Java Enable Preview: Vector API

Daniel Hinojosa

Using SIMD Processing in the JVM

In this Presentation

- Vector API
- SIMD
- Species
- Lanes & Components
- Masking
- Lane-wise Operations
- Cross-lane Operations

Slides and Material:
https://github.com/dhinojosa/java_enable_vector_api

Vector API

Vector API

- Introduce an API to express vector computations that reliably compile at runtime **to vector instructions** on supported CPU architectures.
- The performance from the Vector API is superior to equivalent scalar computations.
- A part of **Project Panama**

https://openjdk.org/jeps/448

SIMD

- Single Instruction/Multiple Data
- Such machines exploit data level parallelism, but not concurrency
- Not all machines offer the ability to process SIMD processes
- A binary operation applied to two vectors with the same number of lanes would, for each lane, apply the equivalent scalar operation on the corresponding two scalar values from each vector
- This is not new, and have been used with supercomputers in the 60s and 70s

https://en.wikipedia.org/wiki/Single_instruction

Use Cases

- Fast Computation of Arrays
- Machine Learning and Ai
- Gaming Arithmetic and Physics Engines
- Imaging and Audio
- Cryptography

Parallel Processing

- **In Standard Java Today:**
  - Distributed Process - Break up the data into small chunks and perform your calculations
  - Carried through with threads
  - Might be performed concurrently and not really in parallel, depending if threads are completely scheduled and have to undergo timesharing
- **Using Vector API:**
  - Fast Parallel Computation, not Concurrent
  - SIMD ***does not use*** `Threads`
  - Uses large amount of operating units, executing the same operation *at the same CPU cycle* at the exact same time
  - Requires the C2 Compiler

Vector API is Waiting

- Vector API is in Project Panama
- What does Project Valhalla have that Vector API under Panama need?
  - Inline Classes - The current implementation has to box/unbox vector objects and store them in heap, which would defeat the purpose of having fast parallelism.
  - Primitive Classes - Allow us to declare classes that behave like primitives

Project Valhalla

1. **Value Classes and Objects**, introducing objects that lack identity and thus can have optimized encodings
2. **Null-Restricted and Nullable Types**, providing language support for null-aware types and runtime enforcement of null restrictions
3. **Null-Restricted Value Class Types**, improving the performance of fields and arrays with null-restricted value class types
4. **Enhanced Primitive Boxing**, allowing primitives to be treated more like objects
5. **Parametric JVM**, preserving and optimizing generic class and method parameterizations at runtime

Latest news in Vector API

- JEP 508: Vector API is on the 10th incubator (as of JDK 25)
- In the latest incubator:
  - `VectorShuffle` now supports access to and from `MemorySegment`.
  - The implementation now links to native mathematical-function libraries via the Foreign Function & Memory API (JEP 454) rather than custom C++ code inside the HotSpot JVM, thereby improving maintainability.
  - Addition, subtraction, division, multiplication, square root, and fused multiply/add operations on Float16 values are now auto-vectorized on supporting x64 CPUs.
- Important that we have upgraded to the latest Java to use these features

Species

Species

- Know the types that we need to use: `byte`, `short`, `int`, `long`, `float`, and `double`
- Know the size of what can be handled, anywhere from 64 bits to 512 bits, *for now*
- Depending on the type that used is called the *species*

https://openjdk.org/jeps/448

Vectors

Vectors

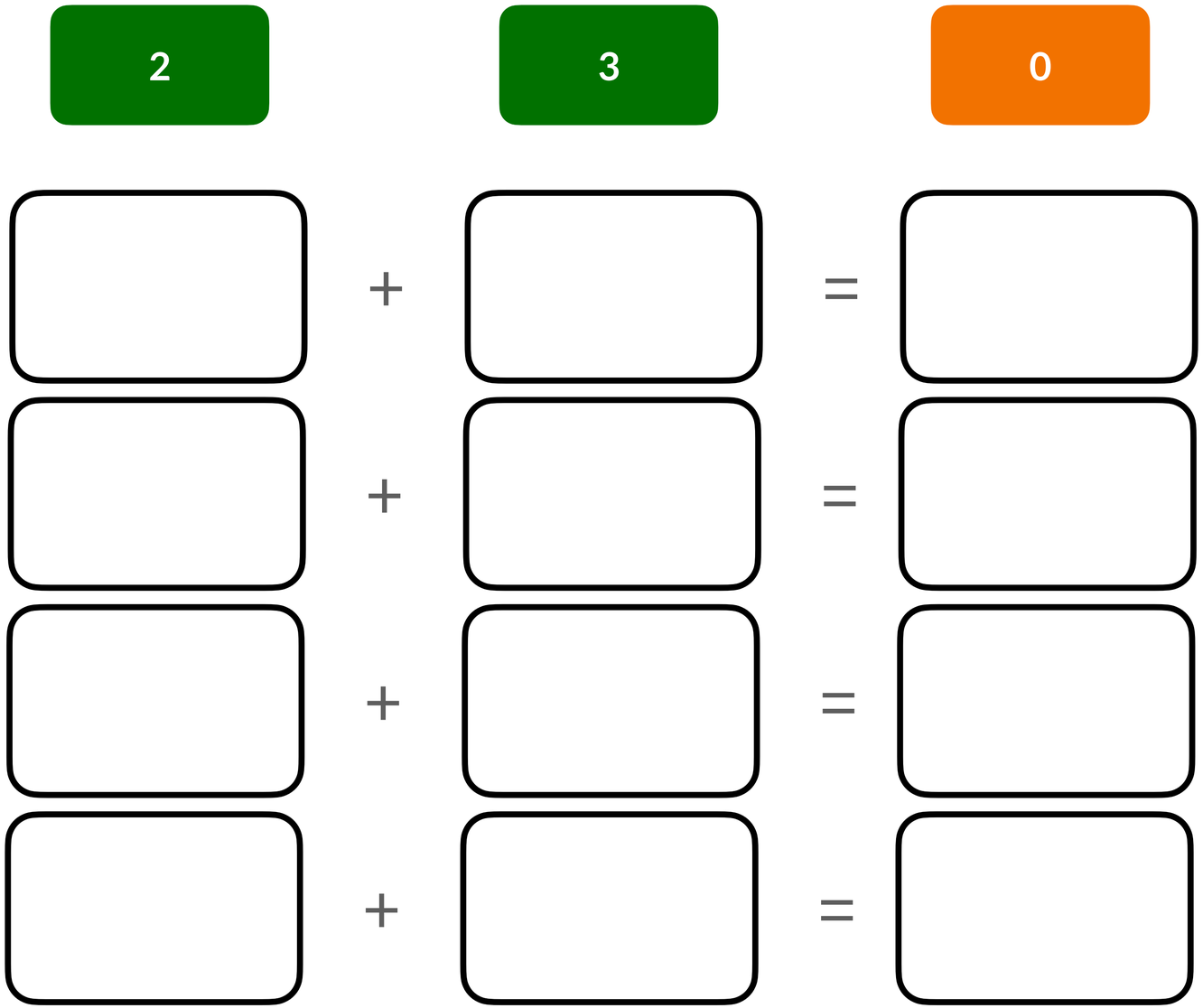
- A vector computation consists of a sequence of operations on vectors
- A vector comprises a (usually) fixed sequence of scalar values, where the scalar values correspond to the number of hardware-defined vector lanes
- A binary operation applied to two vectors with the same number of lanes would, for each lane, apply the equivalent scalar operation on the corresponding two scalar values from each vector.

