pg*, const int16_t **base*, svuint32_t *indices*) svint64_t **svld1sh_gather**_[u64]**index_s64**(svbool_t *pg*, const int16_t **base*, svuint64_t *indices*) svuint32_t **svld1sh_gather**_[u32]**index_u32**(svbool_t *pg*, const int16_t **base*, svuint32_t *indices*) svuint64_t **svld1sh_gather**_[u64]**index_u64**(svbool_t *pg*, const int16_t **base*, svuint64_t *indices*)

6.2.4.7. LD1SH (vector base, scalar index)

Instances
svint32_t **svld1sh_gather**[_u32base]_**index_s32**(svbool_t *pg*, svuint32_t *bases*, int64_t *index*) svint64_t **svld1sh_gather**[_u64base]_**index_s64**(svbool_t *pg*, svuint64_t *bases*, int64_t *index*) svuint32_t **svld1sh_gather**[_u32base]_**index_u32**(svbool_t *pg*, svuint32_t *bases*,

Instances
int64_t *index*) svuint64_t **svld1sh_gather**[_u64base]**_index_u64**(svbool_t *pg*, svuint64_t *bases*, int64_t *index*)

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
svint32_t **svld1uh_s32**(svbool_t *pg*, const uint16_t **base*) svint64_t **svld1uh_s64**(svbool_t *pg*, const uint16_t **base*) svuint32_t **svld1uh_u32**(svbool_t *pg*, const uint16_t **base*) svuint64_t **svld1uh_u64**(svbool_t *pg*, const uint16_t **base*)

6.2.5.2. LD1UH (scalar base, VL displacement)

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

6.2.5.3. LD1UH (vector base)

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

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

Instances
svint32_t **svld1uh_gather**_[s32]**offset_s32**(svbool_t *pg*, const uint16_t **base*, svint32_t *offsets*) svint64_t **svld1uh_gather**_[s64]**offset_s64**(svbool_t *pg*, const uint16_t **base*, svint64_t *offsets*) svuint32_t **svld1uh_gather**_[s32]**offset_u32**(svbool_t *pg*, const uint16_t **base*, svint32_t *offsets*) svuint64_t **svld1uh_gather**_[s64]**offset_u64**(svbool_t *pg*, const uint16_t **base*, svint64_t *offsets*) svint32_t **svld1uh_gather**_[u32]**offset_s32**(svbool_t *pg*, const uint16_t **base*, svuint32_t *offsets*) svint64_t **svld1uh_gather**_[u64]**offset_s64**(svbool_t *pg*, const uint16_t **base*, svuint64_t *offsets*) svuint32_t **svld1uh_gather**_[u32]**offset_u32**(svbool_t *pg*, const uint16_t **base*, svuint32_t *offsets*) svuint64_t **svld1uh_gather**_[u64]**offset_u64**(svbool_t *pg*, const uint16_t **base*, svuint64_t *offsets*)

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

Instances
svint32_t **svld1uh_gather**[_u32base]_**offset_s32**(svbool_t *pg*, svuint32_t *bases*, int64_t *offset*)
svint64_t **svld1uh_gather**[_u64base]_**offset_s64**(svbool_t *pg*, svuint64_t *bases*, int64_t *offset*)
svuint32_t **svld1uh_gather**[_u32base]_**offset_u32**(svbool_t *pg*, svuint32_t *bases*, int64_t *offset*)
svuint64_t **svld1uh_gather**[_u64base]_**offset_u64**(svbool_t *pg*, svuint64_t *bases*, int64_t *offset*)

6.2.5.6. LD1UH (scalar base, vector index)

Instances
svint32_t **svld1uh_gather**_[s32]**index_s32**(svbool_t *pg*, const uint16_t **base*, svint32_t *indices*)
svint64_t **svld1uh_gather**_[s64]**index_s64**(svbool_t *pg*, const uint16_t **base*, svint64_t *indices*)
svuint32_t **svld1uh_gather**_[s32]**index_u32**(svbool_t *pg*, const uint16_t **base*, svint32_t *indices*)
svuint64_t **svld1uh_gather**_[s64]**index_u64**(svbool_t *pg*, const uint16_t **base*, svint64_t *indices*)
svint32_t **svld1uh_gather**_[u32]**index_s32**(svbool_t *pg*, const uint16_t **base*, svuint32_t *indices*)
svint64_t **svld1uh_gather**_[u64]**index_s64**(svbool_t *pg*, const uint16_t **base*, svuint64_t *indices*)
svuint32_t **svld1uh_gather**_[u32]**index_u32**(svbool_t *pg*, const uint16_t **base*, svuint32_t *indices*)
svuint64_t **svld1uh_gather**_[u64]**index_u64**(svbool_t *pg*, const uint16_t **base*, svuint64_t *indices*)

6.2.5.7. LD1UH (vector base, scalar index)

Instances
svint32_t **svld1uh_gather**[_u32base]_**index_s32**(svbool_t *pg*, svuint32_t *bases*, int64_t *index*)
svint64_t **svld1uh_gather**[_u64base]_**index_s64**(svbool_t *pg*, svuint64_t *bases*, int64_t *index*)
svuint32_t **svld1uh_gather**[_u32base]_**index_u32**(svbool_t *pg*, svuint32_t *bases*, int64_t *index*)
svuint64_t **svld1uh_gather**[_u64base]_**index_u64**(svbool_t *pg*, svuint64_t *bases*, int64_t *index*)

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
svint64_t **svld1sw_s64**(svbool_t *pg*, const int32_t **base*)

Instances
svuint64_t **svld1sw_u64**(svbool_t *pg*, const int32_t **base*)

6.2.6.2. LD1SW (scalar base, VL displacement)

Instances
svint64_t **svld1sw_vnum_s64**(svbool_t *pg*, const int32_t **base*, int64_t *vnum*) svuint64_t **svld1sw_vnum_u64**(svbool_t *pg*, const int32_t **base*, int64_t *vnum*)

6.2.6.3. LD1SW (vector base)

Instances
svint64_t **svld1sw_gather**[_u64base]_**s64**(svbool_t *pg*, svuint64_t *bases*) svuint64_t **svld1sw_gather**[_u64base]_**u64**(svbool_t *pg*, svuint64_t *bases*)

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

Instances
svint64_t **svld1sw_gather**_[s64]**offset_s64**(svbool_t *pg*, const int32_t **base*, svint64_t *offsets*) svuint64_t **svld1sw_gather**_[s64]**offset_u64**(svbool_t *pg*, const int32_t **base*, svint64_t *offsets*) svint64_t **svld1sw_gather**_[u64]**offset_s64**(svbool_t *pg*, const int32_t **base*, svuint64_t *offsets*) svuint64_t **svld1sw_gather**_[u64]**offset_u64**(svbool_t *pg*, const int32_t **base*, svuint64_t *offsets*)

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

Instances
svint64_t **svld1sw_gather**[_u64base]_**offset_s64**(svbool_t *pg*, svuint64_t *bases*, int64_t *offset*) svuint64_t **svld1sw_gather**[_u64base]_**offset_u64**(svbool_t *pg*, svuint64_t *bases*, int64_t *offset*)

6.2.6.6. LD1SW (scalar base, vector index)

Instances
svint64_t **svld1sw_gather**_[s64]**index_s64**(svbool_t *pg*, const int32_t **base*, svint64_t *indices*) svuint64_t **svld1sw_gather**_[s64]**index_u64**(svbool_t *pg*, const int32_t **base*, svint64_t *indices*) svint64_t **svld1sw_gather**_[u64]**index_s64**(svbool_t *pg*, const int32_t **base*, svuint64_t *indices*) svuint64_t **svld1sw_gather**_[u64]**index_u64**(svbool_t *pg*, const int32_t **base*, svuint64_t *indices*)

6.2.6.7. LD1SW (vector base, scalar index)

Instances
svint64_t **svld1sw_gather**[_u64base]_**index_s64**(svbool_t *pg*, svuint64_t *bases*, int64_t *index*)

Instances
svuint64_t **svld1sw_gather**[_u64base]_**index_u64**(svbool_t *pg*, svuint64_t *bases*, int64_t *index*)

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
svint64_t **svld1uw_s64**(svbool_t *pg*, const uint32_t **base*) svuint64_t **svld1uw_u64**(svbool_t *pg*, const uint32_t **base*)

6.2.7.2. LD1UW (scalar base, VL displacement)

Instances
svint64_t **svld1uw_vnum_s64**(svbool_t *pg*, const uint32_t **base*, int64_t *vnum*) svuint64_t **svld1uw_vnum_u64**(svbool_t *pg*, const uint32_t **base*, int64_t *vnum*)

6.2.7.3. LD1UW (vector base)

Instances
svint64_t **svld1uw_gather**[_u64base]_**s64**(svbool_t *pg*, svuint64_t *bases*) svuint64_t **svld1uw_gather**[_u64base]_**u64**(svbool_t *pg*, svuint64_t *bases*)

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

Instances
svint64_t **svld1uw_gather**_[s64]**offset_s64**(svbool_t *pg*, const uint32_t **base*, svint64_t *offsets*) svuint64_t **svld1uw_gather**_[s64]**offset_u64**(svbool_t *pg*, const uint32_t **base*, svint64_t *offsets*) svint64_t **svld1uw_gather**_[u64]**offset_s64**(svbool_t *pg*, const uint32_t **base*, svuint64_t *offsets*) svuint64_t **svld1uw_gather**_[u64]**offset_u64**(svbool_t *pg*, const uint32_t **base*, svuint64_t *offsets*)

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

Instances
svint64_t **svld1uw_gather**[_u64base]_**offset_s64**(svbool_t *pg*, svuint64_t *bases*, int64_t *offset*) svuint64_t **svld1uw_gather**[_u64base]_**offset_u64**(svbool_t *pg*, svuint64_t *bases*, int64_t *offset*)

6.2.7.6. LD1UW (scalar base, vector index)

Instances
svint64_t **svld1uw_gather**_[s64]**index_s64**(svbool_t *pg*, const uint32_t **base*, svint64_t *indices*)

Instances

`svuint64_t svld1uw_gather_[s64]index_u64(svbool_t pg, const uint32_t *base, svint64_t indices)`
`svint64_t svld1uw_gather_[u64]index_s64(svbool_t pg, const uint32_t *base, svuint64_t indices)`
`svuint64_t svld1uw_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 svld1uw_gather[_u64base]_index_s64(svbool_t pg, svuint64_t bases, int64_t index)` `svuint64_t svld1uw_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 svld1rq[_s8](svbool_t pg, const int8_t *base)`
`svint16_t svld1rq[_s16](svbool_t pg, const int16_t *base)`
`svint32_t svld1rq[_s32](svbool_t pg, const int32_t *base)`
`svint64_t svld1rq[_s64](svbool_t pg, const int64_t *base)`
`svuint8_t svld1rq[_u8](svbool_t pg, const uint8_t *base)`
`svuint16_t svld1rq[_u16](svbool_t pg, const uint16_t *base)`
`svuint32_t svld1rq[_u32](svbool_t pg, const uint32_t *base)`
`svuint64_t svld1rq[_u64](svbool_t pg, const uint64_t *base)`
`svfloat16_t svld1rq[_f16](svbool_t pg, const float16_t *base)`
`svfloat32_t svld1rq[_f32](svbool_t pg, const float32_t *base)`
`svfloat64_t svld1rq[_f64](svbool_t pg, const float64_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)`

Instances
svuint64_t **svldff1**[_u64](svbool_t *pg*, const uint64_t **base*)
svfloat16_t **svldff1**[_f16](svbool_t *pg*, const float16_t **base*)
svfloat32_t **svldff1**[_f32](svbool_t *pg*, const float32_t **base*)
svfloat64_t **svldff1**[_f64](svbool_t *pg*, const float64_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*)

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*, svint32_t *offsets*)

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*, svint32_t *indices*)

Instances
svfloat64_t **svldff1_gather**_[s64]**index**[_f64](svbool_t *pg*, const float64_t **base*, svint64_t *indices*)
svint32_t **svldff1_gather**_[u32]**index**[_s32](svbool_t *pg*, const int32_t **base*, svuint32_t *indices*)
svint64_t **svldff1_gather**_[u64]**index**[_s64](svbool_t *pg*, const int64_t **base*, svuint64_t *indices*)
svuint32_t **svldff1_gather**_[u32]**index**[_u32](svbool_t *pg*, const uint32_t **base*, svuint32_t *indices*)
svuint64_t **svldff1_gather**_[u64]**index**[_u64](svbool_t *pg*, const uint64_t **base*, svuint64_t *indices*)
svfloat32_t **svldff1_gather**_[u32]**index**[_f32](svbool_t *pg*, const float32_t **base*, svuint32_t *indices*)
svfloat64_t **svldff1_gather**_[u64]**index**[_f64](svbool_t *pg*, const float64_t **base*, svuint64_t *indices*)

6.2.9.7. LDFF1 (vector base, scalar index)

Instances
svint32_t **svldff1_gather**[_u32base]_**index_s32**(svbool_t *pg*, svuint32_t *bases*, int64_t *index*)
svint64_t **svldff1_gather**[_u64base]_**index_s64**(svbool_t *pg*, svuint64_t *bases*, int64_t *index*)
svuint32_t **svldff1_gather**[_u32base]_**index_u32**(svbool_t *pg*, svuint32_t *bases*, int64_t *index*)
svuint64_t **svldff1_gather**[_u64base]_**index_u64**(svbool_t *pg*, svuint64_t *bases*, int64_t *index*)
svfloat32_t **svldff1_gather**[_u32base]_**index_f32**(svbool_t *pg*, svuint32_t *bases*, int64_t *index*)
svfloat64_t **svldff1_gather**[_u64base]_**index_f64**(svbool_t *pg*, svuint64_t *bases*, int64_t *index*)

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
svint16_t **svldff1sb_s16**(svbool_t *pg*, const int8_t **base*)
svint32_t **svldff1sb_s32**(svbool_t *pg*, const int8_t **base*)
svint64_t **svldff1sb_s64**(svbool_t *pg*, const int8_t **base*)
svuint16_t **svldff1sb_u16**(svbool_t *pg*, const int8_t **base*)
svuint32_t **svldff1sb_u32**(svbool_t *pg*, const int8_t **base*)
svuint64_t **svldff1sb_u64**(svbool_t *pg*, const int8_t **base*)

6.2.10.2. LDFF1SB (scalar base, VL displacement)

Instances

```
svint16_t svldff1sb_vnum_s16(svbool_t pg, const int8_t *base, int64_t vnum)
svint32_t svldff1sb_vnum_s32(svbool_t pg, const int8_t *base, int64_t vnum)
svint64_t svldff1sb_vnum_s64(svbool_t pg, const int8_t *base, int64_t vnum)
svuint16_t svldff1sb_vnum_u16(svbool_t pg, const int8_t *base,
                              int64_t vnum)
svuint32_t svldff1sb_vnum_u32(svbool_t pg, const int8_t *base,
                              int64_t vnum)
svuint64_t svldff1sb_vnum_u64(svbool_t pg, const int8_t *base,
                              int64_t vnum)
```

6.2.10.3. LDFF1SB (vector base)

Instances

```
svint32_t svldff1sb_gather[_u32base]_s32(svbool_t pg, svuint32_t bases)
svint64_t svldff1sb_gather[_u64base]_s64(svbool_t pg, svuint64_t bases)
svuint32_t svldff1sb_gather[_u32base]_u32(svbool_t pg, svuint32_t bases)
svuint64_t svldff1sb_gather[_u64base]_u64(svbool_t pg, svuint64_t bases)
```

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

Instances

```
svint32_t svldff1sb_gather_[s32]offset_s32(svbool_t pg, const int8_t *base,
                                           svint32_t offsets)
svint64_t svldff1sb_gather_[s64]offset_s64(svbool_t pg, const int8_t *base,
                                           svint64_t offsets)
svuint32_t svldff1sb_gather_[s32]offset_u32(svbool_t pg, const int8_t *base,
                                            svint32_t offsets)
svuint64_t svldff1sb_gather_[s64]offset_u64(svbool_t pg, const int8_t *base,
                                            svint64_t offsets)
svint32_t svldff1sb_gather_[u32]offset_s32(svbool_t pg, const int8_t *base,
                                           svuint32_t offsets)
svint64_t svldff1sb_gather_[u64]offset_s64(svbool_t pg, const int8_t *base,
                                           svuint64_t offsets)
svuint32_t svldff1sb_gather_[u32]offset_u32(svbool_t pg, const int8_t *base,
                                            svuint32_t offsets)
svuint64_t svldff1sb_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 svldff1sb_gather[_u32base]_offset_s32(svbool_t pg,
                                                svuint32_t bases,
                                                int64_t offset)
svint64_t svldff1sb_gather[_u64base]_offset_s64(svbool_t pg,
                                                svuint64_t bases,
                                                int64_t offset)
svuint32_t svldff1sb_gather[_u32base]_offset_u32(svbool_t pg,
                                                 svuint32_t bases,
                                                 int64_t offset)
svuint64_t svldff1sb_gather[_u64base]_offset_u64(svbool_t pg,
                                                 svuint64_t bases,
                                                 int64_t offset)
```


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 `svrdffr` to determine which elements of the result have defined values.

6.2.11.1. LDFF1UB (scalar base)

Instances

```
svint16_t svldff1ub_s16(svbool_t pg, const uint8_t *base)
svint32_t svldff1ub_s32(svbool_t pg, const uint8_t *base)
svint64_t svldff1ub_s64(svbool_t pg, const uint8_t *base)
svuint16_t svldff1ub_u16(svbool_t pg, const uint8_t *base)
svuint32_t svldff1ub_u32(svbool_t pg, const uint8_t *base)
svuint64_t svldff1ub_u64(svbool_t pg, const uint8_t *base)
```

6.2.11.2. LDFF1UB (scalar base, VL displacement)

Instances

```
svint16_t svldff1ub_vnum_s16(svbool_t pg, const uint8_t *base,
                             int64_t vnum)
svint32_t svldff1ub_vnum_s32(svbool_t pg, const uint8_t *base,
                             int64_t vnum)
svint64_t svldff1ub_vnum_s64(svbool_t pg, const uint8_t *base,
                             int64_t vnum)
svuint16_t svldff1ub_vnum_u16(svbool_t pg, const uint8_t *base,
                              int64_t vnum)
svuint32_t svldff1ub_vnum_u32(svbool_t pg, const uint8_t *base,
                              int64_t vnum)
svuint64_t svldff1ub_vnum_u64(svbool_t pg, const uint8_t *base,
                              int64_t vnum)
```

6.2.11.3. LDFF1UB (vector base)

Instances

```
svint32_t svldff1ub_gather[_u32base]_s32(svbool_t pg, svuint32_t bases)
svint64_t svldff1ub_gather[_u64base]_s64(svbool_t pg, svuint64_t bases)
svuint32_t svldff1ub_gather[_u32base]_u32(svbool_t pg, svuint32_t bases)
svuint64_t svldff1ub_gather[_u64base]_u64(svbool_t pg, svuint64_t bases)
```

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

Instances

```
svint32_t svldff1ub_gather_[s32]offset_s32(svbool_t pg, const uint8_t *base,
                                           svint32_t offsets)
svint64_t svldff1ub_gather_[s64]offset_s64(svbool_t pg, const uint8_t *base,
                                           svint64_t offsets)
svuint32_t svldff1ub_gather_[s32]offset_u32(svbool_t pg,
                                            const uint8_t *base,
                                            svint32_t offsets)
svuint64_t svldff1ub_gather_[s64]offset_u64(svbool_t pg,
                                            const uint8_t *base,
                                            svint64_t offsets)
```


Instances

`svint32_t svldff1ub_gather_[u32]offset_s32(svbool_t pg, const uint8_t *base, svuint32_t offsets)`
`svint64_t svldff1ub_gather_[u64]offset_s64(svbool_t pg, const uint8_t *base, svuint64_t offsets)`
`svuint32_t svldff1ub_gather_[u32]offset_u32(svbool_t pg, const uint8_t *base, svuint32_t offsets)`
`svuint64_t svldff1ub_gather_[u64]offset_u64(svbool_t pg, const uint8_t *base, svuint64_t offsets)`

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

Instances

`svint32_t svldff1ub_gather[_u32base]_offset_s32(svbool_t pg, svuint32_t bases, int64_t offset)`
`svint64_t svldff1ub_gather[_u64base]_offset_s64(svbool_t pg, svuint64_t bases, int64_t offset)`
`svuint32_t svldff1ub_gather[_u32base]_offset_u32(svbool_t pg, svuint32_t bases, int64_t offset)`
`svuint64_t svldff1ub_gather[_u64base]_offset_u64(svbool_t pg, svuint64_t bases, int64_t offset)`

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

`svint32_t svldff1sh_s32(svbool_t pg, const int16_t *base)`
`svint64_t svldff1sh_s64(svbool_t pg, const int16_t *base)`
`svuint32_t svldff1sh_u32(svbool_t pg, const int16_t *base)`
`svuint64_t svldff1sh_u64(svbool_t pg, const int16_t *base)`

6.2.12.2. LDFF1SH (scalar base, VL displacement)

Instances

`svint32_t svldff1sh_vnum_s32(svbool_t pg, const int16_t *base, 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)
```

6.2.12.6. LDFF1SH (scalar base, vector index)

Instances

```
svint32_t svldff1sh_gather_[s32]index_s32(svbool_t pg, const int16_t *base,
                                          svint32_t indices)
svint64_t svldff1sh_gather_[s64]index_s64(svbool_t pg, const int16_t *base,
                                          svint64_t indices)
svuint32_t svldff1sh_gather_[s32]index_u32(svbool_t pg, const int16_t *base,
```


Instances
svint32_t *indices*) svuint64_t **svldff1sh_gather**_[s64]**index_u64**(svbool_t *pg*, const int16_t **base*, svint64_t *indices*) svint32_t **svldff1sh_gather**_[u32]**index_s32**(svbool_t *pg*, const int16_t **base*, svuint32_t *indices*) svint64_t **svldff1sh_gather**_[u64]**index_s64**(svbool_t *pg*, const int16_t **base*, svuint64_t *indices*) svuint32_t **svldff1sh_gather**_[u32]**index_u32**(svbool_t *pg*, const int16_t **base*, svuint32_t *indices*) svuint64_t **svldff1sh_gather**_[u64]**index_u64**(svbool_t *pg*, const int16_t **base*, svuint64_t *indices*)

6.2.12.7. LDFF1SH (vector base, scalar index)

Instances
svint32_t **svldff1sh_gather**[_u32base]_**index_s32**(svbool_t *pg*, svuint32_t *bases*, int64_t *index*) svint64_t **svldff1sh_gather**[_u64base]_**index_s64**(svbool_t *pg*, svuint64_t *bases*, int64_t *index*) svuint32_t **svldff1sh_gather**[_u32base]_**index_u32**(svbool_t *pg*, svuint32_t *bases*, int64_t *index*) svuint64_t **svldff1sh_gather**[_u64base]_**index_u64**(svbool_t *pg*, svuint64_t *bases*, int64_t *index*)

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
svint32_t **svldff1uh_s32**(svbool_t *pg*, const uint16_t **base*) svint64_t **svldff1uh_s64**(svbool_t *pg*, const uint16_t **base*) svuint32_t **svldff1uh_u32**(svbool_t *pg*, const uint16_t **base*) svuint64_t **svldff1uh_u64**(svbool_t *pg*, const uint16_t **base*)

6.2.13.2. LDFF1UH (scalar base, VL displacement)

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

6.2.13.3. LDFF1UH (vector base)

Instances

```
svint32_t svldff1uh_gather[_u32base]_s32(svbool_t pg, svuint32_t bases)
svint64_t svldff1uh_gather[_u64base]_s64(svbool_t pg, svuint64_t bases)
svuint32_t svldff1uh_gather[_u32base]_u32(svbool_t pg, svuint32_t bases)
svuint64_t svldff1uh_gather[_u64base]_u64(svbool_t pg, svuint64_t bases)
```

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

Instances

```
svint32_t svldff1uh_gather_[s32]offset_s32(svbool_t pg,
                                           const uint16_t *base,
                                           svint32_t offsets)
svint64_t svldff1uh_gather_[s64]offset_s64(svbool_t pg,
                                           const uint16_t *base,
                                           svint64_t offsets)
svuint32_t svldff1uh_gather_[s32]offset_u32(svbool_t pg,
                                            const uint16_t *base,
                                            svint32_t offsets)
svuint64_t svldff1uh_gather_[s64]offset_u64(svbool_t pg,
                                            const uint16_t *base,
                                            svint64_t offsets)
svint32_t svldff1uh_gather_[u32]offset_s32(svbool_t pg,
                                           const uint16_t *base,
                                           svuint32_t offsets)
svint64_t svldff1uh_gather_[u64]offset_s64(svbool_t pg,
                                           const uint16_t *base,
                                           svuint64_t offsets)
svuint32_t svldff1uh_gather_[u32]offset_u32(svbool_t pg,
                                            const uint16_t *base,
                                            svuint32_t offsets)
svuint64_t svldff1uh_gather_[u64]offset_u64(svbool_t pg,
                                            const uint16_t *base,
                                            svuint64_t offsets)
```

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

Instances

```
svint32_t svldff1uh_gather[_u32base]_offset_s32(svbool_t pg,
                                                svuint32_t bases,
                                                int64_t offset)
svint64_t svldff1uh_gather[_u64base]_offset_s64(svbool_t pg,
                                                svuint64_t bases,
                                                int64_t offset)
svuint32_t svldff1uh_gather[_u32base]_offset_u32(svbool_t pg,
                                                 svuint32_t bases,
                                                 int64_t offset)
svuint64_t svldff1uh_gather[_u64base]_offset_u64(svbool_t pg,
                                                 svuint64_t bases,
                                                 int64_t offset)
```

6.2.13.6. LDFF1UH (scalar base, vector index)

Instances

```
svint32_t svldff1uh_gather_[s32]index_s32(svbool_t pg, const uint16_t *base,
```


Instances

```
                                                svint32_t indices)
svint64_t svldff1uh_gather_[s64]index_s64(svbool_t pg, const uint16_t *base,
                                                svint64_t indices)
svuint32_t svldff1uh_gather_[s32]index_u32(svbool_t pg,
                                                 const uint16_t *base,
                                                 svint32_t indices)
svuint64_t svldff1uh_gather_[s64]index_u64(svbool_t pg,
                                                 const uint16_t *base,
                                                 svint64_t indices)
svint32_t svldff1uh_gather_[u32]index_s32(svbool_t pg, const uint16_t *base,
                                                svuint32_t indices)
svint64_t svldff1uh_gather_[u64]index_s64(svbool_t pg, const uint16_t *base,
                                                svuint64_t indices)
svuint32_t svldff1uh_gather_[u32]index_u32(svbool_t pg,
                                                 const uint16_t *base,
                                                 svuint32_t indices)
svuint64_t svldff1uh_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 svldff1uh_gather[_u32base]_index_s32(svbool_t pg,
                                                     svuint32_t bases,
                                                     int64_t index)
svint64_t svldff1uh_gather[_u64base]_index_s64(svbool_t pg,
                                                     svuint64_t bases,
                                                     int64_t index)
svuint32_t svldff1uh_gather[_u32base]_index_u32(svbool_t pg,
                                                      svuint32_t bases,
                                                      int64_t index)
svuint64_t svldff1uh_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

```
svint64_t svldff1sw_s64(svbool_t pg, const int32_t *base)
svuint64_t svldff1sw_u64(svbool_t pg, const int32_t *base)
```

6.2.14.2. LDFF1SW (scalar base, VL displacement)

Instances

```
svint64_t svldff1sw_vnum_s64(svbool_t pg, const int32_t *base,
                             int64_t vnum)
```


Instances
svuint64_t **svldff1sw_vnum_u64**(svbool_t *pg*, const int32_t **base*, int64_t *vnum*)

6.2.14.3. LDFF1SW (vector base)

Instances
svint64_t **svldff1sw_gather**[_u64base]**_s64**(svbool_t *pg*, svuint64_t *bases*) svuint64_t **svldff1sw_gather**[_u64base]**_u64**(svbool_t *pg*, svuint64_t *bases*)

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

Instances
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*)

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

Instances
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*)

6.2.14.6. LDFF1SW (scalar base, vector index)

Instances
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*, svuint64_t *indices*)

6.2.14.7. LDFF1SW (vector base, scalar index)

Instances
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*)

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 `svrdffr` to determine which elements of the result have defined values.

