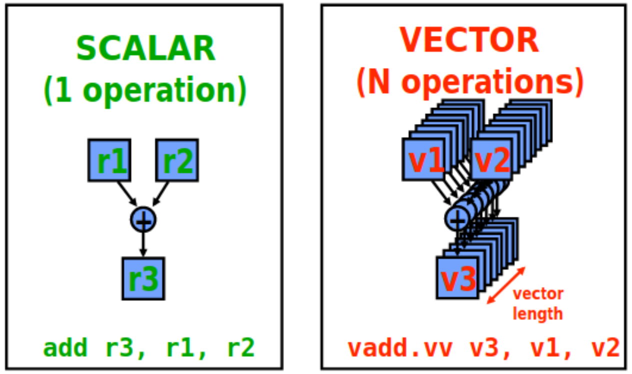
Week 9

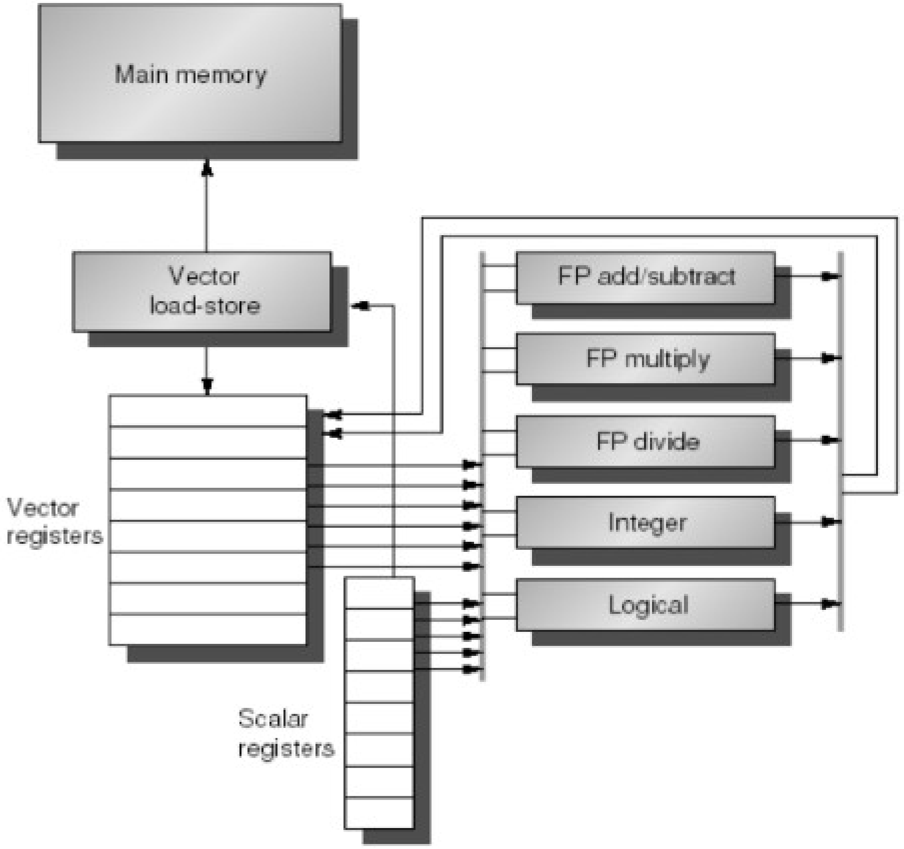
Agenda:

1. Vector Processors and RISCV Vector Extension
2. SiFive Vector Processor

Vector Processors



Source : Patterson Lecture , Vector Processor

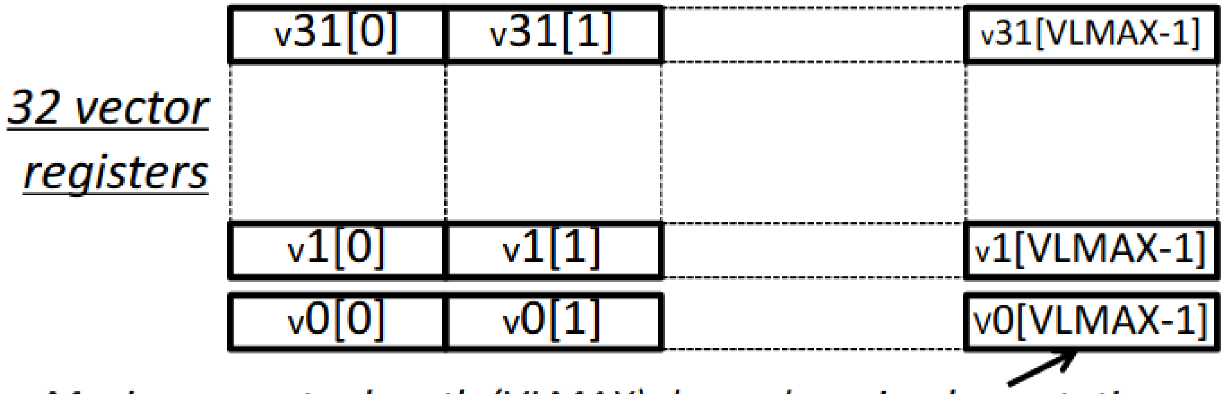


Source : Patterson Hennesy, Comp Arch Text

<u>Constituents of Vector Processor</u>

1. **Vector Register File:** Each vector register is fixed length bank holding a single vector
   - Has atleast 2 read and 1 write ports
   - 32 Vector Regs in RISCV – V Spec
2. **Vector Functional Unit(FU):** Fully pipelined to start new operation every clock. Control unit to manage hazards
3. **Load Store Unit(LSU):** Fully pipelined unit to amortize the memory latency over one vector load.
4. **Crossbars** to connect LSU and FU

RISC-V Foundation Vector Extension Overview



Maximum vector length (VLMAX) depends on implementation, number of vector registers used, and type of each element.

- Unit-stride, strided, scatter-gather, structure load/store instructions
- Rich set of integer, fixed-point, and floating-point instructions
- Vector-vector, vector-scalar, and vector-immediate instructions
- Multiple vector registers can be combined to form longer vectors to reduce instruction bandwidth or support mixed-precision operations (e.g., 16b*16b->32b multiply-accumulate)
- Designed for extension with custom datatypes and widths

Vector CSRs

vtype

Vtype sets width of element in each vector register (e.g., 16-bit, 32-bit, ...)

vl

Vector length CSR sets number of elements active in each instruction

vstart

Resumption element after trap

fcsr (vxrm/vxsat)

Fixed-point rounding mode and saturation flag fields in FP CSR

RISC V Vector ISA

- 32 vector registers (v0 ... v31)
- Each register can hold either a scalar, a vectoror a matrix(shape)
- Each vector register can optionally have an associated type(polymorphic encoding)
- Variable number of registers (dynamically changeable)
- **Vector instruction semantics**
- All instructions controlled by Vector Length (VL) register
- All instructions can be executed under mask
- Intuitive memory ordering model
- Precise exceptions supported
- **Vector instruction set:**
- All instructions present in base line ISA are present in the vector ISA
- Vector memory instructions supporting linear, strided & gather/scatter access patterns
- Optional Fixed-Point set
- Optional Transcendental set

Week 10: Oct 2nd 2020

Actions

1. Study RISC-V Vector Extension
2. Case Study RISCV Vector Processors

Work Done

1. Study RISC-V Vector Extension Spec
2. Detailed study of Ara RISC-V Vector Processor

Ara : RISC-V Vector Processor

Overview

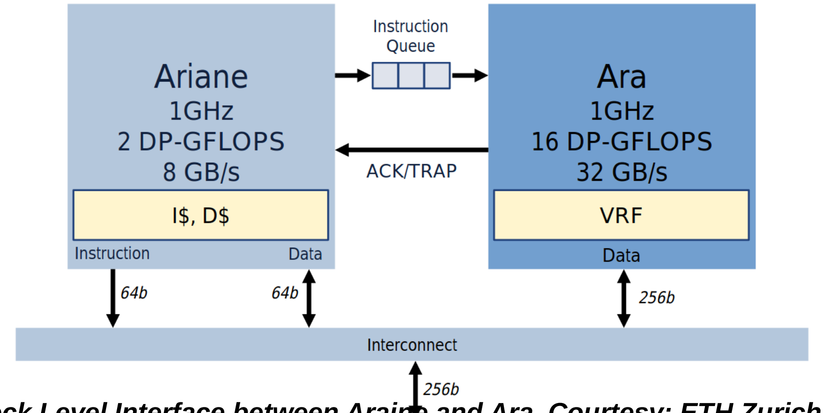
- Developed by PULP team, Integrated Systems Laboratory, ETH-Zurich in 2019
- 64-bit vector co-processor to Ariane (which is RV64GC application processor)
- Based on RISC-V Vector Spec. Version 0.5
- Operating Frequency 1 Ghz
- Designed for low power
- Contains Floating Point Support with 64 bit/cycle throughput (1 DP / 2 SP / 4 HP ouptuts/cycle)
- Peak Performance : 33 DP – GFLOPS (41 DP-GFLOPS/W)
- Exploits Data Level paralellism for efficient execution of Scientific and Matrix oriented computations, as well as DSP and Machine learning algorithms

Topics of discussion

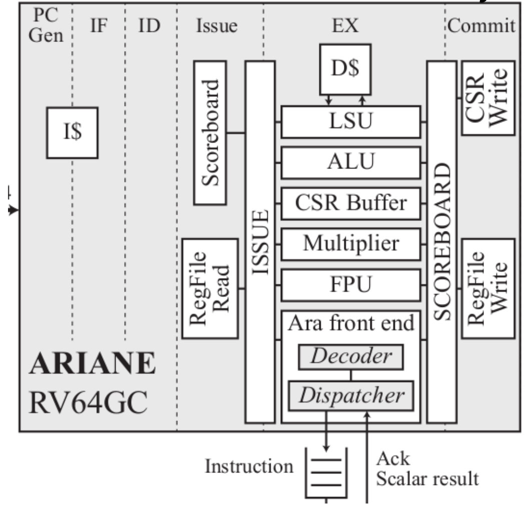
- ARA Microarchitecture
- ARA Performance Benchmarking

Ariane Vector Processor System

- Ariane is a RV64GC based in-order , single issue application class processor having a 6-stage pipeline
- Vector instructions are partially decoded in Ariane’s Instruction Decoder, and then completely in a dedicated Vector Instruction Decoder insider Ara.
- The dispacher controls the interface between Ara and Ariane’s dedicated Scoreboard port.
- While Ariane sends Instructions, Ara sends Ack/ Scalar results through the Scoreboard port.
- The decoupled execution works well, except when Ariane expects a result from Ara, e.g., reading an element of a vector register



Block Level Interface between Araine and Ara. Courtesy: ETH Zurich ppt



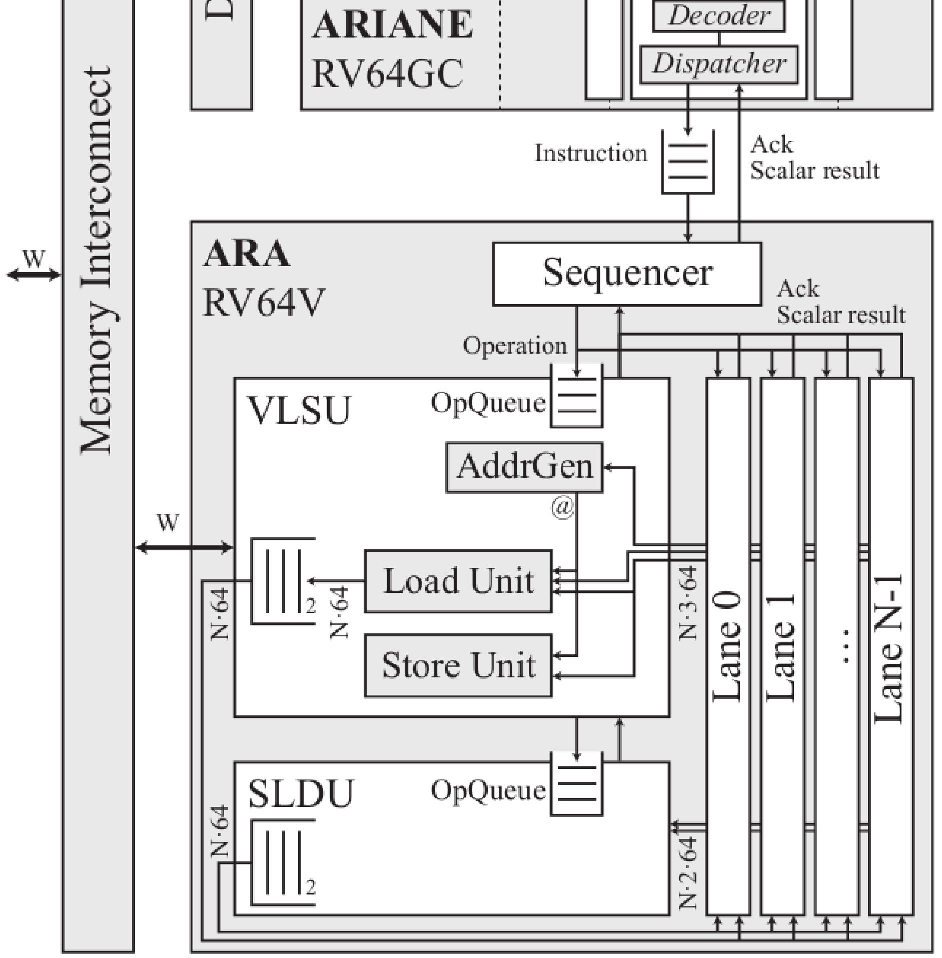
ARA RV64V

ARA RV64V

Fine Grain Block Level Interface between Ariane and Ara. Courtesy: Ara Paper

Components of Ara

- Sequencer
- Slide Unit (SLDU)
- Vector Load Store unit (VLSU)
- Vector Lanes
  - Lane Sequencer
  - Vector Register File
  - Operand Queues
  - Execution Units



(a) Block diagram of an Ara instance with N parallel lanes. Ara receives its commands from Ariane, a RV64GC scalar core. The vector unit has a main sequencer; N parallel lanes; a Slide Unit (SLDU); and a Vector Load/Store Unit (VLSU). The memory interface is W bit wide.

Fig. 3. Top-level block diagram of Ara.

Sequencer

- Responsible for keeping track of vector instructions running on Ara, dispatching them to different execution units and acknowledging them with Ariane.
- This unit has a global view of the instruction execution execution progress across all lanes.
- The sequencer can handle upto 8 parallel instructions
- Hazards among pending vector instructions are resolved by this block.
- In case of a structural hazard, the sequencer delays the issue of vector instructions until the contending instruction releases the shared resource.
- Sequncer stores information about which vector instruction is accessing which vector register. This information is used to resolve data hazards between instructions

Slide Unit (SLDU)

- Vector Register File (VRF) is spread across all lanes. Thus, a given lane cannot access all VRF banks directly.
- SLDU is responsible for handling instructions that must access all Vector Register File (VRF) banks at once.
- Handles tasks such as insertion of an element into a vector, extraction of an element from a vector, vector shuffles, vector slides etc.,
  - vdest[i] <-- vsrc[i + slide_amount]
- **Possible area of improvement**: Vector Reducion operations are given in the latest RISC-V spec. These have not been implemented in Ara. Vector Reuctions are managed using SLDU.

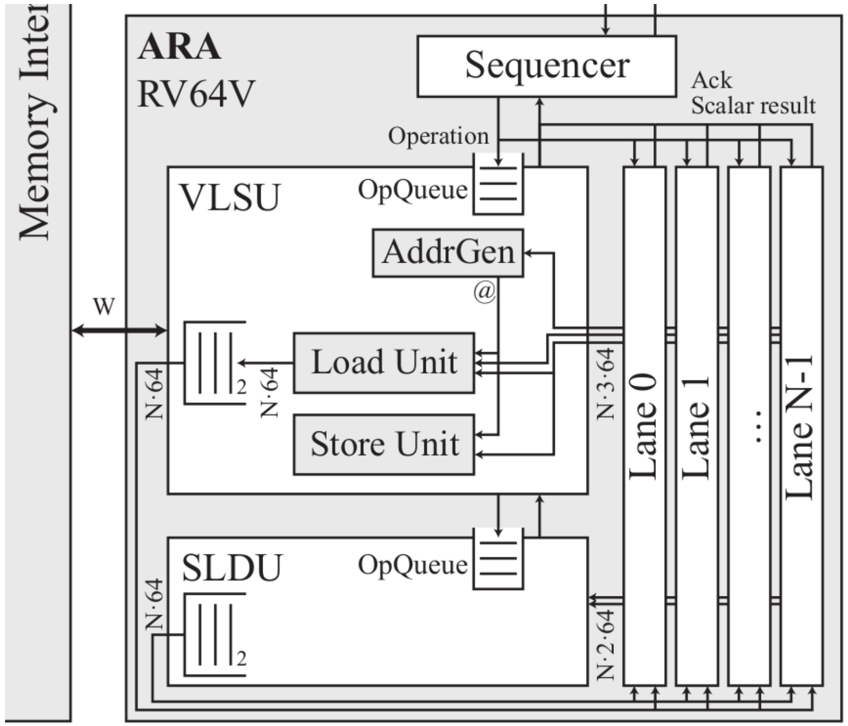
Vector Load Store Unit (VLSU)

- Ara has a single memory port.
- VLSU has an address generator to generate address sequences for :

  1. Unit Stride loads and stores, which access a contiguous chunk of memory

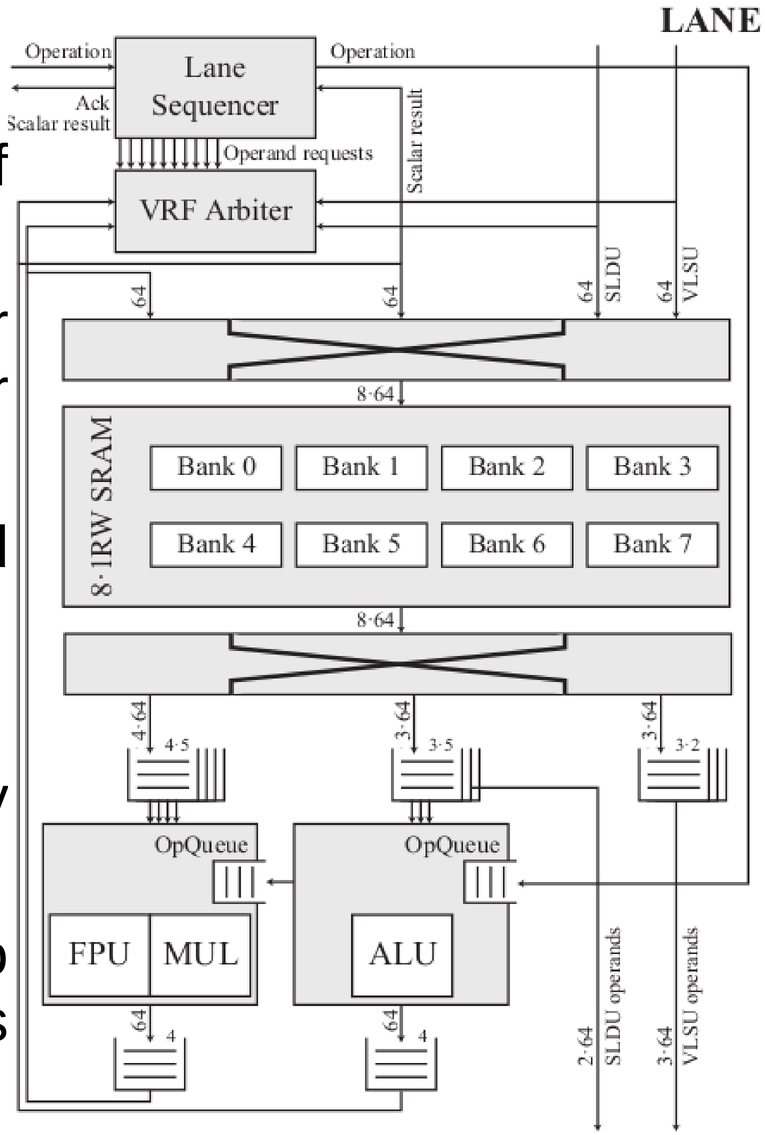
  2. Constant Stride memory Operations, which access memory address spaced by a fixed offset.

