Support for Scalable Vector Architectures in LLVM IR

ARM Manchester Design Center

4th November 2016

Background

ARMv8-A Scalable Vector Extensions is a vector ISA extension for AArch64, featuring lane predication, scatter-gather, and speculative loads. It is intended to scale with hardware such that the same binary running on a processor with wider vector registers can take advantage of the increased compute power without recompilation. This document describes changes made to LLVM that allow its vectorizer to better target SVE.

As the vector length is no longer a statically known value, the way in which the LLVM vectorizer generates code required modifications such that certain values are now runtime evaluated expressions. The following section documents the additional IR instructions needed to achieve this. During the design of these extensions, we endeavoured to strike a balance between elegant and succinct design and the pragmatic realities of generating efficient code for the SVE architecture. We believe that the current design has a relatively small impact on the IR and type system given the unusual characteristics of the target.

Types

Overview:

IR Class Changes:

To represent a vector of unknown length a scaling property is added to the `VectorType` class whose element count becomes an unknown multiple of a known minimum element count. To be specific `unsigned NumElements` is replaced by `class ElementCount` with its two members:

- `unsigned Min`: the minimum number of elements.
- `bool Scalable`: is the element count an unknown multiple of `Min`?

For non-scalable vectors (`Scalable=false`) the scale is assumed to be one and thus `Min` becomes identical to the original `NumElements` property.

This mixture of static and runtime quantities allow us to reason about the relationship between different scalable vector types without knowing their exact length.

IR Interface:

```
static VectorType *get(Type *ELType, ElementCount EC);
// Legacy interface whilst code is migrated.
static VectorType *get(Type *ElType, unsigned NumEls, bool Scalable=false);
```

```
ElementCount getElementCount();
bool isScalable();
```

IR Textual Form:

The textual IR for vectors becomes:

```
<[n x] <m> x <type>>
```

where `type` is the scalar type of each element, `m` corresponds to `Min` and the optional string literal `n x` signifies `Scalable` is *true* when present, false otherwise. The textual IR for non-scalable vectors is thus unchanged.

Scalable vectors with the same `Min` value have the same number of elements, and the same number of bytes if `Min * sizeof(type)` is the same:

`<n x 4 x i32>` and `<n x 4 x i8>` have the same number of elements.

`<n x 4 x i32>` and `<n x 8 x i16>` have the same number of bytes.

SelectionDAG

New scalable vector MVTs are added, one for each existing vector type. Scalable vector MVTs are modelled in the same way as the IR. Hence, `<n x 4 x i32>` becomes `nxv4i32`.

MVT Interface:

```
static MVT getVectorVT(MVT VT, ElementCount EC);
bool isScalableVector() const;
static mvt_range integer_scalable_valuetypes();
static mvt_range fp_scalable_valuetypes();
```

To minimise the effort required for common code to preserve the scalable flag we extend the helper functions within MVT/EVT classes to cover more cases. For example:

```
/// Return a VT for a vector type whose attributes match ourselves
/// but with an element type chosen by the caller.
EVT changeVectorElementType(EVT EltVT)`
```

Instructions

The majority of instructions work seamlessly when applied to scalable vector types. However, on occasion assumptions are made that allow vectorization logic to be reduced directly to constants completely bypassing the IR (e.g. when the element count is known). These assumption are generally unsafe for scalable vectors which forces a requirement to express more logic in IR. To help with this the following instruction are added to the IR.

elementcount

Syntax:

```
<result> = elementcount <n x m x ty> <v1> as <ty2>
```

Overview:

This instruction returns the actual number of elements in the input vector of type `<n x m x ty>` as the scalar type `<ty2>`. This is primarily used to increment induction variables and replaces many uses of `VF` within the loop vectorizer.

IRBuilder Interface:

```
Value *CreateElementCount(Type *Ty, Value *V, const Twine &Name = "");
```

Fixed-Width Behaviour:

A constant with the value of `Min` is created.

SelectionDAG:

See *ISD::ELEMENT_COUNT*.

seriesvector

Syntax:

```
<result> = seriesvector <ty> <v1>, <v2> as <n x m x ty>
```

Overview:

This instruction returns a vector of type `<n x m x ty>` with elements forming the arithmetic series:

```
elt[z] = v1 + z * v2
```

where `z` is the element index and `<ty>` a scalar type. This is primarily used to represent a vector of induction values leading to:

- Predicate creation using vector compares for fully predicated loops (see also: *propff*, *test*).
- Creating offset vectors for gather/scatter via `getelementptr`.
- Creating masks for `shufflevector`.

