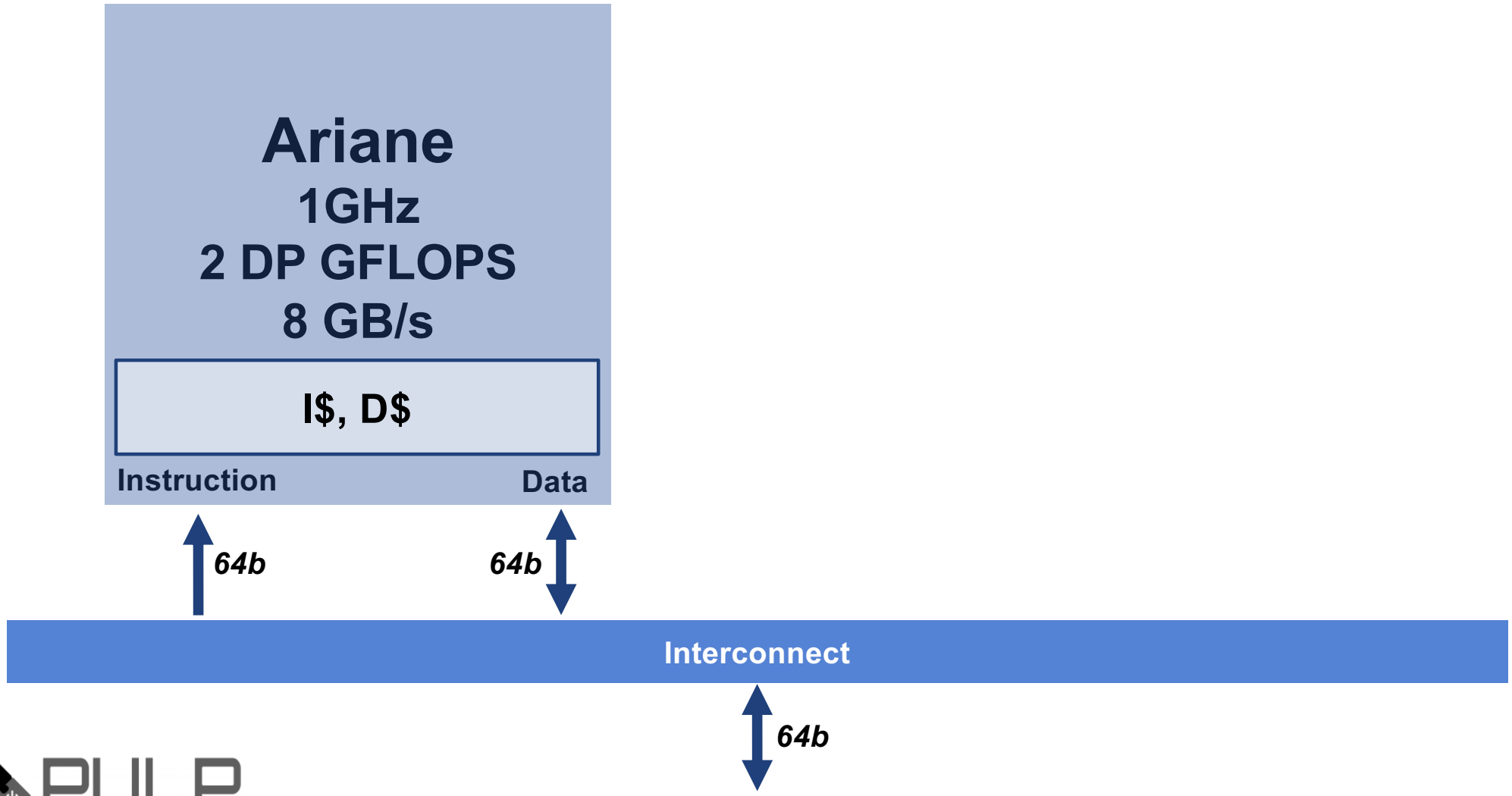
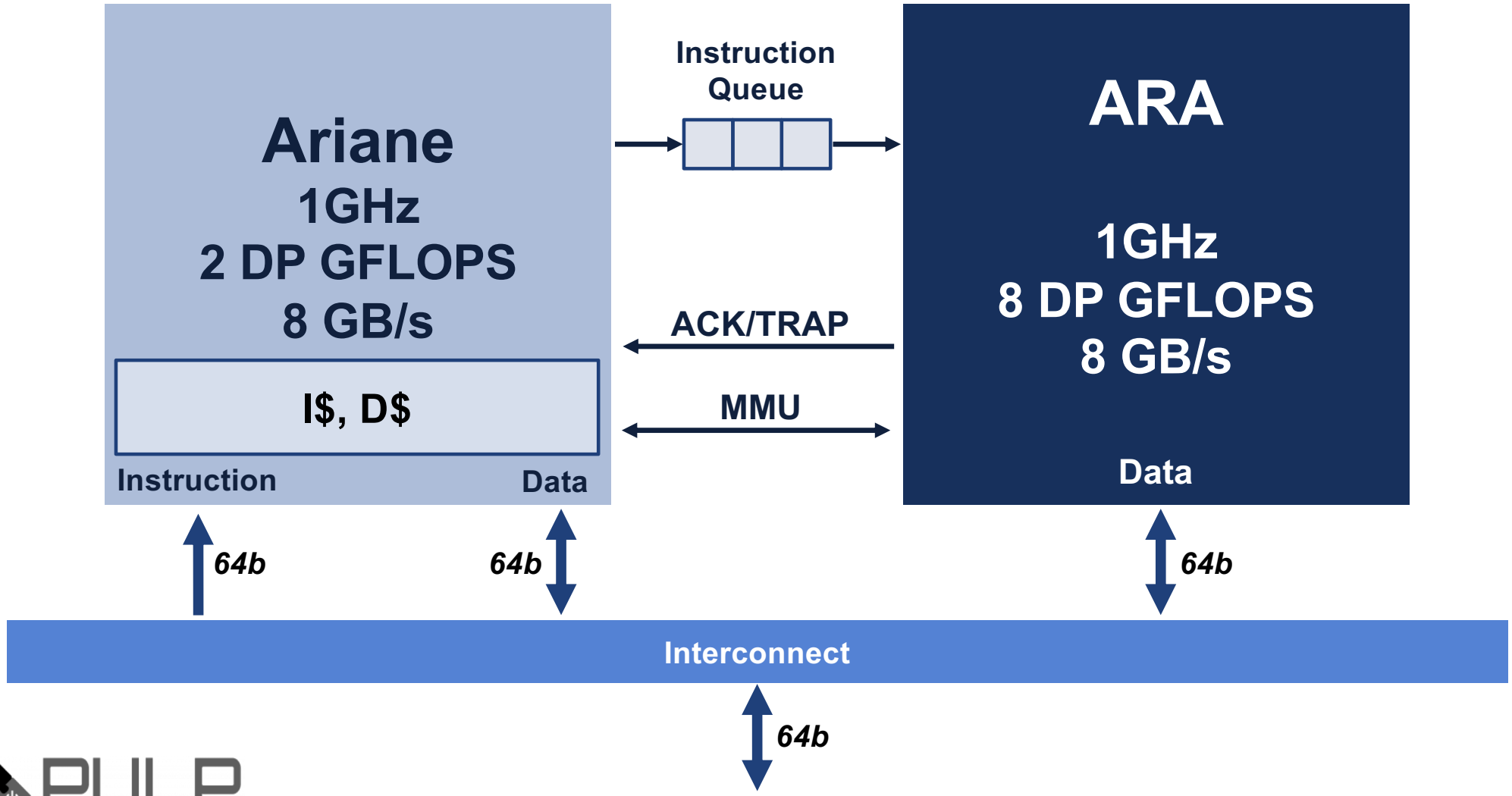
RISC-V Summit

ARA: 64-BIT RISC-V VECTOR IMPLEMENTATION IN 22NM FDSOI

Matheus Cavalcante
PhD Student
ETH Zurich

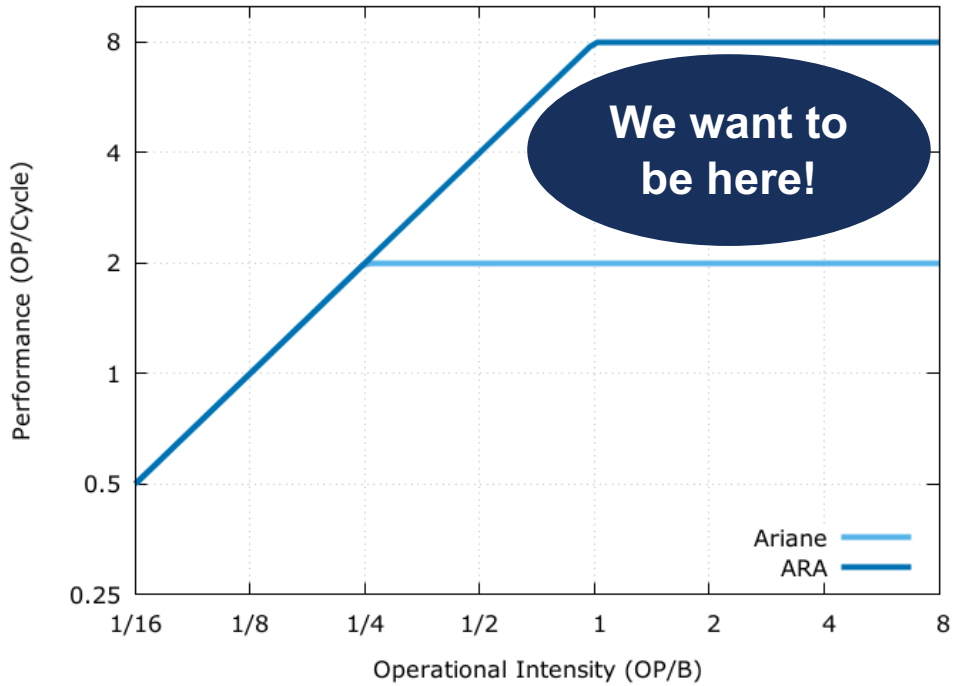
Fabian Schuiki
PhD Student
ETH Zurich


Ariane
1GHz
2 DP GFLOPS
8 GB/s
I$, D$
Instruction
Data
64b
64b
Interconnect
64b


Ariane
1GHz
2 DP GFLOPS
8 GB/s
I$, D$
Instruction
Data
Instruction Queue
ACK/TRAP
MMU
ARA
1GHz
8 DP GFLOPS
8 GB/s
Data
64b
64b
64b
Interconnect
64b

Memory Bandwidth and Performance: ARA and Ariane Rooflines

- Operational Intensity: *operations per byte*
  - Algorithm dependent
  - One FMA = 2 operations
- Memory- and compute-boundness
- Memory bandwidth
- Number of FMAs

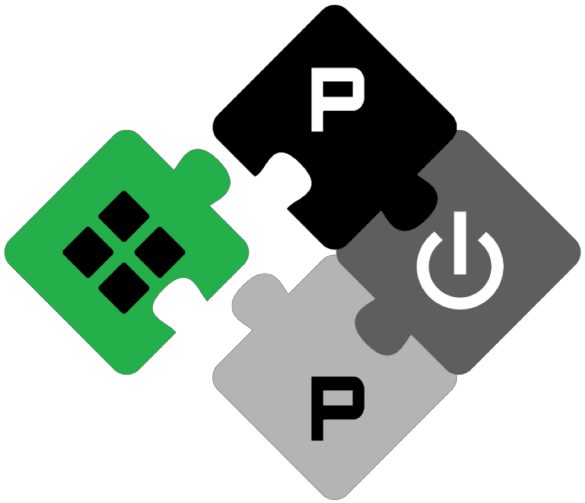


PULP Parallel Ultra Low Power

ARA: High-performance vector processor

- Global Foundries' 22FDX process
  - Master's Thesis (+ a few months of ongoing PhD studies!)
  - Planning to open-source it within the PULP platform (as usual!)