6.2.15.1. LDFF1UW (scalar base)

Instances
`svint64_t` **`svldff1uw_s64`**`(svbool_t` *`pg`*`, const uint32_t *`*`base`*`)` `svuint64_t` **`svldff1uw_u64`**`(svbool_t` *`pg`*`, const uint32_t *`*`base`*`)`

6.2.15.2. LDFF1UW (scalar base, VL displacement)

Instances

`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`*`)`

6.2.15.3. LDFF1UW (vector base)

Instances
`svint64_t` **`svldff1uw_gather`**`[_u64base]_`**`s64`**`(svbool_t` *`pg`*`, svuint64_t` *`bases`*`)` `svuint64_t` **`svldff1uw_gather`**`[_u64base]_`**`u64`**`(svbool_t` *`pg`*`, svuint64_t` *`bases`*`)`

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

Instances

`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`*`,`
`const uint32_t *`*`base`*`,`
`svuint64_t` *`offsets`*`)`

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

Instances

`svint64_t` **`svldff1uw_gather`**`[_u64base]_`**`offset_s64`**`(svbool_t` *`pg`*`,`
`svuint64_t` *`bases`*`,`
`int64_t` *`offset`*`)`
`svuint64_t` **`svldff1uw_gather`**`[_u64base]_`**`offset_u64`**`(svbool_t` *`pg`*`,`
`svuint64_t` *`bases`*`,`
`int64_t` *`offset`*`)`

6.2.15.6. LDFF1UW (scalar base, vector index)

Instances
`svint64_t svldff1uw_gather_[s64]index_s64(svbool_t pg, const uint32_t *base, svint64_t indices)`
`svuint64_t svldff1uw_gather_[s64]index_u64(svbool_t pg, const uint32_t *base, svint64_t indices)`
`svint64_t svldff1uw_gather_[u64]index_s64(svbool_t pg, const uint32_t *base, svuint64_t indices)`
`svuint64_t svldff1uw_gather_[u64]index_u64(svbool_t pg, const uint32_t *base, svuint64_t indices)`

6.2.15.7. LDFF1UW (vector base, scalar index)

Instances
`svint64_t svldff1uw_gather[_u64base]_index_s64(svbool_t pg, svuint64_t bases, int64_t index)`
`svuint64_t svldff1uw_gather[_u64base]_index_u64(svbool_t pg, svuint64_t bases, int64_t index)`

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[5]. 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
`svint8_t svldnf1[_s8](svbool_t pg, const int8_t *base)`
`svint16_t svldnf1[_s16](svbool_t pg, const int16_t *base)`
`svint32_t svldnf1[_s32](svbool_t pg, const int32_t *base)`
`svint64_t svldnf1[_s64](svbool_t pg, const int64_t *base)`
`svuint8_t svldnf1[_u8](svbool_t pg, const uint8_t *base)`
`svuint16_t svldnf1[_u16](svbool_t pg, const uint16_t *base)`
`svuint32_t svldnf1[_u32](svbool_t pg, const uint32_t *base)`
`svuint64_t svldnf1[_u64](svbool_t pg, const uint64_t *base)`
`svfloat16_t svldnf1[_f16](svbool_t pg, const float16_t *base)`
`svfloat32_t svldnf1[_f32](svbool_t pg, const float32_t *base)`
`svfloat64_t svldnf1[_f64](svbool_t pg, const float64_t *base)`

6.2.16.2. LDNF1 (scalar base, VL displacement)

Instances
`svint8_t svldnf1_vnum[_s8](svbool_t pg, const int8_t *base, int64_t vnum)`

[5] 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 **svldnf1_vnum**[_s16](svbool_t *pg*, const int16_t **base*, int64_t *vnum*) svint32_t **svldnf1_vnum**[_s32](svbool_t *pg*, const int32_t **base*, int64_t *vnum*) svint64_t **svldnf1_vnum**[_s64](svbool_t *pg*, const int64_t **base*, int64_t *vnum*) svuint8_t **svldnf1_vnum**[_u8](svbool_t *pg*, const uint8_t **base*, int64_t *vnum*) svuint16_t **svldnf1_vnum**[_u16](svbool_t *pg*, const uint16_t **base*, int64_t *vnum*) svuint32_t **svldnf1_vnum**[_u32](svbool_t *pg*, const uint32_t **base*, int64_t *vnum*) svuint64_t **svldnf1_vnum**[_u64](svbool_t *pg*, const uint64_t **base*, int64_t *vnum*) svfloat16_t **svldnf1_vnum**[_f16](svbool_t *pg*, const float16_t **base*, int64_t *vnum*) svfloat32_t **svldnf1_vnum**[_f32](svbool_t *pg*, const float32_t **base*, int64_t *vnum*) svfloat64_t **svldnf1_vnum**[_f64](svbool_t *pg*, const float64_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 `svrdffr` to determine which elements of the result have defined values.

6.2.17.1. LDNF1SB (scalar base)

Instances
svint16_t **svldnf1sb_s16**(svbool_t *pg*, const int8_t **base*) svint32_t **svldnf1sb_s32**(svbool_t *pg*, const int8_t **base*) svint64_t **svldnf1sb_s64**(svbool_t *pg*, const int8_t **base*) svuint16_t **svldnf1sb_u16**(svbool_t *pg*, const int8_t **base*) svuint32_t **svldnf1sb_u32**(svbool_t *pg*, const int8_t **base*) svuint64_t **svldnf1sb_u64**(svbool_t *pg*, const int8_t **base*)

6.2.17.2. LDNF1SB (scalar base, VL displacement)

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

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 `svrdffr` to determine which elements of the result have defined values.

6.2.18.1. LDNF1UB (scalar base)

Instances

`svint16_t svldnf1ub_s16(svbool_t pg, const uint8_t *base)`
`svint32_t svldnf1ub_s32(svbool_t pg, const uint8_t *base)`
`svint64_t svldnf1ub_s64(svbool_t pg, const uint8_t *base)`
`svuint16_t svldnf1ub_u16(svbool_t pg, const uint8_t *base)`
`svuint32_t svldnf1ub_u32(svbool_t pg, const uint8_t *base)`
`svuint64_t svldnf1ub_u64(svbool_t pg, const uint8_t *base)`

6.2.18.2. LDNF1UB (scalar base, VL displacement)

Instances

`svint16_t svldnf1ub_vnum_s16(svbool_t pg, const uint8_t *base,`
`  int64_t vnum)`
`svint32_t svldnf1ub_vnum_s32(svbool_t pg, const uint8_t *base,`
`  int64_t vnum)`
`svint64_t svldnf1ub_vnum_s64(svbool_t pg, const uint8_t *base,`
`  int64_t vnum)`
`svuint16_t svldnf1ub_vnum_u16(svbool_t pg, const uint8_t *base,`
`  int64_t vnum)`
`svuint32_t svldnf1ub_vnum_u32(svbool_t pg, const uint8_t *base,`
`  int64_t vnum)`
`svuint64_t svldnf1ub_vnum_u64(svbool_t pg, const uint8_t *base,`
`  int64_t vnum)`

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 `svrdffr` 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)`

Instances
`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 `svrdffr` 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 `svrdffr` to determine which elements of the result have defined values.

6.2.21.1. LDNF1SW (scalar base)

Instances
`svint64_t svldnf1sw_s64(svbool_t pg, const int32_t *base)` `svuint64_t svldnf1sw_u64(svbool_t pg, const int32_t *base)`

6.2.21.2. LDNF1SW (scalar base, VL displacement)

Instances
`svint64_t svldnf1sw_vnum_s64(svbool_t pg, const int32_t *base, int64_t vnum)` `svuint64_t svldnf1sw_vnum_u64(svbool_t pg, const int32_t *base, int64_t vnum)`

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 `svrdffr` to determine which elements of the result have defined values.

6.2.22.1. LDNF1UW (scalar base)

Instances
`svint64_t svldnf1uw_s64(svbool_t pg, const uint32_t *base)` `svuint64_t svldnf1uw_u64(svbool_t pg, const uint32_t *base)`

6.2.22.2. LDNF1UW (scalar base, VL displacement)

Instances

`svint64_t svldnf1uw_vnum_s64(svbool_t pg, const uint32_t *base,`
`  int64_t vnum)`
`svuint64_t svldnf1uw_vnum_u64(svbool_t pg, const uint32_t *base,`
`  int64_t vnum)`

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

`svint8_t svldnt1[_s8](svbool_t pg, const int8_t *base)`
`svint16_t svldnt1[_s16](svbool_t pg, const int16_t *base)`
`svint32_t svldnt1[_s32](svbool_t pg, const int32_t *base)`
`svint64_t svldnt1[_s64](svbool_t pg, const int64_t *base)`
`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)`

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,`

Instances

```
                                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)
```

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)
```

6.2.24.2. LD2 (scalar base, VL displacement)

Instances

```
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)
```


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

`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)`

6.2.25.2. LD3 (scalar base, VL displacement)

Instances

`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)`
`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)`

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)`

Instances

```
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)
```

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)
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)
```

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)
```


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*) void **svst1_vnum**[_f32](svbool_t *pg*, float32_t **base*, int64_t *vnum*, svfloat32_t *data*) void **svst1_vnum**[_f64](svbool_t *pg*, float64_t **base*, int64_t *vnum*, svfloat64_t *data*)

6.3.1.3. ST1 (vector base)

Instances
void **svst1_scatter**[_u32base_s32](svbool_t *pg*, svuint32_t *bases*, svint32_t *data*) void **svst1_scatter**[_u64base_s64](svbool_t *pg*, svuint64_t *bases*, svint64_t *data*) void **svst1_scatter**[_u32base_u32](svbool_t *pg*, svuint32_t *bases*, svuint32_t *data*) void **svst1_scatter**[_u64base_u64](svbool_t *pg*, svuint64_t *bases*, svuint64_t *data*) void **svst1_scatter**[_u32base_f32](svbool_t *pg*, svuint32_t *bases*, svfloat32_t *data*) void **svst1_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 **svst1_scatter**_[s32]**offset**[_s32](svbool_t *pg*, int32_t **base*, svint32_t *offsets*, svint32_t *data*) void **svst1_scatter**_[s64]**offset**[_s64](svbool_t *pg*, int64_t **base*, svint64_t *offsets*, svint64_t *data*) void **svst1_scatter**_[s32]**offset**[_u32](svbool_t *pg*, uint32_t **base*, svint32_t *offsets*, svuint32_t *data*) void **svst1_scatter**_[s64]**offset**[_u64](svbool_t *pg*, uint64_t **base*, svint64_t *offsets*, svuint64_t *data*) void **svst1_scatter**_[s32]**offset**[_f32](svbool_t *pg*, float32_t **base*, svint32_t *offsets*, svfloat32_t *data*) void **svst1_scatter**_[s64]**offset**[_f64](svbool_t *pg*, float64_t **base*, svint64_t *offsets*, svfloat64_t *data*)

Instances

```
void svst1_scatter_[u32]offset[_s32](svbool_t pg, int32_t *base,
                                     svuint32_t offsets, svint32_t data)
void svst1_scatter_[u64]offset[_s64](svbool_t pg, int64_t *base,
                                     svuint64_t offsets, svint64_t data)
void svst1_scatter_[u32]offset[_u32](svbool_t pg, uint32_t *base,
                                     svuint32_t offsets, svuint32_t data)
void svst1_scatter_[u64]offset[_u64](svbool_t pg, uint64_t *base,
                                     svuint64_t offsets, svuint64_t data)
void svst1_scatter_[u32]offset[_f32](svbool_t pg, float32_t *base,
                                     svuint32_t offsets, svfloat32_t data)
void svst1_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 svst1_scatter[_u32base]_offset[_s32](svbool_t pg, svuint32_t bases,
                                          int64_t offset, svint32_t data)
void svst1_scatter[_u64base]_offset[_s64](svbool_t pg, svuint64_t bases,
                                          int64_t offset, svint64_t data)
void svst1_scatter[_u32base]_offset[_u32](svbool_t pg, svuint32_t bases,
                                          int64_t offset, svuint32_t data)
void svst1_scatter[_u64base]_offset[_u64](svbool_t pg, svuint64_t bases,
                                          int64_t offset, svuint64_t data)
void svst1_scatter[_u32base]_offset[_f32](svbool_t pg, svuint32_t bases,
                                          int64_t offset, svfloat32_t data)
void svst1_scatter[_u64base]_offset[_f64](svbool_t pg, svuint64_t bases,
                                          int64_t offset, svfloat64_t data)
```

6.3.1.6. ST1 (scalar base, vector index)

Instances

```
void svst1_scatter_[s32]index[_s32](svbool_t pg, int32_t *base,
                                    svint32_t indices, svint32_t data)
void svst1_scatter_[s64]index[_s64](svbool_t pg, int64_t *base,
                                    svint64_t indices, svint64_t data)
void svst1_scatter_[s32]index[_u32](svbool_t pg, uint32_t *base,
                                    svint32_t indices, svuint32_t data)
void svst1_scatter_[s64]index[_u64](svbool_t pg, uint64_t *base,
                                    svint64_t indices, svuint64_t data)
void svst1_scatter_[s32]index[_f32](svbool_t pg, float32_t *base,
                                    svint32_t indices, svfloat32_t data)
void svst1_scatter_[s64]index[_f64](svbool_t pg, float64_t *base,
                                    svint64_t indices, svfloat64_t data)
void svst1_scatter_[u32]index[_s32](svbool_t pg, int32_t *base,
                                    svuint32_t indices, svint32_t data)
void svst1_scatter_[u64]index[_s64](svbool_t pg, int64_t *base,
                                    svuint64_t indices, svint64_t data)
void svst1_scatter_[u32]index[_u32](svbool_t pg, uint32_t *base,
                                    svuint32_t indices, svuint32_t data)
void svst1_scatter_[u64]index[_u64](svbool_t pg, uint64_t *base,
                                    svuint64_t indices, svuint64_t data)
void svst1_scatter_[u32]index[_f32](svbool_t pg, float32_t *base,
                                    svuint32_t indices, svfloat32_t data)
void svst1_scatter_[u64]index[_f64](svbool_t pg, float64_t *base,
                                    svuint64_t indices, svfloat64_t data)
```


6.3.1.7. ST1 (vector base, scalar index)

Instances
void **svst1_scatter**[_u32base]_**index**[_s32](svbool_t *pg*, svuint32_t *bases*, int64_t *index*, svint32_t *data*) void **svst1_scatter**[_u64base]_**index**[_s64](svbool_t *pg*, svuint64_t *bases*, int64_t *index*, svint64_t *data*) void **svst1_scatter**[_u32base]_**index**[_u32](svbool_t *pg*, svuint32_t *bases*, int64_t *index*, svuint32_t *data*) void **svst1_scatter**[_u64base]_**index**[_u64](svbool_t *pg*, svuint64_t *bases*, int64_t *index*, svuint64_t *data*) void **svst1_scatter**[_u32base]_**index**[_f32](svbool_t *pg*, svuint32_t *bases*, int64_t *index*, svfloat32_t *data*) void **svst1_scatter**[_u64base]_**index**[_f64](svbool_t *pg*, svuint64_t *bases*, int64_t *index*, svfloat64_t *data*)

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*)

Instances
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*) 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*,

Instances
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 **svst1h_scatter**_[s32]**offset**[_s32](svbool_t *pg*, int16_t **base*, svint32_t *offsets*, svint32_t *data*) void **svst1h_scatter**_[s64]**offset**[_s64](svbool_t *pg*, int16_t **base*, svint64_t *offsets*, svint64_t *data*) void **svst1h_scatter**_[s32]**offset**[_u32](svbool_t *pg*, uint16_t **base*, svint32_t *offsets*, svuint32_t *data*) void **svst1h_scatter**_[s64]**offset**[_u64](svbool_t *pg*, uint16_t **base*, svint64_t *offsets*, svuint64_t *data*) void **svst1h_scatter**_[u32]**offset**[_s32](svbool_t *pg*, int16_t **base*, svuint32_t *offsets*, svint32_t *data*) void **svst1h_scatter**_[u64]**offset**[_s64](svbool_t *pg*, int16_t **base*, svuint64_t *offsets*, svint64_t *data*) void **svst1h_scatter**_[u32]**offset**[_u32](svbool_t *pg*, uint16_t **base*, svuint32_t *offsets*, svuint32_t *data*) void **svst1h_scatter**_[u64]**offset**[_u64](svbool_t *pg*, uint16_t **base*, svuint64_t *offsets*, svuint64_t *data*)

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

Instances
void **svst1h_scatter**[_u32base]_**offset**[_s32](svbool_t *pg*, svuint32_t *bases*, int64_t *offset*, svint32_t *data*) void **svst1h_scatter**[_u64base]_**offset**[_s64](svbool_t *pg*, svuint64_t *bases*, int64_t *offset*, svint64_t *data*) void **svst1h_scatter**[_u32base]_**offset**[_u32](svbool_t *pg*, svuint32_t *bases*, int64_t *offset*, svuint32_t *data*) void **svst1h_scatter**[_u64base]_**offset**[_u64](svbool_t *pg*, svuint64_t *bases*, int64_t *offset*, svuint64_t *data*)

6.3.3.6. ST1H (scalar base, vector index)

Instances
void **svst1h_scatter**_[s32]**index**[_s32](svbool_t *pg*, int16_t **base*,

Instances
svint32_t *indices*, svint32_t *data*) void **svst1h_scatter**_[s64]**index**[_s64](svbool_t *pg*, int16_t **base*, svint64_t *indices*, svint64_t *data*) void **svst1h_scatter**_[s32]**index**[_u32](svbool_t *pg*, uint16_t **base*, svint32_t *indices*, svuint32_t *data*) void **svst1h_scatter**_[s64]**index**[_u64](svbool_t *pg*, uint16_t **base*, svint64_t *indices*, svuint64_t *data*) void **svst1h_scatter**_[u32]**index**[_s32](svbool_t *pg*, int16_t **base*, svuint32_t *indices*, svint32_t *data*) void **svst1h_scatter**_[u64]**index**[_s64](svbool_t *pg*, int16_t **base*, svuint64_t *indices*, svint64_t *data*) void **svst1h_scatter**_[u32]**index**[_u32](svbool_t *pg*, uint16_t **base*, svuint32_t *indices*, svuint32_t *data*) void **svst1h_scatter**_[u64]**index**[_u64](svbool_t *pg*, uint16_t **base*, svuint64_t *indices*, svuint64_t *data*)

6.3.3.7. ST1H (vector base, scalar index)

Instances
void **svst1h_scatter**[_u32base]_**index**[_s32](svbool_t *pg*, svuint32_t *bases*, 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*)

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
void **svst1w**[_s64](svbool_t *pg*, int32_t **base*, svint64_t *data*) void **svst1w**[_u64](svbool_t *pg*, uint32_t **base*, svuint64_t *data*)

6.3.4.2. ST1W (scalar base, VL displacement)

Instances
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*)

6.3.4.3. ST1W (vector base)

Instances
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*,

Instances
svuint64_t *data*)

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

Instances
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*)

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

Instances
void **svst1w_scatter**[_u64base]_**offset**[_s64](svbool_t *pg*, svuint64_t *bases*, int64_t *offset*, svint64_t *data*) void **svst1w_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 **svst1w_scatter**_[s64]**index**[_s64](svbool_t *pg*, int32_t **base*, svint64_t *indices*, svint64_t *data*) void **svst1w_scatter**_[s64]**index**[_u64](svbool_t *pg*, uint32_t **base*, svint64_t *indices*, svuint64_t *data*) void **svst1w_scatter**_[u64]**index**[_s64](svbool_t *pg*, int32_t **base*, svuint64_t *indices*, svint64_t *data*) void **svst1w_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 **svst1w_scatter**[_u64base]_**index**[_s64](svbool_t *pg*, svuint64_t *bases*, int64_t *index*, svint64_t *data*) void **svst1w_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*)

Instances
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*)

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*)
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*)

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*)

Instances
void **svst2**[_f64](svbool_t *pg*, float64_t **base*, svfloat64x2_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*) 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*)

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*)

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*,

Instances

```
                      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)
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)
```

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)
```