For the following loop, a seriesvector instruction based on `i` is used to create the data vector to store:

```
unsigned a[LIMIT];

for (unsigned i = 0; i < LIMIT; i++) {
  a[i] = i;
}
```

seriesvector instructions can optionally have *NUW* and *NSW* flags attached to them. The semantics of these flags are intended to match those of the usual scalar instruction wrap flags, but apply element-wise to the result vector. If any element addition of the current value and step results in a signed or unsigned overflow with the corresponding flag set, then the result is a poison value for the entire vector. However, this feature is not currently used.

IRBuilder Interface:

```
Value *CreateSeriesVector(VectorType::ElementCount EC, Value *Start,
                          Value* Step, const Twine &Name = "",
                          bool HasNUW = false, bool HasNSW = false);
```

Fixed-Width Behaviour:

A constant vector is created with the same arithmetic series.

SelectionDAG:

See *ISD::SERIES_VECTOR*.

test

Syntax:

```
<result> = test <cond> <ty> <v1>
```

Overview:

This instruction returns a scalar boolean value based on the comparison of a boolean or vector boolean operand. It allows us to reason about a value as a whole rather than the per scalar comparison obtained from say *icmp*. The most common use is testing the result of *icmp*/*fcmp*.

We use this for the controlling predicate of fully predicated loops. Loops with a termination condition as follows:

```
for (i = 0; i < LIMIT; ++i)
```

are handled by testing their control predicate for `first true` to signify another iteration is required.

Support for scalar booleans is simply to provide symmetry so that all variants of *icmp*/*fcmp* can be passed as input to *test*.

Supported Conditions:

- all false
- all true
- any false
- any true
- first false
- first true
- last false
- last true

IRBuilder Interface:

```
Value *CreateTest(TestInst::Predicate P, Value *V, const Twine &Name = "");
```

Fixed-Width Behaviour

Same as scalable.

SelectionDAG:

See *ISD::TEST_VECTOR*.

propff

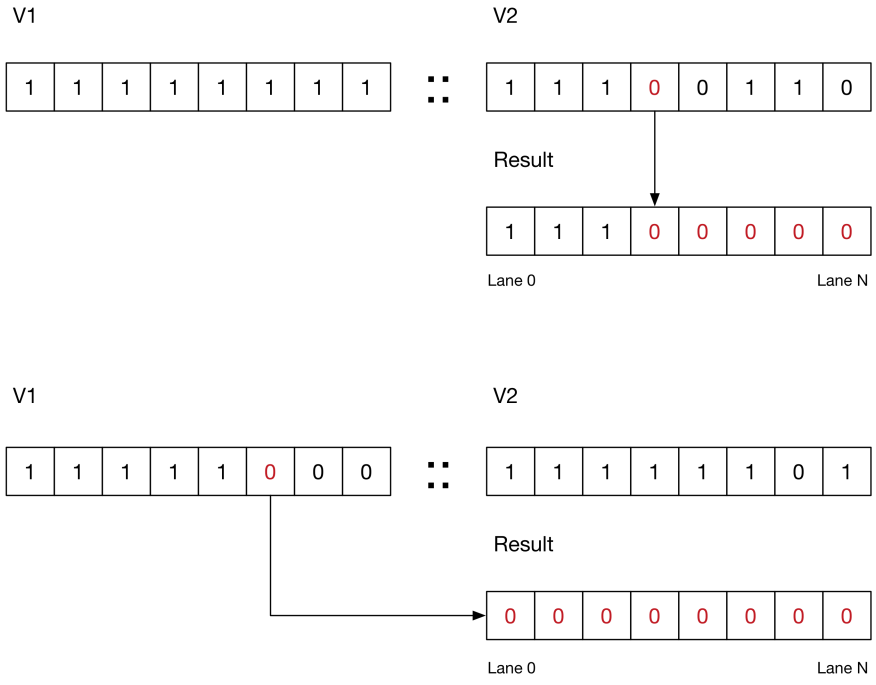
Syntax:

```
<result> = propff <ty> <v1>, <v2>
```

Overview:

This instruction creates a partitioned boolean vector based on the position of the first boolean false value in the concatenation of input boolean vectors `v1` and `v2`. Given vectors of length `k`, an element `x[i]` in the result vector is true if, and only if, each element `y[0]...y[i+k]` in the concatenated input vector `y=v1:v2` is true.

The following examples show the results for a full `v1` and a `v2` which has reached a termination condition, and a `v1` which previously reached a termination condition with any `v2`.