- **Snapshot** of the current development
  - Challenges we faced
  - Results we achieved
  - Insights we gained



Vector processors background

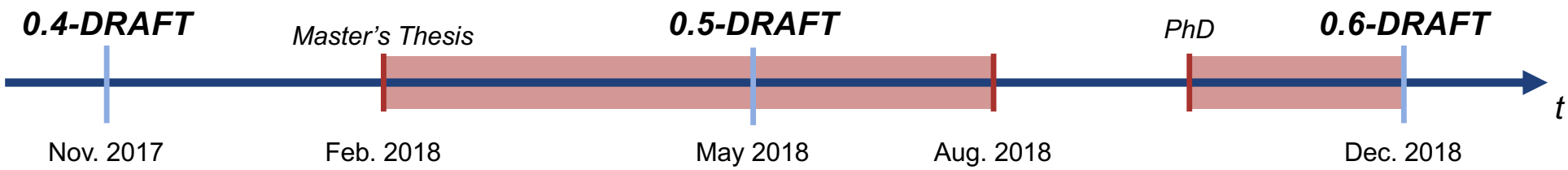
- Vector processing: SIMD
  - Less instruction BW, simpler control, less energy per operation

  - Packed-SIMD vs. Vector “Cray-like” SIMD
- CRAY-I (1977)
- Fujitsu A64FX
  - Based on ARM’s v8-A SVE
  - 512-bit wide packed-SIMD
  - Peak-performance at 2.7 TFLOPS
- Hwacha
  - Vector-fetch architecture
  - More complex: vector unit fetches its instructions and threads can diverge

PULP Parallel Ultra Low Power

RISC-V "V" Extension

- RISC-V "V" Extension: "Cray-like" vector-SIMD approach
- ARA: based on version 0.4-DRAFT



- No full compliance
  - No support to fixed-point and vector atomics – not our focus
  - Limited support to type promotions (e.g., 64b ← 8b + 8b) – hardware cost
    - Eventually dropped in later versions of the Extension

ARA Microarchitecture

Main datapath element: FMA Units

- We support masked FMA instructions (four operands):

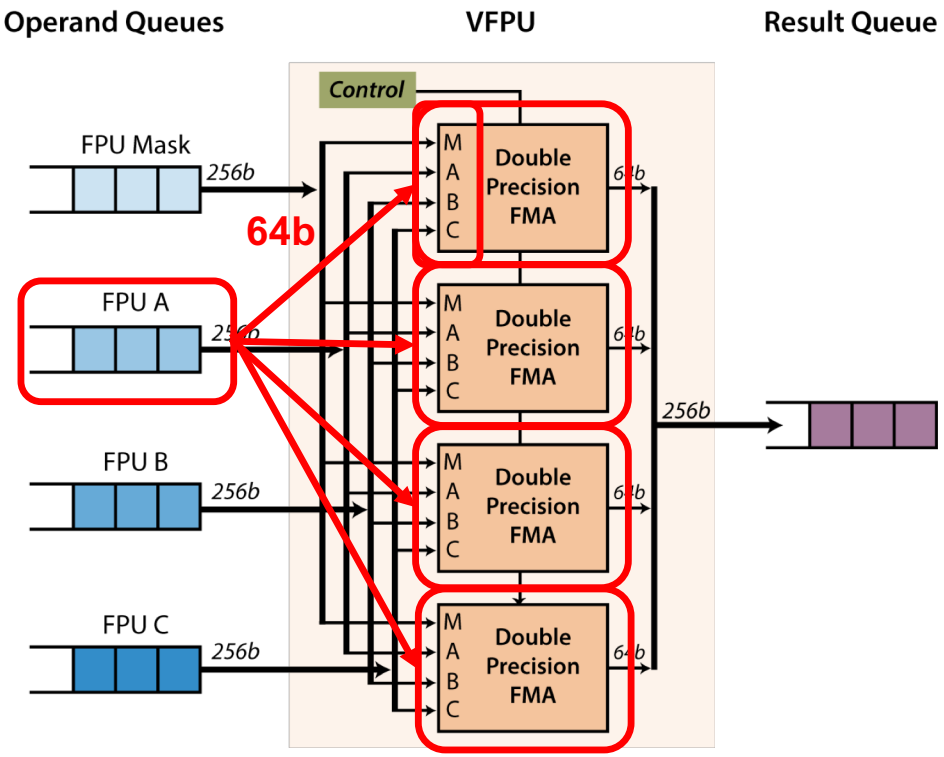
  `vmadd vd, vsa, vsb, vsc, vmask`

  vd[i] = lsb(vmask[i]) ? vsa[i] + vsb[i] + vsc[i] : 0;

- The four lanes operate in lockstep
  - Low control overhead

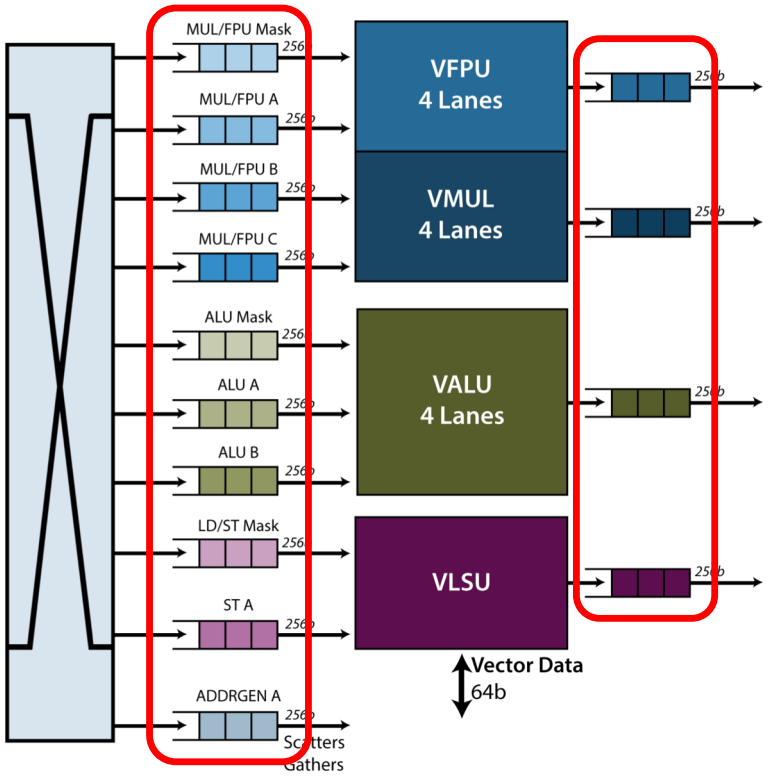
- FMA is pipelined (5 cycles) to meet f_{min} constraint

- Each lane gets 64b operands from four 256b input FIFO buffers (A, B, C, VMASK)
  - Number of lanes determines buffer width



Operand FIFO queues

- One input FIFO buffer provides one operand to all the (four) lanes
  - 256b (4x64b) wide entries
  - One FIFO buffer per operand per multi-lane datapath unit → 10 FIFO buffers
- Output FIFO buffers for output operands, one per multi-lane datapath unit
- Needed to sustain maximum throughput for the lock-step operation of the FUs, while hiding the latency caused by banking conflicts in the VRF (next slides)

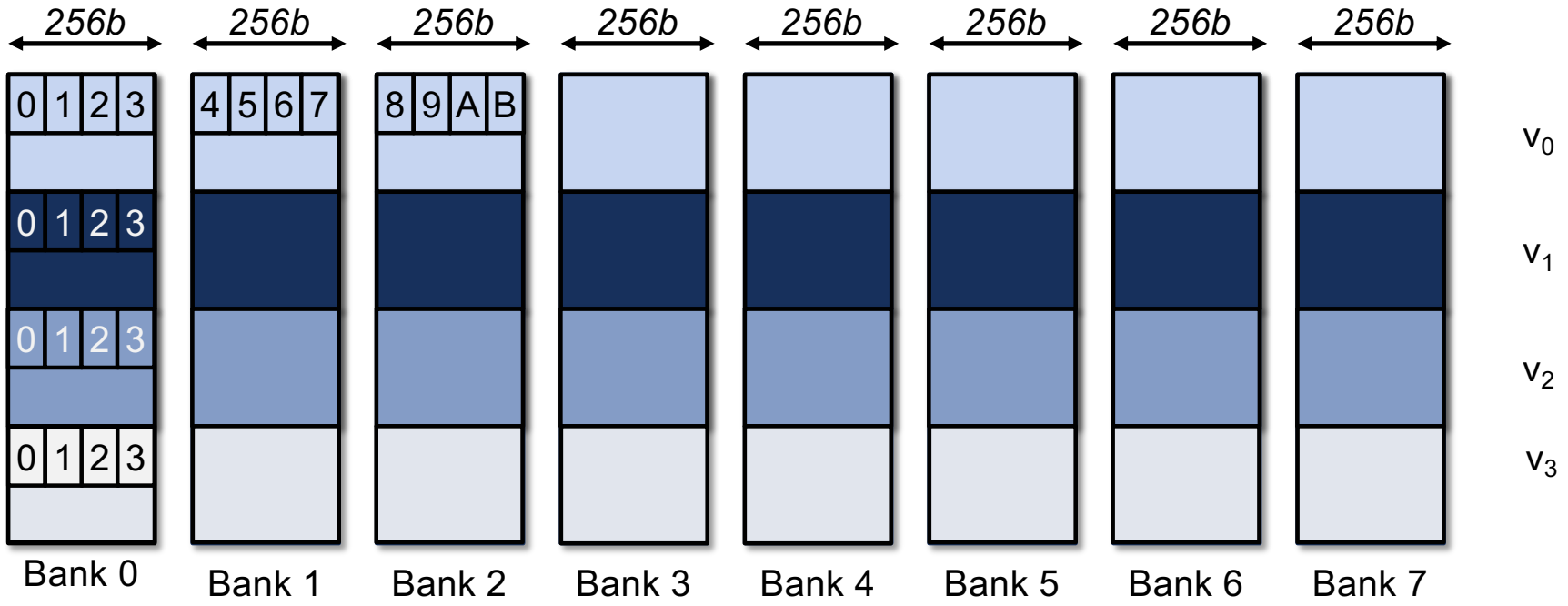


PULP
Parallel Ultra Low Power

Vector register file

256b banks → one bank stores 4 operands consumed in parallel by the 4 lanes

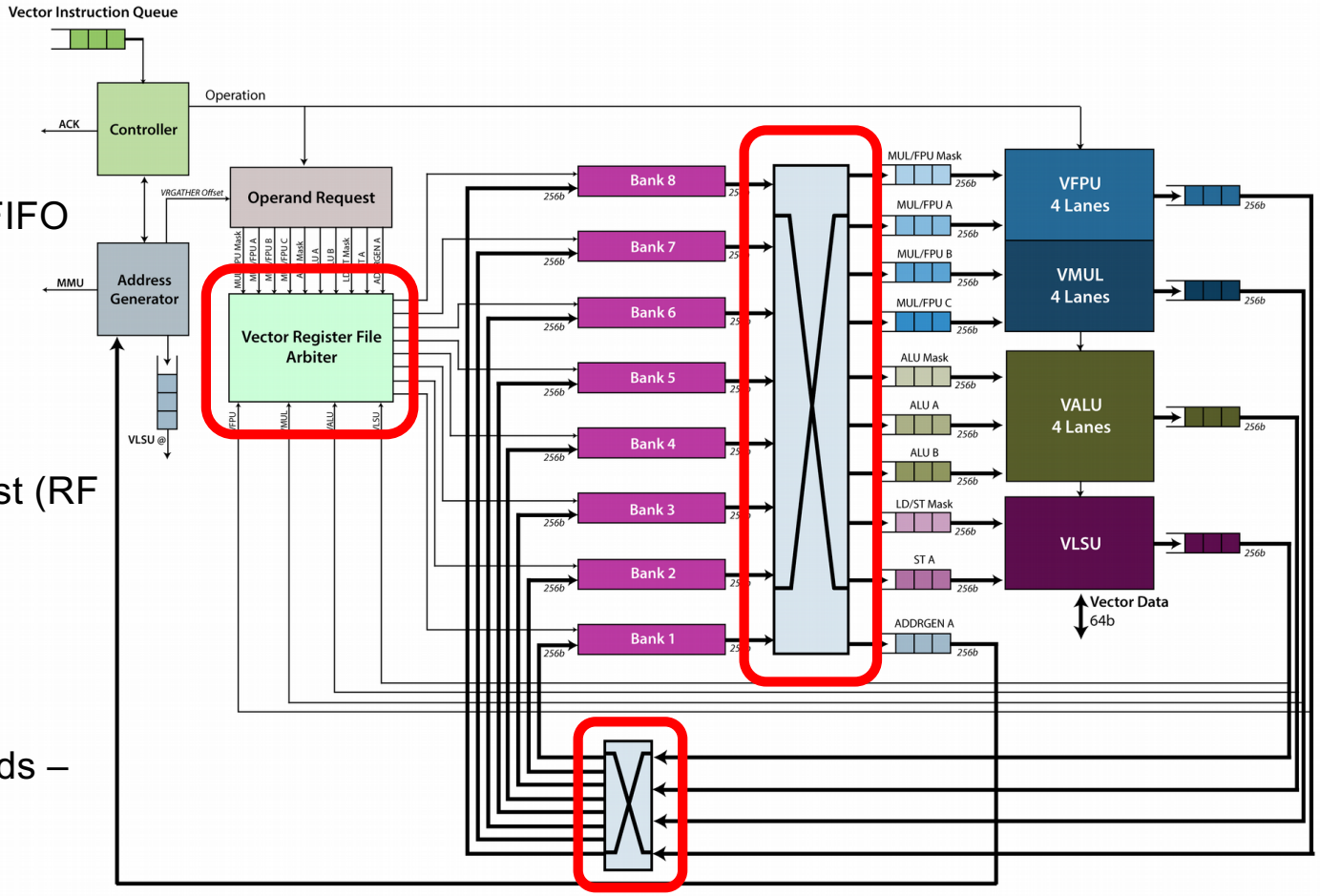
8 banks → BF of 1,6 for the worst-case read BW (FMA is 4R+1W/cycle)



