Vector Extension Proposal

Colin Schmidt, Albert Ou, Yunsup Lee, Krste Asanović

krste@eecs.berkeley.edu

http://www.riscv.org

2nd RISC-V Workshop, Berkeley, CA
June 29, 2015
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

- Non-goal: Look like everyone’s Packed-SIMD
- Non-goal: Look like a GPU
Packed-SIMD versus Traditional Vectors

- 1950s’ Packed-SIMD
- 1970s’ Vectors

Ivan Sutherland is shown working with Sketchpad. The lightpen, shown below, and the keyboard function control are the precursors for the mouse and right and left mouse-clicks prevalent today. The Sketchpad system was the first graphical computer interface. It made it possible for a man and a computer to converse rapidly through the medium of line drawings. Heretofore, most interaction between man and computers had been slowed down by the need to reduce all communication to written statements that could be typed; in the past, we had been writing letters to, rather than conferring with, our computers. For many types of communication, such as describing the shape of a mechanical part or the connections of an electrical circuit, typed statements can prove cumbersome. The Sketchpad system, by eliminating typed statements in favor of line drawings, opened up a new area of man-machine communication. The currently used graphical user interface, or GUI, was based on Sketchpad.

Sketchpad a master's degree from Caltech, my engineering skill had never before been taken seriously; the memory remains vivid. Part of education is learning to take one's place in professional life.

Although enormous both in physical size and in capacity for its day, TX-2 operated like a personal computer. Users sat at the console and debugged code or did experiments. I think the bug was that the reset worked from the keyboard but not when one typed from paper tape. I went to my boss of the time, Jack Mitchell, and reported the bug. I was surprised to hear him ask me to propose a fix. I was even more surprised when my proposed fix, which involved half a dozen additional relays, was adopted and built into all Lincoln writers. Although I had case and one for lower case. There were about a dozen Lincoln writers in the Laboratory, each built into a small table with casters. The bug, as I recall it, had to do with a particular problem with changing case.

§ 1950s

§ 1970s
Packed-SIMD versus Traditional Vectors

[Diagram of packed-SIMD and traditional vector systems, illustrating the differences in structure and memory access.]
GPU versus Traditional Vector

- **GPU**
  - Discrete accelerator running in own memory space
    - More recently support sharing of physical, or pinned virtual, but still in different memory hierarchy and no page faults
  - Terrible at scalar code
  - Only effective on very large data-parallel tasks (>10,000s)
  - ISA/microarchitecture evolved from graphics shader needs, not general compute
  - Only seems efficient compared to out-of-order scalar core

- **Traditional vector**
  - Coprocessor running in same memory hierarchy as scalar
  - Tightly coupled to scalar core
  - Effective with loop counts of 2 or more
  - ISA evolved from general computing needs
  - Very efficient
Rest of Talk Outline

- Why traditional vectors are better than Packed-SIMD or GPUs
- What we’re proposing for V extension
Mostly automatic, possibly some restructuring from the application writer

```c
for (i=0; i<n; i++) {
    y[i] = a*x[i] + y[i];
}
```

**SAXPY**
a0: n, a1: a, a2: *x, a3: *y

vsplat4 4t2, a1
stripmine:
vlw4 4t0, a2
vlw4 4t1, a3
vfma4 4t3, 4t2, 4t0, 4t1
vsw4 4t3, a3
add a2, a2, 4<<2
add a3, a3, 4<<2
sub a0, a0, 4
bgte a0, 4, stripmine
...
handle edge cases

SAXPY on Packed-SIMD Architecture
SAXPY Function Arguments

**Packed-SIMD**

- $a_0$: $n$, $a_1$: $a$, $a_2$: $x$, $a_3$: $y$

  - `vsplat4 4t2, a1`
  - `stripmine:``
    - `vlw4 4t0, a2`
    - `vlw4 4t1, a3`
    - `vfma4 4t3, 4t2, 4t0, 4t1`
    - `vsw4 4t3, a3`
  - `add a2, a2, 4<<2`
  - `add a3, a3, 4<<2`
  - `sub a0, a0, 4`
  - `bgte a0, 4, stripmine`
  - `handle edge cases`
Convert Scalar to Vector