  3. Scatters and Gathers , which uses a vector of offsets to allow general access patters.
- This block converts unit-stride memory requests into burst requests through AXI interface, instead of accessing all elements individually.



Vector Lanes : Organization

- Ara can be configured with variable number of identical lanes
- Each lane has its own *lane-sequencer* for keeping track of upto 8 parallel vector instructions.
- Each lane contains part of Ara's whole *VRF* and *execution units*.
- *VRF arbiter* controls access to VRF
- Inter-lane communication is only required by VLSU and SLDU
- Each lane has a command interface attached to the main sequencer, through which the lanes indicate they finished execution of an instruction.

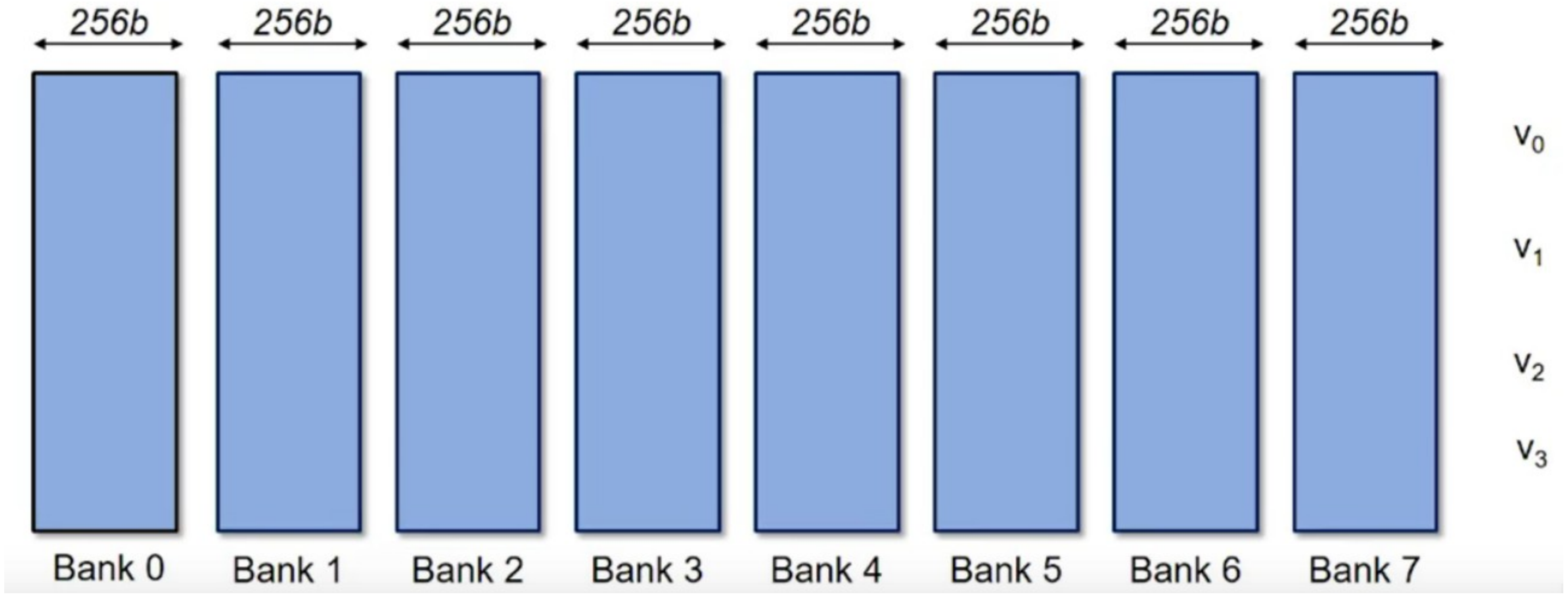


Lane Sequencer

- Responsible for issuing vector instructions to the functional units, controlling theor execution in the context of a single lane.
- Generate requests to read operands from VRF
- Upto 10 operand requests can be generated which are given to the VRF arbiter.
- In case of data hazards, the operand request rate is lowered as there is NO FORWARDING logic.

Vector Register File

- Assume there are 4 lanes (N = 4). Reg. File contains 256b banks. However, physically, each lane contains only 1/4 th of each bank.
- Assuming there re only 4 Vector Registers in the VRF, One bank stores 4 operands consumed in parallel by 4 lanes.

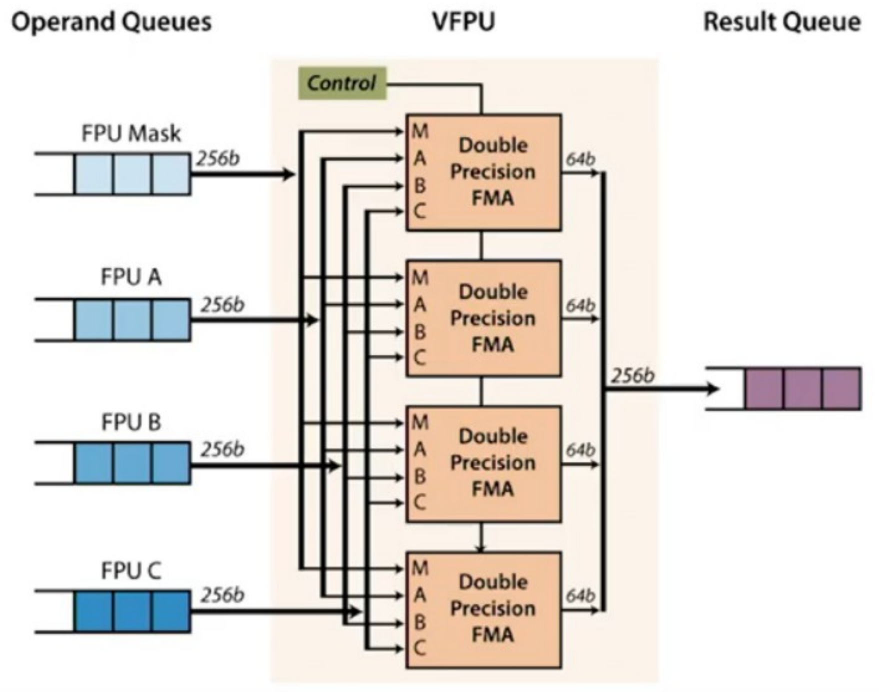


VRF Bank Organization. Courtesy: Ara RISC-V Conference ppt

VRF Bank Organization. Courtesy: Ara RISC-V Conference ppt

Most Complex Instruction: Fused Multiply Add (FMA instruction)

- FMA insructions contain Four operands
- vd[i] = lsb(vmask[i]) ? vsa[i] + vsb[i] + vsc[i] : 0;
- FMA is pipelined (5 cycles) to meet fminconstraint
- The four lanes operate in *lockstep*
  - Low control overhead
- Each lane gets 64b operands from four 256b input FIFO buffers (A, B, C, VMASK)
  - Number of lanes determines buffer width
-

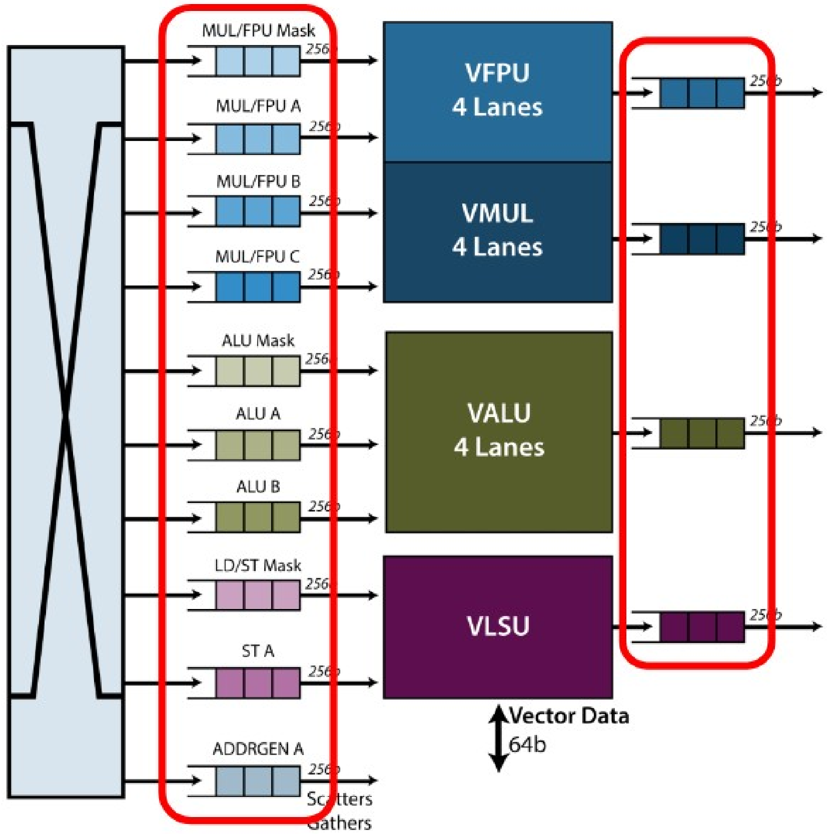


Vector Register File

- Multi-poted Registers are essential for a VRF as several instructions in the pipeline or sevaral data in the same instruction may access the same RF elements independently.
- Due to physical design implemention contraints, Ara’s VRF is implemented as a set of single-ported (1RW) banks.
- The width of each bank is 64-bits
- In RISC-V’s vector instruction, the predicated multiply and add (FMA) instruction is the worst case regarding throughput, which reads 4 operands to produce one result and takes 5 clock cycles(details in next slide)
- Therefore, in steady state, 5 banks are accesses simultaneously to sustain maximum throughput for FMA instruction.
- Ara’s VRF has 8 banks per lane.

Operand Queues

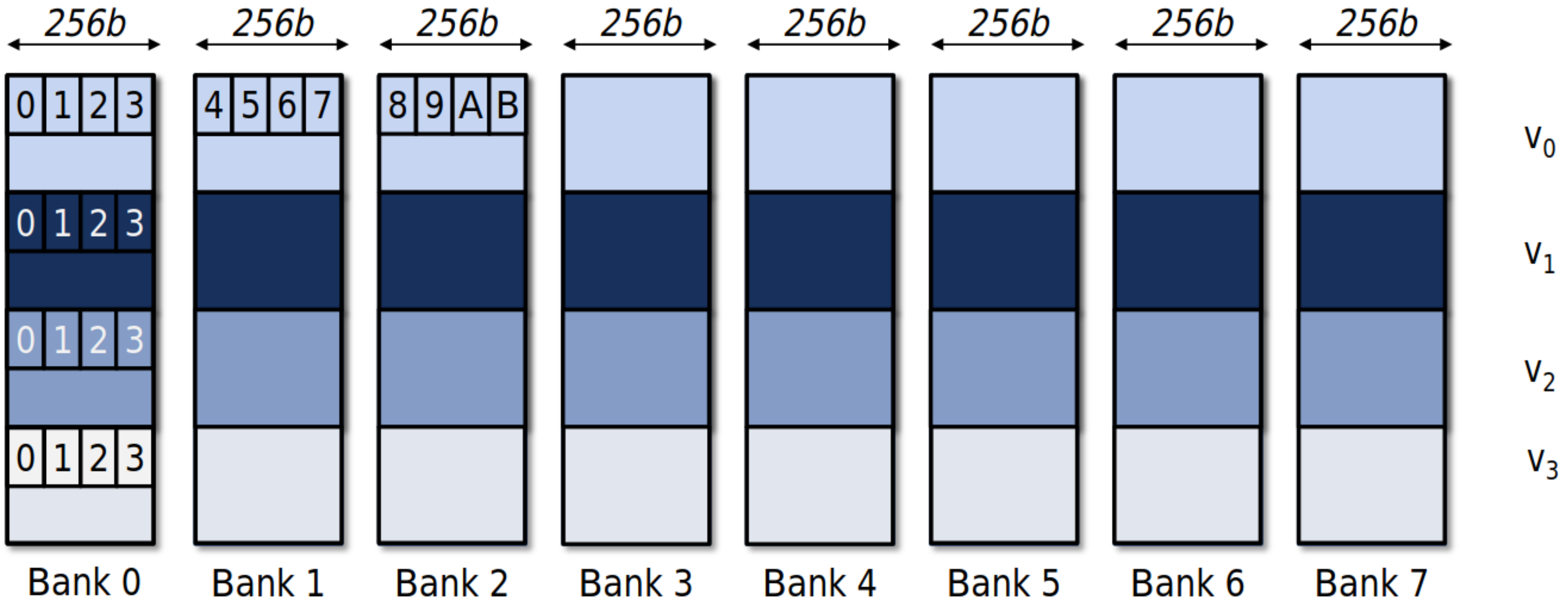
- Queues needed to sustain maximum throughput fo the lock-step operation of the FUs, while hiding th latency caused by banking conflicts in the VRF
- One input queue buffer provides one operand to a the (four) lanes
- 256b (4x64b) wide entries
- One FIFO buffer per operand per multi-lan datapath unit
- 10 FIFO buffers
- Output queue buffers for output operands, one pe multi-lane datapath unit



Operand Q : Courtesy : Ara RISCV Summit

Operand Q : Courtesy : Ara RISCV Summit

Data Organization in VRF

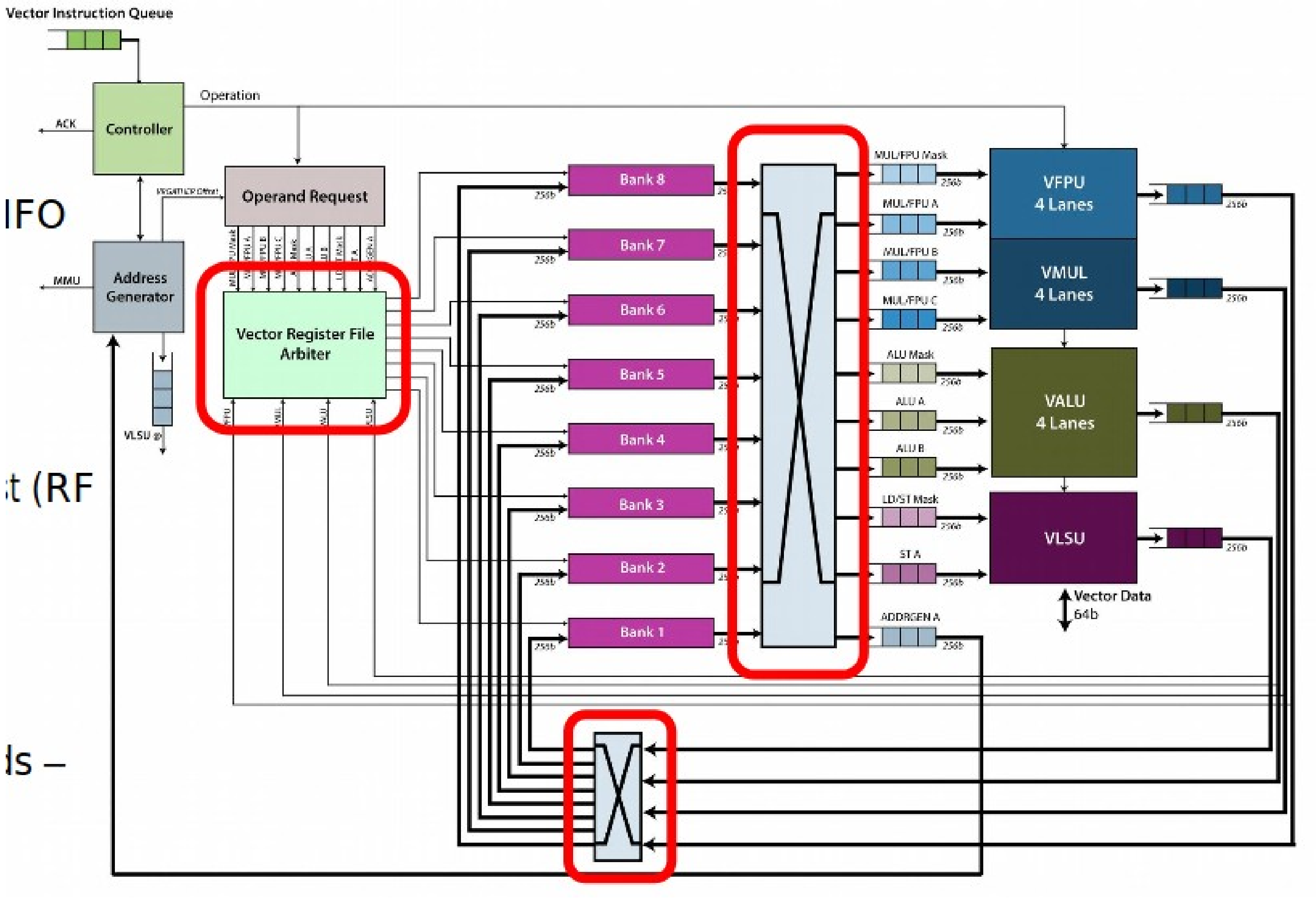


Vector Register File and Operand-Deliver Interconnect

IFO

t (RF

ds –



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
  - VRF is built as 1RW SRAM bank
  - $PM > PA > PB > PC$
  - Writes have lower priority than reads –unless output queue is full

