Overview of the Arm Scalable Vector Extension

The 3rd Isambard hackathon - "Full Steam Ahead!"

Fabrice Dupros Fabrice.Dupros@arm.com

Conrad Hillairet Conrad.Hillairet@arm.com

Phil Ridley Phil.Ridley@arm.com

23rd March 2021

Agenda

- Neon instructions
  - SIMD on Arm
  - Programming with Neon
- SVE
  - Introduction to SVE and Registers
- VLA Programming
  - How to Program SVE
- Simple Instructions
  - Load / Store Operations
  - Intrinsics with ACLE

arm

General Purpose Vector Instructions

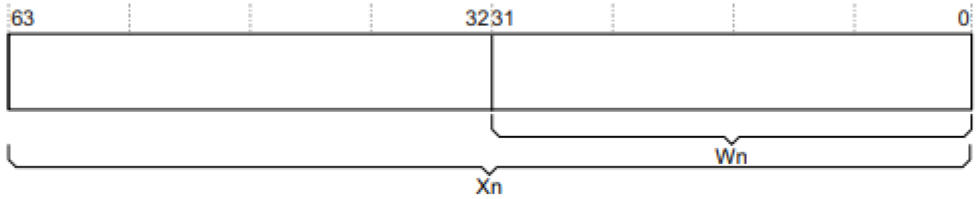
Arm Neon Vector Units

- SIMD Vector Extensions
  - Advanced Single Instruction Multiple Data
  - Fixed width at 128-bit

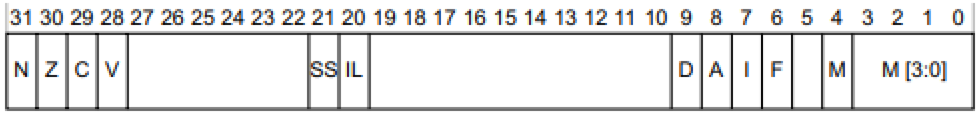
- As of Armv8-a
  - 31x 64-bit general-purpose registers
    - The 32-bit W register is lower half of 64-bit X register
  - 32x 128-bit floating-point registers
    - D is lower 64 bits of 128-bit Q registers

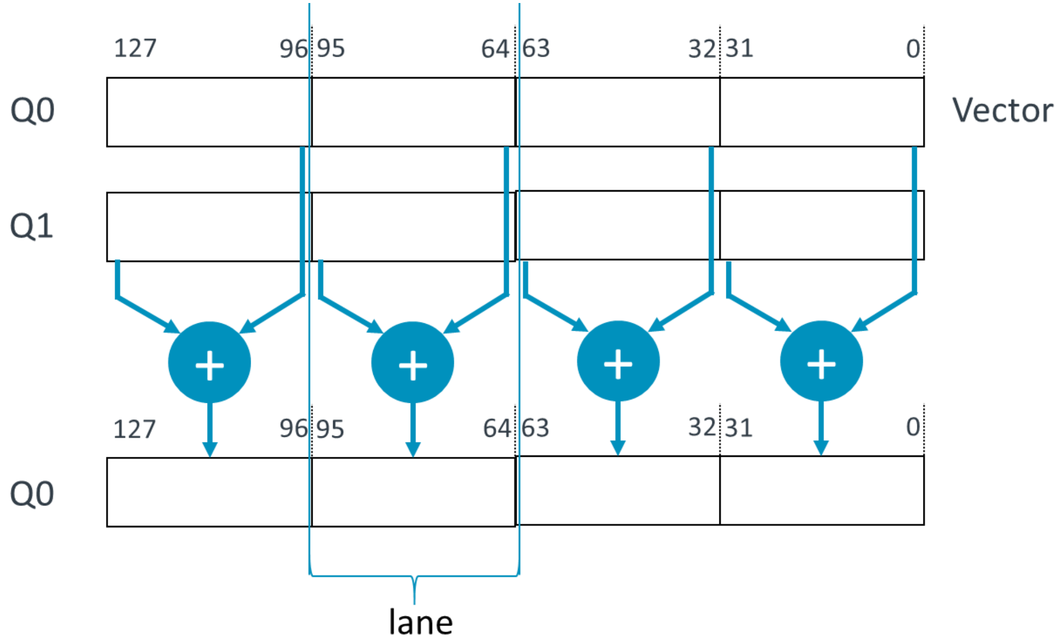
- Example:
  - fadd v0.4s, v0.4s, v1.4s
  - Addition of 4 x 32-bit floats
    - 128-bit Neon/32-bit Int = 4 Lanes

64-bit register layout



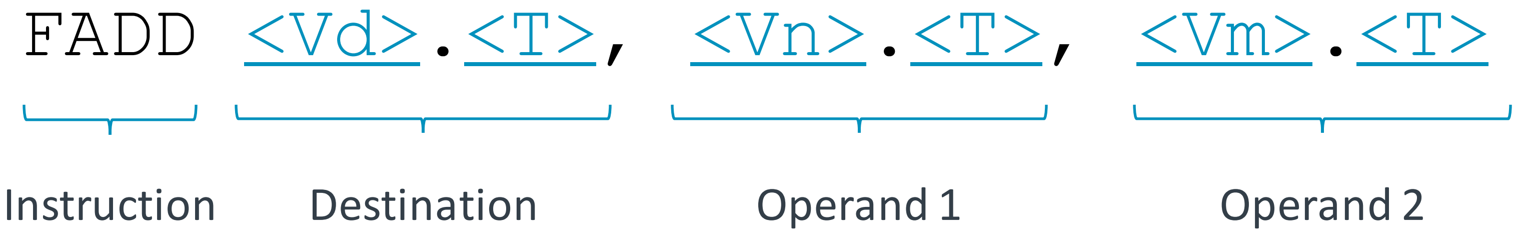
SPSR: Saved Program State Register





Arm Neon Vector Instructions

- Comprehensive set of vector operations
  - Loads, stores and maths operations
  - Scalar and floating point



```
FADD <Vd>.<T>, <Vn>.<T>, <Vm>.<T>
```

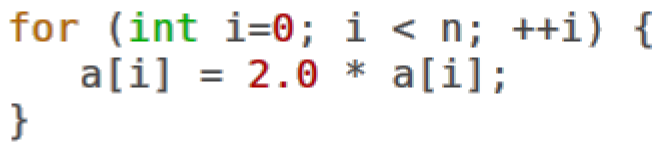
Instruction Destination Operand 1 Operand 2

