The RISC-V Vector ISA Tutorial

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

Roger Espasa, roger.espasa@esperantotech.com , Vector WG Co-Chair

Vector Extension Working Group

8th RISC-V Workshop, May, 2018

Part-I: Gentle Introduction

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 (extension)(*)
- Reconfigurable
- Mixed-type instructions
- Common Vector/SIMD programming model
- Fixed-point support
- Easily Extensible
- Best vector ISA ever ☺

Domains

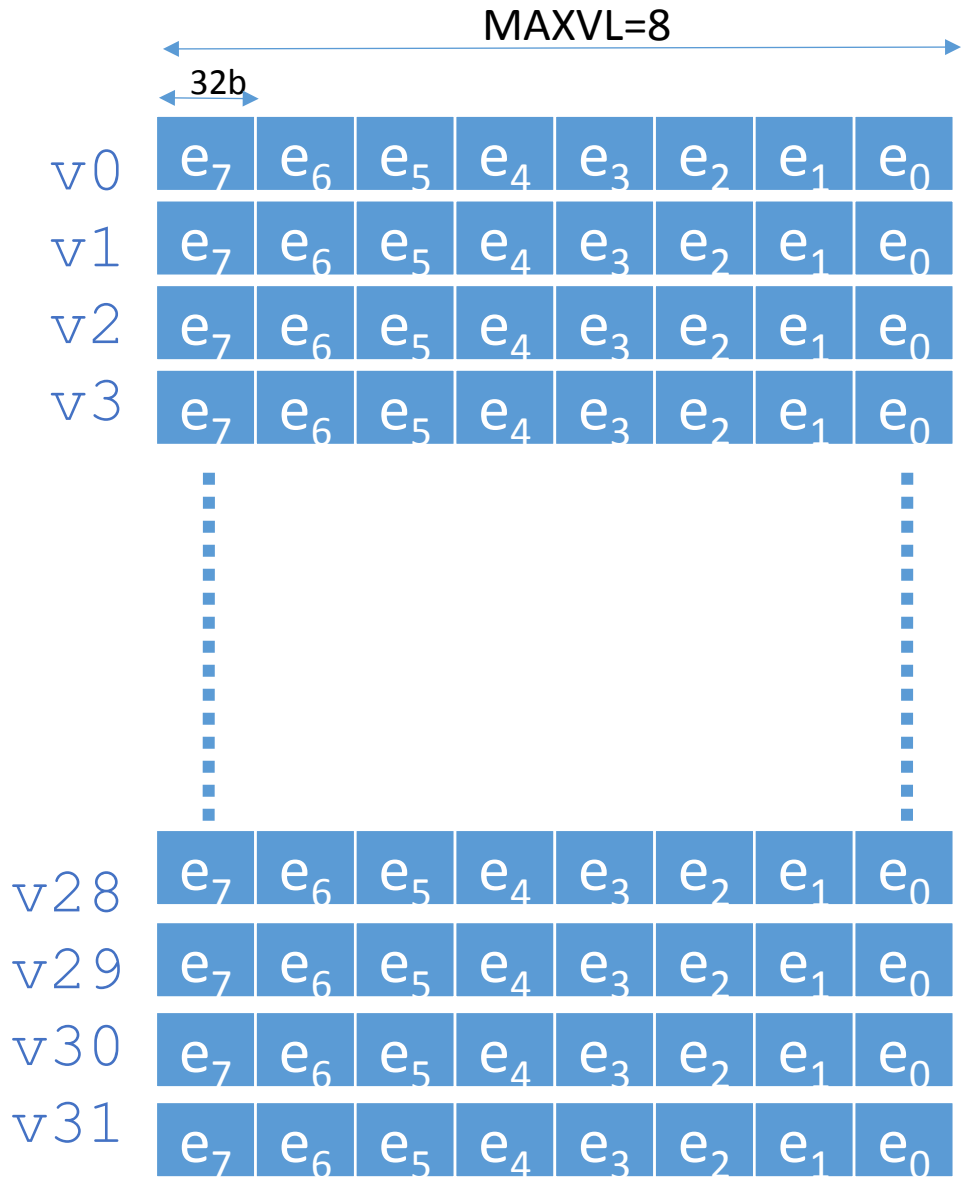
- Machine Learning
- Graphics
- DSP
- Crypto
- Structural analysis
- Climate modeling
- Weather prediction
- Drug design
- And more…

(*)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 can optionally have 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



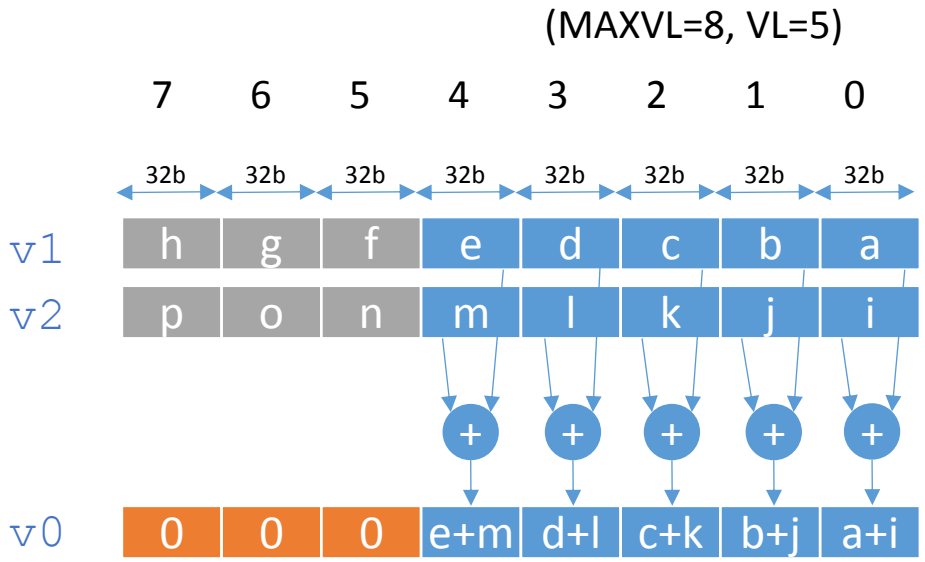
vl	(xlen)
vregmax	(8b)
vemaxw	(3b)
vtypeen	(1b)
vxrm	(2b)
vxcm	(1b)
fcsr.vxsat	(1b)

Note: Floating point flags use the existing scalar flags

Adding two vector registers

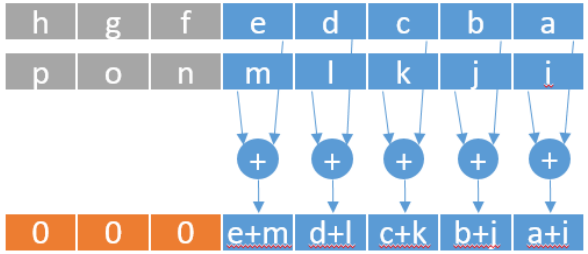
vfadd.s v0, v1, v2

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

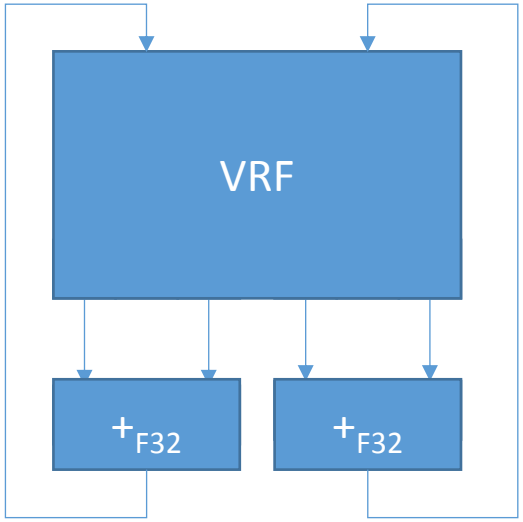


- When VL is zero, dest register is fully cleared
- Operations past ‘vl’ shall not raise exceptions
- Destination can be same as source

