Vector Intermediate Language (VIL)

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

Sonia Kolasinska

November 28, 2025

1 Introduction: The VM^2 Architecture

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

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

1.1 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.

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`).

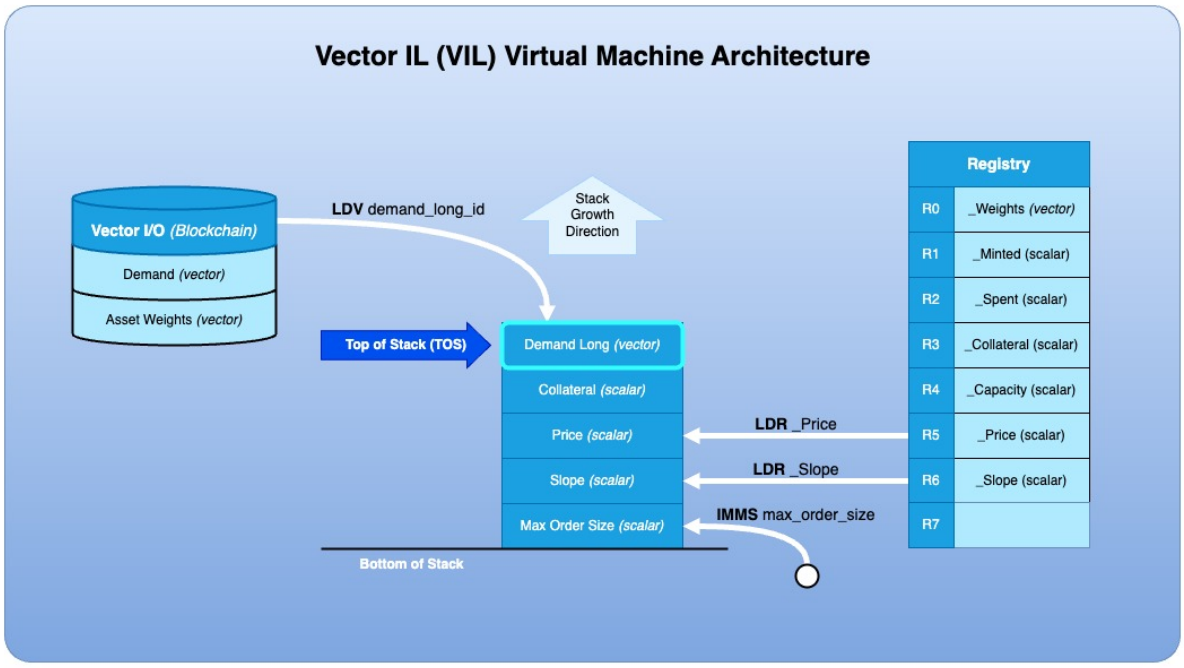

Figure 1: Conceptual VIL VM Stack and Data Flow.

Instruction Set Quick Reference

Quick listing of instructions.

Table 1: VIL — Set Summary

Mnemonic	Brief Description
`LDL`	Load Labels object from VIO by ID. Pushes on TOS..
`LDV`	Load Vector object from VIO by ID. Pushes on TOS..
`LDD`	Load Duplicate (copy) of stack operand at $[T - pos]$. Pushes on TOS..
`LDR`	Load value from Registry (R0-Rn). Pushes on TOS..
`LDM`	Load value moving it out of Registry (R0-Rn). Value is removed from registry. Pushes on TOS..
`STL`	Store Labels object into VIO. Consumes TOS..
`STV`	Store Vector object into VIO. Consumes TOS..
`STR`	Store into Registry (R0-Rn). Consumes TOS..
`PKV`	Pack 'count' values from stack into a new Vector. Consumes 'count' operands from TOS, and replaces them with Vector..