- <Vd> - Destination register, <T> - Type e.g. 4S or 2D

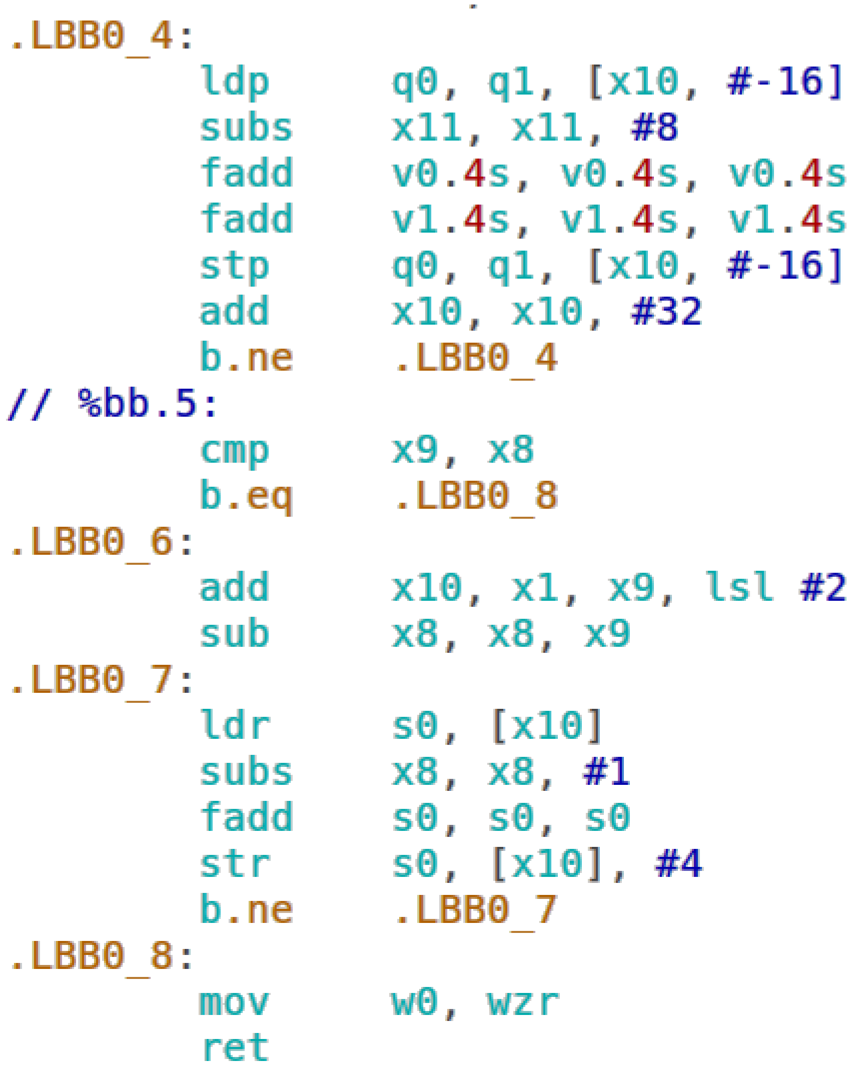
```
FADD V0.4S, V0.4S, V1.4S
```

- Add 4 single precision floating point values from V0 to V1 and store in V0
  - V0 += V1

Programming Neon



```
for (int i=0; i < n; ++i) {
   a[i] = 2.0 * a[i];
}
```



```
.LBB0_4:
        ldp     q0, q1, [x10, #-16]
        subs    x11, x11, #8
        fadd    v0.4s, v0.4s, v0.4s
        fadd    v1.4s, v1.4s, v1.4s
        stp     q0, q1, [x10, #-16]
        add     x10, x10, #32
        b.ne    .LBB0_4
// %bb.5:
        cmp     x9, x8
        b.eq    .LBB0_8
.LBB0_6:
        add     x10, x1, x9, lsl #2
        sub     x8, x8, x9
.LBB0_7:
        ldr     s0, [x10]
        subs    x8, x8, #1
        fadd    s0, s0, s0
        str     s0, [x10], #4
        b.ne    .LBB0_7
.LBB0_8:
        mov     w0, wzr
        ret
```

- .LBB0_4: Start with vector loop (and unroll)
  - Load 8 x 32-bit values (into 2 x 128-bit registers)
  - Subtract 8 from loop counter
  - 2x Neon add instructions (register to itself => 2.0*a[i])
  - Store pair of 128-bit registers back
  - Update array offsets
  - Loop if >=8 iterations left
- .LBB0_7: Remainder (fewer than 8 iterations left)
  - Load a single scalar
  - Add it to itself
  - Store
  - Loop if iterations left

Ease of Use for Neon

Where to start

Neon Intrinsics (ACLE)

- `#include arm_neon.h`
  - Header file of neon intrinsics

- Map to assembly types and instruction names

```
float32x4_t va = vld1q_f32(&a[i]);
va = vmulq_n_f32(va, 2.0);
vst1q_f32(&a[i], va)
```

  - Load 4x 32-bit floats into `*va*` from a[i]
  - Multiply the floats in `*va*` by 2.0
  - Store contents of `*va*` back into a[i]

Auto Vectorisation & Libraries

- Not everyone wants to hand code assembly

- Compilers will generate vector code
  - Generally at optimization levels > -O2
  - Supported in GCC, LLVM, Cray, Arm compiler
  - Vectorisation reports will inform on success

- Designed for ease of use
  - No big gains for Double precision
  - 2x at ***best*** - but very unlikely

- Vectorised libraries
  - Such as ArmPL maths library

Limitations of Neon

- Neon is firstly only 128-bit
  - Not much use to HPC / Scientific Computing

- Want bigger vectors
  - To expand to 256-bit of 512-bit we would need separate instructions
  - Arm like to offer flexibility to customers - Different vector lengths
    - However it is a RISC architecture (32-bit instruction encoding)

- Suffers from same drawbacks as other vector implementations (AVX)
  - Difficulty to auto-vectorise
  - Remainder loops

Scalable Vector Extension: Today’s new Vectorisation Paradigm

Scalable Vector Extension

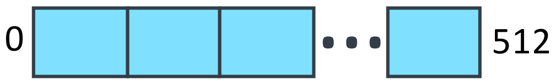
- **SVE is Vector Length Agnostic (VLA)**
  - Vector Length (VL) is a hardware implementation choice from 128 up to 2048 bits.
  - New programming model allows software to scale dynamically to available vector length.
  - No need to define a new ISA, rewrite or recompile for new vector lengths.
- **SVE is not an extension of Advanced SIMD (*aka* Neon)**
  - A separate, optional extension with a new set of instruction encodings.
  - Initial focus is HPC and general-purpose server, not media/image processing.
- **SVE begins to tackle traditional barriers to auto-vectorization**
  - Software-managed speculative vectorization allows uncounted loops to be vectorized.
  - In-vector serialised inner loop permits outer loop vectorization in the presence of dependencies.

Arm’s Scalable Vector Extension (SVE)

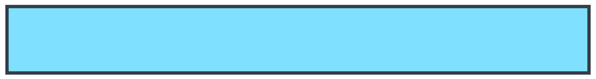
What does the SVE ISA look like?

