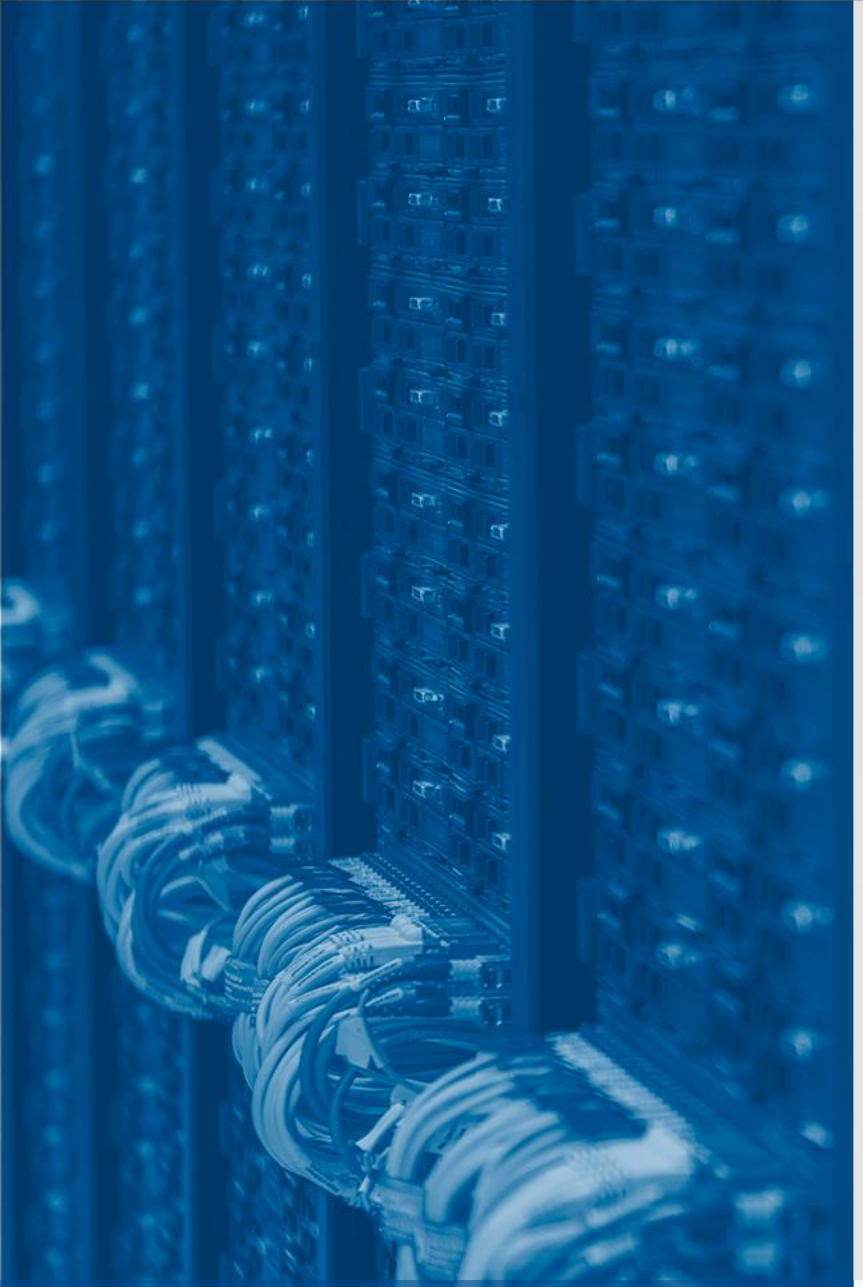


Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Introduction to the RISC-V Vector Extension

Roger Ferrer Ibáñez

August/September 2022

2022 ACM Summer School on HPC and AI

About me

- Research Engineer at the Barcelona Supercomputing Center
  - Compilers and Toolchains for HPC
- Enable research and development in HPC from the compiler side of things
- Some of the things we are looking at these days
  - RISC-V Vector Extension and how to vectorise for it
  - OpenMP and multiple accelerators per node

Outline

- RISC-V
- The Vector Extension
- Compilation
- How to use the Vector Extension

RISC-V

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

RISC-V Architecture

- RISC-V is an open-source-licenced architecture
  - In contrast to many other architectures no licence is required to implement it
  - Created by the University of California at Berkeley in 2010
- RISC-V International fosters the RISC-V Architecture
  - Provides several resources to its members (e.g., educational, compliance, …)
- Unprivileged architecture specification
  - Of interest to any application
- Privileged architecture specification
  - Typically, only of interest to the supervisor (OS), hypervisor and/or firmware

RISC-V ISA Design

- The RISC-V base ISA is very minimal and simple
  - Load/Store-based architecture, one addressing mode
  - Around 50 instructions, only basic integer arithmetic
  - No CPU flags, very similar to MIPS
  - Realistically useful only for very simple CPUs or microcontrollers
- RISC-V instructions are fixed-size 32-bit
  - The encoding allows for 16-bit, 48-bit and 64-bit (and even larger) formats

RISC-V Base ISA

- 32 integer registers of XLEN-bits size (general purpose registers, GPRs)
  - `x0` to `x31`
  - `x0` is constant and hardcoded to all zeros, read-only
- RV32 defines XLEN=32
- RV64 defines XLEN=64
  - RISC-V 64-bit architecture does not provide 32-bit integer registers
- No further state defined

RISC-V Extensions

- RISC-V is augmented via the concept of extensions
  - Extensions can add new instructions and CPU state
- Base ISA is called I (for Integer)
  - RV32I
  - RV64I (XLEN=64, adds a few arithmetic instructions to improve 32-bit integer arithmetic)
- Common Standard Extensions in a RISC-V 64-bit Linux capable core

IMAFD = G

  - M. Integer multiplication and division (mul, div, rem, …)
  - A. Atomic instructions (load reserve + store conditional, atomic read-modify-write)
  - F. Single-Precision Floating-Point (IEEE 754 Binary32)
  - D. Double-Precision Floating-Point (IEEE 754 Binary64)
  - C. Compressed Instructions (16-bit encodings for common I/F/D instructions)

Other extensions of interest

- Zb*. Bit manipulation (adds bitwise operations missing in the base ISA that improve code generation)
- Zfh. Half-Precision Floating Point (IEEE 754 Binary16)
- Zfinx. F in X. (F and D but without a dedicated Floating-Point bank register)
- P. Packed SIMD. (Small-type integer SIMD inside GPR registers)
- Zv*. Vectors (Vector computation)
  - The main topic of this course ☺

- More details about the current specification at
  https://riscv.org/technical/specifications

The Vector Extension

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Vector Extension Design

- The RISC-V Vector Extension (RVV) aims at providing vector computation capabilities to the RISC-V architecture
- RVV wants to have wide applicability, so the design is very flexible
- The flexible design poses some challenges in different aspects when using the ISA
  - Compiler
  - Developers
- The flexibility also allows RVV to be used in more classical SIMD approaches
  - If some assumptions are held

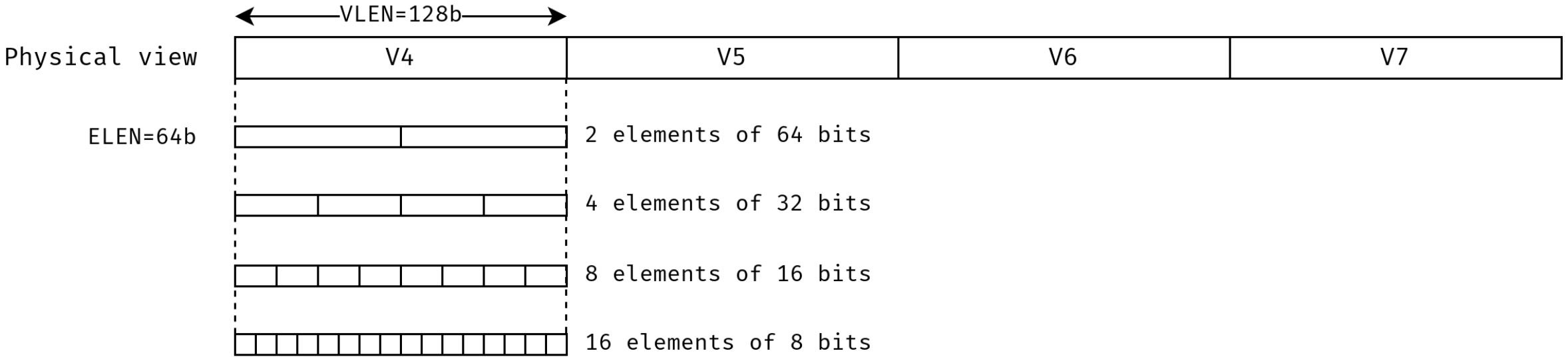
RVV Design

- Vector ISAs are tipically large for several reasons
  - They provide vector equivalents of most scalar arithmetic counterpart instructions
  - They have specific instructions of their own for memory access and vector element manipulation
  - Modern Vector ISAs have some form of predication (masking) support
    - operations to form predicates (e.g., comparisons)
    - additional operands in the instructions
- This impacts the design of RVV
  - Vector operations cannot be encoded in a 32-bit instruction
  - CPU state is used instead

