RISC-V “V” Vector Extension

Sharzy

December 10, 2021

About “V” Extension

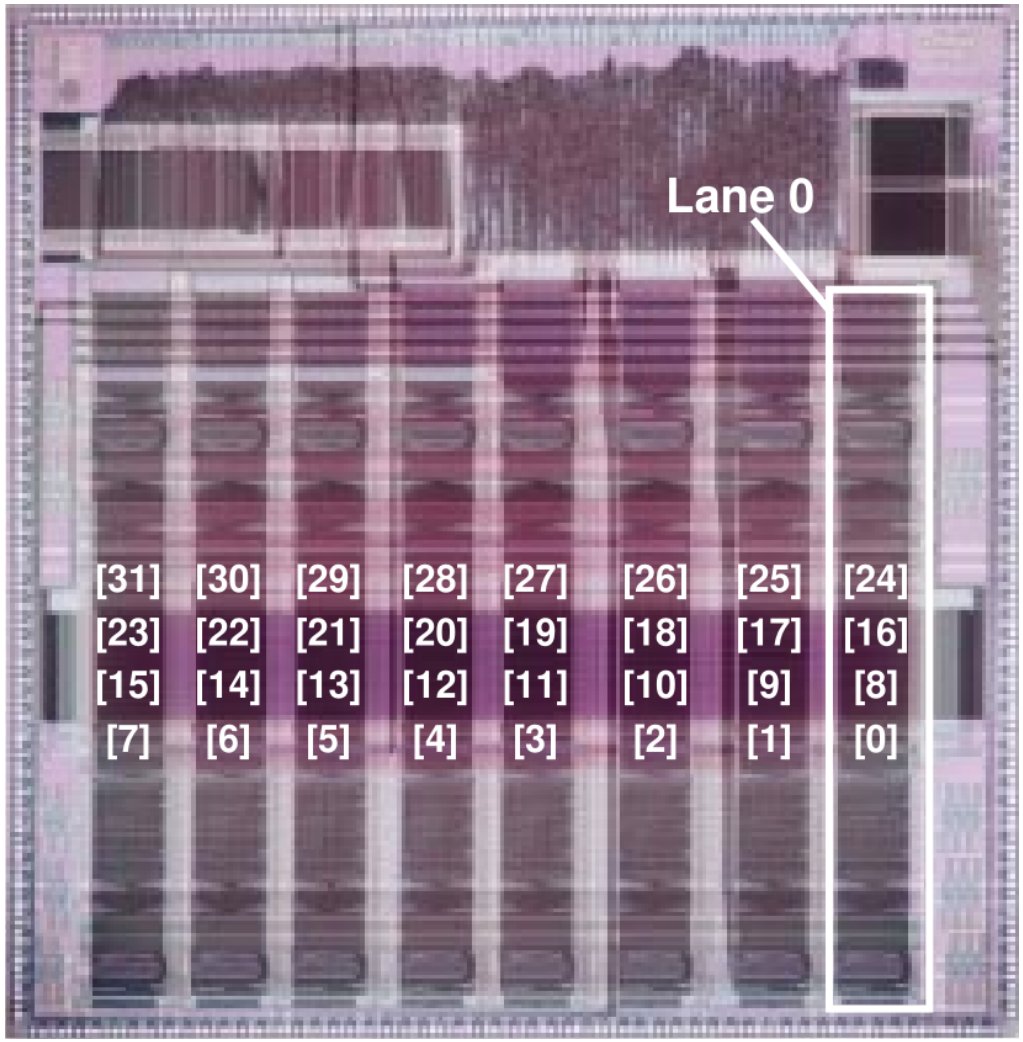
- A standard extension to the RISC-V ISA.
- Modern Cray-style vector ISA.
- Currently in version 1.0, frozen for public review.
- `https://github.com/riscv/riscv-v-spec`

Vector Extension Additinoal State

- 32 vector register, each of length `VLEN` (design-time constant), `vlenb = VLEN / 8` is stored in a CSR

Recall that elements of vector registers are usually distributed in lanes.

