ETH zürich

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

The ARATHON

A Hackathon on vector processors and Ara

Integrated Systems Laboratory (ETH Zürich)

Matteo Perotti mperotti@iis.ee.ethz.ch

PULP Platform
Open Source Hardware, the way it should be!

@pulp_platform
pulp-platform.org
youtube.com/pulp_platform

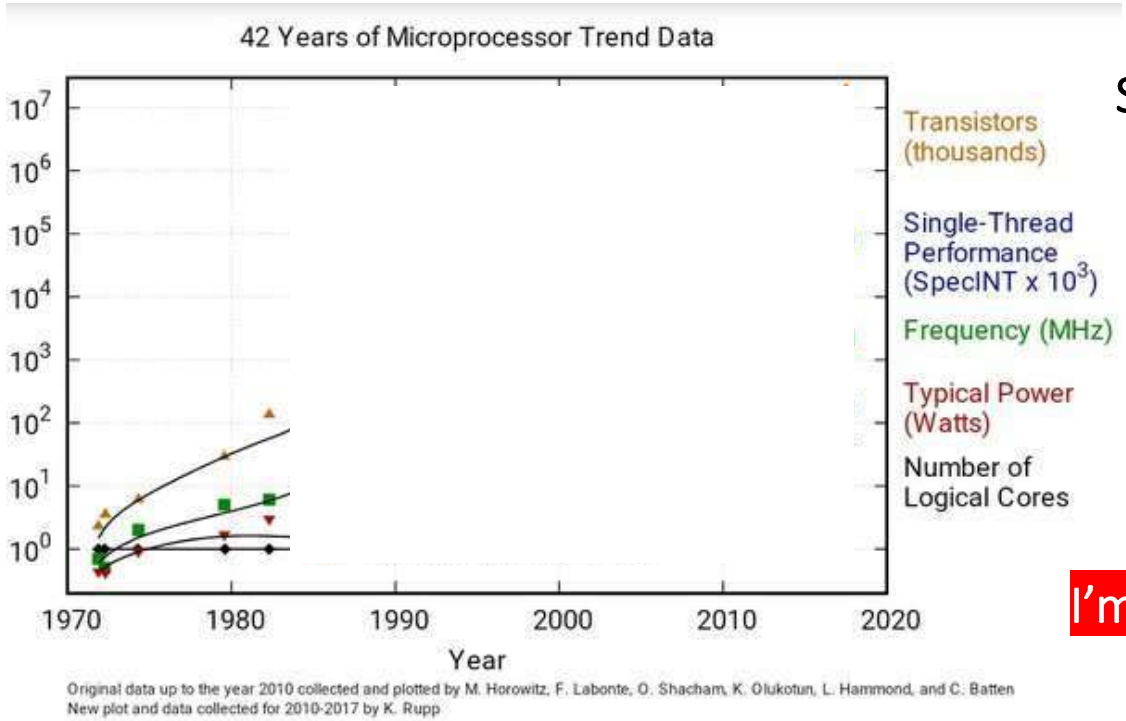
The curious case of Vector Processors

Cray X1

256 MFLOPS

Fastest supercomputer in 1976!

The Cray 1, a vector supercomputer. The first model ran at 80MHz but could retire 2 instructions/cycle for a peak of 160 MIPS. However, it could reach 250 MFLOPS using vectors.



Inexpensive silicon
ILP, OoO, Super Scalar

Hype for Vector Processors

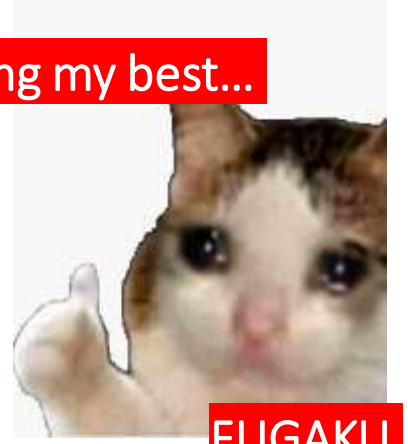
End of Dennard's Scaling
Power wall

SUPERCOMPUTER FUGAKU

500 PFLOPS

Seventh fastest Supercomputer in 2025!

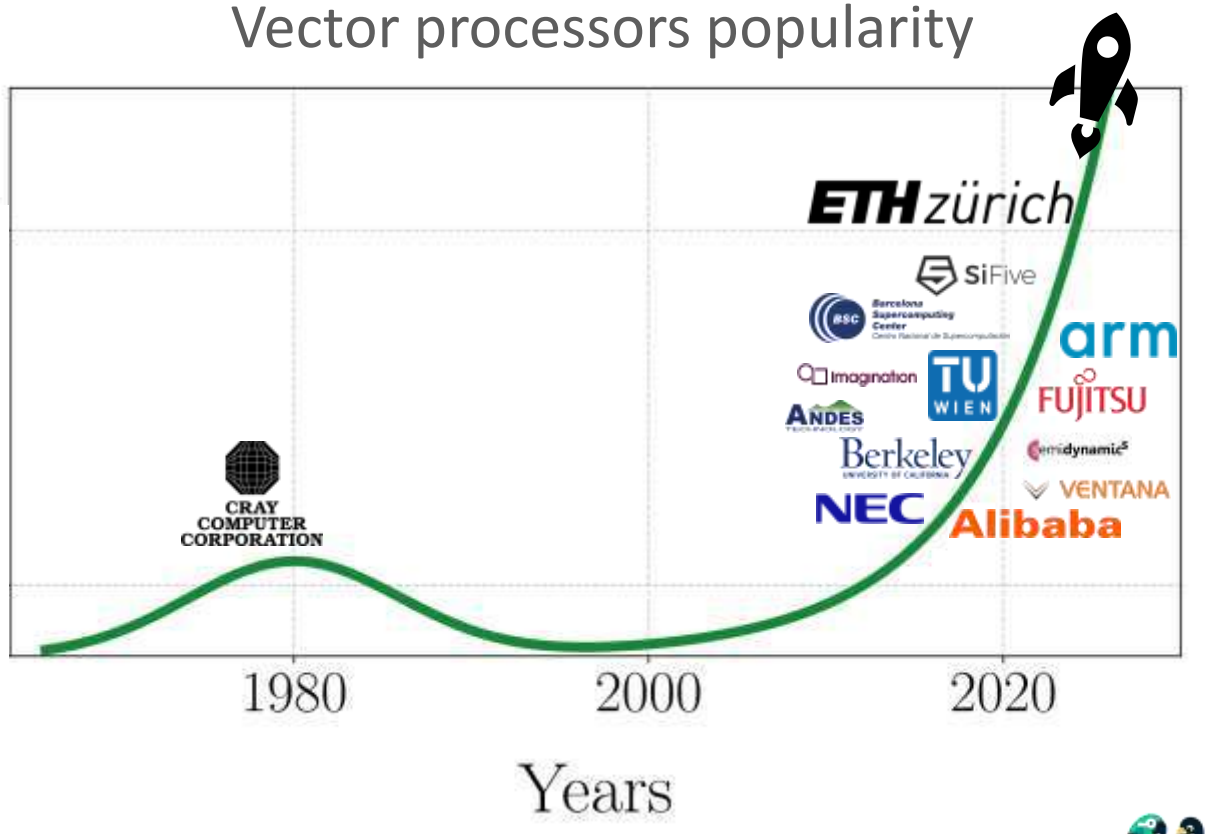
I'm trying my best...



FUGAKU

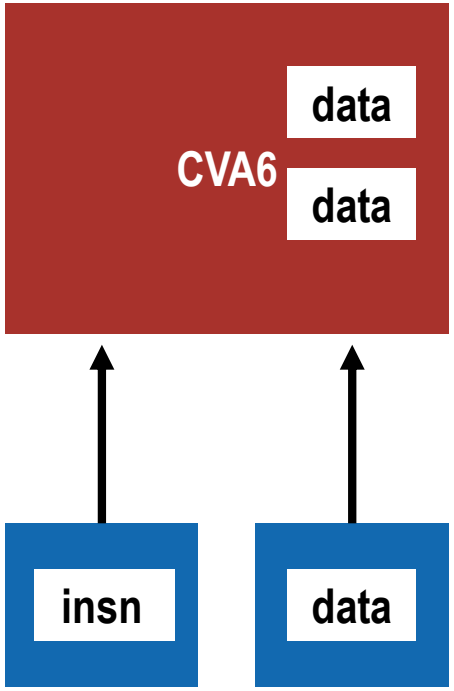
Now… They are all around!

- Tons of implementations in the last years!
- **ISAs** → RISC-V V, Arm SVE, NEC
- **Companies** → SiFive, Andes, NEC, Alibaba
- **Research** → ETHZ, BSC, Berkeley, TU Wien
- Why?
  - They are efficient
  - Great portability of their applications
  - AI and ML (like HPC) are massively parallel
    - Exploit data-level parallelism
    - Adopted in IoT and edge devices



But… let’s first take a step back

Scalar processors should be good enough, right?

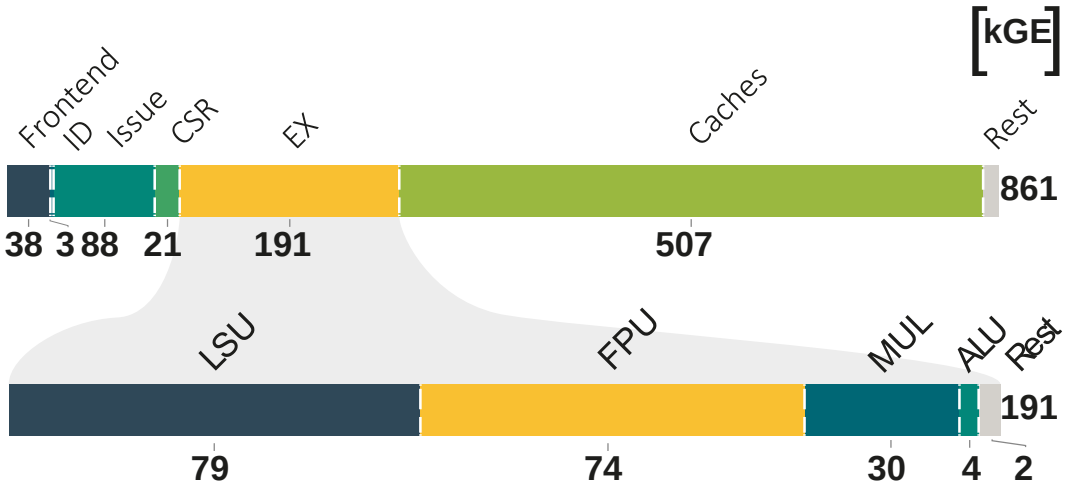


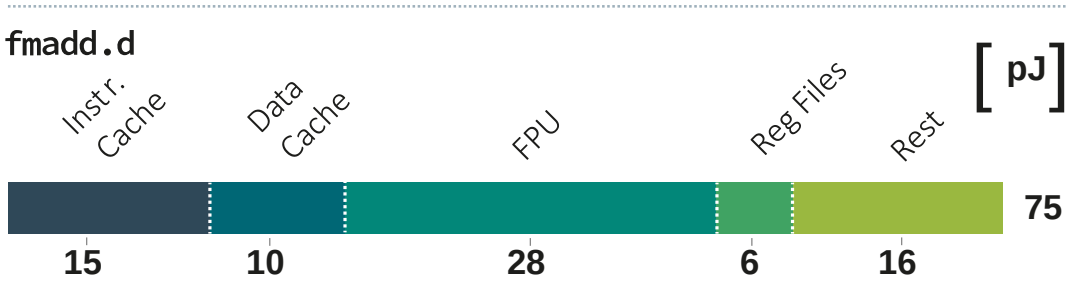

```
for (int i = 0; i < N; ++i)
  acc += a[i] * b[i]
```

```
Loop:
1  ld x1, @1
2  ld x2, @2
3  macc x3, x2, x1
4  bump @1
5  bump @2
6  bump idx
7  branch(idx) → Loop
```

7 instructions for **one "computation"! Not very efficient…**

Scalar processor's flexibility has a cost





Application-class features needed for management, but not the **primary source of energy-efficient** compute!

Von Neumann Bottleneck

- When CVA6 runs a dot-product kernel, **only 28pJ** out of 317pJ are **used for** actual **computation** (9% of energy)

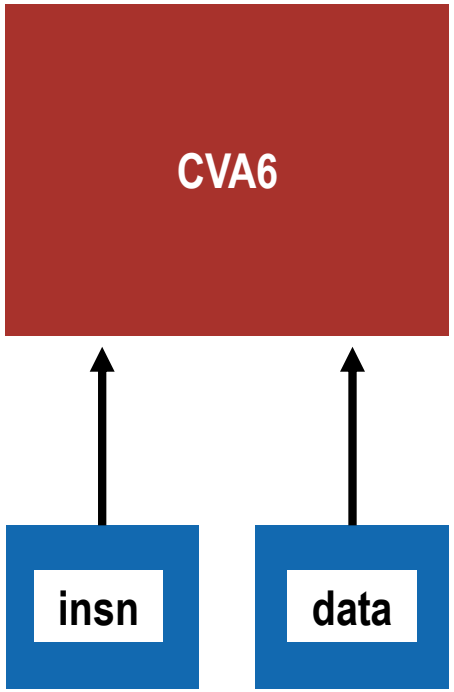
2x load

1x compute

2x addr bump

- A limitation of the von Neumann architecture
- Energy “wasted” by fetching, decoding, and dispatching instructions that operate on single words

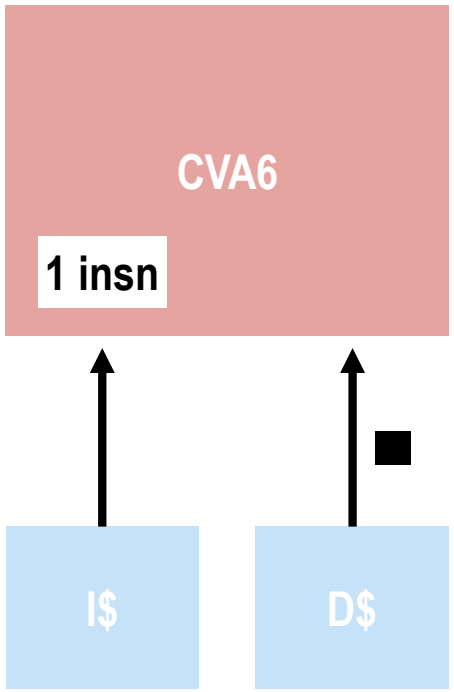
Intuition: one instruction to trigger much computation!



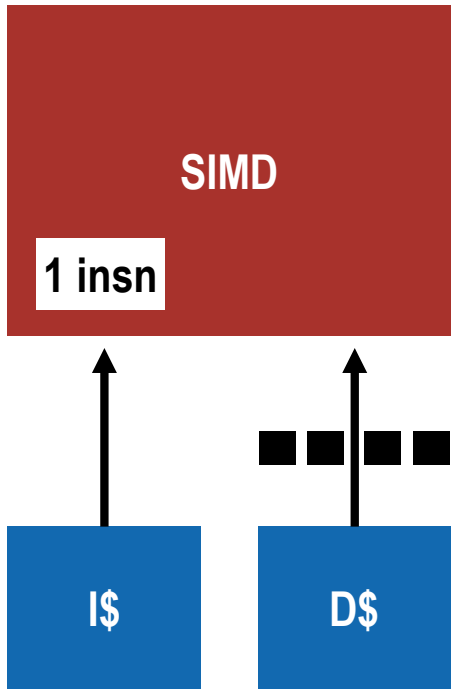

```
for (int i = 0; i < N; ++i)
  acc += a[i] * b[i]
```

```
1  vld x1, @1
```

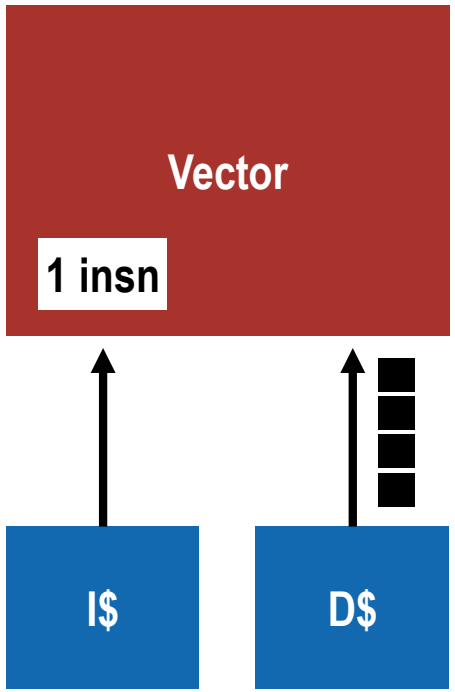
And this can be done through space or time!



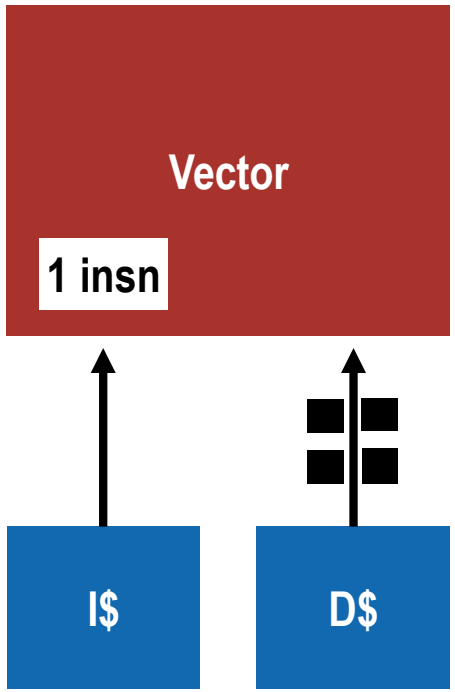
Scalar baseline



SIMD approach (space)

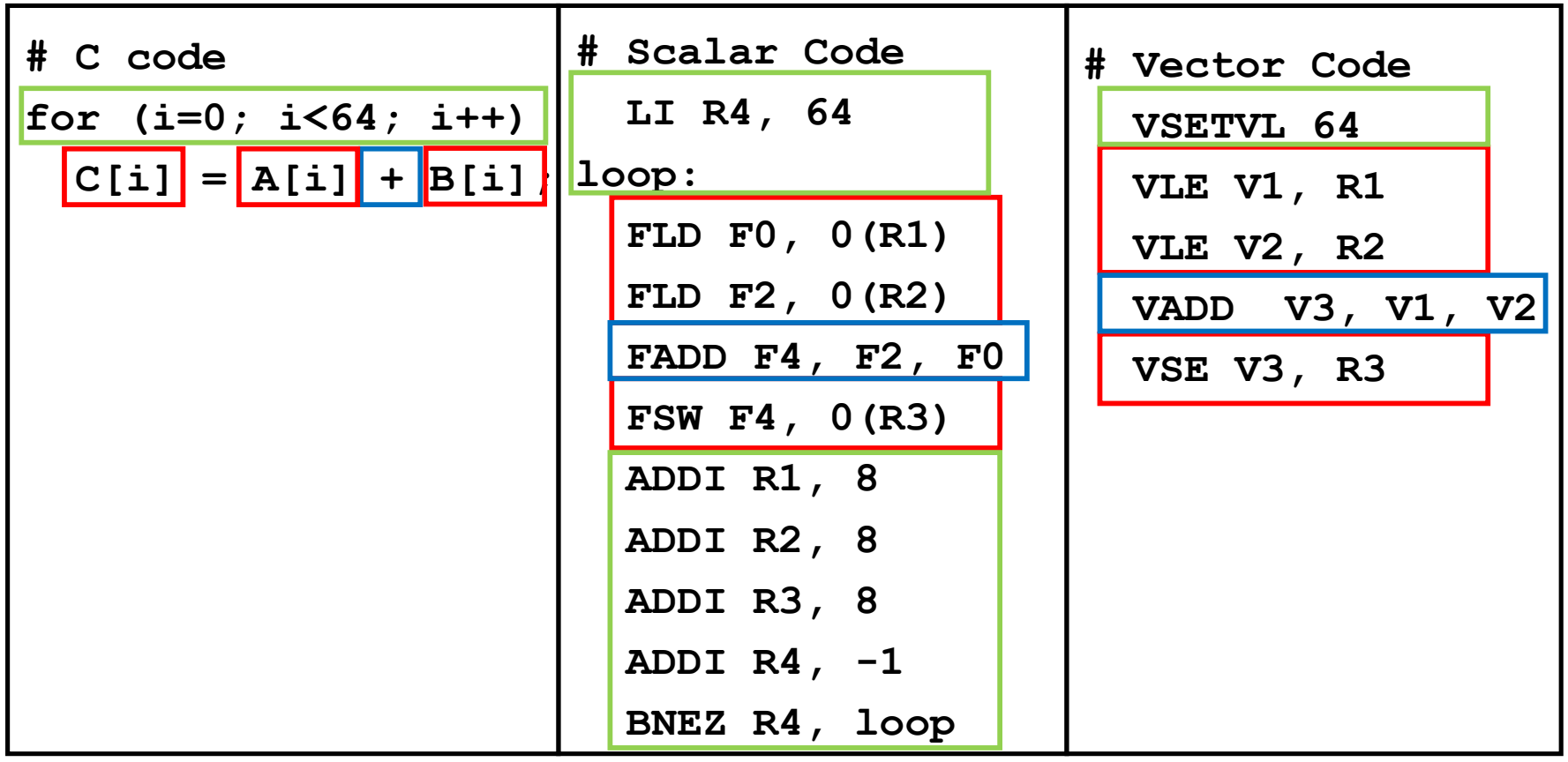


Vector approach (time)



Vector approach (space + time)
Ara is like this!

Vector Code Example



```
# C code
for (i=0; i<64; i++)
  C[i] = A[i] + B[i];
```

```
# Scalar Code
  LI R4, 64
loop:
  FLD F0, 0(R1)
  FLD F2, 0(R2)
  FADD F4, F2, F0
  FSW F4, 0(R3)
  ADDI R1, 8
  ADDI R2, 8
  ADDI R3, 8
  ADDI R4, -1
  BNEZ R4, loop
```

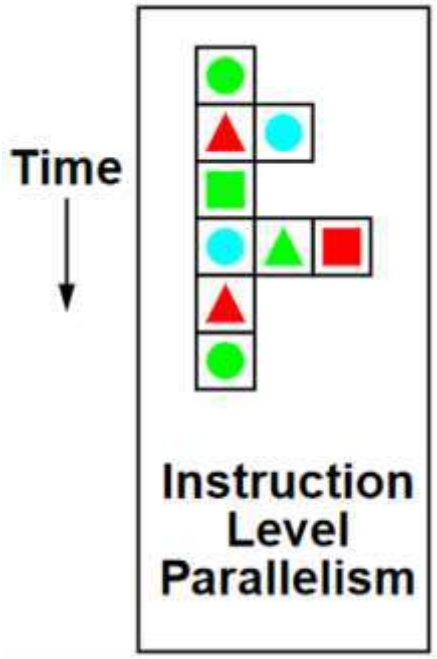
```
# Vector Code
  VSETVL 64
  VLE V1, R1
  VLE V2, R2
  VADD  V3, V1, V2
  VSE V3, R3
```

Hands-on (guided) – Scalar vs. Vector assembly

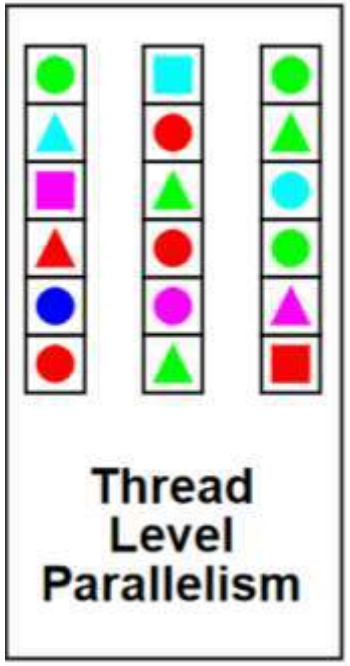
1. `git clone https://github.com/pulp-platform/ara.git`
2. `cd ara`
3. `git checkout mp/arathon`
4. Follow instructions of the main README.md → *Arathon SPECIAL* paragraph
    1. Open the "`apps/arathon/ex/vadd.c`" source code
    2. You will find vector code to implement a vector addition
    3. Write the scalar version of the code (for loop)
    4. Compile the application following the README guide (`ex_name=VADD`)
    5. Inspect the scalar and vector assembly in "`apps/bin/arathon.dump`"


P
P

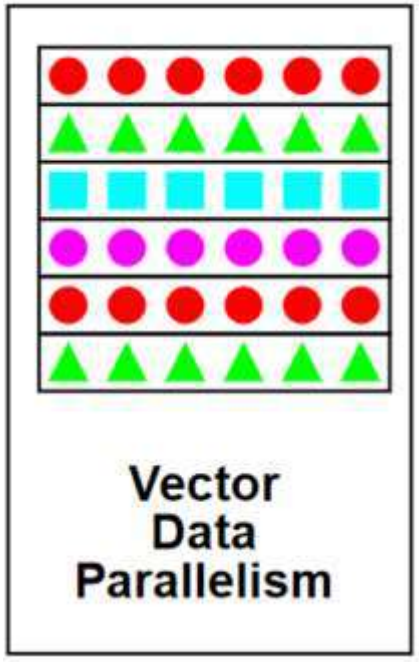
Different types of parallelism



- Pipelining
- OoO + Reg Renaming
- Speculation
- Superscalar
- VLIW



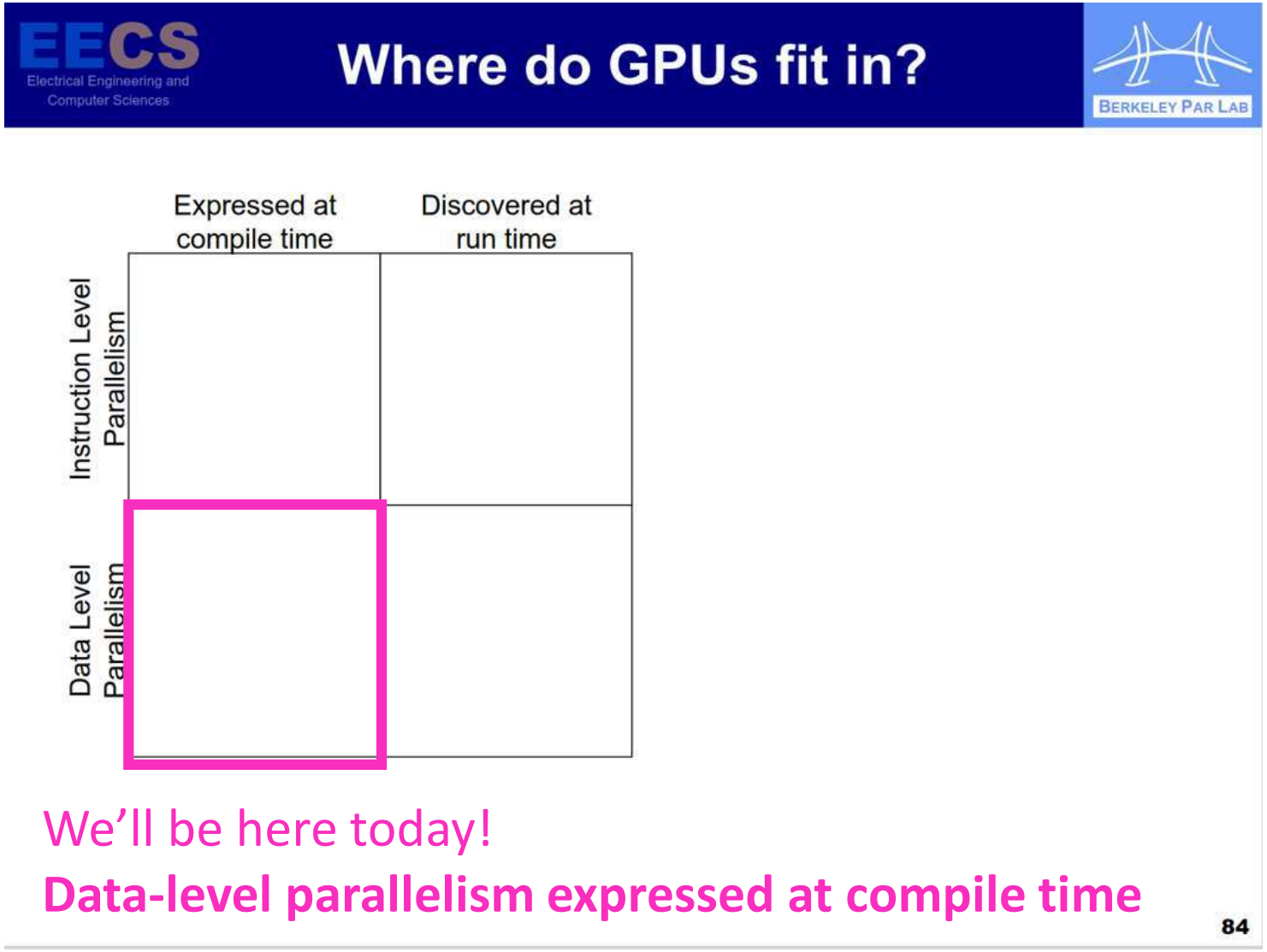
- Multi-core
- HW multi-threading



- SIMD computation
- SIMD processors
- Vector processors

We'll be here today!

Who is responsible for finding parallelism?



Exploiting Data Parallelism: SIMD

- Concurrency arises from performing **the same operation on different elements**
  - **Single-Instruction Multiple-Data (SIMD)**

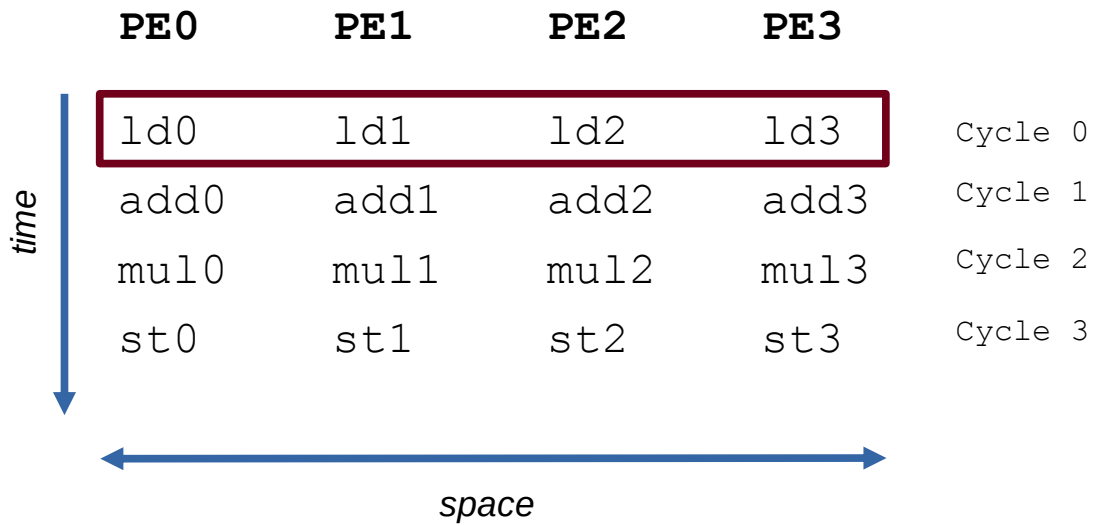
- Single instruction operates on multiple data elements
  - Multiplexed in time or in space

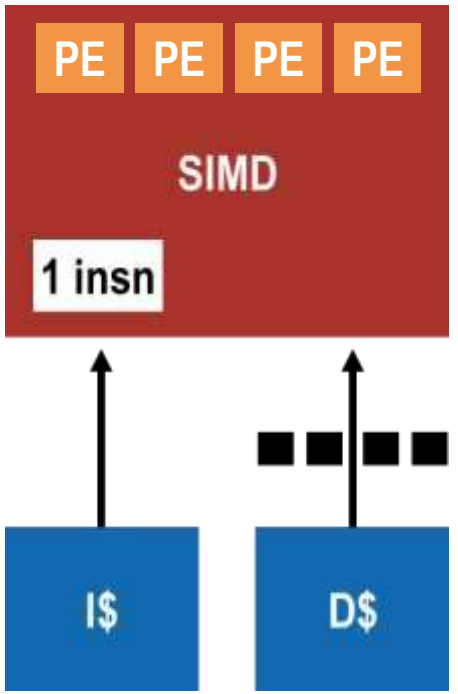
- Time-space duality:
  - **Array processor:** instruction operates on multiple data elements *at the same time* using *different spaces*
  - **Vector processor:** instruction operates on multiple data elements *in consecutive data steps* using the *same space*
  - Hybrid between array and vector processors are more commonly found today.

Parallel Processing: space

- Time-space duality:
  - **Array processor**: instruction operates on multiple data elements at the same time
  - **Vector processor**: instruction operates on multiple data elements in consecutive cycles

```
vload vd, @A[3:0]
vadd vd, vd, 1
vmul vd, vd, 2
vstore @A[3:0], vd
```

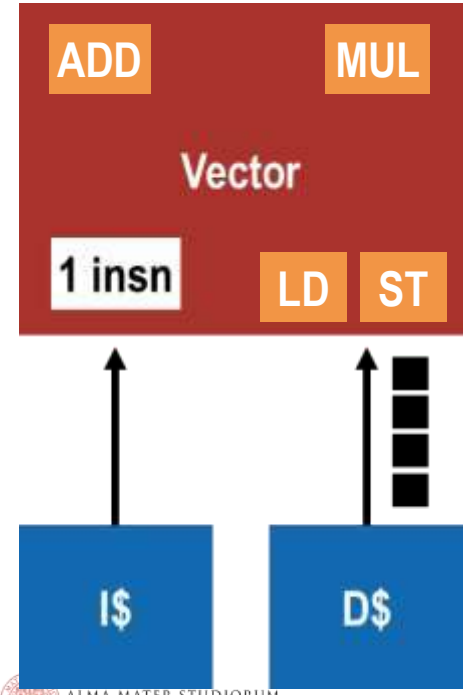


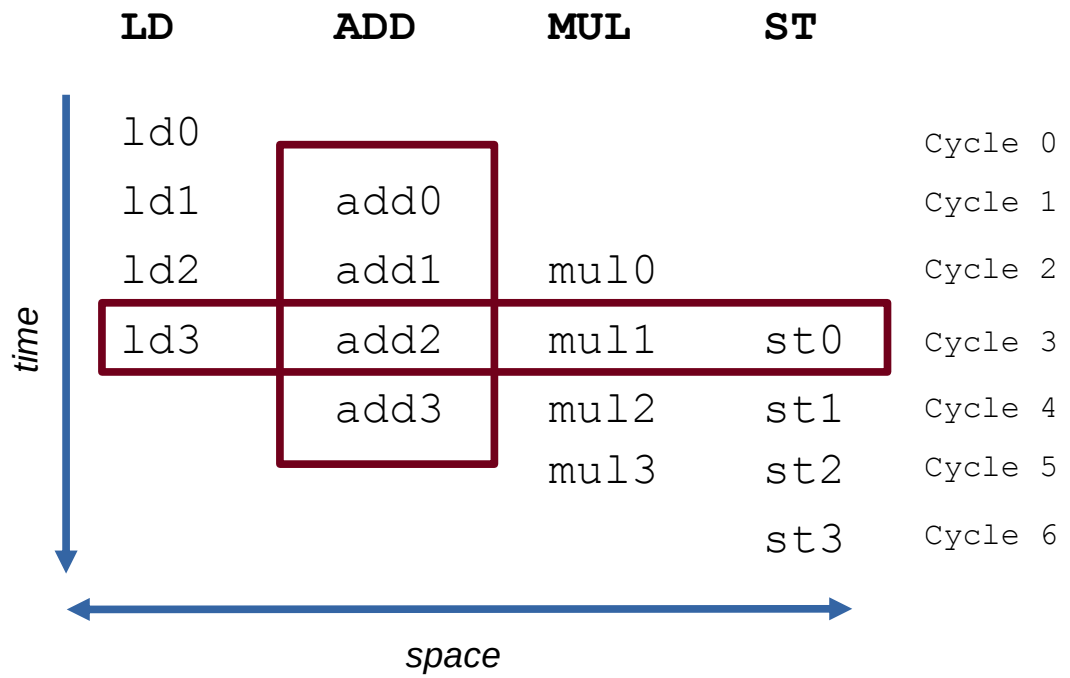


Parallel Processing: time

- Time-space duality:
  - **Array processor**: instruction operates on multiple data elements at the same time
  - **Vector processor**: instruction operates on multiple data elements in consecutive cycles

```
vload vd, @A[3:0]
vadd vd, vd, 1
vmul vd, vd, 2
vstore @A[3:0], vd
```

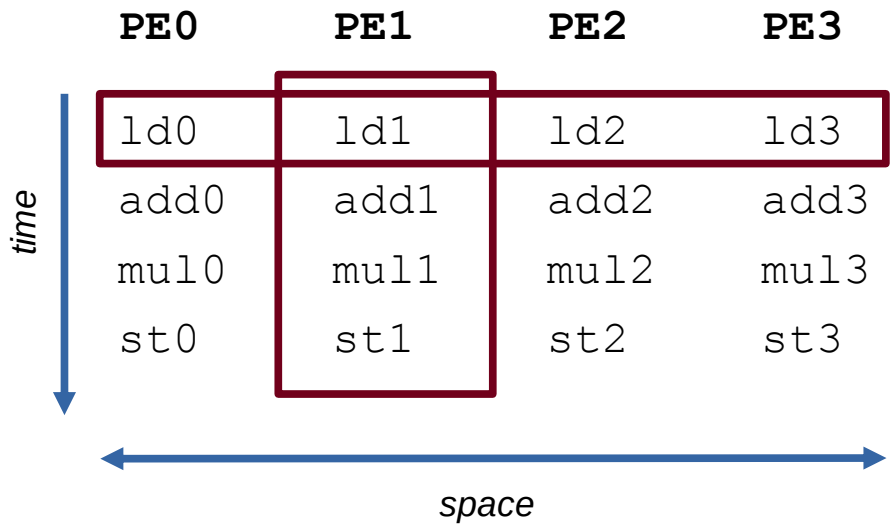




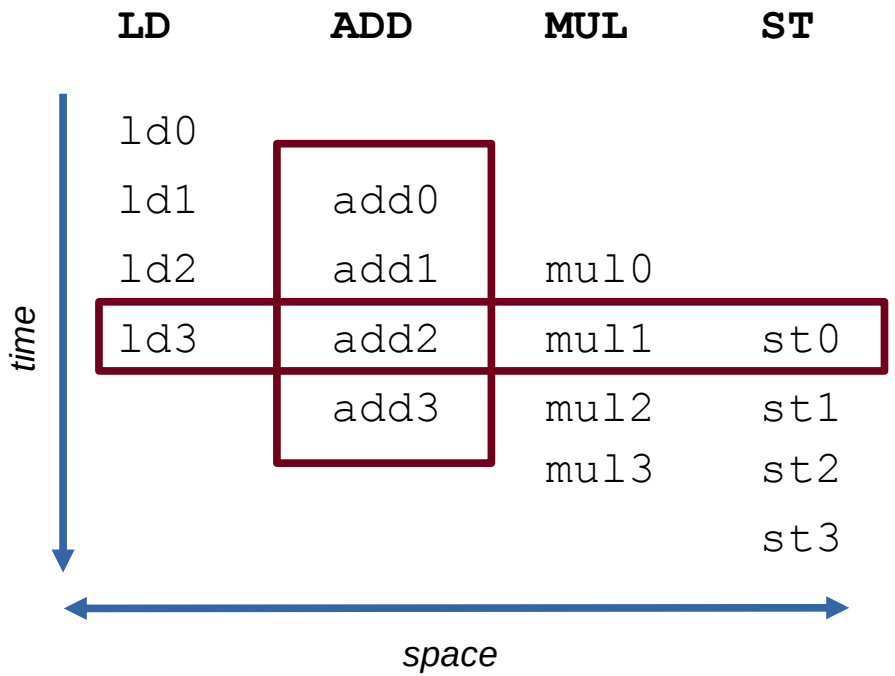
Two extremes?

```
vload vd, @A[3:0]
vadd vd, vd, 1
vmul vd, vd, 2
vstore @A[3:0], vd
```

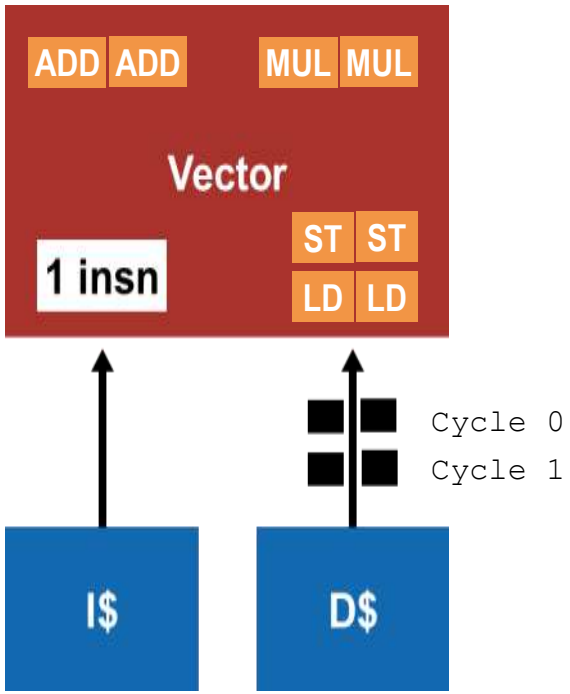
Array Processor



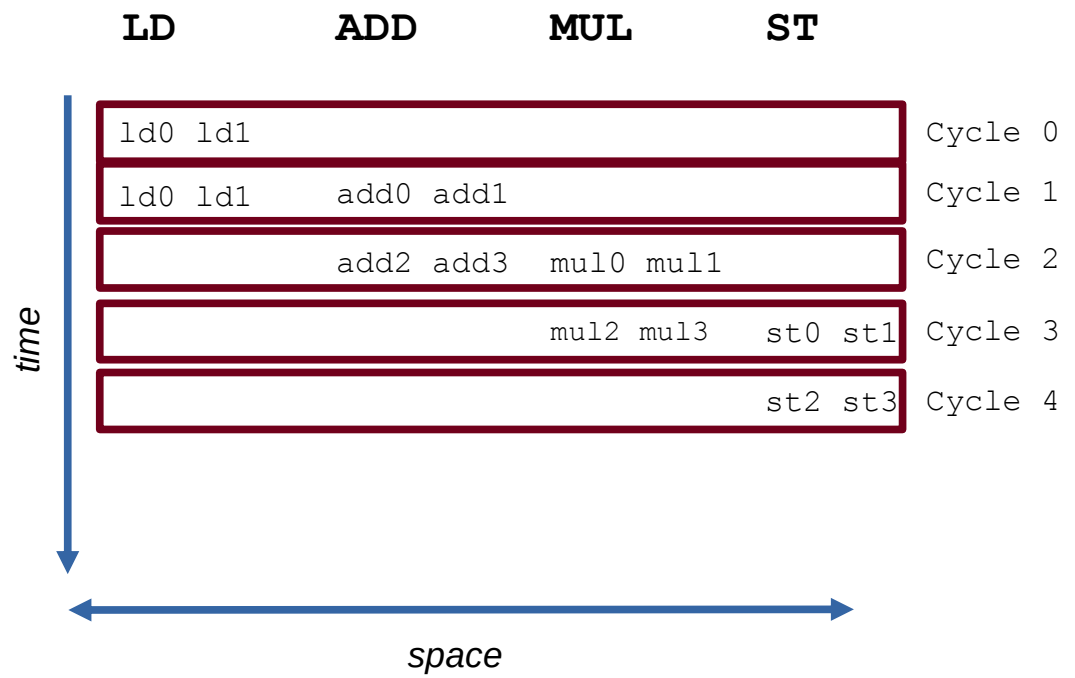
Vector Processor



Vector can be a mixture of the two (Ara!)



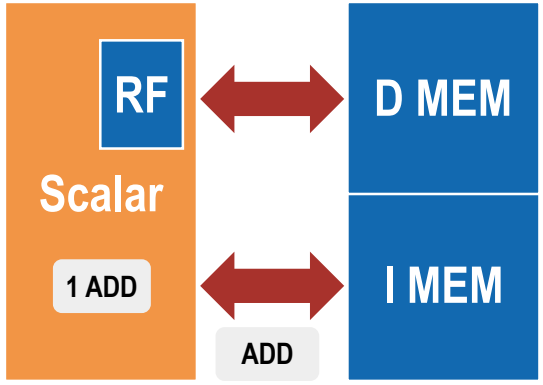

```
vload vd, @A[3:0]
vadd vd, vd, 1
vmul vd, vd, 2
vstore @A[3:0], vd
```



Single-Instruction, Multiple-Data (SIMD) processor

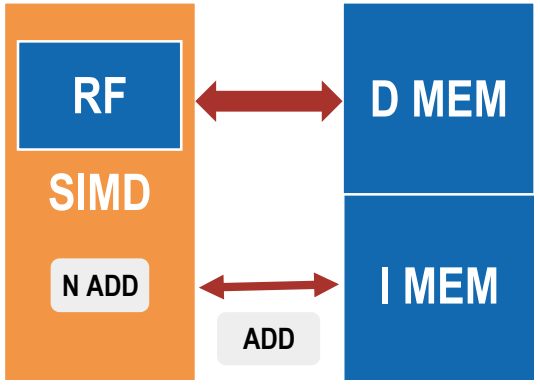
SCALAR CORE

- One instruction – One Operation
  - **BW** and Power on the I-MEM
- RF size is usually fixed
  - One data element per entry
  - Too small for highly-intensive WL
    - BW and Power on the D-MEM



SIMD CORE (space)

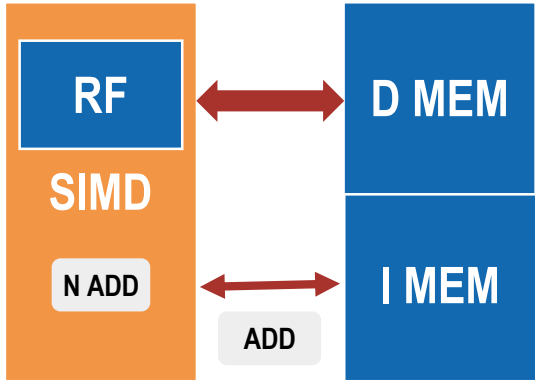
- One instruction – Multiple operations
  - Lower BW and Power on the I-MEM
- RF size is larger
  - Multiple data elements per entry
  - Better buffering, exploit locality
    - Lower BW and Power on the D-MEM



Vector Processors – A flexible and efficient choice

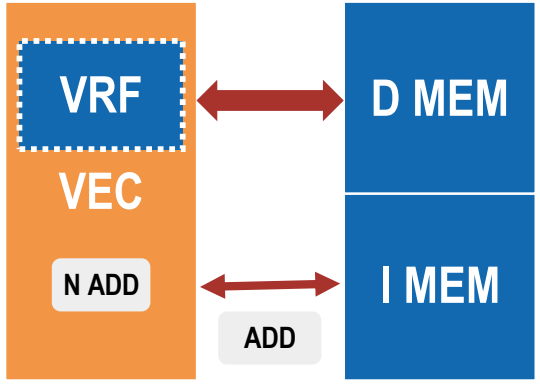
SIMD CORE (space)

- Vector Length (VL) encoded in the instruction!
  - Support new VL? Extend the ISA!
    - e.g., add128, add256, …
  - RF is not flexible



VECTOR CORE (time [+space])

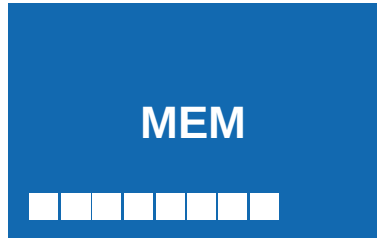
- Programmable VL
  - VRF size decoupled from DP width



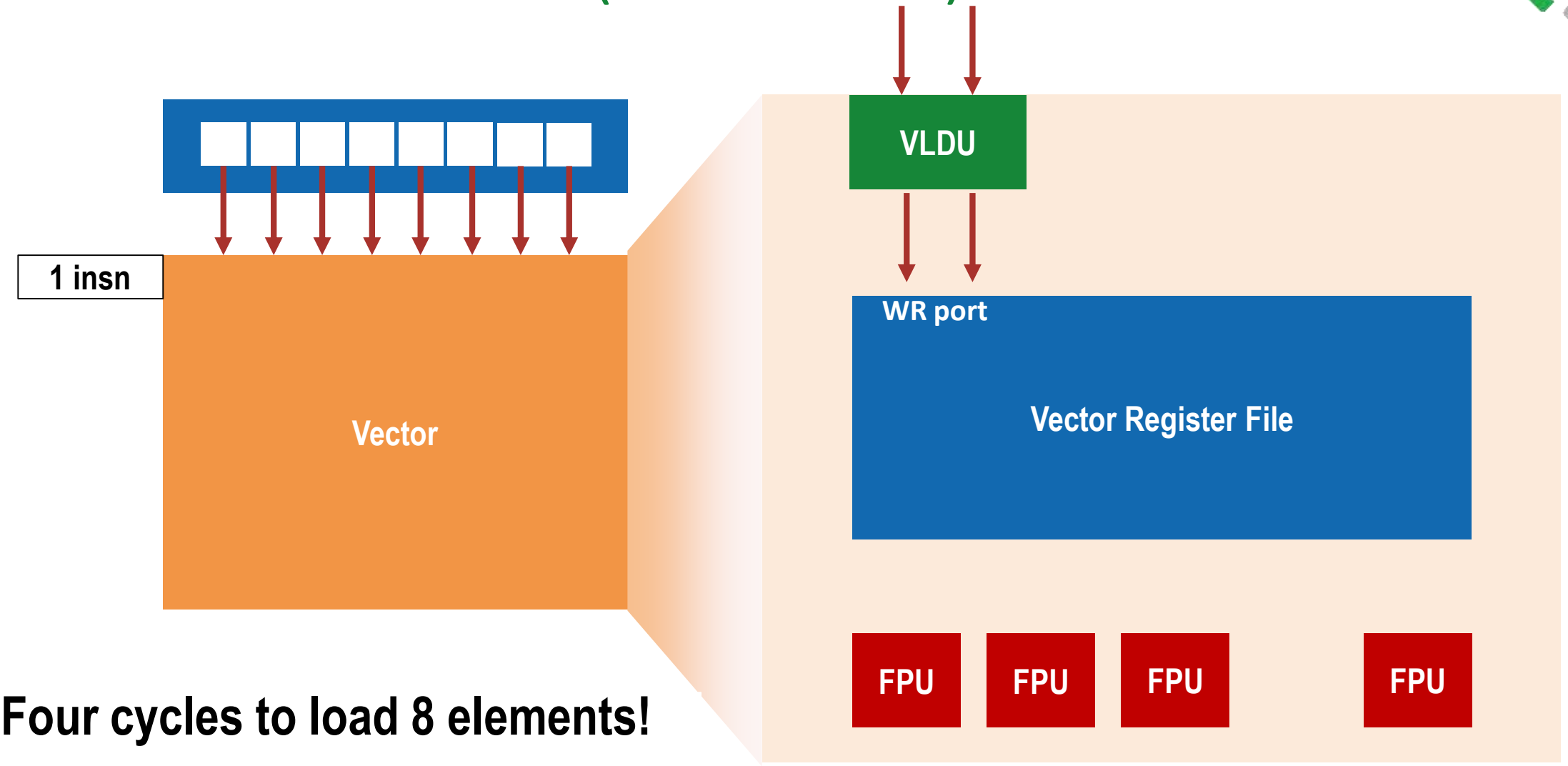

P
P

A brief overview of vector execution

- The **vector elements** to process are **in memory**
- Our vector architecture contains a **Vector Register File (VRF)**
- We load vector elements from memory to the VRF
  - The **VRF buffers** our **vector** close to the processing units
  - The **VRF** is the **bottom** of the **memory hierarchy**!
  - The **VRF increases data reuse** by exploiting data locality
- And if our vector registers are too small to host a full vector?
- We only load and process the first chunk of the vector!
- Then, when we are over with it, we continue with the second chunk
- Until we process the whole vector in memory!

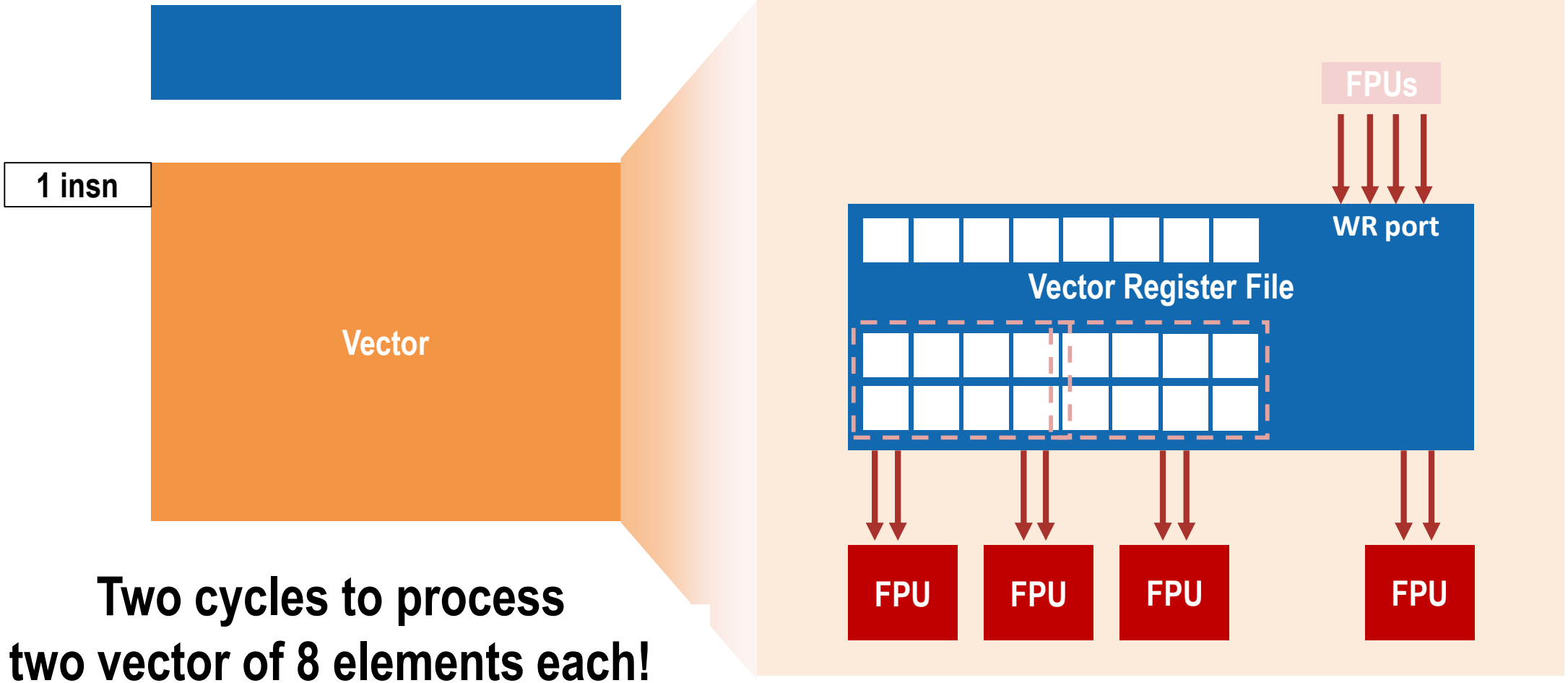


Dataflow – Vector load (mem → VRF)



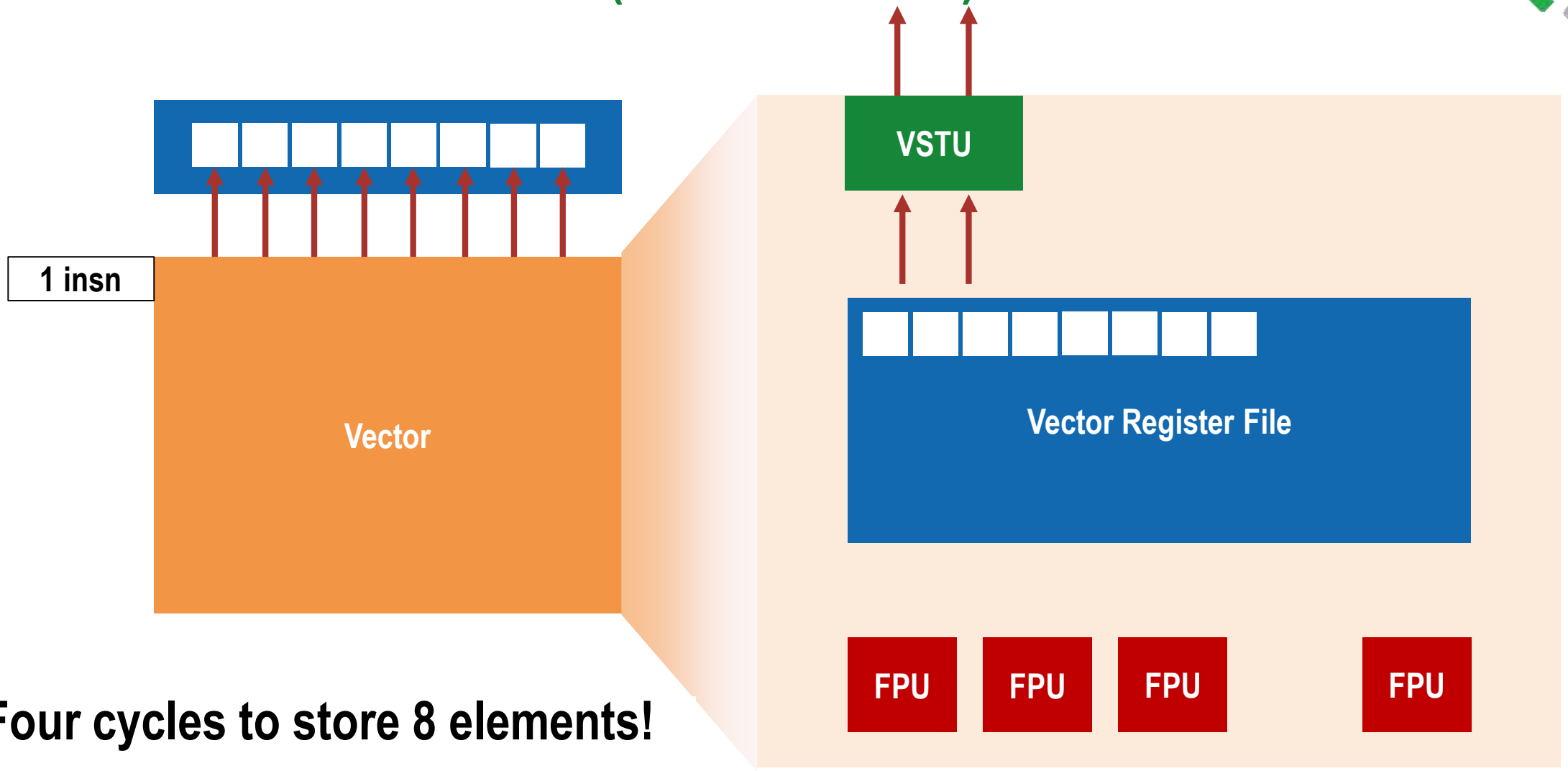
Four cycles to load 8 elements!