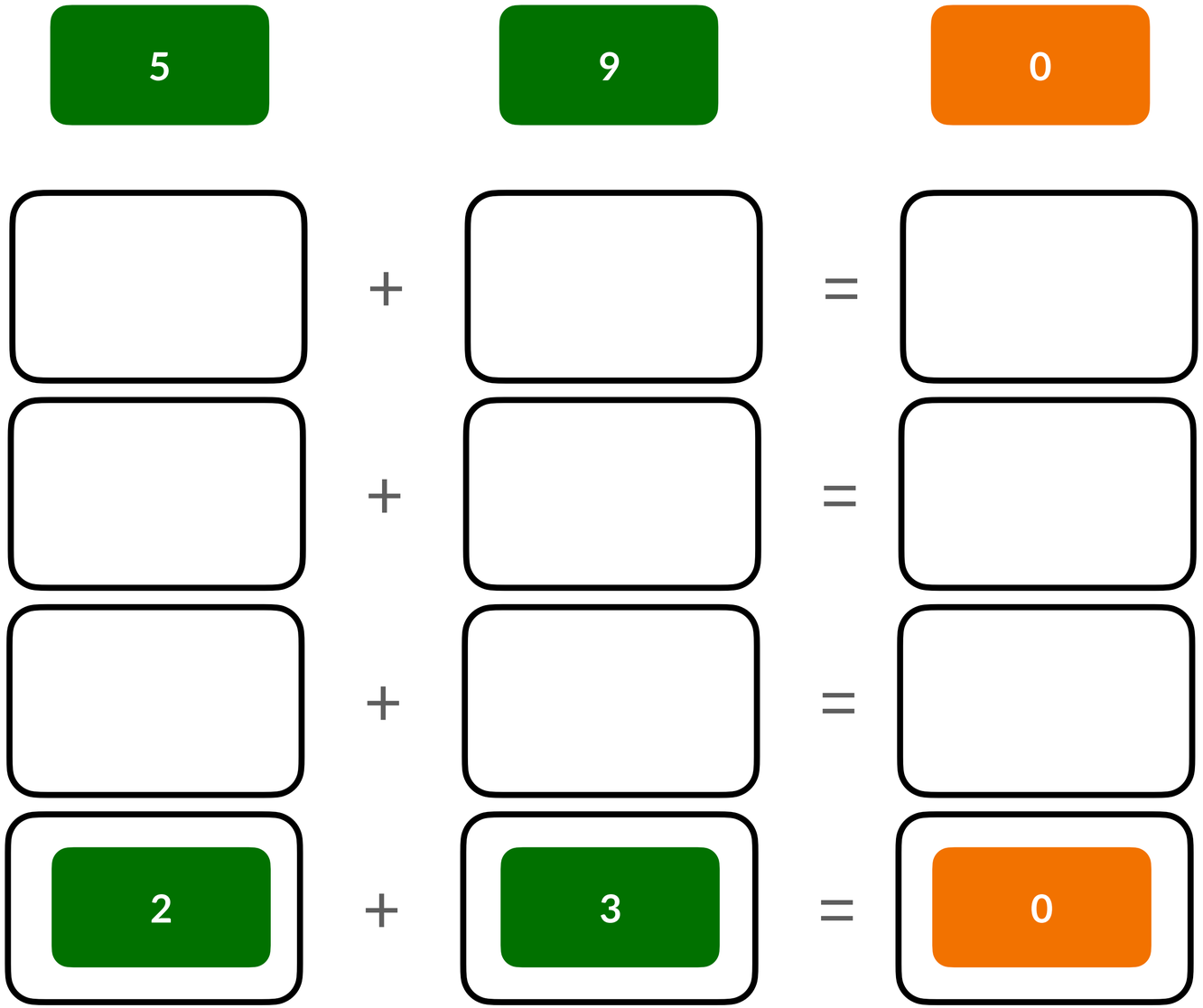
https://openjdk.org/jeps/448

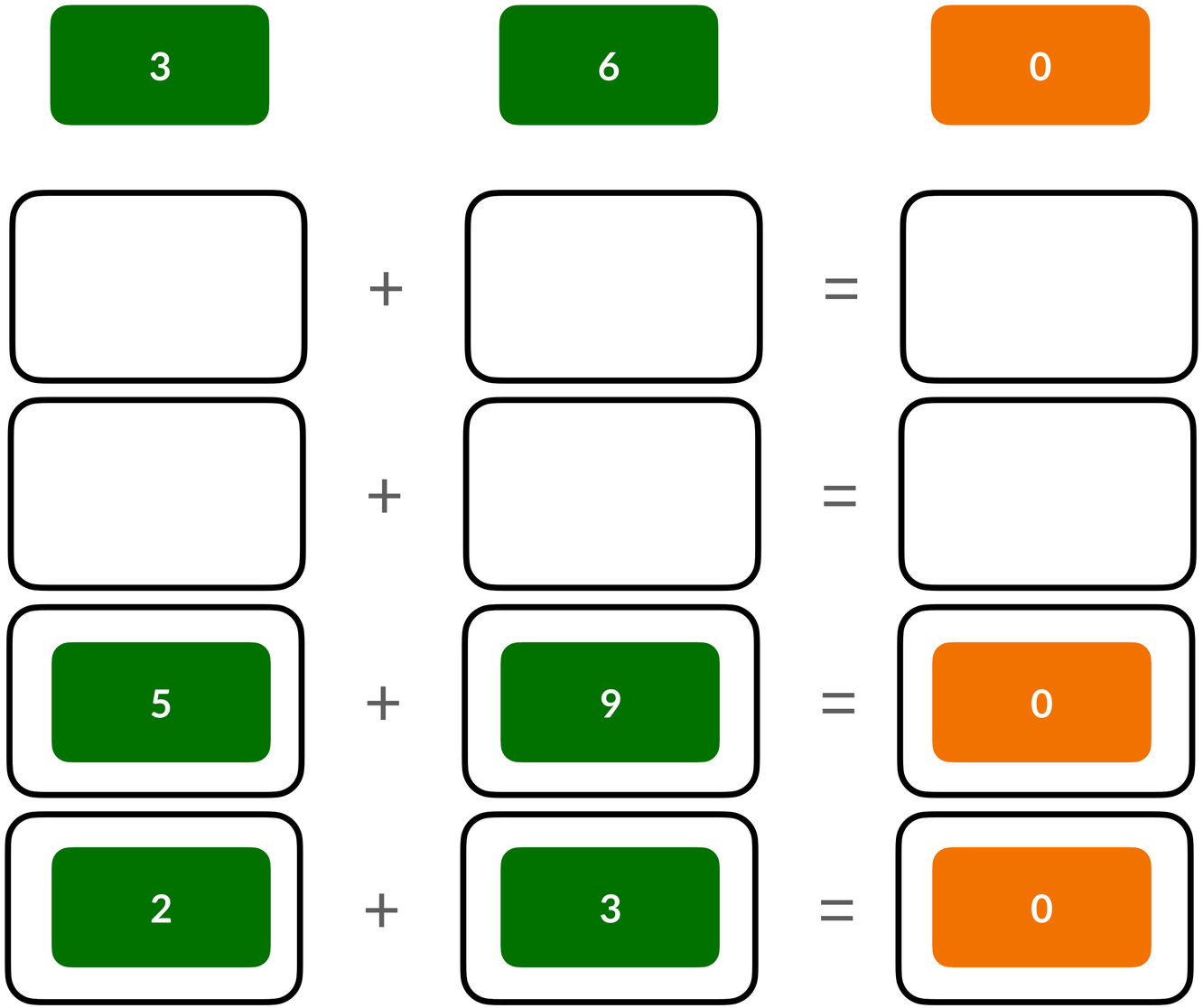
`Vector<E>`

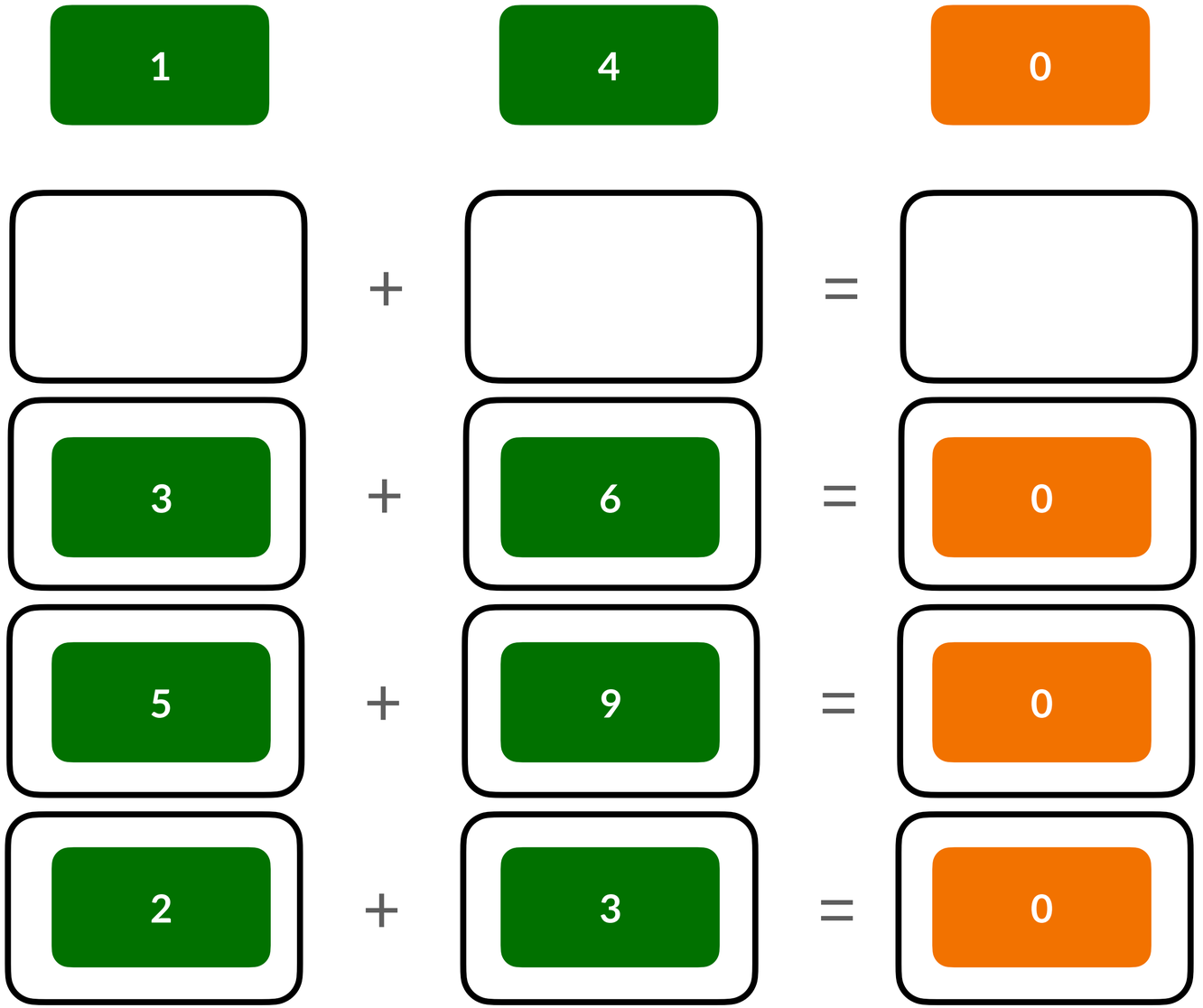
- The `Vector<E>` class has six abstract subclasses for each of the six supporting types: `ByteVector`, `ShortVector`, `IntVector`, `LongVector`, `FloatVector`, and `DoubleVector`.
- Specific implementations are important with SIMD machines, which is why shape-specific subclasses further extend these classes for each type.
- `Vector<E>` is mapped to a hardware vector register when vector computations are compiled by the HotSpot C2 compiler.
- For example `Int128Vector`, `Int512Vector`, etc. These will be abstracted.

