RISC-V Vector Extension Webinar III

August 31st, 2021

Thang Tran, Ph.D.
Principal Engineer

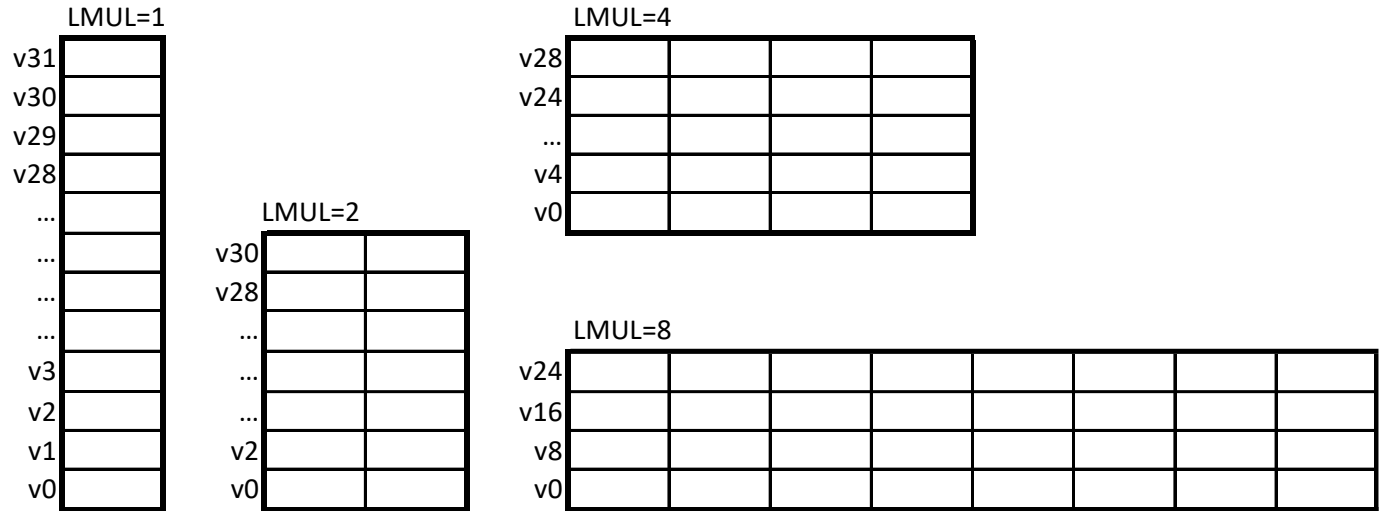
Webinar III - Agenda

- Andes overview
- Vector technology background
  - SIMD/vector concept
  - Vector processor basic
- RISC-V V extension ISA
  - CSR
- **RISC-V V extension ISA**
  - **Memory operations**
  - **Compute instructions**
  - **Permute instructions**
- Sample codes
  - Matrix multiplication
  - Loads with RVV versions 0.8 and 1.0
- AndesCore™ NX27V introduction
- Summary

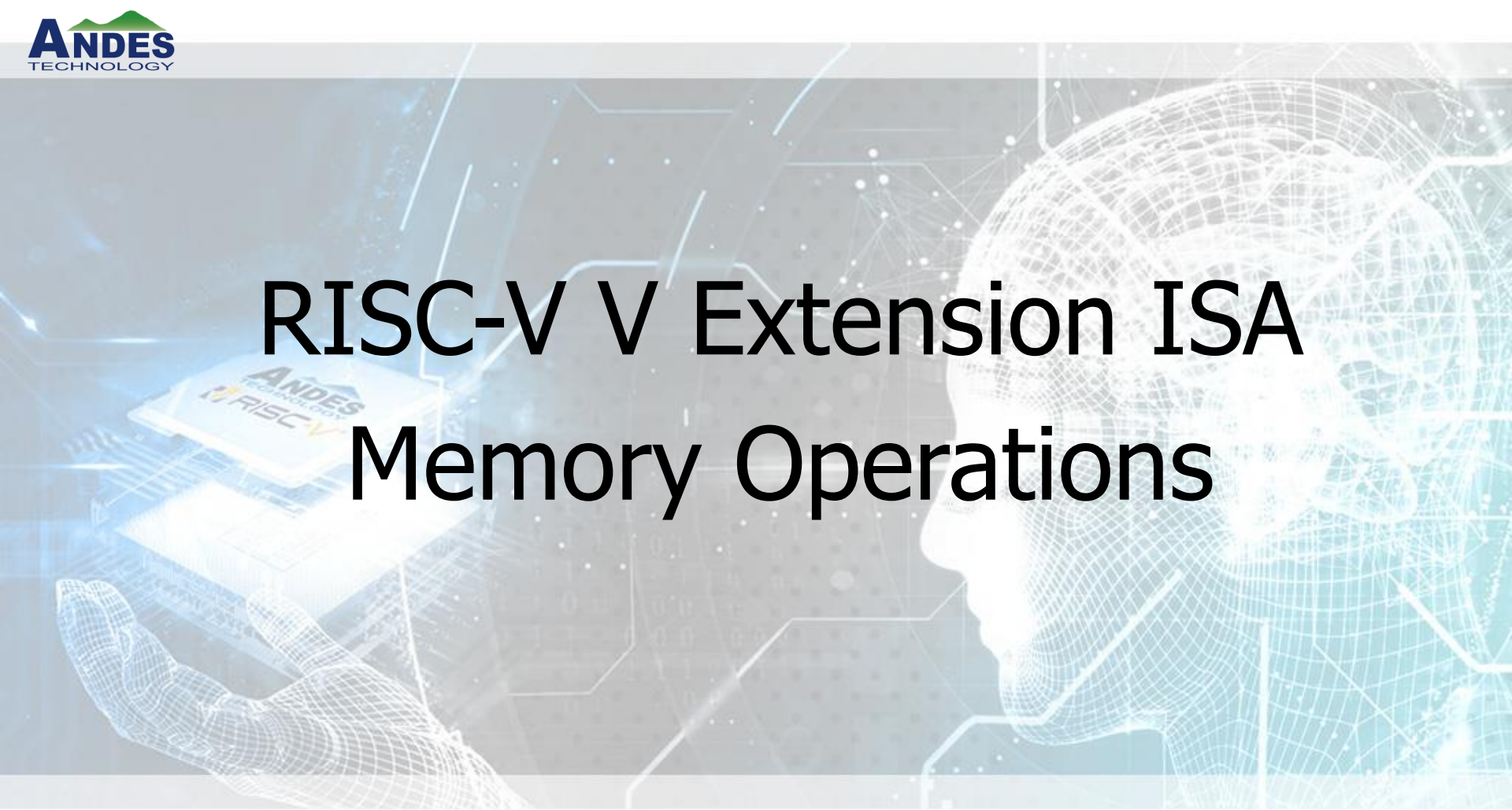
Terminology

- ISA: Instruction Set Architecture
- GOPS: Giga Operations Per Second
- GFLOPS: Giga Floating-Point OPS
- **XRF**: Integer register file
- FRF: Floating-point register file
- **VRF**: Vector register file
- SIMD: Single Instruction Multiple Data
- MMX: Multi Media Extension
- SSE: Streaming SIMD Extension
- AVX: Advanced Vector Extension
- ACE: Andes Custom Extension
- CSR: Control and Status Register
- **SEW**: Element Width (8-64)
- ELEN: Largest Element Width (32 or 64)
- **XLEN**: Scalar register length in bits (64)
- FLEN: FP register length in bits (16-64)
- **VLEN**: Vector register length in bits (128-512)
- **LMUL**: Register grouping multiple (1/8-8)
- EMUL: Effective LMUL
- **VLMAX**/MVL: Vector Length Max
- AVL/**VL**: Application Vector Length
- **Configurable**: parameters are fixed at built time, i.e. cache size
- **Extensible**: added instructions to ISA includes custom instructions to be added by customer
- **Standard extension**: the reserved codes in the ISA for special purposes, i.e. FP, DSP, …
- **Programmable**: parameters can be dynamically changed in the program

LMUL – Registers Grouping



- LMUL=2, instructions with odd register are illegal instructions
- LMUL=4, vector registers are incremented by 4, else illegal instructions
- LMUL=8, only v0, v8, v16, and v24 are valid vector registers



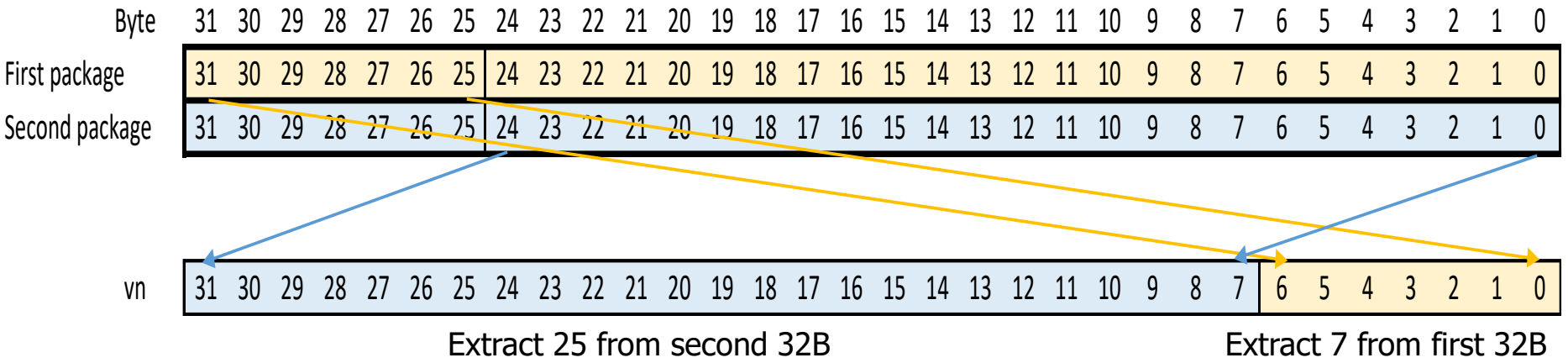
RISC-V V Extension ISA Memory Operations