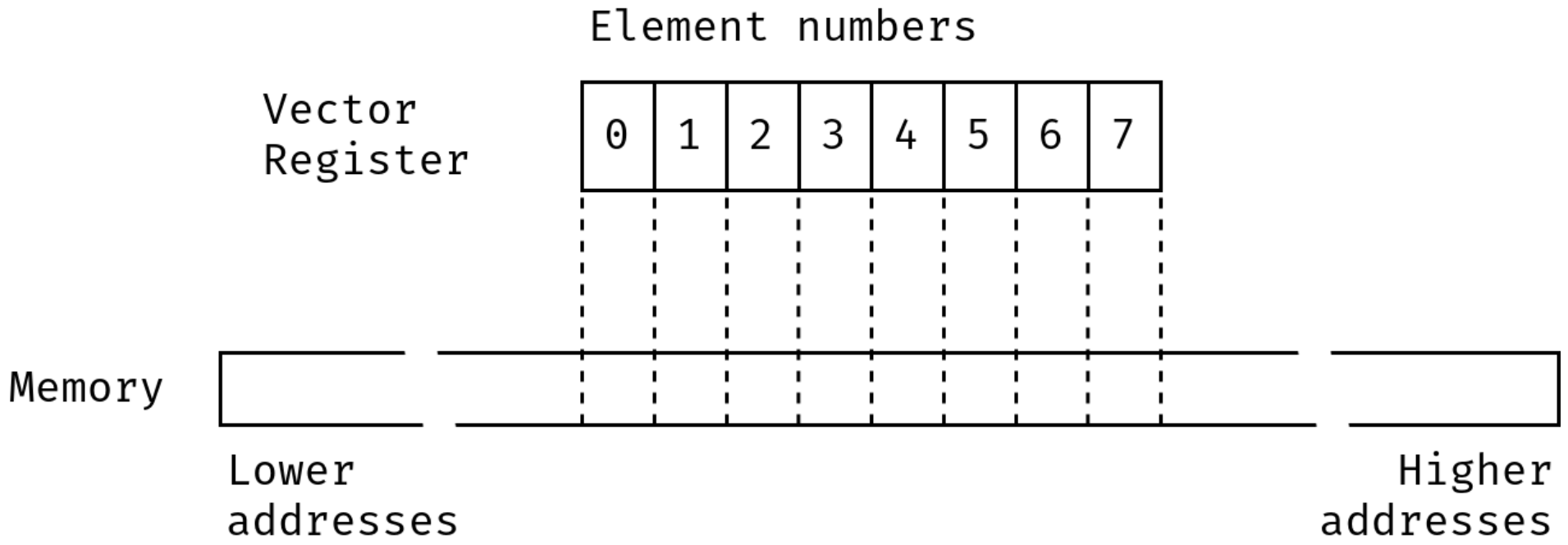
RVV Parameters and Basic State

- RVV defines 32 **vector registers** of size VLEN bits
  - `v0` to `v31`
  - VLEN is a **constant parameter** chosen by the implementor and must be a power of two
    - Zv* standard extensions constraint VLEN to be at least 64 or 128
  - E.g., VLEN=512 would be equivalent in size to Intel AVX-512
  - VLEN is not a great name so read it as “vector register size (in bits)”
- Vectors in RVV are divided in **elements**.
  - The size of elements in bits is at least 8 bits up to ELEN bits
  - ELEN is a **constant parameter** chosen by the implementor
  - Must be a power of two and 8 ≤ ELEN ≤ VLEN
    - Zv* standard extensions constrain ELEN to be at least 32 or 64

Example (1)


VLEN=128b
Physical view
V4
V5
V6
V7
ELEN=64b
2 elements of 64 bits
4 elements of 32 bits
8 elements of 16 bits
16 elements of 8 bits

Convention in these slides

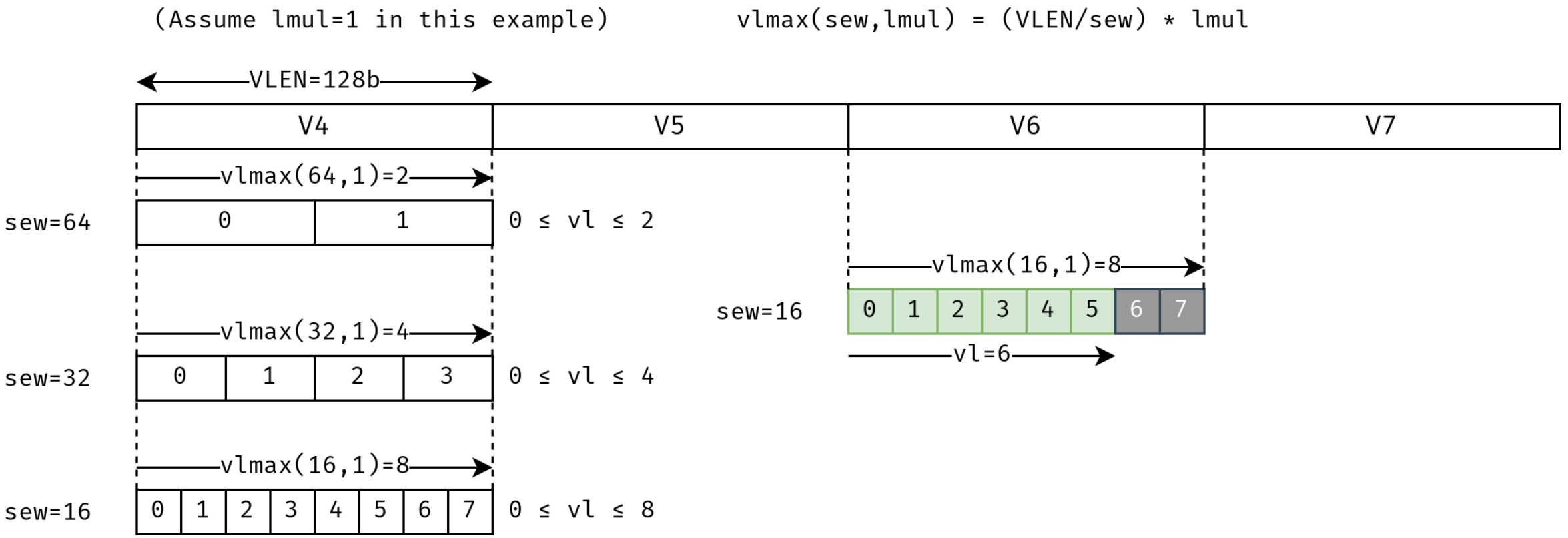


Note: most Vector ISA specifications represent elements in the vector from higher-numbered registers to lower-numbered registers but still map the lower-numbered elements to lower addresses.

RVV Operational State

- There are two **registers** used when operating vectors in RVV
  - **`vtype`**: Vector Type.
  - **`vl`**: Vector Length (not to be confused with VLEN!)
- `vtype` describes **the type of vector we are going to operate** and includes
  - **`sew`**: Standard Element Width. Size in bits of the elements being operated
    - $8 \leq sew \leq ELEN$
  - **`lmul`**: Length Multiplier. Allows grouping registers (more on this later!)
    - $lmul = 2^k$ where $-3 \leq k \leq 3$ (i.e., $lmul \in \{1/8, 1/4, 1/2, 1, 2, 4, 8\}$)
- `vl` describes **how many elements of the vector** (starting from the element zero) **we are going to operate**
  - $0 \leq vl \leq vlmax(sew, lmul)$
  - $vlmax(sew, lmul) = (VLEN / sew) \times lmul$

Example (2)


(Assume lmul=1 in this example)
vlmax(sew,lmul) = (VLEN/sew) * lmul
VLEN=128b
V4
V5
V6
V7
vlmax(64,1)=2
sew=64
0
1
0 ≤ vl ≤ 2
vlmax(32,1)=4
sew=32
0
1
2
3
0 ≤ vl ≤ 4
vlmax(16,1)=8
sew=16
0
1
2
3
4
5
6
7
0 ≤ vl ≤ 8
vlmax(16,1)=8
sew=16
0
1
2
3
4
5
6
7
vl=6

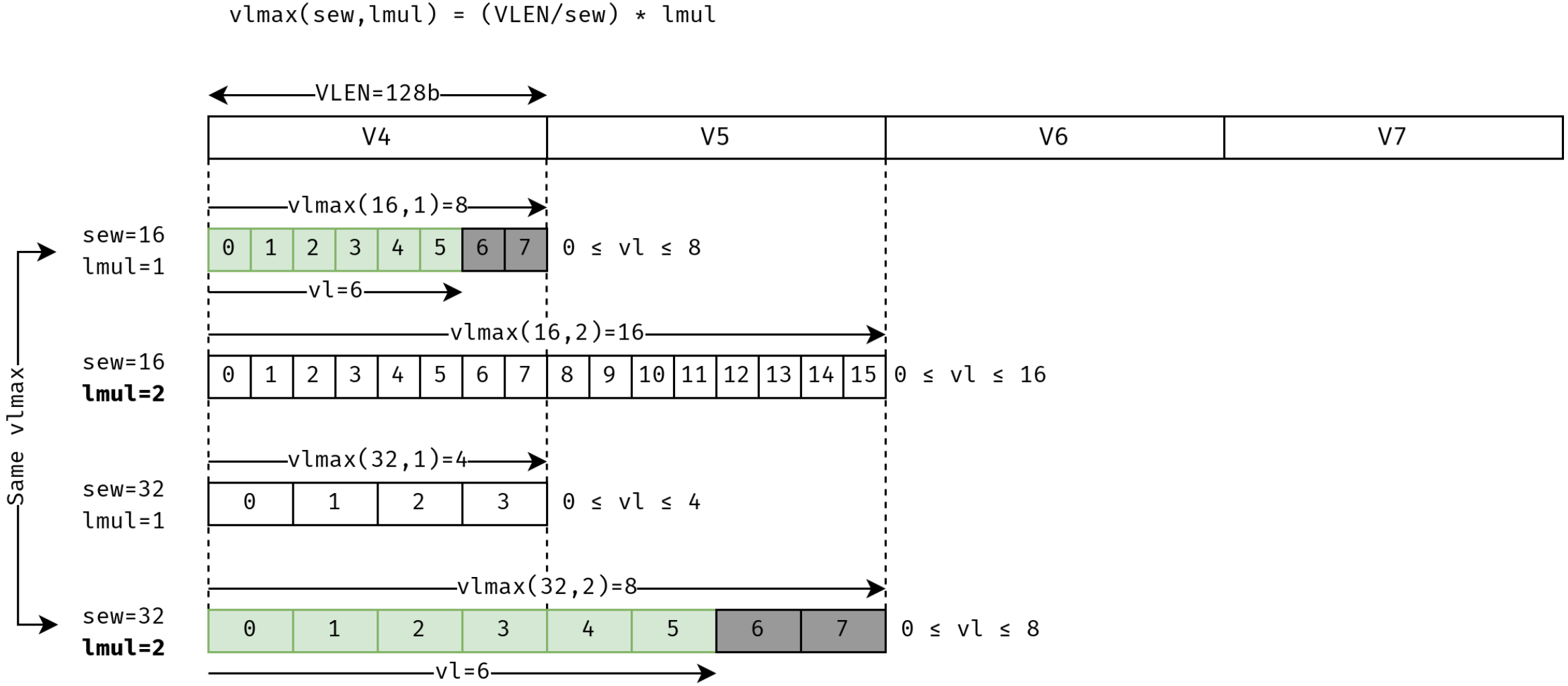
Mixed element sizes

- Vectors with a smaller element size can fit a larger **number of elements**
  - And the opposite: a vector with elements of size ELEN bits can fit the smallest number of elements
- When operating with vectors whose elements are of different size, we have different number of elements
  - This causes problems to algorithms, which want to operate with the same number of elements
- We can "harmonise" the number of elements when the element size is different by either
  - Not using the whole vector register for the small element sizes
  - Use more than one vector register for the large element sizes