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)
```


Instances
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*)

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*, svprfop *op*)

6.4.1.2. PRFB (scalar base, VL displacement)

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

6.4.1.3. PRFB (vector base)

Instances
void **svprfb_gather**[_u32base](svbool_t *pg*, svuint32_t *bases*, svprfop *op*) void **svprfb_gather**[_u64base](svbool_t *pg*, svuint64_t *bases*, 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*, svprfop *op*) void **svprfb_gather**_[s64]**offset**(svbool_t *pg*, const void **base*, svint64_t *offsets*, svprfop *op*) void **svprfb_gather**_[u32]**offset**(svbool_t *pg*, const void **base*, svuint32_t *offsets*, svprfop *op*) void **svprfb_gather**_[u64]**offset**(svbool_t *pg*, const void **base*, svuint64_t *offsets*, 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*, svprfop *op*) void **svprfb_gather**[_u64base]_**offset**(svbool_t *pg*, svuint64_t *bases*, int64_t *offset*, 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*, svprfop *op*)

6.4.2.2. PRFH (scalar base, VL displacement)

Instances
void **svprfh_vnum**(svbool_t *pg*, const void **base*, int64_t *vnum*, svprfop *op*)

6.4.2.3. PRFH (vector base)

Instances
void **svprfh_gather**[_u32base](svbool_t *pg*, svuint32_t *bases*, svprfop *op*) void **svprfh_gather**[_u64base](svbool_t *pg*, svuint64_t *bases*, svprfop *op*)

6.4.2.4. PRFH (scalar base, vector index)

Instances
void **svprfh_gather**_[s32]**index**(svbool_t *pg*, const void **base*, svint32_t *indices*, svprfop *op*) void **svprfh_gather**_[s64]**index**(svbool_t *pg*, const void **base*, svint64_t *indices*, svprfop *op*) void **svprfh_gather**_[u32]**index**(svbool_t *pg*, const void **base*, svuint32_t *indices*, svprfop *op*) void **svprfh_gather**_[u64]**index**(svbool_t *pg*, const void **base*, svuint64_t *indices*, svprfop *op*)

6.4.2.5. PRFH (vector base, scalar index)

Instances
void **svprfh_gather**[_u32base]_**index**(svbool_t *pg*, svuint32_t *bases*, int64_t *index*, svprfop *op*) void **svprfh_gather**[_u64base]_**index**(svbool_t *pg*, svuint64_t *bases*, int64_t *index*, svprfop *op*)

6.4.3. PRFW: Prefetch 32-bit data

6.4.3.1. PRFW (scalar base)

Instances
void **svprfw**(svbool_t *pg*, const void **base*, svprfop *op*)

6.4.3.2. PRFW (scalar base, VL displacement)

Instances
void **svprfw_vnum**(svbool_t *pg*, const void **base*, int64_t *vnum*, svprfop *op*)

6.4.3.3. PRFW (vector base)

Instances
void **svprfw_gather**[_u32base](svbool_t *pg*, svuint32_t *bases*, svprfop *op*) void **svprfw_gather**[_u64base](svbool_t *pg*, svuint64_t *bases*, 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*, svprfop *op*) void **svprfw_gather**_[s64]**index**(svbool_t *pg*, const void **base*, svint64_t *indices*, svprfop *op*) void **svprfw_gather**_[u32]**index**(svbool_t *pg*, const void **base*, svuint32_t *indices*, svprfop *op*) void **svprfw_gather**_[u64]**index**(svbool_t *pg*, const void **base*, svuint64_t *indices*, 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*, svprfop *op*) void **svprfw_gather**[_u64base]_**index**(svbool_t *pg*, svuint64_t *bases*, int64_t *index*, 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*, svprfop *op*)

6.4.4.2. PRFD (scalar base, VL displacement)

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

6.4.4.3. PRFD (vector base)

Instances
void **svprfd_gather**[_u32base](svbool_t *pg*, svuint32_t *bases*, svprfop *op*) void **svprfd_gather**[_u64base](svbool_t *pg*, svuint64_t *bases*, 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*, svprfop *op*) void **svprfd_gather**_[s64]**index**(svbool_t *pg*, const void **base*, svint64_t *indices*, svprfop *op*) void **svprfd_gather**_[u32]**index**(svbool_t *pg*, const void **base*, svuint32_t *indices*, svprfop *op*)

Instances
`void svprfd_gather_[u64]index(svbool_t pg, const void *base, svuint64_t indices, 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, svprfop op)` `void svprfd_gather[_u64base]_index(svbool_t pg, svuint64_t bases, int64_t index, 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)`
`svuint64_t svadrw[_u64base]_[s64]index(svuint64_t bases, svint64_t indices)`
`svuint32_t svadrw[_u32base]_[u32]index(svuint32_t bases,`

Instances
svuint32_t *indices*) svuint64_t **svadrw**[_u64base]_[u64]**index**(svuint64_t *bases*, svuint64_t *indices*)

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 **svadrd**[_u32base]_[s32]**index**(svuint32_t *bases*, svint32_t *indices*) svuint64_t **svadrd**[_u64base]_[s64]**index**(svuint64_t *bases*, svint64_t *indices*) svuint32_t **svadrd**[_u32base]_[u32]**index**(svuint32_t *bases*, svuint32_t *indices*) svuint64_t **svadrd**[_u64base]_[u64]**index**(svuint64_t *bases*, svuint64_t *indices*)

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 **svdup**[_n]_**s8**(int8_t *op*) svint16_t **svdup**[_n]_**s16**(int16_t *op*) svint32_t **svdup**[_n]_**s32**(int32_t *op*) svint64_t **svdup**[_n]_**s64**(int64_t *op*) svuint8_t **svdup**[_n]_**u8**(uint8_t *op*) svuint16_t **svdup**[_n]_**u16**(uint16_t *op*) svuint32_t **svdup**[_n]_**u32**(uint32_t *op*) svuint64_t **svdup**[_n]_**u64**(uint64_t *op*) svfloat16_t **svdup**[_n]_**f16**(float16_t *op*) svfloat32_t **svdup**[_n]_**f32**(float32_t *op*) svfloat64_t **svdup**[_n]_**f64**(float64_t *op*)

6.6.1.2. DUP (scalar), setting inactive to zero

Instances
svint8_t **svdup**[_n]_**s8_z**(svbool_t *pg*, int8_t *op*) svint16_t **svdup**[_n]_**s16_z**(svbool_t *pg*, int16_t *op*) 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*)

Instances
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*)

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*)

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*)

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
svint8_t **svdupq**[_n]_**s8**(int8_t *x0*, int8_t *x1*, int8_t *x2*, int8_t *x3*,

Instances
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*)

6.6.2.2. DUPQ (8 scalars)

Instances
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*)

6.6.2.3. DUPQ (4 scalars)

Instances
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*)

6.6.2.4. DUPQ (2 scalars)

Instances
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*)

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
svint8_t **svindex_s8**(int8_t *base*, int8_t *step*) svint16_t **svindex_s16**(int16_t *base*, int16_t *step*) svint32_t **svindex_s32**(int32_t *base*, int32_t *step*) svint64_t **svindex_s64**(int64_t *base*, int64_t *step*) svuint8_t **svindex_u8**(uint8_t *base*, uint8_t *step*) svuint16_t **svindex_u16**(uint16_t *base*, uint16_t *step*)

Instances
svuint32_t **svindex_u32**(uint32_t *base*, uint32_t *step*)
svuint64_t **svindex_u64**(uint64_t *base*, uint64_t *step*)

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
svint8_t **svadd**[_s8]_**z**(svbool_t *pg*, svint8_t *op1*, svint8_t *op2*)
svint16_t **svadd**[_s16]_**z**(svbool_t *pg*, svint16_t *op1*, svint16_t *op2*)
svint32_t **svadd**[_s32]_**z**(svbool_t *pg*, svint32_t *op1*, svint32_t *op2*)
svint64_t **svadd**[_s64]_**z**(svbool_t *pg*, svint64_t *op1*, svint64_t *op2*)
svuint8_t **svadd**[_u8]_**z**(svbool_t *pg*, svuint8_t *op1*, svuint8_t *op2*)
svuint16_t **svadd**[_u16]_**z**(svbool_t *pg*, svuint16_t *op1*, svuint16_t *op2*)
svuint32_t **svadd**[_u32]_**z**(svbool_t *pg*, svuint32_t *op1*, svuint32_t *op2*)
svuint64_t **svadd**[_u64]_**z**(svbool_t *pg*, svuint64_t *op1*, svuint64_t *op2*)

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

Instances
svint8_t **svadd**[_s8]_**m**(svbool_t *pg*, svint8_t *op1*, svint8_t *op2*)
svint16_t **svadd**[_s16]_**m**(svbool_t *pg*, svint16_t *op1*, svint16_t *op2*)
svint32_t **svadd**[_s32]_**m**(svbool_t *pg*, svint32_t *op1*, svint32_t *op2*)
svint64_t **svadd**[_s64]_**m**(svbool_t *pg*, svint64_t *op1*, svint64_t *op2*)
svuint8_t **svadd**[_u8]_**m**(svbool_t *pg*, svuint8_t *op1*, svuint8_t *op2*)
svuint16_t **svadd**[_u16]_**m**(svbool_t *pg*, svuint16_t *op1*, svuint16_t *op2*)
svuint32_t **svadd**[_u32]_**m**(svbool_t *pg*, svuint32_t *op1*, svuint32_t *op2*)
svuint64_t **svadd**[_u64]_**m**(svbool_t *pg*, svuint64_t *op1*, svuint64_t *op2*)

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

Instances
svint8_t **svadd**[_s8]_**x**(svbool_t *pg*, svint8_t *op1*, svint8_t *op2*)
svint16_t **svadd**[_s16]_**x**(svbool_t *pg*, svint16_t *op1*, svint16_t *op2*)
svint32_t **svadd**[_s32]_**x**(svbool_t *pg*, svint32_t *op1*, svint32_t *op2*)
svint64_t **svadd**[_s64]_**x**(svbool_t *pg*, svint64_t *op1*, svint64_t *op2*)
svuint8_t **svadd**[_u8]_**x**(svbool_t *pg*, svuint8_t *op1*, svuint8_t *op2*)
svuint16_t **svadd**[_u16]_**x**(svbool_t *pg*, svuint16_t *op1*, svuint16_t *op2*)
svuint32_t **svadd**[_u32]_**x**(svbool_t *pg*, svuint32_t *op1*, svuint32_t *op2*)
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*)

Instances
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 **svqadd**[_n_s8](svint8_t *op1*, int8_t *op2*)
svint16_t **svqadd**[_n_s16](svint16_t *op1*, int16_t *op2*)
svint32_t **svqadd**[_n_s32](svint32_t *op1*, int32_t *op2*)

Instances
svint64_t **svqadd**[_n_s64](svint64_t *op1*, int64_t *op2*)
svuint8_t **svqadd**[_n_u8](svuint8_t *op1*, uint8_t *op2*)
svuint16_t **svqadd**[_n_u16](svuint16_t *op1*, uint16_t *op2*)
svuint32_t **svqadd**[_n_u32](svuint32_t *op1*, uint32_t *op2*)
svuint64_t **svqadd**[_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 **svsub**[_s8]_**z**(svbool_t *pg*, svint8_t *op1*, svint8_t *op2*)
svint16_t **svsub**[_s16]_**z**(svbool_t *pg*, svint16_t *op1*, svint16_t *op2*)
svint32_t **svsub**[_s32]_**z**(svbool_t *pg*, svint32_t *op1*, svint32_t *op2*)
svint64_t **svsub**[_s64]_**z**(svbool_t *pg*, svint64_t *op1*, svint64_t *op2*)
svuint8_t **svsub**[_u8]_**z**(svbool_t *pg*, svuint8_t *op1*, svuint8_t *op2*)
svuint16_t **svsub**[_u16]_**z**(svbool_t *pg*, svuint16_t *op1*, svuint16_t *op2*)
svuint32_t **svsub**[_u32]_**z**(svbool_t *pg*, svuint32_t *op1*, svuint32_t *op2*)
svuint64_t **svsub**[_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 **svsub**[_s8]_**m**(svbool_t *pg*, svint8_t *op1*, svint8_t *op2*)
svint16_t **svsub**[_s16]_**m**(svbool_t *pg*, svint16_t *op1*, svint16_t *op2*)
svint32_t **svsub**[_s32]_**m**(svbool_t *pg*, svint32_t *op1*, svint32_t *op2*)
svint64_t **svsub**[_s64]_**m**(svbool_t *pg*, svint64_t *op1*, svint64_t *op2*)
svuint8_t **svsub**[_u8]_**m**(svbool_t *pg*, svuint8_t *op1*, svuint8_t *op2*)
svuint16_t **svsub**[_u16]_**m**(svbool_t *pg*, svuint16_t *op1*, svuint16_t *op2*)
svuint32_t **svsub**[_u32]_**m**(svbool_t *pg*, svuint32_t *op1*, svuint32_t *op2*)
svuint64_t **svsub**[_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 **svsub**[_s8]_**x**(svbool_t *pg*, svint8_t *op1*, svint8_t *op2*)
svint16_t **svsub**[_s16]_**x**(svbool_t *pg*, svint16_t *op1*, svint16_t *op2*)
svint32_t **svsub**[_s32]_**x**(svbool_t *pg*, svint32_t *op1*, svint32_t *op2*)
svint64_t **svsub**[_s64]_**x**(svbool_t *pg*, svint64_t *op1*, svint64_t *op2*)
svuint8_t **svsub**[_u8]_**x**(svbool_t *pg*, svuint8_t *op1*, svuint8_t *op2*)
svuint16_t **svsub**[_u16]_**x**(svbool_t *pg*, svuint16_t *op1*, svuint16_t *op2*)
svuint32_t **svsub**[_u32]_**x**(svbool_t *pg*, svuint32_t *op1*, svuint32_t *op2*)
svuint64_t **svsub**[_u64]_**x**(svbool_t *pg*, svuint64_t *op1*, svuint64_t *op2*)

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

Instances
svint8_t **svsub**[_n_s8]_**z**(svbool_t *pg*, svint8_t *op1*, int8_t *op2*)
svint16_t **svsub**[_n_s16]_**z**(svbool_t *pg*, svint16_t *op1*, int16_t *op2*)

Instances
svint32_t **svsub**[_n_s32]_**z**(svbool_t *pg*, svint32_t *op1*, int32_t *op2*)
svint64_t **svsub**[_n_s64]_**z**(svbool_t *pg*, svint64_t *op1*, int64_t *op2*)
svuint8_t **svsub**[_n_u8]_**z**(svbool_t *pg*, svuint8_t *op1*, uint8_t *op2*)
svuint16_t **svsub**[_n_u16]_**z**(svbool_t *pg*, svuint16_t *op1*, uint16_t *op2*)
svuint32_t **svsub**[_n_u32]_**z**(svbool_t *pg*, svuint32_t *op1*, uint32_t *op2*)
svuint64_t **svsub**[_n_u64]_**z**(svbool_t *pg*, svuint64_t *op1*, uint64_t *op2*)

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

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

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

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

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
svint8_t **svsubr**[_s8]_**z**(svbool_t *pg*, svint8_t *op1*, svint8_t *op2*)
svint16_t **svsubr**[_s16]_**z**(svbool_t *pg*, svint16_t *op1*, svint16_t *op2*)
svint32_t **svsubr**[_s32]_**z**(svbool_t *pg*, svint32_t *op1*, svint32_t *op2*)
svint64_t **svsubr**[_s64]_**z**(svbool_t *pg*, svint64_t *op1*, svint64_t *op2*)
svuint8_t **svsubr**[_u8]_**z**(svbool_t *pg*, svuint8_t *op1*, svuint8_t *op2*)
svuint16_t **svsubr**[_u16]_**z**(svbool_t *pg*, svuint16_t *op1*, svuint16_t *op2*)
svuint32_t **svsubr**[_u32]_**z**(svbool_t *pg*, svuint32_t *op1*, svuint32_t *op2*)
svuint64_t **svsubr**[_u64]_**z**(svbool_t *pg*, svuint64_t *op1*, svuint64_t *op2*)

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

Instances
svint8_t **svsubr**[_s8]_**m**(svbool_t *pg*, svint8_t *op1*, svint8_t *op2*)

Instances
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*)

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

Instances
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*)

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

Instances
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*)

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

Instances
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*)

6.7.4.6. SUBR (vector, scalar), setting inactive to unknown

Instances
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*)
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*)

Instances
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 **svabd**[_s8]_**m**(svbool_t *pg*, svint8_t *op1*, svint8_t *op2*)

Instances
svint16_t **svabd**[_s16]_**m**(svbool_t *pg*, svint16_t *op1*, svint16_t *op2*)
svint32_t **svabd**[_s32]_**m**(svbool_t *pg*, svint32_t *op1*, svint32_t *op2*)
svint64_t **svabd**[_s64]_**m**(svbool_t *pg*, svint64_t *op1*, svint64_t *op2*)
svuint8_t **svabd**[_u8]_**m**(svbool_t *pg*, svuint8_t *op1*, svuint8_t *op2*)
svuint16_t **svabd**[_u16]_**m**(svbool_t *pg*, svuint16_t *op1*, svuint16_t *op2*)
svuint32_t **svabd**[_u32]_**m**(svbool_t *pg*, svuint32_t *op1*, svuint32_t *op2*)
svuint64_t **svabd**[_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 **svabd**[_s8]_**x**(svbool_t *pg*, svint8_t *op1*, svint8_t *op2*)
svint16_t **svabd**[_s16]_**x**(svbool_t *pg*, svint16_t *op1*, svint16_t *op2*)
svint32_t **svabd**[_s32]_**x**(svbool_t *pg*, svint32_t *op1*, svint32_t *op2*)
svint64_t **svabd**[_s64]_**x**(svbool_t *pg*, svint64_t *op1*, svint64_t *op2*)
svuint8_t **svabd**[_u8]_**x**(svbool_t *pg*, svuint8_t *op1*, svuint8_t *op2*)
svuint16_t **svabd**[_u16]_**x**(svbool_t *pg*, svuint16_t *op1*, svuint16_t *op2*)
svuint32_t **svabd**[_u32]_**x**(svbool_t *pg*, svuint32_t *op1*, svuint32_t *op2*)
svuint64_t **svabd**[_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 **svabd**[_n_s8]_**z**(svbool_t *pg*, svint8_t *op1*, int8_t *op2*)
svint16_t **svabd**[_n_s16]_**z**(svbool_t *pg*, svint16_t *op1*, int16_t *op2*)
svint32_t **svabd**[_n_s32]_**z**(svbool_t *pg*, svint32_t *op1*, int32_t *op2*)
svint64_t **svabd**[_n_s64]_**z**(svbool_t *pg*, svint64_t *op1*, int64_t *op2*)
svuint8_t **svabd**[_n_u8]_**z**(svbool_t *pg*, svuint8_t *op1*, uint8_t *op2*)
svuint16_t **svabd**[_n_u16]_**z**(svbool_t *pg*, svuint16_t *op1*, uint16_t *op2*)
svuint32_t **svabd**[_n_u32]_**z**(svbool_t *pg*, svuint32_t *op1*, uint32_t *op2*)
svuint64_t **svabd**[_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 **svabd**[_n_s8]_**m**(svbool_t *pg*, svint8_t *op1*, int8_t *op2*)
svint16_t **svabd**[_n_s16]_**m**(svbool_t *pg*, svint16_t *op1*, int16_t *op2*)
svint32_t **svabd**[_n_s32]_**m**(svbool_t *pg*, svint32_t *op1*, int32_t *op2*)
svint64_t **svabd**[_n_s64]_**m**(svbool_t *pg*, svint64_t *op1*, int64_t *op2*)
svuint8_t **svabd**[_n_u8]_**m**(svbool_t *pg*, svuint8_t *op1*, uint8_t *op2*)
svuint16_t **svabd**[_n_u16]_**m**(svbool_t *pg*, svuint16_t *op1*, uint16_t *op2*)
svuint32_t **svabd**[_n_u32]_**m**(svbool_t *pg*, svuint32_t *op1*, uint32_t *op2*)
svuint64_t **svabd**[_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 **svabd**[_n_s8]_**x**(svbool_t *pg*, svint8_t *op1*, int8_t *op2*)
svint16_t **svabd**[_n_s16]_**x**(svbool_t *pg*, svint16_t *op1*, int16_t *op2*)
svint32_t **svabd**[_n_s32]_**x**(svbool_t *pg*, svint32_t *op1*, int32_t *op2*)
svint64_t **svabd**[_n_s64]_**x**(svbool_t *pg*, svint64_t *op1*, int64_t *op2*)
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*)

Instances
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*)
svint32_t **svmul**[_n_s32]_**z**(svbool_t *pg*, svint32_t *op1*, int32_t *op2*)

Instances
svint64_t **svmul**[_n_s64]_**z**(svbool_t *pg*, svint64_t *op1*, int64_t *op2*)
svuint8_t **svmul**[_n_u8]_**z**(svbool_t *pg*, svuint8_t *op1*, uint8_t *op2*)
svuint16_t **svmul**[_n_u16]_**z**(svbool_t *pg*, svuint16_t *op1*, uint16_t *op2*)
svuint32_t **svmul**[_n_u32]_**z**(svbool_t *pg*, svuint32_t *op1*, uint32_t *op2*)
svuint64_t **svmul**[_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 **svmul**[_n_s8]_**m**(svbool_t *pg*, svint8_t *op1*, int8_t *op2*)
svint16_t **svmul**[_n_s16]_**m**(svbool_t *pg*, svint16_t *op1*, int16_t *op2*)
svint32_t **svmul**[_n_s32]_**m**(svbool_t *pg*, svint32_t *op1*, int32_t *op2*)
svint64_t **svmul**[_n_s64]_**m**(svbool_t *pg*, svint64_t *op1*, int64_t *op2*)
svuint8_t **svmul**[_n_u8]_**m**(svbool_t *pg*, svuint8_t *op1*, uint8_t *op2*)
svuint16_t **svmul**[_n_u16]_**m**(svbool_t *pg*, svuint16_t *op1*, uint16_t *op2*)
svuint32_t **svmul**[_n_u32]_**m**(svbool_t *pg*, svuint32_t *op1*, uint32_t *op2*)
svuint64_t **svmul**[_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 **svmul**[_n_s8]_**x**(svbool_t *pg*, svint8_t *op1*, int8_t *op2*)
svint16_t **svmul**[_n_s16]_**x**(svbool_t *pg*, svint16_t *op1*, int16_t *op2*)
svint32_t **svmul**[_n_s32]_**x**(svbool_t *pg*, svint32_t *op1*, int32_t *op2*)
svint64_t **svmul**[_n_s64]_**x**(svbool_t *pg*, svint64_t *op1*, int64_t *op2*)
svuint8_t **svmul**[_n_u8]_**x**(svbool_t *pg*, svuint8_t *op1*, uint8_t *op2*)
svuint16_t **svmul**[_n_u16]_**x**(svbool_t *pg*, svuint16_t *op1*, uint16_t *op2*)
svuint32_t **svmul**[_n_u32]_**x**(svbool_t *pg*, svuint32_t *op1*, uint32_t *op2*)
svuint64_t **svmul**[_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 **svmulh**[_s8]_**z**(svbool_t *pg*, svint8_t *op1*, svint8_t *op2*)
svint16_t **svmulh**[_s16]_**z**(svbool_t *pg*, svint16_t *op1*, svint16_t *op2*)
svint32_t **svmulh**[_s32]_**z**(svbool_t *pg*, svint32_t *op1*, svint32_t *op2*)
svint64_t **svmulh**[_s64]_**z**(svbool_t *pg*, svint64_t *op1*, svint64_t *op2*)
svuint8_t **svmulh**[_u8]_**z**(svbool_t *pg*, svuint8_t *op1*, svuint8_t *op2*)
svuint16_t **svmulh**[_u16]_**z**(svbool_t *pg*, svuint16_t *op1*, svuint16_t *op2*)
svuint32_t **svmulh**[_u32]_**z**(svbool_t *pg*, svuint32_t *op1*, svuint32_t *op2*)
svuint64_t **svmulh**[_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 **svmulh**[_s8]_**m**(svbool_t *pg*, svint8_t *op1*, svint8_t *op2*)
svint16_t **svmulh**[_s16]_**m**(svbool_t *pg*, svint16_t *op1*, svint16_t *op2*)
svint32_t **svmulh**[_s32]_**m**(svbool_t *pg*, svint32_t *op1*, svint32_t *op2*)

Instances
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

```
svint8_t svmad[_s8]_z(svbool_t pg, svint8_t op1, svint8_t op2,
                      svint8_t op3)
svint16_t svmad[_s16]_z(svbool_t pg, svint16_t op1, svint16_t op2,
                        svint16_t op3)
svint32_t svmad[_s32]_z(svbool_t pg, svint32_t op1, svint32_t op2,
                        svint32_t op3)
svint64_t svmad[_s64]_z(svbool_t pg, svint64_t op1, svint64_t op2,
                        svint64_t op3)
svuint8_t svmad[_u8]_z(svbool_t pg, svuint8_t op1, svuint8_t op2,
                       svuint8_t op3)
svuint16_t svmad[_u16]_z(svbool_t pg, svuint16_t op1, svuint16_t op2,
                         svuint16_t op3)
svuint32_t svmad[_u32]_z(svbool_t pg, svuint32_t op1, svuint32_t op2,
                         svuint32_t op3)
svuint64_t svmad[_u64]_z(svbool_t pg, svuint64_t op1, svuint64_t op2,
                         svuint64_t op3)
```

6.7.9.2. MAD (vector, vector, vector), merging with first input

Instances

```
svint8_t svmad[_s8]_m(svbool_t pg, svint8_t op1, svint8_t op2,
                      svint8_t op3)
svint16_t svmad[_s16]_m(svbool_t pg, svint16_t op1, svint16_t op2,
                        svint16_t op3)
svint32_t svmad[_s32]_m(svbool_t pg, svint32_t op1, svint32_t op2,
                        svint32_t op3)
svint64_t svmad[_s64]_m(svbool_t pg, svint64_t op1, svint64_t op2,
                        svint64_t op3)
svuint8_t svmad[_u8]_m(svbool_t pg, svuint8_t op1, svuint8_t op2,
                       svuint8_t op3)
svuint16_t svmad[_u16]_m(svbool_t pg, svuint16_t op1, svuint16_t op2,
                         svuint16_t op3)
svuint32_t svmad[_u32]_m(svbool_t pg, svuint32_t op1, svuint32_t op2,
                         svuint32_t op3)
svuint64_t svmad[_u64]_m(svbool_t pg, svuint64_t op1, svuint64_t op2,
                         svuint64_t op3)
```

6.7.9.3. MAD (vector, vector, vector), setting inactive to unknown

Instances

```
svint8_t svmad[_s8]_x(svbool_t pg, svint8_t op1, svint8_t op2,
                      svint8_t op3)
svint16_t svmad[_s16]_x(svbool_t pg, svint16_t op1, svint16_t op2,
                        svint16_t op3)
svint32_t svmad[_s32]_x(svbool_t pg, svint32_t op1, svint32_t op2,
                        svint32_t op3)
```


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

```
                          int8_t op3)
svint16_t svmad[_n_s16]_x(svbool_t pg, svint16_t op1, svint16_t op2,
                           int16_t op3)
svint32_t svmad[_n_s32]_x(svbool_t pg, svint32_t op1, svint32_t op2,
                           int32_t op3)
svint64_t svmad[_n_s64]_x(svbool_t pg, svint64_t op1, svint64_t op2,
                           int64_t op3)
svuint8_t svmad[_n_u8]_x(svbool_t pg, svuint8_t op1, svuint8_t op2,
                          uint8_t op3)
svuint16_t svmad[_n_u16]_x(svbool_t pg, svuint16_t op1, svuint16_t op2,
                           uint16_t op3)
svuint32_t svmad[_n_u32]_x(svbool_t pg, svuint32_t op1, svuint32_t op2,
                           uint32_t op3)
svuint64_t svmad[_n_u64]_x(svbool_t pg, svuint64_t op1, svuint64_t op2,
                           uint64_t op3)
```

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

```
svint8_t svmla[_s8]_z(svbool_t pg, svint8_t op1, svint8_t op2,
                      svint8_t op3)
svint16_t svmla[_s16]_z(svbool_t pg, svint16_t op1, svint16_t op2,
                        svint16_t op3)
svint32_t svmla[_s32]_z(svbool_t pg, svint32_t op1, svint32_t op2,
                        svint32_t op3)
svint64_t svmla[_s64]_z(svbool_t pg, svint64_t op1, svint64_t op2,
                        svint64_t op3)
svuint8_t svmla[_u8]_z(svbool_t pg, svuint8_t op1, svuint8_t op2,
                       svuint8_t op3)
svuint16_t svmla[_u16]_z(svbool_t pg, svuint16_t op1, svuint16_t op2,
                         svuint16_t op3)
svuint32_t svmla[_u32]_z(svbool_t pg, svuint32_t op1, svuint32_t op2,
                         svuint32_t op3)
svuint64_t svmla[_u64]_z(svbool_t pg, svuint64_t op1, svuint64_t op2,
                         svuint64_t op3)
```

6.7.10.2. MLA (vector, vector, vector), merging with first input

Instances

```
svint8_t svmla[_s8]_m(svbool_t pg, svint8_t op1, svint8_t op2,
                      svint8_t op3)
svint16_t svmla[_s16]_m(svbool_t pg, svint16_t op1, svint16_t op2,
                        svint16_t op3)
svint32_t svmla[_s32]_m(svbool_t pg, svint32_t op1, svint32_t op2,
                        svint32_t op3)
svint64_t svmla[_s64]_m(svbool_t pg, svint64_t op1, svint64_t op2,
                        svint64_t op3)
svuint8_t svmla[_u8]_m(svbool_t pg, svuint8_t op1, svuint8_t op2,
```


Instances
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

```
                          int16_t op3)
svint32_t svmla[_n_s32]_m(svbool_t pg, svint32_t op1, svint32_t op2,
                          int32_t op3)
svint64_t svmla[_n_s64]_m(svbool_t pg, svint64_t op1, svint64_t op2,
                          int64_t op3)
svuint8_t svmla[_n_u8]_m(svbool_t pg, svuint8_t op1, svuint8_t op2,
                         uint8_t op3)
svuint16_t svmla[_n_u16]_m(svbool_t pg, svuint16_t op1, svuint16_t op2,
                           uint16_t op3)
svuint32_t svmla[_n_u32]_m(svbool_t pg, svuint32_t op1, svuint32_t op2,
                           uint32_t op3)
svuint64_t svmla[_n_u64]_m(svbool_t pg, svuint64_t op1, svuint64_t op2,
                           uint64_t op3)
```

6.7.10.6. MLA (vector, vector, scalar), setting inactive to unknown

Instances

```
svint8_t svmla[_n_s8]_x(svbool_t pg, svint8_t op1, svint8_t op2,
                        int8_t op3)
svint16_t svmla[_n_s16]_x(svbool_t pg, svint16_t op1, svint16_t op2,
                          int16_t op3)
svint32_t svmla[_n_s32]_x(svbool_t pg, svint32_t op1, svint32_t op2,
                          int32_t op3)
svint64_t svmla[_n_s64]_x(svbool_t pg, svint64_t op1, svint64_t op2,
                          int64_t op3)
svuint8_t svmla[_n_u8]_x(svbool_t pg, svuint8_t op1, svuint8_t op2,
                         uint8_t op3)
svuint16_t svmla[_n_u16]_x(svbool_t pg, svuint16_t op1, svuint16_t op2,
                           uint16_t op3)
svuint32_t svmla[_n_u32]_x(svbool_t pg, svuint32_t op1, svuint32_t op2,
                           uint32_t op3)
svuint64_t svmla[_n_u64]_x(svbool_t pg, svuint64_t op1, svuint64_t op2,
                           uint64_t op3)
```

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

```
svint8_t svmsb[_s8]_z(svbool_t pg, svint8_t op1, svint8_t op2,
                      svint8_t op3)
svint16_t svmsb[_s16]_z(svbool_t pg, svint16_t op1, svint16_t op2,
                        svint16_t op3)
svint32_t svmsb[_s32]_z(svbool_t pg, svint32_t op1, svint32_t op2,
                        svint32_t op3)
svint64_t svmsb[_s64]_z(svbool_t pg, svint64_t op1, svint64_t op2,
                        svint64_t op3)
svuint8_t svmsb[_u8]_z(svbool_t pg, svuint8_t op1, svuint8_t op2,
                       svuint8_t op3)
svuint16_t svmsb[_u16]_z(svbool_t pg, svuint16_t op1, svuint16_t op2,
```


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 svmls[_s8]_z(svbool_t pg, svint8_t op1, svint8_t op2,
                      svint8_t op3)
svint16_t svmls[_s16]_z(svbool_t pg, svint16_t op1, svint16_t op2,
                        svint16_t op3)
svint32_t svmls[_s32]_z(svbool_t pg, svint32_t op1, svint32_t op2,
                        svint32_t op3)
svint64_t svmls[_s64]_z(svbool_t pg, svint64_t op1, svint64_t op2,
                        svint64_t op3)
svuint8_t svmls[_u8]_z(svbool_t pg, svuint8_t op1, svuint8_t op2,
                       svuint8_t op3)
svuint16_t svmls[_u16]_z(svbool_t pg, svuint16_t op1, svuint16_t op2,
                         svuint16_t op3)
svuint32_t svmls[_u32]_z(svbool_t pg, svuint32_t op1, svuint32_t op2,
                         svuint32_t op3)
svuint64_t svmls[_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 svmls[_s8]_m(svbool_t pg, svint8_t op1, svint8_t op2,
                      svint8_t op3)
svint16_t svmls[_s16]_m(svbool_t pg, svint16_t op1, svint16_t op2,
                        svint16_t op3)
svint32_t svmls[_s32]_m(svbool_t pg, svint32_t op1, svint32_t op2,
                        svint32_t op3)
svint64_t svmls[_s64]_m(svbool_t pg, svint64_t op1, svint64_t op2,
                        svint64_t op3)
svuint8_t svmls[_u8]_m(svbool_t pg, svuint8_t op1, svuint8_t op2,
                       svuint8_t op3)
svuint16_t svmls[_u16]_m(svbool_t pg, svuint16_t op1, svuint16_t op2,
                         svuint16_t op3)
svuint32_t svmls[_u32]_m(svbool_t pg, svuint32_t op1, svuint32_t op2,
                         svuint32_t op3)
svuint64_t svmls[_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 svmls[_s8]_x(svbool_t pg, svint8_t op1, svint8_t op2,
                      svint8_t op3)
svint16_t svmls[_s16]_x(svbool_t pg, svint16_t op1, svint16_t op2,
                        svint16_t op3)
svint32_t svmls[_s32]_x(svbool_t pg, svint32_t op1, svint32_t op2,
                        svint32_t op3)
svint64_t svmls[_s64]_x(svbool_t pg, svint64_t op1, svint64_t op2,
                        svint64_t op3)
svuint8_t svmls[_u8]_x(svbool_t pg, svuint8_t op1, svuint8_t op2,
                       svuint8_t op3)
svuint16_t svmls[_u16]_x(svbool_t pg, svuint16_t op1, svuint16_t op2,
```


Instances
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*)

6.7.12.4. MLS (vector, vector, scalar), setting inactive to zero

Instances
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*)

6.7.12.5. MLS (vector, vector, scalar), merging with first input

Instances
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*)

6.7.12.6. MLS (vector, vector, scalar), setting inactive to unknown

Instances
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*,

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 *`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