Computation Cycle – Execution of an FMA Instruction

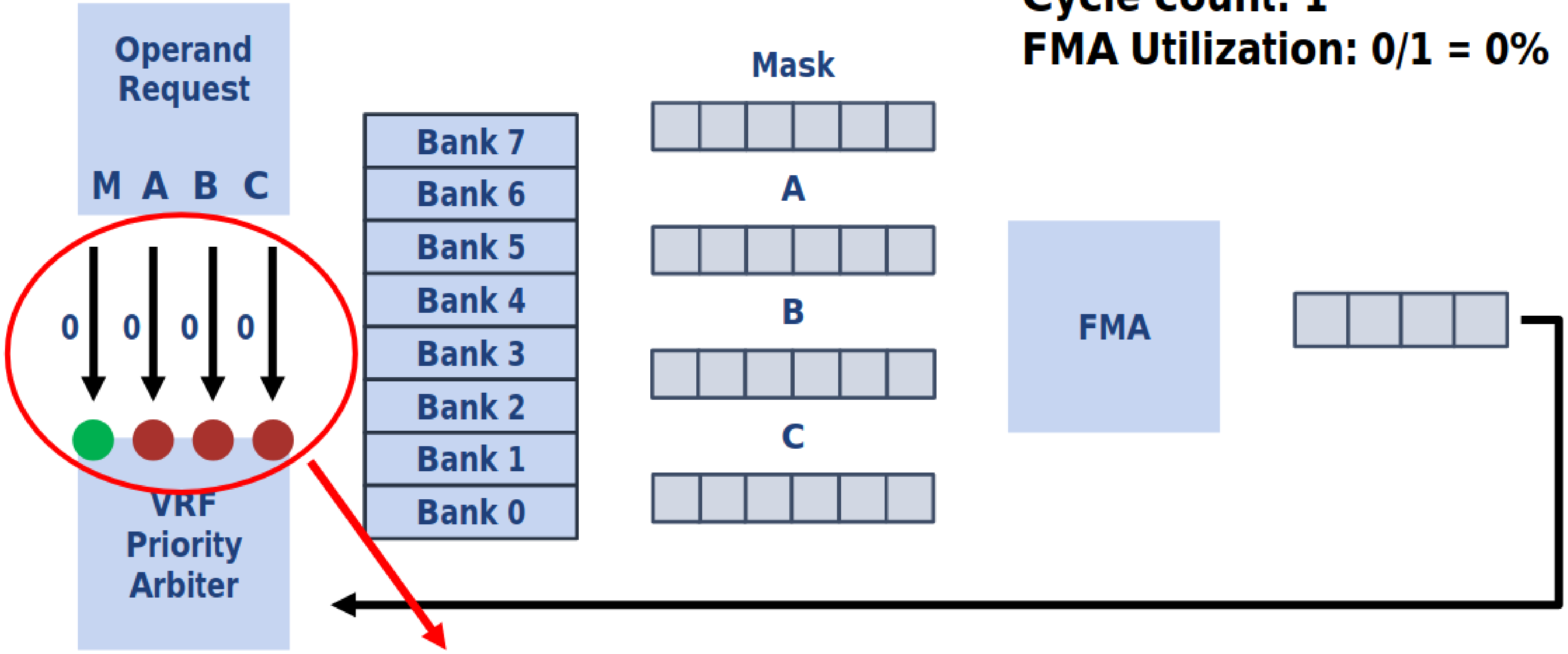
- Consider the execution of the following instruction:

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

- We take a vector of length = 256
- What is the execution time?
- Ideally, with 4 lanes, it should be 256/4 = 64
- However, due to overheads due to pipeline start, register bank clashes, it will be be slightly more.

Execution of a FMA instruction

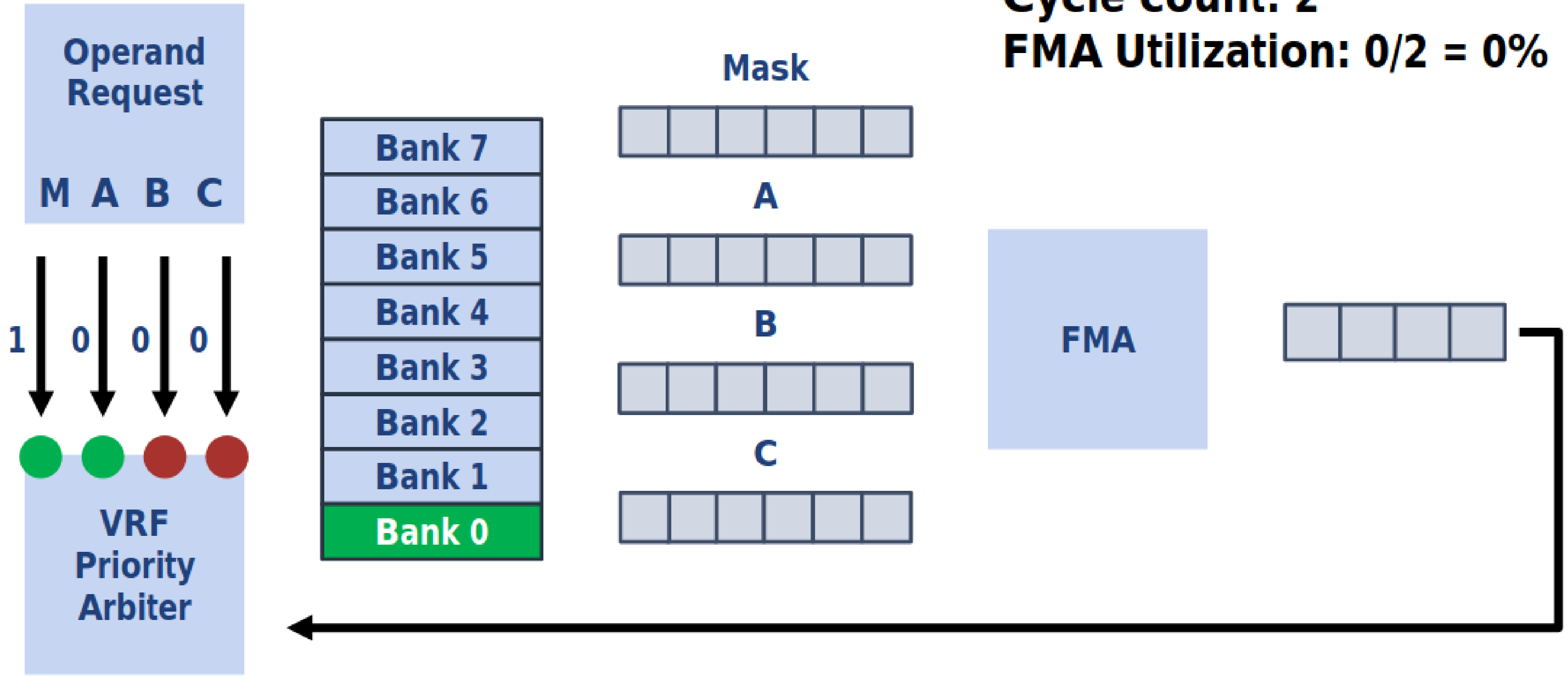
Cycle count: 1
FMA Utilization: 0/1 = 0%



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

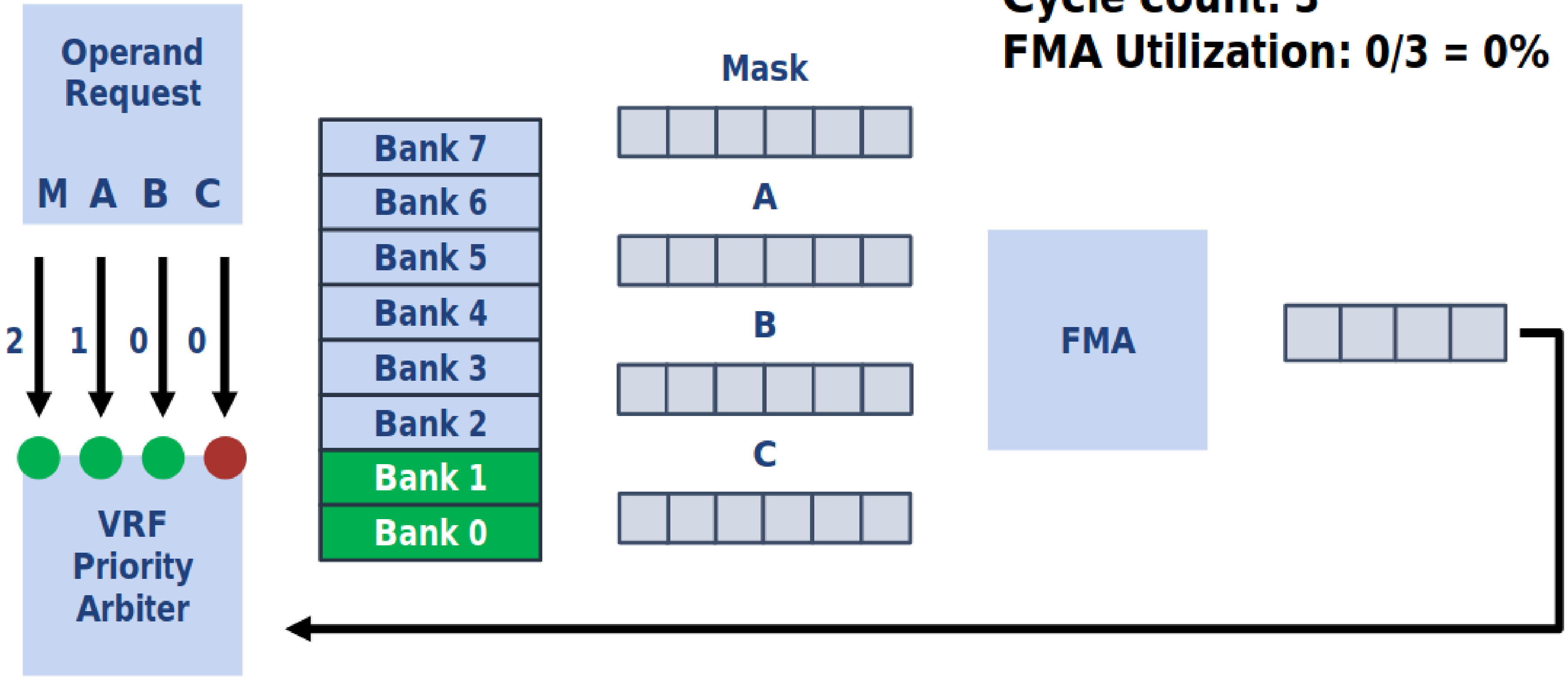
Execution of a FMA instruction

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

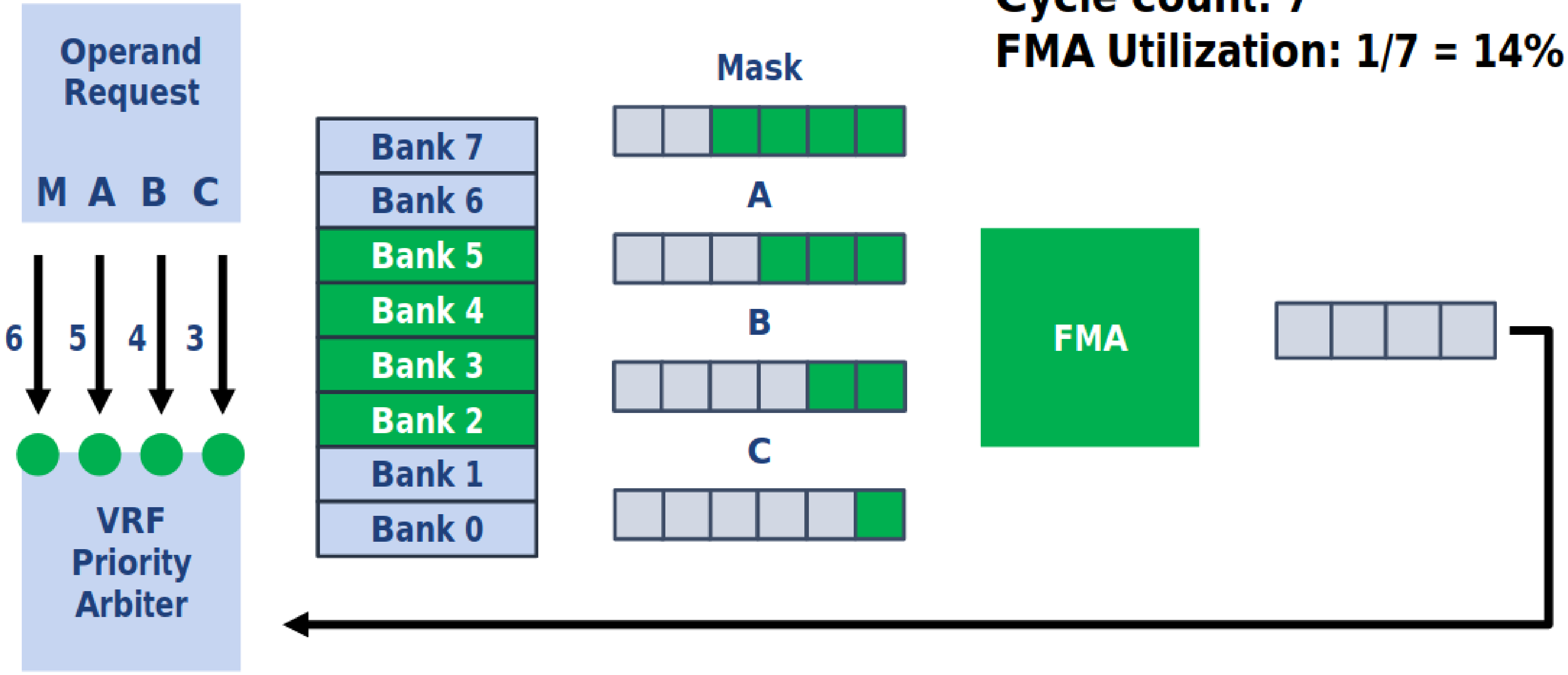


Execution of a FMA instruction

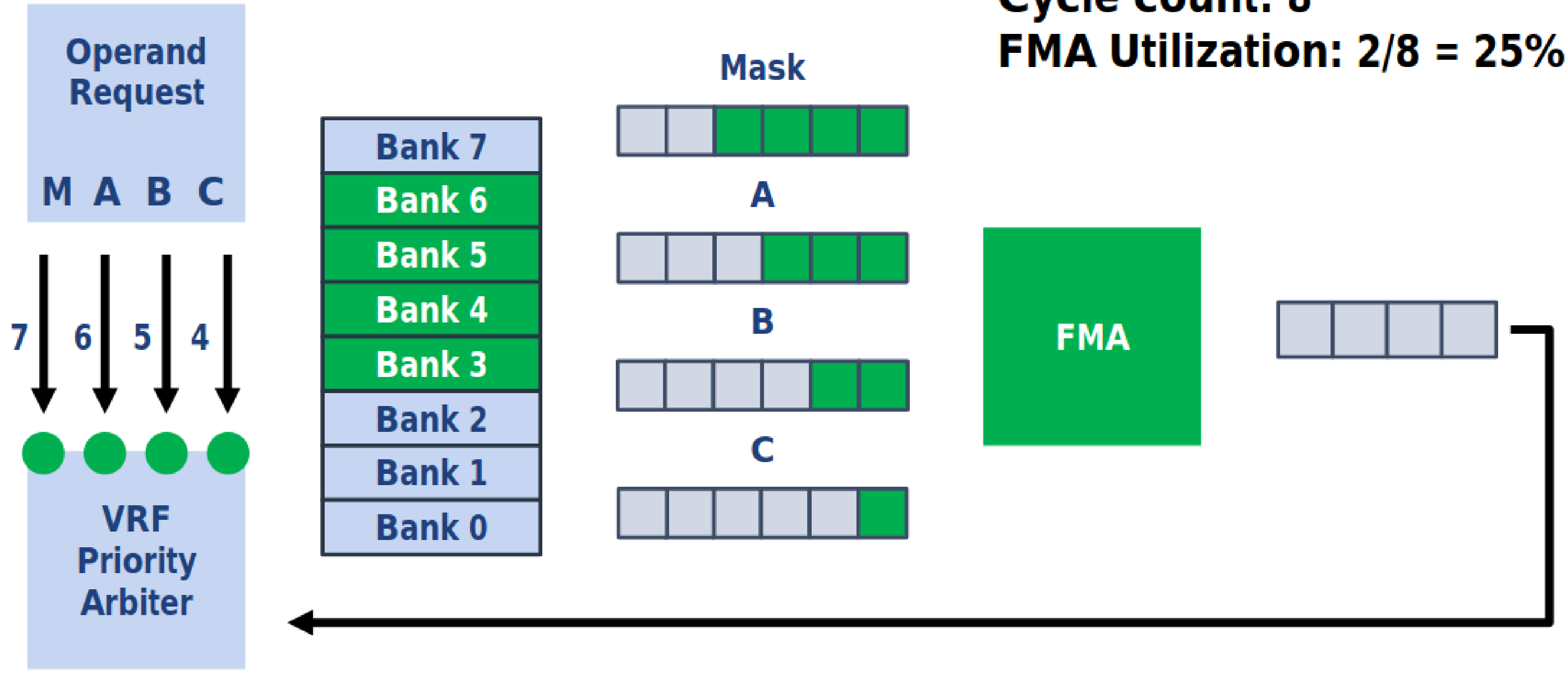
Cycle count: 3
FMA Utilization: 0/3 = 0%