Memory Operations

- Load/store operations move groups of data between the VRF and memory
  - *Andes implements unaligned addressing, no restriction*
- Four basic types of addressing
  - Unit stride: fetch consecutive elements in contiguous memory
  - Non-unit or constant stride: elements are in stride memory; stride is the distance between 2 elements of a vector in memory
  - Indexed (gather-scatter): elements are randomly offset from a base address
  - Segment: move multiple contiguous fields in memory to and from consecutively numbered vector registers – can be unit-stride, constant stride, or index
- *Atomic load/store is not implemented*

Vector Load Data Alignment

Andes processor handles all unaligned addresses



Instruction Format – Load/Store

	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
VL* unit-stride	nf	mw	mop	vm	lumop	rs1	width	vd	0000111
VLS* strided	nf	mw	mop	vm	rs2	rs1	width	vd	0000111
VLX* indexed	nf	mw	mop	vm	vs2	rs1	width	vd	0000111
VS* unit-stride	nf	mw	mop	vm	sumop	rs1	width	vs3	0100111
VSS* strided	nf	mw	mop	vm	rs2	rs1	width	vs3	0100111
VSX* indexed	nf	mw	mop	vm	vs2	rs1	width	vs3	0100111

Segment, multiple registers

mop[1:0]	function
00	unit-stride
01	index_unordered
10	constant-stride
11	index_ordered

lumop/sumop[4:0]	
00000	unit-stride
01000	unit-stride, whole register
01011	unit-stride, mask load/store, EEW=8
10000	load, fault-only-first
others	reserved

mw+width[2:0]		FL/FS
x001	16	H
x010	32	W
x011	64	D
x100	128	Q
	Vector - VLx/VSx	
0000	8	E8
0101	16	E16
0110	32	E32
0111	64	E64
1xxx	Reserved	

FP load/store

EEW=MEW

Vector load/store

Vector load/store, different EMUL

- EMUL is the effective LMUL
- SEW=16, LMUL=1, VLEN=512b
  - elements=512/16=32, **all vector instructions operate on 32 elements**

```
vle16.v v1, (t0)        # Load a vector of sew=16, lmul=1, load to v1
vle32.v v2, (t1)        # Load a vector of sew=32, emul=2, load to v2 and v3
vwadd.wv v4, v2, v1     # v4/v5 ← v2/v3 + sign-extend(v1), emul=2
vse32.v v4, (t1)        # Store a vector of sew=32, emul=2, v4 and v5 store to memory

vle8.v v1, (t1)         # Load a vector of sew=8, emul=1/2, load to elements first half of v1
vle64.v v4, (t0)        # Load a vector of sew=64, emul=4, load to v4, v5, v6, v7
```

Unit-Stride Load/Store Variances

- **Load Fault-only-First:** exception only for first element
  - Set vl=faulted element, no exception
  - Use for exit condition of loop
  - Example: the unit load fetches to the end of the TLB page, all subsequent vector operations will be for the valid elements that were fetched. This is the reversed of the stripped mining for unknown number of application elements
- **Load/store of whole register:**
  - vtype & vl are ignored, SEW=8b, uses NF bits for multiple vector registers
  - vm=1 (inst[25], no masking load all elements), else illegal instruction
  - Vstart >= VLEN/8 (number of 8-bit elements in vector register), no operation
  - Use for fill/spill, context switch, return from interrupt, …

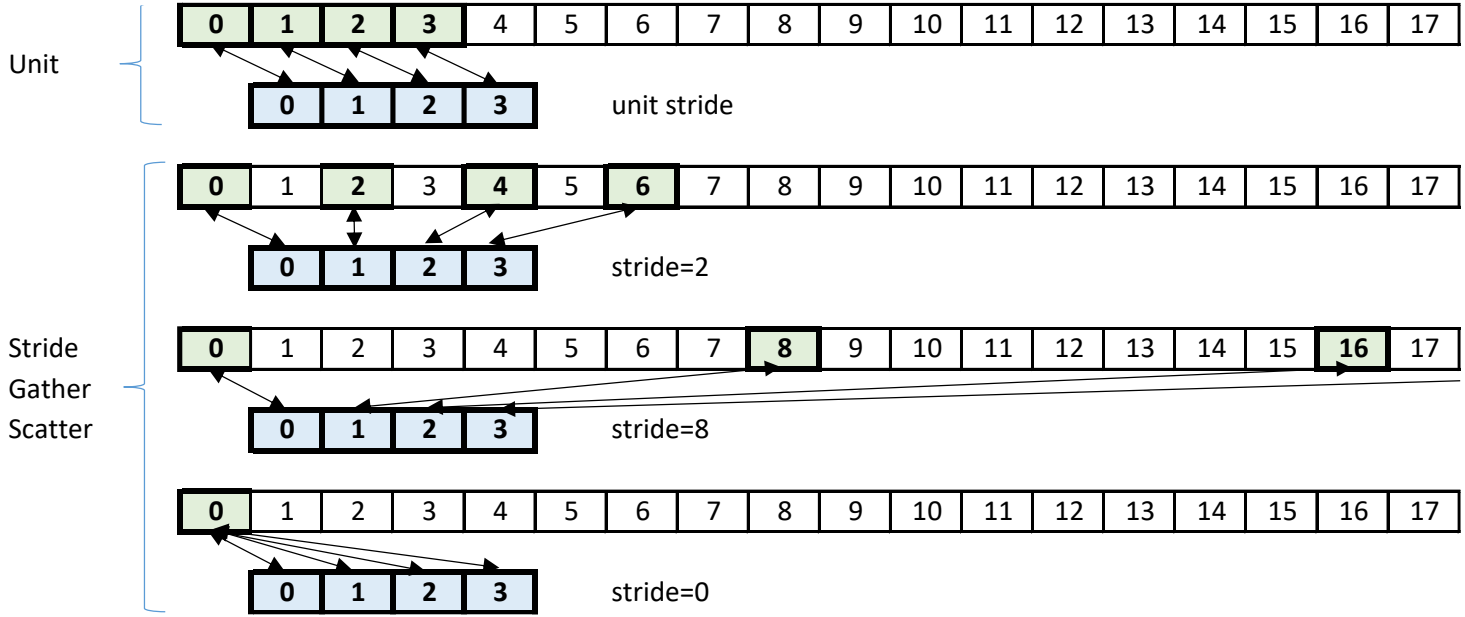
Example: strncpy, Fault-only-First Load

```
strncpy:
    mv a3, a0                          // Copy dst
loop:
    vsetvli x0, a2, e8, m8, ta, ma     // Vectors of bytes, VLEN=512b
    vle8ff.v v8, (a1)                  // Fault-only-First load to 8 vector registers
    vmseq.vi v1, v8, 0                 // Flag zero bytes (mask)
    csrr t1, vl                        // Get number of bytes fetched
```

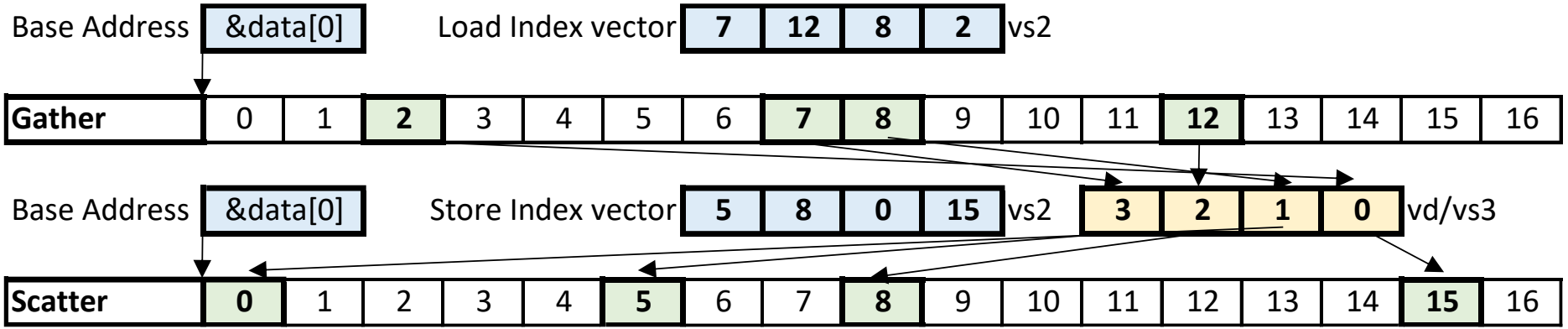
Note: ta is used to write zero to elements after the faulted elements (vl=faulted elements)

Stride Gather/Scatter Memory Access

- Unit stride continuous memory access
- Constant stride: gather scatter (stride can be negative or zero)
  - Support store memory overlapped due to stride=0 or stride<SEW