PULP Parallel Ultra Low Power

Vector Register File and Operand-Deliver Interconnect

- All-to-all input log-interconnect
  - 256b wide (64bx4)
  - 8-source (VRF banks) x 10-dest (FIFO buffers)
  - Registered boundaries (for timing)
- All-to-all output log-interconnect
  - 256b wide (64bx4)
  - 4-source (out FIFO buffers) x 8 dest (RF banks)
- Fixed-priority arbiter
  - $P_M > P_A > P_B > P_C$
  - VRF is built as 1RW SRAM banks
  - Writes have lower priority than reads – unless output queue is full



PULP Parallel Ultra Low Power

Execution of a FMA instruction

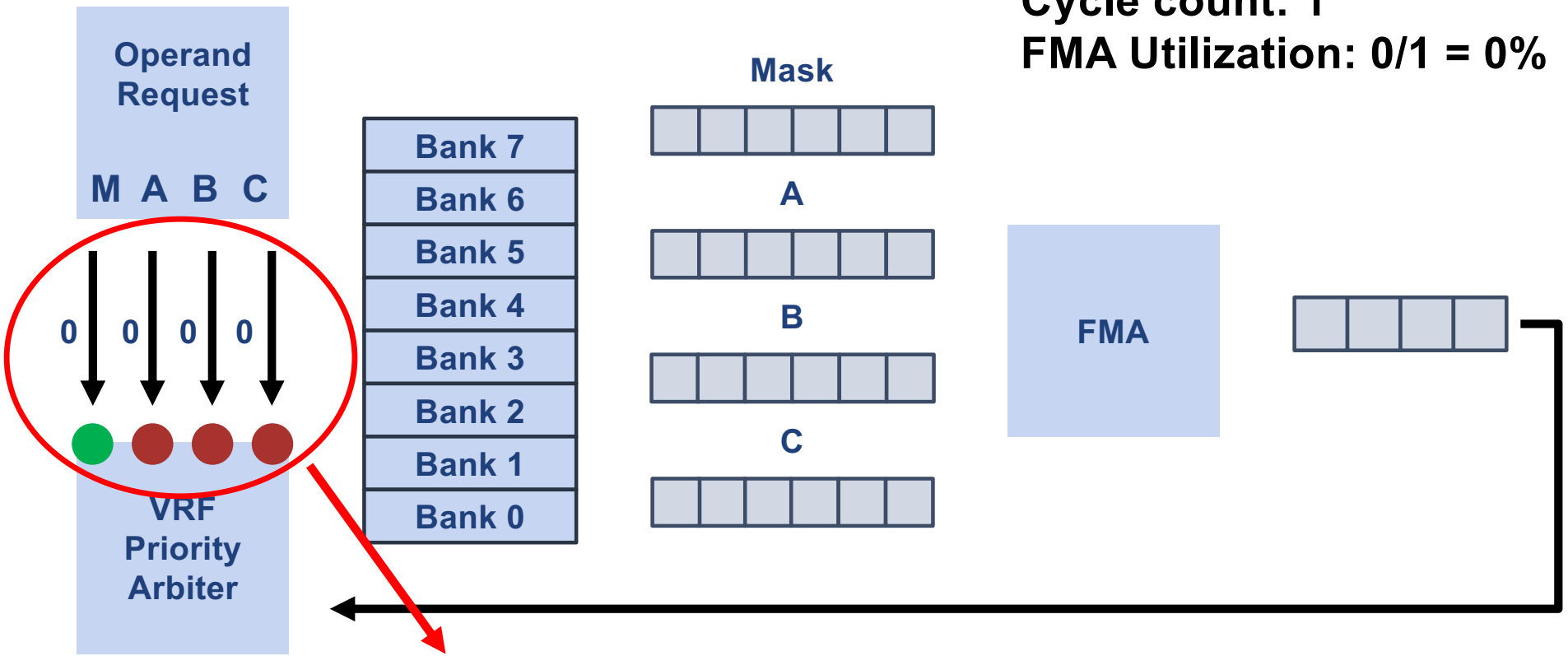
- Consider the execution of the following instruction

```
vmadd vd, vsa, vsb, vsc, vsmask
```

- Worst case in terms of banking factor
  - 4 reads + 1 write per cycle
  - Banking factor = 1,75

- We take a vector of length 256 (ideally 64 cycles to run)

Execution of a FMA instruction

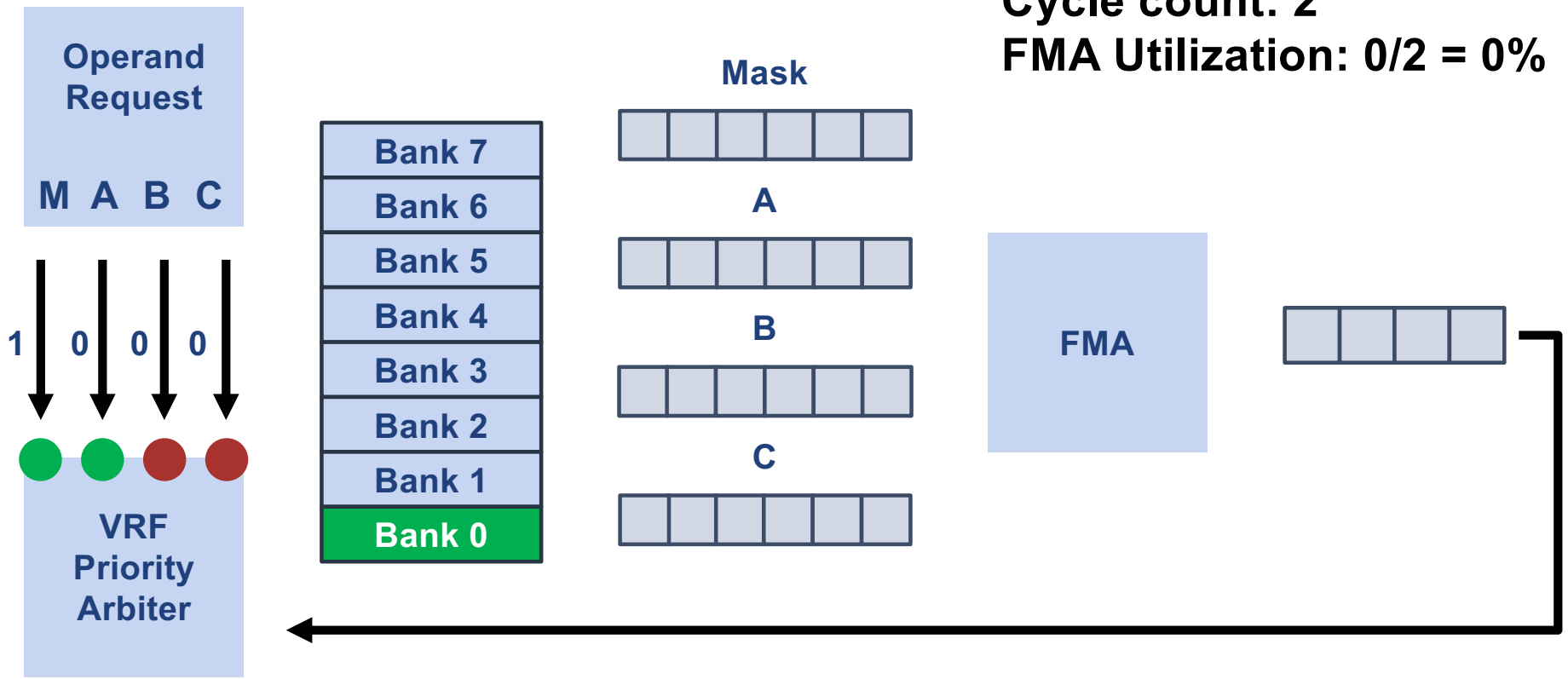


Cycle count: 1
FMA Utilization: 0/1 = 0%

The first 4 elements of all 4 operands are in Bank 0
3 access stalled due to banking conflicts

