Background

• RISC-V is a new open-source ISA rapidly gaining momentum
  • Definition controlled by the RISC-V Foundation
  • No license fee to implement a processor using RISC-V
  • Over 200 companies have joined the foundation

• Very simple and clean ISA, with focus on extensibility
  • Supports RISC-V foundation sponsored extensions
  • As well as your proprietary “secret sauce” extensions

• There's a backend in LLVM
RISC-V Vector Extension (RVV)

• Simple, high performance, high efficiency vector processing
• Scale up & down to large & small cores
• Also base for further domain-specific extensions
• https://github.com/riscv/riscv-v-spec/
• Status: WIP but stable draft, building SW+HW and evaluating
Feature Highlight Reel

• Programmability: lots of support for vectorization
• Mixed-width computations, widening operations
• Fixed-point and f16
• Precise exceptions (with caveats for embedded platforms)
• Base for further specialized extensions, e.g. for matrix math, complex numbers, DSP, ML, graphics, ...
• Wide variety of microarchitecture styles supported, yet portable code
  • Yes, you can build SIMD
  • Yes, you can also build temporal Vectors (Cray anyone?)
Support for Vectorization

• Strip-mined loops – no remainder handling needed
• Masking on (almost) every vector instruction
• Strided loads and stores, scatters, gathers
• Reduction instructions (sum, min/max, and/or, ...)
• Orthogonal set of vector operations, parity with scalar ISA
• fault-only-first loads for loops with data dependent exits
Register State: 32 registers of VLEN bits

- 32 register names: v0 through v31
- Each register is VLEN-bits wide
  - VLEN is chosen by implementation, must be power of 2
    - See spec for additional restrictions in relation to ELEN and SLEN
- Some control registers
  - VL = active vector length
  - SEW = standard element width, hosted in vsew[2:0]
  - LMUL = grouping multiplier
SEW determines number of elements per vector

• SEW = Standard Element Width
• Dynamically settable through ‘vsew[2:0]’
• Each vector register viewed as VLEN/SEW elements, each SEW-bits wide
• Polymorphic instruction
  • vadd can be an i8/i16/i32/… add depending on SEW
• Set up along with VL (vsetvlq t0, a0, e32)

Example: VLEN=256b, vsew=’010, SEW=32b, elements = VLEN/SEW = 8
vfadd.vv v0, v1, v2

for (i = 0; i < VL; ++i)
    v0[i] = v1[i] + v2[i];
v0[VL..VLMAX] = 0;

• Lanes past VL don’t trap, raise exceptions, access memory, etc.
Register Grouping: LMUL

- Groups registers to form “longer vector”
  - Reduces number of valid register names
- Number of registers in each group is LMUL
  - LMUL can be 1, 2, 4, 8
- Example: when LMUL=2
  - vadd v2, v4, v6 really means \((v2,v3) := (v4,v5) + (v6,v7)\)
- Also used for widening operators (32b x 32b → 64b result)
- Like SEW, set with VL \((vsetvli t0, a0, e32, m4)\)
Strip-mining

Increase each array element (length in a0, pointer in a1) by the same amount (a2)

loop:

\[
\begin{align*}
\text{vsetvli} & \ t0, \ a0, \ e32 \quad \# \ t0 = VL = \max(a0, VLMAX) \\
\text{vlw.v} & \ v0, \ (a1) \\
\text{vadd.vs} & \ v2, \ v0, \ a2 \\
\text{vsw.v} & \ v2, \ (a1) \\
\text{sub} & \ a0, \ a0, \ t0 \\
... & ; \text{ advance ptr by VL elements} \\
\text{bnez} & \ a0, \ \text{loop}
\end{align*}
\]

Sets SEW
Polymorphic!
Strip-mining

Increase each array element (length in a0, pointer in a1) by the same amount (a2)

loop:

\[
\begin{align*}
&\text{vsetvli } t0, a0, e32 \quad \# t0 = VL = \max(a0, VLMAX) \\
&\text{vlw.v } v0, (a1) \quad \text{v0, (a1)} \\
&\text{vadd.vs } v2, v0, a2 \quad \text{v2, v0, a2} \\
&\text{vsw.v } v2, (a1) \quad \text{v2, (a1)} \\
&\text{sub } a0, a0, t0 \\
&\ldots \quad ; \text{ advance ptr by VL elements} \\
&\text{bnez } a0, \text{ loop}
\end{align*}
\]
Mixed-precision Calculations