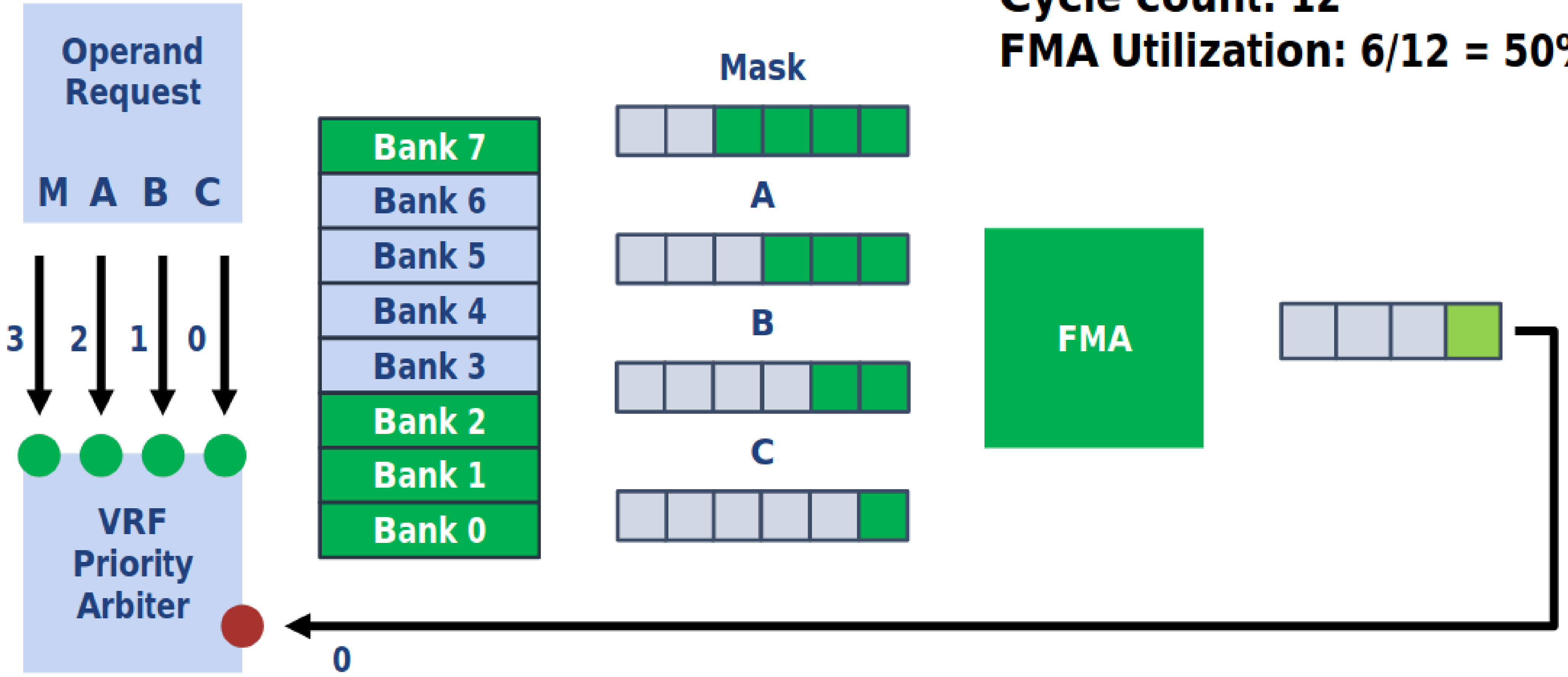
Cycle count: 7
FMA Utilization: 1/7 = 14%



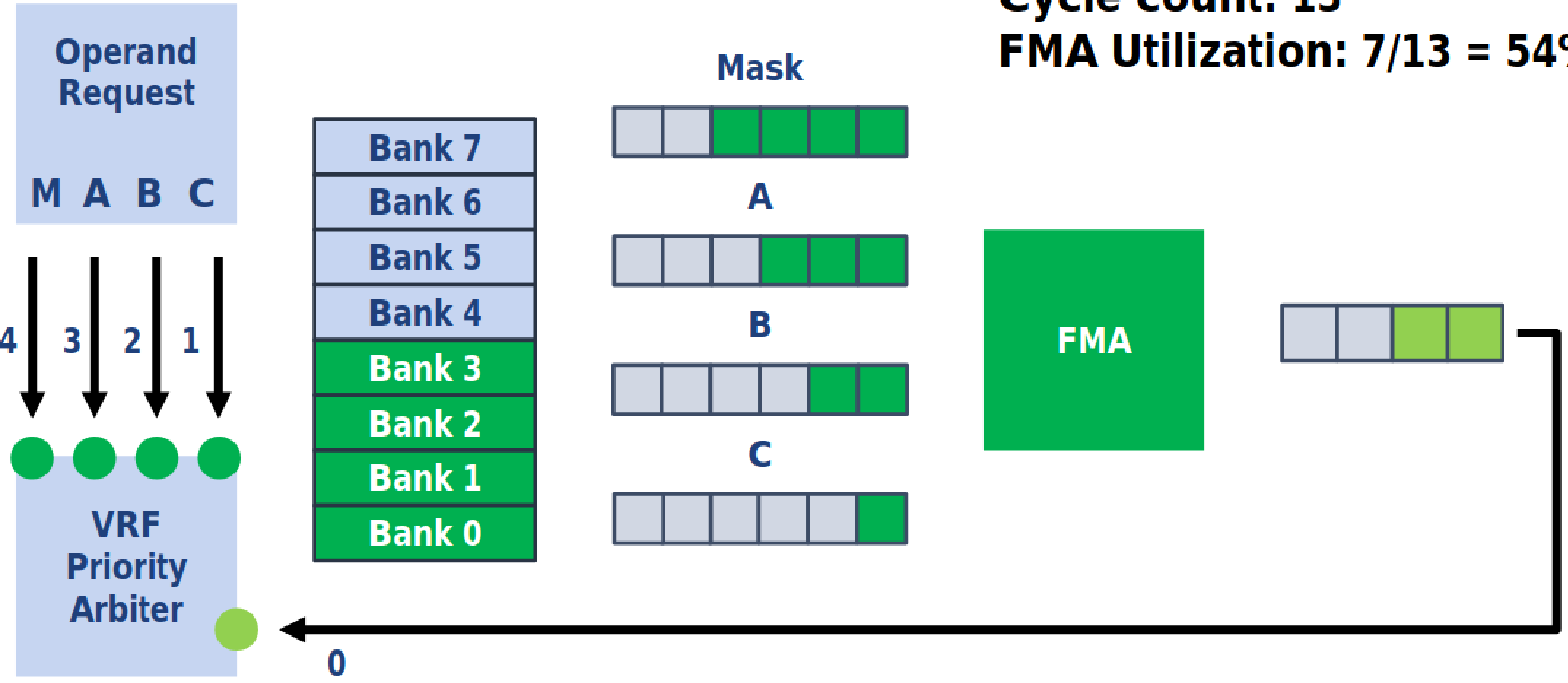
Cycle count: 8
FMA Utilization: 2/8 = 25%



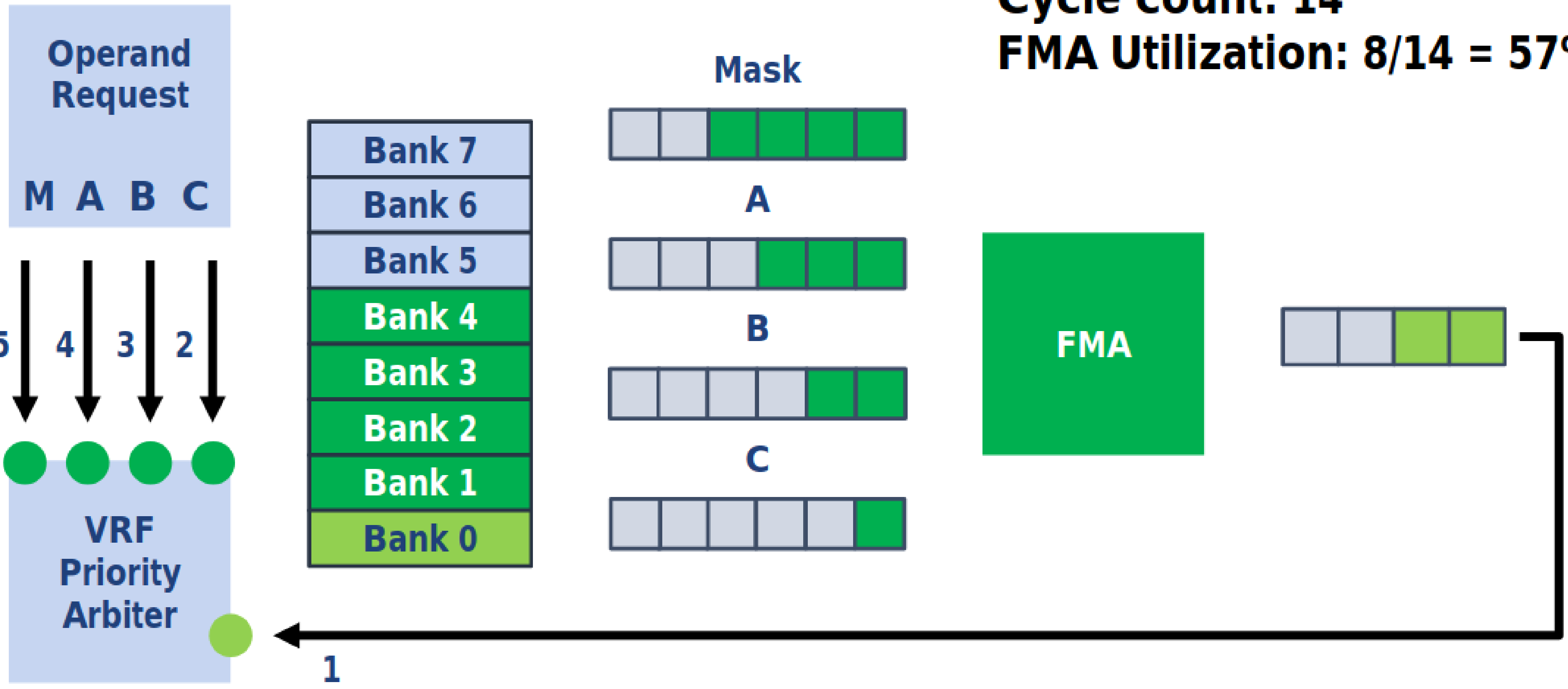
Cycle count: 12
FMA Utilization: 6/12 = 50%



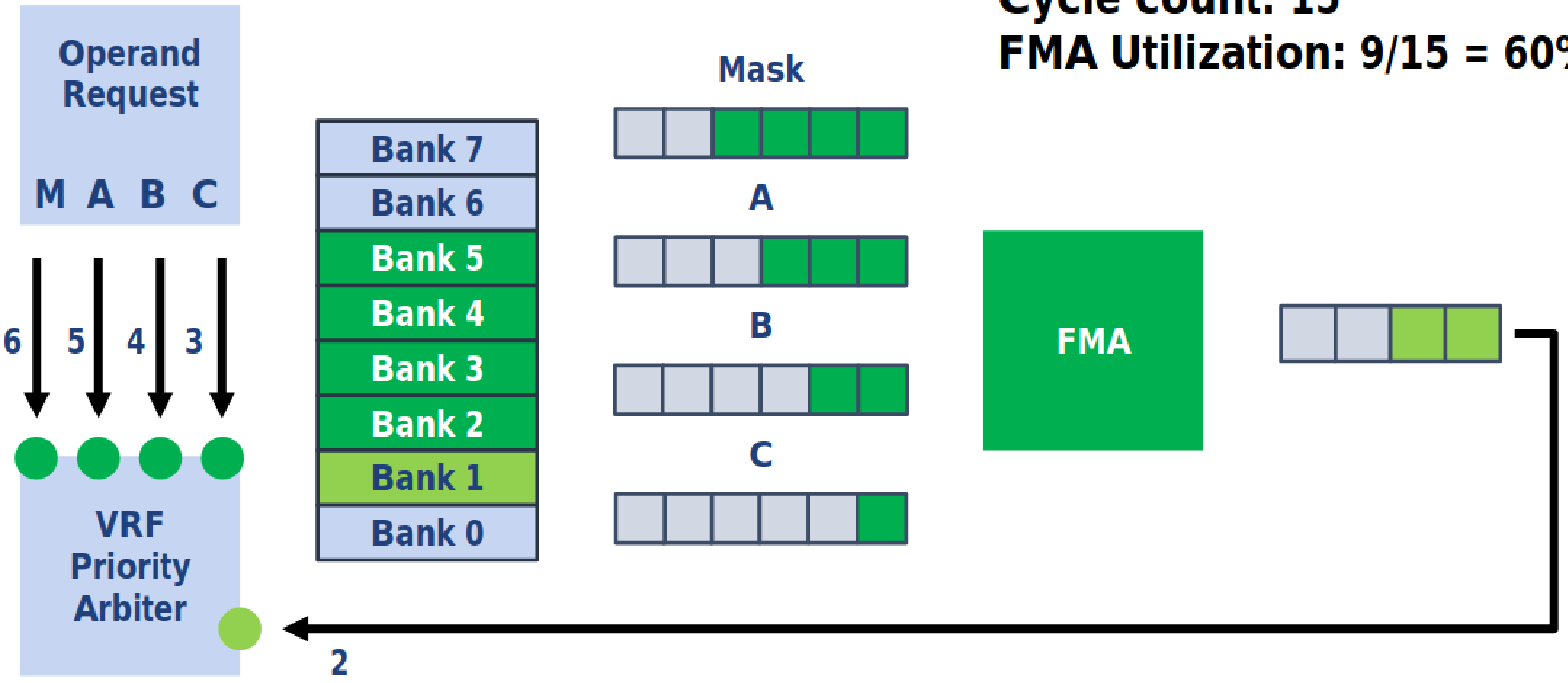
Cycle count: 13
FMA Utilization: 7/13 = 54%



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

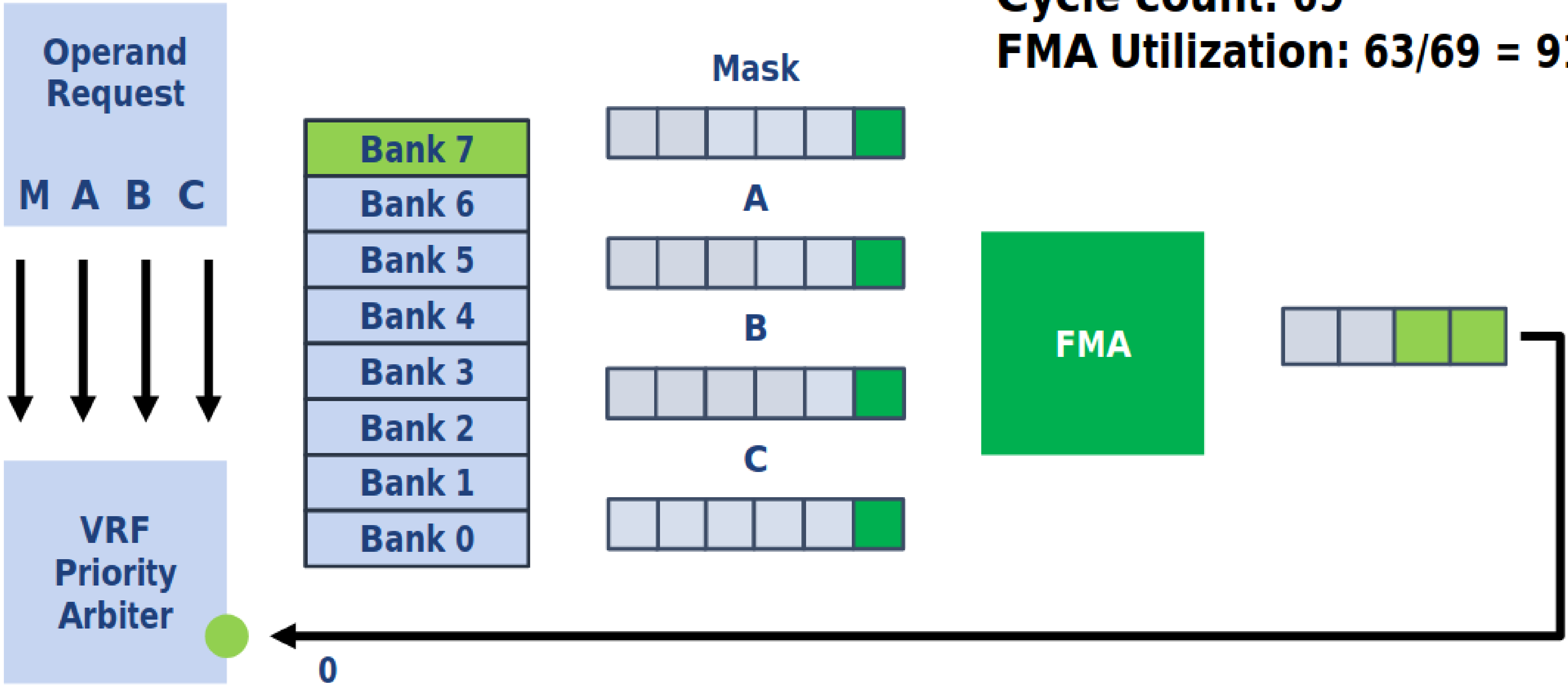


Cycle count: 15
FMA Utilization: 9/15 = 60%



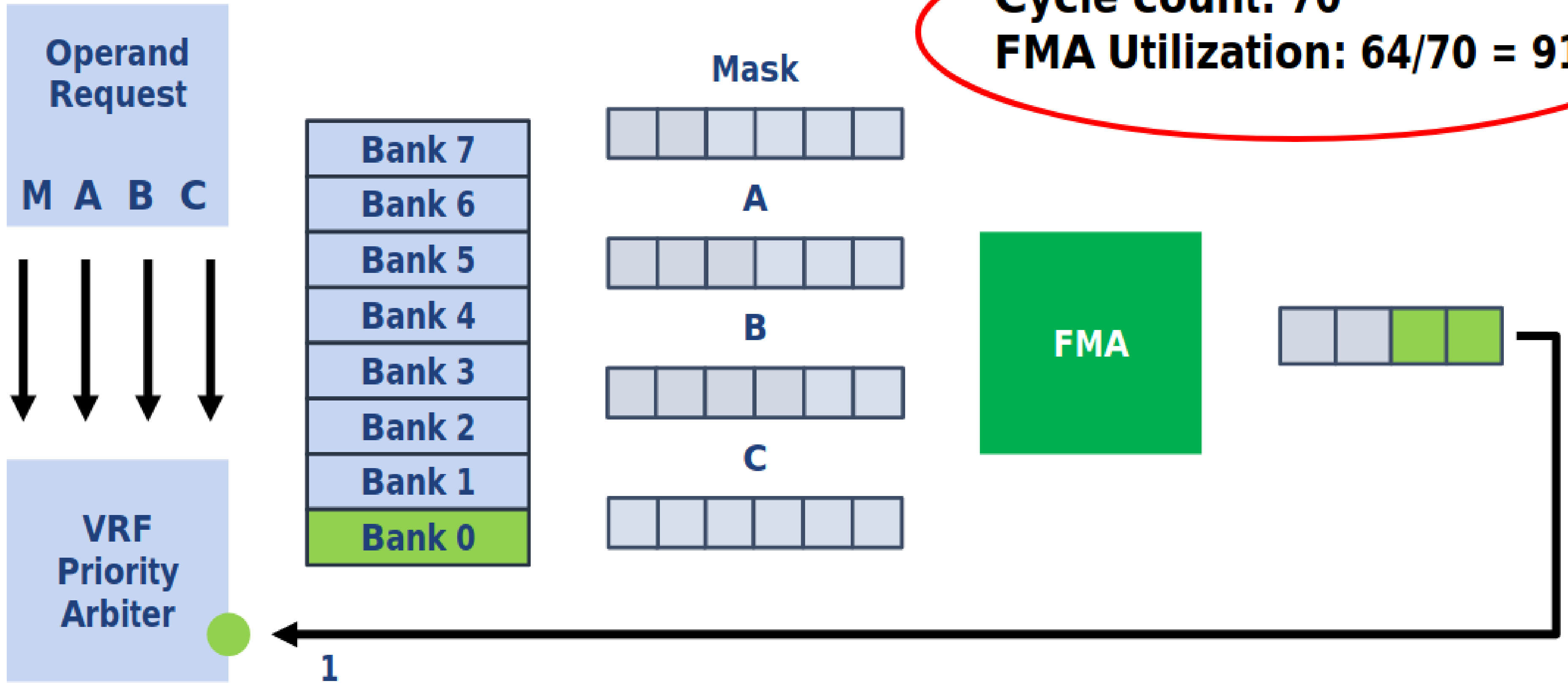
Cycle count: 69

FMA Utilization: 63/69 = 91%



Execution of a FMA instruction

Cycle count: 70
FMA Utilization: 64/70 = 91%



Benchmarking

The relationship between processor performance and memory bandwidth can be analyzed with the *roofline model.*

- Depicts the peak achieveable performance (in OP/cycle) as a function of the arithmetic intensity I, defined as the algorithm- dependent ratio of operations per byte of memory traffic.
- According to this model, computations can be either compute-ound or memory bound
- The peak performance is achieveable only if the algorithm’s arithmetic intensity, in operations per byte, is higher than processor’s performance per memory bandwidth ratio.
- For Ara, it enters compute-bound region when arithmetic intensity > 0.5 DP-FLOP/B.

