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, a la original Cray vectors

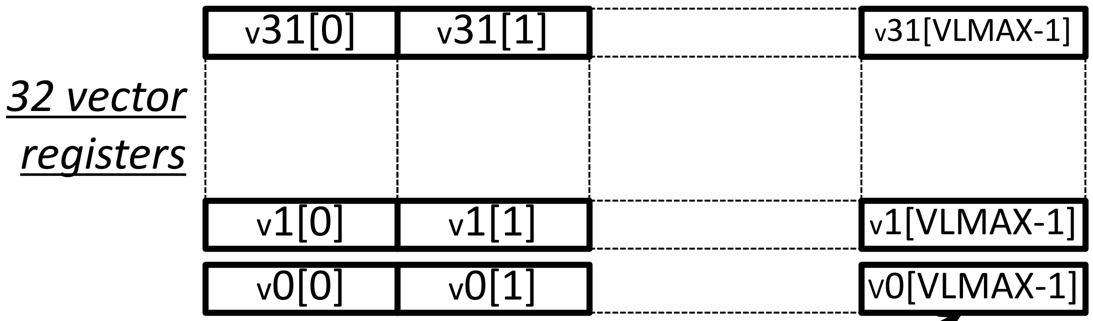
Goals for RISC-V Standard V Extension

- 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
- 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



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

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

vl

Vector length CSR sets number of elements active in each instruction

vstart

Resumption element after trap

fcsr (vxrm/vxsat)

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

Name	Issue Policy	Issue Width	VLEN (bits)	Datapath (bits)	VLEN/Datapath (beats)
Smallest	InO	1	32	32	1
Simple	InO	1	512	128	4
InO-Spatial	InO	2	128	128	1
OoO-Spatial	OoO	2-3	128	128	1
OoO-Temporal	OoO	2-3	512	128	4
OoO-Server	OoO	3-6	2048	512	4
OoO-HPC	OoO	3-6	16384	2048	8

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: `vtype` register

- Added a control register to hold some information about current setting of vector unit
- **`vtype`** fields (total additional 6-7 state bits)
  - **`vsew`**: standard element width (SEW=8,16,32,...,1024)
  - **`vlmul`**: vector length multiplier (LMUL=1,2,4,8)
  - **`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:

  VLMAX = VLEN/SEW

  - E.g., VLEN = 512b, SEW=32b, => VLMAX = 16
- Current active vector length set by **`vl`** register

  0<=**`vl`**<=VLMAX

vsetvli/vsetvl instructions

- **`vsetvli`** instruction sets both vector configuration and vector length:
  - **`vsetvli`** *rd, rs1, imm*

Immediate encodes vtype: <vsew, vlmul, vediv>

Application vector length (AVL)

Returns setting of vector length in scalar register

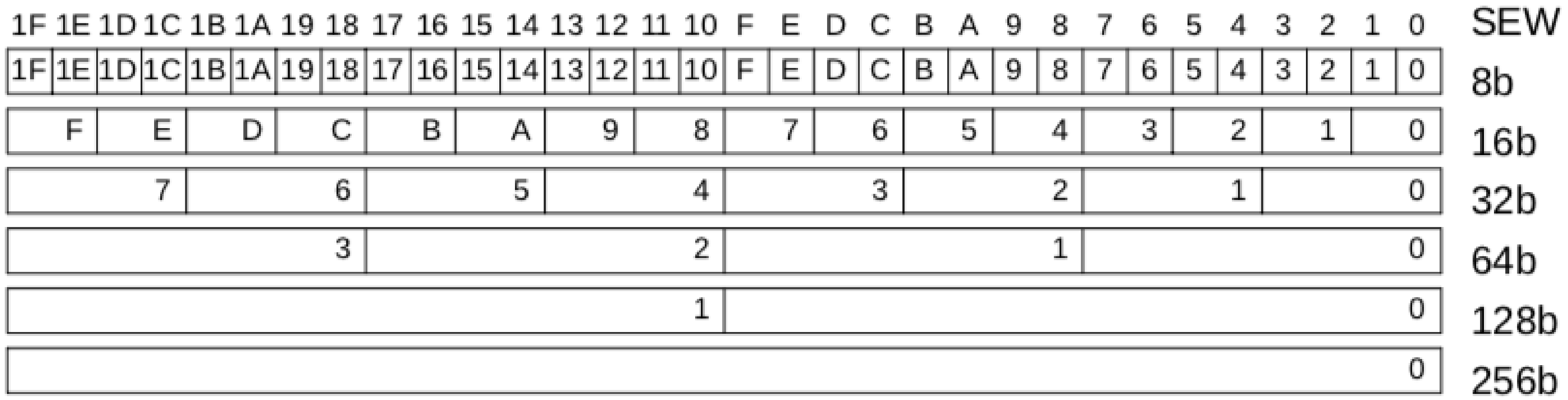
- Vector length **vl** set to *min(AVL, VLMAX)*