Dataflow – Vector arith operation (VRF → FPUs → VRF)



Two cycles to process two vector of 8 elements each!

Dataflow – Vector store (VRF → mem)

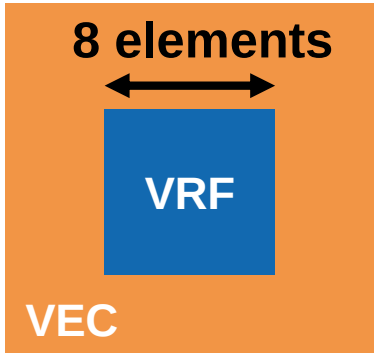


Four cycles to store 8 elements!

How to know the vector register length

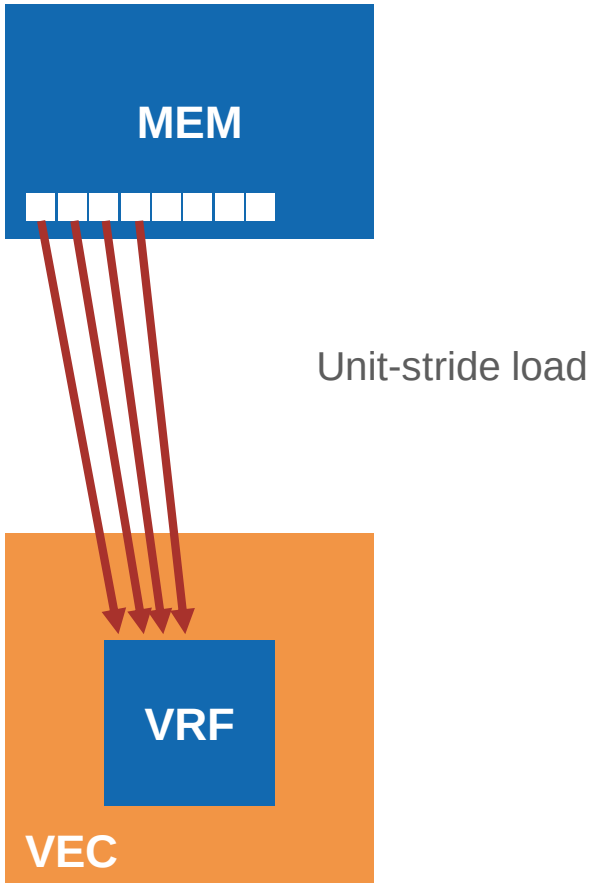
- We can tell the vector architecture which is the length of our vector (AVL)
- The architecture programs itself to handle that AVL
- If AVL is too high, the architecture sets up the maximum vector length it can handle
- The architecture returns back the vector length it will actually handle!
- THIS IS VISIBLE IN SOFTWARE

```
// We have a vector with 17 elements
avl = 17;

// We inform the vector architecture and get back the
// length of the vector that will be processed (vl)
vl = vsetvl(avl)

// If each vreg can host 32 elements -> vl == 17
// If each vreg can host  8 elements -> vl == 8
```



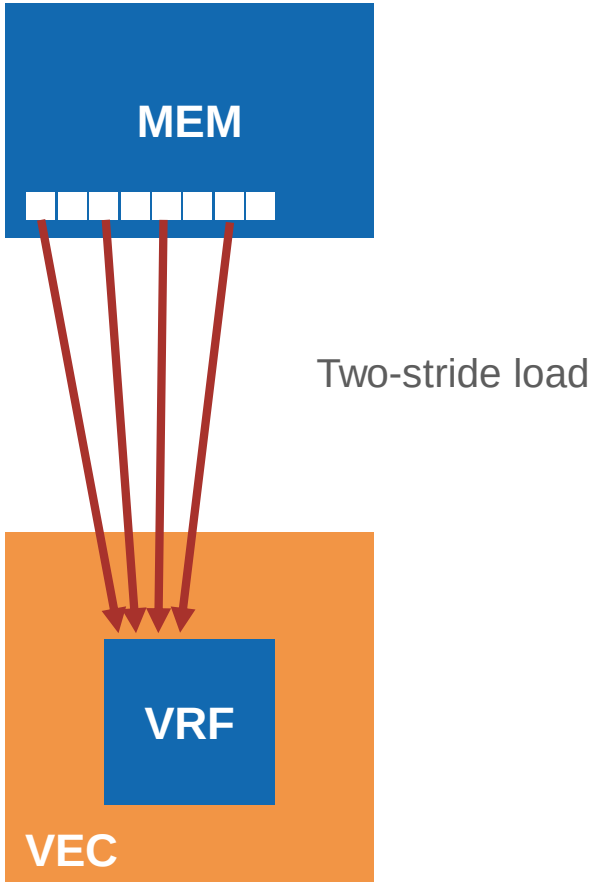
Vector memory operations

- Load/store operations move groups of data between registers and memory
- Three types of addressing
  - Unit stride
    - Contiguous block of information in memory
    - Fastest: always possible to optimize this



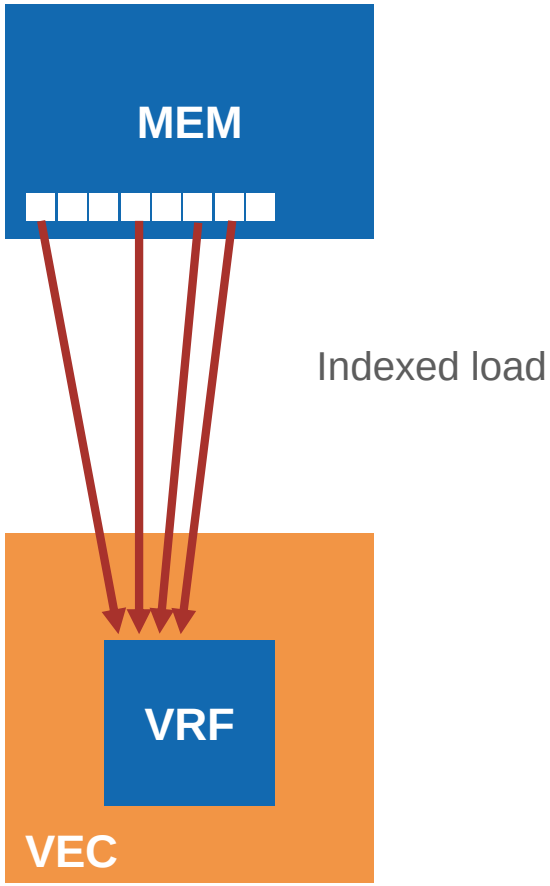
Vector memory operations

- Load/store operations move groups of data between registers and memory
- Three types of addressing
  - Unit stride
    - Contiguous block of information in memory
    - Fastest: always possible to optimize this
  - Non-unit (constant) stride
    - Harder to optimize memory system for all possible strides
    - Prime number of data banks makes it easier to support different strides at full bandwidth



Vector memory operations

- Load/store operations move groups of data between registers and memory
- Three types of addressing
  - Unit stride
    - Contiguous block of information in memory
    - Fastest: always possible to optimize this
  - Non-unit (constant) stride
    - Harder to optimize memory system for all possible strides
    - Prime number of data banks makes it easier to support different strides at full bandwidth
  - Indexed (gather-scatter)
    - Vector equivalent of register indirect
    - Good for sparse arrays of data
    - Increases number of programs that vectorize



Vector Scatter-Gather

- Want to vectorize loops with indirect accesses:

```
for (i=0; i<N; i++)
    A[i] = B[i] + C[D[i]]
```

Vector Scatter-Gather

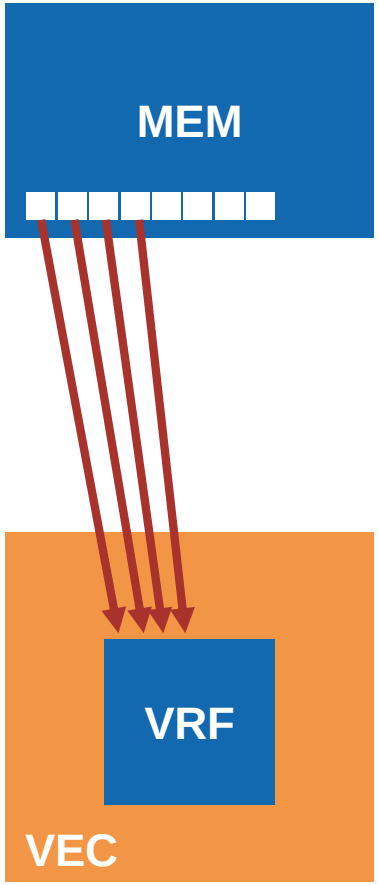
- Want to vectorize loops with indirect accesses:

```
for (i=0; i<N; i++)
    A[i] = B[i] + C[D[i]]
```

```
VLE   vD, rD          # Load indices in D vector
```

`rD` is a scalar register and contains the memory address of D

Unit-stride load



Vector Scatter-Gather

- Want to vectorize loops with indirect accesses:

```
for (i=0; i<N; i++)
    A[i] = B[i] + C[D[i]]
```

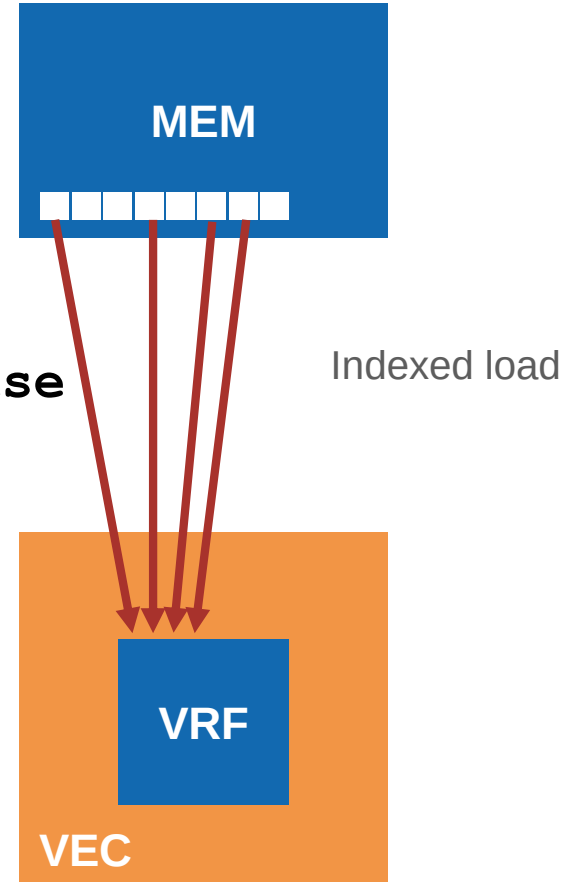
- Indexed load instruction (Gather)

```
VLE  vD, rD          # Load indices in D vector
VLEX vC, rC, vD      # Load indirect from rC base
```

C memory address

D indices

Indexed load



Vector Scatter-Gather

- Want to vectorize loops with indirect accesses:

```
for (i=0; i<N; i++)
    A[i] = B[i] + C[D[i]]
```

- Indexed load instruction (Gather)

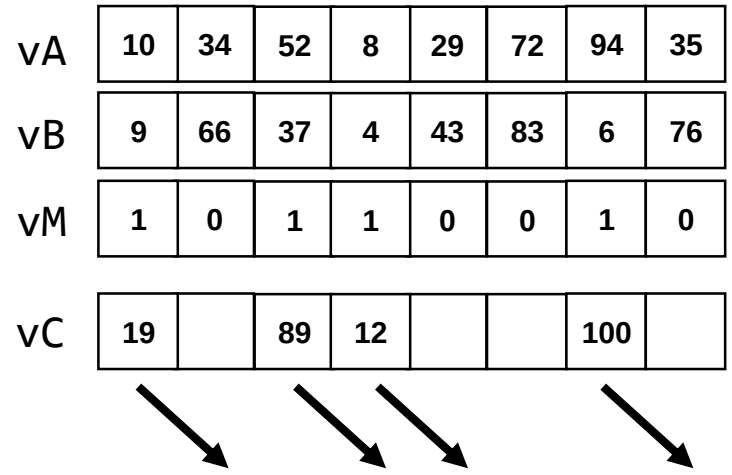
```
VLE  vD, rD          # Load indices in D vector
VLEX vC, rC, vD      # Load indirect from rC base
VLE  vB, rB          # Load B vector
VADD vA, vB, vC      # Do add
VSE  vA, rA          # Store result
```

Vector Conditional Execution

- Problem: Want to vectorize loops with conditional code:

```
for (i=0; i<N; i++)
    if (A[i]>B[i]) then
        C[i] = A[i] + B[i];
```

- Solution: Add vector *mask* (or *flag*) registers
  - vector version of predicate registers, 1 bit per element
- …and *maskable* vector instructions
  - vector operation becomes NOP at elements where mask bit is clear



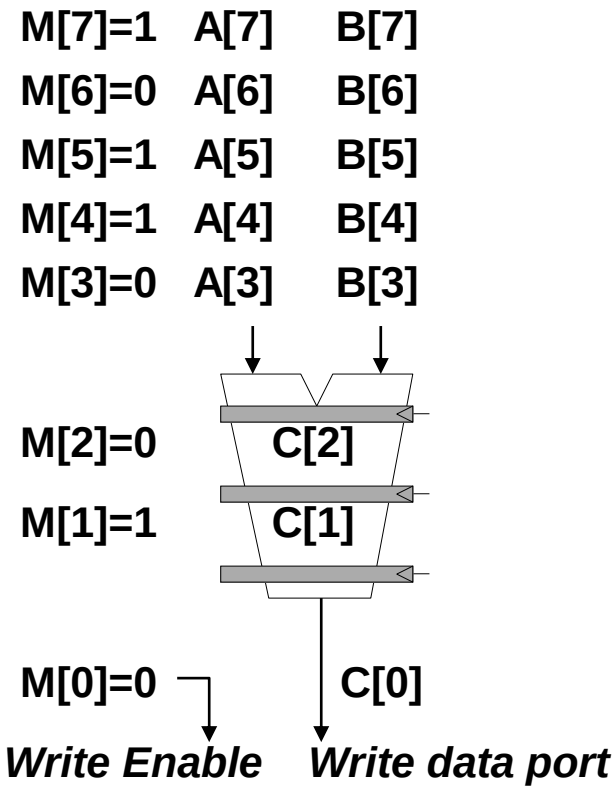
- Code example:

```
VLE      vA, rA            # Load entire A vector
VLE      vB, rB            # Load entire B vector
VMSGT    vM, vA, vB        # Set bits in mask register where A>B
VADD.M   vC, vA, vB, vM    # Add A and B under mask
VSE.M    vC, rC, vM        # Store C back to memory under mask
```

Masked Vector Instructions

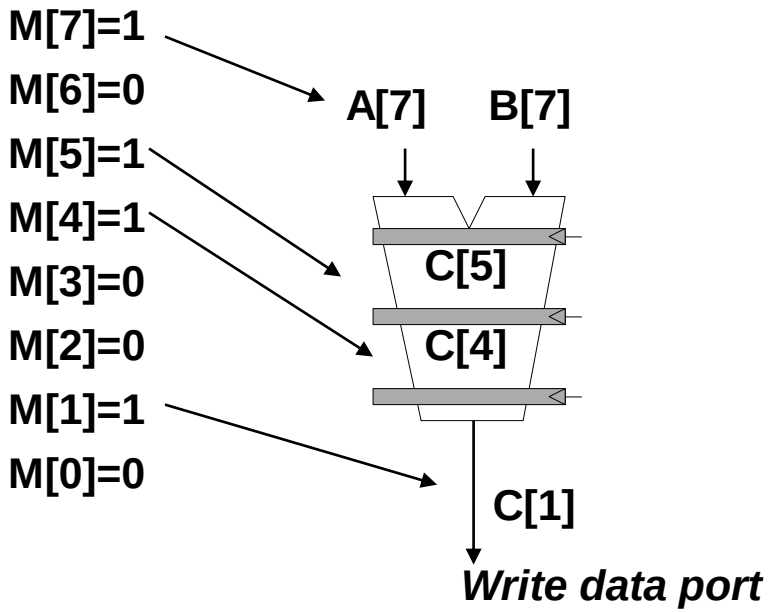
Simple Implementation

Execute all N operations, turn off result writeback according to mask



Density-Time Implementation

Scan mask vector and only execute elements with non-zero masks



Credit: Krste Asanovic

Hands-on (guided) – Predication with masks

1. Follow instructions of the main README.md → *Arathon SPECIAL* paragraph
    1. `As before, but now let's change the exercise name`
    2. `ex_name=MASK`
    3. Write a scalar kernel to add the vector elements if a[i] > b[i]
    4. Starting from the previous vector kernel, let's add predication with masks together!
    5. Inspect apps/bin/arathon.dump
    6. Simulate with waveforms and printf!

Vector Reductions

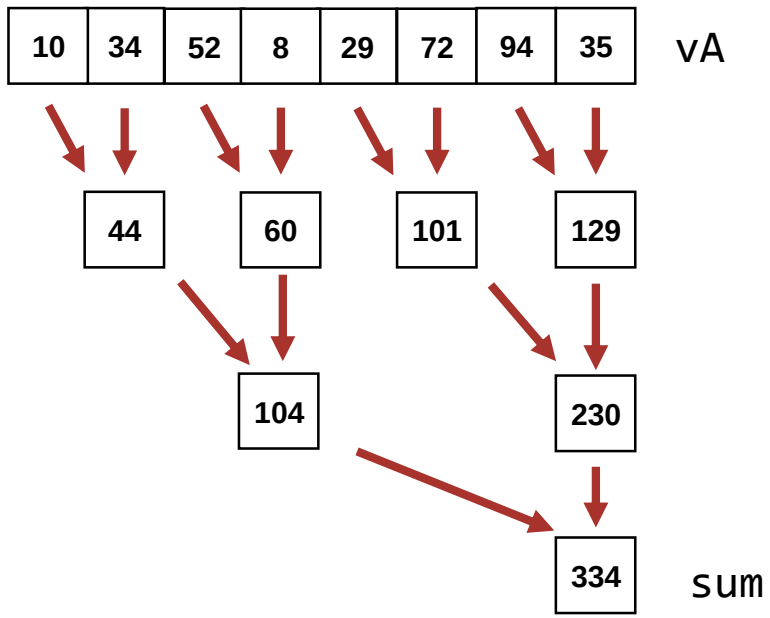
- Problem: Loop-carried dependence on reduction variables

```
sum = 0;
for (i=0; i<N; i++)
    sum += A[i];  # Loop-carried dependence on sum
```

- Solution: vector reduction

```
VLE      vA, rA
VREDSUM sum, vA
```

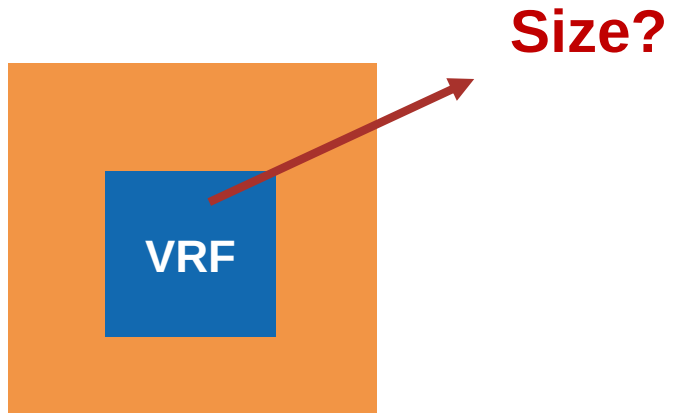
- Reductions are intra-vector operations, whereas most of the other vector instructions are element-wise between two vectors



A vector length problem

- Problem:
  - You want to add two 143-element vectors
  - You don't know your vector architecture and its VRF size
  - You will probably not be able to process the whole vector in one iteration

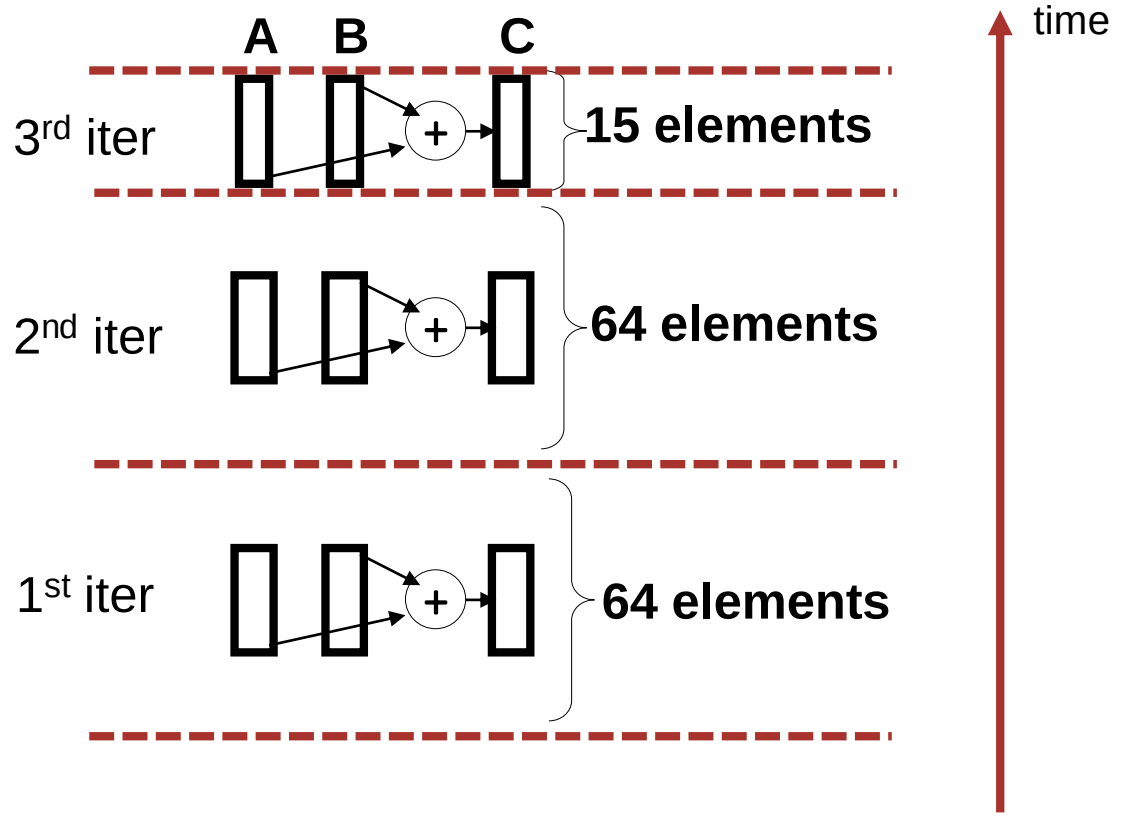
```
// We have a vector with 143 elements
avl = 143;

// We can query the architecture
vl = vsetvl(0)   // vl == 0
vl = vsetvl(8)   // vl == 8
vl = vsetvl(35)  // vl == 35
vl = vsetvl(63)  // vl == 63
vl = vsetvl(64)  // vl == 64
vl = vsetvl(65)  // vl == 64
vl = vsetvl(143) // vl == 64
// We can process up to 64 elements with a vector instruction!
```



Vector stripmining

- What if **# data elements > # elements in a vector register**?
  - Idea: Break loops so that each iteration operates on # elements in a vector register
  - E.g., 143 data elements, 64-element VRF
    - 2 iterations where VLEN = 64
    - 1 iteration where VLEN = 15
- Called **vector stripmining**

```
for (i=0; i<N; i++)
    C[i] = A[i]+B[i];
```



Vector code is portable

- Vector code is agnostic to the vector length…
- ONLY IF YOUR CODE SUPPORTS STRIPMINING!
- Proper vector code can work on different architectures with different VRF sizes

```
void vadd(&a, &b, &c, avl) {
 // Strip-mine loop
 while (avl > 0) {
  // Set up the vector length for the current chunks
  vl = vsetvl(avl);
  // Load the two vector chunks
  // Add the two vector chunks
  // Store the vector chunk
  // Bump pointers
  …
  // Account for the chunks we have already processed!
  avl -= vl;
 }
}
```

The vector length is treated as **a variable,** but **should appear in software** nevertheless**!**

Hands-on – Stripmining

1. Follow instructions of the main README.md → *Arathon SPECIAL* paragraph
    1. `ex_name=VADD_STRIPMINE`
    2. Inspect apps/bin/arathon.dump
    3. Simulate with waveforms or printf!

Vector-vector

```
for (i = 0; i < N; i++) {
    C[i] = A[i] + B[i];
}
```

Register **a0** holds **N**

Register **a1** holds **@A[0]**

Register **a2** holds **@B[0]**

Register **a3** holds **@C[0]**

```
stripmined_loop:
  vsetvli t0, a0, e64 # t0 holds amount done

  vle64.v v0, (a1)    # Load strip of vector A
  vle64.v v1, (a2)    # Load strip of vector B

  vadd.vv v2,v0,v1    # Add vectors

  vse64.v v2, 0(a3)   # Store strip of vector C

  slli t1,t0,3        # t0 * 8 to get bytes
  add a1,a1,t1        # Bump pointers
  add a2,a2,t1
  add a3,a3,t1
  sub a0,a0,t0        # Subtract amount done
  bnez a0, stripmined_loop
```

Vector-vector **fp** add

```
for (i = 0; i < N; i++) {
    C[i] = A[i] + B[i];
}
```

Register **a0** holds **N**

Register **a1** holds **@A[0]**

Register **a2** holds **@B[0]**

Register **a3** holds **@C[0]**

```
stripmined_loop:
  vsetvli t0, a0, e64 # t0 holds amount done

  vle64.v v0, (a1)    # Load strip of vector A
  vle64.v v1, (a2)    # Load strip of vector B

  vfadd.vv v2,v0,v1   # Add vectors

  vse64.v v2, 0(a3)   # Store strip of vector C

  slli t1,t0,3        # t0 * 8 to get bytes
  add a1,a1,t1        # Bump pointers
  add a2,a2,t1
  add a3,a3,t1
  sub a0,a0,t0        # Subtract amount done
  bnez a0, stripmined_loop
```

Vector-scalar add

```
for (i = 0; i < N; i++) {
    C[i] = A[i] + B;
}
```

Register **a0** holds **N**

Register **a1** holds **@A[0]**

Register **a2** holds **B**

Register **a3** holds **@C[0]**

```
stripmined_loop:
  vsetvli t0, a0, e64 # t0 holds amount done

  vle64.v v0, (a1)    # Load strip of vector A


  vfadd.vx v2,v0,a2   # Add vectors

  vse64.v v2, 0(a3)   # Store strip of vector C

  slli t1,t0,3        # t0 * 8 to get bytes
  add a1,a1,t1        # Bump pointers

  add a3,a3,t1
  sub a0,a0,t0        # Subtract amount done
  bnez a0, stripmined_loop
```


P
P

RISC-V Vector Extension

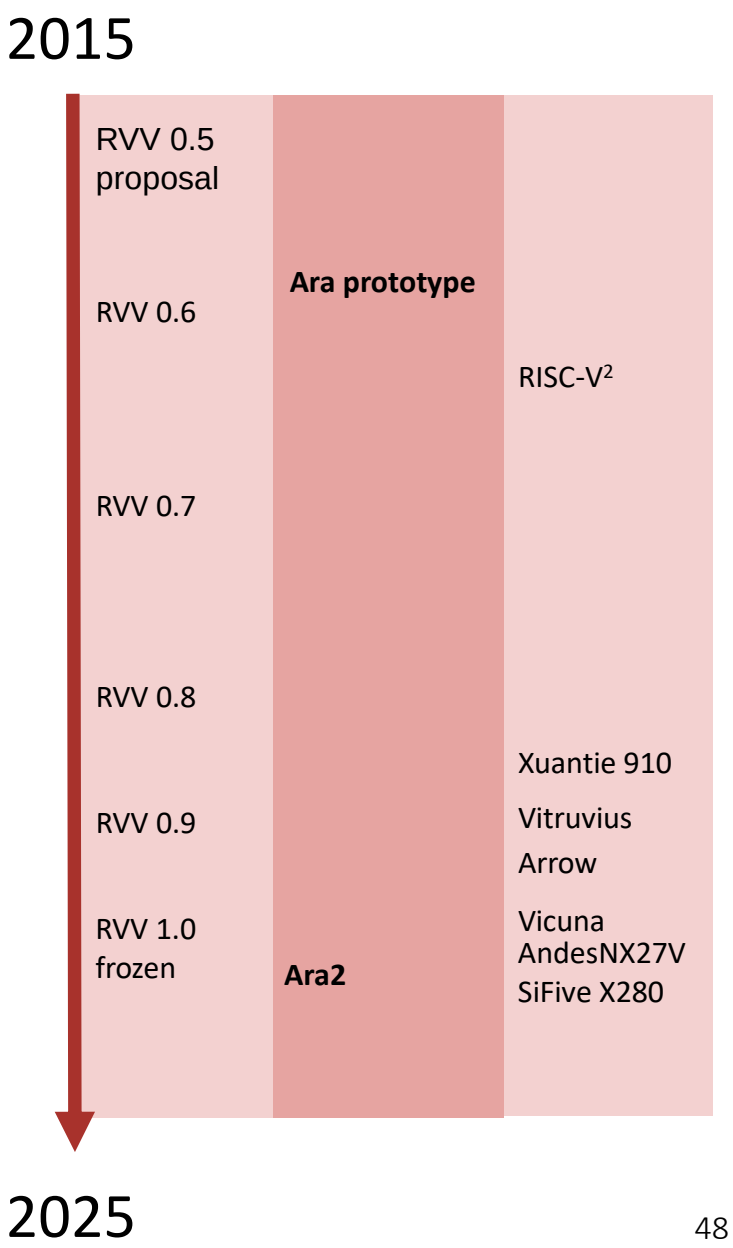
- Standard extension to the **RISC-V ISA**
- An updated form of **Cray-style vectors** for modern microprocessors

- The **latest version** of the **specs** is kept at https://riscv.org/specifications/ratified
  https://drive.google.com/file/d/1uviu1nH-tScFfgrovvFCrj7Omv8tFtkp/view

RISC-V V ISA - A long journey

- Early draft in **2015**
- Years of refinment. Frozen at the end of 2021.
- **Known programming model**

- Comprehensive ISA (+300 instructions)
  - VRF setup instructions (VL, element width, ...)
  - Arithmetic instructions (int, fixpt, fp)
  - Memory operations (unit-stride, strided, indexed)
  - Predicated execution (masks)
  - Permutation instructions (slide, reorder) **Vector length agnostic** – High reusability!



Hands-on – RISCV-V specs

Let's have a look at the **V specifications** structure

- **Two parameters**
  - ELEN → Maximum size [bits] of a vector element
  - VLEN → Number of bits of a single vector register
- Some **new CSRs**
- **VRF byte layout**
- **Instructions**
  - Configuration (vsetvli)
  - Memory
  - Arithmetic (int, fixed-point, floating-point)
  - Permutations
  - Masks

Quick summary of the RISC-V Vector ISA

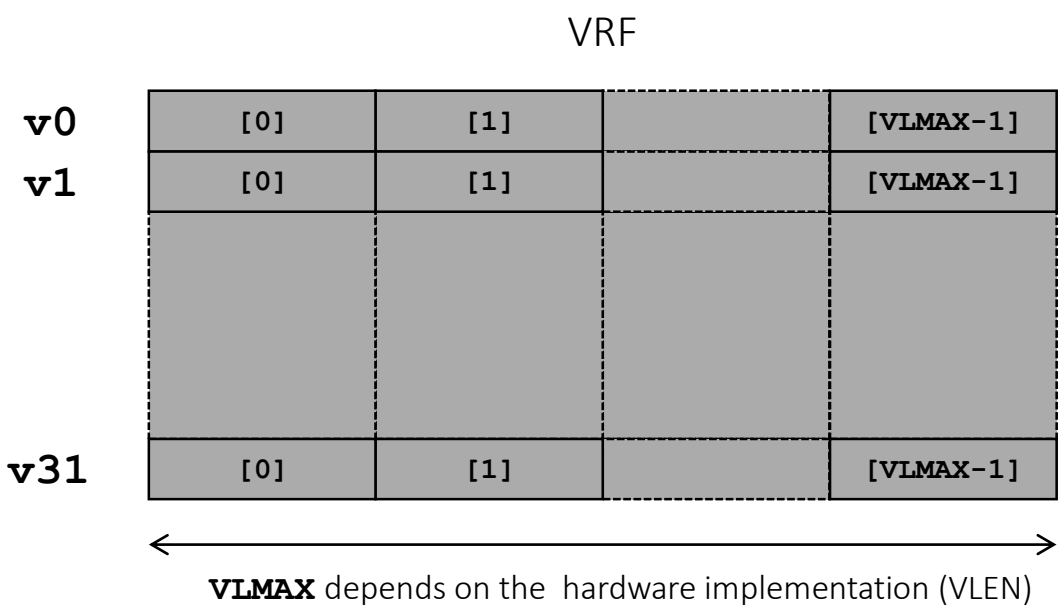
- 32 vector registers, **`v0-v31`**
- **VLEN** is a constant parameter chosen by the HW implementor and must be a power of two - read it as “vector register size (in bits)”
- The vector type **`vtype`** CSR stores the default type to interpret the elements of the vector register
  - Standard Element Width (SEW) in bit:
    - 8, 16, 32, 64-bit…
  - Length multiplier (LMUL):
    - Multiple vector registers can be grouped together, so that a single vector instruction operates on multiple registers
    - LMUL can be 1, 2, 4, 8
  - Vector length register **`vl`** controls the number of elements executed by each instruction

Vector Unit State

vl	Vector length register (writable)
vtype	Vector configuration state (writable)

$0 \le vl \le vlmax(sew, lmul)$

$VLMAX(sew, lmul) = (VLEN / SEW) \times LMUL$



LMUL=1	v0	v1	v2	v3	v4	v5	v6	v7	v8	v9	v10	v11	v12	v13	v14	v15	v16	v17	v18	v19	v20	v21	v22	v23	v24	v25	v26	v27	v28	v29	v30	v31
LMUL=2	v0		v2		v4		v6		v8		v10		v12		v14		v16		v18		v20		v22		v24		v26		v28		v30	
LMUL=4	v0				v4				v8				v12				v16				v20				v24				v28			
LMUL=8	v0								v8								v16								v24							

Vector Unit Configuration

- Can be done through the **`vsetvli`** instruction
- Vector length depends on the element width and length multiplier
  - It should be possible to write assembly code without knowing MAXVL

- **`vsetvli rd, rs1, vtypei`**
  - **`vl`** is set to **`min(VLMAX,rs1)`**, the value is also copied to **`rd`**
  - **`vtype`** is set to **`vtypei`**

  - Example, setting the **`vl`** to 16 elements of 32 bits (if **`VLMAX >= 16`**)

```
li t0, 16
vsetvli t1, t0, e32
```

Hands-on – vsetvli

1. Follow instructions of the main README.md → *Arathon SPECIAL* paragraph
    1. `ex_name=VSETVLI`
    2. `What's the VLEN of Ara?`
    3. What's the longest vector length you can get in Ara?
    4. Can you get something better by changing LMUL?
    5. Can you get something better by changing the element width?
    6. Is VLEN changing?
    7. Also try with a different number of lanes.
    8. Simulate with waveforms and printf! ☺

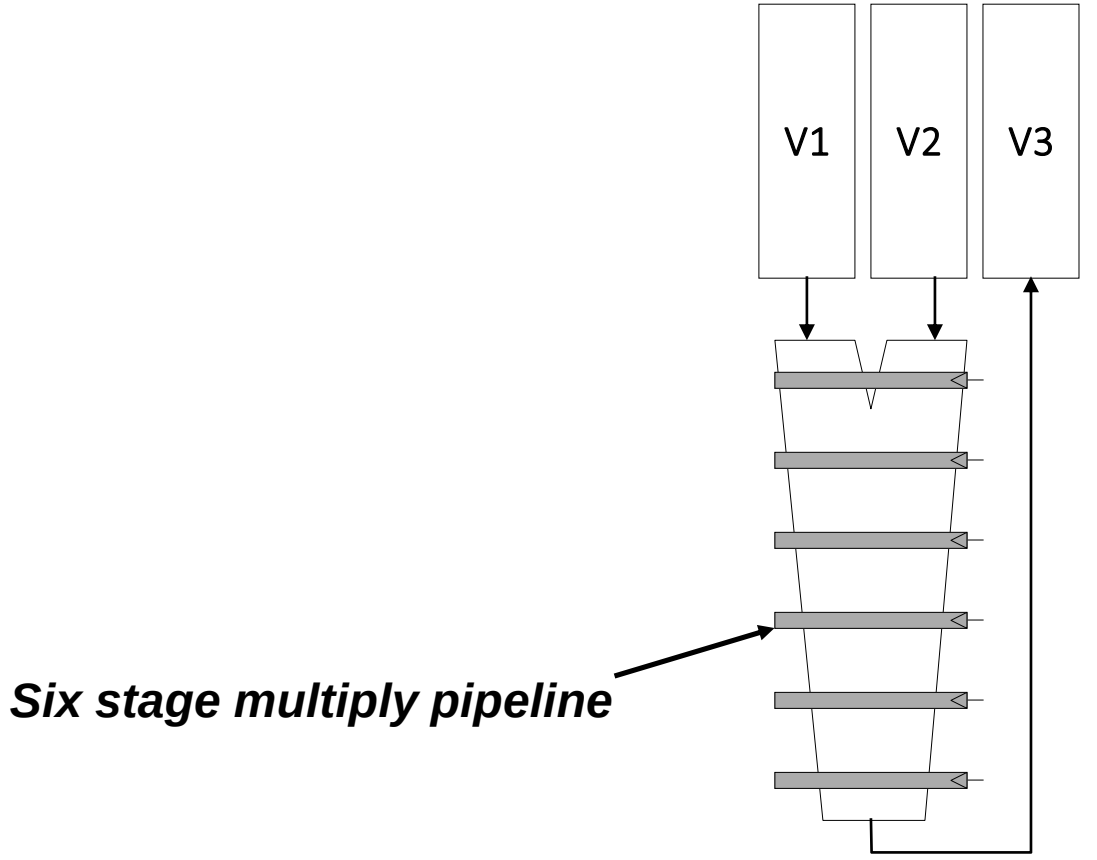

P
P

Vector Instruction Set Advantages

- Compact
  - one short instruction encodes N operations
- Expressive, tells hardware that these N operations:
  - are independent
  - use the same functional unit
  - access disjoint registers
  - access registers in the same pattern as previous instructions
  - access a contiguous block of memory (unit-stride load/store)
  - access memory in a known pattern (strided load/store)
- Scalable
  - can run same object code on more parallel pipelines or lanes

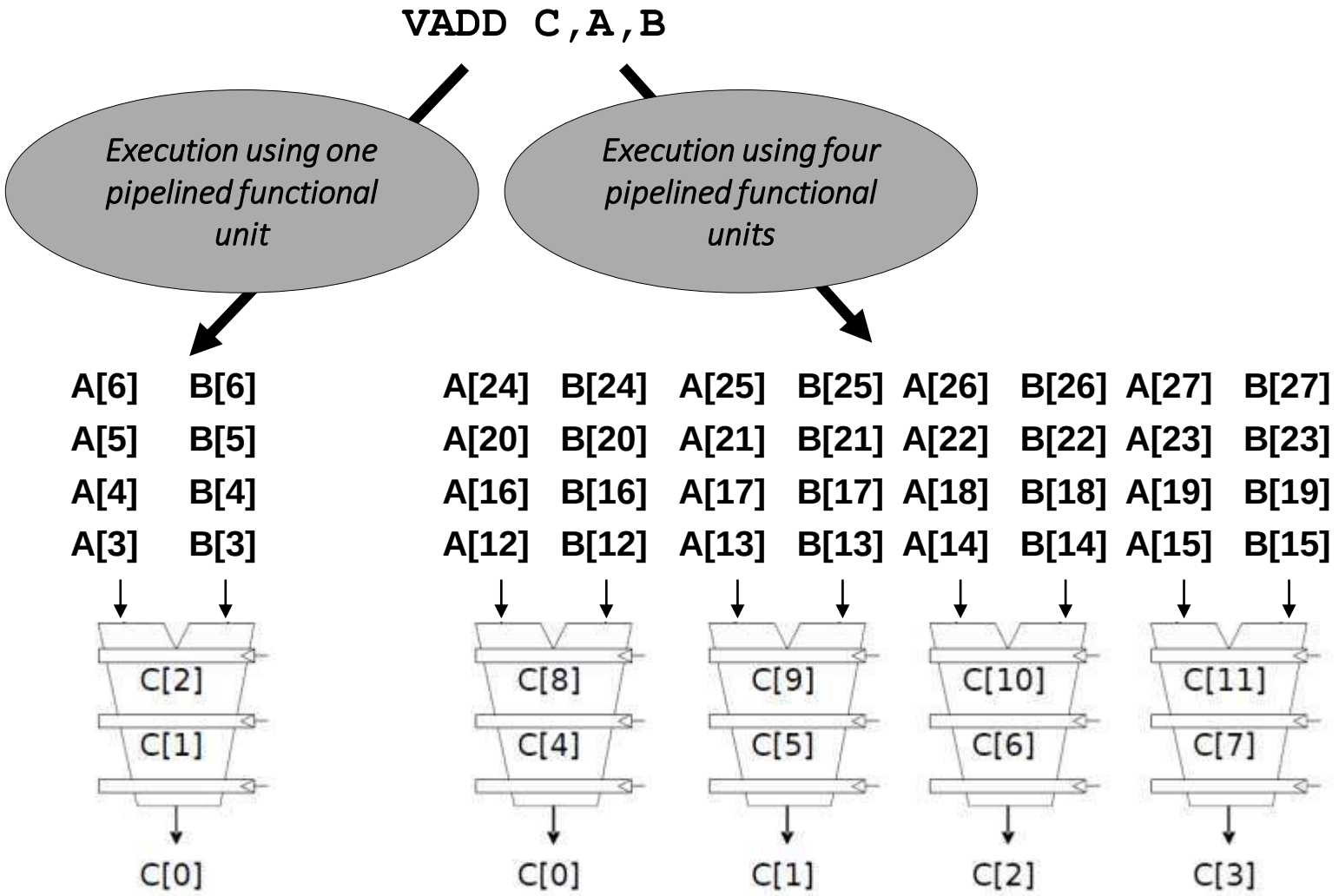
Vector Arithmetic Execution

- Use deep pipeline (**fast clock**) to execute element operations
- Simplifies control of deep pipeline because elements in vector are independent (**no hazards!**)



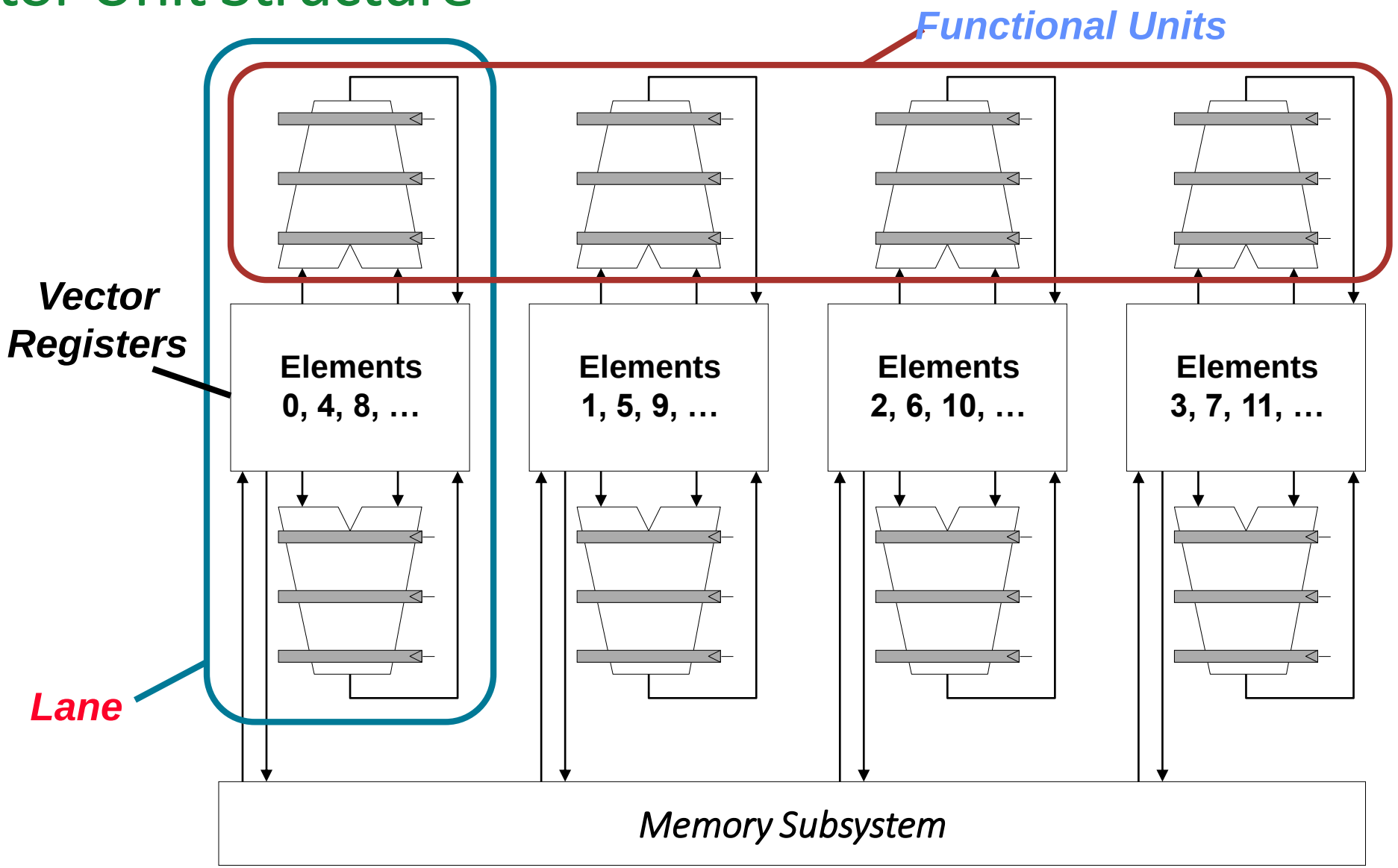
Credit: Krste Asanovic

Vector Instruction Execution



Credit: Krste Asanovic

Vector Unit Structure



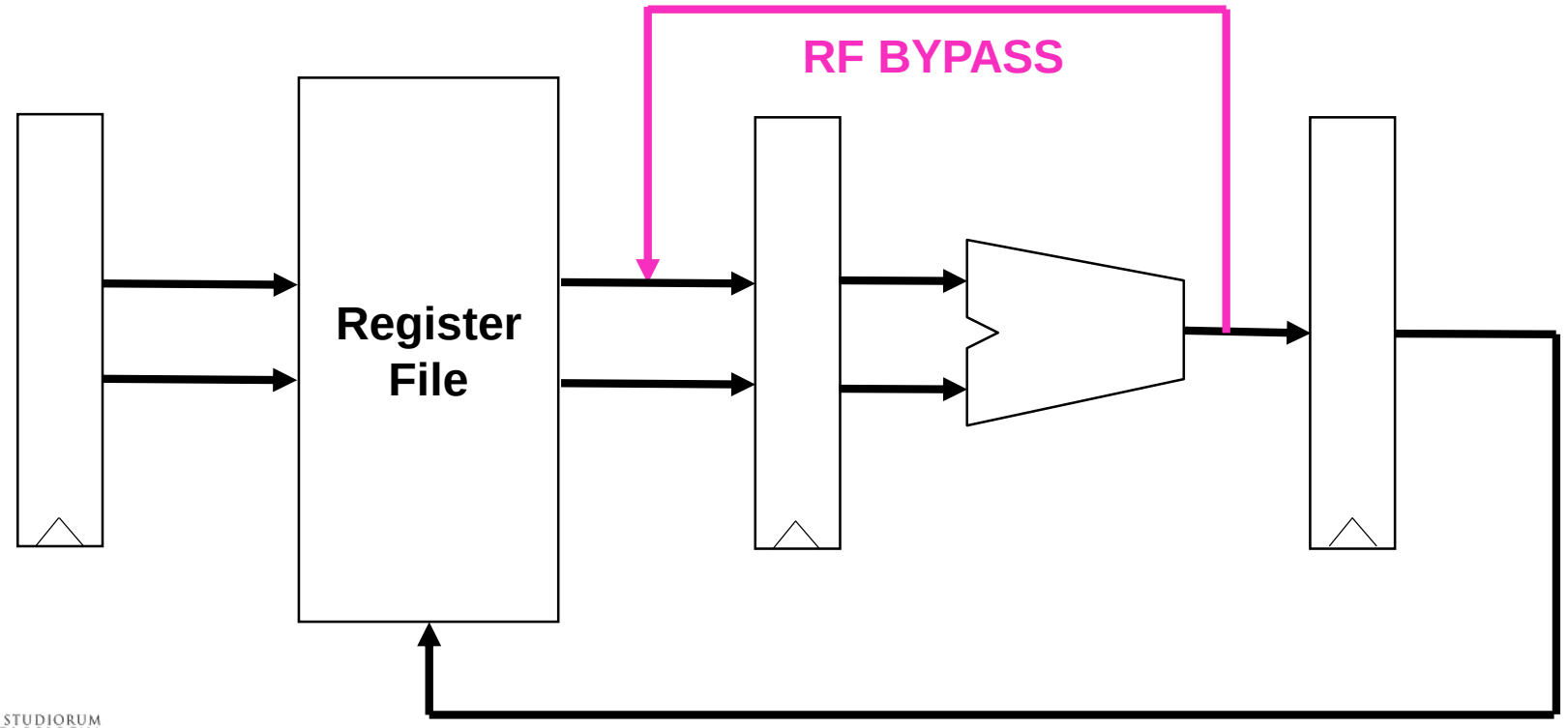
Credit: Krste Asanovic

A refresh: operand forwarding

- Operand forwarding (register bypassing)

```
ADD r1,r2,r2
MUL r4,r1,r3
```

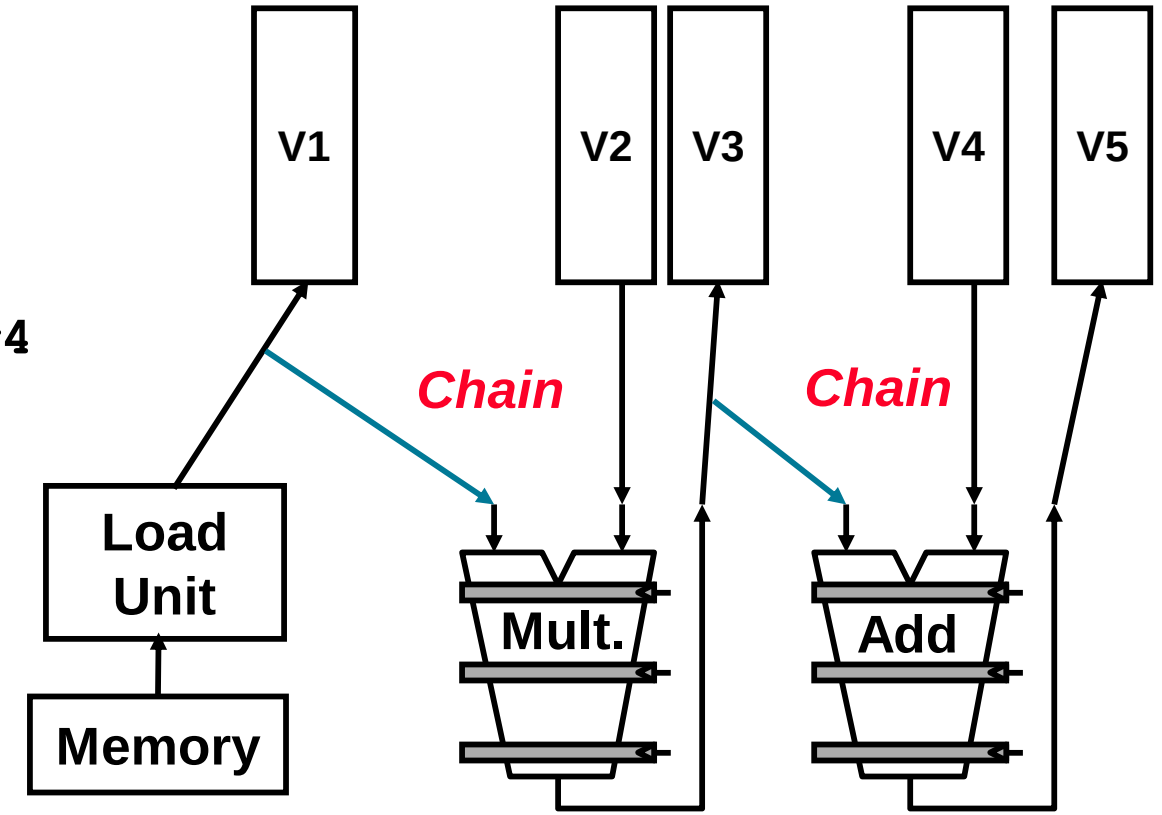
RAW dependency on **r1**!
We should wait until LD has written back before reading operands for MUL