Same element in different registers lies in the same lane.



Vector Extension Additional State Cont'd

- `mstatus` and `vsstatus`: CSR for hypervisor use
- `vtype`: XLEN-wide read-only CSR, providing the default type used to interpret contents of vector
  - `vsew[2:0]`: encoding SEW (**S**elected **E**lement **W**idth)
  - `vlmul[2:0]`: encoding LMUL (**L**ength **Mul**tiplier)
  - `vma`, `vta`: **M**ask/**T**ail **A**gnostic
  - Other bits set to zero, reserved for future use
  - Can be only modified by `vset{i}vl{i}` instruction
  - `vill`: used to encode the return value of illegal `vset{i}vl{i}` instructions

Vector Extension Additional State Cont'd

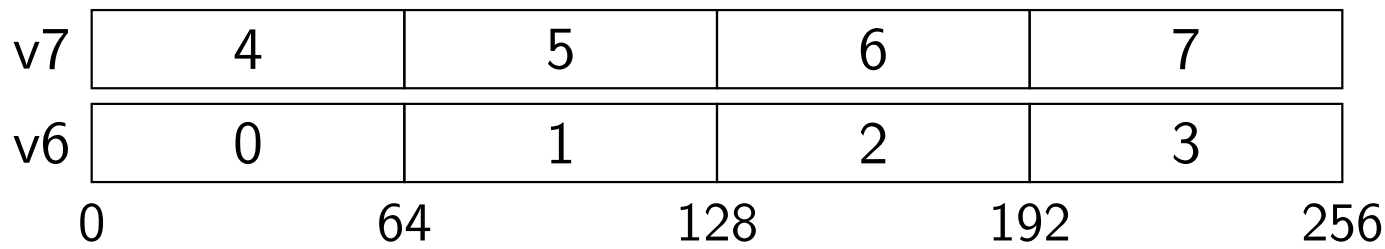
- `vl`: XLEN-wide CSR encoding the number of *userful* elements in a register, because we do not always want to operate on `LUML * VLEN/SEW` elements.
- `vstart`: read-only CSR, storing the index where the trap was taken. Instruction ignores elements preceding `vstart` to ensures forward-progress.
- `vxrm`, `vxsat`: Fixed-point rounding mode and fixed-point accrued saturation flag

SEW and LMUL

`SEW` $= 8, 16, 32, 64$, determined by on `vsew[2:0]`
`LMUL` $= 1, 2, 4, 8, 1/2, 1/4, 1/8$, determined by `vlmul[2:0]`

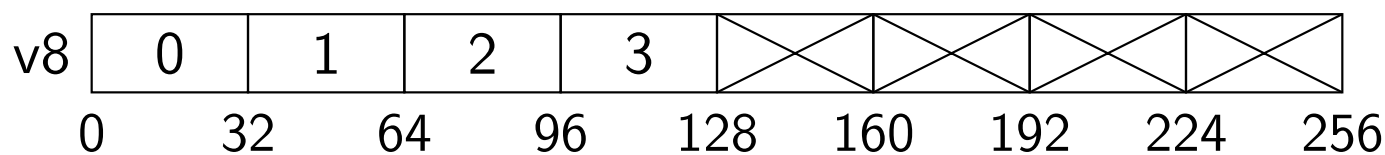
- If `LMUL` ≥ 1, adjacent `LMUL` vector registers forms a group, acts as if they are one register

  Example of LUML=2, VLEN=8, SEW=64b:



- If `LMUL` < 1, only `LMUL` proportion of the vector register is used

  Example of LMUL=1/2, VLEN=4, SEW=32b:



Mask

Many operation support masks:

```
; Vector register-register addition
vadd.vv vd, vs1, vs2, vm

; Usage example
vadd.vv v3, v1, v2, v0.t
vadd.vv v3, v1, v2
```

`vm` is either `v0.t` (encoded by 0) or unspecified (encoded by 1). If set to `v0.t`, the bits in `v0` is used as mask.