A core use of this instruction is to protect against integer overflow that can occur when maintaining the induction vector of a fully predicated loop. For SVE the issue is further compounded with its support of non-power-of-2 vector lengths.

We believe *propff* to be the least complex partitioning instruction to provide sufficient abstraction yet achieve the expected SVE code quality. Although other partitioning schemes can be modelled using *propff* we feel a more capable partitioning instruction is worth highlighting because it can simplify the vectorization of more loops.

IRBuilder Interface:

```
Value *CreatePropFF(Value* P1, Value *P2, const Twine &Name = "");
```

Fixed-Width Behaviour:

Same as scalable.

SelectionDAG:

See *ISD::PROPAGATE_FIRST_ZERO*.

shufflevector

Syntax:

```
<result> = shufflevector <ty1> <v1>, <ty1> <v2>, <ty2> <mask>
```

Overview:

Not a new instruction but *shufflevector* is extended to accept non-constant masks. For code that expects the original restriction `ShuffleVectorInst::getMaskValue` has changed to guard against unsafe uses.

```
// [old] Returns the mask value.
static int getMaskValue(Constant *Mask, unsigned i);
// [new] Returns true when Result contains the mask value, false otherwise.
static bool getMaskValue(Value *Mask, unsigned i, int &Result);
```

Similar changes are made to related interfaces and their users.

IRBuilder Interface:

```
Value *CreateShuffleVector(Value *V1, Value *V2, Value *Mask,
                           const Twine &Name = "");
```

Fixed-Width Behaviour:

No change.

SelectionDAG:

See *ISD::VECTOR_SHUFFLE_VAR*.

Constants

Scalable vectors of known constants cannot be represented within LLVM due to their unknown element count. Optimizations performed on scalable vectors become much more reliant on `ConstantExpr`s. Most of the instructions talked about in this document also have a matching `ConstantExpr` with the most common constant folds duplicated to work with them.

This is most prevalent when considering vector splats. These are simulated via a sequence of `insertelement`and `shufflevector` that are usually resolved to a `Constant`. For scalable vectors this cannot be done and the original sequence is maintained. To mitigate this, `PatternMatch` is extended to better support `ConstantExpr`s along with new helpers like:

```
#define m_SplatVector(X) \
  m_ShuffleVector( \
    m_InsertElement(m_Undef(), X, m_Zero()), \
    m_Value(), \
    m_Zero()) \
```

Zero is the exception with `zeroinitializer` applying equally well to scalable and non-scalable vectors.

Predicated floating point arithmetic

When a loop is fully predicated it becomes necessary to have masked versions of faulting instructions. Today we use the existing masked memory intrinsics to ensure we do not access memory that would not have been accessed by the original scalar loop.

The same principle applies to floating point instructions whereby we should not trigger an exception that would not have been produced by the original scalar loop.

We don't handle this today but are seeking advice as to the preferred method. Options include:

1. Add masked versions of the affected instructions.
2. Extend existing instructions to take a predicate.
3. Construct IR that contains "safe" values for undefined locations.

Given the choice, Option 1 is our preferred route as it's in line with the introduction of the masked memory intrinsics. Although, given the greater potential for optimization, instruction forms may be preferable to intrinsics.

See also *Predicated floating point intrinsics*

Intrinsics

llvm.masked.spec.load

Syntax:

```
<result> = call {<data>, <pred>} @llvm.masked.spec.load.<type>(<ptr>, <align>,
<mask>, <merge>)
```

Overview:

This intrinsic represents a load whereby only the first active lane is expected to succeed with remaining lanes loaded speculatively. Along with the data this intrinsic returns a predicate indicating which lanes of data are valid.

Interface:

```
CallInst *CreateMaskedSpecLoad(Value *Ptr, unsigned Align, Value *Mask,
                               Value *PassThru = 0, const Twine &Name = "");
```

Fixed-Width Behaviour:

Same as scalable.

llvm.ctvpop

Syntax:

Overview:

This intrinsic returns a count of the set bits across a complete vector. It's most useful when operating on predicates as it allows a portable way to create predicate vectors that are partitioned differently (e.g. including the lane with the first change, rather than *propff*'s exclusive behaviour).

Interface:

```
CallInst *CreateCntVPop(Value *Vec, const Twine &Name);
```

Fixed-Width Behaviour

Same as scalable. However, it can also be modelled using `*llvm.ctpop*` followed by a scalarized horizontal reduction. This is not possible with scalable vectors because scalarization is not an option.

horizontal reductions

Loops containing reductions are first vectorized as if no such reduction exists, thus building a vector of accumulated results. When exiting the vector loop a final in-vector reduction is done by effectively scalarizing the operation across each of the result vector's elements.