Index Gather/Scatter Memory Access

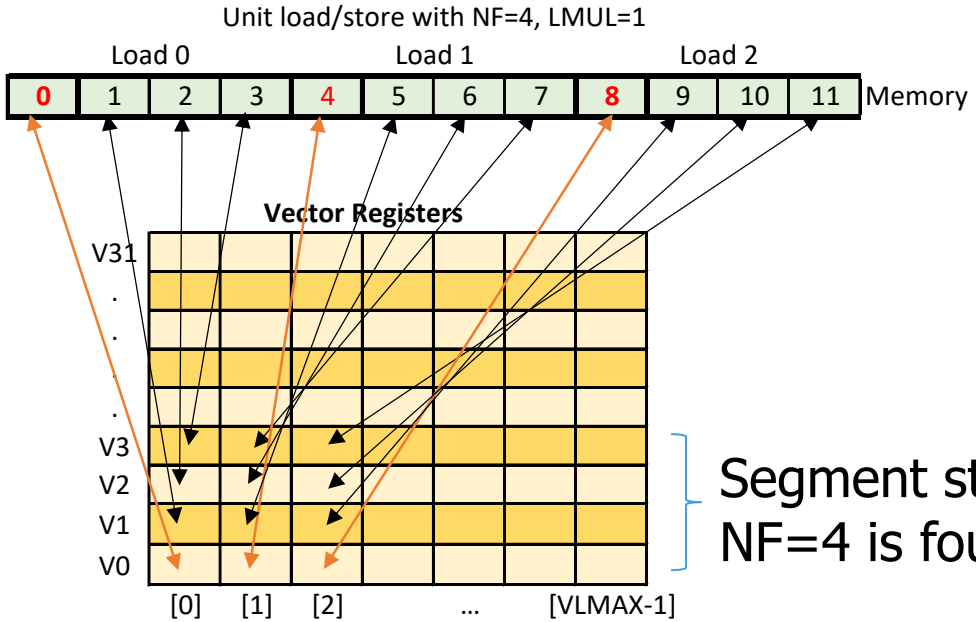


Note:

- The indices are absolute and not accumulative
- The indices are not in-order

Segment Unit Stride Load/Store

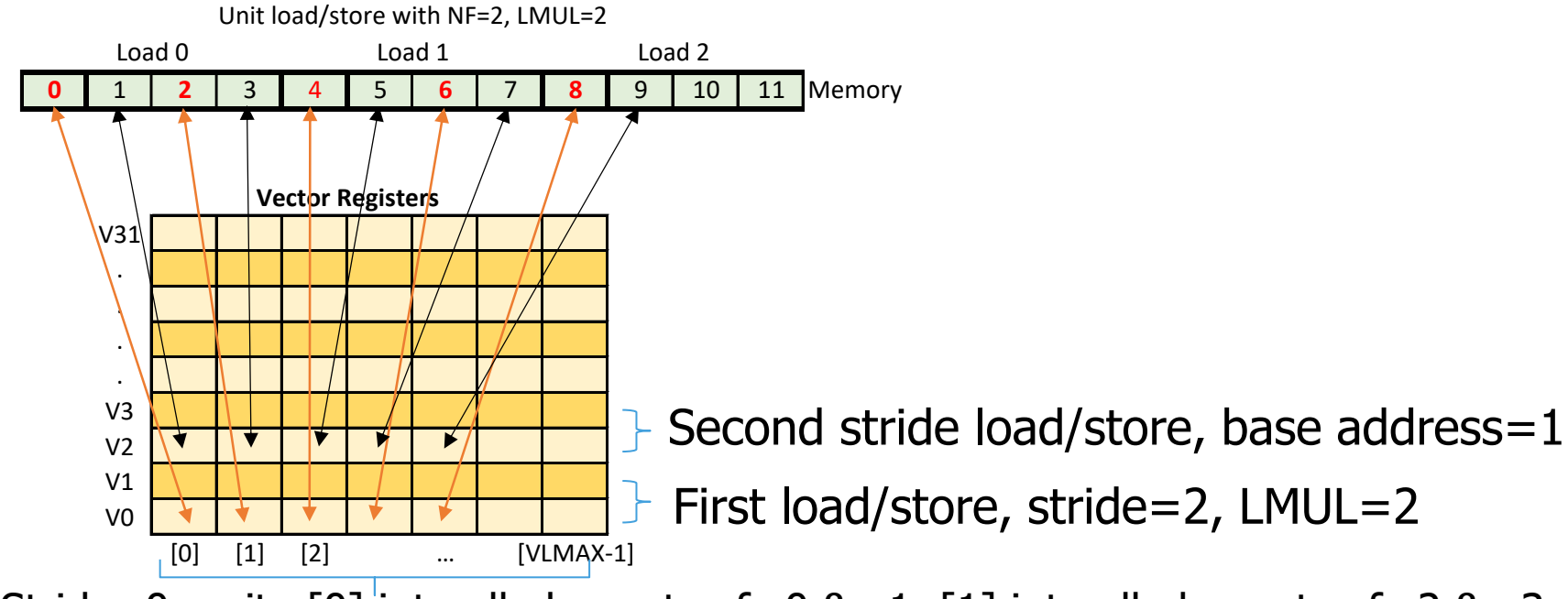
- Multiple vector registers load/store, NF*EMUL <= 8
- Equivalent to 4 stride load/store micro-ops in below example
- Fault-only-first is allowed



Segment stride load/store with NF=4 is four stride loads/stores

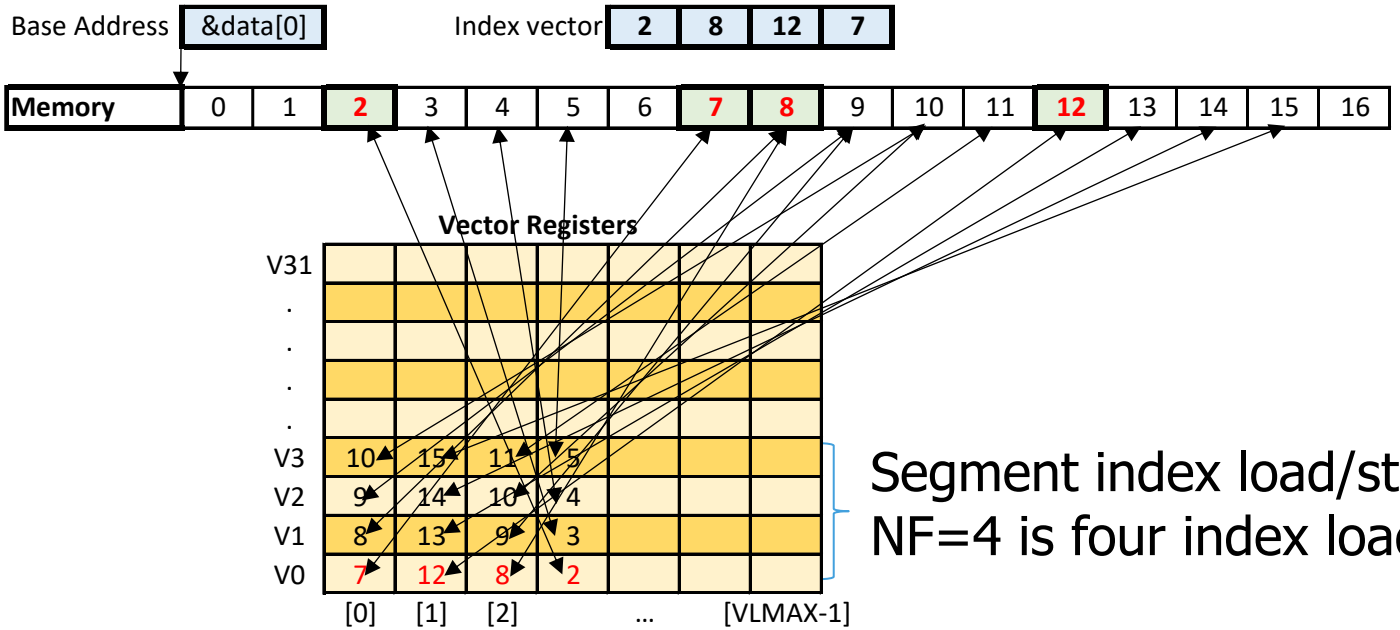
Segment Stride Load/Store, LMUL=2

- Multiple vector registers load/store, NF*EMUL <= 8
- Zero and negative strides are allowed



