Vector Intermediate Language (VIL)

Gas-Efficient Vector VM as a Smart-Contract in Rust / Stylus (WASM)

Sonia Kolasinska, IndexMaker Labs

December 2, 2025

1 Introduction: The VM^2 Architecture

The Vector Intermediate Language (VIL) Virtual Machine (VM) represents a modern approach to deploying complex mathematical and financial logic onto the Arbitrum ecosystem using Rust / Stylus.

This is *VM*^{2} architecture — Virtual Machine (VIL) running as a smart contract within Virtual Machine (WASM, hosted by Arbitrum node).

1.1 Project Location

The complete project can be found at: `github.com/IndexMaker/deindex`

1.2 Motivation: Gas Efficiency through On-Chain Vector Operations

The primary innovation of VIL is the substantial gas savings realized by relocating complex vector algebra directly into the smart contract's state and execution context.

In traditional EVM smart contract systems, performing vector operations often requires:

1. Loading raw data from storage (`SLOAD`).
2. Storing raw data into storage (`SSTORE`).
3. (De)Serializing structured data (e.g. `Vec<u128>`) between contract calls.
4. Passing large data arrays as call data, incurring significant calldata gas costs.
5. Executing memory-intensive loops in the EVM, which is highly gas-inefficient for array processing.

VIL eliminates these bottlenecks:

- **Zero Serialization Overhead:** Vector data, stored as a pure binary blob (`Vec<Amount>`; where `Amount` is `u128` wrapper type), is loaded directly into the VIL execution environment within the same smart contract. All vector operations (addition, subtraction, multiplication, etc.) happen purely within the VIL execution context, **avoiding expensive EVM-based serialization and ABI encoding/decoding between logic steps**.
- **Optimal Data Flow:** Data is loaded **once** from a single storage slot (`SLOAD`), processed entirely, and stored back once (`SSTORE`), maximizing efficiency within the WASM environment.
- **WASM Optimization:** By executing the core logic via a simple, specialized instruction set (VIL), the underlying WASM runtime can execute vector mathematics far more efficiently than general-purpose EVM bytecode.

1.3 Design Philosophy: The Mathematical Co-processor Model

The architecture of the VIL VM is designed under the principle of hardware specialization:

> *„The VIL VM is to the Stylus/WASM smart contract what the x87 FPU co-processor was to the x86 CPU.„*

The VIL's stack-based architecture is prioritizing computational efficiency over general-purpose flexibility. This approach is rooted in the history of specialized math processing and draws explicit architectural inspiration from two sources:

1. The *Intel x87 Floating Point Unit (FPU) co-processor*, which accelerated complex arithmetic using a dedicated register stack to minimize reliance on general-purpose memory access.
2. The stack-based data flow popularized by *Hewlett-Packard (HP) calculators* utilizing *Reverse Polish Notation (RPN)*, which demonstrated the extreme efficiency of a stack-centric model for complex, multi-step calculation.

The VIL VM is architected to function as a specialized *mathematical co-processor* within the Stylus/WASM environment. This design leverages a highly optimized internal stack for vector, scalar, and label manipulation, enabling gas-efficient execution of complex vector algebra (e.g., `ADD`, `MUL` and `JADD`) without incurring the memory and gas overhead associated with deep memory reads/writes inherent in EVM and general WASM array processing.

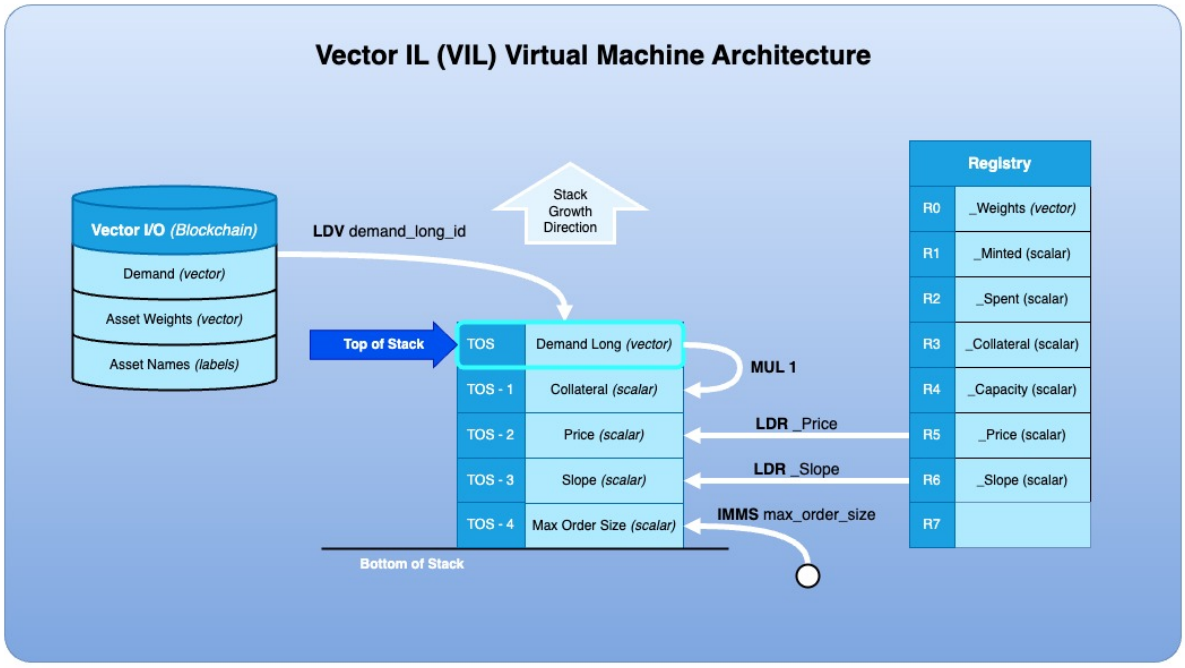

Figure 1: Conceptual VIL VM Stack and Data Flow.