Packed-SIMD

\[ a0: n, a1: a, a2: *x, a3: *y \]

- **vsplat4 4t2, a1**
- **stripmine:**
  - vlw4 4t0, a2
  - vlw4 4t1, a3
  - vfma4 4t3, 4t2, 4t0, 4t1
  - vsw4 4t3, a3
- **add a2, a2, 4<<2**
- **add a3, a3, 4<<2**
- **sub a0, a0, 4**
- **bgte a0, 4, stripmine**
  - handle edge cases

Diagram:

- Inputs: 4t0, 4t1, 4t2
- Outputs: 4t3
- Operations: vsplat4, stripmine, vlw4, vfma4, vsw4, add, sub, bgte
- FMA

...
a0: n, a1: a, a2: \(*x, a3: *y

vsplat4 4t2, a1

stripmine:
vlw4 4t0, a2
vlw4 4t1, a3
vfma4 4t3, 4t2, 4t0, 4t1
vsw4 4t3, a3
add a2, a2, 4<<2
add a3, a3, 4<<2
sub a0, a0, 4
bgte a0, 4, stripmine

handle edge cases

Packed-SIMD
Load Y Vector

\[
\begin{align*}
a0 &: n, a1 &: a, a2 &: *x, a3 &: *y \\
\text{vsplat4 4t2, a1} \\
\text{stripmine:} \\
\text{vlw4 4t0, a2} \\
\text{vlw4 4t1, a3} \\
\text{vfma4 4t3, 4t2, 4t0, 4t1} \\
\text{vsw4 4t3, a3} \\
\text{add a2, a2, 4<<2} \\
\text{add a3, a3, 4<<2} \\
\text{sub a0, a0, 4} \\
\text{bgte a0, 4, stripmine} \\
\ldots \\
\text{handle edge cases}
\end{align*}
\]
Compute Multiply-Add Result

\[ \begin{align*}
a_0 &: n, \ a_1 &: a, \ a_2 &: \ast x, \ a_3 &: \ast y \\
v\text{splat}_4 \ 4t2, \ a_1 \\
\text{stripmine}: & \\
v\text{l}w_4 \ 4t0, \ a_2 \\
v\text{l}w_4 \ 4t1, \ a_3 \\
v\text{f}m\text{a}_4 \ 4t3, \ 4t2, \ 4t0, \ 4t1 \\
v\text{s}w_4 \ 4t3, \ a_3 \\
\text{add} \ a_2, \ a_2, \ 4\ll<2 \\
\text{add} \ a_3, \ a_3, \ 4\ll<2 \\
\text{sub} \ a_0, \ a_0, \ 4 \\
\text{bg}t\text{e} \ a_0, \ 4, \ \text{stripmine} \\
\ldots \\
\text{handle edge cases}
\end{align*} \]

Packed-SIMD
Vector Store of Results

Packed-SIMD

\[ a0: n, a1: a, a2: \,*x, a3: \,*y \]

- vsplat4 4t2, a1
- stripmine:
  - vlw4 4t0, a2
  - vlw4 4t1, a3
  - vfma4 4t3, 4t2, 4t0, 4t1
  - vsw4 4t3, a3
- add a2, a2, 4<<2
- add a3, a3, 4<<2
- sub a0, a0, 4
- bgte a0, 4, stripmine
  
  ... handle edge cases

\[ \text{FMA} \]
Increment Address Pointers

a0: n, a1: a, a2: *x, a3: *y

vsplat4 4t2, a1
stripmine:
vlw4 4t0, a2
vlw4 4t1, a3
vfma4 4t3, 4t2, 4t0, 4t1
vsw4 4t3, a3
add a2, a2, 4<<2
add a3, a3, 4<<2
sub a0, a0, 4
bgte a0, 4, stripmine