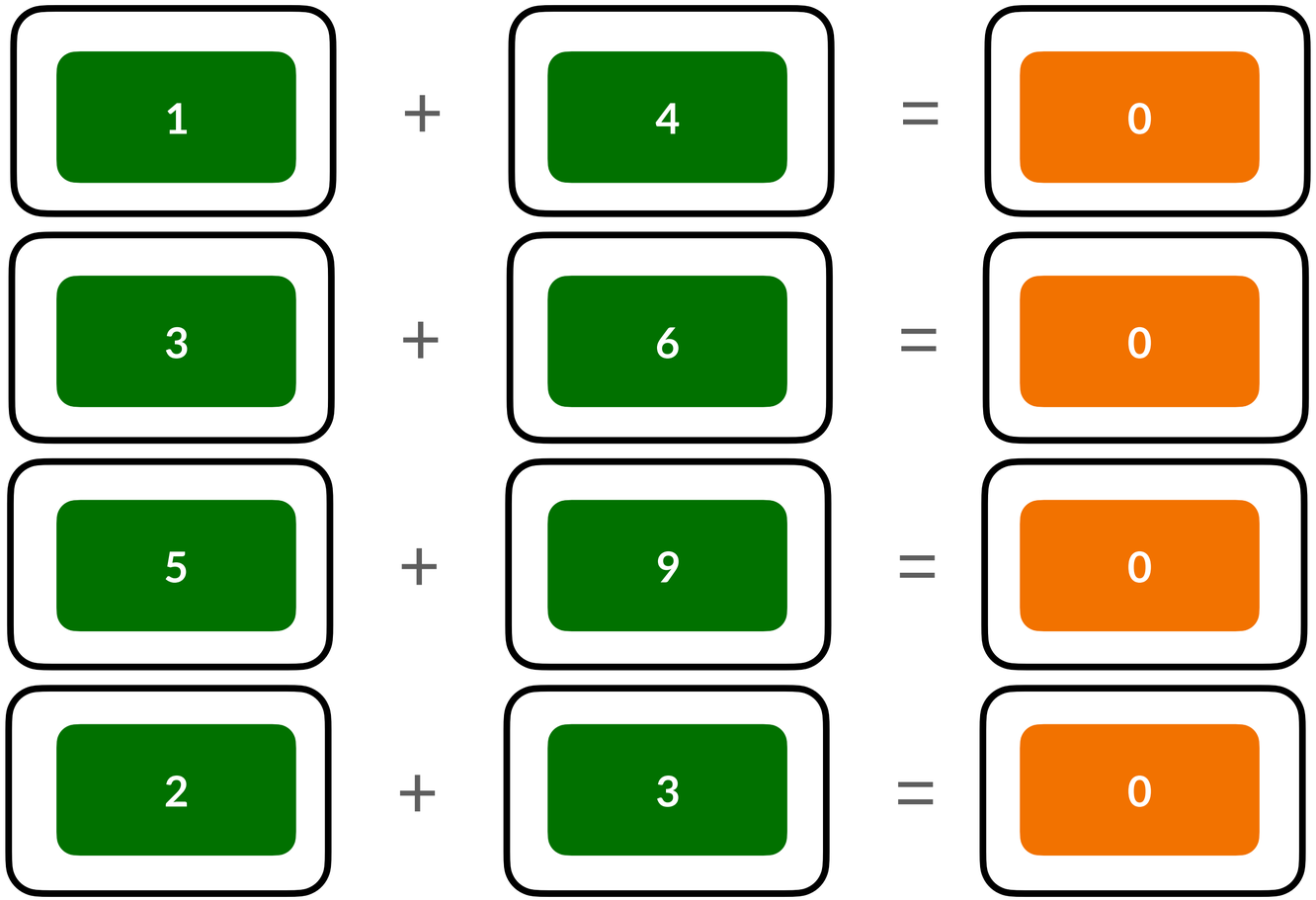
https://openjdk.org/jeps/448

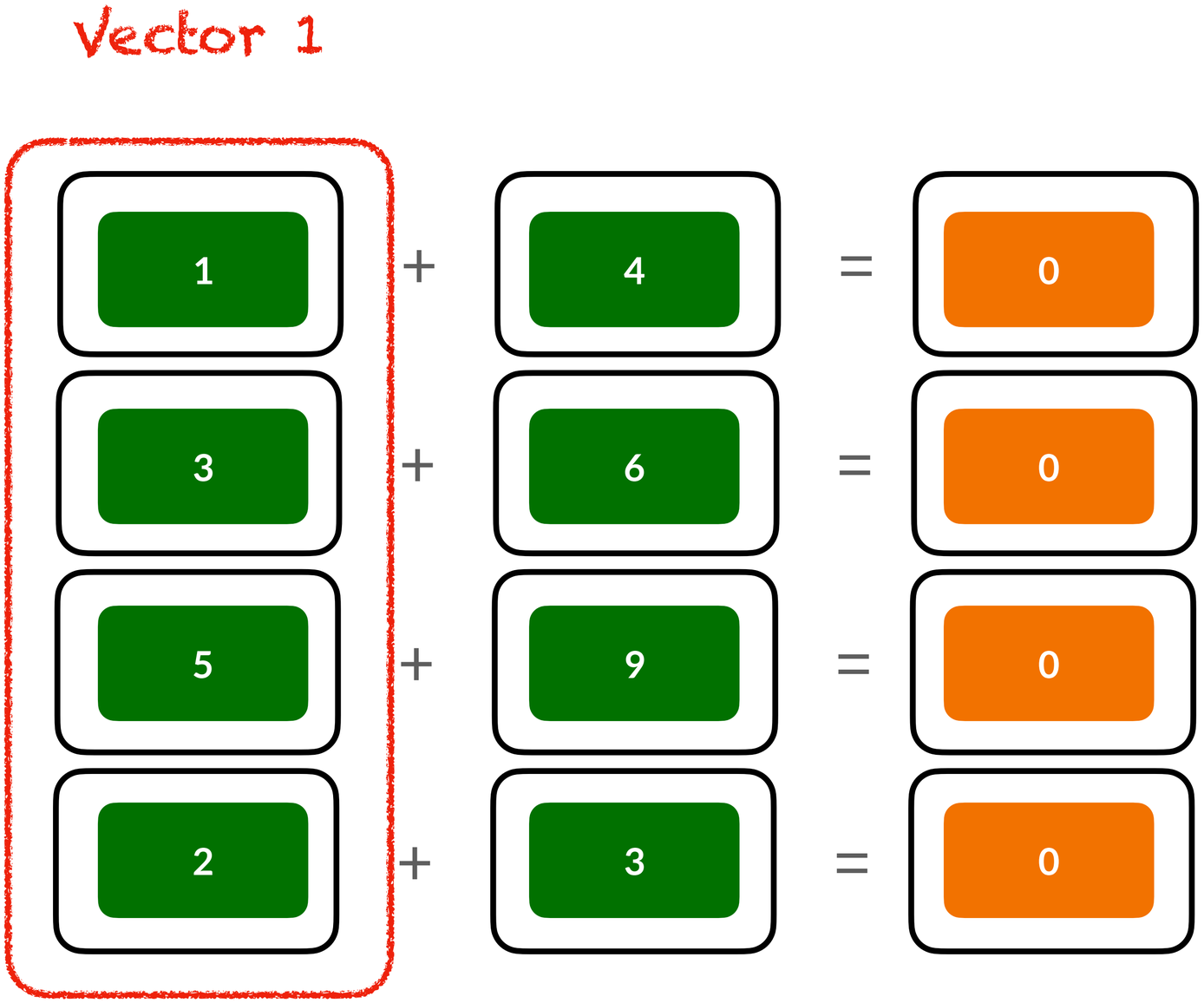

2
3
0
+
=
+
=
+
=
+
=

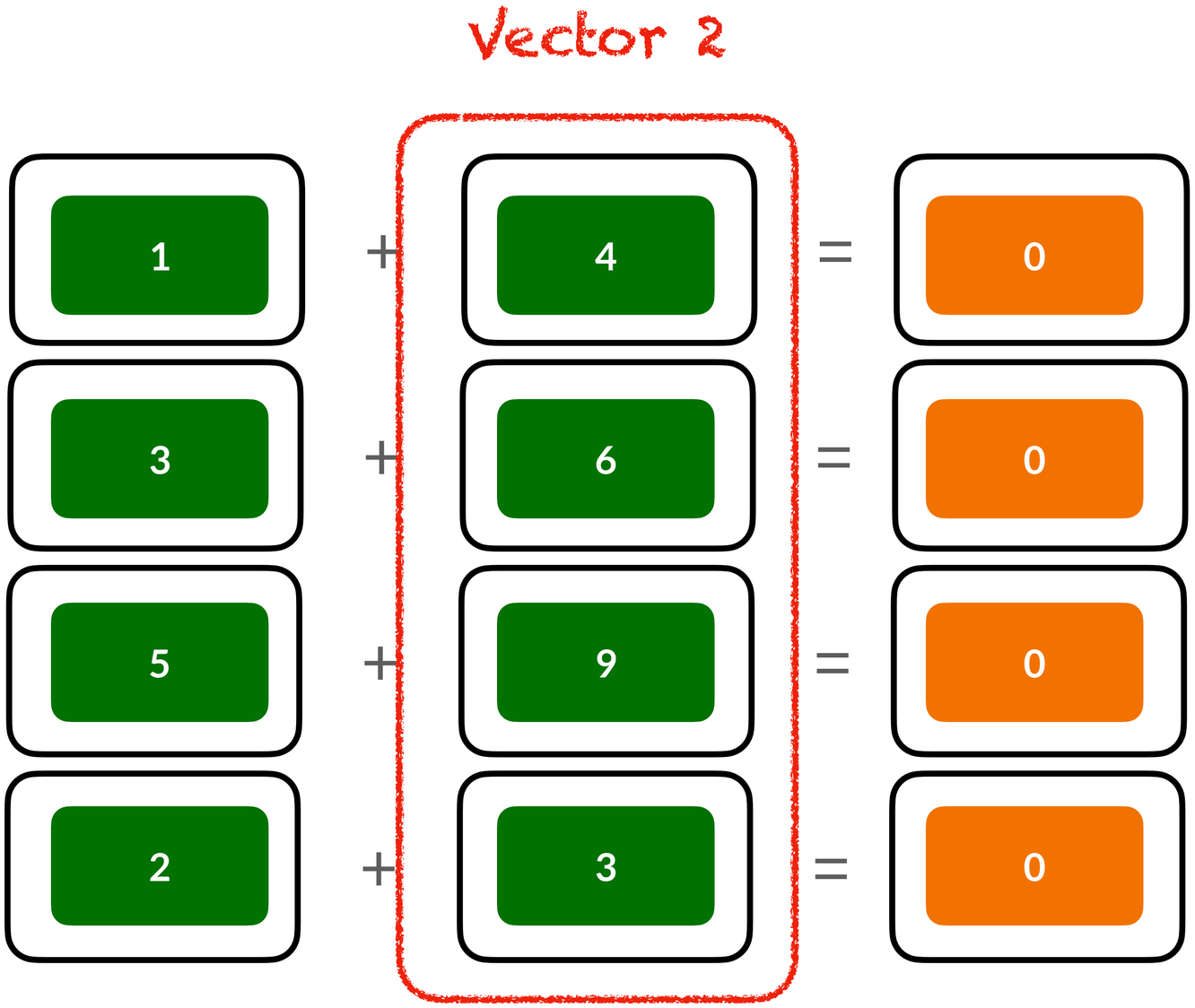

5
9
0
+
=
+
=
+
=
2
+
3
=
0

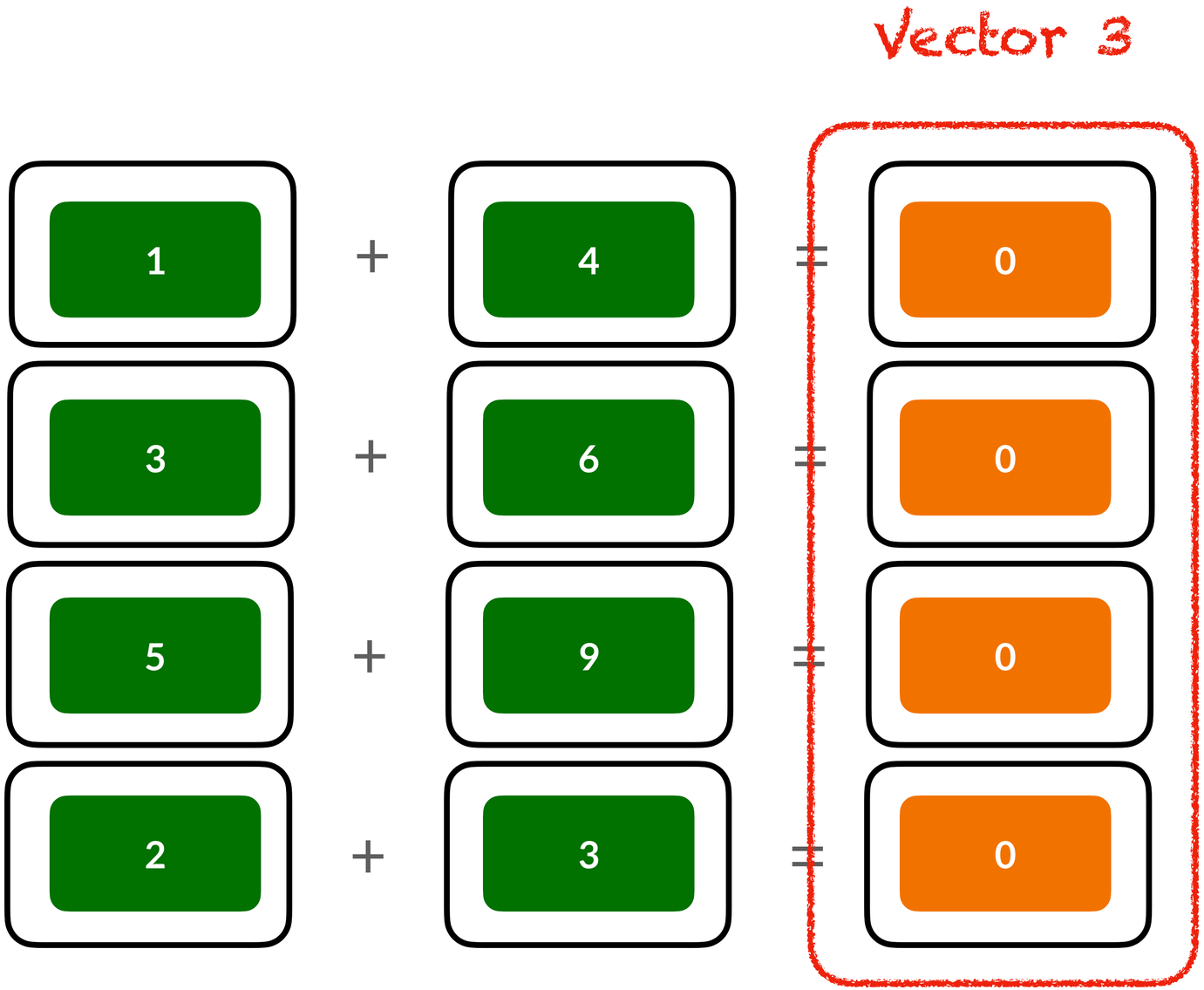

3
6
0
+
=
+
=
5
+
9
=
0
2
+
3
=
0

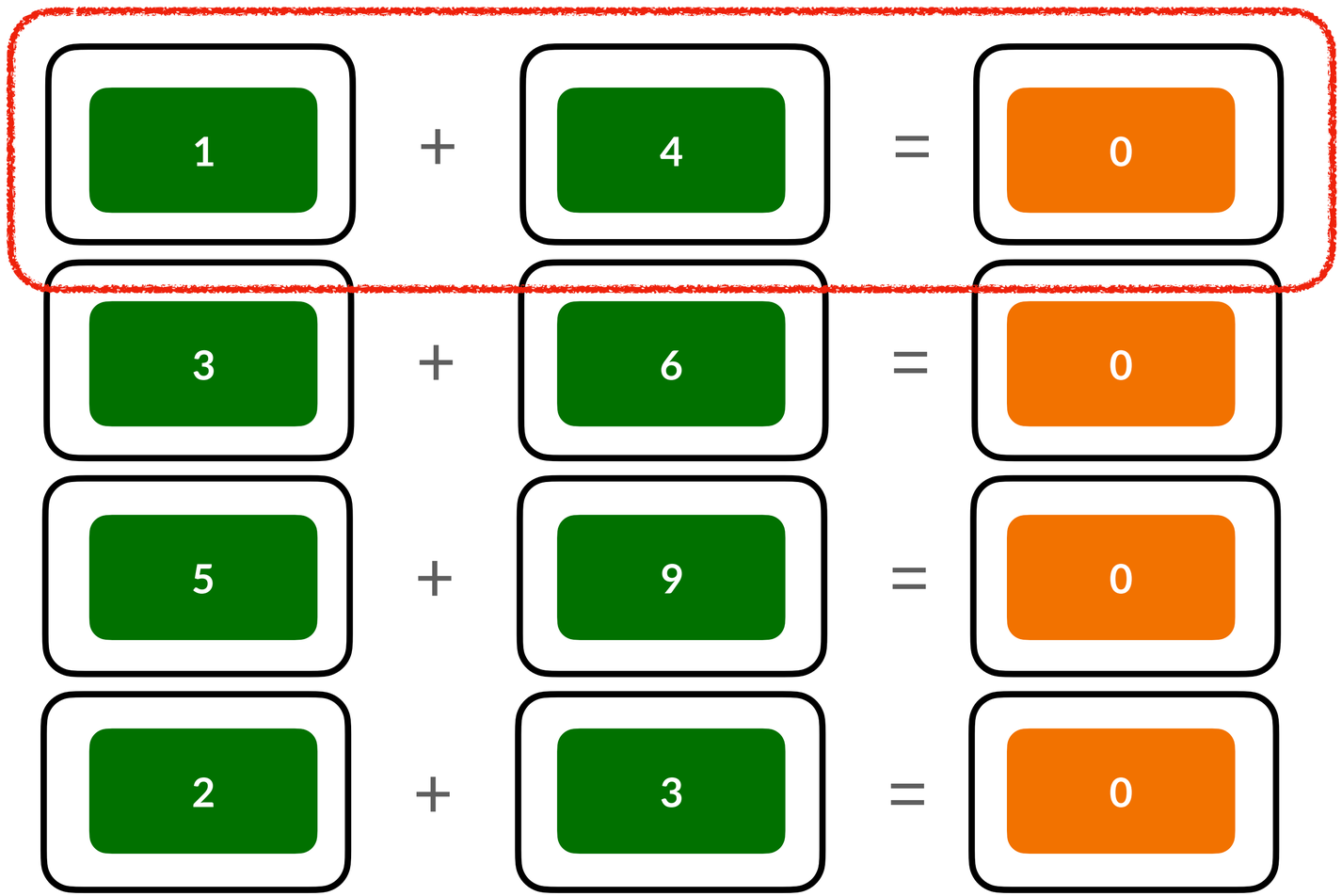

1
4
0
+
=
3
+
6
=
0
5
+
9
=
0
2
+
3
=
0

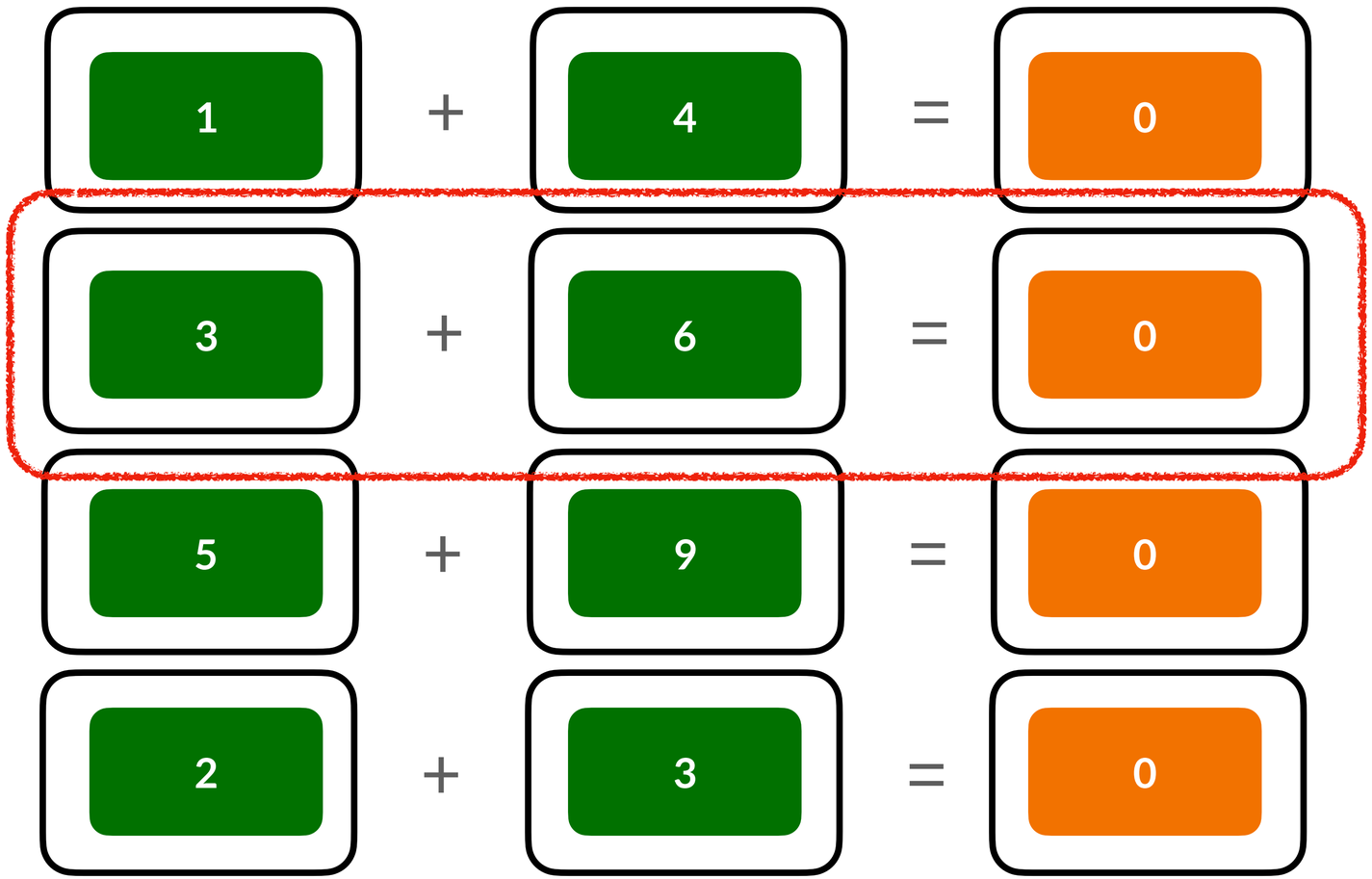

1 + 4 = 0
3 + 6 = 0
5 + 9 = 0
2 + 3 = 0

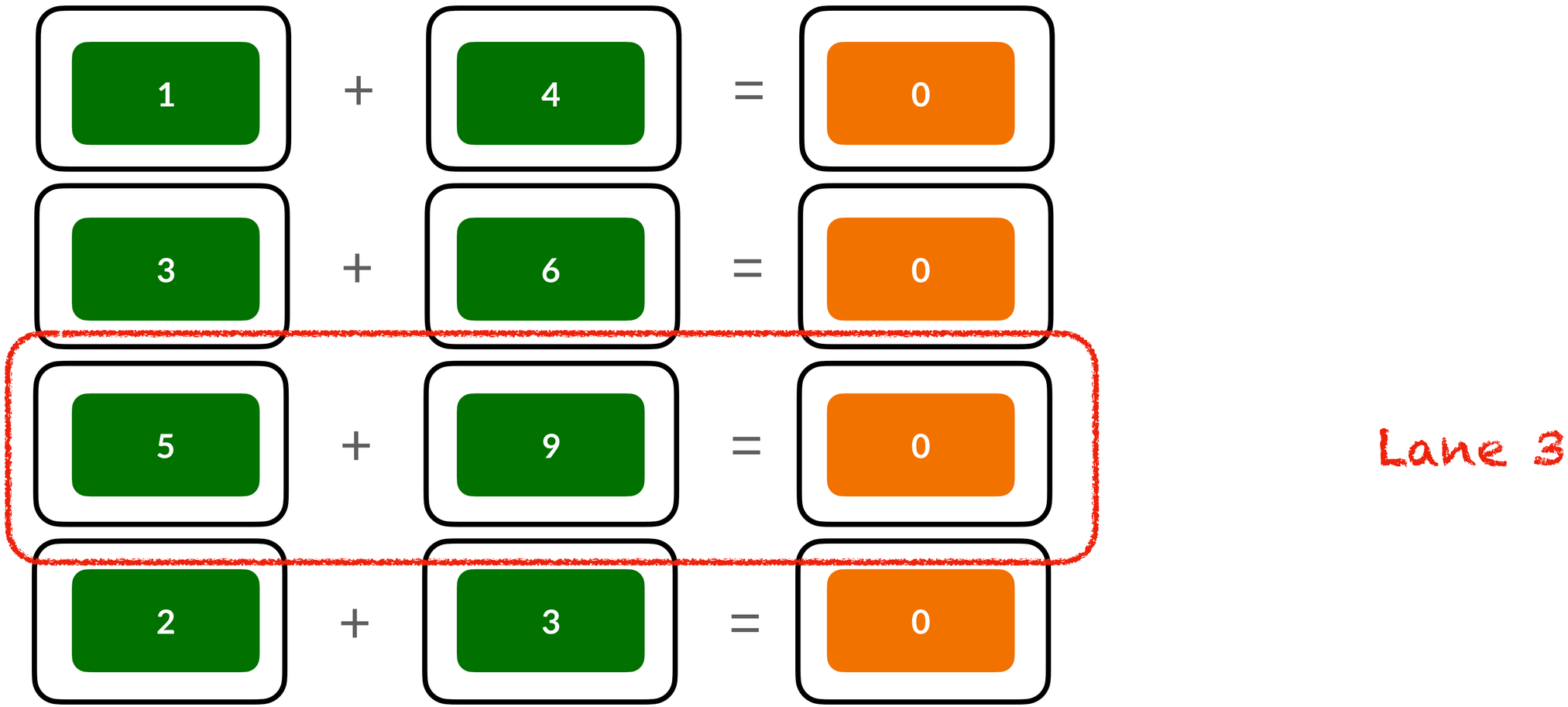

Vector 1
1
