Vector Extension Update

Krste Asanovic
krste@berkeley.edu

ADEPT Retreat, Chaminade, Santa Cruz
May 29, 2019
Vector Extension Prehistory

- RISC-V (2010-) originally designed to explore new accelerators based on top of vector engine (ESP)
- Hwacha was primary research vehicle to develop vector ISA and microarchitecture ideas (2012-)
  - Hwacha taped out multiple times at UCB (v4.5 on EagleX)
- Hwacha was an explicitly decoupled vector-fetch accelerator with own vector instruction stream
- RISC-V “V” extension has more traditional single instruction stream, à la original Cray vectors
Goals for RISC-V Standard V Extension

- Efficient and scalable to all reasonable design points
  - Low-cost or high-performance
  - 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
Vector Extension History

- First proposal (v0.1) presented June 2015 workshop
- Many iterations until recent v0.7 stable draft in Jan 2019
- By far the largest RISC-V extension (larger than sum of everything previously ratified)
- v0.7 now being targeted by community with implementation work and software development
  - [https://github.com/riscv/riscv-v-spec](https://github.com/riscv/riscv-v-spec)
- Stable spec version v0.7.1, matching software release today!
- Plan to spend ~year on trying extension before freezing and attempting to ratify
RISC-V Foundation Vector Extension Overview

32 vector registers

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

Maximum vector length (VLMAX) depends on implementation, number of vector registers used, and type of each element.

- Unit-stride, strided, scatter-gather, structure load/store instructions
- Rich set of integer, fixed-point, and floating-point instructions
- Vector-vector, vector-scalar, and vector-immediate instructions
- Multiple vector registers can be combined to form longer vectors to reduce instruction bandwidth or support mixed-precision operations (e.g., 16b*16b->32b multiply-accumulate)
- Designed for extension with custom datatypes and widths

Vector CSRs

- vtype
- vl
- vstart
- fcsr (vxrm/vxsat)

Vtype sets width of element in each vector register (e.g., 16-bit, 32-bit, ...)

Vector length CSR sets number of elements active in each instruction

Resumption element after trap

Fixed-point rounding mode and saturation flag fields in FP CSR
Vector Unit Implementation - Dependent Parameters

- ELEN: Size of largest element in bits
- VLEN: Number of bits in each vector register
  - VLEN >= ELEN
- SLEN: Striping distance in bits
  - VLEN >= SLEN >= ELEN

- Vector ISA designed to allow same binary code to work across variations in VLEN and SLEN
### Some Microarchitecture Design Points

<table>
<thead>
<tr>
<th>Name</th>
<th>Issue Policy</th>
<th>Issue Width</th>
<th>VLEN (bits)</th>
<th>Datapath (bits)</th>
<th>VLEN/Datapath (beats)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Smallest</td>
<td>InO</td>
<td>1</td>
<td>32</td>
<td>32</td>
<td>1</td>
</tr>
<tr>
<td>Simple</td>
<td>InO</td>
<td>1</td>
<td>512</td>
<td>128</td>
<td>4</td>
</tr>
<tr>
<td>InO-Spatial</td>
<td>InO</td>
<td>2</td>
<td>128</td>
<td>128</td>
<td>1</td>
</tr>
<tr>
<td>OoO-Spatial</td>
<td>OoO</td>
<td>2-3</td>
<td>128</td>
<td>128</td>
<td>1</td>
</tr>
<tr>
<td>OoO-Temporal</td>
<td>OoO</td>
<td>2-3</td>
<td>512</td>
<td>128</td>
<td>4</td>
</tr>
<tr>
<td>OoO-Server</td>
<td>OoO</td>
<td>3-6</td>
<td>2048</td>
<td>512</td>
<td>4</td>
</tr>
<tr>
<td>OoO-HPC</td>
<td>OoO</td>
<td>3-6</td>
<td>16384</td>
<td>2048</td>
<td>8</td>
</tr>
</tbody>
</table>
Design Challenges: Opcode Space

- Community wanted to stay with 32-bit instruction encoding
  - Low-end embedded systems have 32-bit instruction fetch
  - Harder to support mixed-length instruction streams, 16, 32, 48, & 64 bits
  - Static code size matters on embedded platforms
- But wanted vast array of datatypes and custom datatypes, and large set of operations
Opcode Solution: \texttt{vtype} register

- Added a control register to hold some information about current setting of vector unit
- \texttt{vtype} fields (total additional 6-7 state bits)
  - \texttt{vsew}: standard element width (SEW=8,16,32,...,1024)
  - \texttt{vlmul}: vector length multiplier (LMUL=1,2,4,8)
  - \texttt{vediv}: vector element divider (EDIV=1,2,4,8)
- Encoding only occupies 1.5 major opcodes
- Full 64-bit instruction encoding also planned
  - Can view current 32-bit encoding as compressed form of full encoding
Vector Length control

- Current maximum vector length is register length in bits divided by current element width setting:
  \[ \text{VLMAX} = \frac{\text{VLEN}}{\text{SEW}} \]

  - E.g., VLEN = 512b, SEW=32b, => VLMAX = 16
- Current active vector length set by \texttt{vl} register
  \[ 0 \leq \texttt{vl} \leq \text{VLMAX} \]
vsetvli/vsetvl instructions