... handle edge cases

Packed-SIMD
Stripmine Loop Checks

\[
\begin{align*}
&\text{a0: n, a1: a, a2: } \ast x, \text{ a3: } \ast y \\
&\text{vsplat4 4t2, a1} \\
&\text{stripmine:} \\
&\text{vlw4 4t0, a2} \\
&\text{vlw4 4t1, a3} \\
&\text{vfma4 4t3, 4t2, 4t0, 4t1} \\
&\text{vsw4 4t3, a3} \\
&\text{add a2, a2, 4<<2} \\
&\text{add a3, a3, 4<<2} \\
&\text{sub a0, a0, 4} \\
&\text{bgte a0, 4, stripmine} \\
&\ldots \\
&\text{handle edge cases}
\end{align*}
\]
Handle Remainder Strip

\[
a_0: \ n, \ a_1: \ a, \ a_2: \ *x, \ a_3: \ *y
\]

vsplat4 4t2, a1
stripmine:
  vlw4 4t0, a2
  vlw4 4t1, a3
  vfma4 4t3, 4t2, 4t0, 4t1
  vsw4 4t3, a3
  add a2, a2, 4<<2
  add a3, a3, 4<<2
  sub a0, a0, 4
  bgte a0, 4, stripmine
  . . .
handle edge cases

Packed-SIMD
Port from 4-wide to 8-wide Packed-SIMD

Packed-SIMD

a0: n, a1: a, a2: *x, a3: *y

vsplat4 4t2, a1
stripmine:
vlw4 4t0, a2
vlw4 4t1, a3
vfma4 4t3, 4t2, 4t0, 4t1
vsw4 4t3, a3
add a2, a2, 4<<2
add a3, a3, 4<<2
sub a0, a0, 4
bgte a0, 4, stripmine
. . .
handle edge cases

New and Improved Packed-SIMD

a0: n, a1: a, a2: *x, a3: *y

vsplat8 8t2, a1
stripmine:
vlw8 8t0, a2
vlw8 8t1, a3
vfma8 8t3, 8t2, 8t0, 8t1
vsw8 8t3, a3
addi a2, a2, 8<<2
addi a3, a3, 8<<2
sub a0, a0, 8
bgte a0, 8, stripmine
. . .
handle even more edge cases
Packed-SIMD vs. Traditional Vectors

Packed-SIMD

<table>
<thead>
<tr>
<th>a0: n, a1: a, a2: *x, a3: *y</th>
</tr>
</thead>
<tbody>
<tr>
<td>vsplat4 4t2, a1</td>
</tr>
<tr>
<td>stripmine:</td>
</tr>
<tr>
<td>vlw4 4t0, a2</td>
</tr>
<tr>
<td>vlw4 4t1, a3</td>
</tr>
<tr>
<td>vfma4 4t3, 4t2, 4t0, 4t1</td>
</tr>
<tr>
<td>vsw4 4t3, a3</td>
</tr>
<tr>
<td>add a2, a2, 4&lt;&lt;2</td>
</tr>
<tr>
<td>add a3, a3, 4&lt;&lt;2</td>
</tr>
<tr>
<td>sub a0, a0, 4</td>
</tr>
<tr>
<td>bgte a0, 4, stripmine</td>
</tr>
<tr>
<td>. . .</td>
</tr>
<tr>
<td>handle edge cases</td>
</tr>
</tbody>
</table>

Traditional Vectors

| stripmine: |
| vsetvl t0, a0 |
| vlw v0, a2 |
| vlw v1, a3 |
| vfma v1, a1, v0, v1 |
| vsw v1, a3 |
| slli t1, t0, 2 |
| add a2, a2, t1 |
| add a3, a3, t1 |
| sub a0, a0, t0 |
| bnez a0, stripmine |
Set Vector Length Register
\[ \text{min}(\text{app\_length}, \text{hardware\_length}) \]