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

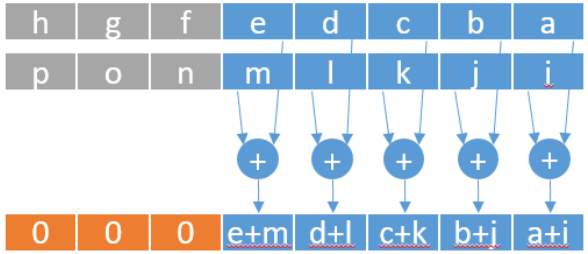


2-lane implementation

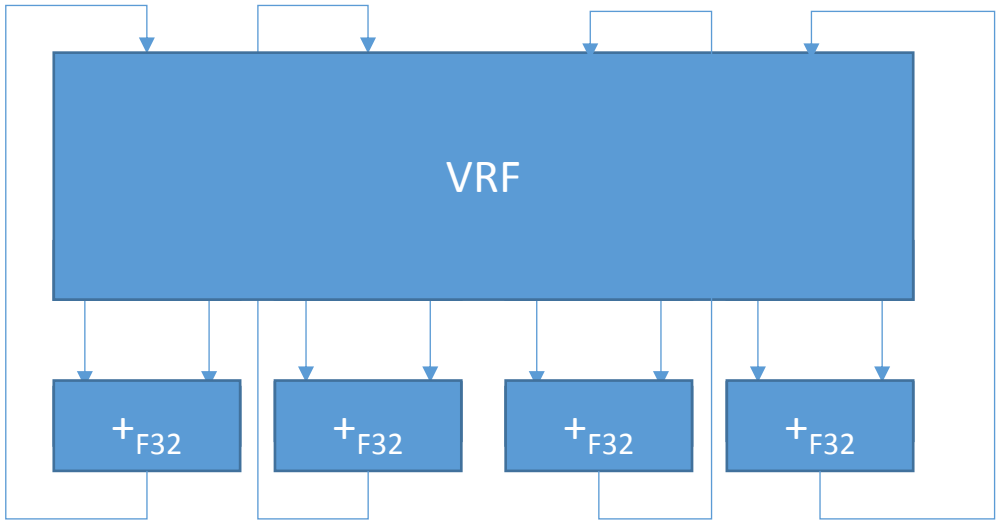


1st clock: a+i, b+j
2nd clock: c+k, d+l
3rd clock: e+m, 0
4th clock: up to you

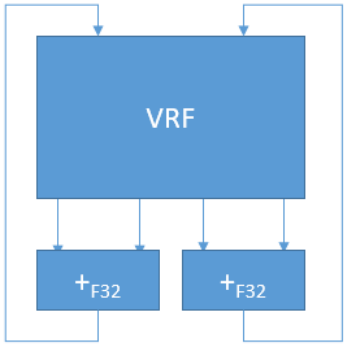
How is this executed? SIMD? Vector? Up to you!



4-lane implementation



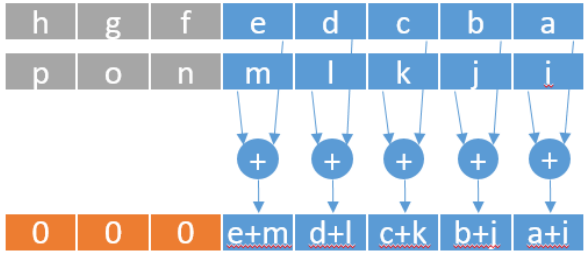
2-lane implementation



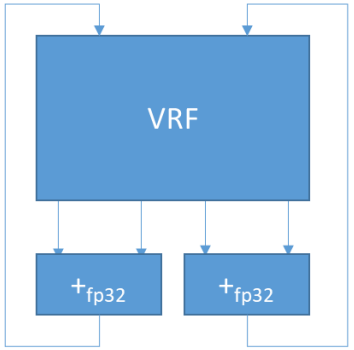
1st clock: a+i, b+j
2nd clock: c+k, d+l
3rd clock: e+m, 0
4th clock: up to you

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!

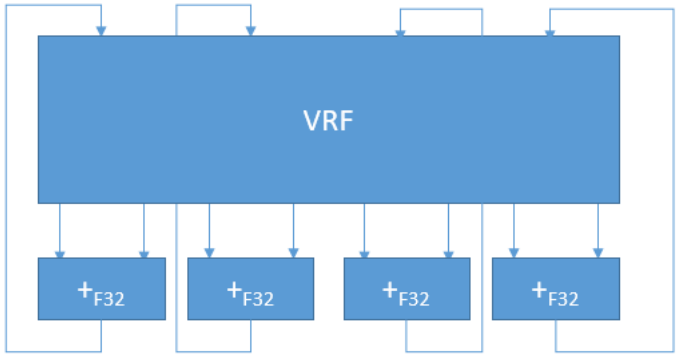


2-lane implementation



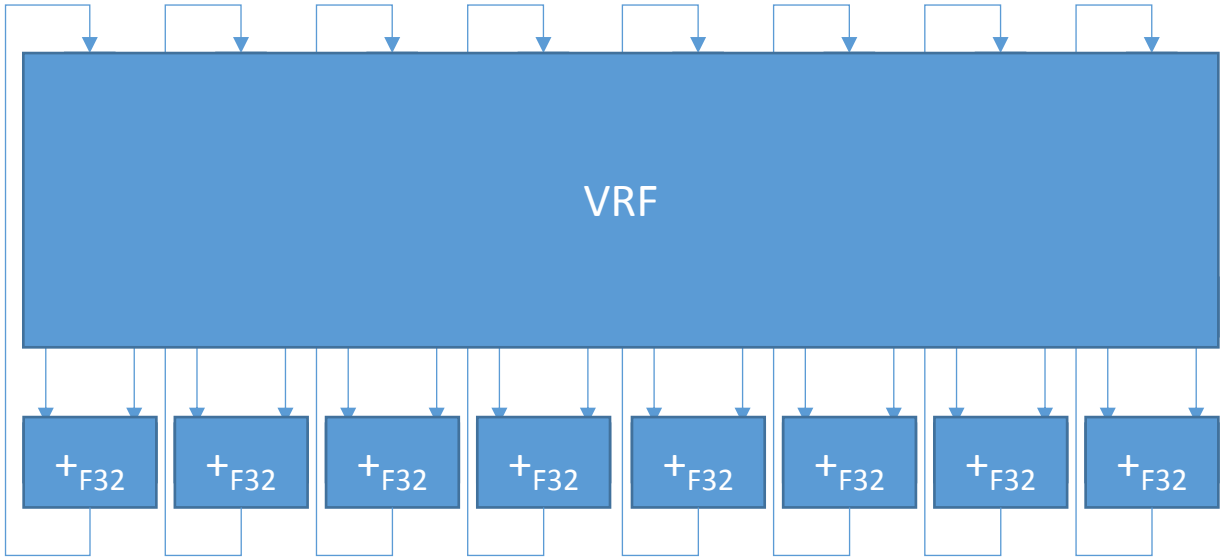
1st clock: a+i, b+j
2nd clock: c+k, d+l
3rd clock: e+m, 0
4th clock: up to you

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)



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

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

Complete Vector Instruction Set

Vector Memory Instructions

operation	instructions
vector load	vlb, vlbu, vlh, vlhu, vlw, vlwu, vld, vflh, vflw, vfld
vector load, strided	vlsb, vlsbu, vlsh, vlshu, vlsw, vlswu, vlsd, vflsh, vflsw, vflsd
vector load, indexed (gather)	vlxb, vlxbu, vlxh, vlxhu, vlxw, vlxwu, vlxd, vflxh, vflxw, vflxd
vector store	vsb, vsh, vsw, vsd
vector store, strided	vssb, vssh, vssw, vssd
vector store, indexed (scatter)	vsxb, vsxh, vsxw, vsxd
vector store, indexed, unordered	vsxub, vsxuh, vsxuw, vsxud

Vector Integer Instructions

operation	instructions
add	vadd, vaddi, vaddw, vaddiw
subtract	vsub, vsubw
multiply	vmul, vmulh, vmulhsu, vmulhu
widening multiply	vmulwdn
divide	vdiv, vdivu, vrem, vremu
shift	vsll, vslli, vsra, vsrai, vsrl, vsrli
logical	vand, vandi, vor, vori, vxor, vxori
compare	vseq, vslt, vsltu
fixed point	vclipb, vclipbu, vcliph, vcliphu, vclipw, vclipwu

Vector Floating Point Instructions

operation	instructions
add	vfadd.h, vfadd.s, vfadd.d
subtract	vfsub.h, vfsub.s, vfsub.d
multiply	vfmul.h, vfmul.s, vfmul.d
divide	vfdiv.h, vfdiv.s, vfdiv.d
sign	vfsgn{j,jn,jx}.h, vfsgn{j,jn,jx}.s, vfsgn{j,jn,jx}.d
max	vfmax.h, vfmax.s, vfmax.d
min	vfmin.h, vfmin.s, vfmin.d
compare	vfeq.h, vfeq.s, vfeq.d, vltq.h, vflt.s, vflt.d, vfle.h, vfle.s, vfle.d
sqrt	vfsqrt.h, vfsqrt.s, vfsqrt.d
class	vfclass.h, vflcass.s, vflcass.d

Vector Floating Point Multiply Add

operation	instructions
add	vfmadd.h, vfmadd.s, vfmadd.d
sub	vfmsub.h, vfmsub.s, vfmsub.d
widening add	vfmaddwdn.h, vfmaddwdn.s, vfmaddwdn.d
widening sub	vfmsubwdn.h, vfmsubwdn.s, vfmsubwdn.d

Vector Convert

From Integer to Float	
To Half	vfcvt.h.i, vfcvt.h.u
To Single	vfcvt.s.i, vfcvt.s.u
To Double	vfcvt.d.i, vfcvt.d.u
From Float to Vemaxw Integer	
To Signed	vfcvt.i.h, vfcvt.i.s, vfcvt.i.d
To Unsigned	vfcvt.u.h, vfcvt.u.s, vfcvt.u.d
From Float to Float	
To Half	vfcvt.h.s, vfcvt.h.d
To Single	vfcvt.s.h, vfcvt.s.d
To Double	vfcvt.d.h, vfcvt.d.s

