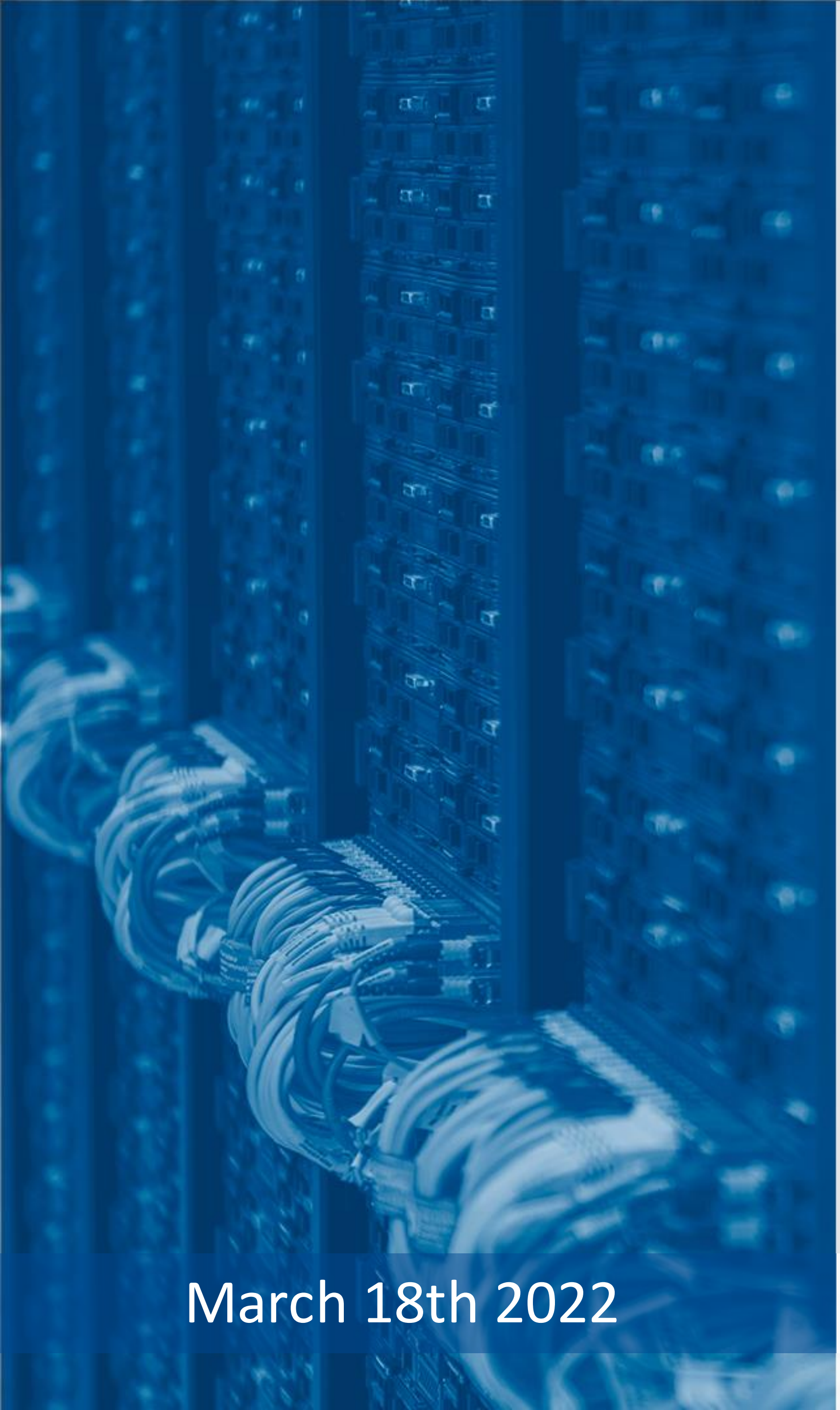


Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Adding support for RISC-V "V" vector extension in LLVM

Roger Ferrer Ibáñez
roger.ferrer@bsc.es

March 18th 2022

OSHEAN - Open-Source Hardware European Alliances and iNitiatives

RISC-V Vector Extension (RVV) in one sentence

Let’s try to summarise the whole RVV extension

Flexible Vector ISA with **vector length** and **masking support**

Implementing it in LLVM poses several challenges ⚠

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Overview

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Flexible

- The whole ISA provides a relatively large number of instructions (180)
  - Mirrors most scalar operations into vector operations
  - Several *forms* per instruction (vector-vector, vector-scalar, vector-immediate)
  - Specific vector instructions for masks, for “shuffles”, etc.
- The ISA supports many element types (SEW) and register grouping
  - Rather regular and orthogonal (not too many *quirks*)
- Lots of C/C++ intrinsics
  - >40,000 at the moment, yikes!