Packed-SIMD

\[
\begin{align*}
\text{a0: } n, &\quad \text{a1: a, } a2: \ast x, \quad \text{a3: } \ast y \\
\text{vsplat4 4t2, a1} \\
\text{stripmine:} \\
\text{vlw4 4t0, a2} \\
\text{vlw4 4t1, a3} \\
\text{vfma4 4t3, 4t2, 4t0, 4t1} \\
\text{vsw4 4t3, a3} \\
\text{add a2, a2, 4<<2} \\
\text{add a3, a3, 4<<2} \\
\text{sub a0, a0, 4} \\
\text{bgte a0, 4, stripmine} \\
\ldots \\
\text{handle edge cases}
\end{align*}
\]

Traditional Vectors

\[
\begin{align*}
\text{a0: } n, &\quad \text{a1: a, } a2: \ast x, \quad \text{a3: } \ast y \\
\text{stripmine:} \\
\quad \text{vsetvl t0, a0} \\
\quad \text{vlw v0, a2} \\
\quad \text{vlw v1, a3} \\
\quad \text{vfma v1, a1, v0, v1} \\
\quad \text{vsw v1, a3} \\
\quad \text{slli t1, t0, 2} \\
\quad \text{add a2, a2, t1} \\
\quad \text{add a3, a3, t1} \\
\quad \text{sub a0, a0, t0} \\
\quad \text{bnez a0, stripmine}
\end{align*}
\]
Vector Loads

Packed-SIMD

\[ a0: n, a1: a, a2: \star x, a3: \star y \]

vsplat4 4t2, a1
stripmine:
  vlw4 4t0, a2
  vlw4 4t1, a3
  vfma4 4t3, 4t2, 4t0, 4t1
  vsw4 4t3, a3
  add a2, a2, 4<<2
  add a3, a3, 4<<2
  sub a0, a0, 4
  bgte a0, 4, stripmine
  ...
  handle edge cases

Traditional Vectors

\[ a0: n, a1: a, a2: \star x, a3: \star y \]

stripmine:
  vsetvl t0, a0
  vlw v0, a2
  vlw v1, a3
  vfma v1, a1, v0, v1
  vsw v1, a3
  slli t1, t0, 2
  add a2, a2, t1
  add a3, a3, t1
  sub a0, a0, t0
  bnez a0, stripmine
Vector Multiply-Add Compute (vector and scalar operands)

Packed-SIMD

\[
\begin{align*}
    a0 &: n, \ a1 &: a, \ a2 &: *x, \ a3 &: *y \\
    &\text{vsplat4 4t2, a1} \\
    &\text{stripmine:} \\
    &\text{vlw4 4t0, a2} \\
    &\text{vlw4 4t1, a3} \\
    &\text{vfma4 4t3, 4t2, 4t0, 4t1} \\
    &\text{vsw4 4t3, a3} \\
    &\text{add a2, a2, 4<<<2} \\
    &\text{add a3, a3, 4<<<2} \\
    &\text{sub a0, a0, 4} \\
    &\text{bgte a0, 4, stripmine} \\
    &\ldots \\
    &\text{handle edge cases}
\end{align*}
\]

Traditional Vectors

\[
\begin{align*}
    a0 &: n, \ a1 &: a, \ a2 &: *x, \ a3 &: *y \\
    &\text{stripmine:} \\
    &\text{vsetvl t0, a0} \\
    &\text{vlw v0, a2} \\
    &\text{vlw v1, a3} \\
    &\text{vfma v1, a1, v0, v1} \\
    &\text{vsw v1, a3} \\
    &\text{slli t1, t0, 2} \\
    &\text{add a2, a2, t1} \\
    &\text{add a3, a3, t1} \\
    &\text{sub a0, a0, t0} \\
    &\text{bnez a0, stripmine}
\end{align*}
\]
Vector Store Results

### Packed-SIMD

- `vsetvl t0, a0`
- `vlw v0, a2`
- `vlw v1, a3`
- `vfma v1, a1, v0, v1`
- `vsw v1, a3`
- `slli t1, t0, 2`
- `add a2, a2, t1`
- `add a3, a3, t1`
- `sub a0, a0, t0`
- `bnez a0, stripmine`

