



Accelerator Architectures for Machine Learning (AAML)

Lecture 5: RISC-V Matrix Extension

Tsung Tai Yeh

Department of Computer Science
National Yang-Ming Chiao Tung University



Acknowledgements and Disclaimer

- Slides was developed in the reference with
Joel Emer, Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang, ISCA 2019
tutorial
Efficient Processing of Deep Neural Network, Vivienne Sze, Yu-Hsin
Chen, Tien-Ju Yang, Joel Emer, Morgan and Claypool Publisher, 2020
Yakun Sophia Shao, EE290-2: Hardware for Machine Learning, UC
Berkeley, 2020
CS231n Convolutional Neural Networks for Visual Recognition,
Stanford University, 2020



Outline

- RISC-V Vector Processor Microarchitecture
- RISC-V Matrix Extension
 - Programming Model
 - Instruction Formats



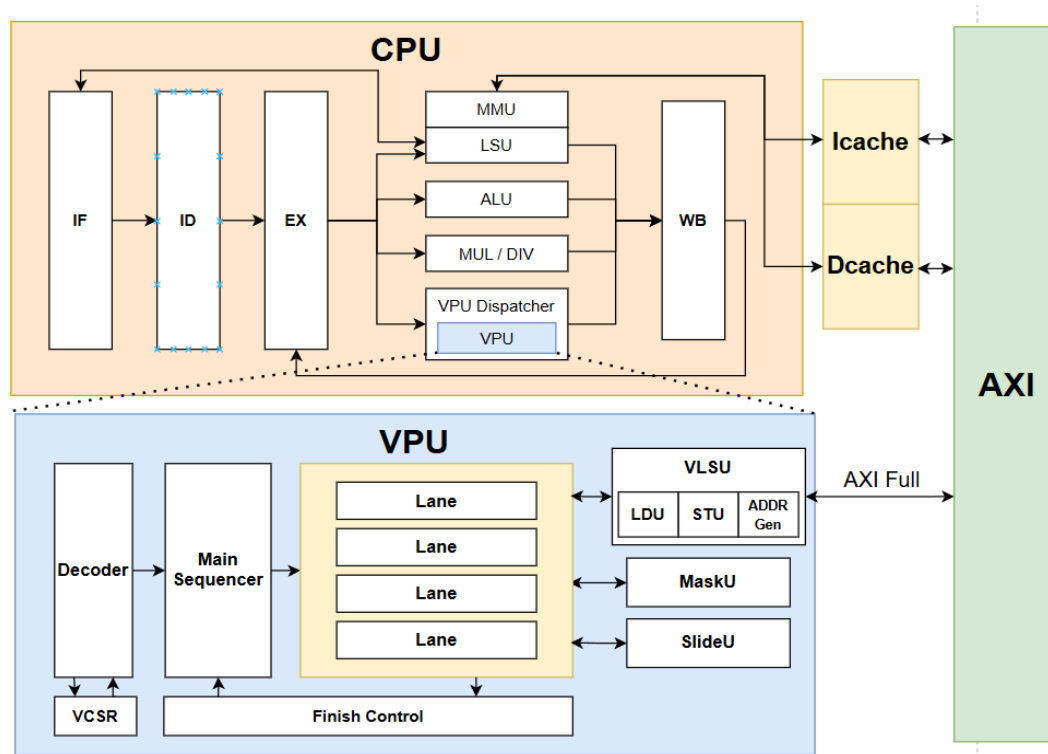
Why Vector Processing?

- **The scalar core processes one element at a time**
 - Fetching, decoding, and executing an instruction per data item wastes cycles when the same operation repeats across thousands of items
- **Vector processing**
 - Lets one instruction operate on many elements at once, amortizing that per-instruction overhead across the whole batch.
- **Multiple parallel vector lanes**
 - Let those elements be computed simultaneously, not just batched — turning data-level parallelism into real speedup.



RISC-V SoC with VPU

- **VPU dispatcher**
 - Distribute Vector ISAs
- **VPU includes**
 - Main sequence
 - Lanes
 - Mask Units
 - Slide Units
 - VLSU

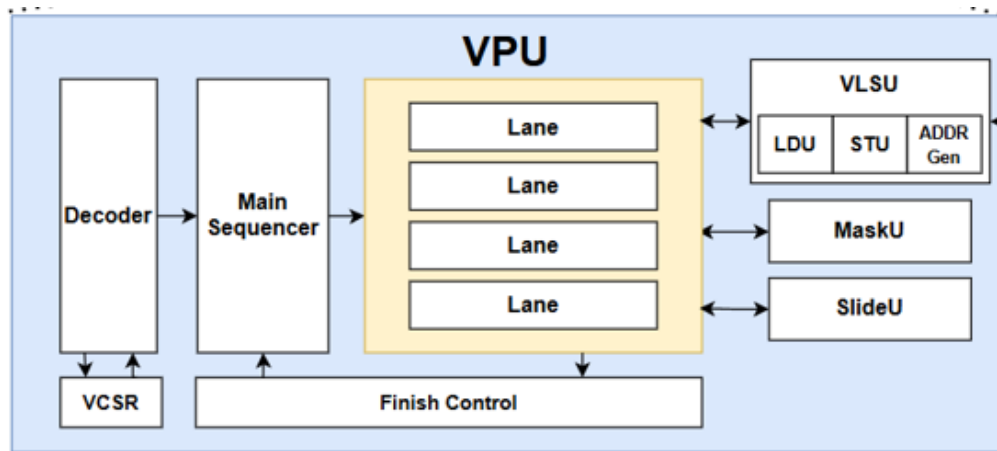




RISC-V VPU Microarchitecture

- **Decoder**

- Re-decodes the vector instruction handed off from the CPU's VPU Dispatcher
- Extracts opcode, source/destination register indices, and other control fields the rest of the VPU needs

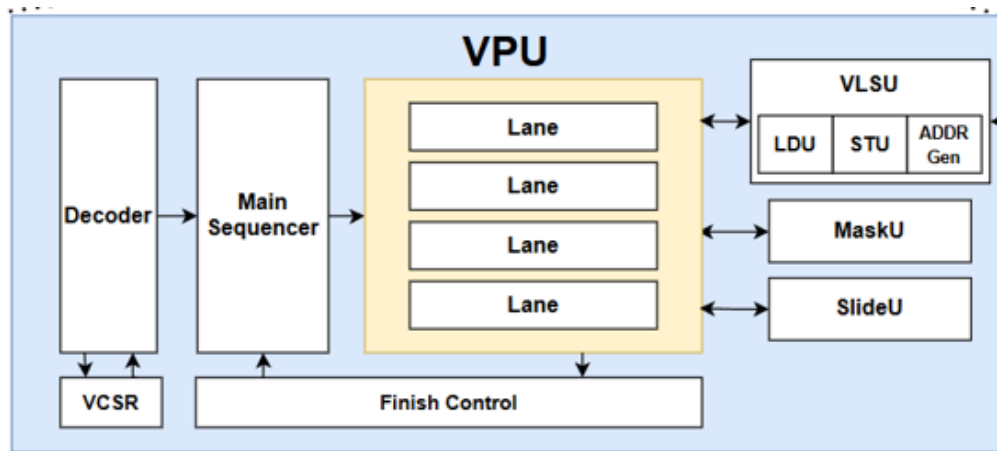




RISC-V VPU Microarchitecture

- **Main Sequencer**

- The VPU's scheduler — decides when and to which functional unit a decoded instruction gets dispatched
- Tracks data dependencies between in-flight instructions to avoid hazards

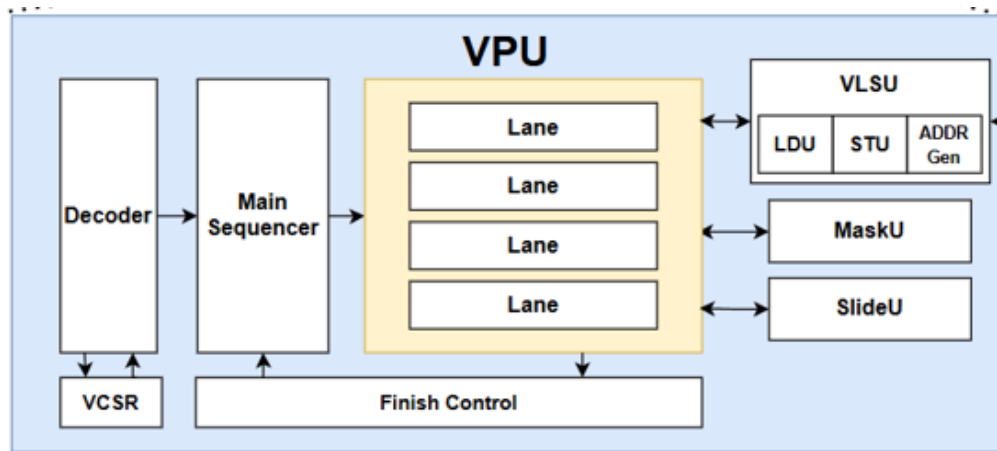




RISC-V VPU Microarchitecture

- **Vector Lanes**

- 4 lanes run in parallel, each handling a slice of the vector's elements, with its own ALU/MUL-DIV and a slice of the VRF
- Connects out to VLSU / MaskU / SlideU for anything that needs cross-lane data

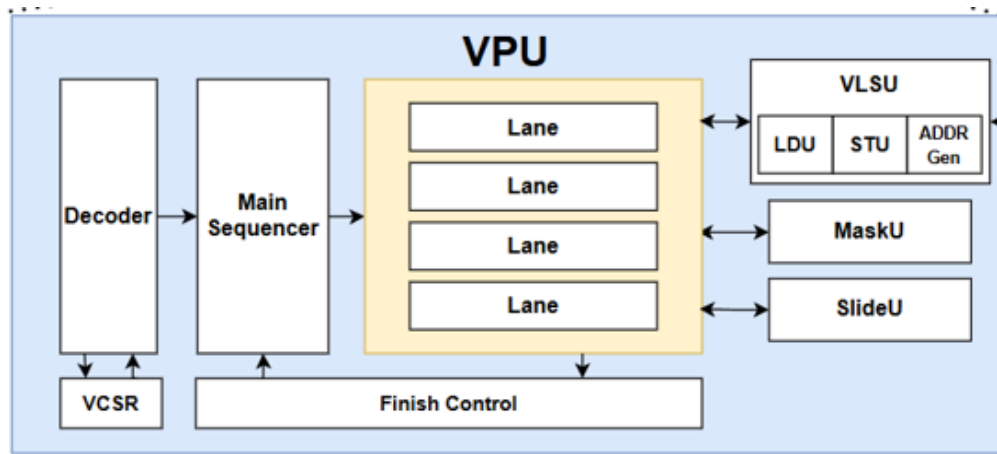




RISC-V VPU Microarchitecture

- **VCSR & Finish Control**

- VCSR holds the current vector configuration — vl, vtype (SEW/LMUL)
- Finish Control aggregates per-lane completion signals; that status is what gets reported back to the CPU's pipeline