2 Development Workflow with the `devil!()` Proc Macro

Writing raw instruction bytes and operands for any VM is tedious and error-prone. The `devil!()` procedural macro simplifies VIL program creation by allowing developers to embed native VIL assembly code directly within Rust functions.

2.1 Macro Functionality

The `devil!()` macro transpiles the human-readable VIL assembly language into an efficient, executable bytecode array (`Vec<u8>`). This allows for:

- **Readability:** Programs are easy to audit and maintain.
- **Type Safety:** The underlying Rust environment ensures correct operand types (e.g., `u128` storage IDs, registry names) are used.
- **Concise Syntax:** Eliminates the need for manual byte array construction.

2.2 Usage Example

The following snippet demonstrates how a complex sequence of data retrieval and vector operations is expressed concisely using the macro:

```
use devil_macros::devil;

pub fn execute_buy_order(...) -> Vec<u8> {
    devil! {
        LDV         asset_weights_id                // Load from Blockchain (SLOAD)
        STR         _Weights                        // Store into registry
        // ...

        LDR         _CappedIndexQuantity            // Load from registry
        LDM         _Weights                        // Move from registry
        MUL         1                               // Multiply
        // ...

        LDM         _AssetQuantities                // Move from registry
        STV         executed_asset_quantities_id    // Store into Blockchain (SSTORE)
    }
}
```

3 VIL Instruction Set Reference

The VIL instruction set is designed around the concept of a stack machine operating on two primary data types: $Scalar$ (`Amount`) and $Vector$ (`Vec<Amount>`), and additionally $Labels$ (`Vec<u128>`), which enable $Join$ operations (e.g. `JADD`, `JUPD`, `JFLT`). The instructions manipulate the stack and the internal registry (`R0`...`Rn`).

Instruction Set Conventions and Notation

The following conventions are used throughout the instruction set documentation to provide a clear, standardized representation of VIL stack operations, register access, and state assignments, reflecting the architecture's stack-based design inspired by the x87 FPU.

`[TOS]` Denotes the operand currently residing on the **Top of the Stack**.

`[TOS - N]` Denotes the operand located N positions below the top of the stack.
For instance, `[TOS - 1]` refers to the operand immediately below the top element.

`_Name` In usage examples, names beginning with an underscore (e.g., `_AssetWeights`) refer to dedicated VIL registers. These names are automatically associated with register indices by the `devil!()` procedural macro.

$A \leftarrow B$ A left arrow in mathematical notation indicates an assignment operation.
The value or result on the right-hand side is assigned to the destination on the left-hand side. For example, $[TOS] \leftarrow [TOS]+[TOS-1]$ means the sum of the top two stack operands is stored back onto the top of the stack.

Instruction Set Quick Reference

Quick listing of instructions.

Table 1: VIL — Set Summary

Mnemonic	Brief Description
LDL	Load Labels from Storage
LDV	Load Vector from Storage
LDD	Load Duplicate from Stack
LDR	Load Duplicate from Registry
LDM	Load Moving from Register
STL	Store Labels into Storage
STV	Store Vector into Storage
STR	Store Moving into Registry
PKV	Pack Scalar values into Vector
PKL	Pack Label values into Labels object
UNPK	Unpack values from Vector / Labels
VPUSH	Push Scalar onto Vector
VPOP	Pop Scalar from Vector
T	Transpose Vectors on Stack
LUNION	Union of Lables
LPUSH	Push Label onto Labels
LPOP	Pop Label from Labels
JUPD	Update (Replace) by Joining on Labels
JADD	Add by Joining on Labels
JFLT	Filter by Joining on Labels
ADD	Addition
SUB	Substraction
SSB	Saturating Subraction
MUL	Multiplication
DIV	Division
SQRT	Square Root
MIN	Minimum
MAX	Maximum
VSUM	Sum of Vector components
VMIN	Minimum component of a Vector
VMAX	Maximum component of a Vector
IMMS	Push Scalar on Stack
IMML	Push Label on Stack
ZEROS	Zero Vector
ONES	Unit Vector
POPN	Pop N levels from Stack
SWAP	Swap levels on Stack
B	Branch into stored procedure
FOLD	Apply stored procedure to all components

3.1 Data Loading & Stack Access (Opcode 10..15)

3.1.1 `LDL` — Load Labels from Storage

⇑ Back to top

Description

Load (`Labels`) object by ID from *Vector I/O (VIO)*.

This is relatively gas-expensive operation as it will load binary blob from *Blockchain* using underlying `(SLOAD)` operatation.

Operands

- `labels_id: u128` – An ID of the $Labels$ object *(vector storage slot)*.

Stack Args

- *(no stack args)*.

Return

- `[TOS]: Labels` – $Labels$ object loaded from vector storage.

Usage Example

```
LDL     asset_names_id
```

Load $Labels$ with 'asset_names_id' ID from vector storage.