The mask of element i is stored at the i-th bit of the mask register. (regardless of `SEW` and `LMUL`)

In older versions of RVV spec, mask bit is aligned with SEW

Tail Agnostic and Mask Agnostic

Element types for index x:

```
prestart(x) = (0 ≤ x < vstart)
body(x)     = (vstart ≤ x < vl)
tail(x)     = (vl ≤ x < max(VLMAX,VLEN/SEW))
mask(x)     = unmasked || v0.mask[x]
active(x)   = body(x) && mask(x)
inactive(x) = body(x) && !mask(x)
```

Recall that `vta` and `vma` is part of vector type

- `vma` $= 0$ tail elements retains original values or overwritten with 1s after operation
- `vma` $= 1$ tail elements retains original values
- `vta` $= 0$ masked elements retains original values or overwritten with 1s after operation
- `vta` $= 1$ masked elements retains original values

Configuration Setting Instruction

Instructions `vset{i}vl{i}` are used to set vector type and length.

AVL: Application Vector Length

```
; rd = new vl, rs1 = AVL, vtypei = new vtype setting
vsetvli rd, rs1, vtypei
; rd = new vl, uimm = AVL, vtypei = new vtype setting
vsetivli rd, uimm, vtypei
; rd = new vl, rs1 = AVL, rs2 = new type value
vsetvl rd, rs1, rs2
```

`vl` is determined in a "smart" way:

`rd`	`rs1`	New `vl`
`-`	`!x0`	`min(x[rs1], VLMAX)`
`!x0`	`x0`	`VLMAX`
`x0`	`x0`	Keep existing `vl`

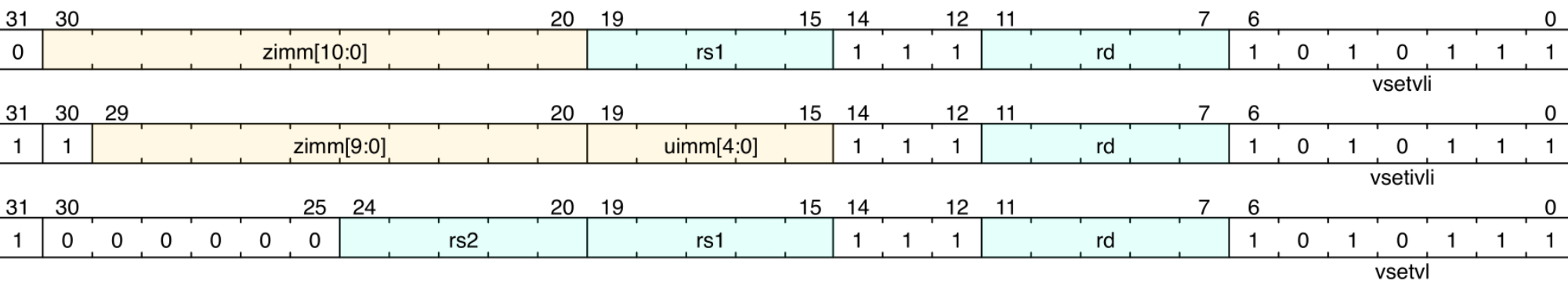
No need to encode `vlen` in binary. Compile once, run everywhere (probably)

Configuration Setting Instruction Cont'd

`vtypei` can be specified by some short notations in assembly

```
vsetvli t0, a0, e8, m2, ta, ma
; SEW = 8, LMUL = 2, tail agnostic, mask agnostic

vsetvli t0, a0, e32, mf2, ta, ma
; SEW = 32, LMUL = 1/2,  ...
```

Instruction encoding:



Load/Store Instruciton

Three addressing modes:

- Unit-stride

  `rd[i] = mem[x[rs1] + i * SEW]`

- Strided

  `rd[i] = mem[x[rs1] + i * x[rs2] * SEW]`

- Vector Indexed

  `rd[i] = mem[x[rs1] + vs2[i] * SEW]`

  - Ordered/Unorderd
  - Signed/Unsigned offset

Note that load/store instruction hard-encoded EEW (**E**ffective **E**lement **W**idth) in instruction.