### Traditional Vectors

- `vsplat4 4t2, a1`
- `stripmine: vlw4 4t0, a2 vlw4 4t1, a3 vfma4 4t3, 4t2, 4t0, 4t1 vsw4 4t3, a3 add a2, a2, 4<<2 add a3, a3, 4<<2 sub a0, a0, 4 bgte a0, 4, stripmine . . . handle edge cases`
Increment Address Pointers

Packed-SIMD

vsplat4 4t2, a1
stripmine:
  vlw4 4t0, a2
  vlw4 4t1, a3
  vfma4 4t3, 4t2, 4t0, 4t1
  vsw4 4t3, a3
add a2, a2, 4<<2
add a3, a3, 4<<2
sub a0, a0, 4
bgte a0, 4, stripmine
  . . .
handle edge cases

Traditional Vectors

a0: n, a1: a, a2: *x, a3: *y

stripmine:
  vsetvl t0, a0
  vlw v0, a2
  vlw v1, a3
  vfma v1, a1, v0, v1
  vsw v1, a3
  slli t1, t0, 2
  add a2, a2, t1
  add a3, a3, t1
  sub a0, a0, t0
  bnez a0, stripmine
Stripmine Loop
(note, no edge cases!)

a0: n, a1: a, a2: *x, a3: *y

vsplat4 4t2, a1
stripmine:
  vlw4 4t0, a2
  vlw4 4t1, a3
  vfma4 4t3, 4t2, 4t0, 4t1
  vsw4 4t3, a3
  add a2, a2, 4<<2
  add a3, a3, 4<<2
  sub a0, a0, 4
  bgte a0, 4, stripmine
  ... handle edge cases

Traditional Vectors

a0: n, a1: a, a2: *x, a3: *y

stripmine:
  vsetvl t0, a0
  vlw v0, a2
  vlw v1, a3
  vfma v1, a1, v0, v1
  vsw v1, a3
  slli t1, t0, 2
  add a2, a2, t1
  add a3, a3, t1
  sub a0, a0, t0
  bnez a0, stripmine
SPMD Programming Model

- Classic model reappears in CUDA/OpenCL
- Same restructuring as autovectorization, plus more on top to get performance

```c
Kernel(int n, float a,
       float* x, float* y) {
    if (tid < n) {
        y[tid] = a*x[tid]+y[tid];
    }
}

Kernel<<<(n+31)/32*32>>> (n, a, x, y);
```
Explicit Thread (Element) Identifier

```
Kernel(int n, float a,
       float* x, float* y) {
    if (tid < n) {
      y[tid] = a*x[tid]+y[tid];
    }
}

Kernel<<<(n+31)/32*32>>> (n, a, x, y);
```

SPMD Programming Model
Explicit Check if Thread Active

Kernel(int n, float a,
       float* x, float* y) {
   if (tid < n) {
     y[tid] = a*x[tid]+y[tid];
   }
}

Kernel<<<(n+31)/32*32>>>
(n, a, x, y);

SPMD
Programming Model
GPU/SIMT Architecture

a0: n, a1: a,
a2: *x, a3: *y

mv t0, tid
bge t0, a0, skip
slli t0, t0, 2
add a2, a2, t0
add a3, a3, t0
lw t1, 0(a2)
lw t2, 0(a3)
fma.s t0, a1, t1, t2
sw t0, 0(a3)

skip:
  stop
Check if active

a0: n, a1: a,
a2: *x, a3: *y

mv t0, tid
bge t0, a0, skip
slli t0, t0, 2
add a2, a2, t0
add a3, a3, t0
lw t1, 0(a2)
lw t2, 0(a3)
fma.s t0, a1, t1, t2
sw t0, 0(a3)
skip:
    stop
Calculate address and load X elements

\[
a0: n, \ a1: a, \\
a2: *x, \ a3: *y
\]

mv t0, tid
bge t0, a0, skip
slli t0, t0, 2
add a2, a2, t0
add a3, a3, t0
lw t1, 0(a2)
lw t2, 0(a3)
fma.s t0, a1, t1, t2
sw t0, 0(a3)

