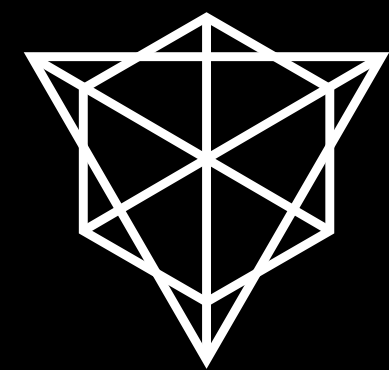


AluVM and its runtime environments

Dr Maxim Orlovsky,



Pandora Core AG

work donated to LNP/BP Standards Association under Apache2/MIT/CC0 licenses



AluVM – from “arithmetic logic unit”

github.com/internet2-org/aluvm-spec

- Purely functional & arithmetical: each operation is an arithmetic function
- No external state; converts set of inputs into false/true validation result
- Extremely robust & deterministic: no exceptions are possible
 - no stack (register-based VM)
 - no random memory access
 - no I/O, memory allocations
- If it compiles, it will always run successfully
- Easy to be implemented in hardware, like in FPGAs
- Any byte sequence presents a valid program
(important for both robustness and genetic algorithms)

Robustness at the core: no UB

- All registers may be in the *undefined* state
- Impossible/incorrect operations put destination into *undefined* state
- Code always extended to 2^{16} bytes with zeros, which corresponds to “set *st0* register to false and stop execution” op-code
- There are no invalid jump operations
- There are no invalid instructions
- Cycles & jumps are counted with 2^{16} limit (bounded-time execution)
- No ambiguity: any two distinct byte strings always represent strictly distinct programs
- Code is signed
- Data segment is signed
- Code commits to the used ISA extensions
- Libraries identified by the signature
- Code does not run if not all libraries are present

	AluVM	Bitcoin script	EVM, kEVM, IELE	WASM	JVM, CLR	LLVM
Type	Register	Stack	Stack	Stack	Stack	Stack
Random memory access	No	No	No	Yes	Yes	Yes
Dynamic memory allocations	No	Yes	Yes	Yes	Yes	Yes
I/O operations	No	No	No	Via extensions	Yes	Yes
Turing completeness	Yes (bounded)	No	Yes (bounded)	Yes	Yes	Yes
Static analysis	Simple	Simple	Complex	Hard	Hard	Hard
Sandboxing	Always	Always	Always	Poor	No native	No native
Runtime environment	Any sandboxed	UTXO blockchain	Account-based blockchain	Internet	OS	Compiler
Library code immutability	Yes	No libraries	Yes	No	No	No
Undefined behavior (UB)	Impossible	Possible	Possible	Possible	Possible	Possible

History

- The need for AluVM recognized as a part of RGB project
24 & 31st of Mar 2021 – youtu.be/JmKNy0Mv68I

RGV VM selection criteria

	Embedded procedures (<i>status quo</i>)	AluVM	WASM	Simplicity
Code audibility	Poor	Good	Average	Excellent (formally provable)
Safety	Endianness is tricky	Good	Average	Excellent
Cryptographic primitives	Grin-based, hard migration	Secp256k1	Not known, but smooth migration	Not ready
Schema validation implementation effort	Medium	Low	High	Extreme high
Can be used by other schema devs	No	Yes	Easily	Not really

History

- The need for AluVM recognized as a part of RGB project
24 & 31st of Mar 2021 – youtu.be/JmKNy0Mv68I
- Concept presented on 19th of May 2021 – youtu.be/Mma0oyiVbSE
- v0.1 release of Rust AluVM implementation on 28th of May 2021
(API docs docs.rs/alure/0.1.0/alure/)
- v0.2 release with multiple enhancements on 9 Jun 2021
(API docs docs.rs/alure/0.2.1/aluvm/)