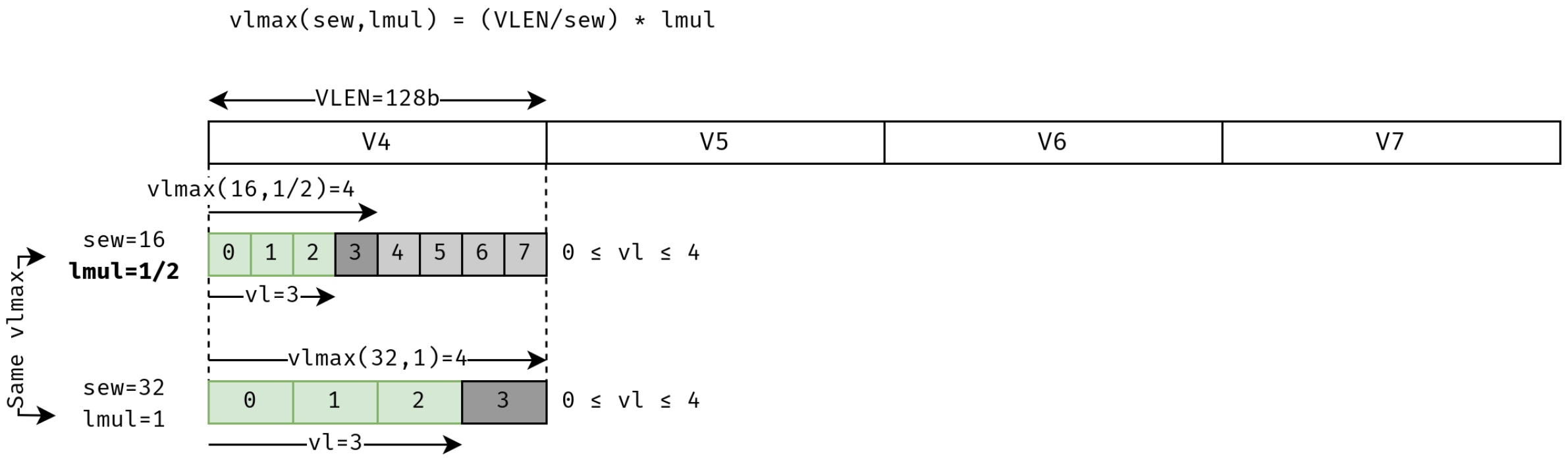
Length multiplier

- RVV allows the two scenarios via the length multiplier
- When `lmul = 1` we can operate up to all the elements of a vector register
- When `lmul < 1` we can operate up to a fraction of all the elements of a vector register `lmul` $\in \{1/2, 1/4, 1/8\}$
- When `lmul > 1` the operation uses a **vector group** of `lmul` vector registers
  - A vector group "gangs" several vector registers. The vector group is identified by the smallest numbered vector register in the group.
  - 16 vector groups of `lmul = 2`
    - `v0, v2, v4, v6, v8, v10, v12, v14, v16, ..., v28, v30`
  - 8 vector groups of `lmul = 4`
    - `v0, v4, v8, v12, v16, v20, v24, v28`
  - 4 vector groups of `lmul = 8`
    - `v0, v4, v8, v16`

Example (3)


vlmax(sew,lmul) = (VLEN/sew) * lmul
VLEN=128b
V4
V5
V6
V7
Same vlmax
sew=16
lmul=1
vlmax(16,1)=8
0 1 2 3 4 5 6 7
0 ≤ vl ≤ 8
vl=6
sew=16
lmul=2
vlmax(16,2)=16
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
0 ≤ vl ≤ 16
sew=32
lmul=1
vlmax(32,1)=4
0 1 2 3
0 ≤ vl ≤ 4
sew=32
lmul=2
vlmax(32,2)=8
0 1 2 3 4 5 6 7
0 ≤ vl ≤ 8
vl=6

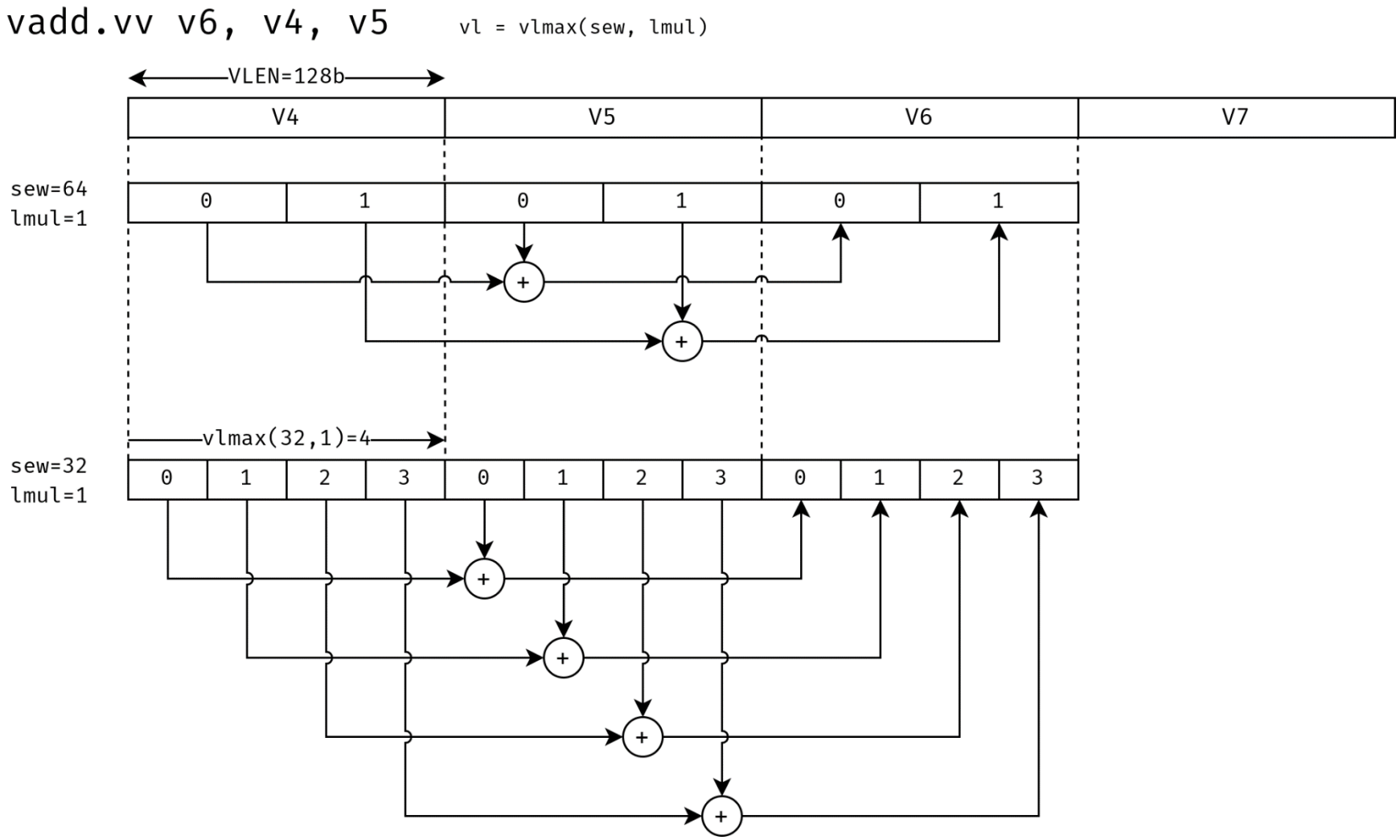
Example (4)


vlmax(sew,lmul) = (VLEN/sew) * lmul
VLEN=128b
V4
V5
V6
V7
vlmax(16,1/2)=4
sew=16
lmul=1/2
0 1 2 3 4 5 6 7
0 ≤ vl ≤ 4
vl=3
Same vlmax
vlmax(32,1)=4
sew=32
lmul=1
0 1 2 3
0 ≤ vl ≤ 4
vl=3

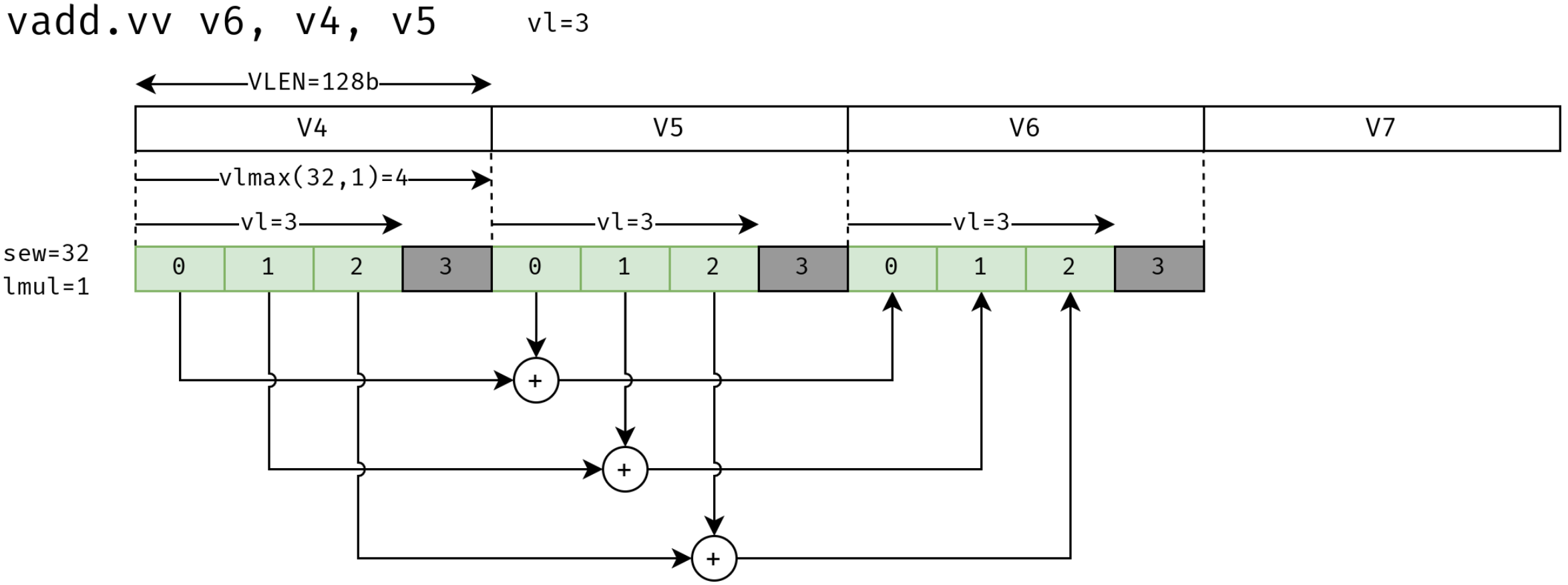
Vector operation

- Vector instructions fully determine the vector operation we are going to execute by using the values of vl and vtype
  - `vl` and `vtype` act as implicit operands of the vector instructions
- When `vl < vlmax` then we have elements that are not operated
  - Those elements are called the **tail elements**
- RVV offers two policies here
  - tail undisturbed. Tail elements in the destination register are left unmodified.
  - tail agnostic. Can behave like tail undisturbed or, alternatively, all the bits of the tail elements of the destination register are set to 1

