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 Instruction 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`).

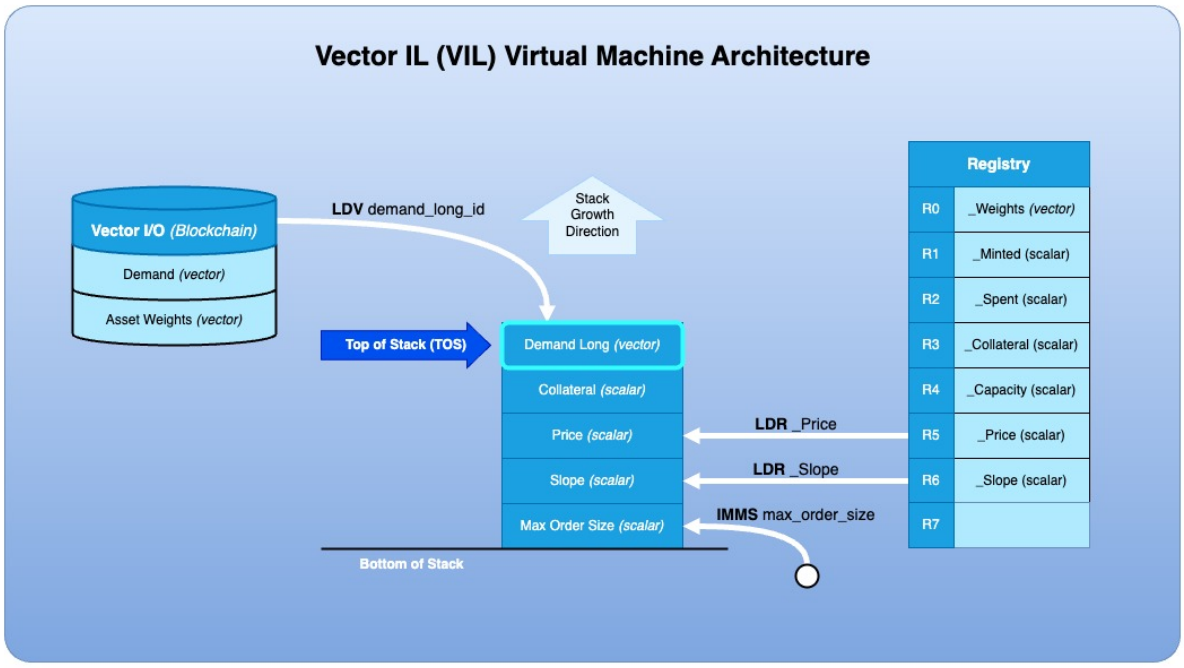

Figure 1: Conceptual VIL VM Stack and Data Flow.

Instruction Set Quick Reference

Quick listing of instructions.

Table 1: VIL Instruction Set Summary

Mnemonic	Brief Description
`LDL`	Loads a `Labels` object from Blockchain storage ID (`SLOAD`).
`LDV`	Loads a `Vector<Amount>` object from Blockchain storage ID (`SLOAD`).
`LDD`	Loads a Duplicate (copy) of a stack operand at a relative position.
`LDR`	Loads a Duplicate (copy) of a value from an internal Register.
`LDM`	Loads a value moving it from an internal Register.
`STL`	Stores the stack's TOS (Labels) to Blockchain storage (`SSTORE`).
`STV`	Stores the stack's TOS (Vector) to Blockchain storage (`SSTORE`).
`STR`	Stores the stack's TOS into a named internal Register.
`PKV`	Packs N stack items into a single Vector operand.
`PKL`	Packs N stack items into a single Labels operand.