Changes since last presentation

AluVM v0.2

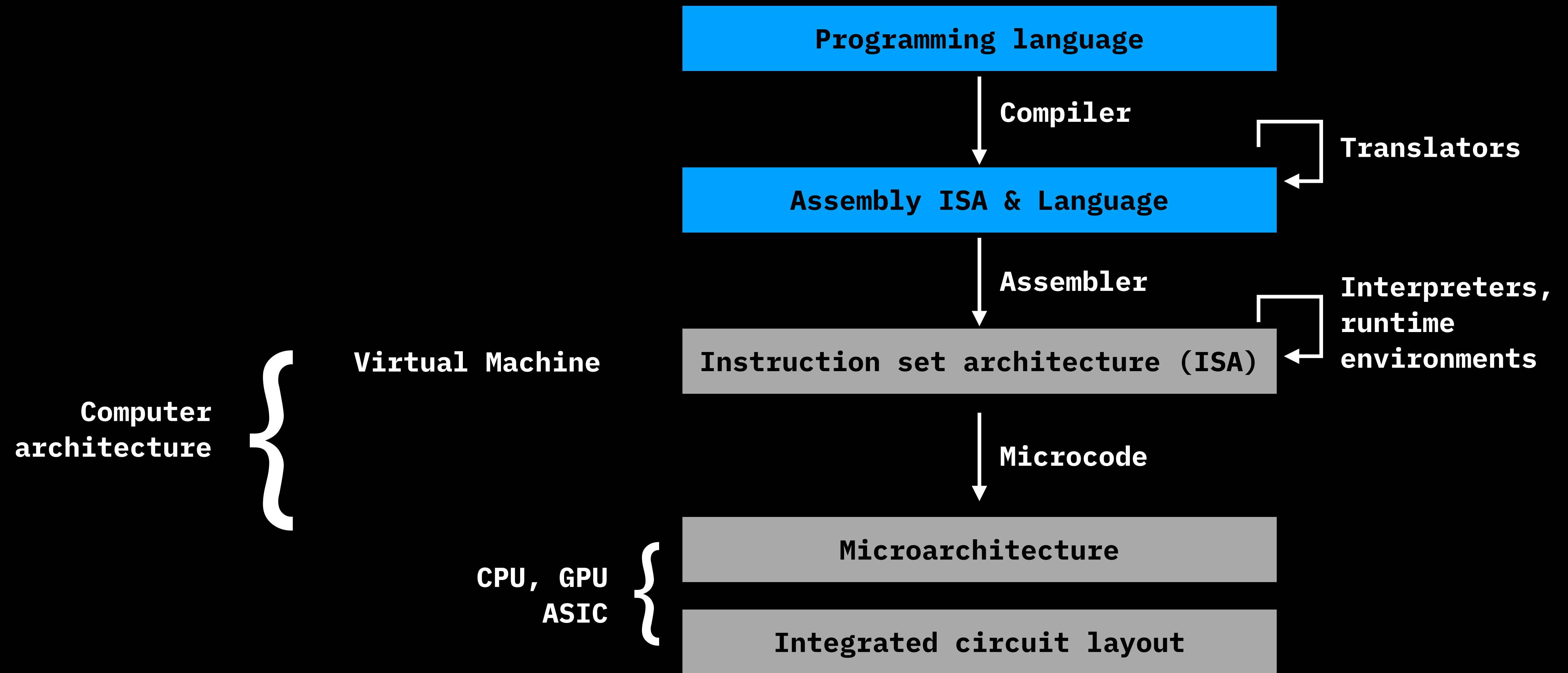
- Better & new float types and dedicated registers
- Instruction set refactoring
- Clear VM & runtime environment separation

AluVM v0.3 (scheduled)

- Introduction of data & code separation: data segment
- Library packaging format
- Bounded execution complexity