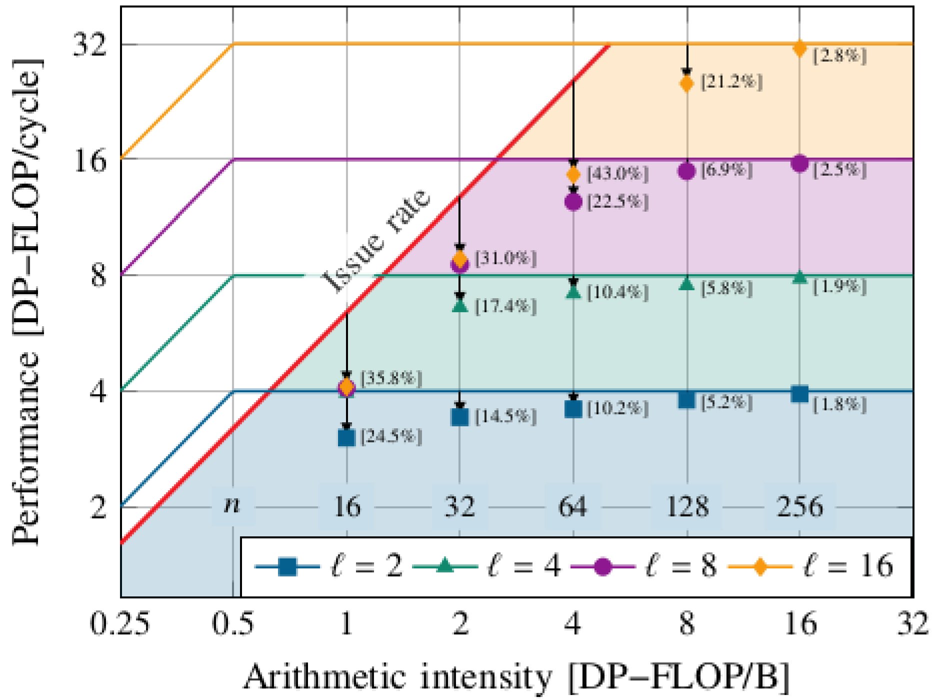


Fig. 5. Performance results for the matrix multiplication $C \leftarrow AB + C$, with different number of lanes ℓ, for several $n \times n$ problem sizes. The bold red line depicts a performance boundary due to the instruction issue rate. The numbers between brackets indicate the performance loss, with respect to the theoretically achievable peak performance.

Courtesy: Ara paper

Three Benchmarks

- **Matrix Multiplication**
  - C <-- AB + C where all matrices are n x n having DP elements
  - The algorithm requires 2n^3 floating-point operations (one FMA is considered as 2 operations) and 32n^2 Bytes of memory transfers.
  - Therefore, the algorithm has **arithmetic intensity 2n^3 / 32n^2 = n/16 DP-FLOPS/B**
- **Convolution(DCONV)**
  - Example frm GoogLeNet CNN, Input Image = 112 x 112 x 3; kernel = 7 x 7 x 3 x 64
  - This results in a total of 7 x 7 x 3 x 64 x 112 x 112 FMAs, or 236 DP-MFLOPS
  - Memory : (3 x 118 x 118 + 7 x 7 x 3 x 64 + 64 x 112 x 112) x 8 Bytes = 6.44 MiB
  - Therefore, **arithmetic intensity = 34.9 DP-FLOPS/B =>Heavily Compute Bound**
- **DAXPY**
  - Y <-- aX+Y where X, Y are n-dim vectors and a is a DP scalar
  - Requires 2^n operations (n FMAs) and 24n bytes of memory transfers
  - Therefore, this algorithm has **arithmetic intensity = 1/12 DP-FLOPS/B =>Memory Bound**

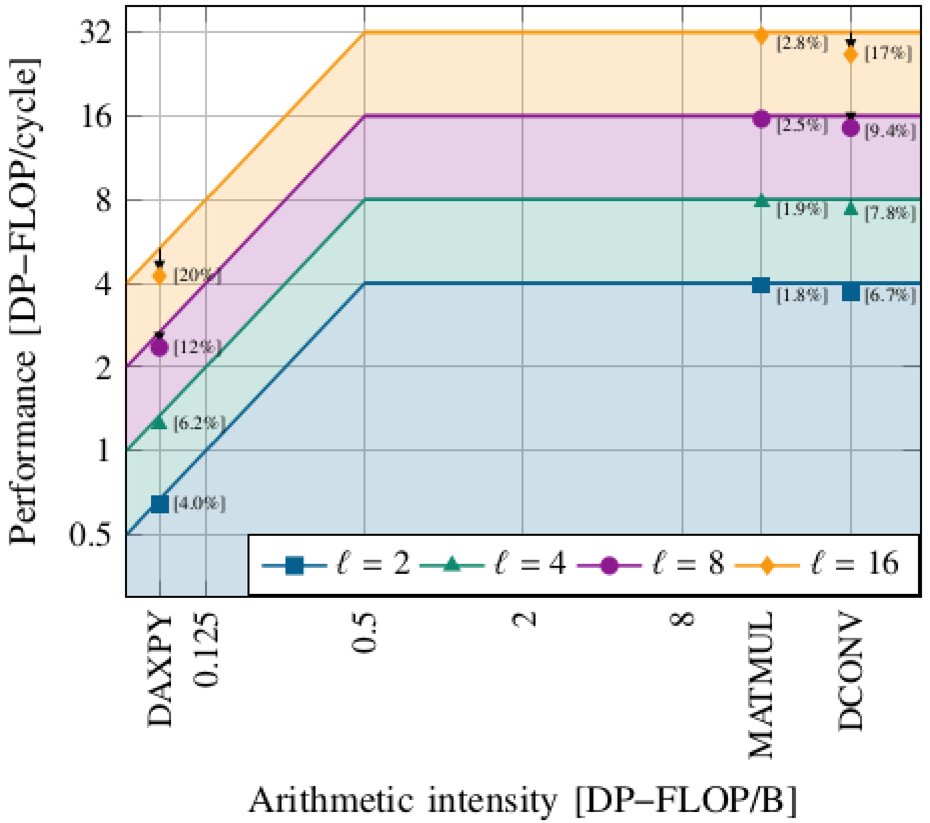


Fig. 6. Performance results for the three considered benchmarks, with different number of lanes ℓ. AXPY uses vectors of length 256, the MATMUL is between matrices of size 256×256, and CONV uses GoogLeNet's sizes. The numbers between brackets indicate the performance loss, with respect to the theoretically achievable peak performance.

Week 11: Oct 9th 2020

Actions

1. Study RISC-V Vector Extension

Work Done

1. Study RISC-V Vector Extension Spec (7 out 20 Chapters)

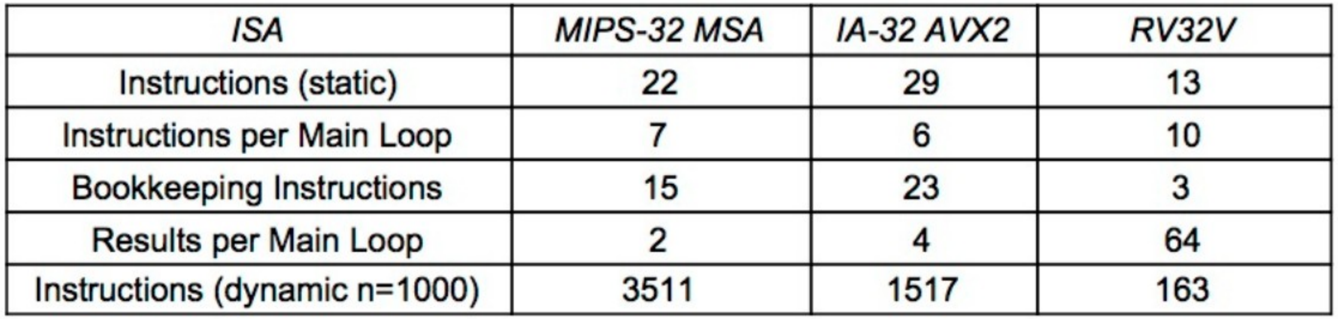
Goals for Standard RISC-V “V” Extension

- Efficient and Scalable to all reasonalble design-points
  - Low Cost Micro-controller or high-performance supercomputer
  - In-order, decoupled, or out-of-order microarchitectures
  - Integer, fixed-point, or floating-point data types
- Compiler Support for Auto-vectorization
- All instructions to fit into Standard 32-bit encoding space
- Be base for future vector++ extensions such as Crypto, Machine Learning extensions on top of RISC-V “V” extension.

***Source**: RISCV “V” Extension Proposal by Krste Asanovic, UC-Berkeley and SIFive, 2016*

V Key features

- Cray-style vectors
  - “The right way” to exploit SIMD parallelism
  - Packed-SIMD, GPU etc., are inefficient in terms of number of instructions in the instruction set, number of instructions executed and lack of portability. (Source: Computer Organization RISC-V Edition , Chapter 8)



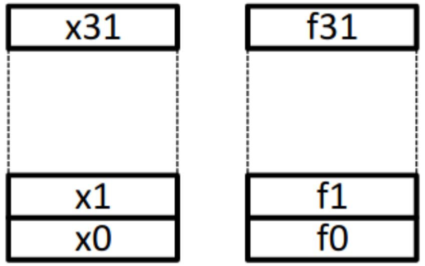
ISA	*MIPS-32 MSA*	*IA-32 AVX2*	*RV32V*
Instructions (static)	22	29	13
Instructions per Main Loop	7	6	10
Bookkeeping Instructions	15	23	3
Results per Main Loop	2	4	64
Instructions (dynamic n=1000)	3511	1517	163

Source: Comp Org. Ch 8

- Implementation-dependent vector length
  - Same binary runs on different CPUs with different hardware vector lengths
- Reconfigurable Vector Register File (allow different vector lengths and elemenet sizes)
- Mixed Precision Support (can operate with operands with different widths in the same op)

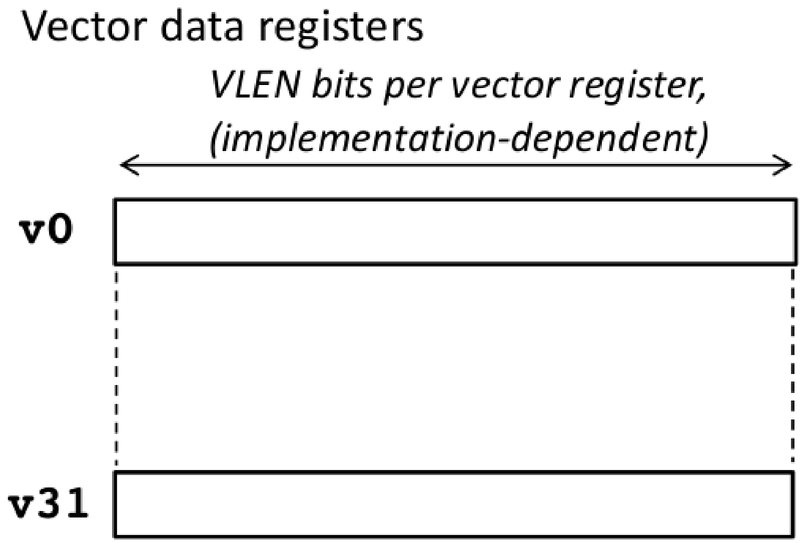
V Extension State

- Standard RISC-V scalar x and f registers



- Vector Configuration
  - Vector Length Register **vl**
  - Vector type register **vtype**
  - Other Control Registers
    - **vstart** (For trap handling)
    - **vrm/vxsat** - Fixed point rounding mode/ sateuration

- Upto 32 vector registers v0-v31 of atleast 4 elements each, with variable bits/element(size in powers of 2)

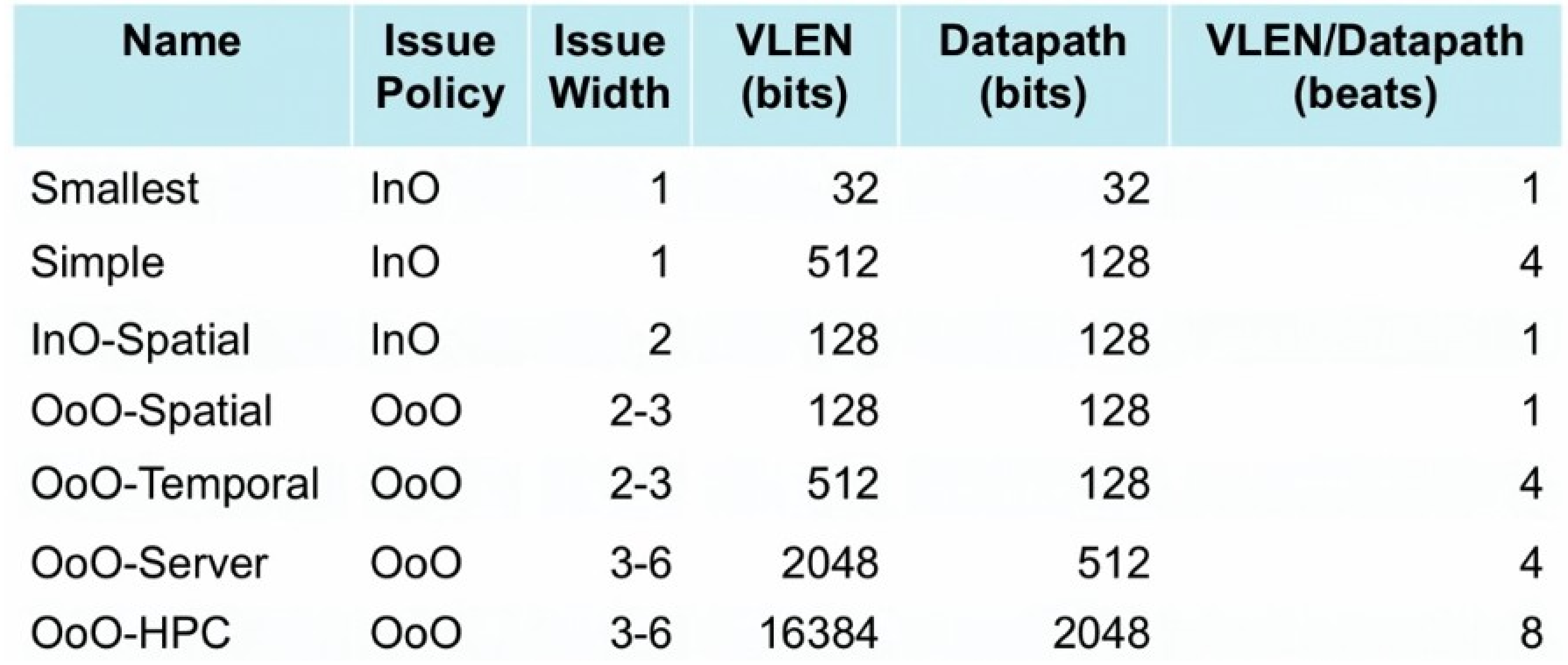


- One Predicate Register - **v0** for containing vector masks

Vector Unit Implementation Dependent Parameters

- **ELEN** : Size of largest element in bits
- **VLEN** : No. Of bits in each vector register
  - VLEN >= ELEN
- **SLEN** : Striping Distance in bits
  - Sets how many bits are packed contiguously into one vector register before moving to the next vector register in the *group*.
  - The striping length SLEN for an implementation is set to optimize the tradeoff between datapath wiring for mixed-width operations and buffering needed to corner-turn wide vector unit-stride memory accesses into parallel accesses for the vector register file.
  - This is generally the width of the memory port.
  - VLEN >= SLEN >= ELEN

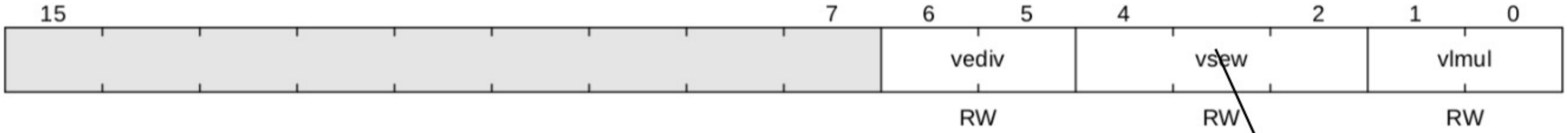
Some Microarchitecture Design Points



Name	Issue Policy	Issue Width	VLEN (bits)	Datapath (bits)	VLEN/Datapath (beats)
Smallest	InO	1	32	32	1
Simple	InO	1	512	128	4
InO-Spatial	InO	2	128	128	1
OoO-Spatial	OoO	2-3	128	128	1
OoO-Temporal	OoO	2-3	512	128	4
OoO-Server	OoO	3-6	2048	512	4
OoO-HPC	OoO	3-6	16384	2048	8