`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)`

6.7.14.2. DIV (vector, vector), merging with first input

Instances

`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)`

6.7.14.3. DIV (vector, vector), setting inactive to unknown

Instances

`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)`

6.7.14.4. DIV (vector, scalar), setting inactive to zero

Instances

`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)`

6.7.14.5. DIV (vector, scalar), merging with first input

Instances

`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)`

6.7.14.6. DIV (vector, scalar), setting inactive to unknown

Instances
`svint32_t svdiv[_n_s32]_x(svbool_t pg, svint32_t op1, int32_t op2)`

Instances
svint64_t **svdiv**[_n_s64]_**x**(svbool_t *pg*, svint64_t *op1*, int64_t *op2*)
svuint32_t **svdiv**[_n_u32]_**x**(svbool_t *pg*, svuint32_t *op1*, uint32_t *op2*)
svuint64_t **svdiv**[_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 **svdivr**[_s32]_**z**(svbool_t *pg*, svint32_t *op1*, svint32_t *op2*)
svint64_t **svdivr**[_s64]_**z**(svbool_t *pg*, svint64_t *op1*, svint64_t *op2*)
svuint32_t **svdivr**[_u32]_**z**(svbool_t *pg*, svuint32_t *op1*, svuint32_t *op2*)
svuint64_t **svdivr**[_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 **svdivr**[_s32]_**m**(svbool_t *pg*, svint32_t *op1*, svint32_t *op2*)
svint64_t **svdivr**[_s64]_**m**(svbool_t *pg*, svint64_t *op1*, svint64_t *op2*)
svuint32_t **svdivr**[_u32]_**m**(svbool_t *pg*, svuint32_t *op1*, svuint32_t *op2*)
svuint64_t **svdivr**[_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 **svdivr**[_s32]_**x**(svbool_t *pg*, svint32_t *op1*, svint32_t *op2*)
svint64_t **svdivr**[_s64]_**x**(svbool_t *pg*, svint64_t *op1*, svint64_t *op2*)
svuint32_t **svdivr**[_u32]_**x**(svbool_t *pg*, svuint32_t *op1*, svuint32_t *op2*)
svuint64_t **svdivr**[_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 **svdivr**[_n_s32]_**z**(svbool_t *pg*, svint32_t *op1*, int32_t *op2*)
svint64_t **svdivr**[_n_s64]_**z**(svbool_t *pg*, svint64_t *op1*, int64_t *op2*)
svuint32_t **svdivr**[_n_u32]_**z**(svbool_t *pg*, svuint32_t *op1*, uint32_t *op2*)
svuint64_t **svdivr**[_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 **svdivr**[_n_s32]_**m**(svbool_t *pg*, svint32_t *op1*, int32_t *op2*)
svint64_t **svdivr**[_n_s64]_**m**(svbool_t *pg*, svint64_t *op1*, int64_t *op2*)
svuint32_t **svdivr**[_n_u32]_**m**(svbool_t *pg*, svuint32_t *op1*, uint32_t *op2*)
svuint64_t **svdivr**[_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 **svdivr**[_n_s32]_**x**(svbool_t *pg*, svint32_t *op1*, int32_t *op2*)
svint64_t **svdivr**[_n_s64]_**x**(svbool_t *pg*, svint64_t *op1*, int64_t *op2*)
svuint32_t **svdivr**[_n_u32]_**x**(svbool_t *pg*, svuint32_t *op1*, uint32_t *op2*)
svuint64_t **svdivr**[_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 **svmax**[_s8]_**z**(svbool_t *pg*, svint8_t *op1*, svint8_t *op2*)
svint16_t **svmax**[_s16]_**z**(svbool_t *pg*, svint16_t *op1*, svint16_t *op2*)
svint32_t **svmax**[_s32]_**z**(svbool_t *pg*, svint32_t *op1*, svint32_t *op2*)
svint64_t **svmax**[_s64]_**z**(svbool_t *pg*, svint64_t *op1*, svint64_t *op2*)
svuint8_t **svmax**[_u8]_**z**(svbool_t *pg*, svuint8_t *op1*, svuint8_t *op2*)
svuint16_t **svmax**[_u16]_**z**(svbool_t *pg*, svuint16_t *op1*, svuint16_t *op2*)
svuint32_t **svmax**[_u32]_**z**(svbool_t *pg*, svuint32_t *op1*, svuint32_t *op2*)
svuint64_t **svmax**[_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 **svmax**[_s8]_**m**(svbool_t *pg*, svint8_t *op1*, svint8_t *op2*)
svint16_t **svmax**[_s16]_**m**(svbool_t *pg*, svint16_t *op1*, svint16_t *op2*)
svint32_t **svmax**[_s32]_**m**(svbool_t *pg*, svint32_t *op1*, svint32_t *op2*)
svint64_t **svmax**[_s64]_**m**(svbool_t *pg*, svint64_t *op1*, svint64_t *op2*)
svuint8_t **svmax**[_u8]_**m**(svbool_t *pg*, svuint8_t *op1*, svuint8_t *op2*)
svuint16_t **svmax**[_u16]_**m**(svbool_t *pg*, svuint16_t *op1*, svuint16_t *op2*)
svuint32_t **svmax**[_u32]_**m**(svbool_t *pg*, svuint32_t *op1*, svuint32_t *op2*)
svuint64_t **svmax**[_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 **svmax**[_s8]_**x**(svbool_t *pg*, svint8_t *op1*, svint8_t *op2*)
svint16_t **svmax**[_s16]_**x**(svbool_t *pg*, svint16_t *op1*, svint16_t *op2*)
svint32_t **svmax**[_s32]_**x**(svbool_t *pg*, svint32_t *op1*, svint32_t *op2*)
svint64_t **svmax**[_s64]_**x**(svbool_t *pg*, svint64_t *op1*, svint64_t *op2*)
svuint8_t **svmax**[_u8]_**x**(svbool_t *pg*, svuint8_t *op1*, svuint8_t *op2*)
svuint16_t **svmax**[_u16]_**x**(svbool_t *pg*, svuint16_t *op1*, svuint16_t *op2*)
svuint32_t **svmax**[_u32]_**x**(svbool_t *pg*, svuint32_t *op1*, svuint32_t *op2*)
svuint64_t **svmax**[_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 **svmax**[_n_s8]_**z**(svbool_t *pg*, svint8_t *op1*, int8_t *op2*)
svint16_t **svmax**[_n_s16]_**z**(svbool_t *pg*, svint16_t *op1*, int16_t *op2*)
svint32_t **svmax**[_n_s32]_**z**(svbool_t *pg*, svint32_t *op1*, int32_t *op2*)
svint64_t **svmax**[_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 **svmin**[_u32]_**m**(svbool_t *pg*, svuint32_t *op1*, svuint32_t *op2*)
svuint64_t **svmin**[_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 **svmin**[_s8]_**x**(svbool_t *pg*, svint8_t *op1*, svint8_t *op2*)
svint16_t **svmin**[_s16]_**x**(svbool_t *pg*, svint16_t *op1*, svint16_t *op2*)
svint32_t **svmin**[_s32]_**x**(svbool_t *pg*, svint32_t *op1*, svint32_t *op2*)
svint64_t **svmin**[_s64]_**x**(svbool_t *pg*, svint64_t *op1*, svint64_t *op2*)
svuint8_t **svmin**[_u8]_**x**(svbool_t *pg*, svuint8_t *op1*, svuint8_t *op2*)
svuint16_t **svmin**[_u16]_**x**(svbool_t *pg*, svuint16_t *op1*, svuint16_t *op2*)
svuint32_t **svmin**[_u32]_**x**(svbool_t *pg*, svuint32_t *op1*, svuint32_t *op2*)
svuint64_t **svmin**[_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 **svmin**[_n_s8]_**z**(svbool_t *pg*, svint8_t *op1*, int8_t *op2*)
svint16_t **svmin**[_n_s16]_**z**(svbool_t *pg*, svint16_t *op1*, int16_t *op2*)
svint32_t **svmin**[_n_s32]_**z**(svbool_t *pg*, svint32_t *op1*, int32_t *op2*)
svint64_t **svmin**[_n_s64]_**z**(svbool_t *pg*, svint64_t *op1*, int64_t *op2*)
svuint8_t **svmin**[_n_u8]_**z**(svbool_t *pg*, svuint8_t *op1*, uint8_t *op2*)
svuint16_t **svmin**[_n_u16]_**z**(svbool_t *pg*, svuint16_t *op1*, uint16_t *op2*)
svuint32_t **svmin**[_n_u32]_**z**(svbool_t *pg*, svuint32_t *op1*, uint32_t *op2*)
svuint64_t **svmin**[_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 **svmin**[_n_s8]_**m**(svbool_t *pg*, svint8_t *op1*, int8_t *op2*)
svint16_t **svmin**[_n_s16]_**m**(svbool_t *pg*, svint16_t *op1*, int16_t *op2*)
svint32_t **svmin**[_n_s32]_**m**(svbool_t *pg*, svint32_t *op1*, int32_t *op2*)
svint64_t **svmin**[_n_s64]_**m**(svbool_t *pg*, svint64_t *op1*, int64_t *op2*)
svuint8_t **svmin**[_n_u8]_**m**(svbool_t *pg*, svuint8_t *op1*, uint8_t *op2*)
svuint16_t **svmin**[_n_u16]_**m**(svbool_t *pg*, svuint16_t *op1*, uint16_t *op2*)
svuint32_t **svmin**[_n_u32]_**m**(svbool_t *pg*, svuint32_t *op1*, uint32_t *op2*)
svuint64_t **svmin**[_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 **svmin**[_n_s8]_**x**(svbool_t *pg*, svint8_t *op1*, int8_t *op2*)
svint16_t **svmin**[_n_s16]_**x**(svbool_t *pg*, svint16_t *op1*, int16_t *op2*)
svint32_t **svmin**[_n_s32]_**x**(svbool_t *pg*, svint32_t *op1*, int32_t *op2*)
svint64_t **svmin**[_n_s64]_**x**(svbool_t *pg*, svint64_t *op1*, int64_t *op2*)
svuint8_t **svmin**[_n_u8]_**x**(svbool_t *pg*, svuint8_t *op1*, uint8_t *op2*)
svuint16_t **svmin**[_n_u16]_**x**(svbool_t *pg*, svuint16_t *op1*, uint16_t *op2*)
svuint32_t **svmin**[_n_u32]_**x**(svbool_t *pg*, svuint32_t *op1*, uint32_t *op2*)
svuint64_t **svmin**[_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 svneg[_s8]_z(svbool_t pg, svint8_t op)`
`svint16_t svneg[_s16]_z(svbool_t pg, svint16_t op)`
`svint32_t svneg[_s32]_z(svbool_t pg, svint32_t op)`
`svint64_t svneg[_s64]_z(svbool_t pg, svint64_t op)`

6.7.18.2. NEG (vector), merging with separate vector

Instances
`svint8_t svneg[_s8]_m(svint8_t inactive, svbool_t pg, svint8_t op)`
`svint16_t svneg[_s16]_m(svint16_t inactive, svbool_t pg, svint16_t op)`
`svint32_t svneg[_s32]_m(svint32_t inactive, svbool_t pg, svint32_t op)`
`svint64_t svneg[_s64]_m(svint64_t inactive, svbool_t pg, svint64_t op)`

6.7.18.3. NEG (vector), setting inactive to unknown

Instances
`svint8_t svneg[_s8]_x(svbool_t pg, svint8_t op)`
`svint16_t svneg[_s16]_x(svbool_t pg, svint16_t op)`
`svint32_t svneg[_s32]_x(svbool_t pg, svint32_t op)`
`svint64_t svneg[_s64]_x(svbool_t pg, svint64_t op)`

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 svabs[_s8]_z(svbool_t pg, svint8_t op)`
`svint16_t svabs[_s16]_z(svbool_t pg, svint16_t op)`
`svint32_t svabs[_s32]_z(svbool_t pg, svint32_t op)`
`svint64_t svabs[_s64]_z(svbool_t pg, svint64_t op)`

6.7.19.2. ABS (vector), merging with separate vector

Instances
`svint8_t svabs[_s8]_m(svint8_t inactive, svbool_t pg, svint8_t op)`
`svint16_t svabs[_s16]_m(svint16_t inactive, svbool_t pg, svint16_t op)`
`svint32_t svabs[_s32]_m(svint32_t inactive, svbool_t pg, svint32_t op)`
`svint64_t svabs[_s64]_m(svint64_t inactive, svbool_t pg, svint64_t op)`

6.7.19.3. ABS (vector), setting inactive to unknown

Instances
`svint8_t svabs[_s8]_x(svbool_t pg, svint8_t op)`
`svint16_t svabs[_s16]_x(svbool_t pg, svint16_t op)`

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
svuint64_t **svnot**[_u64]_**z**(svbool_t *pg*, svuint64_t *op*)

6.8.5.2. NOT (vector), merging with separate vector

Instances
svint8_t **svnot**[_s8]_**m**(svint8_t *inactive*, svbool_t *pg*, svint8_t *op*)
svint16_t **svnot**[_s16]_**m**(svint16_t *inactive*, svbool_t *pg*, svint16_t *op*)
svint32_t **svnot**[_s32]_**m**(svint32_t *inactive*, svbool_t *pg*, svint32_t *op*)
svint64_t **svnot**[_s64]_**m**(svint64_t *inactive*, svbool_t *pg*, svint64_t *op*)
svuint8_t **svnot**[_u8]_**m**(svuint8_t *inactive*, svbool_t *pg*, svuint8_t *op*)
svuint16_t **svnot**[_u16]_**m**(svuint16_t *inactive*, svbool_t *pg*, svuint16_t *op*)
svuint32_t **svnot**[_u32]_**m**(svuint32_t *inactive*, svbool_t *pg*, svuint32_t *op*)
svuint64_t **svnot**[_u64]_**m**(svuint64_t *inactive*, svbool_t *pg*, svuint64_t *op*)

6.8.5.3. NOT (vector), setting inactive to unknown

Instances
svint8_t **svnot**[_s8]_**x**(svbool_t *pg*, svint8_t *op*)
svint16_t **svnot**[_s16]_**x**(svbool_t *pg*, svint16_t *op*)
svint32_t **svnot**[_s32]_**x**(svbool_t *pg*, svint32_t *op*)
svint64_t **svnot**[_s64]_**x**(svbool_t *pg*, svint64_t *op*)
svuint8_t **svnot**[_u8]_**x**(svbool_t *pg*, svuint8_t *op*)
svuint16_t **svnot**[_u16]_**x**(svbool_t *pg*, svuint16_t *op*)
svuint32_t **svnot**[_u32]_**x**(svbool_t *pg*, svuint32_t *op*)
svuint64_t **svnot**[_u64]_**x**(svbool_t *pg*, svuint64_t *op*)

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
svint8_t **svcnot**[_s8]_**z**(svbool_t *pg*, svint8_t *op*)
svint16_t **svcnot**[_s16]_**z**(svbool_t *pg*, svint16_t *op*)
svint32_t **svcnot**[_s32]_**z**(svbool_t *pg*, svint32_t *op*)
svint64_t **svcnot**[_s64]_**z**(svbool_t *pg*, svint64_t *op*)
svuint8_t **svcnot**[_u8]_**z**(svbool_t *pg*, svuint8_t *op*)
svuint16_t **svcnot**[_u16]_**z**(svbool_t *pg*, svuint16_t *op*)
svuint32_t **svcnot**[_u32]_**z**(svbool_t *pg*, svuint32_t *op*)
svuint64_t **svcnot**[_u64]_**z**(svbool_t *pg*, svuint64_t *op*)

6.8.6.2. CNOT (vector), merging with separate vector

Instances
svint8_t **svcnot**[_s8]_**m**(svint8_t *inactive*, svbool_t *pg*, svint8_t *op*)
svint16_t **svcnot**[_s16]_**m**(svint16_t *inactive*, svbool_t *pg*, svint16_t *op*)
svint32_t **svcnot**[_s32]_**m**(svint32_t *inactive*, svbool_t *pg*, svint32_t *op*)
svint64_t **svcnot**[_s64]_**m**(svint64_t *inactive*, svbool_t *pg*, svint64_t *op*)
svuint8_t **svcnot**[_u8]_**m**(svuint8_t *inactive*, svbool_t *pg*, svuint8_t *op*)
svuint16_t **svcnot**[_u16]_**m**(svuint16_t *inactive*, svbool_t *pg*, svuint16_t *op*)
svuint32_t **svcnot**[_u32]_**m**(svuint32_t *inactive*, svbool_t *pg*, svuint32_t *op*)

Instances
svuint64_t **svcnot**[_u64]_**m**(svuint64_t *inactive*, svbool_t *pg*, svuint64_t *op*)

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*)

Instances
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*)
svuint32_t **svlsl**[_u32]_**x**(svbool_t *pg*, svuint32_t *op1*, svuint32_t *op2*)
svuint64_t **svlsl**[_u64]_**x**(svbool_t *pg*, svuint64_t *op1*, svuint64_t *op2*)

6.9.1.4. LSL (vector, scalar), setting inactive to zero

Instances
svint64_t **svlsl**[_n_s64]_**z**(svbool_t *pg*, svint64_t *op1*, uint64_t *op2*)
svuint64_t **svlsl**[_n_u64]_**z**(svbool_t *pg*, svuint64_t *op1*, uint64_t *op2*)

6.9.1.5. LSL (vector, scalar), merging with first input

Instances
svint64_t **svlsl**[_n_s64]_**m**(svbool_t *pg*, svint64_t *op1*, uint64_t *op2*)
svuint64_t **svlsl**[_n_u64]_**m**(svbool_t *pg*, svuint64_t *op1*, uint64_t *op2*)

6.9.1.6. LSL (vector, scalar), setting inactive to unknown

Instances
svint64_t **svlsl**[_n_s64]_**x**(svbool_t *pg*, svint64_t *op1*, uint64_t *op2*)
svuint64_t **svlsl**[_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 **svlsl_wide**[_s8]_**z**(svbool_t *pg*, svint8_t *op1*, svuint64_t *op2*)
svint16_t **svlsl_wide**[_s16]_**z**(svbool_t *pg*, svint16_t *op1*, svuint64_t *op2*)
svint32_t **svlsl_wide**[_s32]_**z**(svbool_t *pg*, svint32_t *op1*, svuint64_t *op2*)
svuint8_t **svlsl_wide**[_u8]_**z**(svbool_t *pg*, svuint8_t *op1*, svuint64_t *op2*)
svuint16_t **svlsl_wide**[_u16]_**z**(svbool_t *pg*, svuint16_t *op1*, svuint64_t *op2*)
svuint32_t **svlsl_wide**[_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 **svlsl_wide**[_s8]_**m**(svbool_t *pg*, svint8_t *op1*, svuint64_t *op2*)
svint16_t **svlsl_wide**[_s16]_**m**(svbool_t *pg*, svint16_t *op1*, svuint64_t *op2*)
svint32_t **svlsl_wide**[_s32]_**m**(svbool_t *pg*, svint32_t *op1*, svuint64_t *op2*)
svuint8_t **svlsl_wide**[_u8]_**m**(svbool_t *pg*, svuint8_t *op1*, svuint64_t *op2*)
svuint16_t **svlsl_wide**[_u16]_**m**(svbool_t *pg*, svuint16_t *op1*, svuint64_t *op2*)
svuint32_t **svlsl_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 **svlsl_wide**[_s8]_**x**(svbool_t *pg*, svint8_t *op1*, svuint64_t *op2*)
svint16_t **svlsl_wide**[_s16]_**x**(svbool_t *pg*, svint16_t *op1*, svuint64_t *op2*)
svint32_t **svlsl_wide**[_s32]_**x**(svbool_t *pg*, svint32_t *op1*, svuint64_t *op2*)
svuint8_t **svlsl_wide**[_u8]_**x**(svbool_t *pg*, svuint8_t *op1*, svuint64_t *op2*)
svuint16_t **svlsl_wide**[_u16]_**x**(svbool_t *pg*, svuint16_t *op1*, svuint64_t *op2*)

Instances
svuint32_t **svlsl_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 **svlsl**[_n_s8]_**z**(svbool_t *pg*, svint8_t *op1*, uint64_t *op2*)
svint16_t **svlsl**[_n_s16]_**z**(svbool_t *pg*, svint16_t *op1*, uint64_t *op2*)
svint32_t **svlsl**[_n_s32]_**z**(svbool_t *pg*, svint32_t *op1*, uint64_t *op2*)
svuint8_t **svlsl**[_n_u8]_**z**(svbool_t *pg*, svuint8_t *op1*, uint64_t *op2*)
svuint16_t **svlsl**[_n_u16]_**z**(svbool_t *pg*, svuint16_t *op1*, uint64_t *op2*)
svuint32_t **svlsl**[_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 **svlsl**[_n_s8]_**m**(svbool_t *pg*, svint8_t *op1*, uint64_t *op2*)
svint16_t **svlsl**[_n_s16]_**m**(svbool_t *pg*, svint16_t *op1*, uint64_t *op2*)
svint32_t **svlsl**[_n_s32]_**m**(svbool_t *pg*, svint32_t *op1*, uint64_t *op2*)
svuint8_t **svlsl**[_n_u8]_**m**(svbool_t *pg*, svuint8_t *op1*, uint64_t *op2*)
svuint16_t **svlsl**[_n_u16]_**m**(svbool_t *pg*, svuint16_t *op1*, uint64_t *op2*)
svuint32_t **svlsl**[_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 **svlsl**[_n_s8]_**x**(svbool_t *pg*, svint8_t *op1*, uint64_t *op2*)
svint16_t **svlsl**[_n_s16]_**x**(svbool_t *pg*, svint16_t *op1*, uint64_t *op2*)
svint32_t **svlsl**[_n_s32]_**x**(svbool_t *pg*, svint32_t *op1*, uint64_t *op2*)
svuint8_t **svlsl**[_n_u8]_**x**(svbool_t *pg*, svuint8_t *op1*, uint64_t *op2*)
svuint16_t **svlsl**[_n_u16]_**x**(svbool_t *pg*, svuint16_t *op1*, uint64_t *op2*)
svuint32_t **svlsl**[_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 **svlsr**[_u8]_**z**(svbool_t *pg*, svuint8_t *op1*, svuint8_t *op2*)
svuint16_t **svlsr**[_u16]_**z**(svbool_t *pg*, svuint16_t *op1*, svuint16_t *op2*)
svuint32_t **svlsr**[_u32]_**z**(svbool_t *pg*, svuint32_t *op1*, svuint32_t *op2*)
svuint64_t **svlsr**[_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 **svlsr**[_u8]_**m**(svbool_t *pg*, svuint8_t *op1*, svuint8_t *op2*)
svuint16_t **svlsr**[_u16]_**m**(svbool_t *pg*, svuint16_t *op1*, svuint16_t *op2*)
svuint32_t **svlsr**[_u32]_**m**(svbool_t *pg*, svuint32_t *op1*, svuint32_t *op2*)

Instances
svuint64_t **svlsr**[_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 **svlsr**[_u8]_**x**(svbool_t *pg*, svuint8_t *op1*, svuint8_t *op2*) svuint16_t **svlsr**[_u16]_**x**(svbool_t *pg*, svuint16_t *op1*, svuint16_t *op2*) svuint32_t **svlsr**[_u32]_**x**(svbool_t *pg*, svuint32_t *op1*, svuint32_t *op2*) svuint64_t **svlsr**[_u64]_**x**(svbool_t *pg*, svuint64_t *op1*, svuint64_t *op2*)

6.9.2.4. LSR (vector, scalar), setting inactive to zero

Instances
svuint64_t **svlsr**[_n_u64]_**z**(svbool_t *pg*, svuint64_t *op1*, uint64_t *op2*)

6.9.2.5. LSR (vector, scalar), merging with first input

Instances
svuint64_t **svlsr**[_n_u64]_**m**(svbool_t *pg*, svuint64_t *op1*, uint64_t *op2*)

6.9.2.6. LSR (vector, scalar), setting inactive to unknown

Instances
svuint64_t **svlsr**[_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 **svlsr_wide**[_u8]_**z**(svbool_t *pg*, svuint8_t *op1*, svuint64_t *op2*) 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**[_n_u8]_**z**(svbool_t *pg*, svuint8_t *op1*, uint64_t *op2*)

Instances
svuint16_t **svlsr**[_n_u16]_**z**(svbool_t *pg*, svuint16_t *op1*, uint64_t *op2*)
svuint32_t **svlsr**[_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**[_n_u8]_**m**(svbool_t *pg*, svuint8_t *op1*, uint64_t *op2*)
svuint16_t **svlsr**[_n_u16]_**m**(svbool_t *pg*, svuint16_t *op1*, uint64_t *op2*)
svuint32_t **svlsr**[_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**[_n_u8]_**x**(svbool_t *pg*, svuint8_t *op1*, uint64_t *op2*)
svuint16_t **svlsr**[_n_u16]_**x**(svbool_t *pg*, svuint16_t *op1*, uint64_t *op2*)
svuint32_t **svlsr**[_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*)
svint16_t **svasr**[_s16]_**z**(svbool_t *pg*, svint16_t *op1*, svuint16_t *op2*)
svint32_t **svasr**[_s32]_**z**(svbool_t *pg*, svint32_t *op1*, svuint32_t *op2*)
svint64_t **svasr**[_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 **svasr**[_s8]_**m**(svbool_t *pg*, svint8_t *op1*, svuint8_t *op2*)
svint16_t **svasr**[_s16]_**m**(svbool_t *pg*, svint16_t *op1*, svuint16_t *op2*)
svint32_t **svasr**[_s32]_**m**(svbool_t *pg*, svint32_t *op1*, svuint32_t *op2*)
svint64_t **svasr**[_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 **svasr**[_s8]_**x**(svbool_t *pg*, svint8_t *op1*, svuint8_t *op2*)
svint16_t **svasr**[_s16]_**x**(svbool_t *pg*, svint16_t *op1*, svuint16_t *op2*)
svint32_t **svasr**[_s32]_**x**(svbool_t *pg*, svint32_t *op1*, svuint32_t *op2*)
svint64_t **svasr**[_s64]_**x**(svbool_t *pg*, svint64_t *op1*, svuint64_t *op2*)

6.9.3.4. ASR (vector, scalar), setting inactive to zero

Instances
svint64_t **svasr**[_n_s64]_**z**(svbool_t *pg*, svint64_t *op1*, uint64_t *op2*)

6.9.3.5. ASR (vector, scalar), merging with first input

Instances
svint64_t **svasr**[_n_s64]_**m**(svbool_t *pg*, svint64_t *op1*, uint64_t *op2*)

6.9.3.6. ASR (vector, scalar), setting inactive to unknown

Instances
svint64_t **svasr**[_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 **svasr_wide**[_s8]_**z**(svbool_t *pg*, svint8_t *op1*, svuint64_t *op2*) svint16_t **svasr_wide**[_s16]_**z**(svbool_t *pg*, svint16_t *op1*, svuint64_t *op2*) svint32_t **svasr_wide**[_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 **svasr_wide**[_s8]_**m**(svbool_t *pg*, svint8_t *op1*, svuint64_t *op2*) svint16_t **svasr_wide**[_s16]_**m**(svbool_t *pg*, svint16_t *op1*, svuint64_t *op2*) svint32_t **svasr_wide**[_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 **svasr_wide**[_s8]_**x**(svbool_t *pg*, svint8_t *op1*, svuint64_t *op2*) svint16_t **svasr_wide**[_s16]_**x**(svbool_t *pg*, svint16_t *op1*, svuint64_t *op2*) svint32_t **svasr_wide**[_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 **svasr**[_n_s8]_**z**(svbool_t *pg*, svint8_t *op1*, uint64_t *op2*) svint16_t **svasr**[_n_s16]_**z**(svbool_t *pg*, svint16_t *op1*, uint64_t *op2*) svint32_t **svasr**[_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 **svasr**[_n_s8]_**m**(svbool_t *pg*, svint8_t *op1*, uint64_t *op2*) svint16_t **svasr**[_n_s16]_**m**(svbool_t *pg*, svint16_t *op1*, uint64_t *op2*) svint32_t **svasr**[_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 **svasr**[_n_s8]_**x**(svbool_t *pg*, svint8_t *op1*, uint64_t *op2*)

Instances
svint16_t **svasr**[_n_s16]_**x**(svbool_t *pg*, svint16_t *op1*, uint64_t *op2*)
svint32_t **svasr**[_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 **svasrd**[_n_s8]_**z**(svbool_t *pg*, svint8_t *op1*, uint64_t *imm2*)
svint16_t **svasrd**[_n_s16]_**z**(svbool_t *pg*, svint16_t *op1*, uint64_t *imm2*)
svint32_t **svasrd**[_n_s32]_**z**(svbool_t *pg*, svint32_t *op1*, uint64_t *imm2*)
svint64_t **svasrd**[_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 **svasrd**[_n_s8]_**m**(svbool_t *pg*, svint8_t *op1*, uint64_t *imm2*)
svint16_t **svasrd**[_n_s16]_**m**(svbool_t *pg*, svint16_t *op1*, uint64_t *imm2*)
svint32_t **svasrd**[_n_s32]_**m**(svbool_t *pg*, svint32_t *op1*, uint64_t *imm2*)
svint64_t **svasrd**[_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 **svasrd**[_n_s8]_**x**(svbool_t *pg*, svint8_t *op1*, uint64_t *imm2*)
svint16_t **svasrd**[_n_s16]_**x**(svbool_t *pg*, svint16_t *op1*, uint64_t *imm2*)
svint32_t **svasrd**[_n_s32]_**x**(svbool_t *pg*, svint32_t *op1*, uint64_t *imm2*)
svint64_t **svasrd**[_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 **svinsr**[_n_s8](svint8_t *op1*, int8_t *op2*)
svint16_t **svinsr**[_n_s16](svint16_t *op1*, int16_t *op2*)
svint32_t **svinsr**[_n_s32](svint32_t *op1*, int32_t *op2*)
svint64_t **svinsr**[_n_s64](svint64_t *op1*, int64_t *op2*)
svuint8_t **svinsr**[_n_u8](svuint8_t *op1*, uint8_t *op2*)
svuint16_t **svinsr**[_n_u16](svuint16_t *op1*, uint16_t *op2*)
svuint32_t **svinsr**[_n_u32](svuint32_t *op1*, uint32_t *op2*)
svuint64_t **svinsr**[_n_u64](svuint64_t *op1*, uint64_t *op2*)
svfloat16_t **svinsr**[_n_f16](svfloat16_t *op1*, float16_t *op2*)
svfloat32_t **svinsr**[_n_f32](svfloat32_t *op1*, float32_t *op2*)

Instances
`svfloat64_t svinsr[_n_f64](svfloat64_t op1, float64_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 svaddv[_s8](svbool_t pg, svint8_t op)`
`int64_t svaddv[_s16](svbool_t pg, svint16_t op)`
`int64_t svaddv[_s32](svbool_t pg, svint32_t op)`
`int64_t svaddv[_s64](svbool_t pg, svint64_t op)`
`uint64_t svaddv[_u8](svbool_t pg, svuint8_t op)`
`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)`

Instances
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*)
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*)

Instances
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*)
svbool_t **svcmpeq**[_u64](svbool_t *pg*, svuint64_t *op1*, svuint64_t *op2*)

6.11.1.2. CMPEQ (vector, scalar)

Instances
svbool_t **svcmpeq**[_n_s64](svbool_t *pg*, svint64_t *op1*, int64_t *op2*)
svbool_t **svcmpeq**[_n_u64](svbool_t *pg*, svuint64_t *op1*, uint64_t *op2*)