Example (5)


vadd.vv v6, v4, v5
vl = vlmax(sew, lmul)
VLEN=128b
V4
V5
V6
V7
sew=64
lmul=1
0
1
0
1
0
1
+
+
vlmax(32,1)=4
sew=32
lmul=1
0
1
2
3
0
1
2
3
0
1
2
3
+
+
+
+

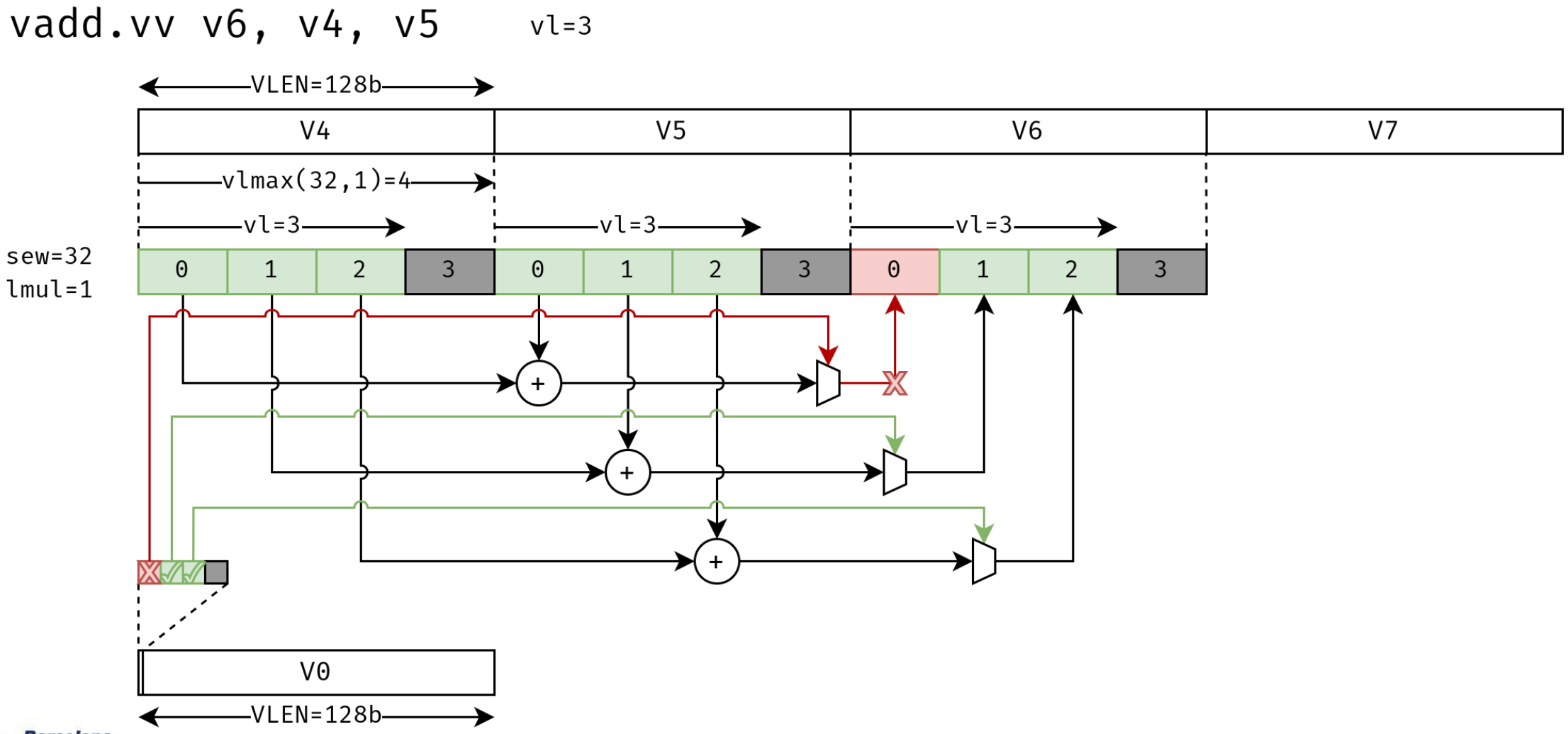
Example (6)


vadd.vv v6, v4, v5
vl=3
VLEN=128b
V4
V5
V6
V7
vlmax(32,1)=4
vl=3
vl=3
vl=3
sew=32
lmul=1
0
1
2
3
0
1
2
3
0
1
2
3
+
+
+

Masking (Predication)

- Control flow may be problematic when using vector instructions. Turning it to data-flow (e.g., if-conversion) allows us to represent the control flow as a value that can be held by a vector
- A mask vector is a vector whose elements are single bits
  - There are no distinguished vector registers for mask vectors (`v0` to `v31` can be used)
  - RVV defines a specific layout for mask vectors where bits are packed contiguously in the vector register, starting from the LSB bit as the 0^{th} element of the mask.
- Instructions can be masked using the `v0` register
  - While it is possible to operate mask vectors in all the other registers, only `v0` can be used as a mask operand when masking a vector instruction

Example (7)


vadd.vv v6, v4, v5
vl=3
VLEN=128b
V4
V5
V6
V7
vlmax(32,1)=4
vl=3
vl=3
vl=3
sew=32
lmul=1
0
1
2
3
0
1
2
3
0
1
2
3
+
+
+
V0
VLEN=128b

Masked Vector operation

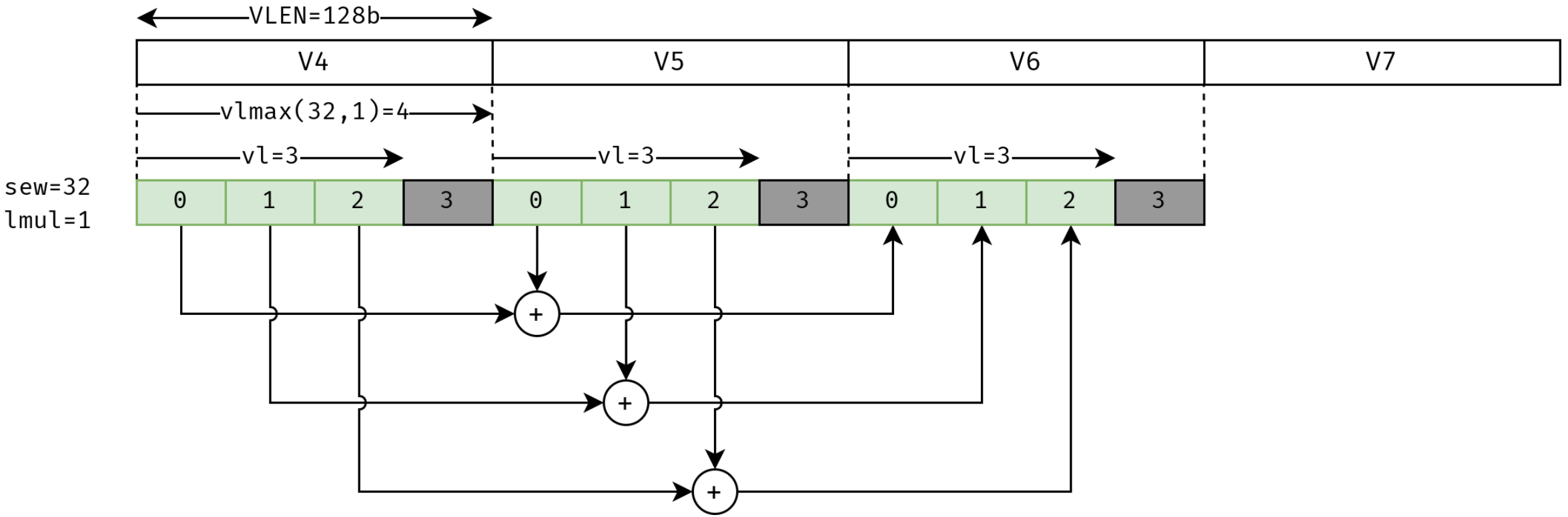
- When executing a vector operation, the `v0` register (interpreted as a mask vector layout) determines whether a non-tail element is **active** or **inactive**.
  - Active elements operate as usual
  - Inactive elements are not operated at all
- Inactive elements have a policy as well
  - mask undisturbed. The corresponding element of the destination register is left unmodified.
  - mask agnostic. The corresponding element of the destination register is either left unmodified or all its bits are set to 1.

Setting `vl` and `vtype`

- Generic RISC-V instructions cannot set `vl` or `vtype`
- There are two cases where vl and/or vtype can change
  - `vsetvl` / `vsetvli` / `vsetivli` instructions ("set vector length")
  - `vle*ff` instructions
- Set vector length instruction set the vl and the vtype
- The most common one is vsetvli
  - `vsetvli rd, rs, eN,mX,tP,mP` (updates `rd` with the vector length computed)
  - `rs` is an input register operand that contains the **application vector length** (AVL) which represents the vector length the program wants to use
    - `vseti`**`i`**`vli` replaces this operand with a small immediate from 0 to 31.
  - `N` in `eN` is the `sew` (8, 16, 32, 64, …)
  - `X` in `mX` is the `lmul` (spelled as `fY` for `1/Y` cases)
  - `P` is the policy for tail (`t`) and mask (`m`): `u` for undisturbed, `a` for agnostic

Example (8)

```
vsetivli x10, 3, e32,m1,ta,ma   # vl ← 3, sew ← 32, lmul ← 1
vadd.vv v6, v4, v5
```



Special cases setting the vl

- `vsetvli rd, x0, eN,mX,tP,mP  # rd != x0`
  - Sets `vl` to `vlmax(sew=N, lmul=X)` and `vtype` to `sew=N, lmul=X`

  Note: If only the VLEN is needed, a dedicated register VLENB exists that returns VLEN in bytes (i.e. VLENB = VLEN/8)
- `vsetvli x0, x0, eN,mX,tP,mP`
  - Only changes `vtype` (assumes the application vector is the current `vl`)
  - Only valid when the new `vlmax` is left unchanged

```
vsetivli x0, 10, e32,m1,ta,ma      # vlmax = (VLEN/32) * 1 = VLEN/32
<vector operations with sew=32, lmul=1>
vsetivli x0, x0, e16,mf2,ta,ma     # vlmax = (VLEN/16) * 1/2 = VLEN/32
<vector operations with sew=16, lmul=1/2>
```

What if AVL > `vlmax(sew,lmul)`?