Table 1 – Continued from previous page

Mnemonic	Brief Description
PKL	Pack 'count' values from stack into a new Labels object. Consumes 'count' operands from TOS, and replaces them with Labels..
UNPK	Unpack a Vector/Labels object onto the stack. Consumes TOS, and replaces with its components..
VPUSH	Push a scalar onto the Vector (TOS). In-place updates Vector on TOS, appending new component at the end..
VPOP	Pop a scalar from the Vector (TOS). In-place updates Vector on TOS, removing last component..
T	Transpose 'count' vectors on stack $[V1, V2]$ -> $[T1, T2]$. In-place updates 'count' operands from TOS by performing transform..
LUNION	Union of two Labels operands (TOS and T-pos). Pushes on TOS..
LPUSH	Push a label value onto the Labels object (TOS). In-place updates Labels on TOS, appending new component at the end..
LPOP	Pop a label value from the Labels object (TOS). In-place updates Labels on TOS, removing last component..
JUPD	Update using Labels. Expands vector at $[TOS - pos\_B]$ using labels at $[TOS - lab\_B]$ to match labels of TOS at $[TOS - lab\_A]$. In-place updates TOS. Consumes TOS..
JADD	Add using Labels. Expands vector at $[TOS - pos\_B]$ using labels at $[TOS - lab\_B]$ to match labels of TOS at $[TOS - lab\_A]$. In-place updates TOS. Consumes TOS..
JFLT	Filter using Labels. Expands vector at $[TOS - 1]$ using labels at $[T - lab\_B]$ to match labels of TOS at $[T - lab\_A]$. In-place updates TOS. Does not consume other operands..
ADD	Add TOS by operand at $[T - pos]$. Works with vectors and scalars. In-place updates operand on TOS. Does not consume the other operand..
SUB	Subtract TOS by operand at $[T - pos]$. Works with vectors and scalars. In-place updates operand on TOS. Does not consume the other operand..
SSB	Saturating subtract TOS by operand at $[T - pos]$. Works with vectors and scalars. In-place updates operand on TOS. Does not consume the other operand..
MUL	Multiply TOS by operand at $[T - pos]$. Works with vectors and scalars. In-place updates operand on TOS. Does not consume the other operand..
DIV	Divide TOS by operand at $[T - pos]$. Works with vectors and scalars. In-place updates operand on TOS. Does not consume the other operand..
SQRT	Square root of TOS (scalar or component-wise vector). Works with vectors and scalars. In-place updates operand on TOS..
MIN	Min between TOS and operand at $[T - pos]$. Works with vectors and scalars. In-place updates operand on TOS. Does not consume the other operand..
MAX	Max between TOS and operand at $[T - pos]$. Works with vectors and scalars. In-place updates operand on TOS. Does not consume the other operand..
VSUM	Sum of all vector components. Pushes on TOS. Does not consume the operand..
VMIN	Minimum value found within vector components. Pushes on TOS. Does not consume the operand..
VMAX	Maximum value found within vector components. Pushes on TOS. Does not consume the operand..

Table 1 – Continued from previous page

Mnemonic	Brief Description
IMMS	Push immediate Scalar value on stack.
IMML	Push immediate Label value on stack.
ZEROS	Create Vector of zeros matching length of Labels at $[T - pos]$. Pushes on TOS. Does not consume the operand..
ONES	Create Vector of ones matching length of Labels at $[T - pos]$. Pushes on TOS. Does not consume the operand..
POPN	Pop 'n' values from the stack.
SWAP	Swap TOS with operand at $[T - n]$.
B	Call sub-routine stored as Labels at 'prg_id', supplying 'N' inputs and taking 'M' outputs from stack. 'N' inputs are consumed from stack. 'M' outputs are moved from sub-routine's TOS to caller's TOS..
FOLD	Fold (iterate) over vector/label operands. Same as 'B' except sub-routine is called repeatedly over components of Vector at TOS..

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

3.1.1 `LDL` — Load Labels from Storage

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

Notice

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

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

Notice

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

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

Notice

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

Load duplicate (copy) from registry `reg`.

Notice

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

Load value moving it out of registry `reg`.

Notice

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

3.2.1 `STL` — Store Labels into Storage

Store Labels object into VIO. Consumes TOS.

