The RISC-V Vector ISA

Krste Asanovic, krste@berkeley.edu, Vector WG Chair

Roger Espasa, roger.espasa@esperanto.ai, Vector WG Co-Chair

Vector Extension Working Group

Why a Vector Extension?

Vector ISA Goodness

- Reduced instruction bandwidth
- Reduced memory bandwidth
- Lower energy
- Exposes DLP
- Masked execution
- Gather/Scatter
- From small to large VPU

RISC-V Vector Extension

- Small
- Natural memory ordering
- Masks folded into vregs(*)
- Scalar, Vector & Matrix(*)
- Typed registers
- Reconfigurable
- Mixed-type instructions
- Common Vector/SIMD programming model
- Fixed-point support
- Easily Extensible
- Best vector ISA ever ☺

Domains

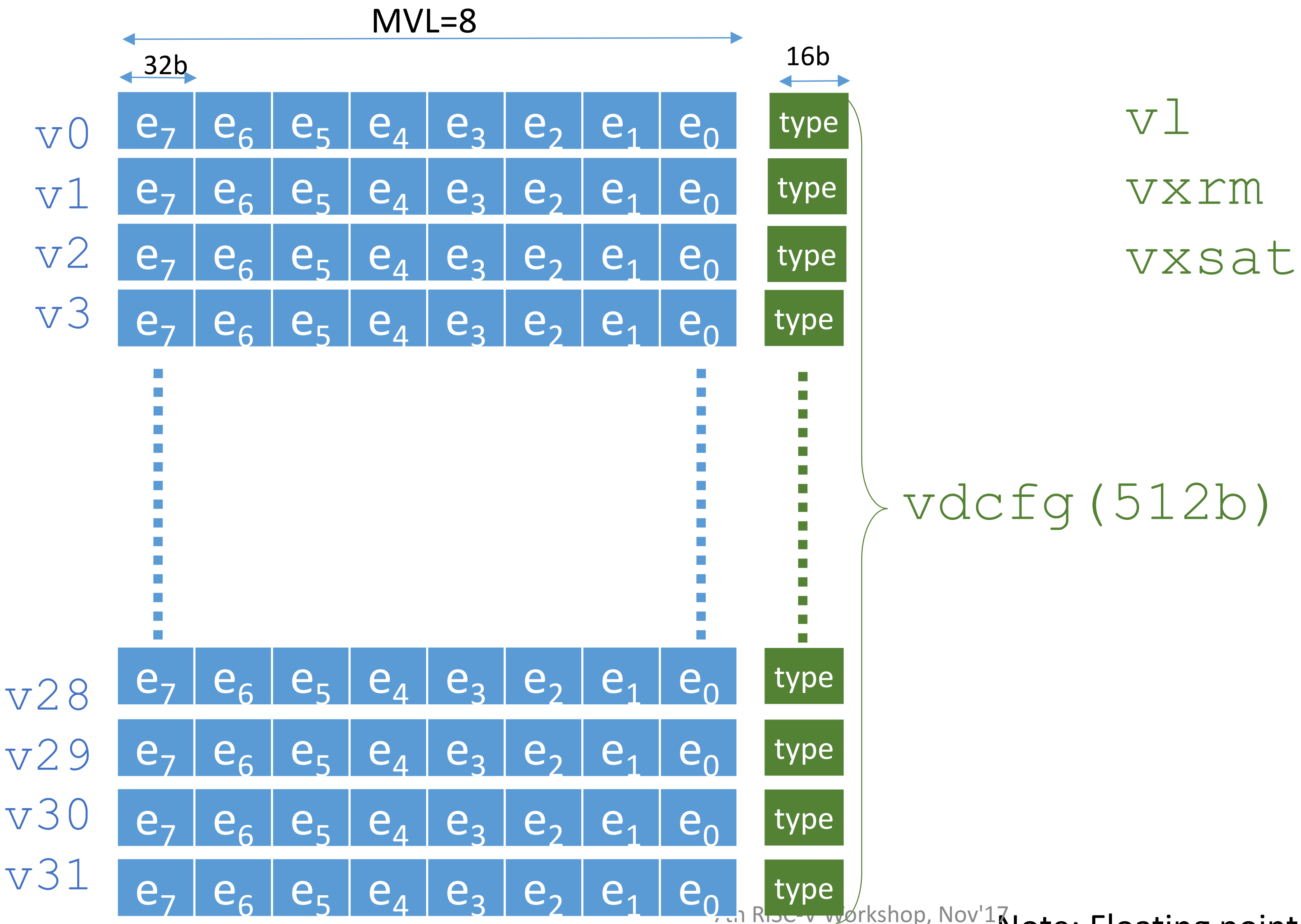
- Machine Learning
- Graphics
- DSP
- Crypto
- Structural analysis
- Climate modeling
- Weather prediction
- Drug design
- And more…

(*)Changed since last Workshop Presentation

The Vector ISA in a nutshell

- 32 vector registers (v0 … v31)
  - Each register can hold either a scalar, a vector or a matrix (shape)
  - Each vector register has an associated type (polymorphic encoding)
  - Variable number of registers (dynamically changeable)
- Vector instruction semantics
  - All instructions controlled by Vector Length (VL) register
  - All instructions can be executed under mask
  - Intuitive memory ordering model
  - Precise exceptions supported
- Vector instruction set:
  - All instructions present in base line ISA are present in the vector ISA
  - Vector memory instructions supporting linear, strided & gather/scatter access patterns
  - Optional Fixed-Point set
  - Optional Transcendental set

New Architectural State



vl	(XLEN)
vxrm	(3b)
vxsat	(1b)

Note: Floating point flags use the existing scalar flags

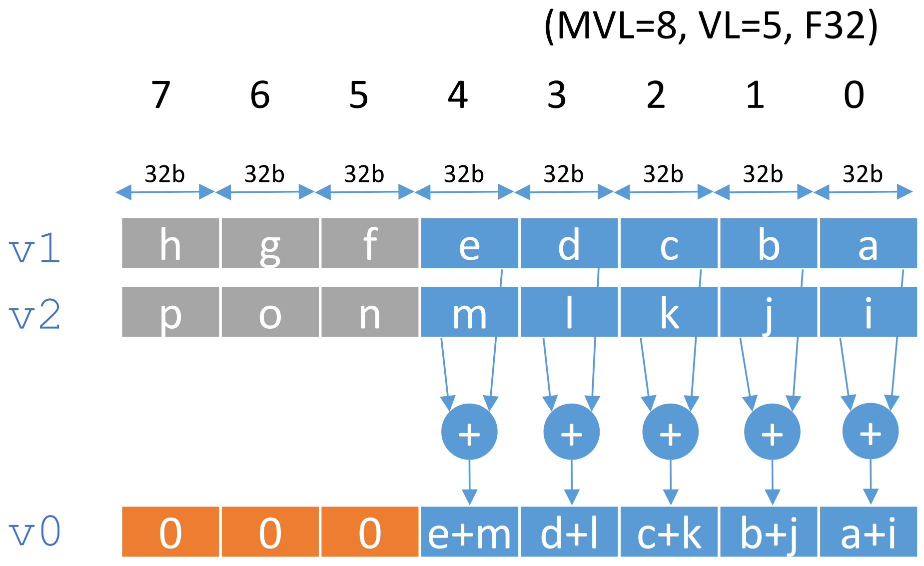
Complete Vector Instruction List

VOP						VMEM	
vmadd	vadd	vmerge	vsll	vclass	vround	vld	vamoswap
vnmadd	vaddi	vmin	vslli	vpopc	vclip	vst	vamoadd
vmsub	vand	vmul	vsra	vsgnj	vextract	vlds	vamoand
vnmsub	vandi	vmulh	vsrai	vsgnjn	vmv	vsts	vamoor
	vdiv	vsne	vsrl	vsgnjx		vldx	vamoxor
	vseq	vor	vsrli	vsqrt		vstx	vamomax
	vsge	vori	vsub	vcvt			vamomin
	vslt	vrem	vxor				
	vmax	vselect	vxori				

Adding two vector registers

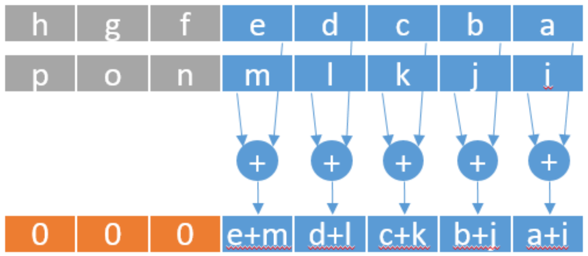
vadd v1, v2 → v0

```
for (i = 0; i < vl; i++ )
{
   v0[i] = v1[i] +F32 v2[i]
}
for (i = vl; i < MVL; i++ )
{
   v0[i] = 0
}
```

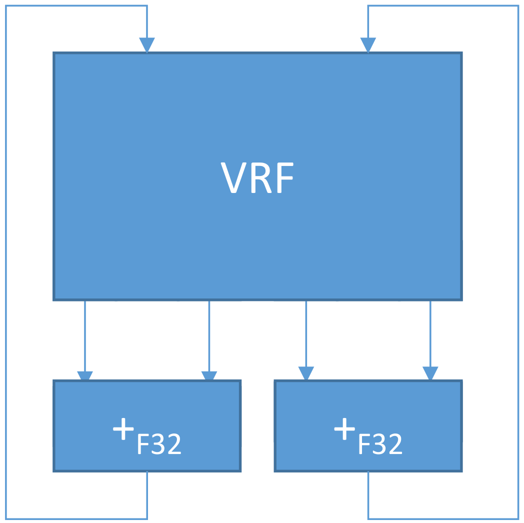


- When VL is zero, dest register is fully cleared
- Operations past ‘vl’ shall not raise exceptions
- Destination can be same as source

How is this executed? SIMD? Vector? Up to you!