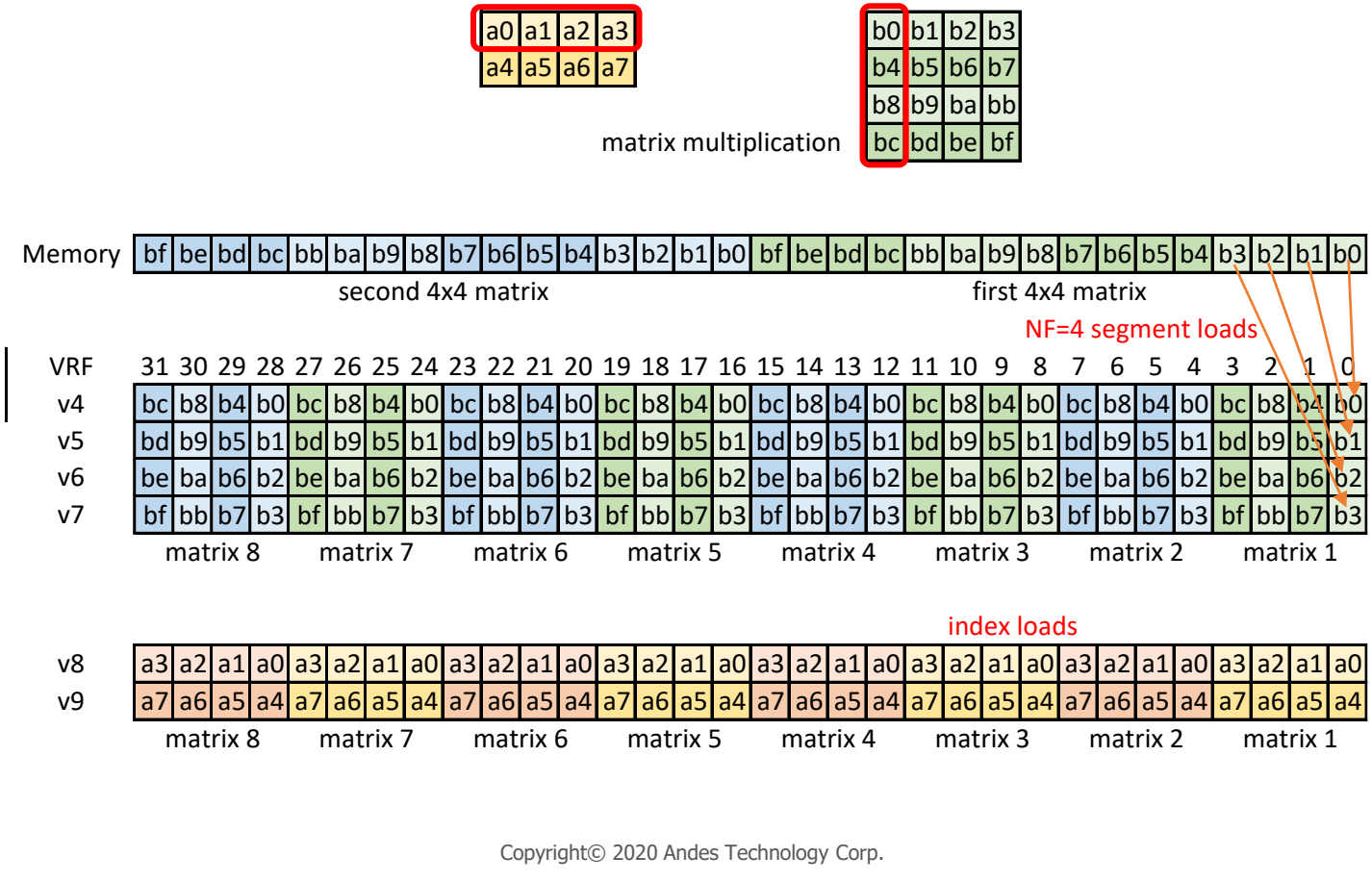
Segment Index Load/Store

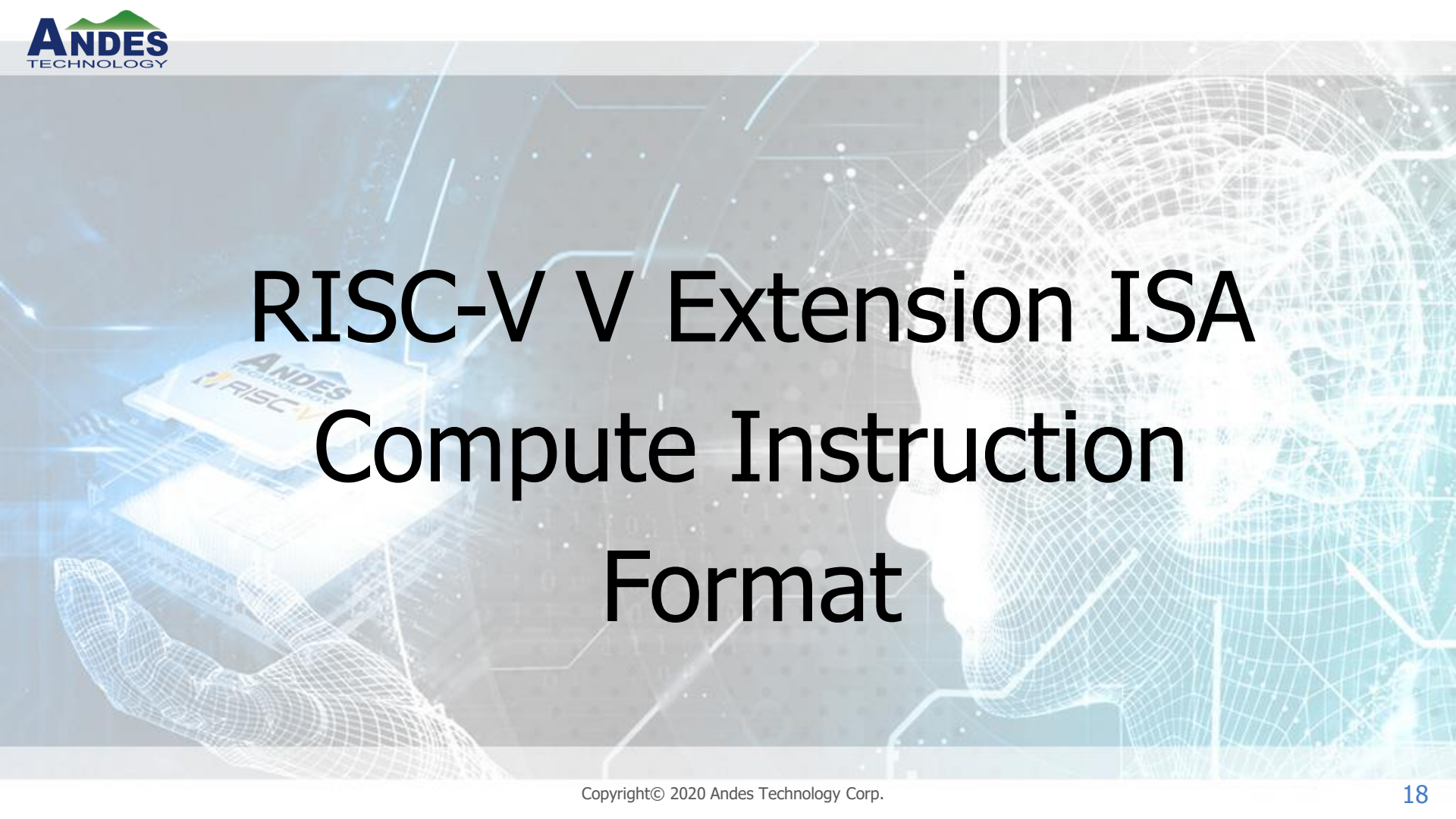
- Multiple vector registers load/store, NF*EMUL <= 8
  - Overlapping of store data to memory
  - NF number of index load/store micro-ops, the index is incremented by SEW



Segment index load/store with NF=4 is four index loads/stores

Matrix Multiplication Loads Example

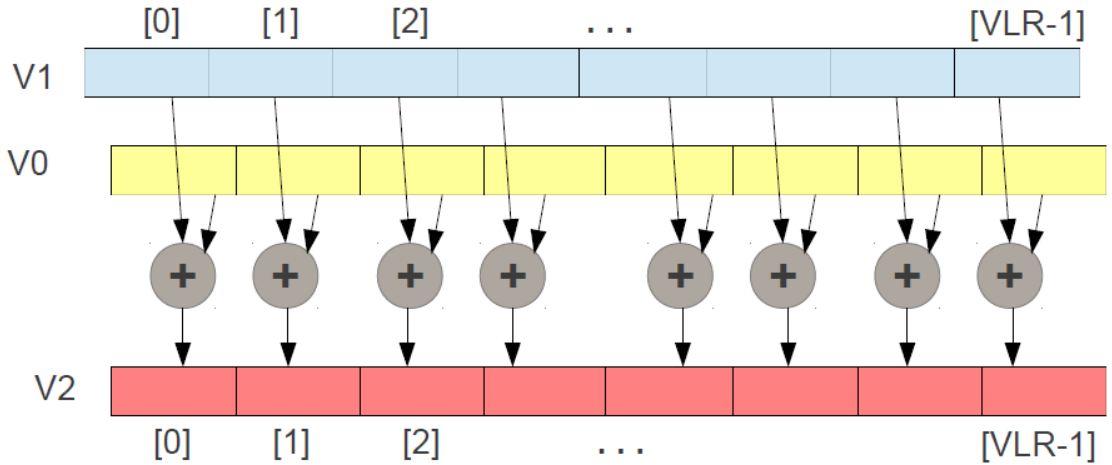




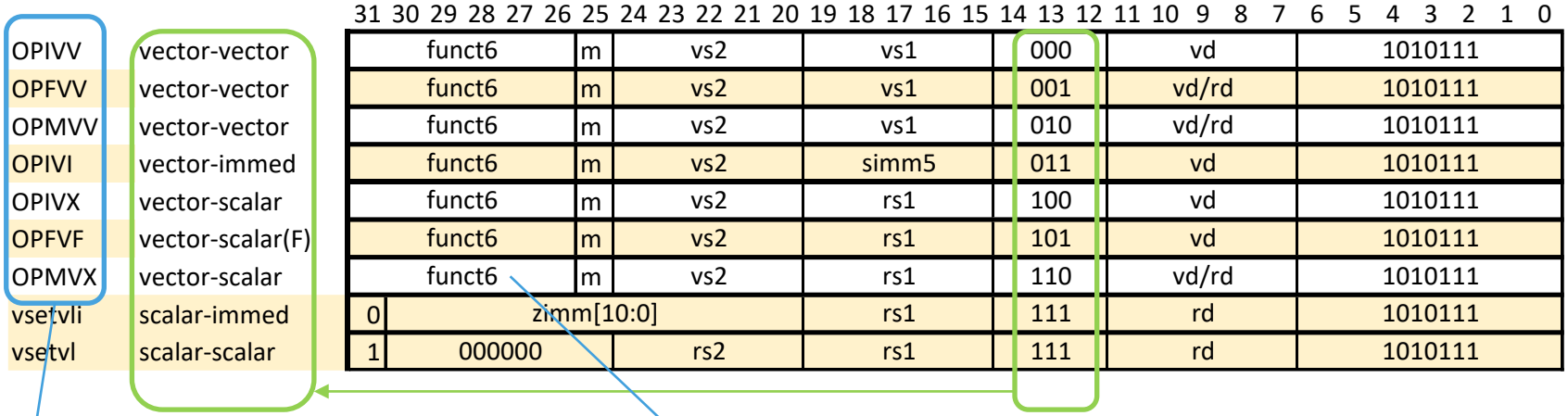
RISC-V V Extension ISA Compute Instruction Format