3.1.2 `LDV` — Load Vector from Storage

⇑ Back to top

Description

Load (`Vector`) object by ID from *Vector I/O (VIO)*.

This is relatively gas-expensive operation as it will load binary blob from *Blockchain* using underlying `(SLOAD)` operatation.

Operands

- `labels_id: u128` – An ID of the $Vector$ object *(vector storage slot)*.

Stack Args

- *(no stack args)*.

Return

- `[TOS]: Vector` – $Vector$ object loaded from vector storage.

Usage Example

```
LDV     asset_weights_id
```

Load $Vector$ with 'asset_weights_id' ID from vector storage.

3.1.3 `LDD` — Load Duplicate from Stack

⇑ Back to top

Description

Load duplicate (copy) of stack operand at `T - pos`.

This is mildly gas-expensive operation, much less gas-expensive than load from vector storage, but still it requires cloning of a vector.

Operands

- `pos: u8` – Stack position relative to `TOS` from where we want to duplicate an operand.

Stack Args

- `[TOS - pos]` – Source operand to be cloned.

Return

- `[TOS]` – Duplicate of the operand at `TOS - pos`.

Usage Example

```
LDD     3
```

Create a duplicate of the operand three levels deep on the stack.

3.1.4 `LDR` — Load Duplicate from Registry

⇑ Back to top

Description

Load duplicate (copy) from registry `reg`.

This is mildly gas-expensive operation, much less gas-expensive than load from vector storage, but still it requires cloning of a vector.

Operands

- `reg: u8` – Register number `0..num_registry`.

Stack Args

- *(no stack args)*.

Return

- `[TOS]` – Duplicate of the registry `reg`.

Usage Example

```
    LDR     _AssetWeights
```

Create a duplicate of the register allocated for `_AssetWeights`.

The `_AssetWeights` is a convenience label we associate with some register when we use `devil!()` macro.

3.1.5 `LDM` — Load Moving from Register

⇑ Back to top

Description

Load value moving it out of registry `reg`.

This is highly gas-effective operation, as operand is moved from register onto stack without copying.

Operands

- `reg: u8` – Register number `0..num_registry`.

Stack Args

- *(no stack args)*.

Return

- `[TOS]` – Value moved out of the registry `reg`.

Usage Example

```
LDM     _MarketAssetPrices
```

Move `_MarketAssetPrices` value out of register, and place it on stack.

The `_MarketAssetPrices` is a convenience label we associate with some register when we use `devil!()` macro.

3.2 Data Storage & Register Access (Opcode 20..23)

3.2.1 `STL` — Store Labels into Storage

⇑ Back to top

Description

Store (`Labels`) object by ID into *Vector I/O (VIO)*.

This is relatively gas-expensive operation as it will store binary blob into *Blockchain* using underlying `(SSTORE)` operatation.

Operands

- `labels_id: u128` – An ID of the $Labels$ object *(vector storage slot)*.

Stack Args

- `[TOS]` – Operand on stack to be moved into vector storage `labels_id`.

Return

- *(stack shifted by one level)*.

Usage Example

```
STL     market_asset_names_id
```

Pop value from stack `[TOS]` and store it into vector storage with 'market_asset_names' ID.

3.2.2 `STV` — Store Vector into Storage

⇑ Back to top

Description

Store (`Vector`) object by ID into *Vector I/O (VIO)*.

This is relatively gas-expensive operation as it will store binary blob into *Blockchain* using underlying `(SSTORE)` operatation.

Operands

- `vector_id: u128` – An ID of the $Vector$ object *(vector storage slot)*.

Stack Args

- `[TOS]` – Operand on stack to be moved into vector storage `vector_id`.

Return

- *(stack shifted by one level)*.

Usage Example

```
STV     market_asset_prices
```

Pop value from stack `[TOS]` and store it into vector storage with 'market_asset_prices' ID.

3.2.3 `STR` — Store Moving into Registry

⇑ Back to top

Description

Move value from stack and store it into registry `reg`.

This is highly gas-effective operation, as operand is moved from stack into register without copying.

Operands

- `reg: u8` – Register number `0..num_registry`.

Stack Args

- `[TOS]` – Operand on stack to be moved into registry `reg`.

Return

- *(stack shifted by one level)*.

Usage Example

```
STR     _AssetWeights
```

Pop value from stack `[TOS]` and store it in register allocated for `_AssetWeights`.

3.3 Data Structure Manipulation (Opcode 30..35)

3.3.1 `PKV` — Pack Scalar values into Vector

⇑ Back to top

Description

Pack `count` $Scalar$ type values from stack into a new $Vector$.

Operands

- `count: u8` – Number of stack levels starting from `[TOS]` that we want to pack into $Vector$.

Stack Args

- `[TOS]: Scalar` – Operand on stack to be packed as last element of resulting $Vector$.

...

- `[TOS - count]: Scalar` – Operand on stack to be packed as first element of resulting $Vector$.

Return

- `[TOS]` – $Vector$ contructed from stack operands `[TOS - count], ..., [TOS]`
- *(stack shifted by 'count' levels)*.

Usage Example

```
LDR     _Capacity
LDR     _Price
LDR     _Slope
PKV     3
```

Load three values from registry: `_Capacity`, `_Price`, `_Slope`, and then pack them into single $Vector$: `[_Capacity, _Price, _Slope]`.

3.3.2 `PKL` — Pack Label values into Labels object

⇑ Back to top

Description

Pack `count` $Label$ type values from stack into a new $Labels$.

