Vector Extension Proposal
v0.2

Krste Asanović
UC Berkeley / SiFive Inc.
5th RISC-V Workshop, Berkeley, CA
November 30, 2016
Goals for RISC-V Standard V Extension

- Efficient and scalable to all reasonable design points
  - Low-cost microcontroller or high-performance supercomputer
  - In-order, decoupled, or out-of-order microarchitectures
  - Integer, fixed-point, and/or floating-point data types
- Good compiler target
- Support both implicit auto-vectorization (OpenMP) and explicit SPMD (OpenCL) programming models
- Work with virtualization layers
- Fit into standard fixed 32-bit encoding space
- Be base for future vector++ extensions
RISC-V Vector Extension Update Summary

- Last presentation v0.1, 2\textsuperscript{nd} RISC-V Workshop, June 2015
- Progress slow last year due to other higher priority parts of standard, but time to work on this now
- Working group forming now (I’m chair)
- Goal is to ratify 12 months from now at 7\textsuperscript{th} workshop
V Key Features

- Cray-style vectors
  - “The right way” to exploit SIMD parallelism
- Implementation-dependent vector length
  - Same binary runs with different hardware vector lengths
  - Support wide range of implementations, microcontroller to supercomputer
- Reconfigurable vector register file
- Mixed-precision support
V Extension State

Standard RISC-V scalar x and f registers

<table>
<thead>
<tr>
<th>x31</th>
<th>f31</th>
</tr>
</thead>
<tbody>
<tr>
<td>x1</td>
<td>f1</td>
</tr>
<tr>
<td>x0</td>
<td>f0</td>
</tr>
</tbody>
</table>

Up to 32 vector data registers, v0-v31, of at least 4 elements each, with variable bits/element (8,16,32,64,128)

<table>
<thead>
<tr>
<th>v31[0]</th>
<th>v31[1]</th>
<th>v31[MVL-1]</th>
</tr>
</thead>
<tbody>
<tr>
<td>v1[0]</td>
<td>v1[1]</td>
<td>v1[MVL-1]</td>
</tr>
<tr>
<td>v0[0]</td>
<td>v0[1]</td>
<td>v0[MVL-1]</td>
</tr>
</tbody>
</table>

Vector configuration CSRs

- vcmaxw
- vctype
- vcnpred

Vector length CSR

- vl

Vector fixed-point rounding mode and saturation flag CSRs

- vxrm
- vxsat

Up to 8 vector predicate registers, with 1 bit per element

<table>
<thead>
<tr>
<th>p7[0]</th>
<th>p7[1]</th>
<th>p7[MVL-1]</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1[0]</td>
<td>p1[1]</td>
<td>p1[MVL-1]</td>
</tr>
<tr>
<td>p0[0]</td>
<td>p0[1]</td>
<td>p0[MVL-1]</td>
</tr>
</tbody>
</table>
Each vector data register is configured with a width and type, or disabled.

- Configurable number of predicate registers (0-8)
- Maximum vector length (MVL) function of configuration, physical register storage, and microarchitecture
### Supported Fixed-Point Widths

<table>
<thead>
<tr>
<th>ISA</th>
<th>Supported Widths</th>
</tr>
</thead>
<tbody>
<tr>
<td>RV32I</td>
<td>X8, X16, X32</td>
</tr>
<tr>
<td>RV64I</td>
<td>X8, X16, X32, X64</td>
</tr>
<tr>
<td>RV128I</td>
<td>X8, X16, X32, X64, X128</td>
</tr>
</tbody>
</table>

### Supported Floating-Point Widths

<table>
<thead>
<tr>
<th>Prefix</th>
<th>Supported Widths</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>F16, F32</td>
</tr>
<tr>
<td>FD</td>
<td>F16, F32, F64</td>
</tr>
<tr>
<td>FDQ</td>
<td>F16, F32, F64, F128</td>
</tr>
</tbody>
</table>

Adding V extension to scalar floating-point extension adds scalar half-precision (IEEE 16-bit FP) instructions.
Each vector data register has a 4-bit field in the \( \text{vcmaxw} \) CSR that describes the maximum width of elements in that vector.