skip:
    stop
Address calc and load Y elements

\[
a0: n, a1: a, \\
a2: *x, a3: *y
\]

\[
\begin{align*}
\text{mv t0, tid} \\
\text{bge t0, a0, skip} \\
\text{slli t0, t0, 2} \\
\text{add a2, a2, t0} \\
\text{add a3, a3, t0} \\
\text{lw t1, 0(a2)} \\
\text{lw t2, 0(a3)} \\
\text{fma.s t0, a1, t1, t2} \\
\text{sw t0, 0(a3)} \\
\text{skip:} \\
\text{stop}
\end{align*}
\]
Compute Element Results

\[ a_0: n, \ a_1: a, \ \ a_2: \ x, \ a_3: \ y \]

\begin{verbatim}
mv t0, tid
bge t0, a0, skip
slli t0, t0, 2
add a2, a2, t0
add a3, a3, t0
lw t1, 0(a2)
lw t2, 0(a3)
fma.s t0, a1, t1, t2
sw t0, 0(a3)
skip:
\end{verbatim}

Stop

SIMT
a0: n, a1: a, a2: *x, a3: *y

mv t0, tid
bge t0, a0, skip
slli t0, t0, 2
add a2, a2, t0
add a3, a3, t0
lw t1, 0(a2)
lw t2, 0(a3)
fma.s t0, a1, t1, t2
sw t0, 0(a3)

skip:
stop

SIMT
GPU/SIMT vs. Traditional Vectors

**SIMT**

\[ a_0: n, \ a_1: a, \ a_2: x, \ a_3: y \]

- \texttt{mv t0, tid}
- \texttt{bge t0, a0, skip}
- \texttt{slli t0, t0, 2}
- \texttt{add a2, a2, t0}
- \texttt{add a3, a3, t0}
- \texttt{lw t1, 0(a2)}
- \texttt{lw t2, 0(a3)}
- \texttt{fma.s t0, a0, t1, t2}
- \texttt{sw t0, 0(a3)}

**Traditional Vectors**

\[ a_0: n, \ a_1: a, \ a_2: x, \ a_3: y \]

- \texttt{stripmine:}
  - \texttt{vsetvl t0, a0}
  - \texttt{vlw vr0, a2}
  - \texttt{vlw vr1, a3}
  - \texttt{vfma vr1, a1, vr0, vr1}
  - \texttt{vsw vr1, a3}
  - \texttt{slli t1, t0, 2}
  - \texttt{add a2, a2, t1}
  - \texttt{add a3, a3, t1}
  - \texttt{sub a0, a0, t0}
  - \texttt{bnez a0, stripmine}

**Example:**

\[ \texttt{stop} \]
Vector Length avoids Activity Branch

SIMT

```
a0: n, a1: a, a2: *x, a3: *y

mv t0, tid
bge t0, a0, skip
slli t0, t0, 2
add a2, a2, t0
add a3, a3, t0
lw t1, 0(a2)
lw t2, 0(a3)
fma.s t0, a0, t1, t2
sw t0, 0(a3)
skip:
  stop
```

Traditional Vectors

```
a0: n, a1: a, a2: *x, a3: *y

stripmine:
  vsetvl t0, a0
  vlw vr0, a2
  vlw vr1, a3
  vfma vr1, a1, vr0, vr1
  vsw vr1, a3
  slli t1, t0, 2
  add a2, a2, t1
  add a3, a3, t1
  sub a0, a0, t0
  bnez a0, stripmine
```
GPU/SIMT replicates scalar operands

**SIMT**

\[
\begin{align*}
a0: \ n, & \quad a1: \ a, \quad a2: \ *x, \quad a3: \ *y \\
& \quad mv \ t0, \ tid \\
& \quad bge \ t0, \ a0, \ skip \\
& \quad slli \ t0, \ t0, \ 2 \\
& \quad add \ a2, \ a2, \ t0 \\
& \quad add \ a3, \ a3, \ t0 \\
& \quad lw \ t1, \ 0(a2) \\
& \quad lw \ t2, \ 0(a3) \\
& \quad fma.s \ t0, \ a0, \ t1, \ t2 \\
& \quad sw \ t0, \ 0(a3) \\
& \quad \text{skip:} \\
& \quad \text{stop}
\end{align*}
\]