From Vemaxw Int to Narrow Int	
To Byte	vcvt.{b,bu}.i
To Half	vcvt.{h,hu}.i
To Word	vcvt.{w,wu}.i

Vector Data Movement

operation	instructions	action
insert gpr into vector	vins vd, rs1, rs2	vd[rs2] = rs1
insert fp into vector	vins vd, fs1, rs2	vd[rs2] = fs1
extract velem to gpr	vext rd, vs1, rs2	rd = vs1[rs2]
extract velem to fp	vext fd, vs1, rs2	fd = vs1[rs2]
vector-vector merge	vmerge vd, vs1, vs2, vm	mask picks src
vector-gpr merge	vmergex vd, rs1, vs2, vm	mask picks src
vector-fp merge	vmergef vd, fs1, vs2, vm	mask picks src
vector register gather	vrgather vd, vs1, vs2, vm	vd[i] = vs1[vs2[i]]
Gpr splat/bcast	vsplatx vd, rs1	Vd[0..MAXVL] = rs1
fpr splat/bcast	vsplatf vd, fs1	Vd[0..MAXVL] = fs1
vector slide down	vslidedwn vd, vs1, rs2, vm	vd[i] = vs1[rs2+i]
vector slide up	vslideup vd, vs1, rs2, vm	vd[rs2+i] = vs1[i]

Vector Mask Operations

operation	instructions
Find first set bit in mask	Vmfirst rd, vs1
Mask pop count	Vmpopc rd, vs1
Count preceding mask bits	Vmiota vd, vm
Flag before first	Vmfbf vd, vs1, vm
Flag including first	Vmfif vd, vs1, vm

Let’s vectorize a simple loop using RISC-V’s vector ISA

Loop #1: Add vector and scalar

```
float a;
float vec[1714];
for (i = 0; i < 1714; i++)
{
  vec[i] = vec[i] + a;
}
```

We need to understand how to…

1. Initialize the vector unit
2. Set the ‘vl’ register
3. Load a portion of ‘vec’ into a vector register v0
4. Load scalar constant ‘a’ into a vector register v1
5. Add v0 and v1, result going into v2
6. Store back v2 to memory
7. Loop back until we’re done

```
float a;
float vec[1714];
for (i = 0; i < 1714; i++)
{
  vec[i] = vec[i] + a;
}
```

Initializing the vector unit

Vector Unit Initialization

- Before using the vector unit, we must “configure” it
  - Must tell hardware how many vector registers we want to enable
    - From 2 to 32 currently
  - Must indicate the Maximum Element Width across all vectors
    - 01 – 8 bit data
    - 10 – 16 bit data
    - 11 – 32 bit data
    - 00 – 64 bit data
- Configuration info kept in different WARL fields in the vcfg CSR
  - vregmax (8b) :holds highest numbered (-1) enabled register. When 0, vector unit disabled
  - vemaxw (3b): holds maximum element width
  - vtypeen (1b): enable type extension
  - Vxcm (1b): fixed point clip mode
  - Vxrm (2b): fixed point rounding mode

Vconfig imm (*)

(*) Exact field and imm encoding TBD

- Fast instruction to configure/disable vector unit
- Vconfig 0 → disables the vector unit
- Vconfig imm
  - Imm[4:0] = Highest numbered vector register (-1) enabled
    - If 0, vector unit disabled
    - If 1, 2 vector registers, v0 and v1, enabled
    - If 2, 3 vector registers, v0, v1 and v2, enabled
    - If 31, 32 vector registers, v0..v31, enabled
  - Imm[6:5] = Maximum Element Width across all vectors
    - 01 – 8 bit data
    - 10 – 16 bit data
    - 11 – 32 bit data
    - 00 – 64 bit data
  - Imm[7] = Fixed point clip mode
  - Fixed point rounding mode always set to ’10 (RNE). Can be changed with CSRRSI
- Vconfig automatically sets vl to the MAXVL available in the machine

We need to understand how to…

1. Initialize the vector unit: 4 vregs, 32b data
   - vconfig 0x63
2. Set the ‘vl’ register
3. Load a portion of ‘vec’ into a vector register v0
4. Load scalar constant ‘a’ into a vector register v1
5. Add v0 and v1, result going into v2
6. Store back v2 to memory
7. Loop back until we’re done

```
float a;
float vec[1714];
for (i = 0; i < 1714; i++)
{
  vec[i] = vec[i] + a;
}
```

Using the ‘vl’ register

vsetvl rd, rs1

- Sets vl based on the current vector configuration and the value in rs1 treated as an unsigned integer, and also writes this value to rd.
- Hardware will guarantee that the vl setting must be:
  1. greater than 0, if rs1 is greater than 0
  2. monotonically increasing with the value in rs1, but need not be strictly increasing
  3. bounded above by min(rs1,MAXVL)
  4. deterministic for any given configuration
- If the vector unit is disabled, vsetvl or any read or write of vl will raise an illegal instruction exception.
- The vsetvl instruction is not encoded as a regular CSRRW instruction as the value returned depends on the input value, but regular CSR instructions can be used to read and write vl.
  - The value written to vl on a CSR write is capped at MAXVL (vl is WARL)
- In our example

We need to understand how to…

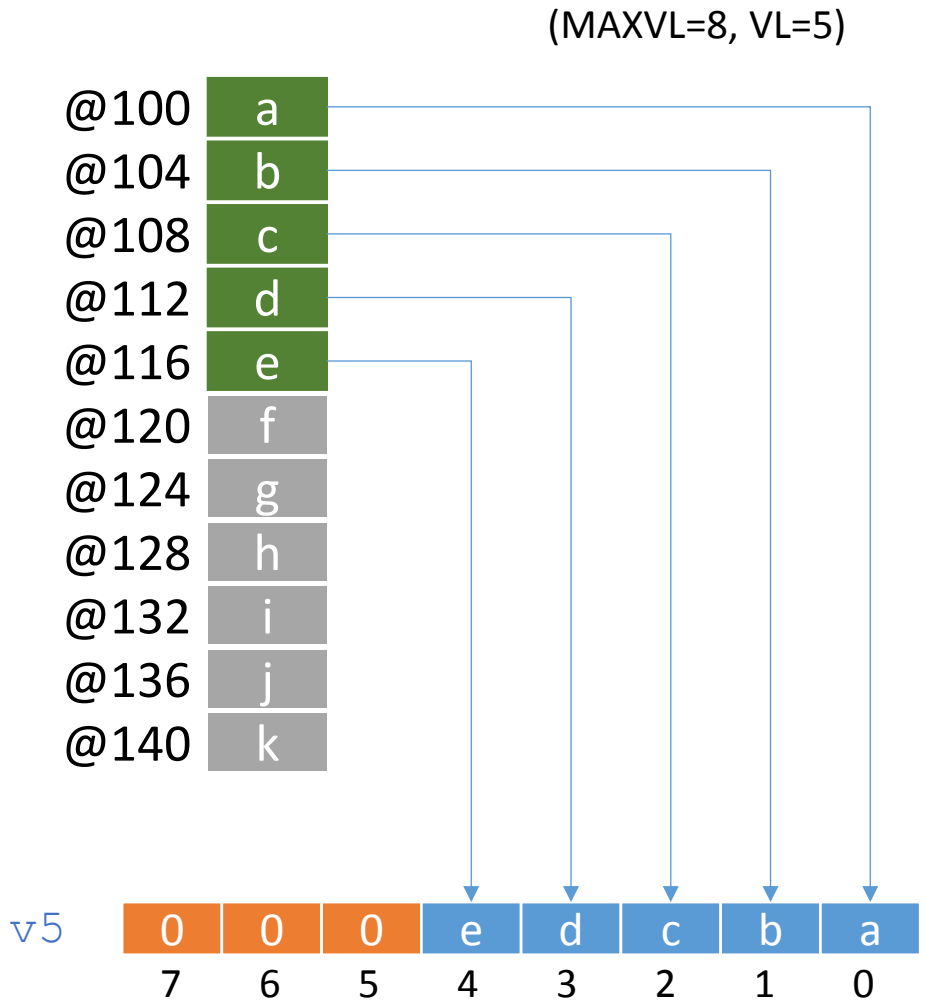
1. Initialize the vector unit: 4 vregs, 32b data
   - vconfig 0x63
2. Set the ‘vl’ register
   - addi x1, x0, 1714
   - vsetvl x2, x1 // will set vl and x2 both to min(MAXVL, 1714)
3. Load a portion of ‘vec’ into a vector register v0
4. Load scalar constant ‘a’ into a vector register v1
5. Add v0 and v1, result going into v2
6. Store back v2 to memory
7. Loop back until we’re done

```
float a;
float vec[1714];
for (i = 0; i < 1714; i++)
{
  vec[i] = vec[i] + a;
}
```