Vector Arithmetic Instruction Overview

- vop.**vv** vd, vs2, vs1, vm // vector-vector, vd[i]=vs2[i] op vs1[i]
- vop.**vx** vd, vs2, rs1, vm // vector-scalar, vd[i]=vs2[i] op x[rs1]
- vop.**vi** vd, vs2, imm, vm // vector-imm, vd[i]=vs2[i] op imm
- vfop.**vv** vd, vs2, vs1, vm // vector-vector, vd[i]=vs2[i] fop vs1[i]
- vfop.**vx** vd, vs2, rs1, vm // vector-scalar, vd[i]=vs2[i] fop f[rs1]



Compute Instruction Format



OPI – integer operation

OPF – floating-point operation

OFM – mask operation

Funct6 is used to encode all vector instructions, most of the time, the same funct6 code is used for all 3 forms for vector instructions (vector, scalar, immediate)

Vector Widening Instructions

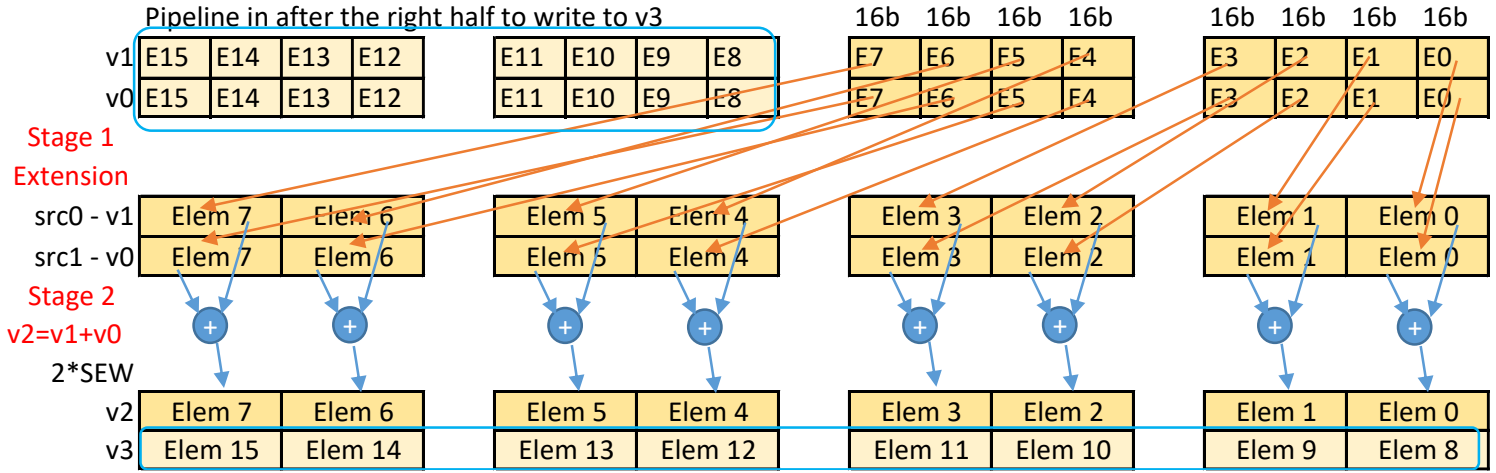
- Widening: double width
  - vwop.v{v,x} vd, vs2, vs1/rs1 // 2*SEW = sign/zero(SEW) op sign/zero(SEW)
  - vwop.w{v,x} vd, vs2, vs1/rs1 // 2*SEW = 2*SEW op sign/zero(SEW)
  - vfwop.v{v,f} vd, vs2, vs1/rs1 // 2*SEW = SEW op SEW
  - vfwop.w{v,f} vd, vs2, vs1/rs1 // 2*SEW = 2*SEW op SEW (vfwadd & vfwsub)
- Widening operations
  - Integer ALU: unsigned/signed extended source operands, double width operation
  - Integer MUL: single width operation, then return double-width results
  - Floating-point: single width operation, then return double-width results
  - **Same number of elements rule as defined in vtype**:
    - Double width → writing back to twice the number of vector registers

Vector Widening Instructions – Extension

- Vector integer extension
  - V{z,s}ext.vf{2,4,8} vd, vs2 // Zero/sign **extend to SEW**
  - The result is the same number of elements set by vtype
  - Example: SEW=32b, then only V{z,s}ext.vf{**2,4**} are legal instructions

Shifting of Source Operands for Double-Width

- The data are extended and folded into the source data from VRF before sending to execution
  - Cycle 1: operate on full-width data and write back to v2
    shift data from left-half and extension to full width
  - Cycle 2: operate on full-width data and write back to v3



Vector Narrow Instructions

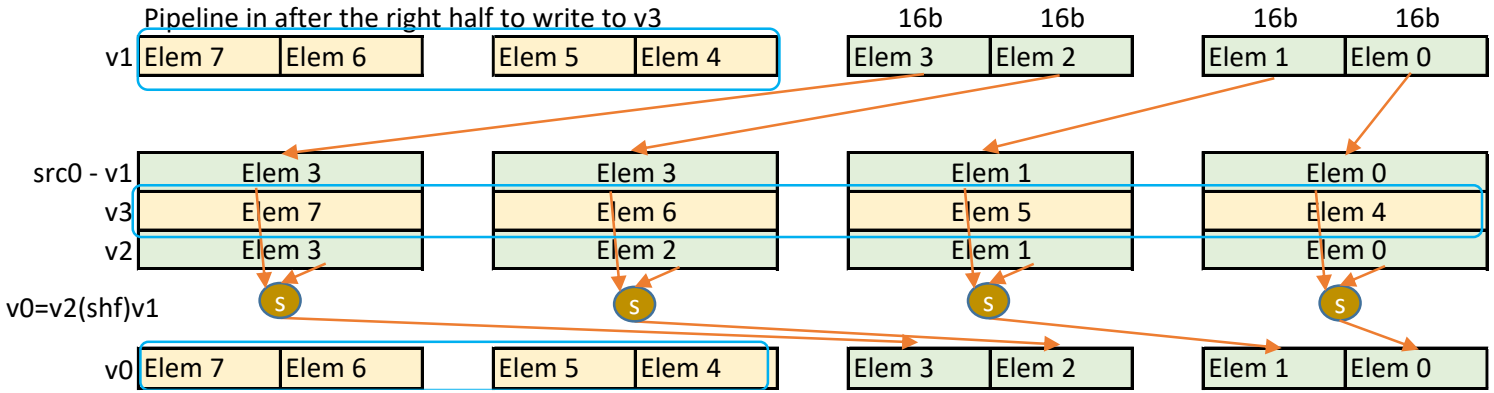
- Narrowing: from double-width
  - vnop.w{v,x,i} vd, vs2, vs1/rs1 // SEW = 2*SEW op SEW
  - Logical and arithmetic right shift operations