EMUL is calculated as EEW/SEW*LMUL

Stripmining Example

```
  ; void *memcpy(void *dest, const void *src, size_t n)
  ; a0=dest, a1=src, a2=n
memcpy:
    mv a3, a0
loop:
  vsetvli t0, a2, e8,m8  ; element width = 8bit
                         ; length multiplier = 8
  vle8.v v0, (a1)        ; load bytes
    add a1, a1, t0       ; bump pointer
    sub a2, a2, t0       ; decrement count
  vse8.v v0, (a3)        ; store bytes
    add a3, a3, t0       ; bump pointer
    bnex a2, loop        ; any more?
    ret
```

Load/Store Segment Instruciton

"Segment" means contiguous fields in memory. For example

```
struct Color {
    uint_8 R;
    uint_8 G;
    uint_8 B;
};
```

Suppose we want to load an array of `RGB` to vector registers

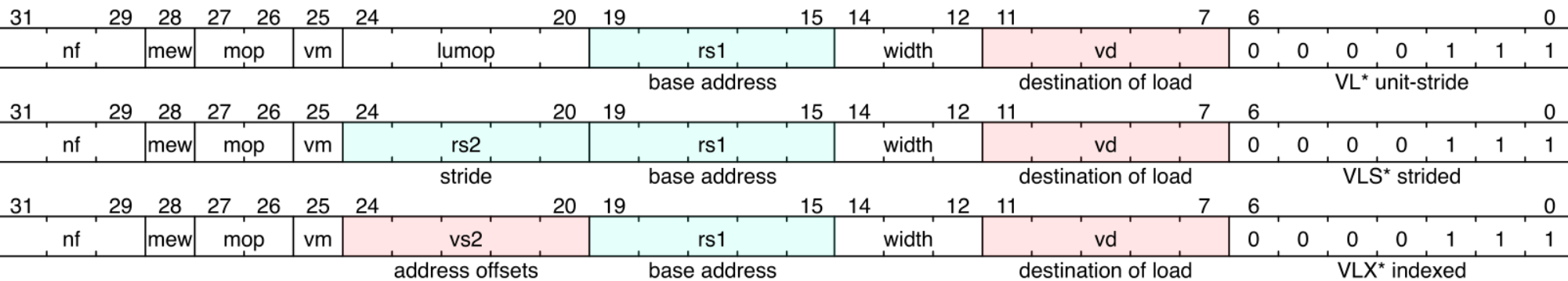
```
; load R to v4, v5, load G to v6, v7, load B to v8, v9

vsetvli a1, t0, e8, m2, ta, ma
; set SEW=8, LMUL=2

vlseg3e8.v v4, (x5)
; Load byte at addresses x5 + 3 * i     into (v4, v5)[i]
;  and byte at addresses x5 + 3 * i + 1 into (v6, v7)[i]
;  and byte at addresses x5 + 3 * i + 2 into (v8, v9)[i]
```

Load/Store Segment Instruciton

It also supports three addressing modes:

```
; <nf> number of fields
; <eew> effective element width
; nf * emul ≤ 8
vlseg<nf>e<eew>.v vd, (rs1), vm
vlsseg<nf>e<eew>.v vd, (rs1), rs2, vm
vluxseg<nf>ei<eew>.v vd, (rs1), rs2, vm
```

Load/Store Instruction Encoding



Fault-Only-First Loads

`vle{8,16,32,64}ff.v  vd, (rs1), vm`

- Only take a trap caused by synchronous exception on element 0.
- If element of index `i` > 0 raises an exception, trap is ignored and `vl` is set to `i`
- Register of index `i` $> i$ may be spuriously updated.
- Useful for functions such as unaligned **`strlen`**

  You do no know exactly how many bytes you need. And fixed-length vector load may access data in inaccessible pages

Memory Consistency Model

- Keep PC order on the local hart
- RVWMO at the instruction level
- Memory access is unordered except for ordered vector-indexed load/store.