Vector Load (unit stride)

vflw v5,80(x3)

```
sz = 4   // based on opcode vflw
tmp = x3 + 80;                  // x3 = 20
for (i = 0; i < vl; i++ )
{
   v5[i] = read_mem(tmp, sz);
   tmp = tmp + sz;
}
for (i = vl; i < MAXVL; i++ )
{
   v5[i] = 0
}
```



- Unaligned addresses are legal, likely very slow

We need to understand how to…

1. Initialize the vector unit: 4 vregs, 32b data
   - vconfig 0x63
2. Set the ‘vl’ register
   - addi x1, x0, 1714
   - vsetvl x2, x1 // will set vl and x2 both to min(MAXVL, 1714)
3. Load a portion of ‘vec’ into a vector register v0
   - Assume x3 contains address of ‘vec’
   - vflw v0, 0(x3) // will load ‘VL’ elements out of ‘vec’
4. Load scalar constant ‘a’ into a vector register v1
5. Add v0 and v1, result going into v2
6. Store back v2 to memory
7. Loop back until we’re done

```
float a;
float vec[1714];
for (i = 0; i < 1714; i++)
{
  vec[i] = vec[i] + a;
}
```

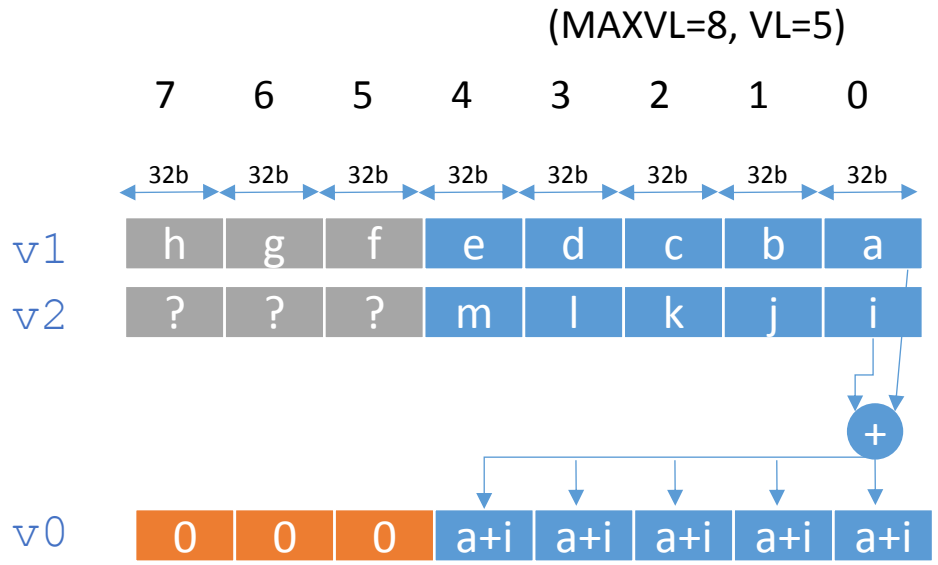
Scalar support

Scalar support

- The data inside a VREG can have 3 possible shapes:
  - A vector (i.e., what you'd expect)
  - A single scalar value (or, equivalently, the same value in all vector elements)
  - A matrix (optional, not in the base spec)
- Base ISA only supports vector shape
- Base ISA does offer some support for scalar shape
  - In a compatible/extensible manner with type+shape extension
- In the base ISA: vop vd.s, vs1, vs2
  - vd[*] = vs1[0] op vs2[0]
  - The dest vreg of an operation can be marked as 'scalar'
  - Only one operation performed: vs1[0] op vs2[0]
  - Result is copied/broadcast to all positions of vd (up until vl)

vfadd.s v0.s, v1, v2

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



- Implementations are free to replicate the scalar value across all elements in the vector register or microarchitecturally “remember” that v0 contains a single scalar value. The architectural visible result must be equivalent whichever implementation is chosen.

Adding a vector and a scalar

- Vector data located in v1, final result in v0
- A) Load scalar value from mem, replicate across vector, then add
  - vfld.s v2.s, 80(r5)
  - vfadd.s v0, v1, v2
- B) Move f0 to vreg using vmergef, then add
  - fld.s f0, 80(r5)
  - vmergef v2, f0, v2 # with mask=all true, will select f0
  - vfadd.s v0, v1, v2

We need to understand how to…

1. Initialize the vector unit: 4 vregs, 32b data
   - vconfig 0x63
2. Set the 'vl' register
   - addi x1, x0, 1714
   - vsetvl x2, x1 // will set vl and x2 both to min(MAXVL, 1714)
3. Load a portion of 'vec' into a vector register v0
   - Assume x3 contains address of 'vec'
   - Vflw v0, 0(x3) // will load 'VL' elements out of 'vec'
4. Load scalar constant 'a' into a vector register v1
   - Assume x4 contains address of 'a'
   - vfld.s v2.s, 0(x4)
5. Add v0 and v1, result going into v2
6. Store back v2 to memory
7. Loop back until we're done

```
float a;
float vec[1714];
for (i = 0; i < 1714; i++)
{
  vec[i] = vec[i] + a;
}
```

We need to understand how to…

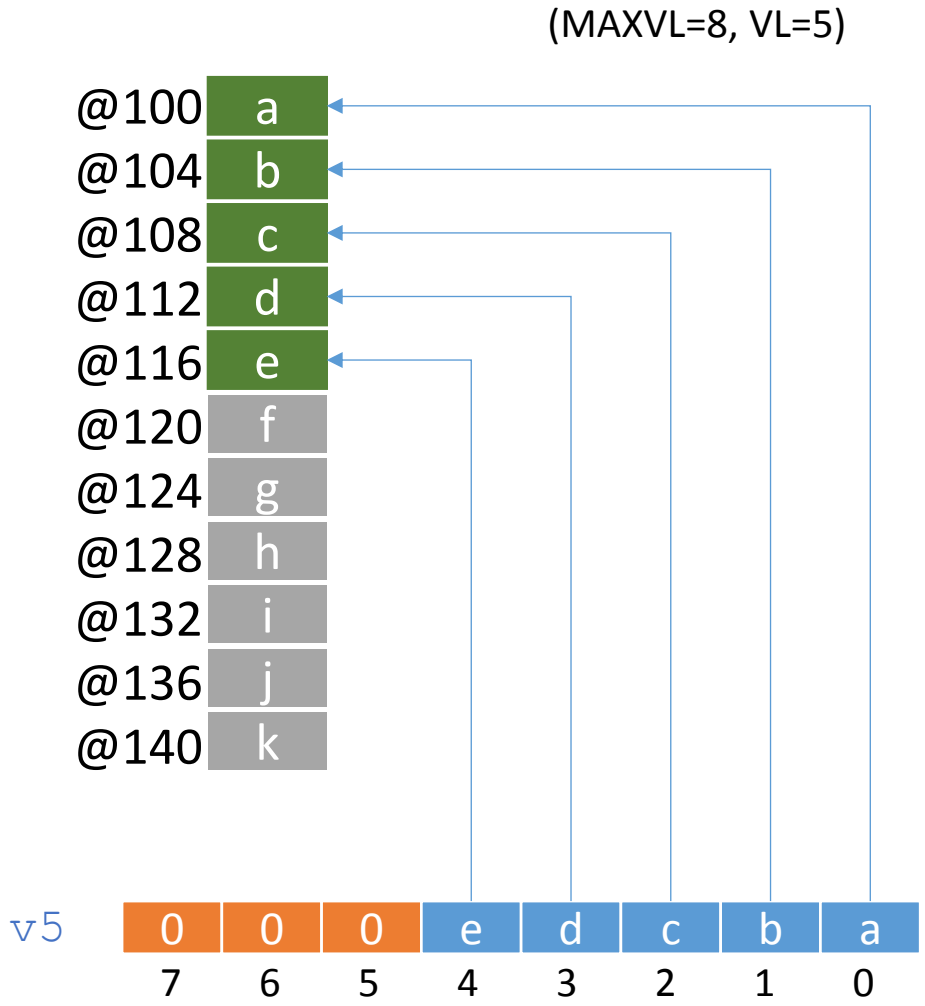
1. Initialize the vector unit: 4 vregs, 32b data
   - vconfig 0x63
2. Set the 'vl' register
   - addi x1, x0, 1714
   - vsetvl x2, x1 // will set vl and x2 both to min(MAXVL, 1714)
3. Load a portion of 'vec' into a vector register v0
   - Assume x3 contains address of 'vec'
   - Vflw v0, 0(x3) // will load 'VL' elements out of 'vec'
4. Load scalar constant 'a' into a vector register v1
   - Assume x4 contains address of 'a'
   - vfld.s v1.s, 0(x4)
5. Add v0 and v1, result going into v2
   - vfadd.s v2, v1, v0
6. Store back v2 to memory
7. Loop back until we're done

```
float a;
float vec[1714];
for (i = 0; i < 1714; i++)
{
  vec[i] = vec[i] + a;
}
```