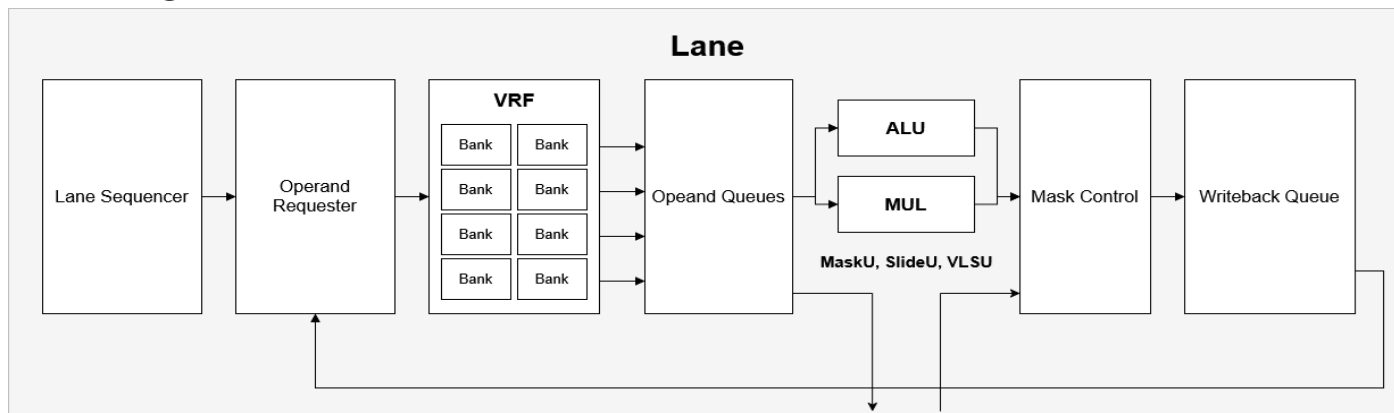




RISC-V VPU Microarchitecture

- **VPU Vector Lanes**

- Process instruction info (ID/OP/SEW)
- Prioritize requests with Arbiter
- Fetch operands from VRF
- Process data via Functional Units
- Manage writebacks with Mask Ctrl





RISC-V VPU Microarchitecture

SEW(Selected Element Width): represents the width of the currently selected element within the vector register

● VPU Vector Lanes — VRF

- LSU rearranges the sequence of elements, ensuring correct ordering
- Special organization improves VRF utilization within each lane by efficiently managing unused tail elements

VLEN = 128

	Lane 3				Lane 2				Lane 1				Lane 0			
sew = 32	3				2				1				0			
sew = 16	7	3			6	2			5	1			4	0		
sew = 8	15	11	7	3	14	10	6	2	13	9	5	1	12	8	4	0

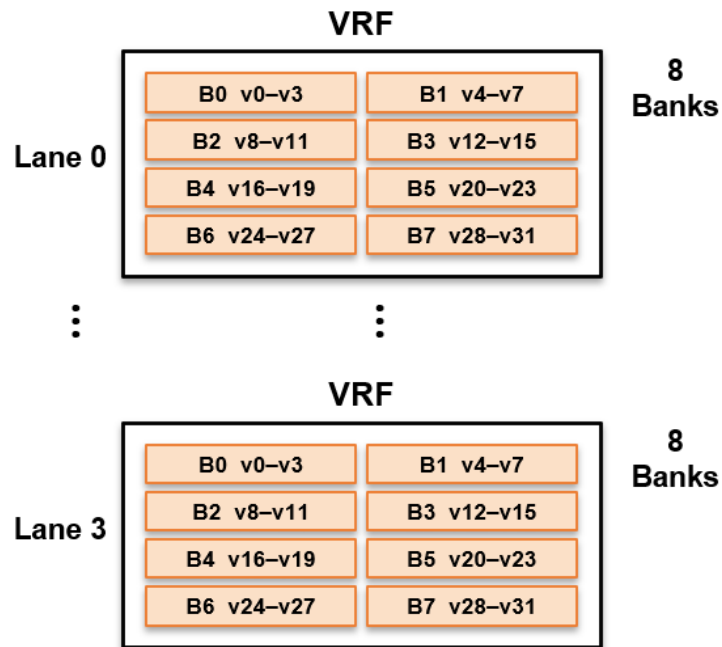
	Lane 3				Lane 2				Lane 1				Lane 0			
Lane Organization	15	11	7	3	14	10	6	2	13	9	5	1	12	8	4	0
Natural	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0



RISC-V VPU Microarchitecture

● VPU Vector Lanes — VRF

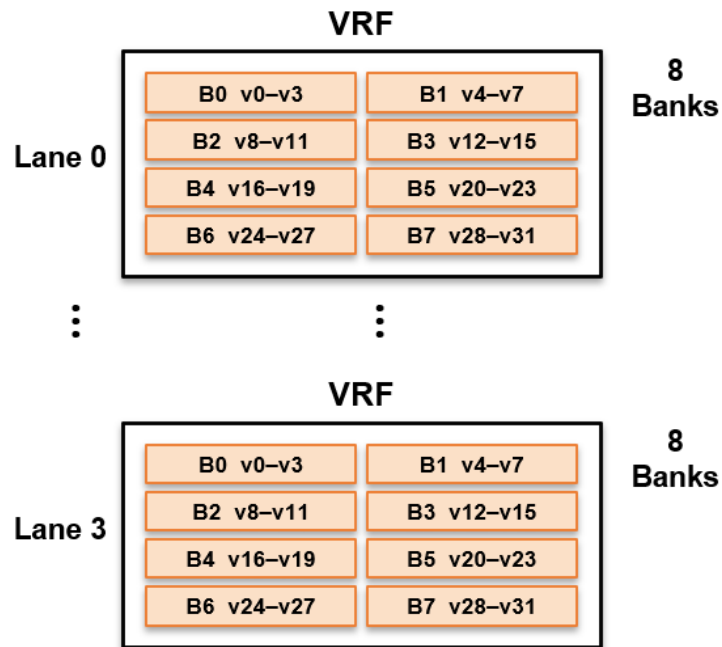
- VPU has 32 vector registers (v0–v31), each VLEN = 1024 bits.
- Total VRF = 32×1024 bits = 4 KB, split evenly across the 4 lanes $\rightarrow$ 1 KB per lane.
- SEW sets the element width for this instruction —
- e.g. SEW=16 $\rightarrow 1024 \div 16 = 64$ elements max (VLMAX).





RISC-V VPU Microarchitecture

- **VPU Vector Lanes — VRF**
 - vl (vector length) =
 - Elements actually processed = $\min(\text{requested count}, \text{VLMAX})$ — not always equal to VLMAX.





RISC-V VPU Microarchitecture

```
#include <riscv_vector.h>

void vec_add_i16(int16_t *A, int16_t *B, int16_t *C, int n) {
    while (n > 0) {
        size_t vl = __riscv_vsetvl_e16m1(n);
        vint16m1_t va = __riscv_vle16_v_i16m1(A, vl);
        vint16m1_t vb = __riscv_vle16_v_i16m1(B, vl);
        vint16m1_t vc = __riscv_vadd_vv_i16m1(va, vb, vl);
        __riscv_vse16_v_i16m1(C, vc, vl);

        n -= vl;  A += vl;  B += vl;  C += vl;
    }
}

int16_t A[85], B[85], C[85];
vec_add_i16(A, B, C, 85);      // <- n = 85
```

How each setting is decided:

SEW = 16

the array type is `int16_t` — each element is a 16-bit value

LMUL = 1

the default choice: no need to group multiple registers together for this loop

VLMAX = VLEN ÷ SEW = 1024 ÷ 16 = 64

the most elements one instruction can touch, given SEW=16 and LMUL=1

Iteration 1: $n=85$, so $vl = \min(85, 64) = 64$ — processes elements 0–63, hits VLMAX exactly

Iteration 2: $n = 85 - 64 = 21$ remain, so $vl = \min(21, 64) = 21$ — this leftover is exactly the $vl=21$ case examined on the next slide

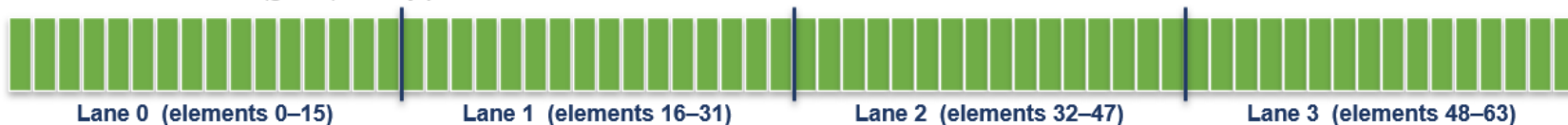


RISC-V VPU Microarchitecture

- In 1st iteration

`__riscv_vadd_vv_i16m1(va, vb, vl)` — loop's 1st call: $SEW=16$, $LMUL=1$, $VLMAX=64$, $vl=64$

$vl = 64$ active elements (green) — fully packed



Per-lane breakdown

Lane	Elements it owns	Active this instruction	Status
Lane 0	0 – 15	0 – 15 (all 16)	Fully active
Lane 1	16 – 31	16 – 31 (all 16)	Fully active
Lane 2	32 – 47	32 – 47 (all 16)	Fully active
Lane 3	48 – 63	48 – 63 (all 16)	Fully active

All 4 lanes have work this instruction — $vl=64$ exactly fills $VLMAX$, so every lane is fully active with no tail.



RISC-V VPU Microarchitecture

- In 2nd iteration

`__riscv_vadd_vv_i16m1(va, vb, vl)` — loop's 2nd call: $SEW=16$, $LMUL=1$, $vl = \min(85-64, VLMAX) = 21 = 21$

$vl = 21$ active elements (green)

tail = elements 21–63 (gray, not processed)



Per-lane breakdown

Lane	Elements it owns	Active this instruction	Status
Lane 0	0 – 15	0 – 15 (all 16)	Fully active
Lane 1	16 – 31	16 – 20 (5 of 16)	Partly active + tail
Lane 2	32 – 47	none	All tail — idle
Lane 3	48 – 63	none	All tail — idle

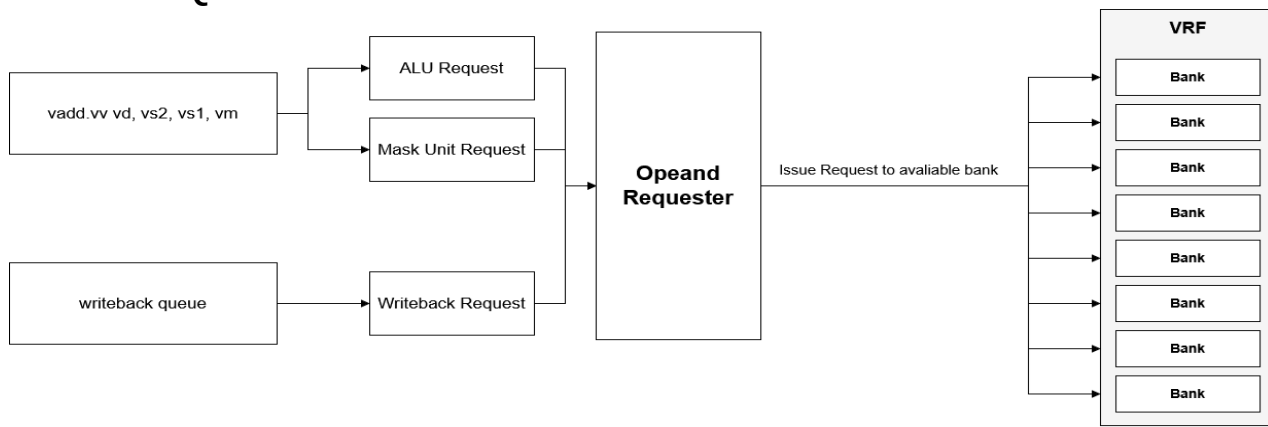
Only Lane 0 and Lane 1 have any work this instruction — Lane 2 and Lane 3 sit idle, since none of their elements fall within $vl=21$.