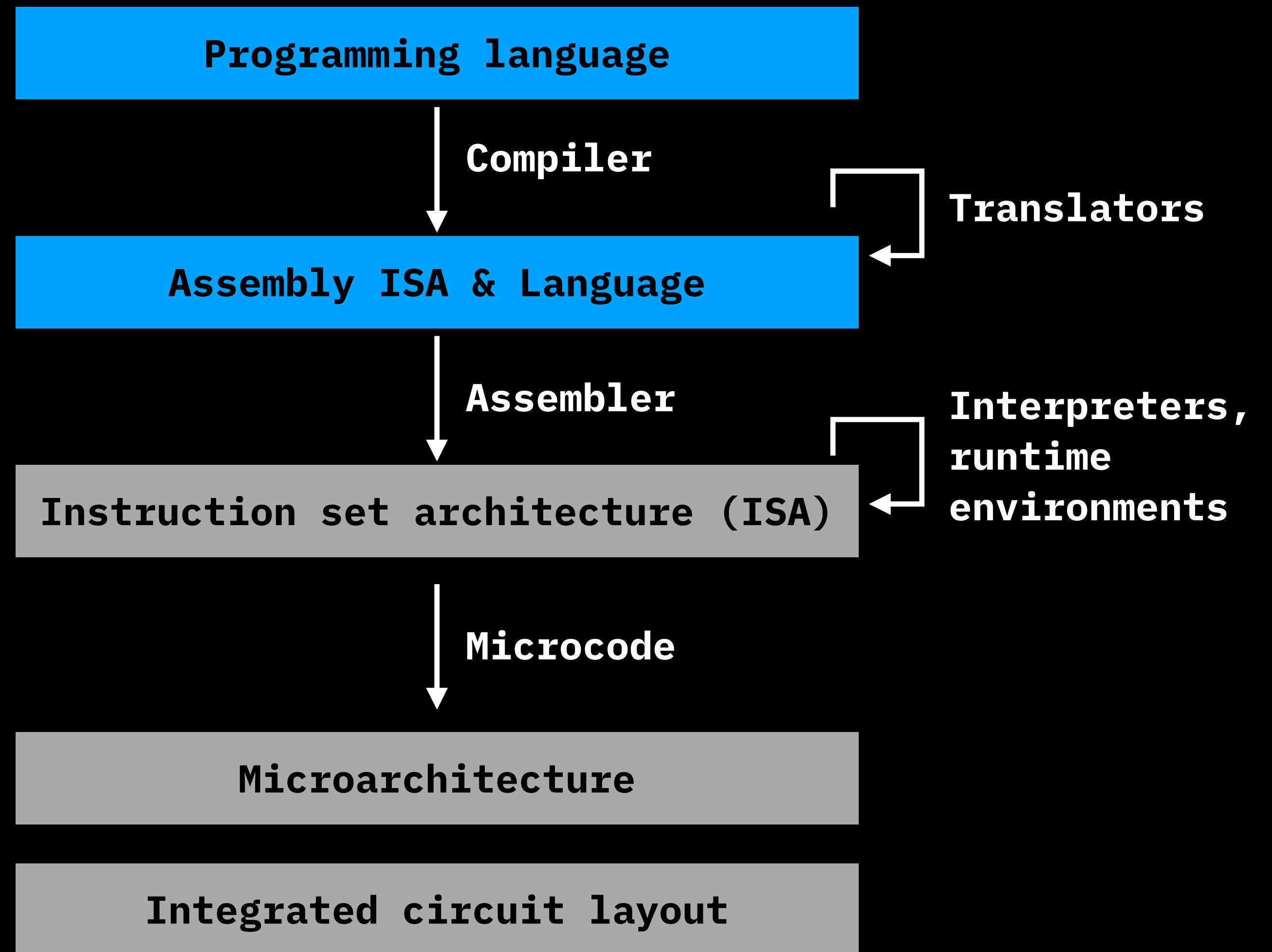
AluVM architecture

Programming & computing architectures



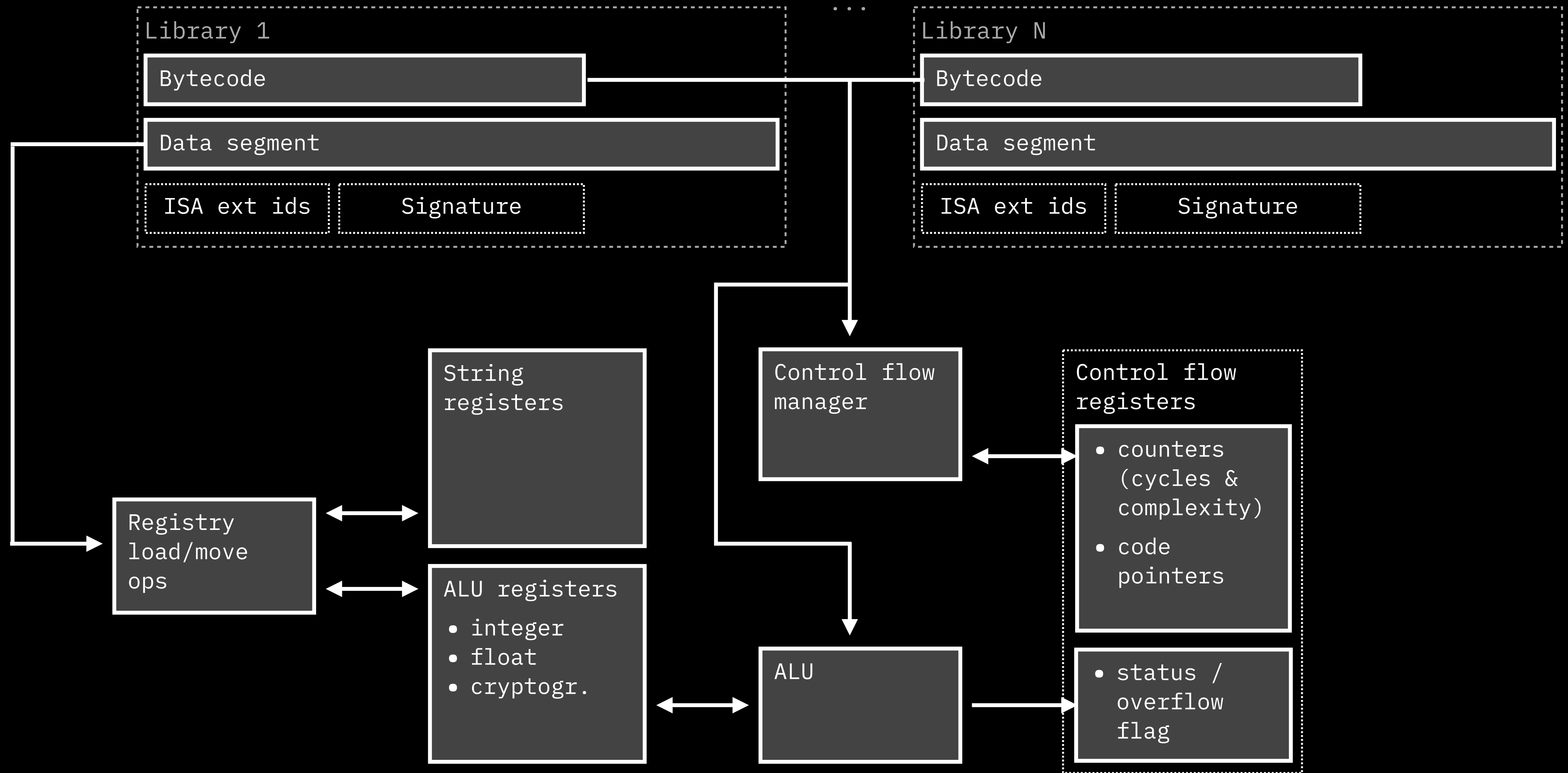
AluVM Architecture

- Programming language: none (yet)
 - Use with Contractum
 - Potentially any LLVM-compiling language
- Assembly ISA: Alu Assembly
- ISA:
 - AluVM
 - ISA extensions:
ALURE, RGB, BP, LNP, SIMD, WEB4, REBICA
- Microarchitecture: none (yet)
 - Uses runtime environments to run on
x86_64 and ARM64
- IC Layout: planned (first FPGA, then ASIC)



AluVM ISA

Instruction set architecture



Instructions

- RISC: only 256 instructions
- 3 families of core instructions:
 - Control flow
 - Data load / movement between registers
 - ALU (including cryptography)
- Extensible with ISA extensions
127 of the operations are reserved for extensions
 - More cryptography
 - Custom data I/O (blockchain, LN, client-side-validation)
 - Genetic algorithms / code self-modification

ISA Extensions

- Standardized & maintained by LNP/BP Standards Association
- Identified by ISAE Id:
 - Consists of capital letters
 - Up to 32 bytes in total length
 - Kept as part of the library as a zero-separated list
 - Assigned by LNP/BP Standards Association via new LNPBP standard

Currently defined:

- ALU – Base instruction set (always present & not explicitly given)
- ALURE – AluVM runtime environment
- RGB – RGB Core runtime
- BP – access to bitcoin blockchain data
- LNP – access to LN channel state and tx structure
- WEB4 – sandboxed networking & local storage
- SIMD – parallel or scientific computing (linear algebra), machine learning
- REBICA – for biologically-inspired forms of computing

ALU instruction robustness

- Impossible arithmetic operation (0/0, Inf/inf)
always sets the destination register into *undefined* state
(unlike *NaN* in IEEE-754 it has only a single unique value)
- Operation resulting in the value which can't fit the bit dimensions under a used encoding, including representation of infinity for integer encodings ($x/0$ if $x \neq 0$) results in:
 - for float underflows, subnormally encoded number,
 - for $x/0$ if $x \neq 0$ on float numbers, $\pm Inf$ float value,
 - for overflows in integer checked operations and floats: *undefined* value, setting *st0* to *false*,
 - for overflows in integer wrapped operations, modulo division on the maximum register value