Vector Store (unit stride)

vsw v5,80(x3)

```
sz  = 4;  // Based on vsw opcode
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

We need to understand how to…

1. Initialize the vector unit: 4 vregs, 32b data
   - vconfig 0x63
2. Set the 'vl' register
   - addi x1, x0, 1714
   - vsetvl x2, x1 // will set vl and x2 both to min(MAXVL, 1714)
3. Load a portion of 'vec' into a vector register v0
   - Assume x3 contains address of 'vec'
   - Vflw v0, 0(x3) // will load 'VL' elements out of 'vec'
4. Load scalar constant 'a' into a vector register v1
   - Assume x4 contains address of 'a'
   - vfld.s v1.s, 0(x4)
5. Add v0 and v1, result going into v2
   - vfadd.s v2, v1, v0
6. Store back v2 to memory
   - vsw v2, 0(x3)
7. Loop back until we're done

```
float a;
float vec[1714];
for (i = 0; i < 1714; i++)
{
  vec[i] = vec[i] + a;
}
```

Final loop

```
float a;
float vec[1714];
for (i = 0; i < 1714; i++)
{
  vec[i] = vec[i] + a;
}
```

```
        // assume x1 contains constant 1714
        // assume x3 contains address of 'vec'
        // assume x4 contains address of 'a'
        vconfig  0x63 // 4 vregs, 32b data
        vfld.s  v1.s, 0(x4)   // load 'a' constant and bcst into v1
loop:
        vsetvl x2, x1         // will set vl and x2 both to min(maxvl, x1)
        vflw v0, 0(x3)        // will load 'vl' elements out of 'vec'
        vfadd.s v2, v1, v0    // do the add
        vsw v2, 0(x3)         // store result back to 'vec'
        slli x5, x2, 2        // bytes consumed from 'vec' (x2 * sizeof(float))
        add x3, x3, x5        // increment 'vec' pointer
        sub x1, x1, x2        // subtract from total (x1) work done this iteration (x2)
        bne x1, x0, loop      // if x1 not yet zero, still work to do
        vconfig  0x0          // optional, DISABLE vector unit
```

Part-II: The rest of the spec

Masked execution

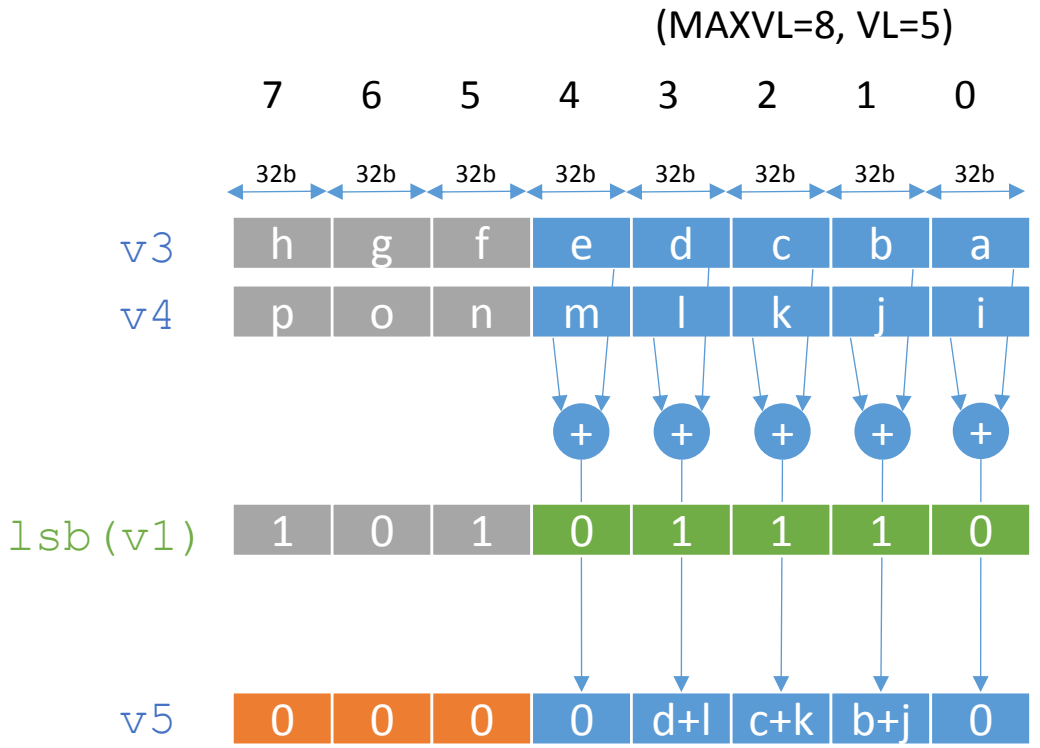
Masked execution

- 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, vslt, vsltu)
  - veq v6, v7 → v1
  - Comparison results are integer “0” or “1”
  - Encoded with as many bits as the destination register element size
  - Other compare operations can be realized using the opposite mask encoding
- Instructions use 2 bits of encoding to select masked execution
  - 00 : Destination register is scalar
  - 01 : Destination register is vector, operation unmasked
  - 10 : Use v0’s elements lsb as the mask
  - 11 : Use ~v0’s elements lsb as the mask

vfadd.s v5, v3, v4, v0.t

```
for (i = 0; i < vl; i++ )
{
   v5[i] = lsb(v0[i]) ? v3[i] +F32 v4[i] : 0;
}
for (i = vl; i < MAXVL; i++ )
{
   v5[i] = 0
}
```

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

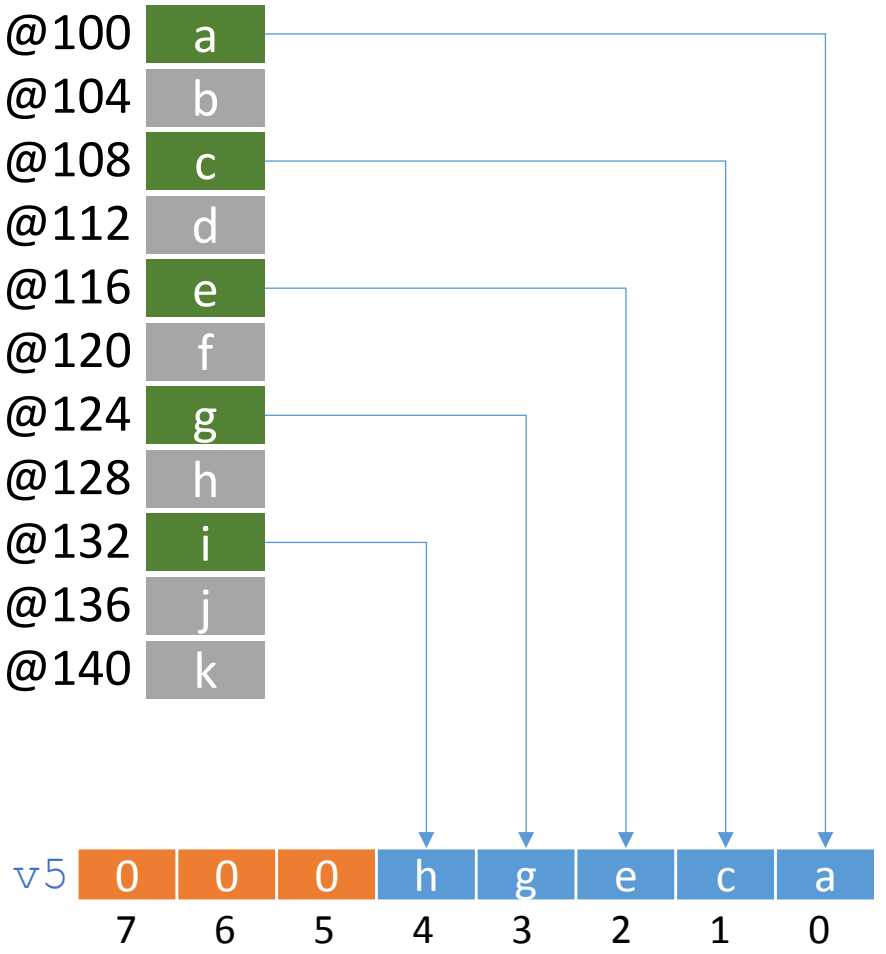


Strided Vector Load

vlsw v5,80(x3,x9)