Operands

- `count: u8` – Number of stack levels starting from `[TOS]` that we want to pack into $Labels$.

Stack Args

- `[TOS]: Label` – Operand on stack to be packed as last element of resulting $Labels$.

...

- `[TOS - count]: Label` – Operand on stack to be packed as first element of resulting $Labels$.

Return

- `[TOS]` – $Labels$ contructed from stack operands `[TOS - count], ..., [TOS]`
- *(stack shifted by 'count' levels)*.

Usage Example

```
IMML    btc_label
IMML    eth_label
IMML    xrp_label
PKL     3
```

Load three $Label$ constants: `btc_label`, `eth_label`, `xrp_label`, and then pack them into single $Labels$: `[btc_label, eth_label, xrp_label]`.

3.3.3 `UNPK` — Unpack values from Vector / Labels

⇑ Back to top

Description

Unpack a $Vector$ / $Labels$ object onto the stack.

Operands

- *(no operands)*

Stack Args

- `[TOS]: Vector|Labels` – An object to be unpacked onto stack $Vector$ / $Labels$: `[[0], ..., [count - 1]]`.

Return

- `[TOS]: Scalar|Label` – Component unpacked as last from $Vector$ / $Labels$.

...

- `[TOS - count]: Scalar|Label` – Component unpacked as first from $Vector$ / $Labels$.
- *(stack shifted by one level)*.

Usage Example

```
LDV     index_quote_id
UNPK
```

Load a $Vector$ from vector storage with 'index_quote_id' ID, and unpack its components onto stack. Given that 'index_quote_id' points to a vector, the component of which are: `[capacity, price, slope]`, the resulting stack will have structure: `[(previous stack state)..., capacity, price, slope]`, where the value of 'slope' will be on `[TOS]`.

3.3.4 `VPUSH` — Push Scalar onto Vector

⇑ Back to top

Description

Pushes a scalar onto the $Vector$.

This is gas-efficient in-place operation, as it updates a $Vector$ on `[TOS]`, appending new component at the end.

Operands

- `scalar: Scalar` – Constant $Scalar$ value to be pushed onto $Vector$ at `[TOS]`.

Stack Args

- `[TOS]: Vector[0..len]` – A $Vector$ of length 'len'.

Return

- `[TOS]: Vector[0..len + 1]` – Resulting $Vector$ of length $len + 1$.

Usage Example

```
LDV     weights_id
VPUSH   10.0
```

Load from storage a vector with 'weights_id' ID, and push scalar constant 10.0 to the end of it.

Note This does not modify the vector in-storage, and only its on-stack copy is modified. To store this modified vector back into storage separate `STV` instruction is required.

3.3.5 `VPOP` — Pop Scalar from Vector

⇑ Back to top

Description

Pop a scalar from the $Vector$.

This is gas-efficient in-place operation, as it updates a $Vector$ on `[TOS]`, removing its last component.

Operands

- *(no operands)*

Stack Args

- `[TOS]: Vector[0..len]` – A $Vector$ of length 'len'.

Return

- `[TOS]: Vector[0..len - 1]` – Resulting $Vector$ of length $len - 1$.

Usage Example

```
LDV     weights_id
VPOP
```

Load from storage a vector with 'weights_id' ID, and pop last element from the end of it.

Note This does not modify the vector in-storage, and only its on-stack copy is modified. To store this modified vector back into storage separate `STV` instruction is required.

3.3.6 `T` — Transpose Vectors on Stack

⇑ Back to top

Description

Perform matrix transposition of M vectors on the stack, each of length N.

$$\begin{pmatrix} a_{1,1} & a1,2 & \cdots & a_{1,N} \\ a_{2,1} & a2,2 & \cdots & a_{2,N} \\ \vdots & \vdots & \ddots & \vdots \\ a_{M,1} & a_{M,2} & \cdots & a_{M,N} \end{pmatrix}^{T} \Rightarrow \begin{pmatrix} a_{1,1} & a2,1 & \cdots & a_{M,1} \\ a_{1,2} & a2,2 & \cdots & a_{M,2} \\ \vdots & \vdots & \ddots & \vdots \\ a_{1,N} & a_{2,N} & \cdots & a_{M,N} \end{pmatrix}$$

This is gas-efficient in-place operation, which updates M vectors on the stack.

Operands

- `count: u8` – Number of vectors on the stack to transpose together.

Stack Args

- `[TOS]: Vector` – A $Vector$ of length 'len'.

...

- `[TOS - count]: Vector` – A $Vector$ of length 'len'.

Return

- `[TOS]: Vector` – A $Vector$ of length 'count'.

...

- `[TOS - len]: Vector` – A $Vector$ of length 'count'.

Usage Example

```
LDV     index_a_quote_id
LDV     index_b_quote_id
LDV     index_c_quote_id
T       3
```

Load from vector storage three vectors with IDs:

```
index_a_quote_id, index_b_quote_id, index_c_quote_id,
```

and then transpose them into vectors:

```
    [index_a_capacity, index_b_capacity, index_c_capacity],
    [index_a_price, index_b_price, index_c_price],
    [index_a_slope, index_b_slope, index_c_slope],
```

or simply:

```
capacities, prices, slopes.
```

3.4 Labels Manipulation (Opcode 40..46)

3.4.1 `LUNION` — Union of Lables

⇑ Back to top

Description

Union of two operands of $Labels$ type.

This is gas-efficient in-place operation of complexity $O(N + M)$.