Vector Chaining

- Vector version of register bypassing
- The VRF acts as a collector point

```
VLE  v1
VMUL v3,v1,v2
VADD v5, v3, v4
```

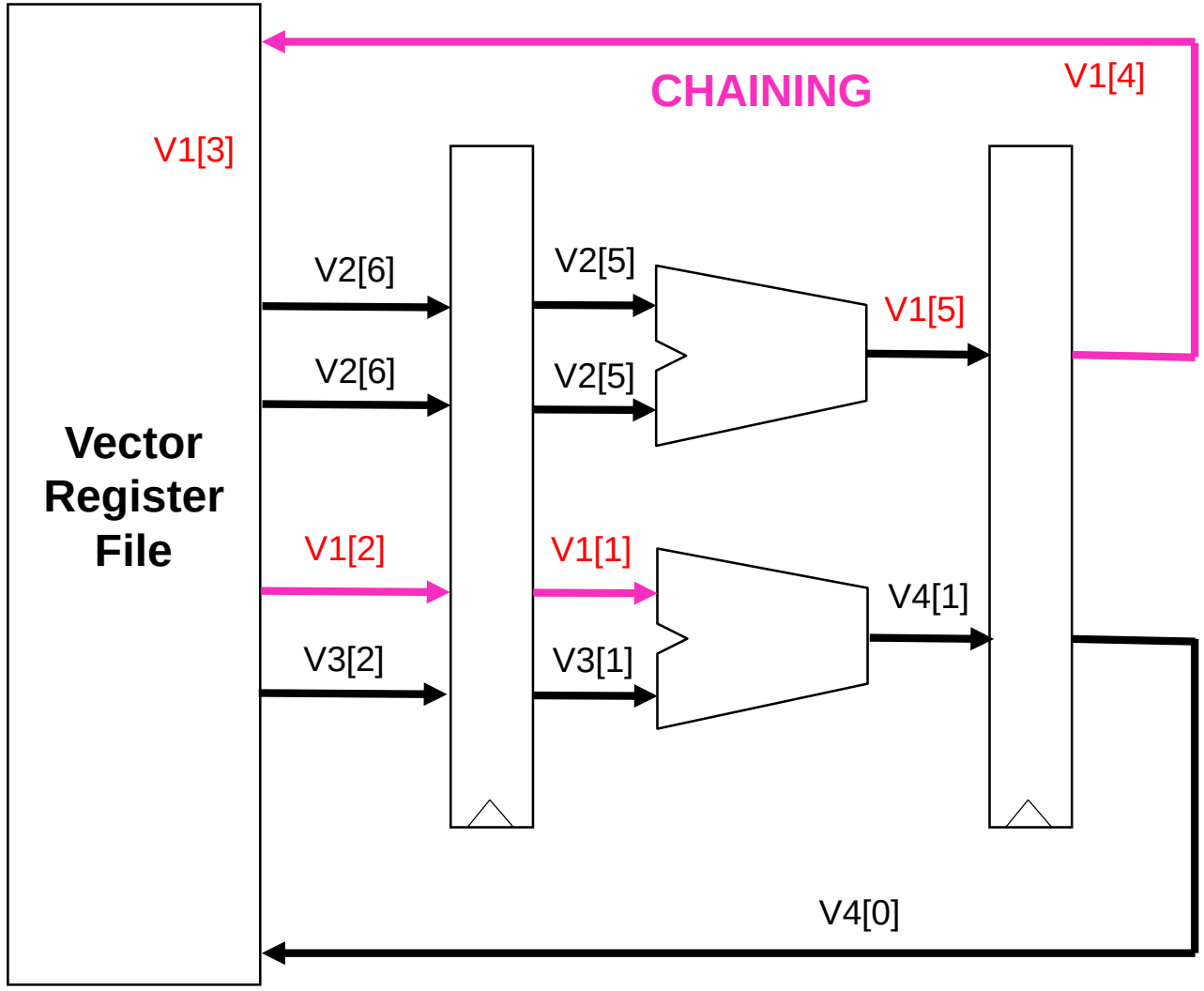


Credit: Krste Asanovic

Vector Flexible Chaining

- We can use VRF as chaining node

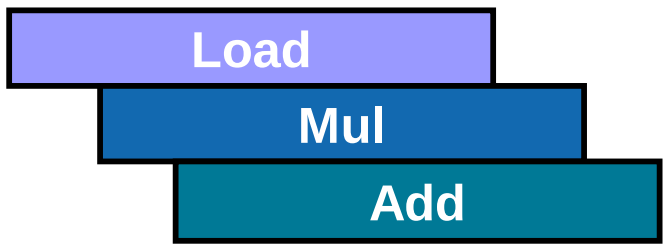
```
VADD v1,v2,v2
VMUL v4,v1,v3
```



Vector Chaining

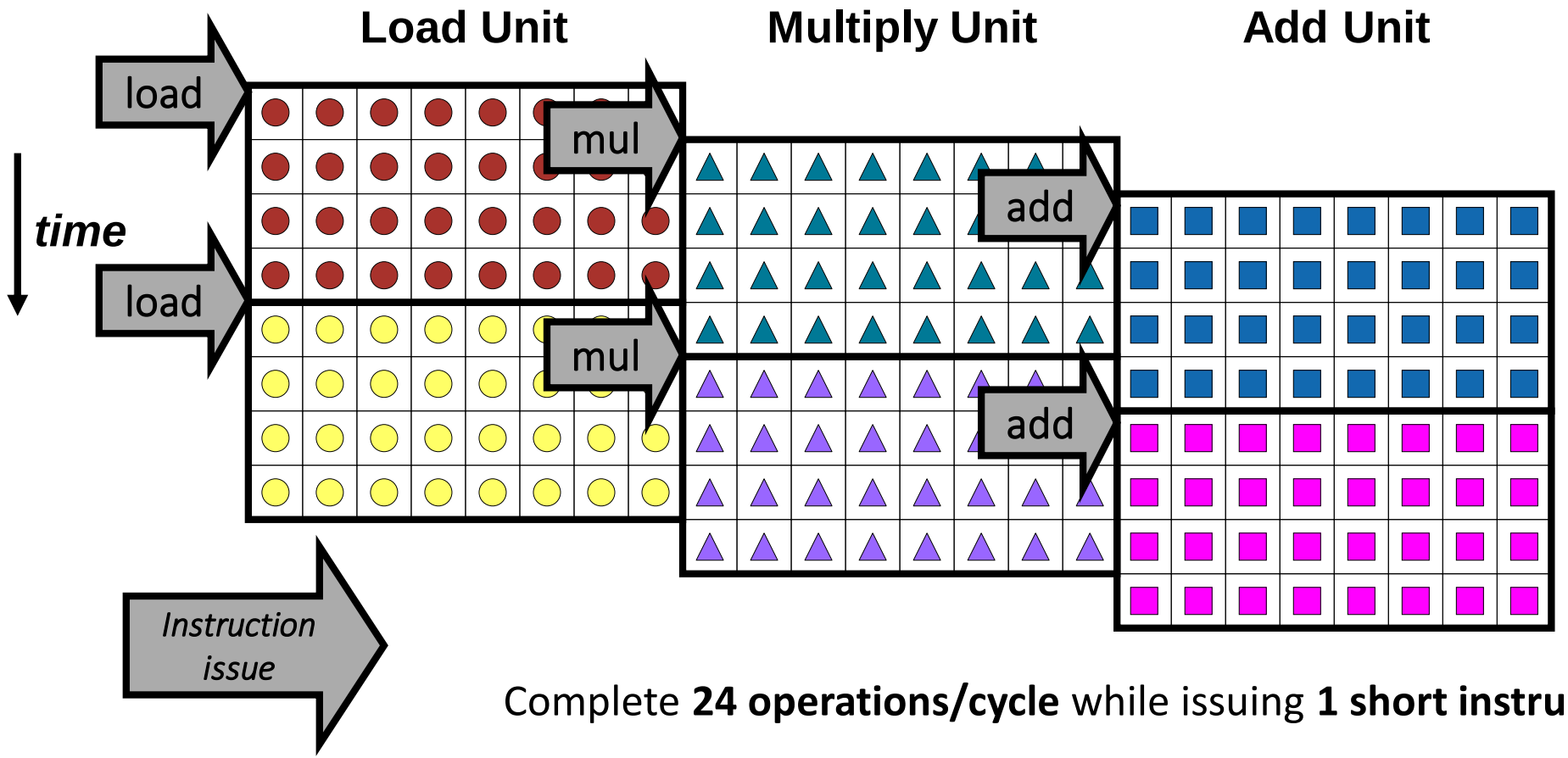
- Without chaining, must wait for last element of result to be written before starting dependent instruction

- With chaining, can start dependent instruction as soon as first result appears



Vector instruction parallelism

- Can overlap execution of multiple vector instructions
  - Example machine has 32 elements per vector register and 8 lanes



Complete **24 operations/cycle** while issuing **1 short instruction/cycle**

Credit: Krste Asanovic

Hands-on – Chaining

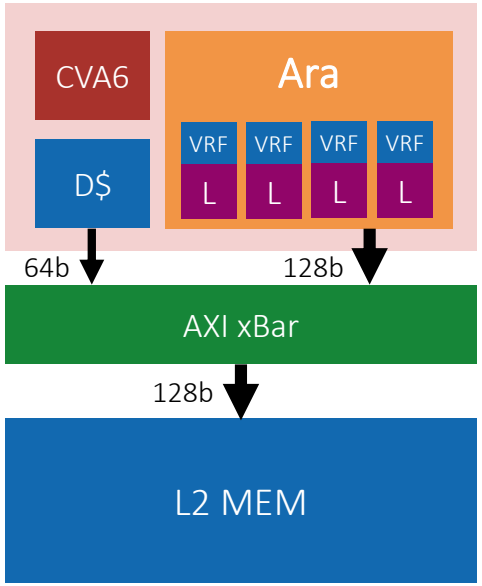
1. Follow instructions of the main README.md → *Arathon SPECIAL* paragraph
    1. `ex_name=CHAINING`
    2. Inspect apps/bin/arathon.dump
    3. Simulate with waveforms or printf!


P
P

Ara2 SoC – High-level architecture

RISC-V V 1.0 vector processor

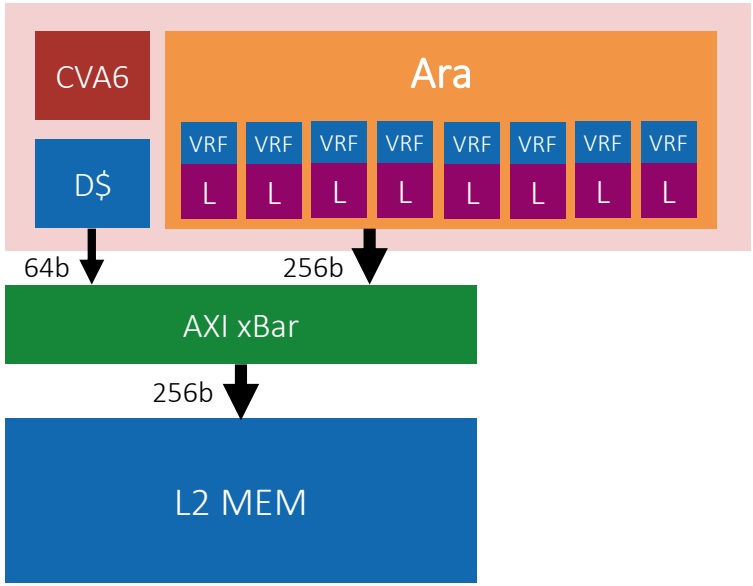
- Decoupled architecture (CVA6 + Ara)
- Private memory ports
- Ara accesses the L2 memory
- Ara uses a wide AXI memory bus
- Lane-based design
- Each lane – 1 FPU
- Split vector register file (VRF)



Ara2 SoC – High-level architecture

RISC-V V 1.0 vector processor

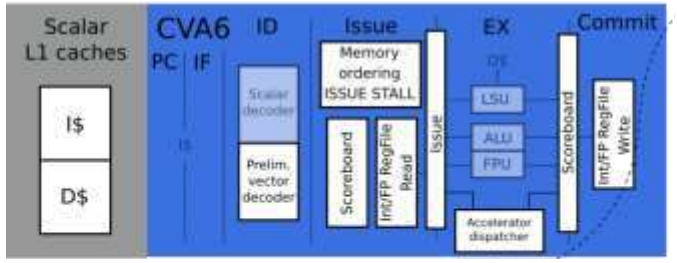
- Decoupled architecture (CVA6 + Ara)
- Private memory ports
- Ara accesses the L2 memory
- Ara uses a wide AXI memory bus
- Lane-based design
- Each lane – 1 FPU
- Split vector register file (VRF)
- Ara is scalable: from 2 to 16 lanes!



Double the lanes → Double the maximum performance!

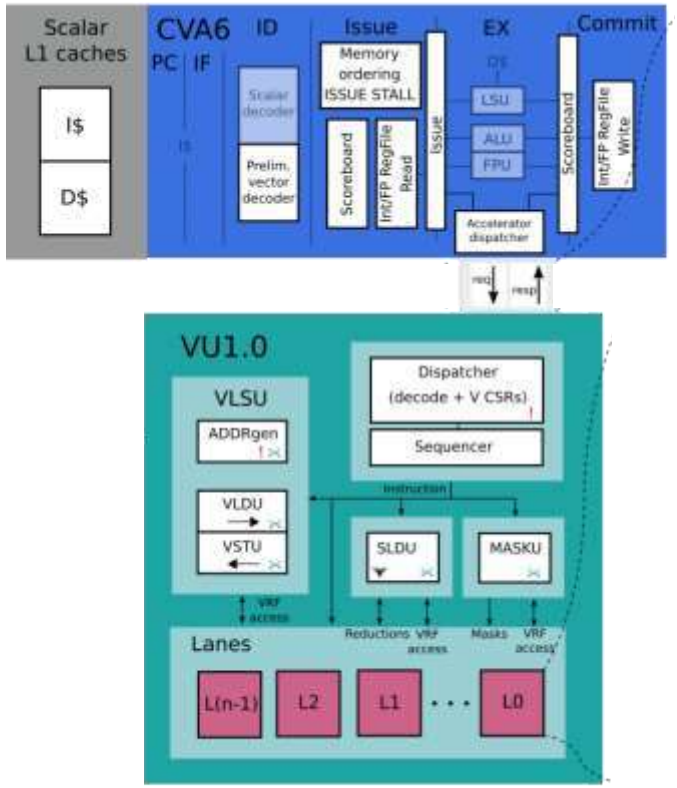
Ara System – Take a scalar core

- Scalar core: **CVA6**



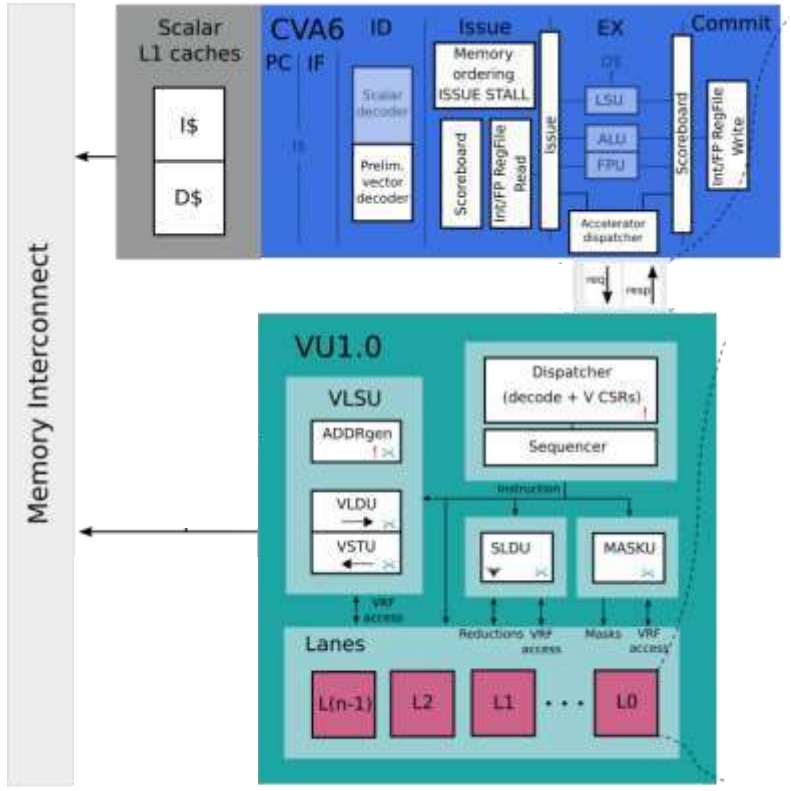
Ara System – Add a vector unit

- Scalar core: **CVA6**
- Vector core: **Ara**



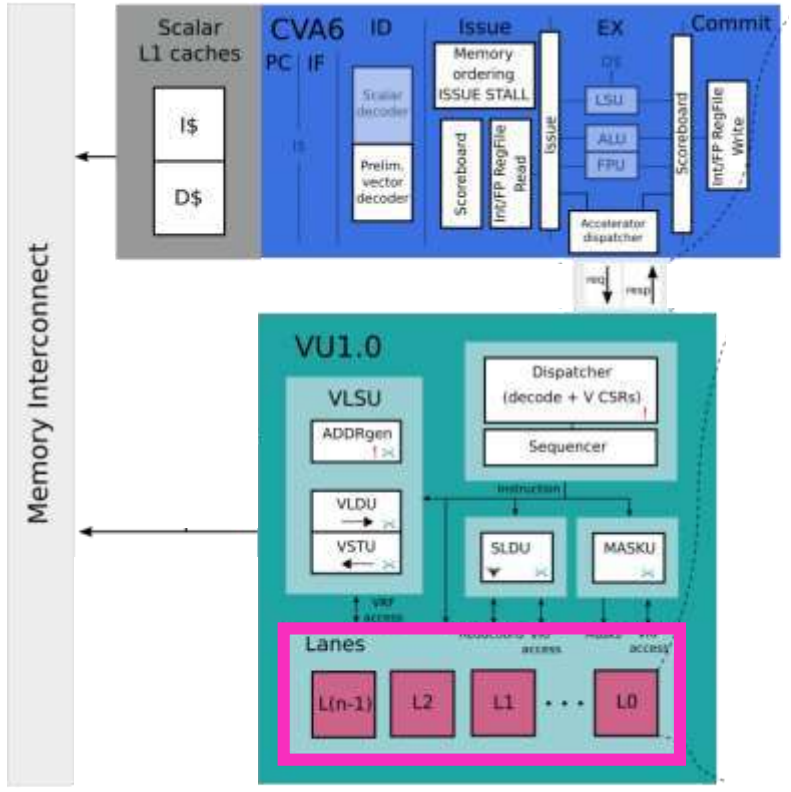
Ara System – Connect to memory

- Scalar core: **CVA6**
- Vector core: **Ara**
- Same **AXI** interconnect



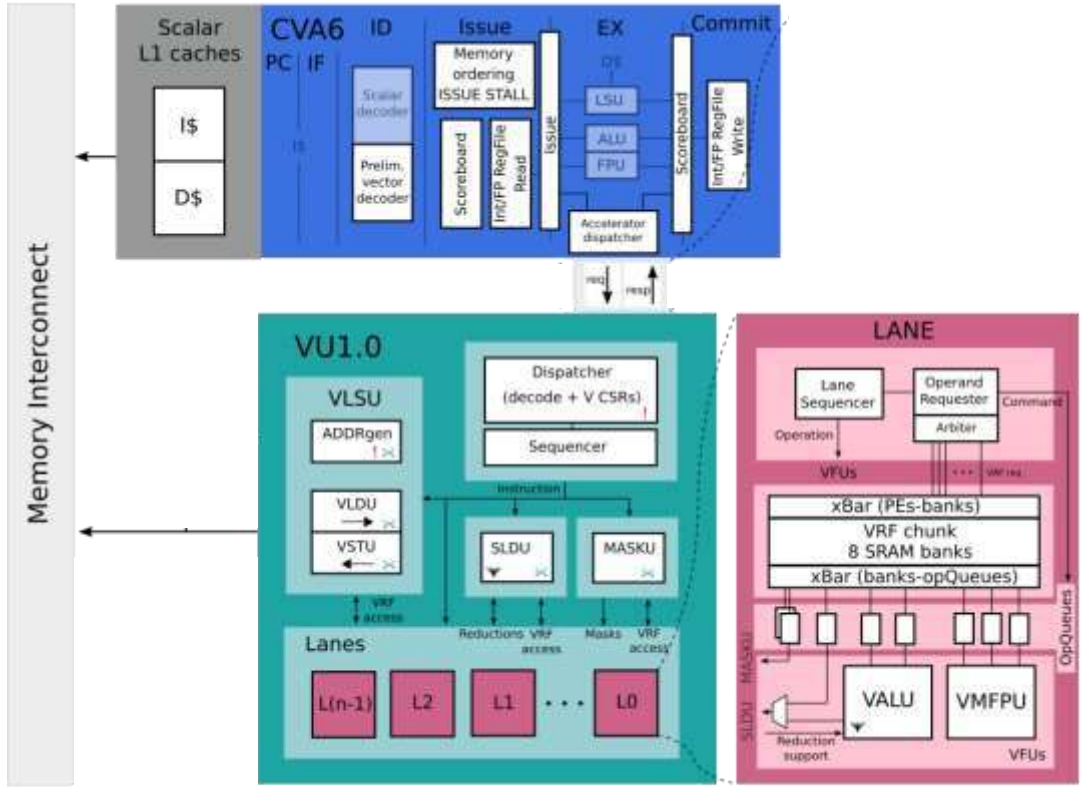
Ara System – Make it scalable with lanes

- Scalar core: **CVA6**
- Vector core: **Ara**
- Same **AXI** interconnect
- Parametric number of Lanes
  - Each lane contains:
    - FPU, ALU (computation)
    - Chunk of the VRF (buffering)



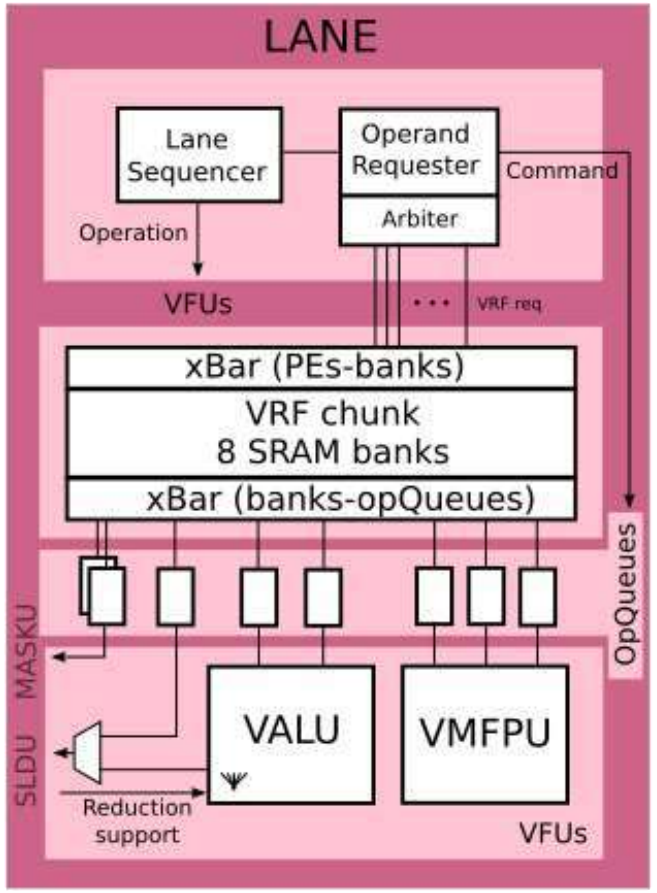
Ara System – Wait… what's in a lane?

- Scalar core: **CVA6**
- Vector core: **Ara**
- Same **AXI** interconnect
- Parametric number of Lanes
  - Each lane contains:
    - FPU, ALU (computation)
    - Chunk of the VRF (buffering)



Ara System – The Lane

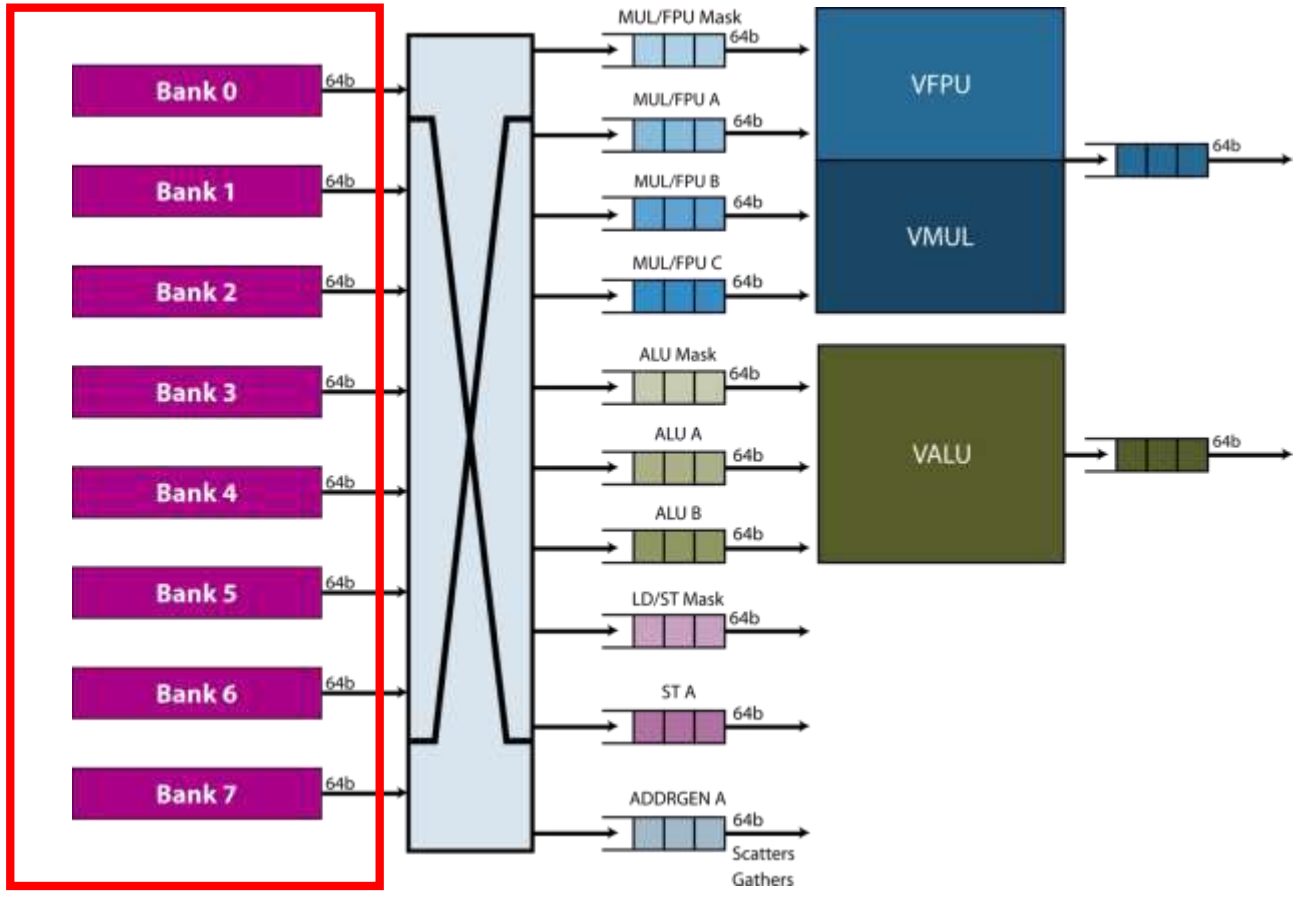
- **Computational** heart
  - One 64-bit SIMD FPU/lane
  - One 64-bit SIMD ALU/lane
- Keep a **chunk** of the **VRF**
- **Max**imize **locality**
  - Element 0 remains in lane 0
  - Minimize inter-lane communication
- VRF in-lane reads/writes



16 Lanes: 32 DP-FLOP/Cycle – 128 HP-FLOP/Cycle!

Ara System - The Lane

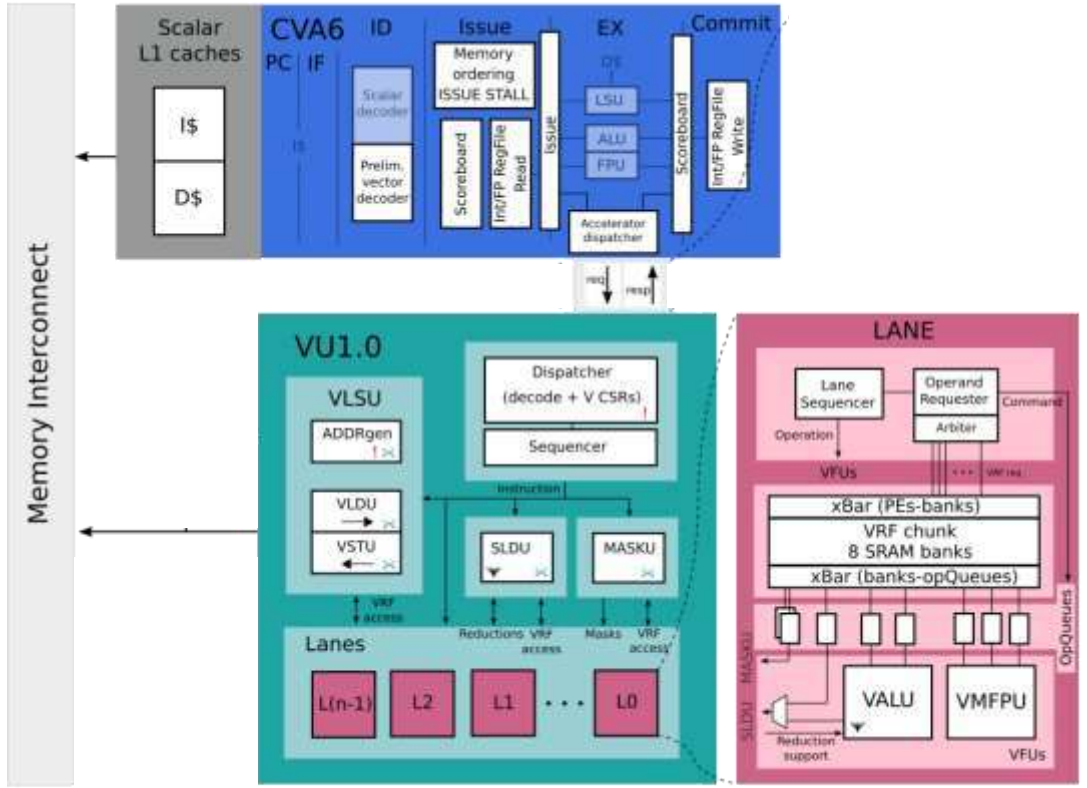
- **Modular VRF**
- **4 KiB/lane**
- **8 SRAM banks/lane**
- **Operand queues**
  - Amortize VRF bank conflicts



VRF Chunk (4 KiB)

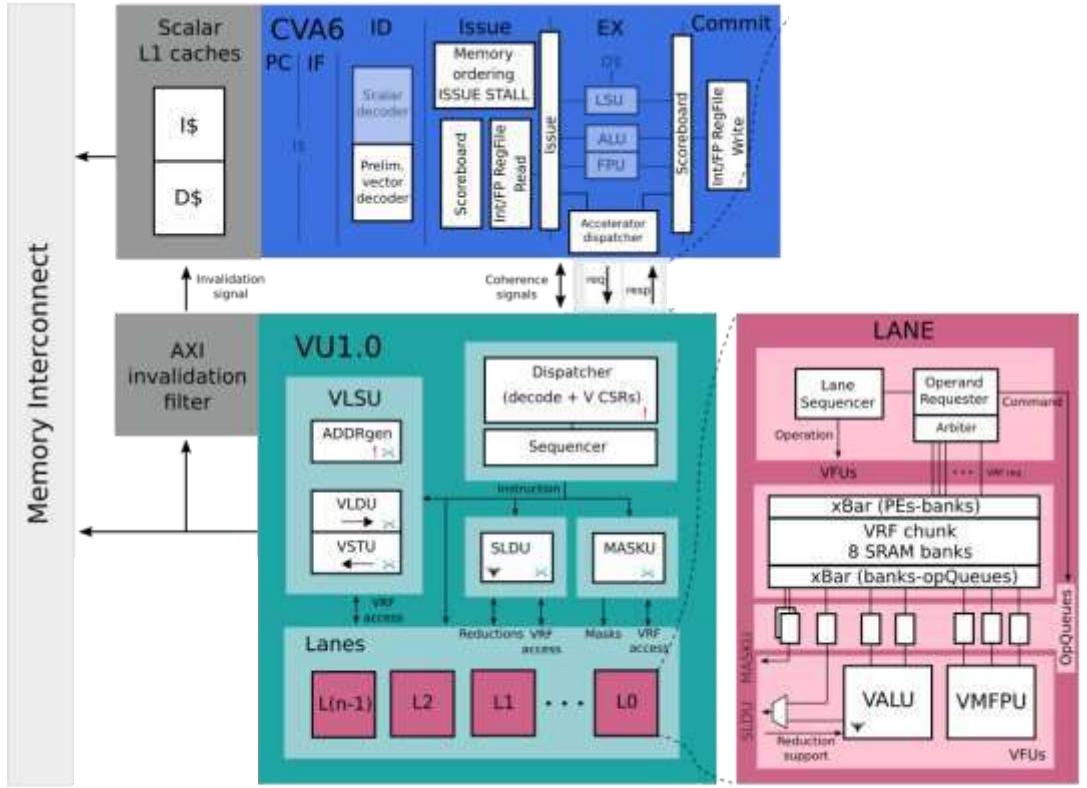
Ara System – Let's add some interfaces

- Scalar core: **CVA6**
- Vector core: **Ara**
- Same **AXI** interconnect
- Parametric number of Lanes
  - Each lane contains:
    - FPU, ALU (computation)
    - Chunk of the VRF (buffering)



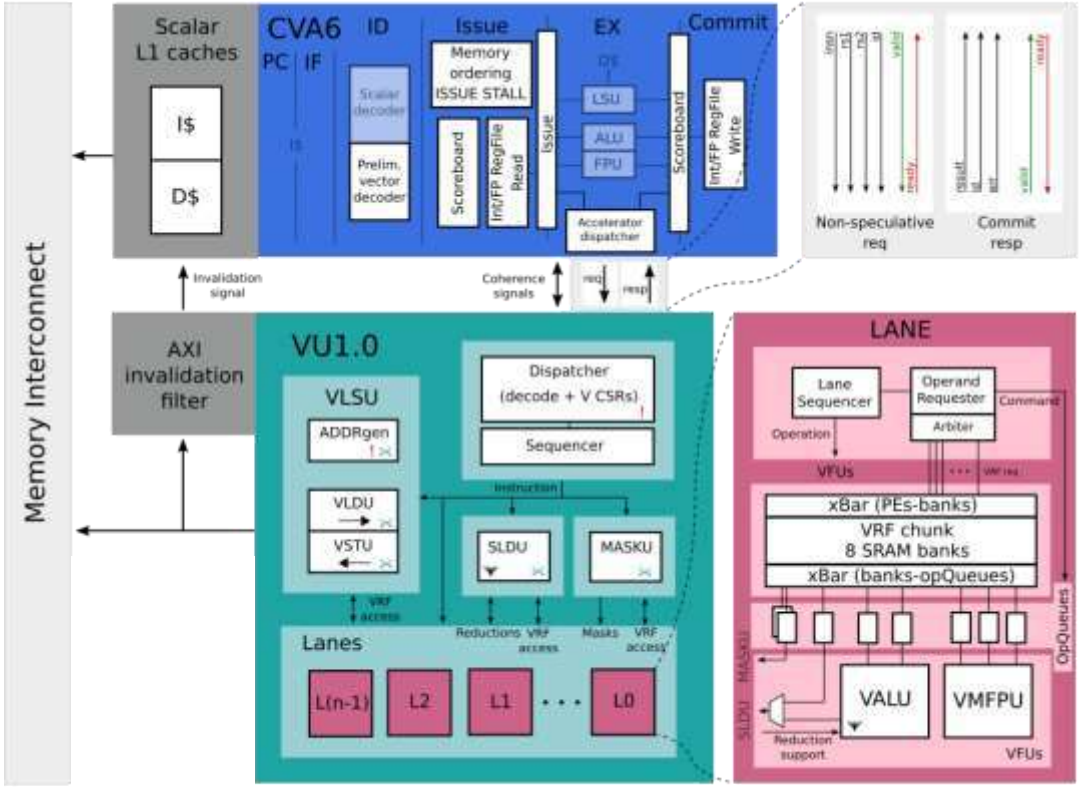
Ara System – Let's add some interfaces

- Scalar core: **CVA6**
- Vector core: **Ara**
- Same **AXI** interconnect
- Parametric number of Lanes
  - Each lane contains:
    - FPU, ALU (computation)
    - Chunk of the VRF (buffering)
- Memory coherence



Ara System – Let's add some interfaces

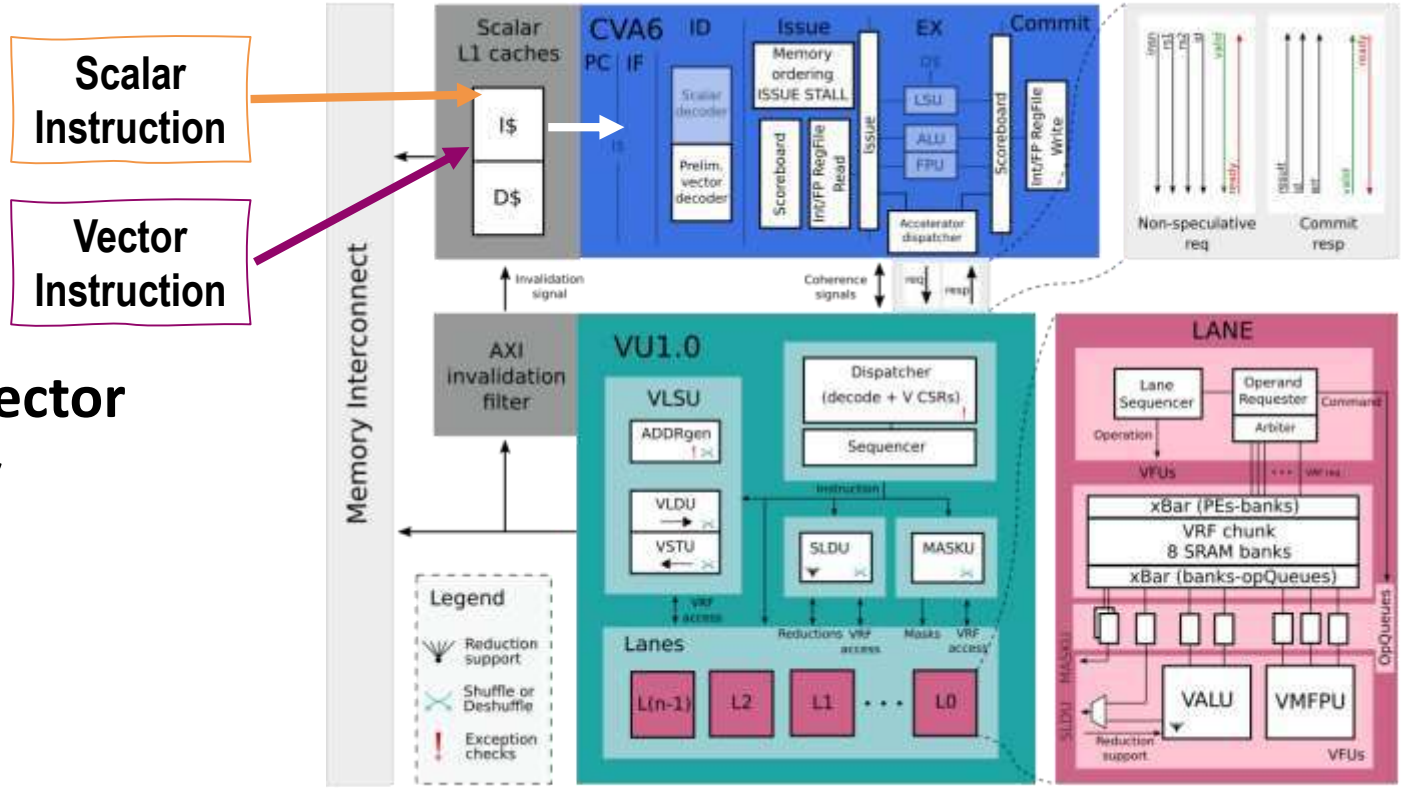
- Scalar core: **CVA6**
- Vector core: **Ara**
- Same **AXI** interconnect
- Parametric number of Lanes
  - Each lane contains:
    - FPU, ALU (computation)
    - Chunk of the VRF (buffering)
- Memory coherence
- Non-speculative interface



Ara System – Let's run some code!

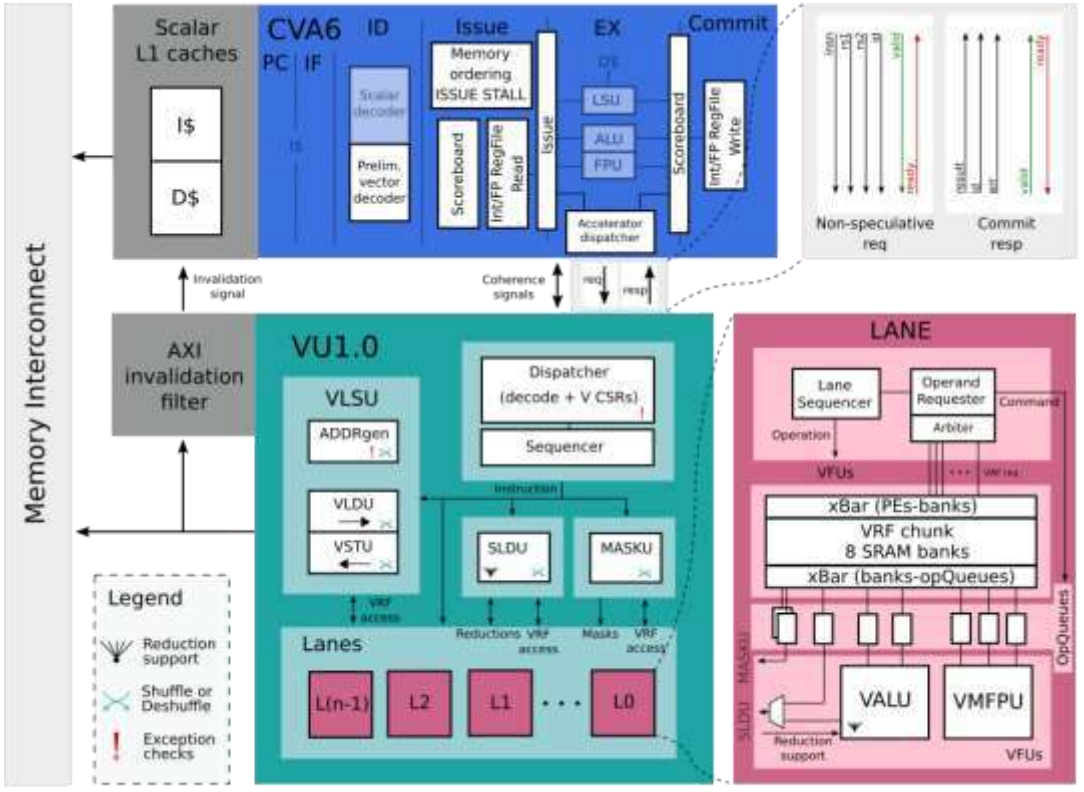
CVA6 and **Ara** work **in tandem!**

CVA6 fetches **scalar** and **vector** instructions from memory



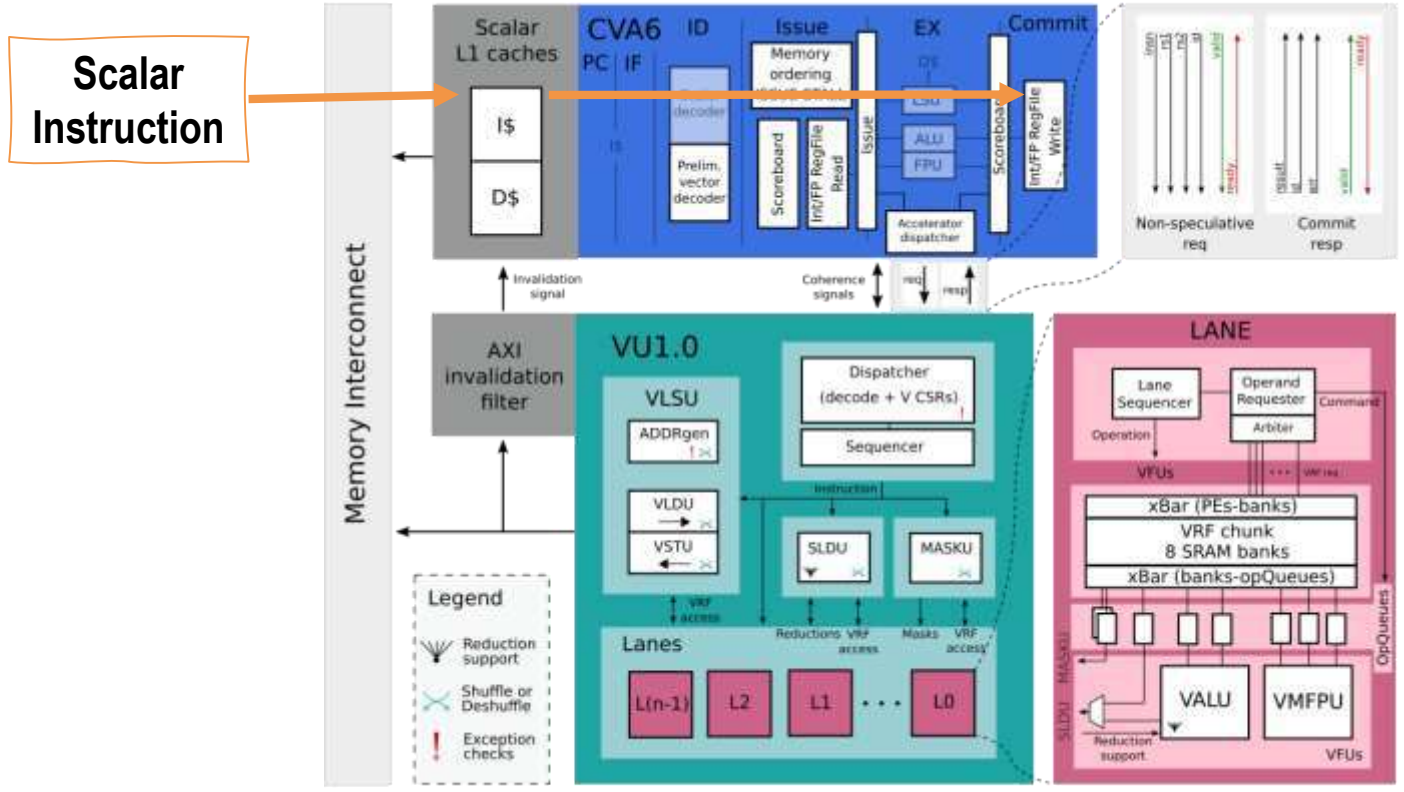
Ara System – Scalar instructions

Scalar Instruction



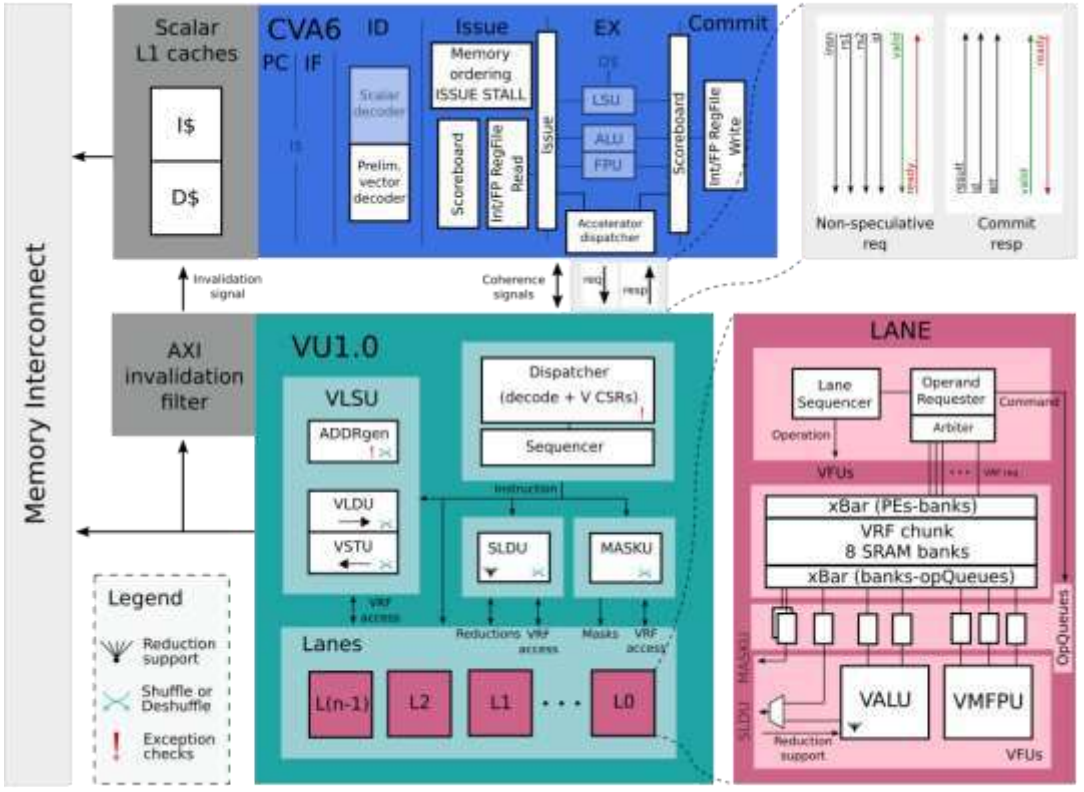
Ara System – Scalar instructions

Scalar execution – **CVA6**



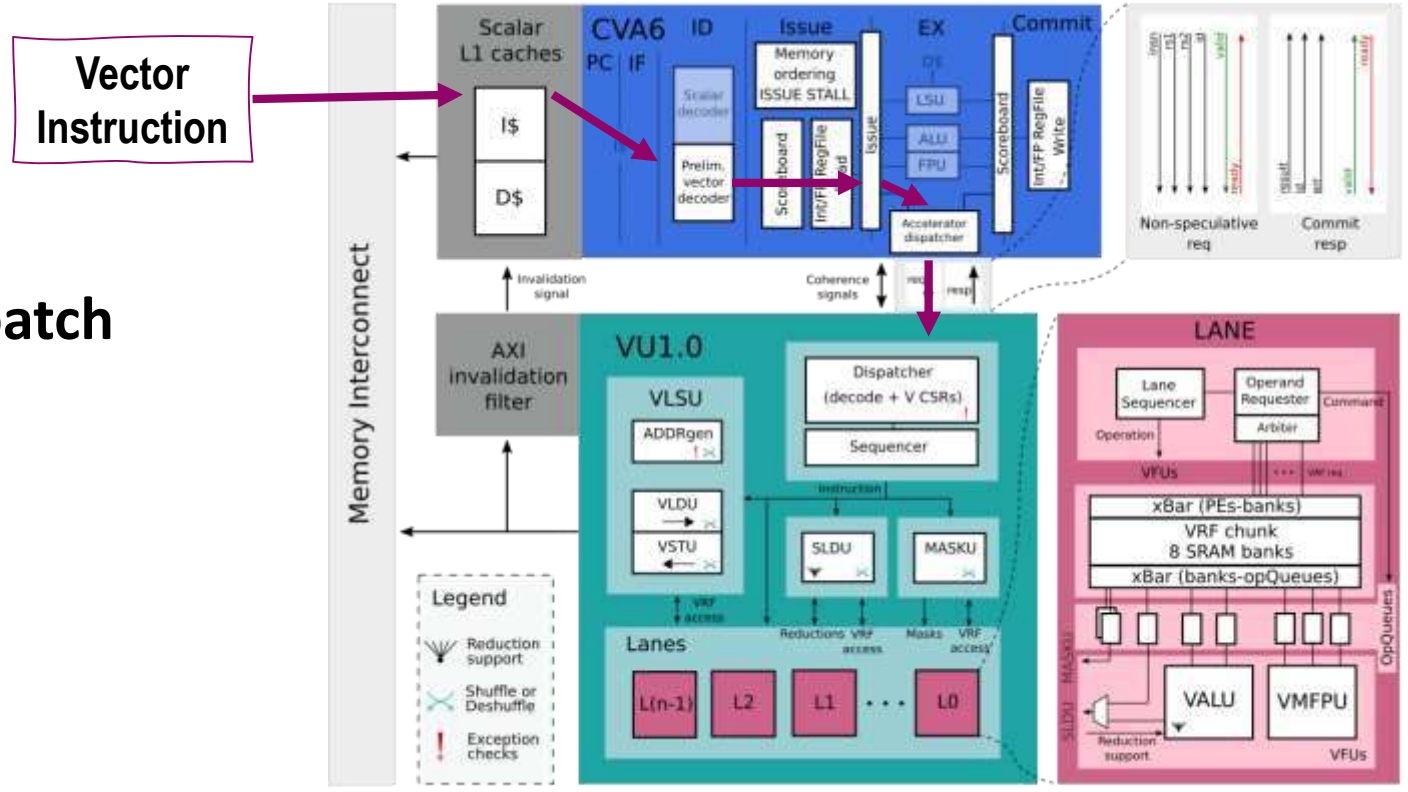
Ara System – Vector instructions

Vector Instruction



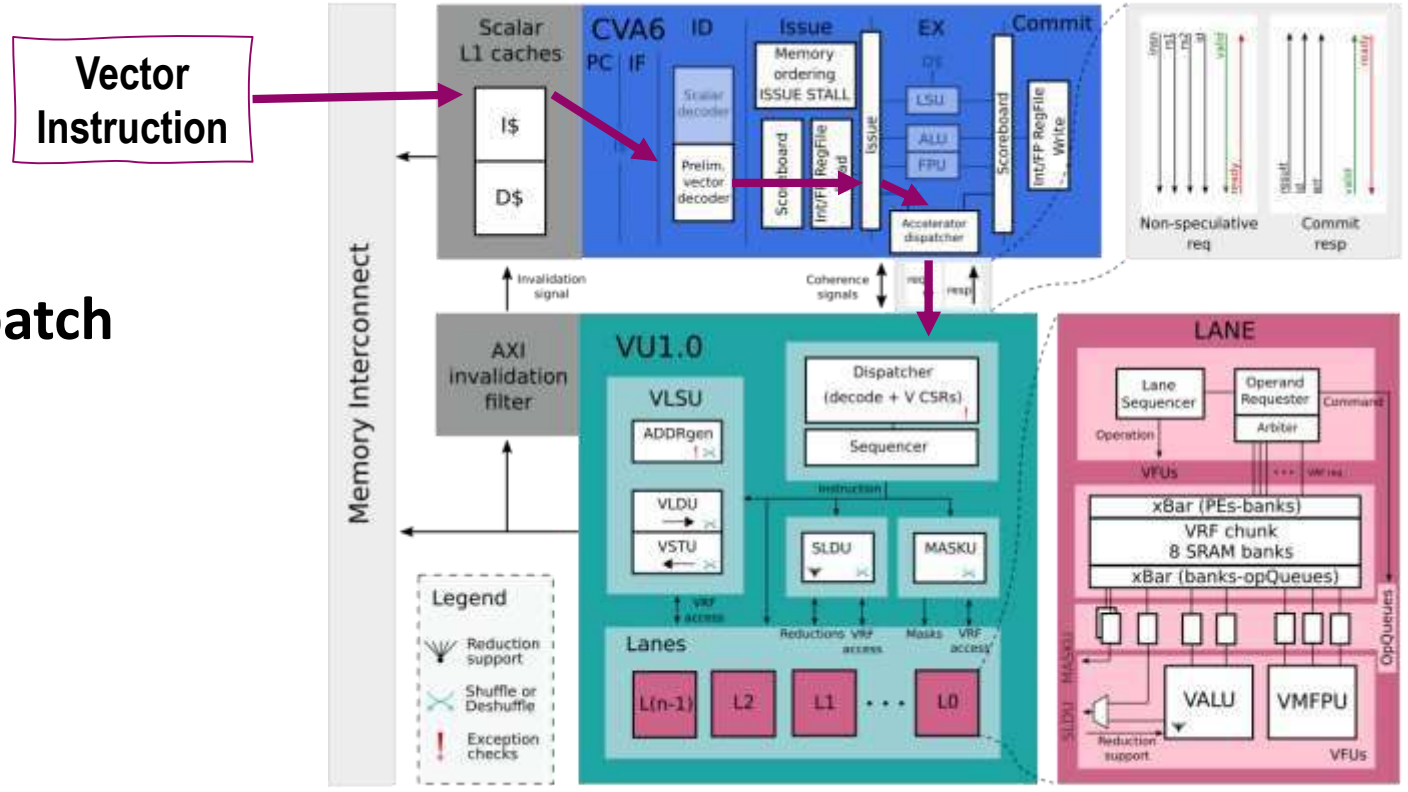
Ara System – Vector instructions

- Vector instruction **dispatch**



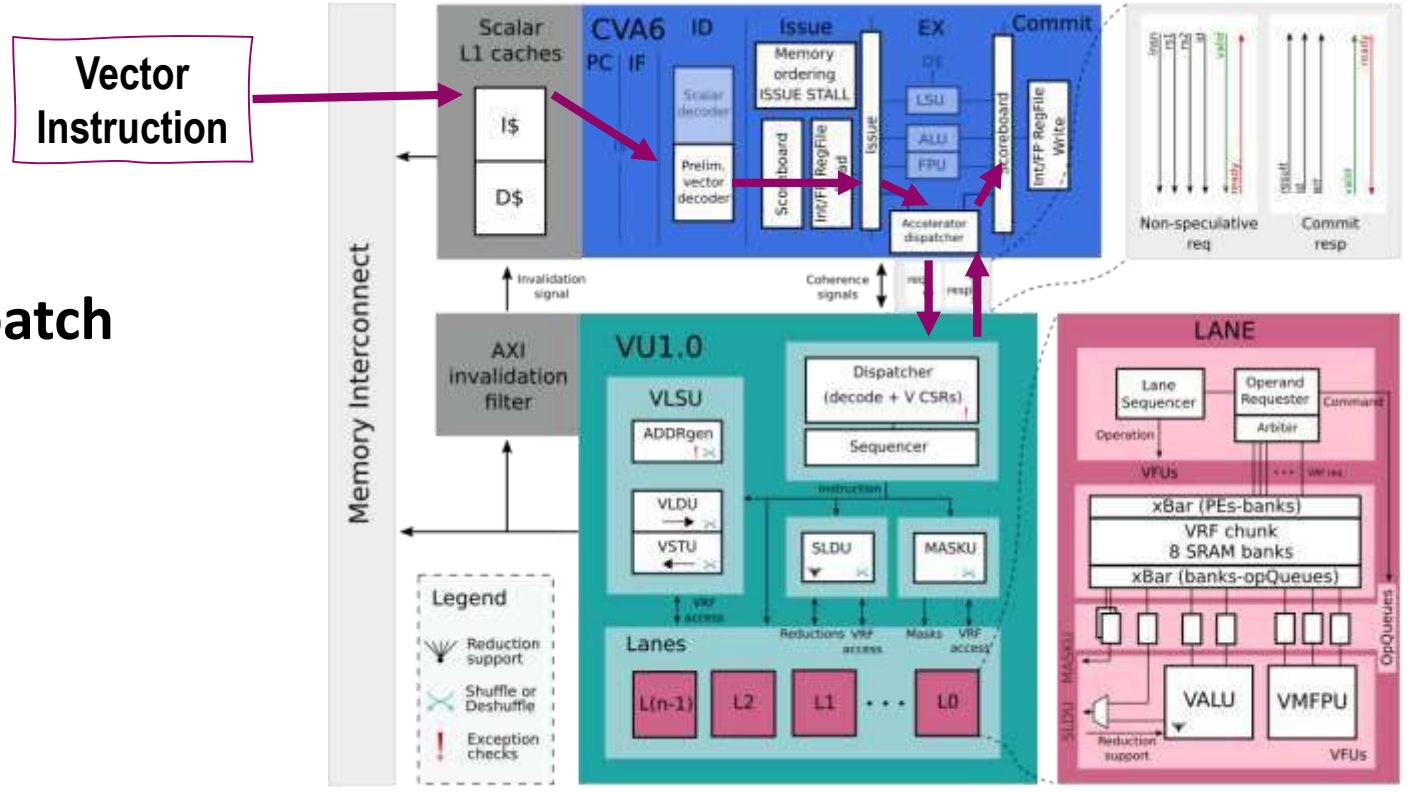
Ara System – Vector instructions

- Vector instruction **dispatch**
- **Top** of the **scoreboard**
- **Non speculative!**