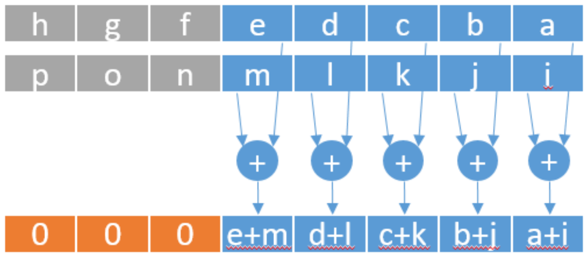


2-lane implementation



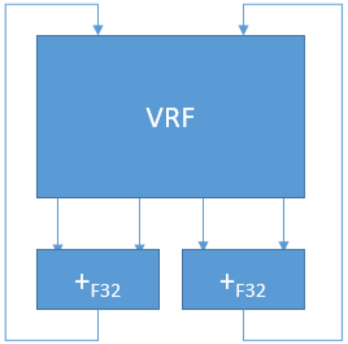
1st clock:	a+i, b+j
2nd clock:	c+k, d+l
3rd clock:	e+m, 0
4th clock:	up to you

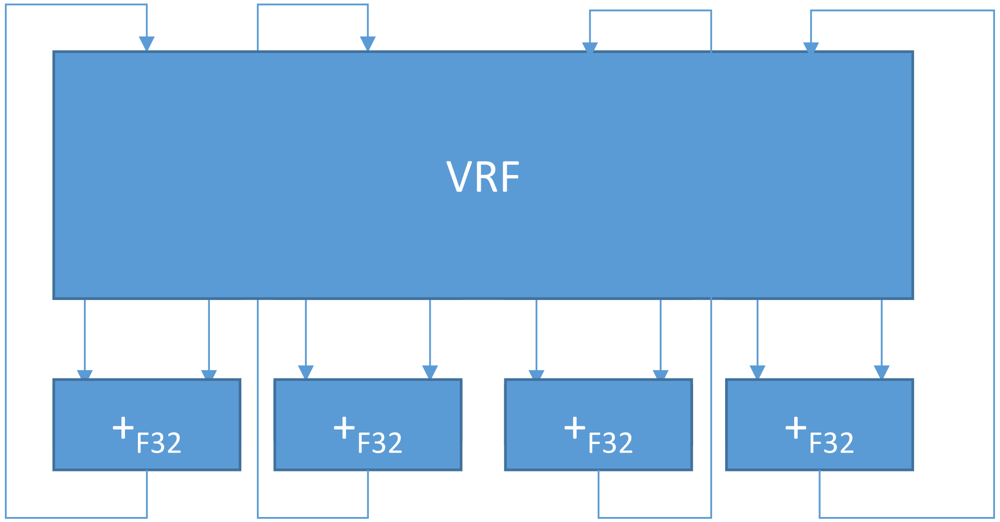
How is this executed? SIMD? Vector? Up to you!



4-lane implementation

2-lane implementation

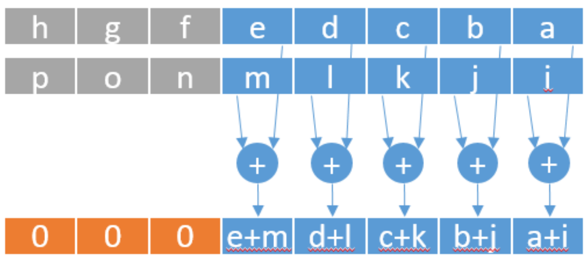




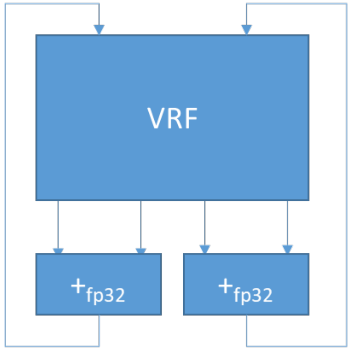
1st clock:	a+i, b+j
2nd clock:	c+k, d+l
3rd clock:	e+m, 0
4th clock:	up to you

1st clock:	a+i, b+j, c+k, d+l
2nd clock:	e+m, 0, 0, 0

How is this executed? SIMD? Vector? Up to you!

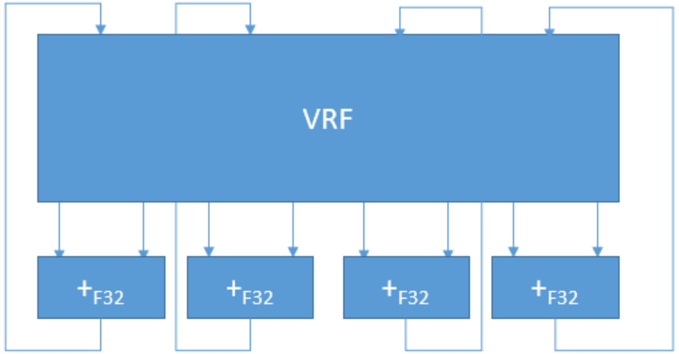


2-lane implementation



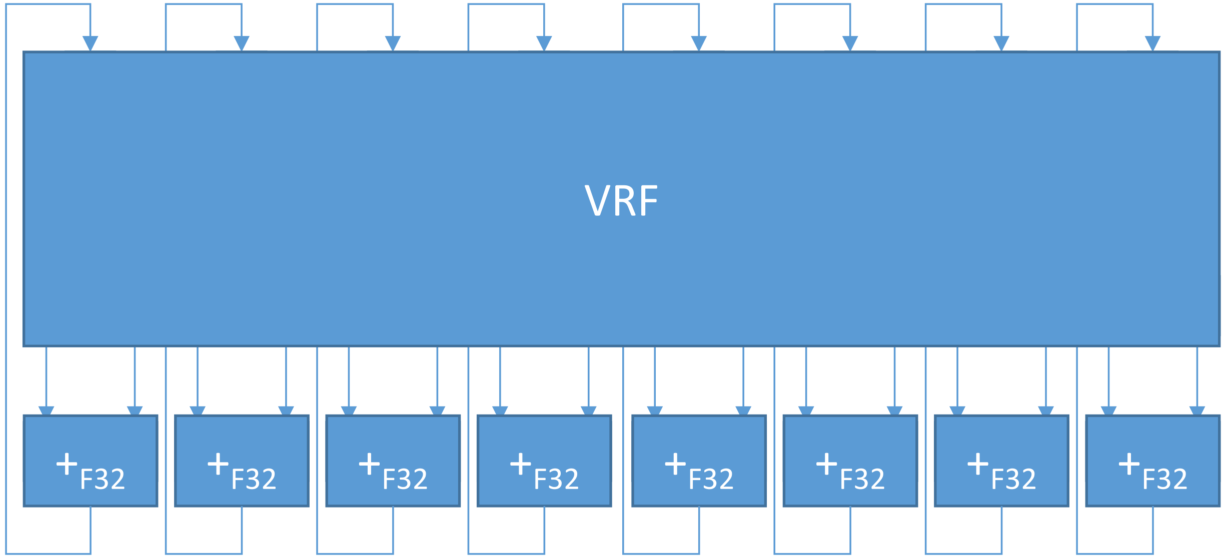
1st clock:	a+i, b+j
2nd clock:	c+k, d+l
3rd clock:	e+m, 0
4th clock:	up to you

4-lane implementation



1st clock:	a+i, b+j, c+k, d+l
2nd clock:	e+m, 0, 0, 0

8-lane implementation (a.k.a. SIMD)



1st clock: a+i, b+j, c+k, d+l, e+m, 0, 0, 0

Number of lanes is transparent to programmer
Same code runs independent of # of lanes