6.11.1.3. CMPEQ (vector, wide vector)

Instances
svbool_t **svcmpeq_wide**[_s8](svbool_t *pg*, svint8_t *op1*, svint64_t *op2*)
svbool_t **svcmpeq_wide**[_s16](svbool_t *pg*, svint16_t *op1*, svint64_t *op2*)
svbool_t **svcmpeq_wide**[_s32](svbool_t *pg*, svint32_t *op1*, svint64_t *op2*)
svbool_t **svcmpeq_wide**[_u8](svbool_t *pg*, svuint8_t *op1*, svuint64_t *op2*)
svbool_t **svcmpeq_wide**[_u16](svbool_t *pg*, svuint16_t *op1*, svuint64_t *op2*)
svbool_t **svcmpeq_wide**[_u32](svbool_t *pg*, svuint32_t *op1*, svuint64_t *op2*)

6.11.1.4. CMPEQ (vector, wide scalar)

Instances
svbool_t **svcmpeq**[_n_s8](svbool_t *pg*, svint8_t *op1*, int64_t *op2*)
svbool_t **svcmpeq**[_n_s16](svbool_t *pg*, svint16_t *op1*, int64_t *op2*)
svbool_t **svcmpeq**[_n_s32](svbool_t *pg*, svint32_t *op1*, int64_t *op2*)
svbool_t **svcmpeq**[_n_u8](svbool_t *pg*, svuint8_t *op1*, uint64_t *op2*)
svbool_t **svcmpeq**[_n_u16](svbool_t *pg*, svuint16_t *op1*, uint64_t *op2*)
svbool_t **svcmpeq**[_n_u32](svbool_t *pg*, svuint32_t *op1*, uint64_t *op2*)

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 **svcmpne**[_s8](svbool_t *pg*, svint8_t *op1*, svint8_t *op2*)
svbool_t **svcmpne**[_s16](svbool_t *pg*, svint16_t *op1*, svint16_t *op2*)
svbool_t **svcmpne**[_s32](svbool_t *pg*, svint32_t *op1*, svint32_t *op2*)
svbool_t **svcmpne**[_s64](svbool_t *pg*, svint64_t *op1*, svint64_t *op2*)
svbool_t **svcmpne**[_u8](svbool_t *pg*, svuint8_t *op1*, svuint8_t *op2*)
svbool_t **svcmpne**[_u16](svbool_t *pg*, svuint16_t *op1*, svuint16_t *op2*)
svbool_t **svcmpne**[_u32](svbool_t *pg*, svuint32_t *op1*, svuint32_t *op2*)
svbool_t **svcmpne**[_u64](svbool_t *pg*, svuint64_t *op1*, svuint64_t *op2*)

6.11.2.2. CMPNE (vector, scalar)

Instances
svbool_t **svcmpne**[_n_s64](svbool_t *pg*, svint64_t *op1*, int64_t *op2*)
svbool_t **svcmpne**[_n_u64](svbool_t *pg*, svuint64_t *op1*, uint64_t *op2*)

6.11.2.3. CMPNE (vector, wide vector)

Instances
svbool_t **svcmpne_wide**[_s8](svbool_t *pg*, svint8_t *op1*, svint64_t *op2*)
svbool_t **svcmpne_wide**[_s16](svbool_t *pg*, svint16_t *op1*, svint64_t *op2*)
svbool_t **svcmpne_wide**[_s32](svbool_t *pg*, svint32_t *op1*, svint64_t *op2*)
svbool_t **svcmpne_wide**[_u8](svbool_t *pg*, svuint8_t *op1*, svuint64_t *op2*)
svbool_t **svcmpne_wide**[_u16](svbool_t *pg*, svuint16_t *op1*, svuint64_t *op2*)
svbool_t **svcmpne_wide**[_u32](svbool_t *pg*, svuint32_t *op1*, svuint64_t *op2*)

6.11.2.4. CMPNE (vector, wide scalar)

Instances
svbool_t **svcmpne**[_n_s8](svbool_t *pg*, svint8_t *op1*, int64_t *op2*)
svbool_t **svcmpne**[_n_s16](svbool_t *pg*, svint16_t *op1*, int64_t *op2*)
svbool_t **svcmpne**[_n_s32](svbool_t *pg*, svint32_t *op1*, int64_t *op2*)
svbool_t **svcmpne**[_n_u8](svbool_t *pg*, svuint8_t *op1*, uint64_t *op2*)
svbool_t **svcmpne**[_n_u16](svbool_t *pg*, svuint16_t *op1*, uint64_t *op2*)
svbool_t **svcmpne**[_n_u32](svbool_t *pg*, svuint32_t *op1*, uint64_t *op2*)

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 **svcmplt**[_s8](svbool_t *pg*, svint8_t *op1*, svint8_t *op2*)
svbool_t **svcmplt**[_s16](svbool_t *pg*, svint16_t *op1*, svint16_t *op2*)
svbool_t **svcmplt**[_s32](svbool_t *pg*, svint32_t *op1*, svint32_t *op2*)
svbool_t **svcmplt**[_s64](svbool_t *pg*, svint64_t *op1*, svint64_t *op2*)
svbool_t **svcmplt**[_u8](svbool_t *pg*, svuint8_t *op1*, svuint8_t *op2*)
svbool_t **svcmplt**[_u16](svbool_t *pg*, svuint16_t *op1*, svuint16_t *op2*)

Instances
svbool_t **svcmplt**[_u32](svbool_t *pg*, svuint32_t *op1*, svuint32_t *op2*)
svbool_t **svcmplt**[_u64](svbool_t *pg*, svuint64_t *op1*, svuint64_t *op2*)

6.11.3.2. CMPLT (vector, scalar)

Instances
svbool_t **svcmplt**[_n_s64](svbool_t *pg*, svint64_t *op1*, int64_t *op2*)
svbool_t **svcmplt**[_n_u64](svbool_t *pg*, svuint64_t *op1*, uint64_t *op2*)

6.11.3.3. CMPLT (vector, wide vector)

Instances
svbool_t **svcmplt_wide**[_s8](svbool_t *pg*, svint8_t *op1*, svint64_t *op2*)
svbool_t **svcmplt_wide**[_s16](svbool_t *pg*, svint16_t *op1*, svint64_t *op2*)
svbool_t **svcmplt_wide**[_s32](svbool_t *pg*, svint32_t *op1*, svint64_t *op2*)
svbool_t **svcmplt_wide**[_u8](svbool_t *pg*, svuint8_t *op1*, svuint64_t *op2*)
svbool_t **svcmplt_wide**[_u16](svbool_t *pg*, svuint16_t *op1*, svuint64_t *op2*)
svbool_t **svcmplt_wide**[_u32](svbool_t *pg*, svuint32_t *op1*, svuint64_t *op2*)

6.11.3.4. CMPLT (vector, wide scalar)

Instances
svbool_t **svcmplt**[_n_s8](svbool_t *pg*, svint8_t *op1*, int64_t *op2*)
svbool_t **svcmplt**[_n_s16](svbool_t *pg*, svint16_t *op1*, int64_t *op2*)
svbool_t **svcmplt**[_n_s32](svbool_t *pg*, svint32_t *op1*, int64_t *op2*)
svbool_t **svcmplt**[_n_u8](svbool_t *pg*, svuint8_t *op1*, uint64_t *op2*)
svbool_t **svcmplt**[_n_u16](svbool_t *pg*, svuint16_t *op1*, uint64_t *op2*)
svbool_t **svcmplt**[_n_u32](svbool_t *pg*, svuint32_t *op1*, uint64_t *op2*)

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 **svcmple**[_s8](svbool_t *pg*, svint8_t *op1*, svint8_t *op2*)
svbool_t **svcmple**[_s16](svbool_t *pg*, svint16_t *op1*, svint16_t *op2*)
svbool_t **svcmple**[_s32](svbool_t *pg*, svint32_t *op1*, svint32_t *op2*)
svbool_t **svcmple**[_s64](svbool_t *pg*, svint64_t *op1*, svint64_t *op2*)
svbool_t **svcmple**[_u8](svbool_t *pg*, svuint8_t *op1*, svuint8_t *op2*)
svbool_t **svcmple**[_u16](svbool_t *pg*, svuint16_t *op1*, svuint16_t *op2*)
svbool_t **svcmple**[_u32](svbool_t *pg*, svuint32_t *op1*, svuint32_t *op2*)
svbool_t **svcmple**[_u64](svbool_t *pg*, svuint64_t *op1*, svuint64_t *op2*)

6.11.4.2. CMPLE (vector, scalar)

Instances
svbool_t **svcmple**[_n_s64](svbool_t *pg*, svint64_t *op1*, int64_t *op2*)
svbool_t **svcmple**[_n_u64](svbool_t *pg*, svuint64_t *op1*, uint64_t *op2*)

6.11.4.3. CMPLE (vector, wide vector)

Instances
svbool_t **svcmple_wide**[_s8](svbool_t *pg*, svint8_t *op1*, svint64_t *op2*)
svbool_t **svcmple_wide**[_s16](svbool_t *pg*, svint16_t *op1*, svint64_t *op2*)
svbool_t **svcmple_wide**[_s32](svbool_t *pg*, svint32_t *op1*, svint64_t *op2*)
svbool_t **svcmple_wide**[_u8](svbool_t *pg*, svuint8_t *op1*, svuint64_t *op2*)
svbool_t **svcmple_wide**[_u16](svbool_t *pg*, svuint16_t *op1*, svuint64_t *op2*)
svbool_t **svcmple_wide**[_u32](svbool_t *pg*, svuint32_t *op1*, svuint64_t *op2*)

6.11.4.4. CMPLE (vector, wide scalar)

Instances
svbool_t **svcmple**[_n_s8](svbool_t *pg*, svint8_t *op1*, int64_t *op2*)
svbool_t **svcmple**[_n_s16](svbool_t *pg*, svint16_t *op1*, int64_t *op2*)
svbool_t **svcmple**[_n_s32](svbool_t *pg*, svint32_t *op1*, int64_t *op2*)
svbool_t **svcmple**[_n_u8](svbool_t *pg*, svuint8_t *op1*, uint64_t *op2*)
svbool_t **svcmple**[_n_u16](svbool_t *pg*, svuint16_t *op1*, uint64_t *op2*)
svbool_t **svcmple**[_n_u32](svbool_t *pg*, svuint32_t *op1*, uint64_t *op2*)

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 **svcmpge**[_s8](svbool_t *pg*, svint8_t *op1*, svint8_t *op2*)
svbool_t **svcmpge**[_s16](svbool_t *pg*, svint16_t *op1*, svint16_t *op2*)
svbool_t **svcmpge**[_s32](svbool_t *pg*, svint32_t *op1*, svint32_t *op2*)
svbool_t **svcmpge**[_s64](svbool_t *pg*, svint64_t *op1*, svint64_t *op2*)
svbool_t **svcmpge**[_u8](svbool_t *pg*, svuint8_t *op1*, svuint8_t *op2*)
svbool_t **svcmpge**[_u16](svbool_t *pg*, svuint16_t *op1*, svuint16_t *op2*)
svbool_t **svcmpge**[_u32](svbool_t *pg*, svuint32_t *op1*, svuint32_t *op2*)
svbool_t **svcmpge**[_u64](svbool_t *pg*, svuint64_t *op1*, svuint64_t *op2*)

6.11.5.2. CMPGE (vector, scalar)

Instances
svbool_t **svcmpge**[_n_s64](svbool_t *pg*, svint64_t *op1*, int64_t *op2*)
svbool_t **svcmpge**[_n_u64](svbool_t *pg*, svuint64_t *op1*, uint64_t *op2*)

6.11.5.3. CMPGE (vector, wide vector)

Instances
svbool_t **svcmpge_wide**[_s8](svbool_t *pg*, svint8_t *op1*, svint64_t *op2*)
svbool_t **svcmpge_wide**[_s16](svbool_t *pg*, svint16_t *op1*, svint64_t *op2*)
svbool_t **svcmpge_wide**[_s32](svbool_t *pg*, svint32_t *op1*, svint64_t *op2*)
svbool_t **svcmpge_wide**[_u8](svbool_t *pg*, svuint8_t *op1*, svuint64_t *op2*)
svbool_t **svcmpge_wide**[_u16](svbool_t *pg*, svuint16_t *op1*, svuint64_t *op2*)
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**[_n_s8](svbool_t *pg*, svint8_t *op1*, int64_t *op2*)
svbool_t **svcmpge**[_n_s16](svbool_t *pg*, svint16_t *op1*, int64_t *op2*)
svbool_t **svcmpge**[_n_s32](svbool_t *pg*, svint32_t *op1*, int64_t *op2*)
svbool_t **svcmpge**[_n_u8](svbool_t *pg*, svuint8_t *op1*, uint64_t *op2*)
svbool_t **svcmpge**[_n_u16](svbool_t *pg*, svuint16_t *op1*, uint64_t *op2*)
svbool_t **svcmpge**[_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_s64](svbool_t *pg*, svint64_t *op1*, int64_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 **svcmpgt**[_n_s8](svbool_t *pg*, svint8_t *op1*, int64_t *op2*)
svbool_t **svcmpgt**[_n_s16](svbool_t *pg*, svint16_t *op1*, int64_t *op2*)
svbool_t **svcmpgt**[_n_s32](svbool_t *pg*, svint32_t *op1*, int64_t *op2*)
svbool_t **svcmpgt**[_n_u8](svbool_t *pg*, svuint8_t *op1*, uint64_t *op2*)
svbool_t **svcmpgt**[_n_u16](svbool_t *pg*, svuint16_t *op1*, uint64_t *op2*)
svbool_t **svcmpgt**[_n_u32](svbool_t *pg*, svuint32_t *op1*, uint64_t *op2*)

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*×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 **svwhilelt_b8**[_s32](int32_t *op1*, int32_t *op2*)
svbool_t **svwhilelt_b16**[_s32](int32_t *op1*, int32_t *op2*)
svbool_t **svwhilelt_b32**[_s32](int32_t *op1*, int32_t *op2*)
svbool_t **svwhilelt_b64**[_s32](int32_t *op1*, int32_t *op2*)
svbool_t **svwhilelt_b8**[_u32](uint32_t *op1*, uint32_t *op2*)
svbool_t **svwhilelt_b16**[_u32](uint32_t *op1*, uint32_t *op2*)
svbool_t **svwhilelt_b32**[_u32](uint32_t *op1*, uint32_t *op2*)
svbool_t **svwhilelt_b64**[_u32](uint32_t *op1*, uint32_t *op2*)
svbool_t **svwhilelt_b8**[_s64](int64_t *op1*, int64_t *op2*)
svbool_t **svwhilelt_b16**[_s64](int64_t *op1*, int64_t *op2*)
svbool_t **svwhilelt_b32**[_s64](int64_t *op1*, int64_t *op2*)
svbool_t **svwhilelt_b64**[_s64](int64_t *op1*, int64_t *op2*)
svbool_t **svwhilelt_b8**[_u64](uint64_t *op1*, uint64_t *op2*)
svbool_t **svwhilelt_b16**[_u64](uint64_t *op1*, uint64_t *op2*)
svbool_t **svwhilelt_b32**[_u64](uint64_t *op1*, uint64_t *op2*)
svbool_t **svwhilelt_b64**[_u64](uint64_t *op1*, uint64_t *op2*)

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 **svwhilele_b8**[_s32](int32_t *op1*, int32_t *op2*)

Instances
svbool_t **svwhilele_b16**[_s32](int32_t *op1*, int32_t *op2*)
svbool_t **svwhilele_b32**[_s32](int32_t *op1*, int32_t *op2*)
svbool_t **svwhilele_b64**[_s32](int32_t *op1*, int32_t *op2*)
svbool_t **svwhilele_b8**[_u32](uint32_t *op1*, uint32_t *op2*)
svbool_t **svwhilele_b16**[_u32](uint32_t *op1*, uint32_t *op2*)
svbool_t **svwhilele_b32**[_u32](uint32_t *op1*, uint32_t *op2*)
svbool_t **svwhilele_b64**[_u32](uint32_t *op1*, uint32_t *op2*)
svbool_t **svwhilele_b8**[_s64](int64_t *op1*, int64_t *op2*)
svbool_t **svwhilele_b16**[_s64](int64_t *op1*, int64_t *op2*)
svbool_t **svwhilele_b32**[_s64](int64_t *op1*, int64_t *op2*)
svbool_t **svwhilele_b64**[_s64](int64_t *op1*, int64_t *op2*)
svbool_t **svwhilele_b8**[_u64](uint64_t *op1*, uint64_t *op2*)
svbool_t **svwhilele_b16**[_u64](uint64_t *op1*, uint64_t *op2*)
svbool_t **svwhilele_b32**[_u64](uint64_t *op1*, uint64_t *op2*)
svbool_t **svwhilele_b64**[_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 **svcls**[_s8]_**z**(svbool_t *pg*, svint8_t *op*)
svuint16_t **svcls**[_s16]_**z**(svbool_t *pg*, svint16_t *op*)
svuint32_t **svcls**[_s32]_**z**(svbool_t *pg*, svint32_t *op*)
svuint64_t **svcls**[_s64]_**z**(svbool_t *pg*, svint64_t *op*)

6.13.1.2. CLS (vector), merging with separate vector

Instances
svuint8_t **svcls**[_s8]_**m**(svuint8_t *inactive*, svbool_t *pg*, svint8_t *op*)
svuint16_t **svcls**[_s16]_**m**(svuint16_t *inactive*, svbool_t *pg*, svint16_t *op*)
svuint32_t **svcls**[_s32]_**m**(svuint32_t *inactive*, svbool_t *pg*, svint32_t *op*)
svuint64_t **svcls**[_s64]_**m**(svuint64_t *inactive*, svbool_t *pg*, svint64_t *op*)

6.13.1.3. CLS (vector), setting inactive to unknown

Instances
svuint8_t **svcls**[_s8]_**x**(svbool_t *pg*, svint8_t *op*)
svuint16_t **svcls**[_s16]_**x**(svbool_t *pg*, svint16_t *op*)
svuint32_t **svcls**[_s32]_**x**(svbool_t *pg*, svint32_t *op*)
svuint64_t **svcls**[_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
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*)

6.13.2.2. CLZ (vector), merging with separate vector

Instances
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*)

6.13.2.3. CLZ (vector), setting inactive to unknown

Instances
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*)

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
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*)
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*)

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*)

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*)

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*)
svint64_t **svextb**[_s64]_**z**(svbool_t *pg*, svint64_t *op*)
svuint16_t **svextb**[_u16]_**z**(svbool_t *pg*, svuint16_t *op*)
svuint32_t **svextb**[_u32]_**z**(svbool_t *pg*, svuint32_t *op*)
svuint64_t **svextb**[_u64]_**z**(svbool_t *pg*, svuint64_t *op*)

6.14.1.2. EXTB (vector), merging with separate vector

Instances
svint16_t **svextb**[_s16]_**m**(svint16_t *inactive*, svbool_t *pg*, svint16_t *op*)
svint32_t **svextb**[_s32]_**m**(svint32_t *inactive*, svbool_t *pg*, svint32_t *op*)
svint64_t **svextb**[_s64]_**m**(svint64_t *inactive*, svbool_t *pg*, svint64_t *op*)
svuint16_t **svextb**[_u16]_**m**(svuint16_t *inactive*, svbool_t *pg*, svuint16_t *op*)

Instances
svuint32_t **svextb**[_u32]_**m**(svuint32_t *inactive*, svbool_t *pg*, svuint32_t *op*)
svuint64_t **svextb**[_u64]_**m**(svuint64_t *inactive*, svbool_t *pg*, svuint64_t *op*)

6.14.1.3. EXTB (vector), setting inactive to unknown

Instances
svint16_t **svextb**[_s16]_**x**(svbool_t *pg*, svint16_t *op*)
svint32_t **svextb**[_s32]_**x**(svbool_t *pg*, svint32_t *op*)
svint64_t **svextb**[_s64]_**x**(svbool_t *pg*, svint64_t *op*)
svuint16_t **svextb**[_u16]_**x**(svbool_t *pg*, svuint16_t *op*)
svuint32_t **svextb**[_u32]_**x**(svbool_t *pg*, svuint32_t *op*)
svuint64_t **svextb**[_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 **svexth**[_s32]_**z**(svbool_t *pg*, svint32_t *op*)
svint64_t **svexth**[_s64]_**z**(svbool_t *pg*, svint64_t *op*)
svuint32_t **svexth**[_u32]_**z**(svbool_t *pg*, svuint32_t *op*)
svuint64_t **svexth**[_u64]_**z**(svbool_t *pg*, svuint64_t *op*)

6.14.2.2. EXTH (vector), merging with separate vector

Instances
svint32_t **svexth**[_s32]_**m**(svint32_t *inactive*, svbool_t *pg*, svint32_t *op*)
svint64_t **svexth**[_s64]_**m**(svint64_t *inactive*, svbool_t *pg*, svint64_t *op*)
svuint32_t **svexth**[_u32]_**m**(svuint32_t *inactive*, svbool_t *pg*, svuint32_t *op*)
svuint64_t **svexth**[_u64]_**m**(svuint64_t *inactive*, svbool_t *pg*, svuint64_t *op*)

6.14.2.3. EXTH (vector), setting inactive to unknown

Instances
svint32_t **svexth**[_s32]_**x**(svbool_t *pg*, svint32_t *op*)
svint64_t **svexth**[_s64]_**x**(svbool_t *pg*, svint64_t *op*)
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*)
svint64_t **svrbit**[_s64]_**m**(svint64_t *inactive*, svbool_t *pg*, svint64_t *op*)
svuint8_t **svrbit**[_u8]_**m**(svuint8_t *inactive*, svbool_t *pg*, svuint8_t *op*)
svuint16_t **svrbit**[_u16]_**m**(svuint16_t *inactive*, svbool_t *pg*, svuint16_t *op*)
svuint32_t **svrbit**[_u32]_**m**(svuint32_t *inactive*, svbool_t *pg*, svuint32_t *op*)
svuint64_t **svrbit**[_u64]_**m**(svuint64_t *inactive*, svbool_t *pg*, svuint64_t *op*)

6.15.1.3. RBIT (vector), setting inactive to unknown

Instances
svint8_t **svrbit**[_s8]_**x**(svbool_t *pg*, svint8_t *op*)
svint16_t **svrbit**[_s16]_**x**(svbool_t *pg*, svint16_t *op*)
svint32_t **svrbit**[_s32]_**x**(svbool_t *pg*, svint32_t *op*)
svint64_t **svrbit**[_s64]_**x**(svbool_t *pg*, svint64_t *op*)
svuint8_t **svrbit**[_u8]_**x**(svbool_t *pg*, svuint8_t *op*)
svuint16_t **svrbit**[_u16]_**x**(svbool_t *pg*, svuint16_t *op*)
svuint32_t **svrbit**[_u32]_**x**(svbool_t *pg*, svuint32_t *op*)
svuint64_t **svrbit**[_u64]_**x**(svbool_t *pg*, svuint64_t *op*)

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 svrevb[_s16]_z(svbool_t pg, svint16_t op)`
`svint32_t svrevb[_s32]_z(svbool_t pg, svint32_t op)`
`svint64_t svrevb[_s64]_z(svbool_t pg, svint64_t op)`
`svuint16_t svrevb[_u16]_z(svbool_t pg, svuint16_t op)`
`svuint32_t svrevb[_u32]_z(svbool_t pg, svuint32_t op)`
`svuint64_t svrevb[_u64]_z(svbool_t pg, svuint64_t op)`

6.15.2.2. REVB (vector), merging with separate vector

Instances
`svint16_t svrevb[_s16]_m(svint16_t inactive, svbool_t pg, svint16_t op)`
`svint32_t svrevb[_s32]_m(svint32_t inactive, svbool_t pg, svint32_t op)`
`svint64_t svrevb[_s64]_m(svint64_t inactive, svbool_t pg, svint64_t op)`
`svuint16_t svrevb[_u16]_m(svuint16_t inactive, svbool_t pg, svuint16_t op)`
`svuint32_t svrevb[_u32]_m(svuint32_t inactive, svbool_t pg, svuint32_t op)`
`svuint64_t svrevb[_u64]_m(svuint64_t inactive, svbool_t pg, svuint64_t op)`

6.15.2.3. REVB (vector), setting inactive to unknown

Instances
`svint16_t svrevb[_s16]_x(svbool_t pg, svint16_t op)`
`svint32_t svrevb[_s32]_x(svbool_t pg, svint32_t op)`
`svint64_t svrevb[_s64]_x(svbool_t pg, svint64_t op)`
`svuint16_t svrevb[_u16]_x(svbool_t pg, svuint16_t op)`
`svuint32_t svrevb[_u32]_x(svbool_t pg, svuint32_t op)`
`svuint64_t svrevb[_u64]_x(svbool_t pg, svuint64_t op)`

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)`

Instances
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*) svuint64_t **svrevw**[_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.

Signalling 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 **svadd**[_f16]_**z**(svbool_t *pg*, svfloat16_t *op1*, svfloat16_t *op2*) svfloat32_t **svadd**[_f32]_**z**(svbool_t *pg*, svfloat32_t *op1*, svfloat32_t *op2*) svfloat64_t **svadd**[_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 **svadd**[_f16]_**m**(svbool_t *pg*, svfloat16_t *op1*, svfloat16_t *op2*) svfloat32_t **svadd**[_f32]_**m**(svbool_t *pg*, svfloat32_t *op1*, svfloat32_t *op2*) svfloat64_t **svadd**[_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 **svadd**[_f16]_**x**(svbool_t *pg*, svfloat16_t *op1*, svfloat16_t *op2*) svfloat32_t **svadd**[_f32]_**x**(svbool_t *pg*, svfloat32_t *op1*, svfloat32_t *op2*) svfloat64_t **svadd**[_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 **svadd**[_n_f16]_**z**(svbool_t *pg*, svfloat16_t *op1*, float16_t *op2*) svfloat32_t **svadd**[_n_f32]_**z**(svbool_t *pg*, svfloat32_t *op1*, float32_t *op2*) svfloat64_t **svadd**[_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 **svadd**[_n_f16]_**m**(svbool_t *pg*, svfloat16_t *op1*, float16_t *op2*) svfloat32_t **svadd**[_n_f32]_**m**(svbool_t *pg*, svfloat32_t *op1*, float32_t *op2*) svfloat64_t **svadd**[_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 **svadd**[_n_f16]_**x**(svbool_t *pg*, svfloat16_t *op1*, float16_t *op2*) svfloat32_t **svadd**[_n_f32]_**x**(svbool_t *pg*, svfloat32_t *op1*, float32_t *op2*) 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.

Signalling 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*,

Instances
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.

Signalling 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*) svfloat32_t **svsub**[_f32]_**z**(svbool_t *pg*, svfloat32_t *op1*, svfloat32_t *op2*) svfloat64_t **svsub**[_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 **svsub**[_f16]_**m**(svbool_t *pg*, svfloat16_t *op1*, svfloat16_t *op2*) svfloat32_t **svsub**[_f32]_**m**(svbool_t *pg*, svfloat32_t *op1*, svfloat32_t *op2*) svfloat64_t **svsub**[_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 **svsub**[_f16]_**x**(svbool_t *pg*, svfloat16_t *op1*, svfloat16_t *op2*) svfloat32_t **svsub**[_f32]_**x**(svbool_t *pg*, svfloat32_t *op1*, svfloat32_t *op2*) svfloat64_t **svsub**[_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 **svsub**[_n_f16]_**z**(svbool_t *pg*, svfloat16_t *op1*, float16_t *op2*)
svfloat32_t **svsub**[_n_f32]_**z**(svbool_t *pg*, svfloat32_t *op1*, float32_t *op2*)
svfloat64_t **svsub**[_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 **svsub**[_n_f16]_**m**(svbool_t *pg*, svfloat16_t *op1*, float16_t *op2*)
svfloat32_t **svsub**[_n_f32]_**m**(svbool_t *pg*, svfloat32_t *op1*, float32_t *op2*)
svfloat64_t **svsub**[_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 **svsub**[_n_f16]_**x**(svbool_t *pg*, svfloat16_t *op1*, float16_t *op2*)
svfloat32_t **svsub**[_n_f32]_**x**(svbool_t *pg*, svfloat32_t *op1*, float32_t *op2*)
svfloat64_t **svsub**[_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.