+
4
=
0
3
+
6
=
0
5
+
9
=
0
2
+
3
=
0

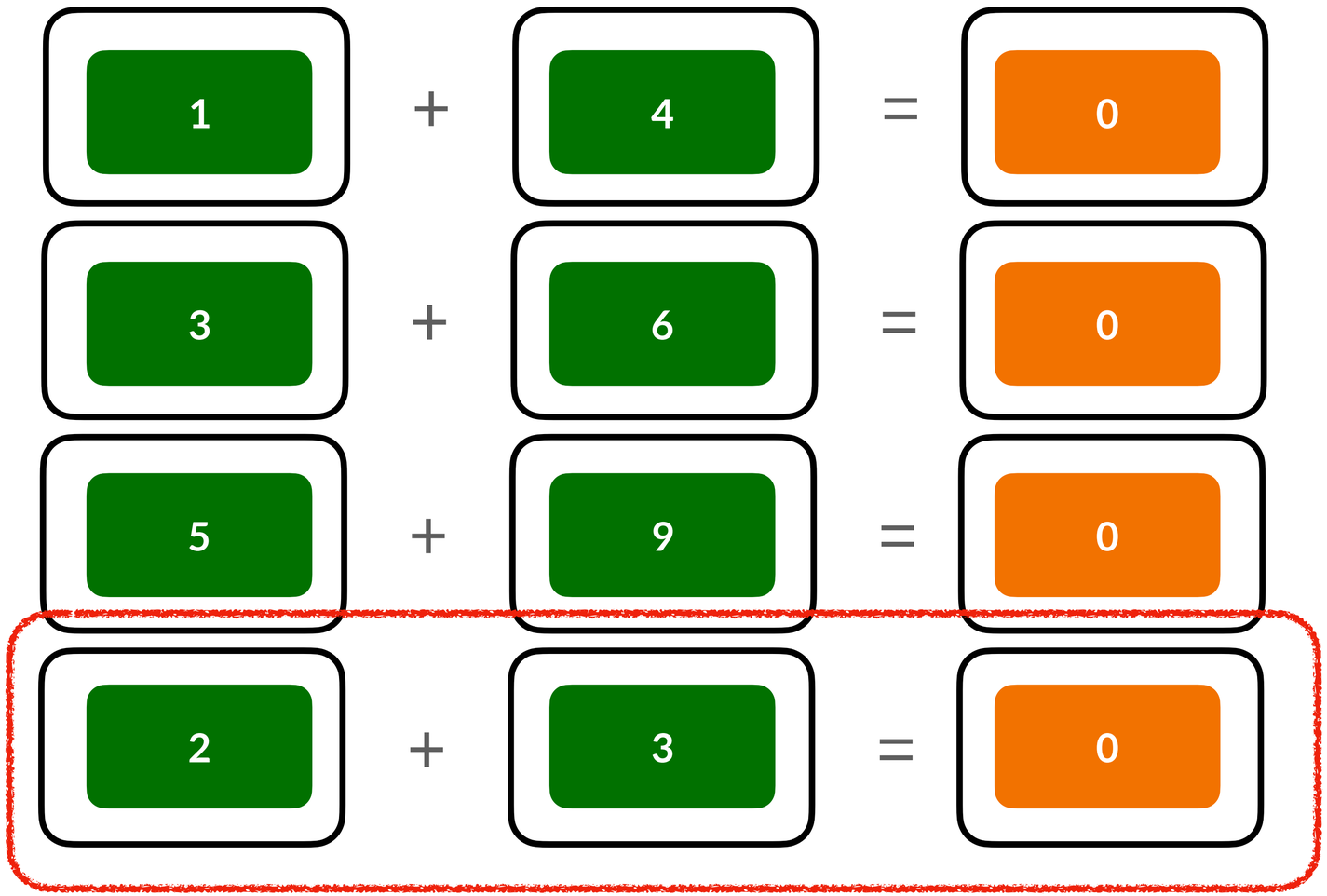

Vector 2
1
+
4
=
0
3
+
6
=
0
5
+
9
=
0
2
+
3
=
0

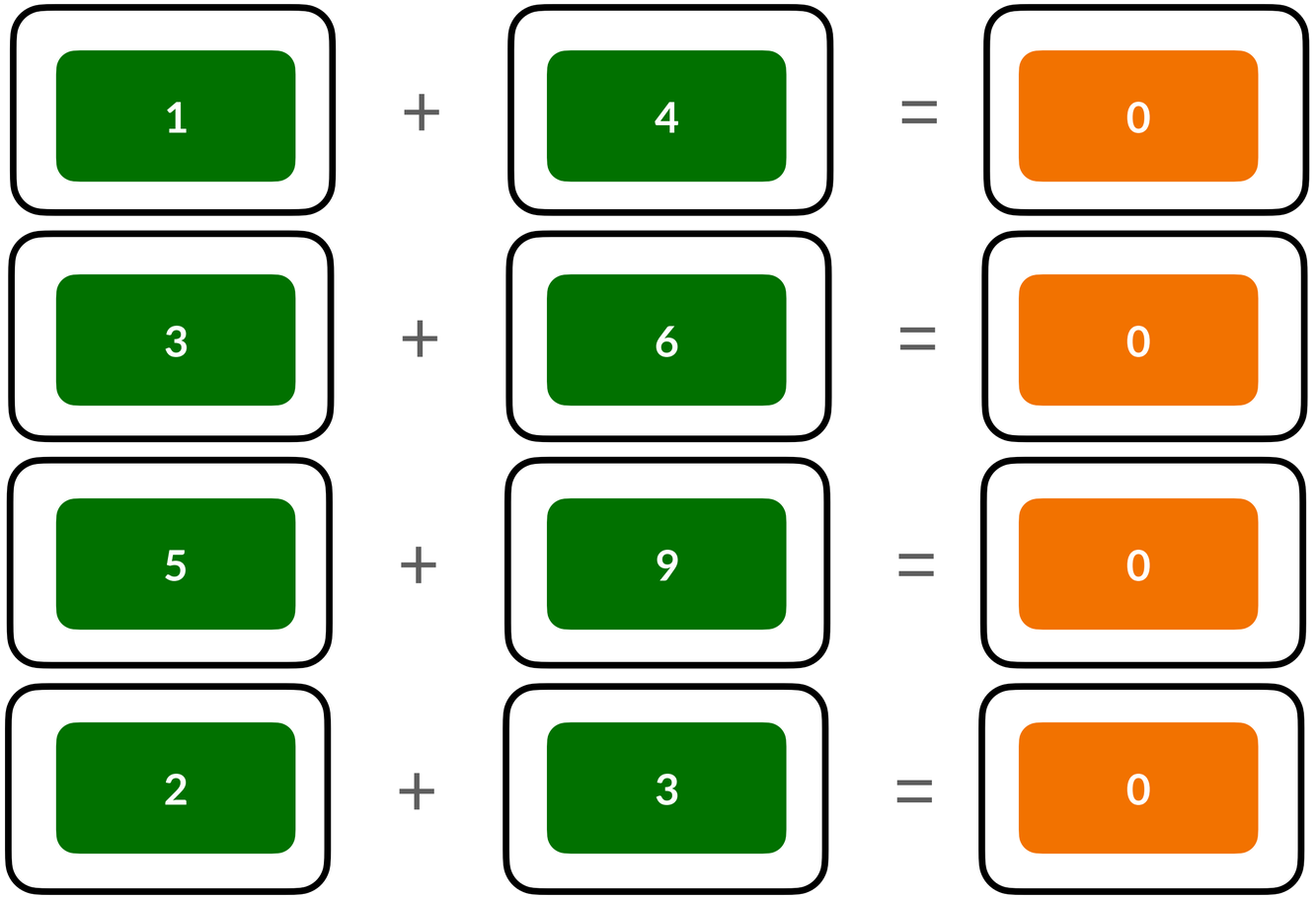

Vector 3
1
+
4
=
0
3
+
6
=
0
5
+
9
=
0
2
+
3
=
0


1 + 4 = 0
3 + 6 = 0
5 + 9 = 0
2 + 3 = 0
Lane 1


1 + 4 = 0
3 + 6 = 0
5 + 9 = 0
2 + 3 = 0
Lane 2


1 + 4 = 0
3 + 6 = 0
5 + 9 = 0
2 + 3 = 0
Lane 3


1 + 4 = 0
3 + 6 = 0
5 + 9 = 0
2 + 3 = 0
Lane 4

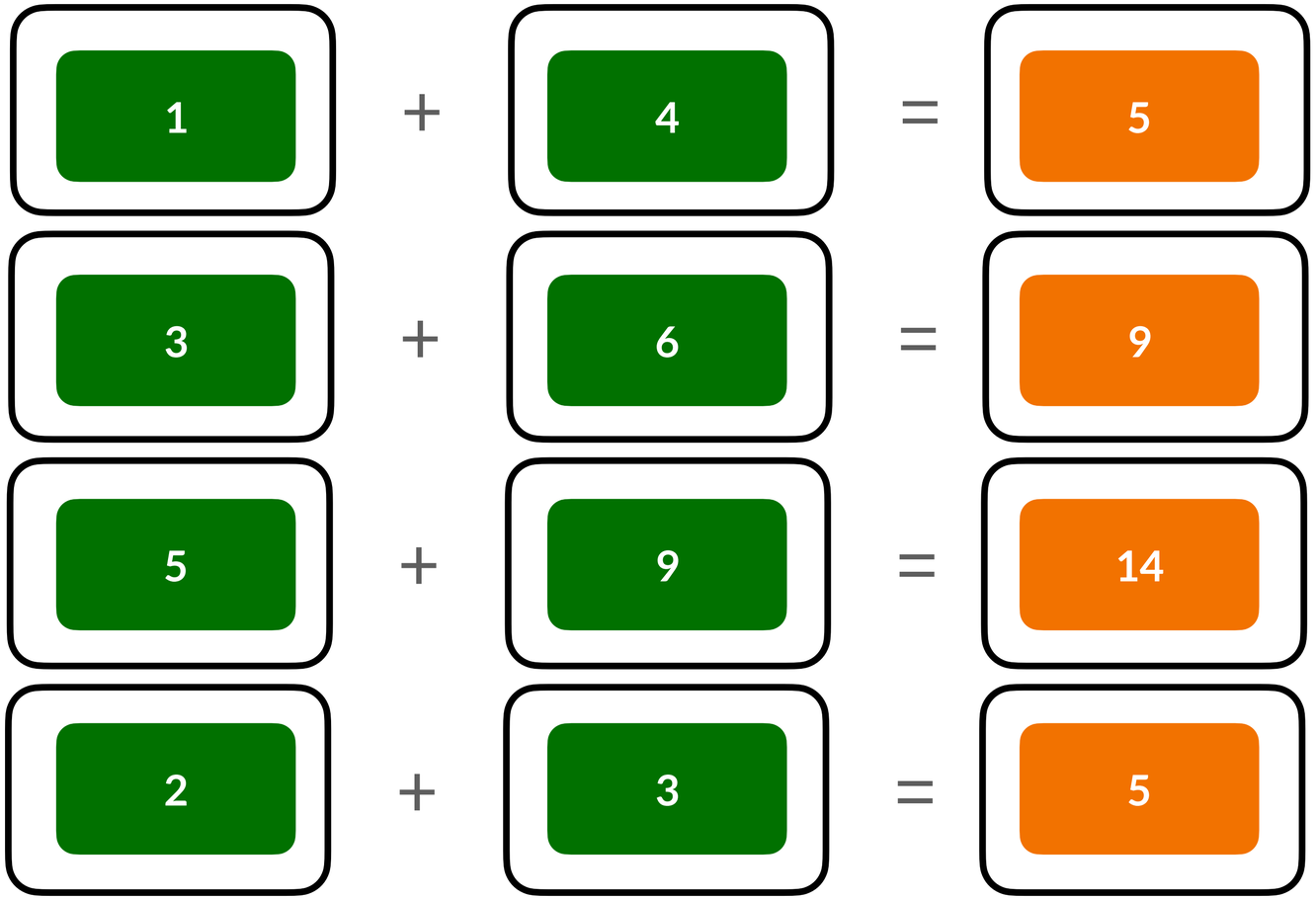


Four Total Components

(In this example)