- **Lots of cases to support which bloats the compiler** ⚠

Barcelona
Supercomputing
Center
Centro Nacional de Supercomputación

Vector length

- Most Vector/SIMD ISAs assume that one always wants to operate all the elements/lanes of a vector register
  - Compiler intermediate representations (IR) have great support for this case
- RVV (re)introduces the concept of vector length (VL)
  - first VL elements of the vector operands participate in the operation
  - setting the VL is linked to the vector element type
  - this is not an explicit operand of the instructions

- **Compilers do not really like implicit state** ⚠

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Masking support

- Masking (or predication) is more general than the vector length idea
    - only enabled elements participate in the operation
- **LLVM IR does not have great support for this** (yet) ⚠
    - Currently only load/store/gather/scatter have masked versions
- At the semantical level this is unnecessary for most instructions
- At the machine level, masking (and vector length) are relevant for performance
    - e.g. avoids unnecessary exceptions or slow paths in FP

More in detail

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Flexibility

- RVV provides 32 vector registers (v0, …, v31)
  - They are in their own register class VR
- RVV provides vector groups, made up of vector registers (LMUL > 1)
  - We currently represent the vector groups as (super)register classes
  - LMUL is the length multiplier and enables register grouping to align the number of elements
- LLVM has machinery to determine the conflicts between registers to ensure that code generation is correct
- Unfortunately, LLVM Machine IR cannot generalise an instruction over different register classes, so we need one Machine IR instruction per register class
  - Despite them being mapped to the same instruction

Vector Length

- Vector length is not an explicit operand of the vector instructions
  - Part of the RVV state, must be set using vsetvl instructions
- It is convenient to model the vector length as an explicit operand of the vector instruction
  - Makes the instruction not to have a hidden side-effect which makes easier to reason with it (e.g., during scheduling)
- A later pass
  - inserts the needed `vsetvl` instructions
  - nullifies the explicit vector length operand
  - adds the register VL as an implicit operand
- This gives us correct and (reasonably) efficient code! ☺

Example

- Consider `vadd.vv vdest, vsrc1, vsrc2`
  - we must have one Machine IR instruction per register class
  - we need enough information so `vsetvl` (which needs LMUL and the SEW) can be inserted
  - SEW is an additional operand, along with the explicit vector length
- So we end with
  - `PseudoVADD_VV_M1`
  - `PseudoVADD_VV_M2`
  - `PseudoVADD_VV_M4`
  - `PseudoVADD_VV_M8`
  - `PseudoVADD_VV_MF2` (LMUL=1/2)
  - `PseudoVADD_VV_MF4` (LMUL=1/4)
  - `PseudoVADD_VV_MF8` (LMUL=1/8)
- Plus tail undisturbed and masked versions of the above → 21 internal pseudo instructions!

Example

Right after Instruction Selection

```
%4:vr = PseudoVADD_VV_M1 %0:vr, %1:vr, %3:gprnox0, 5
```

After `vsetvli` insertion

```
dead $x0 = PseudoVSETVLI %3:gprnox0, 80, implicit-def $vl, implicit-def $vtype
%4:vr = PseudoVADD_VV_M1 %0:vr, %1:vr, $noreg, 5, implicit $vl, implicit $vtype
```

Masking support

- Vector Predication is currently used to address this!
  - https://llvm.org/docs/Proposals/VectorPredication.html
- This fits well our code generation needs (but other SIMD ISAs as well!)
  - Vector Predication will be used by the vectorizer

```
define <vscale x 2 x i32> @vadd_vv_nxv2i32(<vscale x 2 x i32> %va,
                                           <vscale x 2 x i32> %b,
                                           <vscale x 2 x i1> %mask,
                                           i32 zeroext %evl) {
  %v = call <vscale x 2 x i32> @llvm.vp.add.nxv2i32(<vscale x 2 x i32> %va,
                                                    <vscale x 2 x i32> %b,
                                                    <vscale x 2 x i1> %mask,
                                                    i32 %evl)
  ret <vscale x 2 x i32> %v
}
```

$\Rightarrow$

```
vadd_vv_nxv2i32:
        vsetvli zero, a0, e32, m1, ta, mu
        vadd.vv v8, v8, v9, v0.t
        ret
```

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

European Processor Initiative

A glimpse to the future

- As part of the EPI project, we extended the LoopVectorizer to use Vector Predication
  - **Note**: this is **not** the only way to vectorise this code!