• Usually, biggest data type limits vector length
  • Unless you want lots of shuffles
Mixed-precision Calculations

• Usually, biggest data type limits vector length
• Alternative with RISC-V V:
  • pack 16b elements tightly
  • 32b elements span two registers
  • Switch LMUL to work with both
• No need to shuffle in registers
• Tradeoff: not a win on all uarchs
LLVM Support

• Out-of-tree patches @ [https://github.com/rkruppe/rvv-llvm](https://github.com/rkruppe/rvv-llvm)
• Want to start upstreaming when spec frozen
• Mostly MC and CodeGen work so far
• Very interested in autovectorization, but needs groundwork
• Status: can manually write vector code in IR and CodeGen it
Strip-mined Loop in IR

loop:

%n = phi ...
%ptr = phi ...
%vl = call i32 @llvm.riscv.vsetvl(i32 %n)
%v1 = call <scalable 1 x i32> @llvm.riscv.vlw(%ptr, i32 %vl)
%v2 = call ... @llvm.riscv.vadd.sv1i32(%v1, %splat, i32 %vl)
call void @llvm.riscv.vsw(%ptr, %v2, i32 %vl)
%n.new = sub i32 %n, %vl
%ptr.new = ...
%done = icmp eq i32 %n.new, 0
IR Vector Type

• `<scalable k x T>` type proposed by Arm for their Scalable Vector Extension (SVE)

• Lots of common ground (even more than last year!)
  • vector register size unknown at compile time, constant at runtime
  • but: known constant factor, e.g., VLEN multiple of 64b

• Want to use whatever gets accepted upstream for SVE

• References
  • [https://llvm.org/D32530](https://llvm.org/D32530)
IR Intrinsics

• `@llvm.riscv.vadd.sv1i32(op1, op2, i32 vl, mask)`
  • Active vector length is just another argument
  • Masking as part of every operation, not external select
• Essentially like Simon Moll’s Vector Predication proposal
• Note: no mention of SEW/LMUL
• References
  • [https://llvm.org/D57504](https://llvm.org/D57504)
  • Simon Moll’s talk earlier today
CodeGen Perspective

• VL is just another (allocatable) integer register
  • Copies to/from GPR supported
  • Input to most vector instructions, output of vsetvl
  • Need to figure out how to “spill” it

• vtype is reserved physical register
  • Implicitly used by everything, defined by vsetvl
  • Managed by backend, no IR representation
  • SEW, LMUL dictated by vector types used in IR
Instruction Selection

• Straightforward mapping of intrinsics to (pseudo-)instructions
  • Hardware instructions are polymorphic, but compiler needs static info
  • Pseudos for each element width and LMUL
  • Different LMUL also means different register classes (e.g., pairs for LMUL=2)
    • e.g. <scalable 4 x i32> add \rightarrow vadd_e32_m4

• VL modelled as normal integer value

• Don’t set up configuration (SEW, LMUL) yet
After ISel

• Place instruction that set up necessary SEW and LMUL
  • Fold into existing vsetvl’s where possible
• MIR optimizations, e.g., removing redundant vl ↔ GPR copies
• Copying vector registers is a mess
  • Need to copy whole register (vl = MAX) in general
  • Should usually prove that elements past current vl won’t be read
  • Not yet sure how to best achieve this
Next Steps needed

• Fill in more backend features
• Automatic vectorization (cf. SVE)
• Software ecosystem: vendor-tuned libraries
• Evaluate & adjust ISA
• Implementations will start popping out soon

• Please come help!
Conclusion

• RISC-V has a great, flexible vector extension
  • https://github.com/riscv/riscv-v-spec/

• LLVM backend for it already started
  • https://github.com/rkruppe/rvv-llvm

• Lots of industrial activity around it (even if you don’t see it)