Simple memcpy example

```
# 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
```

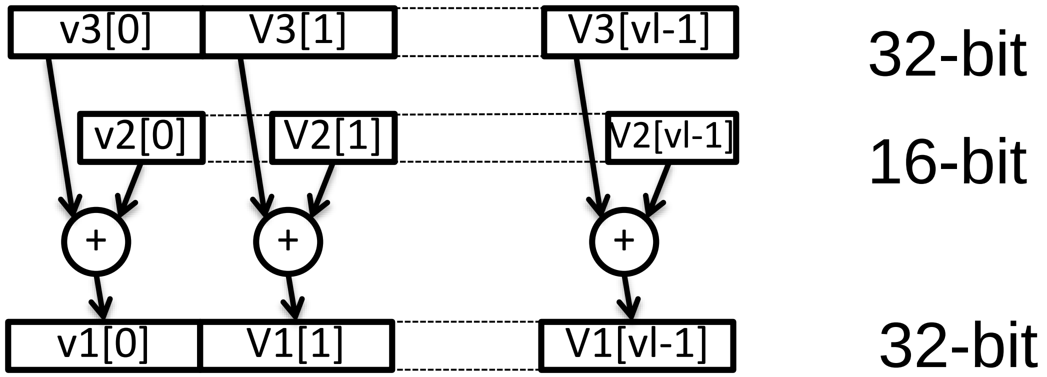
Mapping Elements to Vector State

VLEN=256b


1F 1E 1D 1C 1B 1A 19 18 17 16 15 14 13 12 11 10 F E D C B A 9 8 7 6 5 4 3 2 1 0
SEW
1F 1E 1D 1C 1B 1A 19 18 17 16 15 14 13 12 11 10 F E D C B A 9 8 7 6 5 4 3 2 1 0
8b
F E D C B A 9 8 7 6 5 4 3 2 1 0
16b
7 6 5 4 3 2 1 0
32b
3 2 1 0
64b
1 0
128b
0
256b