- Under consideration for upstreaming to LLVM

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

⇒

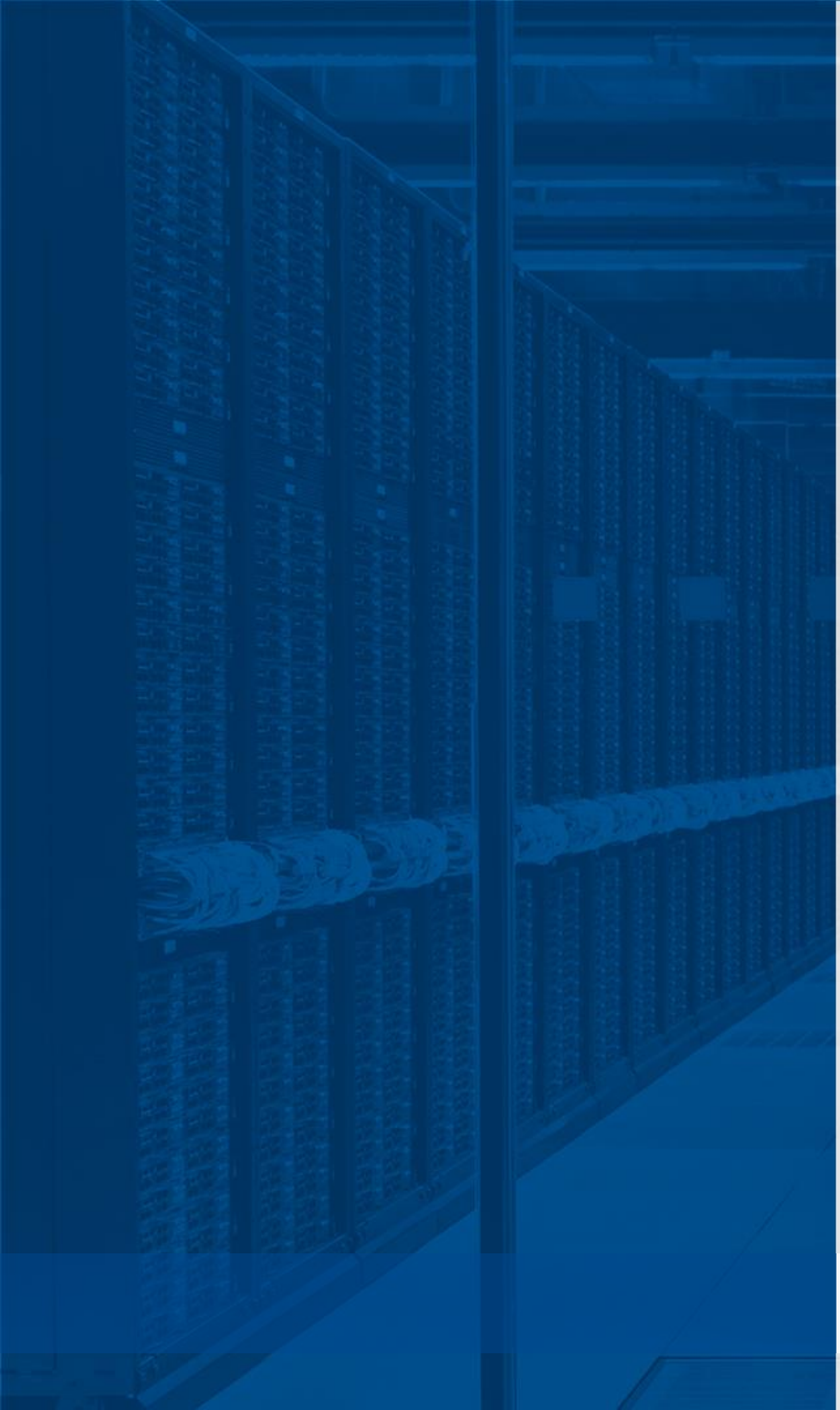
```
add_ref:
        blez    a0, .LBB0_3
        li      a4, 0
        slli    a0, a0, 32
        srli    a6, a0, 32
.LBB0_2:
        slli    a5, a4, 2
        add     a7, a2, a5
        sub     a0, a6, a4
        vsetvli t0, a0, e32, m1, ta, mu
        vle32.v v8, (a7)
        add     a0, a3, a5
        vle32.v v9, (a0)
        vadd.vv v8, v9, v8
        add     a0, a1, a5
        add     a4, a4, t0
        vse32.v v8, (a0)
        bne     a4, a6, .LBB0_2
.LBB0_3:
        ret
```

Play with the vectorizer here!
http://tiny.cc/epi-vector

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

In summary

- RVV is very **flexible**.
  - Great for the developer but implies <u>complexity</u> in the compiler engineering
- **Vector length** (and the vector type) is a big piece of implicit state in RVV
  - Creative approach that retains the desired freedom (e.g., scheduling) during code generation
- **Masking** is still a pending task in LLVM IR
  - Vector Predication is very promising in this space
  - Not only RISC-V benefits from it: <u>other mainstream SIMD ISAs should be using it</u>
  - We know how to teach LLVM to optimise Vector Predication at the same level of the existing IR but this will take time



Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Thank you!

This work has been done as part of the European Processor Initiative project.

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.

roger.ferrer@bsc.es