PULP Parallel Ultra Low Power

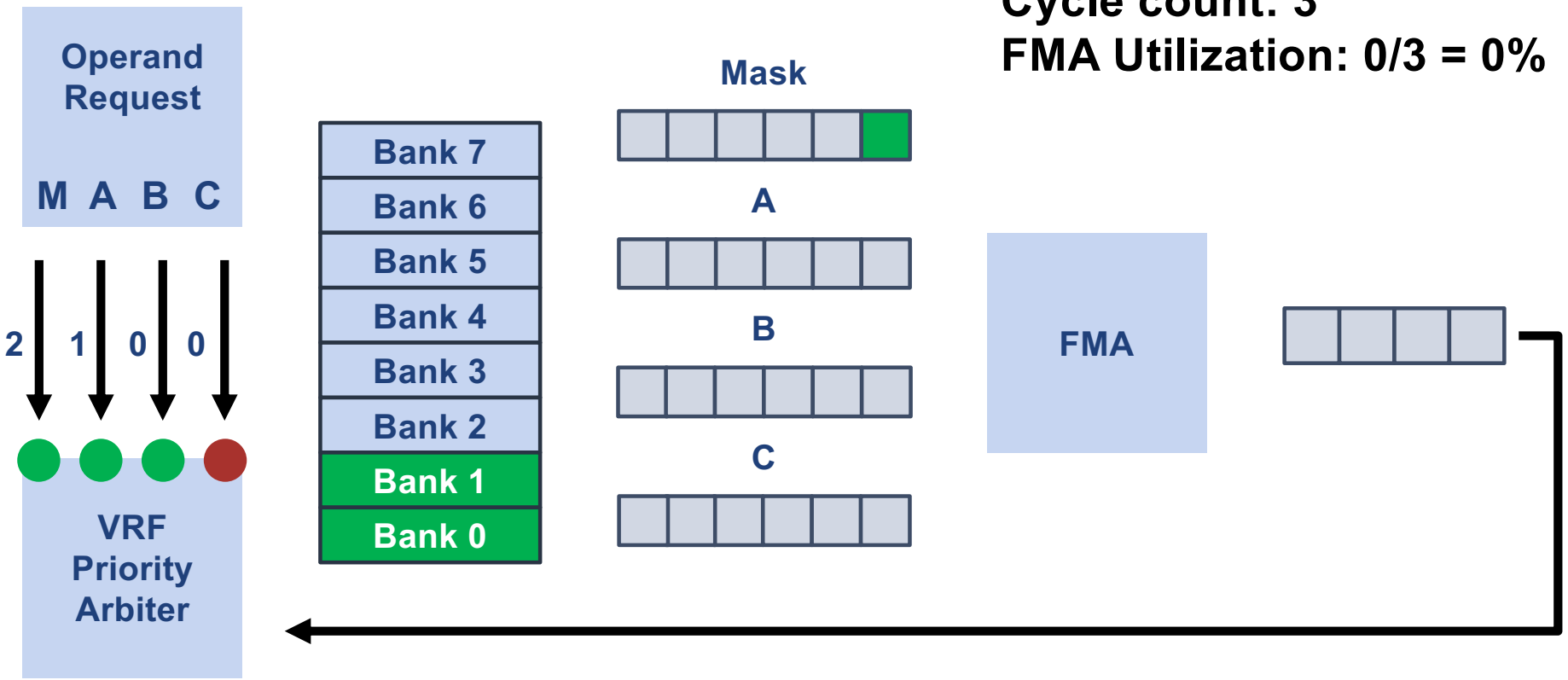
Execution of a FMA instruction



Cycle count: 2
FMA Utilization: 0/2 = 0%

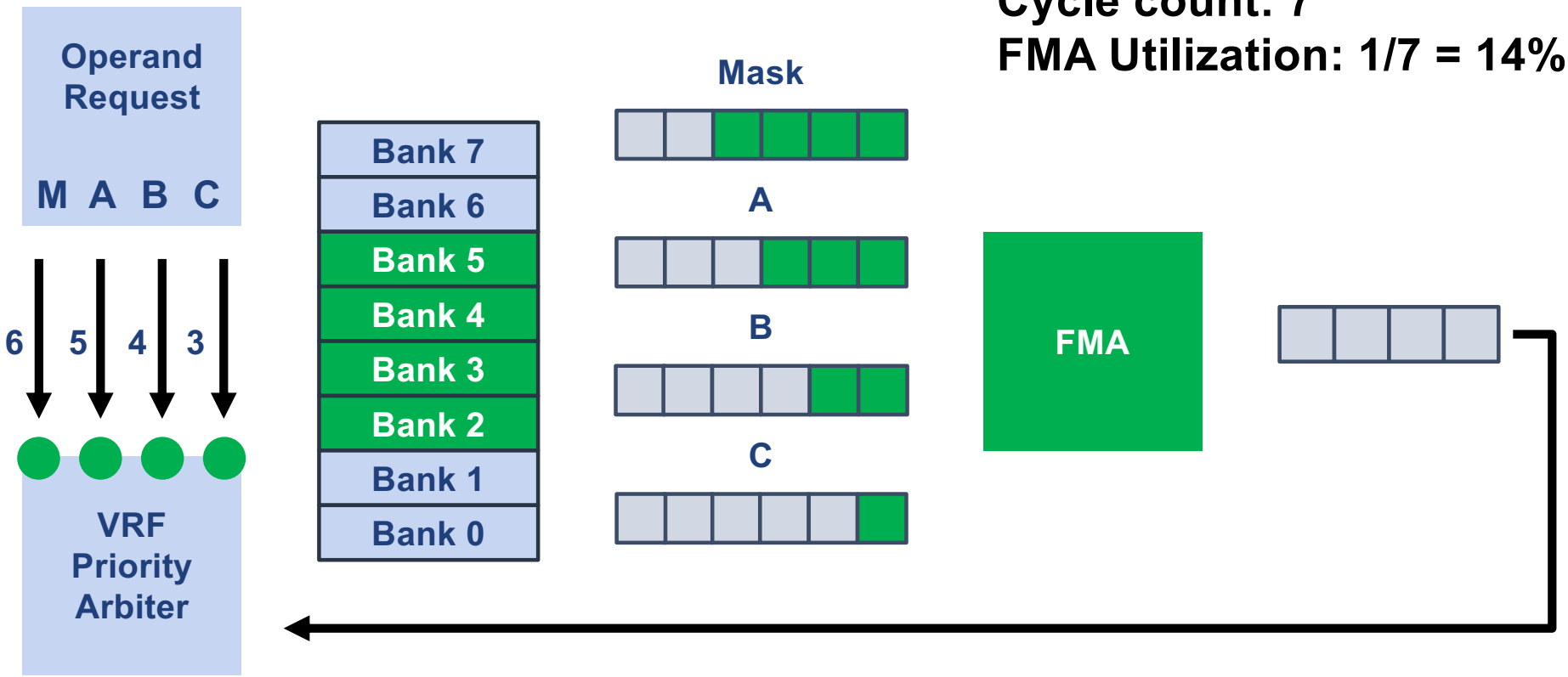
PULP Parallel Ultra Low Power

Execution of a FMA instruction



Cycle count: 3
FMA Utilization: 0/3 = 0%

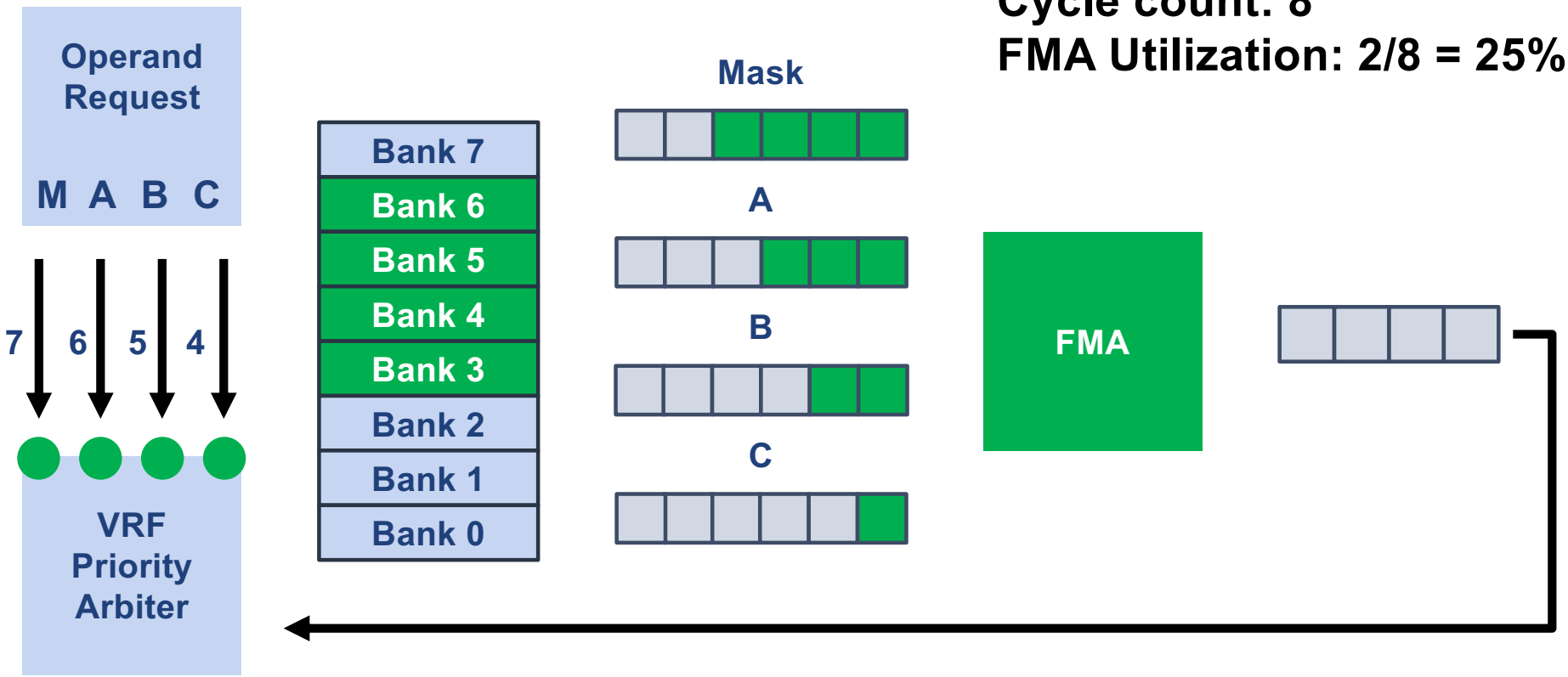
Execution of a FMA instruction



Cycle count: 7
FMA Utilization: 1/7 = 14%

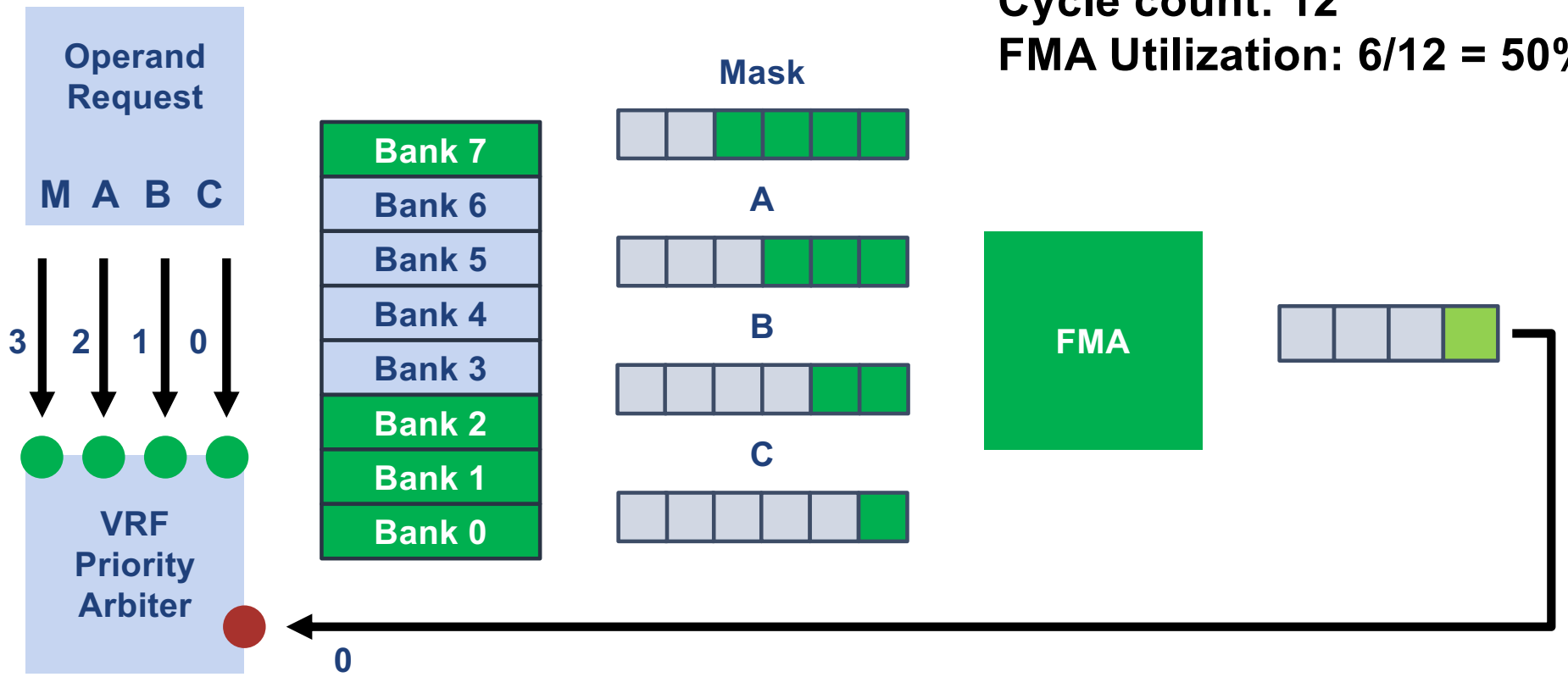
PULP Parallel Ultra Low Power

Execution of a FMA instruction



Cycle count: 8
FMA Utilization: 2/8 = 25%

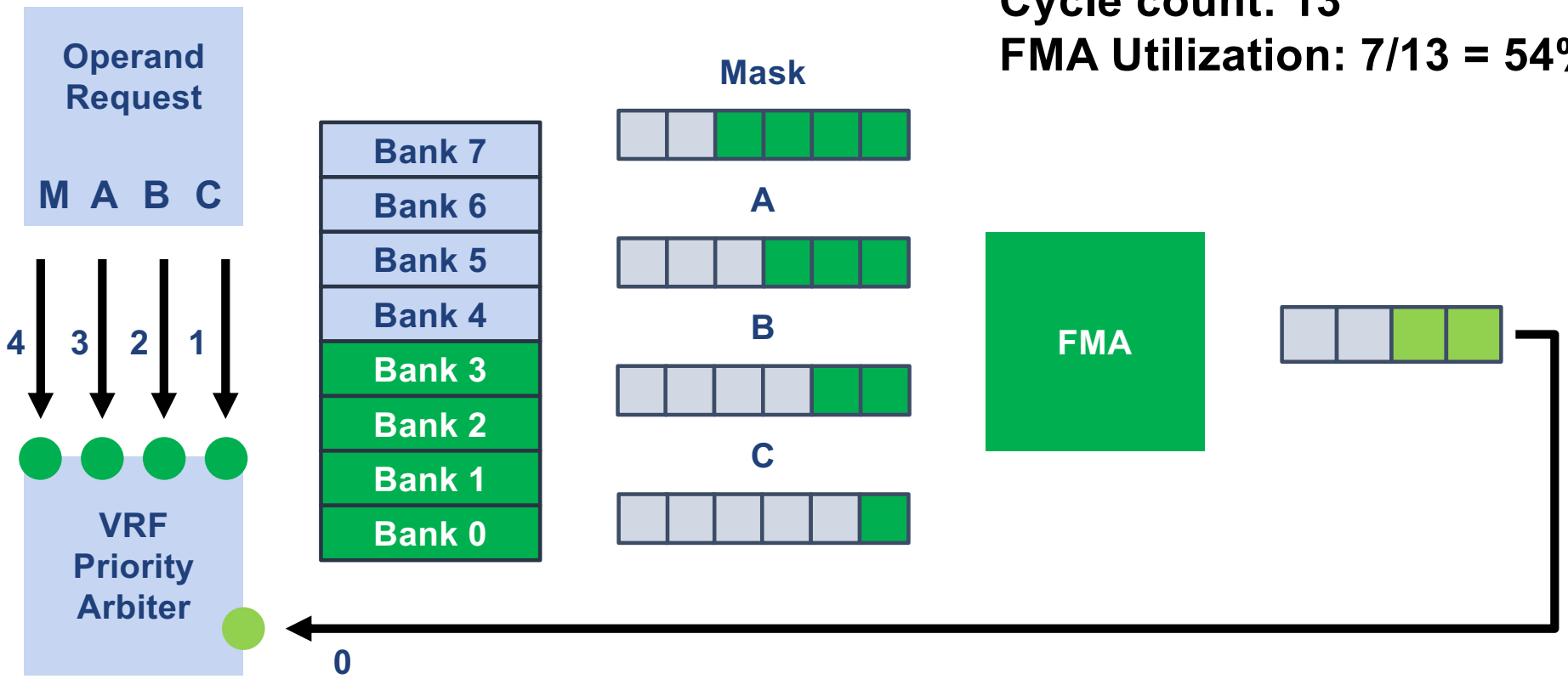
Execution of a FMA instruction



Cycle count: 12
FMA Utilization: 6/12 = 50%

PULP Parallel Ultra Low Power

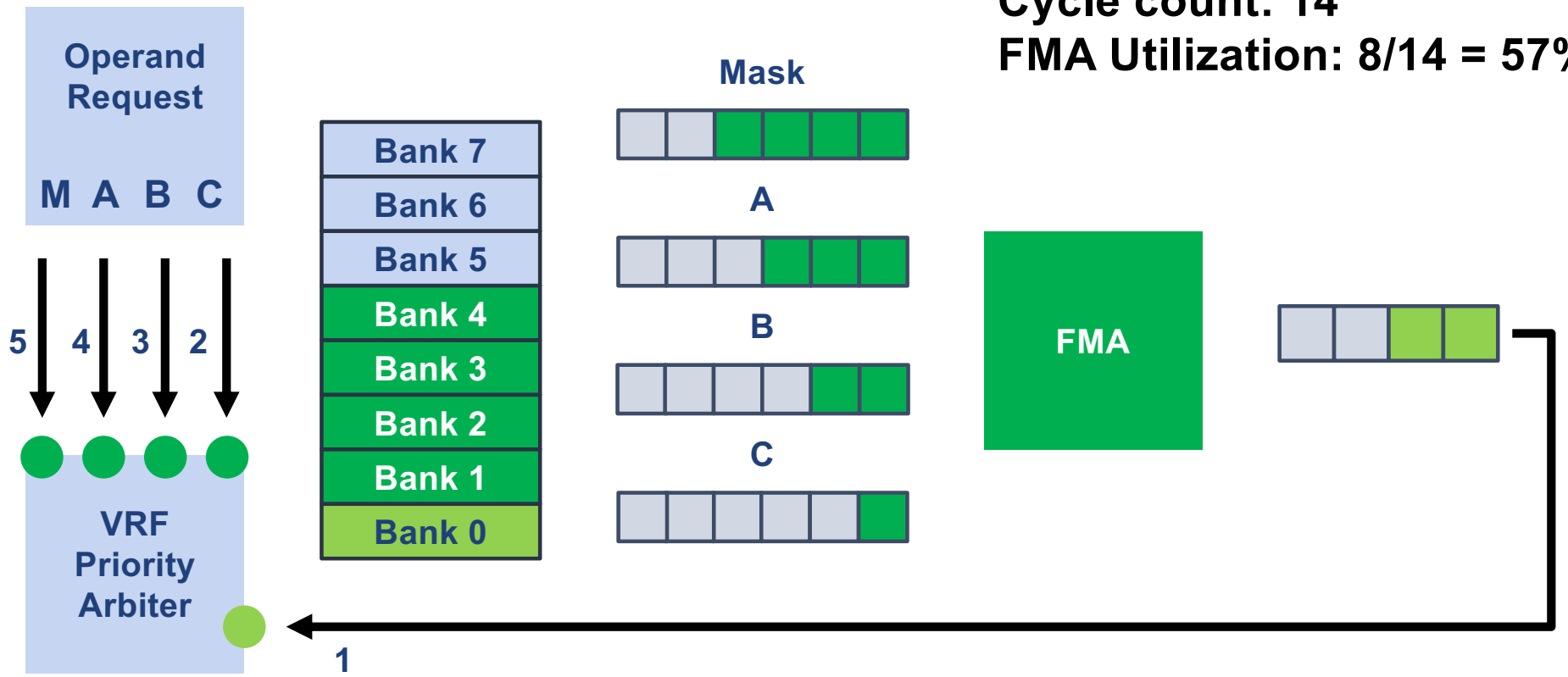
Execution of a FMA instruction



Cycle count: 13