Operands

- `<label_id>`: `u128` Stack pos

Stack Args

- : Input

Return

- : Output

Usage Example

```
STL <label_id>
```

3.2.2 `STV` — Store Vector into Storage

Store Vector object into VIO. Consumes TOS.

Operands

- `<vector_id>`: `u128` Stack pos

Stack Args

- : Input

Return

- : Output

Usage Example

```
STV <vector_id>
```

3.2.3 `STR` — Store Moving into Registry

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

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 3. Data Structure Manipulation (Opcode 30..35)

3.3.1 `PKV` —

Pack ‘count‘ values from stack into a new Vector. Consumes ‘count‘ operands from TOS, and replaces them with Vector.

Operands

- `<count>`: `u128` Stack pos

Stack Args

- `stack args` = $[TOS - count, ..., TOS : Scalar]$: Input

Return

- `result` $[TOS]$: Output

Usage Example

```
PKV <count>
```

3.3.2 `PKL` —

Pack ‘count‘ values from stack into a new Labels object. Consumes ‘count‘ operands from TOS, and replaces them with Labels.

Operands

- `<count>`: `u128` Stack pos

Stack Args

- `stack args` = $[TOS - count, ..., TOS : Scalar]$: Input

Return

- `result` $[TOS]$: Output

Usage Example

```
PKL <count>
```

3.3.3 `UNPK` —

Unpack a Vector/Labels object onto the stack. Consumes TOS, and replaces with its components.

Operands

- (no operands)

Stack Args

- `stack args =` $[TOS : Vector|Labels]$: Input

Return

- `result` $[TOS - len, ..., TOS]$: Output

Usage Example

```
UNPK
```

3.3.4 `VPUSH` —

Push a scalar onto the Vector (TOS). In-place updates Vector on TOS, appending new component at the end.

Operands

- `<immediate`: `u128` Stack pos
- `(scalar)>`: `u128` Stack pos

Stack Args

- `stack args =` $[TOS : Vector[0..len]]$: Input

Return

- `result =` $[TOS : Vector[0..len + 1]]$: Output

Usage Example

```
VPUSH <immediate (scalar)>
```

3.3.5 `VPOP` —

Pop a scalar from the Vector (TOS). In-place updates Vector on TOS, removing last component.

Operands

- (no operands)

Stack Args

- `stack args =` $[TOS : Vector[0..len]]$: Input

Return

- `result =` $[TOS : Vector[0..len-1]]$: Output

Usage Example

```
VPOP
```

3.3.6 `T` —

Transpose 'count' vectors on stack $[V1, V2]$ -> $[T1, T2]$. In-place updates 'count' operands from TOS by performing transform.

Operands

- `<count>`: `u128` Stack pos

Stack Args

- `stack args =` $[TOS - count, ..., TOS : Vector[0..len]]$: Input

Return

- `result =` $[TOS - len, ..., TOS : Vector[0..count]]$: Output

Usage Example

```
T <count>
```

3.4 4. Labels Manipulation (Opcode 40..46)

3.4.1 `LUNION` —

Union of two Labels operands (TOS and T-pos). Pushes on TOS.

Operands

- `<pos>`: `u128` Stack pos

Stack Args

- `stack args` = $[TOS - pos, TOS]$: Input

Return

- `result` = $[TOS]$: Output

Usage Example

```
LUNION <pos>
```

3.4.2 `LPUSH` —

Push a label value onto the Labels object (TOS). In-place updates Labels on TOS, appending new component at the end.

Operands

- `<immediate`: `u128` Stack pos
- `(label)>`: `u128` Stack pos

Stack Args

- `stack args` = $[TOS : Labels[0..len]]$: Input

Return

- `result` = $[TOS : Labels[0..len + 1]]$: Output

Usage Example

```
LPUSH <immediate (label)>
```

3.4.3 `LPOP` —

Pop a label value from the Labels object (TOS). In-place updates Labels on TOS, removing last component.