- Total of 32x4b=128 bits of width state held in one (RV128), two (RV64) or four (RV32) CSRs.
- Any writes to \( \text{vcmaxw} \) initializes all vector unit state.
Each data register has current type encoded in 4-bit field in vctype register

- Writes to vcmaxw set both vcmaxw and vctype, vcmaxw retains only width not type
- Writes to vctype only zeros associated vector register

<table>
<thead>
<tr>
<th>Type</th>
<th>vctype encoding</th>
<th>vcmaxw equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>Disabled</td>
<td>0000</td>
<td>0000</td>
</tr>
<tr>
<td>F16</td>
<td>0001</td>
<td>1001</td>
</tr>
<tr>
<td>F32</td>
<td>0010</td>
<td>1010</td>
</tr>
<tr>
<td>F64</td>
<td>0011</td>
<td>1011</td>
</tr>
<tr>
<td>F128</td>
<td>0100</td>
<td>1100</td>
</tr>
<tr>
<td>X8</td>
<td>1000</td>
<td>1000</td>
</tr>
<tr>
<td>X16</td>
<td>1001</td>
<td>1001</td>
</tr>
<tr>
<td>X32</td>
<td>1010</td>
<td>1010</td>
</tr>
<tr>
<td>X64</td>
<td>1011</td>
<td>1011</td>
</tr>
<tr>
<td>X128</td>
<td>1100</td>
<td>1100</td>
</tr>
</tbody>
</table>
Vector Predicate Configuration

- The `vcnpred` CSR holds number of predicate registers (0-8)
- Writes to `vcnpred` initializes all vector unit state
Faster configuration

- Setting all configuration bits directly via `vcmaxw` requires creating/loading long immediates and writing possibly multiple CSRs (RV32/64)
- A `vcfgd` CSR alias is defined for faster writes of common vector data configurations
- One 5-bit field per supported type, set to highest vector register number with that type or zero for none

<table>
<thead>
<tr>
<th></th>
<th>F64</th>
<th>F32</th>
<th>F16</th>
<th>X32</th>
<th>X16</th>
<th>X8</th>
</tr>
</thead>
<tbody>
<tr>
<td>RV32</td>
<td>0</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>RV64</td>
<td>0</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>24</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>RV128</td>
<td>0</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>83</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
</tbody>
</table>
### Fast configuration example

<table>
<thead>
<tr>
<th>RV32</th>
<th>RV64</th>
<th>RV128</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>F64</td>
<td>F128</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>X64</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>F32</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>F16</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>X32</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>X16</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>X8</td>
</tr>
<tr>
<td>83</td>
<td>5</td>
<td>F128</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>F32</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>F16</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>X32</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>X16</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>X8</td>
</tr>
</tbody>
</table>

**Table: Vector registers**

<table>
<thead>
<tr>
<th>Vector registers</th>
<th>vcmaxw</th>
<th>vctype</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>v31–v19</td>
<td>0000</td>
<td>0000</td>
<td>Disabled</td>
</tr>
<tr>
<td>v18–v13</td>
<td>1011</td>
<td>0011</td>
<td>F64</td>
</tr>
<tr>
<td>v12–v2</td>
<td>1010</td>
<td>0010</td>
<td>F32</td>
</tr>
<tr>
<td>v1–v0</td>
<td>1010</td>
<td>1010</td>
<td>X32</td>
</tr>
</tbody>
</table>
Maximum Vector Length

- Setting `vcmaxw` and `vcnpred` determines current maximum vector length (MVL)
  - `vctype` does not affect MVL
- Any change to `vcmaxw` or `vcnpred` initializes all vector unit state
  - Must not rely on state inbetween reconfigurations
  - Gives flexibility to implementations
  - Avoid security holes from leaking state
- `CSRRW / CSRRWI` instructions to change `vcmaxw` / `vcnpred` return resulting MVL
  - This is different than plain `CSRRW` that returns old value
  - Most code will not use MVL directly
Set Vector Length

- Active vector length held in `vl` CSR, a WARL register holding values between 0 and MVL inclusive.
- Any configuration changes initialize `vl` to MVL.
- Usually `vl` modified with `setvl` instruction encoded as CSRRW/CSRRWI instruction
- Source argument to `setvl` is application vector length (AVL), returns value placed in `vl`

<table>
<thead>
<tr>
<th>AVL Value</th>
<th><code>vl</code> setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>$AVL \geq 2 \text{MVL}$</td>
<td>MVL</td>
</tr>
<tr>
<td>$2 \text{MVL} &gt; AVL &gt; \text{MVL}$</td>
<td>$\lfloor AVL/2 \rfloor$</td>
</tr>
<tr>
<td>$\text{MVL} \geq AVL$</td>
<td>AVL</td>
</tr>
</tbody>
</table>
32-bit integer vector-vector add example

```
vcfgd 2*X32  # Only need two vector registers
stripmine:
vsetvl t0, a0  # a0 holds vector length
vld v0, a1    # Get first vector
vld v1, a2    # Get second vector
vadd v1, v0   # Add vectors
vst v1, a3    # Store result vector
sll t1,t0,2   # Multiply count by 4 to get byte
add a1, t1    # Bump pointers
add a2, t1
add a3, t1
sub a0, t0    # Subtract number done
bnez a0, stripmine  # Any more?
vuncfg  # Turn off vector unit by zeroing config
```
16-bit integer vector-vector add example