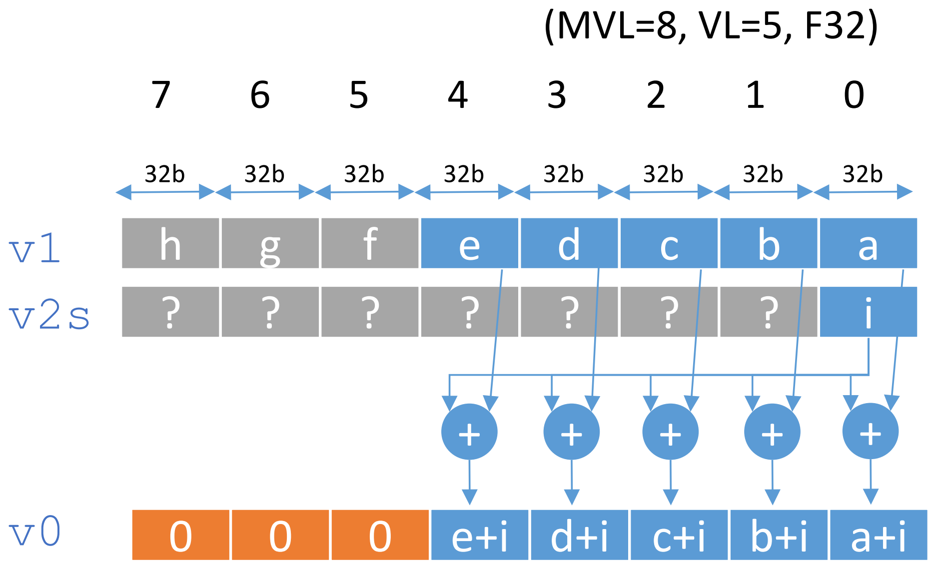
Adding a vector and a scalar

Scalar values in the Vector Register File

- The data inside a VREG can have 3 possible shapes:
  - A single scalar value
  - A vector (i.e., what you'd expect)
  - A matrix (optional, not in the base spec)
- The current shape is held in the per-vreg type field
  - Shape changes cause a VRF reset (discussed later)
- A vector register with shape scalar
  - Only holds one value
  - Implementation choice: where exactly this one value is stored within the vector is not defined by the spec. Whether the value is replicated to every lane is also implementation dependent.

vadd v1, v2.s → v0

```
for (i = 0; i < vl; i++ )
{
   v0[i] = v1[i] +F32 v2[0]
}
for (i = vl; i < MVL; i++ )
{
   v0[i] = 0
}
```



- Implementations are free to replicate the scalar value across all elements in the vector register
- Assembly notation for indicating scalar operands still T.B.D

Masked execution

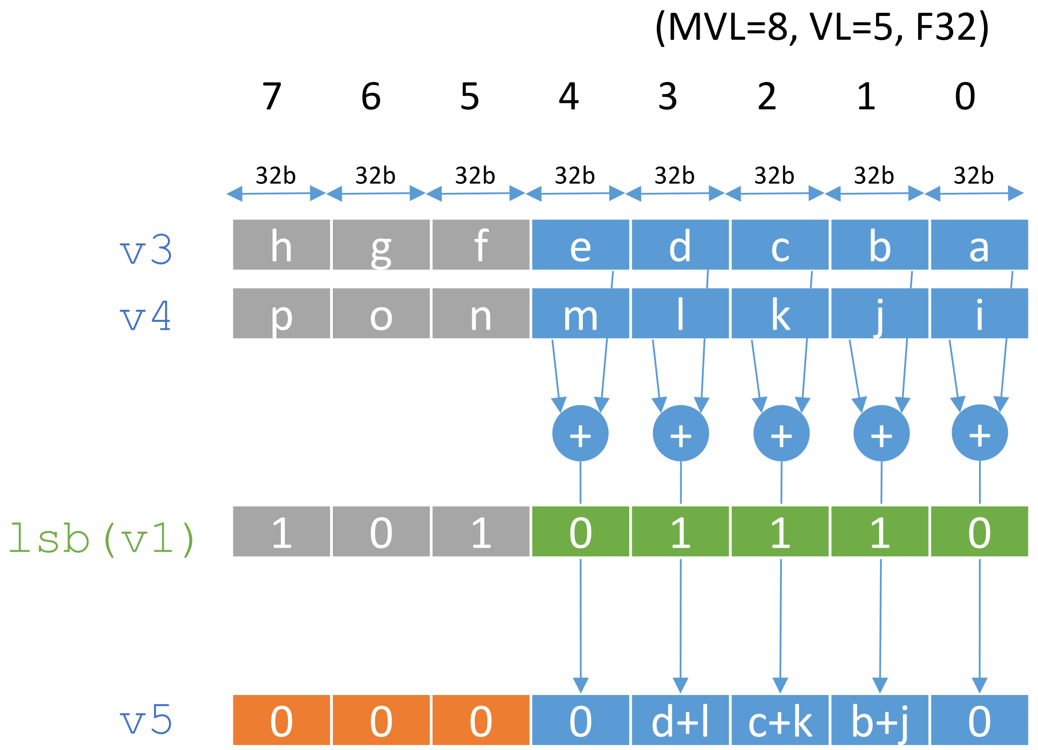
Masked execution

- Masks are stored in regular vector registers
  - The LSB of each element is used as a boolean “0” or “1” value
  - Other bits ignored
- Masks are computed with compare operations (vseq, vsne, vslt, vsge)
  - veq v6, v7 → v1
  - Comparison results are integer “0” or “1” (can’t be assigned to float types)
  - Encoded with as many bits as the destination register element size
- Instructions use 2 bits of encoding to select masked execution
  - 00 : No masking (== assume masking is 0xFFFF…FFFF)
  - 01 : unused (used for other encodings)
  - 10 : Use v1’s elements lsb as the mask
  - 11 : Use ~v1’s elements lsb as the mask

vadd v3, v4, v1.t → v5

```
for (i = 0; i < vl; i++ )
{
   v5[i] = lsb(v1[i]) ? v3[i] +F32 v4[i] : 0;
}
for (i = vl; i < MVL; i++ )
{
   v5[i] = 0
}
```

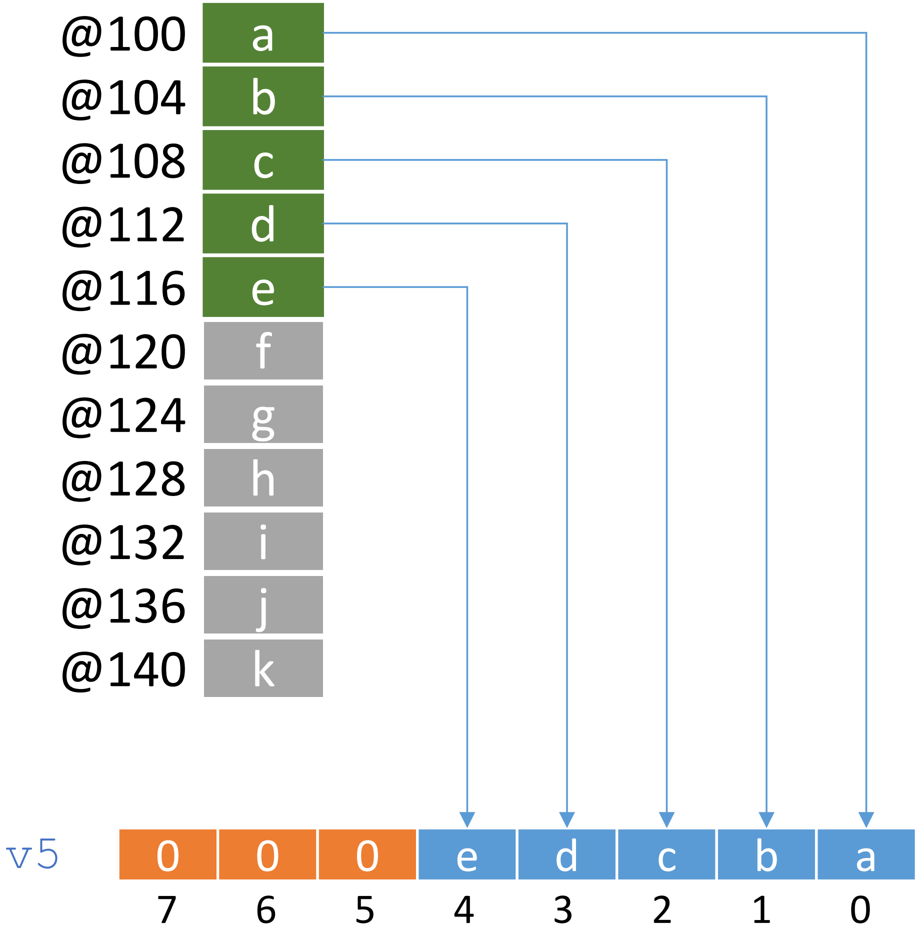
- Remember: v1 is the only register used as mask source
- Masked-out operations shall not raise any exceptions
- Assembly notation still TBD