Operands

- `pos: u8`: Position on stack of the other $Labels$ object relative to `[TOS]`.

Stack Args

- `[TOS]: Labels` – The $Labels$ object to be extended.
- `[TOS - pos]: Labels` – The $Labels$ objects containing additional labels.

Return

- `[TOS]` – Resulting $Labels$ object containig original and additional labels.

Usage Example

```
LDL         asset_weights
LDL         market_asset_weights
LUNION      1
```

Load two objects of type $Labels$ from storage with IDs: `asset_weights` and `market_asset_weights`, and then modify last labels on `[TOS]` *`market_asset_weights`* by adding labels from `[TOS - 1]` stack level *`asset_weights`*. Resulting labels will contain all original labels from `market_asset_weights` and new labels from `asset_weights`.

3.4.2 `LPUSH` — Push Label onto Labels

⇑ Back to top

Description

Pushes a $Label$ onto the $Labels$.

This is gas-efficient in-place operation, as it updates a $Labels$ on `[TOS]`, appending new component at the end.

Operands

- `label: Label` – Constant $Label$ to be pushed onto $Labels$ at `[TOS]`.

Stack Args

- `[TOS]: Labels[0..len]` – A $Labels$ of length 'len'.

Return

- `[TOS]: Labels[0..len + 1]` – Resulting $Labels$ of length $len + 1$.

Usage Example

```
LDL     asset_names_id
LPUSH   eth_label
```

Load from storage the labels with 'asset_names_id' ID, and push label constant 'eth_label' to the end of it.

Note This does not modify the labels in-storage, and only its on-stack copy is modified. To store this modified labels back into storage separate `STL` instruction is required.

3.4.3 `LPOP` — Pop Label from Labels

⇑ Back to top

Description

Pop a label from the $Labels$.

This is gas-efficient in-place operation, as it updates a $Labels$ on `[TOS]`, removing its last component.

Operands

- *(no operands)*

Stack Args

- `[TOS]: Labels[0..len]` – A $Labels$ of length 'len'.

Return

- `[TOS]: Labels[0..len - 1]` – Resulting $Labels$ of length $len - 1$.

Usage Example

```
LDL     asset_names_id
VPOP
```

Load from storage the labels with 'asset_names_id' ID, and pop a label from the end of it.

Note This does not modify the labels in-storage, and only its on-stack copy is modified. To store this modified labels back into storage separate `STL` instruction is required.

3.4.4 `JUPD` — Update (Replace) by Joining on Labels

⇑ Back to top

Description

Update $Vector$ on top of the stack (`[TOS]`) by replacing its selected components with values from another vector on stack at `[TOS - pos_B]` joining on $Labels$.

This is an efficient operation of complexity $O(N)$ *(for contiguous overlaps)* or $O(N \times \log M)$ *(for overlaps with gaps)*.

Operands

- `pos_B: u8` – Stack position of $VectorB$ relative to `[TOS]`.
- `lab_A: u8` – Stack position of $LabelsA$ relative to `[TOS]`.
- `lab_B: u8` – Stack position of $LabelsB$ relative to `[TOS]`.

Stack Args

- `[TOS: Vector]` – Vector A
- `[TOS - pos_B: Vector]` – Vector B
- `[TOS - lab_A: Labels]` – Labels A
- `[TOS - lab_B: Labels]` – Labels B

Return

- `[TOS: Vector]`: – $VectorA$ updated by replacing values from $VectorB$ mapping them through projection of $LabelsB$ onto $LabelsA$.

Usage Example

```
JUPD    2   4   5
```

Update vector on stack at `[TOS]` by replacing its selected components with components of a vector at `[TOS - 2]`, using labels at `[TOS - 4]` for vector at `[TOS]`, and using labels at `[TOS - 5]` for vector at `[TOS - 2]`, where labels overlap.

3.4.5 `JADD` — Add by Joining on Labels

⇑ Back to top

Description

Add two $Vectors$ joining on $Labels$. Expands vector at `[TOS - pos_B]` using labels at `[TOS - lab_B]` to match labels of vector at `TOS` located on stack at `[TOS - lab_A]`.

This is an efficient operation of complexity $O(N)$ *(for contiguous overlaps)* or $O(N \times \log M)$ *(for overlaps with gaps)*.

Operands

- `pos_B: u8` – Stack position of $VectorB$ relative to `[TOS]`.
- `lab_A: u8` – Stack position of $LabelsA$ relative to `[TOS]`.
- `lab_B: u8` – Stack position of $LabelsB$ relative to `[TOS]`.

Stack Args

- `[TOS: Vector]` – $VectorA$
- `[TOS - pos_B: Vector]` – $VectorB$
- `[TOS - lab_A: Labels]` – $LabelsA$
- `[TOS - lab_B: Labels]` – $LabelsB$

Return

- `[TOS: Vector]`: – $VectorA$ updated by adding values from $VectorB$ mapping them through projection of $LabelsB$ onto $LabelsA$.

Usage Example

```
JADD    2   4   5
```

Add to vector on stack at `[TOS]` selected components of a vector at `[TOS - 2]`, using labels at `[TOS - 4]` for vector at `[TOS]`, and using labels at `[TOS - 5]` for vector at `[TOS - 2]`, where labels overlap.

3.4.6 `JFLT` — Filter by Joining on Labels

⇑ Back to top

Description

Filter vector on stack at `[TOS]` retaining only components corresponding to labels at `[TOS - lab_B]` using labels at `[TOS - lab_A]` for vector at `[TOS]`, and where labels overlap.