- The specification allows computing the new `vl` as
  `vl = min(vlmax(sew, lmul), AVL)`
- However, when `vlmax(sew,lmul) < AVL < 2*vlmax(sew,lmul)` an implementation may compute `ceil(AVL / 2) ≤ vl ≤ vlmax(sew, lmul)`
- Example: `VLEN=128 → vlmax(sew=16, lmul=1) = 8`
  `vsetivli x0, 9, e16,m1,ta,ma`
  `vl` will be such that `5 ≤ vl ≤ 8`
- This makes the precise value of `vl` a bit unpredictable if we cannot assert that `AVL ≤ vlmax(sew, lmul)`
  - Note: due to the lower bound being `ceil(AVL / 2)` and `AVL < 2*vlmax(sew,lmul)`, we know the remaining elements to process is going to be less than `vlmax(sew, lmul)`.

Many more details we cannot cover

- We cannot cover many more details of the ISA of the V extension
- If I piqued your interest, I recommend you look at the full specification here:
  - https://github.com/riscv/riscv-v-spec/releases/download/v1.0/riscv-v-spec-1.0.pdf

Comparison to other Vector/SIMD ISAs

Feature	Intel AVX-512	Arm SVE	NEC SX-Aurora TSUBASA	RISC-V Vector
Is the vector register size defined by the architecture?	**Yes**. 512 bit VLE extension allows using 128-bit (SSE) and 256-bit (AVX-2) registers.	**No**. From 128 bit to 208 bits (in multiples of 128 bits)	**Yes**. Current generation is 16,384 bits.	**No**. Powers of two, from 64/128 up to 65,536 bits.
Predication/Masking	**Yes**. 8 mask registers `k0-k7` (`k0` hardcoded to all ones)	**Yes**. 16 vector predicate registers `p0-p15`. (`p0-p7` masking, `p8-p15` loops)	**Yes**. 16 vector mask registers.	**Yes**. Only `v0` as an implicit operand if the instruction is masked.
Set vector length	**No**	**No** (Privileged, compatibility-only)	**Yes**	**Yes**

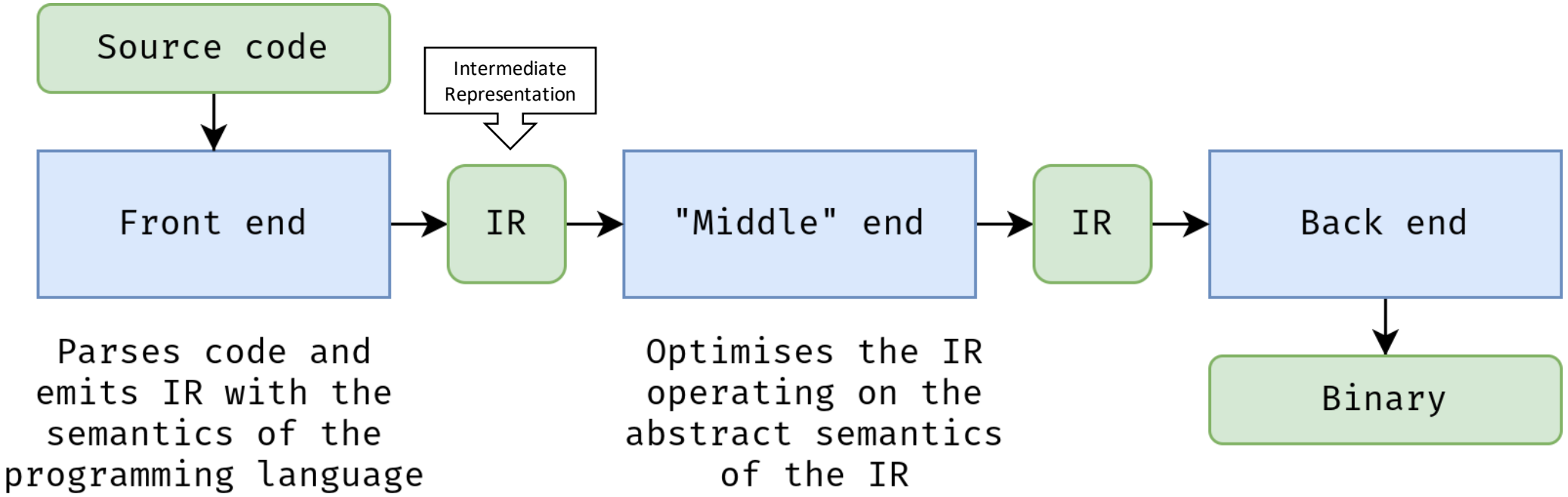
This table is by no means meant to be exhaustive, there are other important differences like the set of data types supportedby the ISA (e.g. fixed floating types, complex, polynomials, saturated arithmetic, etc).

Compilation

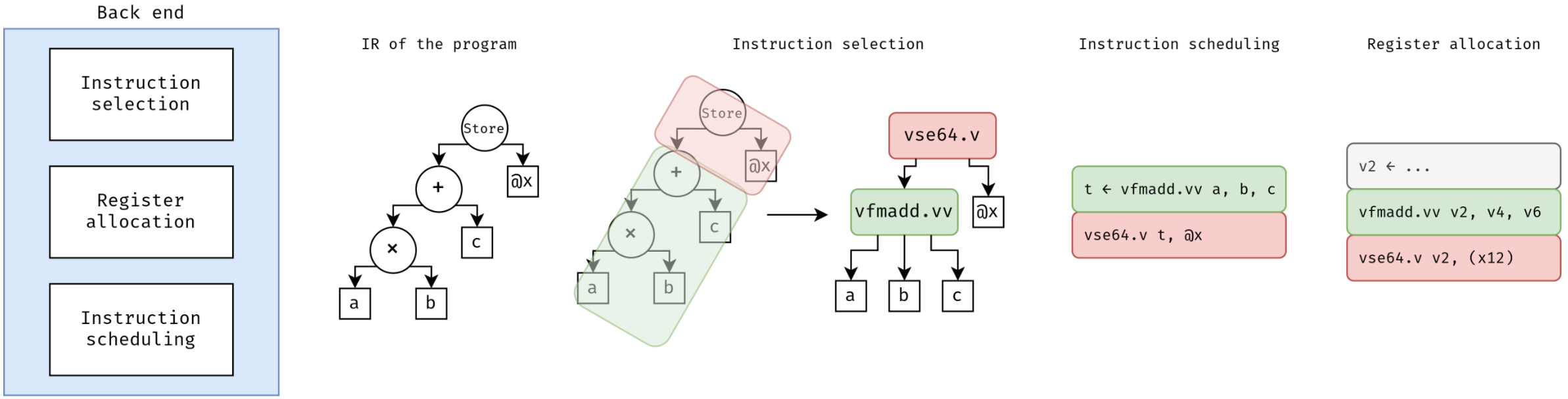
(What we did in LLVM)

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Super quick summary of how compilers work



Back end



Note: this is very simplified!

Modern compilation

- Modern compiler infrastructures build on top of the concept of **virtual registers**
- The compiler assumes it has an infinite amount of virtual registers
- Virtual registers are then mapped onto **physical** (architectural) **registers** of a specific kind in a process called **register allocation**
- This process assumes that a **temporary area** exists to temporarily keep values in case there are not enough physical registers of a given kind
- This area is commonly the **memory** in a process called "**register spilling**" but it could be other (kinds of) registers too (at risk of spilling those too!)

Modern compilation and CPU state

- CPU state in the form of specific registers does not fit well the virtual register compilation model
  - Examples of such registers: vl, vtype, FP status, etc.
- There is just one register, so nothing must be assigned
- Often pack many details
  - FP status register might contain bits for rounding mode and bits for the result of the last FP instruction
  - FP arithmetic instructions commonly use the former bits and define the latter bits
- Often there is an expectation that those registers **are** implicit
  - An FP routine can use the same instructions but use a different rounding mode by setting it beforehand
- All these details impact how the backend of the compiler represents the instructions

What we did for RVV in LLVM

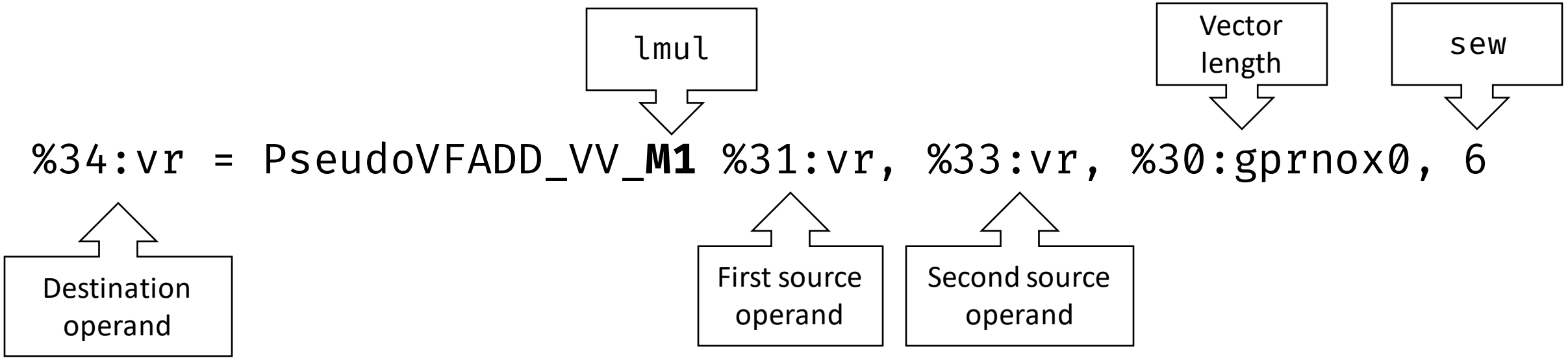
- In order to have a sensible representation of the instructions in the RISC-V backend we chose to do the following:
- `vl` and `sew` are explicit operands of the instructions
  - `sew` are always an immediate
- `lmul` is represented in the instruction opcode used by the compiler
  - each (ISA) instruction has 7 opcodes associated in the compiler
  - (the reason is that we defined 4 vector register classes, one for each lmul ≥ 1 but LLVM does not allow overloading opcodes over different register classes)
- Instruction selection must select the right instruction based on the `lmul` and set the right `sew` (those depend on the vector types used by the IR) and provide a vector length.

Where does the vector length come from?

- If we are operating as whole vectors, the vector length is basically the `vlmax(sew, lmul)`
  - This happens because LLVM IR has an elementwise extension from scalar type operations to vector type operations for almost every arithmetic instruction
- If not, the vector length is somehow provided by the user
  - RVV IR intrinsics almost verbatim from the RVV C/C++ builtins
  - Vector Predication IR that includes a vector length operand

What does this give us?