FMA Utilization: 7/13 = 54%

PULP Parallel Ultra Low Power

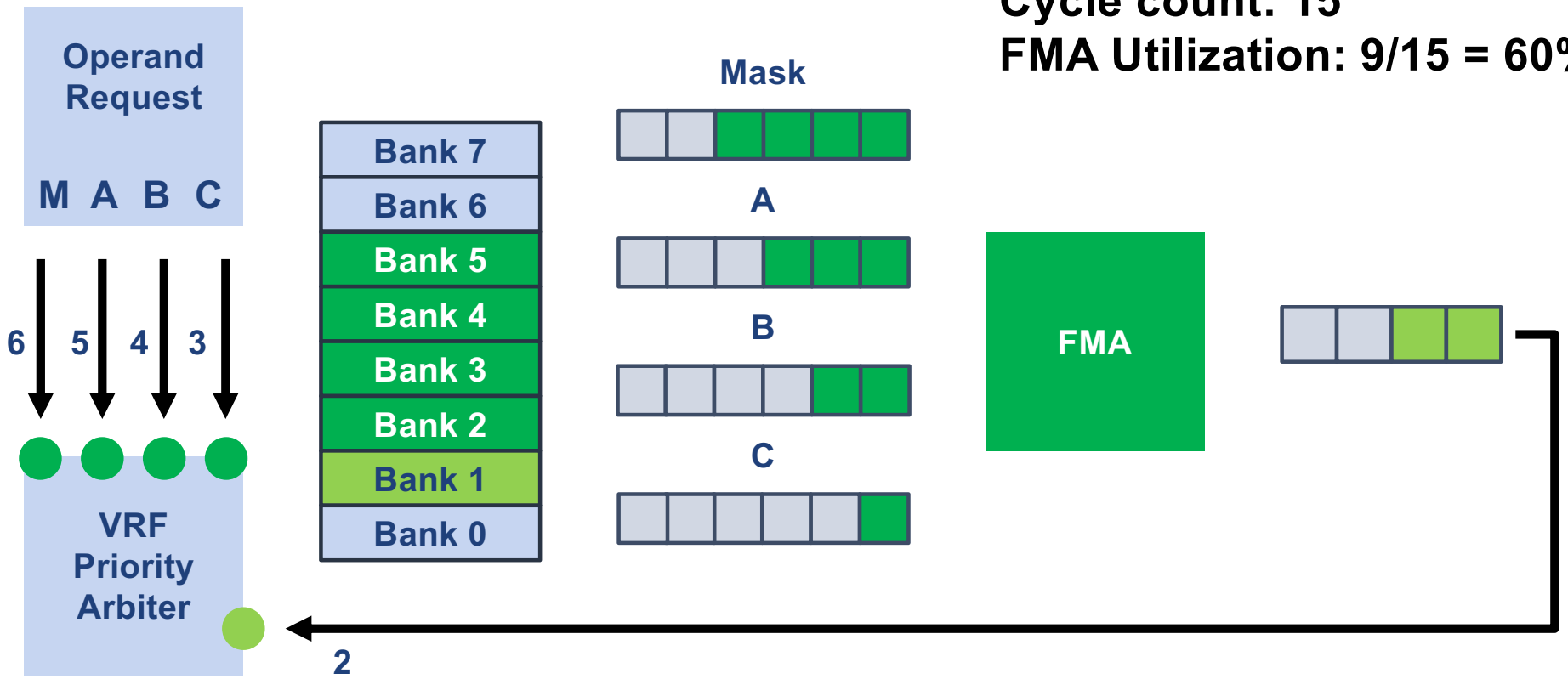
Execution of a FMA instruction



Cycle count: 14
FMA Utilization: 8/14 = 57%

PULP Parallel Ultra Low Power

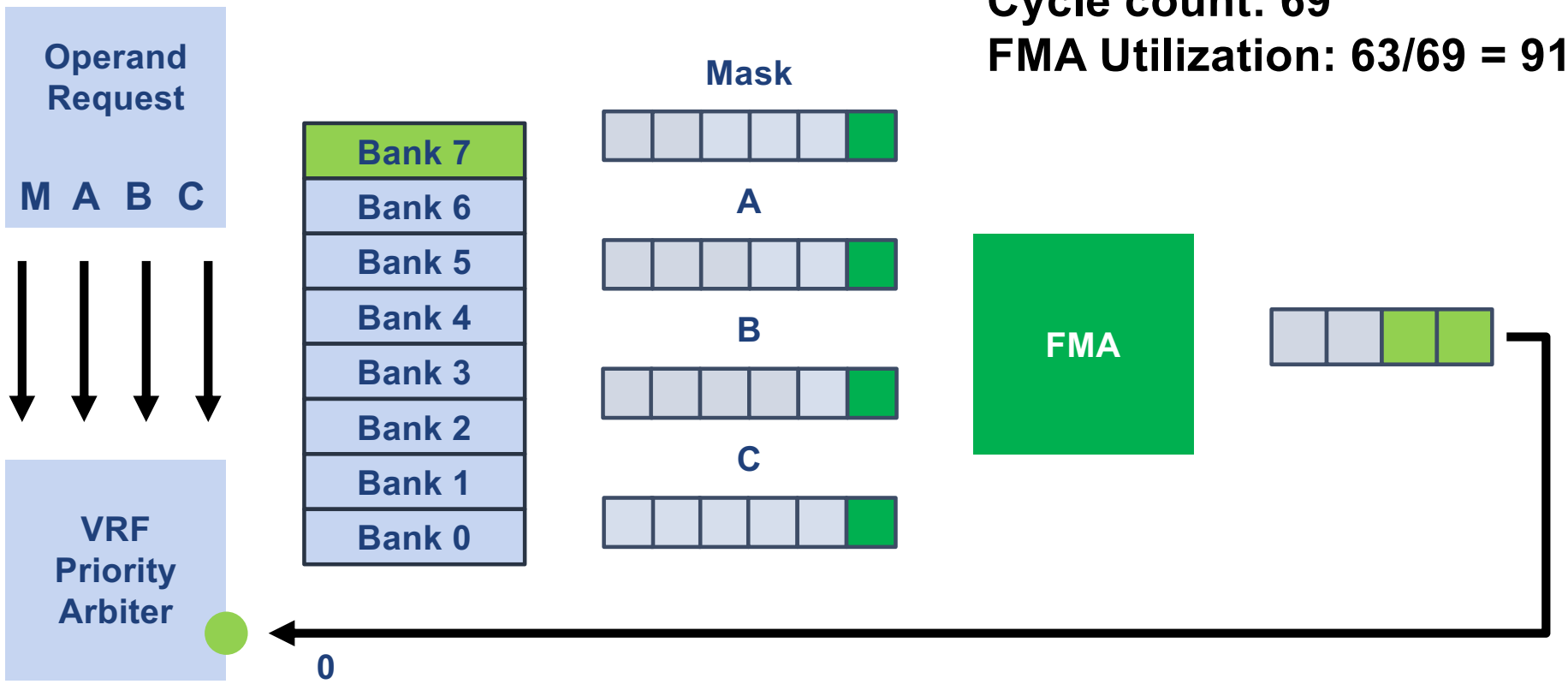
Execution of a FMA instruction



Cycle count: 15
FMA Utilization: 9/15 = 60%

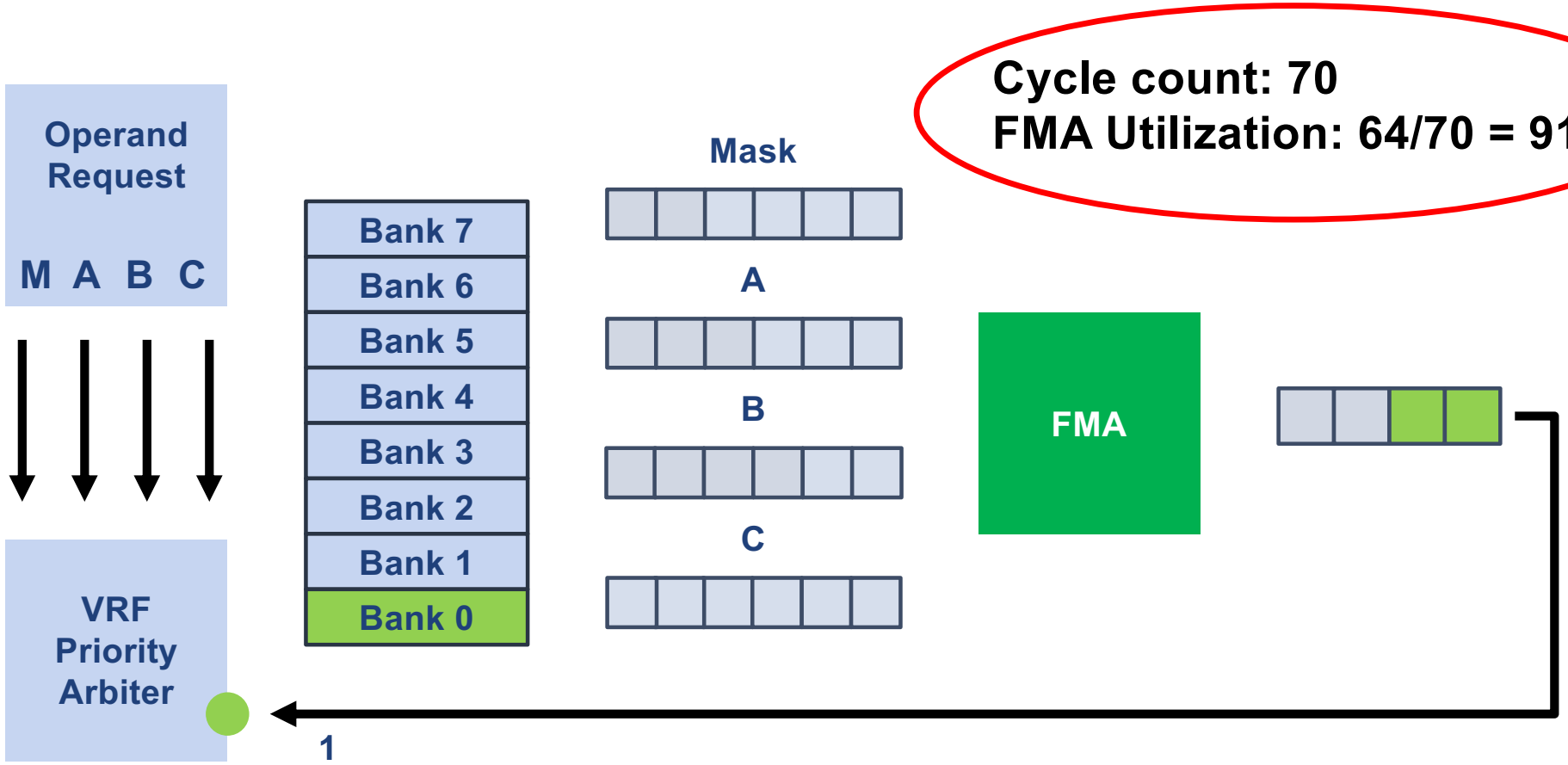
PULP Parallel Ultra Low Power

Execution of a FMA instruction



Cycle count: 69
FMA Utilization: 63/69 = 91%

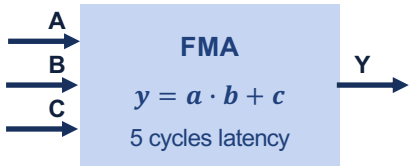
Execution of a FMA instruction

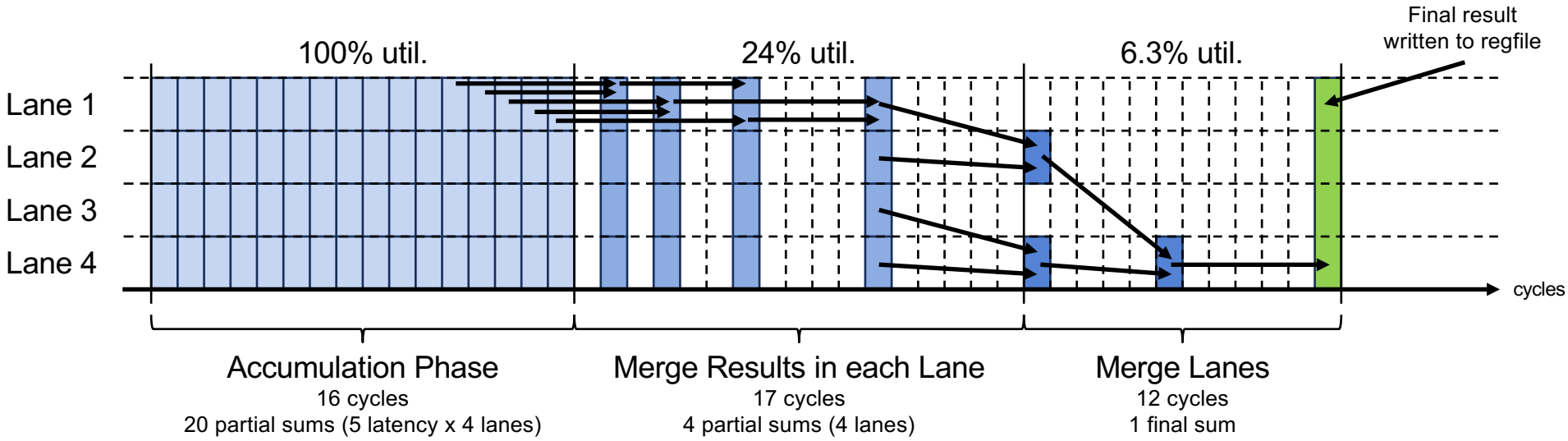


Hardware Support for Vector Reductions

$$y = \sum_i A_i \cdot B_i$$

- Triggered by VMADD instruction with scalar result register
- Executed on FMA units by feeding results back in as operand C