This is an efficient operation of complexity $O(N)$ *(for contiguous overlaps)* or $O(N \times \log M)$ *(for overlaps with gaps)*.

Operands

- `lab_A: u8` – Stack position of $LabelsA$ relative to `[TOS]`.
- `lab_B: u8` – Stack position of $LabelsB$ relative to `[TOS]`.

Stack Args

- `[TOS: Vector]` – $VectorA$
- `[TOS - lab_A: Labels]` – $LabelsA$
- `[TOS - lab_B: Labels]` – $LabelsB$

Return

- `[TOS: Vector]`: – $VectorA$ filtered by retaining components through projection of $LabelsB$ onto $LabelsA$.

Usage Example

```
JFLT    4   5
```

Filter vector on stack at `[TOS]` retaining only components corresponding to labels at `[TOS - 5]` using labels at `[TOS - 4]` for vector at `[TOS]`, and where labels overlap.

3.5 Arithmetic & Core Math (Opcode 50..55)

Description

3.5.1 `ADD` — Addition

⇑ Back to top

Add operand at `[TOS]` to operand at `[TOS - pos]`.

$$[TOS] \leftarrow [TOS] + [TOS - pos]$$

- Works with vectors and scalars.
- In-place updates operand on `[TOS]`.
- Does not consume the other operand.

Operands

- `pos: u8` – Stack position relative to `[TOS]`.

Stack Args

- `[TOS: Vector|Scalar]` – Operand to be updated in-place, with addition.
- `[TOS - pos: Vector|Scalar]` – Second operand of addition.

Return

- `[TOS]` – Updated first operand.

Usage Example

```
IMMS    3.0
IMMS    5.0
ADD     1
```

Add 3.0 to 5.0, and replace 5.0 with result of the addition (8.0).

3.5.2 `SUB` — Substraction

⇑ Back to top

Description

Subtract from operand at `[TOS]` an operand at `[TOS - pos]`.

- Works with vectors and scalars.
- In-place updates operand on `[TOS]`.
- Does not consume the other operand.

Operands

- `pos: u8` – Stack position relative to `[TOS]`.

Stack Args

- `[TOS: Vector|Scalar]` – Operand to be updated in-place, with subtraction.
- `[TOS - pos: Vector|Scalar]` – Second operand of subtraction.

Return

- `[TOS]` – Updated first operand.

Usage Example

```
IMMS    3.0
IMMS    5.0
SUB     1
```

Subtract 3.0 from 5.0, and replace 5.0 with result of the subtraction (2.0).

3.5.3 `SSB` — Saturating Subraction

⇑ Back to top

Description

Subtract from operand at `[TOS]` an operand at `[TOS - pos]` saturating to 0.

$$[TOS] \leftarrow \mathsf{max}([TOS] - [TOS - pos], 0)$$

- Works with vectors and scalars.
- In-place updates operand on `[TOS]`.
- Does not consume the other operand.

Operands

- `pos: u8` – Stack position relative to `[TOS]`.

Stack Args

- `[TOS: Vector|Scalar]` – Operand to be updated in-place, with saturating subtraction.
- `[TOS - pos: Vector|Scalar]` – Second operand of saturating subtraction.

Return

- `[TOS]` – Updated first operand.

Usage Example

```
IMMS    5.0
IMMS    3.0
SSB     1
```

Subtract 5.0 from 3.0 saturating at 0.0, and replace 3.0 with result of the saturating subtraction (0.0).

3.5.4 `MUL` — Multiplication

⇑ Back to top

Description

Multiply operand at `[TOS]` by operand at `[TOS - pos]`.

$$[TOS] \leftarrow [TOS] \odot [TOS - pos]$$

- Works with vectors and scalars.
- Multiplication of two vectors is performed component-wise.
- Multiplication of vector by scalar is possible, but not scalar by vector as it is an in-place operation.
- In-place updates operand on `[TOS]`.
- Does not consume the other operand.

Operands

- `pos: u8` – Stack position relative to `[TOS]`.

Stack Args

- `[TOS: Vector|Scalar]` – Operand to be updated in-place, with multiplication.
- `[TOS - pos: Vector|Scalar]` – Second operand of multiplication.

Return

- `[TOS]` – Updated first operand.

Usage Example

```
IMMS    3.0
IMMS    5.0
MUL     1
```

Multiply 3.0 by 5.0, and replace 5.0 with result of the multiplication (15.0).

3.5.5 `DIV` — Division

⇑ Back to top

Description

Divide operand at `[TOS]` by operand at `[TOS - pos]`.

$$[TOS] \leftarrow \frac{[TOS]}{[TOS - pos]}$$

- Works with vectors and scalars.
- Division of two vectors is performed component-wise.
- Division of vector by scalar is possible, but not scalar by vector as it is an in-place operation.
- In-place updates operand on `[TOS]`.
- Does not consume the other operand.

Operands

- `pos: u8` – Stack position relative to `[TOS]`.

Stack Args

- `[TOS: Vector|Scalar]` – Operand to be updated in-place, with division.
- `[TOS - pos: Vector|Scalar]` – Second operand of division.

Return

- `[TOS]` – Updated first operand.

Usage Example

```
IMMS    3.0
IMMS    15.0
DIV     1
```

Divide 15.0 by 3.0, and replace 15.0 with result of the multiplication (3.0).