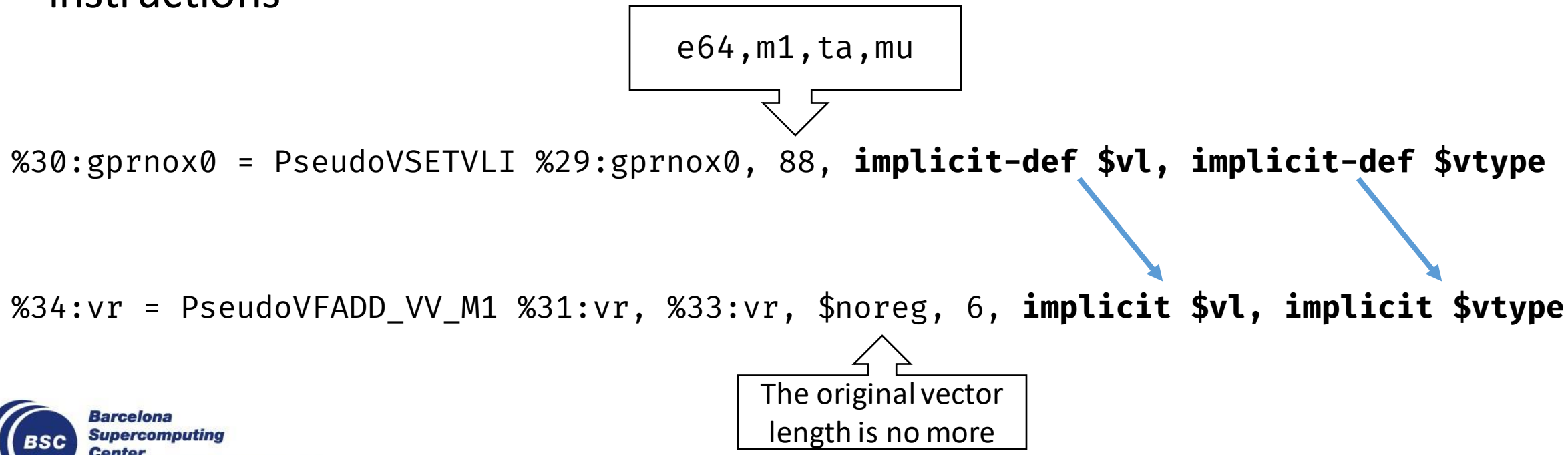
- Right after instruction selection, instructions in the low-level IR of the compiler look like this



- This is not valid RVV code but represents the intent correctly

Code generation

- Now that we have associated each instruction with the sew and vector length they need, it is time to make sure both vtype and vl are correctly set in the CPU state.
- A pass analyses the instructions and inserts the needed `vsetvli` instructions



Final emission

- After inserting the `vsetvli` instructions, register allocation can run as usual
- The extra operands can be ignored for the final emission of the instructions

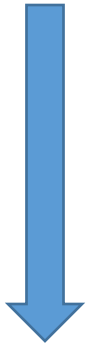
```
vsetvli  t0, a5, e64,m1,ta,mu
vfadd.vv v8, v8, v9
```

How to program with the Vector Extension

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Ways to use RVV

- There are many different ways to use the RVV instructions
  - Assembly
  - C/C++ Builtins
  - Automatic (or semi-automatic) vectorization
  - Libraries and/or kernels



- Productivity
+ Control

+ Productivity
- Control

- Some of those approaches may rely on JIT
  - OpenCL
  - SYCL

EPI built-ins

- As part of the EPI project, at BSC we implemented an initial set of intrinsics (even before any other intrinsics for RVV existed)
- Explicit vector length obtained via vsetvl built-ins
- Not all RVV instructions are accessible, LMUL<1 not possible
- https://repo.hca.bsc.es/gitlab/rferrer/epi-builtins-ref

```
void saxpy(size_t n, const float a,
           const float *x, float *y) {
  size_t i;
  for (i = 0; i < n; i++) {
    y[i] = a * x[i] + y[i];
  }
}
```

```
void saxpy_epi(size_t n, const float a, const float *x, float *y) {
  for (size_t i = 0; i < n; /* */) {
    size_t vl = __builtin_epi_vsetvl(n - i, __epi_e32, __epi_m8);
    __epi_16xf32 vx = __builtin_epi_vload_16xf32(&x[i], vl);
    __epi_16xf32 vy = __builtin_epi_vload_16xf32(&y[i], vl);
    __epi_16xf32 va = __builtin_epi_vfmv_v_f_16xf32(a, vl);
    vy = __builtin_epi_vfmacc_16xf32(vy, va, vx, vl);
    __builtin_epi_vstore_16xf32(&y[i], vy, vl);
    i += vl;
  }
}
```

RVV C/C++ Built-ins

- Covers all the RVV instructions (~40,000 built-ins)
- Same philosophy as EPI with respect to the explicit vector length
- Full specification at
  - https://github.com/riscv-non-isa/rvv-intrinsic-doc

```
void saxpy(size_t n, const float a,
           const float *x, float *y) {
  size_t i;
  for (i = 0; i < n; i++) {
    y[i] = a * x[i] + y[i];
  }
}
```

```
void saxpy_intrinsics(size_t n, const float a,
                      const float *x, float *y) {
  for (size_t i = 0; i < n; ) {
    size_t vl = vsetvl_e32m8(n - i);
    vfloat32m8_t vx = vle32_v_f32m8(&x[i], vl);
    vfloat32m8_t vy = vle32_v_f32m8(&y[i], vl);
    vy = vfmacc_vf_f32m8(vy, a, vx, vl);
    vse32_v_f32m8(&y[i], vy, vl);
    i += vl;
  }
}
```

Beyond built-ins

- Built-ins may be unavoidable in some situations, but they require taking care of low-level concerns
  - Libraries typically will use them in their optimised implementations
- Typical alternatives here involve some amount of vectorization
  - SLP vectorization
  - Loop vectorization: `#pragma omp simd`
  - Whole function vectorization: OpenCL, SYCL, `#pragma omp declare simd`

Loop vectorization

- Loops may be (intuitively) vectorized if
  - They do not have loop-carried dependences (i.e., a parallel loop)
  - Or they do, but the “source” and the “sink” of all loop-carried dependences are many iterations apart through the iteration space
  - Note that there are some cases where this definition would not apply yet the loop would still be vectorizable, e.g., reductions.
- During vectorization, the compiler replaces scalar operations as vector operations
  - Those vector operations are ultimately mapped to vector instructions

Vector-length specific / agnostic

- Traditionally vectorization targets a specific vector register size known to the compiler
  - This is the natural approach for vector ISAs that prescribe the size of the vector register such as AVX-512
  - This has been called “vector-length specific”(VLS) vectorization
- Arm SVE introduced “vector-length agnostic” (VLA) vectorization
  - Useful for architectures where the implementation determines the size of the vector register as it avoids having many versions for the different vector sizes
- RISC-V (and SVE) can use either approach
  - The compiler must be told the minimum vector register size it can assume
  - For RISC-V, at BSC we have focused on VLA
- A JIT can be used as a hybrid scheme that at runtime can do “adaptive” VLS

Current Vectorization Schemes in LLVM

- In a vector loop we must consider what to do when the number of iterations is lower than the number of elements that can fit in the vector register.
- Classical scheme implemented in LLVM: a first loop that operates with full vectors followed by a scalar loop (epilog) that processes the remainder elements.
  - Cons: two loops, long vectors increase the risk that the program only executes the epilog (without entering the vector loop). The epilog can be vectorized again with shorter vectors.
- "Tail folding": only one vector loop, but first compute a mask that disables the elements that would be past the loop boundary. Use the mask in all the operations that need it.
  - Pro: one loop
  - Cons: needs to compute a mask, the mask is only used for loads/stores

Loop vectorization with RVV

- As part of the EPI project, we extended the LLVM loop vectorizer.
- Uses the Vector Predication IR proposed by Simon Moll
  - Operations in this IR have an explicit vector length and mask
  - https://llvm.org/docs/Proposals/VectorPredication.html
- Vector Predication IR maps well to RISC-V
  - But it is of application for other vector ISAs such as AVX-512 or SVE

Vector length-based vectorization

- Following the style of the “tail folding” we can compute the vector length using the remaining number of iterations
- Like tail folding passes the mask to all the vector operations, we pass the vector length of the current vector loop iteration to all the vector operations
  - Pros: one loop
  - Cons: need to compute the vector length in each iteration

Example: LLVM IR

```
void add_ref(int N, double *c, double *a, double *b) {
  for (int i = 0; i < N; i++)
    c[i] = a[i] + b[i];
}
```

```
vector.body: ; preds = %for.body.preheader, %vector.body
  %index = phi i64 [ %index.next, %vector.body ], [ 0, %for.body.preheader ], !dbg !28
  %3 = getelementptr inbounds double, ptr %a, i64 %index
  %4 = sub i64 %wide.trip.count, %index
  %5 = tail call i64 @llvm.epi.vsetvl(i64 %4, i64 3, i64 0)
  %6 = trunc i64 %5 to i32
  %vp.op.load = tail call <vscale x 1 x double> @llvm.vp.load.nxv1f64.p0(ptr %3, <vscale x 1 x i1> %true_mask, i32 %6)
  %7 = getelementptr inbounds double, ptr %b, i64 %index
  %vp.op.load13 = tail call <vscale x 1 x double> @llvm.vp.load.nxv1f64.p0(ptr %7, <vscale x 1 x i1> %true_mask, i32 %6)
  %vp.op = tail call <vscale x 1 x double> @llvm.vp.fadd.nxv1f64(<vscale x 1 x double> %vp.op.load,
      <vscale x 1 x double> %vp.op.load13, <vscale x 1 x i1> %true_mask, i32 %6)
  %8 = getelementptr inbounds double, ptr %c, i64 %index
  tail call void @llvm.vp.store.nxv1f64.p0(<vscale x 1 x double> %vp.op, ptr %8, <vscale x 1 x i1> %true_mask, i32 %6)
  %9 = and i64 %5, 4294967295
  %index.next = add i64 %index, %9
  %10 = icmp eq i64 %index.next, %wide.trip.count
  br i1 %10, label %for.cond.cleanup, label %vector.body
```

Example: assembly

```
void add_ref(int N, double *c, double *a, double *b) {
  for (int i = 0; i < N; i++)
    c[i] = a[i] + b[i];
}
```

```
.LBB0_4: # %vector.body
  slli a7, a4, 3
  add a6, a2, a7
  sub a5, a0, a4
  vsetvli t0, a5, e64, m1, ta, mu
  vle64.v v8, (a6)
  add a5, a3, a7
  vle64.v v9, (a5)
  vfadd.vv v8, v8, v9
  add a5, a1, a7
  add a4, a4, t0
  vse64.v v8, (a5)
  bne a4, a0, .LBB0_4
```

Strided Accesses

- Sometimes loops have to do “strided accesses”
  - For instance, loops that operate with arrays of complex numbers
- However, LLVM does not have the notion of strided memory access
- Complex memory accesses are handled as scatter/gather.
  - This works for VLS vectorization (by analysing the vector indices), but it is harder to do under VLA
- We have contributed experimental Vector Predication support to LLVM
  - The loop vectorizer can identify such accesses and use strided memory accesses

Example

```
void zaxpy(_Complex float a,
           _Complex float * __restrict dx,
           _Complex float * __restrict dy,
           int n) {
for (int i = 0; i < n; i++) {
    dy[i] += a * dx[i];
  }
}
```

`_Complex` types are pairs of the real and imaginary parts.