How SVE works

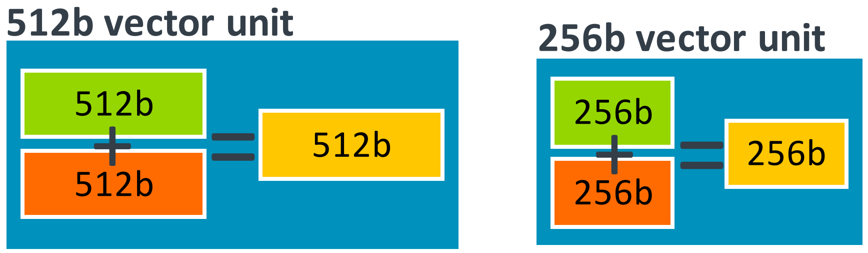
The hardware sets the vector length



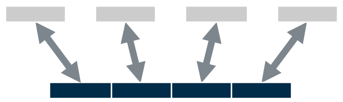
In software, vectors have no length



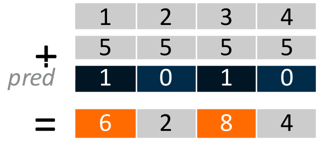
The *exact same* binary code runs on hardware with different vector lengths



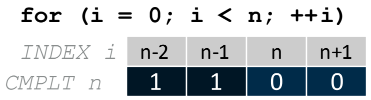
SVE improves auto-vectorization



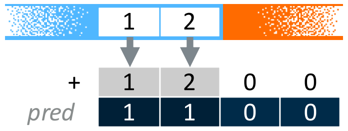
Gather-load and scatter-store



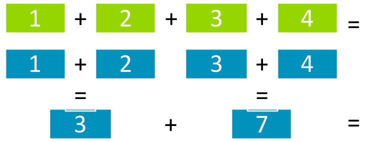
Per-lane predication



Predicate-driven loop control and management



Vector partitioning and software-managed speculation



Extended floating-point horizontal reductions

SVE vs Traditional ISA

How do we compute data which has ten chunks of 4-bytes?

Aarch64 (scalar)

- □ Ten iterations over a 4-byte register

Neon (128-bit vector engine)

- □ Two iterations over a 16-byte register + two iterations of a drain loop over a 4-byte register

SVE (128-bit VLA vector engine)

- □ Three iterations over a 16-byte **VLA register** with an adjustable **predicate**

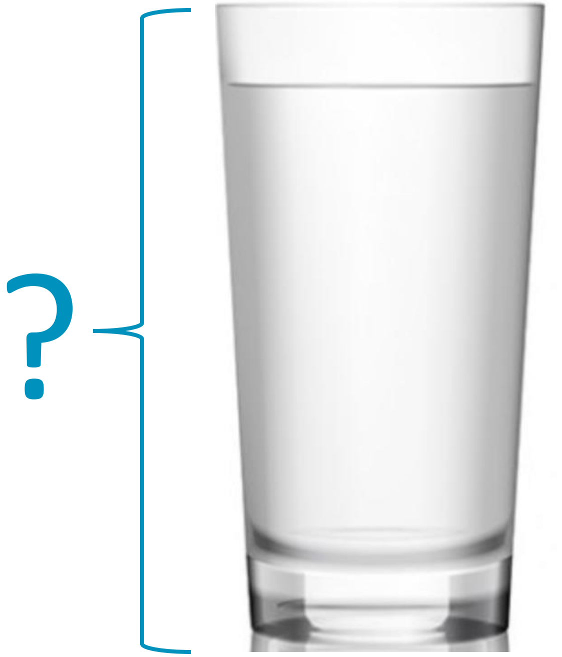
How big can an SVE vector be?

Any multiple of 128 bits up to 2048 bits, and it can be dynamically reduced.

```
(A) VL = LEN x 128
  (B) VL <= 2048
```

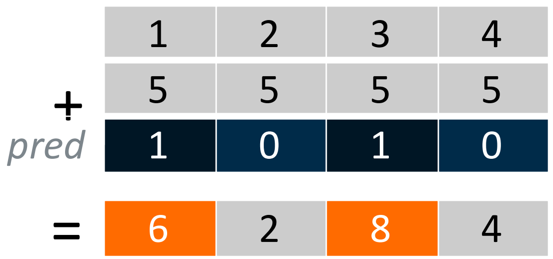
`VL` is *implementation dependent*,

can be *reduced by the OS/Hypervisor*.



How can you program when the vector length is unknown?

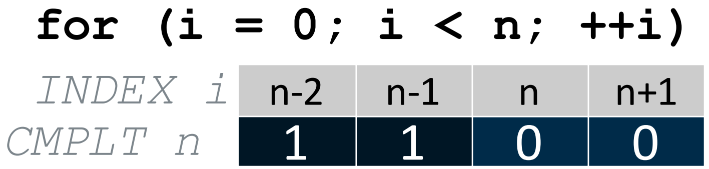
SVE provides features to enable VLA programming from the assembly level and up



	1	2	3	4
+	5	5	5	5
pred	1	0	1	0
=	6	2	8	4

Per-lane predication

Operations work on individual lanes under control of a predicate register.

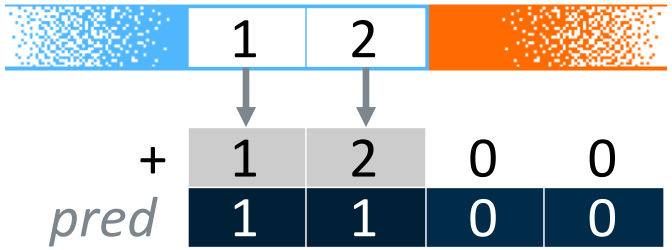


```
for (i = 0; i < n; ++i)
```

INDEX i	n-2	n-1	n	n+1
CMPLT n	1	1	0	0

Predicate-driven loop control and management

Eliminate scalar loop heads and tails by processing partial vectors.