Signalling 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 **svsubr**[_f16]_**z**(svbool_t *pg*, svfloat16_t *op1*, svfloat16_t *op2*)
svfloat32_t **svsubr**[_f32]_**z**(svbool_t *pg*, svfloat32_t *op1*, svfloat32_t *op2*)
svfloat64_t **svsubr**[_f64]_**z**(svbool_t *pg*, svfloat64_t *op1*, svfloat64_t *op2*)

6.16.4.2. SUBR (vector, vector), merging with first input

Instances
svfloat16_t **svsubr**[_f16]_**m**(svbool_t *pg*, svfloat16_t *op1*, svfloat16_t *op2*)
svfloat32_t **svsubr**[_f32]_**m**(svbool_t *pg*, svfloat32_t *op1*, svfloat32_t *op2*)
svfloat64_t **svsubr**[_f64]_**m**(svbool_t *pg*, svfloat64_t *op1*, svfloat64_t *op2*)

6.16.4.3. SUBR (vector, vector), setting inactive to unknown

Instances
svfloat16_t **svsubr**[_f16]_**x**(svbool_t *pg*, svfloat16_t *op1*, svfloat16_t *op2*)
svfloat32_t **svsubr**[_f32]_**x**(svbool_t *pg*, svfloat32_t *op1*, svfloat32_t *op2*)
svfloat64_t **svsubr**[_f64]_**x**(svbool_t *pg*, svfloat64_t *op1*, svfloat64_t *op2*)

6.16.4.4. SUBR (vector, scalar), setting inactive to zero

Instances
svfloat16_t **svsubr**[_n_f16]_**z**(svbool_t *pg*, svfloat16_t *op1*, float16_t *op2*)

Instances
svfloat32_t **svsubr**[_n_f32]_**z**(svbool_t *pg*, svfloat32_t *op1*, float32_t *op2*)
svfloat64_t **svsubr**[_n_f64]_**z**(svbool_t *pg*, svfloat64_t *op1*, float64_t *op2*)

6.16.4.5. SUBR (vector, scalar), merging with first input

Instances
svfloat16_t **svsubr**[_n_f16]_**m**(svbool_t *pg*, svfloat16_t *op1*, float16_t *op2*)
svfloat32_t **svsubr**[_n_f32]_**m**(svbool_t *pg*, svfloat32_t *op1*, float32_t *op2*)
svfloat64_t **svsubr**[_n_f64]_**m**(svbool_t *pg*, svfloat64_t *op1*, float64_t *op2*)

6.16.4.6. SUBR (vector, scalar), setting inactive to unknown

Instances
svfloat16_t **svsubr**[_n_f16]_**x**(svbool_t *pg*, svfloat16_t *op1*, float16_t *op2*)
svfloat32_t **svsubr**[_n_f32]_**x**(svbool_t *pg*, svfloat32_t *op1*, float32_t *op2*)
svfloat64_t **svsubr**[_n_f64]_**x**(svbool_t *pg*, svfloat64_t *op1*, float64_t *op2*)

6.16.5. ABD: Floating-point absolute difference

These functions compute the absolute difference of two floating-point inputs.

Signalling 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 **svabd**[_f16]_**z**(svbool_t *pg*, svfloat16_t *op1*, svfloat16_t *op2*)
svfloat32_t **svabd**[_f32]_**z**(svbool_t *pg*, svfloat32_t *op1*, svfloat32_t *op2*)
svfloat64_t **svabd**[_f64]_**z**(svbool_t *pg*, svfloat64_t *op1*, svfloat64_t *op2*)

6.16.5.2. ABD (vector, vector), merging with first input

Instances
svfloat16_t **svabd**[_f16]_**m**(svbool_t *pg*, svfloat16_t *op1*, svfloat16_t *op2*)
svfloat32_t **svabd**[_f32]_**m**(svbool_t *pg*, svfloat32_t *op1*, svfloat32_t *op2*)
svfloat64_t **svabd**[_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 **svabd**[_f16]_**x**(svbool_t *pg*, svfloat16_t *op1*, svfloat16_t *op2*)
svfloat32_t **svabd**[_f32]_**x**(svbool_t *pg*, svfloat32_t *op1*, svfloat32_t *op2*)
svfloat64_t **svabd**[_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 **svabd**[_n_f16]_**z**(svbool_t *pg*, svfloat16_t *op1*, float16_t *op2*)
svfloat32_t **svabd**[_n_f32]_**z**(svbool_t *pg*, svfloat32_t *op1*, float32_t *op2*)
svfloat64_t **svabd**[_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 **svabd**[_n_f16]_**m**(svbool_t *pg*, svfloat16_t *op1*, float16_t *op2*)
svfloat32_t **svabd**[_n_f32]_**m**(svbool_t *pg*, svfloat32_t *op1*, float32_t *op2*)
svfloat64_t **svabd**[_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 **svabd**[_n_f16]_**x**(svbool_t *pg*, svfloat16_t *op1*, float16_t *op2*)
svfloat32_t **svabd**[_n_f32]_**x**(svbool_t *pg*, svfloat32_t *op1*, float32_t *op2*)
svfloat64_t **svabd**[_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 *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.

Signalling 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 **svmul**[_f16]_**z**(svbool_t *pg*, svfloat16_t *op1*, svfloat16_t *op2*)
svfloat32_t **svmul**[_f32]_**z**(svbool_t *pg*, svfloat32_t *op1*, svfloat32_t *op2*)
svfloat64_t **svmul**[_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*) svfloat64_t **svmulx**[_f64]_**z**(svbool_t *pg*, svfloat64_t *op1*, svfloat64_t *op2*)

6.16.7.2. MULX (vector, vector), merging with first input

Instances
svfloat16_t **svmulx**[_f16]_**m**(svbool_t *pg*, svfloat16_t *op1*, svfloat16_t *op2*) svfloat32_t **svmulx**[_f32]_**m**(svbool_t *pg*, svfloat32_t *op1*, svfloat32_t *op2*) svfloat64_t **svmulx**[_f64]_**m**(svbool_t *pg*, svfloat64_t *op1*, svfloat64_t *op2*)

6.16.7.3. MULX (vector, vector), setting inactive to unknown

Instances
svfloat16_t **svmulx**[_f16]_**x**(svbool_t *pg*, svfloat16_t *op1*, svfloat16_t *op2*) svfloat32_t **svmulx**[_f32]_**x**(svbool_t *pg*, svfloat32_t *op1*, svfloat32_t *op2*) svfloat64_t **svmulx**[_f64]_**x**(svbool_t *pg*, svfloat64_t *op1*, svfloat64_t *op2*)

6.16.7.4. MULX (vector, scalar), setting inactive to zero

Instances
svfloat16_t **svmulx**[_n_f16]_**z**(svbool_t *pg*, svfloat16_t *op1*, float16_t *op2*) svfloat32_t **svmulx**[_n_f32]_**z**(svbool_t *pg*, svfloat32_t *op1*, float32_t *op2*)

Instances
`svfloat64_t svmulx[_n_f64]_z(svbool_t pg, svfloat64_t op1, float64_t op2)`

6.16.7.5. MULX (vector, scalar), merging with first input

Instances
`svfloat16_t svmulx[_n_f16]_m(svbool_t pg, svfloat16_t op1, float16_t op2)`
`svfloat32_t svmulx[_n_f32]_m(svbool_t pg, svfloat32_t op1, float32_t op2)`
`svfloat64_t svmulx[_n_f64]_m(svbool_t pg, svfloat64_t op1, float64_t op2)`

6.16.7.6. MULX (vector, scalar), setting inactive to unknown

Instances
`svfloat16_t svmulx[_n_f16]_x(svbool_t pg, svfloat16_t op1, float16_t op2)`
`svfloat32_t svmulx[_n_f32]_x(svbool_t pg, svfloat32_t op1, float32_t op2)`
`svfloat64_t svmulx[_n_f64]_x(svbool_t pg, svfloat64_t op1, float64_t op2)`

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.

Signalling 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 svmad[_f16]_z(svbool_t pg, svfloat16_t op1, svfloat16_t op2, 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,`

Instances
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*) 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 *`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.

Signalling 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*,

Instances
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*) 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 *`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.

Signalling 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 svcmla[_f16]_z(svbool_t pg, svfloat16_t op1, svfloat16_t op2, svfloat16_t op3, uint64_t imm_rotation)`
`svfloat32_t svcmla[_f32]_z(svbool_t pg, svfloat32_t op1, svfloat32_t op2, svfloat32_t op3, uint64_t imm_rotation)`
`svfloat64_t svcmla[_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 svcmla[_f16]_m(svbool_t pg, svfloat16_t op1, svfloat16_t op2, svfloat16_t op3, uint64_t imm_rotation)`
`svfloat32_t svcmla[_f32]_m(svbool_t pg, svfloat32_t op1, svfloat32_t op2, svfloat32_t op3, uint64_t imm_rotation)`
`svfloat64_t svcmla[_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 svcmla[_f16]_x(svbool_t pg, svfloat16_t op1, svfloat16_t op2,`

Instances

`                          svfloat16_t op3, uint64_t imm_rotation)`
`svfloat32_t svcmla[_f32]_x(svbool_t pg, svfloat32_t op1, svfloat32_t op2,`
`                          svfloat32_t op3, uint64_t imm_rotation)`
`svfloat64_t svcmla[_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 svcmla_lane[_f16](svfloat16_t op1, svfloat16_t op2,`
`                            svfloat16_t op3, uint64_t imm_index,`
`                            uint64_t imm_rotation)`
`svfloat32_t svcmla_lane[_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 (multiplicand 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.

Signalling 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 svmsb[_f16]_z(svbool_t pg, svfloat16_t op1, svfloat16_t op2,`
`                           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,`

Instances
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*, float32_t *op3*) svfloat64_t **svmsb**[_n_f64]_**x**(svbool_t *pg*, svfloat64_t *op1*, svfloat64_t *op2*, float64_t *op3*)

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 *`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.

Signalling 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
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*,

Instances
svfloat32_t *op3*) svfloat64_t **svmls**[_f64]_**z**(svbool_t *pg*, svfloat64_t *op1*, svfloat64_t *op2*, svfloat64_t *op3*)

6.16.12.2. MLS (vector, vector, vector), merging with first input

Instances
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*)

6.16.12.3. MLS (vector, vector, vector), setting inactive to unknown

Instances
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*)

6.16.12.4. MLS (vector, vector, scalar), setting inactive to zero

Instances
svfloat16_t **svmls**[_n_f16]_**z**(svbool_t *pg*, svfloat16_t *op1*, svfloat16_t *op2*, 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*)

6.16.12.5. MLS (vector, vector, scalar), merging with first input

Instances
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*)

6.16.12.6. MLS (vector, vector, scalar), setting inactive to unknown

Instances
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*)

6.16.12.7. MLS (vector, vector, vector, lane)

Instances
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*)

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.

Signalling 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
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*) svfloat64_t **svnmad**[_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 **svnmad**[_f16]_**m**(svbool_t *pg*, svfloat16_t *op1*, svfloat16_t *op2*, svfloat16_t *op3*) svfloat32_t **svnmad**[_f32]_**m**(svbool_t *pg*, svfloat32_t *op1*, svfloat32_t *op2*, svfloat32_t *op3*) svfloat64_t **svnmad**[_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 **svnmad**[_f16]_**x**(svbool_t *pg*, svfloat16_t *op1*, svfloat16_t *op2*, svfloat16_t *op3*) svfloat32_t **svnmad**[_f32]_**x**(svbool_t *pg*, svfloat32_t *op1*, svfloat32_t *op2*, svfloat32_t *op3*) svfloat64_t **svnmad**[_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 **svnmad**[_n_f16]_**z**(svbool_t *pg*, svfloat16_t *op1*, svfloat16_t *op2*, float16_t *op3*)

Instances
`svfloat32_t svnmad[_n_f32]_z(svbool_t pg, svfloat32_t op1, svfloat32_t op2, float32_t op3)` `svfloat64_t svnmad[_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 svnmad[_n_f16]_m(svbool_t pg, svfloat16_t op1, svfloat16_t op2, float16_t op3)`
`svfloat32_t svnmad[_n_f32]_m(svbool_t pg, svfloat32_t op1, svfloat32_t op2, float32_t op3)`
`svfloat64_t svnmad[_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 svnmad[_n_f16]_x(svbool_t pg, svfloat16_t op1, svfloat16_t op2, float16_t op3)`
`svfloat32_t svnmad[_n_f32]_x(svbool_t pg, svfloat32_t op1, svfloat32_t op2, float32_t op3)`
`svfloat64_t svnmad[_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.

Signalling 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 svnmla[_f16]_z(svbool_t pg, svfloat16_t op1, svfloat16_t op2, svfloat16_t op3)`
`svfloat32_t svnmla[_f32]_z(svbool_t pg, svfloat32_t op1, svfloat32_t op2, svfloat32_t op3)`
`svfloat64_t svnmla[_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 svnmla[_f16]_m(svbool_t pg, svfloat16_t op1, svfloat16_t op2, svfloat16_t op3)`
`svfloat32_t svnmla[_f32]_m(svbool_t pg, svfloat32_t op1, svfloat32_t op2, svfloat32_t op3)`
`svfloat64_t svnmla[_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 **svnmla**[_f16]_**x**(svbool_t *pg*, svfloat16_t *op1*, svfloat16_t *op2*, svfloat16_t *op3*) svfloat32_t **svnmla**[_f32]_**x**(svbool_t *pg*, svfloat32_t *op1*, svfloat32_t *op2*, svfloat32_t *op3*) svfloat64_t **svnmla**[_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 **svnmla**[_n_f16]_**z**(svbool_t *pg*, svfloat16_t *op1*, svfloat16_t *op2*, float16_t *op3*) svfloat32_t **svnmla**[_n_f32]_**z**(svbool_t *pg*, svfloat32_t *op1*, svfloat32_t *op2*, float32_t *op3*) svfloat64_t **svnmla**[_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 **svnmla**[_n_f16]_**m**(svbool_t *pg*, svfloat16_t *op1*, svfloat16_t *op2*, float16_t *op3*) svfloat32_t **svnmla**[_n_f32]_**m**(svbool_t *pg*, svfloat32_t *op1*, svfloat32_t *op2*, float32_t *op3*) svfloat64_t **svnmla**[_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 **svnmla**[_n_f16]_**x**(svbool_t *pg*, svfloat16_t *op1*, svfloat16_t *op2*, float16_t *op3*) svfloat32_t **svnmla**[_n_f32]_**x**(svbool_t *pg*, svfloat32_t *op1*, svfloat32_t *op2*, float32_t *op3*) svfloat64_t **svnmla**[_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.

Signalling 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 **svnmsb**[_f16]_**z**(svbool_t *pg*, svfloat16_t *op1*, svfloat16_t *op2*,

Instances
svfloat16_t *op3*) svfloat32_t **svnmsb**[_f32]_**z**(svbool_t *pg*, svfloat32_t *op1*, svfloat32_t *op2*, svfloat32_t *op3*) svfloat64_t **svnmsb**[_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 **svnmsb**[_f16]_**m**(svbool_t *pg*, svfloat16_t *op1*, svfloat16_t *op2*, svfloat16_t *op3*) svfloat32_t **svnmsb**[_f32]_**m**(svbool_t *pg*, svfloat32_t *op1*, svfloat32_t *op2*, svfloat32_t *op3*) svfloat64_t **svnmsb**[_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 **svnmsb**[_f16]_**x**(svbool_t *pg*, svfloat16_t *op1*, svfloat16_t *op2*, svfloat16_t *op3*) svfloat32_t **svnmsb**[_f32]_**x**(svbool_t *pg*, svfloat32_t *op1*, svfloat32_t *op2*, svfloat32_t *op3*) 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*,

Instances
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.

Signalling 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 **svnmls**[_f16]_**z**(svbool_t *pg*, svfloat16_t *op1*, svfloat16_t *op2*, svfloat16_t *op3*) svfloat32_t **svnmls**[_f32]_**z**(svbool_t *pg*, svfloat32_t *op1*, svfloat32_t *op2*, svfloat32_t *op3*) svfloat64_t **svnmls**[_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 **svnmls**[_f16]_**m**(svbool_t *pg*, svfloat16_t *op1*, svfloat16_t *op2*, svfloat16_t *op3*) svfloat32_t **svnmls**[_f32]_**m**(svbool_t *pg*, svfloat32_t *op1*, svfloat32_t *op2*, svfloat32_t *op3*) svfloat64_t **svnmls**[_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 **svnmls**[_f16]_**x**(svbool_t *pg*, svfloat16_t *op1*, svfloat16_t *op2*, svfloat16_t *op3*) svfloat32_t **svnmls**[_f32]_**x**(svbool_t *pg*, svfloat32_t *op1*, svfloat32_t *op2*, svfloat32_t *op3*) svfloat64_t **svnmls**[_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 **svnmls**[_n_f16]_**z**(svbool_t *pg*, svfloat16_t *op1*, svfloat16_t *op2*, float16_t *op3*) svfloat32_t **svnmls**[_n_f32]_**z**(svbool_t *pg*, svfloat32_t *op1*, svfloat32_t *op2*, float32_t *op3*) svfloat64_t **svnmls**[_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 **svnmls**[_n_f16]_**m**(svbool_t *pg*, svfloat16_t *op1*, svfloat16_t *op2*, float16_t *op3*) svfloat32_t **svnmls**[_n_f32]_**m**(svbool_t *pg*, svfloat32_t *op1*, svfloat32_t *op2*, float32_t *op3*) svfloat64_t **svnmls**[_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 **svnmls**[_n_f16]_**x**(svbool_t *pg*, svfloat16_t *op1*, svfloat16_t *op2*, float16_t *op3*) svfloat32_t **svnmls**[_n_f32]_**x**(svbool_t *pg*, svfloat32_t *op1*, svfloat32_t *op2*, float32_t *op3*) svfloat64_t **svnmls**[_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.

Signalling 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 **svdiv**[_f16]_**z**(svbool_t *pg*, svfloat16_t *op1*, svfloat16_t *op2*) svfloat32_t **svdiv**[_f32]_**z**(svbool_t *pg*, svfloat32_t *op1*, svfloat32_t *op2*) svfloat64_t **svdiv**[_f64]_**z**(svbool_t *pg*, svfloat64_t *op1*, svfloat64_t *op2*)

6.16.17.2. DIV (vector, vector), merging with first input

Instances
svfloat16_t **svdiv**[_f16]_**m**(svbool_t *pg*, svfloat16_t *op1*, svfloat16_t *op2*) svfloat32_t **svdiv**[_f32]_**m**(svbool_t *pg*, svfloat32_t *op1*, svfloat32_t *op2*) svfloat64_t **svdiv**[_f64]_**m**(svbool_t *pg*, svfloat64_t *op1*, svfloat64_t *op2*)

6.16.17.3. DIV (vector, vector), setting inactive to unknown

Instances
svfloat16_t **svdiv**[_f16]_**x**(svbool_t *pg*, svfloat16_t *op1*, svfloat16_t *op2*) svfloat32_t **svdiv**[_f32]_**x**(svbool_t *pg*, svfloat32_t *op1*, svfloat32_t *op2*) svfloat64_t **svdiv**[_f64]_**x**(svbool_t *pg*, svfloat64_t *op1*, svfloat64_t *op2*)

6.16.17.4. DIV (vector, scalar), setting inactive to zero

Instances
svfloat16_t **svdiv**[_n_f16]_**z**(svbool_t *pg*, svfloat16_t *op1*, float16_t *op2*)

Instances
svfloat32_t **svdiv**[_n_f32]_**z**(svbool_t *pg*, svfloat32_t *op1*, float32_t *op2*)
svfloat64_t **svdiv**[_n_f64]_**z**(svbool_t *pg*, svfloat64_t *op1*, float64_t *op2*)

6.16.17.5. DIV (vector, scalar), merging with first input

Instances
svfloat16_t **svdiv**[_n_f16]_**m**(svbool_t *pg*, svfloat16_t *op1*, float16_t *op2*)
svfloat32_t **svdiv**[_n_f32]_**m**(svbool_t *pg*, svfloat32_t *op1*, float32_t *op2*)
svfloat64_t **svdiv**[_n_f64]_**m**(svbool_t *pg*, svfloat64_t *op1*, float64_t *op2*)

6.16.17.6. DIV (vector, scalar), setting inactive to unknown

Instances
svfloat16_t **svdiv**[_n_f16]_**x**(svbool_t *pg*, svfloat16_t *op1*, float16_t *op2*)
svfloat32_t **svdiv**[_n_f32]_**x**(svbool_t *pg*, svfloat32_t *op1*, float32_t *op2*)
svfloat64_t **svdiv**[_n_f64]_**x**(svbool_t *pg*, svfloat64_t *op1*, float64_t *op2*)

6.16.18. DIVR: Floating-point division, reversed

These functions divide the second floating-point value by the first input.

Signalling 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 **svdivr**[_f16]_**z**(svbool_t *pg*, svfloat16_t *op1*, svfloat16_t *op2*)
svfloat32_t **svdivr**[_f32]_**z**(svbool_t *pg*, svfloat32_t *op1*, svfloat32_t *op2*)
svfloat64_t **svdivr**[_f64]_**z**(svbool_t *pg*, svfloat64_t *op1*, svfloat64_t *op2*)

6.16.18.2. DIVR (vector, vector), merging with first input

Instances
svfloat16_t **svdivr**[_f16]_**m**(svbool_t *pg*, svfloat16_t *op1*, svfloat16_t *op2*)
svfloat32_t **svdivr**[_f32]_**m**(svbool_t *pg*, svfloat32_t *op1*, svfloat32_t *op2*)
svfloat64_t **svdivr**[_f64]_**m**(svbool_t *pg*, svfloat64_t *op1*, svfloat64_t *op2*)

6.16.18.3. DIVR (vector, vector), setting inactive to unknown

Instances
svfloat16_t **svdivr**[_f16]_**x**(svbool_t *pg*, svfloat16_t *op1*, svfloat16_t *op2*)
svfloat32_t **svdivr**[_f32]_**x**(svbool_t *pg*, svfloat32_t *op1*, svfloat32_t *op2*)
svfloat64_t **svdivr**[_f64]_**x**(svbool_t *pg*, svfloat64_t *op1*, svfloat64_t *op2*)

6.16.18.4. DIVR (vector, scalar), setting inactive to zero

Instances
svfloat16_t **svdivr**[_n_f16]_**z**(svbool_t *pg*, svfloat16_t *op1*, float16_t *op2*)

Instances
svfloat32_t **svdivr**[_n_f32]_**z**(svbool_t *pg*, svfloat32_t *op1*, float32_t *op2*)
svfloat64_t **svdivr**[_n_f64]_**z**(svbool_t *pg*, svfloat64_t *op1*, float64_t *op2*)

6.16.18.5. DIVR (vector, scalar), merging with first input

Instances
svfloat16_t **svdivr**[_n_f16]_**m**(svbool_t *pg*, svfloat16_t *op1*, float16_t *op2*)
svfloat32_t **svdivr**[_n_f32]_**m**(svbool_t *pg*, svfloat32_t *op1*, float32_t *op2*)
svfloat64_t **svdivr**[_n_f64]_**m**(svbool_t *pg*, svfloat64_t *op1*, float64_t *op2*)

6.16.18.6. DIVR (vector, scalar), setting inactive to unknown

Instances
svfloat16_t **svdivr**[_n_f16]_**x**(svbool_t *pg*, svfloat16_t *op1*, float16_t *op2*)
svfloat32_t **svdivr**[_n_f32]_**x**(svbool_t *pg*, svfloat32_t *op1*, float32_t *op2*)
svfloat64_t **svdivr**[_n_f64]_**x**(svbool_t *pg*, svfloat64_t *op1*, float64_t *op2*)

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.

Signalling 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 **svmax**[_f16]_**z**(svbool_t *pg*, svfloat16_t *op1*, svfloat16_t *op2*)
svfloat32_t **svmax**[_f32]_**z**(svbool_t *pg*, svfloat32_t *op1*, svfloat32_t *op2*)
svfloat64_t **svmax**[_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 **svmax**[_f16]_**m**(svbool_t *pg*, svfloat16_t *op1*, svfloat16_t *op2*)
svfloat32_t **svmax**[_f32]_**m**(svbool_t *pg*, svfloat32_t *op1*, svfloat32_t *op2*)
svfloat64_t **svmax**[_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 **svmax**[_f16]_**x**(svbool_t *pg*, svfloat16_t *op1*, svfloat16_t *op2*)
svfloat32_t **svmax**[_f32]_**x**(svbool_t *pg*, svfloat32_t *op1*, svfloat32_t *op2*)
svfloat64_t **svmax**[_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 **svmax**[_n_f16]_**z**(svbool_t *pg*, svfloat16_t *op1*, float16_t *op2*)
svfloat32_t **svmax**[_n_f32]_**z**(svbool_t *pg*, svfloat32_t *op1*, float32_t *op2*)

Instances
svfloat64_t **svmax**[_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 **svmax**[_n_f16]_**m**(svbool_t *pg*, svfloat16_t *op1*, float16_t *op2*)
svfloat32_t **svmax**[_n_f32]_**m**(svbool_t *pg*, svfloat32_t *op1*, float32_t *op2*)
svfloat64_t **svmax**[_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 **svmax**[_n_f16]_**x**(svbool_t *pg*, svfloat16_t *op1*, float16_t *op2*)
svfloat32_t **svmax**[_n_f32]_**x**(svbool_t *pg*, svfloat32_t *op1*, float32_t *op2*)
svfloat64_t **svmax**[_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.

Signalling 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 **svmaxnm**[_f16]_**z**(svbool_t *pg*, svfloat16_t *op1*, svfloat16_t *op2*)
svfloat32_t **svmaxnm**[_f32]_**z**(svbool_t *pg*, svfloat32_t *op1*, svfloat32_t *op2*)
svfloat64_t **svmaxnm**[_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 **svmaxnm**[_f16]_**m**(svbool_t *pg*, svfloat16_t *op1*, svfloat16_t *op2*)
svfloat32_t **svmaxnm**[_f32]_**m**(svbool_t *pg*, svfloat32_t *op1*, svfloat32_t *op2*)
svfloat64_t **svmaxnm**[_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 **svmaxnm**[_f16]_**x**(svbool_t *pg*, svfloat16_t *op1*, svfloat16_t *op2*)
svfloat32_t **svmaxnm**[_f32]_**x**(svbool_t *pg*, svfloat32_t *op1*, svfloat32_t *op2*)
svfloat64_t **svmaxnm**[_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 **svmaxnm**[_n_f16]_**z**(svbool_t *pg*, svfloat16_t *op1*, float16_t *op2*)
svfloat32_t **svmaxnm**[_n_f32]_**z**(svbool_t *pg*, svfloat32_t *op1*, float32_t *op2*)
svfloat64_t **svmaxnm**[_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 **svmaxnm**[_n_f16]_**m**(svbool_t *pg*, svfloat16_t *op1*, float16_t *op2*)
svfloat32_t **svmaxnm**[_n_f32]_**m**(svbool_t *pg*, svfloat32_t *op1*, float32_t *op2*)
svfloat64_t **svmaxnm**[_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 **svmaxnm**[_n_f16]_**x**(svbool_t *pg*, svfloat16_t *op1*, float16_t *op2*)
svfloat32_t **svmaxnm**[_n_f32]_**x**(svbool_t *pg*, svfloat32_t *op1*, float32_t *op2*)
svfloat64_t **svmaxnm**[_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.

Signalling 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 **svmin**[_f16]_**z**(svbool_t *pg*, svfloat16_t *op1*, svfloat16_t *op2*)
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*)

Instances
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.

Signalling 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*)
svfloat32_t **svminnm**[_f32]_**z**(svbool_t *pg*, svfloat32_t *op1*, svfloat32_t *op2*)
svfloat64_t **svminnm**[_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 **svminnm**[_f16]_**m**(svbool_t *pg*, svfloat16_t *op1*, svfloat16_t *op2*)
svfloat32_t **svminnm**[_f32]_**m**(svbool_t *pg*, svfloat32_t *op1*, svfloat32_t *op2*)
svfloat64_t **svminnm**[_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 **svminnm**[_f16]_**x**(svbool_t *pg*, svfloat16_t *op1*, svfloat16_t *op2*)
svfloat32_t **svminnm**[_f32]_**x**(svbool_t *pg*, svfloat32_t *op1*, svfloat32_t *op2*)
svfloat64_t **svminnm**[_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 **svminnm**[_n_f16]_**z**(svbool_t *pg*, svfloat16_t *op1*, float16_t *op2*)
svfloat32_t **svminnm**[_n_f32]_**z**(svbool_t *pg*, svfloat32_t *op1*, float32_t *op2*)
svfloat64_t **svminnm**[_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 **svminnm**[_n_f16]_**m**(svbool_t *pg*, svfloat16_t *op1*, float16_t *op2*)
svfloat32_t **svminnm**[_n_f32]_**m**(svbool_t *pg*, svfloat32_t *op1*, float32_t *op2*)

Instances
svfloat64_t **svminnm**[_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 **svminnm**[_n_f16]_**x**(svbool_t *pg*, svfloat16_t *op1*, float16_t *op2*) svfloat32_t **svminnm**[_n_f32]_**x**(svbool_t *pg*, svfloat32_t *op1*, float32_t *op2*) svfloat64_t **svminnm**[_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.

Signalling 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 **svscale**[_f16]_**z**(svbool_t *pg*, svfloat16_t *op1*, svint16_t *op2*) svfloat32_t **svscale**[_f32]_**z**(svbool_t *pg*, svfloat32_t *op1*, svint32_t *op2*) 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.

Signalling NaNs trigger an IEEE Invalid exception but quiet NaNs do not.

6.16.24.1. TSMUL (vector, vector)

Instances
`svfloat16_t svtsmul[_f16](svfloat16_t op1, svuint16_t op2)`
`svfloat32_t svtsmul[_f32](svfloat32_t op1, svuint32_t op2)`
`svfloat64_t svtsmul[_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.