- E.g. Reduction of 64-element vector:
- Avg. utilization in this case 36% ($\frac{N}{N+4\cdot 29}, N = 64$)



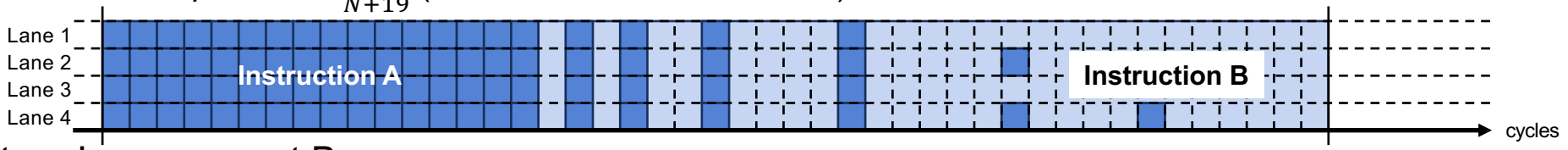


PULP Parallel Ultra Low Power

Ways to Improve Reductions

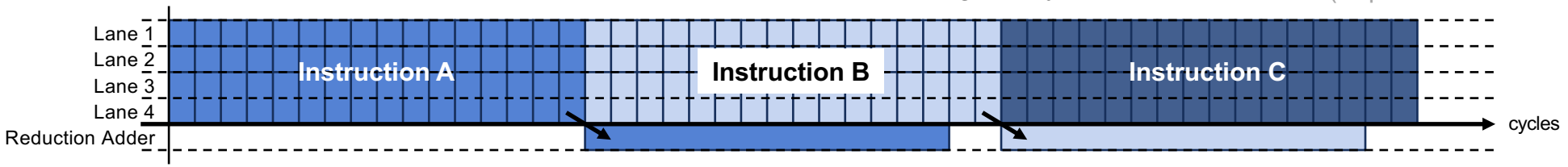
- Current Implementation (constant 29 cycle tail):
  - $\frac{N}{N+4\cdot 29}$ (36% for 64-element vector)
- Future Improvement A:
  - Schedule FPU operations of next instruction in gaps of the reduction
  - Utilization improves to $\frac{N}{N+19}$ (77% for 64-element vector)

Low Hardware Cost
(More Complex FSM)



- Future Improvement B:
  - Add separate reduction adder
  - Utilization improves up to 100%, since reduction tail not eating away performance

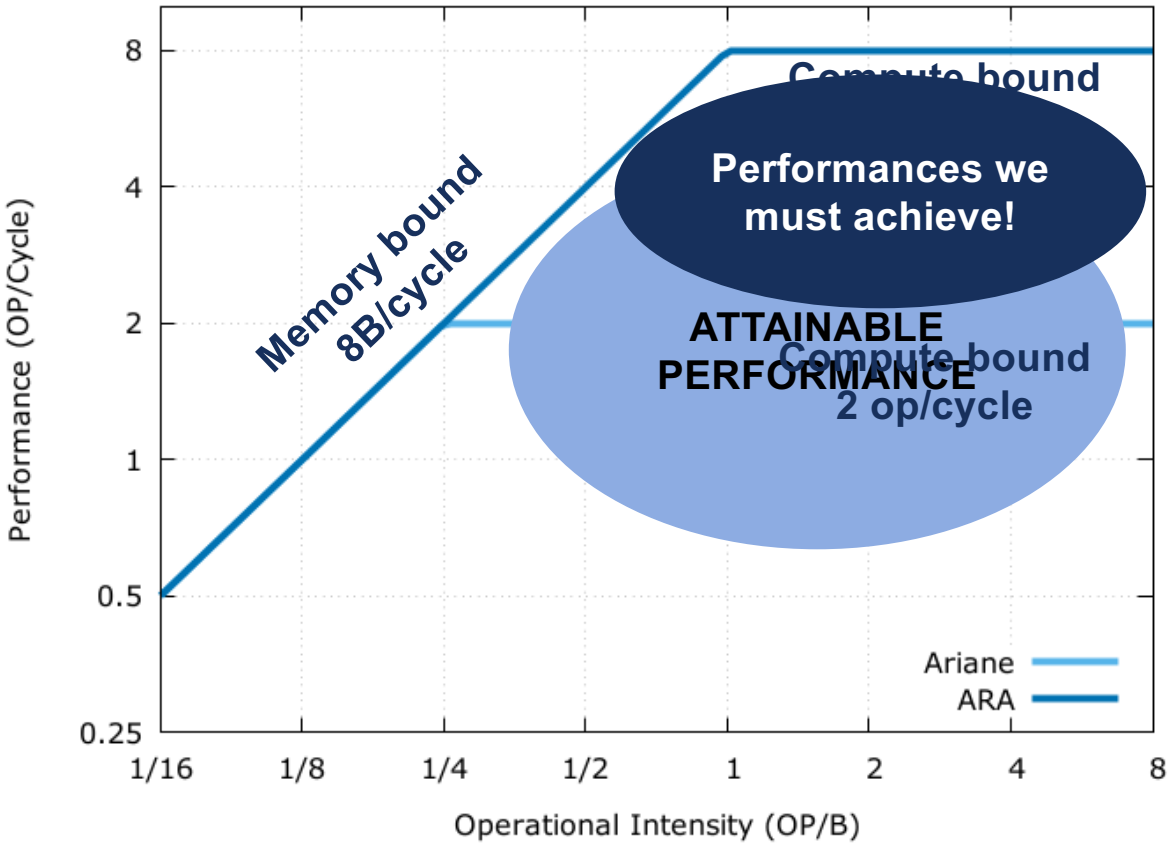
Significant Hardware Cost
(Separate FP ADD Unit)