3.5.6 `SQRT` — Square Root

⇑ Back to top

Description

Take square root of `[TOS]`.

$$[TOS] \leftarrow \sqrt{[TOS]}$$

- Works with vectors and scalars.
- Square root of two vector is performed component-wise.
- In-place updates operand on TOS.

Notice

This operation is implemented using *Babylonian method* of $O(\log k)$ complexity where k is number of bits. Since it is fixed point 128-bit number, this is constant $O(1)$.

Operands

- (no operands)

Stack Args

- `[TOS: Vector|Scalar]` – Operand of square root.

Return

- `[TOS]` – Result of square root.

Usage Example

```
IMMS    4.0
SQRT
```

Take square root of 4.0, and replace 4.0 with the result (2.0).

3.6 Logic & Comparison (Opcode 60..61)

3.6.1 `MIN` — Minimum

⇑ Back to top

Description

Take minimum between operand at `[TOS]` and operand at `[TOS - pos]`.

$$[TOS] \leftarrow \min([TOS], [TOS - pos])$$

- Works with vectors and scalars.
- Minimum of two vectors is performed component-wise.
- Minimum of vector by scalar is possible, but not scalar by vector as it is an in-place operation.
- In-place updates operand on `[TOS]`.
- Does not consume the other operand.

Operands

- `pos: u8` – Stack position relative to `[TOS]`.

Stack Args

- `[TOS: Vector|Scalar]` – Operand to be updated in-place, with minimum.
- `[TOS - pos: Vector|Scalar]` – Second operand of minimum.

Return

- `[TOS]` – Updated first operand.

Usage Example

```
IMMS    3.0
IMMS    5.0
MIN     1
```

Take minimum of 5.0 and 3.0, and replace 5.0 with result of the minimum (3.0).

3.6.2 `MAX` — Maximum

⇑ Back to top

Description

Take maximum between operand at `[TOS]` and operand at `[TOS - pos]`.

$$[TOS] \leftarrow \max([TOS], [TOS - pos])$$

- Works with vectors and scalars.
- Maximum of two vectors is performed component-wise.
- Maximum of vector by scalar is possible, but not scalar by vector as it is an in-place operation.
- In-place updates operand on `[TOS]`.
- Does not consume the other operand.

Operands

- `pos: u8` – Stack position relative to `[TOS]`.

Stack Args

- `[TOS: Vector|Scalar]` – Operand to be updated in-place, with maximum.
- `[TOS - pos: Vector|Scalar]` – Second operand of maximum.

Return

- `[TOS]` – Updated first operand.

Usage Example

```
IMMS    3.0
IMMS    5.0
MAX     1
```

Take minimum of 5.0 and 3.0, and replace 5.0 with result of the minimum (5.0).

3.7 Vector Aggregation (Opcode 70..72)

3.7.1 `VSUM` — Sum of Vector components

⇑ Back to top

Description

Sum of all vector components.

$$[TOS] \leftarrow \sum_{i=0}^{len([TOS])} ([TOS][i])$$

- (no operands)

Stack Args

- `[TOS: Vector]` – $Vector$ of which components to sum.

Return

- `[TOS: Scalar]` – $Scalar$ sum of input vector components.

Usage Example

```
LDV     asset_weights_id
VSUM
```

Load from storage a $Vector$ with ID 'asset_weights_id', and compute sum of all components, i.e. sum of all weights, and place result on `[TOS]`.

3.7.2 `VMIN` — Minimum component of a Vector

⇑ Back to top

Description

Minimum value found within vector components.

$$[TOS] \leftarrow \min_{i=0}^{len([TOS])} ([TOS][i])$$

- (no operands)

Stack Args

- `[TOS: Vector]` – $Vector$ of which components minimum to find.

Return

- `[TOS: Scalar]` – $Scalar$ minium of input vector components.

Usage Example

```
LDV     asset_weights_id
VMIN
```

Load from storage a $Vector$ with ID ‘asset_weights_id’, find minimum weight, and place result on `[TOS]`.

3.7.3 `VMAX` — Maximum component of a Vector

⇑ Back to top

Description

Maximum value found within vector components.

$$[TOS] \leftarrow \max_{i=0}^{len([TOS])} ([TOS][i])$$

- (no operands)

Stack Args

- `[TOS: Vector]` – $Vector$ of which components maximum to find.

Return

- `[TOS: Scalar]` – $Scalar$ maximum of input vector components.

Usage Example

```
LDV     asset_weights_id
VMAX
```

Load from storage a $Vector$ with ID 'asset_weights_id', find maximum weight, and place result on `[TOS]`.

3.8 Immediate Values & Vector Creation (Opcode 80..83)

3.8.1 `IMMS` — Push Scalar on Stack

⇑ Back to top

Description

Push immediate $Scalar$ value on stack.

Operands

- `value: Scalar` – Constant $Scalar$ value to be pushed on stack.

Stack Args

- *(no stack args)*

Return

- `[TOS: Scalar]` – $Scalar$ operand pushed.

Usage Example

```
IMMS    4.0
```

Push constant value 4.0 on `[TOS]`.

3.8.2 `IMML` — Push Label on Stack

⇑ Back to top

Description

Push immediate $Label$ value on stack.

Operands

- `value: Label` – Constant $Label$ value to be pushed on stack.

Stack Args

- *(no stack args)*

Return

- `[TOS: Label]` – $Label$ operand pushed.