RISC-V VPU Microarchitecture

- **VPU Operand (Register) Arbiter**

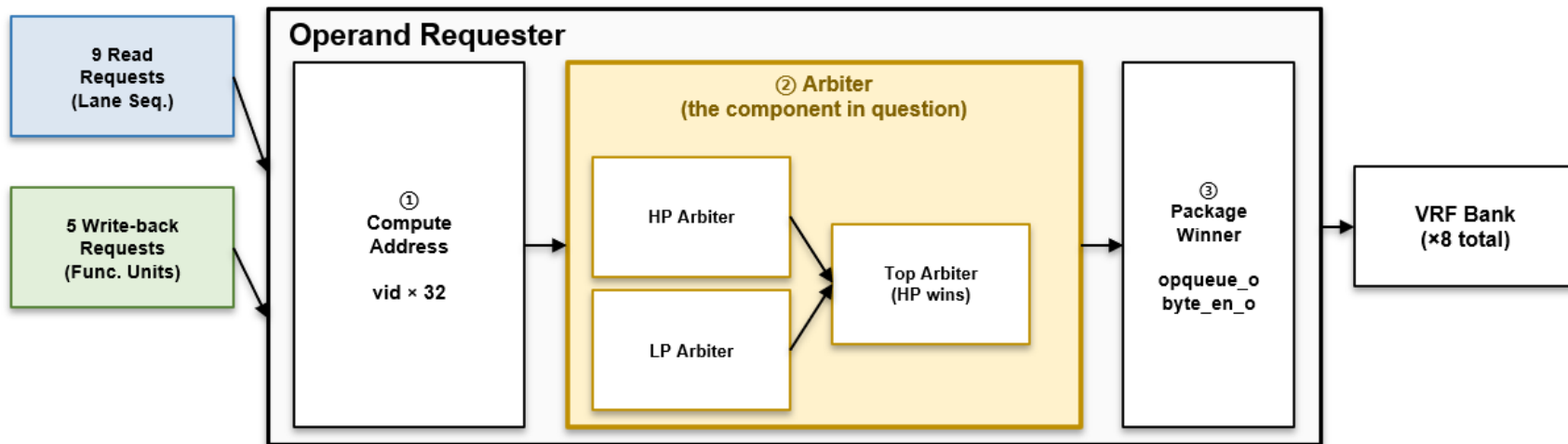
- Resolves structural hazards when multiple units compete for the VRF
- Routes the accepted request to an available memory bank, maximizing bandwidth
- Fixed-priority policy balances forwarding new operands vs. clearing the Writeback Queue





RISC-V VPU Microarchitecture

- Inside the operand requester



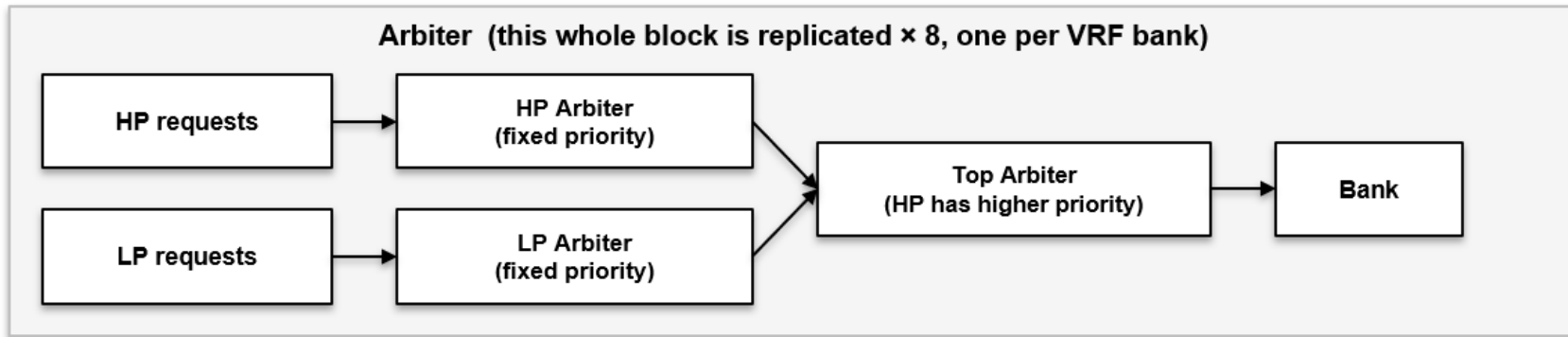
- The Arbiter (gold) is only ONE of three things happening inside the Operand Requester — it doesn't act alone.
- Before arbitration, every read request already has its address computed (①); the Arbiter itself never touches addresses, only decides whose turn it is.
- After arbitration, the winner still needs routing to the right Operand Queue and a byte-write mask (③) before the bank can actually be touched.



RISC-V VPU Microarchitecture

- **VPU Operand (Register) Arbiter**

- The HP Arbiter, LP Arbiter, and Top Arbiter are all the SAME arbiter module, instantiated 3 times — not three different designs.
- “Bank” here means one of 8 — there are really 8 independent copies of this whole HP/LP/Top structure, 24 rr_arbiter instances in total.





RISC-V VPU Microarchitecture

- VPU Operand (Register) Arbiter

Priority: high ↓ low

#	HP (High Priority)	Description
1	ALUA	vs2 → ALU
2	ALUB	vs1 / imm → ALU
3	MULA	vs2 → MUL/DIV
4	MULB	vs1 / imm → MUL/DIV
5	ADDRGENA	vs2 → VLSU address gen
6	StA	store data → VLSU Store Unit
7	MaskM	fixed v0 → MaskU predication

#	LP (Low Priority)	Description
1	MaskB	vs2 → MaskU exec
2	SlideAddrGenA	vs2 → SlideU
3	ALU-wb	ALU result → VRF
4	MUL-wb	MUL/DIV result → VRF
5	MaskU-wb	MaskU result → VRF
6	LDU-wb	Load data → VRF
7	SlideU-wb	Slide result → VRF

Final stage: Top Arbiter

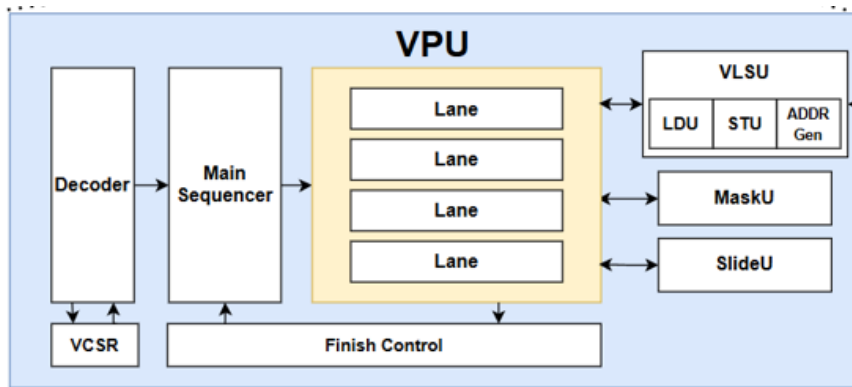
Whatever wins HP and whatever wins LP are both fed into a third arbiter (Top), which picks between the two winners — HP always wins when both are present.



RISC-V VPU Microarchitecture

- **Slide Unit (SLDU)**

- Each vector register is physically split across 4 lanes — a lane's ALU can only reach its own slice.
- vslideup/vslidedown need to shift elements across lane boundaries — data which a single ALU can't reach.
- SlideU is the dedicated datapath built specifically to move bytes between lanes.





RISC-V VPU Microarchitecture

- How does SLDU work?

before:



after shift:



```
// vslidedown.vi vd, vs2, 1  
// rule: vd[i] = vs2[i + 1], for i = 0 .. vl-2  
//       vd[vl-1] = unspecified (tail)
```

```
vs2 = [10, 20, 30, 40, 50, 60, 70, 80]
```

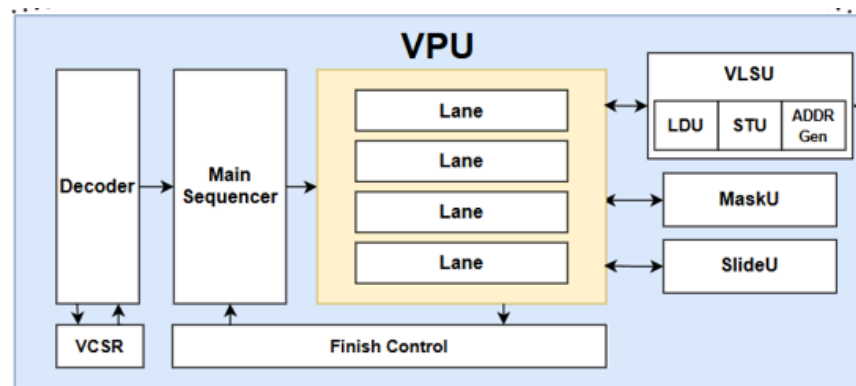
```
vd[0] = vs2[1] = 20  
vd[1] = vs2[2] = 30  
vd[2] = vs2[3] = 40  
vd[3] = vs2[4] = 50  
vd[4] = vs2[5] = 60  
vd[5] = vs2[6] = 70  
vd[6] = vs2[7] = 80  
vd[7] = vs2[8] = ?    // out of range -> tail
```



RISC-V VPU Microarchitecture

- **Mask Unit (MASKU)**

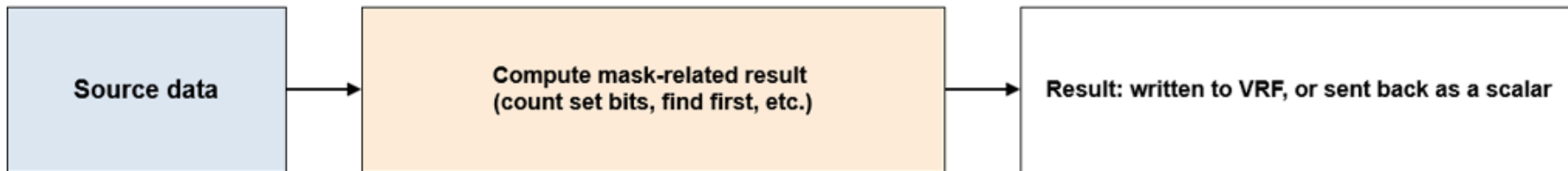
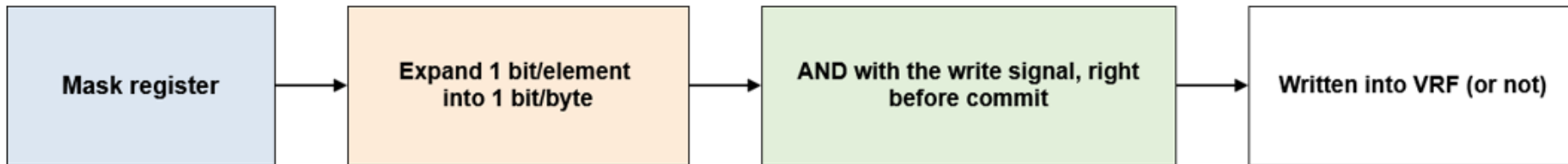
- RVV predicates instructions with a mask (v0) — only elements where the mask bit is 1 get written back.
- MaskU converts that 1-bit-per-element mask into a byte-level write-enable signal, applied right before the Writeback Queue commits.





RISC-V VPU Microarchitecture

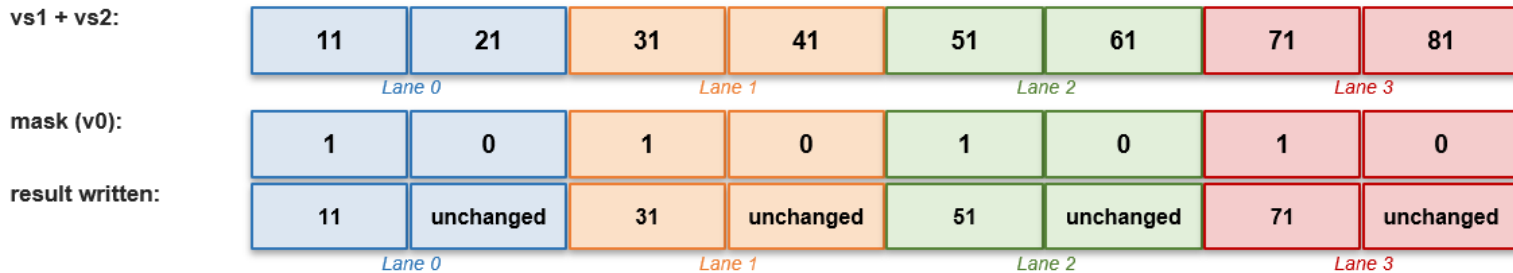
- How does MASKU work?