This code has been compiled using `-Ofast` so the complex multiplication does not check for NaN or infinite values (involves a runtime call that prevents vectorization)

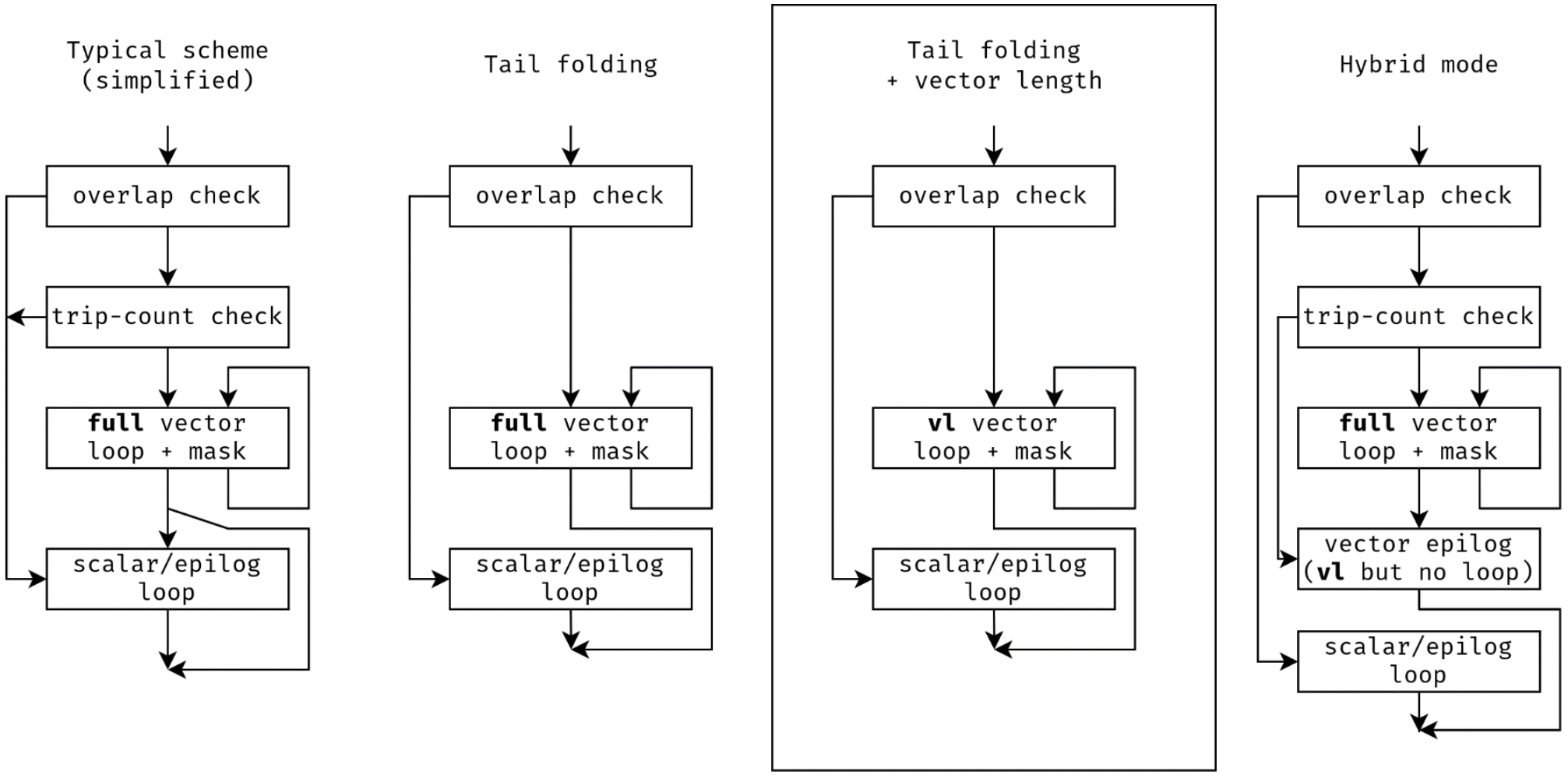
Excerpt of the vector body using gather (`vluxe`) and scatter (`vsoxe`)

```
…
vluxei64.v  v11, (zero), v11
vluxei64.v  v10, (zero), v10
vfmul.vf    v12, v11, fa0
vfmacc.vf   v12, fa1, v10
vfmul.vf    v11, v11, fa1
vfmsac.vf   v11, fa0, v10
vsetvli     a5, zero, e64, m1, ta, mu
vadd.vx     v9, v9, a1
vsetvli     zero, a4, e64, m1, ta, mu
vluxei64.v  v10, (zero), v9
vsetvli     a5, zero, e64, m1, ta, mu
vadd.vi     v13, v9, 8
vsetvli     zero, a4, e64, m1, ta, mu
vluxei64.v  v14, (zero), v13
vfadd.vv    v10, v11, v10
vfadd.vv    v11, v12, v14
vsoxei64.v  v10, (zero), v9
vsoxei64.v  v11, (zero), v13
…
```

The whole loop using strided load (`vlse`) and strided store (`vsse`)

```
.LBB0_5:
  sub         a5, a2, a3
  vsetvli     t0, a5, e64, m1, ta, mu
  slli        a5, a3, 4
  add         a4, a6, a5
  vlse64.v    v8, (a4), t1
  add         a4, a0, a5
  vlse64.v    v9, (a4), t1
  vfmul.vf    v10, v8, fa0
  vfmacc.vf   v10, fa1, v9
  add         a4, a1, a5
  vlse64.v    v11, (a4), t1
  add         a5, a5, a7
  vlse64.v    v12, (a5), t1
  vfmul.vf    v8, v8, fa1
  vfmsac.vf   v8, fa0, v9
  vfadd.vv    v8, v8, v11
  vfadd.vv    v9, v10, v12
  vsse64.v    v8, (a4), t1
  add         a3, a3, t0
  vsse64.v    v9, (a5), t1
  bne         a3, a2, .LBB0_5
```

Summary of vectorization schemes



Vectorization may fail

- Compilers must be conservative with the analyses they perform to avoid breaking the program semantics.
- Sometimes the compiler will not vectorize a loop.
- clang supports the following flags that can help identifying the reasons
  - `-Rpass=loop-vectorize`
    - Reports successfully vectorized loops.
  - `-Rpass-missed=loop-vectorize`
    - Reports loops that were not vectorized.
  - `-Rpass-analysis=loop-vectorize`
    - Reports extra details about why a loop failed to vectorize

Vectorization report

```
 1  void works(long N, double *c) {
 2    long i;
 3    for (i = 0; i < N-1; i++) {
 4      c[i] = c[i+4]*0.5;
 5    }
 6  }
 7
 8  void fails(long N, double *c) {
 9    long i;
10    for (i = 4; i < N; i++) {
11      c[i] = c[i-4]*0.5;
12    }
13  }
14
```

```
test.c:3:3: remark: the cost-model indicates that
interleaving is not beneficial [-Rpass-analysis=loop-
vectorize]

  for (i = 0; i < N-1; i++) {

  ^


test.c:3:3: remark: vectorized loop (vectorization
width: vscale x 1, interleaved count: 1) [-Rpass=loop-
vectorize]

test.c:10:3: remark: loop not vectorized: Scalable
vectorization does not support vectorizing loops that
are not parallel yet [-Rpass-analysis=loop-vectorize]

  for (i = 4; i < N; i++) {

  ^
```

VLS Loop Vectorization

- Vector Length Specific (VLS) can be used in RISC-V as well
- One must tell the compiler, the minimum size of the register it can assume
  - `-mllvm -riscv-v-vector-bits-min=<vlen>`
  - (At some point `–mcpu=<cpu-name>` will internally do that as well)
- Con: the code will only work in CPUs with equal or larger `VLEN`
  - We could always use the scalar epilogue as a fallback if need be
- Pro: `vl` is only set outside the loop (though `vtype` might still have to change)
- Pro: The compiler can generate better code because it is fully aware of the size of the vector register.

Example

```
void daxpy(double a,
           double * __restrict dx,
           double * __restrict dy,
           int n) {
  int i;
  for (i = 0; i < n; i++) {
    dy[i] += a * dx[i];
  }
}
```

`-mllvm -riscv-v-vector-bits-min=256`

The compiler has chosen to do “interleaving” in which it effectively uses several (in this case 2) vector operations per each scalar operation.

LMUL=2 could be an alternative here if we care about code size.

Using VLEN ≥ 256 we can fit 4 doubles in each register

```
.LBB0_3:
        andi    a6, a2, -8
        vsetivli        zero, 4, e64, m1, ta, mu
        vfmv.v.f        v8, fa0
        mv      a4, a6
        mv      a5, a1
        mv      a3, a0
.LBB0_4:
        addi    a7, a5, 32
        addi    t0, a3, 32
        vle64.v v9, (a3)
        vle64.v v10, (t0)
        vle64.v v11, (a5)
        vle64.v v12, (a7)
        vfmacc.vv       v11, v8, v9
        vfmacc.vv       v12, v8, v10
        vse64.v v11, (a5)
        vse64.v v12, (a7)
        addi    a3, a3, 64
        addi    a4, a4, -8
        addi    a5, a5, 64
        bnez    a4, .LBB0_4
        beq     a6, a2, .LBB0_8
```

Scalar epilogue not shown!

SLP Vectorization

- SLP (Superlevel Word Parallelism) is an alternate vectorization approach that identifies repeated scalar operations and coalesces them using vector instructions
- SLP is most practical only when we know the size of the vector register
  - A scalable version is possible: check that the VLEN is large enough and branch to the original scalar code though this has a code-size impact.