Operands

- (no operands)

Stack Args

- `stack args =` $[TOS : Labels[0..len]]$: Input

Return

- `result =` $[TOS : Labels[0..len-1]]$: Output

Usage Example

```
LPOP
```

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

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 $Labels$.

Notice

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

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]`.

Notice

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

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.

Notice

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

- `result` = $[TOS :' Afilteredmapped' LBto' LA]$: Output
- `[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 5. Arithmetic & Core Math (Opcode 50..55)

3.5.1 `ADD` —

Add TOS by operand at $[T - pos]$. Works with vectors and scalars. In-place updates operand on TOS. Does not consume the other operand.

Operands

- `<pos>`: `u128` Stack pos

Stack Args

- `stack args =` $[TOS - pos, TOS : Vector|Scalar]$: Input

Return

- `result =` $[TOS]$: Output

Usage Example

```
ADD <pos>
```

3.5.2 `SUB` —

Subtract TOS by operand at $[T - pos]$. Works with vectors and scalars. In-place updates operand on TOS. Does not consume the other operand.

Operands

- `<pos>`: `u128` Stack pos

Stack Args

- `stack args =` $[TOS - pos, TOS : Vector|Scalar]$: Input

Return

- `result =` $[TOS]$: Output

Usage Example

```
SUB <pos>
```

3.5.3 `SSB` —

Saturating subtract TOS by operand at $[T - pos]$. Works with vectors and scalars. In-place updates operand on TOS. Does not consume the other operand.

Operands

- `<pos>`: `u128` Stack pos

Stack Args

- `stack args =` $[TOS - pos, TOS : Vector|Scalar]$: Input

Return

- `result =` $[TOS]$: Output

Usage Example

```
SSB <pos>
```

3.5.4 `MUL` —

Multiply TOS by operand at $[T - pos]$. Works with vectors and scalars. In-place updates operand on TOS. Does not consume the other operand.

Operands

- `<pos>`: `u128` Stack pos

Stack Args

- `stack args =` $[TOS - pos, TOS : Vector|Scalar]$: Input

Return

- `result =` $[TOS]$: Output

Usage Example

```
MUL <pos>
```

3.5.5 `DIV` —

Divide TOS by operand at $[T - pos]$. Works with vectors and scalars. In-place updates operand on TOS. Does not consume the other operand.

Operands

- `<pos>`: `u128` Stack pos

Stack Args

- `stack args` = $[TOS - pos, TOS : Vector|Scalar]$: Input

Return

- `result` = $[TOS]$: Output

Usage Example

```
DIV <pos>
```

3.5.6 `SQRT` —

Square root of TOS (scalar or component-wise vector). Works with vectors and scalars. In-place updates operand on TOS.

Operands

- (no operands)

Stack Args

- `stack args` = $[TOS : Vector|Scalar]$: Input

Return

- `result` = $[TOS]$: Output

Usage Example

```
SQRT
```

3.6 6. Logic & Comparison (Opcode 60..61)

3.6.1 `MIN` —

Min between TOS and operand at $[T - pos]$. Works with vectors and scalars. In-place updates operand on TOS. Does not consume the other operand.

Operands

- `<pos>`: `u128` Stack pos

Stack Args

- `stack args =` $[TOS - pos, TOS : Vector|Scalar]$: Input

Return

- `result =` $[TOS : Vector|Scalar]$: Output

Usage Example

```
MIN <pos>
```

3.6.2 `MAX` —

Max between TOS and operand at $[T - pos]$. Works with vectors and scalars. In-place updates operand on TOS. Does not consume the other operand.

Operands

- `<pos>`: `u128` Stack pos

Stack Args

- `stack args =` $[TOS - pos, TOS : Vector|Scalar]$: Input

Return

- `result =` $[TOS : Vector|Scalar]$: Output

Usage Example

```
MAX <pos>
```

3.7 7. Vector Aggregation (Opcode 70..72)

3.7.1 `VSUM` —

Sum of all vector components. Pushes on TOS. Does not consume the operand.