1	+	4	=	0
3	+	6	=	0
5	+	9	=	0
2	+	3	=	0

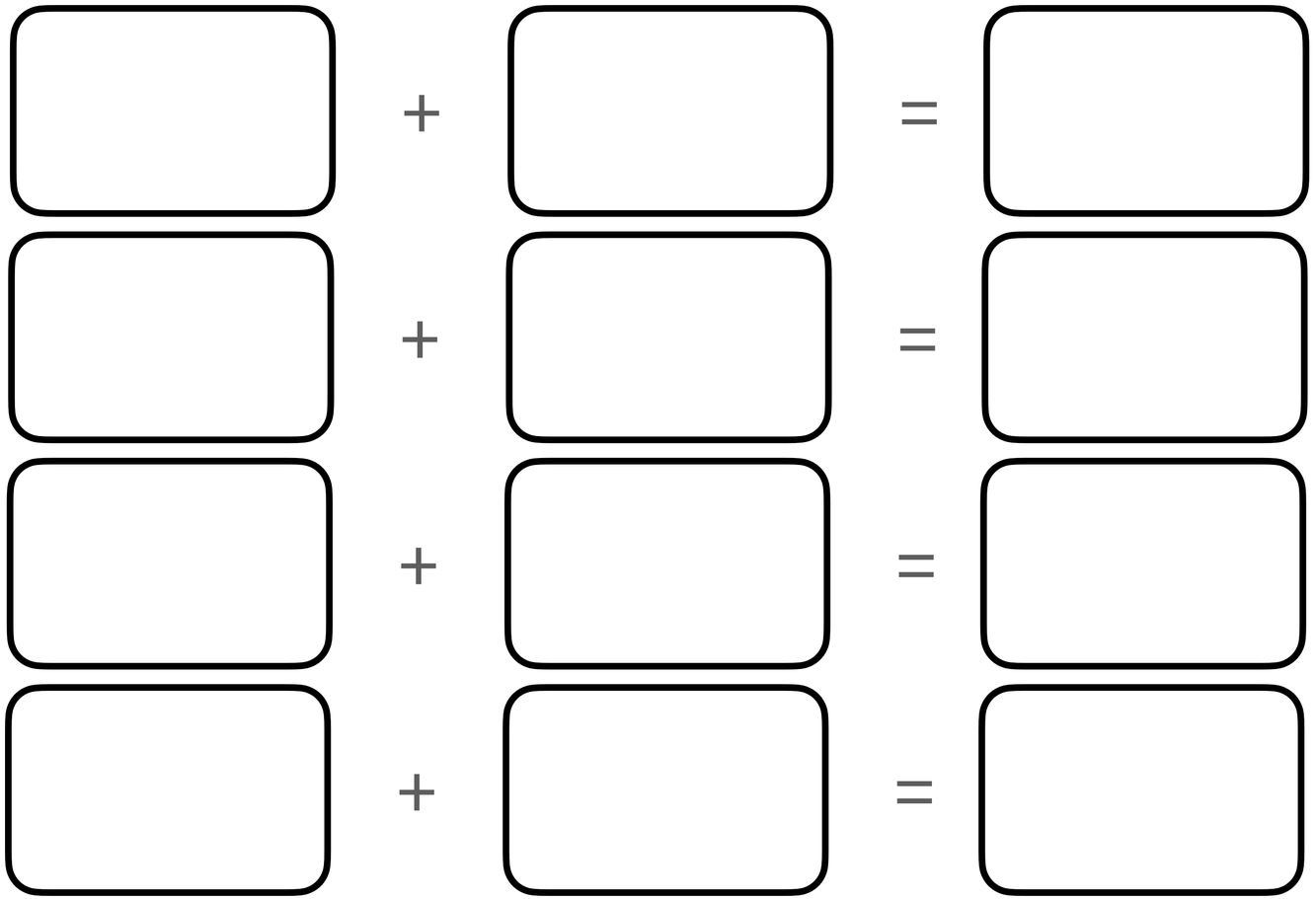
1	+	4	=	5
3	+	6	=	9
5	+	9	=	14
2	+	3	=	5