Vector Load (unit stride)

vld 80(x3)→ v5

```
sz  = sizeof_type(v5);     // 4
tmp = x3 + 80;             // x3 = 20
for (i = 0; i < vl; i++ )
{
    v5[i] = read_mem(tmp, sz);
    tmp = tmp + sz;
}
for (i = vl; i < MVL; i++ )
{
    v0[i] = 0
}
```



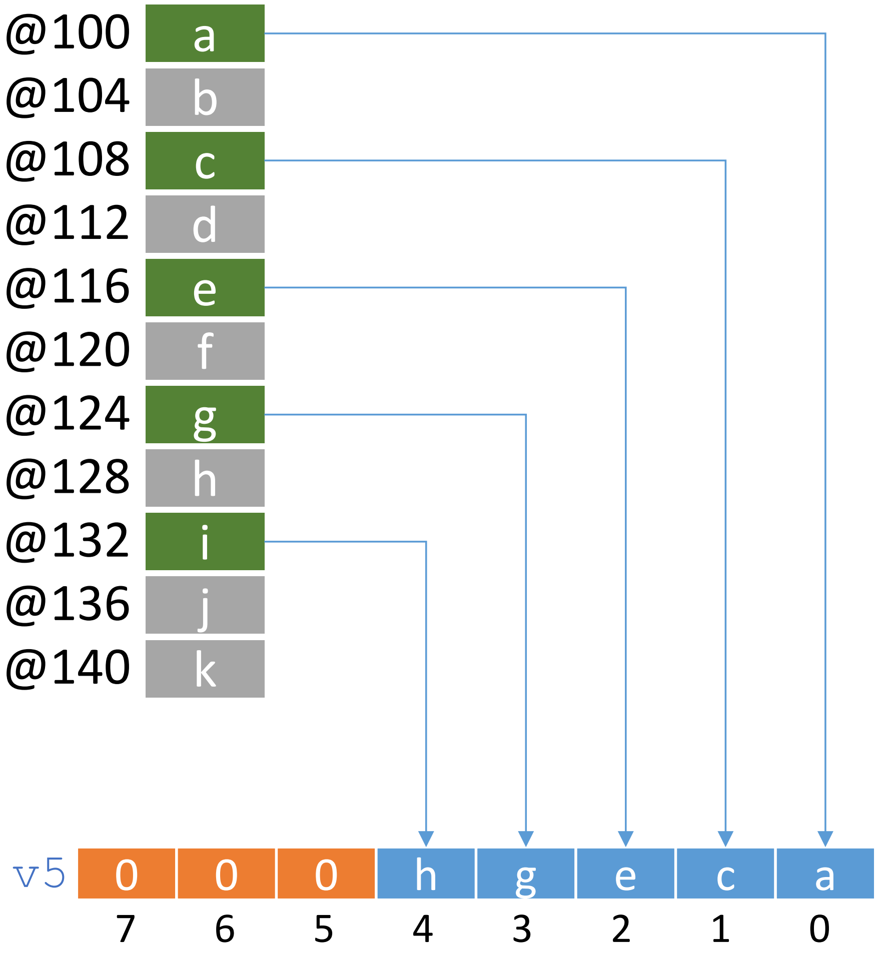
- Unaligned addresses are legal, likely very slow

Strided Vector Load

vlds 80(x3,x9) → v5

```
sz  = sizeof_type(v5);          // 4
tmp = x3 + 80;                  // x3 = 20
for (i = 0; i < vl; i++ )
{
    v5[i] = read_mem(tmp, sz);
    tmp = tmp + x9; // x9 = 8 = stride in bytes

}
for (i = vl; i < MVL; i++ )
{
    v0[i] = 0
}
```

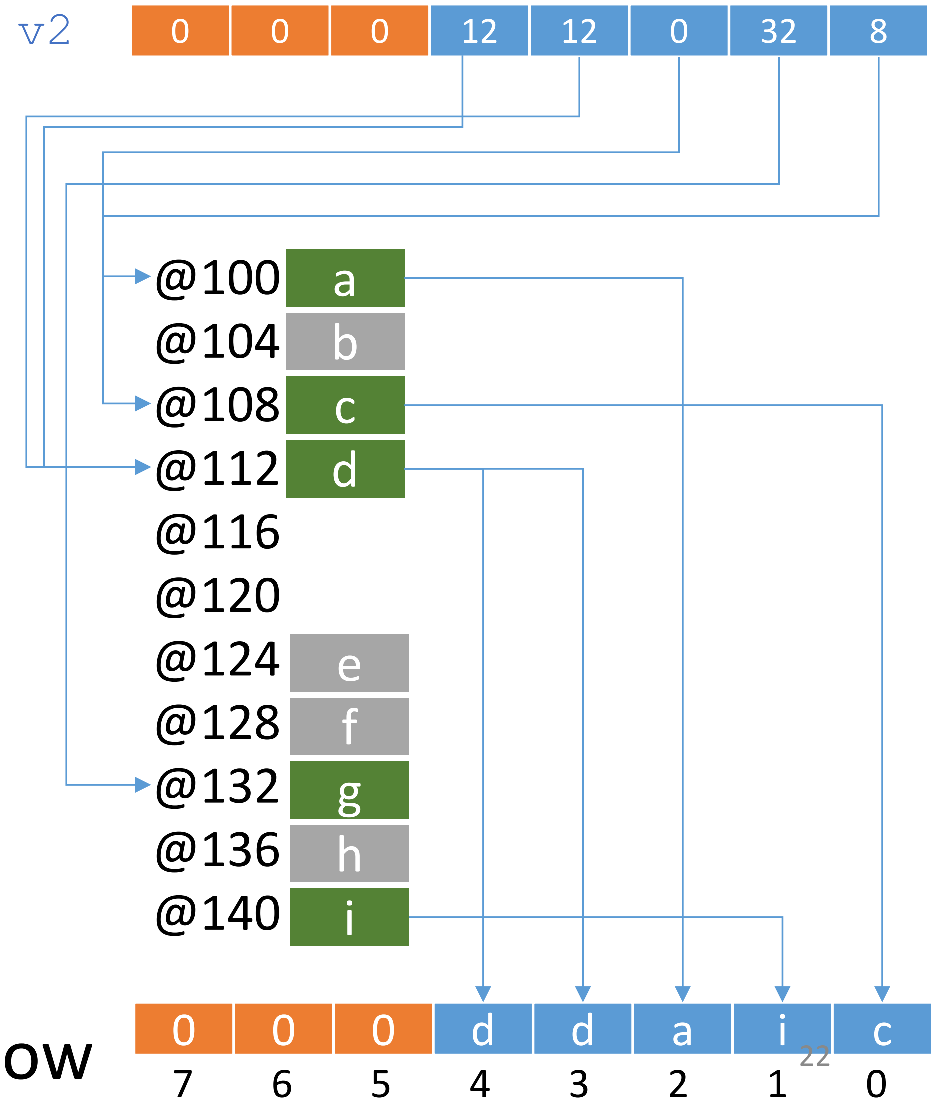


- Stride 0 is legal
- Strides that result in unaligned accesses are legal
  - likely very slow

Gather (indexed vector load)

vldx 80(x3,v2) → v5

```
sz  = sizeof_type(v5);        // 4
tmp = x3 + 80                 // 100
for (i = 0; i < vl; i++ )
{
    addr = tmp + sext(v2[i]);
    v5[i] = read_mem(addr, sz);
}
for (i = vl; i < MVL; i++ )
{
    v0[i] = 0
}
```