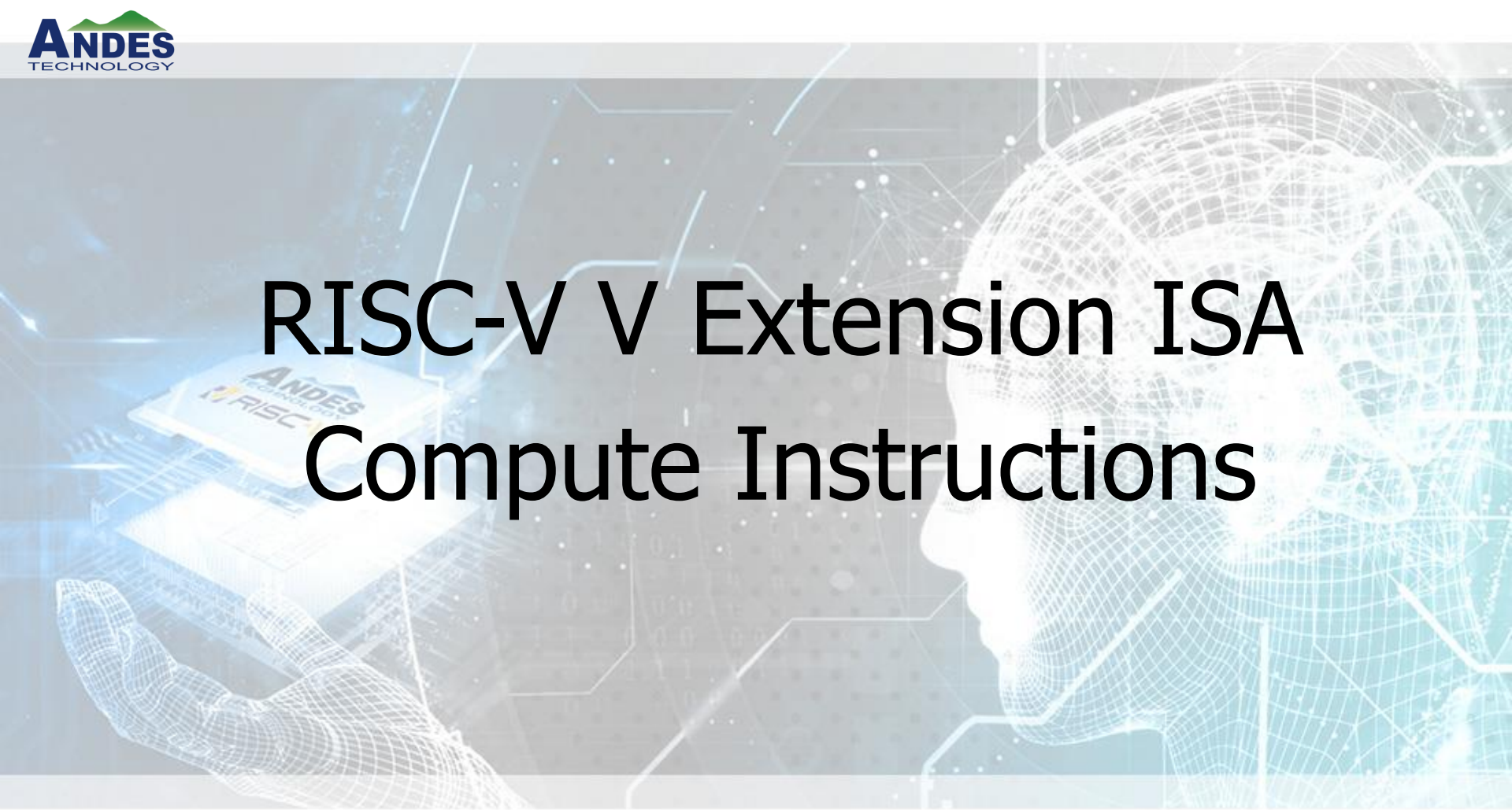
Example of operations without changing vtype, LMUL=1, SEW=16b, 32 elements in all vector instruction operations:

vle16.v v1, (t0) // **Load to v1** full vector registers
vle32.v v2, (t1) // Load a vector of sew=32b, emul=2, **load to v2 and v3**
vwadd.wv v0, v2, v1 // v0/v1 ← v2/v3 + sign-extend(v1), emul=2
vnsra.wx v7, v0, t2 // v7 ← v0/v1 (shift right by t2)
vse16.v v7, (t0) // **Store v7** to memory

Shifting of Source Operands for Narrow-Width

- The data are extended and folded into the source data from VRF before sending to execution
  - Cycle 1: operate on full-width data
  - Cycle 2: Shift result data to write back to right-half of v0





RISC-V V Extension ISA
Compute Instructions

Vector Integer Arithmetic Instructions

Vector Integer **Add, Subtract, Reverse Subtract** (including Double Width)
Vector Integer **Extension**
Vector Integer **Add-with-Carry** / **Subtract-with-Borrow**
Vector Bitwise **Logical** Instructions
Vector Bit **Shift** Instructions (including Narrow Width)
Vector Integer **Comparison** Instructions, write 1/0 to destination register
Vector Integer **Min/Max** Instructions, write min/max to destination register
Vector Integer **Multiply** (including Double Width)
Vector Integer **Divide** and Remain
Vector Integer **Multiply-Add/Sub**, **Multiply-Accumulate** (including Double Width)
Vector Integer **Merge** Instructions (vector – vs1, scalar – rs1, imm)
Vector Integer **Move** Instructions (vector – vs1, scalar – rs1, imm)

Vector Arithmetic with Carry Instructions

vadc.vv vd, vs2, vs1, vm=0 // vd[i] = vs2[i] + vs1[i] + v0[i].mask
vmadc.vvm vd, vs2, vs1, vm=0 // vd[i].mask = **carry_out**(vs2[i] + vs1[i] + v0[i].mask)

- The carry-in is in v0 same as mask register
- Vector instructions oriented to implementation :
  - Single vector destination register
  - Single write port per vector instruction 
- Two vector instructions to complete the add with carry, rules:
  - **vadc**: destination and source registers cannot be overlapped
  - **vadc**: destination register cannot be v0
  - **vmadc**: destination register can be v0 or any vector register

Vector MAC Instructions

v(w)**madd**.vv vd, vs2, vs1, vm // vd[i] = **vs2[i]** + (vs1[i] * vd[i])

v(w)**madd**.vx vd, vs2, rs1, vm // vd[i] = **vs2[i]** + (rs1 * vd[i])

v(w)**macc**.vv vd, vs2, vs1, vm // vd[i] = **vd[i]** + (vs1[i] * vs2[i])

v(w)**macc**.vx vd, vs2, rs1, vm // vd[i] = **vd[i]** + (rs1 * vs2[i])

v(w)**msub** and v(w)**msac** have the same format

Vector Merge Instructions

```
vmerge.vx vd, vs2, rs1, vm=0      // vd[i] = vm[i] ? rs1     : vs2[i]
vmerge.vv vd, vs2, vs1, vm=0      // vd[i] = vm[i] ? vs1[i] : vs2[i]
vmerge.vi vd, vs2, imm, vm=0      // vd[i] = vm[i] ? imm   : vs2[i]
```

- Each element uses vm[i] to select the input element

Vector Integer Move Instructions

vmv.v.{v,x,i} vd, {vs1, rs1, imm}, vm=1

- Same encoding as with Vector Merge Instructions with vm=1 and vs2=v0
- {x,i} splat a scalar register or immediate to all elements
- vs1=vd is NOP
- Other vs2 values are reserved

Vector Fixed-Point Arithmetic Instructions

- Vector Saturating Add and Subtract (set CSR vxsat bit)
- Vector Averaging Add and Subtract (overflow is ignored)
  - Add/subtract, then shift right by 1, round off according to CSR vxrm bit
- Vector Fractional Multiply with Rounding and Saturation
- Vector Scaling Shift Instructions
- Vector Narrowing Clip Instructions

Vector Fractional Multiply with Rounding and Saturation

vsmul.v{v,x} vd, vs2, {vs1,rs1}, vm

- Double-width multiplication vs2[i]*{vs1[i], rs1}
- Shift right by SEW-1
- Roundoff_signed according to CSR vxrm bit
- Saturate the result to SEW bits, set CSR vxsat bit if saturated
- vd[i] = clip(roundoff_signed(vs2[i]*vs1[i], SEW-1))

Vector Scale-Shift and Clip Instructions

vssrl.v{v/x/i} vd, vs2, {vs1,rs1,imm}, vm // scale-shift right logical

- vd[i] = round_unsigned((vs2[i] + rnd) >> {vs1[i],rs1,imm})

vssra.v{v/x/i} vd, vs2, {vs1,rs1,imm}, vm // scale-shift right arithmetic

- vd[i] = round_signed((vs2[i] + rnd) >> {vs1[i],rs1,imm})

vnclip.v{v/x/i} vd, **vs2**, {vs1,rs1,imm}, vm // vs2 is 2*SEW, narrow clip

- vd[i] = clip(round_signed((vs2[i] + rnd) >> {vs1[i],rs1,imm}))

vnclipu.v{v/x/i} vd, **vs2**, {vs1,rs1,imm}, vm // vs2 is 2*SEW, narrow clip

- vd[i] = clip(round_unsigned((vs2[i] + rnd) >> {vs1[i],rs1,imm}))

Vector Floating-Point Arithmetic Instructions

Vector FP **Add, Subtract, Reverse Subtract** (including Double Width)
Vector FP **Multiply** (including Double Width)
Vector FP **Divide, Reverse Divide**
Vector FP **Multiply-Add/Sub**, **Multiply-Accumulate** (including Double Width)
Vector FP **Square-Root, Reciprocal Square-Root Estimate, Reciprocal Estimate**
Vector FP **Comparison** Instructions, write 1/0 to destination register
Vector FP **Min/Max** Instructions, write min/max to destination register
Vector FP **Merge, Move** Instructions (only fp.scalar – f[rs1])
Vector FP **Sign-Injection** Instructions
Vector FP **Classify** Instructions
Vector **FP/Integer Type-Convert** (including Double Width and Narrowing)

Vector Reduction Instructions

Vector **Integer Reduction** Instructions

- Reduction **sum** (including Double Width)
- Reduction **min/max** (including signed and unsigned)
- Reduction bitwise **AND, OR, XOR**

Vector **FP Reduction** Instructions

- Reduction **ordered sum** (including Double Width)