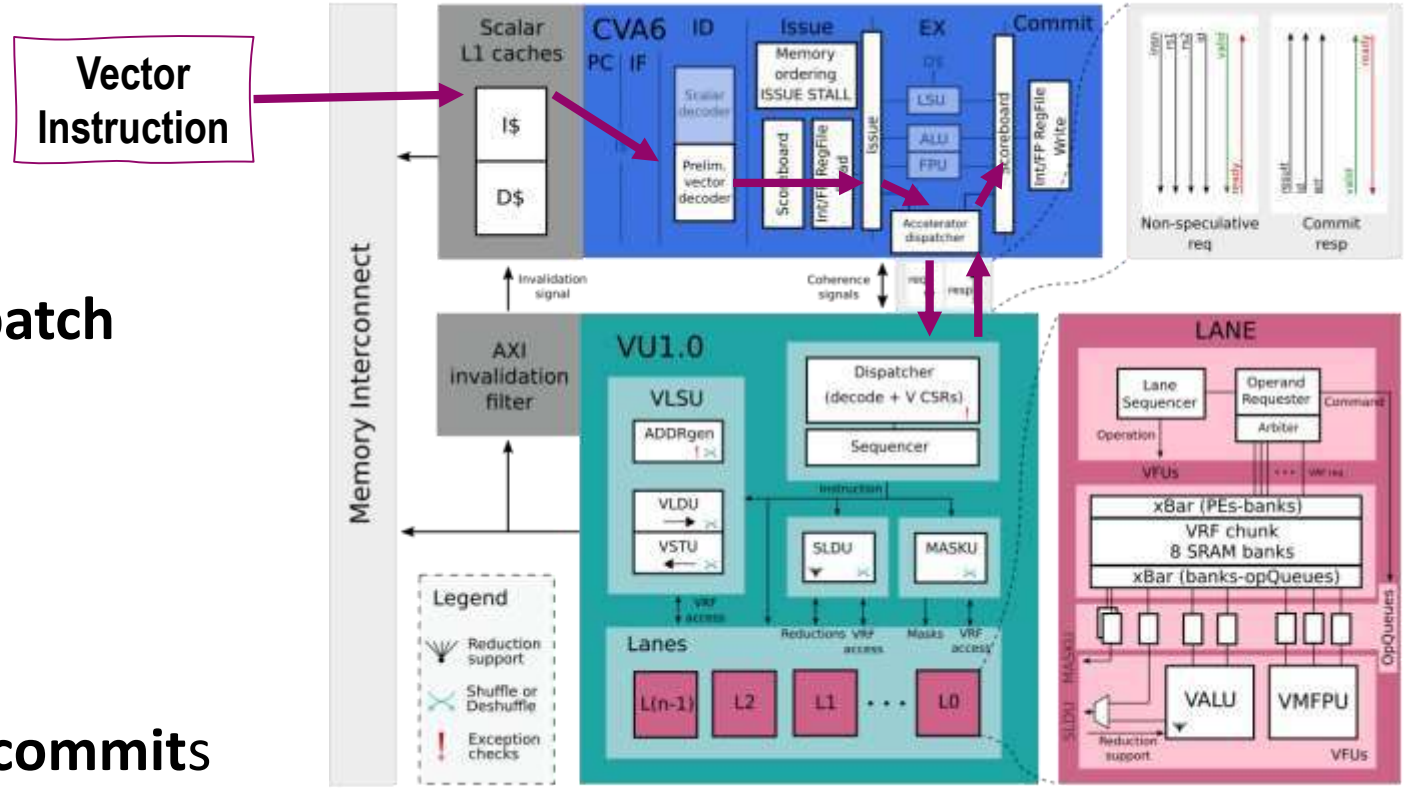
Ara System – Vector instructions

- Vector instruction **dispatch**
- **Top** of the **scoreboard**
- **Non speculative!**
- **Ara answers** back



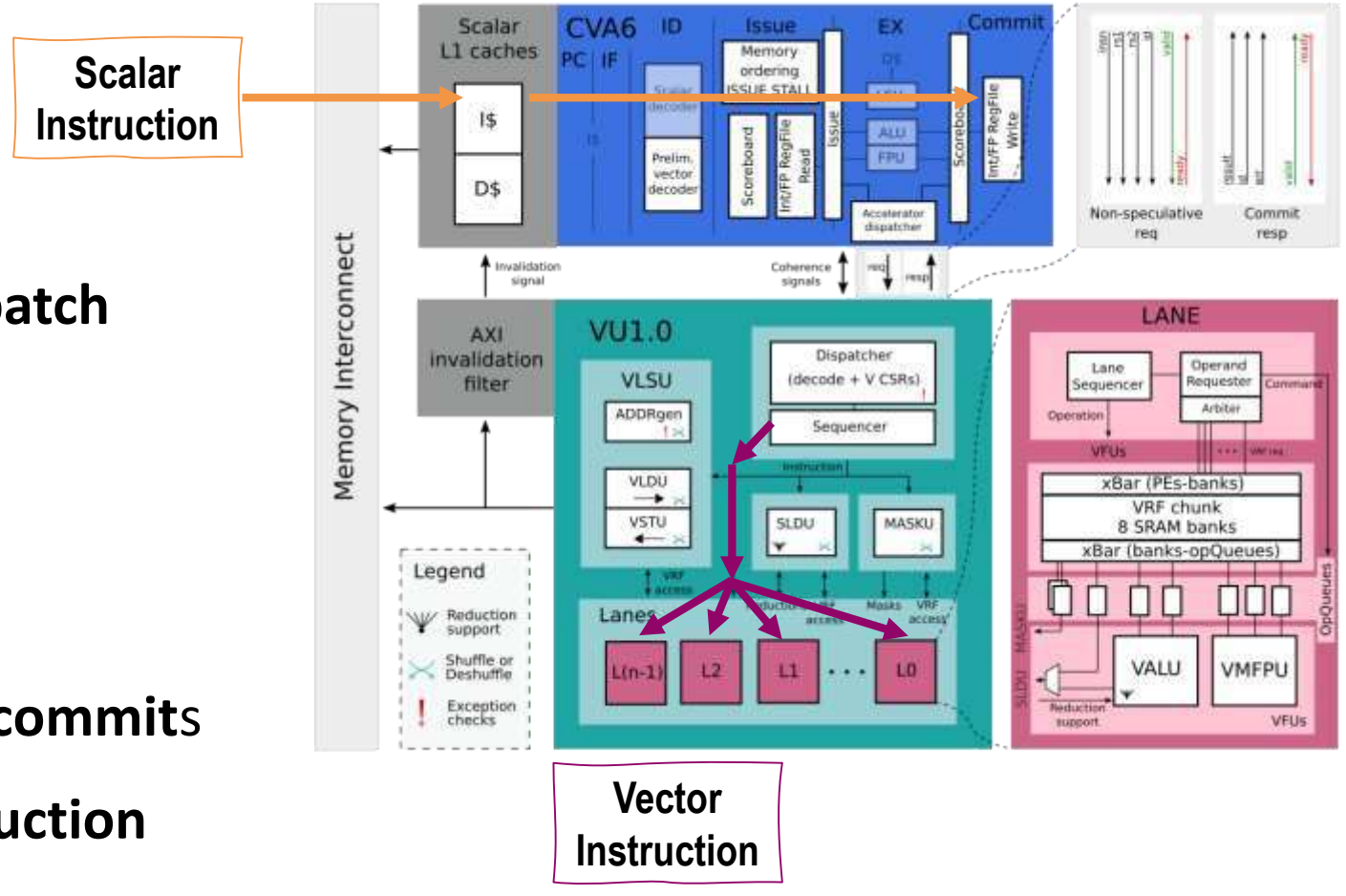
Ara System – Vector instructions

- Vector instruction **dispatch**
- **Top** of the **scoreboard**
- **Non speculative!**
- **Ara answers** back
- **No exceptions**? CVA6 **commit**s



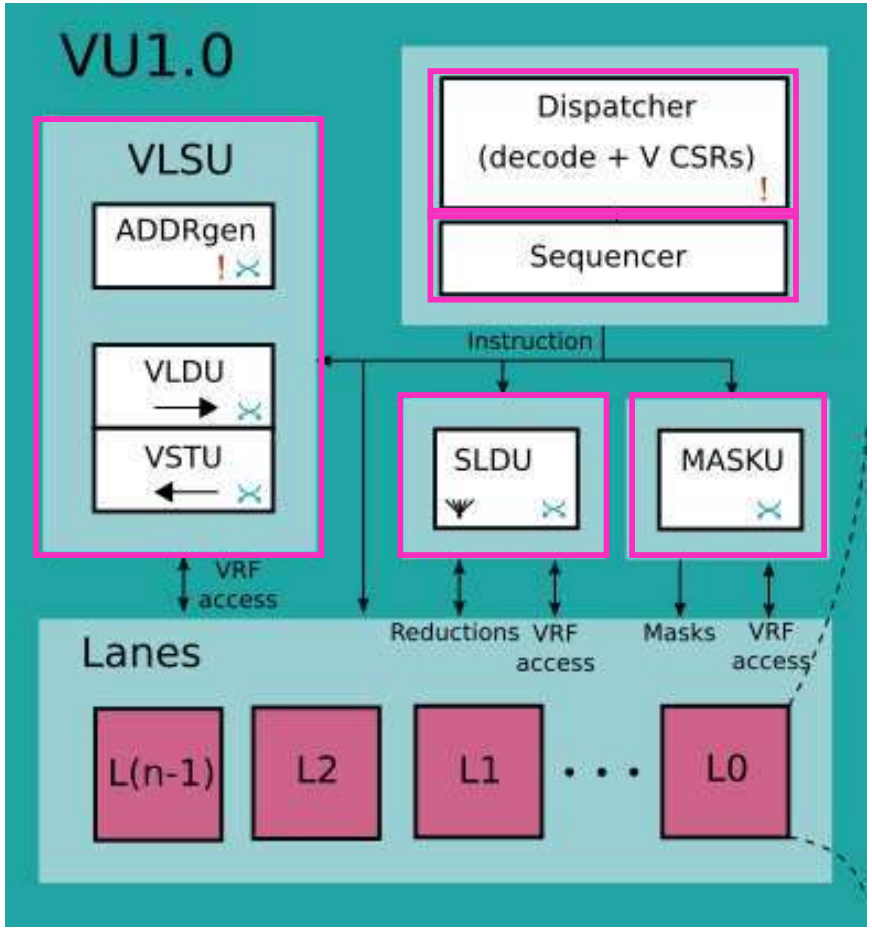
Ara System – Vector instructions

- Vector instruction **dispatch**
- **Top** of the **scoreboard**
- **Non speculative!**
- **Ara answers** back
- **No exceptions**? CVA6 **commit**s
- **Ara executes** the **instruction**



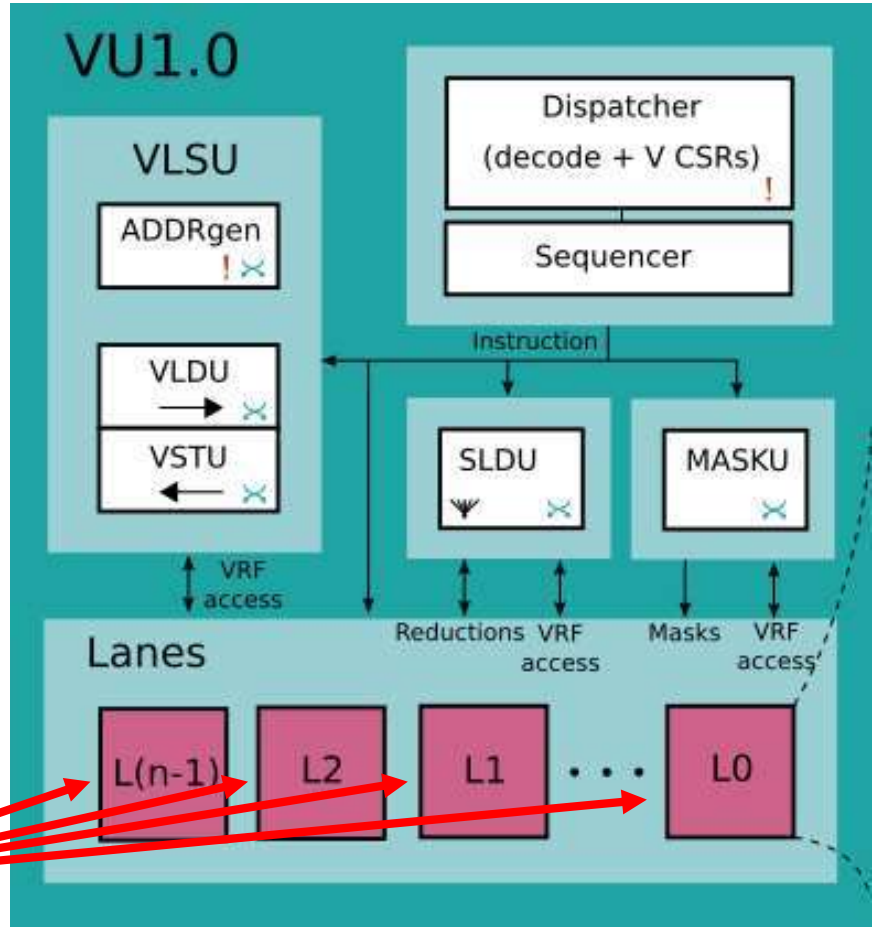
Ara

- Dispatcher (decode + CSRs)
- Sequencer (issue + hazards)
- Private VLSU (vload + vstore)
- Slide Unit (permutations)
- Mask Unit (predication)



Ara

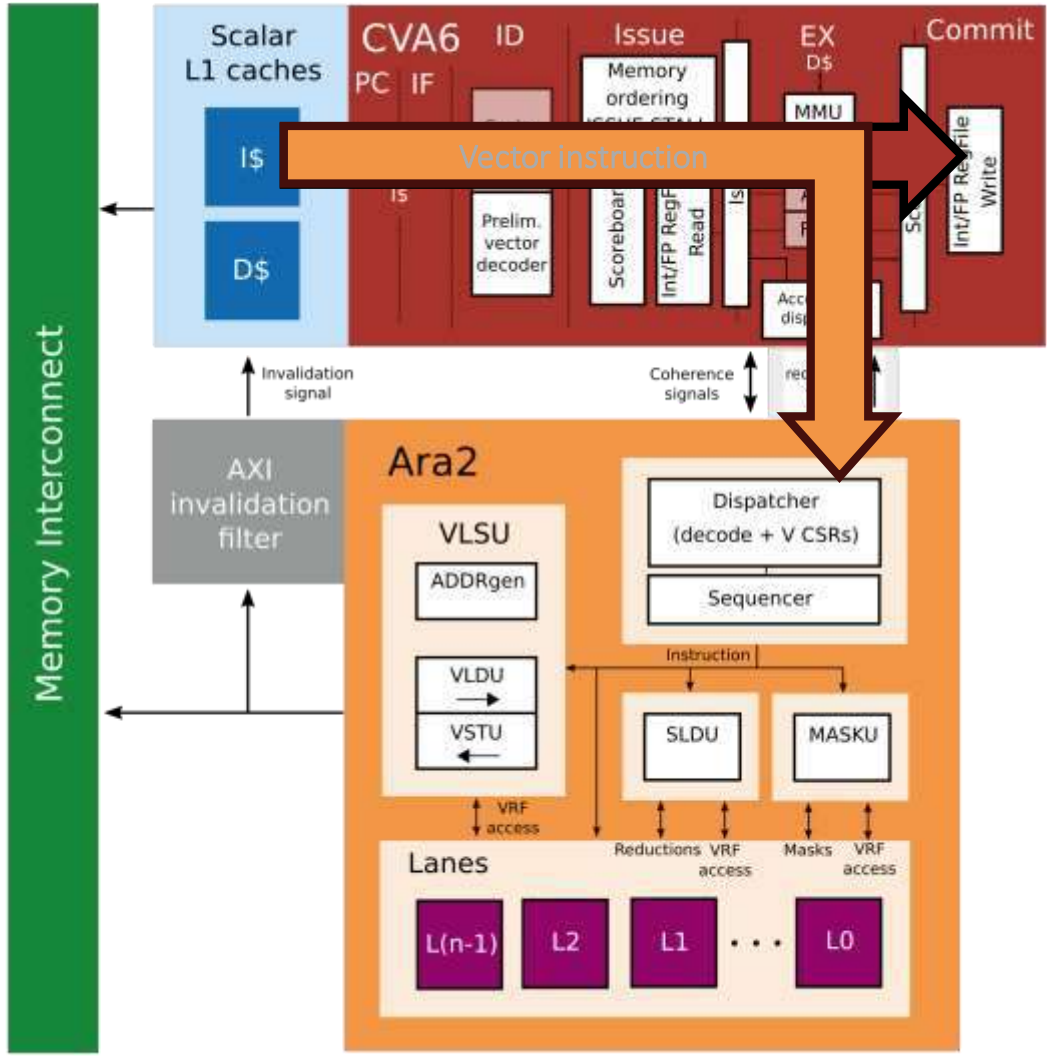
- Dispatcher (decode + CSRs)
- Sequencer (issue + hazards)
- Private VLSU (vload + vstore)
- Slide Unit (permutations)
- Mask Unit (predication)
- Lanes (computation + VRF)



Parametric number of lanes!
From 2 to 16

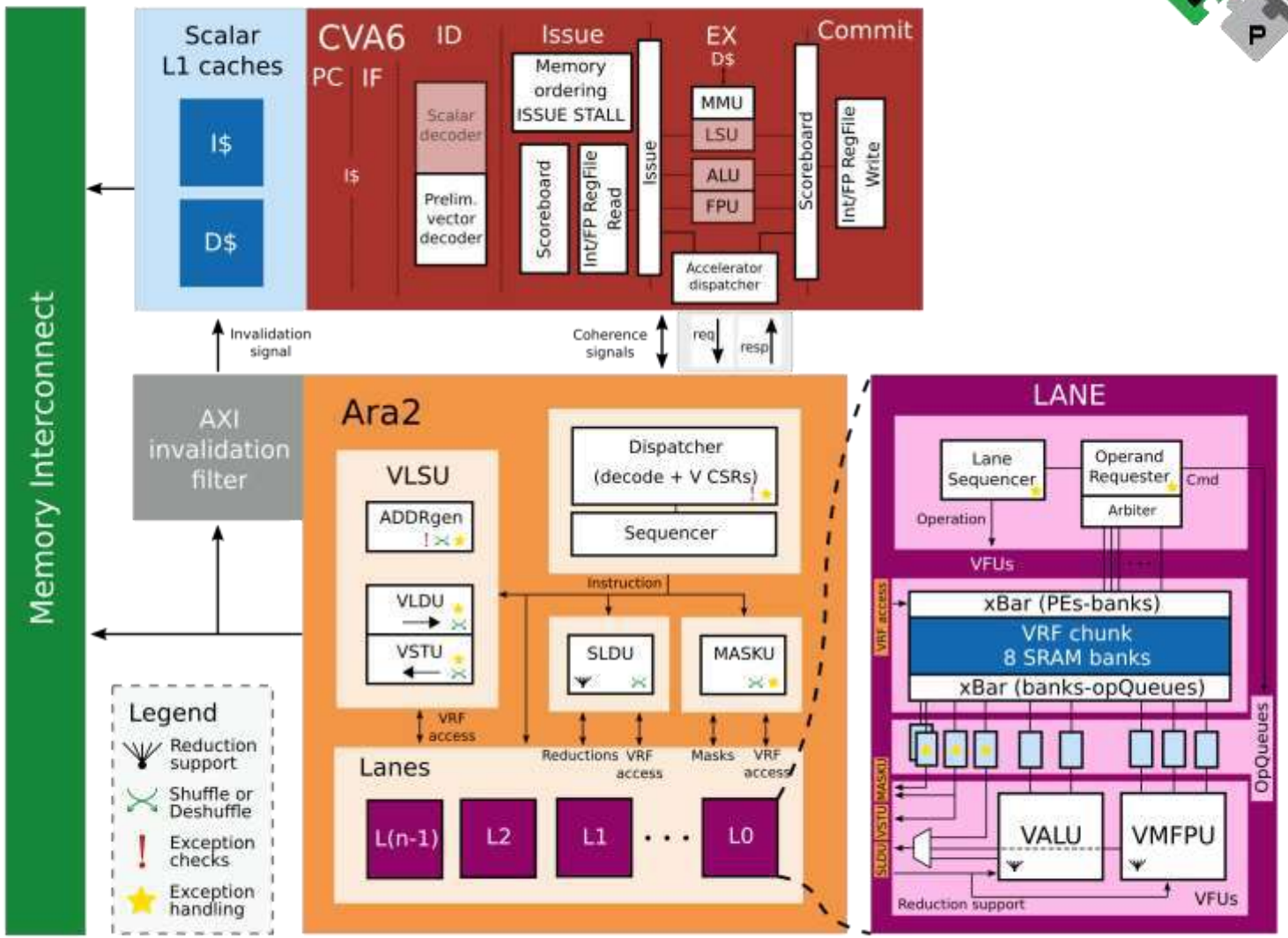
Ara2 architecture

- CVA6 + **Ara2**: RV64GC**V**
- **CVA6**
  - Access to I-MEM
  - Non-speculative V Dispatch
  - Fire-and-forget V instructions
  - **INSN** and **MMU** interfaces
- **Ara2**
  - Check for exceptions and ack.
  - Execute the V instruction

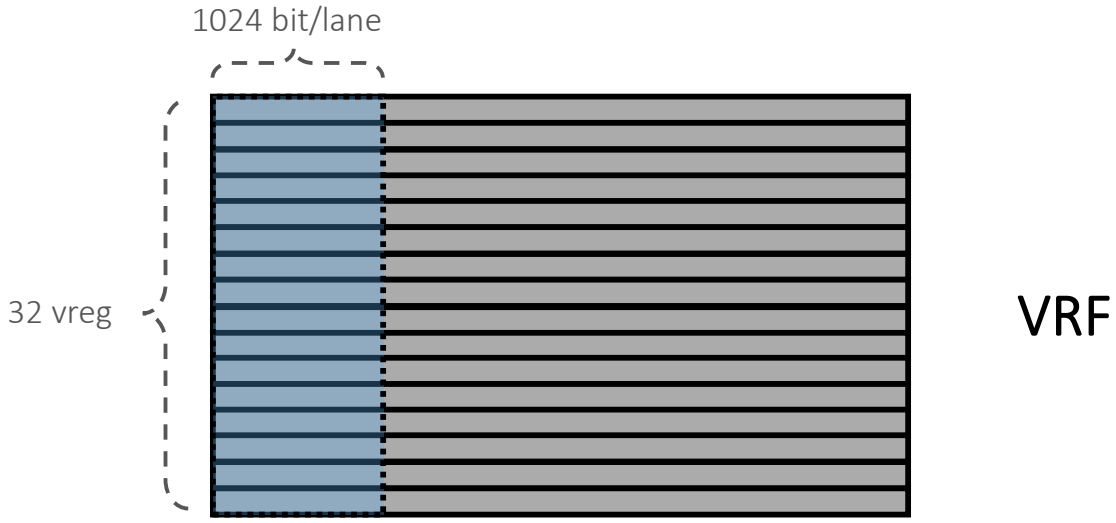


Ara2 architecture

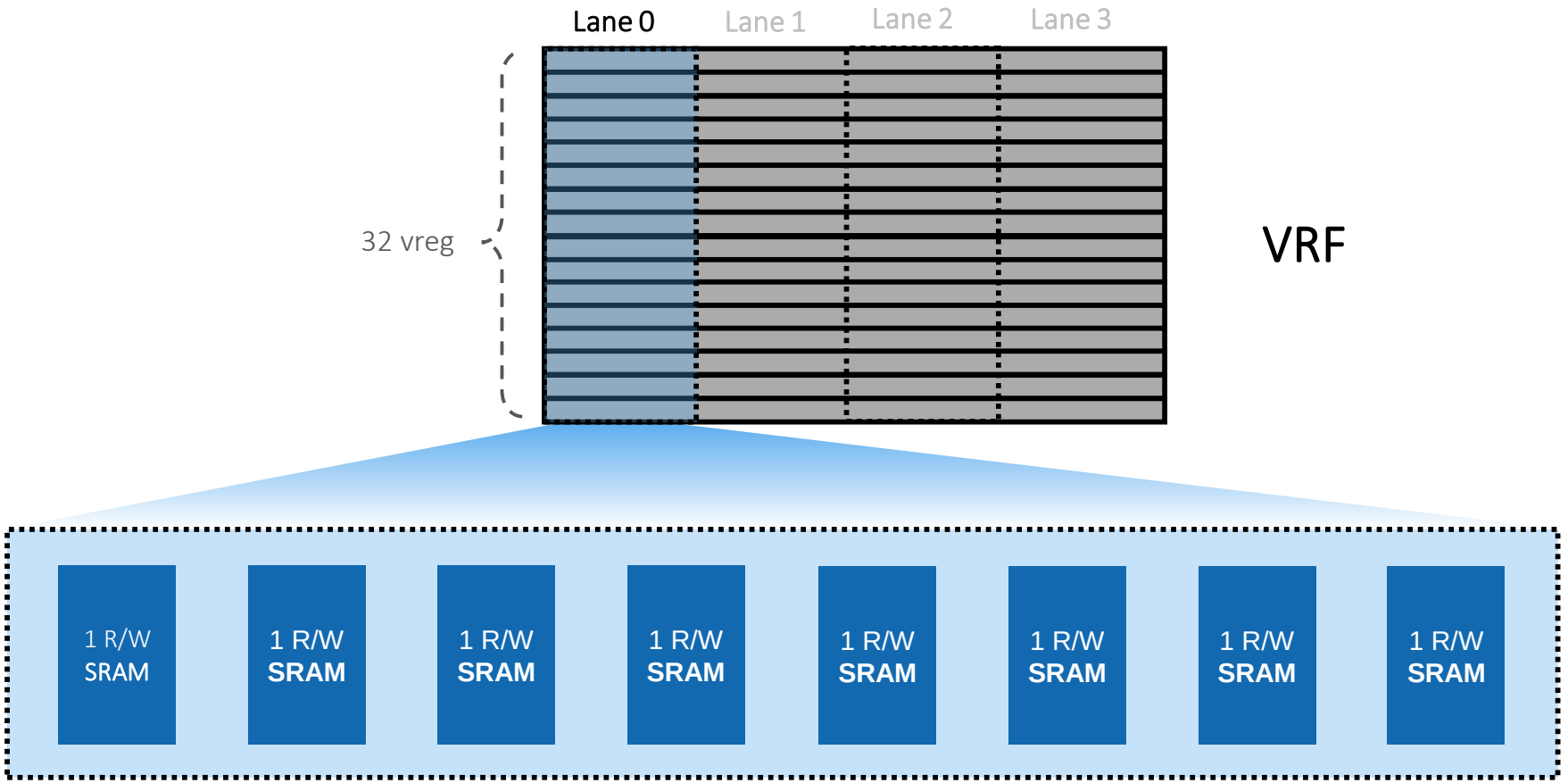
- CVA6 + **Ara2**: RV64GC**V**
- **CVA6**
  - Access to I-MEM
  - Non-speculative V Dispatch
  - Fire-and-forget V instructions
  - INSN and MMU interfaces
- **Ara2**
  - Check for exceptions and ack.
  - Execute the V instruction
  - Lane: computational heart
  - One FPU per lane
  - One VRF chunk per lane



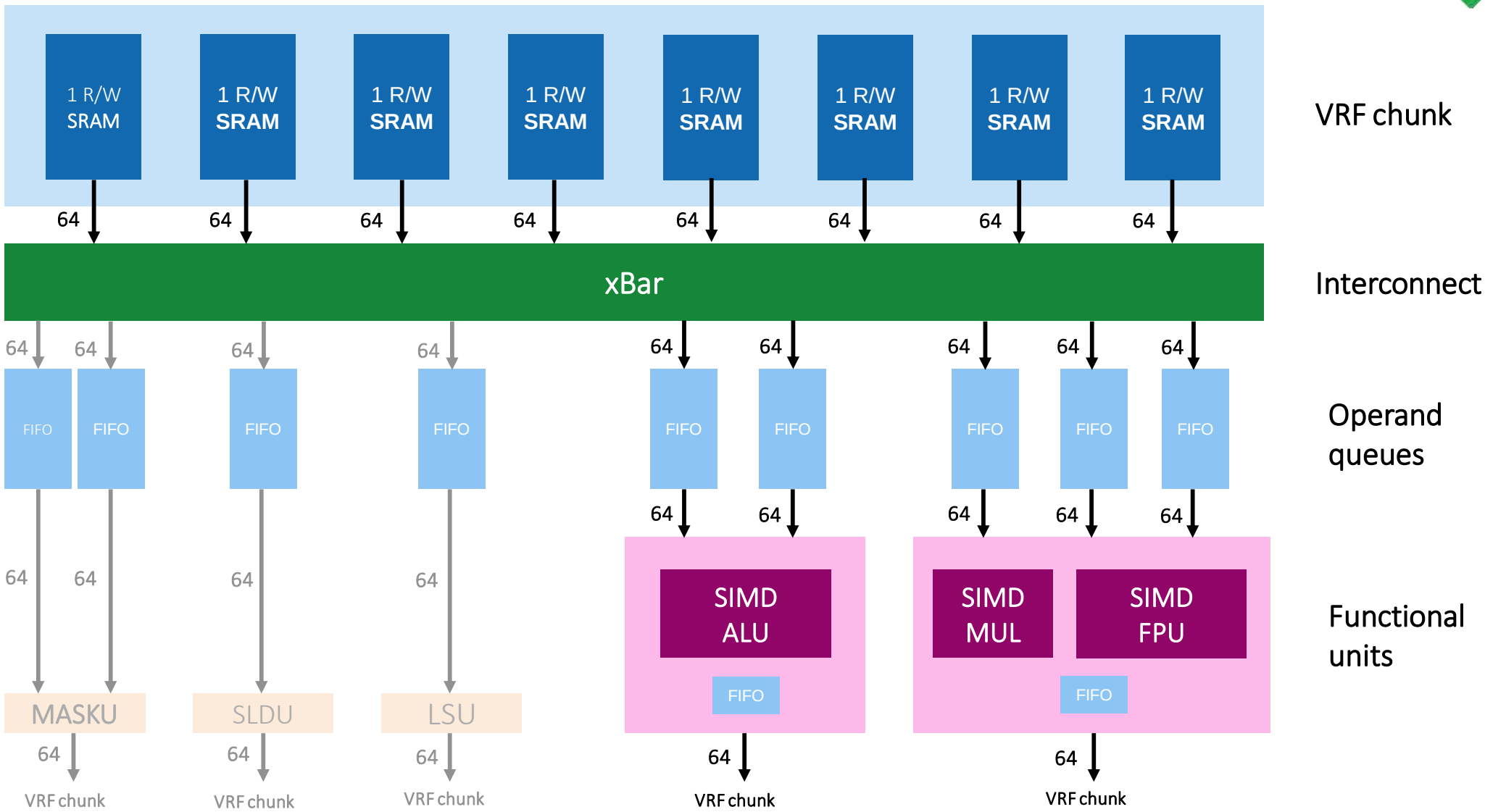
The vector register file (VRF)



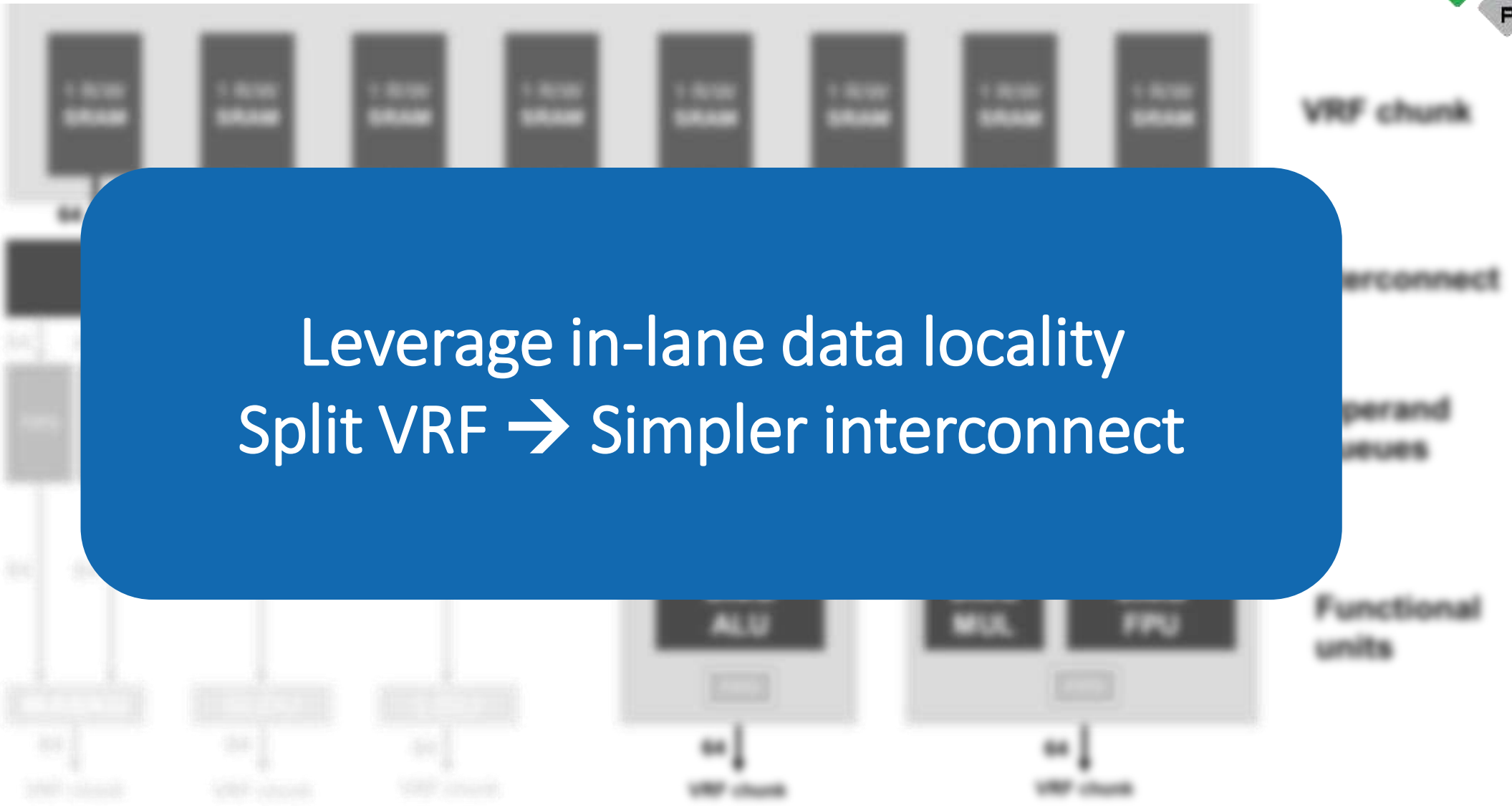
The vector register file (VRF)



Into the LANE

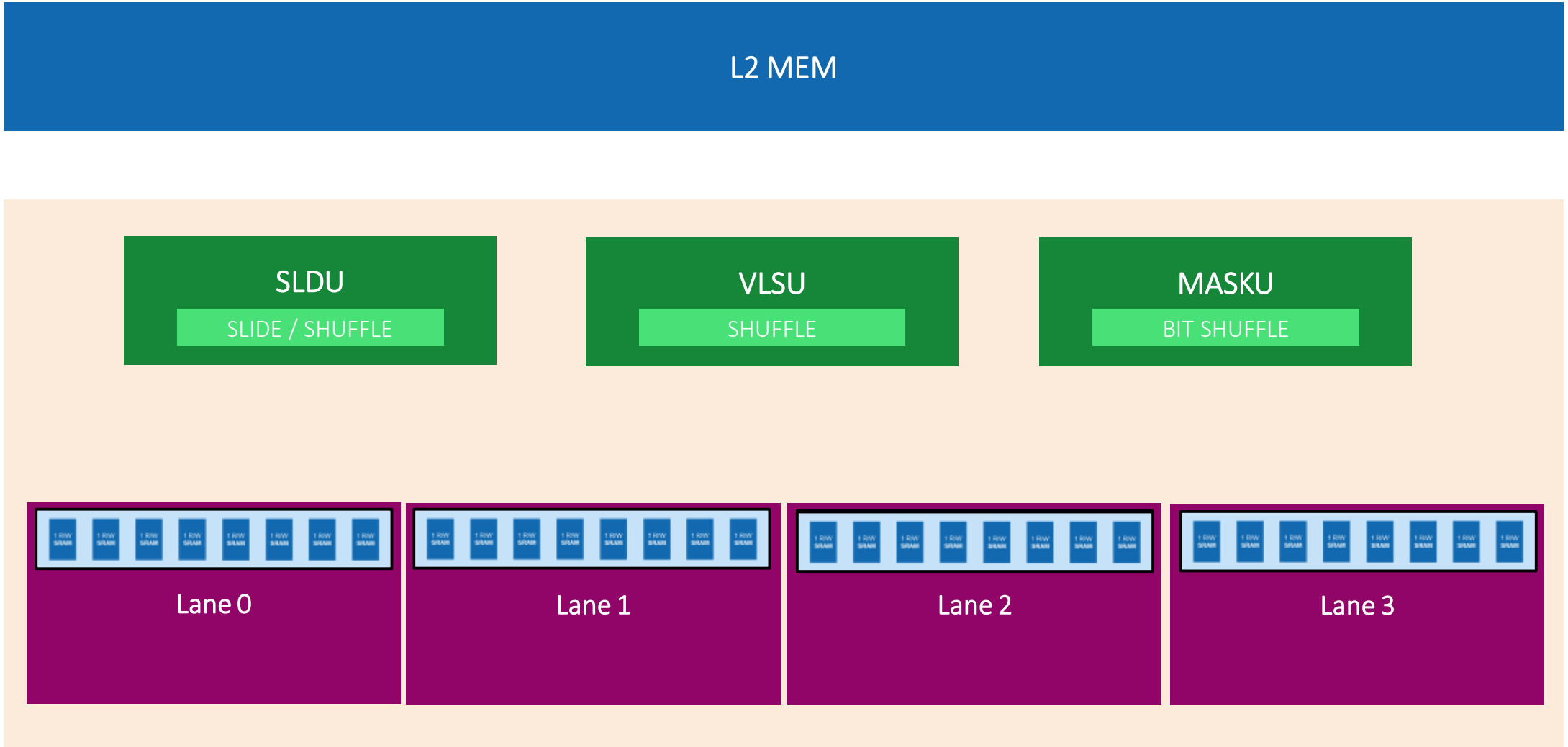


Into the LANE

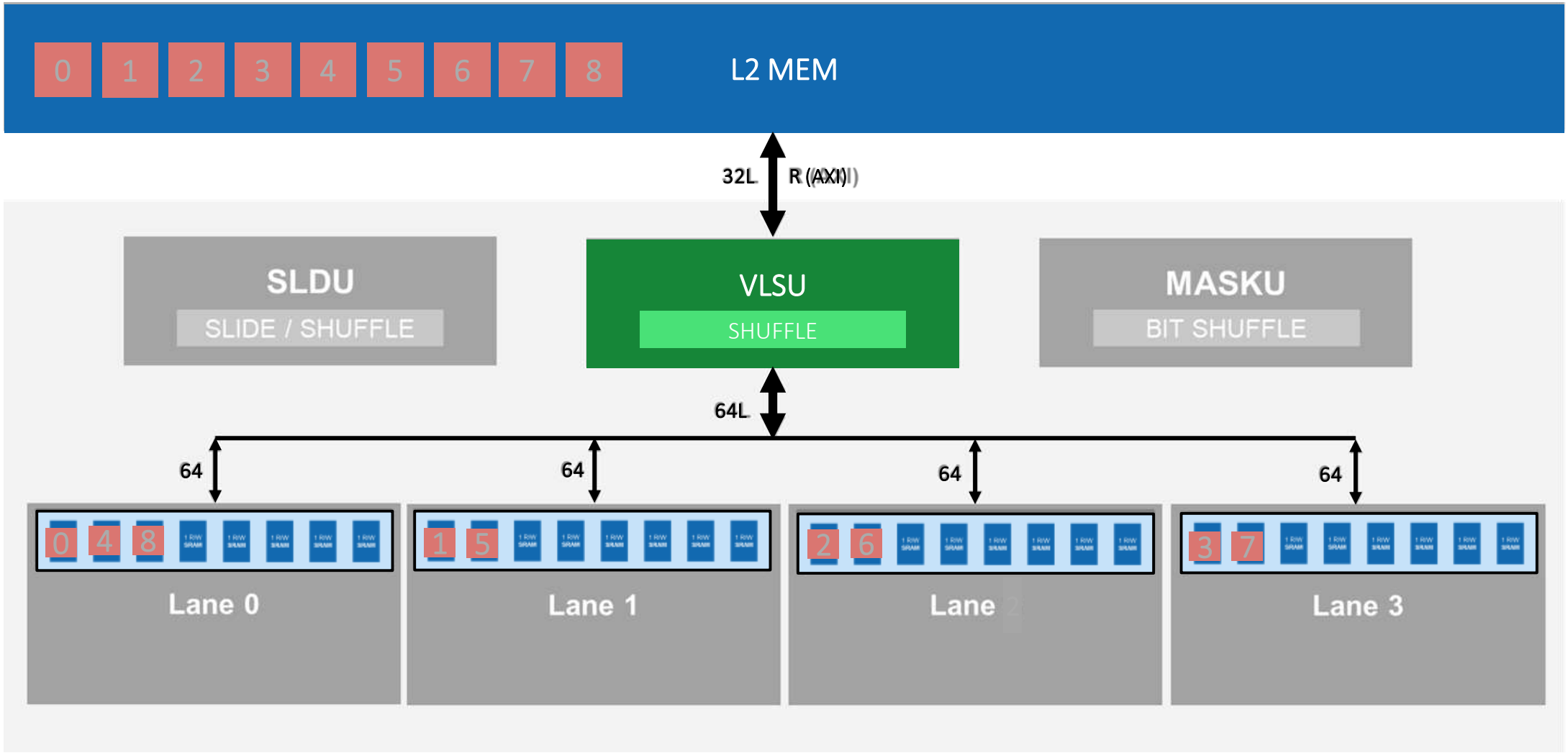


Leverage in-lane data locality
Split VRF → Simpler interconnect

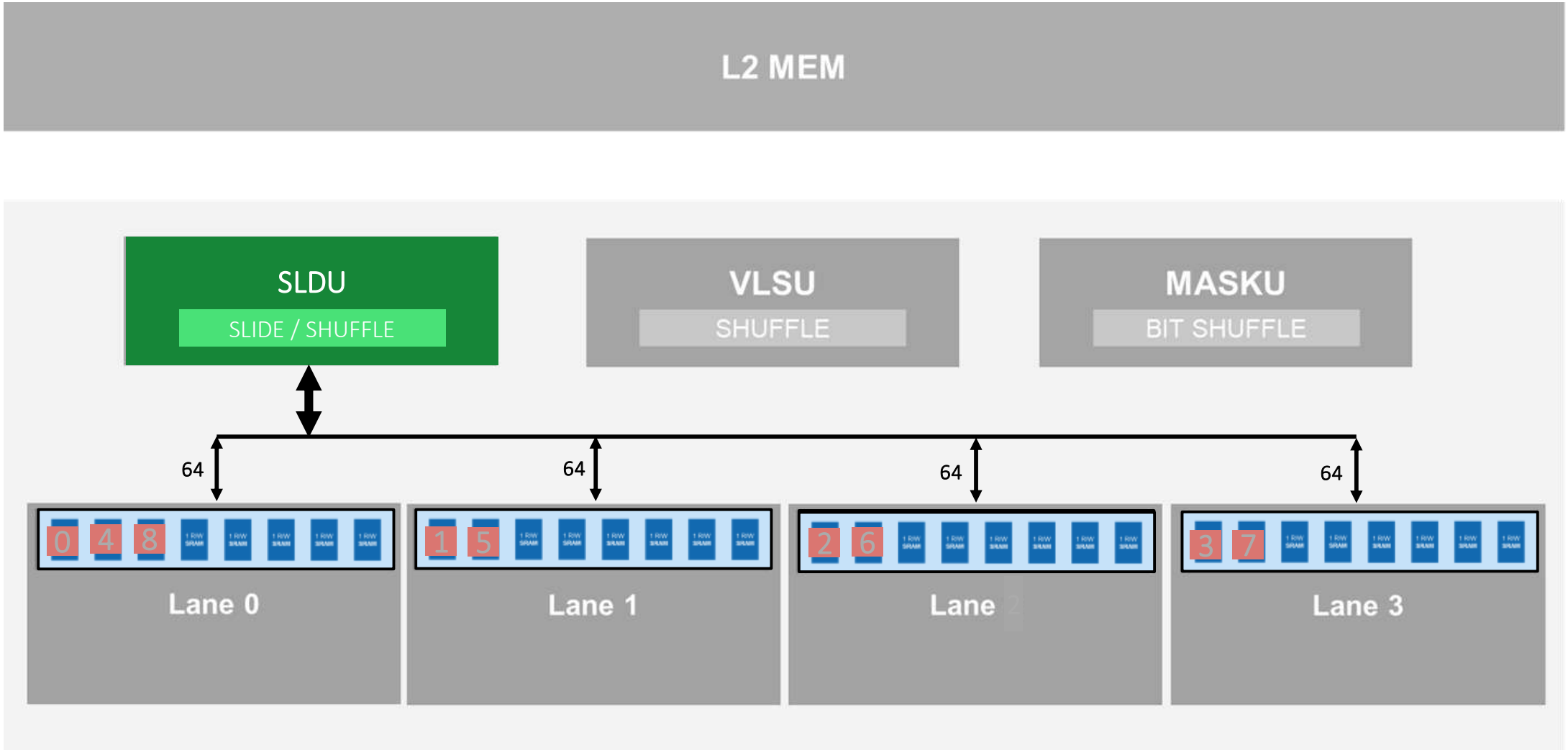
How to move data outside of the lanes?