RISC-V VPU Microarchitecture

- How does MASKU work?



```
// vadd.vv vd, vs2, vs1, v0.t
// rule: vd[i] = mask[i] ? (vs1[i] + vs2[i]) : vd[i]    (old value kept)

vs1 = [1, 1, 1, 1, 1, 1, 1, 1]
vs2 = [10, 20, 30, 40, 50, 60, 70, 80]
mask = [1, 0, 1, 0, 1, 0, 1, 0]

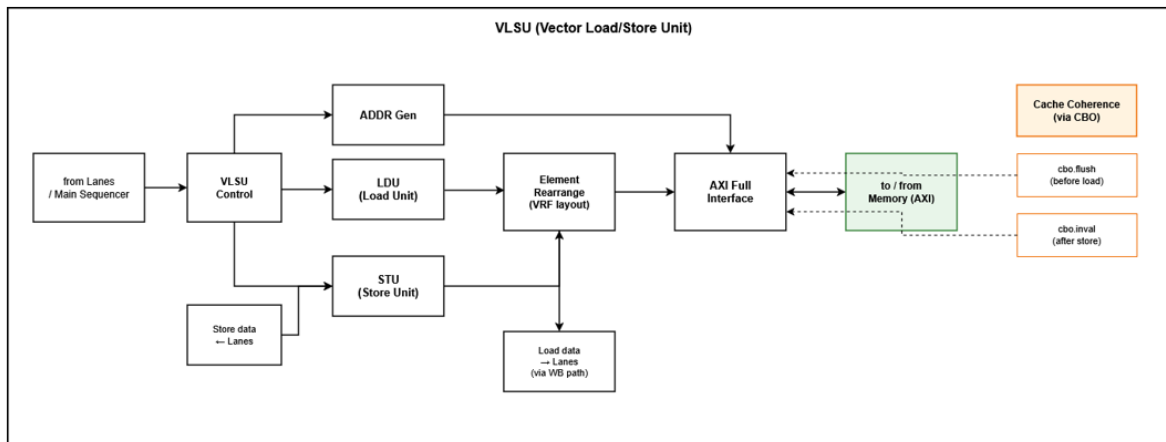
vd[0] = mask[0] ? 1+10 : vd[0] = 11
vd[1] = mask[1] ? 1+20 : vd[1] = unchanged
vd[2] = mask[2] ? 1+30 : vd[2] = 31
vd[3] = mask[3] ? 1+40 : vd[3] = unchanged
vd[6] = mask[6] ? 1+70 : vd[6] = 71
vd[7] = mask[7] ? 1+80 : vd[7] = unchanged
```



RISC-V VPU Microarchitecture

- **Vector Load/Store Unit (VLSU)**

- Handles all vector memory traffic — LDU (load) + STU (store) + ADDRGEN (address generation for unit-stride / strided / indexed access)
- Own AXI Full path straight to DRAM — bypasses CPU cache, which is why cache-coherence (CBO) handling is needed





RISC-V VPU Microarchitecture

- Vector Load/Store Unit (VLSU)

Component	Description
Input / Control	
from Lanes / Main Seq.	Triggers a load or store instruction
VLSU Control	Decides load vs. store; dispatches to ADDR Gen, LDU, or STU
Address & Data Path	
ADDR Gen	Computes the memory address — unit-stride, strided, or indexed
LDU (Load Unit)	Reads data from memory, hands it to Element Rearrange
STU (Store Unit)	Writes lane data to memory; wins priority over LDU same cycle
Store data ← Lanes	The operand data STU writes out, sourced from all 4 lanes
Element Rearrange	Arbitrates the 4 lanes, splits the winner's word by EEW
Load data → Lanes (WB path)	Shares the same writeback queue as ALU/MUL results
External Memory Interface	
AXI Full Interface	The bus interface between VLSU and external memory
to / from Memory (AXI)	The actual DRAM, reached only through this AXI link
Cache Coherence (CBO)	
Cache Coherence note	VLSU bypasses the CPU D-cache entirely via AXI
cbo.flush (before load)	Software flushes dirty D-cache lines so VLSU reads fresh data
cbo.inval (after store)	Software invalidates stale D-cache lines after VLSU writes



RISC-V Matrix Extensions

- **Why Matrix Extensions?**
- **Driven by AI:** GEMM is one of the most common operations in AI
- **Matrix multiplication is a bottleneck in many AI algorithms**
 - Optimal Matrix execution on processors required to boost performance
- **Other ISAs already implement them**
 - IBM Matrix-Multiply Assist (2020)
 - Intel Advanced Matrix Extension (2020)
 - Arm Scalable Matrix Extension (2021)

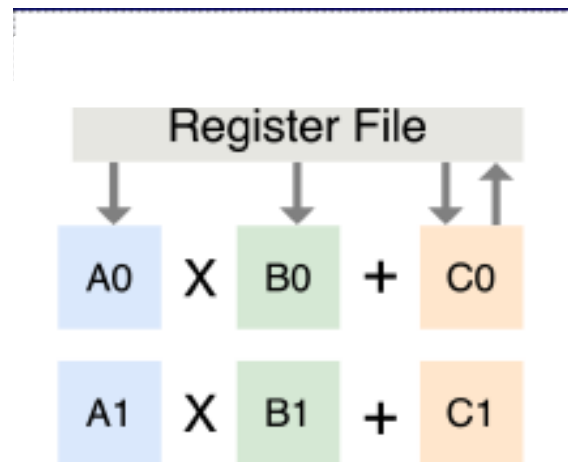


RISC-V Matrix Extensions



Vector processing

N elements – N operations



Matrix processing

N^2 elements – N^3 operations



RISC-V Matrix Extensions

- **Matrix Extension Advantages?**
 - Reduce memory bandwidth requirement
 - Less data to generate more operations (data memory bandwidth)
 - One instruction contains N^3 operations (instruction memory bandwidth)
 - Inherently more power efficiency with less data movement



RISC-V Matrix Extensions

- **Two types of standard matrix extensions being developed:**
 - **Integrated Matrix Extension (IME)**
 - Matrix operations using vector registers as input and output state
 - **Attached Matrix Extension (AME)**
 - Additional matrix state used to hold accumulators to support higher throughput



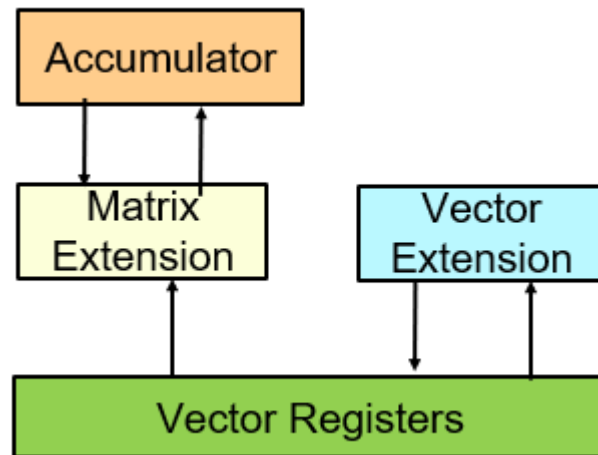
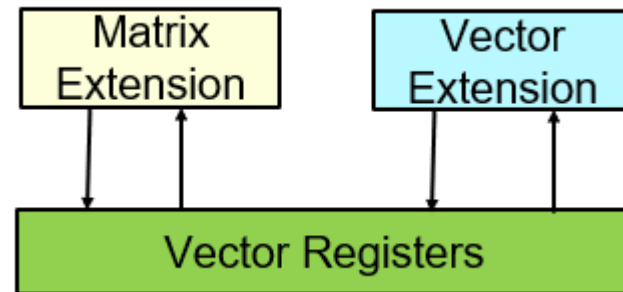
RISC-V Matrix in CPU

- **Fully Integrated Facility**

- Reuse vector registers for source operands
- Reuse vector registers for accumulators

- **Hybrid Facility**

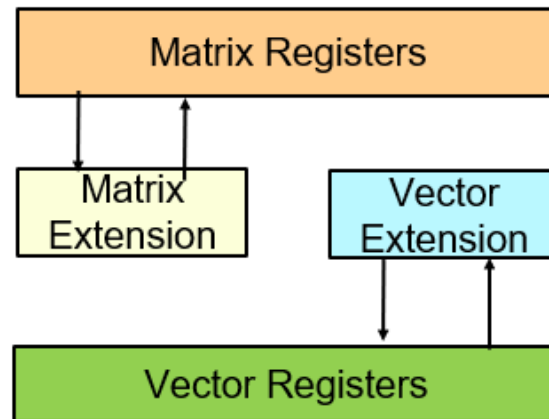
- Reuse vector registers for source operands
- Independent accumulator registers





RISC-V Matrix in CPU

- **Attached Facility**
 - Additional matrix registers
- **What others do**

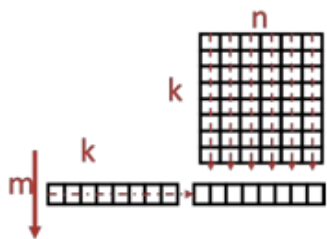


<i>Reuse vector registers for source operands and accumulators</i>	SiFive Intelligence Extension
<i>Reuse vector registers for source operands independent accumulator register</i>	Power MMA, ARM SME
<i>independent source operands and accumulators</i>	Intel AMX, Apple AMX, Stream computing Matrix extension



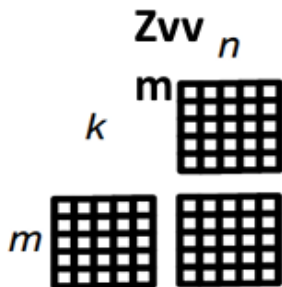
RISC-V Matrix Extensions

Dot-product



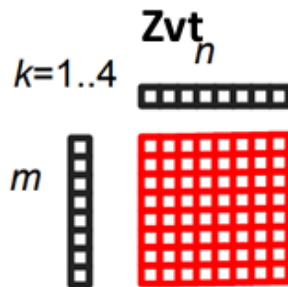
- No new state
- Just one additional dot-p insn

IME



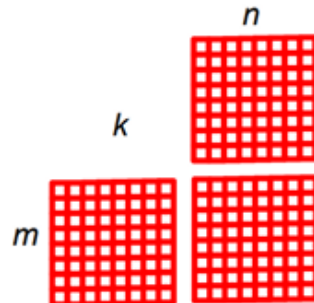
- No new state
- Integrated within vector architecture
- VRF interpreted as 2D tiles

VME



- New state for accumulators
- Within RVV
- Outer-product oriented Matrix engine

AME



- Completely separate co-processor
- Dedicated state for input matrix regs and accumulators
- Can work without RVV



RISC-V Matrix Extensions

- **The Archi Difference is where the data lives**

	Operands A, B	Accumulator C	Abstraction
BDP	RVV regs	RVV regs	vector-matrix primitives
IME	RVV regs (tiled)	RVV regs (tiled)	tile GEMM
VME	RVV regs	Acc regs	tile GEMM
AME	Matrix regs	Acc regs	tile GEMM