Signalling 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 svtmad[_f16](svfloat16_t op1, svfloat16_t op2, uint64_t imm3)`
`svfloat32_t svtmad[_f32](svfloat32_t op1, svfloat32_t op2, uint64_t imm3)`
`svfloat64_t svtmad[_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 svtssel[_f16](svfloat16_t op1, svuint16_t op2)`
`svfloat32_t svtssel[_f32](svfloat32_t op1, svuint32_t op2)`

Instances
`svfloat64_t svtssel[_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 svabs[_f16]_z(svbool_t pg, svfloat16_t op)`
`svfloat32_t svabs[_f32]_z(svbool_t pg, svfloat32_t op)`
`svfloat64_t svabs[_f64]_z(svbool_t pg, svfloat64_t op)`

6.16.27.2. ABS (vector), merging with separate vector

Instances

`svfloat16_t svabs[_f16]_m(svfloat16_t inactive, svbool_t pg,`
`  svfloat16_t op)`
`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,`

Instances
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.

Signalling 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*) svfloat32_t **svsqrt**[_f32]_**z**(svbool_t *pg*, svfloat32_t *op*) svfloat64_t **svsqrt**[_f64]_**z**(svbool_t *pg*, svfloat64_t *op*)

6.16.29.2. SQRT (vector), merging with separate vector

Instances
svfloat16_t **svsqrt**[_f16]_**m**(svfloat16_t *inactive*, svbool_t *pg*, svfloat16_t *op*) svfloat32_t **svsqrt**[_f32]_**m**(svfloat32_t *inactive*, svbool_t *pg*, svfloat32_t *op*) svfloat64_t **svsqrt**[_f64]_**m**(svfloat64_t *inactive*, svbool_t *pg*, svfloat64_t *op*)

6.16.29.3. SQRT (vector), setting inactive to unknown

Instances
svfloat16_t **svsqrt**[_f16]_**x**(svbool_t *pg*, svfloat16_t *op*) svfloat32_t **svsqrt**[_f32]_**x**(svbool_t *pg*, svfloat32_t *op*) svfloat64_t **svsqrt**[_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 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 **svrecpx**[_f16]_**z**(svbool_t *pg*, svfloat16_t *op*) svfloat32_t **svrecpx**[_f32]_**z**(svbool_t *pg*, svfloat32_t *op*) svfloat64_t **svrecpx**[_f64]_**z**(svbool_t *pg*, svfloat64_t *op*)

6.16.33.2. RECPX (vector), merging with separate vector

Instances
svfloat16_t **svrecpx**[_f16]_**m**(svfloat16_t *inactive*, svbool_t *pg*, svfloat16_t *op*) svfloat32_t **svrecpx**[_f32]_**m**(svfloat32_t *inactive*, svbool_t *pg*, svfloat32_t *op*) svfloat64_t **svrecpx**[_f64]_**m**(svfloat64_t *inactive*, svbool_t *pg*, svfloat64_t *op*)

6.16.33.3. RECPX (vector), setting inactive to unknown

Instances
svfloat16_t **svrecpx**[_f16]_**x**(svbool_t *pg*, svfloat16_t *op*) svfloat32_t **svrecpx**[_f32]_**x**(svbool_t *pg*, svfloat32_t *op*) svfloat64_t **svrecpx**[_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 **svrsqrte**[_f16](svfloat16_t *op*) svfloat32_t **svrsqrte**[_f32](svfloat32_t *op*) svfloat64_t **svrsqrte**[_f64](svfloat64_t *op*)

6.16.35. RSQRTS: Floating-point reciprocal square root step

6.16.35.1. RSQRTS (vector, vector)

Instances
svfloat16_t **svrsqrts**[_f16](svfloat16_t *op1*, svfloat16_t *op2*) svfloat32_t **svrsqrts**[_f32](svfloat32_t *op1*, svfloat32_t *op2*) svfloat64_t **svrsqrts**[_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.

Signalling NaNs trigger an IEEE Invalid exception but quiet NaNs do not.

6.16.36.1. RINTA (vector), setting inactive to zero

Instances
svfloat16_t **svrinta**[_f16]_**z**(svbool_t *pg*, svfloat16_t *op*) svfloat32_t **svrinta**[_f32]_**z**(svbool_t *pg*, svfloat32_t *op*) svfloat64_t **svrinta**[_f64]_**z**(svbool_t *pg*, svfloat64_t *op*)

6.16.36.2. RINTA (vector), merging with separate vector

Instances
svfloat16_t **svrinta**[_f16]_**m**(svfloat16_t *inactive*, svbool_t *pg*, svfloat16_t *op*) svfloat32_t **svrinta**[_f32]_**m**(svfloat32_t *inactive*, svbool_t *pg*, svfloat32_t *op*) svfloat64_t **svrinta**[_f64]_**m**(svfloat64_t *inactive*, svbool_t *pg*, svfloat64_t *op*)

6.16.36.3. RINTA (vector), setting inactive to unknown

Instances
svfloat16_t **svrinta**[_f16]_**x**(svbool_t *pg*, svfloat16_t *op*) svfloat32_t **svrinta**[_f32]_**x**(svbool_t *pg*, svfloat32_t *op*) svfloat64_t **svrinta**[_f64]_**x**(svbool_t *pg*, svfloat64_t *op*)

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.

Signalling NaNs trigger an IEEE Invalid exception but quiet NaNs do not.

6.16.37.1. RINTI (vector), setting inactive to zero

Instances
svfloat16_t **svrinti**[_f16]_**z**(svbool_t *pg*, svfloat16_t *op*)

Instances
`svfloat32_t svrinti[_f32]_z(svbool_t pg, svfloat32_t op)`
`svfloat64_t svrinti[_f64]_z(svbool_t pg, svfloat64_t op)`

6.16.37.2. RINTI (vector), merging with separate vector

Instances
`svfloat16_t svrinti[_f16]_m(svfloat16_t inactive, svbool_t pg, svfloat16_t op)`
`svfloat32_t svrinti[_f32]_m(svfloat32_t inactive, svbool_t pg, svfloat32_t op)`
`svfloat64_t svrinti[_f64]_m(svfloat64_t inactive, svbool_t pg, svfloat64_t op)`

6.16.37.3. RINTI (vector), setting inactive to unknown

Instances
`svfloat16_t svrinti[_f16]_x(svbool_t pg, svfloat16_t op)`
`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.

Signalling 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.

Signalling 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,`
`  svfloat16_t op)`
`svfloat32_t svrintn[_f32]_m(svfloat32_t inactive, svbool_t pg,`
`  svfloat32_t op)`
`svfloat64_t svrintn[_f64]_m(svfloat64_t inactive, svbool_t pg,`
`  svfloat64_t op)`

6.16.39.3. RINTN (vector), setting inactive to unknown

Instances

`svfloat16_t svrintn[_f16]_x(svbool_t pg, svfloat16_t op)`
`svfloat32_t svrintn[_f32]_x(svbool_t pg, svfloat32_t op)`
`svfloat64_t svrintn[_f64]_x(svbool_t pg, svfloat64_t op)`

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.

Signalling NaNs trigger an IEEE Invalid exception but quiet NaNs do not.

6.16.40.1. RINTP (vector), setting inactive to zero

Instances

`svfloat16_t svrintp[_f16]_z(svbool_t pg, svfloat16_t op)`
`svfloat32_t svrintp[_f32]_z(svbool_t pg, svfloat32_t op)`
`svfloat64_t svrintp[_f64]_z(svbool_t pg, svfloat64_t op)`

6.16.40.2. RINTP (vector), merging with separate vector

Instances

`svfloat16_t svrintp[_f16]_m(svfloat16_t inactive, svbool_t pg,`
`  svfloat16_t op)`
`svfloat32_t svrintp[_f32]_m(svfloat32_t inactive, svbool_t pg,`
`  svfloat32_t op)`
`svfloat64_t svrintp[_f64]_m(svfloat64_t inactive, svbool_t pg,`

Instances
svfloat64_t *op*)

6.16.40.3. RINTP (vector), setting inactive to unknown

Instances
svfloat16_t **svrintp**[_f16]_**x**(svbool_t *pg*, svfloat16_t *op*) svfloat32_t **svrintp**[_f32]_**x**(svbool_t *pg*, svfloat32_t *op*) svfloat64_t **svrintp**[_f64]_**x**(svbool_t *pg*, svfloat64_t *op*)

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.

Signalling 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.

Signalling 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*)

Instances
`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)`
`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)`
`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.

Signalling NaNs trigger an IEEE Invalid exception but quiet NaNs do not.

6.18.1.1. CMPEQ (vector, vector)

Instances
svbool_t **svcmpeq**[_f16](svbool_t *pg*, svfloat16_t *op1*, svfloat16_t *op2*)
svbool_t **svcmpeq**[_f32](svbool_t *pg*, svfloat32_t *op1*, svfloat32_t *op2*)
svbool_t **svcmpeq**[_f64](svbool_t *pg*, svfloat64_t *op1*, svfloat64_t *op2*)

6.18.1.2. CMPEQ (vector, scalar)

Instances
svbool_t **svcmpeq**[_n_f16](svbool_t *pg*, svfloat16_t *op1*, float16_t *op2*)
svbool_t **svcmpeq**[_n_f32](svbool_t *pg*, svfloat32_t *op1*, float32_t *op2*)
svbool_t **svcmpeq**[_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.

Signalling NaNs trigger an IEEE Invalid exception but quiet NaNs do not.

6.18.2.1. CMPNE (vector, vector)

Instances
svbool_t **svcmpne**[_f16](svbool_t *pg*, svfloat16_t *op1*, svfloat16_t *op2*)
svbool_t **svcmpne**[_f32](svbool_t *pg*, svfloat32_t *op1*, svfloat32_t *op2*)
svbool_t **svcmpne**[_f64](svbool_t *pg*, svfloat64_t *op1*, svfloat64_t *op2*)

6.18.2.2. CMPNE (vector, scalar)

Instances
svbool_t **svcmpne**[_n_f16](svbool_t *pg*, svfloat16_t *op1*, float16_t *op2*)
svbool_t **svcmpne**[_n_f32](svbool_t *pg*, svfloat32_t *op1*, float32_t *op2*)
svbool_t **svcmpne**[_n_f64](svbool_t *pg*, svfloat64_t *op1*, float64_t *op2*)

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 signalling and quiet NaNs trigger an IEEE Invalid exception.

6.18.3.1. CMPLT (vector, vector)

Instances
svbool_t **svcmplt**[_f16](svbool_t *pg*, svfloat16_t *op1*, svfloat16_t *op2*)
svbool_t **svcmplt**[_f32](svbool_t *pg*, svfloat32_t *op1*, svfloat32_t *op2*)
svbool_t **svcmplt**[_f64](svbool_t *pg*, svfloat64_t *op1*, svfloat64_t *op2*)

6.18.3.2. CMPLT (vector, scalar)

Instances
svbool_t **svcmplt**[_n_f16](svbool_t *pg*, svfloat16_t *op1*, float16_t *op2*)

Instances
svbool_t **svcmplt**[_n_f32](svbool_t *pg*, svfloat32_t *op1*, float32_t *op2*)
svbool_t **svcmplt**[_n_f64](svbool_t *pg*, svfloat64_t *op1*, float64_t *op2*)

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 signalling and quiet NaNs trigger an IEEE Invalid exception.

6.18.4.1. CMPLE (vector, vector)

Instances
svbool_t **svcmple**[_f16](svbool_t *pg*, svfloat16_t *op1*, svfloat16_t *op2*)
svbool_t **svcmple**[_f32](svbool_t *pg*, svfloat32_t *op1*, svfloat32_t *op2*)
svbool_t **svcmple**[_f64](svbool_t *pg*, svfloat64_t *op1*, svfloat64_t *op2*)

6.18.4.2. CMPLE (vector, scalar)

Instances
svbool_t **svcmple**[_n_f16](svbool_t *pg*, svfloat16_t *op1*, float16_t *op2*)
svbool_t **svcmple**[_n_f32](svbool_t *pg*, svfloat32_t *op1*, float32_t *op2*)
svbool_t **svcmple**[_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 signalling and quiet NaNs trigger an IEEE Invalid exception.

6.18.5.1. CMPGE (vector, vector)

Instances
svbool_t **svcmpge**[_f16](svbool_t *pg*, svfloat16_t *op1*, svfloat16_t *op2*)
svbool_t **svcmpge**[_f32](svbool_t *pg*, svfloat32_t *op1*, svfloat32_t *op2*)
svbool_t **svcmpge**[_f64](svbool_t *pg*, svfloat64_t *op1*, svfloat64_t *op2*)

6.18.5.2. CMPGE (vector, scalar)

Instances
svbool_t **svcmpge**[_n_f16](svbool_t *pg*, svfloat16_t *op1*, float16_t *op2*)
svbool_t **svcmpge**[_n_f32](svbool_t *pg*, svfloat32_t *op1*, float32_t *op2*)
svbool_t **svcmpge**[_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 signalling and quiet NaNs trigger an IEEE Invalid exception.

6.18.6.1. CMPGT (vector, vector)

Instances
svbool_t **svcmpgt**[_f16](svbool_t *pg*, svfloat16_t *op1*, svfloat16_t *op2*)
svbool_t **svcmpgt**[_f32](svbool_t *pg*, svfloat32_t *op1*, svfloat32_t *op2*)
svbool_t **svcmpgt**[_f64](svbool_t *pg*, svfloat64_t *op1*, svfloat64_t *op2*)

6.18.6.2. CMPGT (vector, scalar)

Instances
svbool_t **svcmpgt**[_n_f16](svbool_t *pg*, svfloat16_t *op1*, float16_t *op2*)
svbool_t **svcmpgt**[_n_f32](svbool_t *pg*, svfloat32_t *op1*, float32_t *op2*)
svbool_t **svcmpgt**[_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 **svcmpuo**[_f16](svbool_t *pg*, svfloat16_t *op1*, svfloat16_t *op2*)
svbool_t **svcmpuo**[_f32](svbool_t *pg*, svfloat32_t *op1*, svfloat32_t *op2*)
svbool_t **svcmpuo**[_f64](svbool_t *pg*, svfloat64_t *op1*, svfloat64_t *op2*)

6.18.7.2. CMPUO (vector, scalar)

Instances
svbool_t **svcmpuo**[_n_f16](svbool_t *pg*, svfloat16_t *op1*, float16_t *op2*)
svbool_t **svcmpuo**[_n_f32](svbool_t *pg*, svfloat32_t *op1*, float32_t *op2*)
svbool_t **svcmpuo**[_n_f64](svbool_t *pg*, svfloat64_t *op1*, float64_t *op2*)

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 signalling and quiet NaNs trigger an IEEE Invalid exception.

6.18.8.1. ACLT (vector, vector)

Instances
svbool_t **svaclt**[_f16](svbool_t *pg*, svfloat16_t *op1*, svfloat16_t *op2*)
svbool_t **svaclt**[_f32](svbool_t *pg*, svfloat32_t *op1*, svfloat32_t *op2*)
svbool_t **svaclt**[_f64](svbool_t *pg*, svfloat64_t *op1*, svfloat64_t *op2*)

6.18.8.2. ACLT (vector, scalar)

Instances
svbool_t **svaclt**[_n_f16](svbool_t *pg*, svfloat16_t *op1*, float16_t *op2*)

Instances
svbool_t **svaclt**[_n_f32](svbool_t *pg*, svfloat32_t *op1*, float32_t *op2*)
svbool_t **svaclt**[_n_f64](svbool_t *pg*, svfloat64_t *op1*, float64_t *op2*)

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 signalling and quiet NaNs trigger an IEEE Invalid exception.

6.18.9.1. ACLE (vector, vector)

Instances
svbool_t **svacle**[_f16](svbool_t *pg*, svfloat16_t *op1*, svfloat16_t *op2*)
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 signalling 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 signalling 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*) 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 signalling 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*,

Instances
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*) 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*)

Instances
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*, svuint16_t *op*) svfloat16_t **svcvt_f16**[_s32]_**m**(svfloat16_t *inactive*, svbool_t *pg*, svint32_t *op*) svfloat32_t **svcvt_f32**[_s32]_**m**(svfloat32_t *inactive*, svbool_t *pg*, svint32_t *op*) svfloat64_t **svcvt_f64**[_s32]_**m**(svfloat64_t *inactive*, svbool_t *pg*, svint32_t *op*) svfloat16_t **svcvt_f16**[_s64]_**m**(svfloat16_t *inactive*, svbool_t *pg*, svint64_t *op*) svfloat32_t **svcvt_f32**[_s64]_**m**(svfloat32_t *inactive*, svbool_t *pg*, svint64_t *op*) svfloat64_t **svcvt_f64**[_s64]_**m**(svfloat64_t *inactive*, svbool_t *pg*, svint64_t *op*) svfloat16_t **svcvt_f16**[_u32]_**m**(svfloat16_t *inactive*, svbool_t *pg*, svuint32_t *op*) svfloat32_t **svcvt_f32**[_u32]_**m**(svfloat32_t *inactive*, svbool_t *pg*, svuint32_t *op*) svfloat64_t **svcvt_f64**[_u32]_**m**(svfloat64_t *inactive*, svbool_t *pg*, svuint32_t *op*) svfloat16_t **svcvt_f16**[_u64]_**m**(svfloat16_t *inactive*, svbool_t *pg*, svuint64_t *op*) svfloat32_t **svcvt_f32**[_u64]_**m**(svfloat32_t *inactive*, svbool_t *pg*, svuint64_t *op*) svfloat64_t **svcvt_f64**[_u64]_**m**(svfloat64_t *inactive*, svbool_t *pg*, svuint64_t *op*)

6.19.2.3. CVT (vector), setting inactive to unknown

Instances
svfloat16_t **svcvt_f16**[_s16]_**x**(svbool_t *pg*, svint16_t *op*) svfloat16_t **svcvt_f16**[_u16]_**x**(svbool_t *pg*, svuint16_t *op*) svfloat16_t **svcvt_f16**[_s32]_**x**(svbool_t *pg*, svint32_t *op*) svfloat32_t **svcvt_f32**[_s32]_**x**(svbool_t *pg*, svint32_t *op*) svfloat64_t **svcvt_f64**[_s32]_**x**(svbool_t *pg*, svint32_t *op*) svfloat16_t **svcvt_f16**[_s64]_**x**(svbool_t *pg*, svint64_t *op*) svfloat32_t **svcvt_f32**[_s64]_**x**(svbool_t *pg*, svint64_t *op*) svfloat64_t **svcvt_f64**[_s64]_**x**(svbool_t *pg*, svint64_t *op*) svfloat16_t **svcvt_f16**[_u32]_**x**(svbool_t *pg*, svuint32_t *op*) svfloat32_t **svcvt_f32**[_u32]_**x**(svbool_t *pg*, svuint32_t *op*) svfloat64_t **svcvt_f64**[_u32]_**x**(svbool_t *pg*, svuint32_t *op*) svfloat16_t **svcvt_f16**[_u64]_**x**(svbool_t *pg*, svuint64_t *op*) svfloat32_t **svcvt_f32**[_u64]_**x**(svbool_t *pg*, svuint64_t *op*)

Instances
`svfloat64_t svcvt_f64[_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.

Signalling NaNs trigger an IEEE Invalid exception but quiet NaNs do not.

6.19.3.1. CVT (vector), setting inactive to zero

Instances

`svfloat32_t svcvt_f32[_f16]_z(svbool_t pg, svfloat16_t op)`
`svfloat64_t svcvt_f64[_f16]_z(svbool_t pg, svfloat16_t op)`
`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.

Signalling 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,`

Instances
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
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*)

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
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*)

Instances
`float64_t svlastb[_f64](svbool_t pg, svfloat64_t op)`

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

`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)`
`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)`

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)`

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)`

Instances
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*, svfloat16_t *data*) svfloat32_t **svclastb**[_f32](svbool_t *pg*, svfloat32_t *fallback*, svfloat32_t *data*) svfloat64_t **svclastb**[_f64](svbool_t *pg*, svfloat64_t *fallback*, svfloat64_t *data*)

6.20.4.2. CLASTB (scalar, vector)

Instances
int8_t **svclastb**[_n_s8](svbool_t *pg*, int8_t *fallback*, svint8_t *data*) int16_t **svclastb**[_n_s16](svbool_t *pg*, int16_t *fallback*, svint16_t *data*) int32_t **svclastb**[_n_s32](svbool_t *pg*, int32_t *fallback*, svint32_t *data*) int64_t **svclastb**[_n_s64](svbool_t *pg*, int64_t *fallback*, svint64_t *data*) uint8_t **svclastb**[_n_u8](svbool_t *pg*, uint8_t *fallback*, svuint8_t *data*) uint16_t **svclastb**[_n_u16](svbool_t *pg*, uint16_t *fallback*, svuint16_t *data*) uint32_t **svclastb**[_n_u32](svbool_t *pg*, uint32_t *fallback*, svuint32_t *data*) uint64_t **svclastb**[_n_u64](svbool_t *pg*, uint64_t *fallback*, svuint64_t *data*) float16_t **svclastb**[_n_f16](svbool_t *pg*, float16_t *fallback*, svfloat16_t *data*) float32_t **svclastb**[_n_f32](svbool_t *pg*, float32_t *fallback*, svfloat32_t *data*) float64_t **svclastb**[_n_f64](svbool_t *pg*, float64_t *fallback*, svfloat64_t *data*)

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
svint32_t **svcompact**[_s32](svbool_t *pg*, svint32_t *op*) svint64_t **svcompact**[_s64](svbool_t *pg*, svint64_t *op*) svuint32_t **svcompact**[_u32](svbool_t *pg*, svuint32_t *op*) svuint64_t **svcompact**[_u64](svbool_t *pg*, svuint64_t *op*) svfloat32_t **svcompact**[_f32](svbool_t *pg*, svfloat32_t *op*) svfloat64_t **svcompact**[_f64](svbool_t *pg*, svfloat64_t *op*)

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
svint8_t **svsplice**[_s8](svbool_t *pg*, svint8_t *op1*, svint8_t *op2*)
svint16_t **svsplice**[_s16](svbool_t *pg*, svint16_t *op1*, svint16_t *op2*)
svint32_t **svsplice**[_s32](svbool_t *pg*, svint32_t *op1*, svint32_t *op2*)
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*)

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*)

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*)

Instances
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*)
svfloat64_t **svsel**[_f64](svbool_t *pg*, svfloat64_t *op1*, svfloat64_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*)

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
`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)`

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
`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)`

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
`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)`

6.20.12.2. REV (predicate)

Instances
svbool_t **svrev_b8**(svbool_t *op*) svbool_t **svrev_b16**(svbool_t *op*) svbool_t **svrev_b32**(svbool_t *op*) svbool_t **svrev_b64**(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 **svtrn1**[_s8](svint8_t *op1*, svint8_t *op2*) svint16_t **svtrn1**[_s16](svint16_t *op1*, svint16_t *op2*) svint32_t **svtrn1**[_s32](svint32_t *op1*, svint32_t *op2*) svint64_t **svtrn1**[_s64](svint64_t *op1*, svint64_t *op2*) svuint8_t **svtrn1**[_u8](svuint8_t *op1*, svuint8_t *op2*) svuint16_t **svtrn1**[_u16](svuint16_t *op1*, svuint16_t *op2*) svuint32_t **svtrn1**[_u32](svuint32_t *op1*, svuint32_t *op2*) svuint64_t **svtrn1**[_u64](svuint64_t *op1*, svuint64_t *op2*) svfloat16_t **svtrn1**[_f16](svfloat16_t *op1*, svfloat16_t *op2*) svfloat32_t **svtrn1**[_f32](svfloat32_t *op1*, svfloat32_t *op2*) svfloat64_t **svtrn1**[_f64](svfloat64_t *op1*, svfloat64_t *op2*)

6.20.13.2. TRN1 (predicate, predicate)

Instances
svbool_t **svtrn1_b8**(svbool_t *op1*, svbool_t *op2*) svbool_t **svtrn1_b16**(svbool_t *op1*, svbool_t *op2*) svbool_t **svtrn1_b32**(svbool_t *op1*, svbool_t *op2*) svbool_t **svtrn1_b64**(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 **svtrn2**[_s8](svint8_t *op1*, svint8_t *op2*) svint16_t **svtrn2**[_s16](svint16_t *op1*, svint16_t *op2*) svint32_t **svtrn2**[_s32](svint32_t *op1*, svint32_t *op2*) svint64_t **svtrn2**[_s64](svint64_t *op1*, svint64_t *op2*) svuint8_t **svtrn2**[_u8](svuint8_t *op1*, svuint8_t *op2*) svuint16_t **svtrn2**[_u16](svuint16_t *op1*, svuint16_t *op2*) svuint32_t **svtrn2**[_u32](svuint32_t *op1*, svuint32_t *op2*) svuint64_t **svtrn2**[_u64](svuint64_t *op1*, svuint64_t *op2*) svfloat16_t **svtrn2**[_f16](svfloat16_t *op1*, svfloat16_t *op2*) svfloat32_t **svtrn2**[_f32](svfloat32_t *op1*, svfloat32_t *op2*)

Instances
svfloat64_t **svtrn2**[_f64](svfloat64_t *op1*, svfloat64_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
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*)

6.20.17.2. UZP1 (predicate, predicate)

Instances
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*)

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
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*)

6.20.18.2. UZP2 (predicate, predicate)

Instances
svbool_t **svuzp2_b8**(svbool_t *op1*, svbool_t *op2*)
svbool_t **svuzp2_b16**(svbool_t *op1*, svbool_t *op2*)

Instances
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*)

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*)
svfloat64_t **svzip2**[_f64](svfloat64_t *op1*, svfloat64_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. Predicate creation

6.21.1. 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`.

6.21.1.1. PTRUE (inherent)

Instances
svbool_t **svptrue_b8**()
svbool_t **svptrue_b16**()
svbool_t **svptrue_b32**()
svbool_t **svptrue_b64**()

6.21.1.2. PTRUE (pattern)

Instances
svbool_t **svptrue_pat_b8**(svpattern *pattern*)
svbool_t **svptrue_pat_b16**(svpattern *pattern*)
svbool_t **svptrue_pat_b32**(svpattern *pattern*)
svbool_t **svptrue_pat_b64**(svpattern *pattern*)

6.21.2. PFALSE: Return an all-false predicate

This function returns a predicate in which every bit is false.

6.21.2.1. PFALSE (inherent)

Instances
svbool_t **svpfalse**[_b]()

6.21.3. DUP: Duplicate boolean value

These functions copy a boolean value into every element of a predicate. More precisely, the behavior of:

`svdup_n_b`*`NN`*`(`*`op`*`)`

is identical to:

`op` `? svptrue_b`*`NN`*`() : svpfalse_b()`

although the implementation can use any instruction sequence that achieves the same effect.

6.21.3.1. DUP (scalar)

Instances
svbool_t **svdup**[_n]_**b8**(bool *op*) svbool_t **svdup**[_n]_**b16**(bool *op*) svbool_t **svdup**[_n]_**b32**(bool *op*) svbool_t **svdup**[_n]_**b64**(bool *op*)

6.21.4. 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.

6.21.4.1. 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*)

6.21.4.2. 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*)

6.21.4.3. DUPQ (4 scalars)

Instances
svbool_t **svdupq**[_n]_**b32**(bool *x0*, bool *x1*, bool *x2*, bool *x3*)

6.21.4.4. DUPQ (2 scalars)

Instances
svbool_t **svdupq**[_n]_**b64**(bool *x0*, bool *x1*)

6.22. Predicate operations

6.22.1. MOV: Copy predicate

These functions take a copy of the active elements of a predicate.

6.22.1.1. MOV (predicate), setting inactive to zero

Instances
svbool_t **svmov**[_b]_**z**(svbool_t *pg*, svbool_t *op*)

6.22.2. AND: Predicate AND

These functions perform an AND of two predicate inputs.

6.22.2.1. AND (predicate, predicate), setting inactive to zero

Instances
svbool_t **svand**[_b]_**z**(svbool_t *pg*, svbool_t *op1*, svbool_t *op2*)

6.22.3. BIC: Predicate AND NOT

These functions invert the second predicate input and AND it with the first predicate input.

6.22.3.1. BIC (predicate, predicate), setting inactive to zero

Instances
svbool_t **svbic**[_b]_**z**(svbool_t *pg*, svbool_t *op1*, svbool_t *op2*)

6.22.4. NAND: Predicate NAND

These functions perform an AND of two predicate inputs and invert the result.

6.22.4.1. NAND (predicate, predicate), setting inactive to zero

Instances
svbool_t **svnand**[_b]_**z**(svbool_t *pg*, svbool_t *op1*, svbool_t *op2*)

6.22.5. ORR: Predicate OR

These functions perform an OR of two predicate inputs.

6.22.5.1. ORR (predicate, predicate), setting inactive to zero

Instances
svbool_t **svorr**[_b]_**z**(svbool_t *pg*, svbool_t *op1*, svbool_t *op2*)

6.22.6. ORN: Predicate OR NOT

These functions invert the second predicate input and OR it with the first predicate input.