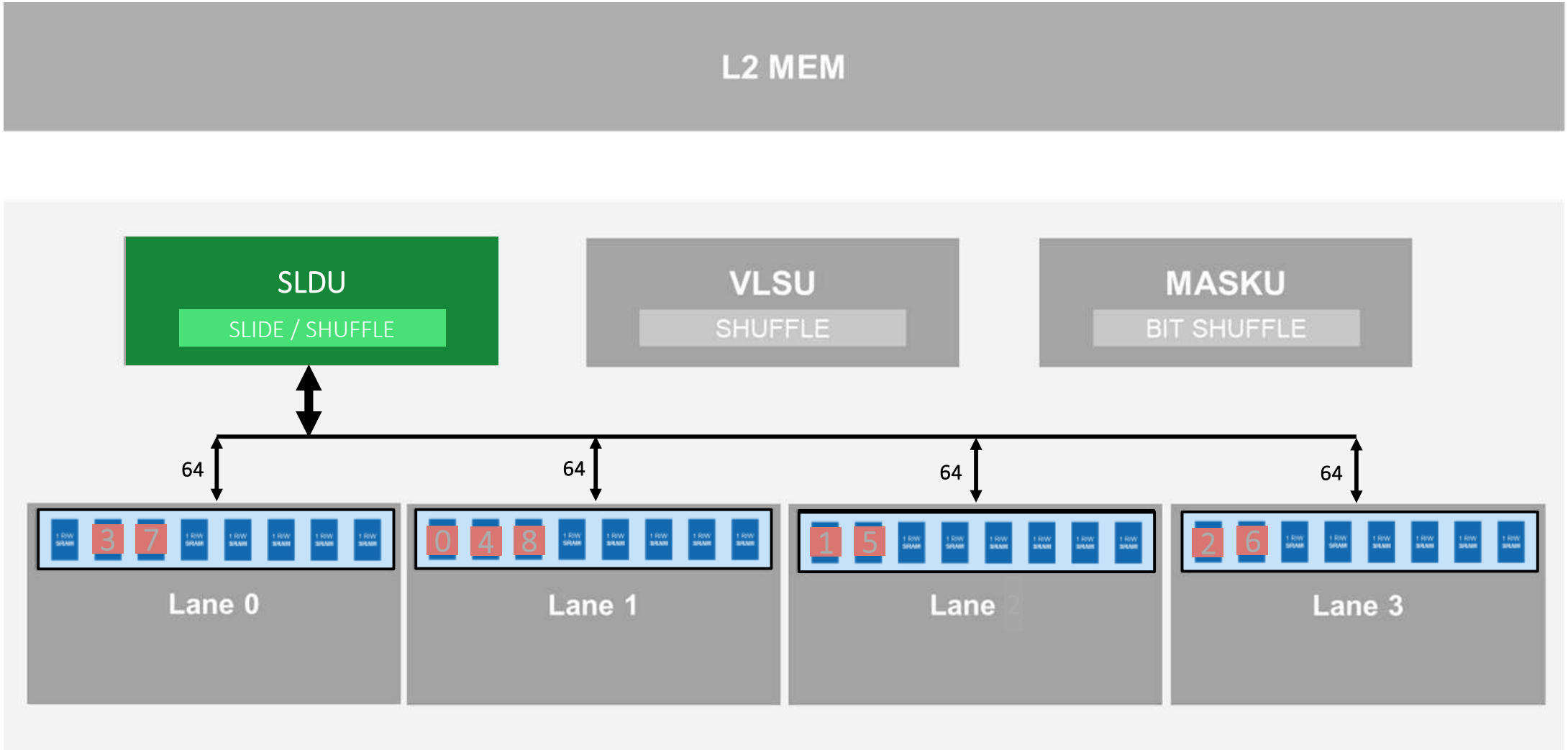
VLSU – Transfer elements between MEM and VRF



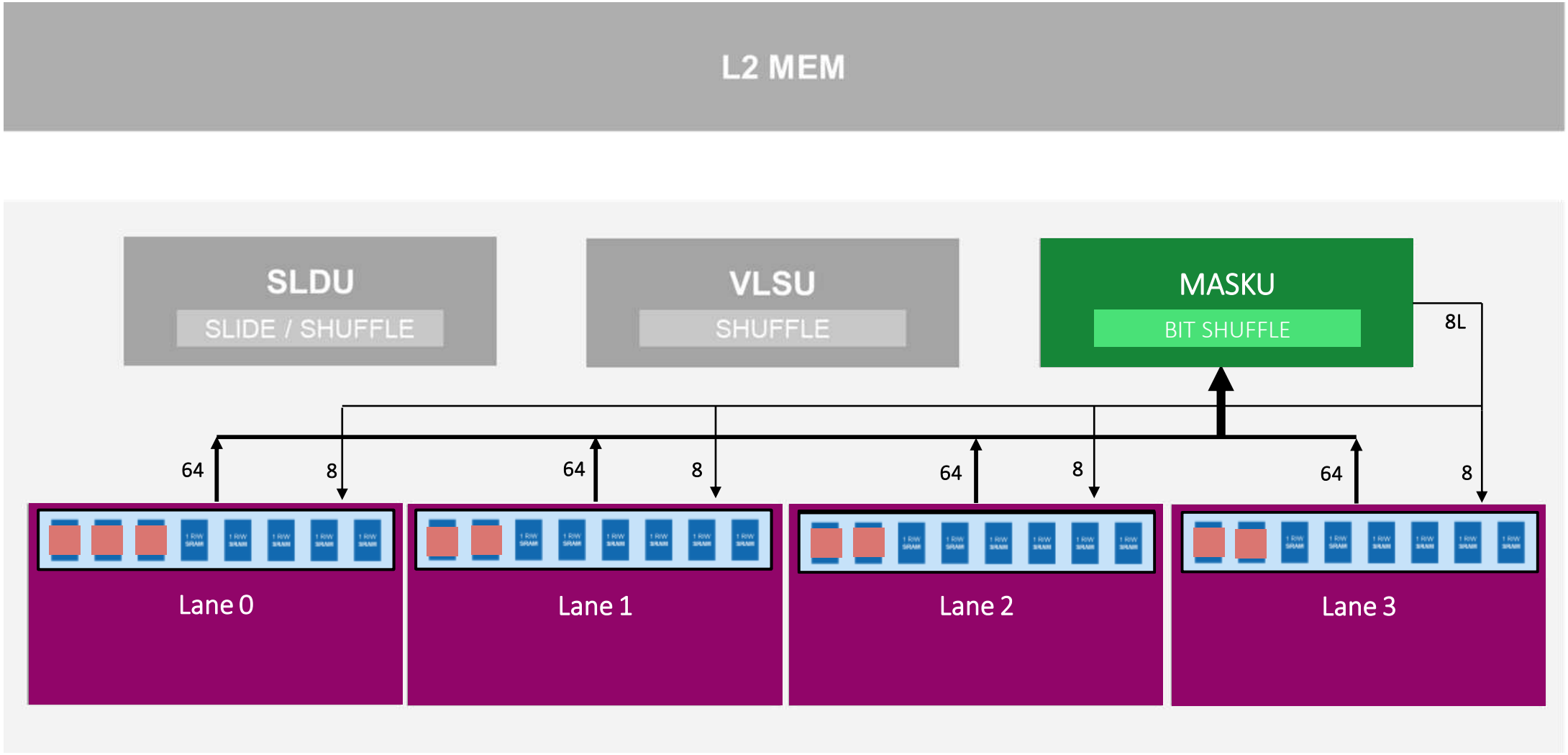
SLDU – Move vector elements among lanes



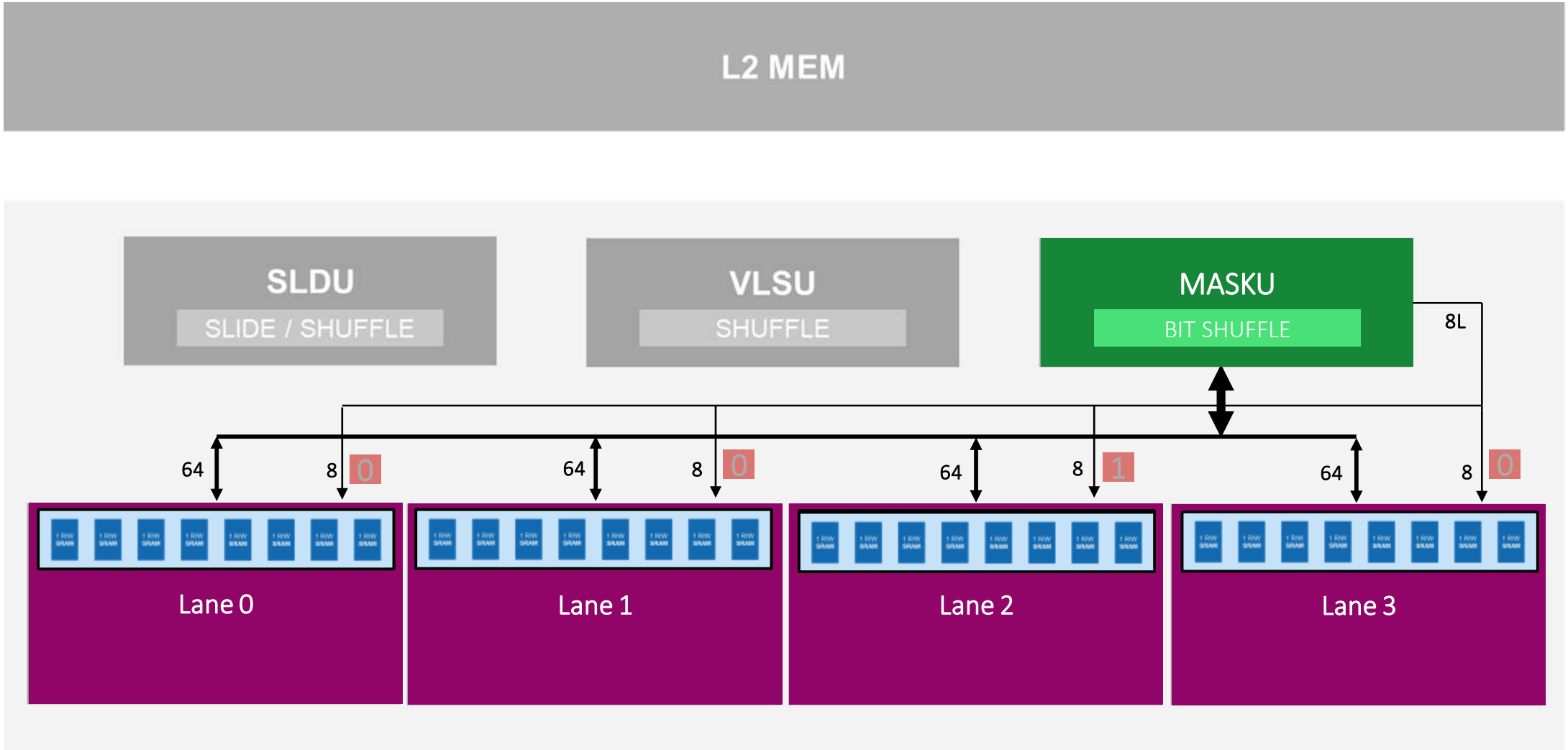
SLDU – Move vector elements among lanes



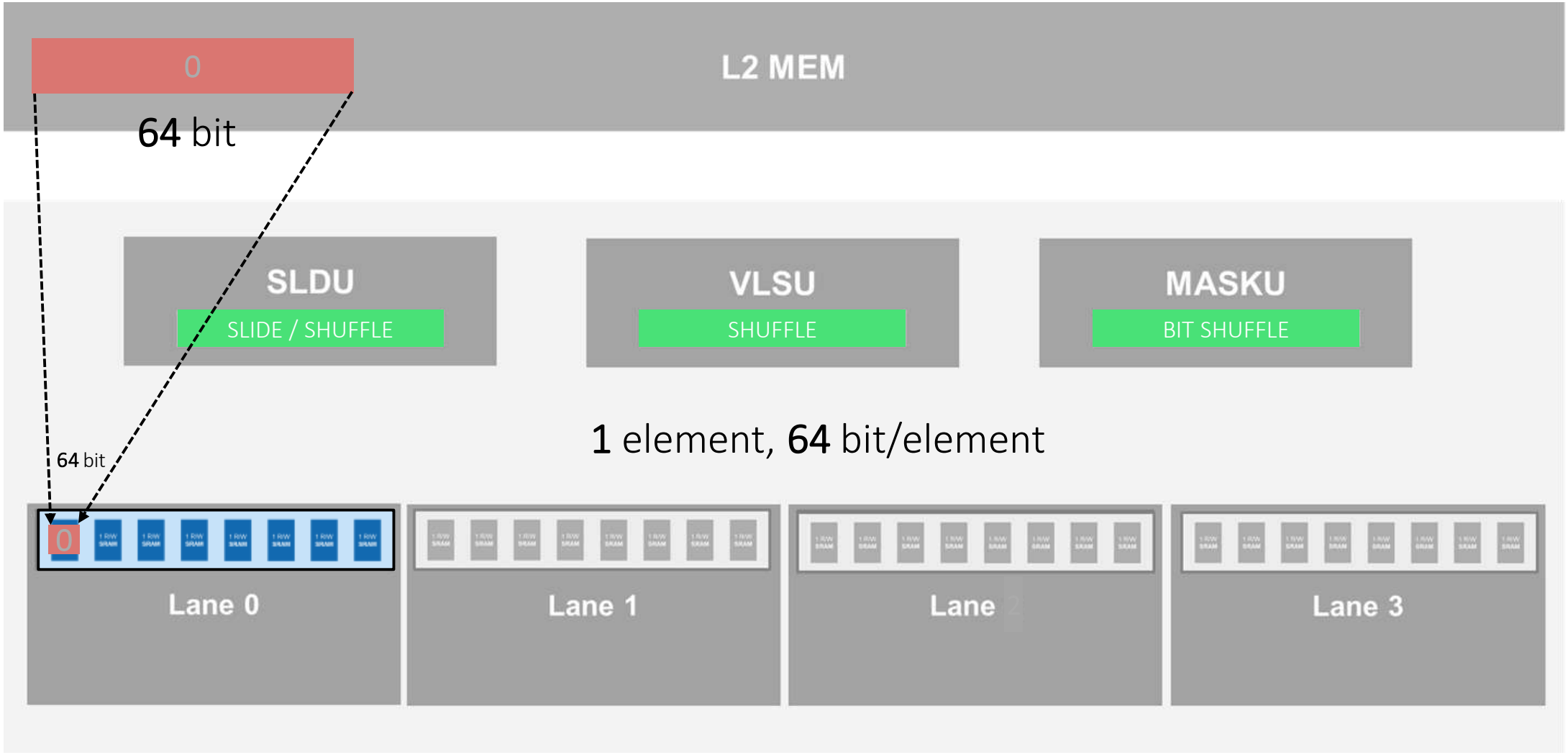
MASKU – Support bit-level predicated execution



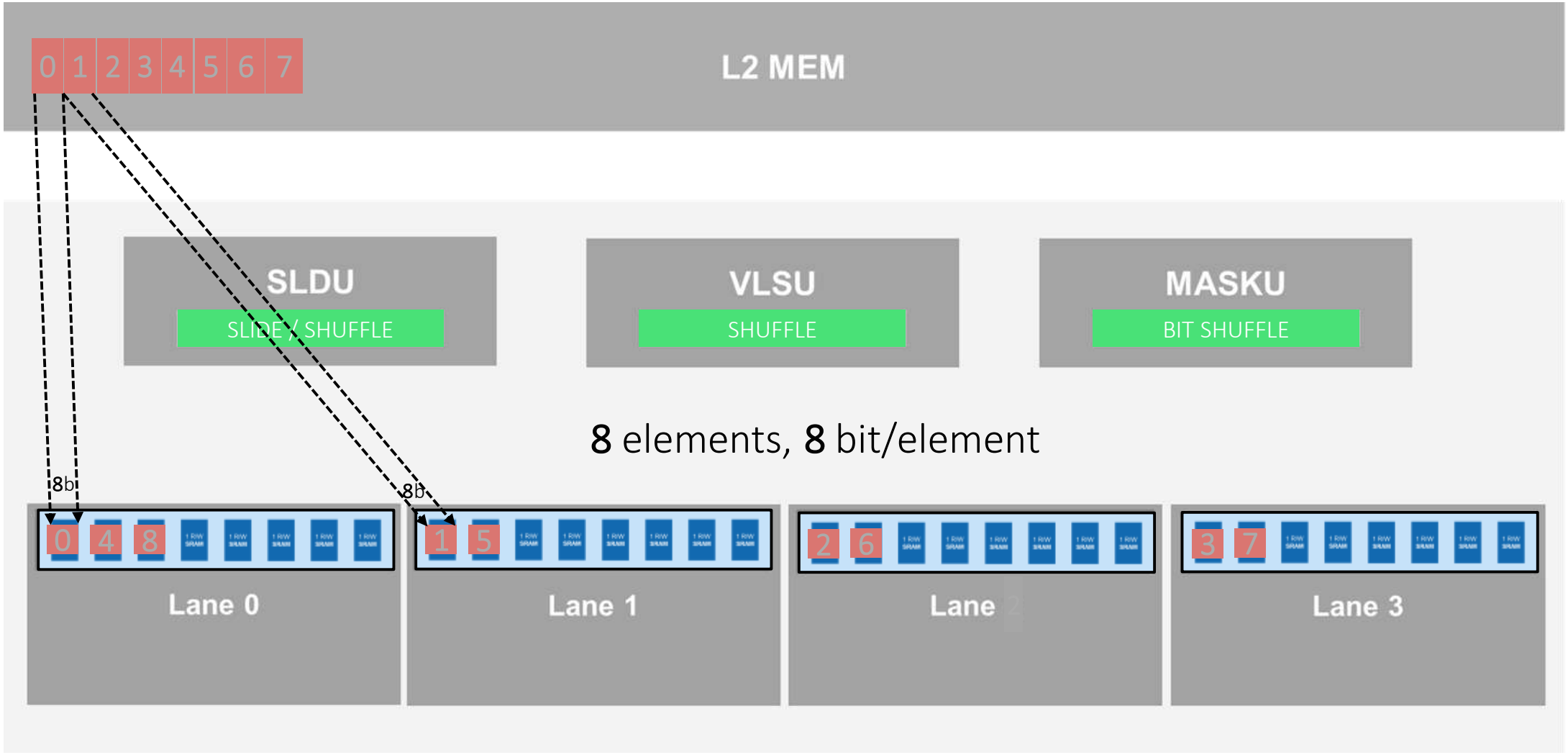
MASKU – Support bit-level predicated execution



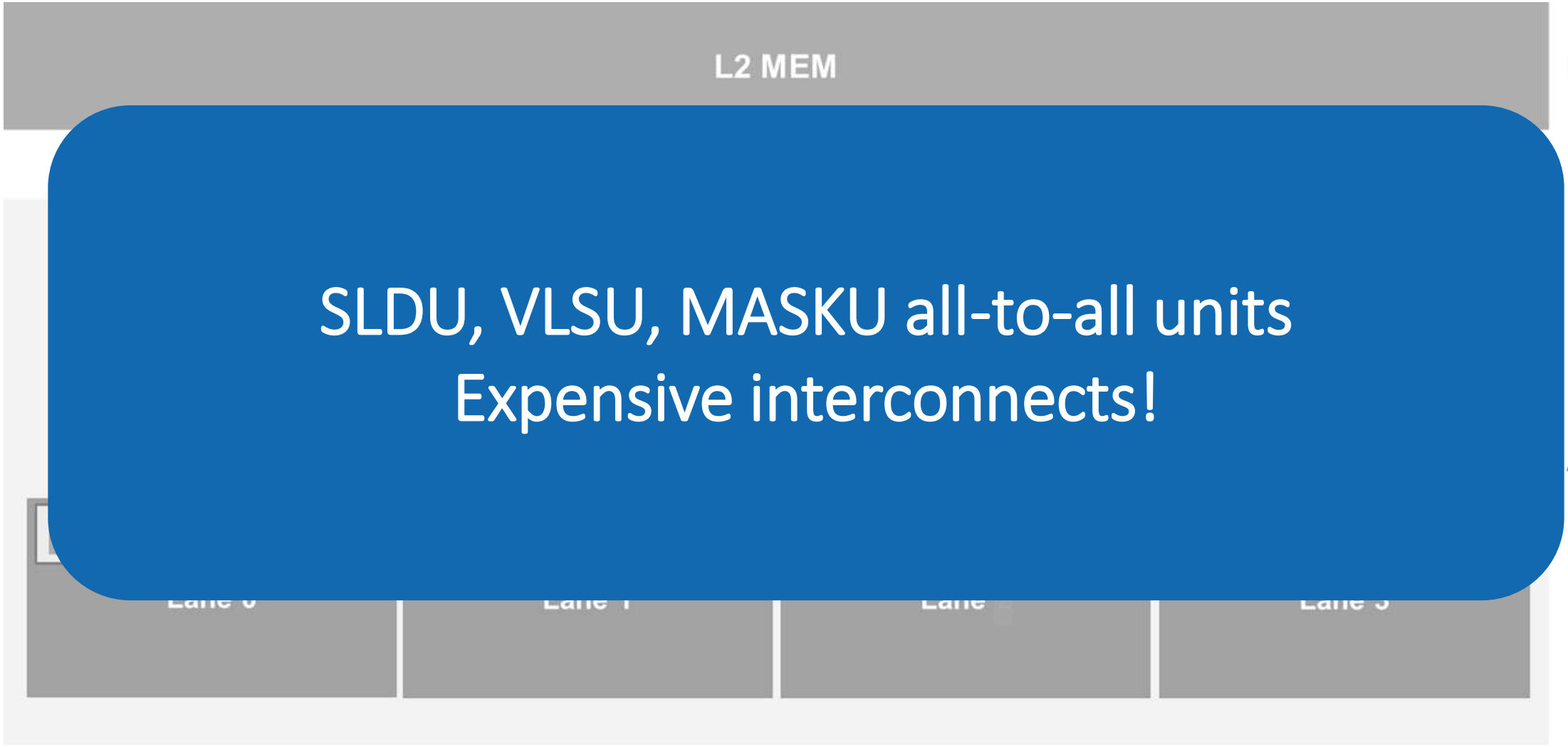
The vector length and data widths can change at runtime!



The vector length and data widths can change at runtime!



All-to-all connected units – Takeaway



SLDU, VLSU, MASKU all-to-all units
Expensive interconnects!

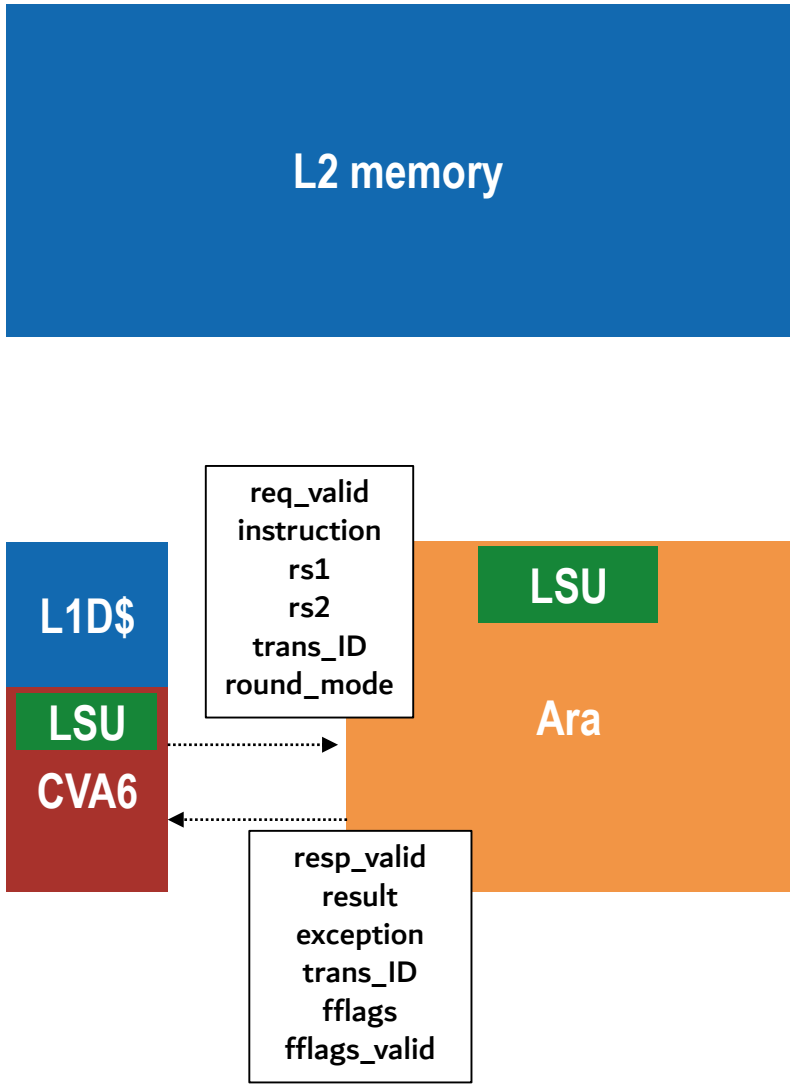

P
P

Deep dive into Ara's interfaces


P
P

Instruction request – response

```
module cva6 (
  output logic          req_valid,
  input  logic          req_ready,
  output logic [31:0]
instruction,
  output logic [63:0] rs1,
  output logic [63:0] rs2,
  output logic [3:0]  trans_ID,
  output roundmode_e
round_mode,

  input  logic
resp_valid,
  output logic
resp_ready,
  input  [63:0]         result,
  input  exception_t    exception,
  input  logic [3:0]    trans_ID,
  input  logic [4:0]    fflags,
  input  logic
```

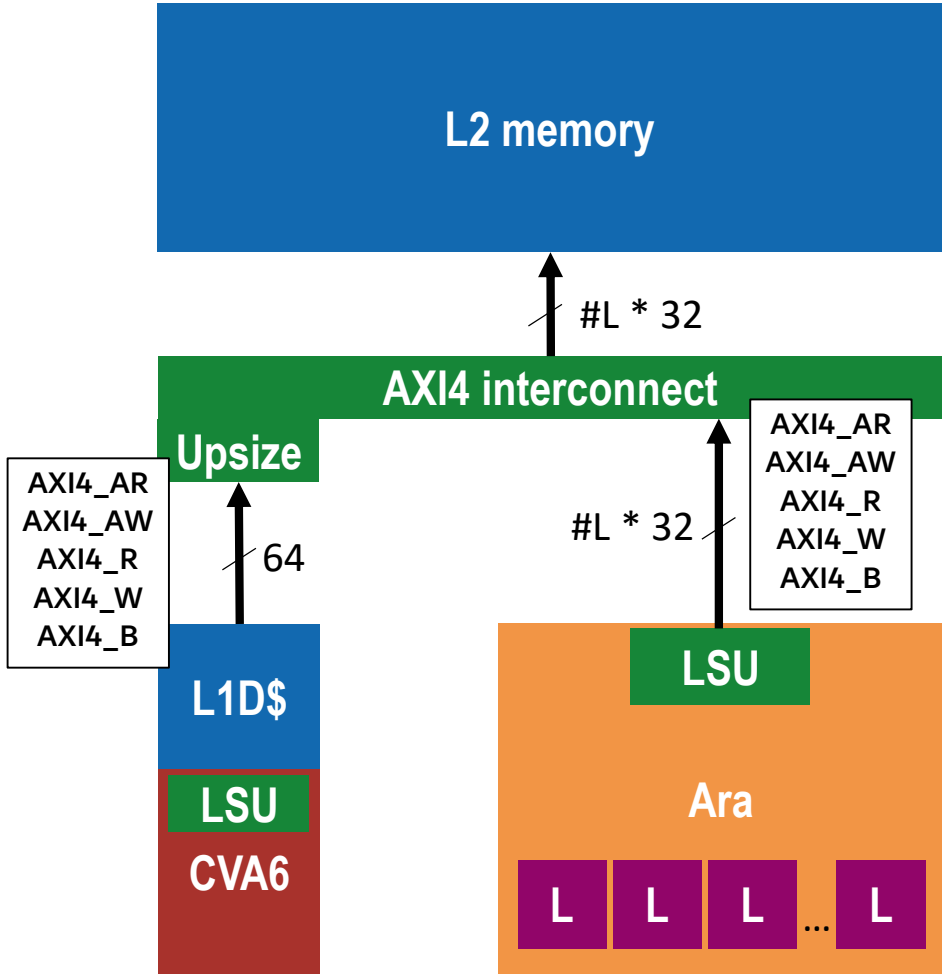



P
P

Memory interface

- **CVA6** and **Ara** have **independent LSUs**
- **AXI4** protocol and interconnect
- **Serialization** to/from memory

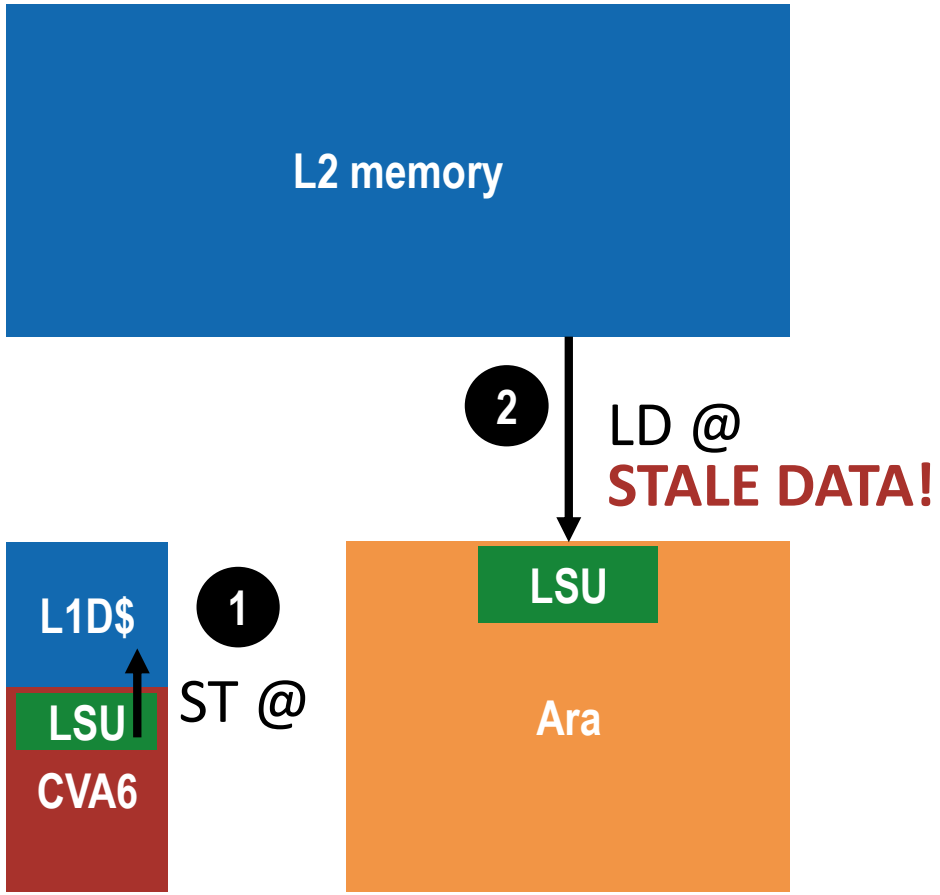
- **Wide-data port** → #Lanes * 32-bit
  e.g., 8-lane Ara: 256-bit data bus
- Wide-data port → **Simple interconnect**




P
P

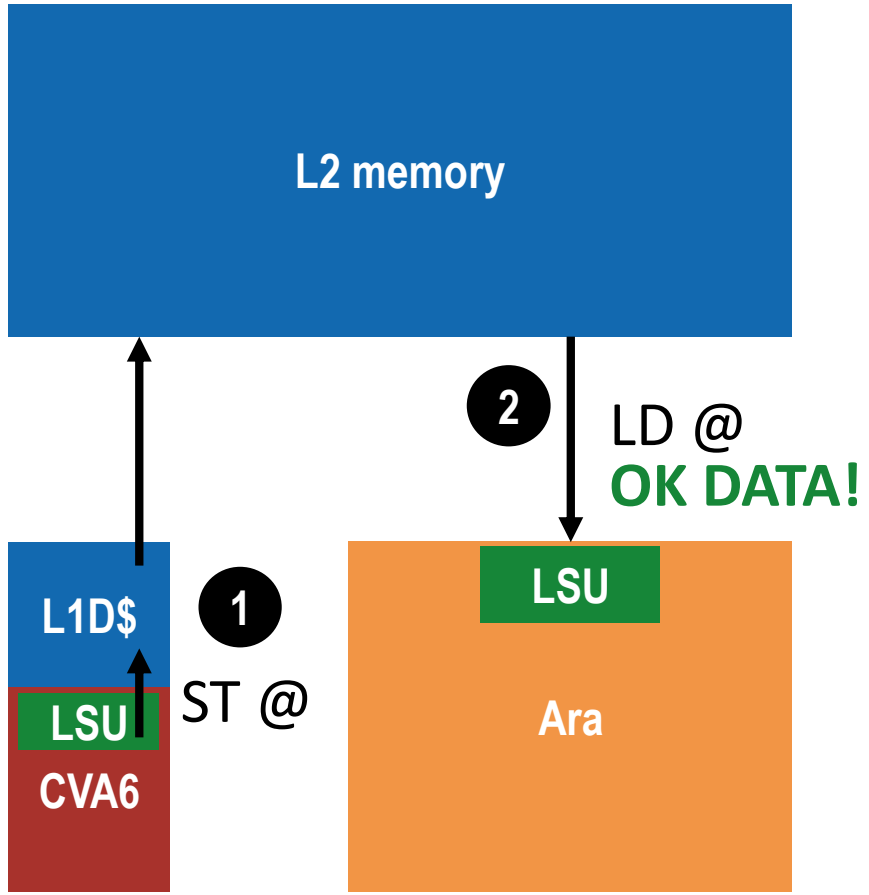
The coherence problems

1. CVA6 writes to *addr0*, Ara reads *addr0*



The coherence problems

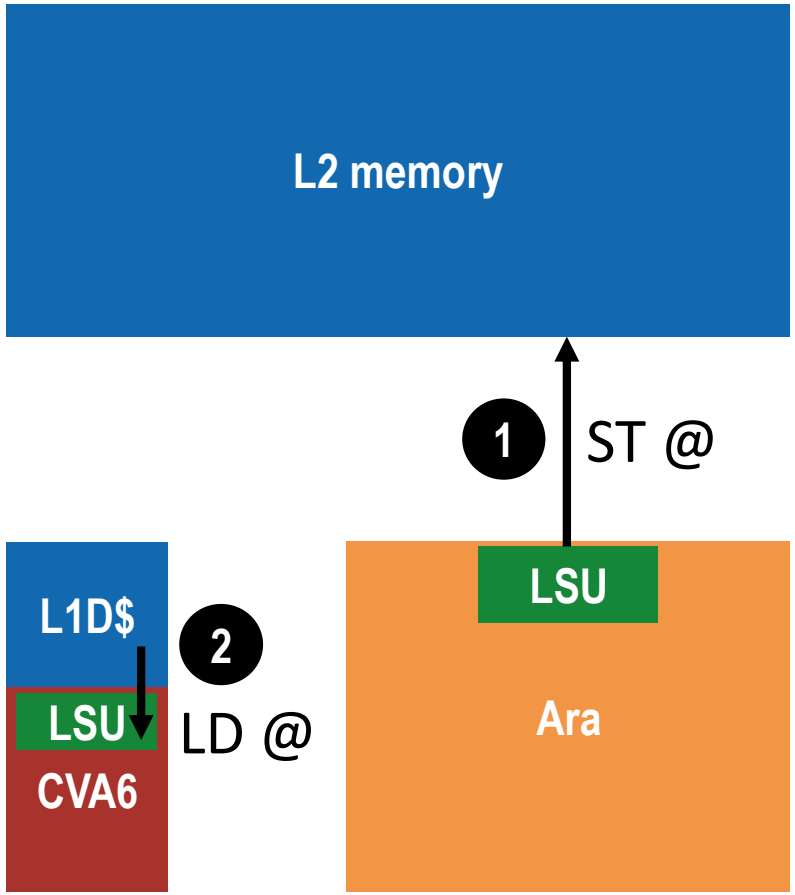
1. CVA6 writes to *addr0*, Ara reads *addr0*
   SOLUTION: write-through L1D$



The coherence problems

1. CVA6 writes to *addr0*, Ara reads *addr0*
   SOLUTION: write-through L1D$

2. Ara writes to addr0, CVA6 reads addr0

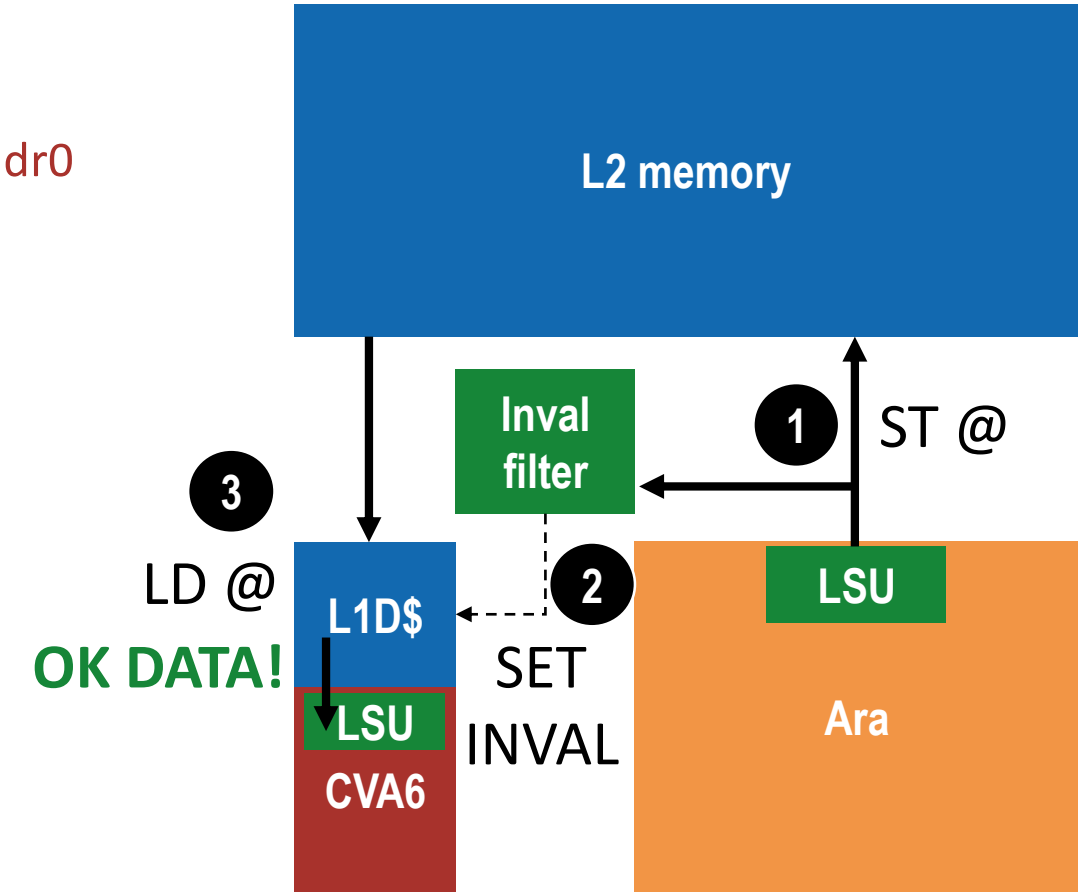


STALE DATA!

The coherence problems

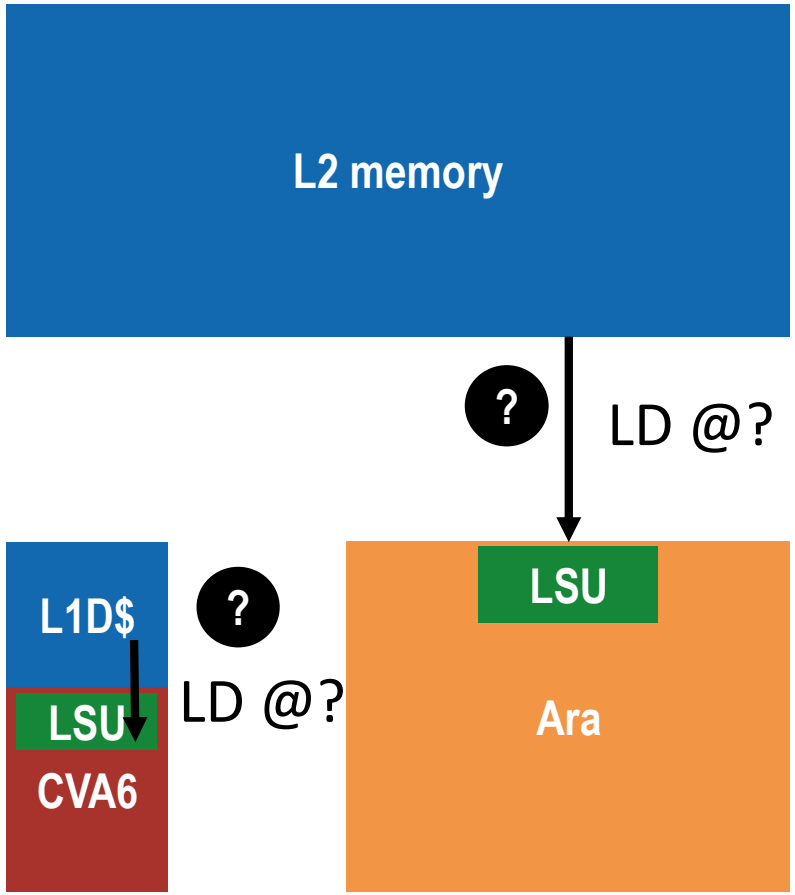
1. CVA6 writes to *addr0*, Ara reads *addr0*
   SOLUTION: write-through L1-D$

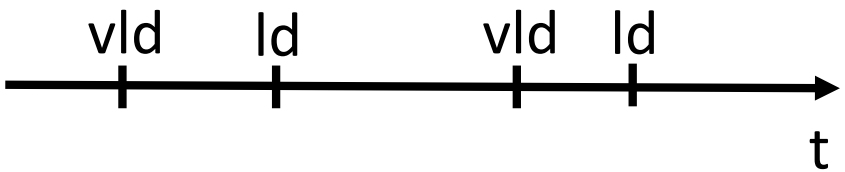
2. Ara writes to addr0, CVA6 reads addr0
   SOLUTION: L1-D$ set invalidation



The memory-ordering problem

1. How to order scalar and vector loads?
   SOLUTION: not a problem

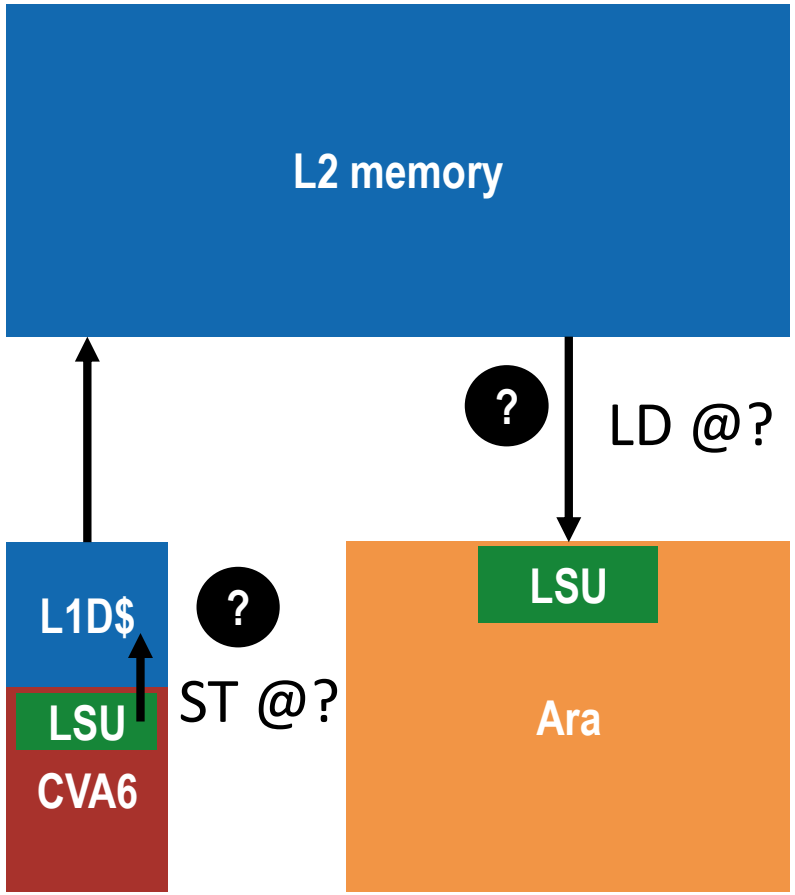


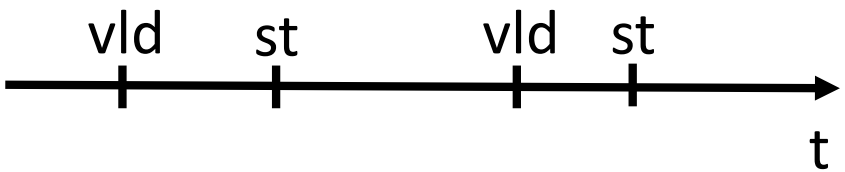


The memory-ordering problem

1. How to order scalar and vector loads?
   SOLUTION: not a problem

2. How to order scalar st and vector ld?
   st after vld: issue st only if no pending vld
   vld after st: start vld only if no pending st



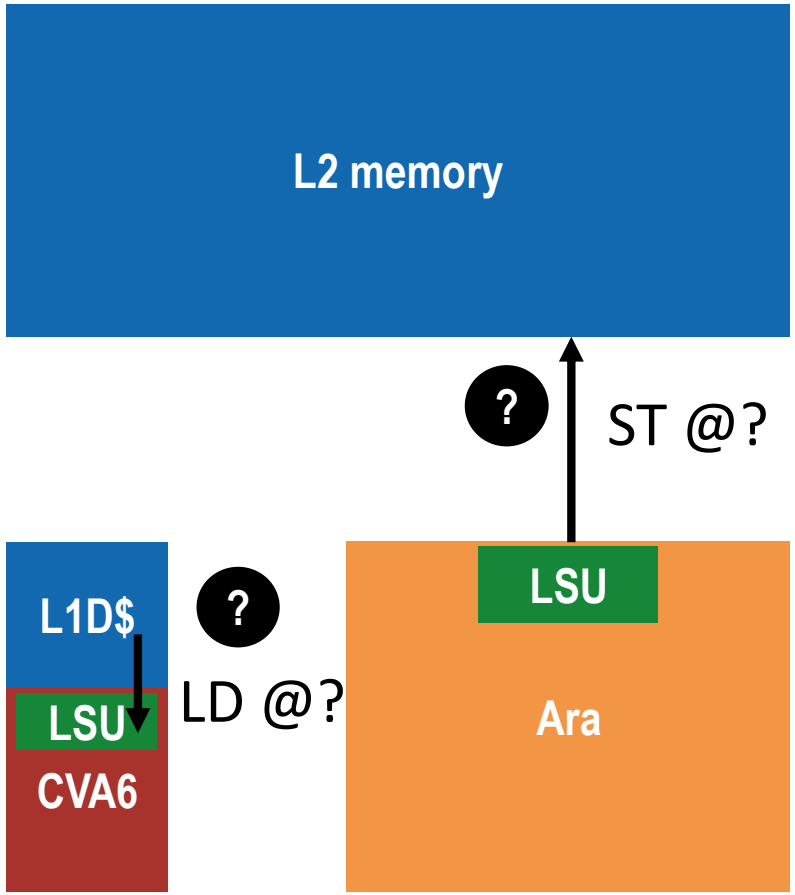


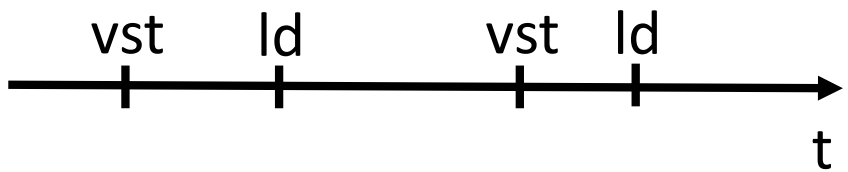
The memory-ordering problem

1. How to order scalar and vector loads?
   SOLUTION: not a problem

2. How to order scalar st and vector ld?
   st after vld: issue st only if no pending vld
   vld after st: start vld only if no pending st

3. How to order vector st and scalar ld?
   ld after vst: issue ld only if no pending vst
   vst after ld: ordered by the scoreboard





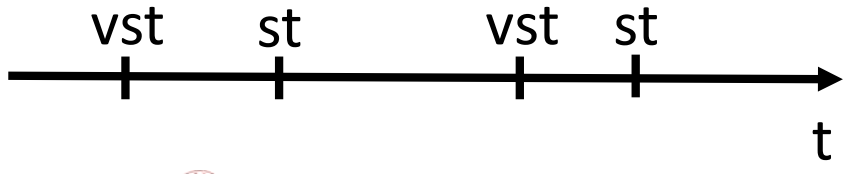
The memory-ordering problem

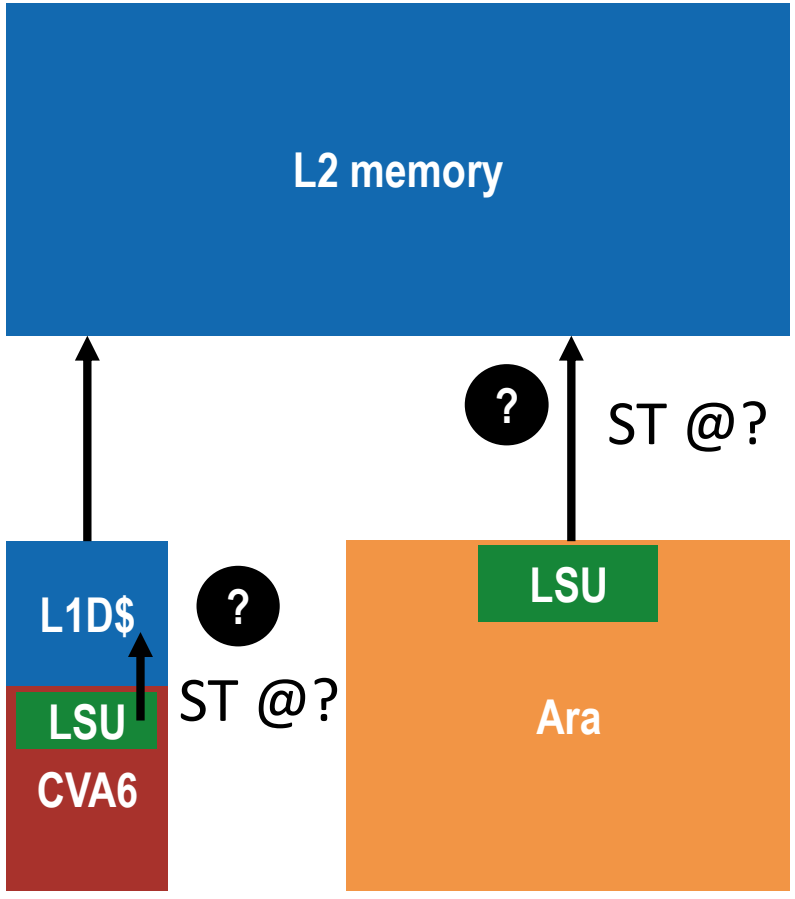
1. How to order scalar and vector loads?
   SOLUTION: not a problem

2. How to order scalar st and vector ld?
   st after vld: issue st only if no pending vld
   vld after st: start vld only if no pending st

3. How to order vector st and scalar ld?
   ld after vst: issue ld only if no pending vst
   vst after ld: ordered by the scoreboard

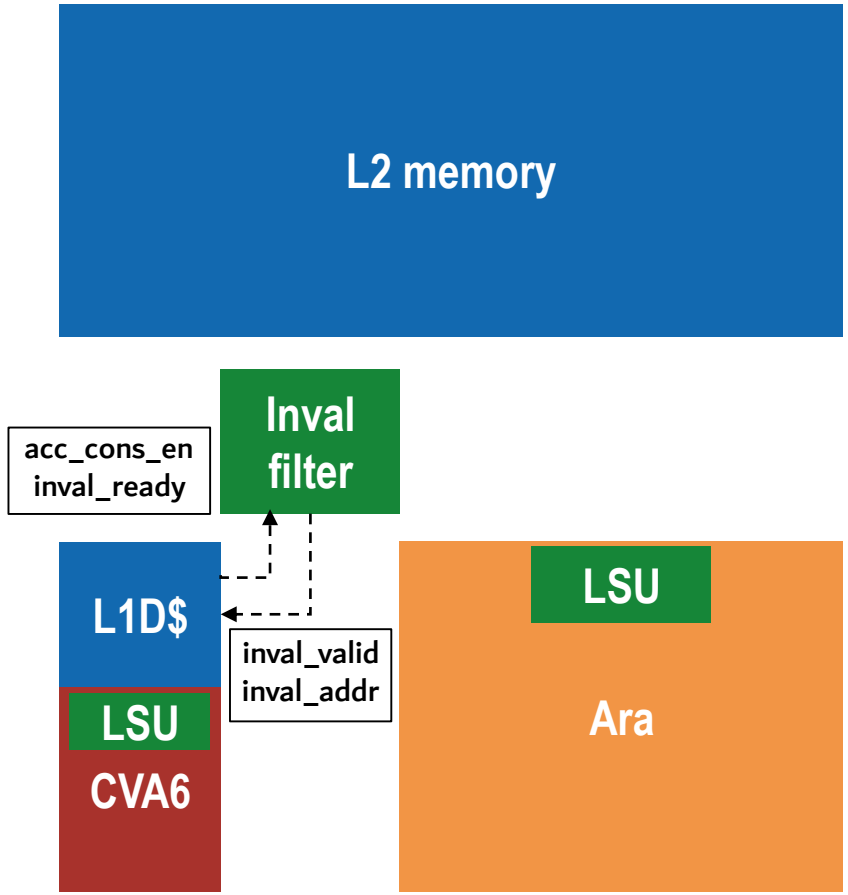
4. How to order scalar st and vector st?
   st after vst: issue st only if no pending vst
   vst after st: start vst only if no pending st





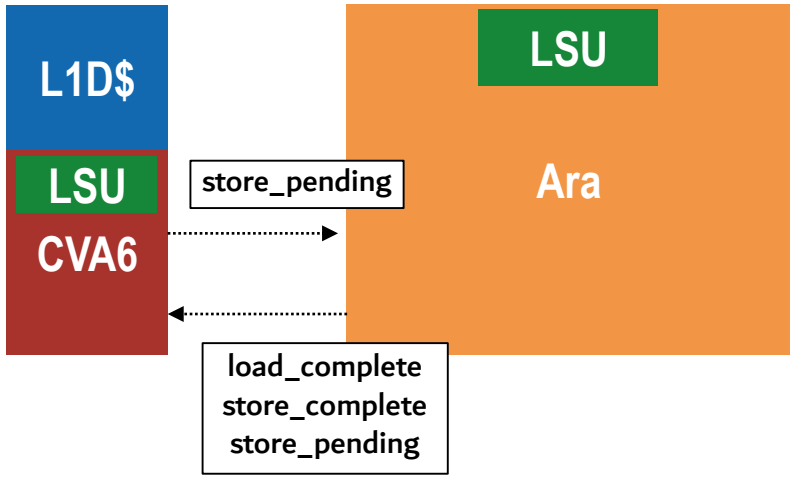
Coherence interface

```
module cva6 (
  input  logic
inval_valid,
  input  logic [63:0]
inval_addr,
  …
  output logic
inval_ready,
  output logic
acc_cons_en
)
```



Consistency interface

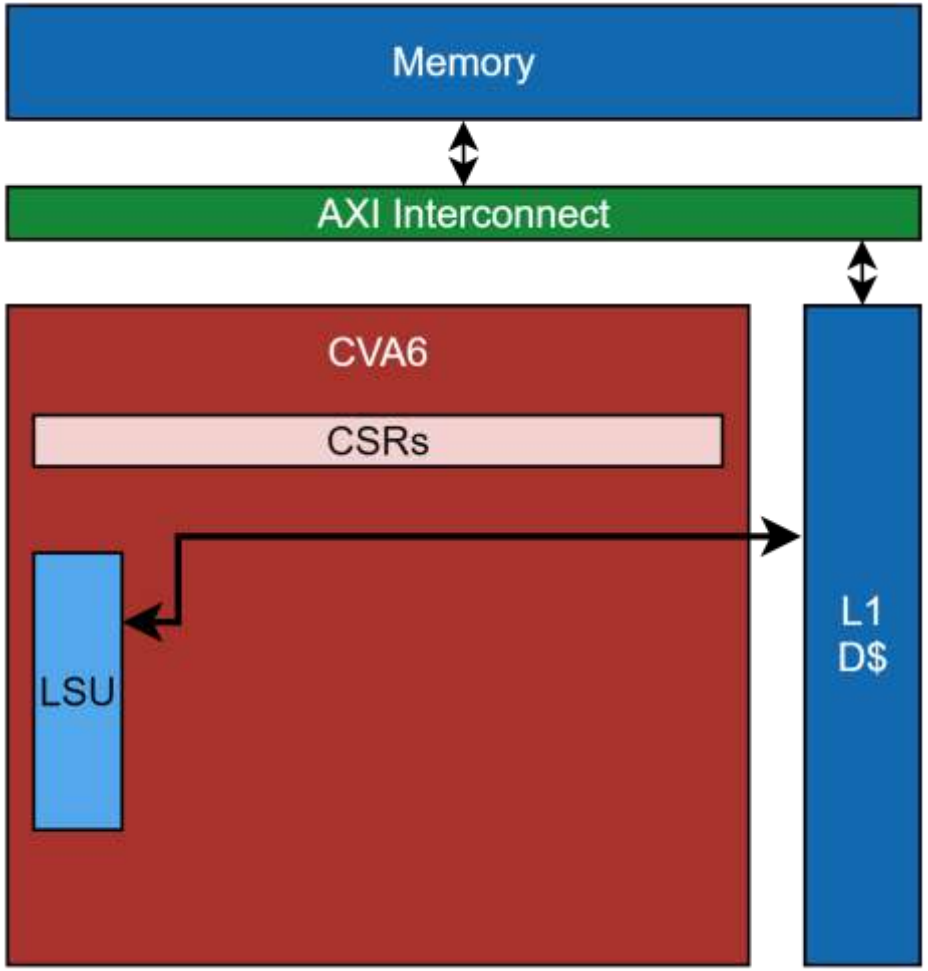
```
module cva6 (
  input  logic store_complete,
  input  logic load_complete,
  input  logic store_pending,
  …
  output logic store_pending
)
```