PULP
Parallel Ultra Low Power

Benchmarks

ARA and Ariane – Peak performance



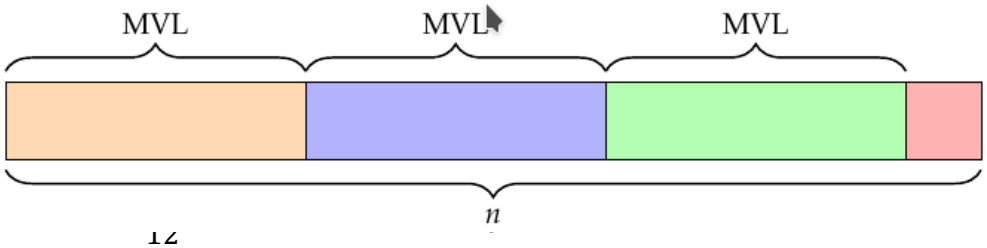
PULP
Parallel Ultra Low Power

Benchmarks

- Can we achieve 8 GFLOPs peak performance?
  - Upper-bound: four FMAs working at 100%
- Three key kernels:
  - Multiply-add (DAXPY): heavily memory-bound
  - Convolution (DCONV): compute-bound
  - Matrix-multiplication (DGEMM): compute-bound
- Cycle-accurate simulation from the RTL
  - We ignore the initial set-up cycles (around 40 cycles)
  - Startup, instruction fetch, decoding, vector unit configuration…

DAXPY: $Y \leftarrow aX + Y$

- Strip-mined loop over the n elements of vectors x and y



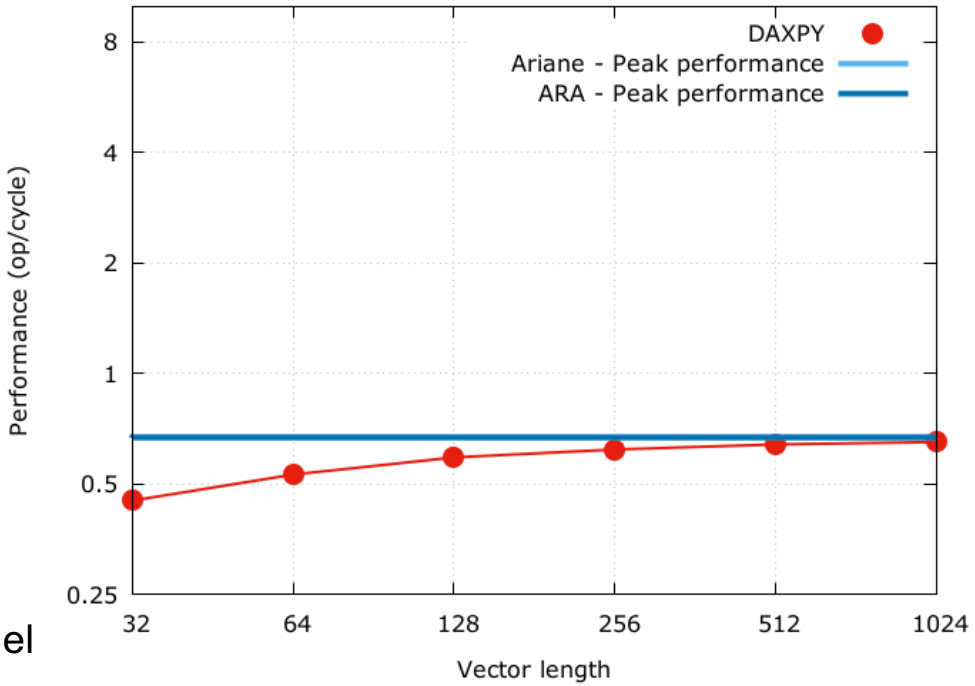
- Memory-bound
  - We'll be far from 8 GFLOPs!
  - But are we close to the performance limit?

```
// Read scalar a
vins va, a, zero;

while (n != 0) {
    // Stripmined loop
    size_t vl = setvl(n);

    // Read x and y
    vld vx, x;
    vld vy, y;

    // vy = va . vx + vy
    vmadd vy, va, vx, vy;

    // Store y
    vst vy, y;

    // Bump pointers
    x += vl; y += vl; n -= vl;
}
```

PULP Parallel Ultra Low Power

DAXPY: Performance

Vector Length	FPU Utilization (%)	Performance (op/cycle)
32	5,6%	**0,45**
64	6,6%	**0,53**
128	7,3%	**0,59**
256	7,7%	**0,62**
512	8,0%	**0,64**
1024	8,1%	**0,65**



- We achieve what we could in terms of perf
  - Can't expect 8 GFLOPs from a memory-bound kernel
- Ops/cycle grows to 8 if we increase memory port width (e.g. 128b → 2x perf)

PULP
Parallel Ultra Low Power

DCONV: $Y = K * X$

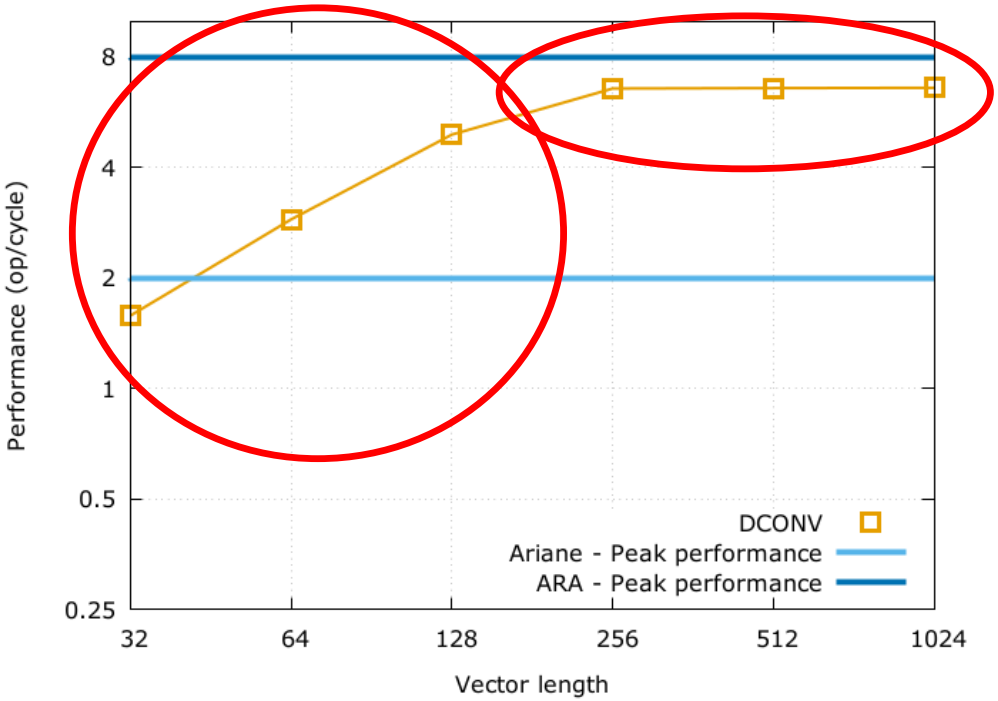
- Kernel particular for CNNs
  - Convolution kernel size: 7 channels, each 3×3
  - Image size: 7 channels, each $n\times1$
- Operational intensity
  - $3\times3\times8\times7n = 504n$ bytes of memory transfers
  - $882n$ operations (multiply-adds)
  - $1{,}75$ operations per byte
- Compute-bound kernel
  - It should be possible to achieve 8 ops/cycle
  - Scheduling is key

PULP
Parallel Ultra Low Power

DCONV: Performance

Vector Length	FPU Utilization (%)	Performance (op/cycle)
32	19,8%	**1,58**
64	36,1%	**2,89**
128	61,5%	**4,92**
256	82,1%	**6,57**
512	82,3%	**6,59**
1024	82,5%	**6,60**

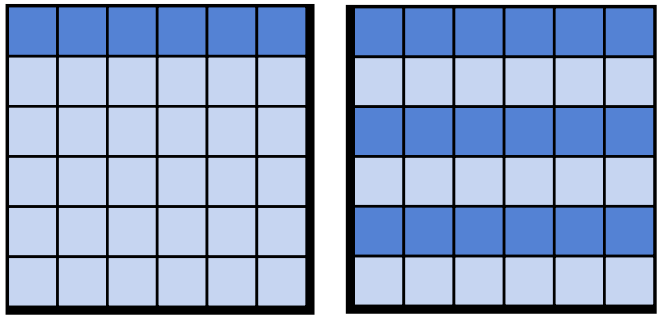
- Initial banking conflicts limit performance
- Performance goes up until strip-mining loop comes to play
  - Unroll strip-mining: programmability?
  - Hard to hide all the memory transfers (initial loads and final stores)



PULP
Parallel Ultra Low Power

DGEMM: $C \leftarrow \alpha AB + \beta C$

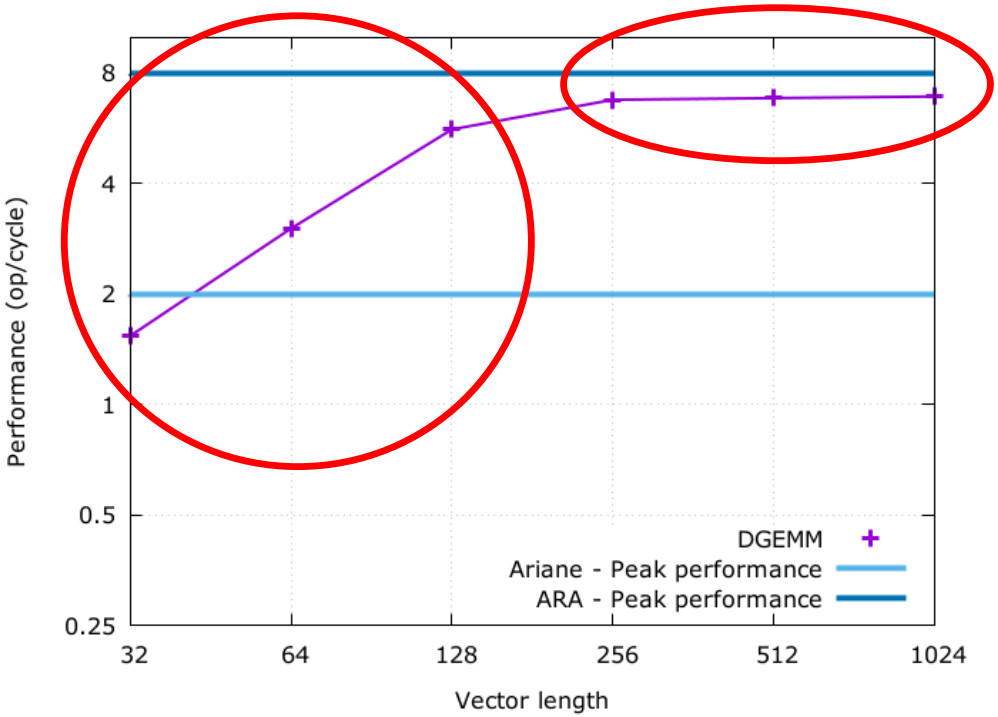
- BLAS-3 routine
  - Common kernel in several applications
- High data reuse
  - When the kernel is compute-bound, it should be possible to achieve 8 ops/cycle
- Operational intensity
  - $8 \times 3n^2 = 24n^2$ bytes of memory transfers
  - $2n^3$ operations (multiply-adds)
  - $\frac{n}{12}$ operations per byte
  - If $n \leq 12$, kernel is memory-bound by ARA's VLSU unit