Source: Vector Extension 0.7 Slides , Krste Asanovic, UC Berkeley, RISCV Summit - 2019

Vector Type Register



Bits	Contents
1:0	vlmul[1:0]
4:2	vsew[2:0]
6:5	vediv[1:0]
XLEN-1:7	Reserved (write 0)

vsew[2:0]			SEW
0	0	0	8
0	0	1	16
0	1	0	32
0	1	1	64
1	0	0	128
1	0	1	256
1	1	0	512
1	1	1	1024

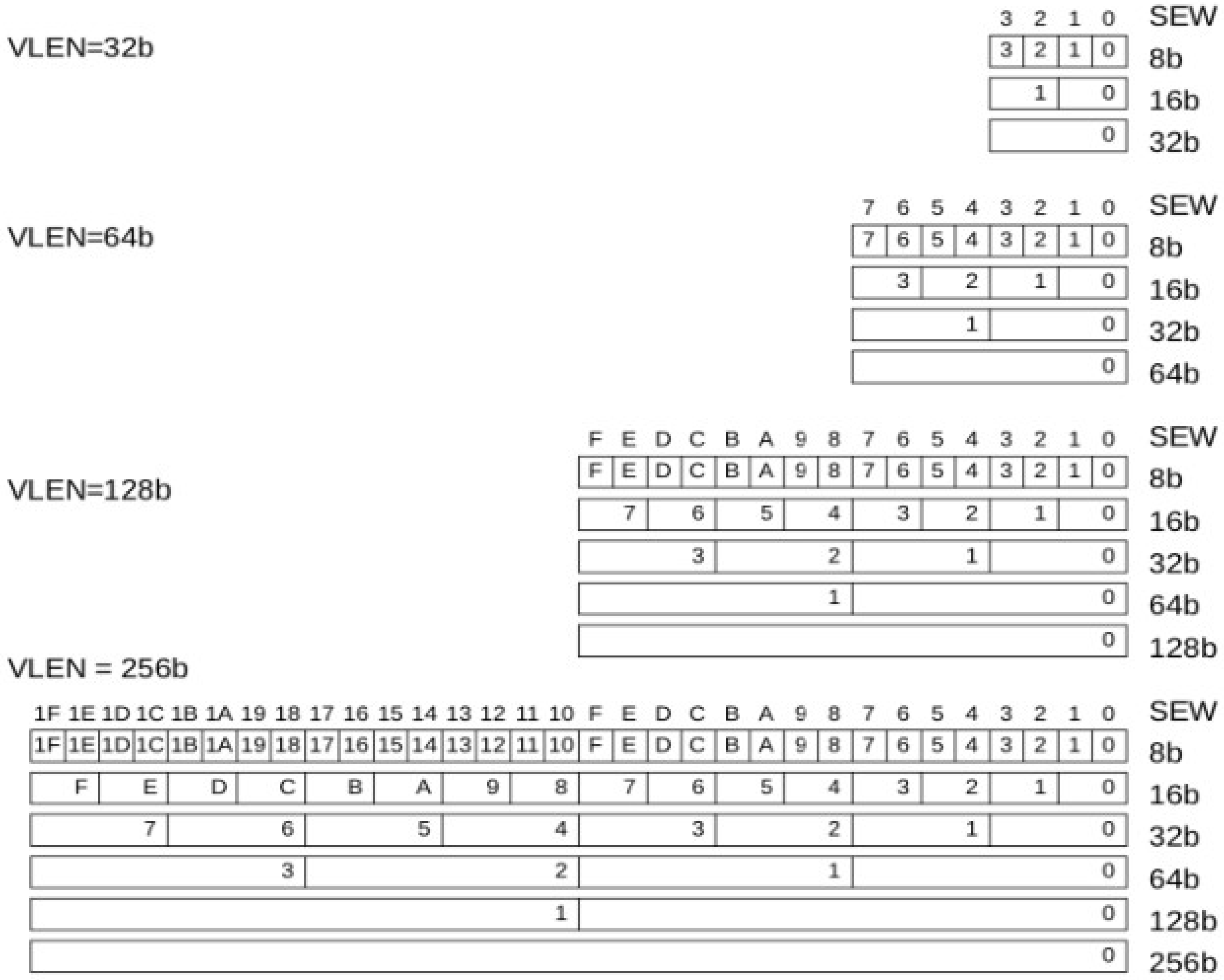
`vsew[2:0]` field encodes standard element width (SEW) in bits of elements in vector register (SEW = $8*2^{\texttt{vsew}}$)

`vlmul[1:0]` encodes vector register length multiplier (LMUL = $2^{\texttt{vlmul}}$ = 1-8) *(v0.9 adds "fractional LMUL" < 1)*

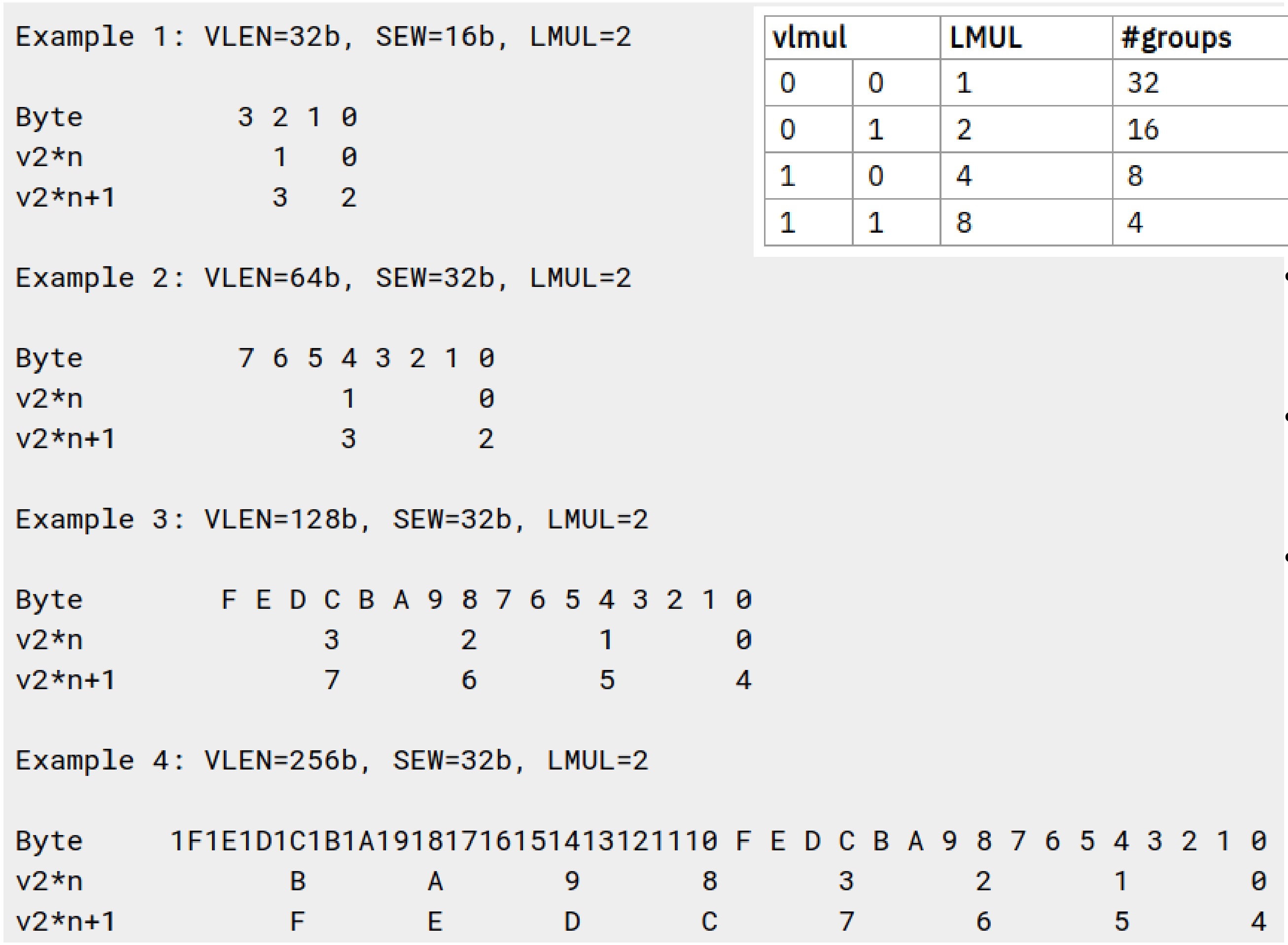
`vediv[1:0]` encodes how vector elements are divided into equal sub-elements (EDIV = $2^{\texttt{vediv}}$ = 1-8)

Slide adapted from :https://inst.eecs.berkeley.edu/~cs152/sp20/lectures/L17-RISCV-Vectors.pdf -2020

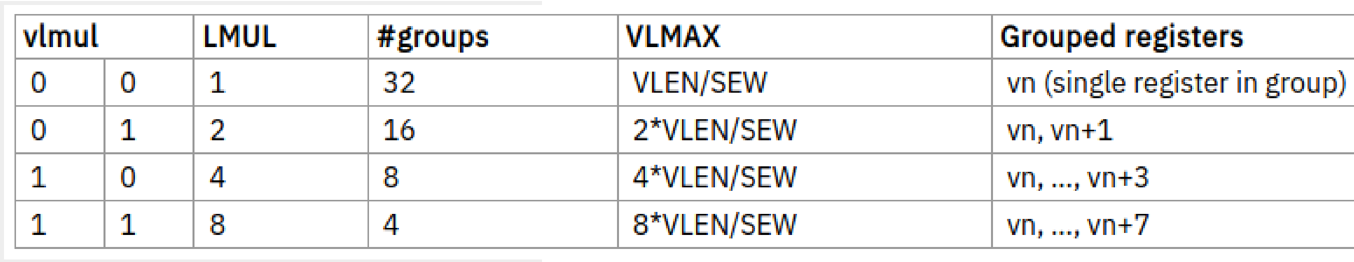
Example Vector Register Layouts



Vector Register Grouping (lmul)



```
Example 1: VLEN=32b, SEW=16b, LMUL=2

Byte         3 2 1 0
v2*n           1   0
v2*n+1         3   2

Example 2: VLEN=64b, SEW=32b, LMUL=2

Byte         7 6 5 4 3 2 1 0
v2*n               1       0
v2*n+1             3       2

Example 3: VLEN=128b, SEW=32b, LMUL=2

Byte        F E D C B A 9 8 7 6 5 4 3 2 1 0
v2*n              3       2       1       0
v2*n+1            7       6       5       4

Example 4: VLEN=256b, SEW=32b, LMUL=2

Byte     1F1E1D1C1B1A19181716151413121110 F E D C B A 9 8 7 6 5 4 3 2 1 0
v2*n            B       A       9       8       3       2       1       0
v2*n+1          F       E       D       C       7       6       5       4
```



vlmul		LMUL	#groups	VLMAX	Grouped registers
0	0	1	32	VLEN/SEW	vn (single register in group)
0	1	2	16	2*VLEN/SEW	vn, vn+1
1	0	4	8	4*VLEN/SEW	vn, ..., vn+3
1	1	8	4	8*VLEN/SEW	vn, ..., vn+7

- Multiple registers can be grouped together so that a single instruction can operate on multiple vector registers
- Term “*Vector Register Group*” referes to one or more registers that form a single operand
- Used for :
  - Accomodate mixed-width precision ops
  - To increase efficiency by using longer vectors when fewer separate registers needed

LMUL with shorter striping length (SLEN < VLEN)

```
Example 3: VLEN=128b, SLEN=64b, SEW=32b, LMUL=4

Byte          F E D C B A 9 8 7 6 5 4 3 2 1 0
v4*n                9       8       1       0   32b elements
v4*n+1              B       A       3       2
v4*n+2              D       C       5       4
v4*n+3              F       E       7       6

Example 4: VLEN=128b, SLEN=128b, SEW=32b, LMUL=4

Byte          F E D C B A 9 8 7 6 5 4 3 2 1 0
v4*n                3       2       1       0   32b elements
v4*n+1              7       6       5       4
v4*n+2              B       A       9       8
v4*n+3              F       E       D       C

Example 5: VLEN=256b, SLEN=128b, SEW=32b, LMUL=4

Byte     1F1E1D1C1B1A19181716151413121110 F E D C B A 9 8 7 6 5 4 3 2 1 0
v4*n           13      12      11      10       3       2       1       0
v4*n+1         17      16      15      14       7       6       5       4
v4*n+2         1B      1A      19      18       B       A       9       8
v4*n+3         1F      1E      1D      1C       F       E       D       C

Example 6: VLEN=256b, SLEN=128b, SEW=256b, LMUL=4

Byte     1F1E1D1C1B1A19181716151413121110 F E D C B A 9 8 7 6 5 4 3 2 1 0
v4*n                                                                    0
v4*n+1                                                                  1
v4*n+2                                                                  2
v4*n+3                                                                  3
```

Setting vector configuration, vsetvli/vsetvl

The **`vsetvl{i}`** configuration instructions set the **`vtype`** register, and also set the **`vl`** register, returning the **`vl`** value in a scalar register

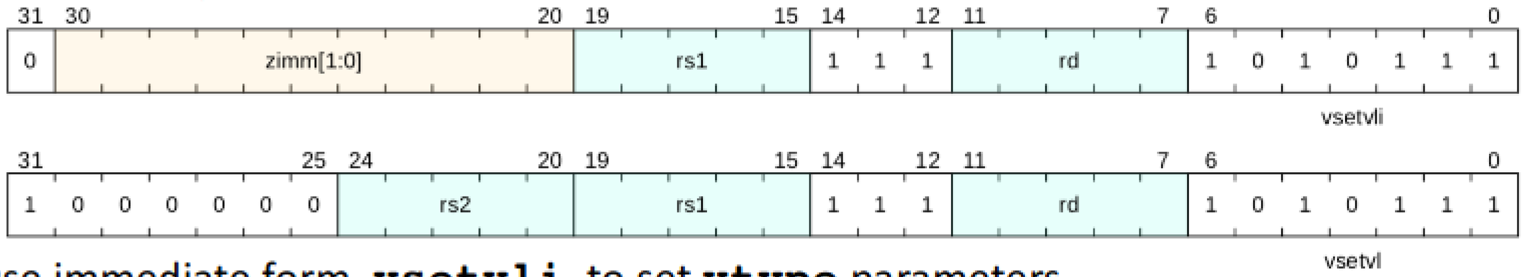
```
vsetvli rd, rs1, e8 # Set SEW=8, vl=min(VLEN/SEW,rs1), rd=vl
```

`vtype` parameters (SEW,LMUL,EDIV) encoded as immediate in instruction

Resulting machine vector length setting

Requested application vector length

Instruction encoding



31 30 20 19 15 14 12 11 7 6 0
0 | zimm[1:0] | rs1 | 1 1 1 | rd | 1 0 1 0 1 1 1
vsetvli
31 25 24 20 19 15 14 12 11 7 6 0
1 0 0 0 0 0 0 | rs2 | rs1 | 1 1 1 | rd | 1 0 1 0 1 1 1
vsetvl

Usually use immediate form, **`vsetvli`**, to set **`vtype`** parameters.

The register version **`vsetvl`** is usually used only for context save/restore

vsetvl{i} Instruction

- The first scalar register argument, *rs1*, is the requested application vector length (AVL)
  - The type argument (either immediate or second register *rs2*) indicates how the vector registers should be configured
- Configuration includes size of each element, SEW, and LMUL value
- The vector length is set to the *minimum* of requested AVL and the maximum supported vector length (VLMAX) in the new configuration
  - VLMAX = LMUL*VLEN/SEW
  - **vl** = min(AVL, VLMAX)
- The value placed in **vl** is also written to the scalar destination register *rd*

Stripmined vector Memory Copy Example

```
  # void *memcpy(void* dest, const void* src, size_t n)
  # a0=dest, a1=src, a2=n
  #
memcpy:
    mv a3, a0 # Copy destination
loop:
  vsetvli t0, a2, e8     # Vectors of 8b
  vle.v v0, (a1)                # Load bytes
    add a1, a1, t0              # Bump pointer
    sub a2, a2, t0              # Decrement count
  vse.v v0, (a3)                # Store bytes
    add a3, a3, t0              # Bump pointer
    bnez a2, loop               # Any more?
    ret                         # Return
```

Set configuration, calculate vector strip length → vsetvli

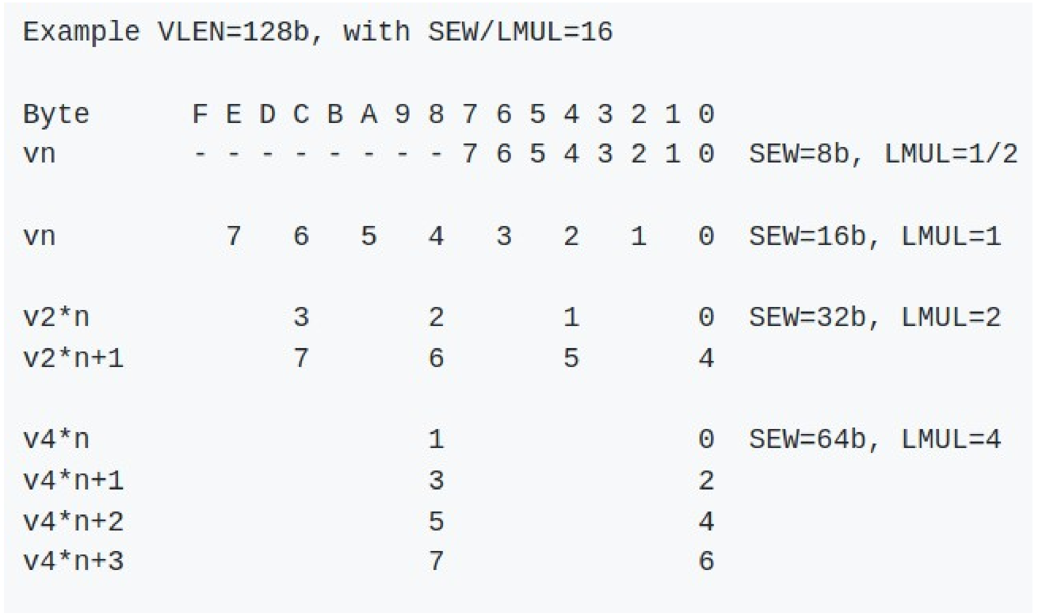
Unit-stride vector load elements (bytes) → vle.v

Unit-stride vector store elements (bytes) → vse.v

Same binary machine code can run on machines with any VLEN!