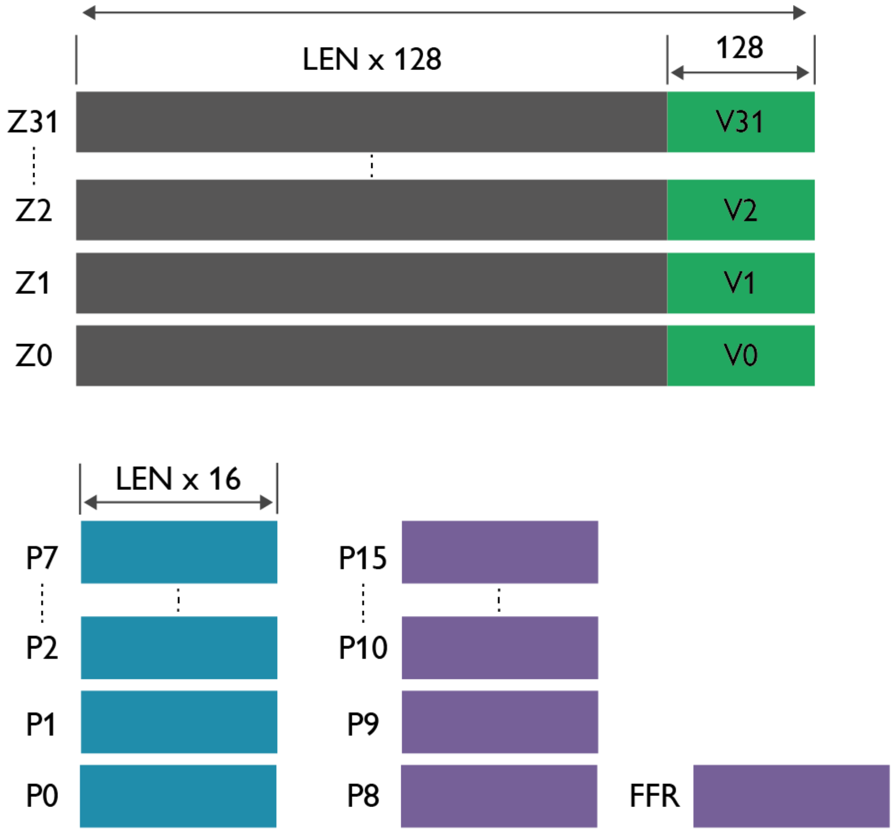
	1	2		
+	1	2	0	0
pred	1	1	0	0

Vector partitioning & software-managed speculation

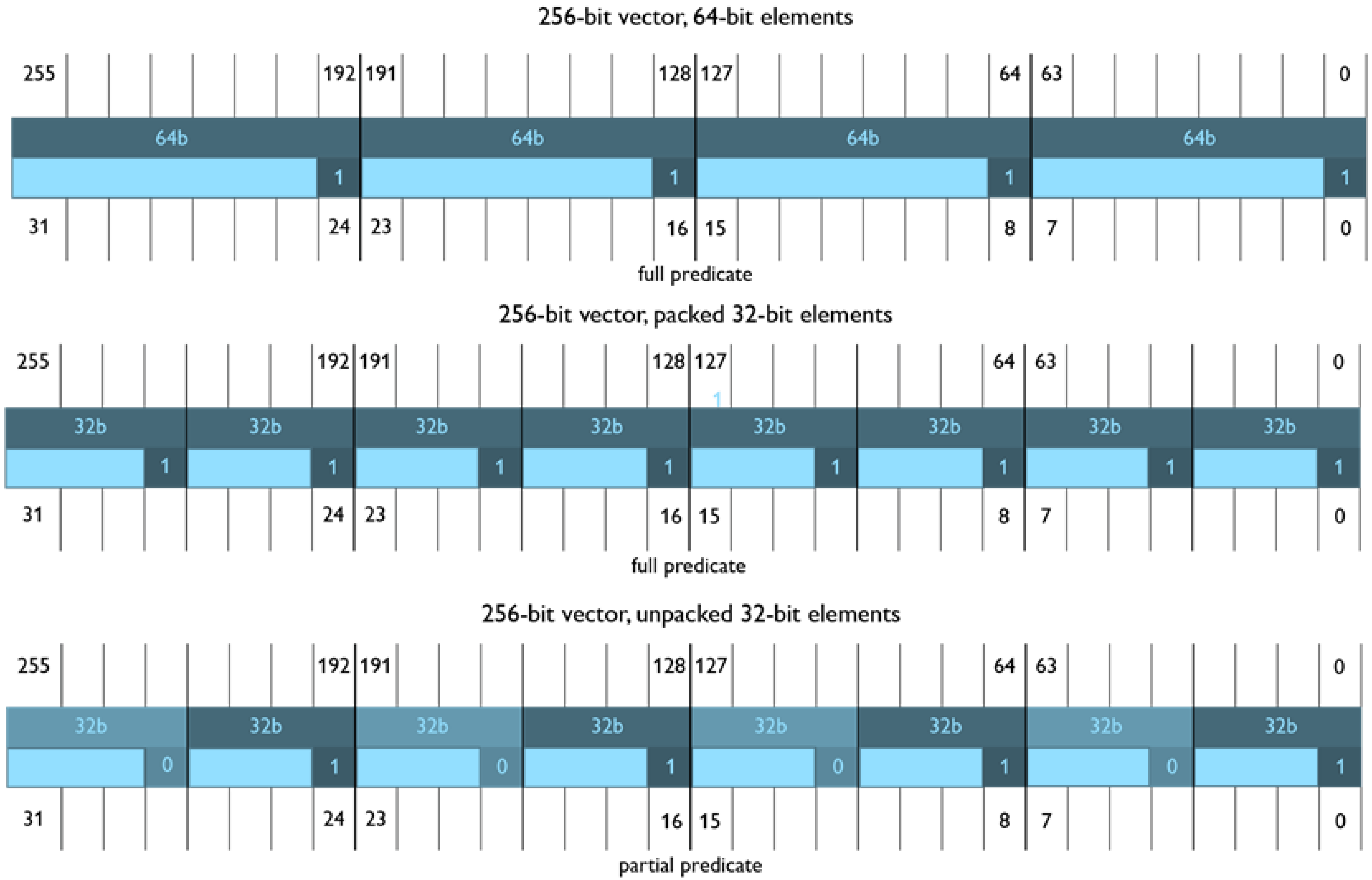
First Faulting Load instructions allow memory accesses to cross into invalid pages.

SVE Registers

- **Scalable vector registers**
  - `Z0-Z31` extending NEON's 128-bit `V0-V31`.
  - Packed DP, SP & HP floating-point elements.
  - Packed 64, 32, 16 & 8-bit integer elements.
- **Scalable predicate registers**
  - `P0-P7` governing predicates for load/store/arithmetic.
  - `P8-P15` additional predicates for loop management.
  - `FFR` first fault register for software speculation.



SVE vector & predicate register organization



256-bit vector, 64-bit elements

full predicate

256-bit vector, packed 32-bit elements

full predicate

256-bit vector, unpacked 32-bit elements

partial predicate

SVE Predicate condition flags

SVE is a *predicate-centric* architecture

- Predicates are central, not an afterthought
- Support complex nested conditions and loops.
- Predicate generation also sets condition flags.
- Reduces vector loop management overhead.

Overloading the A64 NZCV condition flags

Flag	SVE	Condition
N	First	Set if first active element is true
Z	None	Set if no active element is true
C	!Last	Set if last active element is false
V		Scalarized loop state, else zero

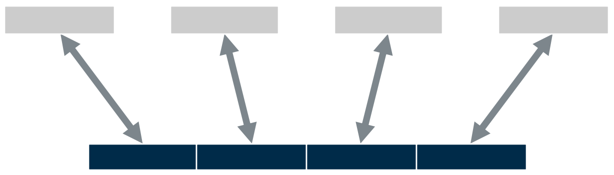
Reuses the A64 conditional instructions

- Conditional branches B.EQ → B.NONE
- Conditional select, set, increment, etc.