“Obvious” mapping in case of LMUL=1

Challenge: Mixed-Precision Arithmetic

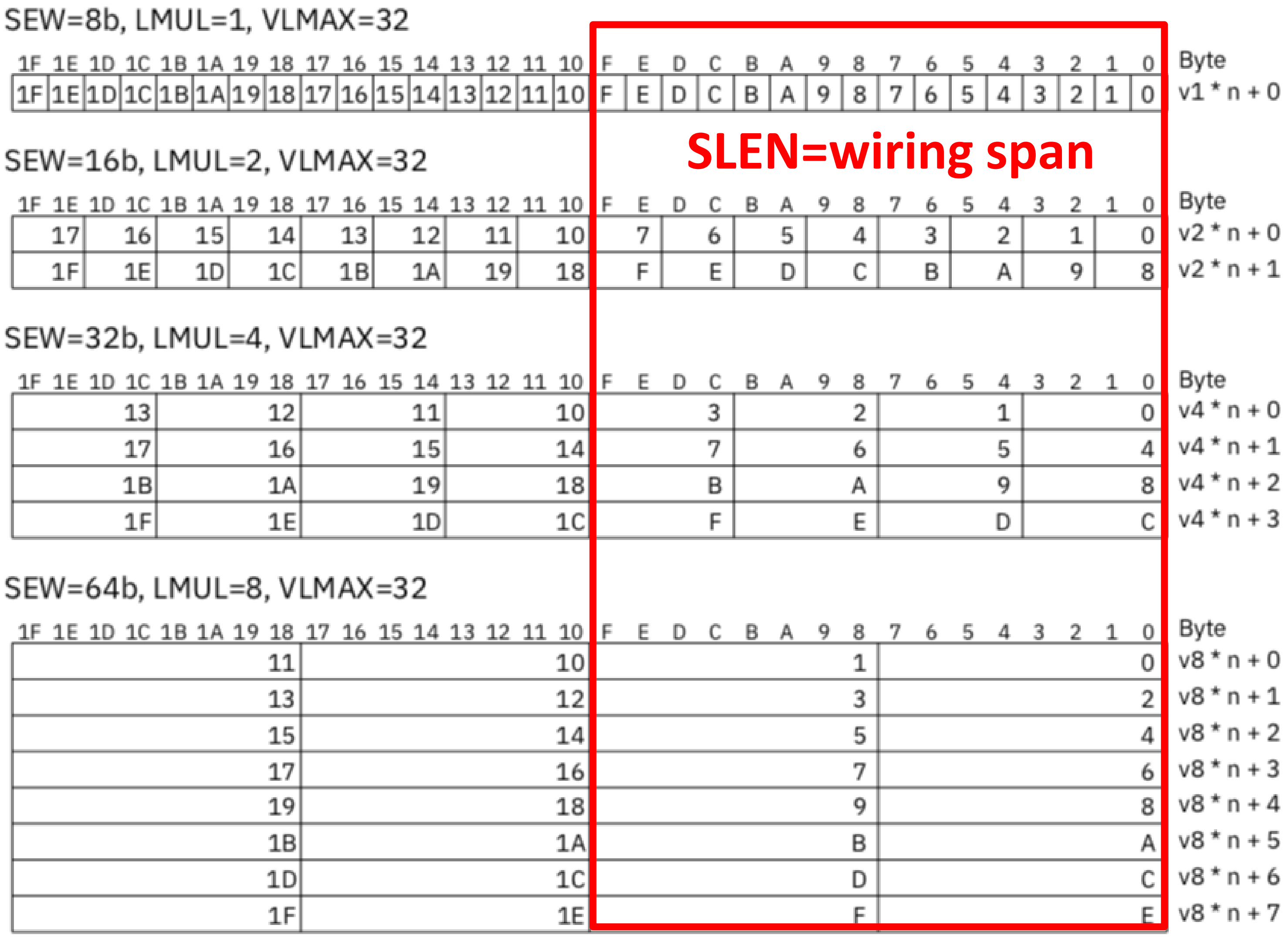


- To preserve accuracy with fewest bits, use mixed-precision, so some operations have 2*SEW operands or results (also 4*SEW)
- Problem 1: To keep same VLMAX, need more bits in registers
- Problem 2: Operands should “line up” to avoid datapath wiring

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

Mapping Elements LMUL>1 VLEN=256 SLEN=128


SLEN=wiring span

SEW=8b, LMUL=1, VLMAX=32

1F	1E	1D	1C	1B	1A	19	18	17	16	15	14	13	12	11	10	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0	Byte
1F	1E	1D	1C	1B	1A	19	18	17	16	15	14	13	12	11	10	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0	v1 * n + 0

SEW=16b, LMUL=2, VLMAX=32

1F	1E	1D	1C	1B	1A	19	18	17	16	15	14	13	12	11	10	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0	Byte
	17		16		15		14		13		12		11		10		7		6		5		4		3		2		1		0	v2 * n + 0
	1F		1E		1D		1C		1B		1A		19		18		F		E		D		C		B		A		9		8	v2 * n + 1

SEW=32b, LMUL=4, VLMAX=32

1F	1E	1D	1C	1B	1A	19	18	17	16	15	14	13	12	11	10	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0	Byte
			13				12				11				10				3				2				1				0	v4 * n + 0
			17				16				15				14				7				6				5				4	v4 * n + 1
			1B				1A				19				18				B				A				9				8	v4 * n + 2
			1F				1E				1D				1C				F				E				D				C	v4 * n + 3