Overview of Vector Operations

A vector operation be like:

vwaddu.wu

Destination type normal (), wide[1] (w), narrow (n), mask (m)

Operation name add, sub, xor, ...

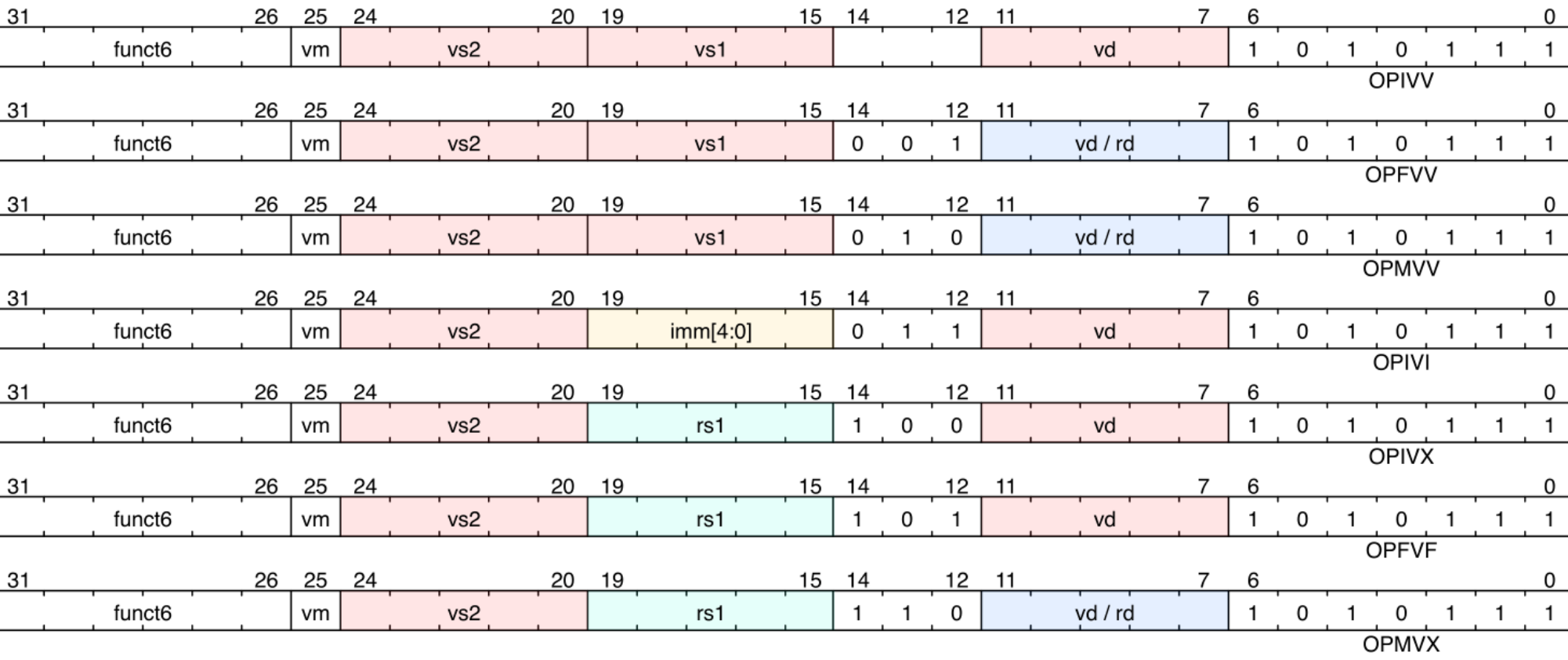
Signed / Unsigned signed (), unsigned (u)