P
P

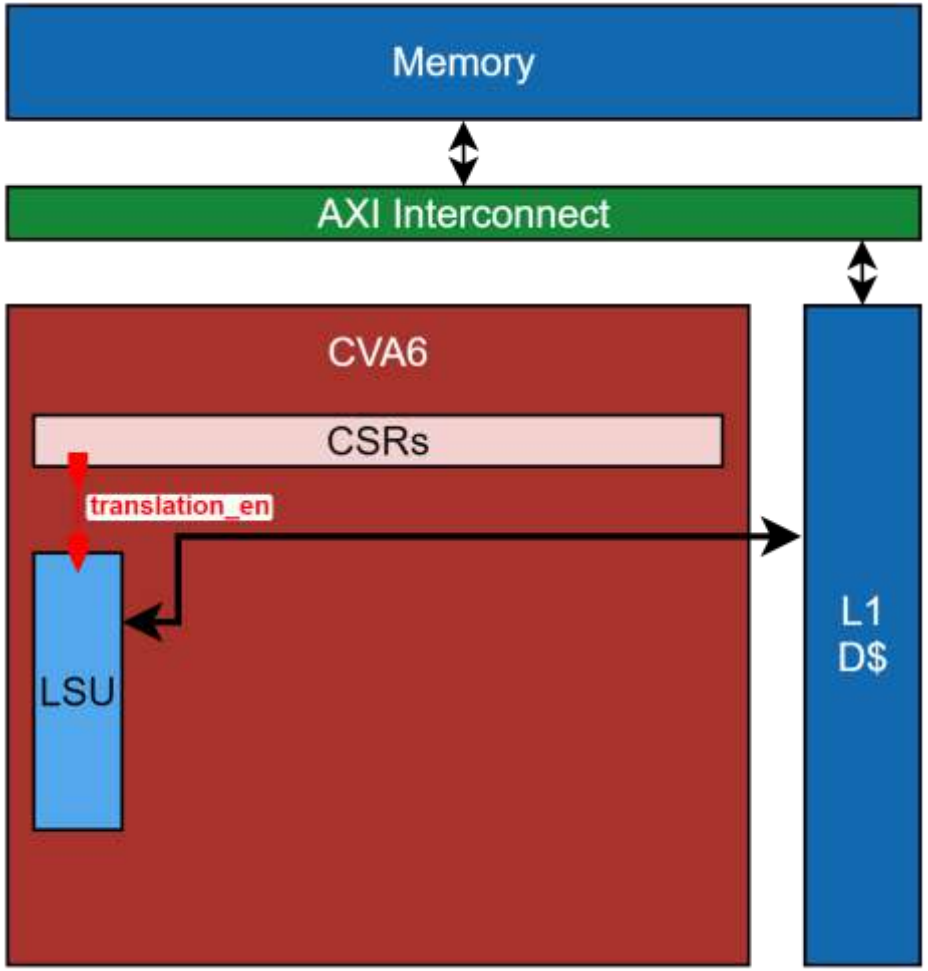
Background – CVA6 MMU

- CVA6 – RV64GC with OS support



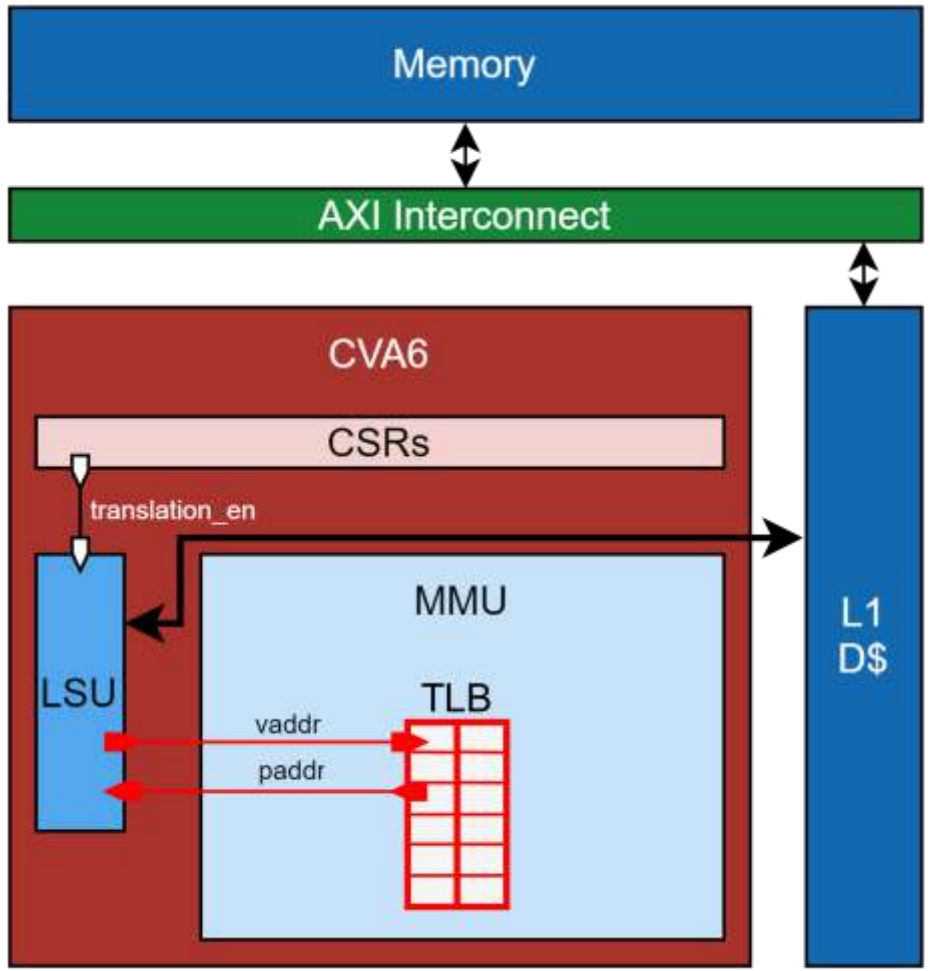
Background – CVA6 MMU

- CVA6 – RV64GC with OS support



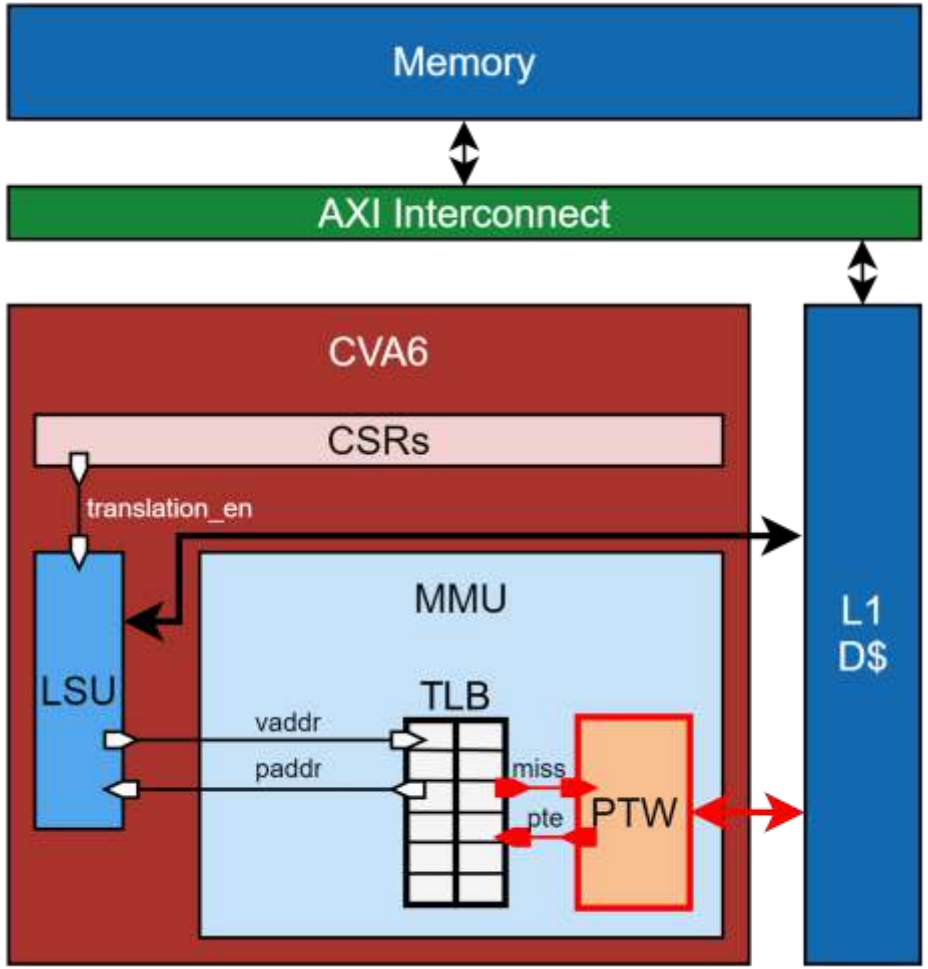
Background – CVA6 MMU

- CVA6 – RV64GC with OS support
- **Private MMU** (Memory Management Unit)
  - Translate a **virtual address into** a **physical** one
  - **TLB** – A cache for the page table entries



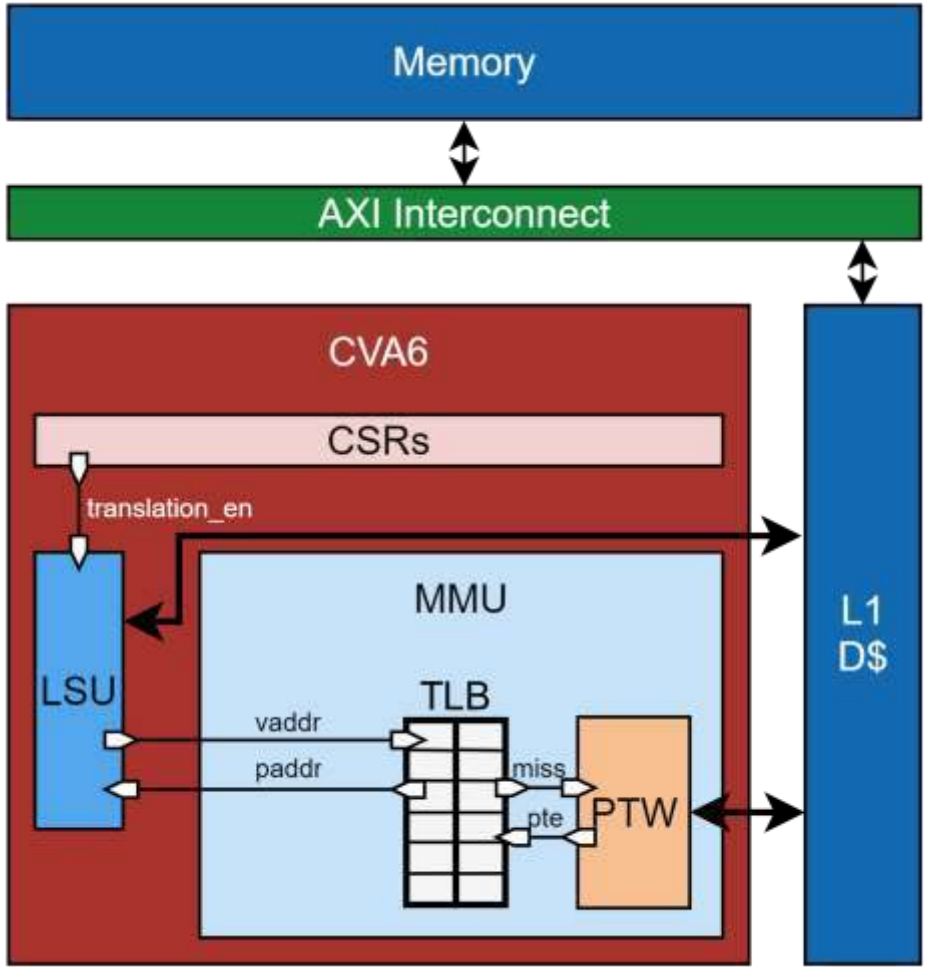
Background – CVA6 MMU

- CVA6 – RV64GC with OS support
- Private MMU (Memory Management Unit)
  - Translate a virtual address into a physical one
  - TLB – A cache for the page table entries
- If **TLB** does **not** have the **entry** (**TLB miss**)
  - PTW (Page Table Walker) fetches it from memory



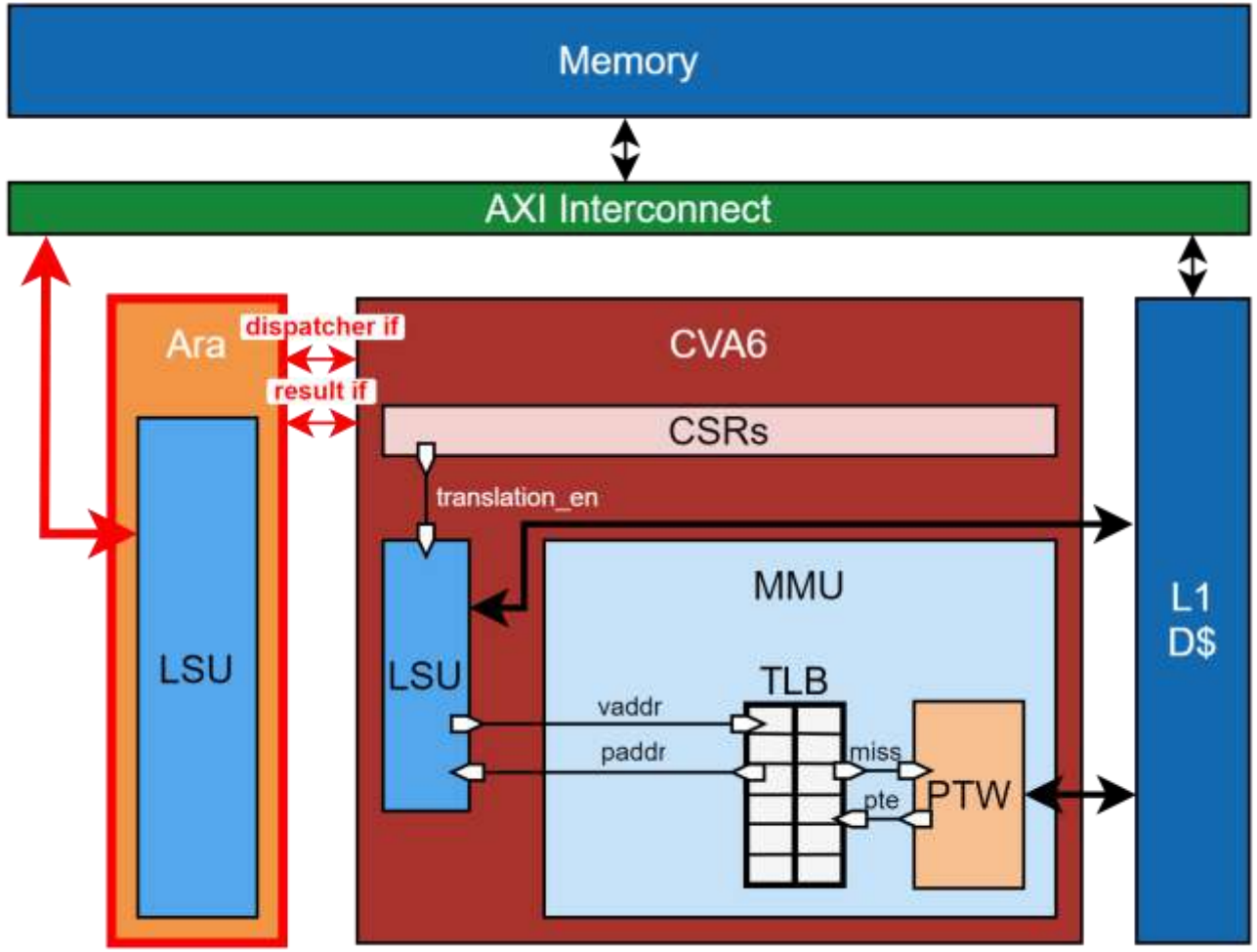
Background – CVA6 MMU

- CVA6 – RV64GC with OS support
- Private MMU (Memory Management Unit)
  - Translate a virtual address into a physical one
  - TLB – A cache for the page table entries
- If TLB does not have the entry (TLB miss)
  - PTW (Page Table Walker) fetches it from memory



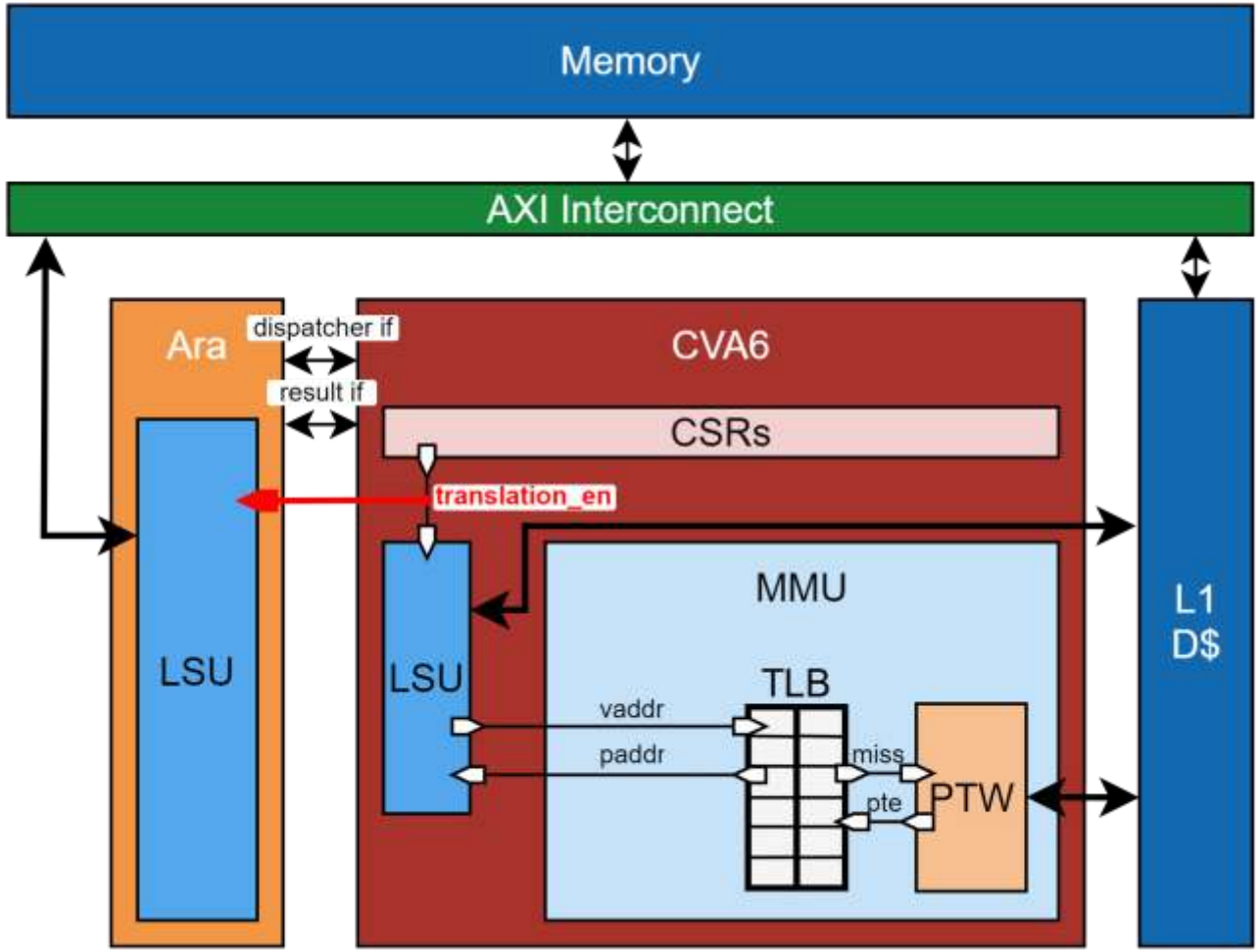
AraOS – How to translate virtual addresses?

- Ara2 – RV64GC**V** without OS support



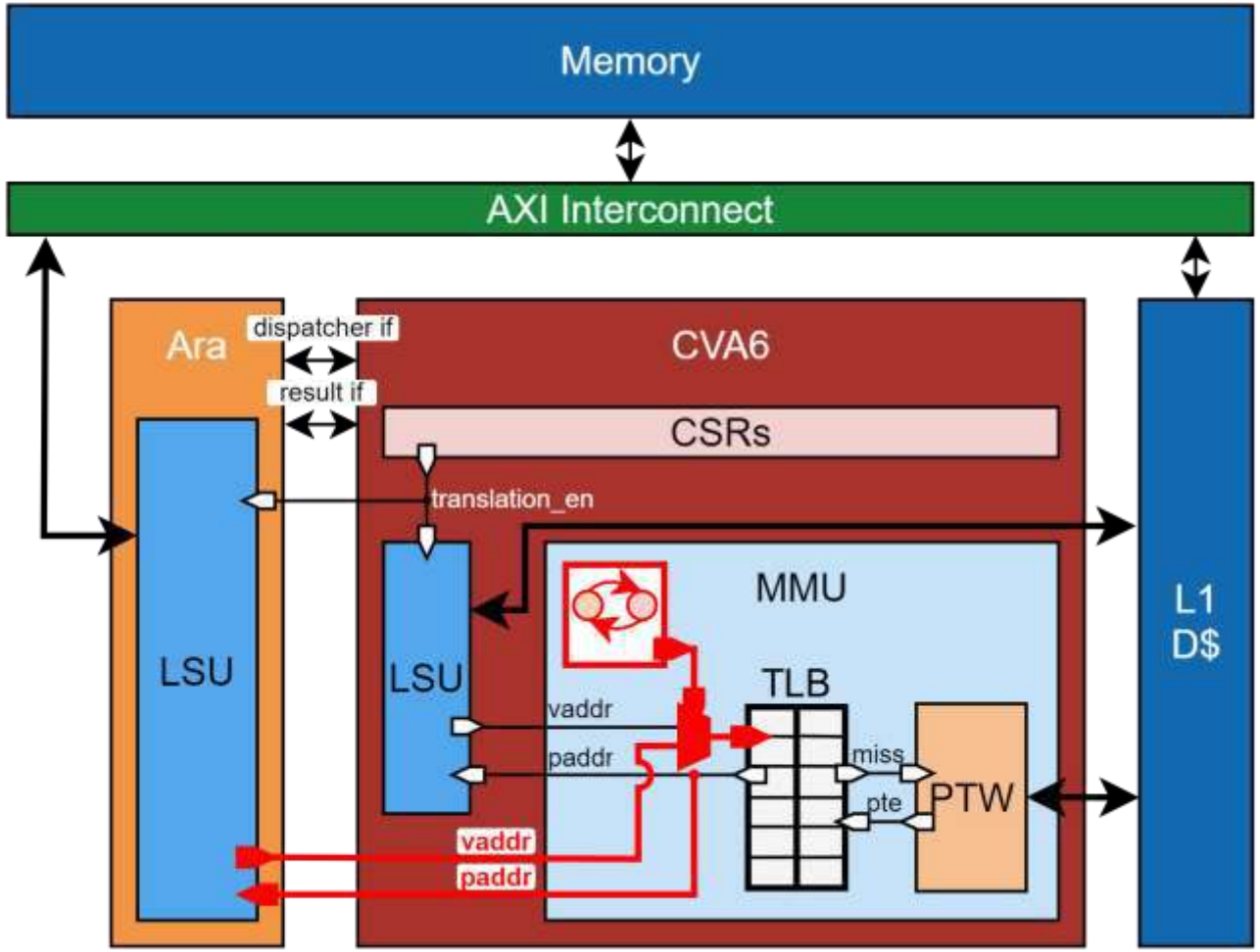
AraOS – Address translation in Ara2

- Ara2 – RV64GCV without OS support
- **Add OS support** to Ara2
  - Controlled by CVA6



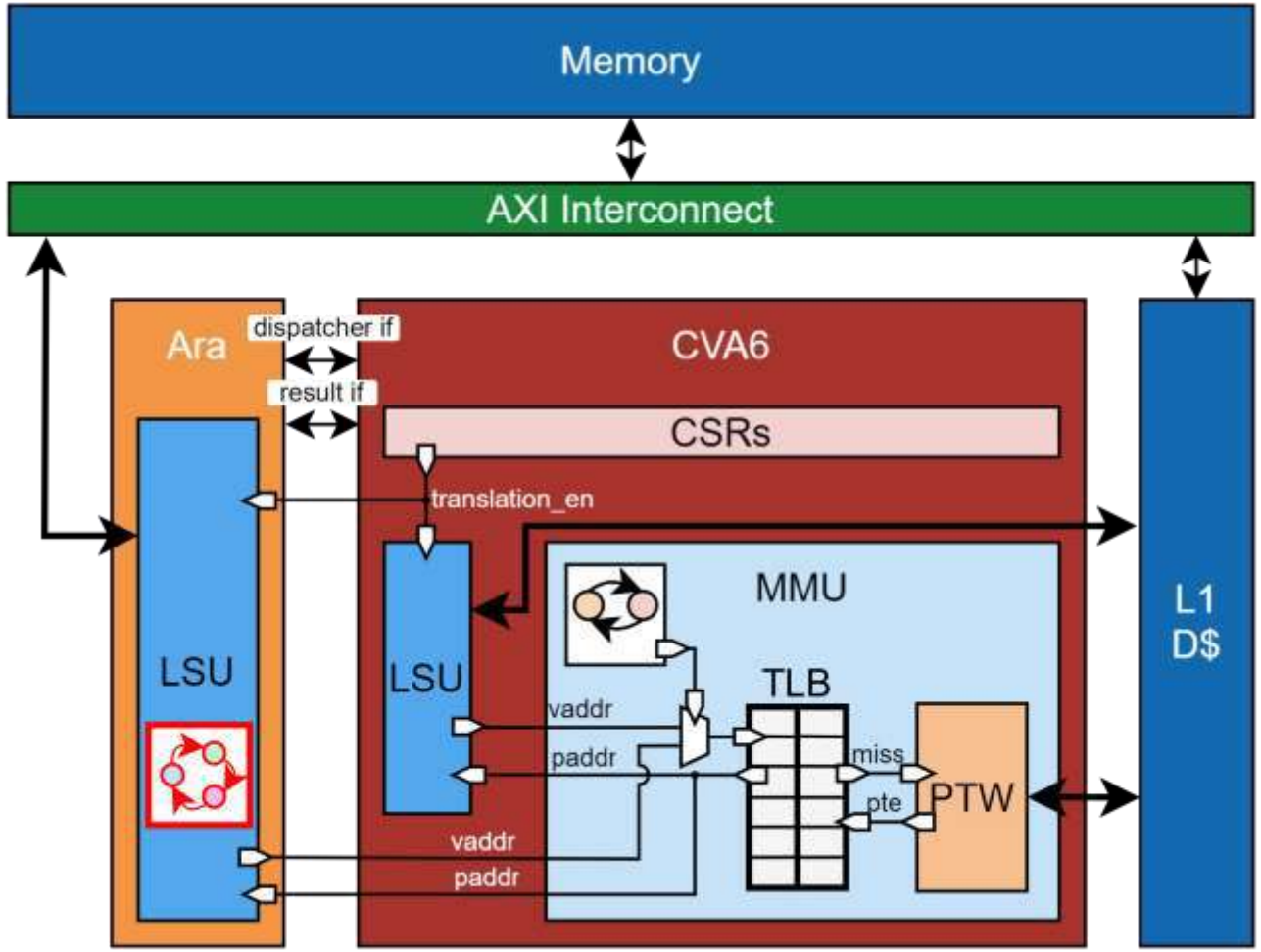
AraOS – Address translation in Ara2

- Ara2 – RV64GCV without OS support
- Add OS support to Ara2
  - Controlled by CVA6
  - Share CVA6 MMU
  - FSM arbitrates the accesses (CVA6 priority)



AraOS – Address translation in Ara2

- Ara2 – RV64GCV without OS support
- Add OS support to Ara2
  - Controlled by CVA6
  - Share CVA6 MMU
  - FSM arbitrates the accesses (CVA6 priority)
- FSM inside Ara2's private vector LSU
  - **Translate one address per AXI burst**
  - Already **aligned** to **4-KiB boundary** (AXI specs)
  - **Gather/Scatter → One translation per element**



AraOS – Handling page-fault exceptions

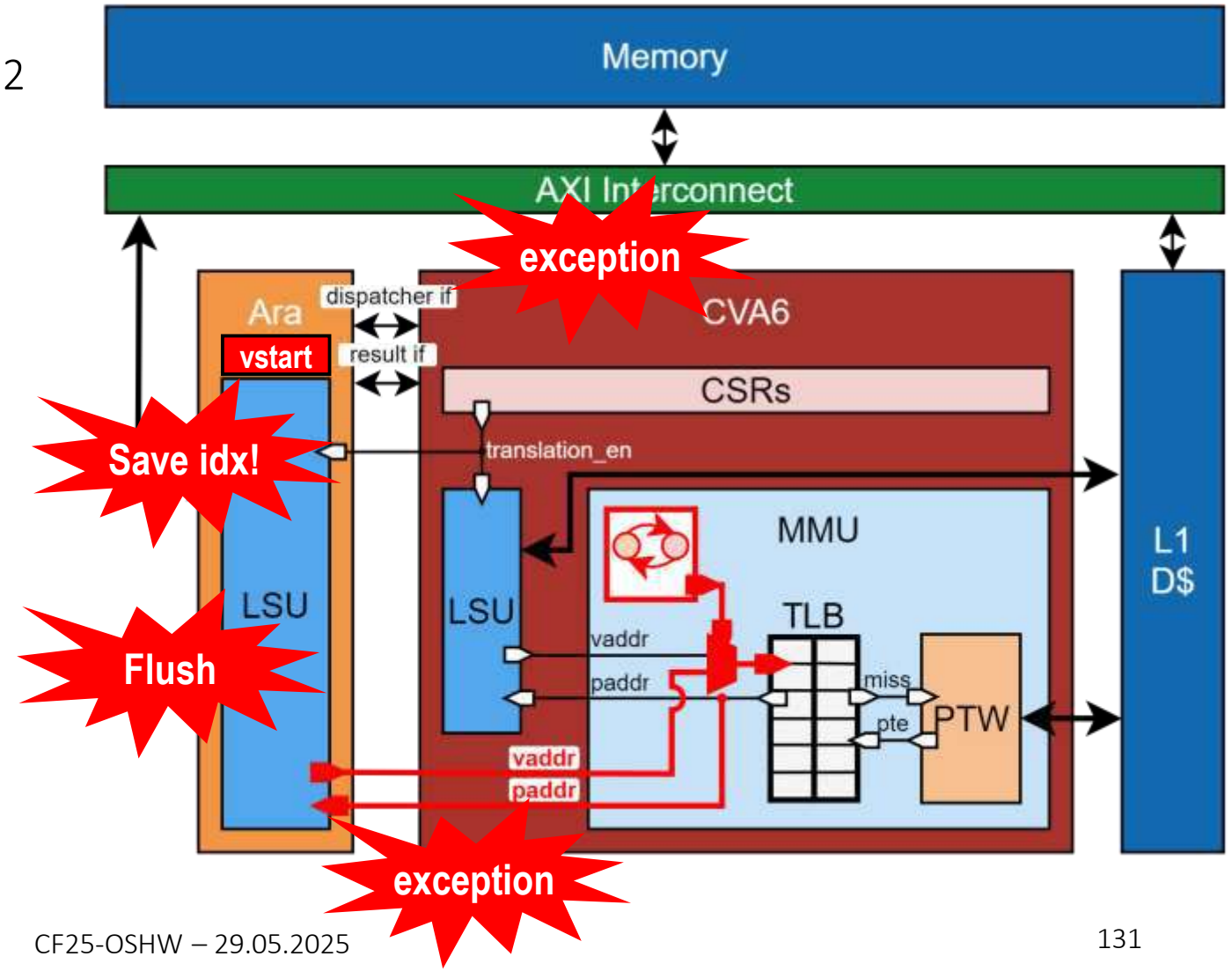
In case of a **page-fault exception**?

The **MMU reports** an **exception** to Ara2

Ara2

- Doesn't issue further AXI requests
- Save faulty element idx in vstart
- Flush microarchitectural state
- Report the exception to CVA6

CVA6 → handle the exception



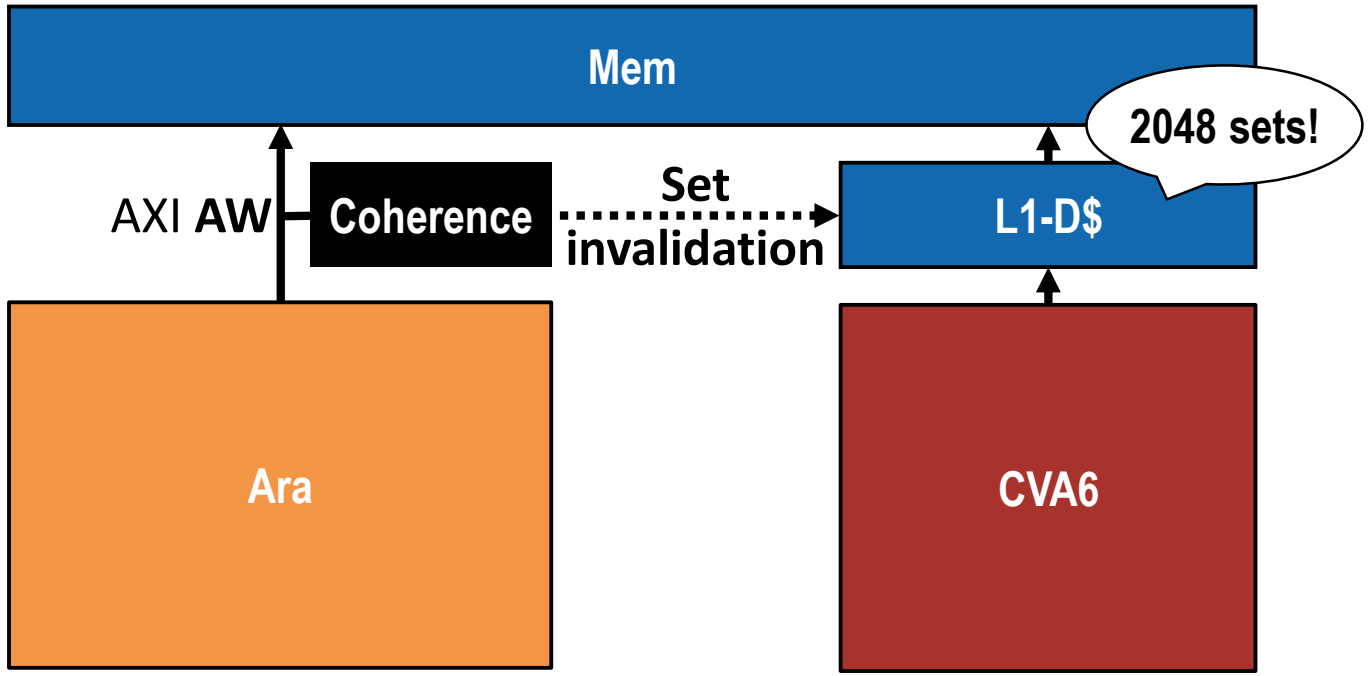
AraOS – Cache coherence

- Upon vector store → **Invalidate** related **cache sets** in CVA6's write-through L1-D$
- Check **AXI AW** channel and generate invalidation signals
- But… how can this work? CVA6 L1-D$ is **virtually indexed**, physically tagged
- AXI AW **is physical!**

12 LSBs of address are PHYSICAL!

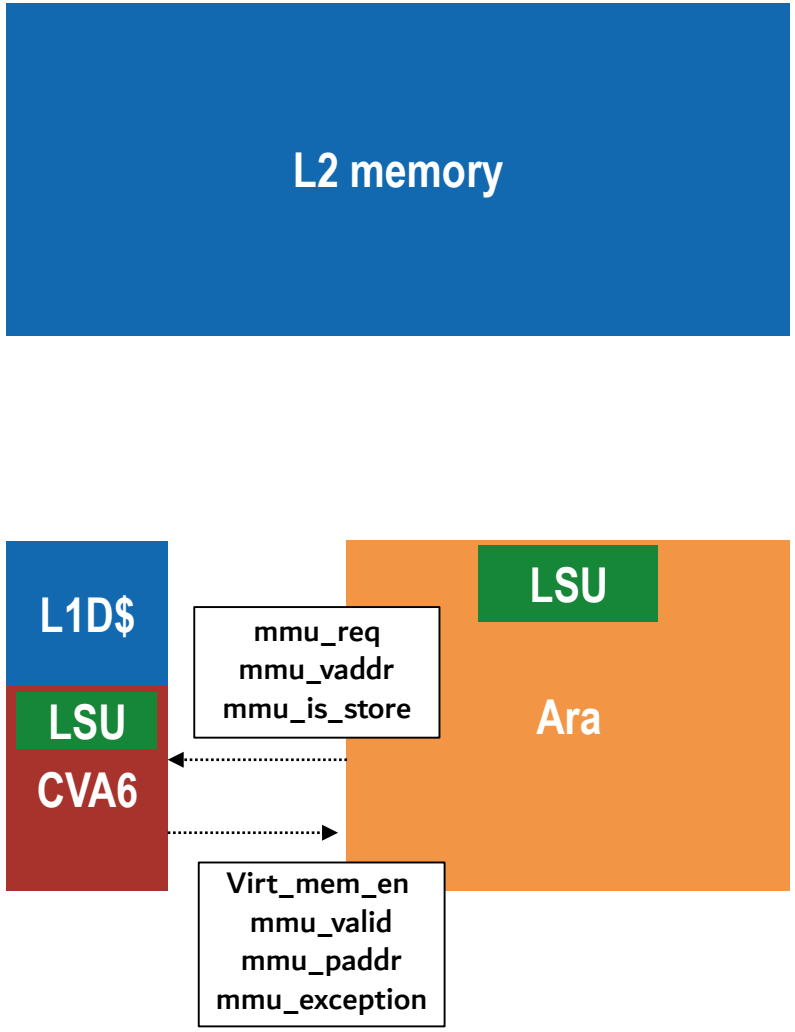
If L1-D$ cache sets ≤ 4096
Cache address index bits ≤ 12

Cache "is" physically indexed



MMU interface

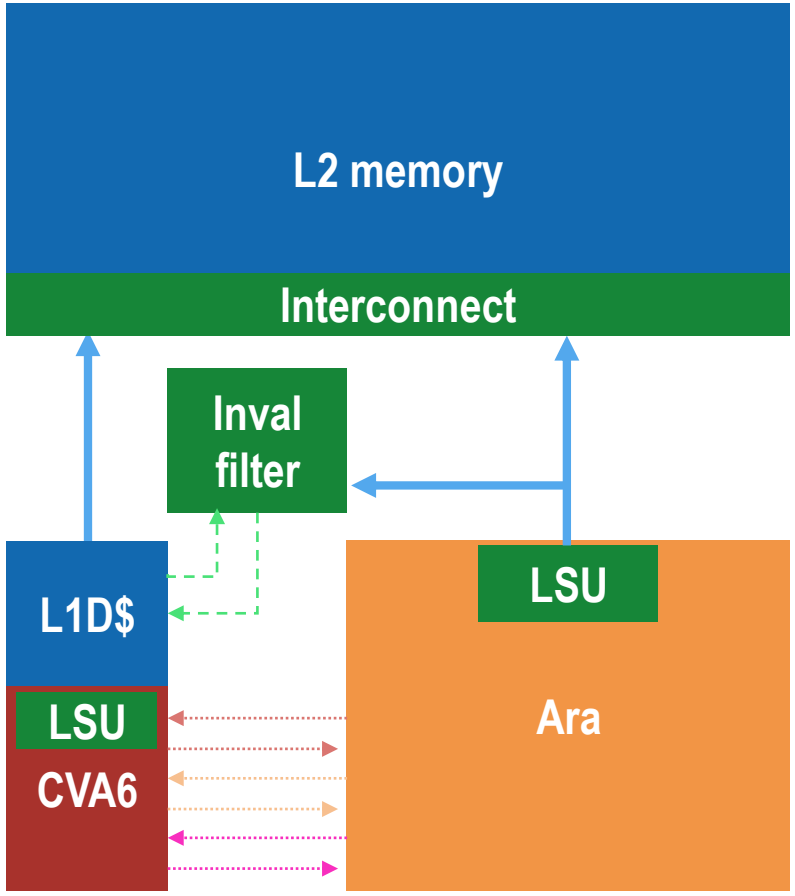
```
module cva6 (
  input  logic         mmu_req,
  input  vLen-1:0      mmu_vaddr,
  input  logic
mmu_is_store,
  …
  output logic
virt_mem_en,
  output logic         mmu_valid,
  output [pLen-1:0]    mmu_paddr,
  output exception_t
mmu_exception,
)
```




P
P

Interfaces: putting it all together

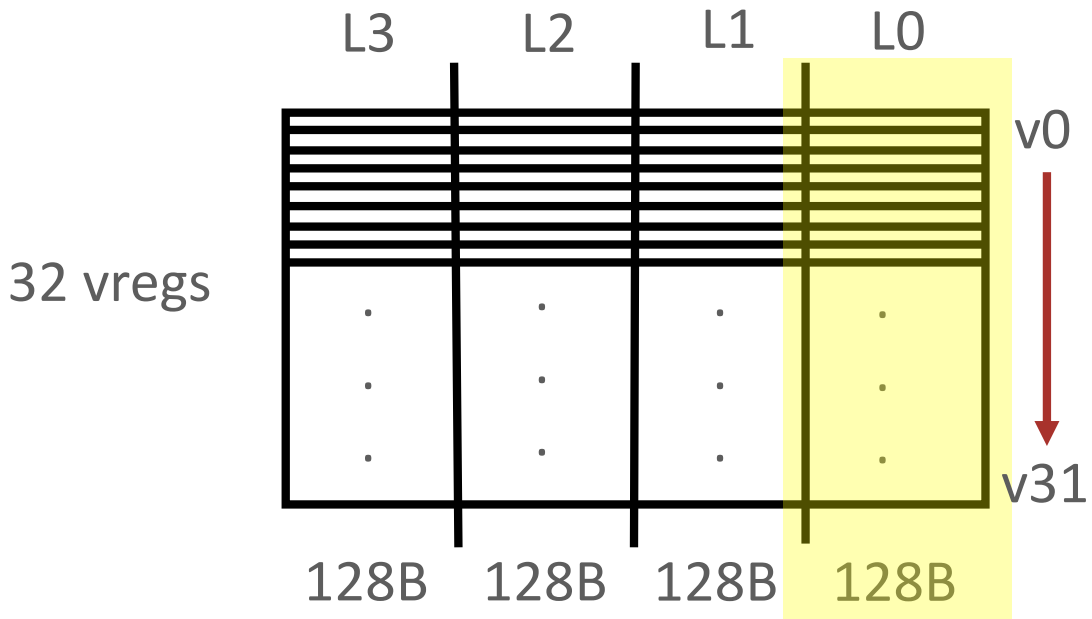
```
module cva6 (
  axi_req_intf_t
  axi_rsp_intf_t

  coh_req_intf_t
  coh_rsp_intf_t

  main_req_intf_t
  main_rsp_intf_t

  mmu_req_intf_t
  mmu_rsp_intf_t

  cons_req_intf_t
  cons_rsp_intf_t
)
```

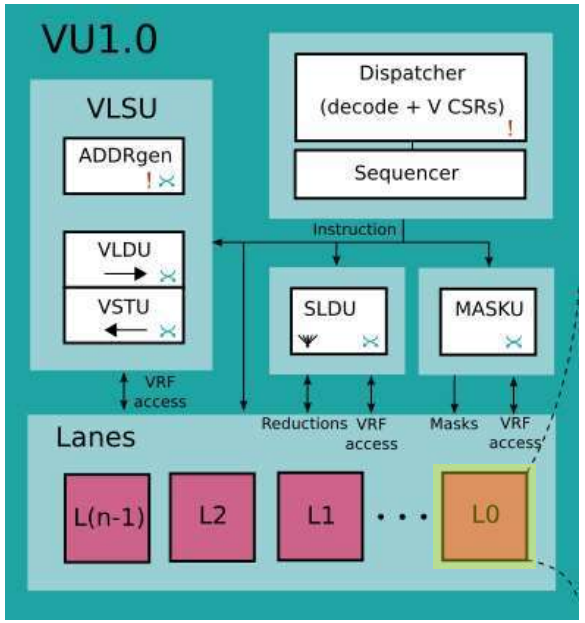



P
P

Ara's VRF Byte Layout

Ara 4 Lanes VRF
Size: 16 KiB
Split among the lanes!





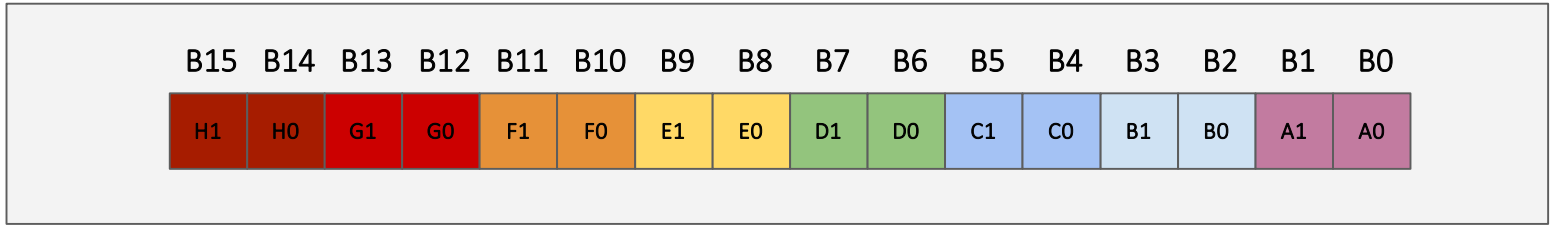
RISC-V V – VRF byte layout

The VRF byte layout should be exactly like in memory

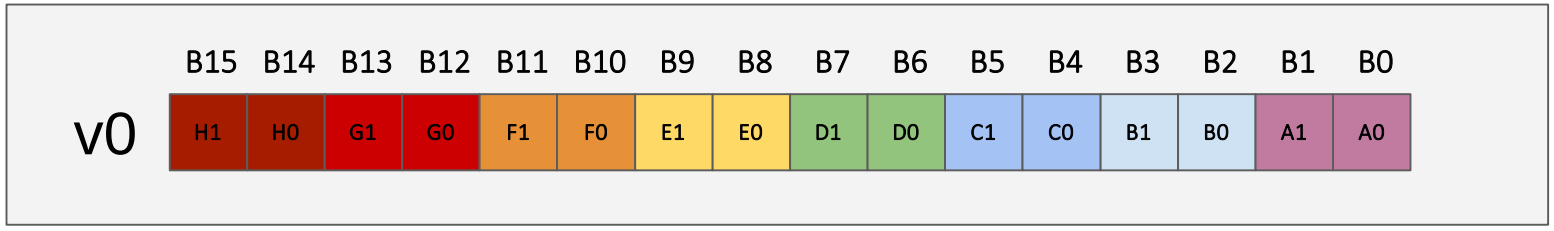
In Ara, we only emulate this behavior, but the internal mapping is more complex

This is how it should be, and how logically you should imagine it

Memory



VRF



Ara VRF byte layout

- Vector with 8 elements
- 2 Byte/element

Ara VRF

- Vector with 8 elements
- 2 Byte/element

Memory (outside Ara)

B15	B14	B13	B12	B11	B10	B9	B8	B7	B6	B5	B4	B3	B2	B1	B0
H1	H0	G1	G0	F1	F0	E1	E0	D1	D0	C1	C0	B1	B0	A1	A0

Ara VRF

- Vector with 8 elements
- 2 Byte/element

```
…
vld v0, (Memory)
…
```

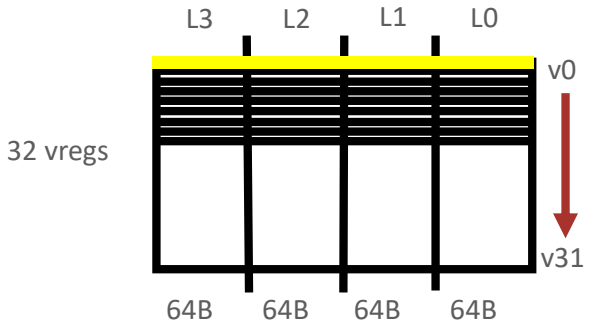
Memory (outside Ara)

B15	B14	B13	B12	B11	B10	B9	B8	B7	B6	B5	B4	B3	B2	B1	B0
H1	H0	G1	G0	F1	F0	E1	E0	D1	D0	C1	C0	B1	B0	A1	A0

Ara VRF

- Vector with 8 elements
- 2 Byte/element

```
…
vld v0, (Memory)
…
```



Memory (outside Ara)

B15	B14	B13	B12	B11	B10	B9	B8	B7	B6	B5	B4	B3	B2	B1	B0
H1	H0	G1	G0	F1	F0	E1	E0	D1	D0	C1	C0	B1	B0	A1	A0

v0 (VRF, inside Ara)

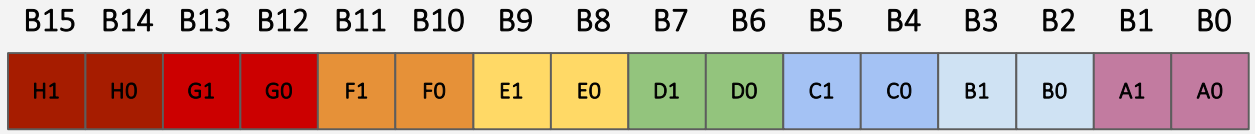
	B15	B14	B13	B12	B11	B10	B9	B8	B7	B6	B5	B4	B3	B2	B1	B0
L0																
L1																
L2																
L3																

Ara VRF

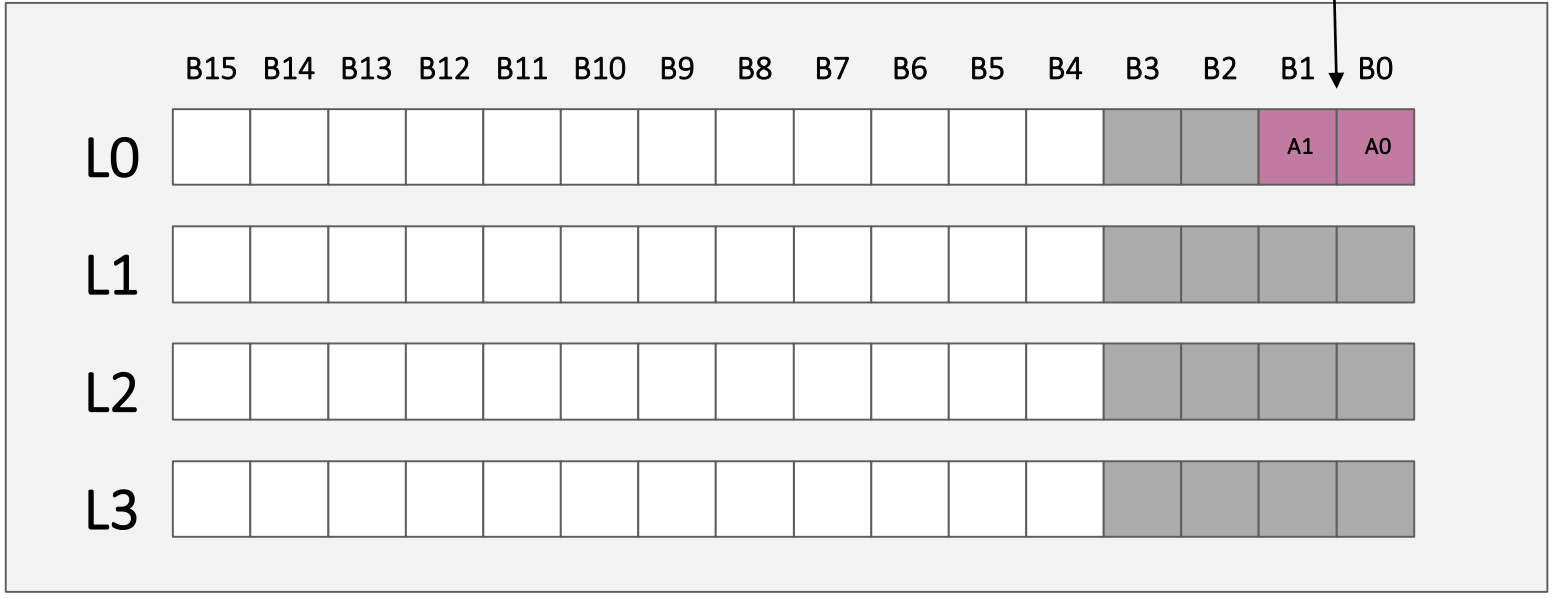
- Vector with 8 elements
- 2 Byte/element

```
…
vld v0, (Memory)
…
```

Memory (outside Ara)



v0 (VRF, inside Ara)

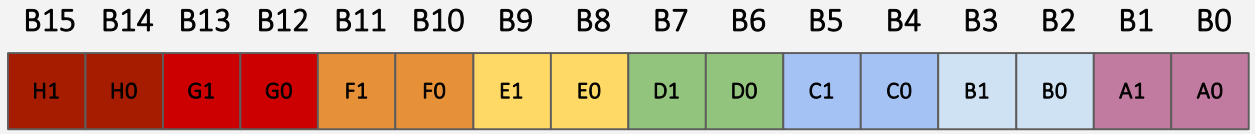


Ara VRF

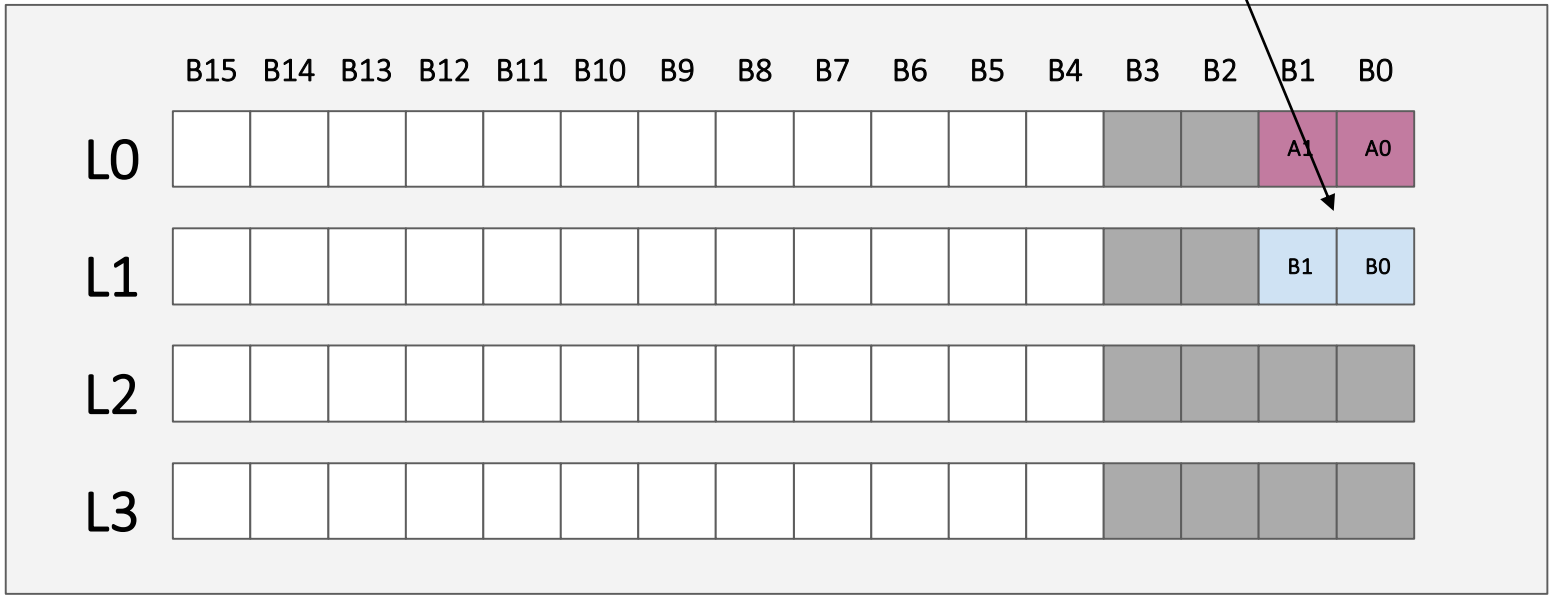
- Vector with 8 elements
- 2 Byte/element

```
…
vld v0, (Memory)
…
```

Memory (outside Ara)



v0 (VRF, inside Ara)

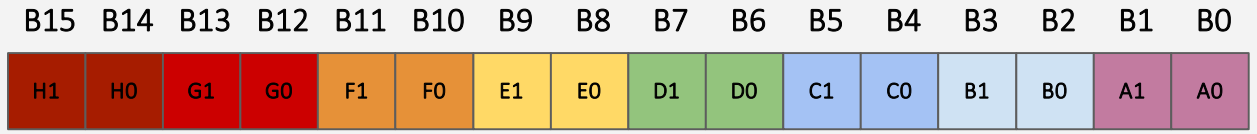


Ara VRF

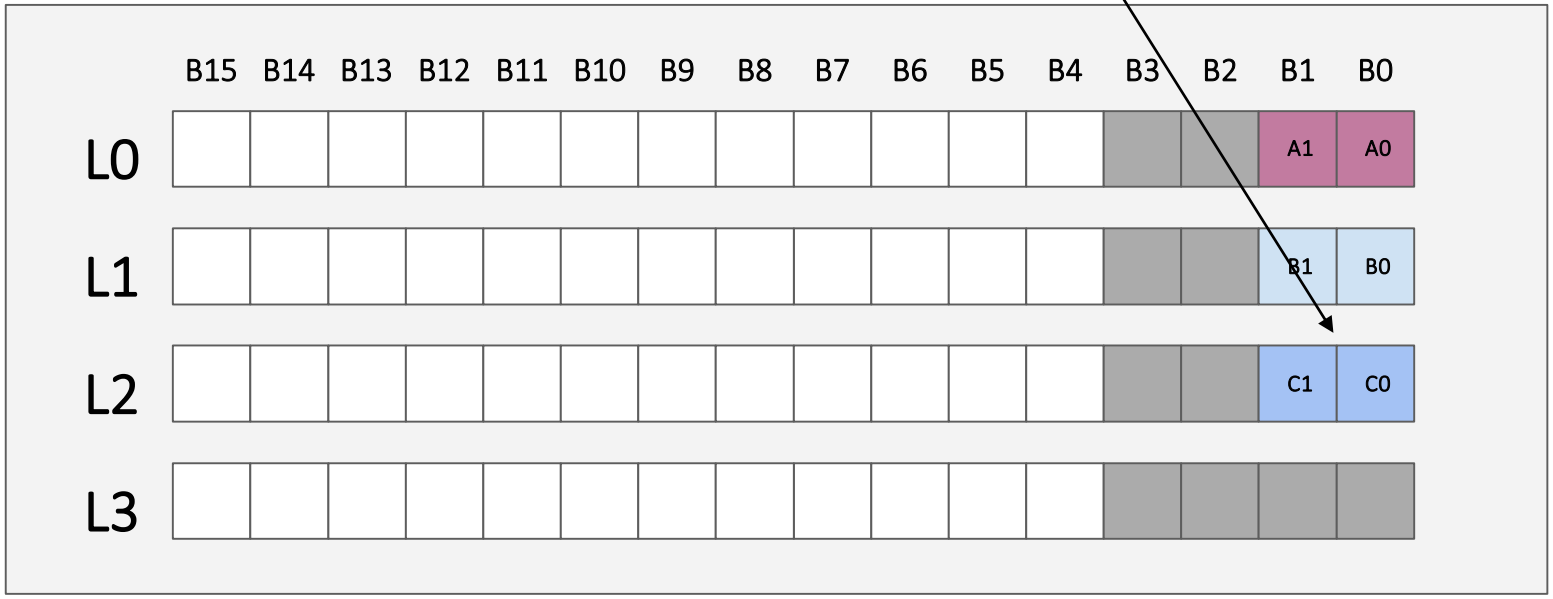
- Vector with 8 elements
- 2 Byte/element

```
…
vld v0, (Memory)
…
```