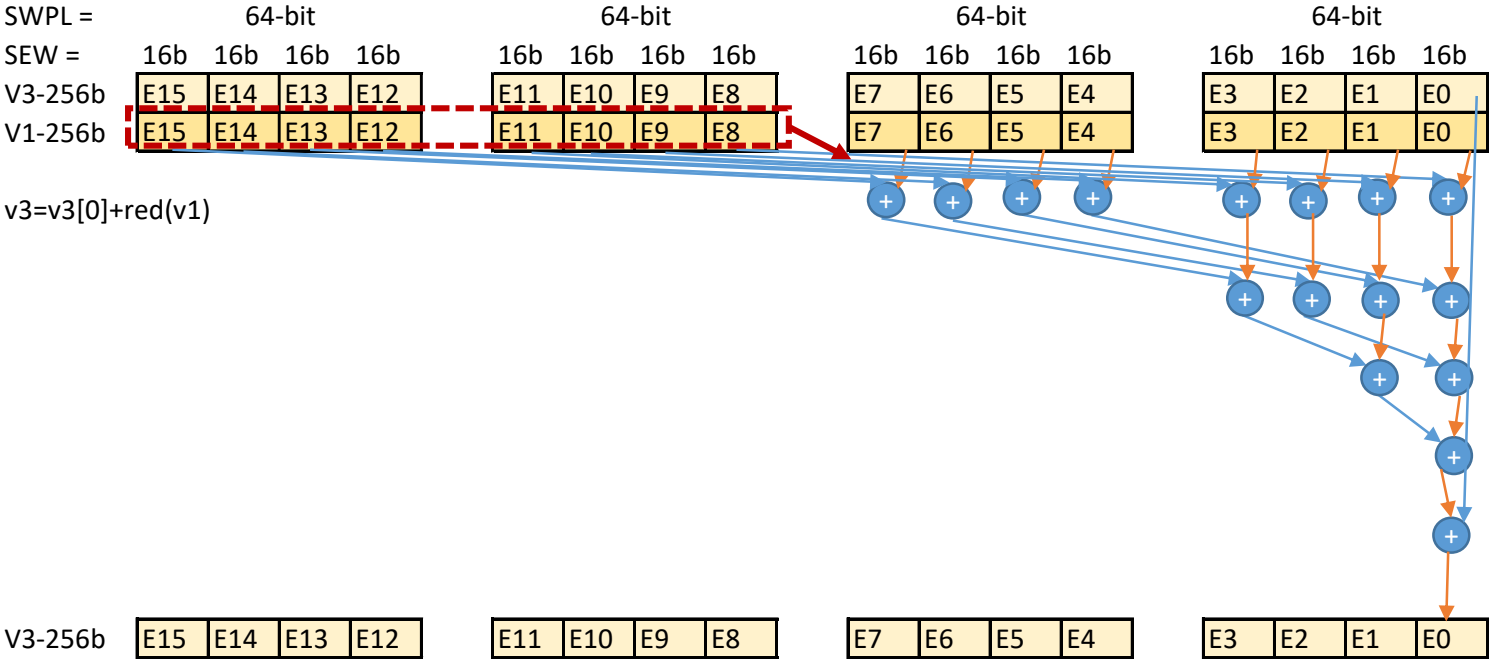
  (((((vs1[0] + vs2[0]) + vs2[1]) + vs2[2]) + vs2[3]) + …
- Reduction **unordered sum** (including Double Width)
- Reduction **min/max**

vd[0] = op(vs1[0] , vs2[*]) where * are all active elements (mask enabled)

The **w** form writes the double-width result

Vector Reduction Sum

- Reduction of VLEN/SEW=256-bit/16-bit=16 elements
- LMUL>1 can be pipelined



Vector Mask Instructions

Vector **Mask-Register Logical** Instructions

AND, NAND, AND-NOT, OR, NOR, OR-NOT, XOR, XNOR

Vector **mask population count** vpopc

vfirst **find-first**-set mask bit

vmsbf.m **set-before**-first mask bit

vmsif.m **set-including**-first mask bit

vmsof.m **set-only**-first mask bit

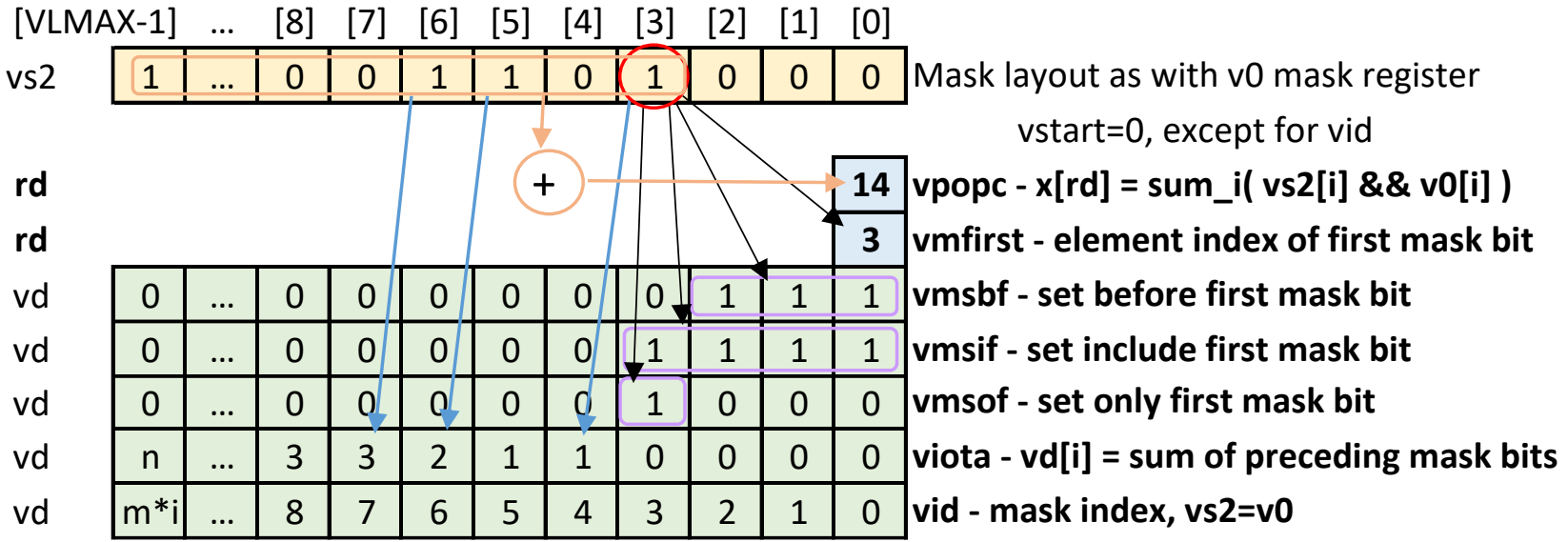
Vector **Iota** Instruction

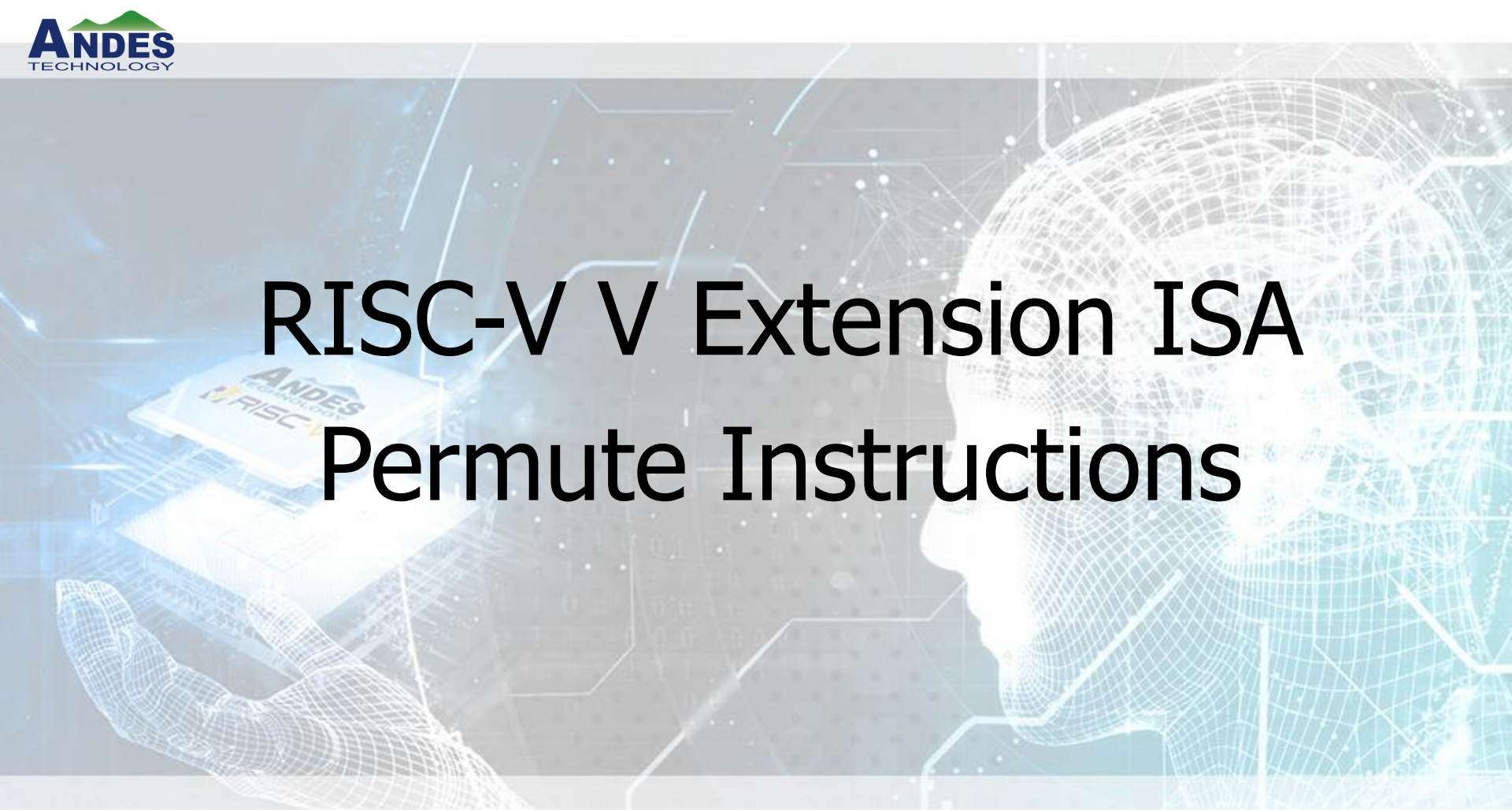
Vector **Element Index** Instruction

Vector Mask Logical

- **Mask logical** instructions, vd = vs2 (logical-op) vs1
  - vmmv.m = vmand.mm vd, vs, vs // copy mask register
  - vmclr.m = vmxor.mm vd, vd, vd // clear mask register
  - vmset.m = vmxnor.mm vd, vd, vd // set mask register
  - vmnot.m = vmnand.mm vd, vs, vs // invert mask bits
- vl and vstart are used as with normal vector operations
- Single vector register operation, LMUL is ignored, bit operations, each bit is for an element