Condition Test	A64 Name	SVE Alias	SVE Interpretation
Z=1	EQ	NONE	No active elements are true
Z=0	NE	ANY	Any active element is true
C=1	CS	NLAST	Last active element is not true
C=0	CC	LAST	Last active element is true
N=1	MI	FIRST	First active element is true
N=0	PL	NFRST	First active element is not true
C=1 & Z=0	HI	PMORE	More partitions: some active elements are true but not the last one
C=0 | Z=1	LS	PLAST	Last partition: last active element is true or none are true
N=V	GE	TCONT	Continue scalar loop
N!=V	LT	TSTOP	Stop scalar loop

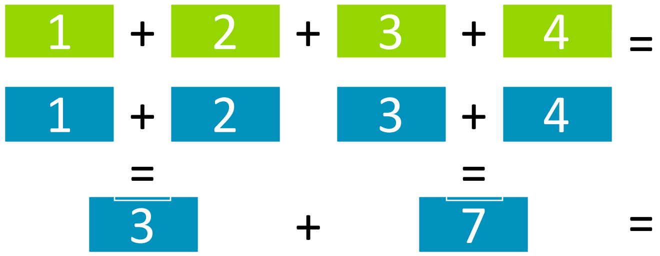
SVE supports vectorization in complex code

Right from the start, SVE was engineered to handle codes that usually won't vectorize



Gather-load and scatter-store

Loads a single register from several non-contiguous memory locations.



Extended floating-point horizontal reductions

In-order and tree-based reductions trade-off performance and repeatability.

Vector Length Agnostic Programming

VLA

Vector **L**ength **A**gnostic
programming model

- Write once
- Compile once
- Vectorize more loops

Open source support

- **Arm actively posting SVE open source patches upstream**
  - Beginning with first public announcement of SVE at HotChips 2016

- **Available upstream**

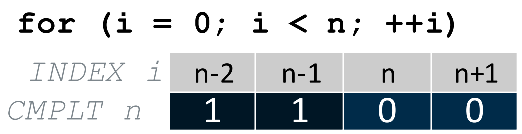
GNU Binutils-2.28:	released Feb 2017, includes SVE assembler & disassembler
GCC 8:	Full assembly, disassembly and basic auto-vectorization
LLVM 7:	Full assembly, disassembly
QEMU 3:	User space SVE emulation
GDB 8.2	HPC use cases fully included

- **Under upstream review**

LLVM:	Since Nov 2016, as presented at LLVM conference
Linux kernel:	Since Mar 2017, LWN article on SVE support

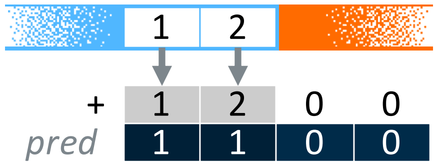
Quick Recap

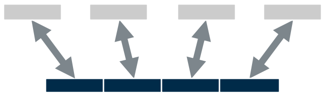
- SVE enables Vector Length Agnostic (VLA) programming
- VLA enables portability, scalability, and optimization
- Predicates control which operations affect which vector lanes
  - Predicates are not bitmasks
  - You can think of them as dynamically resizing the vector registers
- The actual vector length is set by the CPU architect
  - Any multiple of 128 bits up to 2048 bits
  - May be dynamically reduced by the OS or hypervisor
- SVE was designed for HPC and can vectorize complex structures
- Many open source and commercial tools currently support SVE

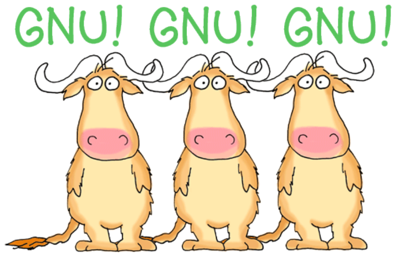


```
for (i = 0; i < n; ++i)
```

INDEX i	n-2	n-1	n	n+1
CMPLT n	1	1	0	0







Vector Length Agnostic Programming

A paradigm shift for developers

Advantages

- Not thinking about vector length
  - Rather just vectorisation
- No peel/remainder loops
  - All handled by predication
- Key is loop structures
  - Predicates are powerful

Considerations

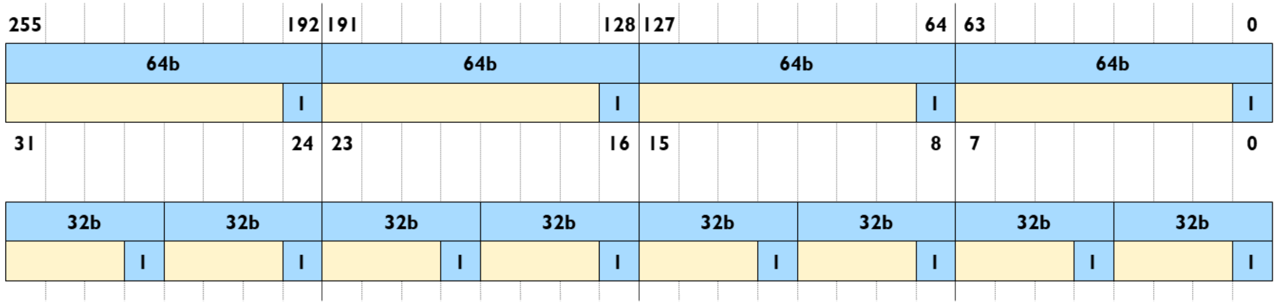
- Should not be writing fixed width
  - Applies to data structures and instructions
  - More portable for different hardware
- Can the compiler identify loop structure?
  - Generate predicated instructions
- However: VLA *may* be slower
  - Cost of generating predicates
  - Near empty loops

Introduction to SVE Instructions

Simple Instructions: Generating Predicates

- PTRUE <Pd>.<T>
  - Predicate of all 1’s
  - Suffix of type single or double

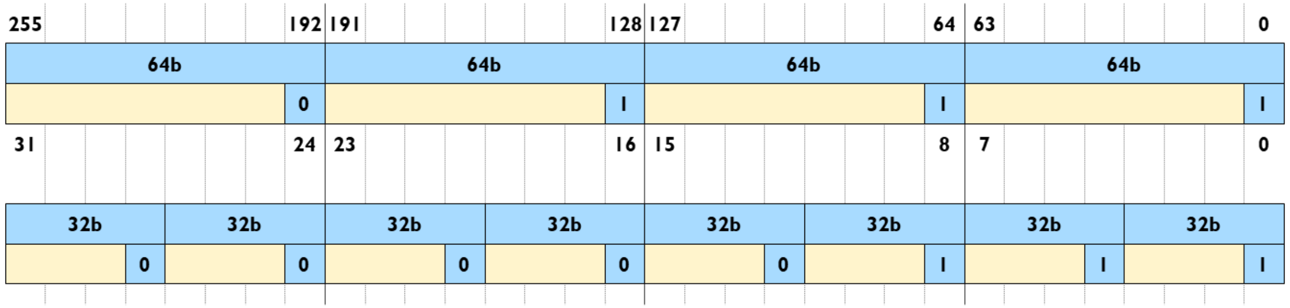
```
PTRUE P0.D

PTRUE P0.S
```



- WHILELT <Pd>.<T>, <R><n>, <R><m>
  - Counter less than
  - Used for loops
  - for(int i=8; i<11;++i)