Memory (outside Ara)



v0 (VRF, inside Ara)

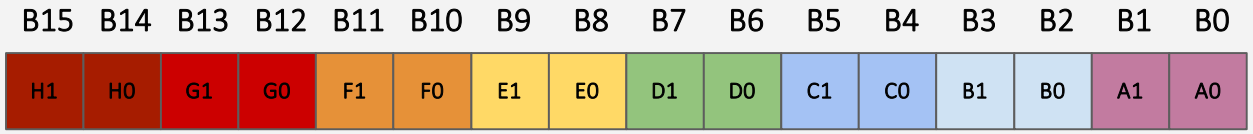


Ara VRF

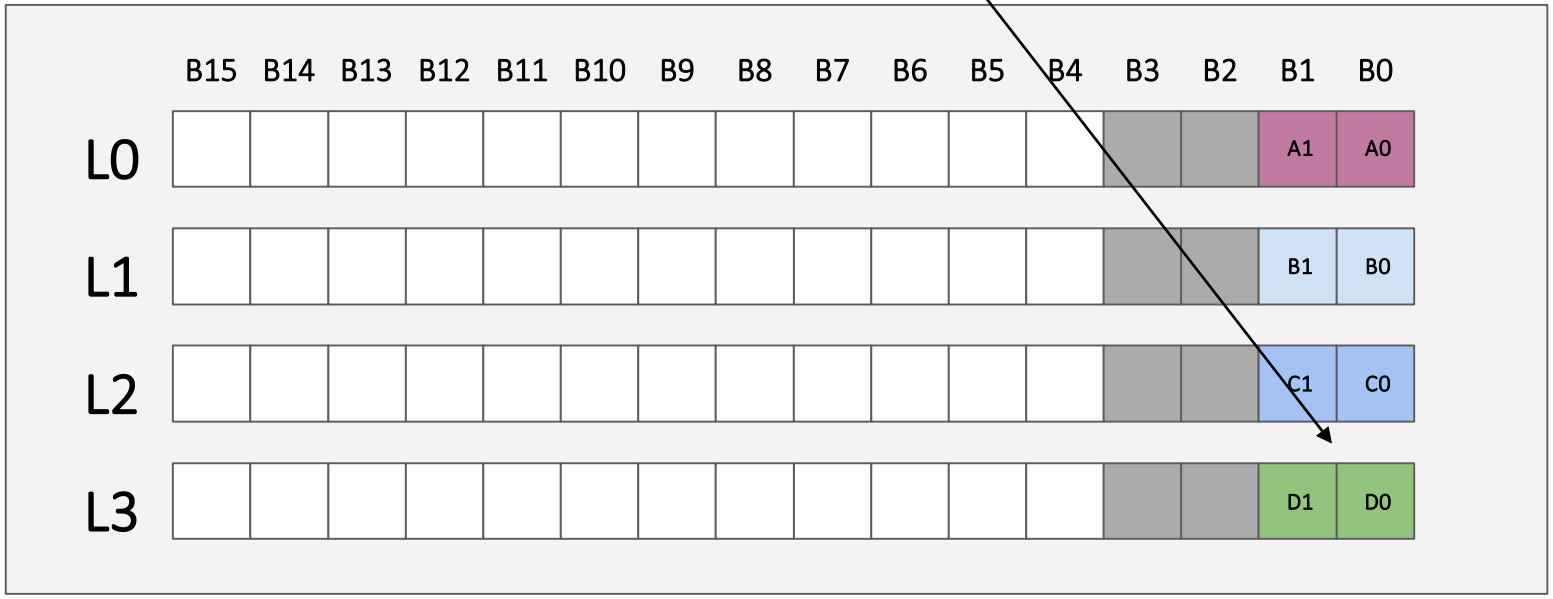
- Vector with 8 elements
- 2 Byte/element

```
…
vld v0, (Memory)
…
```

Memory (outside Ara)



v0 (VRF, inside Ara)

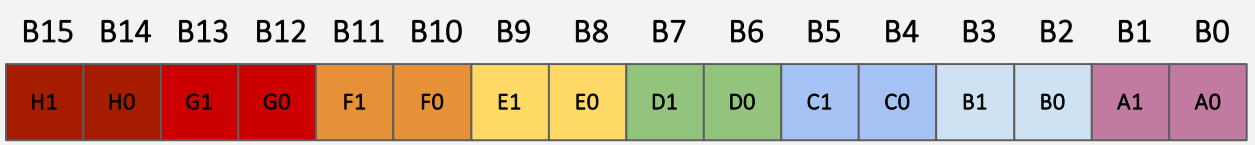


Ara VRF

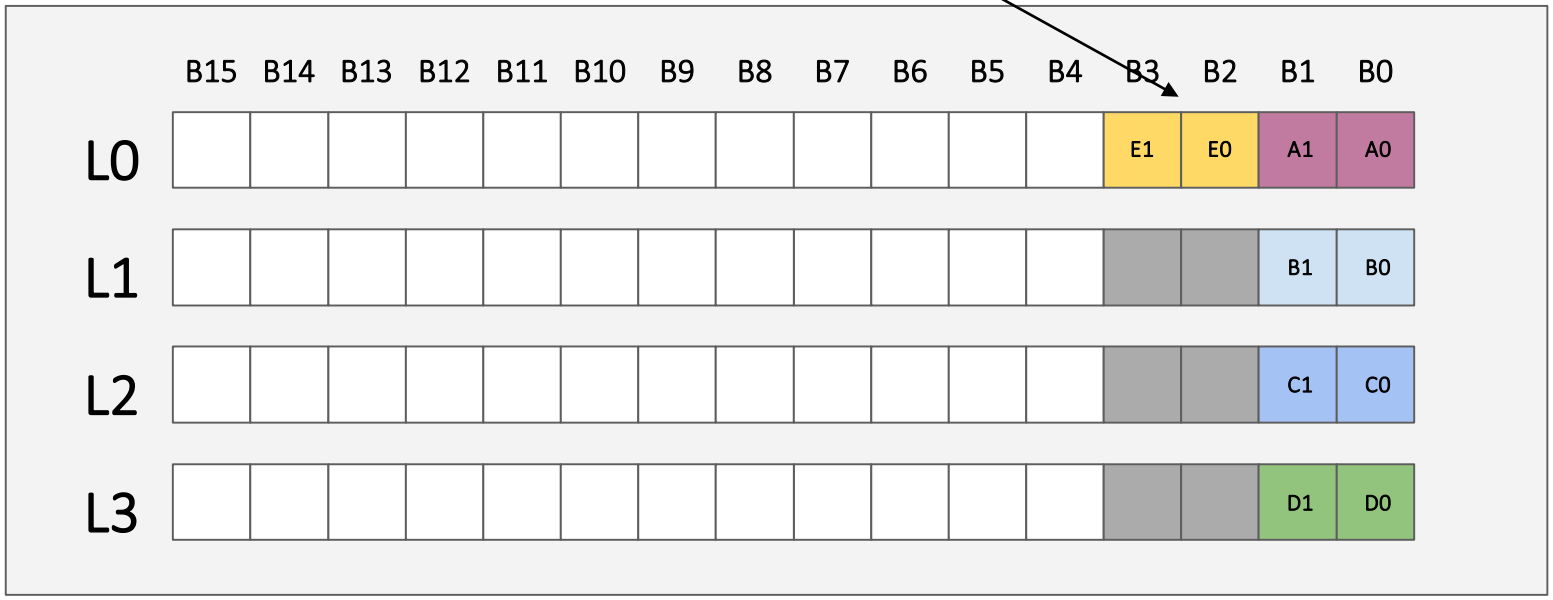
- Vector with 8 elements
- 2 Byte/element

```
…
vld v0, (Memory)
…
```

Memory (outside Ara)



v0 (VRF, inside Ara)

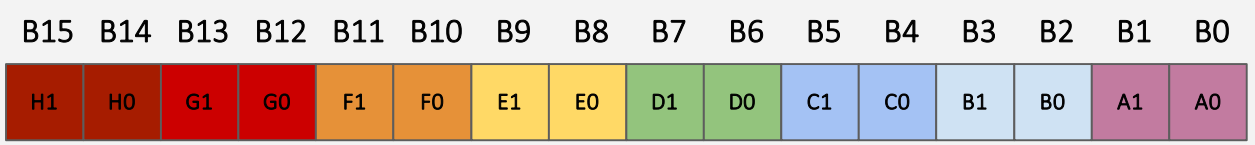


Ara VRF

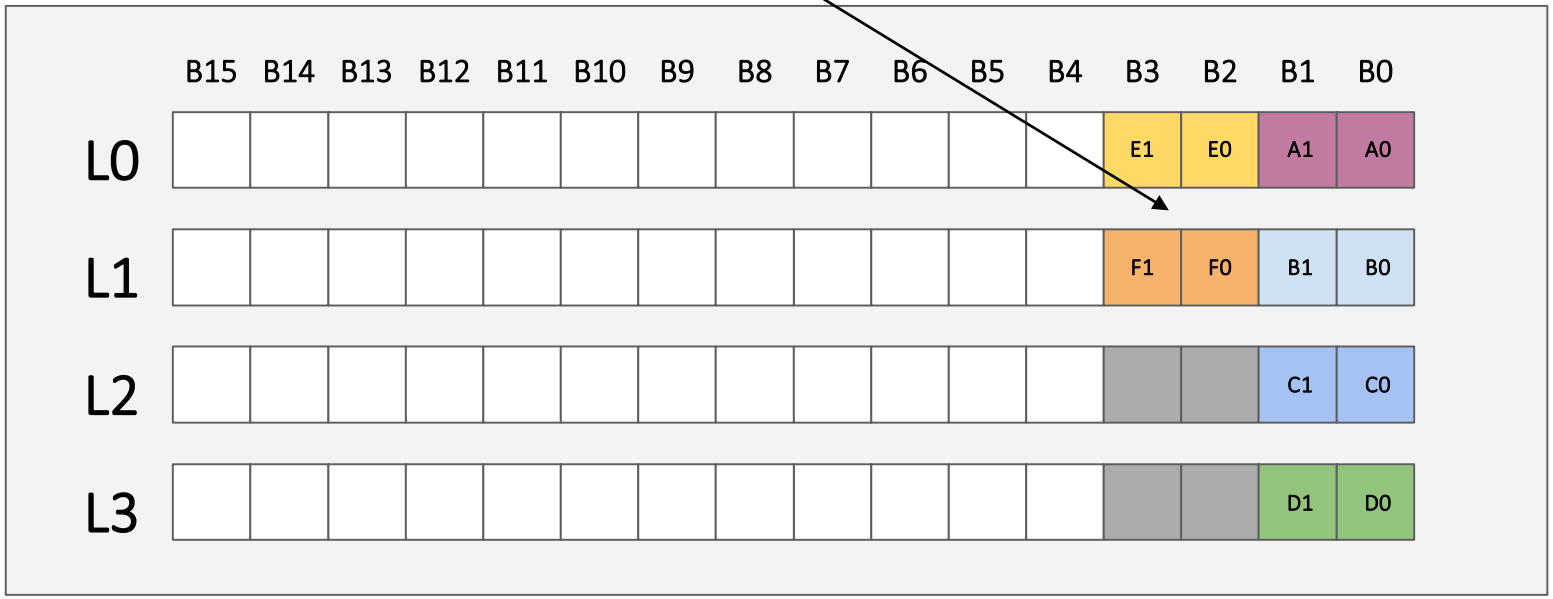
- Vector with 8 elements
- 2 Byte/element

```
…
vld v0, (Memory)
…
```

Memory (outside Ara)



v0 (VRF, inside Ara)

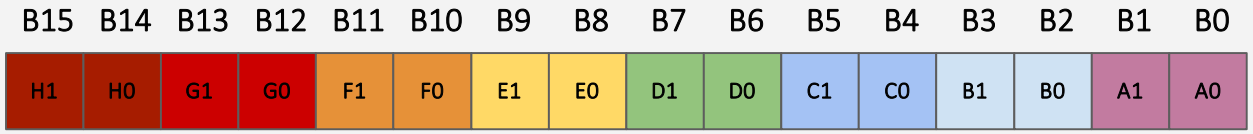


Ara VRF

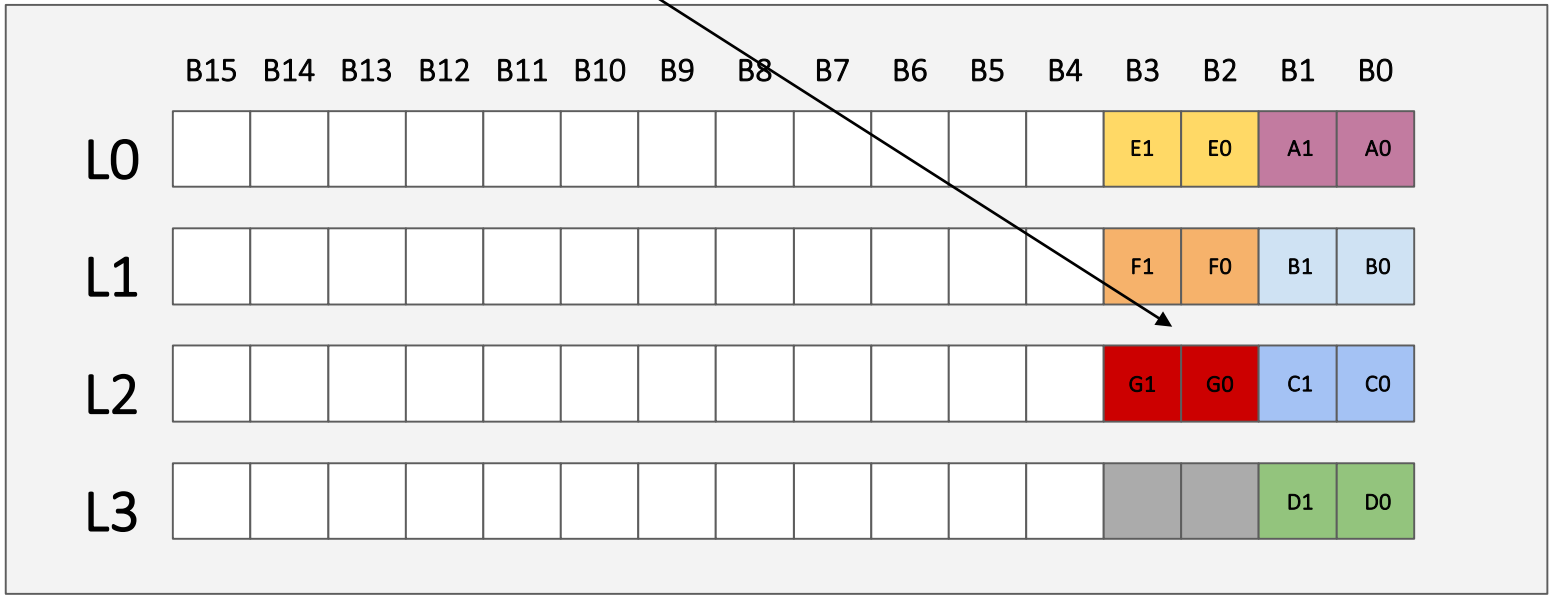
- Vector with 8 elements
- 2 Byte/element

```
…
vld v0, (Memory)
…
```

Memory (outside Ara)



v0 (VRF, inside Ara)

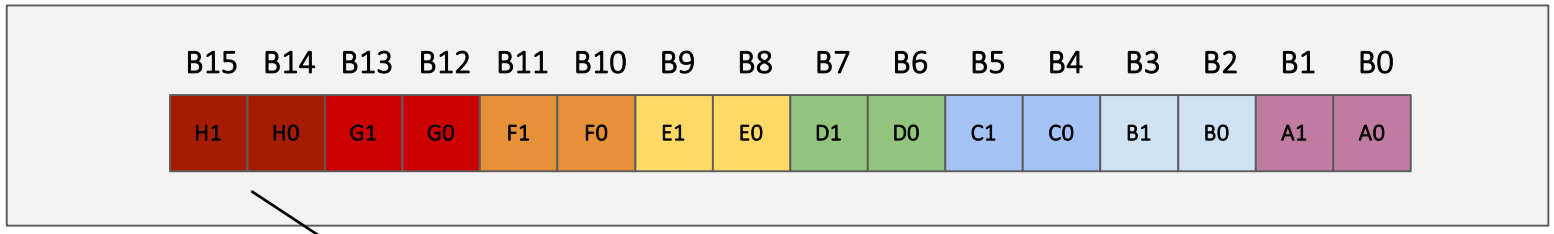


Ara VRF

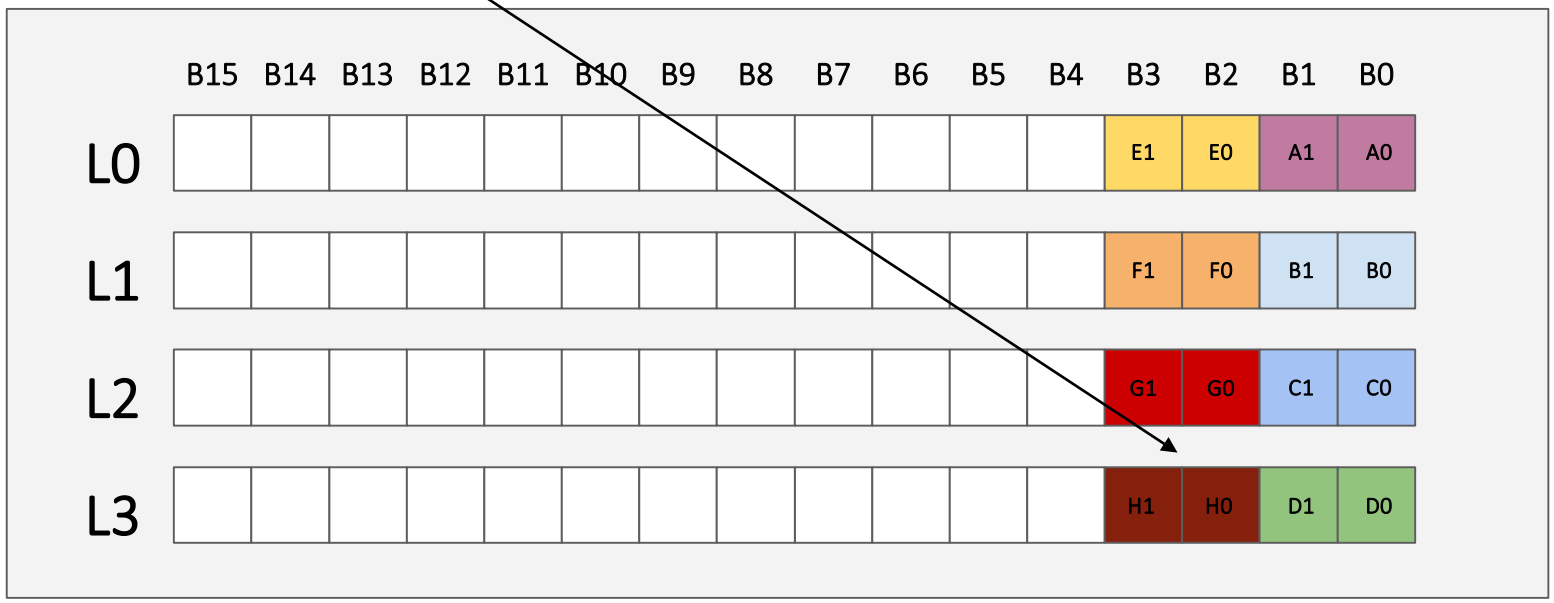
- Vector with 8 elements
- 2 Byte/element

```
…
vld v0, (Memory)
…
```

Memory (outside Ara)



v0 (VRF, inside Ara)



Ara VRF

- Vector with 8 elements
- 2 Byte/element

```
…
vld v0, (Memory)
…
```

Memory (outside Ara)

B15	B14	B13	B12	B11	B10	B9	B8	B7	B6	B5	B4	B3	B2	B1	B0
H1	H0	G1	G0	F1	F0	E1	E0	D1	D0	C1	C0	B1	B0	A1	A0

v0 (VRF, inside Ara)

	B15	B14	B13	B12	B11	B10	B9	B8	B7	B6	B5	B4	B3	B2	B1	B0
L0													E1	E0	A1	A0
L1													F1	F0	B1	B0
L2													G1	G0	C1	C0
L3													H1	H0	D1	D0

Ara VRF

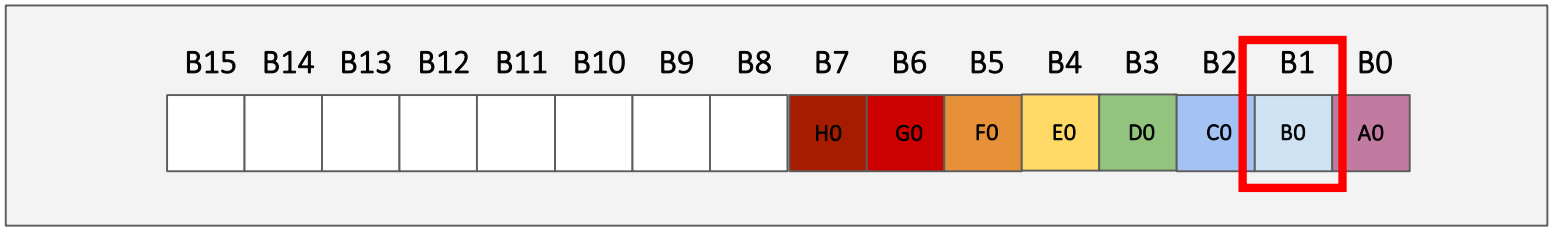
- Vector with 8 elements
- 1 Byte/element

```
…
vld v0, (Memory)
…
```

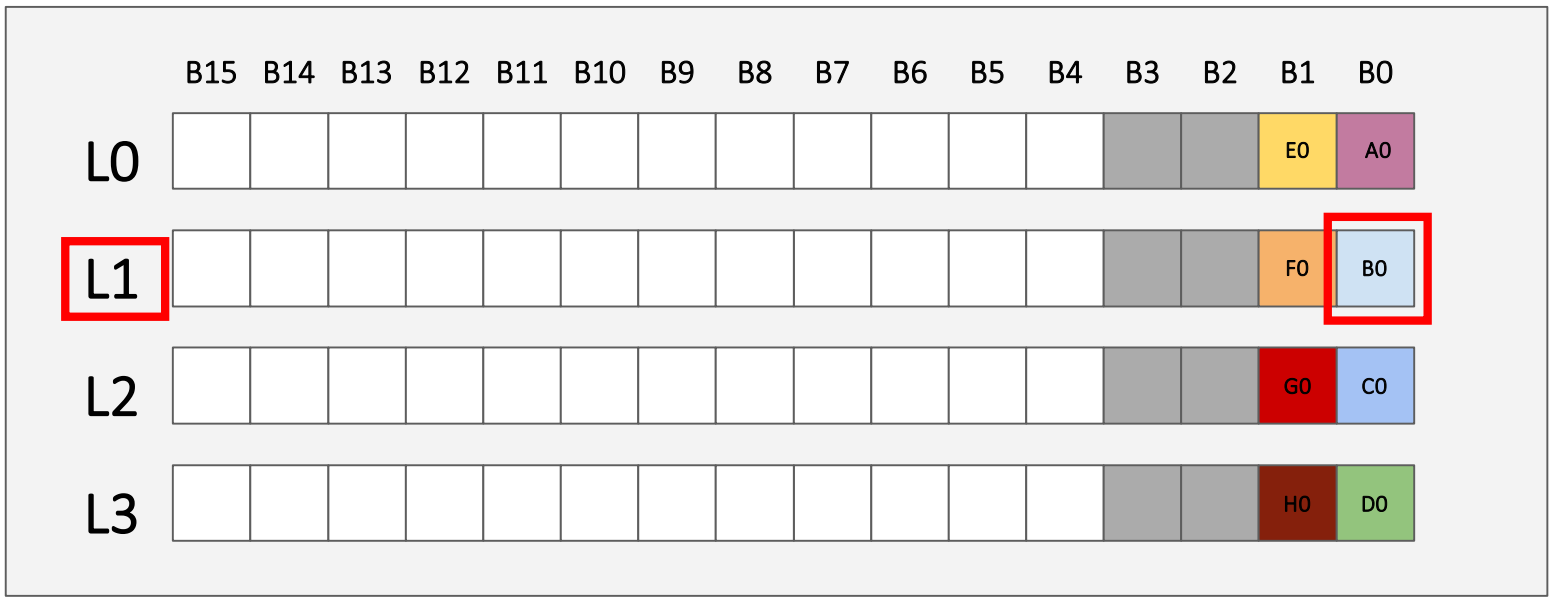
SAME BYTE IN MEMORY MAPPED TO DIFFERENT LANES

DEPENDING ON THE ELEMENT WIDTH

Memory (outside Ara)



v0 (VRF, inside Ara)



Reshuffle problems

In particular situations, Ara needs to change byte layout of its vector registers

- To **preserve tail elements**
  - When we don't overwrite the whole vector register
- For **consistent vector-vector in-lane** operations
  - Lanes can only access local data!
- When **loading/storing**
  - This is done on-the-fly in the VLSU

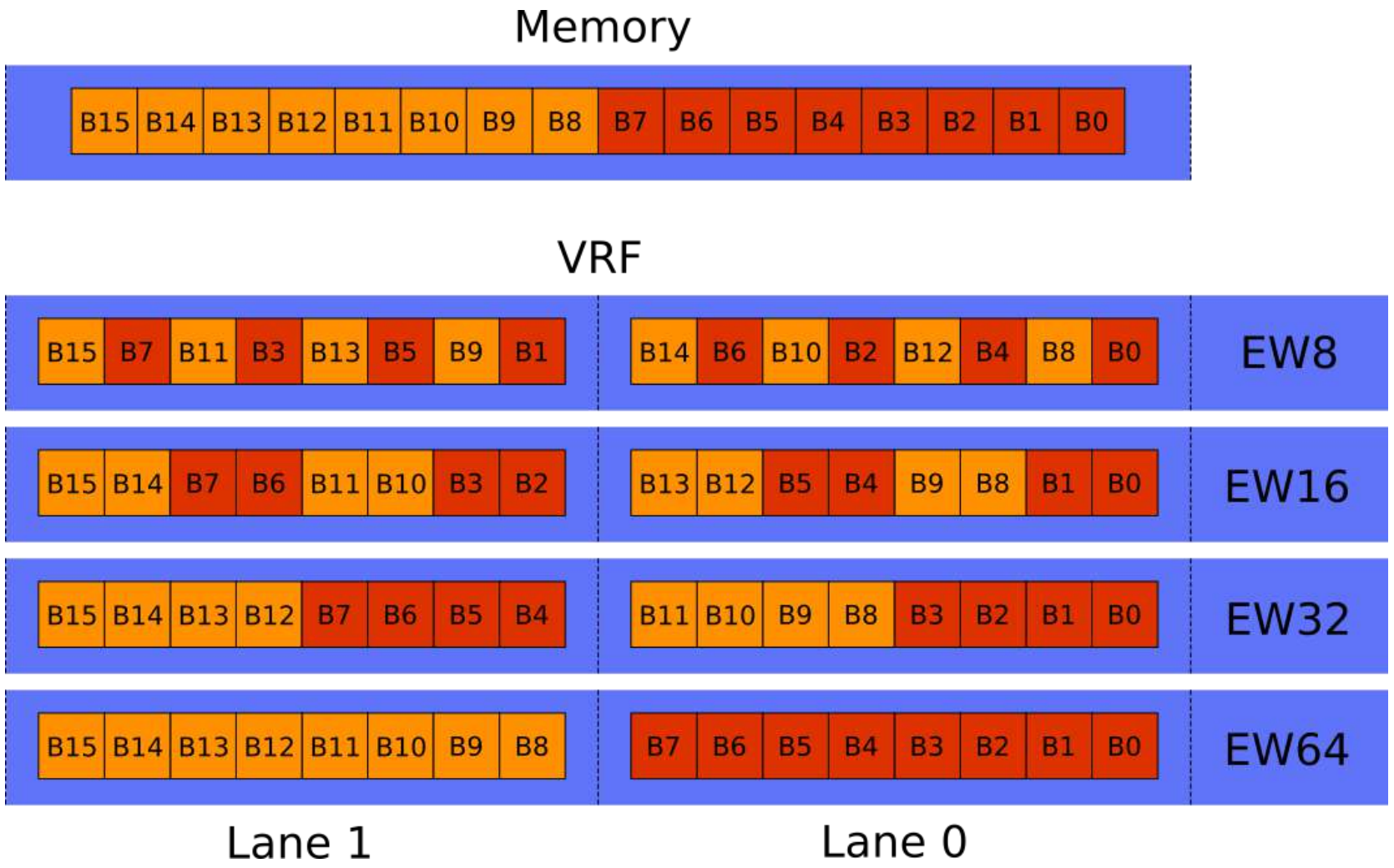
Byte-layout changes (reshuffles) are done by the Slide Unit

The reshuffles are injected by Ara's dispatcher

Hands-on – Reshuffle

1. Follow instructions of the main README.md → *Arathon SPECIAL* paragraph
    1. ex_name=RESHUFFLE
    2. Try to trigger a reshuffle in Ara
    3. For example, initialize 16-bit vectors, but load one of them with vle8 and 2*vl. Then add them together.
    4. Does the reshuffle impact performance?
    5. Inspect apps/bin/arathon.dump
    6. Simulate with waveforms or printf!

Real byte layout mapping



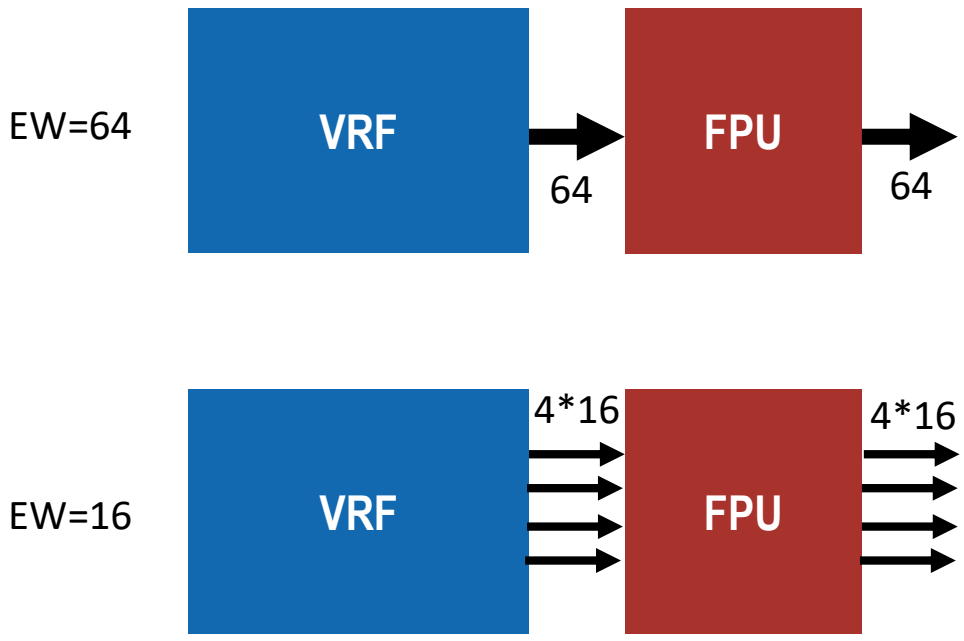
Hands-on – VRF

1. Follow instructions of the main README.md → *Arathon SPECIAL* paragraph
    1. ex_name=VRF
    2. Inspect apps/bin/arathon.dump
    3. Simulate with waveforms or printf!


P
P

SIMD computation

- Every lane's functional unit operates on a **64-bit datapath**
- Every functional unit works in SIMD fashion
  - **One 64-bit element/cycle**
  - Two 32-bit elements/cycle
  - **Four 16-bit elements/cycle**
  - Eight 8-bit elements/cycle



Hands-on – SIMD

1. Follow instructions of the main README.md → *Arathon SPECIAL* paragraph
    1. ex_name=SIMD
    2. Inspect apps/bin/arathon.dump
    3. Simulate with waveforms or printf!


P
P

Main dependency checks and scoreboard

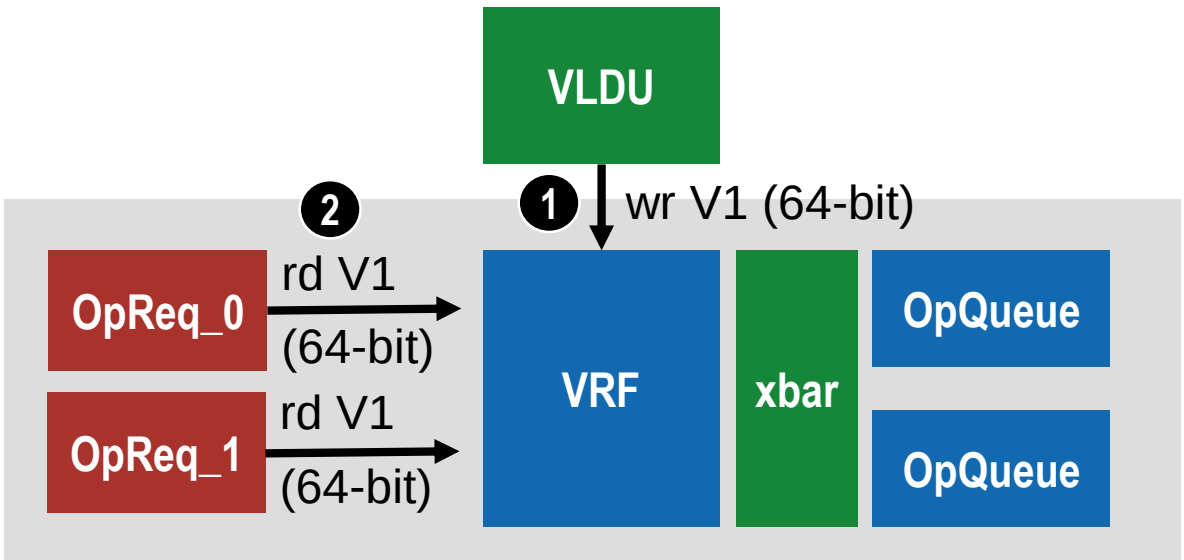
- Scoreboard in the main sequencer → Which instruction depends on which one
- Every instruction is broadcasted to Ara's backend
  - Attach hazard metadata for each src/dst vreg → This vreg has a dep. on which instruction?
- **All hazards** (RAW, WAR, WAW) enforce a **throttling at the source operand fetch** of the dependent instruction!
  - If source operands are not fetched, results cannot be generated either!
- Chaining is implemented by controlling source operand fetch
- **Cannot chain specific instructions**: some slides and instructions without source operands!

In-lane chaining

- Operands are fetched from in-lane VRF slices
- An instruction can fetch from a vector register with a hazard only if dependent instruction has written the same register at least once (1-credit-based system)