Operands

- (no operands)

Stack Args

- `stack args` = $[TOS : Vector]$: Input

Return

- `result` = $[TOS : Scalar]$: Output

Usage Example

```
VSUM
```

3.7.2 `VMIN` —

Minimum value found within vector components. Pushes on TOS. Does not consume the operand.

Operands

- (no operands)

Stack Args

- `stack args` = $[TOS : Vector]$: Input

Return

- `result` = $[TOS : Scalar]$: Output

Usage Example

```
VMIN
```

3.7.3 `VMAX` —

Maximum value found within vector components. Pushes on TOS. Does not consume the operand.

Operands

- (no operands)

Stack Args

- `stack args` = $[TOS : Vector]$: Input

Return

- `result` = $[TOS : Scalar]$: Output

Usage Example

```
VMAX
```

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

3.8.1 `IMMS` —

Push immediate Scalar value on stack

Operands

- `<immediate`: `u128` Stack pos
- `(scalar)>`: `u128` Stack pos

Stack Args

- `no stack args`: Input

Return

- `result` = $[TOS : Scalar]$: Output

Usage Example

```
IMMS <immediate (scalar)>
```

3.8.2 `IMML` —

Push immediate Label value on stack

Operands

- `<immediate`: `u128` Stack pos
- `(label)>`: `u128` Stack pos

Stack Args

- `no stack args`: Input

Return

- `result` = $[TOS : Label]$: Output

Usage Example

```
IMML <immediate (label)>
```

3.8.3 `ZEROS` —

Create Vector of zeros matching length of Labels at $[T - pos]$. Pushes on TOS. Does not consume the operand.

Operands

- `<pos>`: `u128` Stack pos

Stack Args

- `stack args` = $[TOS - pos : Vector|Labels]$: Input

Return

- `result` = $[TOS : Vector]$: Output

Usage Example

```
ZEROS <pos>
```

3.8.4 `ONES` —

Create Vector of ones matching length of Labels at $[T - pos]$. Pushes on TOS. Does not consume the operand.

Operands

- `<pos>`: `u128` Stack pos

Stack Args

- `stack args` = $[TOS - pos : Vector|Labels]$: Input

Return

- `result` = $[TOS : Vector]$: Output

Usage Example

```
ONES <pos>
```

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

3.9.1 `POPN` —

Pop 'n' values from the stack

Operands

- `<count>`: `u128` Stack pos

Stack Args

- `stack args` = $['B..., TOS - count, ..., TOS]$: Input

Return

- `result` = $['B...]$: Output

Usage Example

```
POPN <count>
```

3.9.2 `SWAP` —

Swap TOS with operand at $[T - n]$

Operands

- `<pos>`: `u128` Stack pos

Stack Args

- `stack args` = $[TOS - pos :' A, TOS :' B]$: Input

Return

- `result` = $[TOS - pos :' B, TOS :' A]$: Output

Usage Example

```
SWAP <pos>
```

3.9.3 `B` —

Call sub-routine stored as Labels at ‘prg_id‘, supplying ‘N‘ inputs and taking ‘M‘ outputs from stack. ‘N‘ inputs are consumed from stack. ‘M‘ outputs are moved from sub-routine’s TOS to caller’s TOS.

Operands

- `<prg_id>`: `u128` Stack pos
- `<N>`: `u128` Stack pos
- `<M>`: `u128` Stack pos
- `<R>`: `u128` Stack pos

Stack Args

- `stack args =` $[TOS - N]$: Input

Return

- `result =` $[TOS - M]$: Output

Usage Example

```
B <prg_id> <N> <M> <R>
```

3.9.4 `FOLD` —

Fold (iterate) over vector/label operands. Same as ‘B‘ except sub-routine is called repeatedly over components of Vector at TOS.

Operands

- `<prg_id>`: `u128` Stack pos
- `<N>`: `u128` Stack pos
- `<M>`: `u128` Stack pos
- `<R>`: `u128` Stack pos

Stack Args

- : Input

Return

- : Output

Usage Example

```
FOLD <prg_id> <N> <M> <R>
```