SEW=64b, LMUL=8, VLMAX=32

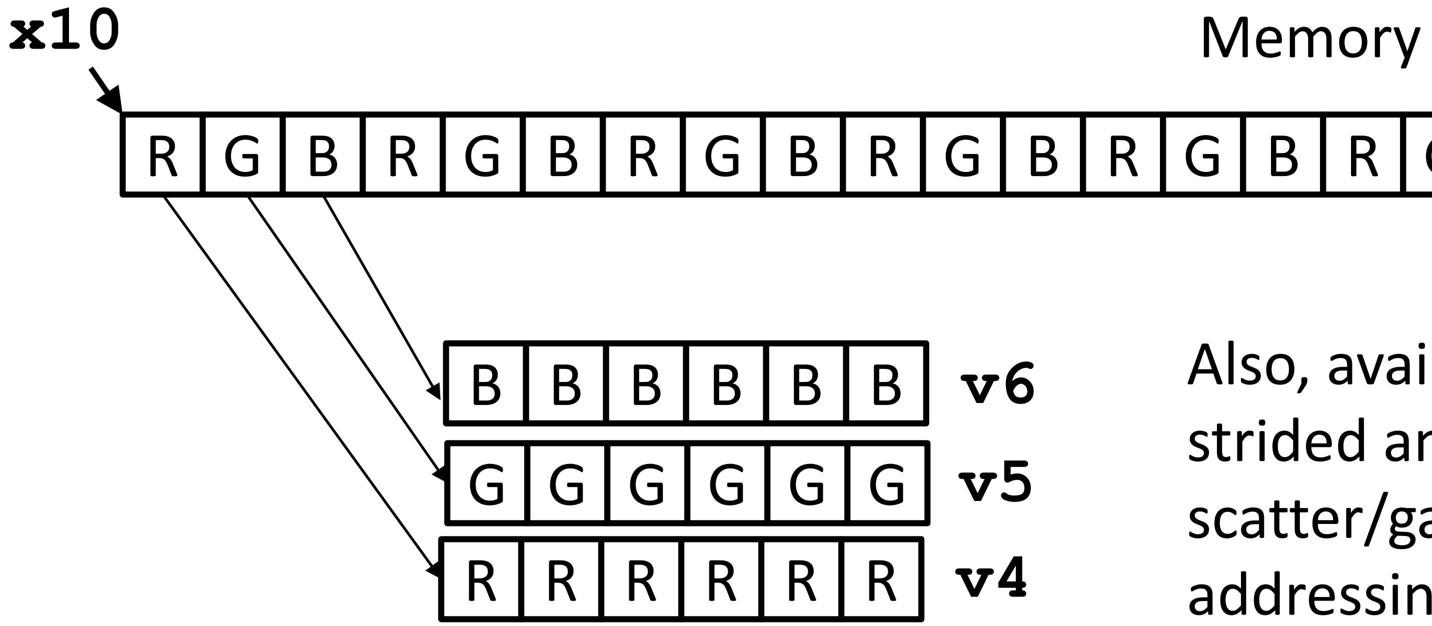
1F	1E	1D	1C	1B	1A	19	18	17	16	15	14	13	12	11	10	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0	Byte
							11								10								1								0	v8 * n + 0
							13								12								3								2	v8 * n + 1
							15								14								5								4	v8 * n + 2
							17								16								7								6	v8 * n + 3
							19								18								9								8	v8 * n + 4
							1B								1A								B								A	v8 * n + 5
							1D								1C								D								C	v8 * n + 6
							1F								1E								F								E	v8 * n + 7

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)

Vector Segment Loads/Stores

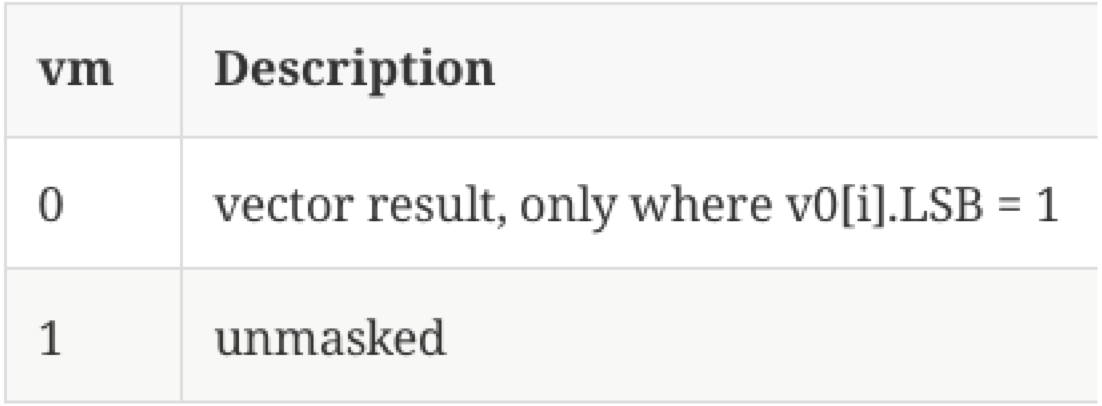
```
vlseg3b.v v4, (x10)
 # Load packed 8b RGB values into v4, v5, v6
```



Also, available for strided and scatter/gather addressing modes

Predicated Execution

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



vm	Description
0	vector result, only where v0[i].LSB = 1
1	unmasked

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 **vstart** register holds the first element to be processed by next vector instruction, and every vector instruction resets **`vstart`** to 0.
- On a trap, EPC points to faulting instruction and **`vstart`** points to element at which to continue execution; preceding elements <**vstart** unchanged
- User software should not try to use **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?

- Contributors include: Alon Amid, Krste Asanovic, Allen Baum, Alex Bradbury, Tony Brewer, Chris Celio, Aliaksei Chapyzhenka, Silviu Chiricescu, Ken Dockser, Bob Dreyer, Roger Espasa, Sean Halle, John Hauser, David Horner, Bruce Hoult, Bill Huffman, Constantine Korikov, Ben Korpan, Robin Kruppe, Yunsup Lee, Guy Lemieux, Filip Moc, Rich Newell, Albert Ou, David Patterson, Colin Schmidt, Alex Solomatnikov, Steve Wallach, Andrew Waterman, Jim Wilson.