SLP Vectorization: Example

```
void saxpy8(float *__restrict A,
            float *__restrict B,
            float C) {
  A[0] += C*B[0];
  A[1] += C*B[1];
  A[2] += C*B[2];
  A[3] += C*B[3];
  A[4] += C*B[4];
  A[5] += C*B[5];
  A[6] += C*B[6];
  A[7] += C*B[7];
}
```

```
saxpy8: # -mllvm -riscv-v-vector-bits-min=128
        vsetivli        zero, 4, e32, m1, ta, mu
        vle32.v v8, (a1)
        vle32.v v9, (a0)
        vfmacc.vf       v9, fa0, v8
        vse32.v v9, (a0)
        addi    a1, a1, 16
        addi    a0, a0, 16
        vle32.v v8, (a1)
        vle32.v v9, (a0)
        vfmacc.vf       v9, fa0, v8
        vse32.v v9, (a0)
        ret
```

```
saxpy8: # -mllvm -riscv-v-vector-bits-min=256 (or larger)
        vsetivli        zero, 8, e32, m1, ta, mu
        vle32.v v8, (a1)
        vle32.v v9, (a0)
        vfmacc.vf       v9, fa0, v8
        vse32.v v9, (a0)
        ret
```

OpenMP SIMD

- As of version 4.0, OpenMP has constructs that assist with vectorization
  - `#pragma omp simd`
    - Loop Vectorization
  - `#pragma omp declare simd`
    - For functions called in “#pragma omp simd” loops
- `#pragma omp simd` reuses the existing Loop Vectorization infrastructure
  - May relax a few legality checks
- We have been implementing support for functions in `#pragma omp declare simd` where the function receives a vector length
  - This way it can be used in loops vectorized using vector length

Example

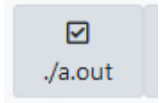
BSC suggestion

```
#pragma omp declare simd simdlen(omp_max_simdlen : 2) notinbranch
double example(double a, double b) {
   return 1/(1/a + 1/b);
}
```

Experimental clause for VLA vectorization under discussion at OpenMP

```
_ZGVENk2vv_example:
  lui a1, %hi(.LCPI1_0)
  fld ft0, %lo(.LCPI1_0)(a1)
  vsetvli zero, a0, e64, m2, ta, mu
  vfrdiv.vf v8, v8, ft0
  vfrdiv.vf v10, v10, ft0
  vfadd.vv v8, v8, v10
  vfrdiv.vf v8, v8, ft0
  ret
```

Try our compiler!

- You can toy with our compiler for RVV in our Compiler Explorer instance
  - https://repo.hca.bsc.es/epic
- Make sure you enable at least `-O2` to enable vectorization
  - Defaults to vector length-based vectorization
- Click on 

to execute using qemu with VLEN=512 bits
- Click on 

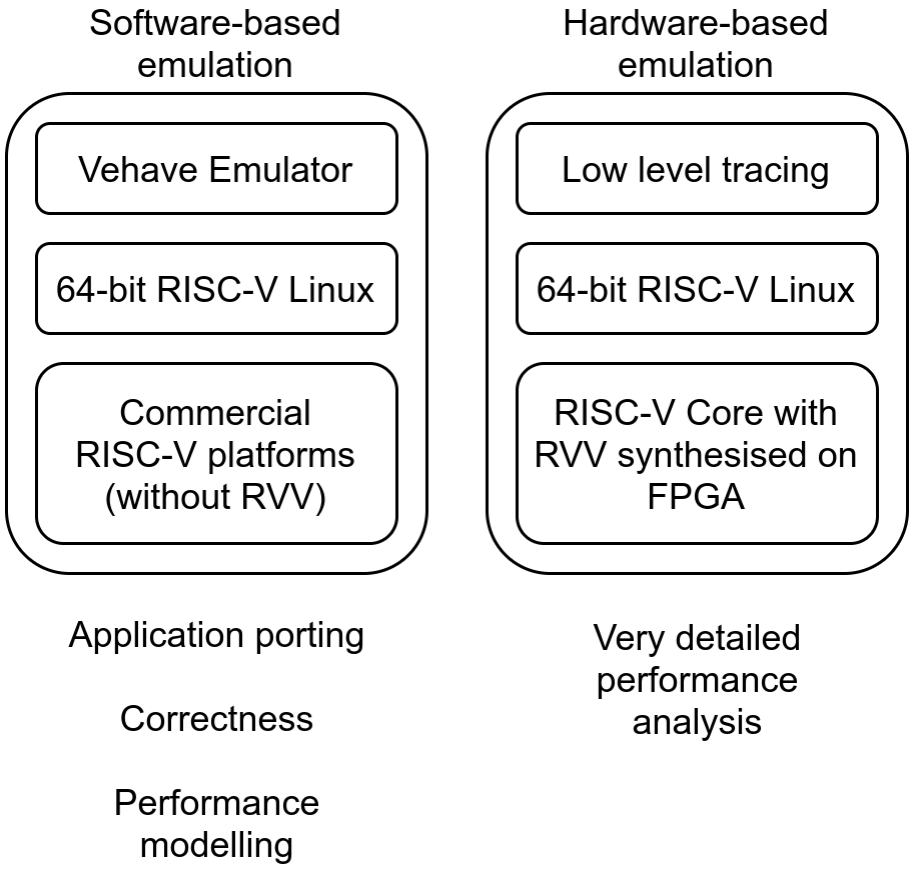
for examples

- To vectorize functions `#pragma omp declare simd` make sure you pass

  `–fopenmp-simd –Xclang –vectorize-wfv`
- `linear` and `uniform` clauses not implemented yet
- Functions cannot contain loops
  - This is a limitation of LLVM's Loop Vectorizer currently

Software Development Vehicles

- How to create a quick loop of feedback for hardware-software co-design?
- Software Development Vehicles with progressive performance fidelity



Vehave emulator

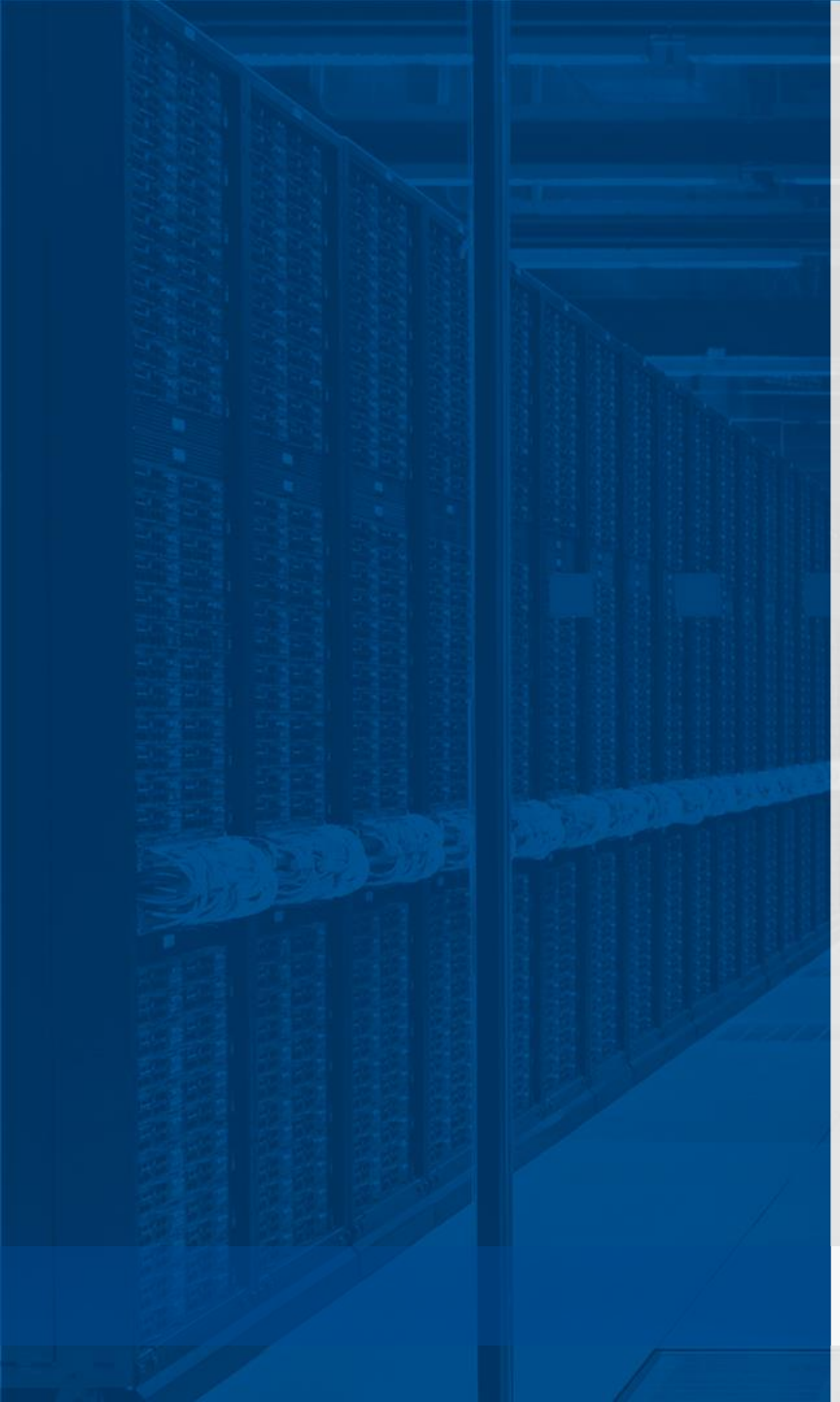
- Trap-based emulator (slow!)
- Allow us to check correctness of the compiler and the porting of applications to RVV
  - qemu can be used as an alternative for this
- Generates traces that can be used for performance modelling and compiler code generation analysis using tools like Paraver and MUSA
- Runs on any Linux RISC-V 64-bit platform that does not have RVV support

Wrap-up

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Conclusions

- RISC-V Vector Extension is a powerful and flexible vector ISA
- Vector length as a convenient way to vectorize loops and functions
- It is possible to generate efficient code in RVV using LLVM
- Comprehensive set of C/C++ built-ins
- LLVM Loop Vectorizer can be used to vectorize for RVV
- Now working on making OpenMP SIMD useable too



Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Thank you!

The European Processor Initiative (EPI) has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement EPI-SGA1: 826647 and under EPI-SGA2: 101036168. Please see http://www.european-processor-initiative.eu for more information.

The European PILOT project has received funding from the European High-Performance Computing Joint Undertaking (JU) under grant agreement No.101034126. The JU receives support from the European Union's Horizon 2020 research and innovation programme and Spain, Italy, Switzerland, Germany, France, Greece, Sweden, Croatia and Turkey.

The MEEP project has received funding from the European High-Performance Computing Joint Undertaking (JU) under grant agreement No 946002. The JU receives support from the European Union's Horizon 2020 research and innovation programme and Spain, Croatia, Turkey

roger.ferrer@bsc.es