Table 1 – Continued from previous page

Mnemonic	Brief Description
UNPK	Unpacks elements of the stack's TOS Vector/Labels
VPUSH	Pushes immediate value to the end of the Vector on TOS
VPOP	Pops last value from end of the Vector on TOS
T	Transpose N vectors starting from TOS, into M vectors using transposition *(swapping rows with columns)*
LUNION	Merge-join two operands of Labels type
LPUSH	Push immediate label onto Labels on TOS
LPOP	Pop last label from Labels on TOS
JUPD	Perform Join-Update.
JADD	Perform Join-Add.
JFLT	Perform Join-Filter.
ADD	Performs element-wise or scalar addition *(binary)*.
SUB	Performs element-wise or scalar subtraction *(binary)*.
SSB	Performs element-wise or scalar saturating subtraction *(binary)*.
MUL	Performs element-wise or scalar multiplication *(binary)*.
DIV	Performs element-wise or scalar division *(binary)*.
SQRT	Take square root element-wise or from scalar *(binary)*.
MIN	Take minimum element-wise or from scalar *(binary)*.
MAX	Take maximum element-wise or from scalar *(binary)*.
VSUM	Sum all elements of a vector *(unary)*.
VMIN	Find minimum in the vector *(unary)*.
VMAX	Find maximum in the vector *(unary)*.
IMMS	Load immediate value.
IMML	Load immediate label.
ZEROS	Creates a Vector of zeros, matching the length of a reference object.
ONES	Creates a Vector of ones, matching the length of a reference object.
POPN	Removes a specified number of items from the stack.
SWAP	Swaps TOS with an operand at a relative stack position.
B	Branches execution to a subroutine, passing N inputs and receiving M outputs.
FOLD	Iteratively applies a subroutine (Fold/Reduce) over elements of a Vector operand.

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

These instructions are responsible for fetching data from on-chain storage or internal registers and pushing it onto the execution stack.

3.1.1 LDL (Load Labels)

Loads a Labels object (metadata, program code) from a designated Blockchain storage slot (SLOAD) and pushes it onto the stack (TOS).

Operands

- `<label_id>`: `u128` Storage ID.

Usage Example

```
LDL     asset_names_id
```

3.1.2 `LDV` (Load Vector)

Loads a `Vector<Amount>` object from a designated Blockchain storage slot (`SLOAD`). This involves fetching the binary blob from storage and deserializing it into a `Vec<Amount>` struct. Pushes the Vector onto the stack.

Operands

- `<vector_id>`: `u128` Storage ID.

Usage Example

```
LDV     asset_weights_id
```

3.1.3 `LDD` (Load Duplicate)

Creates a copy of the operand at a specified stack position and pushes the duplicate onto the stack (TOS). Useful for reusing an operand without recalculating or reloading it.

Operands

- `<pos>`: Stack index relative to TOS (0 is TOS, 1 is the item below TOS).

Usage Example

```
LDD     1
```

3.1.4 `LDR` (Load from Registry - Read)

Loads a value from a named internal register (`R0–Rn`) and pushes it onto the stack. The value remains in the registry, allowing multiple reads.

Operands

- `<reg>`: Registry name (e.g., `_Weights`).

Usage Example

```
LDR     _Weights
```

3.1.5 `LDM` (Load from Registry - Move)

Loads a value from a named internal register (`R0`–`Rn`) and pushes it onto the stack. The value is *removed* from the registry, similar to a move operation in high-level languages.

Operands

- `<reg>`: Registry name (e.g., `_Spent`).

Usage Example

```
LDM     _Spent
```

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

These instructions are responsible for persisting data to the blockchain or internal registers and removing data from the stack.

3.2.1 `STL` (Store Labels)

Stores the Labels object currently on the stack (TOS) into a designated Blockchain storage slot (`SSTORE`). Consumes TOS.

Operands

- `<label_id>`: `u128` Storage ID.

Usage Example

```
STL     new_labels_id
```

3.2.2 `STV` (Store Vector)

Stores the Vector object currently on the stack (TOS) into a designated Blockchain storage slot (`SSTORE`). This serializes the `Vector<Amount>` back to a binary blob for efficient storage. Consumes TOS.

Operands

- `<vector_id>`: `u128` Storage ID.