**Traditional Vectors**

\[
\begin{align*}
a0: \ n, & \quad a1: \ a, \quad a2: \ *x, \quad a3: \ *y \\
& \quad \text{stripmine:} \\
& \quad \quad vsetvl \ t0, \ a0 \\
& \quad \quad vlw \ vr0, \ a2 \\
& \quad \quad vlw \ vr1, \ a3 \\
& \quad \quad vfma \ vr1, \ a1, \ vr0, \ vr1 \\
& \quad \quad vsw \ vr1, \ a3 \\
& \quad \quad slli \ t1, \ t0, \ 2 \\
& \quad \quad add \ a2, \ a2, \ t1 \\
& \quad \quad add \ a3, \ a3, \ t1 \\
& \quad \quad sub \ a0, \ a0, \ t0 \\
& \quad \quad bnez \ a0, \ \text{stripmine}
\end{align*}
\]
GPU/SIMT redundant address calculations, dynamic memory coalescing

**SIMT**

a0: n, a1: a, a2: *x, a3: *y

mv t0, tid
bge t0, a0, skip
slli t0, t0, 2
add a2, a2, t0
add a3, a3, t0
lw t1, 0(a2)
lw t2, 0(a3)
fma.s t0, a0, t1, t2
sw t0, 0(a3)

**Traditional Vectors**

a0: n, a1: a, a2: *x, a3: *y

stripmine:
  vsetvl t0, a0
  vlw vr0, a2
  vlw vr1, a3
  vfma vr1, a1, vr0, vr1
  vsw vr1, a3
  slli t1, t0, 2
  add a2, a2, t1
  add a3, a3, t1
  sub a0, a0, t0
  bnez a0, stripmine

stop
Don’t need SIMT for SPMD

- Original CUDA model developed for NVIDIA SIMT engines, but don’t need SIMT hardware to run SPMD (CUDA/OpenCL) programs well
  - Check out Yunsup’s MICRO-2014 paper and upcoming thesis
Proposed 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>

Vector configuration

<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>

MVL is maximum vector length, implementation and configuration dependent, but MVL >= 4

<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>

8 vector predicate registers, with 1 bit per element

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

- Reconfigurable vector registers
  - Exchange unused architectural registers for longer vectors
- Mixed-precision support
  - From 8-bit to MAX(XLEN,FLEN) in powers of 2
- Integer, fixed-point, floating-point arithmetic
  - Floating-point requires corresponding scalar extension
  - Fixed-point to include rounding, saturation, scaling
- Unit-stride, Strided, Indexed Load/Stores
- Predication
Reconfigurable Vector Register File

- Programming model allows specifying number of architectural registers (1-32)
- Maximum hardware vector length automatically extends to fill capacity of register file

- Exchange unused architectural registers for longer hardware vectors

```
vsetcfg 4
vlen = 2
```

```
vsetcfg 2
vlen = 4
```
Mixed-Precision Support

Hardware subdivides physical register into multiple narrower architectural registers as requested
- Subword packing transparent to software
- Improved utilization of operand communication bandwidth
- Spatial functional-unit parallelism

- Only support elements up to \( \text{MAX}(XLEN,FLEN) \)
  - E.g. RV32IM would only support 8,16,32

\[
\begin{array}{c}
\text{vv0}[0] \\
\text{vv0}[1] \\
\text{vv1}[0] \\
\text{vv1}[1] \\
v0[2] \\
v0[1] \\
v1[0] \\
v1[1] \\
v1[2] \\
v1[3] \\
v2[0] \\
v2[1] \\
v3[0] \\
v3[1] \\
\end{array}
\]
Programmer’s View of Reconfigurability