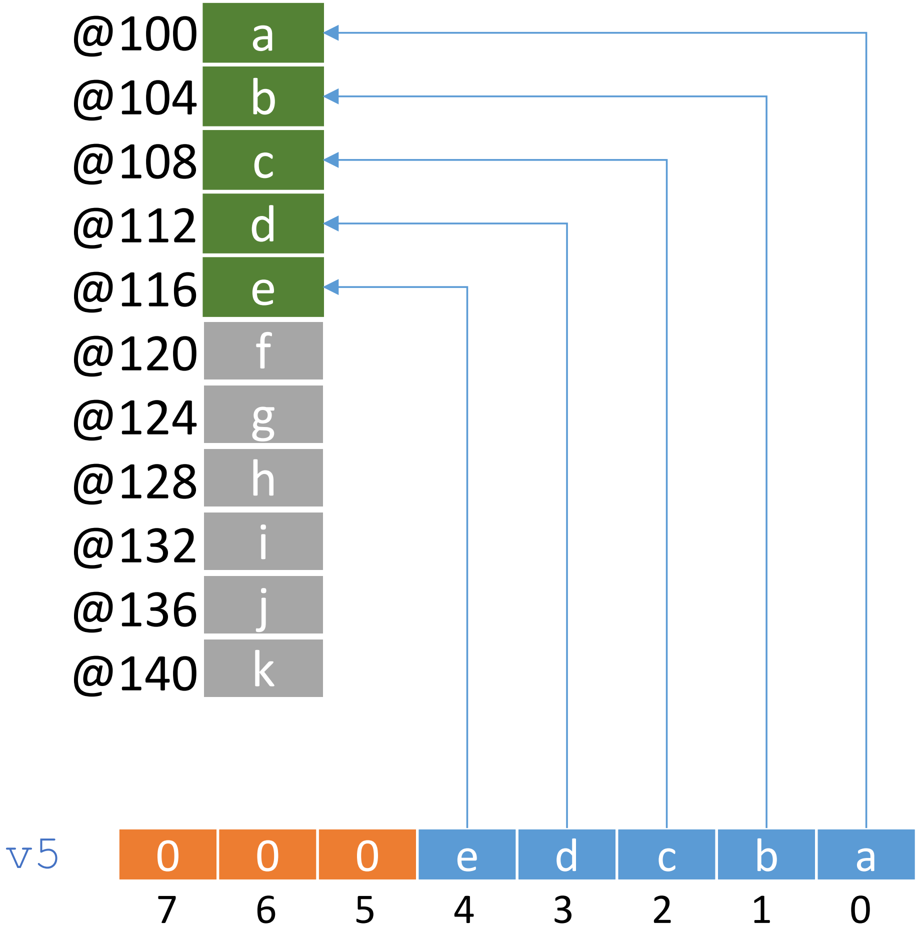


- Repeated addresses are legal
- Unaligned addresses are legal, likely very slow

Vector Store (unit stride)

vst v5 → 80(x3)

```
sz  = sizeof_type(v5);     // 4
tmp = x3 + 80;             // x3 = 20
for (i = 0; i < vl; i++ )
{
    write_mem(tmp, sz, v5[i]);
    tmp = tmp + sz;
}
```

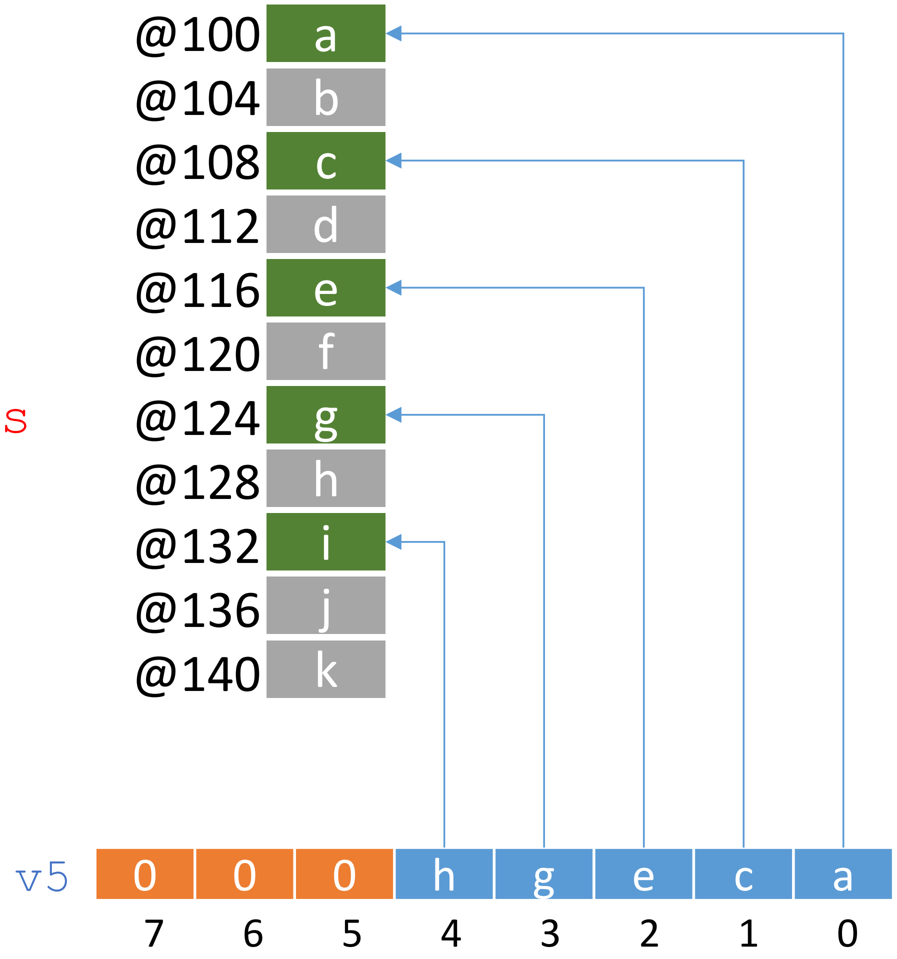


- Unaligned addresses are legal, likely very slow

Strided Vector Store

vsts v5 → 80(x3,x9)

```
// x9 = stride in bytes
sz  = sizeof_type(v5);  // 4
tmp = x3 + 80;                  // x3 = 20
for (i = 0; i < vl; i++ )
{
   write_mem(tmp, sz, v5[i]);
   tmp = tmp + x9; // x9 = 8 = stride in bytes
}
```

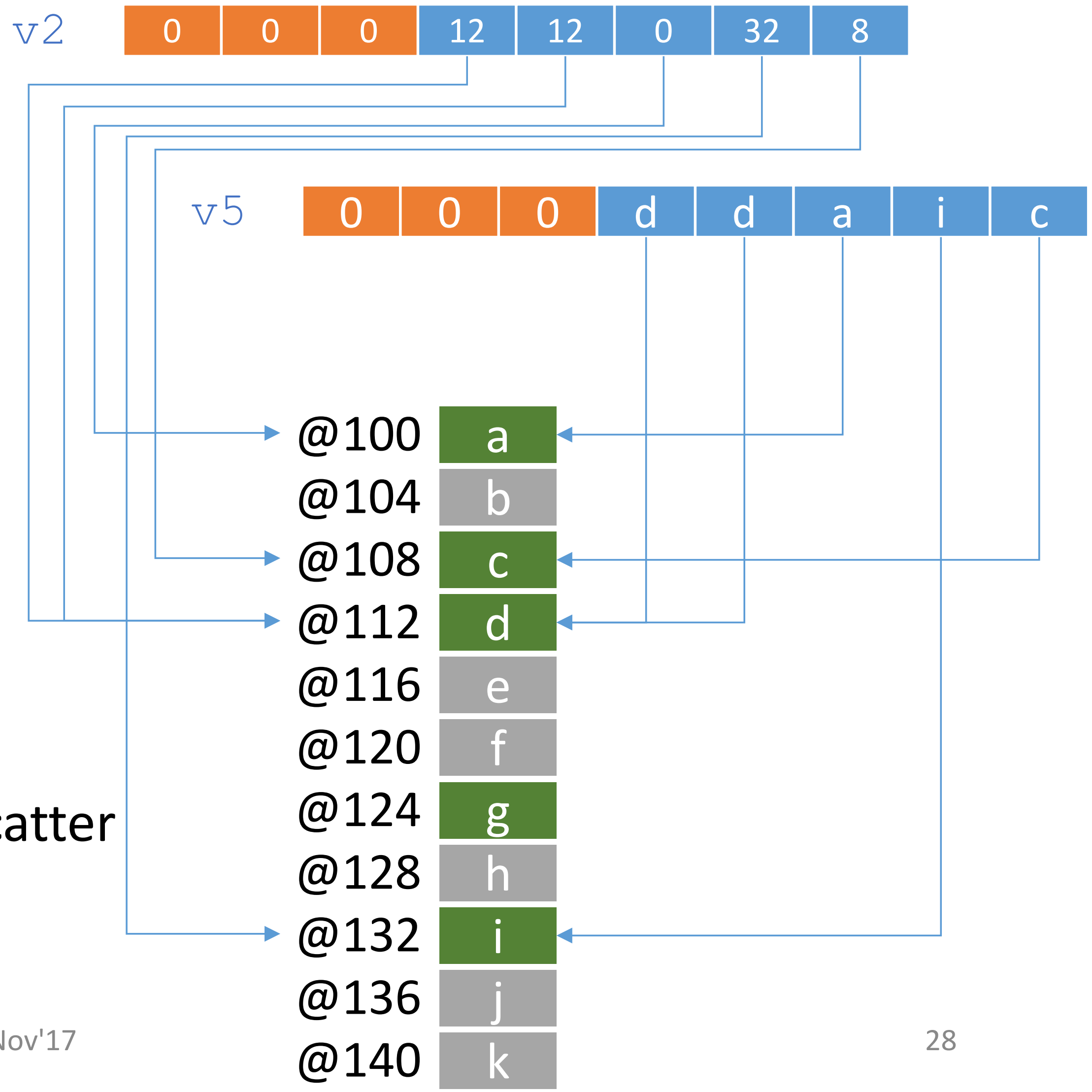


- Stride 0 is legal
- Strides that result in unaligned accesses are legal
  - likely very slow