vcfgd 2*X16  # Only need two vector registers

stripmine:
  vsetvl t0, a0  # a0 holds vector length
  vld v0, a1    # Get first vector
  vld v1, a2    # Get second vector
  vadd v1, v0   # Add vectors
  vst v1, a3    # Store result vector
  sll t1,t0,1   # Multiply count by 2 to get byte
  add a1, t1    # Bump pointers
  add a2, t1
  add a3, t1
  sub a0, t0    # Subtract number done
  bnez a0, stripmine  # Any more?
  vuncfg  # Turn off vector unit by zeroing config
16-bit + 32-bit vector add example

vcfgd 1\times 32|1\times 16

stripmine:
  vsetvl t0, a0  # a0 holds vector length
  vld v0, a1    # Get first 16-bit vector
  vld v1, a2    # Get second 32-bit vector
  vadd v1, v0   # Add vectors
  vst v1, a3    # Store result vector
  sll t1,t0,1   # Multiply count by 2 to get byte
  sll t2,t0,2   # Multiply count by 4 to get byte
  add a1, t1    # Bump pointers
  add a2, t2
  add a3, t2
  sub a0, t0    # Subtract number done
  bnez a0, stripmine # Any more?
  vuncfg # Turn off vector unit by zeroing config
Vector Length Portability

- Same binary code works regardless of:
  - Number of physical register bits
  - Number of physical lanes
  - Mixed-precision packing strategy

- Architecture guarantees minimum vector length of four regardless of configuration to avoid stripmine overhead for short vectors
  - E.g., if use 32 * 64-bit vector registers,
  - need 128 * 8-byte physical element registers
  - 1KB SRAM
Polymorphic Instruction Encoding

- Single signed integer ADD opcode works on different size inputs and outputs
  - Size of inputs and outputs inherent in register number
  - Sign-extend smaller input
  - Modulo arithmetic on overflow to destination
  - Restrict supported combinations to simplify hardware

- Integer, Fixed-point, Floating-point arithmetic
Vector Loads and Stores

Addressing modes:
- Unit-stride (scalar base)
- Constant stride (scalar base, scalar stride)
- Indexed (scalar base, vector offset)

Types inherent in destination register number (for integers, signed/unsigned determined at use)

Support vector AMOs:
- E.g, Vector fetch-and-add
Vector Predication

- Up to eight vector predicate registers p0-p7, one bit per element
- Logical operations between predicate registers
- All vector instructions are predicated under p0
  - Implicit predicate due to encoding constraints
- Instruction to swap two predicate registers
  - Reduce overhead of scheduling complex control flow
  - Can implement just in rename table if OoO core
- Popcount instruction returns number of active bits in predicate register to scalar integer register
  - Used for divergent control flow optimizations
- Other cross-element flag operations to support complex loop optimizations
- Support for software vector length speculation
Vector Predication and Vector Register Renaming

Previous approaches in vector archs:

1) Destination has old value if predicate false
   – Simpler spec, better for in-order/no renaming
   – Have to copy old value to new destination with renaming

2) Destination has zero value if predicate false
   – Better for out-of-order with renaming
   – Need additional merge(s) to rebuild complete vector

3) Destination has undefined value if predicate false
   – More complex code, better for out-of-order with renaming
   – Need additional merge(s) to rebuild complete vector
   – Messy definition

- We’re choosing 1), as simpler and safer.
- Use microarchitectural tricks for OoO machines to reduce amount of data transfer.
Vector Function Calls

```c
for (i=0; i<N; i++)
    x[i] = exp(y[i]/z[i]);
```

- In auto-vectorized code, want to make vector calls to function library with separate vector calling convention
  - Args in vector registers
  - Active elements communicated by vector length and vp0
- Need to abstract callee register usage from caller
- Caller has to allocate registers for callee to use
- Set \texttt{vcmaw} to largest value, then callee can change type with \texttt{vctype}
- Vector runtime can optimize calling convention within vector runtime library
OpenCL / CUDA / SPMD programming

- Not a great programming model, should move community back to autovectorization/autoparallelization, but needed for compatibility
- Predication used to handle divergent control flow
  - See Yunsup’s thesis
- Configuration must be set at kernel launch to maximum width used anywhere in call tree
- Need general vector function call capability with standard callee/caller save protocol
OS Support

- Restartable page faults via microcode state dump, opaque to OS
  - Similar to DEC Vector Vax implementation
  - If implementation has precise traps, can skip
- Privileged specification describes XS sstatus field used to encode coprocessor status (Off, Initial, Clean, Dirty) to reduce context save/restore overhead.
Questions?