6.22.6.1. ORN (predicate, predicate), setting inactive to zero

Instances
svbool_t **svorn**[_b]_**z**(svbool_t *pg*, svbool_t *op1*, svbool_t *op2*)

6.22.7. NOR: Predicate NOR

These functions perform an OR of two predicate inputs and invert the result.

6.22.7.1. NOR (predicate, predicate), setting inactive to zero

Instances
svbool_t **svnor**[_b]_**z**(svbool_t *pg*, svbool_t *op1*, svbool_t *op2*)

6.22.8. EOR: Predicate exclusive OR

These functions perform an exclusive OR of two predicate inputs.

6.22.8.1. EOR (predicate, predicate), setting inactive to zero

Instances
svbool_t **sveor**[_b]_**z**(svbool_t *pg*, svbool_t *op1*, svbool_t *op2*)

6.22.9. NOT: Predicate NOT

These functions invert a predicate input.

6.22.9.1. NOT (predicate), setting inactive to zero

Instances
svbool_t **svnot**[_b]_**z**(svbool_t *pg*, svbool_t *op*)

6.22.10. 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.

6.22.10.1. BRKA (predicate), setting inactive to zero

Instances
svbool_t **svbrka**[_b]_**z**(svbool_t *pg*, svbool_t *op*)

6.22.10.2. BRKA (predicate), merging with separate predicate

Instances
svbool_t **svbrka**[_b]_**m**(svbool_t *inactive*, svbool_t *pg*, svbool_t *op*)

6.22.11. 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.

6.22.11.1. BRKB (predicate), setting inactive to zero

Instances
svbool_t **svbrkb**[_b]_**z**(svbool_t *pg*, svbool_t *op*)

6.22.11.2. BRKB (predicate), merging with separate predicate

Instances
svbool_t **svbrkb**[_b]_**m**(svbool_t *inactive*, svbool_t *pg*, svbool_t *op*)

6.22.12. 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.

6.22.12.1. BRKN (predicate, predicate), setting inactive to zero

Instances
svbool_t **svbrkn**[_b]_**z**(svbool_t *pg*, svbool_t *op1*, svbool_t *op2*)

6.22.13. 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.

6.22.13.1. BRKPA (predicate, predicate), setting inactive to zero

Instances
svbool_t **svbrkpa**[_b]_**z**(svbool_t *pg*, svbool_t *op1*, svbool_t *op2*)

6.22.14. 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.

6.22.14.1. BRKPB (predicate, predicate), setting inactive to zero

Instances
svbool_t **svbrkpb**[_b]_**z**(svbool_t *pg*, svbool_t *op1*, svbool_t *op2*)

6.22.15. 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.

6.22.15.1. PFIRST (predicate)

Instances
svbool_t **svpfirst**[_b](svbool_t *pg*, svbool_t *op*)

6.22.16. 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.

6.22.16.1. PNEXT (predicate)

Instances
svbool_t **svpnext_b8**(svbool_t *pg*, svbool_t *op*)

Instances

`svbool_t svpnext_b16(svbool_t pg, svbool_t op)`
`svbool_t svpnext_b32(svbool_t pg, svbool_t op)`
`svbool_t svpnext_b64(svbool_t pg, svbool_t op)`

6.23. Testing predicates

6.23.1. 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` Return true if the first active element is true.

`last` 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.

6.23.1.1. 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)`

6.24. FFR manipulation

6.24.1. 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.

6.24.1.1. RDFFR (inherent)

Instances
`svbool_t svrdffr()`

6.24.2. 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” a description of FFR groups.

6.24.2.1. SETFFR (inherent)

Instances
`void svsetffr()`

6.24.3. 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.

6.24.3.1. WRFFR (predicate)

Instances
`void svwrffr(svbool_t op)`

6.25. Counting elements

6.25.1. 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.

6.25.1.1. CNTP (predicate)

Instances

`uint64_t svcntp_b8(svbool_t pg, svbool_t op)`
`uint64_t svcntp_b16(svbool_t pg, svbool_t op)`
`uint64_t svcntp_b32(svbool_t pg, svbool_t op)`
`uint64_t svcntp_b64(svbool_t pg, svbool_t op)`

6.25.2. 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`.

6.25.2.1. CNTB (inherent)

Instances
`uint64_t svcntb()`

6.25.2.2. CNTB (pattern)

Instances
`uint64_t svcntb_pat(svpattern pattern)`

6.25.3. 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`.

6.25.3.1. CNTH (inherent)

Instances
`uint64_t svcnth()`

6.25.3.2. CNTH (pattern)

Instances
`uint64_t` **`svcnth_pat`**`(svpattern` *`pattern`*`)`

6.25.4. 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`.

6.25.4.1. CNTW (inherent)

Instances
`uint64_t` **`svcntw`**`()`

6.25.4.2. CNTW (pattern)

Instances
`uint64_t` **`svcntw_pat`**`(svpattern` *`pattern`*`)`

6.25.5. 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`.

6.25.5.1. CNTD (inherent)

Instances
`uint64_t` **`svcntd`**`()`

6.25.5.2. CNTD (pattern)

Instances
`uint64_t` **`svcntd_pat`**`(svpattern` *`pattern`*`)`

6.25.6. 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.

6.25.6.1. LEN (vector)

Instances

`uint64_t` **`svlen`**`[_s8](svint8_t` *`op`*`)`
`uint64_t` **`svlen`**`[_s16](svint16_t` *`op`*`)`
`uint64_t` **`svlen`**`[_s32](svint32_t` *`op`*`)`

Instances
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*)

6.26. Saturating scalar arithmetic

6.26.1. 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].

6.26.1.1. 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*)

6.26.1.2. QINCB (scalar, pattern, multiplier)

Instances
int32_t **svqincb_pat**[_n_s32](int32_t *op*, svpattern *pattern*, uint64_t *imm_factor*)
int64_t **svqincb_pat**[_n_s64](int64_t *op*, svpattern *pattern*, uint64_t *imm_factor*)
uint32_t **svqincb_pat**[_n_u32](uint32_t *op*, svpattern *pattern*, uint64_t *imm_factor*)
uint64_t **svqincb_pat**[_n_u64](uint64_t *op*, svpattern *pattern*, uint64_t *imm_factor*)

6.26.2. 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].

6.26.2.1. QINCH (scalar, multiplier)

Instances
int32_t **svqinch**[_n_s32](int32_t *op*, uint64_t *imm_factor*) int64_t **svqinch**[_n_s64](int64_t *op*, uint64_t *imm_factor*) uint32_t **svqinch**[_n_u32](uint32_t *op*, uint64_t *imm_factor*) uint64_t **svqinch**[_n_u64](uint64_t *op*, uint64_t *imm_factor*)

6.26.2.2. QINCH (scalar, pattern, multiplier)

Instances
int32_t **svqinch_pat**[_n_s32](int32_t *op*, svpattern *pattern*, uint64_t *imm_factor*) int64_t **svqinch_pat**[_n_s64](int64_t *op*, svpattern *pattern*, uint64_t *imm_factor*) uint32_t **svqinch_pat**[_n_u32](uint32_t *op*, svpattern *pattern*, uint64_t *imm_factor*) uint64_t **svqinch_pat**[_n_u64](uint64_t *op*, svpattern *pattern*, uint64_t *imm_factor*)

6.26.2.3. QINCH (vector, multiplier)

Instances
svint16_t **svqinch**[_s16](svint16_t *op*, uint64_t *imm_factor*) svuint16_t **svqinch**[_u16](svuint16_t *op*, uint64_t *imm_factor*)

6.26.2.4. QINCH (vector, pattern, multiplier)

Instances
svint16_t **svqinch_pat**[_s16](svint16_t *op*, svpattern *pattern*, uint64_t *imm_factor*) svuint16_t **svqinch_pat**[_u16](svuint16_t *op*, svpattern *pattern*, uint64_t *imm_factor*)

6.26.3. 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].

6.26.3.1. QINCW (scalar, multiplier)

Instances
int32_t **svqincw**[_n_s32](int32_t *op*, uint64_t *imm_factor*) int64_t **svqincw**[_n_s64](int64_t *op*, uint64_t *imm_factor*) uint32_t **svqincw**[_n_u32](uint32_t *op*, uint64_t *imm_factor*) uint64_t **svqincw**[_n_u64](uint64_t *op*, uint64_t *imm_factor*)

6.26.3.2. QINCW (scalar, pattern, multiplier)

Instances
int32_t **svqincw_pat**[_n_s32](int32_t *op*, svpattern *pattern*, uint64_t *imm_factor*) int64_t **svqincw_pat**[_n_s64](int64_t *op*, svpattern *pattern*, uint64_t *imm_factor*) uint32_t **svqincw_pat**[_n_u32](uint32_t *op*, svpattern *pattern*, uint64_t *imm_factor*) uint64_t **svqincw_pat**[_n_u64](uint64_t *op*, svpattern *pattern*, uint64_t *imm_factor*)

6.26.3.3. QINCW (vector, multiplier)

Instances
svint32_t **svqincw**[_s32](svint32_t *op*, uint64_t *imm_factor*) svuint32_t **svqincw**[_u32](svuint32_t *op*, uint64_t *imm_factor*)

6.26.3.4. QINCW (vector, pattern, multiplier)

Instances
svint32_t **svqincw_pat**[_s32](svint32_t *op*, svpattern *pattern*, uint64_t *imm_factor*) svuint32_t **svqincw_pat**[_u32](svuint32_t *op*, svpattern *pattern*, uint64_t *imm_factor*)

6.26.4. 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].

6.26.4.1. QINCD (scalar, multiplier)

Instances
int32_t **svqincd**[_n_s32](int32_t *op*, uint64_t *imm_factor*) int64_t **svqincd**[_n_s64](int64_t *op*, uint64_t *imm_factor*) uint32_t **svqincd**[_n_u32](uint32_t *op*, uint64_t *imm_factor*) uint64_t **svqincd**[_n_u64](uint64_t *op*, uint64_t *imm_factor*)

6.26.4.2. QINCD (scalar, pattern, multiplier)

Instances
int32_t **svqincd_pat**[_n_s32](int32_t *op*, svpattern *pattern*, uint64_t *imm_factor*) int64_t **svqincd_pat**[_n_s64](int64_t *op*, svpattern *pattern*, uint64_t *imm_factor*) uint32_t **svqincd_pat**[_n_u32](uint32_t *op*, svpattern *pattern*,

Instances
uint64_t *imm_factor*) uint64_t **svqincd_pat**[_n_u64](uint64_t *op*, svpattern *pattern*, uint64_t *imm_factor*)

6.26.4.3. 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*)

6.26.4.4. QINCD (vector, pattern, multiplier)

Instances
svint64_t **svqincd_pat**[_s64](svint64_t *op*, svpattern *pattern*, uint64_t *imm_factor*) svuint64_t **svqincd_pat**[_u64](svuint64_t *op*, svpattern *pattern*, uint64_t *imm_factor*)

6.26.5. 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.

6.26.5.1. 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*)

6.26.5.2. 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*) 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*)

6.26.6. 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].

6.26.6.1. QDECB (scalar, multiplier)

Instances
int32_t **svqdecb**[_n_s32](int32_t *op*, uint64_t *imm_factor*) int64_t **svqdecb**[_n_s64](int64_t *op*, uint64_t *imm_factor*) uint32_t **svqdecb**[_n_u32](uint32_t *op*, uint64_t *imm_factor*) uint64_t **svqdecb**[_n_u64](uint64_t *op*, uint64_t *imm_factor*)

6.26.6.2. QDECB (scalar, pattern, multiplier)

Instances
int32_t **svqdecb_pat**[_n_s32](int32_t *op*, svpattern *pattern*, uint64_t *imm_factor*) int64_t **svqdecb_pat**[_n_s64](int64_t *op*, svpattern *pattern*, uint64_t *imm_factor*) uint32_t **svqdecb_pat**[_n_u32](uint32_t *op*, svpattern *pattern*, uint64_t *imm_factor*) uint64_t **svqdecb_pat**[_n_u64](uint64_t *op*, svpattern *pattern*, uint64_t *imm_factor*)

6.26.7. 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].

6.26.7.1. 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*)

6.26.7.2. QDECH (scalar, pattern, multiplier)

Instances
int32_t **svqdech_pat**[_n_s32](int32_t *op*, svpattern *pattern*,

Instances
uint64_t *imm_factor*) int64_t **svqdech_pat**[_n_s64](int64_t *op*, svpattern *pattern*, uint64_t *imm_factor*) uint32_t **svqdech_pat**[_n_u32](uint32_t *op*, svpattern *pattern*, uint64_t *imm_factor*) uint64_t **svqdech_pat**[_n_u64](uint64_t *op*, svpattern *pattern*, uint64_t *imm_factor*)

6.26.7.3. QDECH (vector, multiplier)

Instances
svint16_t **svqdech**[_s16](svint16_t *op*, uint64_t *imm_factor*) svuint16_t **svqdech**[_u16](svuint16_t *op*, uint64_t *imm_factor*)

6.26.7.4. QDECH (vector, pattern, multiplier)

Instances
svint16_t **svqdech_pat**[_s16](svint16_t *op*, svpattern *pattern*, uint64_t *imm_factor*) svuint16_t **svqdech_pat**[_u16](svuint16_t *op*, svpattern *pattern*, uint64_t *imm_factor*)

6.26.8. 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].

6.26.8.1. QDECW (scalar, multiplier)

Instances
int32_t **svqdecw**[_n_s32](int32_t *op*, uint64_t *imm_factor*) int64_t **svqdecw**[_n_s64](int64_t *op*, uint64_t *imm_factor*) uint32_t **svqdecw**[_n_u32](uint32_t *op*, uint64_t *imm_factor*) uint64_t **svqdecw**[_n_u64](uint64_t *op*, uint64_t *imm_factor*)

6.26.8.2. QDECW (scalar, pattern, multiplier)

Instances
int32_t **svqdecw_pat**[_n_s32](int32_t *op*, svpattern *pattern*, uint64_t *imm_factor*) int64_t **svqdecw_pat**[_n_s64](int64_t *op*, svpattern *pattern*, uint64_t *imm_factor*) uint32_t **svqdecw_pat**[_n_u32](uint32_t *op*, svpattern *pattern*, uint64_t *imm_factor*) uint64_t **svqdecw_pat**[_n_u64](uint64_t *op*, svpattern *pattern*,

Instances
uint64_t *imm_factor*)

6.26.8.3. 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*)

6.26.8.4. QDECW (vector, pattern, multiplier)

Instances
svint32_t **svqdecw_pat**[_s32](svint32_t *op*, svpattern *pattern*, uint64_t *imm_factor*) svuint32_t **svqdecw_pat**[_u32](svuint32_t *op*, svpattern *pattern*, uint64_t *imm_factor*)

6.26.9. 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].

6.26.9.1. 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*)

6.26.9.2. QDECD (scalar, pattern, multiplier)

Instances
int32_t **svqdecd_pat**[_n_s32](int32_t *op*, svpattern *pattern*, uint64_t *imm_factor*) int64_t **svqdecd_pat**[_n_s64](int64_t *op*, svpattern *pattern*, uint64_t *imm_factor*) uint32_t **svqdecd_pat**[_n_u32](uint32_t *op*, svpattern *pattern*, uint64_t *imm_factor*) uint64_t **svqdecd_pat**[_n_u64](uint64_t *op*, svpattern *pattern*, uint64_t *imm_factor*)

6.26.9.3. 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*)

6.26.9.4. QDECD (vector, pattern, multiplier)

Instances
`svint64_t svqdecd_pat[_s64](svint64_t op, svpattern pattern, uint64_t imm_factor)`
`svuint64_t svqdecd_pat[_u64](svuint64_t op, svpattern pattern, uint64_t imm_factor)`

6.26.10. 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.

6.26.10.1. QDECP (scalar)

Instances
`int32_t svqdecp[_n_s32]_b8(int32_t op, svbool_t pg)`
`int32_t svqdecp[_n_s32]_b16(int32_t op, svbool_t pg)`
`int32_t svqdecp[_n_s32]_b32(int32_t op, svbool_t pg)`
`int32_t svqdecp[_n_s32]_b64(int32_t op, svbool_t pg)`
`int64_t svqdecp[_n_s64]_b8(int64_t op, svbool_t pg)`
`int64_t svqdecp[_n_s64]_b16(int64_t op, svbool_t pg)`
`int64_t svqdecp[_n_s64]_b32(int64_t op, svbool_t pg)`
`int64_t svqdecp[_n_s64]_b64(int64_t op, svbool_t pg)`
`uint32_t svqdecp[_n_u32]_b8(uint32_t op, svbool_t pg)`
`uint32_t svqdecp[_n_u32]_b16(uint32_t op, svbool_t pg)`
`uint32_t svqdecp[_n_u32]_b32(uint32_t op, svbool_t pg)`
`uint32_t svqdecp[_n_u32]_b64(uint32_t op, svbool_t pg)`
`uint64_t svqdecp[_n_u64]_b8(uint64_t op, svbool_t pg)`
`uint64_t svqdecp[_n_u64]_b16(uint64_t op, svbool_t pg)`
`uint64_t svqdecp[_n_u64]_b32(uint64_t op, svbool_t pg)`
`uint64_t svqdecp[_n_u64]_b64(uint64_t op, svbool_t pg)`

6.26.10.2. QDECP (vector)

Instances
`svint16_t svqdecp[_s16](svint16_t op, svbool_t pg)`
`svint32_t svqdecp[_s32](svint32_t op, svbool_t pg)`
`svint64_t svqdecp[_s64](svint64_t op, svbool_t pg)`
`svuint16_t svqdecp[_u16](svuint16_t op, svbool_t pg)`
`svuint32_t svqdecp[_u32](svuint32_t op, svbool_t pg)`
`svuint64_t svqdecp[_u64](svuint64_t op, svbool_t pg)`

6.27. Reinterpreting data

6.27.1. 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.

6.27.1.1. 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*)
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*)
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*)
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*)
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*)
svuint8_t **svreinterpret_u8**[_f16](svfloat16_t *op*)
svuint8_t **svreinterpret_u8**[_f32](svfloat32_t *op*)
svuint8_t **svreinterpret_u8**[_f64](svfloat64_t *op*)
svuint16_t **svreinterpret_u16**[_s8](svint8_t *op*)

Instances

```
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)
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)
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)
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)
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)
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)
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)
```


Instances
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*)

7. Mapping of SVE instructions to functions

7.1. List of instructions

This section contains a list of all SVE 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”

ADD (immediate)
ADD (vectors, predicated)
ADD (vectors, unpredicated)

> Section 6.7.1, “ADD: Modular integer addition”

ADDPL
ADDVL

> No direct support; see Section 7.3, “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, “ADRD: Compute vector address for 64-bit data”

AND (immediate)
AND (vectors, predicated)
AND (vectors, unpredicated)

> Section 6.8.1, “AND: Bitwise AND”

AND (predicates)
ANDS

> Section 6.22.2, “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”

BIC (immediate)
BIC (vectors, predicated)
BIC (vectors, unpredicated)

Section 6.8.2, “BIC: Bitwise AND NOT”

BIC (predicates)
BICS

Section 6.22.3, “BIC: Predicate AND NOT”

BRKA
BRKAS

Section 6.22.10, “BRKA: Break after first true condition”

BRKB
BRKBS

Section 6.22.11, “BRKB: Break before first true condition”

BRKN
BRKNS

Section 6.22.12, “BRKN: Propagate break to next partition”

BRKPA
BRKPAS

Section 6.22.13, “BRKPA: Propagate and break after first true condition”

BRKPB
BRKPBS

Section 6.22.14, “BRKPB: Propagate and break before first true condition”

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”

CMPEQ (immediate)
CMPEQ (vectors)
CMPEQ (wide elements)

Section 6.11.1, “CMPEQ: Integer compare equal”

CMPGE (immediate)
CMPGE (vectors)
CMPGE (wide elements)

Section 6.11.5, “CMPGE: 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”

CMPLE (immediate)
CMPLE (vectors)
CMPLE (wide elements)

Section 6.11.4, “CMPLE: 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, “CMPLE: 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 6.25.2, “CNTB: Count the number of 8-bit elements in a pattern”

CNTD

Section 6.25.5, “CNTD: Count the number of 64-bit elements in a pattern”

CNTH

Section 6.25.3, “CNTH: Count the number of 16-bit elements in a pattern”

CNTP

Section 6.25.1, “CNTP: Count active elements”

CNTW

Section 6.25.4, “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 7.2, “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 7.3, “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 6.21.3, “DUP: Duplicate boolean value”
Section 6.21.4, “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 6.22.8, “EOR: Predicate exclusive OR”

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”

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”

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”

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”

FMAXNMV

Section 6.17.4, “MAXNMV: Floating-point maximum number reduction”

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”

FMINNMV

Section 6.17.6, “MINNMV: Floating-point minimum number reduction”

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)”

FMLS (indexed)
FMLS (vectors)

Section 6.16.12, “MLS: Fused floating-point subtraction of product (minuend first)”

FMSB

Section 6.16.11, “MSB: Fused floating-point subtraction of product (multiplicand first)”

FMOV (immediate, predicated)
FMOV (immediate, unpredicated)
FMOV (zero, predicated)
FMOV (zero, unpredicated)

Section 6.6.1, “DUP: Duplicate scalar value”

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”

FRSQRTE

Section 6.16.34, “RSQRTE: Floating-point reciprocal square root estimate”

FRSQRTS

Section 6.16.35, “RSQRTS: 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”

INCB
INCD (scalar)
INCD (vector)
INCH (scalar)
INCH (vector)
INCP (scalar)
INCP (vector)
INCW (scalar)
INCW (vector)

No direct support; see Section 7.3, “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 `svdup`s in which the scalar value comes from memory; see Section 6.6.1, “DUP: Duplicate scalar value”.

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 6.21.4, “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”

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)”

MLA

Section 6.7.10, “MLA: Integer addition of product (addend first)”

MLS

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 6.22.1, “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)

Section 6.7.7, “MUL: Integer multiplication, returning low half”

NAND
NANDS

Section 6.22.4, “NAND: Predicate NAND”

NEG

Section 6.7.18, “NEG: Integer negation”

NOR
NORS

Section 6.22.7, “NOR: Predicate NOR”

NOT (vector)

Section 6.8.5, “NOT: Bitwise inverse”

NOT (predicate)
NOTS

Section 6.22.9, “NOT: Predicate NOT”

ORN (predicates)
ORNS

Section 6.22.6, “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 6.22.5, “ORR: Predicate OR”

ORV

Section 6.10.5, “ORV: Integer OR reduction”

PFALSE

Section 6.21.2, “PFALSE: Return an all-false predicate”

PFIRST

Section 6.22.15, “PFIRST: Set first active predicate element to true”

PNEXT

Section 6.22.16, “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 6.23.1, “PTEST: Test active elements”

PTRUE
PTRUES

Section 6.21.1, “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”

RBIT

Section 6.15.1, “RBIT: Reverse bits within elements”

RDFFR (predicated)
RDFFR (unpredicated)
RDFFRS

Section 6.24.1, “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 6.25.2, “CNTB: Count the number of 8-bit elements in a pattern”
Section 6.25.3, “CNTH: Count the number of 16-bit elements in a pattern”
Section 6.25.4, “CNTW: Count the number of 32-bit elements in a pattern”
Section 6.25.5, “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”

REVH

Section 6.15.3, “REVH: Reverse halfwords within elements”

REVW

Section 6.15.4, “REVW: Reverse words within elements”

SABD

Section 6.7.6, “ABD: Integer absolute difference”

SADDV

Section 6.10.1, “ADDV: Integer addition reduction”

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 6.24.2, “SETFFR: Set the first-fault register”

SMAX (immediate)
SMAX (vectors)

Section 6.7.16, “MAX: Integer maximum”

SMAXV

Section 6.10.2, “MAXV: Integer maximum reduction”

SMIN (immediate)
SMIN (vectors)

Section 6.7.17, “MIN: Integer minimum”

SMINV

Section 6.10.3, “MINV: Integer minimum reduction”

SMULH

Section 6.7.8, “MULH: Integer multiplication, returning high half”

SPLICE

Section 6.20.6, “SPLICE: Splice two vectors under predicate control”

SQADD (immediate)
SQADD (vectors)

Section 6.7.2, “QADD: Saturating integer addition”

SQDECB

Section 6.26.6, “QDECB: Saturating decrement by a multiple of `svcntb`”

SQDECD (scalar)
SQDECD (vector)

Section 6.26.9, “QDECD: Saturating decrement by a multiple of `svcntd`”

SQDECH (scalar)
SQDECH (vector)

Section 6.26.7, “QDECH: Saturating decrement by a multiple of `svcnth`”

SQDECP (scalar)
SQDECP (vector)

Section 6.26.10, “QDECP: Saturating decrement by a multiple of `svcntp`”

SQDECW (scalar)
SQDECW (vector)

Section 6.26.8, “QDECW: Saturating decrement by a multiple of `svcntw`”

SQINCB

Section 6.26.1, “QINCB: Saturating increment by a multiple of `svcntb`”

SQINCD (scalar)
SQINCD (vector)

Section 6.26.4, “QINCD: Saturating increment by a multiple of `svcntd`”

SQINCH (scalar)
SQINCH (vector)

Section 6.26.2, “QINCH: Saturating increment by a multiple of `svcnth`”

SQINCP (scalar)
SQINCP (vector)

Section 6.26.5, “QINCP: Saturating increment by a multiple of `svcntp`”

SQINCW (scalar)
SQINCW (vector)

Section 6.26.3, “QINCW: Saturating increment by a multiple of `svcntw`”

SQSUB (immediate)
SQSUB (vectors)

Section 6.7.5, “QSUB: Saturating integer subtraction”

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”

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”

SUBR (immediate)
SUBR (vectors)

Section 6.7.4, “SUBR: Modular integer subtraction, reversed”

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”

TRN1 (predicates)
TRN1 (vectors)

Section 6.20.13, “TRN1: Interleave even elements from two inputs”

TRN2 (predicates)
TRN2 (vectors)

Section 6.20.14, “TRN2: Interleave odd elements from two inputs”

UABD

Section 6.7.6, “ABD: Integer absolute difference”

UADDV

Section 6.10.1, “ADDV: Integer addition reduction”

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”

UMAX (immediate)
UMAX (vectors)

Section 6.7.16, “MAX: Integer maximum”

UMAXV

Section 6.10.2, “MAXV: Integer maximum reduction”

UMIN (immediate)
UMIN (vectors)

Section 6.7.17, “MIN: Integer minimum”

UMINV

Section 6.10.3, “MINV: Integer minimum reduction”

UMULH

Section 6.7.8, “MULH: Integer multiplication, returning high half”

UQADD (immediate)
UQADD (vectors)

Section 6.7.2, “QADD: Saturating integer addition”

UQDECB

Section 6.26.6, “QDECB: Saturating decrement by a multiple of `svcntb`”

UQDECD (scalar)
UQDECD (vector)

Section 6.26.9, “QDECD: Saturating decrement by a multiple of `svcntd`”

UQDECH (scalar)
UQDECH (vector)

Section 6.26.7, “QDECH: Saturating decrement by a multiple of `svcnth`”

UQDECP (scalar)
UQDECP (vector)

Section 6.26.10, “QDECP: Saturating decrement by a multiple of `svcntp`”

UQDECW (scalar)
UQDECW (vector)

Section 6.26.8, “QDECW: Saturating decrement by a multiple of `svcntw`”

UQINCB

Section 6.26.1, “QINCB: Saturating increment by a multiple of `svcntb`”

UQINCD (scalar)
UQINCD (vector)

Section 6.26.4, “QINCD: Saturating increment by a multiple of `svcntd`”

UQINCH (scalar)
UQINCH (vector)

Section 6.26.2, “QINCH: Saturating increment by a multiple of `svcnth`”

UQINCP (scalar)
UQINCP (vector)

Section 6.26.5, “QINCP: Saturating increment by a multiple of `svcntp`”

UQINCW (scalar)
UQINCW (vector)

Section 6.26.3, “QINCW: Saturating increment by a multiple of `svcntw`”

UQSUB (immediate)
UQSUB (vectors)

Section 6.7.5, “QSUB: Saturating integer subtraction”

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”

UZP2 (predicates)
UZP2 (vectors)

Section 6.20.18, “UZP2: Select odd elements from two inputs”

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”

WRFFR

Section 6.24.3, “WRFFR: Write to the first-fault register”

ZIP1 (predicates)
ZIP1 (vectors)

Section 6.20.19, “ZIP1: Interleave elements from low halves of two inputs”

ZIP2 (predicates)
ZIP2 (vectors)

Section 6.20.20, “ZIP2: Interleave elements from high halves of two inputs”

7.2. 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 != 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.

7.3. 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 6.25.2, “CNTB: Count the number of 8-bit elements in a pattern”) or from a predicate (Section 6.25.1, “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.