Scatter (indexed vector store)

vstx v5 → 80(x3,v2)

```
sz  = sizeof_type(v5);     // 4
tmp = x3 + 80;            // 100
for (i = 0; i < vl; i++ )
{
    addr = tmp + sext(v2[i]);
    write_mem(addr, sz, v5[i]);
}
```



- Repeated addresses are legal
    - Provision for both ordered and unordered scatter
- Unaligned addresses are legal
    - likely very slow

Ordering

- From the point of view of a given HART
  - Vector loads & stores instructions happen in order
  - You don’t need any fences to see your own stores
- From the point of view of other HART’s
  - Other harts see the vector memory accesses as if done by a scalar loop
  - So, they can be seen out-of-order by other harts

Typed Vector Registers

Typed Vector Registers

- Each vector register has an associated type
  - Yes, different registers can have different types (i.e., v2 can have type F16 and v3 have type F32)
  - Types can be mixed in an instruction under certain rules
    - Hardware will automatically promote some types to others (see next slide)
  - Types can be dynamically changed by the vcvt instruction
    - If the type change does not required more bits per element than in current configuration
- Rationale for typed registers
  - Register types enable a "polymorphic" encoding for all vector instructions
  - Saves large space of convert from "type A" to "type B"
  - More scalable into the future: Supports custom types without additional encodings
- Supported types depend on the baseline ISA your implementation supports
  - RV32I → I8, U8, I16, U16, I32, U32
  - RV64I → I8, U8, I16, U16, I32, U32, I64, U64
  - RV128I → I8, U8, I16, U16, I32, U32, I64, U64, X128, X128U
  - F →F16, F32
  - FD → F16, F32, F64
  - FDQ → F16, F32, F64, F128
  - Provision for custom type extensions

Type & data conversions: vcvt

- To convert data into a different format
  - Use vcvt between registers of the appropriate type
  - `vcvt` $v1_{F32} \rightarrow v0_{F16}$
  - `vcvt` $v1_{u8} \rightarrow v0_{F32}$
  - `vcvt` $v1_{F32} \rightarrow v0_{I32}$
- Additional feature: changing the dest register type with vcvt
  - `vcvt` $v1_{F32} \rightarrow v0_{F32}$, `I32`
  - Ignores the current dest type, and sets it to the type requested in immediate
  - Legal if requested type size is not bigger than current configured element width

Mixing Types: promoting small into large

- When any source is smaller than dest, that source is “promoted” to dest size
  - If allowed by promotion table. Otherwise, instruction shall trap
- Promotion examples
  - vadd $v1_{I8}$, $v2_{I8}$ → $v0_{I16}$
  - vadd $v1_{I8}$, $v2_{I64}$ → $v0_{I64}$
  - vadd $v1_{F16}$, $v2_{F32}$ → $v0_{F32}$
  - vmadd $v1_{F16}$, $v2_{F16}$, $v3_{F32}$ → $v3_{F32}$
- Table on the right defines valid promotions
  - Zero extend
  - Sign extend
  - Re-bias exponent and pad mantissa with 0’s

se = sign extend
ze = zero extend
p = pass through
rb = re-bias
t = trap

Dest Type	Source Type promotion										
	I64	I32	I16	I8	U64	U32	U16	U8	F64	F32	F16
I64	p	se	se	se	t	ze	ze	ze	t	t	t
I32	t	p	se	se	t	t	ze	ze	t	t	t
I16	t	t	p	se	t	t	t	ze	t	t	t
I8	t	t	t	p	t	t	t	t	t	t	t
U64	t	t	t	t	p	ze	ze	ze	t	t	t
U32	t	t	t	t	t	p	ze	ze	t	t	t
U16	t	t	t	t	t	t	p	ze	t	t	t
U8	t	t	t	t	t	t	t	p	t	t	t
F64	t	t	t	t	t	t	t	t	p	rb	rb
F32	t	t	t	t	t	t	t	t	t	p	rb
F16	t	t	t	t	t	t	t	t	t	t	p

Reconfigurable Vector Register File

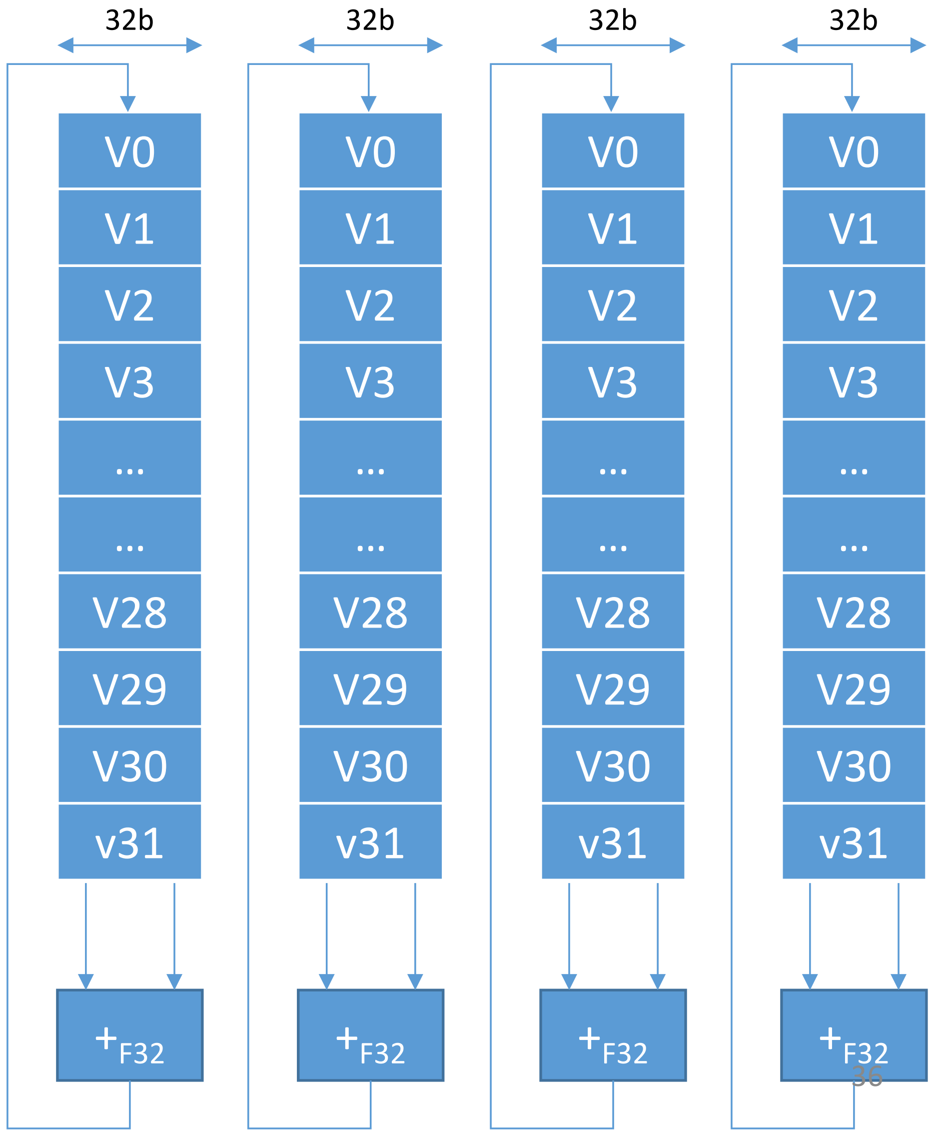
Reconfigurable, variable-length Vector RF

- The vector unit is configured with a `csrrw x1, vdcfg → x2`
  - x1 contains the new configuration indicating
    - Number of logical registers (from 2 to 32)
    - Type for each vector register, using an incremental scheme
  - Hardware resets all vector state to zero
  - Hardware computes Maximum Vector Length (MVL)
    - based on x1 and available vector register file storage
  - MVL returned in x2
  - Can be done in user mode
  - Expected to be fast
- The vector unit is unconfigured writing a 0 to vdcfg
  - Very good to save kernel save & restore!
  - Useful for low power state
- Implementation choices
  - Always return the same MVL, regardless of config
  - Split storage across logical registers, maybe losing some space
  - Pack logical registers as tightly as possible

IMPORTANT: ALL vector registers ALWAYS have the same NUMBER OF ELEMENTS (MVL)