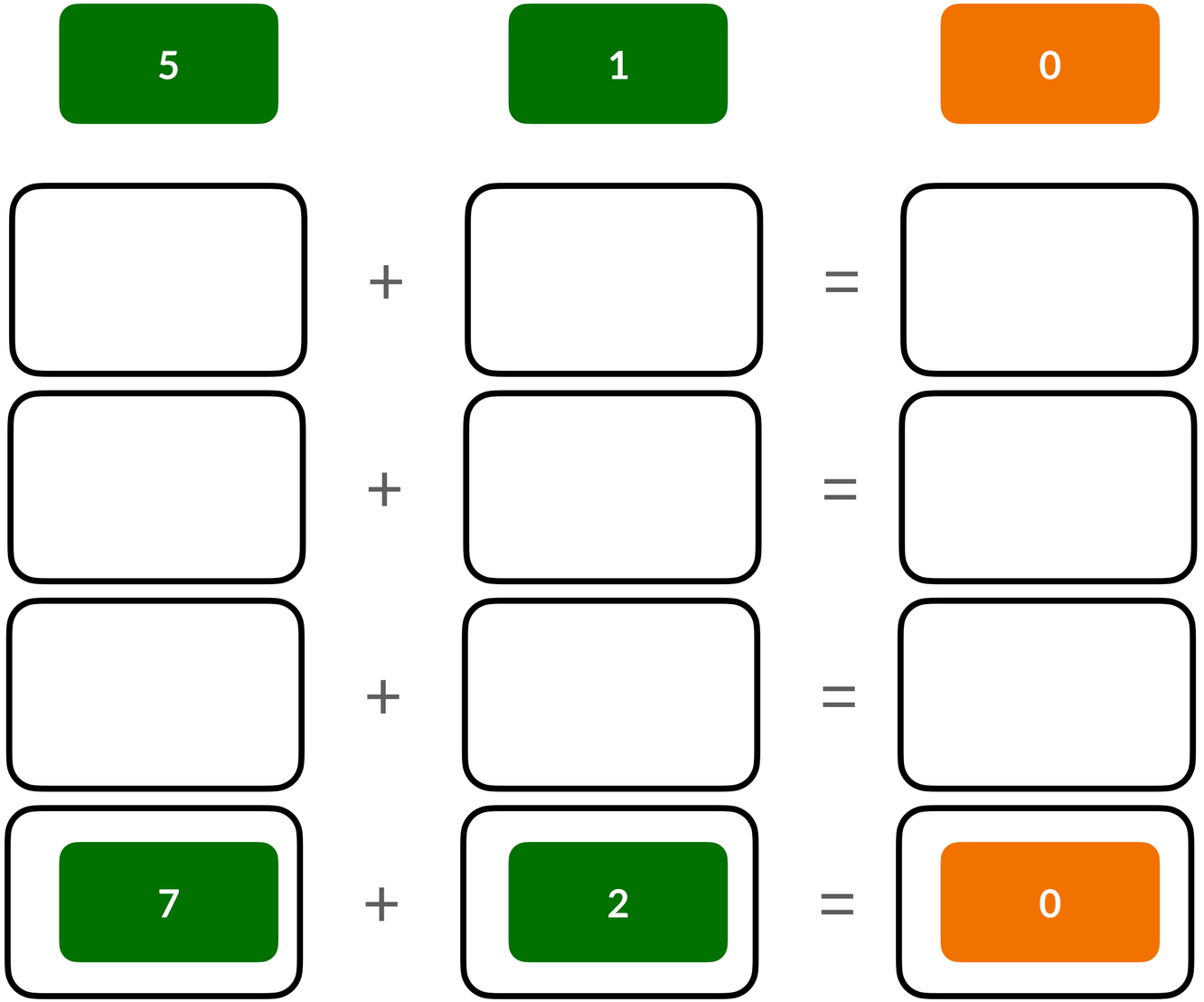

1 + 4 = 5
3 + 6 = 9
5 + 9 = 14
2 + 3 = 5

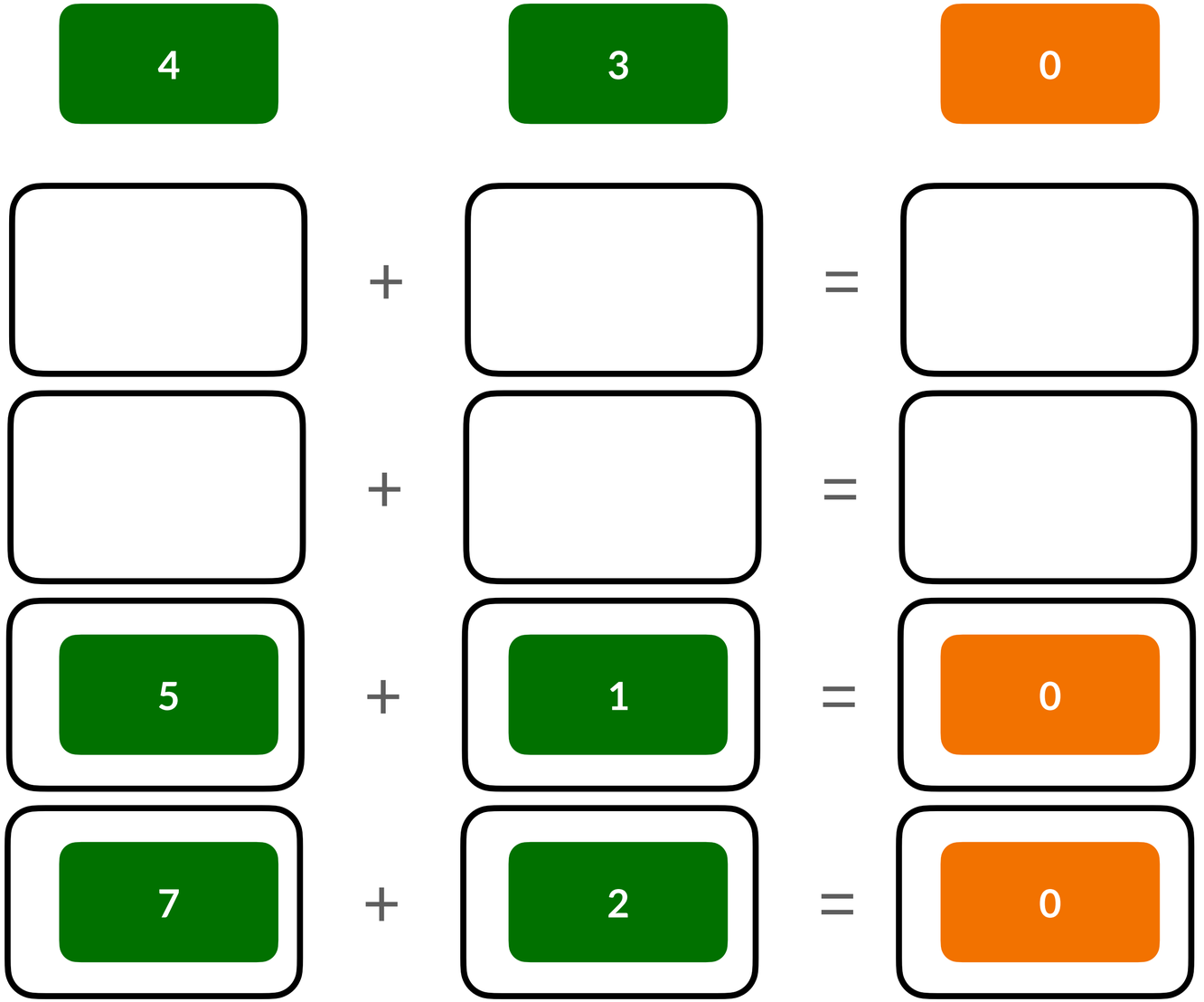
7

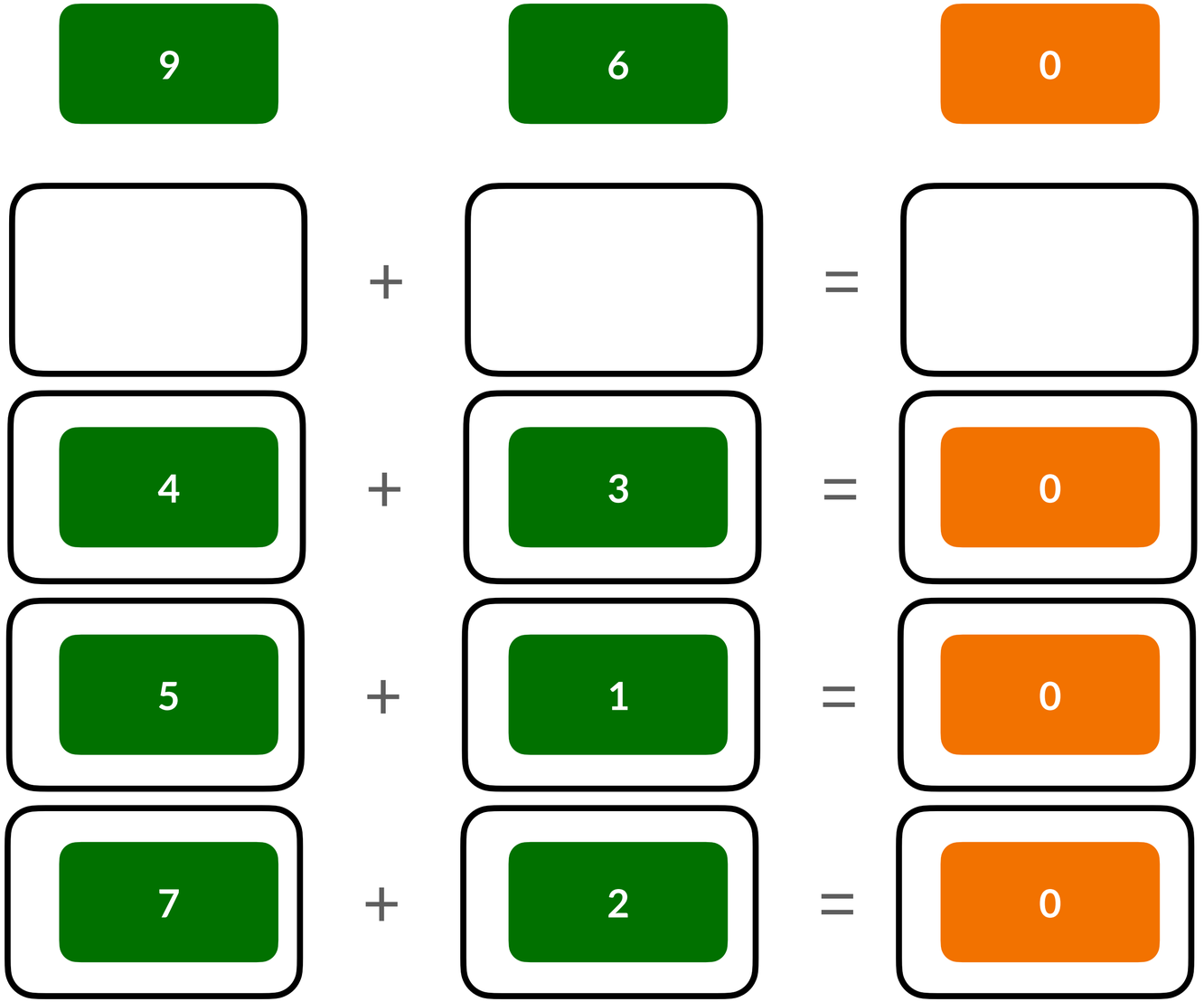
2

0




5
1
0
+
=
+
=
+
=
7
+
2
=
0


4
3
0
+
=
+
=
5
+
1
=
0
7
+
2
=
0


9
6
0
+
=
4
+
3
=
0
5
+
1
=
0
7
+
2
=
0

9	+	6	=	0
4	+	3	=	0
5	+	1	=	0
7	+	2	=	0

9	+	6	=	15
4	+	3	=	7
5	+	1	=	6
7	+	2	=	9

Hardware

What Modern Machinery has SIMD?

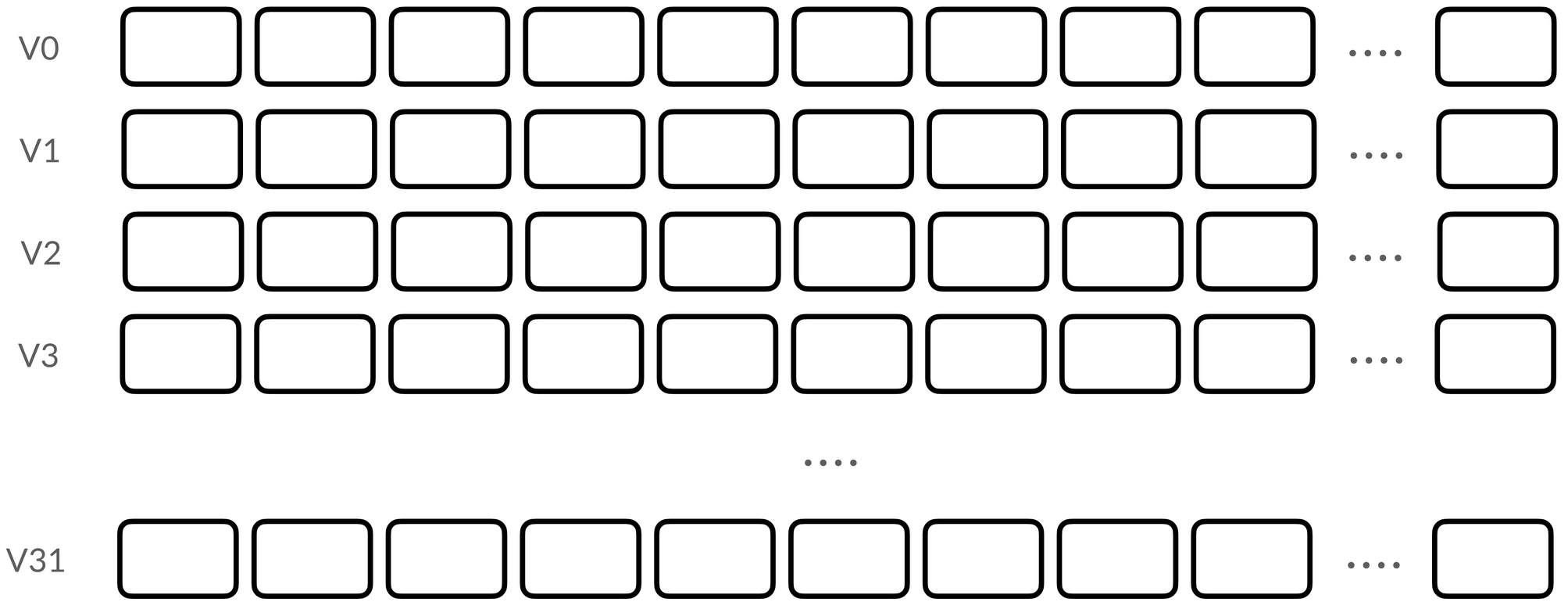
- All x86 processors with AVX Based Instruction Sets
- ARM processor with Neon Instruction Set
- Other processors are available, although it is not a goal to support vector instructions on CPU architectures other than x86 and AArch64

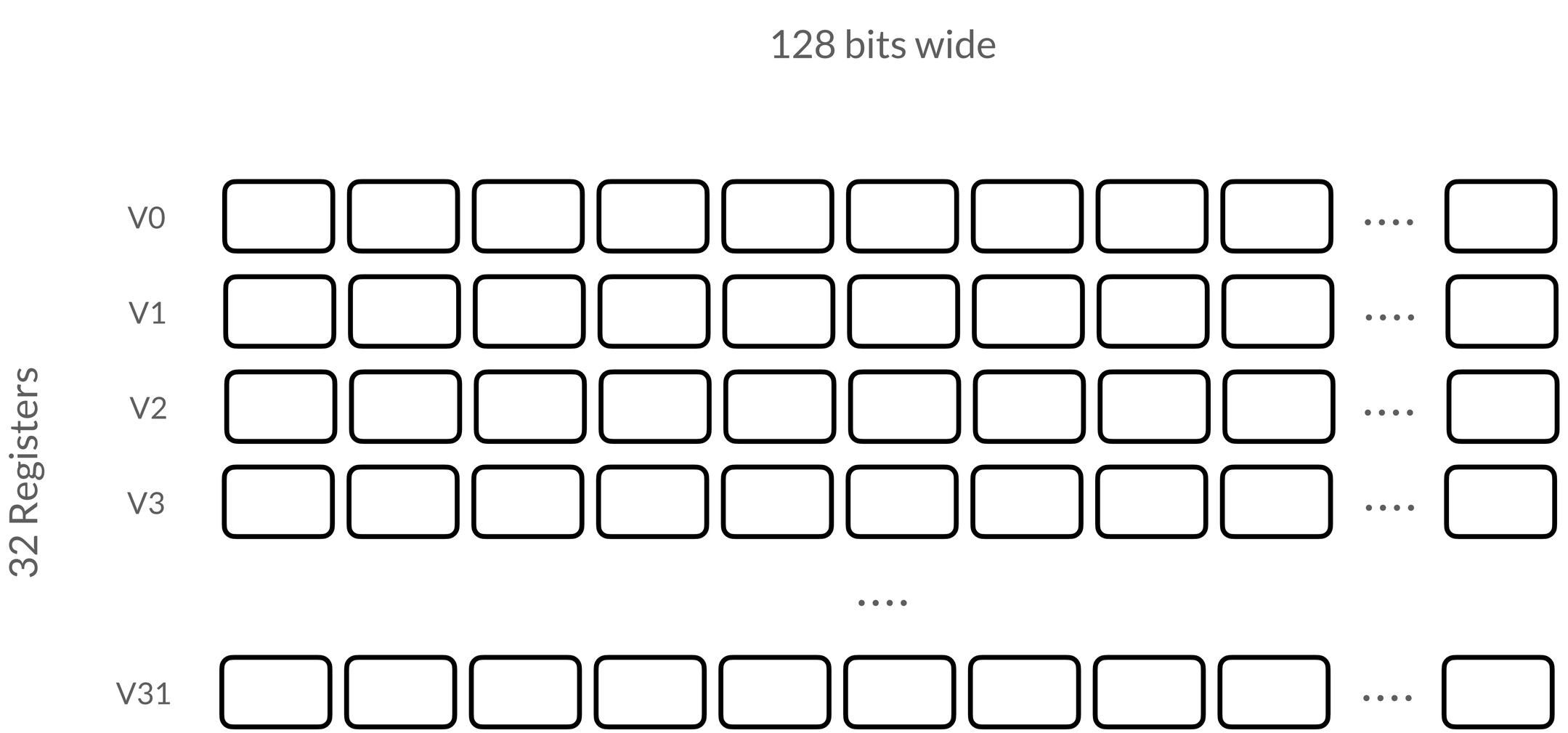
https://en.wikipedia.org/wiki/Single_instruction

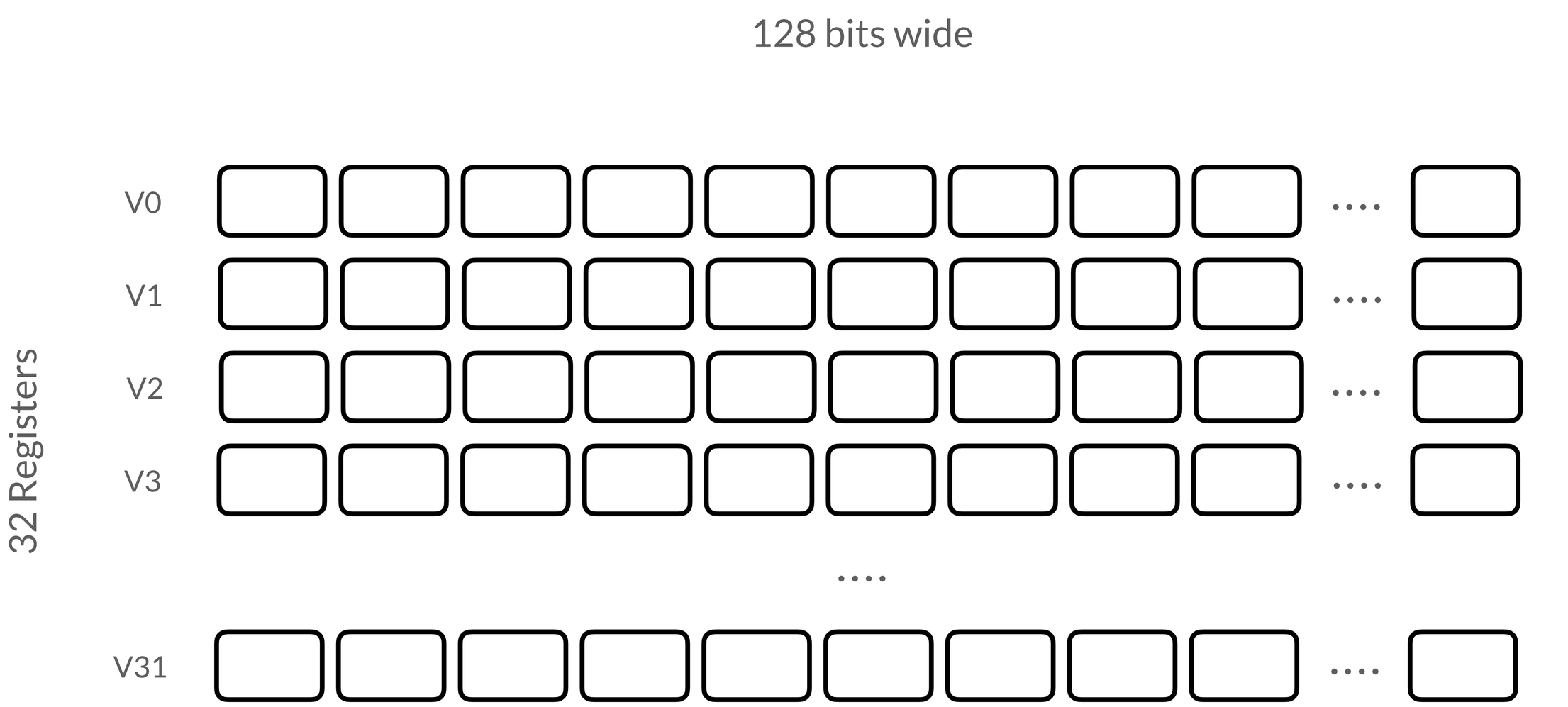
- Each core (Firestorm and Icestorm) has 128-bit wide SIMD registers:
  - **Firestorm:** (~ 3.2 GHz) Fast, intensive processing
  - **Icestorm:** (~2.0 GHz) Background Tasks, idle loops
  - SIMD registers: 32 registers (V0–V31), each 128 bits wide ***in each core, and not shared***
  - These are used by NEON, the SIMD unit on ARM chips.