Usage Example

```
STV     executed_quantities_id
```

3.2.3 `STR` (Store to Registry)

Stores the stack operand (TOS) into a named internal register (`R0`–`Rn`). Consumes TOS. This is the primary way to temporarily save intermediate results.

Operands

- `<reg>`: Registry name (e.g., `_Weights`).

Usage Example

```
STR     _Weights
```

3.2.4 `PKV` (Pack Vector)

Takes N items from the top of the stack and packs them into a single `Vector` or `Tuple` operand, replacing the N items with the single packed item. This is crucial for saving multi-field objects into a single storage slot.

Operands

- `<count>`: The number of stack items to pack.

Usage Example

```
PKV     3
```

3.3 3. Arithmetic Operations (Opcode 30-34)

These instructions perform core arithmetic logic, optimized for vector and scalar interactions.

3.3.1 `ADD` (Addition)

Performs element-wise addition (Vector + Vector) or scalar addition (Scalar + Scalar, or Vector + Scalar). The second operand is implicitly the one found at `[TOS - pos]`.

Operands

- `<pos>`: Stack index of the second operand relative to TOS.

Usage Example

```
ADD     1
```

Adds the top two stack items.

3.3.2 `SUB` (Subtraction)

Performs element-wise subtraction (Vector - Vector) or scalar subtraction (Scalar - Scalar). The second operand is implicitly the one found at `[TOS - pos]`.

Operands

- `<pos>`: Stack index of the second operand relative to TOS.

Usage Example

```
SUB     1
```

Subtracts the second item on stack from TOS.

3.3.3 `ONES` (Create Vector of Ones)

Creates a new Vector where every element is the value `1`, matching the length of a Vector or Labels object found at `[TOS - pos]`. Pushes the new Vector on TOS. This is useful for initialization or masks.

Operands

- `<pos>`: Stack index of the reference object whose length is used.

Usage Example

```
ONES    0
```

Creates a vector of ones matching the length of the current TOS.

3.4 4. Stack Control & Program Flow (Opcode 90-94)

These instructions manage the flow of execution and the state of the stack.

3.4.1 `POPN` (Pop N)

Removes a specified number of items from the top of the stack. Used to clean up temporary results after they are consumed or stored.

Operands

- `<count>`: Number of values to pop.

Usage Example

```
POPN    3
```

Removes the three topmost elements.

3.4.2 `SWAP` (Swap)

Swaps the top element of the stack (TOS) with the element located at the specified position. Essential for re-ordering operands for arithmetic operations.

Operands

- `<pos>`: Stack index of the item to swap with TOS.

Usage Example

```
SWAP    3
```

Swaps TOS with the element three positions beneath it (the 4th element down).

3.4.3 `B` (Branch / Subroutine Call)

Calls a sub-routine (another VIL program) identified by `prg_id`. It moves N inputs from the current stack to the subroutine's stack and, upon completion, moves M outputs from the subroutine's stack back to the current stack.

Operands

- `<prg_id>`: `u128` Program/Labels ID.
- `<N>`: Number of inputs consumed from the caller's stack.
- `<M>`: Number of outputs produced to the caller's stack.
- `<R>`: R-value (Reserved/Placeholder).

Usage Example

```
B   solve_quad  3   1   8
```

Calls the solver program with 3 inputs, expects 1 output, and uses 8 registers.

3.4.4 `FOLD` (Fold / Reduce)

A powerful instruction that applies a subroutine (`prg_id`) iteratively over a Vector operand on the stack, accumulating a result (e.g., sum, product, or complex reduction). This implements functional programming's 'fold' or 'reduce' pattern.

Operands

- `<prg_id>`: `u128` Program/Labels ID.
- `<N>`: Input/Accumulator management (controls how the accumulator and next element are fed into the subroutine).

- <M>: Output management.
- <R>: R-value (Reserved/Placeholder).

Usage Example

```
FOLD    sum_subroutine  2   1   8
```

Fold over elements of a vector on TOS using sub routine taking initially 2 inputs from stack, and returning one output at the end, and using 8 registers.