PULP Parallel Ultra Low Power

DGEMM: Performance

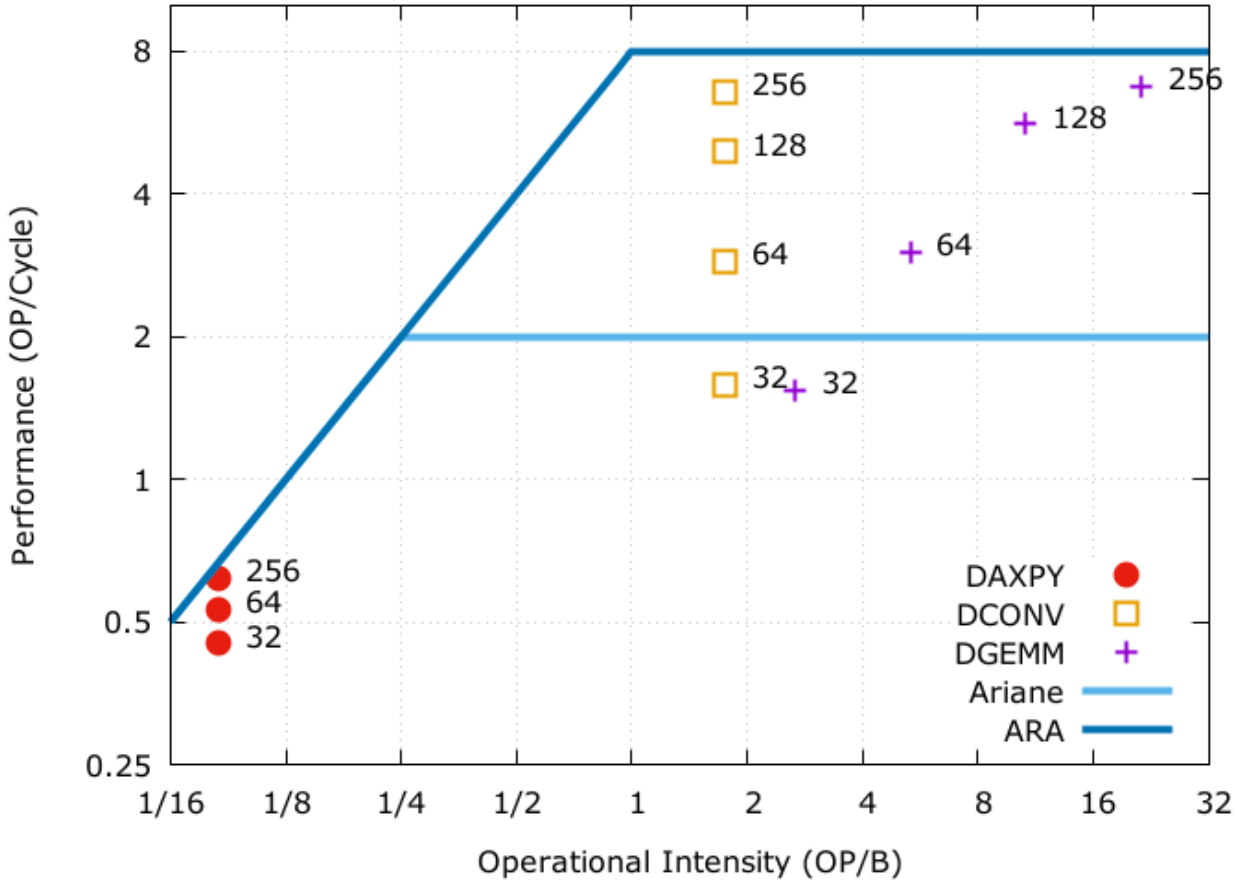
Vector Length	FPU Utilization (%)	Performance (op/cycle)
32	19,2%	**1,54**
64	37,8%	**3,02**
128	70,3%	**5,62**
256	*84,7%*	***6,77***
512	*85,5%*	***6,84***
1024	*86.3%*	***6,91***



- We see the same phenomena seen with DCONV
  - Initial banking conflicts limiting performance with shorter vectors
  - Strip-mining and unmaskable memory transfers limiting steady performance

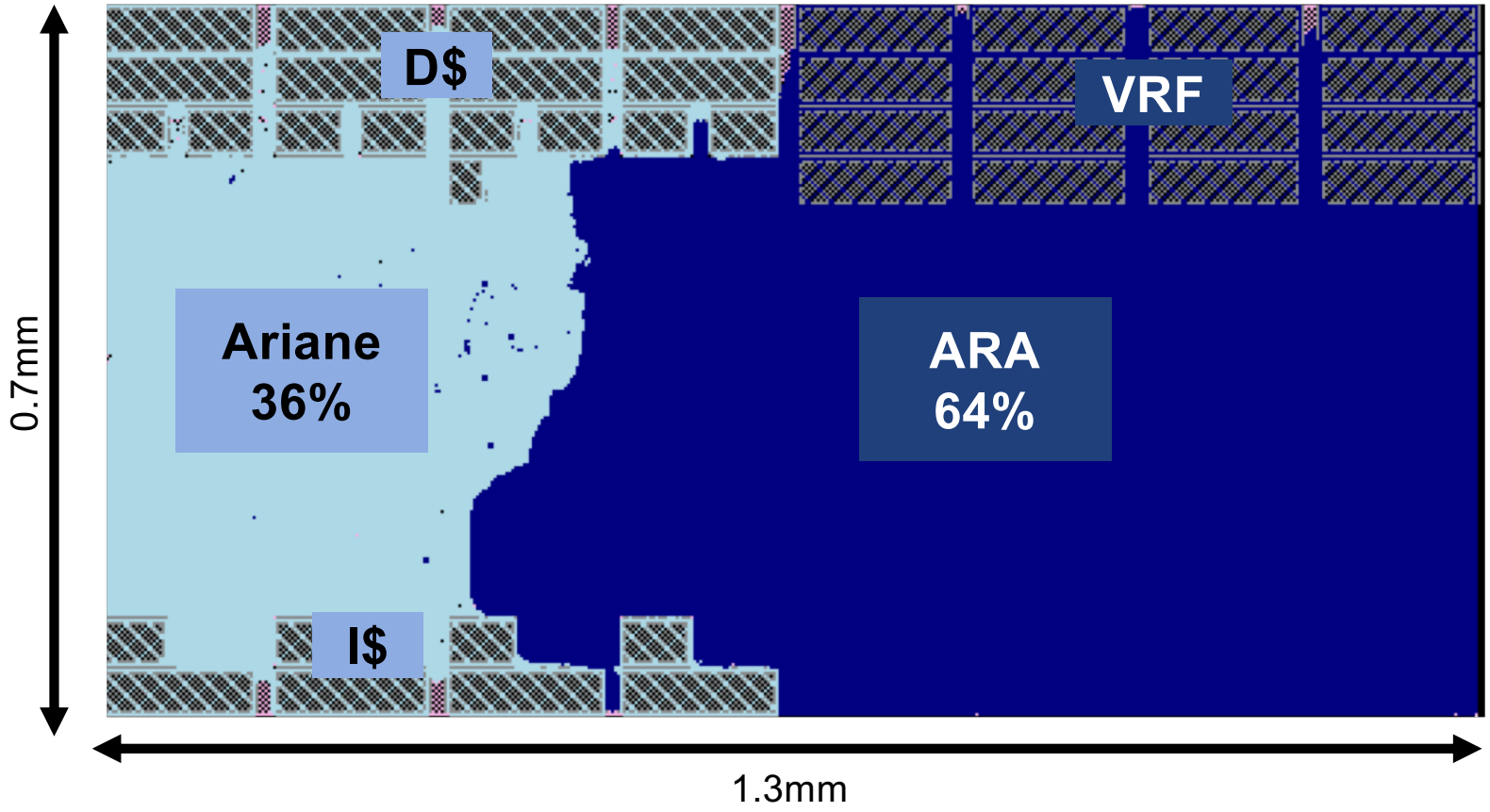
PULP
Parallel Ultra Low Power

So.. can we achieve 8 GFLOPs?



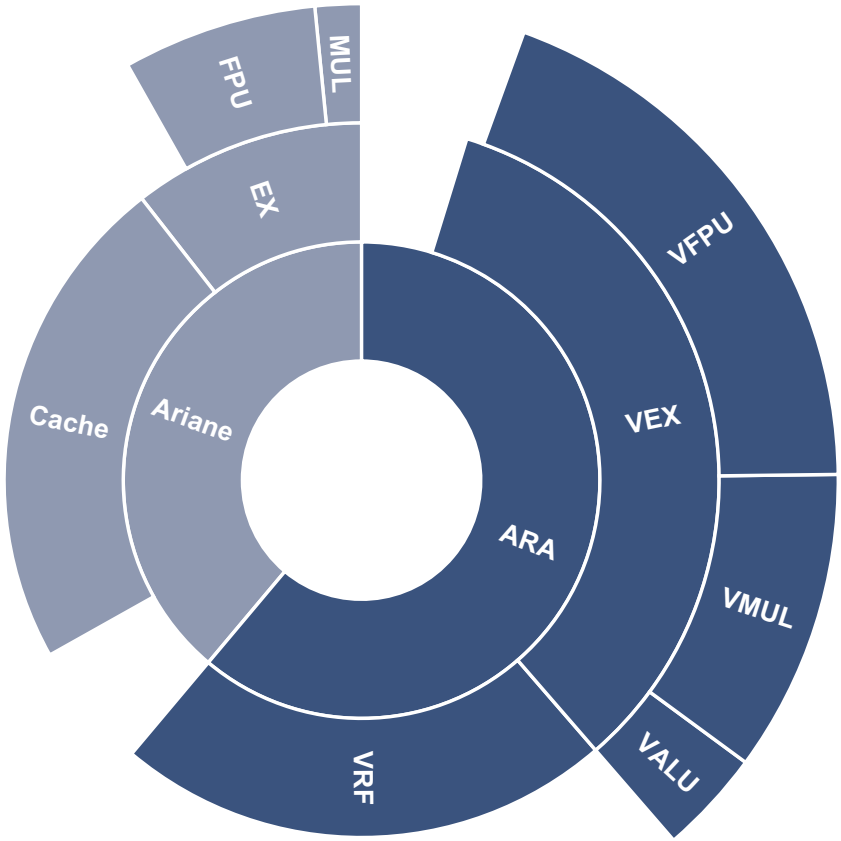
Implementation results

ARA: GF FDX22 1GHz implementation (SS, 0.72V, 125 ºC)



Critical path
45 gates

Ara and Ariane – Area breakdown



- ARA is $1.8\times$ bigger than Ariane…
- and has $4\times$ its computational power

- Operation density:
  - Ariane: $7{,}27\ \mathrm{GFLOPS/mm^2}$
  - ARA: $16{,}23\ \mathrm{GFLOPS/mm^2}$

PULP
Parallel Ultra Low Power

Conclusions

Shuffling instructions

- Higher operational intensity → minimize data transfers
  - By shuffling and reordering data inside vector registers
- Only two* instructions available
  - `vslide`: vd{i} = vs1{i + rs2}
  - `vrgather`: vd{i} = vs1{ vs2{i} }
- Register-gather is too general → hard to optimize!
- Dedicated instructions to more specific shuffling: permutations, rotations?

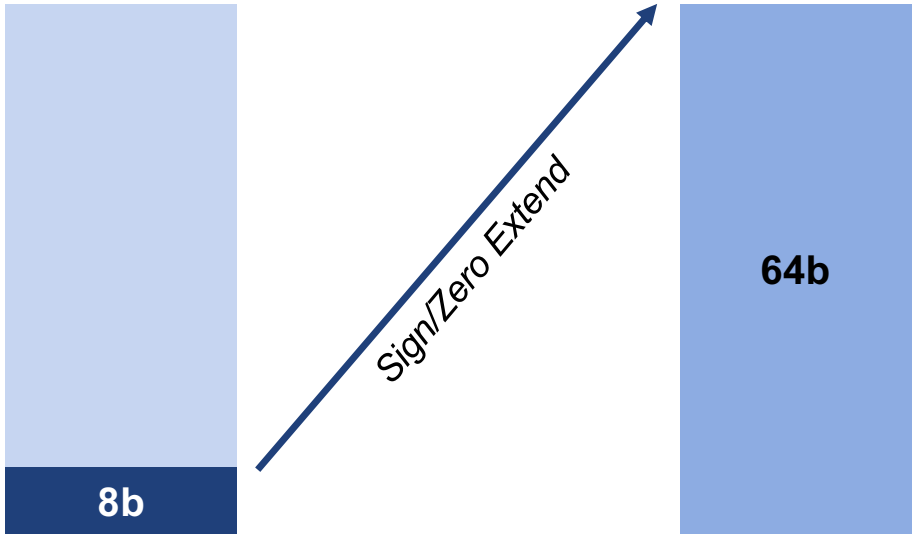
*(three, more recently, as `vslide` was split into `vslideup` and `vslidedown`)

Decoupling between scalar and vector units

- We did benefit from decoupling the scalar and the vector unit
- Different “worlds”
  - Scalar unit: speculative, one instruction issued per cycle, several in-flight instructions
  - Vector unit: non-speculative, latency-tolerant, high throughput, a few in-flight vector instructions
- We see with apprehension ISA decisions that push towards their recoupling
  - E.g., the recent decision of mapping the vector registers over the floating-point registers

Mixed-precision

- ARA supports mixed-precision *to a certain extent*
- Previous versions allowed for a mixed-precision instruction as $64\text{b} \leftarrow 8\text{b} + 8\text{b}$
  - 8b, 16b and 32b operands could be promoted to 64b operands
  - High hardware cost!
- We now allow for a more restricted set of type promotions
  - $8 \rightarrow 16\text{b}$, $16 \rightarrow 32\text{b}$ and $32 \rightarrow 64\text{b}$
  - Aligned with newer revisions of the V Extension



PULP
Parallel Ultra Low Power

Wrapping up...

- ARA: 64-bit vector processor

RISC-V Summit

ARA: 64-BIT RISC-V VECTOR IMPLEMENTATION IN 22NM FDSOI

Matheus Cavalcante
PhD Student
ETH Zurich

Fabian Schuiki
PhD Student
ETH Zurich

https://tmt.knect365.com/risc-v-summit