Vector Mask Instructions





RISC-V V Extension ISA
Permute Instructions

Vector Permutation Instructions

Integer Scalar **Move** Instructions – between vector element 0 and x[rd/rs1]

Floating-Point Scalar **Move** Instructions – between vector element 0 and f[rd/rs1]

Vector **Slide** Instructions

Vector Register **Gather** Instructions

Vector **Compress** Instruction

Whole Vector Register **Move**

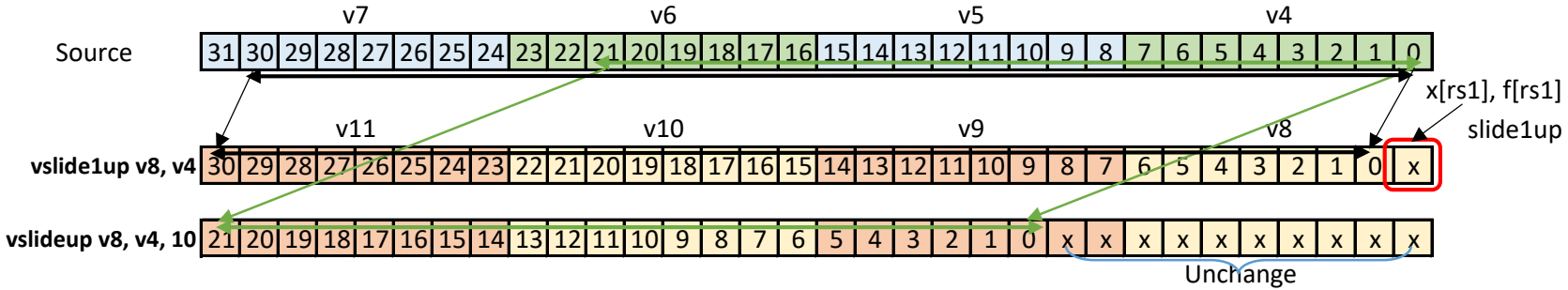
- Move instructions with decoded number of vector registers to copy, ignoring vtype LMUL

Vector Slide Instructions

- vslideup/down – move elements up and down a vector
  - vslide1up.vx vd, vs2, rs1, vm // vd[0]=x[rs1], vd[i+1] = vs2[i]
  - vslideup.vx vd, vs2, rs1, vm // vd[i+rs1] = vs2[i]
  - vslideup.vi vd, vs2, uimm, vm // vd[i+uimm] = vs2[i]
  - vslide1down.vx vd, vs2, rs1, vm // vd[i] = vs2[i+1], vd[vl-1]=x[rs1]
  - vslidedown.vx vd, vs2, rs1, vm // vd[i] = vs2[i+rs1]
  - vslidedown.vi vd, vs2, uimm, vm // vd[i] = vs2[i+uimm]

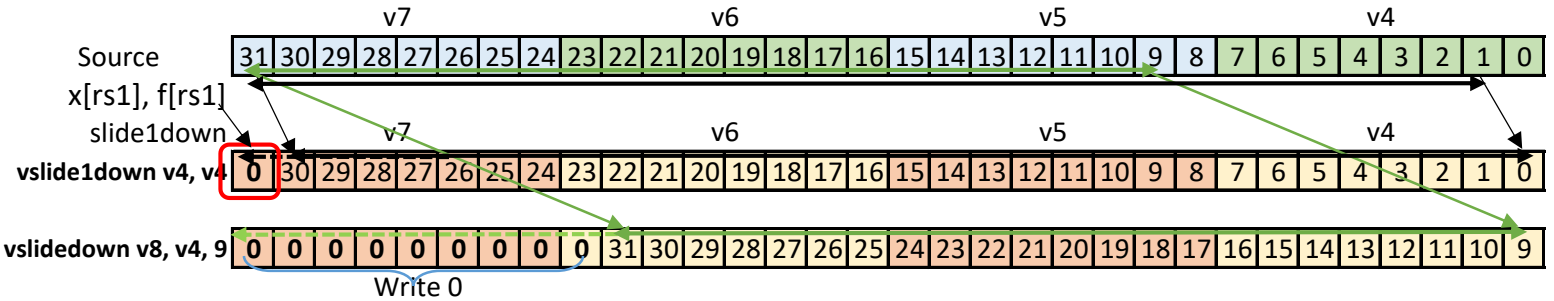
Vector Slideup Instruction

- No source and destination registers overlapping, destination register can be written before the source register is read (RAW)
  - vslideup.vx vd, vs2, rs1, vm // vd[i+rs1] = vs2[i]
  - vslideup.vi vd, vs2, uimm, vm // vd[i+uimm] = vs2[i]
  - vslide1up.vx vd, vs2, rs1, vm // **vd[0]=x[rs1]**, vd[i+1] = vs2[i]
  - Slide1up is from XRF or FRF
  - All lower elements are not modified (or sta)



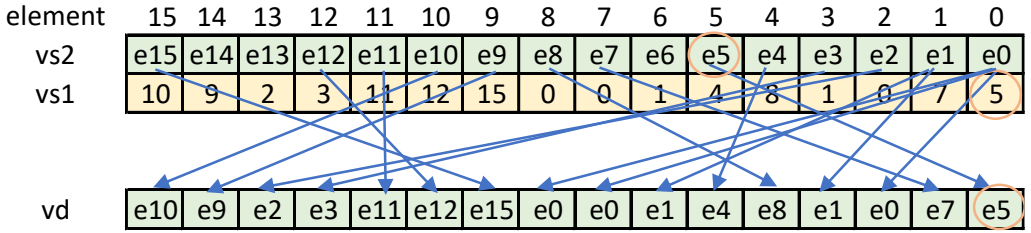
Vector Slidedown Instruction

- Source and destination registers overlapping is okay, source register is read before the destination register is written
  - vslidedown.vx vd, vs2, rs1, vm // vd[i] = vs2[i+rs1]
  - vslidedown.vi vd, vs2, uimm, vm // vd[i] = vs2[i+uimm]
  - vslide1down.vx vd, vs2, rs1, vm // vd[i] = vs2[i+1], **vd[vl-1]=x[rs1]**
  - Slide1down is from XRF or FRF
  - All the upper elements fill in with zero



Vector Register Gather Instructions

- vrgather – read elements from a source vector with index register to write destination vector
  - vrgather.vv vd, vs2, vs1, vm // vd[i] = vs2[vs1[i]]
  - vrgatherei16.vv vd, vs2, vs1, vm // vd[i] = vs2[vs1[i]], vs1 is 16b
  - vrgather.vx vd, vs2, rs1, vm // vd[i] = vs2[rs1]
  - vrgather.vi vd, vs2, uimm, vm // vd[i] = vs2[uimm]
  - If index >= VLMAX, then vd[i] = 0
  - For SEW=8, vs1 can address only 256 elements
  - Can be used in combination with vector unit-stride load/store instead of index load/store

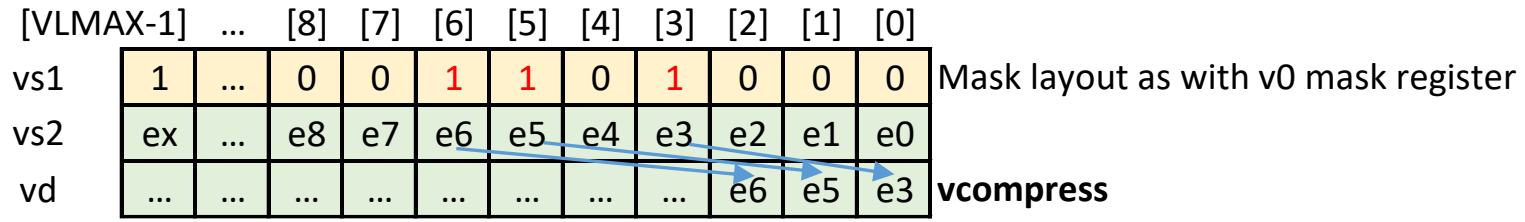


The element number in vd is the same as the index in vs1

Vector Compress Instructions

- vcompress – shift the unmask elements from source vector to destination register
  - vcompress.vm vd, vs2, vs1

    // vd[i] = Compress into vd elements of vs2 where vs1(LSB) is enabled
  - vm=1 (inst[25]), else illegal instruction



Follow Andes, Find Latest Trends

Thank you!

ANDES
TECHNOLOGY