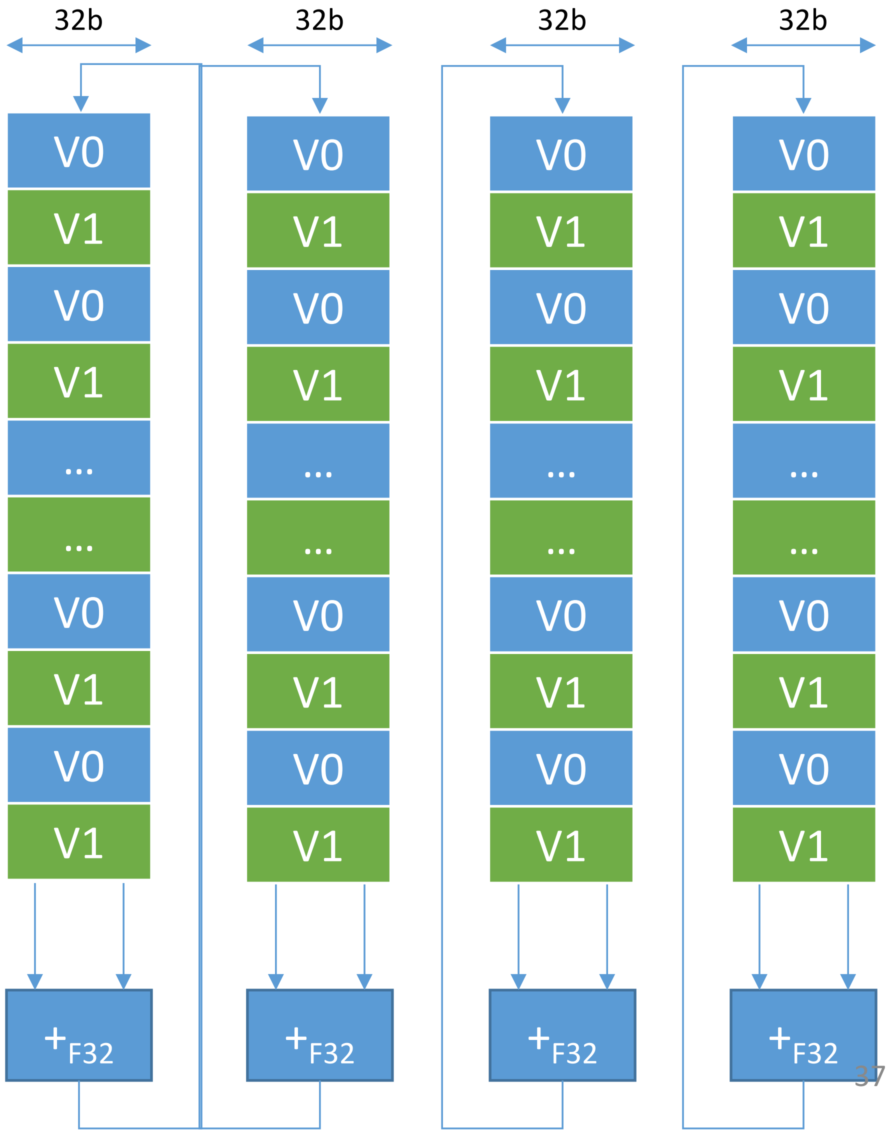
Users asks for 32 F32 registers

- Hardware has 32r x 4e x 4B = 512B
- Need
  - 4 bytes per v0 element
  - 4 bytes per v1 element
  - …
  - 4 bytes per v31 element
- Therefore
  - MVL = 512B / (32 * 4) = 4
- How is the VRF organized?
  - Many possible ways
  - Showing one possible organization



Users asks for only 2 F32 registers

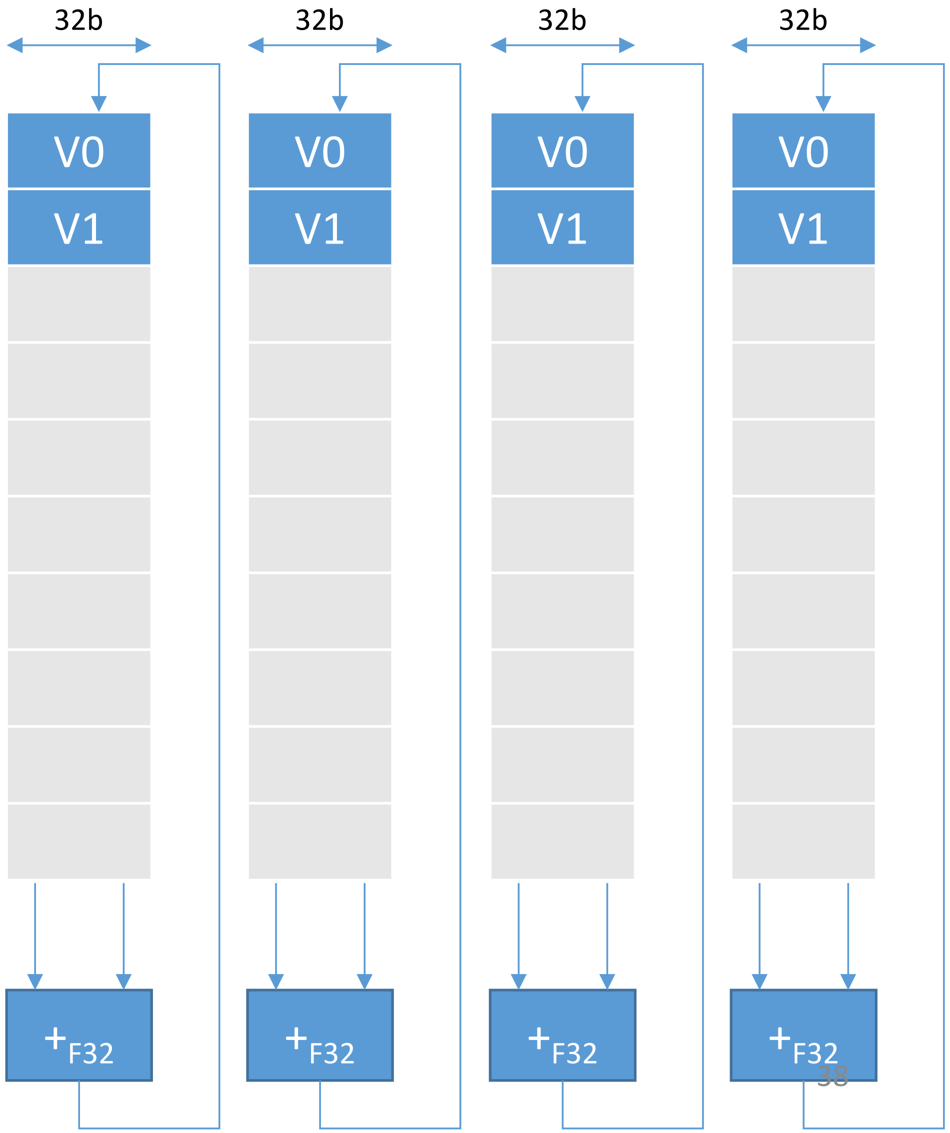
- Hardware has 32r x 4e x 4B = 512B
- Need
  - 4 bytes per v0 element
  - 4 bytes per v1 element
- Therefore
  - MVL = 512B / (4+4) = 64
- How is the VRF organized?
  - Many possible ways
  - Showing an INTERLEAVED organization



Users asks for only 2 F32 registers (also legal!)

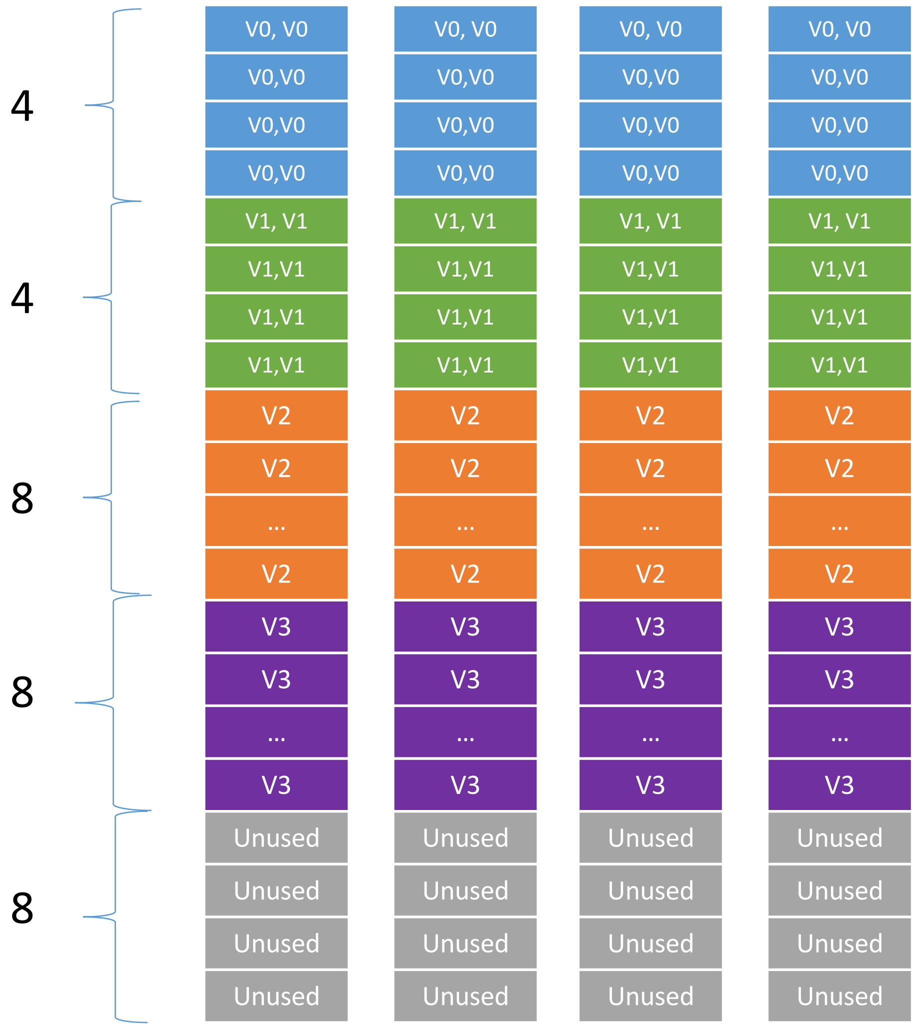
- Hardware has 32r x 4e x 4B = 512B
- Need
  - 4 bytes per v0 element
  - 4 bytes per v1 element
- Therefore
  - MVL = 512B / (4+4) = 64
- And yet, implementation…
  - …answers with MVL = 4
  - Absolutely legal!