- This affects how you shape you matmul kernel code

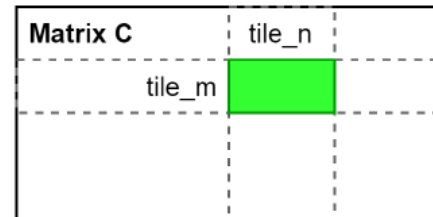
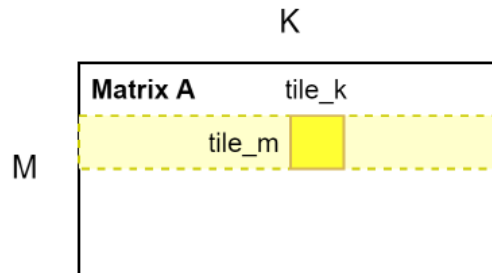
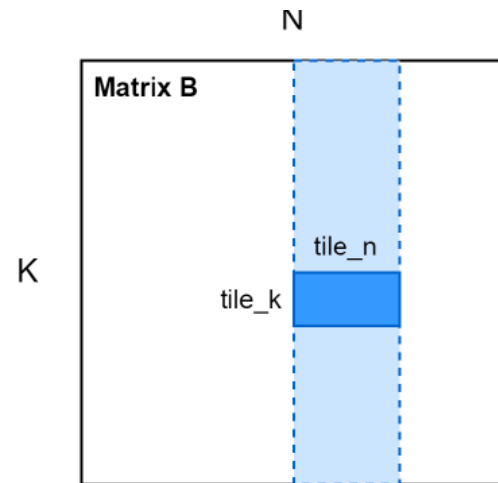
	New state	Micro-archi hints
BDP	No	simplicity, minmal uarchi change
IME	just minimal control	richer abstraction than BDP
VME	dedicated accumulators	aggressive performance boost (outerp-product oriented..)
AME	new matrix state (input + accum.)	a distinct subsystem – not necessarily coupled with RVV

- Not mutually exclusive



Tiled Matrix Multiplication

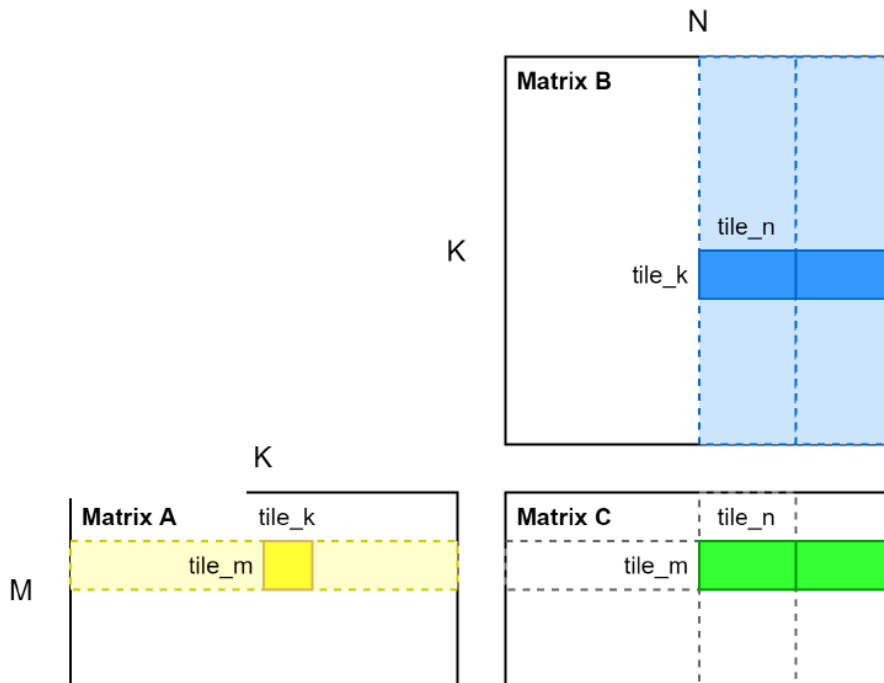
```
for (i=0; i<m; i+=tile_m) {  
  for (j=0; j<n; j+=tile_n) {  
  
    for (s=0; s<k; s+=tile_k) {  
      c_tile[i,j] += a_tile[i,s] * b_tile[s,j];  
    }  
  }  
}
```





RISC-V Matrix Extensions

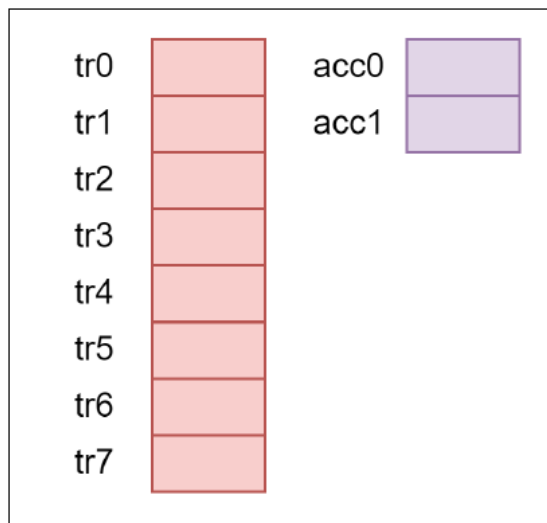
- **Memory access friendly**
 - Taking advantage of unit-stride performance
- **Explore data reuse**
 - Load an element in a tile once while use several times
 - Load an entire tile once and use several times in parallel implementations



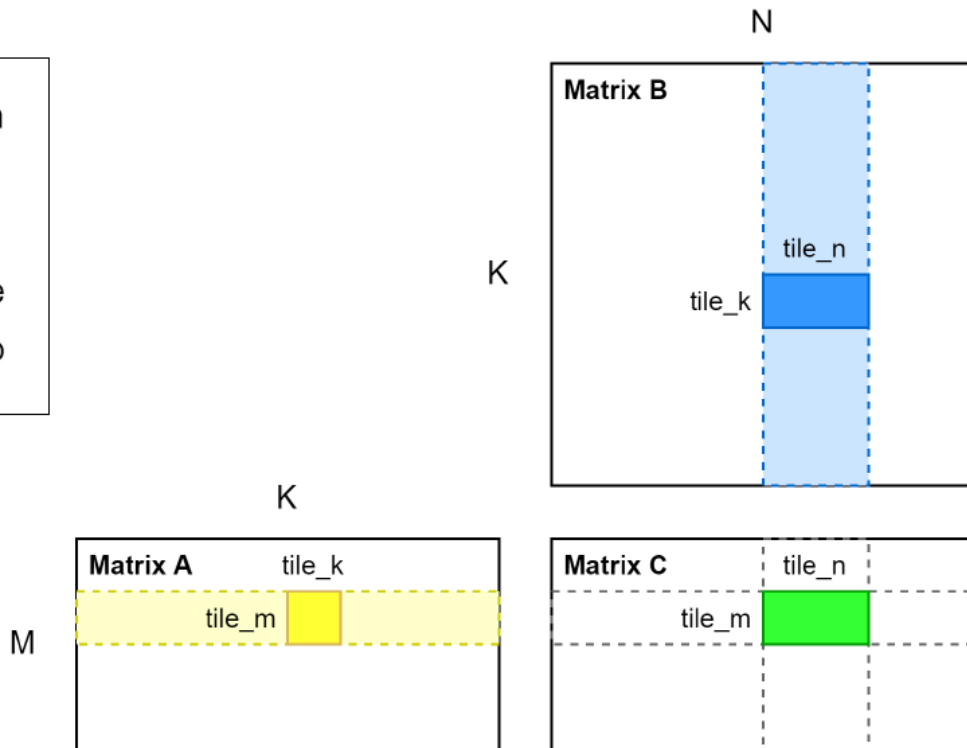
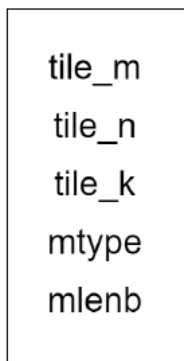


Registers and CSRs

Registers



CSRs





CSRs

- **tile_m/tile_k/tile_n: Matrix the configure registers**
 - Unsigned integers specifying the shapes for tiled matrix
 - Updated by **msettile[m|k|n]{i}** instructions
- **mtype: Matrix type register**
 - Updated by **msettype{i}** instructions

Table 2. **mtype** register layout

Bits	Name	Description
XLEN-1	mill	Illegal value if set
XLEN-2:4	0	Reserved if non-zero
3	maccq	Support quad-width accumulator element
2:0	msew[2:0]	Selected element width (SEW) setting



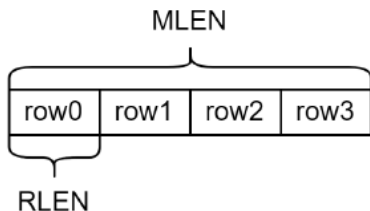
Implementation-defined Constant Parameters

- **ELEN**: Maximum bits of a matrix element, $ELEN > 8$
 - **MLEN**: Bits in a single matrix tile register, $MLEN \leq 2^{32}$
 - **RLEN**: Bits in a row of a single matrix tile register, $RLEN \leq 2^{16}$
-
- $ELEN < RLEN < MLEN$
 - Support matrix tile size from 2×2 to $2^{16} \times 2^{16}$



Tile and Accumulator Registers

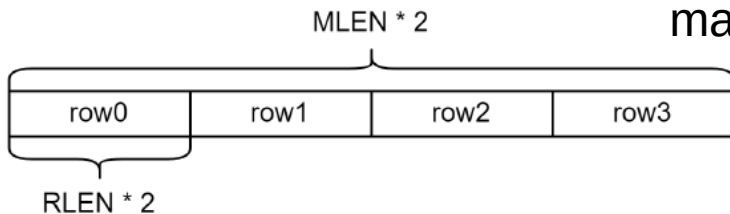
A Tile Register



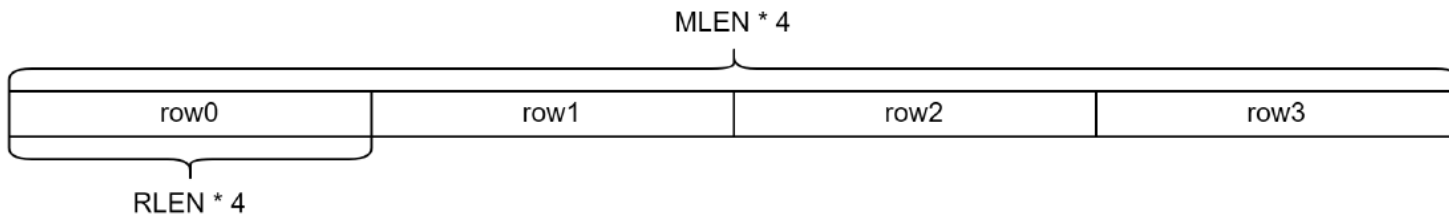
MLEN: Bits in a single matrix tile register, $MLEN \leq 2^{32}$

RLEN: Bits in a row of a single matrix tile register, $RLEN \leq 2^{16}$

An Acc Register
double-width acc type
implementation



An Acc Register
quad-width acc type
implementation





Instruction Formats

- Using the reserved instruction fields

- Opcode = 1110111

inst[4:2]	000	001	010	011	100	101	110	111
inst[6:5]								(> 32b)
00	LOAD	LOAD-FP	custom-0	MISC-MEM	OP-IMM	AUIPC	OP-IMM-32	48b
01	STORE	STORE-FP	custom-1	AMO	OP	LUI	OP-32	64b
10	MADD	MSUB	NMSUB	NMADD	OP-FP	reserved	custom-2/rv128	48b
11	BRANCH	JALR	reserved	JAL	SYSTEM	reserved	custom-3/rv128	≥ 80b

Table 24.1: RISC-V base opcode map, inst[1:0]=11



Instruction Formats

- **Opcode = 1110111**
- **Type by inst[14:12]**
 - 111: Configuration
 - 110: Arithmetic
 - 101: Data Move
 - 000-011: Load & Store

Configuration instructions, funct3 = 111

31 28	27 20	19 15	14 12	11 7	6 0
funct4	imm13			funct3	rd
funct4	00000000	rs1	funct3	rd	1110111

Load & Store instructions, eew = 000 - 011

31 26	25	24 20	19 15	14 12	11 7	6 0
funct6	ls	rs2	rs1	eew	md	1110111

Arithmetic & Type-Convert instructions, funct3 = 110

31 26	25	24 20	19 15	14 12	11 7	6 0
funct6	fp	ms2	ms1	funct3	md	1110111

Data Move instructions, funct3 = 101

31 26	25	24 20	19 15	14 12	11 7	6 0
funct6	di	rs2	ms1	funct3	md	1110111



Configuration-Setting Instructions – msettype{i}

- **msettype{i} instructions**
 - Set the mtype CSR based on their arguments

```
msettypei rd, mtypei    # rd = new mtype, mtypei = new mtype setting  
msettype rd, rs1        # rd = new mtype, rs1 = new mtype value
```