128 bits wide

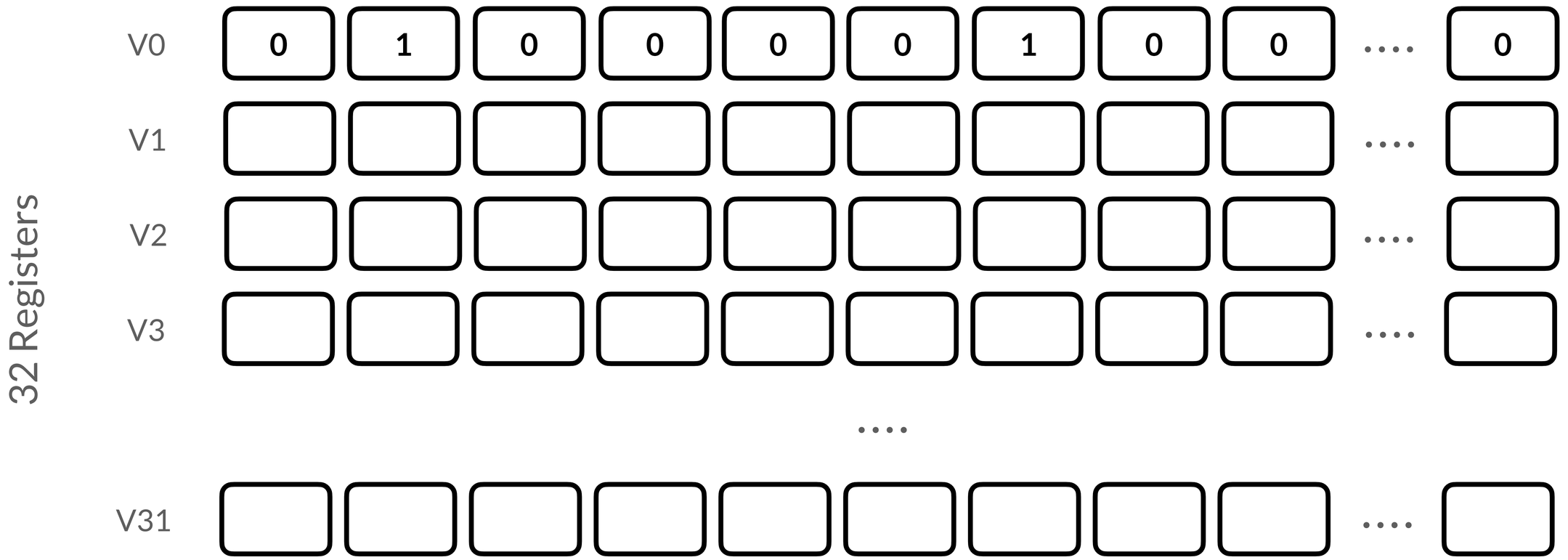
128 bits wide



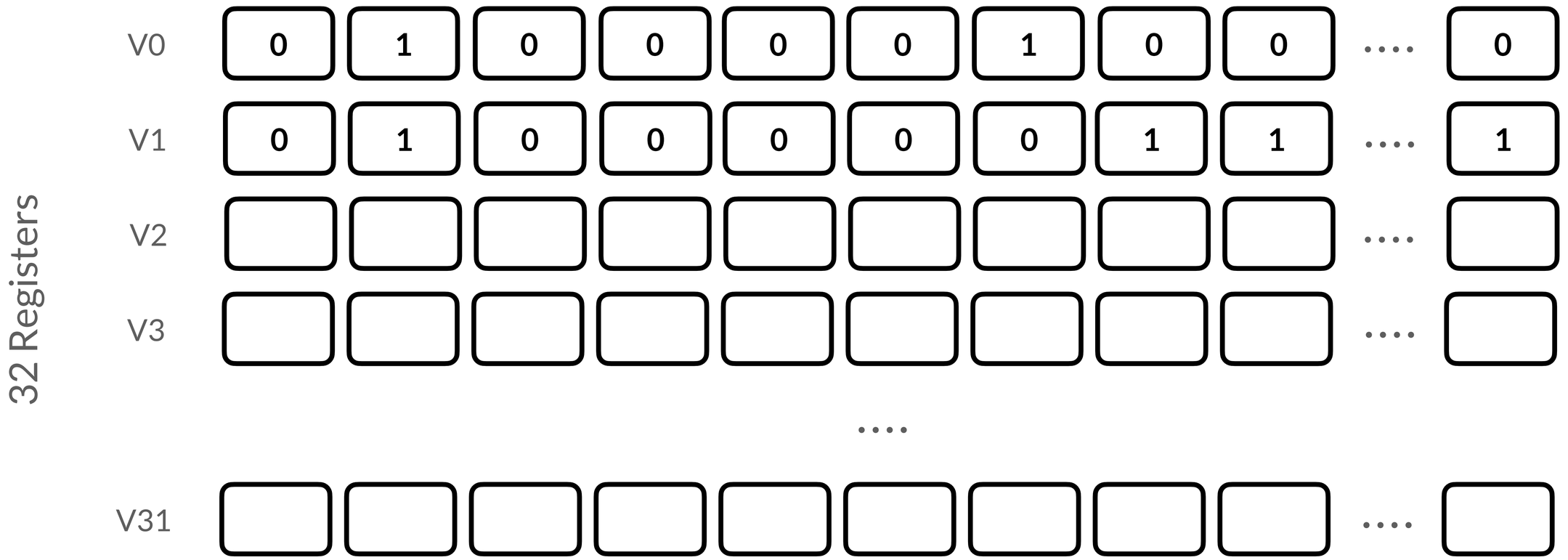

128 bits wide
32 Registers
V0
V1
V2
V3
V31



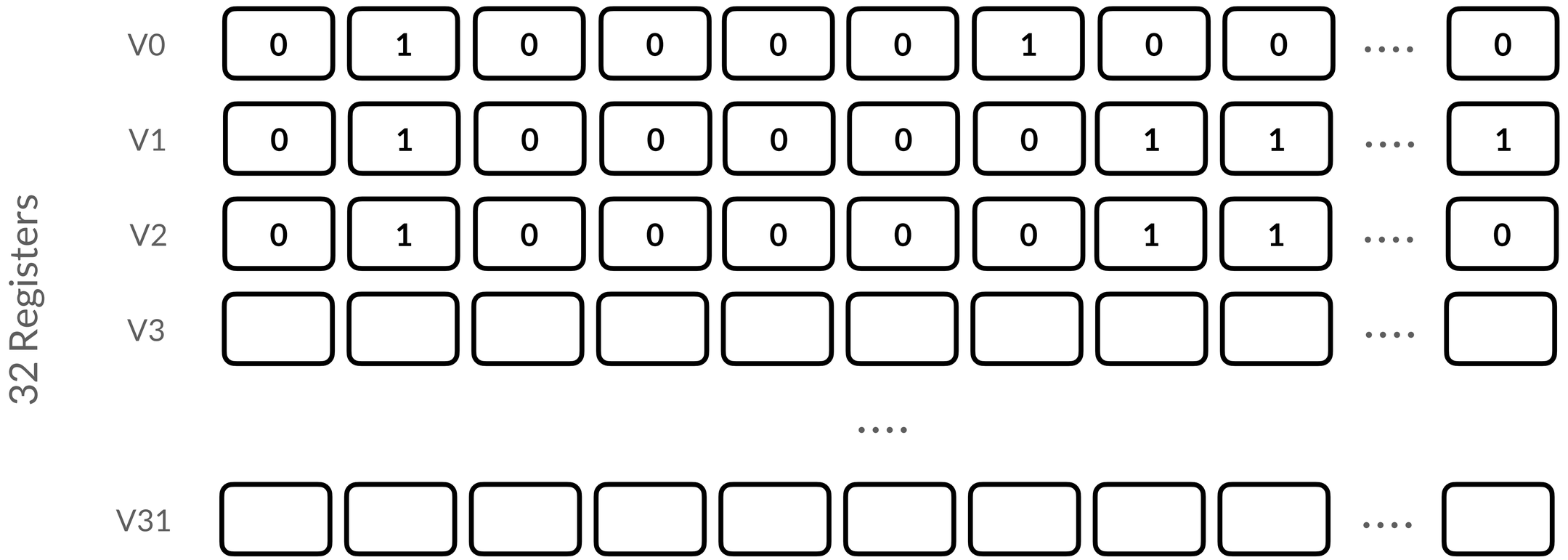

```
a = 34.2f, 89.8f, 45.9f, 78.3f, 67.5f, 2.6f, 23.9f, 101.8f, 90.7f, 85.6f
```




```
a = 34.2f, 89.8f, 45.9f, 78.3f, 67.5f, 2.6f, 23.9f, 101.8f, 90.7f, 85.6f
```




```
b = {24.5f, 32.6f, 38.9f, 28.2f, 14.3f, 67.5f, 54.5f, 77.8f, 46.5f, 89.5f}
```




```
c = {58.7f, 122.4f, 84.8f, 106.5f}
```

SIMD Register Width (w) = 128 bits

Size for Float (s) = 32 bits

Number of Lanes = w/s

SIMD Register Width (w) = 128 bits

Size for Float (s) = 32 bits

Number of Lanes = 128/32

SIMD Register Width (w) = 128 bits

Size for Float (s) = 32 bits

Number of Lanes = 4

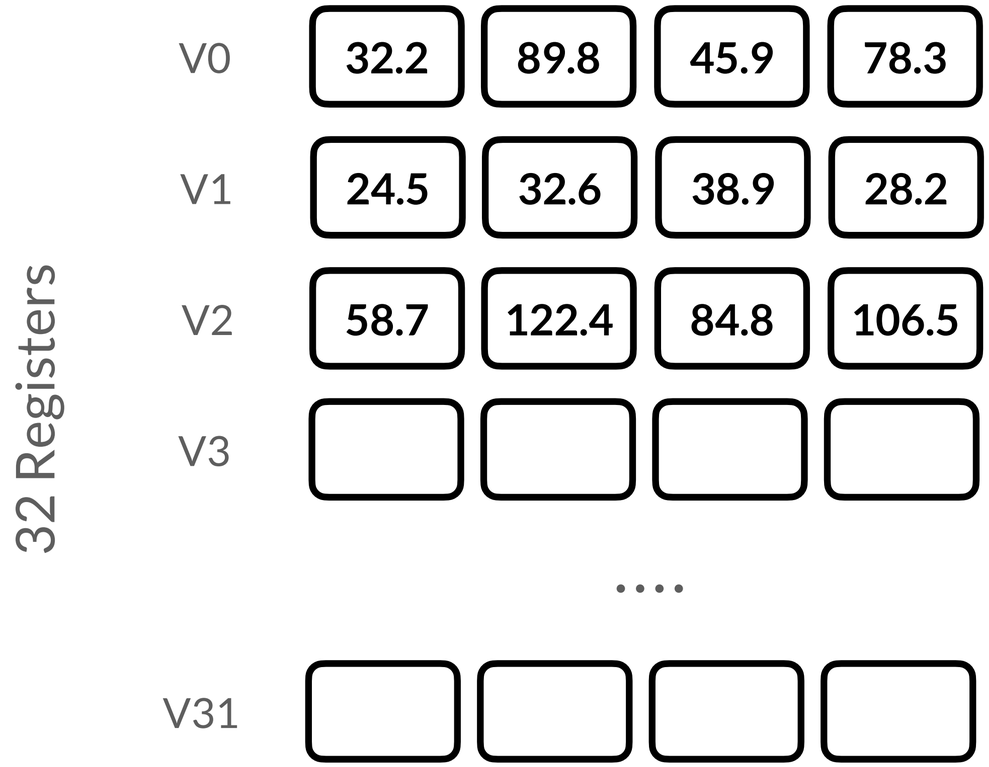
SIMD Register Width (w) = 128 bits

Size for Float (s) = 32 bits

Number of Lanes = 4

(On my machine)

128 bits wide (grouped by 4 since 128/32 = 4)




```
c = {58.7f, 122.4f, 84.8f, 106.5f}
```

Demo: Creating a Vector API Application

- Here we will create a Vector API application in a simple form that will be an exact fit with the number of components
- We will run `AddStandardVector` and `AddStandardVectorLoop`