Mixed Precesion Operations Support

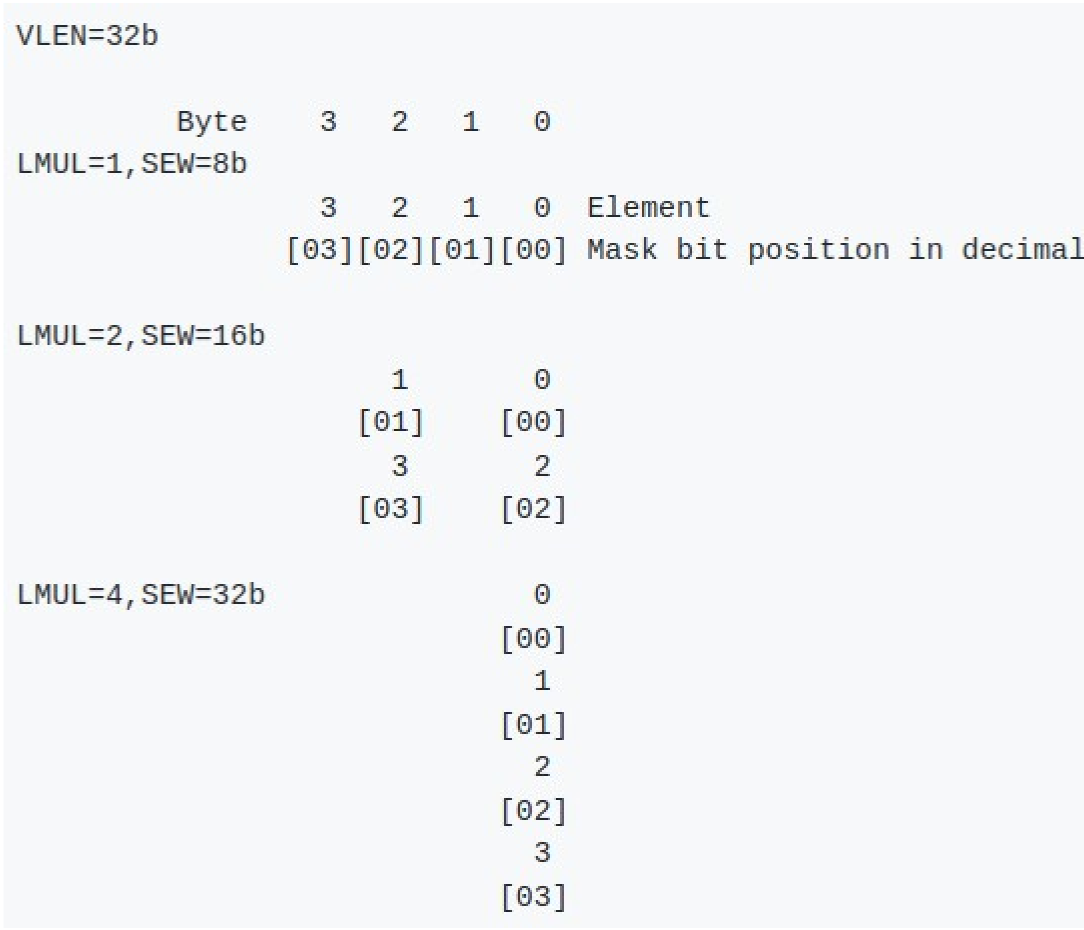
- The vector ISA is designed to support mixed-width operations without requiring explicit additional rearrangement instructions.
- The recommended software strategy is to modify vtype dynamically to keep SEW/LMUL constant (and hence VLMAX constant) when operating on vectors of different precision values.
- The following example shows four different packed element widths (8b, 16b, 32b, 64b) in a VLEN=128b implementation. The vector register grouping factor (LMUL) is increased by the relative element size such that each group can hold the same number of vector elements (VLMAX=8 in this example) to simplify stripmining code.



```
Example VLEN=128b, with SEW/LMUL=16

Byte      F E D C B A 9 8 7 6 5 4 3 2 1 0
vn        - - - - - - - - 7 6 5 4 3 2 1 0  SEW=8b, LMUL=1/2

vn          7   6   5   4   3   2   1   0  SEW=16b, LMUL=1

v2*n            3       2       1       0  SEW=32b, LMUL=2
v2*n+1          7       6       5       4

v4*n                    1               0  SEW=64b, LMUL=4
v4*n+1                  3               2
v4*n+2                  5               4
v4*n+3                  7               6
```

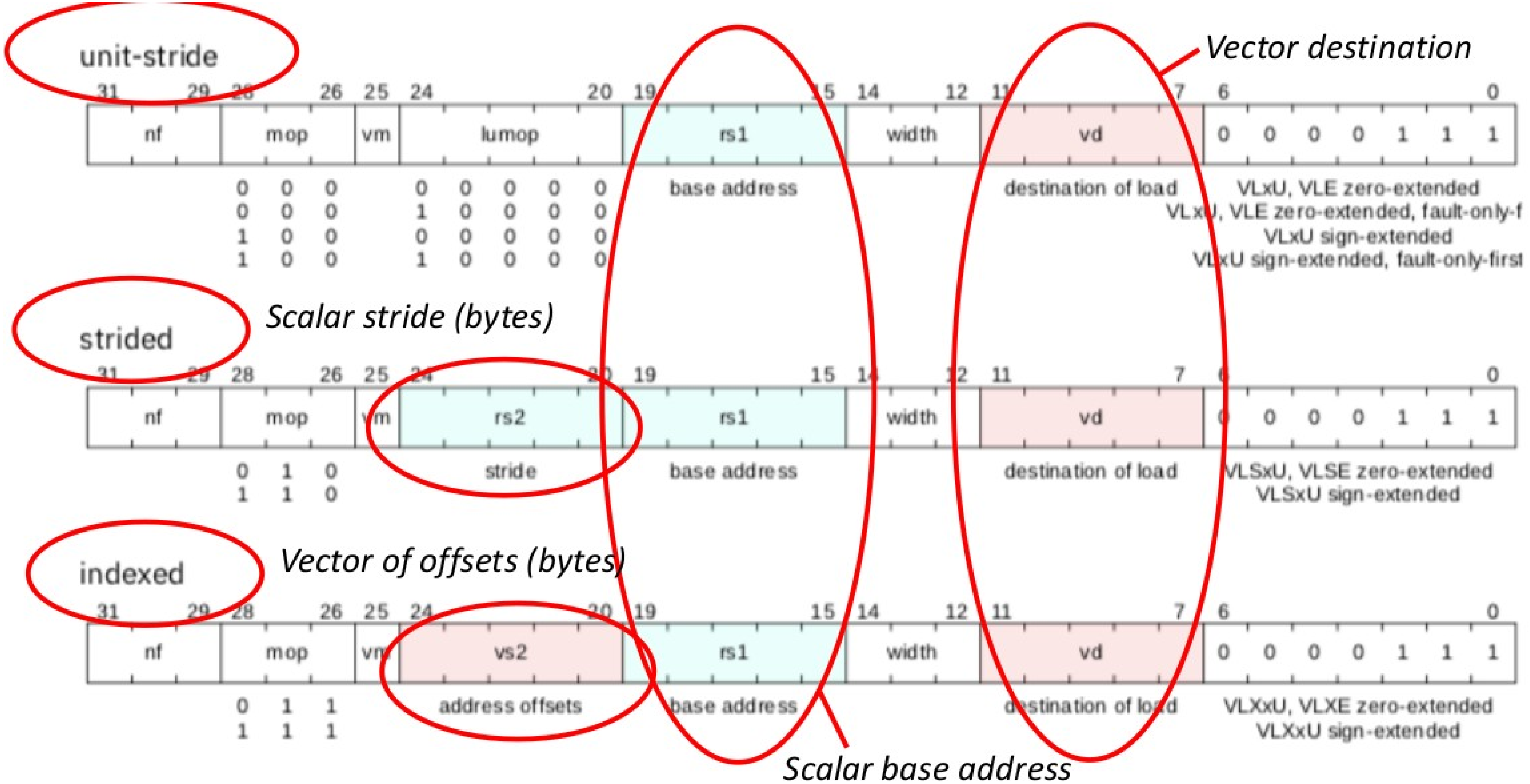
Mask Register

- Nearly all operations can be optionally under a mask (or predicate) held in vector register v0
- A single vm bit in instruction encoding selects whether unmasked or under control of v0
- A vector mask occupies only one vector register regardless of SEW and LMUL.
- Each element is allocated a single mask bit in a mask vector register.
- The mask bit for element i is located in bit i of the mask register, independent of SEW or LMUL.



```
VLEN=32b

         Byte    3   2   1   0
LMUL=1,SEW=8b
                 3   2   1   0  Element
               [03][02][01][00] Mask bit position in decimal

LMUL=2,SEW=16b
                     1       0
                   [01]    [00]
                     3       2
                   [03]    [02]

LMUL=4,SEW=32b               0
                           [00]
                             1
                           [01]
                             2
                           [02]
                             3
                           [03]
```

Vector Load Instructions



unit-stride

31 29	28 26	25	24 20	19 15	14 12	11 7	6 0
nf	mop	vm	lumop	rs1	width	vd	0 0 0 0 1 1 1
	0 0 0		0 0 0 0 0	base address		destination of load	VLxU, VLE zero-extended
	0 0 0		1 0 0 0 0				VLxU, VLE zero-extended, fault-only-f
	1 0 0		0 0 0 0 0				VLxU sign-extended
	1 0 0		1 0 0 0 0				VLxU sign-extended, fault-only-first

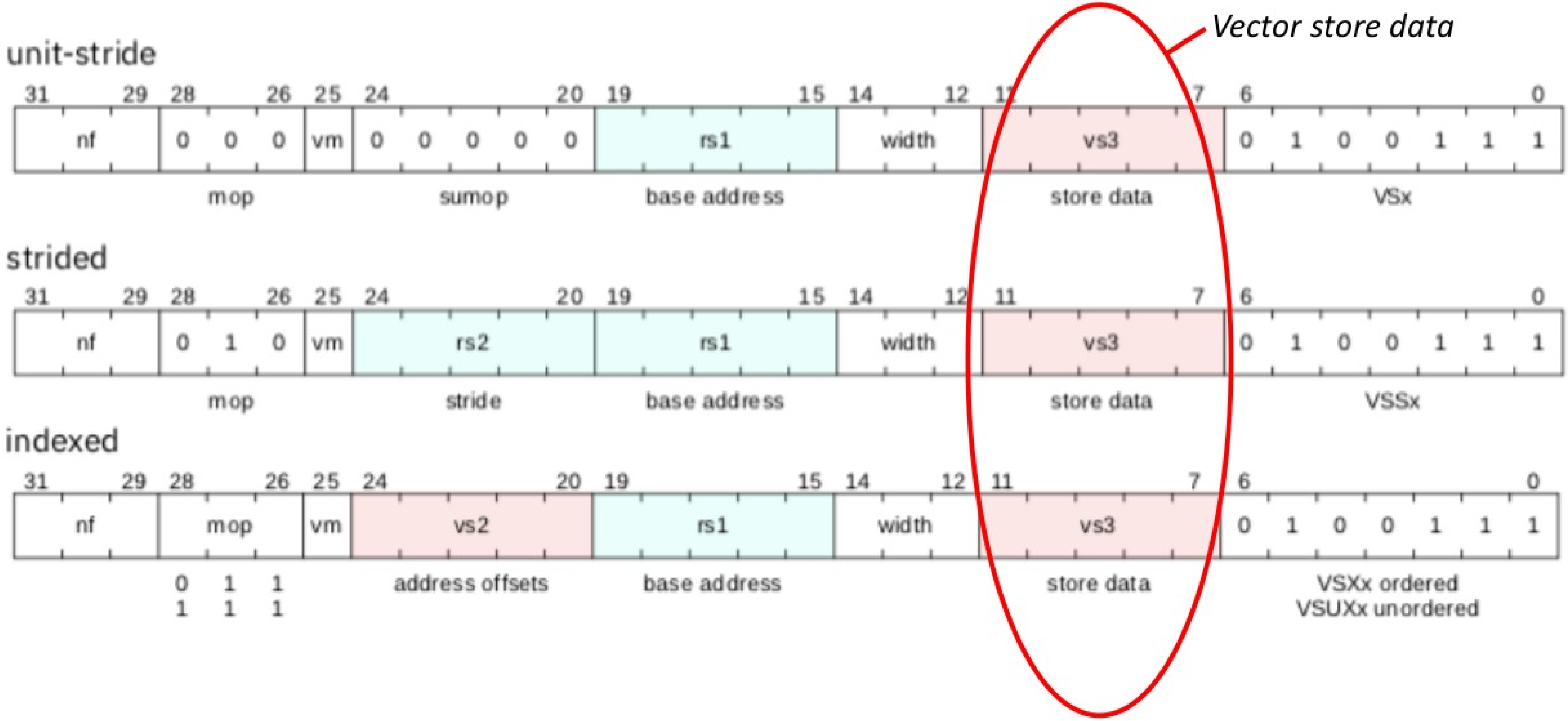
strided

31 29	28 26	25	24 20	19 15	14 12	11 7	6 0
nf	mop	vm	rs2	rs1	width	vd	0 0 0 0 1 1 1
	0 1 0		stride	base address		destination of load	VLSxU, VLSE zero-extended
	1 1 0						VLSxU sign-extended

indexed

31 29	28 26	25	24 20	19 15	14 12	11 7	6 0
nf	mop	vm	vs2	rs1	width	vd	0 0 0 0 1 1 1
	0 1 1		address offsets	base address		destination of load	VLXxU, VLXE zero-extended
	1 1 1						VLXxU sign-extended

Vector store Instructions



Vector Unit Stride Load Stores

```
# vd destination, rs1 base address, vm is mask encoding (v0.t or <missing>)
    vle.v           vd, (rs1), vm      # SEW
# vs3 store data, rs1 base address, vm is mask encoding (v0.t or <missing>)
    vse.v           vs3, (rs1), vm     # SEW store
```

Vector Strided Load Stores

```
# vd destination, rs1 base address, rs2 byte stride
    vlse.v          vd, (rs1), rs2, vm  # SEW
# vs3 store data, rs1 base address, rs2 byte stride
    vsse.v          vs3, (rs1), rs2, vm # SEW
```

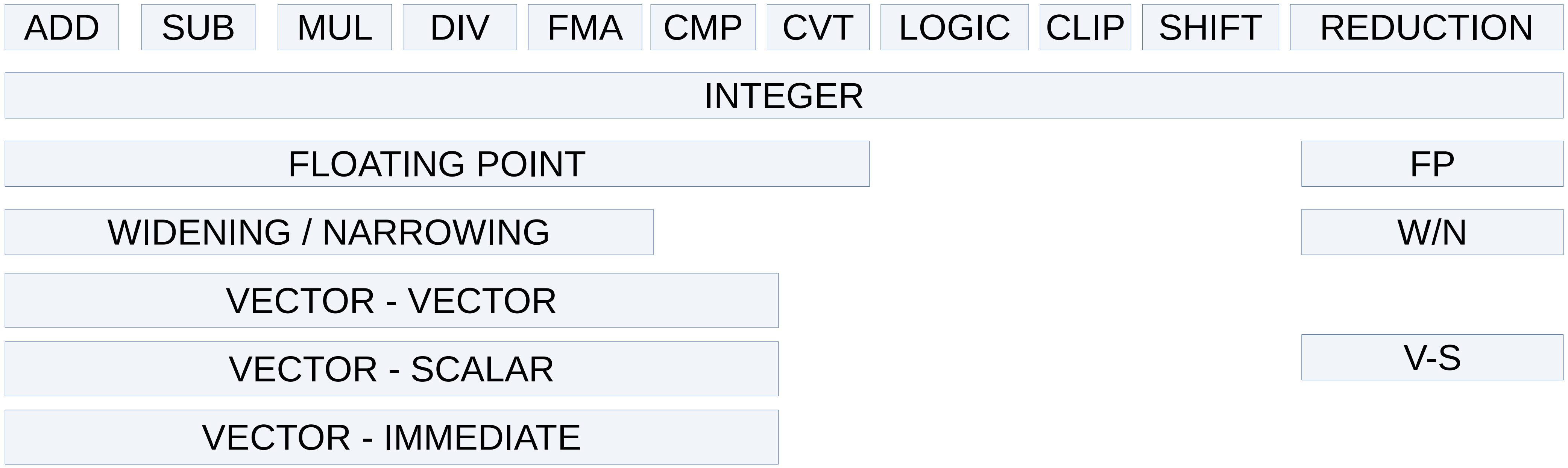
Vector Indexed Load Stores

```
# vd destination, rs1 base address, vs2 indices
    vlxe.v          vd, (rs1), vs2, vm  # SEW
# vs3 store data, rs1 base address, vs2 indices
    vsxe.v          vs3, (rs1), vs2, vm # SEW
```

Week-12 16 Oct 2020

1. RISC-V ISA Vector Extension Instruction Set {Chapters 7 – 17}
2. Further Course of Action

ARITHMETIC AND LOGICAL VECTOR INSTRUCTIONS



1. There is NO SPERERATE *FP Vector Regsiter file*. Both Integer and FP values are stored in the same Vector Register files.
2. Nearly all operations can be optionally under a mask (or predicate) held in vector register v0.
3. A single vm bit in instruction encoding selects whether unmasked or under control of v0

Vector Arithmetic Instruction Encodings

31 26	25	24 20	19 15	14 12	11 7	6 0
funct6	vm	vs2	vs1	funct3	vd	1 0 1 0 1 1 1
				0 0 0		OPIVV
				0 0 1		OPFVV