Table 3. **mtype** register layout

Bits	Name	Description
XLEN-1	mill	Illegal value if set
XLEN-2:4	0	Reserved if non-zero
3	maccq	Support quad-width accumulator element
2:0	msew[2:0]	Selected element width (SEW) setting



Configuration-Setting Instructions – msettype{i}

- **msettype{i} instructions**

- Set the mtvpe CSR based on their arguments

Suggested assembler names used for msettypei mtvpei immediate

e8 # SEW = 8b

e16 # SEW = 16b

e32 # SEW = 32b

e64 # SEW = 64b

accq # support 32-bit accumulator element

Examples:

msettypei t0, e8 # SEW = 8

msettypei t0, e32 # SEW = 16

msettypei t0, e8, accq # SEW = 8, support 32-bit accumulator

element



Configuration-Setting Instructions – msettile [m|k|n] {i}

- **msettile [m|k|n] {i}** instructions
 - Set the tile_m/tile_k/tile_n CSRs based on their arguments.

```
msettilemi rd, mleni      # rd = new tile_m, mleni = ATM
msettilem rd, rs1         # rd = new tile_m, rs1 = ATM

msettileki rd, mleni      # rd = new tile_k, mleni = ATN
msettilek rd, rs1         # rd = new tile_k, rs1 = ATN

msettileni rd, mleni      # rd = new tile_n, mleni = ATK
msettilen rd, rs1         # rd = new tile_n, rs1 = ATK
```

ATM denotes the Application Tile length M (or application matrix M dimension), representing the configured/requested row dimension size for matrix tiles during execution



Configuration-Setting Instructions – msettile [m|k|n] {i}

- **TMMAX, TKMAX, TNMAX**

- represent the maximum shapes of matrix tiles that could be stored in matrix registers
 - $TMMAX = MLEN / RLEN$
 - $TKMAX = \min(MLEN / RLEN, RLEN / SEW)$
 - $TNMAX = RLEN / SEW$

- **MLEN=256, RLEN=64**

- When $SEW=8b$, $TMMAX=4$, $TKMAX=4$, $TNMAX=8$ # 4x4x8 8bit matmul
- When $SEW=16b$, $TMMAX=4$, $TKMAX=4$, $TNMAX=4$ # 4x4x4 16bit matmul
- When $SEW=32b$, $TMMAX=4$, $TKMAX=2$, $TNMAX=2$ # 4x2x2 32bit matmul



Configuration-Setting Instructions – msettile [m|k|n] {i}

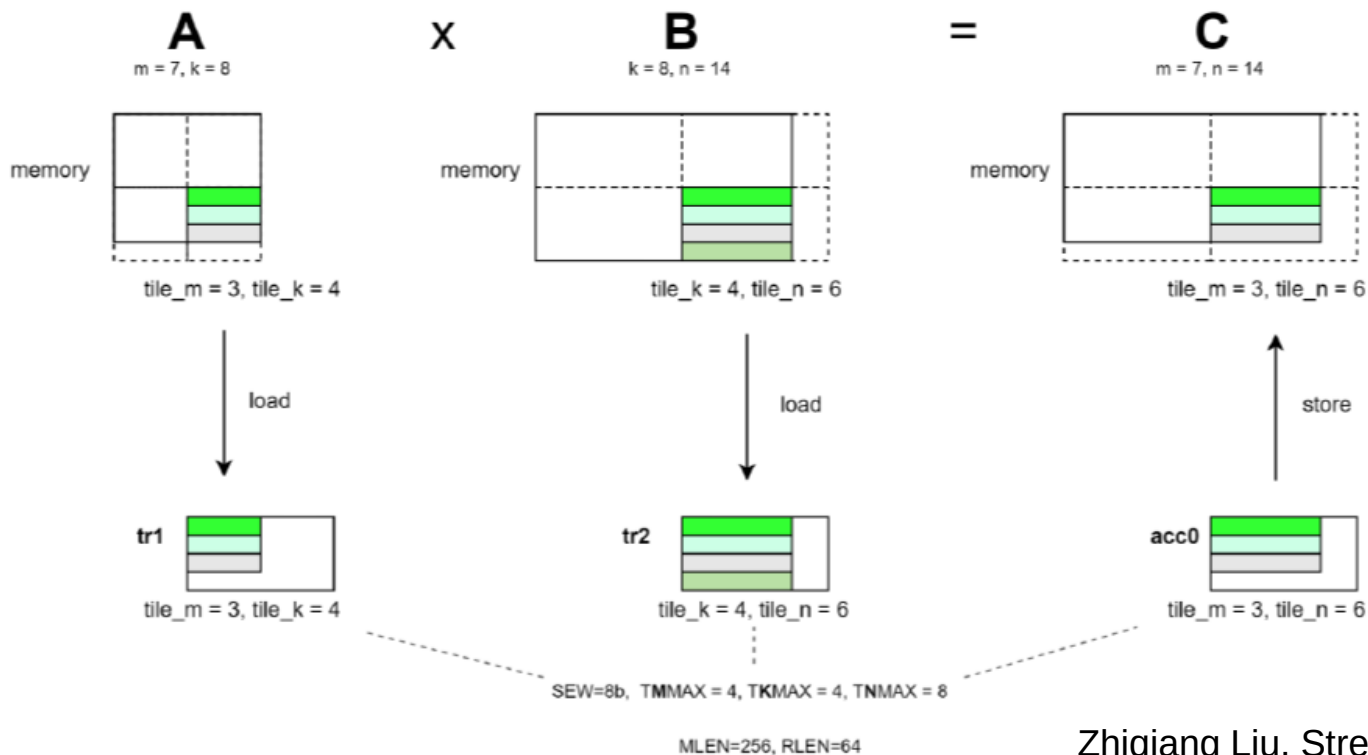
- **msettile [m|k|n] {i} instructions**

- Set tile_m tile_k tile_n obeying the following constraints

1. `tile_m = ATM` if `ATM <= TMMAX`
2. `ceil(ATM / 2) |<= tile_m <= TMMAX` if `ATM < (2 * TMMAX)`
3. `tile_m = TMMAX` if `ATM >= (2 * TMMAX)`



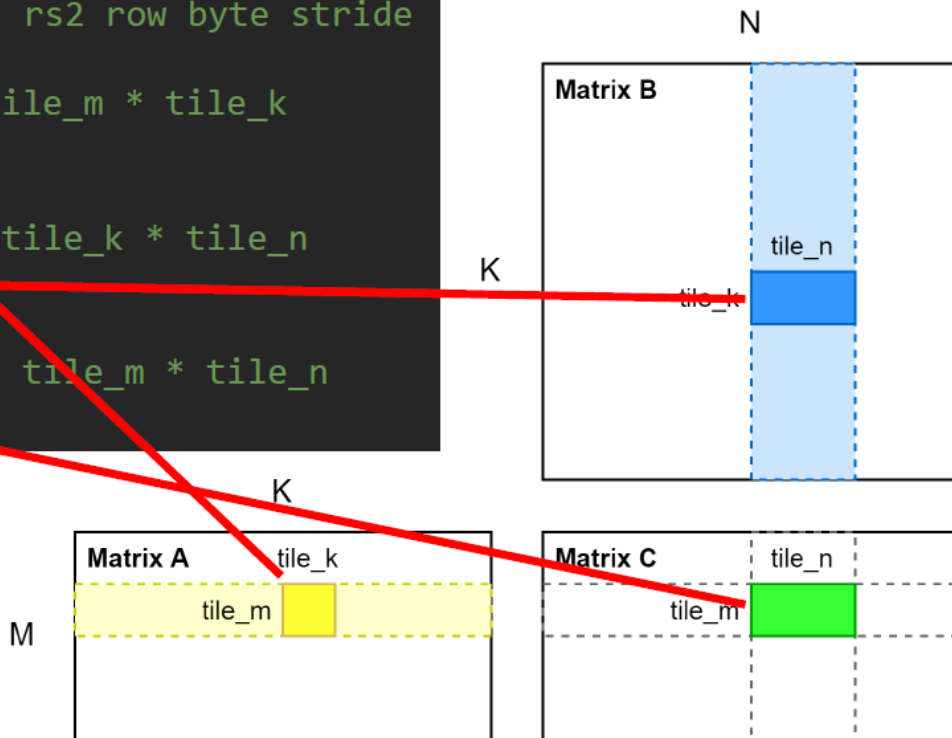
Configuration-Setting Instructions Example





Load Instructions

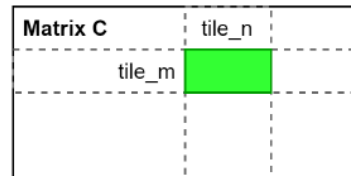
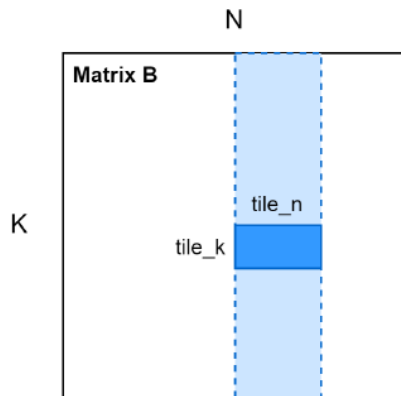
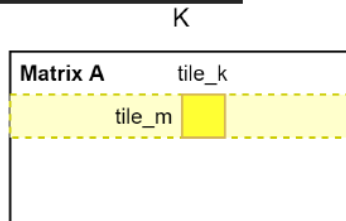
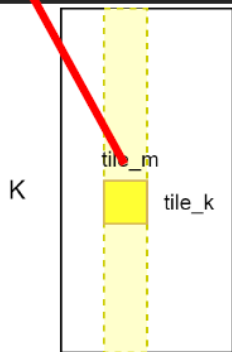
```
# md destination, rs1 base address, rs2 row byte stride  
  
# for left matrix, a, tile size = tile_m * tile_k  
mlae<eew>.m md, (rs1), rs2  
  
# for right matrix, b, tile size = tile_k * tile_n  
mlbe<eew>.m md, (rs1), rs2  
  
# for output matrix, c, tile size = tile_m * tile_n  
mlce<eew>.m md, (rs1), rs2
```





Load Instructions with transposition

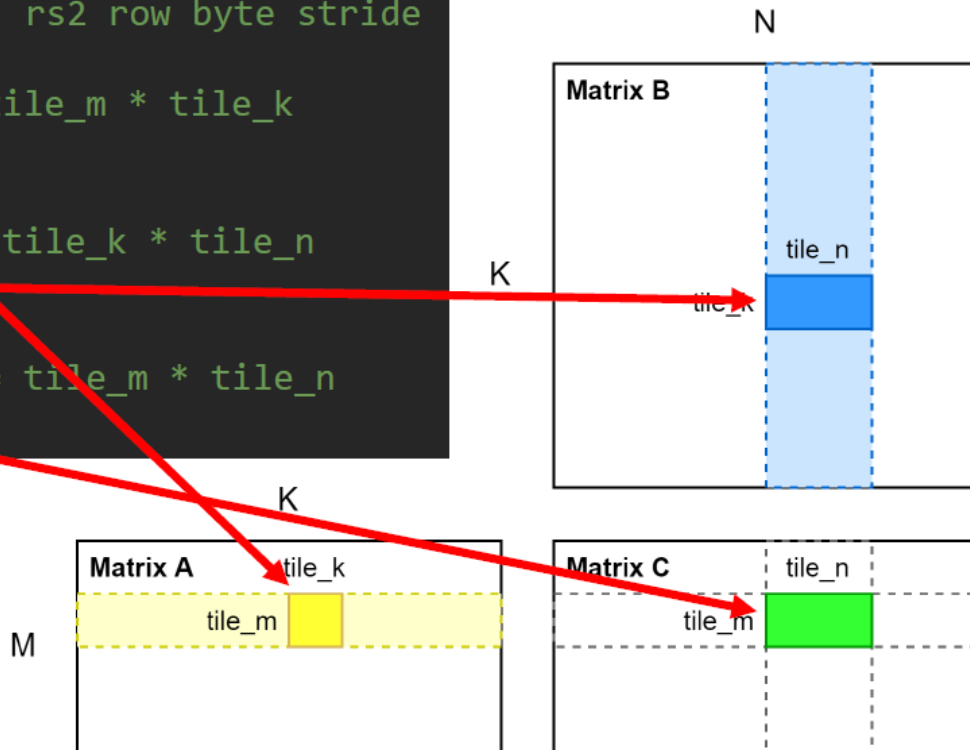
```
# md destination, rs1 base address, rs2 row byte stride  
  
# for left matrix, a, tile size = tile_k * tile_m  
mlate<eew>.m md, (rs1), rs2  
  
# for right matrix, b, tile size = tile_n * tile_k  
mlbte<eew>.m md, (rs1), rs2  
  
# for output matrix, c, tile size = tile_n * tile_m  
mlcte<eew>.m md, (rs1), rs2
```