For scalable vectors, whose element count is unknown, such scalarization is not possible. For this reason we introduce target specific intrinsics to support common reductions (e.g. horizontal add), along with TargetTransformInfo extensions to query their availability. If a scalable vector loop requires a reduction that's not provided by the target its vectorization is prevented.

When fully predicated vectorization is required, additional work is done within the vector loop to ensure inactive lanes don't affect the accumulated result. See also Predicated floating point arithmetic.

Predicated floating point intrinsics

Libcalls not directly supported by the code generator must be serialized (See here). Also, for fully predicated loops the scalar calls must only occur for active lanes so we introduce predicated versions of the most common routines (e.g. `*llvm.pow`). This is mostly transparent to the loop vectorizer beyond the requirement to pass an extra parameter.

TTI Interface:

```
bool canReduceInVector(const RecurrenceDescriptor &Desc, bool NoNaN) const;
Value* getReductionIntrinsic(IRBuilder<> &Builder,
                             const RecurrenceDescriptor& Desc, bool NoNaN,
                             Value* Src) const;
```

SelectionDAG Nodes

ISD::BUILD_VECTOR

```
BUILD_VECTOR(ELT0, ELT1, ELT2, ELT3,...)
```

This node takes as input a set of scalars to concatenate into a single vector. By its very nature this node is incompatible with scalable vectors because we don't know how many scalars it will takes to fill one. All common code using this node has either been rewritten or bypassed.

ISD::ELEMENT_COUNT

```
ELEMENT_COUNT(TYPE)
```

See *elementcount*.

ISD::EXTRACT_SUBVECTOR

```
EXTRACT_SUBVECTOR(VECTOR, IDX)
```

Usage of this node is typically linked to legalization where it's used to split vectors. The index parameter is often absolute and proportional to the input vector's element count.

For scalable vectors an absolute index makes little sense. We have changed this parameter's meaning to become implicitly multiplied by `n` to match its main usage.

The change has no effect when applied to non-scalable vectors, because `n == 1`. No target specific code is affected and in many cases common code becomes compatible with scalable vectors. For example:

```
nxv2i64 extract_subvector(nxv4i64, 2)
```

The real first lane becomes `n * 2`, resulting in the extraction of the top half of the input vector. This maintains the intension of the original code for both scalable and non-scalable vectors.

For common code that truly requires an absolute index we recommend a new distinct ISD node to better differentiate such patterns.

ISD::INSERT_SUBVECTOR

```
INSERT_SUBVECTOR(VECTOR1, VECTOR2, IDX)
```

We have made the equivalent change to this node's index parameter to match the behaviour of ISD::EXTRACT_SUBVECTOR.

ISD::PROPAGATE_FIRST_ZERO

```
PROPAGATE_FIRST_ZERO(VEC1, VEC2)
```

See *propff*.

ISD::SERIES_VECTOR

```
SERIES_VECTOR(INITIAL, STEP)
```

See *seriesvector*.

ISD::SPLAT_VECTOR

```
SPLAT_VECTOR(VAL)
```

Within the code generator a splat is represented by ISD::BUILD_VECTOR and is thus incompatible with scalable vectors.

We initially made use of ISD::SERIES_VECTOR with a zero stride but that brings with it a requirement for ISD::SERIES_VECTOR to support floating-point types. For this reason we created an explicit node that also allowed less complex looking legalization/selection code.

ISD::TEST_VECTOR

```
TEST_VECTOR(VEC, PRED)
```

`VEC` is the boolean vector being tested, and `PRED` is a `TestCode` enum under the `llvm::ISD` namespace which contains all the supported conditions.

See *test*.

ISD::VECTOR_SHUFFLE_VAR

```
VECTOR_SHUFFLE_VAR(VEC1, VEC2, VEC3)
```

See *shufflevector*.

AArch64 Specific Changes

Instruction Selection

In order to allow proper instruction selection there must be a direct mapping from MVTs to SVE registers. SVE data registers have a length in multiples of 128bits (with 128bits being the minimum) and predicate registers have a bit for every byte of a data register.

Given the 128bit minimum we map scalable vector MVTs whose static component is also 128bits (e.g. MVT::nxv4i32) directly to SVE data registers. Scalable vector MVTs with an `i1` element type and a static element count of 16 (128/8 = 16) or fewer (e.g. MVT::nxv4i1) are mapped to SVE predicate registers.