```
sz  = 4;              // based on opcode vlsw
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 < MAXVL; i++ )
{
    v5[i] = 0
}
```

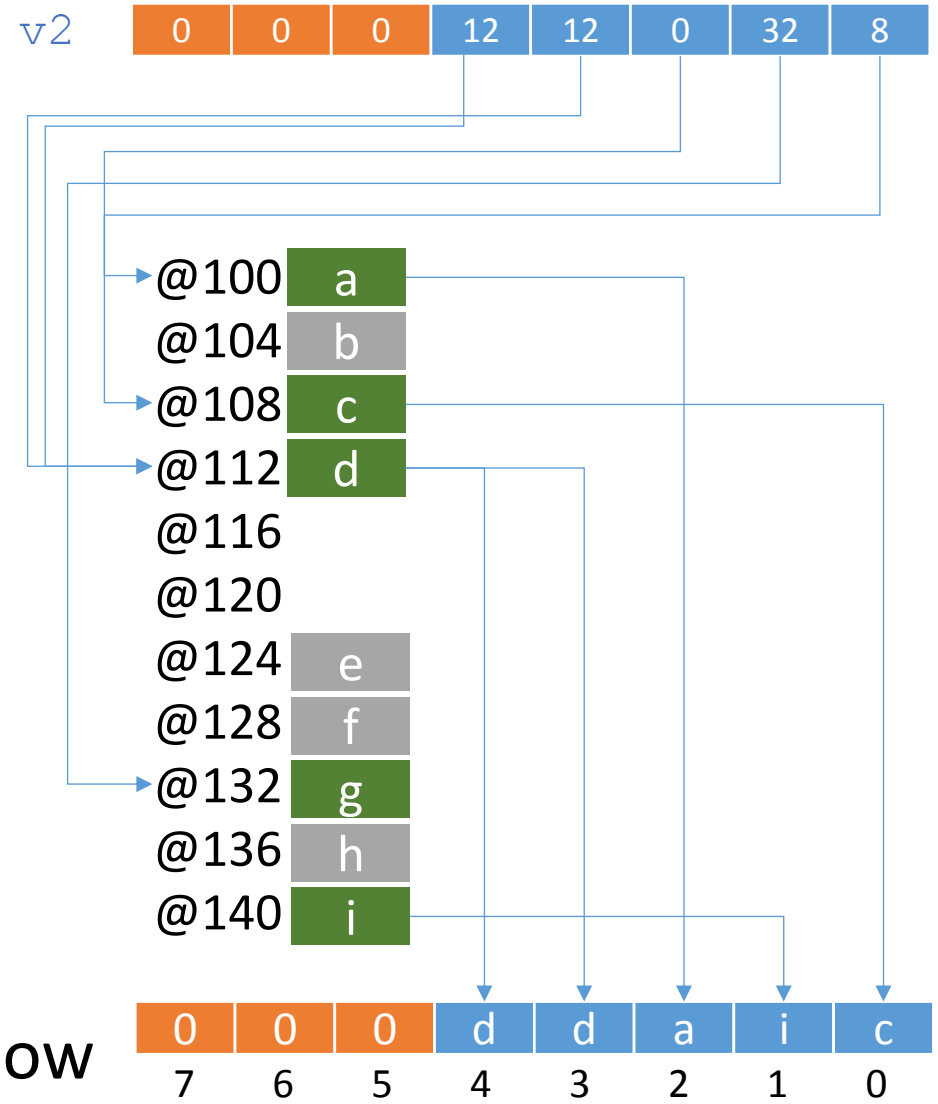


- Stride 0 is legal
- Strides that result in unaligned accesses are legal
  - likely very slow

Gather (indexed vector load)

vflxw v5, 80(x3,v2)

```
sz  = 4;    // based on opcode vflxw
tmp = x3 + 80    // x3 = 20
for (i = 0; i < vl; i++ )
{
   addr = tmp + sext(v2[i]);
   v5[i] = read_mem(addr, sz);
}
for (i = vl; i < MAXVL; i++ )
{
   v5[i] = 0
}
```

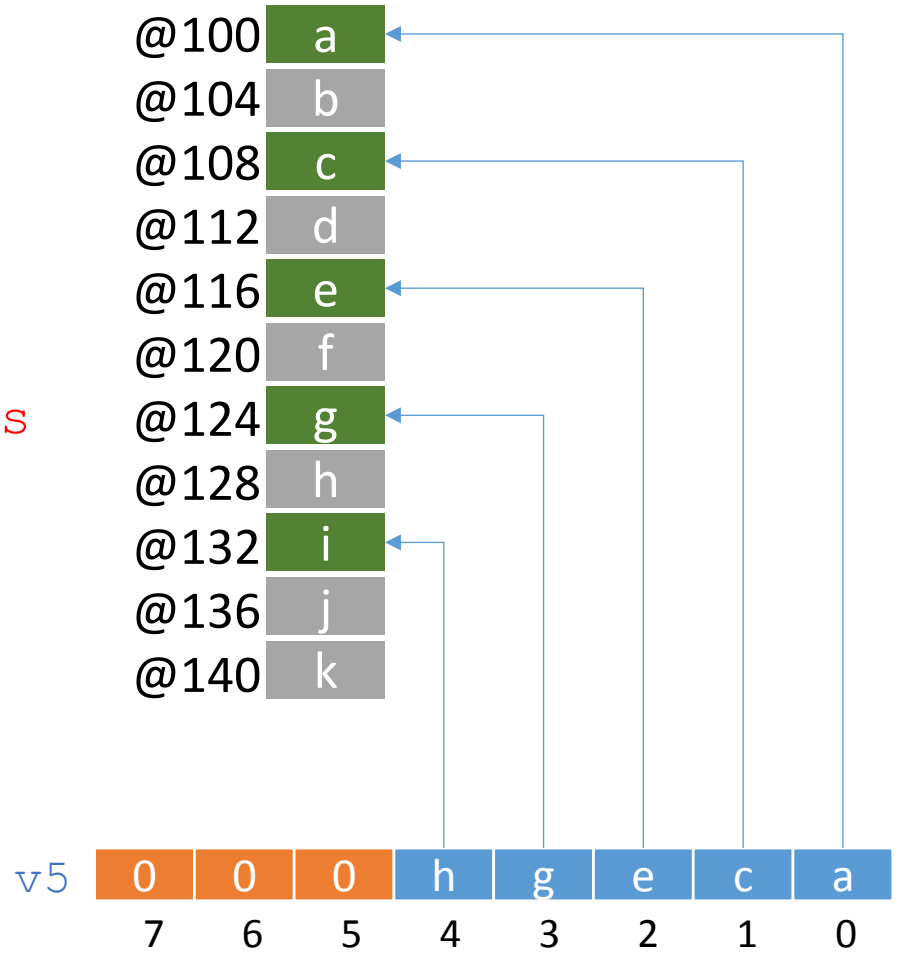


- Repeated addresses are legal
- Unaligned addresses are legal, likely very slow

Strided Vector Store

vssw v5,80(x3,x9)

```
// x9 = stride in bytes
sz  = 4;              // based on opcode vssw
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
}
```

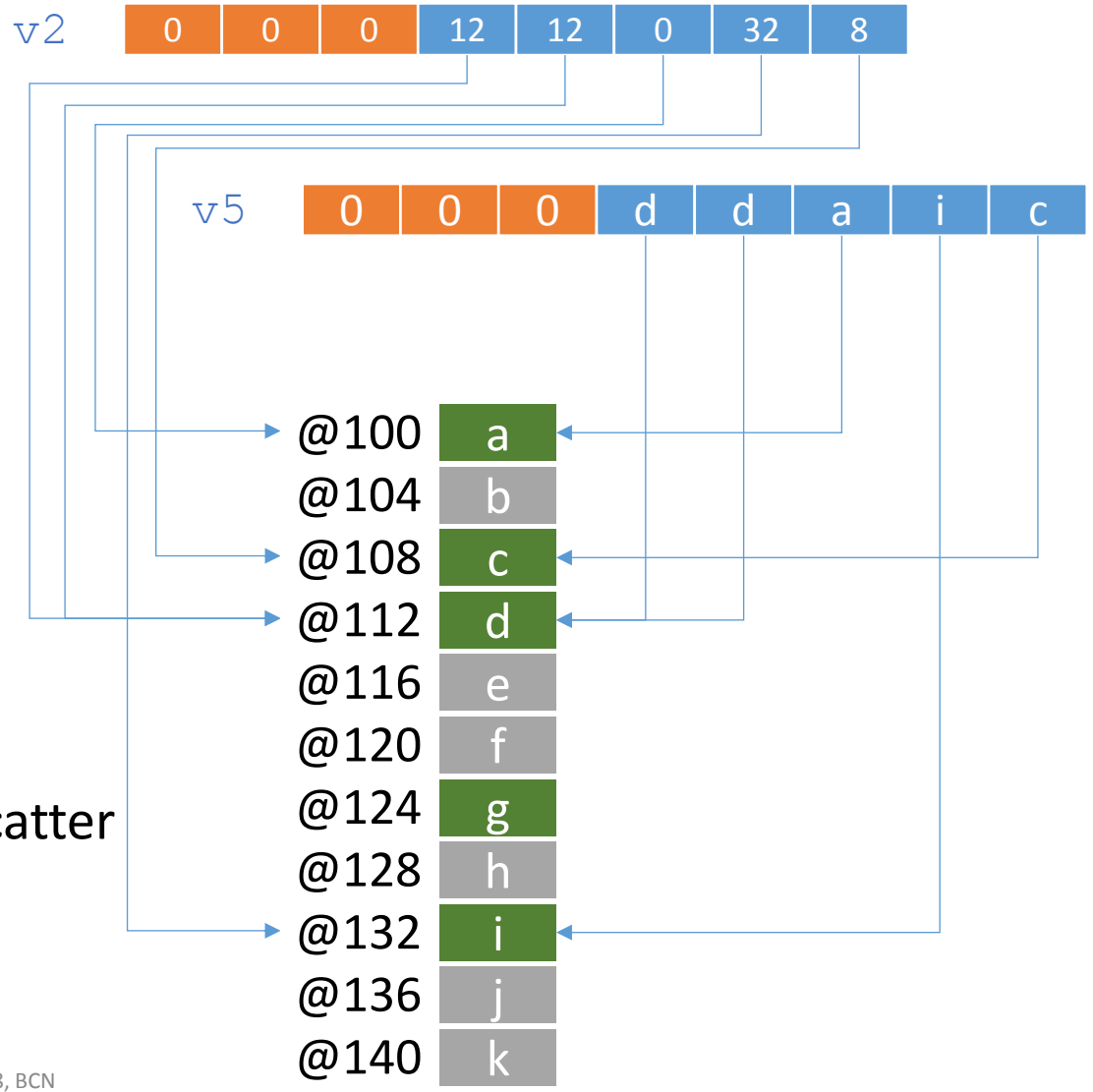


- Stride 0 is legal
- Strides that result in unaligned accesses are legal
  - likely very slow

Scatter (indexed vector store)

vsxw v5, 80(x3,v2)