Store Instructions

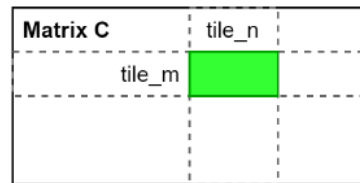
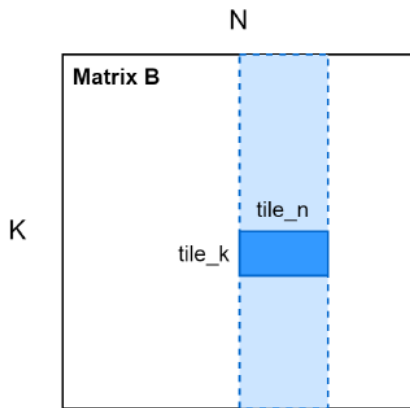
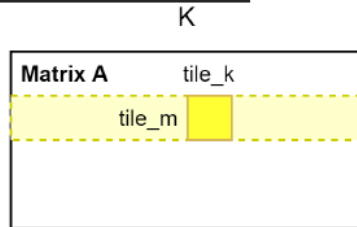
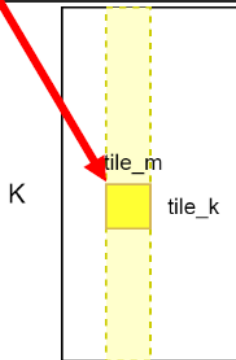
```
# md destination, rs1 base address, rs2 row byte stride  
  
# for left matrix, a, tile size = tile_m * tile_k  
msae<ew>.m md, (rs1), rs2  
  
# for right matrix, b, tile size = tile_k * tile_n  
msbe<ew>.m md, (rs1), rs2  
  
# for output matrix, c, tile size = tile_m * tile_n  
msce<ew>.m md, (rs1), rs2
```





Store Instructions with transposition

```
# ms3 store data, rs1 base address, rs2 row byte stride  
  
# for left matrix, a, tile size = tile_k * tile_m  
msate<ew>.m ms3, (rs1), rs2  
  
# for right matrix, b, tile size = tile_n * tile_k  
msbte<ew>.m ms3, (rs1), rs2  
  
# for output matrix, c, tile size = tile_n * tile_m  
mscte<ew>.m ms3, (rs1), rs2
```

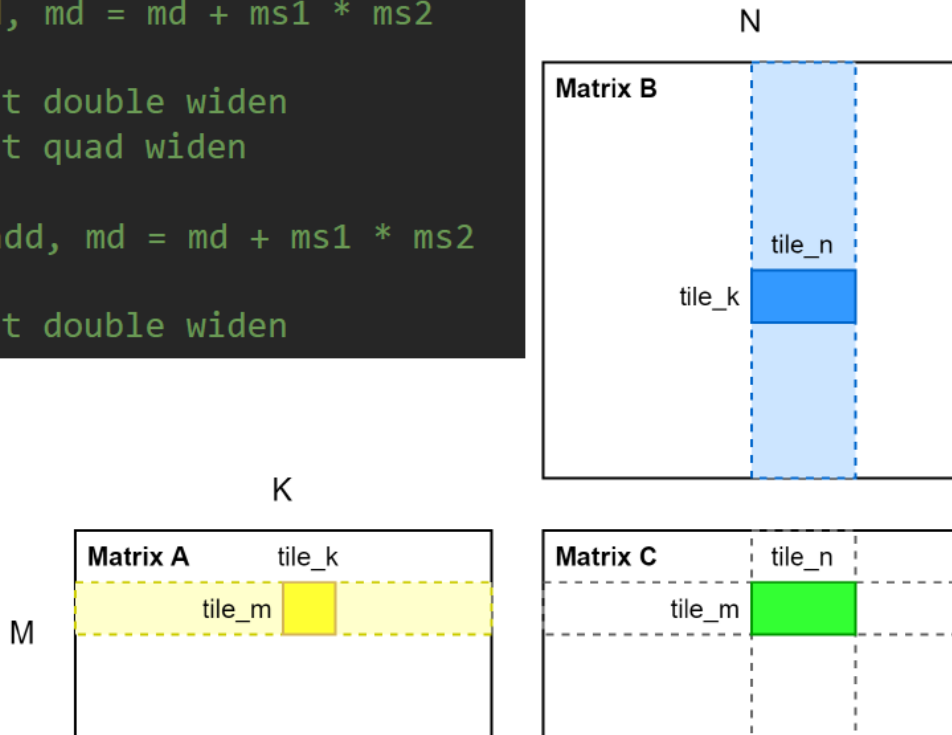




Matrix Multiplication Instructions

```
# int matrix multiplication and add, md = md + ms1 * ms2
mma.mm md, ms1, ms2
mwma.mm md, ms1, ms2      # output double widen
mqma.mm md, ms1, ms2      # output quad widen

# float matrix multiplication and add, md = md + ms1 * ms2
mfma.mm md, ms1, ms2
mfwma.mm md, ms1, ms2     # output double widen
```





Element-Wise Add/Sub/Multiply Instructions

```
# md[i,j] = md[i,j] + ms1[i,j]  
m{w|q}addc.mm md, ms1  
mf{w}addc.mm md, ms1
```

```
# md[i,j] = md[i,j] - ms1[i,j]  
m{w|q}subc.mm md, ms1  
mf{w}subc.mm md, ms1
```

```
# md[i,j] = ms1[i,j] - md[i,j]  
m{w|q}rsubc.mm md, ms1  
mf{w}rsubc.mm md, ms1
```

```
# md[i,j] = ms1[i,j] * rs2  
m{w|q}emulc.mx md, ms1, rs2  
mf{w}emulc.mx md, ms1, rs2
```

```
# md[i,j] = ms1[i,j] * imm  
m{w|q}emulc.mi md, ms1, imm
```



Type-Convert Instructions

```
# convert float to float
mfncvtc.f.fw.m md, ms1 # double-width float to single-width float
mfwcvtc.fw.f.m md, ms1 # single-width float to double-width float

# convert int to float
mfcvtc.f.x.m md, ms1 # int to float
mfncvtc.f.xw.m md, ms1 # double-width int to float
mfncvtc.f.xq.m md, ms1 # quad-width int to float
mfwcvtc.fw.x.m md, ms1 # single-width int to double-width float
mfcvtc.fw.xw.m md, ms1 # double-width int to double-width float
mfncvtc.fw.xq.m md, ms1 # quad-width int to double-width float

# convert float to int
mfcvtc.x.f.m md, ms1 # float to int
mfwcvtc.xw.f.m md, ms1 # float to double-width int
mfwcvtc.xq.f.m md, ms1 # float to quad-width int
mfncvtc.x.fw.m md, ms1 # double-width float to single-width int
mfcvtc.xw.fw.m md, ms1 # double-width float to double-width int
mfwcvtc.xq.fw.m md, ms1 # double-width float to quad-width int
```



Matrix Multiplication with RISC-V Matrix Extension

```
void matmul_float16(c, a, b, m, k, n) {  
    msettype(e16);                                // use 16bit input matrix element  
  
    for (i=0; i<m; i+=tile_m) {                    // loop at dim m with tiling  
        tile_m = msettile_m(m-i);  
        for (j=0; j<n; j+=tile_n) {                // loop at dim n with tiling  
            tile_n = msettile_n(n-j);  
  
            acc = mfemul_mf(acc, 0.f);              // clear acc reg  
            for (s=0; s<k; s+=tile_k) {            // loop at dim k with tiling  
                tile_k = msettile_k(k-s);  
  
                tr1 = mlae16_m(&a[i][s], k*2);      // load left matrix a  
                tr2 = mlbe16_m(&b[s][j], n*2);      // load right matrix b  
                acc = mfwma_mm(tr1, tr2);           // tiled matrix multiply,  
                                                    // double widen output acc  
            }  
  
            acc = mfncvt_f_fw_m(acc);              // convert widen result  
            msce16_m(acc, &c[i][j], n*2);          // store to matrix c  
        }  
    }  
}
```

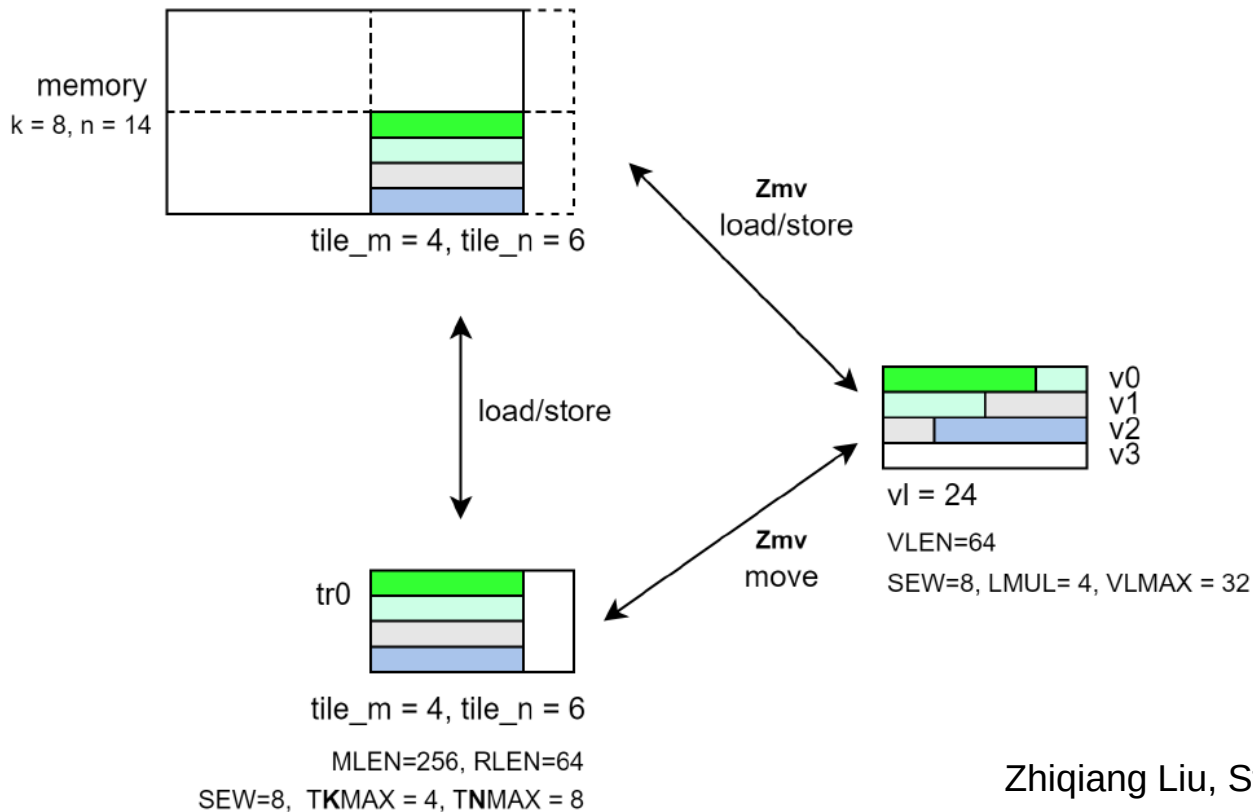


Zmv: Matrix for Vector operations

- **Provide matrix support with the RISC V Vector "V" extension**
 - Load matrix tile slices into vector registers
 - Store matrix tile slices from vector registers
 - Move data between slices of a matrix register and vector registers
 - Element wise multiply with a matrix register and a vector
 - register(broadcast to a matrix).