All other integer MVTs are considered illegal and are either split or promoted accordingly. A similar rule applies to vector floating point MVTs but those types whose static component is less that 128bits (MVT::nx2f32) are also mapped directly to SVE data registers but in a form whereby elements are effectively interleaved with enough undefined elements to fulfil the 128bit requirement.

We do this so that predicate bits correctly align to their data counterpart. For example, for all vector MVTs with two elements, a predicate of nxv2i1 is used, regardless of the data vector's element type.

Stack Regions for SVE Objects

As SVE registers have an unknown size their associated fill/spill instructions require an offset that will be implicitly scaled by the vector length instead of bytes or element size. To accommodate this we introduce the concept of `Stack Regions` that are areas on the stack associated with specific data types or register classes.

Each `Stack Region` controls its own allocation, deallocation and internal offset calculations. For SVE we create separate `Stack Region`s for its data and predicate registers. Objects not belonging to a `Stack Region` use a default so that existing targets are not affected.

SVE-Specific Optimizations

The following SVE specific IR transformation passes are added to better guide code generation. In some instances this has affected how loops are vectorized because it's assumed they will occur. They also point to work other target's may wish to consider.

SVE Post-Vectorization Optimization Pass

This pass is responsible for converting generically vectorized loops into a form that more closely resembles SVE's style of vectorization. For example, rewriting a vectorized loop's control flow to utilize SVE's `while` instruction.

NOTE: This pass is very sensitive to the wrap flags (i.e. NSW/NUW). Much effort has gone into ensuring they are preserved during vectorization to the extent of introducing `llvm.assume` calls when beneficial.

SVE Expand Libcall Pass

In order to keep the loop vectorizer generic we maintain the ability to vectorize libcalls we know the code generator cannot handle because scalable vectors cannot be scalarized. This pass serializes such calls by introducing loops using SVE's predicate iteration instructions.

Note that although such serialization can be achieved generically using `extractelement`/`insertelement`, our experiments showed no route to efficient code generation.

SVE Addressing Modes Pass

This pass alleviates the negative effects of `Loop Strength Reduction` on address computations for SVE targeted loops, leading to better code quality.

Full Example

The following section shows how a C loop with a sum reduction is represented in scalable IR (assuming -Ofast) and the final generated code.

```
int SimpleReduction(int *a, int count) {
  int res = 0;
  for (int i = 0; i < count; ++i)
    res += a[i];

  return res;
}
```

The IR representation shows each of the new instructions being used to control the loop, along with one of the horizontal reduction intrinsics. The main sequence (noted by the *;; Control* comments below) for controlling loop iterations is:

1. Using *elementcount* to increment the induction variable
2. Using *seriesvector* starting from the current induction variable value to compare against a splat of the loop's trip count
3. Using *propff* to ensure the resulting predicate strictly partitions the predicate and does not wrap
4. Using *test* to determine whether the first lane is active (and thus at least one more iteration is required)

```
define i32 @SimpleReduction(i32* nocapture readonly %a, i32 %count) #0 {
entry:
  %cmp6 = icmp sgt i32 %count, 0
  br i1 %cmp6, label %min.iters.checked, label %for.cond.cleanup

min.iters.checked:
  %0 = add i32 %count, -1
  %1 = zext i32 %0 to i64
  %2 = add nuw nsw i64 %1, 1
  %wide.end.idx.splatinsert = insertelement <n x 4 x i64> undef, i64 %2, i32 0
  %wide.end.idx.splat = shufflevector <n x 4 x i64> %wide.end.idx.splatinsert,
                            <n x 4 x i64> undef, <n x 4 x i32> zeroinitializer
  %3 = icmp ugt <n x 4 x i64> %wide.end.idx.splat, seriesvector (i64 0, i64 1)
  %predicate.entry = propff <n x 4 x i1> shufflevector (<n x 4 x i1> insertelement
                            (<n x 4 x i1> undef, i1 true, i32 0), <n x 4 x i1> undef,
                            <n x 4 x i32> zeroinitializer), %3
  br label %vector.body

vector.body:
                                                      %min.iters.checked
  %index = phi i64 [ 0, %min.iters.checked ], [ %index.next, %vector.body ]
  %predicate = phi <n x 4 x i1> [ %predicate.entry, %min.iters.checked ],
                                      [ %predicate.next, %vector.body ]
  %vec.phi = phi <n x 4 x i32> [ zeroinitializer, %min.iters.checked ],
                                                    [ %8, %vector.body ]
  %4 = icmp ult i64 %index, 4294967296
  call void @llvm.assume(i1 %4)
  %5 = getelementptr inbounds i32, i32* %a, i64 %index
  %6 = bitcast i32* %5 to <n x 4 x i32>*
  %wide.masked.load = call <n x 4 x i32> @llvm.masked.load.nxv4i32(<n x 4 x i32>* %6,
                      i32 4, <n x 4 x i1> %predicate, <n x 4 x i32> undef), !tbaa !1
  %7 = select <n x 4 x i1> %predicate, <n x 4 x i32> %wide.masked.load,
```

```
                                          <n x 4 x i32> zeroinitializer
  %8 = add nsw <n x 4 x i32> %vec.phi, %7
  %index.next = add nuw nsw i64 %index, elementcount (<n x 4 x i64> undef) ;; Control 1
  %9 = add nuw nsw i64 %index, elementcount (<n x 4 x i64> undef)
  %10 = seriesvector i64 %9, i64 1 as <n x 4 x i64>                          ;; Control 2
  %11 = icmp ult <n x 4 x i64> %10, %wide.end.idx.splat
  %predicate.next = propff <n x 4 x i1> %predicate, %11                      ;; Control 3
  %12 = test first true <n x 4 x i1> %predicate.next                         ;; Control 4
  br i1 %12, label %vector.body, label %middle.block, !llvm.loop !5

middle.block:
  %.lcssa = phi <n x 4 x i32> [ %8, %vector.body ]
  %13 = call i64 @llvm.aarch64.sve.uaddv.nxv4i32(<n x 4 x i1> shufflevector
       (<n x 4 x i1> insertelement (<n x 4 x i1> undef, i1 true, i32 0),
       <n x 4 x i1> undef, <n x 4 x i32> zeroinitializer), <n x 4 x i32> %.lcssa)
  %14 = trunc i64 %13 to i32
  br label %for.cond.cleanup

for.cond.cleanup:
  %res.0.lcssa = phi i32 [ 0, %entry ], [ %14, %middle.block ]
  ret i32 %res.0.lcssa
}
```

The main vector body of the resulting code is one instruction longer than it would be for NEON, but no scalar tail is required and performance will scale with register length. The *seriesvector*, *shufflevector*(splat), *icmp*, *propff*, *test* sequence has been recognized and transformed into the `whilelo` instruction.

```
SimpleReduction:
// BB#0:
          subs    w9, w1, #1
          b.lt    .LBB0_4
// BB#1:
          add     x9, x9, #1
          mov     x8, xzr
          whilelo p0.s, xzr, x9
          mov     z0.s, #0
.LBB0_2:
          ld1w    {z1.s}, p0/z, [x0, x8, lsl #2]
          incw    x8
          add     z0.s, p0/m, z0.s, z1.s
          whilelo p0.s, x8, x9
          b.mi    .LBB0_2
// BB#3:
          ptrue   p0.s
          uaddv   d0, p0, z0.s
          fmov    w0, s0
          ret
.LBB0_4:
          mov     w0, wzr
          ret
```

Appendix

The following is an alternative to the *propff* instruction described above. We haven't implemented this (or worked out all the specific details) since we can synthesize the initial partitions required with extra operations, although matching those and transforming to appropriate AArch64 SVE instructions is fragile given other optimization passes. This instruction is more complex, but would allow us to explicitly specify the exact partition desired.

partition

Syntax:

```
<result> = partition first <part_on> [inclusive] <ty> <p>  [, propagating
<pprop>]

<result> = partition next <part_on> [inclusive] <ty> <p> from <pcont>
```

Overview:

This instruction creates a partitioned boolean vector based on an input vector `p`. There are two main modes to consider. The *first* mode is an extended version of *propff* where `part_on` is a boolean which determines whether the partition is made based on the first false value or the first true value. Propagating from a previous vector is optional. The `inclusive` option produces a partition which includes the first true or false value; the default is an exclusive partition which only covers the elements before the first true or false value.

In cases where there are multiple subsets of data in a vector which must be processed independently, we need a way to take the original boolean vector and an existing partition to create the next partition to work

on. This is provided by the *next* form of *partition*. The `pcont` argument is the existing partitioned vector, generated either by a *partition first* or another *partition next*. If there are no more partitions after the current one, a boolean vector of all false will be returned.

Fixed-Width Behaviour:

Same as scalable.