```
sz  =4; // Based on opcode vsxw
tmp = x3 + 80; // x3 = 20
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

Reconfigurable Vector Register File

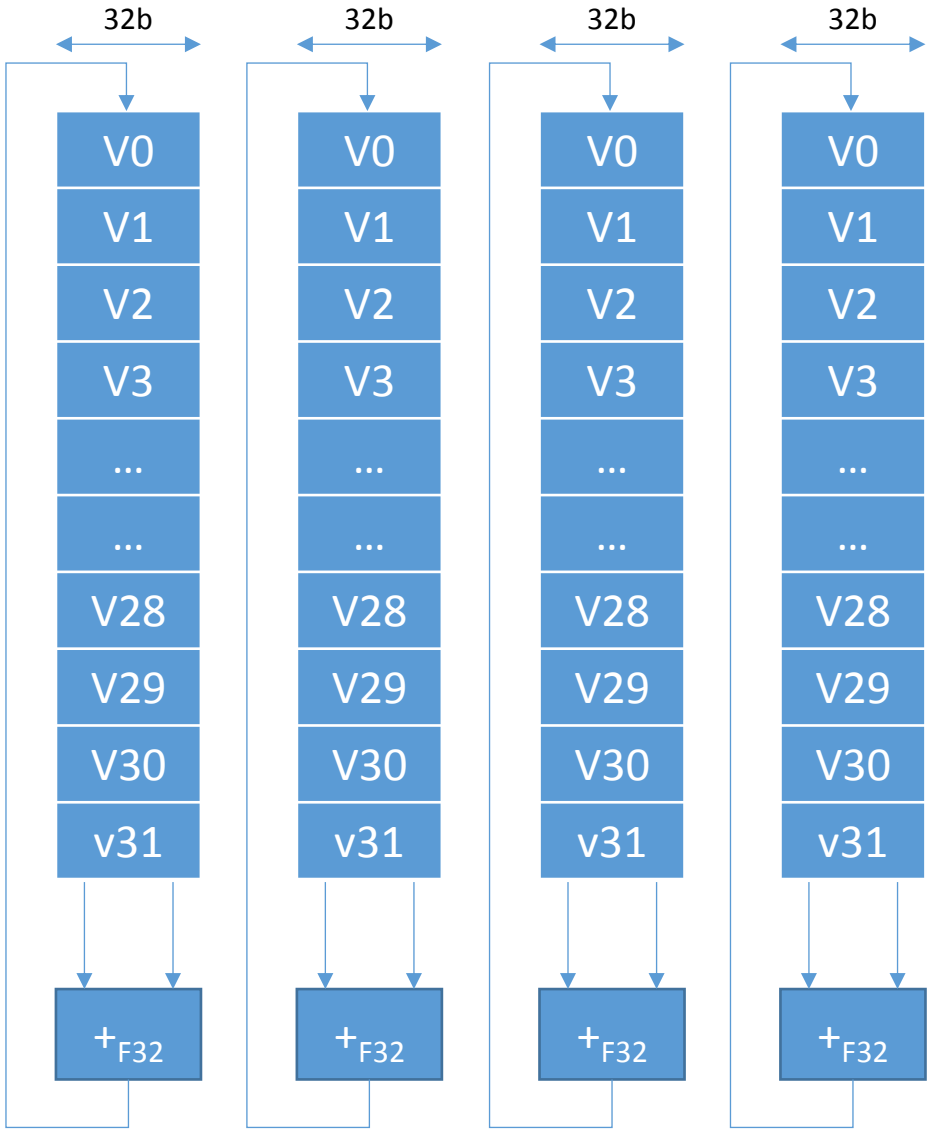
Reconfigurable, variable-length Vector RF

- The vector unit is configured with a `vconfig rd, imm`
  - Imm contains the new configuration indicating
    - Number of logical registers (from 2 to 32)
    - Max element with
  - Hardware resets all vector state to zero
  - Hardware computes Maximum Vector Length (MAXVL)
    - based on imm and available vector register file storage
  - MAXVL returned in rd
  - Can be done in user mode, expected to be fast
- The vector unit is disabled with vconfig x0, 0x0
  - Very good to save kernel save & restore, and low power state
- Implementation choices
  - Always return the same MAXVL, 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 (MAXVL)

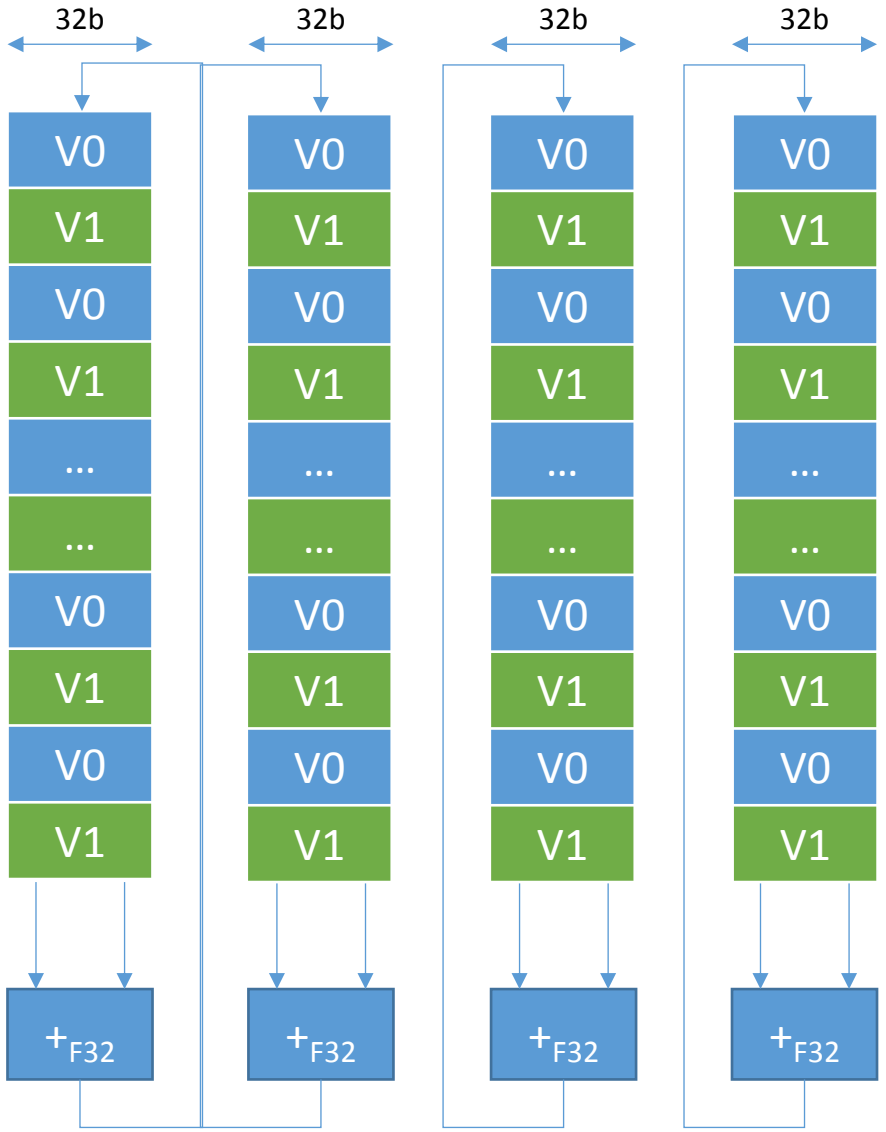
Users asks for 32 registers, vemaxw=32b

- 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
  - MAXVL = 512B / (32 * 4) = 4
- How is the VRF organized?
  - Many possible ways
  - Showing one possible organization