Masking

Masking

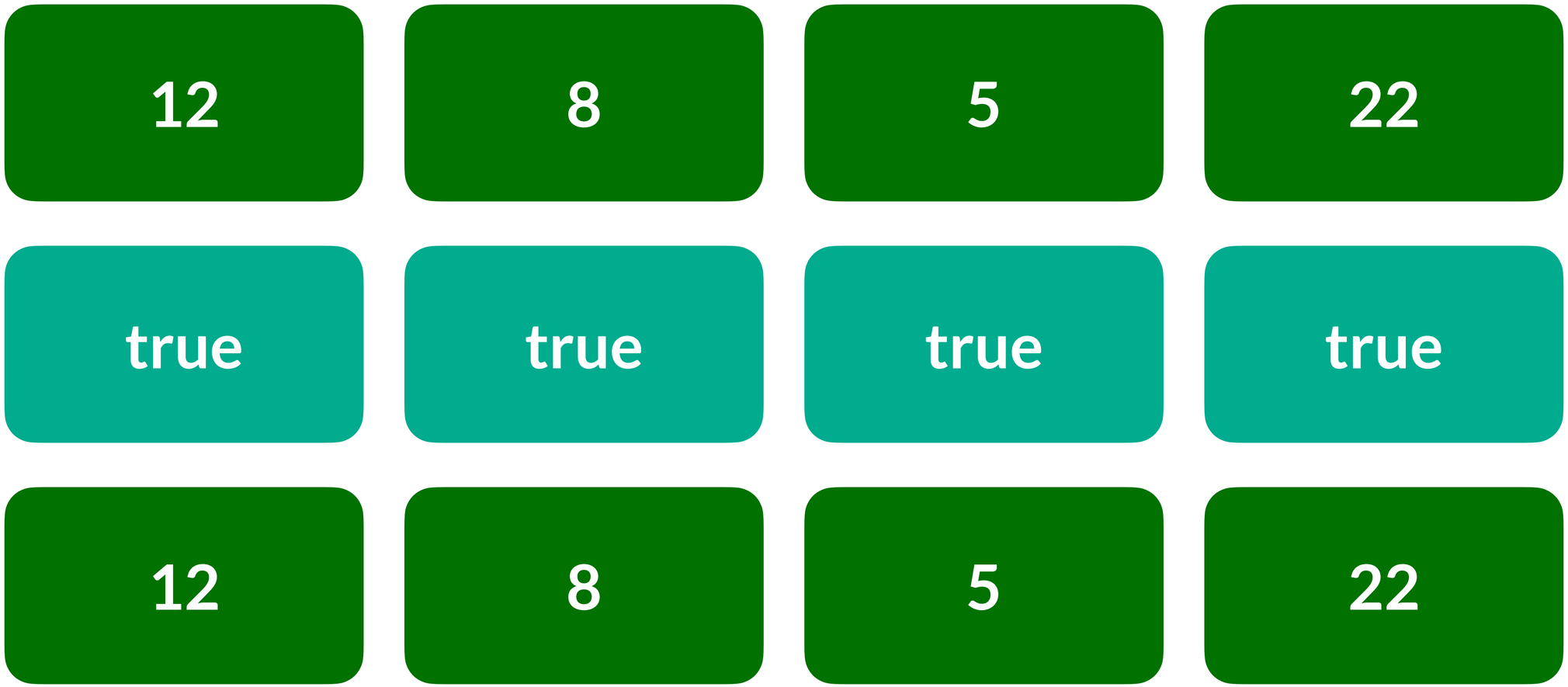
- To support control flow, some vector operations optionally accept masks represented by the public abstract class `VectorMask<E>`
- Each element in a **mask** is a boolean value corresponding to a vector lane.
- A **mask** selects the lanes to which an operation is applied: It is applied if the mask element for the lane is `true`, and some alternative action is taken if the mask is `false`.
- In other words, it's like `window` where each slice will be processed in parallel

https://openjdk.org/jeps/448

Applying Masks

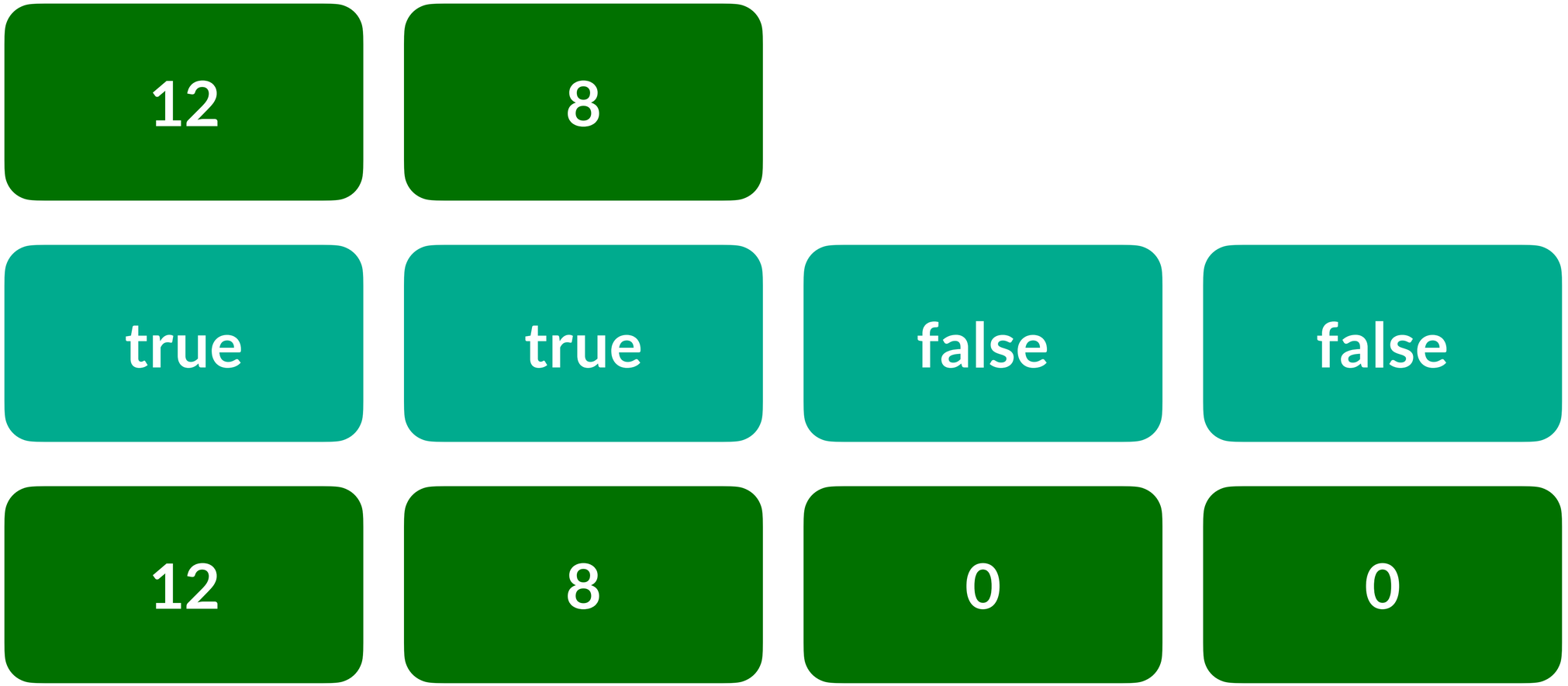
- Similar to vectors, instances of `VectorMask<E>` are instances of non-public concrete subclasses defined for each element type and length combination.
- The instance of `VectorMask<E>` used in an operation should have the same type and length as the vector instances involved in the operation.
- Vector comparison operations produce masks, which can then be used as input to other operations to selectively operate on certain lanes and thereby emulate flow control.
- Masks can also be created using static factory methods in the `VectorMask<E>` class.

https://openjdk.org/jeps/448


12
8
5
22
true
true
true
true
12
8
5
22

12	8	5	22
false	true	false	true
0	8	0	22

12	8	5	22
true	true	false	false
12	8	0	0




```
SPECIES.indexInRange(index, a.length)
```

Demo: Using Masking

- Let’s use some masking so that we can be sure to process all the data in chunks.
- We will run:
  - `AddStandardVectorLoopWithMask`,
  - `AddStandardVectorLoopWithoutMask`

Vector Operations

Vector Operations

- There are two types of vector operations:
  - Lane-wise operations
  - Cross-lane operations

https://openjdk.org/jeps/448

Lane Wise Operations

- Performs a scalar operation, like addition, on a single lane on one or more vectors at a time.
- These operations can combine one lane of a vector with a lane of a second vector, for instance, during an add operation.
- Operations are further classified as unary, binary, ternary, test, or conversion operations

https://openjdk.org/jeps/448

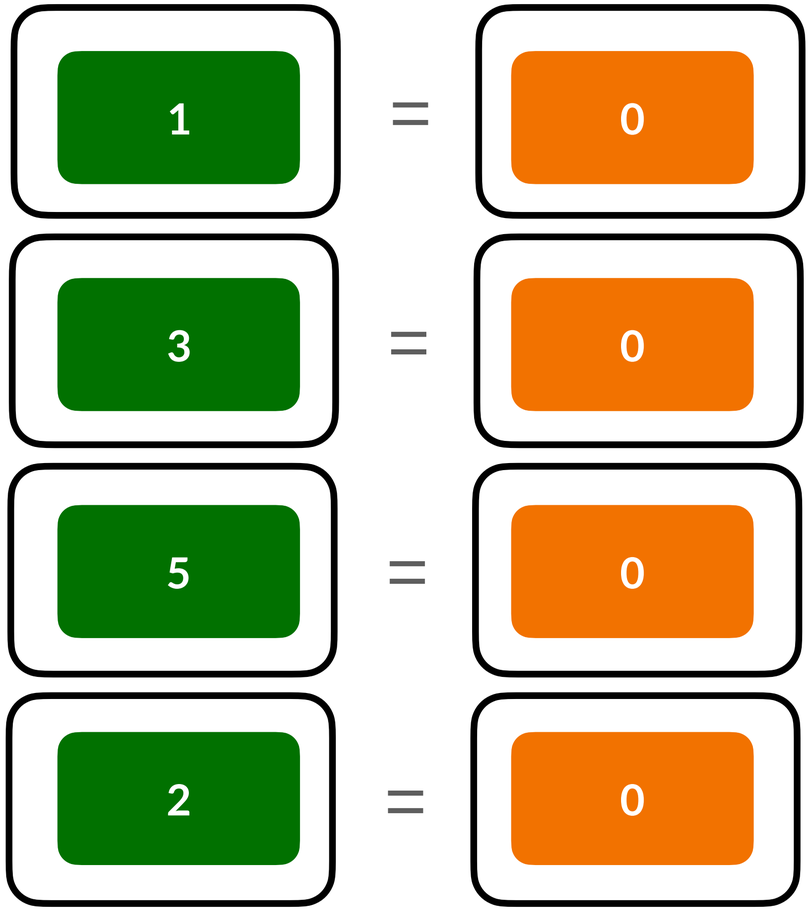
1	+	4	=	0
3	+	6	=	0
5	+	9	=	0
2	+	3	=	0

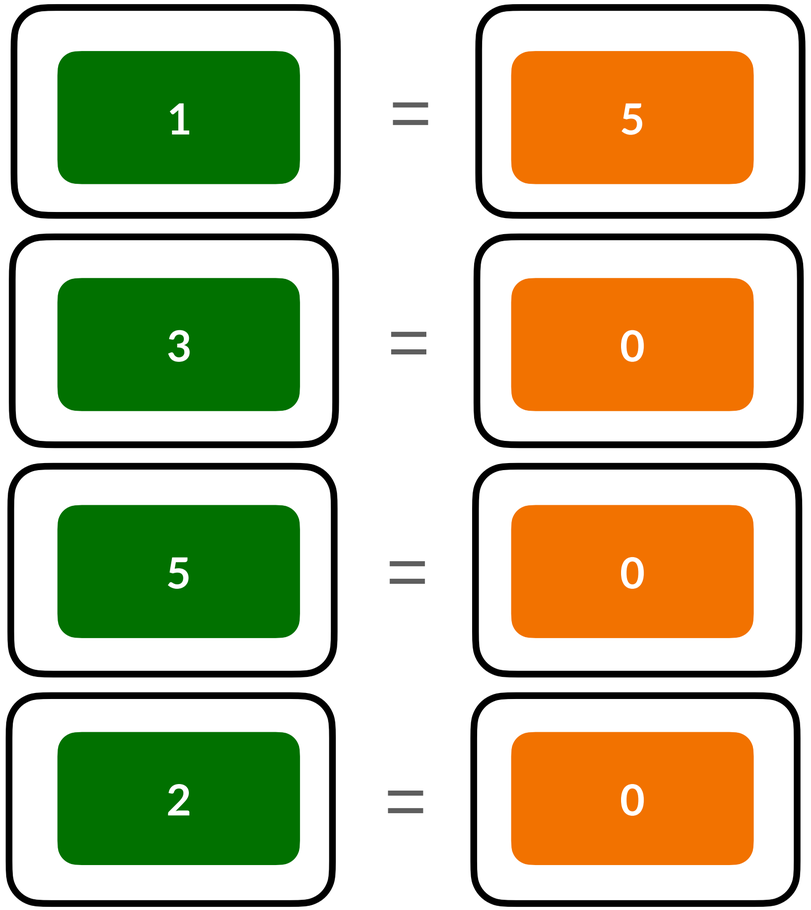
1	+	4	=	5
3	+	6	=	9
5	+	9	=	14
2	+	3	=	5

Cross Lane Operations

- Compute or modify data from different lanes of an entire vector.
- Sorting the components of a vector is an example of a cross-lane operation.
- Cross-lane operations can produce scalars or vectors of different shapes from the source vectors.
- Cross-lane operations can be further classified into permutation and reduction operations

https://openjdk.org/jeps/448


1
=
0
3
=
0
5
=
0
2
=
0


1
=
5
3
=
0
5
=
0
2
=
0

Cross Lane Permutation

- To support cross-lane permutation operations, some vector operations accept shuffles represented by the public abstract class `VectorShuffle<E>`.
- Each element in a shuffle is an `int` value corresponding to a lane index.
- A shuffle is a mapping of lane indexes, describing the movement of lane elements from a given vector to a result vector.
- Similar to vectors and masks, instances of `VectorShuffle<E>` are instances of non-public concrete subclasses defined for each element type and length combination.
- The instance of `VectorShuffle<E>` used in an operation should have the same type and length as the vector instances involved in the operation.

https://openjdk.org/jeps/448

Demo: Vector Shuffling

- Let’s perform some vector cross-lane operations
  - `CompareVector`
  - `BroadcastVector`
  - `ReduceVector`
  - `ReduceVectorBlend`
  - `ShuffleVector`

Demo: Let’s Make a Machine Learning Model!

- Let’s take a look at rudimentary Machine Learning Model!

Multicore Processing

- Each core (Firestorm and Icestorm) has 128-bit wide SIMD registers:
  - **Firestorm:** (~ 3.2 GHz) Fast, intensive processing
  - **Icestorm:** (~2.0 GHz) Background Tasks, idle loops
  - SIMD registers: 32 registers (V0–V31), each 128 bits wide ***in each core, and not shared***
  - These are used by NEON, the SIMD unit on ARM chips.

Multicore Processing

- I have 32 registers, and 128 bits ***in each core***
- I have 10 cores, so can I divide my work and give it to each core?
- I think so.... (I will admit, this is still fuzzy to me at the moment)
- Some strategies can include, `ForkJoin` and `RecursiveAction` to divide up the payload onto multiple cores

Demo: Let’s Parallelize across CPUs

- Let’s take some very large values and divide and conquer across multiple CPUs

Thank You

- Email: dhinojosa@evolutionnext.com
- Github: https://www.github.com/dhinojosa
- Linked In: http://www.linkedin.com/in/dhevolutionnext