X8 = #8, X2 = #11

```
WHILELT P0.D, X8, X2

WHILELT P0.S, X8, X2
```



Simple Instructions: Loads

- LD1D { <Zt>.D }, <Pg>/Z, [<Xn|SP>, <Xm>, LSL #3]
  - Load double word (64-bit) into <Zt> register
  - Using predicate <Pg>
  - Load ‘zeros’ for all non active lanes of <Pg>
  - From address of <Xn|SP> register
    - With offset of counter <Xm>
    - With a logical shift left of 3 (multiplied by 8)

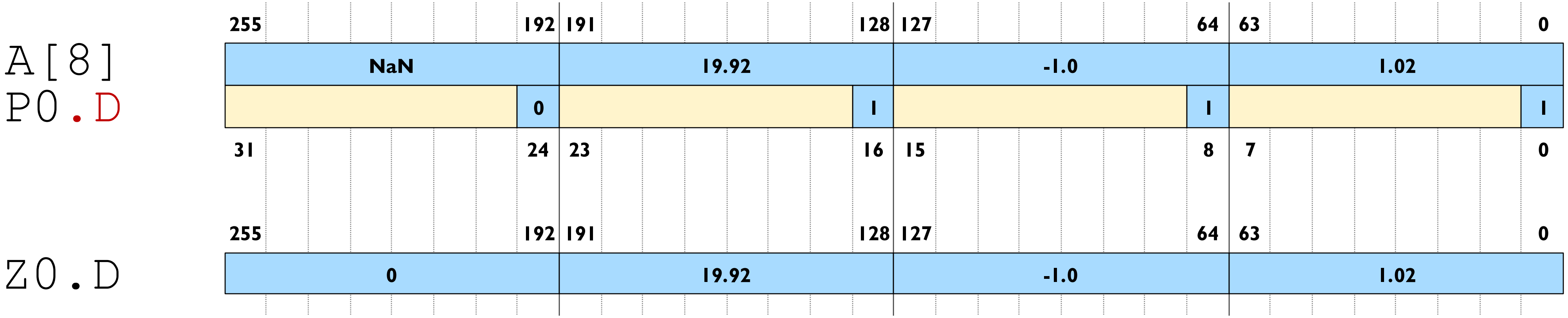
X0 = double *A

X8 = #8

X2 = #11

WHILELT P0.D, X8, X2

LD1D { Z0.D }, P0/z, [X0, X8, LSL #3]



	255 … 192	191 … 128	127 … 64	63 … 0
A[8]	NaN	19.92	-1.0	1.02

	31 … 24	23 … 16	15 … 8	7 … 0
P0.D	0	1	1	1

	255 … 192	191 … 128	127 … 64	63 … 0
Z0.D	0	19.92	-1.0	1.02

How do you count by vector width?

No need for multi-versioning: one increment to rule all vector sizes

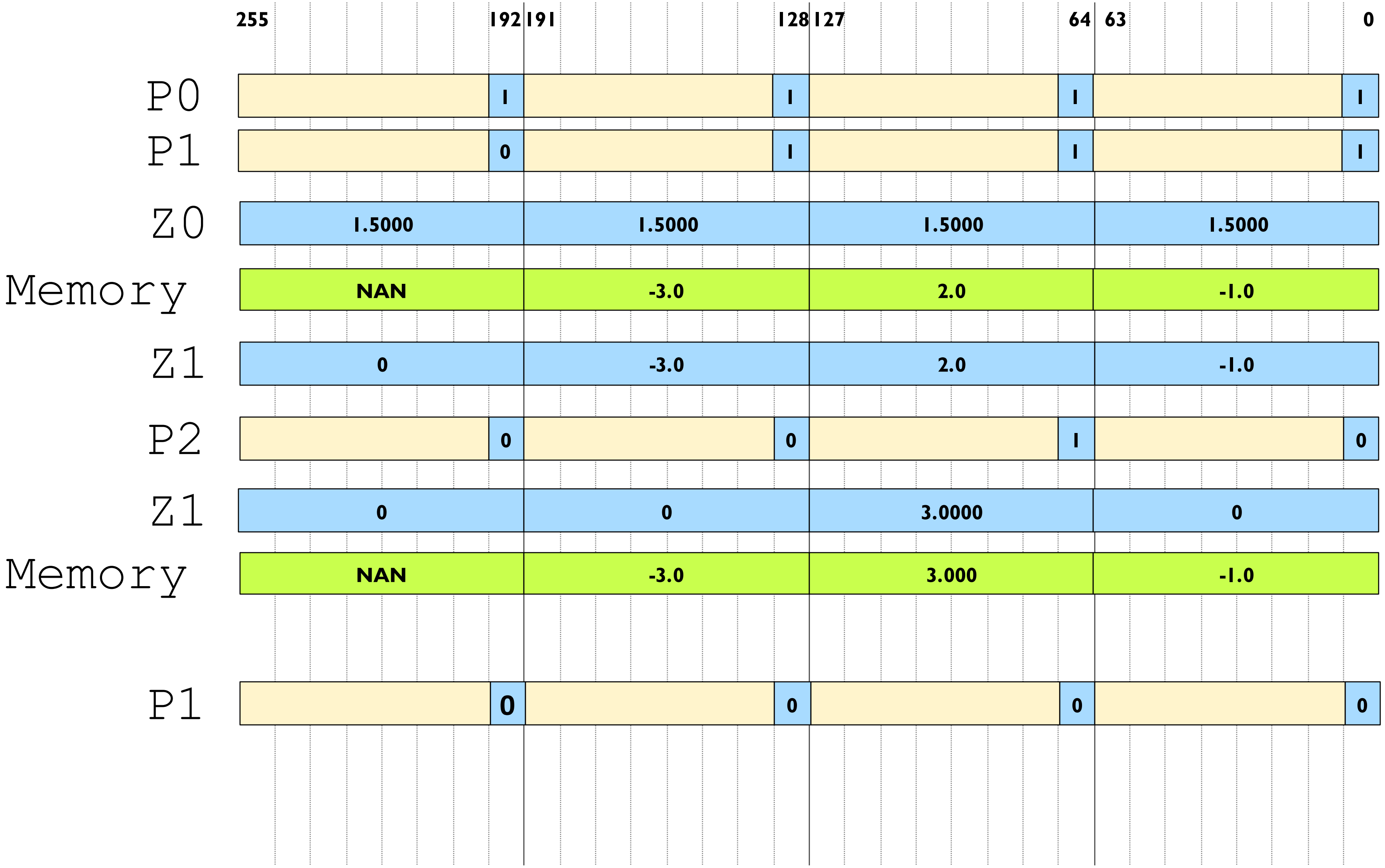
```
ld1w z1.s, p0/z, [x0,x4,lsl 2]      // p0:z1=x[i]
ld1w z2.s, p0/z, [x1,x4,lsl 2]      // p0:z2=y[i]
fmla z2.s, p0/m, z1.s, z0.s         // p0?z2+=x[i]*a
st1w z2.s, p0, [x1,x4,lsl 2]        // p0?y[i]=z2

incw x4                             // i+=(VL/32)
// or
incd x4                             // i+=(VL/64)
```