vle v1, addr

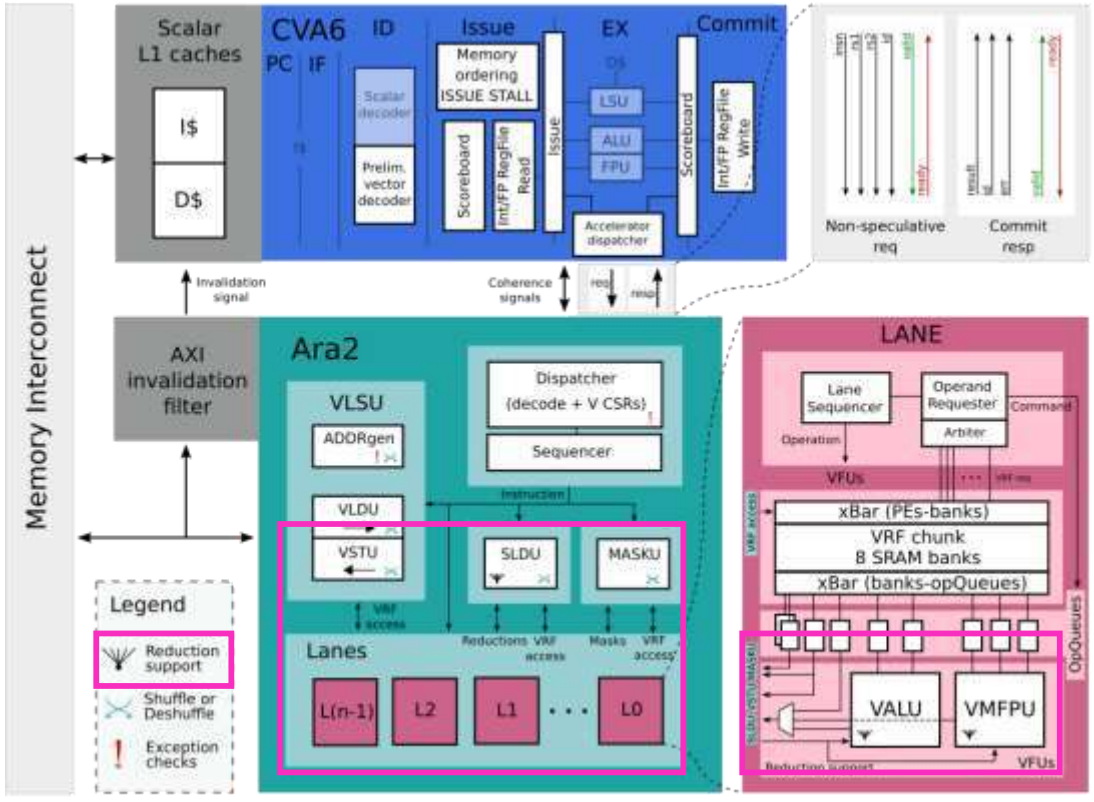
vadd v2, v1, v1



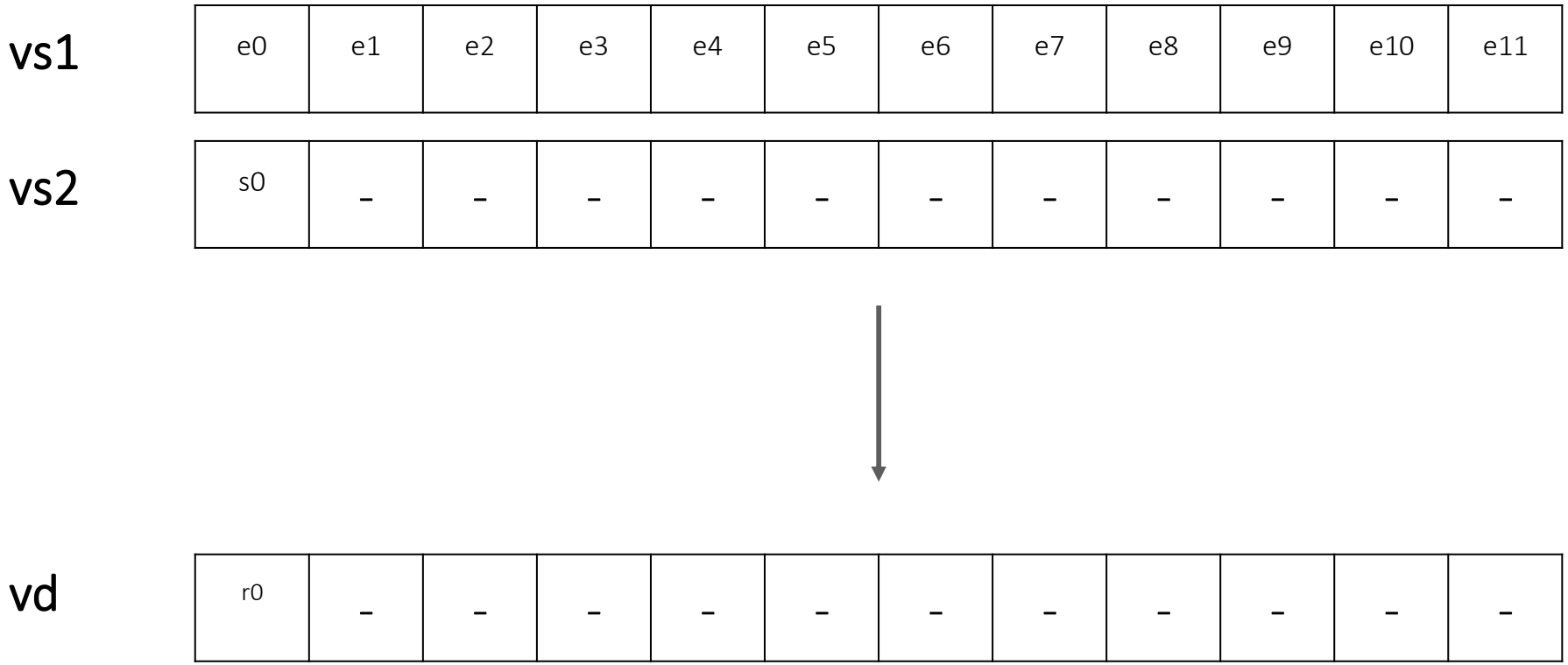

P
P

Vector Reductions

- Horizontal operation on a vector!
- Requires byte exchange among lanes

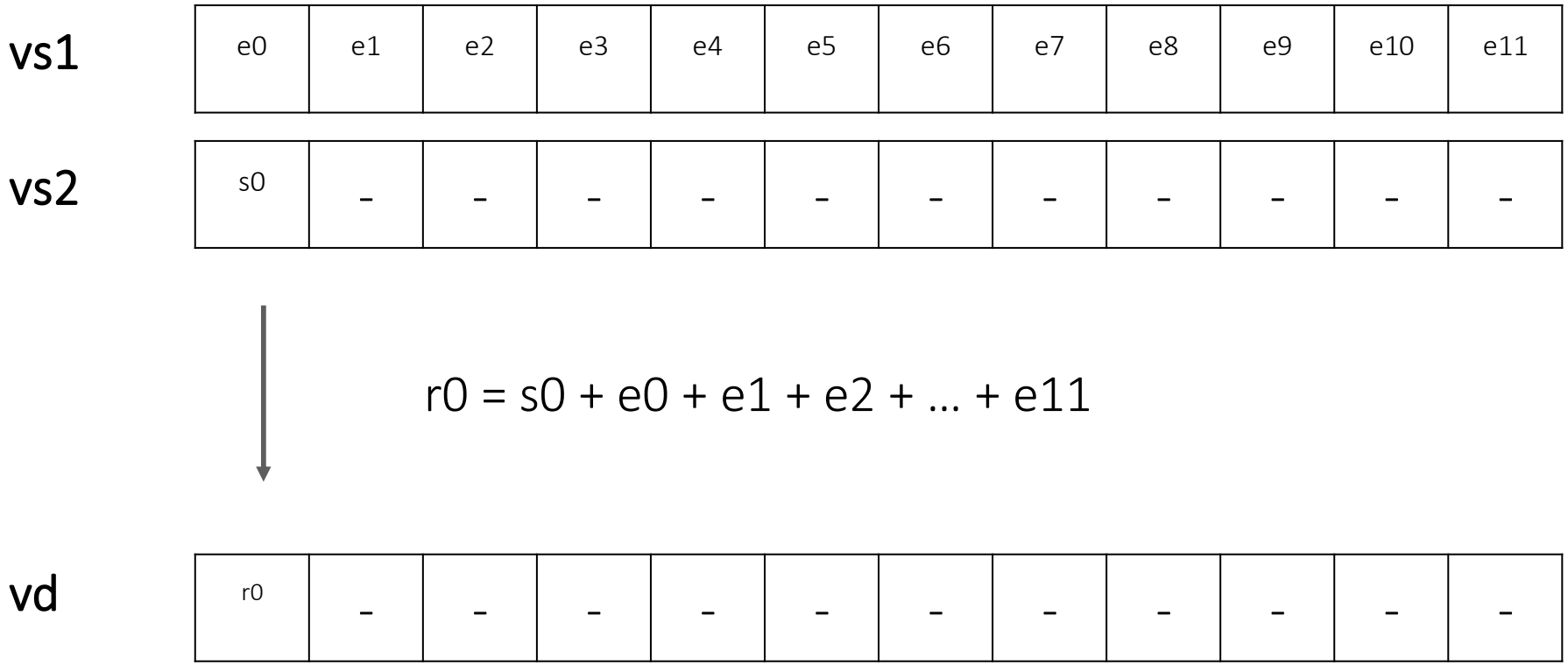


Vector Reductions

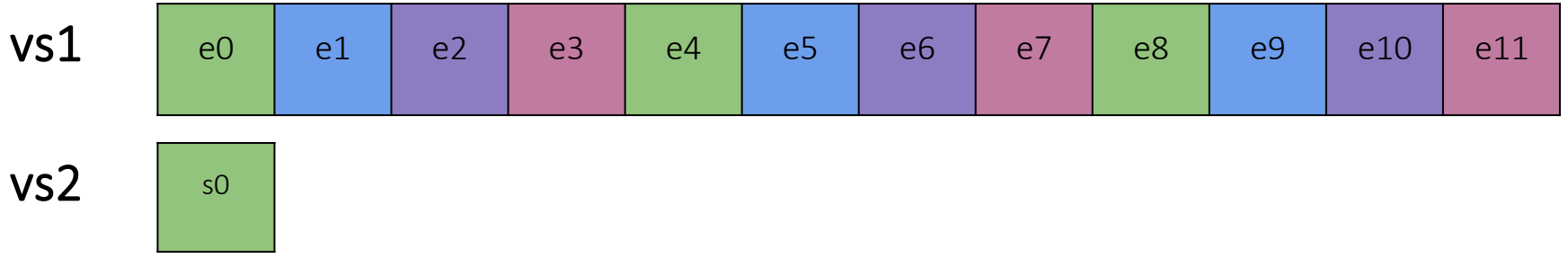


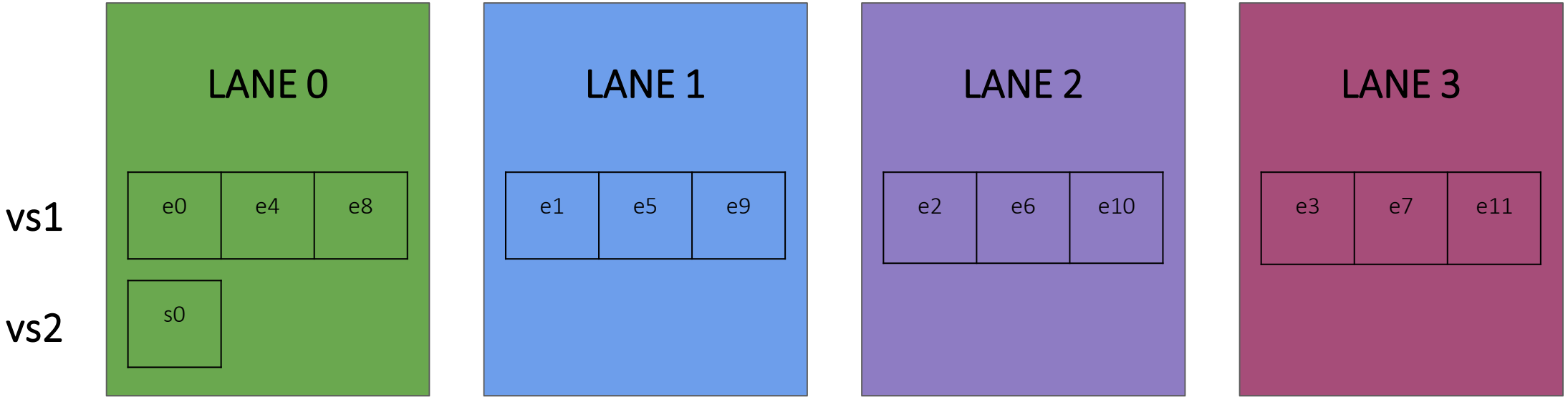
Vector Reductions

Example: vredsum vd, vs1, vs2

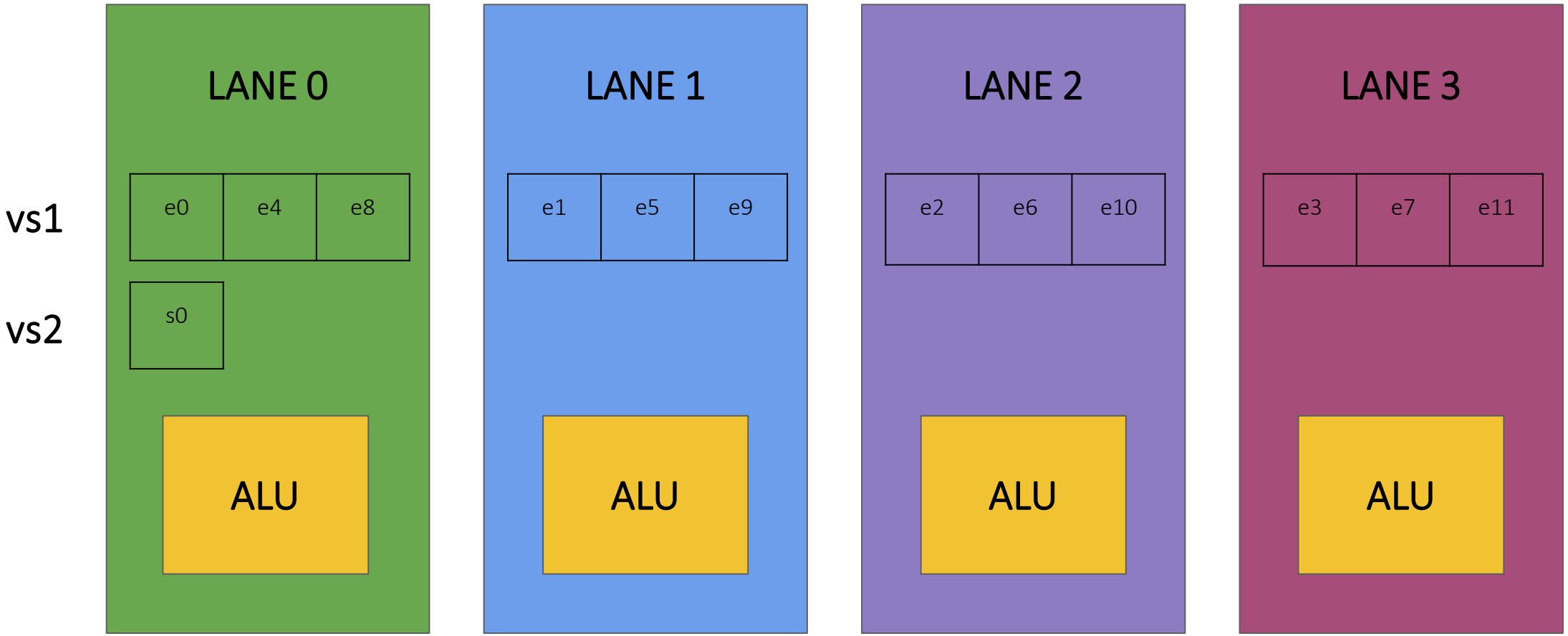


Vector Reductions – Intra-lane phase start





Vector Reductions



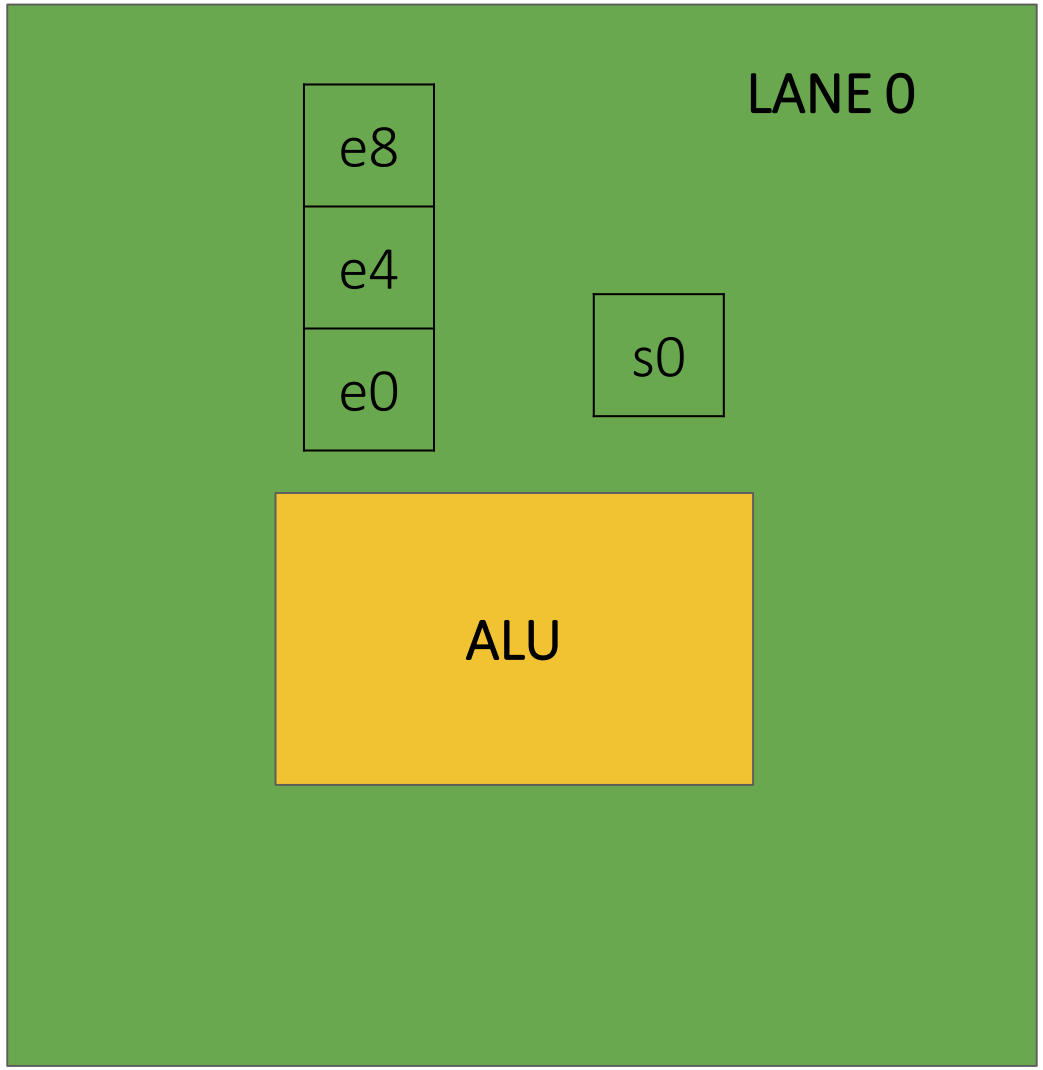
temp0 = s0 + e0 + e4 + e8

temp1 = e1 + e5 + e9

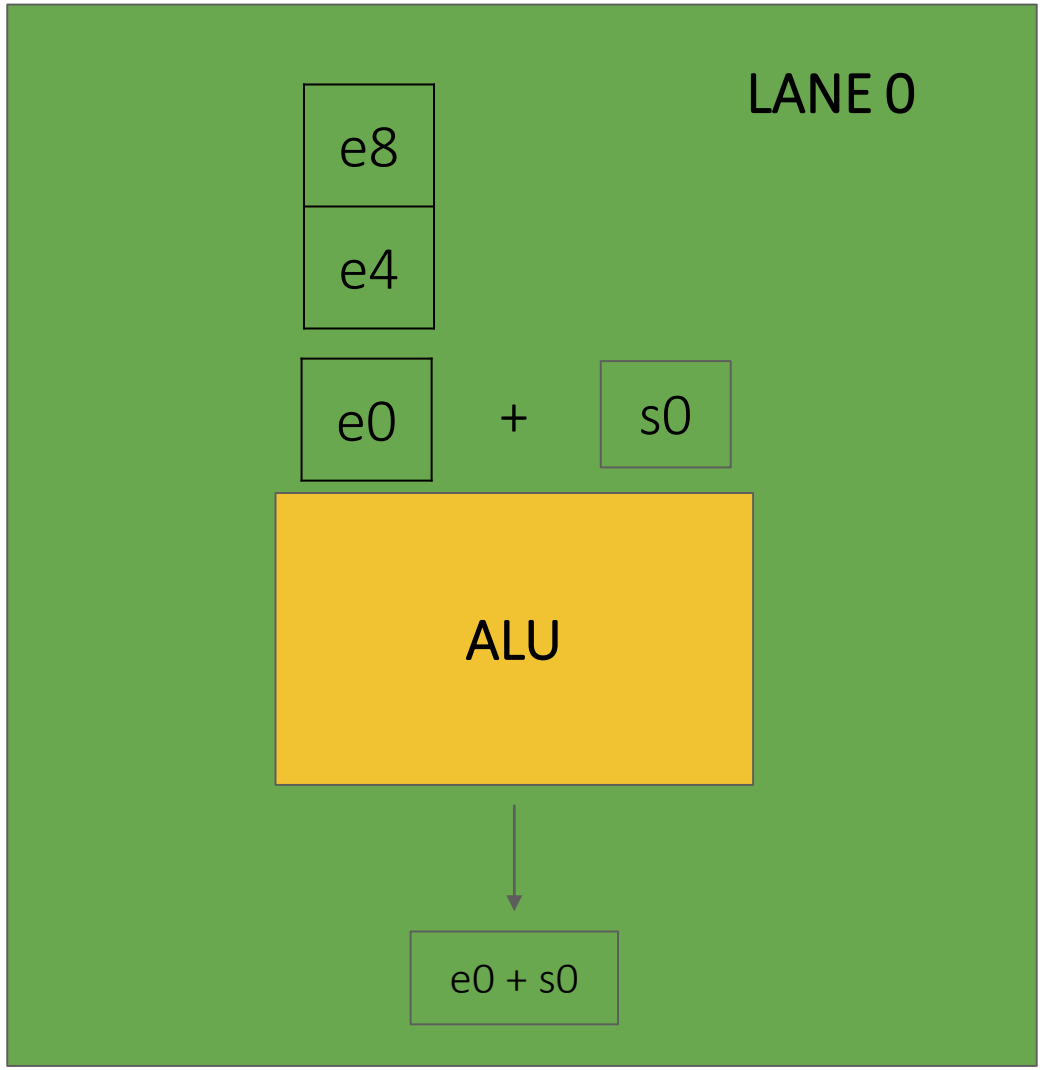
temp2 = e2 + e6 + e10

temp3 = e3 + e7 + e11

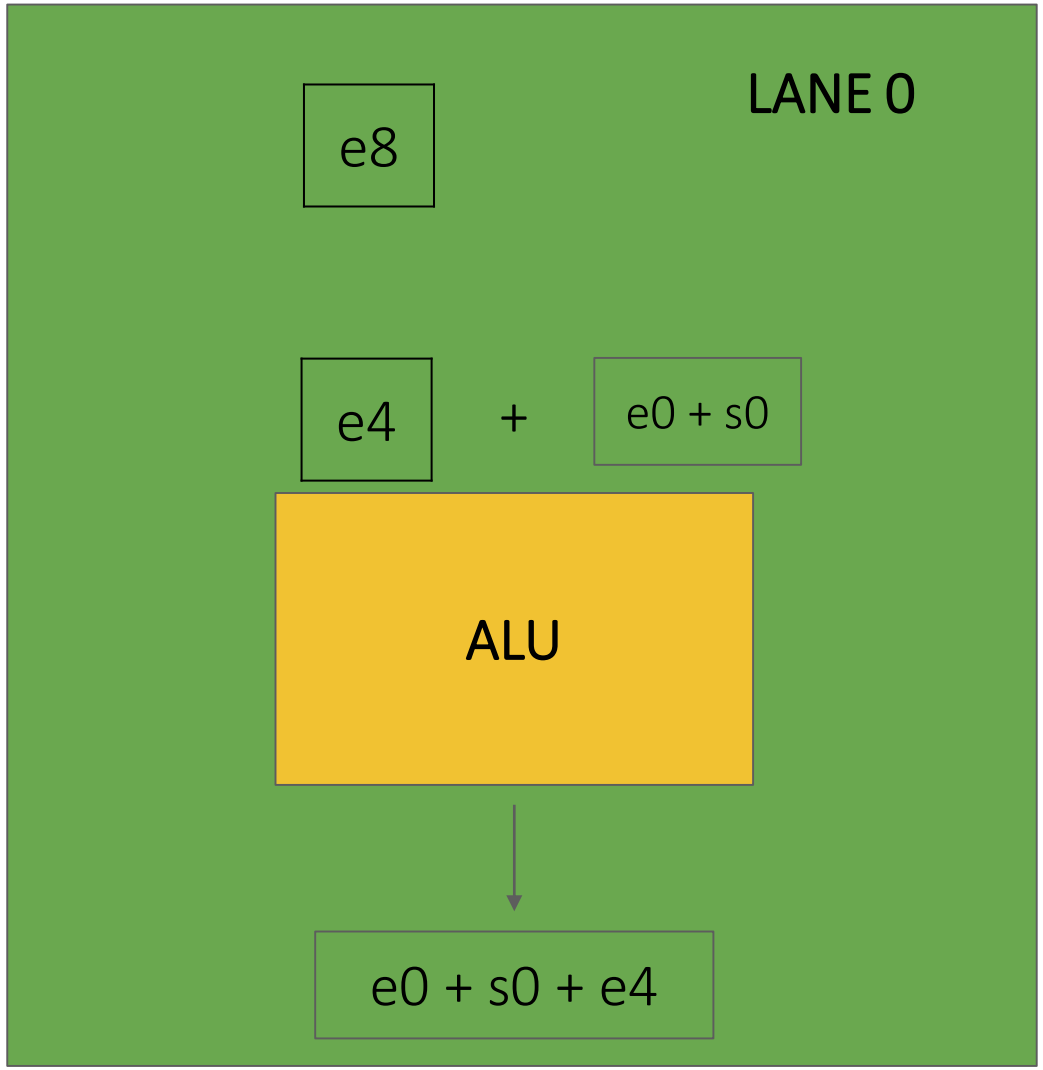
Vector Reductions



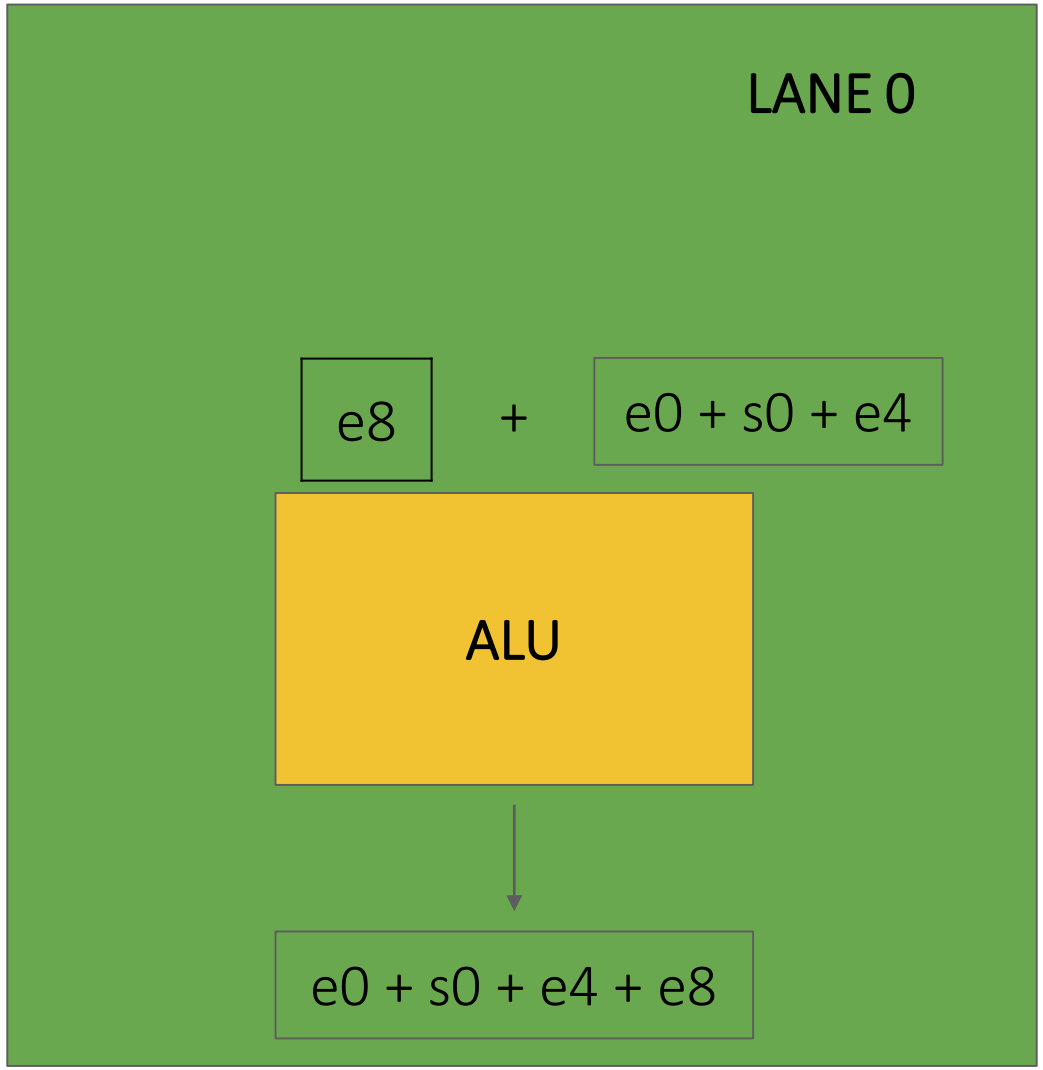
Vector Reductions



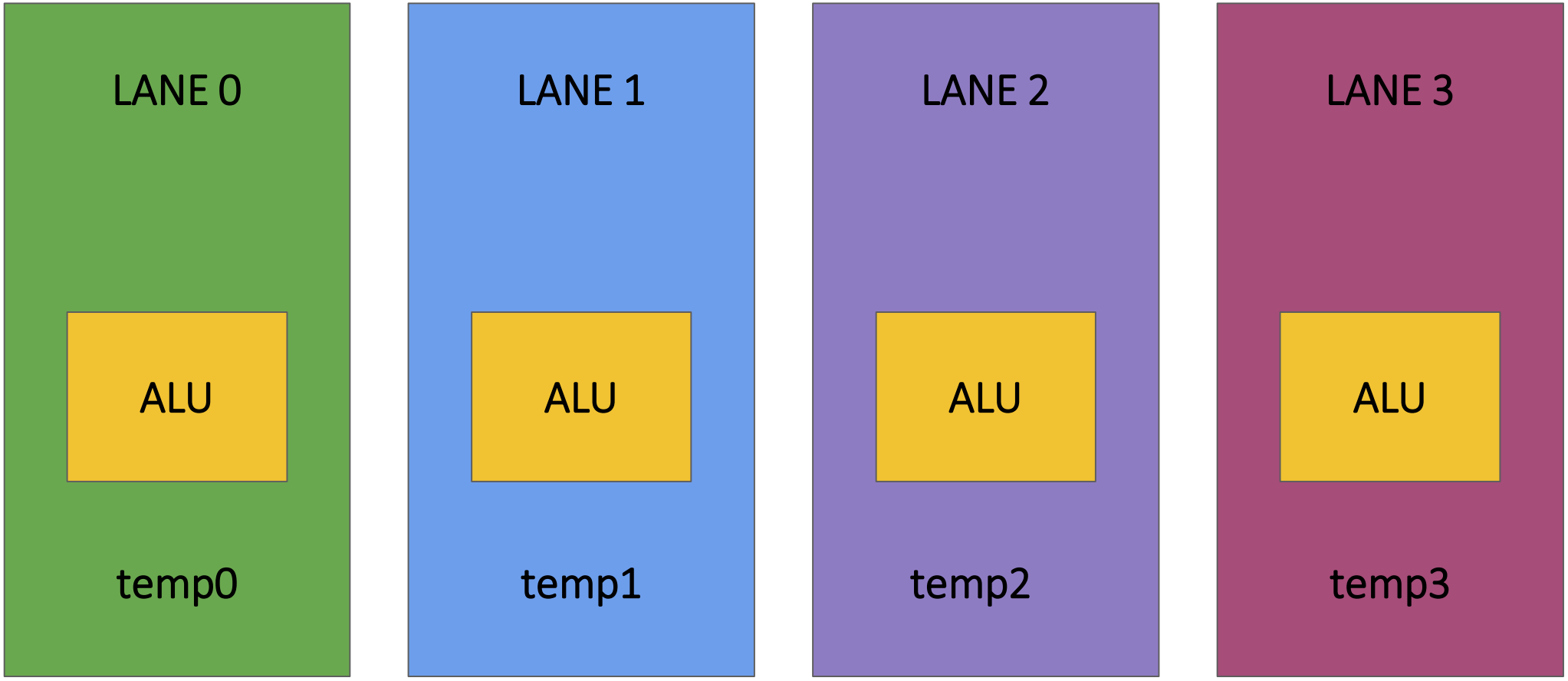
Vector Reductions



Vector Reductions



Vector Reductions – Intra-Lane phase end



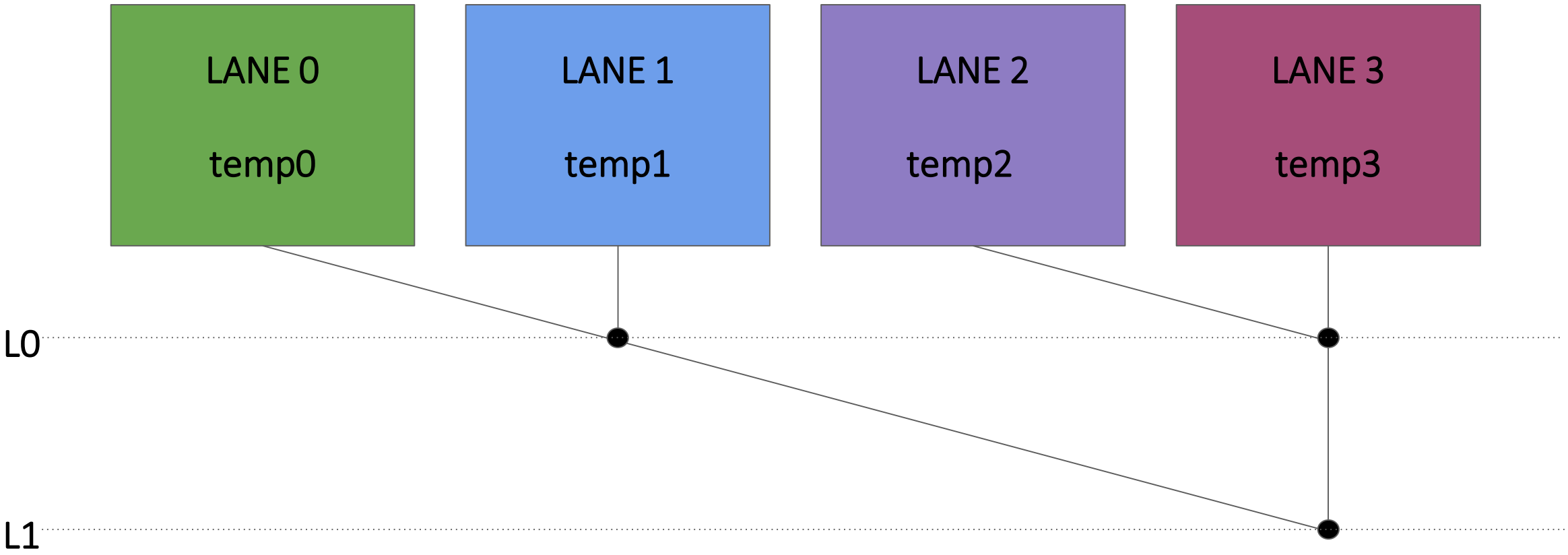
temp0 = s0 + e0 + e4 + e8

temp1 = e1 + e5 + e9

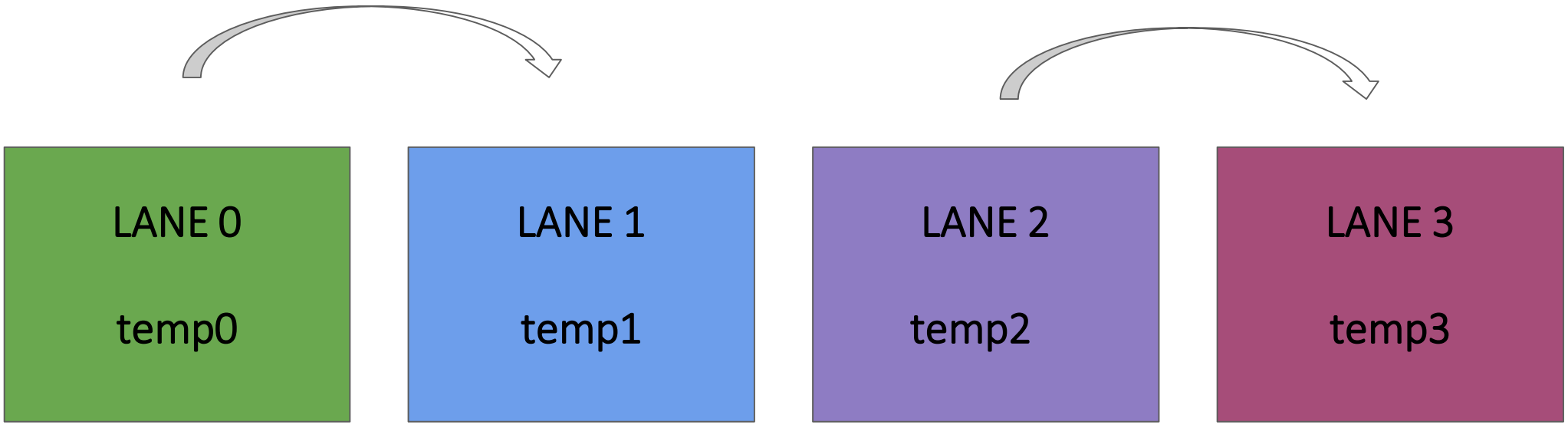
temp2 = e2 + e6 + e10

temp3 = e3 + e7 + e11

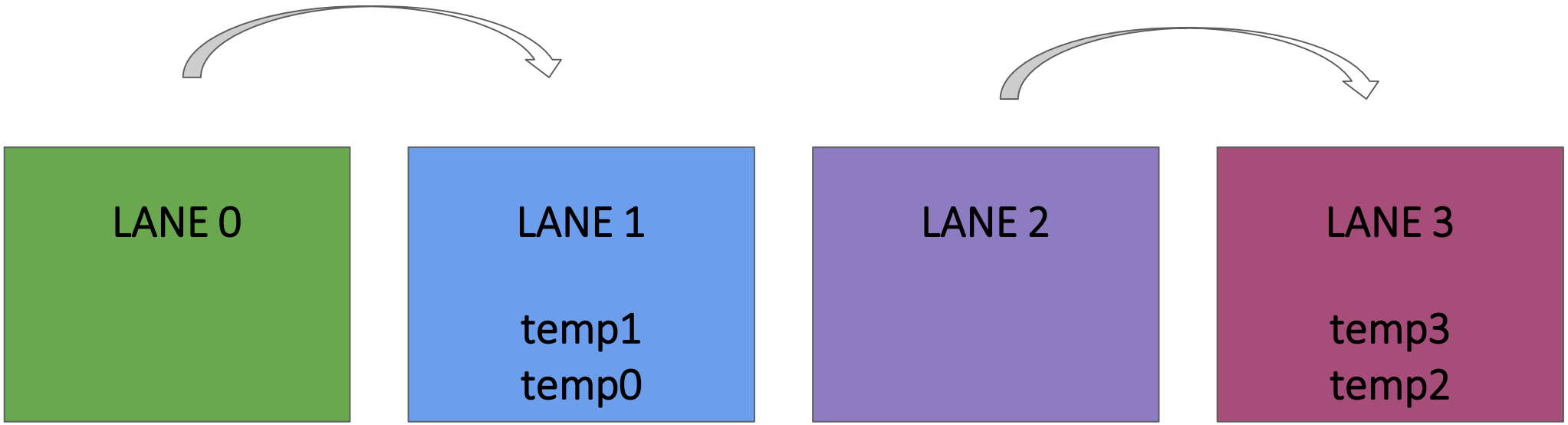
Vector Reductions – Inter-Lane phase start



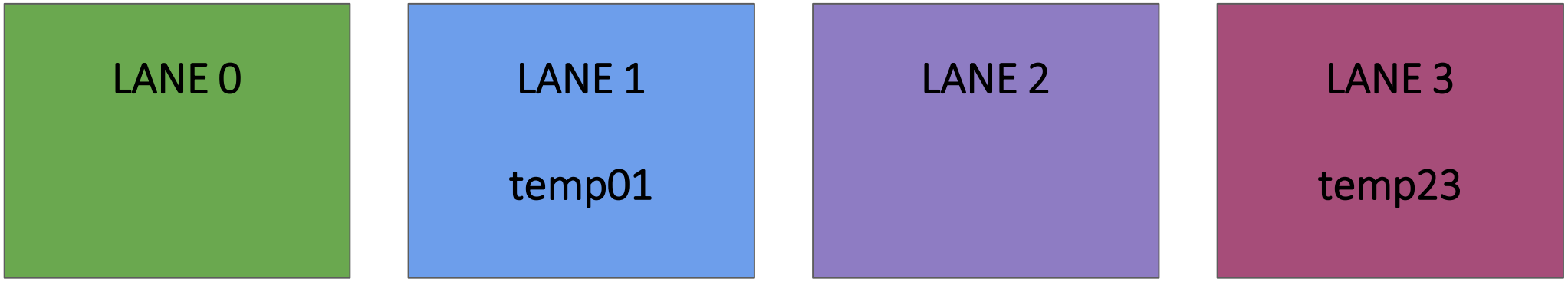
Vector Reductions



Vector Reductions



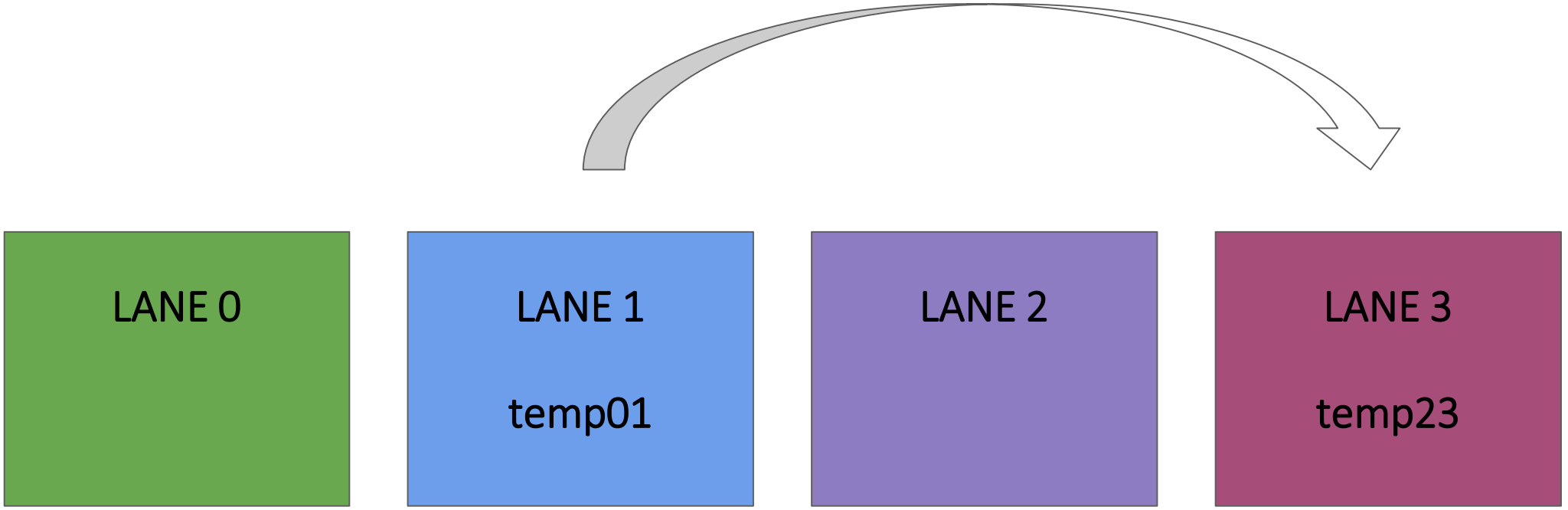
Vector Reductions



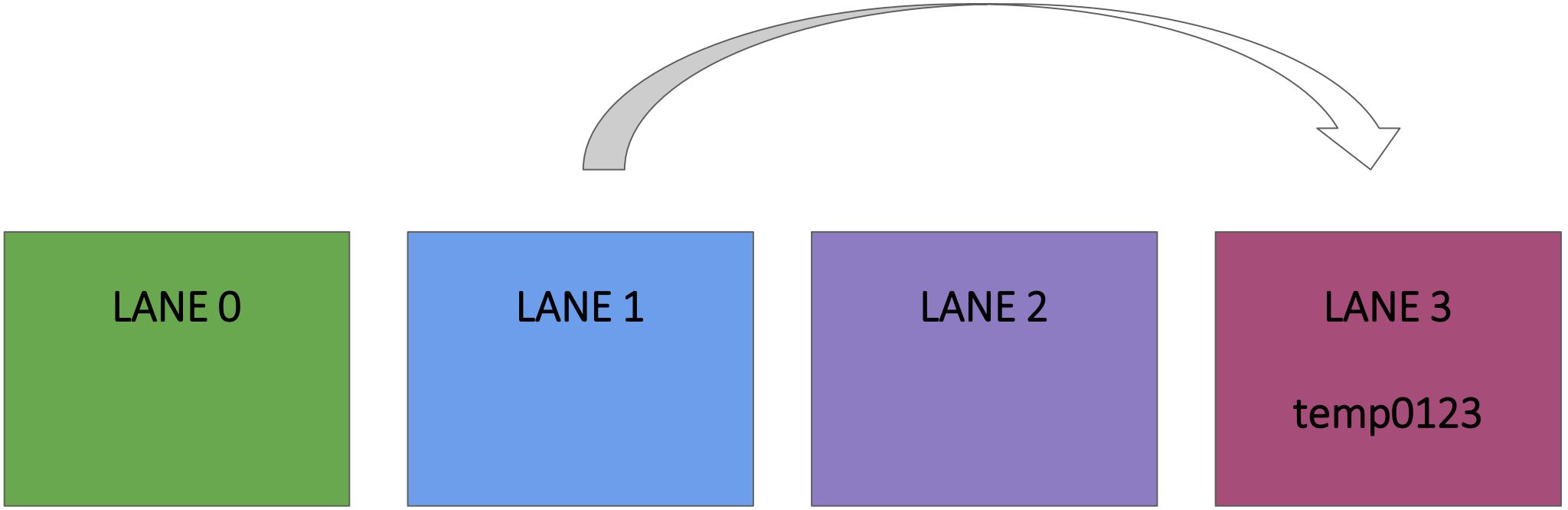
temp01 = temp0 + temp1

temp23 = temp2 + temp3

Vector Reductions

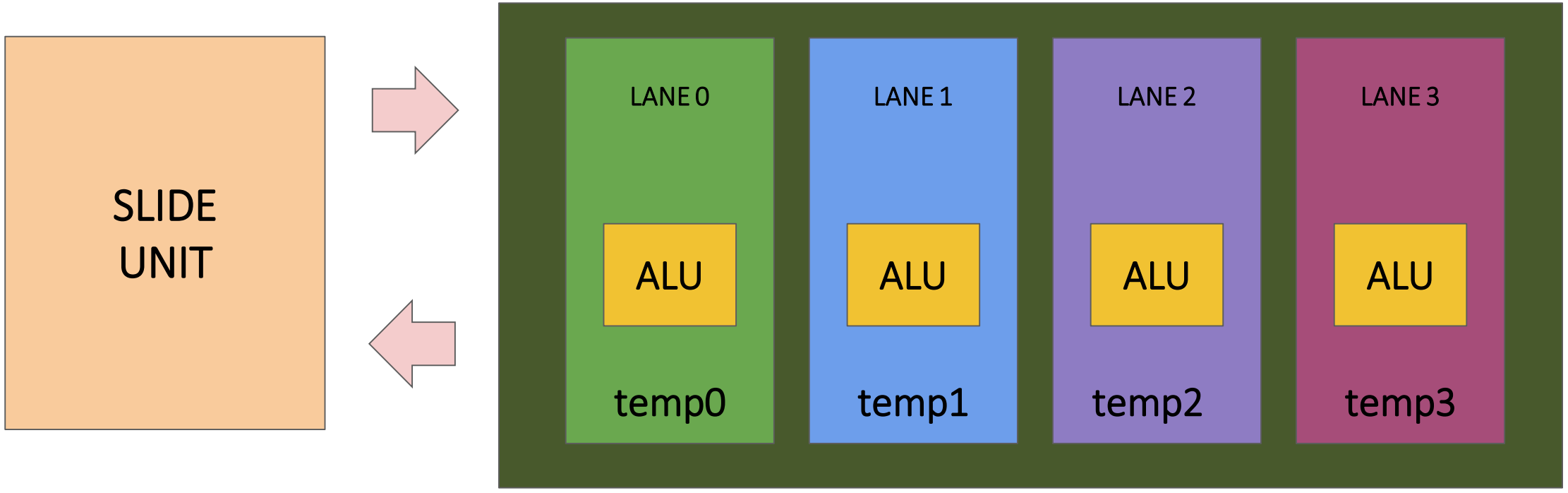


Vector Reductions

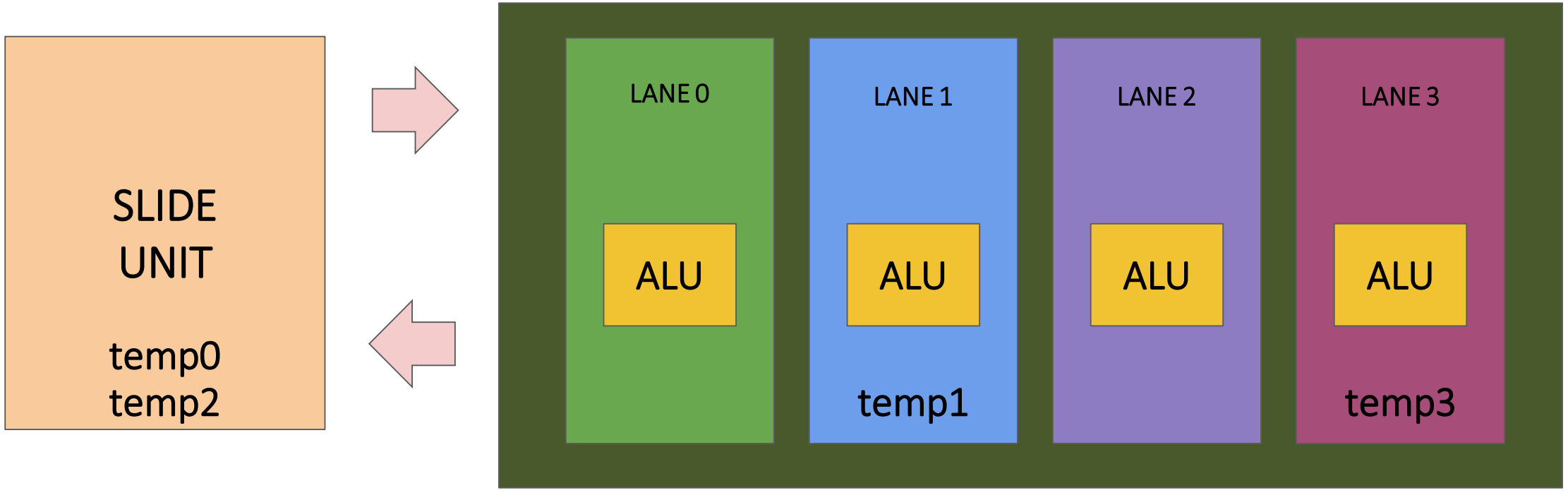


temp0123 = temp01 + temp23

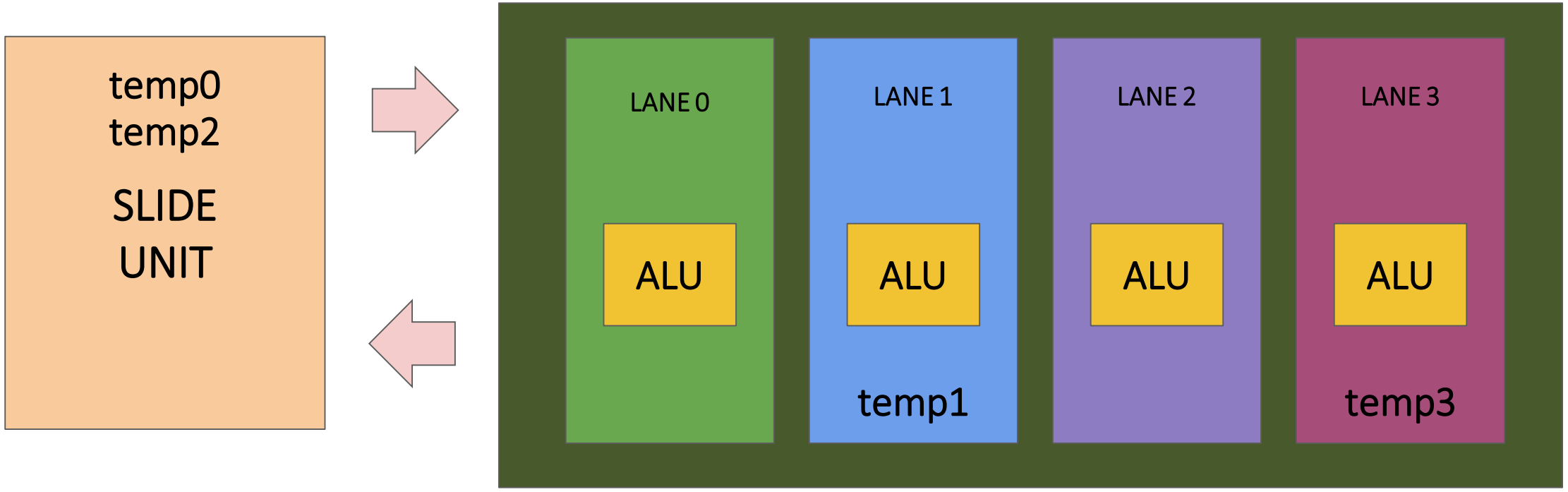
Vector Reductions



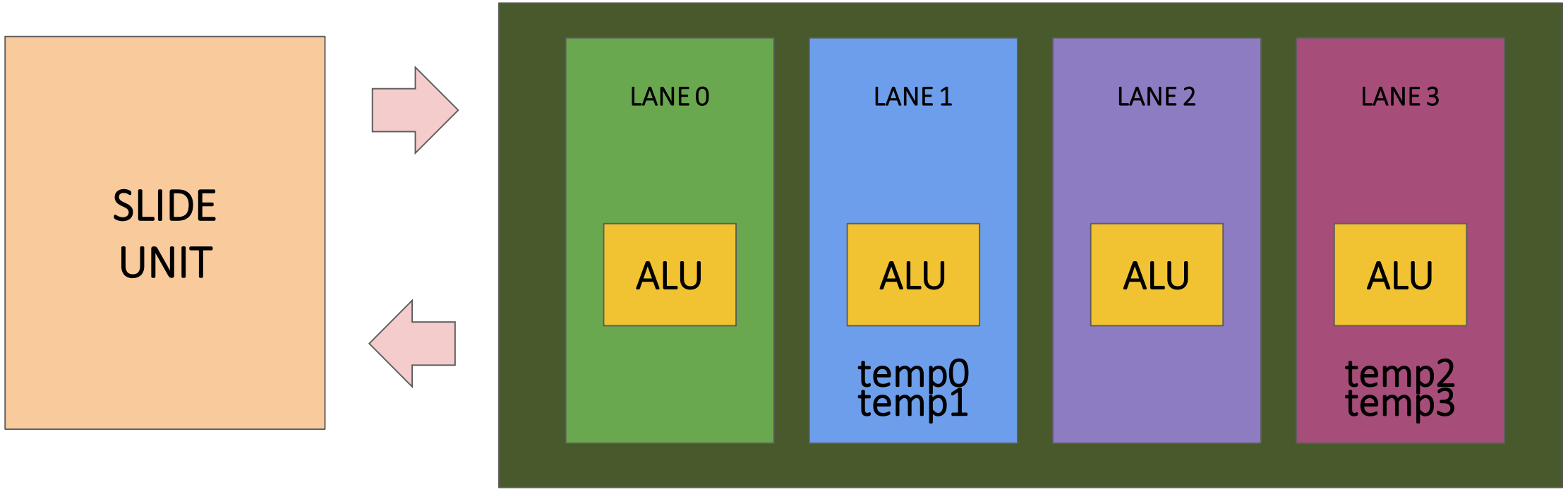
Vector Reductions



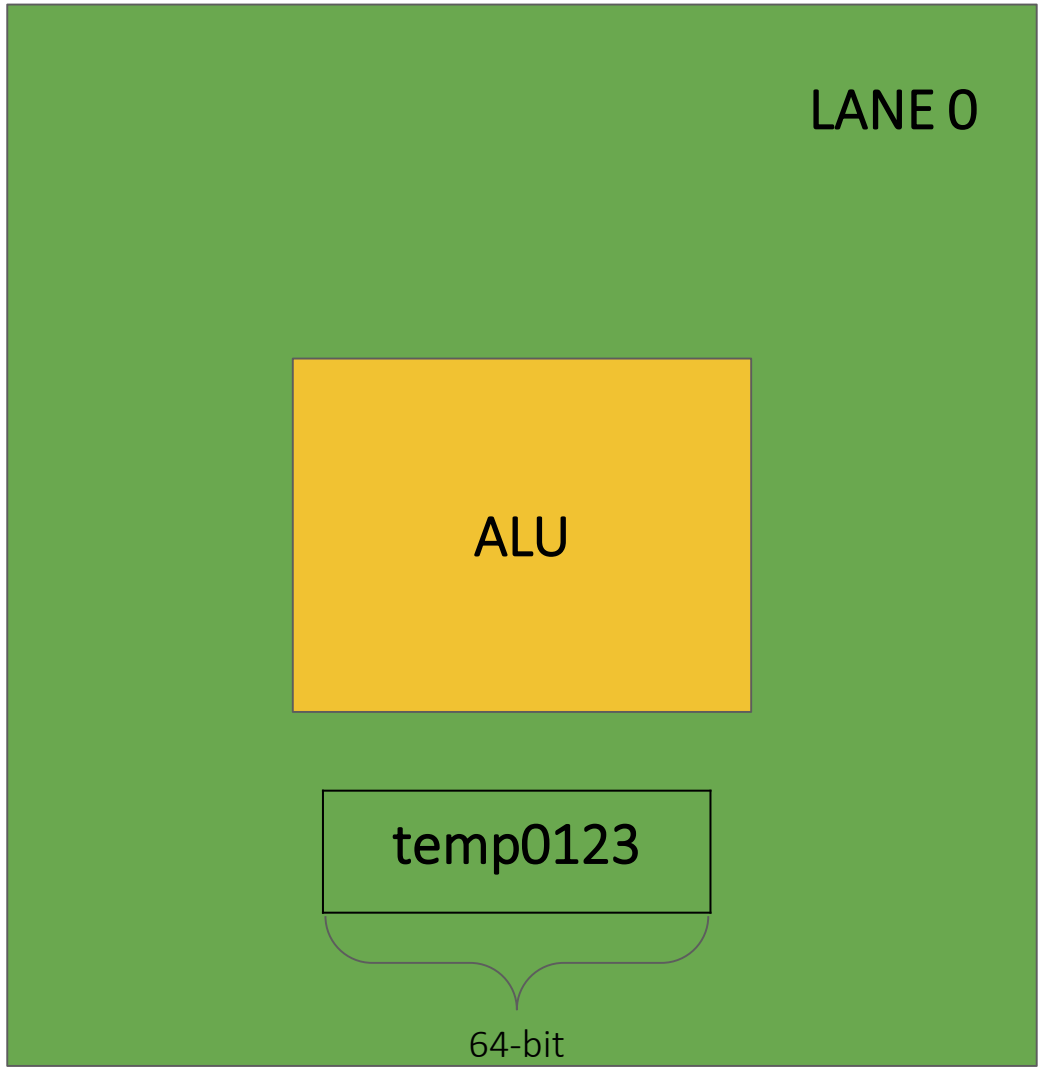
Vector Reductions



Vector Reductions – Inter-Lane phase end

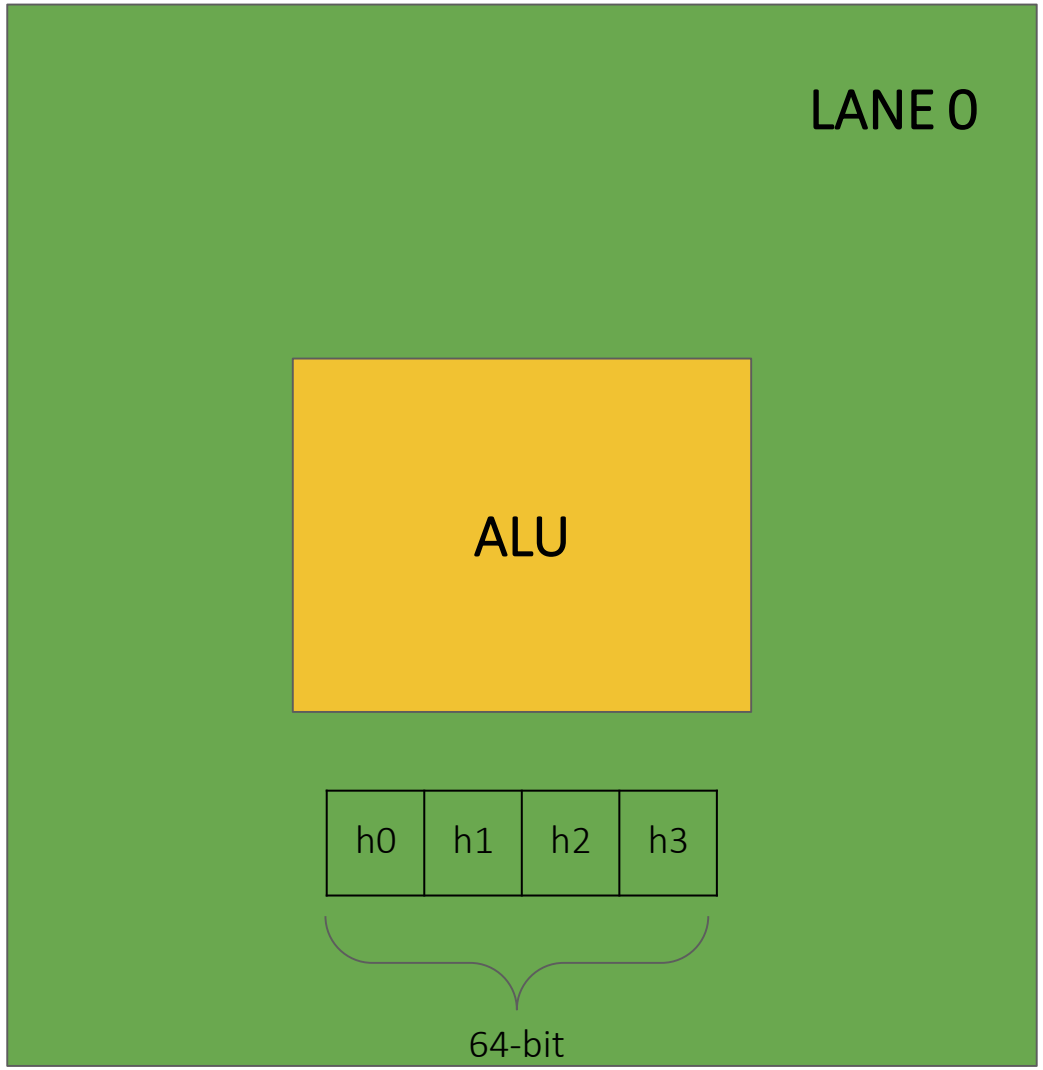


Vector Reductions – SIMD phase start



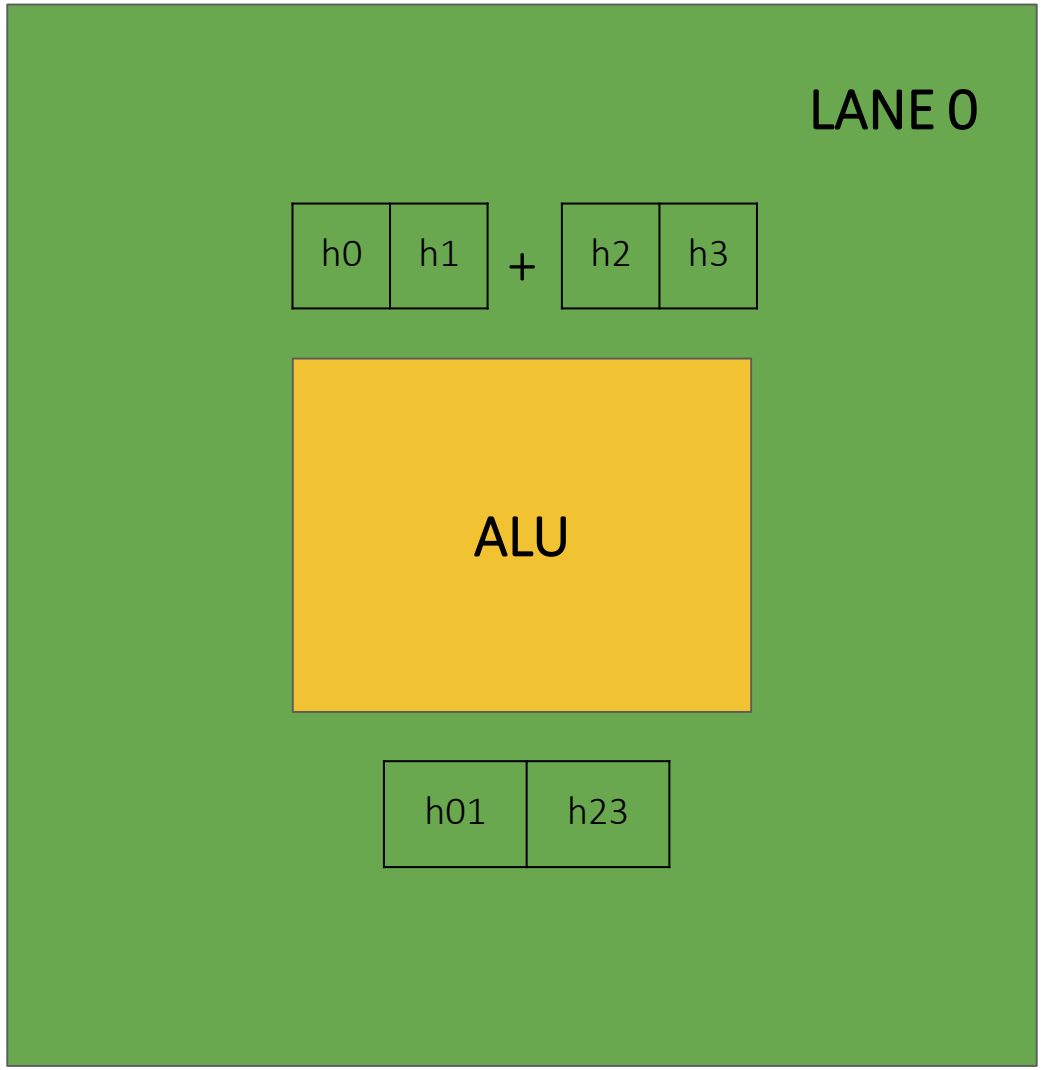
Vector Reductions

- Reduce the 64-bit packet of N elements
- Example: four 16-bit elements
- log2(N) operations for N sub-elements
- Example: log2(4) = 2 operations



Vector Reductions

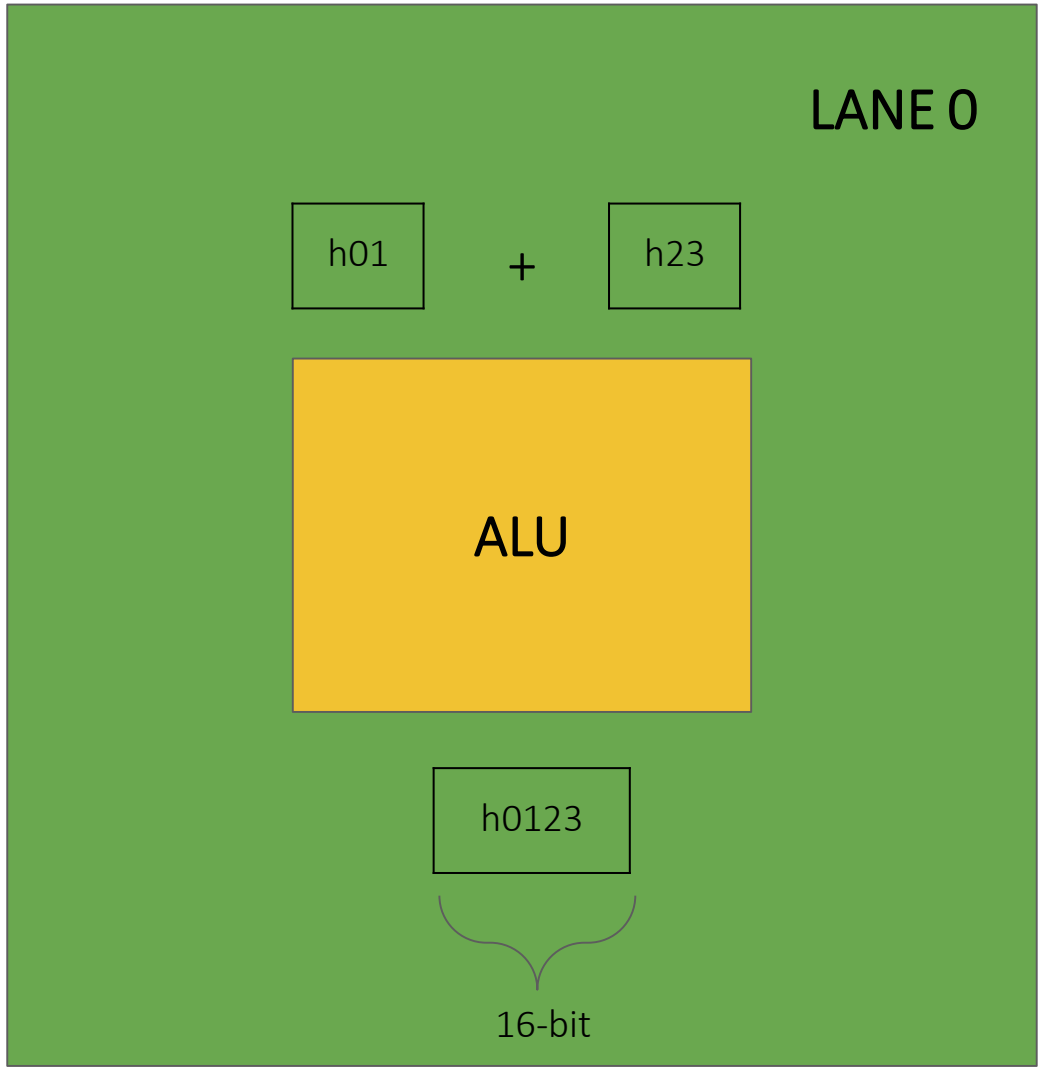
- Reduce the 64-bit packet of N elements
- Example: four 16-bit elements
- log2(N) operations for N sub-elements
- Example: log2(4) = 2 operations



First operation

Vector Reductions – SIMD phase end

- Reduce the 64-bit packet of N elements
- Example: four 16-bit elements
- log2(N) operations for N sub-elements
- Example: log2(4) = 2 operations



Second operation

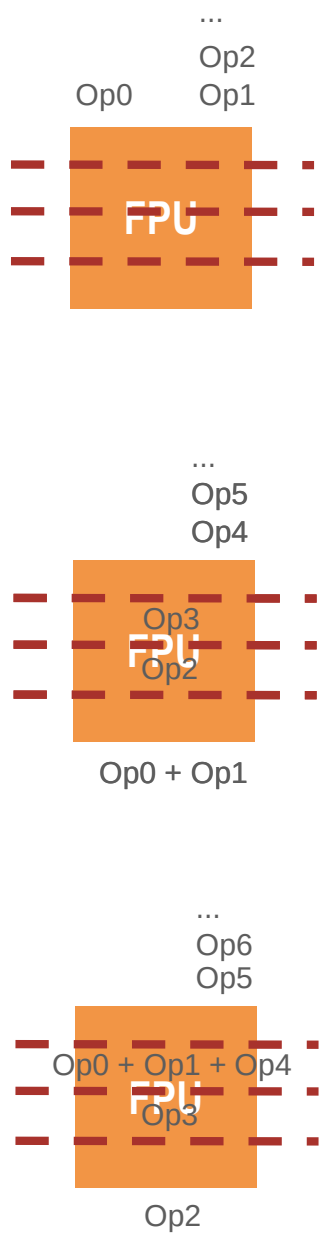
Floating-point reductions?

- FPUs are pipelined! Pipeline levels (R) depend on SEW
- Wait R cycles before a new operation?
- Trick: use pipeline registers to store partial accumulators
- In the end: final round of accumulation
- Intra-Lane reduction latency

$$\frac{N}{L} + R \times (1 + \log_2(\lceil R \rceil)) - (\lceil R \rceil - R)$$

- For R power of 2 (superlinear)

$$\frac{N}{L} + R \times (1 + \log_2(R))$$



Hands-on – [f]DOTP

1. Follow instructions of the main README.md → *Arathon SPECIAL* paragraph
    1. ex_name=scratch
    2. Write an integer dot-product and a floating-point dot-product
    3. Inspect apps/bin/arathon.dump
    4. Simulate with waveforms or printf!


P
P

Repository structure

- README.md → Get started guide
- .github → CI source (you can see how scripts are called, here!)
- apps → SW source code
- cheshire → FPGA and Linux flow
- config → Ara's supported configurations
- docs → Documentation
- hardware → Ara's HW source code and testbench

apps – Ara's SW (applications)

- common – Common files and settings (compiler flags, etc.)
- benchmarks – Benchmark infrastructure to measure application HW cycles
- applications – fmatmul, jacobi2d, conv, spmv, fdotp, etc.
- riscv_tests – Testing suite (instruction-wise)
- rvv_bench – External benchmark suite
- rivec-bmarks – External benchmark suite

hardware – SystemVerilog files

- src, include → Ara's SystemVerilog codebase
- tb → Testbench infrastructure
- spyglass → Spyglass lint flow

cheshire – FPGA flow

- Demonstration (README.md in cheshire/README.md)
- Compile applications, rivec, etc.
- Get to a Linux image (fast-forward)
- Patch and build bitstream (fast-forward)
- Load payload on FPGA SD
- Load bitstream and run programs

Vectorize (or understand) fp-matmul

- Try to think of how to vectorize a matrix multiplication between two N*N mtx
- If you want to know how we do it, take a look at apps/fmatmul
- Try to understand what's happening

Find a bug through riscv-tests!

- Modify one of the riscv-tests, boosting it with new combinations of instructions!
- Test that your code works on a golden model

```
make –C apps riscv_tests_spike –j16
```

- Then, test it on Ara and find new bugs!

```
make –C apps riscv_tests –j16
make –C hardware sim app=rv64uv-ara-${name_of_the_test}
# e.g., make –C hardware sim app=rv64uv-ara-vadd
```