- Before loop nest, vector configure instruction sets number required of each width of register
- After loop nest, unconfigure disables vector unit and avoid save/restore at context swap, potentially power down

```
vsetcfg #64, #32, #16, #8

stripmine:
  vsetvl t0, a0
  # ...
  # Code for loop body
  # ...
  sub a0, a0, t0
  bnez a0, stripmine

vuncfg # Turn off vector unit
```
Higher Performance from Parallel Lanes

Functional Unit

Vector Registers

Lane

Memory Subsystem
Vector Length Portability

- Same binary code works regardless of:
  - Number of physical register bits
  - Number of physical lanes

- 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

- Pros:
  - Denser encoding, sizes inherent in register number
  - Eliminates many difficult cases

- Cons:
  - Can’t reuse register for different sizes
  - Can’t initialize from memory with smaller type
**Vector Loads and Stores**

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

**Types:**
- Separate integer and floating-point loads and stores
  - Support FPU internal recoding
- Size inherent in destination register number (for integers, signed/unsigned determined by use)

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

- 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
Two 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 undefined value if predicate false
   - More complex code, better for out-of-order with renaming
   - Need additional merge(s) to rebuild complete vector

We’re choosing 1), as simpler and safer.

Use microarchitectural tricks for OoO machines to reduce amount of data transfer.
V versus Xhwacha

- V is proposal for a standard RISC-V vector extension
- Xhwacha is a non-standard Berkeley vector extension designed to push state-of-the-art in-order/decoupled vector machines
  - V and Xhwacha lane microarchitecture very similar
  - Multiple versions of Xhwacha have been fabricated
    - up to 34 GFLOPS/W running DGEMM with IEEE-2008 64-bit fused muladds

- Current Berkeley focus on bringing up OpenCL for Xhwacha
- V to follow
Example Vector Lane Organization (from Hwacha)

- Compact register file of four 1R1W SRAM banks
- Per-bank integer ALU
- Two independently scheduled FMA clusters
  - Total of four double-precision FMAs per cycle
- Pipelined integer multiplier
- Variable-latency decoupled functional units
  - Integer divide
  - Floating-point divide with square root
Systolic Bank Execution

- Sustains $n$ operands/cycle after $n$-cycle initial latency

- “Fire and forget” after hazards are cleared upon sequencing
- Chaining follows naturally from interleaving μops belonging to dependent instructions
Physical Vector Register File

- Bank partitioned into different segments for each supported data type width
  - vsetcfg 2, 3, 1, 0 (#64, #32, #16, #8) ➞ vlen = 16

Bank 1

| vv0[3]       | vv0[2]       |
| vv0[7]       | vv0[6]       |
| vv0[11]      | vv0[10]      |
| vv0[15]      | vv0[14]      |

Bank 0

| vv0[1]       | vv0[0]       |
| vv1[1]       | vv1[0]       |
| vv0[5]       | vv0[4]       |
| vv1[9]       | vv1[8]       |
| vv0[13]      | vv0[12]      |

- doubleword
- word
- halfword
OS Support

- Restartable page faults via microcode state dump, opaque to OS
  - Similar to DEC Vector Vax implementation
- Privileged specification describes XS sstatus field used to encode coprocessor status (Off, Initial, Clean, Dirty) to reduce context save/restore overhead.
Minimal V Implementation

![Diagram of Minimal V Implementation]
Minimal V Coprocessor Implementation

Standard RISC-V scalar x and f registers

- x0
- x1
- x31

- f0
- f1
- f31

32 vector data registers, v0-v31, four elements each, \( \text{MAX}(XLEN,FLEN) \)-bits wide

- v0[0]
- v0[1]
- v0[2]
- v0[3]
- v1[0]
- v1[1]
- v1[2]
- v1[3]
- v31[0]
- v31[1]
- v31[2]
- v31[3]

4K-8K bits

Vector configuration

CSR vcfg

10-15 bits

Vector length

CSR vlr

3 bits

32 bits

8 vector predicate registers, each four elements, with 1 bit per element

- p0[0]
- p0[1]
- p0[2]
- p0[3]
- p1[0]
- p1[1]
- p1[2]
- p1[3]
Questions?