“Increment `x4` by the number of 32-bit lanes (`w`) that fit in a VL.”
“Increment `x4` by the number of 64-bit lanes (`d`) that fit in a VL.”

Simple Instructions: Conditional Multiply

```
A = {-1.0, 2.0, -3.0}
for(int i=0; i< N; ++i)
        if(a[i] > 0) a[i] *=1.5;
```

```
    mov     x8, xzr
    ptrue   p0.d
    whilelo p1.d, xzr, x9
    fmov    z0.d, #1.50000000
.LBB0_2:
    ld1d    { z1.d }, p1/z, [x1, x8, lsl #3]
    fcmgt   p2.d, p1/z, z1.d, #0.0
    fmul    z1.d, p2/z, z1.d, z0.d
    st1d    { z1.d }, p2, [x1, x8, lsl #3]
    incd    x8
    whilelo p1.d, x8, x9
    b.mi    .LBB0_2
```



	255 – 192	191 – 128	127 – 64	63 – 0
P0	1	1	1	1
P1	0	1	1	1
Z0	1.5000	1.5000	1.5000	1.5000
Memory	NAN	-3.0	2.0	-1.0
Z1	0	-3.0	2.0	-1.0
P2	0	0	1	0
Z1	0	0	3.0000	0
Memory	NAN	-3.0	3.000	-1.0
P1	0	0	0	0

Arm C Language Extensions for SVE

Compiler Intrinsics for SVE

SVE ACLE

SVE Arm C Language Extensions – aka *C intrinsics*

```
#include <arm_sve.h>
```

- VLA Data types:
  - svfloat64_t, svfloat16_t, svuint32_t, …
  - ***sv<datatype><datasize>_t***

- Predication:
  - Merging: _m
  - Zeroing: _z
  - Don’t care: _x
  - Predicate type: svbool_t

- Intrinsics are **not 1-1 with the ISA.**

- But *Nearly* one intrinsic per SVE instruction

- VLA functions:
  - svfloat64_t svld1_f64(svbool_t pg, const float64_t *base)
  - svfloat32_t svadd[_n_f32]_z(svbool_t pg,
    ```
                           svfloat32_t op1,
                           float32_t op2);
    ```
  - **svbase[*disambiguator*][*type0*][*type1*]…[*pred*]**
  - *base* is the lower-case name of an SVE instruction
  - *disambiguator* distinguishes between different forms of a function
  - *typeN* lists the types of vectors and predicates
  - *pred* describes the inactive elements in the result of a predicated operation

Arm C Language Extensions for SVE

Vectorizing a scalar loop with ACLE

a[:] = 2.0 * a[:]

Original Code

```
for (int i=0; i < N; ++i) {
  a[i] = 2.0 * a[i];
}
```

128-bit Neon ACLE

```
int i;

// vector loop
for (i=0; (i<N-3) && (N&~3); i+=4) {
  float32x4_t va = vld1q_f32(&a[i]);
  va = vmulq_n_f32(va, 2.0);
  vst1q_f32(&a[i], va)
}
// drain loop
for (; i < N; ++i)
  a[i] = 2.0 * a[i];
```

SVE ACLE

```
for (int i = 0 ; i<N ; i += svcntw()){
  svbool_t Pg = svwhilelt_b32(i, N);
  svfloat32_t va = svld1(Pg, &a[i]);
  va = svmul_x(Pg, va, 2.0);
  svst1(Pg, &a[i], va);
}
```

Arm C Language Extensions for SVE

Vectorizing A Scalar Loop With ACLE

a[:] = 2.0 * a[:]

Original Code

```
for (int i=0; i < N; ++i) {
 a[i] = 2.0 * a[i];
}
```

128-bit Neon vectorization

```
int i;

// vector loop
for (i=0; (i<N-3) && (N&~3); i+=4) {
  float32x4_t va = vld1q_f32(&a[i]);
  va = vmulq_n_f32(va, 2.0);
  vst1q_f32(&a[i], va)
}
// drain loop
for (; i < N; ++i)
  a[i] = 2.0 * a[i];
```

This is Neon, ***not*** SVE!

```
for (int i=0; i < N; ++i) {
  a[i] = 2.0 * a[i];
}
```

Vectorizing A Scalar Loop With ACLE

a[:] = 2.0 * a[:]

128-bit Neon vectorization

```
int i;

// vector loop
for (i=0; (i<N-3) && (N&~3); i+=4) {
   float32x4_t va = vld1q_f32(&a[i]);
   va = vmulq_n_f32(va, 2.0);
   vst1q_f32(&a[i], va)
}
// drain loop
for (; i < N; ++i)
   a[i] = 2.0 * a[i];
```

SVE vectorization

```
for (int i = 0 ; i < N; i += svcntw() )

{

 svbool_t Pg = svwhilelt_b32(i, N);

 svfloat32_t va = svld1(Pg, &a[i]);

 va = svmul_x(Pg, va, 2.0);

 svst1(Pg, &a[i], va);

}
```

Vectorizing A Scalar Loop With ACLE

a[:] = 2.0 * a[:]

```
for (int i=0; i < N; ++i) {
  a[i] = 2.0 * a[i];
}
```

SVE vectorization

```
for (int i = 0 ; i < N; i += svcntw() )
{
  svbool_t Pg = svwhilelt_b32(i, N);
  svfloat32_t va = svld1(Pg, &a[i]);
  va = svmul_x(Pg, va, 2.0);
  svst1(Pg, &a[i], va);
}
```

SVE vectorization with fewer branches

```
svbool_t all = svptrue_b32();
svbool_t Pg;
for (int i=0;
     svptest_first(all, Pg=svwhilelt_b32(i, N));
     i += svcntw())
{
  svfloat32_t va = svld1(Pg, &a[i]);
  va = svmul_x(Pg, va, 2.0);
  svst1(Pg, &a[i], va);
}
```

```
for (int i=0; i < N; ++i) {
  a[i] = 2.0 * a[i];
}
```