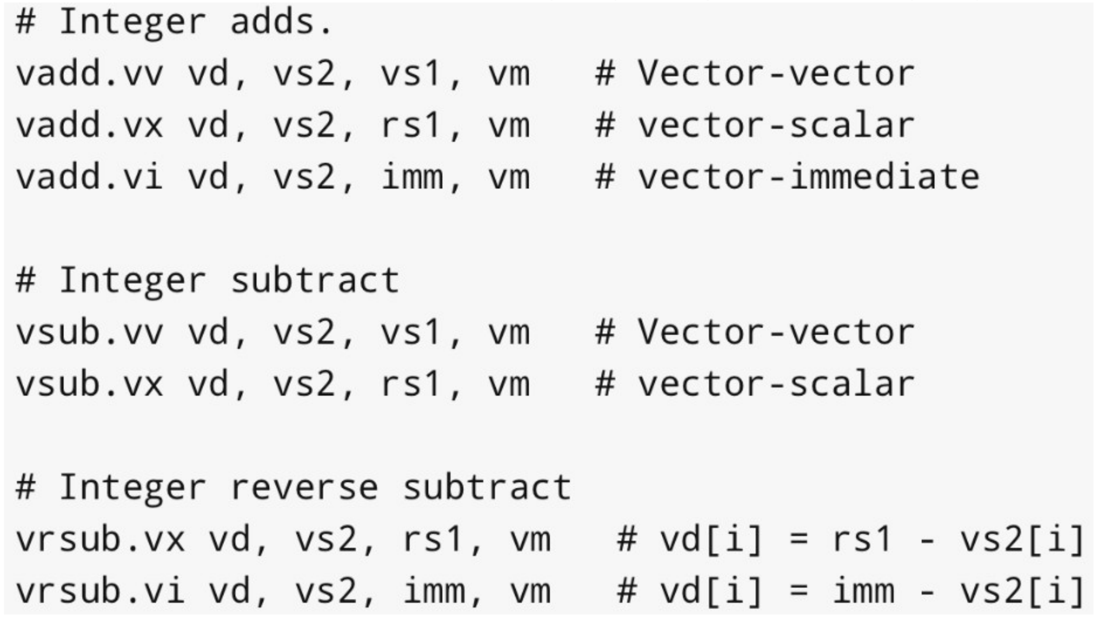
31 26	25	24 20	19 15	14 12	11 7	6 0
funct6	vm	vs2	vs1	0 1 0	vd / rd	1 0 1 0 1 1 1
						OPMVV

31 26	25	24 20	19 15	14 12	11 7	6 0
funct6	vm	vs2	simm5	0 1 1	vd	1 0 1 0 1 1 1
						OPIVI

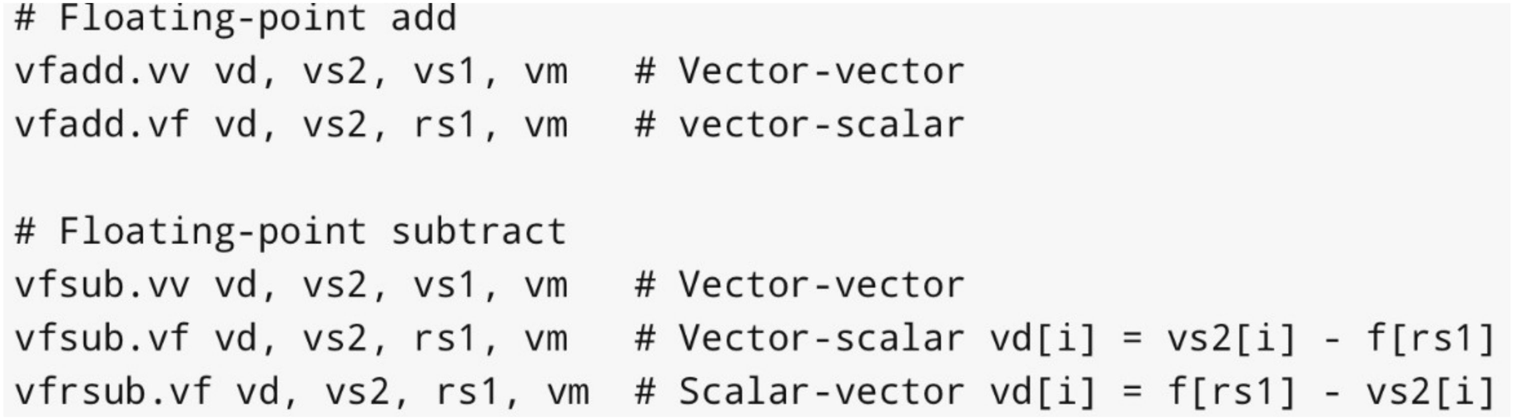
31 26	25	24 20	19 15	14 12	11 7	6 0
funct6	vm	vs2	rs1	funct3	vd	1 0 1 0 1 1 1
				1 0 0		OPIVX
				1 0 1		OPFVF

31 26	25	24 20	19 15	14 12	11 7	6 0
funct6	vm	vs2	rs1	1 1 0	vd / rd	1 0 1 0 1 1 1
						OPMVX

Vector Add Instructions

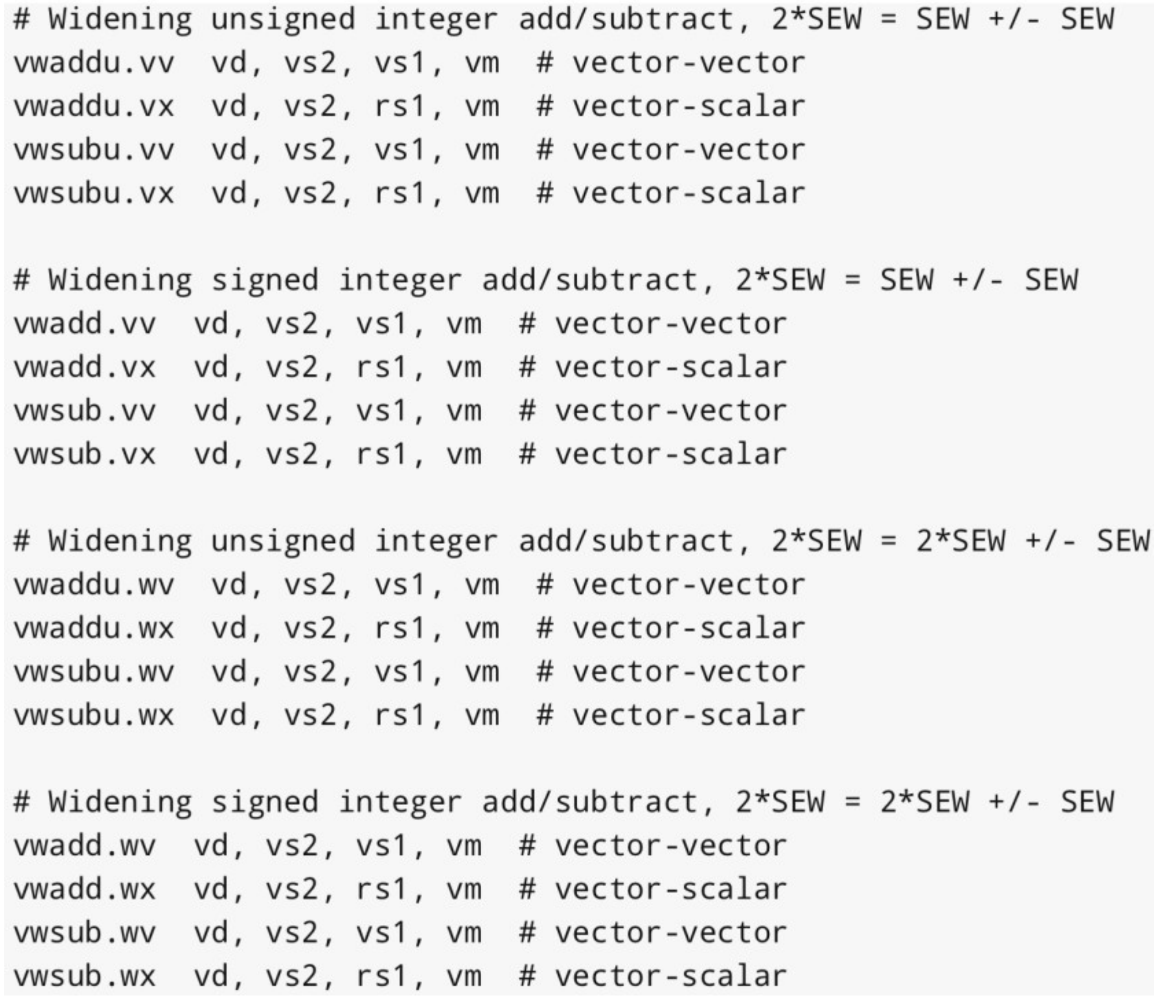


```
# Integer adds.
vadd.vv vd, vs2, vs1, vm   # Vector-vector
vadd.vx vd, vs2, rs1, vm   # vector-scalar
vadd.vi vd, vs2, imm, vm   # vector-immediate

# Integer subtract
vsub.vv vd, vs2, vs1, vm   # Vector-vector
vsub.vx vd, vs2, rs1, vm   # vector-scalar

# Integer reverse subtract
vrsub.vx vd, vs2, rs1, vm   # vd[i] = rs1 - vs2[i]
vrsub.vi vd, vs2, imm, vm   # vd[i] = imm - vs2[i]
```



```
# Floating-point add
vfadd.vv vd, vs2, vs1, vm   # Vector-vector
vfadd.vf vd, vs2, rs1, vm   # vector-scalar

# Floating-point subtract
vfsub.vv vd, vs2, vs1, vm   # Vector-vector
vfsub.vf vd, vs2, rs1, vm   # Vector-scalar vd[i] = vs2[i] - f[rs1]
vfrsub.vf vd, vs2, rs1, vm  # Scalar-vector vd[i] = f[rs1] - vs2[i]
```

Widening Integer Add Instructions



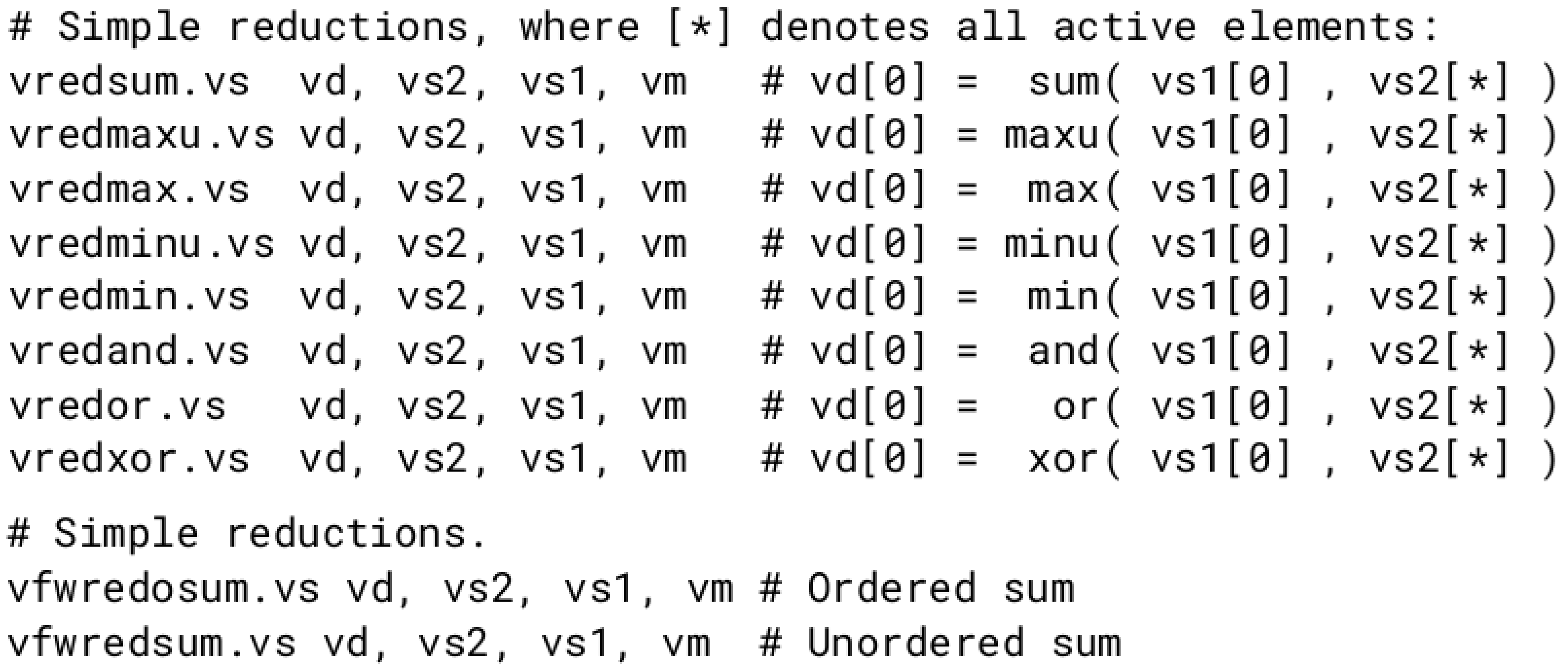
```
# Widening unsigned integer add/subtract, 2*SEW = SEW +/- SEW
vwaddu.vv  vd, vs2, vs1, vm  # vector-vector
vwaddu.vx  vd, vs2, rs1, vm  # vector-scalar
vwsubu.vv  vd, vs2, vs1, vm  # vector-vector
vwsubu.vx  vd, vs2, rs1, vm  # vector-scalar

# Widening signed integer add/subtract, 2*SEW = SEW +/- SEW
vwadd.vv  vd, vs2, vs1, vm  # vector-vector
vwadd.vx  vd, vs2, rs1, vm  # vector-scalar
vwsub.vv  vd, vs2, vs1, vm  # vector-vector
vwsub.vx  vd, vs2, rs1, vm  # vector-scalar

# Widening unsigned integer add/subtract, 2*SEW = 2*SEW +/- SEW
vwaddu.wv  vd, vs2, vs1, vm  # vector-vector
vwaddu.wx  vd, vs2, rs1, vm  # vector-scalar
vwsubu.wv  vd, vs2, vs1, vm  # vector-vector
vwsubu.wx  vd, vs2, rs1, vm  # vector-scalar

# Widening signed integer add/subtract, 2*SEW = 2*SEW +/- SEW
vwadd.wv  vd, vs2, vs1, vm  # vector-vector
vwadd.wx  vd, vs2, rs1, vm  # vector-scalar
vwsub.wv  vd, vs2, vs1, vm  # vector-vector
vwsub.wx  vd, vs2, rs1, vm  # vector-scalar
```

Widening FP Fused Mult-Acc

```
# FP widening multiply-accumulate, overwrites addend
vfwmacc.vv vd, vs1, vs2, vm    # vd[i] = +(vs1[i] * vs2[i]) + vd[i]
vfwmacc.vf vd, rs1, vs2, vm    # vd[i] = +(f[rs1] * vs2[i]) + vd[i]

# FP widening negate-(multiply-accumulate), overwrites addend
vfwnmacc.vv vd, vs1, vs2, vm   # vd[i] = -(vs1[i] * vs2[i]) - vd[i]
vfwnmacc.vf vd, rs1, vs2, vm   # vd[i] = -(f[rs1] * vs2[i]) - vd[i]

# FP widening multiply-subtract-accumulator, overwrites addend
vfwmsac.vv vd, vs1, vs2, vm    # vd[i] = +(vs1[i] * vs2[i]) - vd[i]
vfwmsac.vf vd, rs1, vs2, vm    # vd[i] = +(f[rs1] * vs2[i]) - vd[i]

# FP widening negate-(multiply-subtract-accumulator), overwrites addend
vfwnmsac.vv vd, vs1, vs2, vm   # vd[i] = -(vs1[i] * vs2[i]) + vd[i]
vfwnmsac.vf vd, rs1, vs2, vm   # vd[i] = -(f[rs1] * vs2[i]) + vd[i]
```

Vector Reduction

- Take a Vector Register group of elements and a scalar held in element 0 of a vector register(vs1[0]), and perform a reduction operation using some binary operator to produce a scalar result in element 0 of a vector register(vd[0]).
- Supports FP, Widening, Ordered and Un-ordered sums



```
# Simple reductions, where [*] denotes all active elements:
vredsum.vs  vd, vs2, vs1, vm   # vd[0] =  sum( vs1[0] , vs2[*] )
vredmaxu.vs vd, vs2, vs1, vm   # vd[0] = maxu( vs1[0] , vs2[*] )
vredmax.vs  vd, vs2, vs1, vm   # vd[0] =  max( vs1[0] , vs2[*] )
vredminu.vs vd, vs2, vs1, vm   # vd[0] = minu( vs1[0] , vs2[*] )
vredmin.vs  vd, vs2, vs1, vm   # vd[0] =  min( vs1[0] , vs2[*] )
vredand.vs  vd, vs2, vs1, vm   # vd[0] =  and( vs1[0] , vs2[*] )
vredor.vs   vd, vs2, vs1, vm   # vd[0] =   or( vs1[0] , vs2[*] )
vredxor.vs  vd, vs2, vs1, vm   # vd[0] =  xor( vs1[0] , vs2[*] )

# Simple reductions.
vfwredosum.vs vd, vs2, vs1, vm # Ordered sum
vfwredsum.vs vd, vs2, vs1, vm  # Unordered sum
```

Recent Developents in RISCV-V Ext

- Andes Group develops Vector based Core for RISCV -V Specifications (First Company to do so) - May 26, 2020
- GNU-tool chain for Vector Spec v0.8 available on GitHub
- RISC-V V likely to finallize in Ver1.0

Future Course of Action(s)

- Micro-architecture design
  - Very basic outline provided in Patterson Hennessy text(5th Edition)
  - Krste Asanovac’s PhD thesis (1998) discusses microarchitectural design in slightly more detail (reg-file, memory design (w/o cache),
  - Couple of papers such as Hwacha, Ara (Open Source, not gone thru yet)
- High level design for integrating Vector processor into RISCV core
  - Interface between main core and vector core
  - How closely coupled ? On which bus should the Vector Unit be present Main bus or peripheral(AXI) bus?
- Design of FPU (16,32)
  - I’m unfamiliar
  - IP-Core available to integrate OR else, design from scratch?
- ISA-Simulator coding required ? yes
- Resources on FPGA . Require Part number details

Ordered and Unordered Sums

- **Ordered Sum** Must sum the values in element order, starting with the scalar in vs1[0]--that is, it performs the computation: (((vs1[0] + vs2[0]) + vs2[1]) + ...) vs2[vl-1], where each addition operates identically to the scalar floating-point instructions in terms of raising exception flags and generating or propagating special values. Code good for compiler to auto-vectorize.
- **Unordered Sum** provides freedom to the implementation to perform reduction – Binary Tree of all vector operands followedby log2(n) steps of addition. This is faster

Vector Permutation Instructions

- Provided to move elements around within the vector registers.
- **Integer Scalar Move Instruction** transfers a single value between a scalar x register and element 0 of a vector register.
- **Vector Slide Instructions** move vector elements up or down in a vector register group
- **Vector Register Gather Instruction** reads elements from a first source vector register group at locations given by a second source vector register group.
- **Vector Compress Instruction** allows elements selected by a vector mask register from a source vector register group to be packed into contiguous elements at the start of the destination vector register group.
- **Whole Vector Register Move Instruction** copies all VLEN bits in rs -->rd