Load/store, data move example





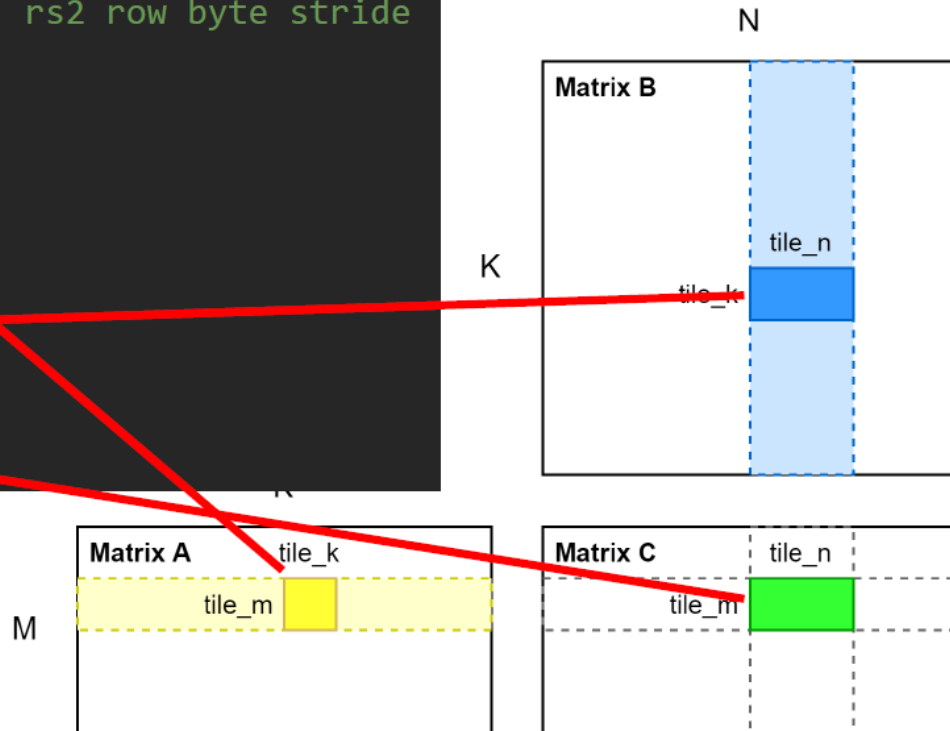
Load Instructions

```
# vd destination, rs1 base address, rs2 row byte stride  
# lmul / (eew/sew) rows or columns
```

```
# for left matrix, a  
mlae<eew>.v vd, (rs1), rs2
```

```
# for right matrix, b  
mlbe<eew>.v vd, (rs1), rs2
```

```
# for output matrix, c  
mlce<eew>.v vd, (rs1), rs2
```





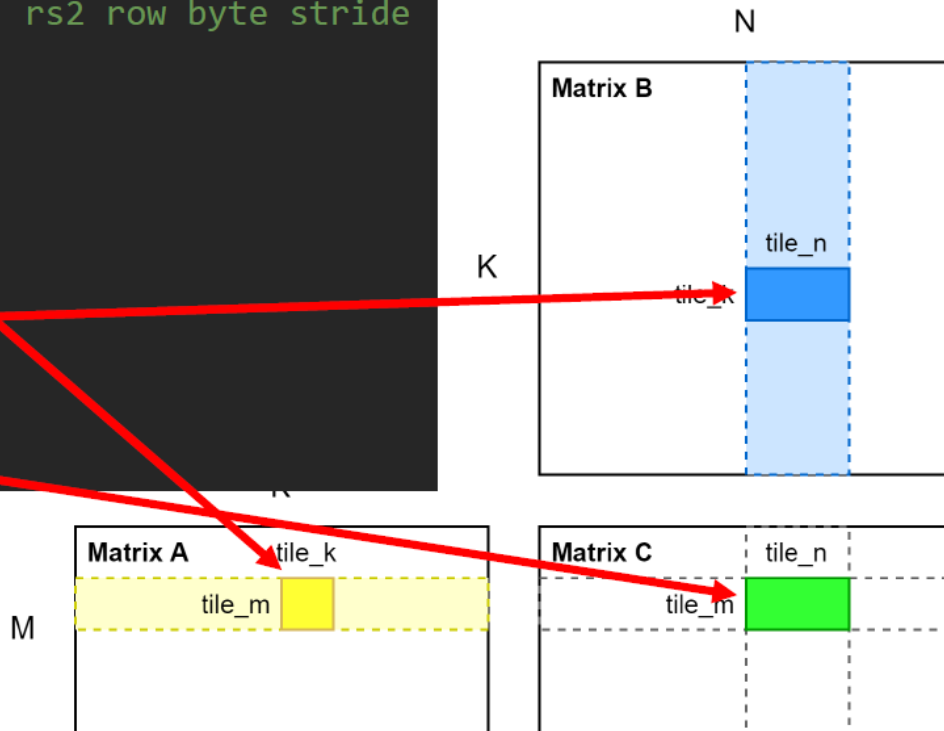
Store Instructions

```
# vs3 store data, rs1 base address, rs2 row byte stride  
# lmul / (eew/sew) rows or columns
```

```
# for left matrix, a  
msae<eew>.v vs3, (rs1), rs2
```

```
# for right matrix, b  
msbe<eew>.v vs3, (rs1), rs2
```

```
# for output matrix, c  
msce<eew>.v vs3, (rs1), rs2
```





Data Move Instructions

between matrix **register** rows and vector registers.

$vd[(i-rs2) * tile_n + j] = md[i, j], i = rs2 .. rs2 + lmul$

mmvar.v.m vd, ms1, rs2

mmvbr.v.m vd, ms1, rs2

mmvcr.v.m vd, ms1, rs2

$md[i, j] = vd[(i-rs2) * tile_n + j], i = rs2 .. rs2 + lmul$

mmvar.m.v md, vs1, rs2

mmvbr.m.v md, vs1, rs2

mmvcr.m.v md, vs1, rs2

between matrix **register** columns and vector registers.

$vd[(j-rs2) * tile_m + i] = md[i, j], j = rs2 .. rs2 + lmul$

mmvac.v.m vd, ms1, rs2

mmvbc.v.m vd, ms1, rs2

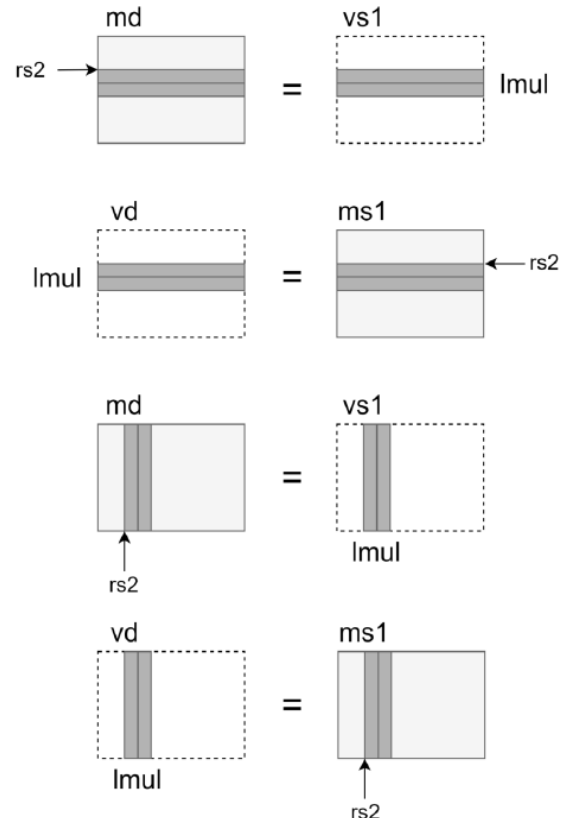
mmvcc.v.m vd, ms1, rs2

$md[i, j] = vd[(j-rs2) * tile_m + i], j = rs2 .. rs2 + lmul$

mmvac.m.v md, vs1, rs2

mmvbc.m.v md, vs1, rs2

mmvcc.m.v md, vs1, rs2





Matrix element-wise multiply

```
# matrix ele-wise multiply with a row of vector
```

```
# md[i,j] = ms1[i,j] * vs2[j]
```

```
memulcr.mv md, ms1, vs2
```

```
mwemulcr.mv md, ms1, vs2 # output double widen
```

```
mqemulcr.mv md, ms1, vs2 # output quadruple widen
```

```
mfemulcr.mv md, ms1, vs2
```

```
mfwemulcr.mv md, ms1, vs2 # output double widen
```

```
# matrix ele-wise multiply with a column of vector
```

```
# md[i,j] = ms1[i,j] * vs2[i]
```

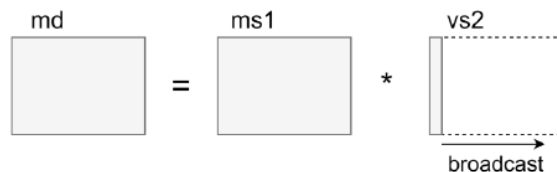
```
memulcc.mv md, ms1, vs2
```

```
mwemulcc.mv md, ms1, vs2 # output double widen
```

```
mqemulcc.mv md, ms1, vs2 # output quadruple widen
```

```
mfemulcc.mv md, ms1, vs2
```

```
mfwemulcc.mv md, ms1, vs2 # output double widen
```





Fused-Matmul_Relu Operation

```
void fused_matmul_relu_float16(c, a, b, m, k, n) {  
    msettype(e16); // use 16bit input matrix element  
    for (i=0; i<m; i+=tile_m) { // loop at dim m with tiling  
        tile_m = msettile_m(m-i);  
        for (j=0; j<n; j+=tile_n) { // loop at dim n with tiling  
            tile_n = msettile_n(n-j);  
            acc = mfemul_mf(acc, 0.f) // clear acc reg  
            for (s=0; s<k; s+=tile_k) { // loop at dim k with tiling  
                tile_k = msettile_k(k-s);  
                tr1 = mlae16_m(&a[i][s]); // load left matrix a  
                tr2 = mlbe16_m(&b[s][j]); // load right matrix b  
                acc = mfwma_mm(tr1, tr2); // tiled matrix multiply  
            }  
  
            acc = mfnvct_f_fw_m(acc); // convert widen result to normal width  
  
            for (s=0; s<tile_m; s+=rows) {  
                rows = min(tile_m - s, 8*vlenb/rlenb); // max rows could move into 8 vregs  
  
                vsetvl(tile_n*rows, e16, m8);  
                v1 = mmvcr_v_m(acc, s); // move acc rows to vreg  
                v1 = vfmax_vf(0.f, v1); // vfmax.vf for relu  
                msce16_v(v1, &c[i+s][j], n); // store output tile slices  
            }  
        }  
    }  
}
```



Takeaway Questions

- What is the job of slide unit in the VPU?
 - (A) Decides when and to which functional unit a decoded instruction gets dispatched
 - (B) Shift elements across lane boundaries
 - (C) Used for predicates instructions
- Given 1KB VREG, how many elements when setting SEW=32?
 - (A) 8
 - (B) 16
 - (C) 32



Takeaway Questions

- What are advantages of Matrix Extension?
 - (A) Reduce the instruction counts for MatMul computations
 - (B) Improve the L1 cache hit of the CPU
 - (C) Shorten the latency of memory access