Vectorizing A Scalar Loop With ACLE

a[:] = 2.0 * a[:]

SVE vectorization with fewer branches

```
svbool_t all = svptrue_b32();
svbool_t Pg;
for (int i=0;
    svptest_first(all, Pg=svwhilelt_b32(i, N));
    i += svcntw())
{
  svfloat32_t va = svld1(Pg, &a[i]);
  va = svmul_x(Pg, va, 2.0);
  svst1(Pg, &a[i], va);
}
```

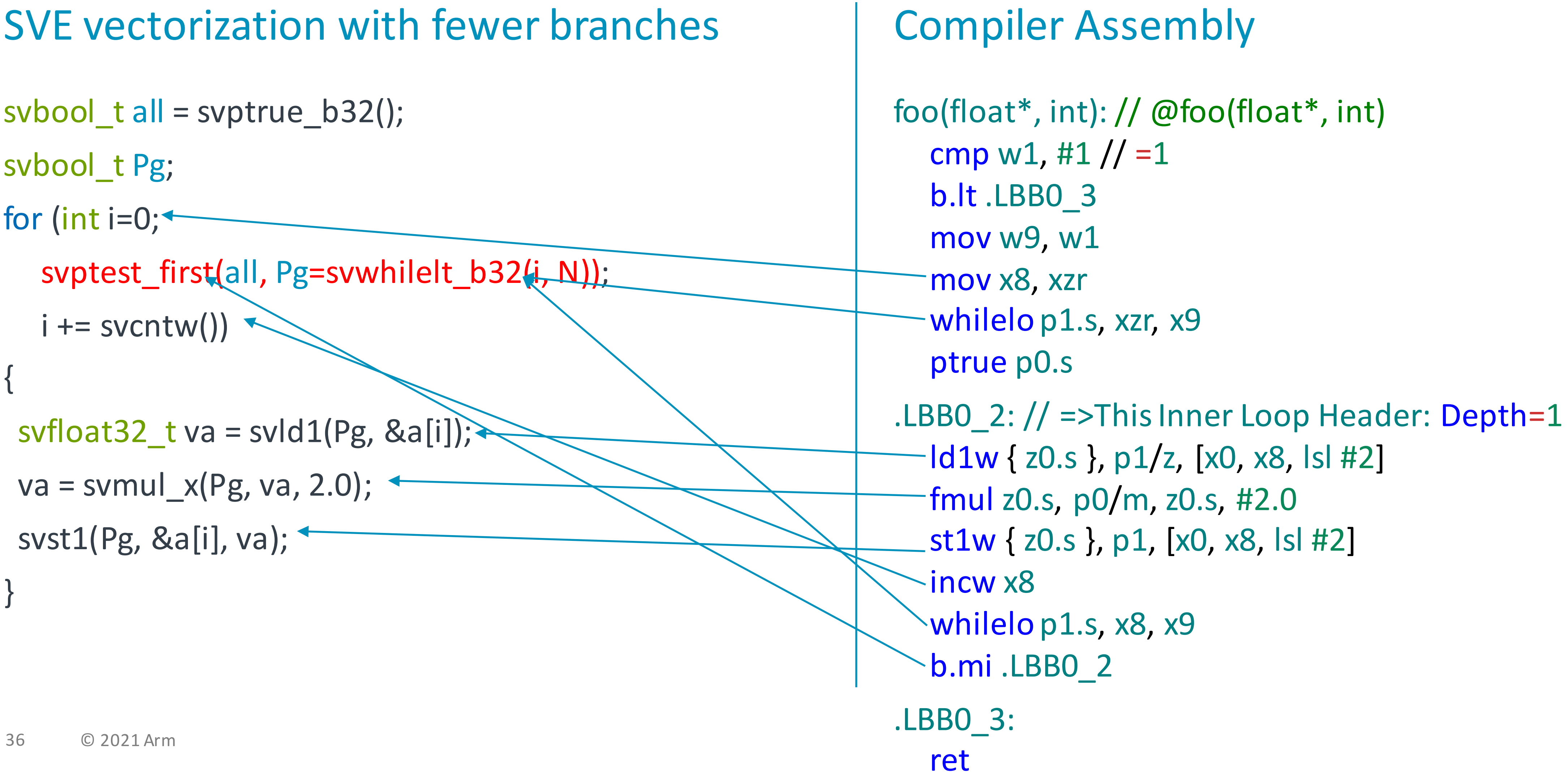
Compiler Assembly

```
foo(float*, int): // @foo(float*, int)
    cmp w1, #1 // =1
    b.lt .LBB0_3
    mov w9, w1
    mov x8, xzr
    whilelo p1.s, xzr, x9
    ptrue p0.s
.LBB0_2: // =>This Inner Loop Header: Depth=1
    ld1w { z0.s }, p1/z, [x0, x8, lsl #2]
    fmul z0.s, p0/m, z0.s, #2.0
    st1w { z0.s }, p1, [x0, x8, lsl #2]
    incw x8
    whilelo p1.s, x8, x9
    b.mi .LBB0_2
.LBB0_3:
    ret
```

Vectorizing A Scalar Loop With ACLE

a[:] = 2.0 * a[:]

```
for (int i=0; i < N; ++i) {
  a[i] = 2.0 * a[i];
}
```



SVE vectorization with fewer branches

```
svbool_t all = svptrue_b32();
svbool_t Pg;
for (int i=0;
    svptest_first(all, Pg=svwhilelt_b32(i, N));
    i += svcntw())
{
  svfloat32_t va = svld1(Pg, &a[i]);
  va = svmul_x(Pg, va, 2.0);
  svst1(Pg, &a[i], va);
}
```

Compiler Assembly

```
foo(float*, int): // @foo(float*, int)
    cmp w1, #1 // =1
    b.lt .LBB0_3
    mov w9, w1
    mov x8, xzr
    whilelo p1.s, xzr, x9
    ptrue p0.s
.LBB0_2: // =>This Inner Loop Header: Depth=1
    ld1w { z0.s }, p1/z, [x0, x8, lsl #2]
    fmul z0.s, p0/m, z0.s, #2.0
    st1w { z0.s }, p1, [x0, x8, lsl #2]
    incw x8
    whilelo p1.s, x8, x9
    b.mi .LBB0_2
.LBB0_3:
    ret
```

SVE Gives You More

- SVE is really powerful (mainly due to predicates)
  - Compilers can exploit this power
  - Auto-vectorisation getting much better

- Power is also being able to vectorise new things
  - Previously hard to vectorise
  - Mapping IF statements to predicates

arm

Thank You

Danke

Gracias

谢谢

ありがとう

Asante

Merci

감사합니다

धन्यवाद

Kiitos

شكرًا

ধন্যবাদ

תודה

arm