First operand type vector (v), wide vector (w), mask (m)

Second operand type vector (v), scaler register (s), fp register (f), immediate (i), unsigned immediate (u), mask (m)

[1] "wide" means $\texttt{EEW} = 2 \times \texttt{SEW}$, while "narrow" means $\texttt{EEW} = \texttt{SEW}/2$

Vector Operations Instruction Encoding



Arithmetic Operations

Type	Listing
Arithmetic	`add`, `sub`, `madc`, `sbc`, `max`, `min`, `mul`, `div`, `rem`
Compare	`s{eq, ne, lt, le, gt}`
Bit Operation	`zext`, `sext`, `and`, `or`, `xor`, `sll`, `srl`, `sra`
Composed Operation	`macc`, `nmsac`, `madd`, `nmsub`
Register Merge	`merge`
Reduction	`red{sum, max, min, and, or, xor}`
Floating Point	`fcvt`, `fwcvt`, `fwncvt`, `fsqrt`, `frsqrt`, `frec`

- Compare operations write result to a register with mask format.
- All operations support masks, `merge` interprets mask not the usual way
- Most integer instruction except Bit Operation have their f-versions (`fadd`, `fmacc`, `fmerge`, `fredmin`, ...).
- Because floating-point addition is not associative, `redsum` has both ordered and unordered f-version.

Mask Operation

Functionality	Instruction
Logical Operation	`and`, `nand`, `andn`, `xor`, `nxor`, `or`, `nor`, `orn`, `xnor`
Count Population	`cpop`
Find first	`first`
Mask first	`s{b,i,o}f` (set-{before,including,only}-first)
Prefix sum	`iota` (treat mask as 0-1 array)
Element Index	`id`

Permutation

These are instruction that re-permute elements in a vector register (group)

```
vslideup.vx    vd, vs2, rs1, vm   ; vd[i+rs1] = vs2[i], head unchanged
vslideup.vi    vd, vs2, uimm, vm  ; vd[i+uimm] = vs2[i], head unchanged

vslidedown.vx  vd, vs2, rs1, vm   ; vd[i-rs1] = vs2[i], tail policy
vslidedown.vi  vd, vs2, uimm, vm  ; vd[i-uimm] = vs2[i], tail policy

vslide1up.vx   vd, vs2, rs1, vm   ; vd[0] = x[rs1], vd[i+1] = vs2[i]
vslide1down.vx vd, vs2, rs1, vm   ; vd[i] = rs2[i+1], vd[vl-1] = x[rs1]

vrgather.vv    vd, vs2, vs1, vm   ; vd[i] = vs2[vs1[i]]
vrgather.vx    vd, vs2, rs1, vm   ; vd[i] = vs2[x[rs1]]
vrgather.vi    vd, vs2, uimm, vm  ; vd[i] = vs2[uimm]

vcompress.vm   vd, vs2, vs1       ; collect unmasked elements in rs1
; No `decompress` because it can be implemented by `iota` and `gather`
```

A lot of inter-lane operation, difficult for hardware design.

Exception

Precise vector trap:

- Older instructions have committed their results
- No newer instructions altered architectural state
- Operations preceding `vstart` have committed their results
- Operations at or following `vstart` may altered architectural state, but can produce the correct state if the instruction is restarted.

Current standard extension does not have support for other type of exceptions.

Conclusion

- RVV has a flexible design. Easy to write code that efficiently runs on hardware of different configurations.
- Comprehensive support for mixed precision and dynamic precision.
- A lot of inter-lane operations which make hardware design difficult.

```
\Thanks
\immediate\write18{wget questions}
\end{presentation}
```