- How is the VRF organized?
  - Many possible ways
  - Showing one possible organization



Users asks for 2 F16 regs & 2 F32 regs

- Hardware has 32r x 4e x 4B = 512B
- Need
  - 2 bytes per v0 element
  - 2 bytes per v1 element
  - 4 bytes per v2 element
  - 4 bytes per v3 element
  - 4 'unused bytes' to nearest power of 2
- Therefore
  - MVL = 512B / (12B + 4B) = 32
- How is the VRF organized?
  - Many possible ways
  - Showing one possible organization



MVL is transparent to software!

- Code can be portable across
  - Different number of lanes
  - Different values of MVL
  - If using setvl instruction
- SETVL rs1, rd
  - vl = rs1 > MVL ? MVL : rs1
  - Encoded as csrrw

```
      # Vector-vector 32-bit add loop.
      # Assume vector unit configured with cor
      # a0 holds N
      # a1 holds pointer to result vector
      # a2 holds pointer to first source vecto
      # a3 holds pointer to second source vect
loop: setvl t0, a0
      vld v0, a2        # Load first vector
      sll t1, t0, 2     # multiply by bytes
      add a2, t1        # Bump pointer
      vld v1, a3        # Load second vector
      add a3, t1        # Bump pointer
      vadd v0, v1       # Add elements
      sub a0, t0        # Decrement elements c
      vst  v0, a1       # Store result vector
      add a1, t1        # Bump pointer
      bnez a0, loop     # Any more?
```

Encoding Summary

31 30 29	28	27	26	25	24 23 22 21 20	19 18 17 16 15	14	13	12	11 10 9 8 7	6 5 4 3 2 1 0	
src3			n	sub	src2	src1	3s	m	m	dest	OPCODE	Example
vs3			0	0	vs2	vs1	1	m	m	vd	VOP	vmadd
func6				i	src2	src1	3s	m	m	dest	OPCODE	Example
func6				0	vs2	vs1	0	m	m	vd	VOP	vadd
func6				0	0	vs1	0	m	m	vd	VOP	vsqrt
func6				0	new dest type	vs1	0	m	m	vd	VOP	vcvt
func6				0	rs2	rs1	0	m	m	vd	VOP	vmov.v.x vd[rs2] = rs1
func6				0	rs2	vs1	0	m	m	xd	VOP	vmov.x.v xd = vs1[rs2]
func3	imm			1	imm	vs1	0	m	m	vd	VOP	vaddi
imm			op		src2	src1	op	m	m	dest	OPCODE	Example
imm			0	0	imm	rs1	0	m	m	vd	VMEM	vld
imm			0	0	imm	rs1	1	m	m	vs1	VMEM	vst
imm			0	1	rs2	rs1	0	m	m	vd	VMEM	vlds
imm			0	1	rs2	rs1	1	m	m	vs1	VMEM	vsts
imm			1	0	vs2	rs1	0	m	m	vd	VMEM	vldx
imm			1	0	vs2	rs1	1	m	m	vs1	VMEM	vstx
func3	a	r	1	1	vs2	rs1	1	m	m	vd	VMEM	vamoadd

Not covered today – ask offline

- Exceptions
- Kernel save & restore
- Custom types
  - Crypto WG has a good list of extended types that fit within 16b encoding
  - GFX has additional types
- Matrix shapes (coming soon)
  - Using the same vregs, don't panic!
  - Vadd "matrix", "matrix" → "matrix"
  - Vmul "matrix", "matrix" → "matrix"

Status & Plans

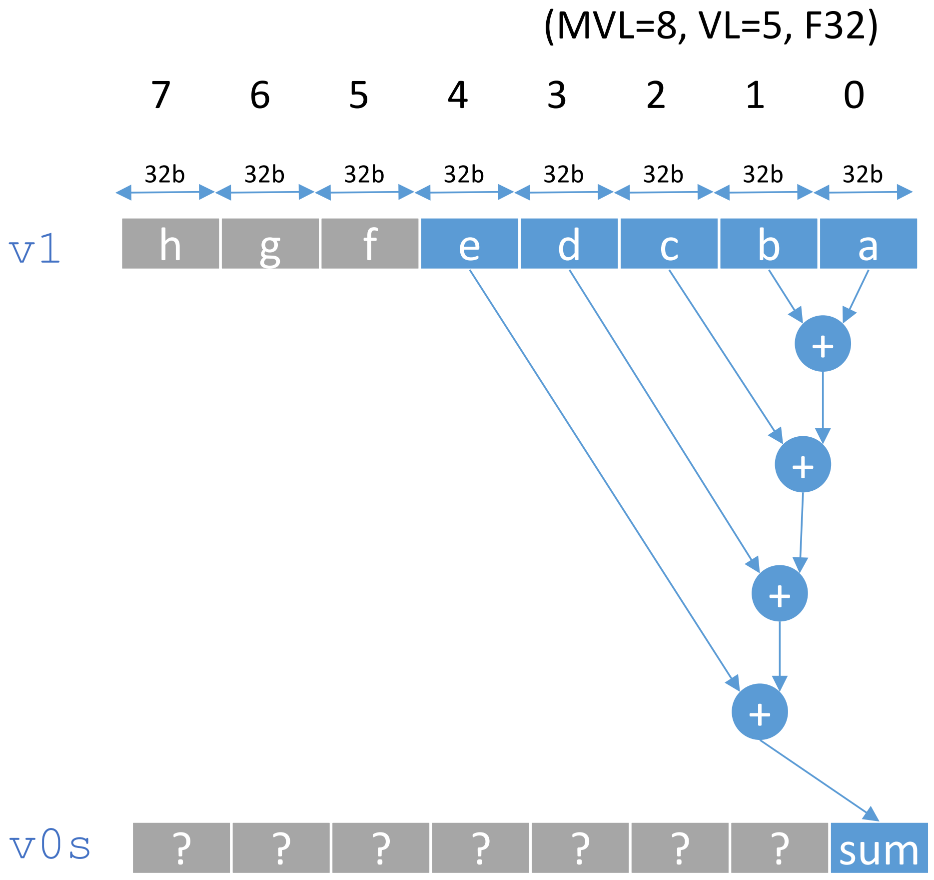
- Best Vector ISA ever! ☺
- Goal is to have spec ready to be ratified by next workshop
  - Week of May 7^{th}, 2018 in Barcelona
- Software
  - Expect LLVM to support it
  - Expect GCC auto-vectorizer to support it
- Please join the vector working group to participate
  - Meeting every 2^{nd} Friday 8am PST
  - Warning: Github spec is out-of-date: WIP to update to this presentation

BACKUP SLIDES

Reductions

vadd v1 → v0.s

```
tmp = 0;
for (i = 0; i < vl; i++ )
{
   tmp = tmp + v1[i]
}
v0[0] = tmp;
```



- Implementations are free to replicate the final "sum" across all elements in the dest vector register

Promotion Table (large font)

		Source Type promotion										
		I64	I32	I16	I8	U64	U32	U16	U8	F64	F32	F16
Dest Type	I64	p	se	se	se	t	ze	ze	ze	t	t	t
	I32	t	p	se	se	t	t	ze	ze	t	t	t
	I16	t	t	p	se	t	t	t	ze	t	t	t
	I8	t	t	t	p	t	t	t	t	t	t	t
	U64	t	t	t	t	p	ze	ze	ze	t	t	t
	U32	t	t	t	t	t	p	ze	ze	t	t	t
	U16	t	t	t	t	t	t	p	ze	t	t	t
	U8	t	t	t	t	t	t	t	p	t	t	t
	F64	t	t	t	t	t	t	t	t	p	rb	rb
	F32	t	t	t	t	t	t	t	t	t	p	rb
	F16	t	t	t	t	t	t	t	t	t	t	p