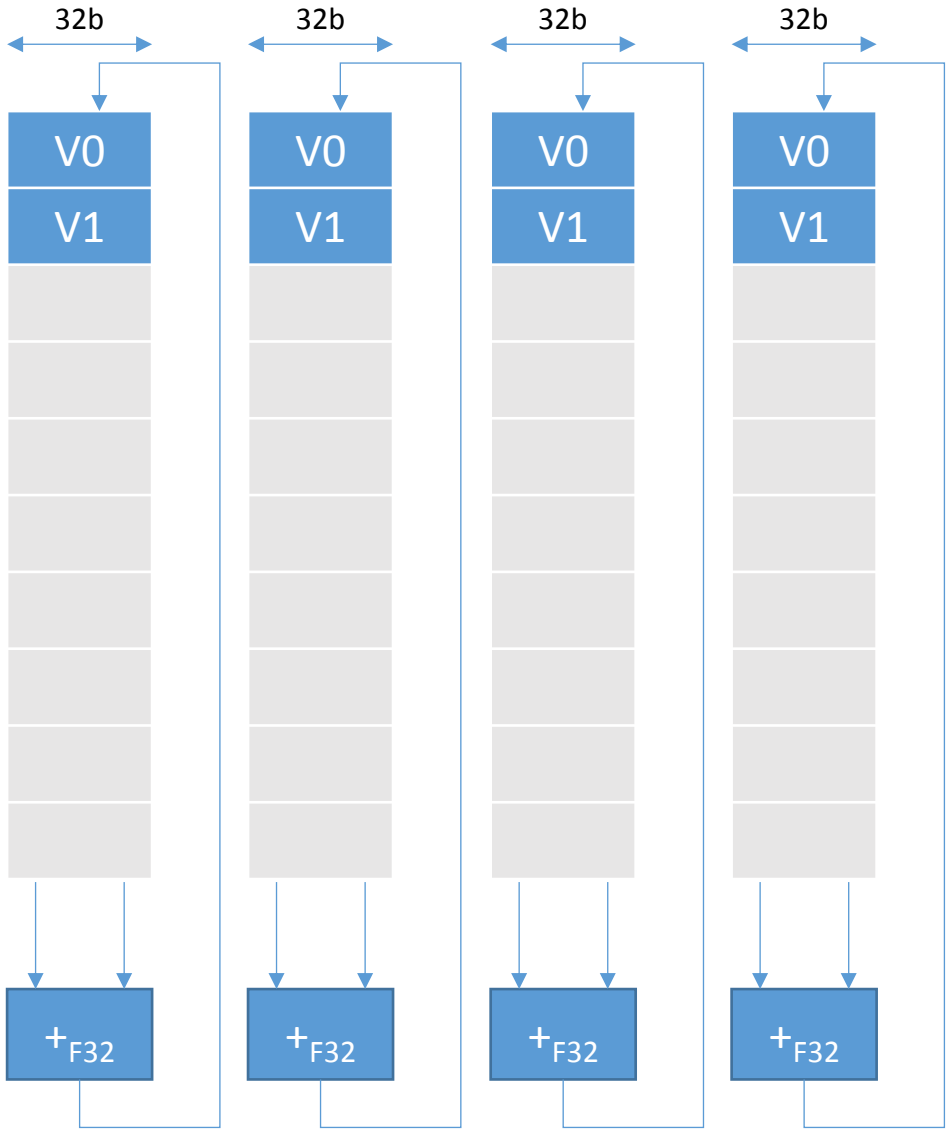
Users asks for 2 registers, vemaxw=32b

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



Users asks for 2 registers, vemaxw=32b

- Hardware has 32r x 4e x 4B = 512B
- Need
  - 4 bytes per v0 element
  - 4 bytes per v1 element
- Therefore
  - MAXVL = 512B / (4+4) = 64
- And yet, implementation…
  - …answers with MAXVL = 4
  - Absolutely legal!
- How is the VRF organized?
  - Many possible ways
  - Showing one possible organization



Users asks for 2 registers, vemaxw=16b

- Hardware has 32r x 4e x 4B = 512B
- Need
  - 2 bytes per v0 element
  - 2 bytes per v1 element
- Therefore
  - MAXVL = 512B / (2+2) = 128

- How is the VRF organized?
  - Many possible ways
  - Showing an INTERLEAVED organization, where hardware PACKS two 16b elements per RF entry

MAXVL is transparent to software!

- If using setvl instruction, code can be portable across
  - Different number of lanes
  - Different values of MAXVL
- vsetvl rd, rs1
  - vl = rd = rs1 > MAXVL ? MAXVL : rs1

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

Part-III: Detailed Instruction Semantics

Helper Functions

```
size2opcode(opcode)
    switch (opcode)
    {
      case vlb, vlbu:          size = 1;
      case vlh, vlhu, vflh:    size = 2;
      case vlw, vlwu, vflw:    size = 4;
      case vld, vfld:          size = 8;
    }
```

`zext(val, sz, fullsz) : sign extend val from sz to full sz`

`zext(val, sz, fullsz) : zero extend val from sz to full sz`

`nanext(val, sz, fullsz) : nan box-extend val from sz to full sz, adding 1's from sz+1 to fullsz`

Opcodes: vlb, vlh, vlw, vld

Format: opcode vd, rs1, imm, vm

```
size = opcode2size(opcode);
if ( size > bytes(vemaxw) ) excep(ILLEGAL);
tmp = rs1 + imm;
len = vl == 0 ? 0 : ( vm == 0x0 ? 1 : vl );
for (i = 0; i < len; i++ )
{
  if ( vm[1] == 0 || LSB(v0[i]) == vm[0] )
  {
    vd[i] = sext(read_mem(tmp, sz), sz, vemaxw);
  }
  tmp = tmp + size;
}
for (i = len; i < maxvl; i++ ){ vd[i] = vm == 0 ? vd[0] : 0; }
```

Opcodes: vlbu, vlhu, vlwu, vldu

Format: opcode vd, rs1, imm, vm

```
size = opcode2size(opcode);
if ( size > bytes(vemaxw) ) excep(ILLEGAL);
tmp = rs1 + imm;
len = vl == 0 ? 0 : ( vm == 0x0 ? 1 : vl );
for (i = 0; i < len; i++ )
{
  if ( vm[1] == 0 || LSB(v1[i]) == vm[0] )
  {
    vd[i] = zext(read_mem(tmp, sz), sz, vemaxw);
  }
  tmp = tmp + size;
}
for (i = 1; i < vl && vm == 0; i++ )   { vd[i] = vd[0]; }
for (i = vl; i < maxvl; i++ )          { vd[i] = 0; }
```

Opcodes: vflh, vfls, vfld

Format: opcode vd, rs1, imm, vm

```
size = opcode2size(opcode);
if ( size > bytes(vemaxw) ) excep(ILLEGAL);
tmp = rs1 + imm;
len = vl == 0 ? 0 : ( vm == 0x0 ? 1 : vl );
for (i = 0; i < len ; i++ )
{
  if ( vm[1] == 0 || LSB(v1[i]) == vm[0] )
  {
    vd[i] = nanext(read_mem(tmp, sz), sz, vemaxw);
  }
  tmp = tmp + size;
}
for (i = 1; i < vl && vm == 0; i++ )   { vd[i] = vd[0]; }
for (i = vl; i < maxvl; i++ )          { vd[i] = 0; }
```

Floating point loads perform NaN boxing, adding ‘1’s up to vemaxw to fill up each vector element .

UNFINISHED!!! TO BE COMPLETED

Part-IV: Extensions

Reductions

Vector Integer Reductions (not in base)

operation	instructions
add	vredsum
max	vredmax, vredmaxu
min	vredmin, vredminu
logical	vredand, vredor, vredxor

Part-IV: Extensions

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`$_{F32}$ → `v0`$_{F16}$
  - `vcvt v1`$_{u8}$ → `v0`$_{F32}$
  - `vcvt v1`$_{F32}$ → `v0`$_{I32}$
- Additional feature: changing the dest register type with vcvt
  - `vcvt v1`$_{F32}$ → `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

		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

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 7th, 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 2nd Friday 8am PST
  - Warning: Github spec is out-of-date: WIP to update to this presentation

Vector Data Movement

operation	instructions	action
move from fp register to vector	vfmv.v.f vd, fs1, vm	vd[*] = fs1
move to fp register from vector	vfmv.f.v fd, vs1	fd = vs1[0]
insert gpr into vector	vinsx vd, rs1, rs2, vm	vd[rs2] = rs1
extract gpr from vector	vextx rd, vs1, rs2	rd = vs1[rs2]
insert vector element into vector	vinsv vd, vs1, rs2	vd[rs2] = vs1[0]
extract vector element from vector	vextv vd, vs1, rs2, vm	Vd[*] = vs1[rs2]
Vector-Vector Merge	Vmerge vd, vs1, vs2, vm	Mask picks src
Vector-GPR Merge	Vmergex vd, rs1, vs2, vm	Mask picks src
Vector Register Gather	Vrgather vd, vs1, vs2, vm	Vd[i] = vs1[vs2[i]]
Vector Slide Down	vslidedwn vd, vs1, rs2, vm	vd[i] = vs1[rs2+i]
Vector Slide Up	vslideup vd, vs1, rs2, vm	vd[rs2+i] = vs1[i]