Usage Example

```
IMML    btc_label
```

Push constant label value ‘btc_label’ from Rust code onto stack.

3.8.3 `ZEROS` — Zero Vector

⇑ Back to top

Description

Create Vector of zeros matching length of the operand at `[TOS - pos]`.

- Works for $Vector$ and $Labels$ type of operands.
- Pushes new zero $Vector$ on `[TOS]`.
- Does not consume the operand.

Operands

- `pos`: `u8` – Stack position relative to `[TOS]`.

Stack Args

- `[TOS - pos: Vector|Labels]` – $Vector$ or $Labels$ the length of which to use.

Return

- `[TOS: Vector]` – $Vector$ of zeros with same length as input operand.

Usage Example

```
LDL     asset_names_id
ZEROS   1
```

Load from storage $Labels$ with ID 'asset_names_id', and then create zero $Vector$ of same length.

3.8.4 `ONES` — Unit Vector

⇑ Back to top

Description

Create Vector of ones matching length of the operand at `[TOS - pos]`.

- Works for $Vector$ and $Labels$ type of operands.
- Pushes new unit $Vector$ on `[TOS]`.
- Does not consume the operand.

Operands

- `pos`: `u8` – Stack position relative to `[TOS]`.

Stack Args

- `[TOS - pos: Vector|Labels]` – $Vector$ or $Labels$ the length of which to use.

Return

- `[TOS: Vector]` – Unit $Vector$ with same length as input operand.

Usage Example

```
LDL     asset_names_id
ONES    1
```

Load from storage $Labels$ with ID 'asset_names_id', and then create unit $Vector$ of same length.

3.9 Stack Control & Program Flow (Opcode 90..94)

3.9.1 `POPN` — Pop N levels from Stack

⇑ Back to top

Description

Pop $count$ values from the stack.

Operands

- `count: u8` – Number of levels to pop.

Stack Args

- `[TOS]` – First level to pop from stack.
- `[...]`
- `[TOS - count]`: Last level to pop from stack.
- *(rest of stack...)*

Return

- *(rest of stack...)*

Usage Example

```
IMMS    1.0
IMMS    2.0
IMMS    3.0
IMMS    4.0
POPN    3
```

Pop 3 levels from stack, i.e. poped elements are *2.0, 3.0, 4.0*, leaving 1.0 on stack at `[TOS]`.

3.9.2 `SWAP` — Swap levels on Stack

⇑ Back to top

Description

Swap TOS with operand at `[TOS - pos]`.

Operands

- `pos: u8` – Stack position relative to `[TOS]`.

Stack Args

- `[TOS]` – First level to swap.
- `[...]`
- `[TOS - pos]`: Second level to swap.
- *(rest of stack...)*

Return

- `[TOS]` – Operand from previously second level.
- `[...]`
- `[TOS - pos]`: Operand from previously first level.
- *(rest of stack...)*

Usage Example

```
IMMS    1.0
IMMS    2.0
IMMS    3.0
IMMS    4.0
SWAP    2
```

Swap 4.0 with 2.0, which results in stack `[1.0, 4.0, 3.0, 2.0]`, where 2.0 is at `[TOS]`.

3.9.3 `B` — Branch into stored procedure

⇑ Back to top

Description

Call sub-routine stored as $Code$ at 'prg_id', supplying 'N' inputs and taking 'M' outputs from stack.

The 'N' inputs are consumed from stack. 'M' outputs are moved from sub-routine's `[TOS]` to caller's `[TOS]`.

Operands

- `prg_id: u128` – An ID of the stored procedure $Code$ object in vector storage.
- `N: u8` – Number of arguments to move from our stack to stack of the stored procedure before its execution.
- `M: u8` – Number of returned values to move back from stack of the stored procedure after its execution.
- `R: u8` – Number of registers to allocate for stored procedure before its execution.

Stack Args

- `[TOS]` – Last argument
- `[...]`
- `[TOS - N]` – First argument

Return

- `[TOS]` – Last returned value
- `[...]`
- `[TOS - M]` – First returned value

Usage Example

```
LDR         _Slope
LDR         _Price
LDR         _Collateral
B           solve_quadratic_id  3  1  4
STR         _IndexQuantity
```

Load from registry '_Slope', '_Price', and '_Collateral', and pass them as arguments to function loaded from storage with ID 'solve_quadratic_id'. Function returns single value, which we store into registry as '_IndexQuantity'.

3.9.4 `FOLD` — Apply stored procedure to all components

⇑ Back to top

Description

Fold (iterate) over vector/label operands.

The behaviour of the operation is the same as `B` except sub-routine is called repeatedly over components of Vector at `[TOS]`.

This is gas-efficient method of stored procedure invocation multiple times for batch of items.

Operands

- `prg_id: u128` – An ID of the stored procedure $Code$ object in vector storage.
- `N: u8` – Number of arguments to move from our stack to stack of the stored procedure before its execution.
- `M: u8` – Number of returned values to move back from stack of the stored procedure after its execution.
- `R: u8` – Number of registers to allocate for stored procedure before its execution.

Stack Args

- `[TOS]: Vector|Labels` – Iterable object of type $Vector$ or $Labels$.
- `[TOS - 1]` – Last argument
- `[...]`
- `[TOS - N - 1]` – First argument

Return

- `[TOS]` – Last returned value
- `[...]`
- `[TOS - M]` – First returned value

Usage Example

```
FOLD    prg_id  N   M   R
```