- **vsetvli** instruction sets both vector configuration and vector length:
  - **vsetvli** \(rd, rs1, imm\)  
  
  Immediate encodes vtype:  
  \(<\text{vsew, vlmul, vediv}>\)

  Application vector length (AVL)

  Returns setting of vector length in scalar register

- Vector length \(vl\) set to \(\text{min}(AVL, VLMAX)\)
# Simple memcpy example

```c
# void*memcpy(void*dest,const void*src, size_t n)
# a0=dest, a1=src, a2=n
memcpy:
    mv a3, a0          # Copy destination
loop:
    vsetvli t0, a2, e8 # Vectors of 8b
    vlb.v v0, (a1)    # Load bytes
    add a1, a1, t0    # Bump pointer
    sub a2, a2, t0    # Decrement count
    vsb.v v0, (a3)    # Store bytes
    add a3, a3, t0    # Bump pointer
    bnez a2, loop     # Any more?
    ret                # Return
```
VLEN=256b

“Obvious” mapping in case of LMUL=1
Challenge: Mixed-Precision Arithmetic

- To preserve accuracy with fewest bits, use mixed-precision, so some operations have 2*SEW operands or results (also 4*SEW)
- Problem 1: To keep same VLMAX, need more bits in registers
- Problem 2: Operands should “line up” to avoid datapath wiring
LMUL, vector length multiplier

- **vlmul** field in vtype register sets LMUL, which is how many vector registers in a “group” (LMUL=1,2,4,8)
- Vector instructions execute all elements in a vector register group
- Now: VLMAX = LMUL*VLEN/SEW
<table>
<thead>
<tr>
<th>SEW</th>
<th>LMUL</th>
<th>VLEN</th>
<th>SLEN</th>
<th>SLEN = wiring span</th>
</tr>
</thead>
<tbody>
<tr>
<td>8b</td>
<td>1</td>
<td>256</td>
<td>128</td>
<td></td>
</tr>
<tr>
<td>16b</td>
<td>2</td>
<td>256</td>
<td>128</td>
<td></td>
</tr>
<tr>
<td>32b</td>
<td>4</td>
<td>256</td>
<td>128</td>
<td></td>
</tr>
<tr>
<td>64b</td>
<td>8</td>
<td>256</td>
<td>128</td>
<td></td>
</tr>
</tbody>
</table>
Instruction Types

- Vector arithmetic instructions:
  - Integer and floating-point operations of SEW width
  - Vector-vector
  - Vector-scalar (with x or f registers)
  - Vector-immediate (for integer ops)
  - Widening SEW*SEW → 2*SEW or 4*SEW
  - Narrowing 2*SEW → SEW

- Vector load and store instructions:
  - Unit-stride, strided, indexed (scatter/gather)
  - Either fixed 8b, 16b, 32b, or variable SEW sized elements
  - Plus segments (load 1,2,...,8 vector registers)
vlseg3b.v v4, (x10)
# Load packed 8b RGB values into v4, v5, v6

x10

Also, available for strided and scatter/gather addressing modes
Predicated Execution

- In tight 32-bit encoding, only a single bit available for masking

<table>
<thead>
<tr>
<th>vm</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>vector result, only where v0[i].LSB = 1</td>
</tr>
<tr>
<td>1</td>
<td>unmasked</td>
</tr>
</tbody>
</table>

What happens to masked-off elements?
Balancing In-Order and Out-of-Order Designs

What to do with masked-off elements and tail elements, past current vector length vl?

Fundamental conflict:
- OoO renamed designs allocate new physical destination vector register, have to write all elements
- InO designs prefer to leave elements undisturbed
Solution: Zeroing versus Preserving?

- Did not want to allow implementation-dependent behavior

- Trade off microarch pain against software cost

- Destination elements past vector length are zeroed
  - InO/OoO machines can implement microarch tricks to not actually write all the zeros

- Masked-off elements inside vector length preserved
  - OoO machines have to copy old destination results, but ISA designed to have only destructive multiply-adds, so no increase in number of read ports
Precise Traps

- Supporting page faults and interrupts
- The unprivileged \texttt{vstart} register holds the first element to be processed by next vector instruction, and every vector instruction resets \texttt{vstart} to 0.
- On a trap, EPC points to faulting instruction and \texttt{vstart} points to element at which to continue execution; preceding elements \texttt{<vstart} unchanged
- User software should not try to use \texttt{vstart} explicitly
Divided Elements

- Vtype field `vediv` sets EDIV=1,2,4,8
- EDIV is number of ways that current SEW is divided into sub-elements
- Supports very small vector types, e.g., 8b divided into 8x1b sub-elements
- Some instruction behaviors change with EDIV
  - Reductions across element
  - Dot products across element
  - Register gather within element
Other Features

- Load-fault-on-first (for safely vectorizing while loops)
- Reduction operations
- Vector register permute instructions
- Register compress instructions
- Fixed-point rounding and saturation
Software Support

- Assembler/binutils for v0.7.1, open-sourced
- Spike ISA simulator for v0.7.1, open-sourced
- Imperas binary release of simulator with v0.7.1 extensions

- Still need to build libraries, compilers, other tooling
Questions?