ALU instruction robustness

Most of the arithmetic operations have to be provided with flags specifying which of the encoding and exception handling should be used:

- Integer encodings has two flags:
 - one for signed/unsigned variant of the encoding
 - one for checked or wrapped variant of exception handling
- Float encoding has 4 variants of rounding, matching IEEE-754 options

Thus, many arithmetic instructions have 8 variants, indicating the used encoding (unsigned, signed integer or float) and operation behavior in situation when resulting value does not fit into the register (overflow or wrap for integers and one of four rounding options for floats).

ISA documentation

←

→

↺

🏠

https://docs.rs/aluvm

🔍 Search

📖 ⓘ ☰

DOCS.RS

aluvm-0.2.1


Platform

Feature flags

Releases

Rust

Find crate



Enum BytesOp

Variants

Cnt

Con

Del

Eq

Extr

Fill

Find

Inj

Ins

Join

Len

Mov

Put

Rev

Splt

Swp

Trait Implementations

Bytecode

Clone

Debug

Display

Eq

Splt(SplitFlag, Reg32, u8, u8, u8)

Split bytestring at a given offset taken from a16 register into two destination strings, overwriting their value. If offset exceeds the length of the string in the register, then the behaviour is determined by the SplitFlag value.

+-----+
| |
+-----+
 ^ ^
 | +-- Split offset (`offset`)
 +-- Source string length (`src_len`)

offset == 0: (1) first, second <- None; st0 <- false (2) first <- None, second <- src_len > 0 ? src : None; st0 <- false (3) first <- None, second <- src_len > 0 ? src : zero-len; st0 <- false (4) first <- zero-len, second <- src_len > 0 ? src : zero-len
offset > 0 && offset < src_len: st0 always set to false (1) first, second <- None (5) first <- short, second <- None (6) first <- short, second <- zero-len (7) first <- zero-ext, second <- None (8) first <- zero-ext, second <- zero-len offset = src_len: (1) first, second <- None; st0 <- false (5,7) first <- ok, second <- None; st0 <- false (6,8) first <- ok, second <- zero-len offset > src_len: operation succeeds anyway, st0 value is not changed

Rule on st0 changes: if at least one of the destination registers is set to None, or offset value exceeds source string length, st0 is set to false; otherwise its value is not modified

Ins(InsertFlag, Reg32, u8, u8)

Insert value from one of bytestring register at a given index of other bytestring register, shifting string bytes. If the destination register does not fit the length of the new string, or the offset exceeds the length of destination string operation behaviour is defined by the provided InsertFlag.

+-----+
| |
+-----+
 ^ ^
 | +-- Insert offset (`offset`)
 +-- Destination string length (`dst_len`)

offset < dst_len && src_len + dst_len > 2^16: (6) Set destination to None (7) Cut destination string part exceeding 2^16 (8) Reduce src_len such that it will fit the destination offset > dst_len && src_len + dst_len + offset <= 2^16: (1) Set destination to None (2) Fill destination from dst_len to offset with zeros (3) Use src_len instead of offset
offset > dst_len && src_len + dst_len + offset > 2^16: (4) Set destination to None (5) Fill destination from dst_len to offset with zeros and cut source string part exceeding 2^16 (6-8) Use src_len instead of offset and use flag value from the first section

Registers

ALU & String

- ALU registers: 8 blocks of 32 registers
 - Integer arithmetic (A-registers)
blocks: 8, 16, 32, 64, 128, 256, 512, 1024 bits
 - Float arithmetic (F-registers)
blocks:
 - IEEE: binary-half, single, double, quad, oct precision
 - IEEE extension: 80-bit X87 register
 - Machine learning BFloat16 register
 - Cryptographic operations (R-registers)
blocks: 128, 160, 256, 512, 1024, 2048, 4096, 8192 bits
- String registers (S-registers): 1 block of 256 registers, 64kb each

Control Flow

- Status (**st0**),
boolean (one bit)
- Cycle counter (**cy0**),
16 bits
- Instruction complexity accumulator (**ca0**),
16 bits
- Call stack register (**cs0**),
3*2¹⁶ bits (192kB block)
- Call stack pointer register (**cp0**),
16 bits

Data segment

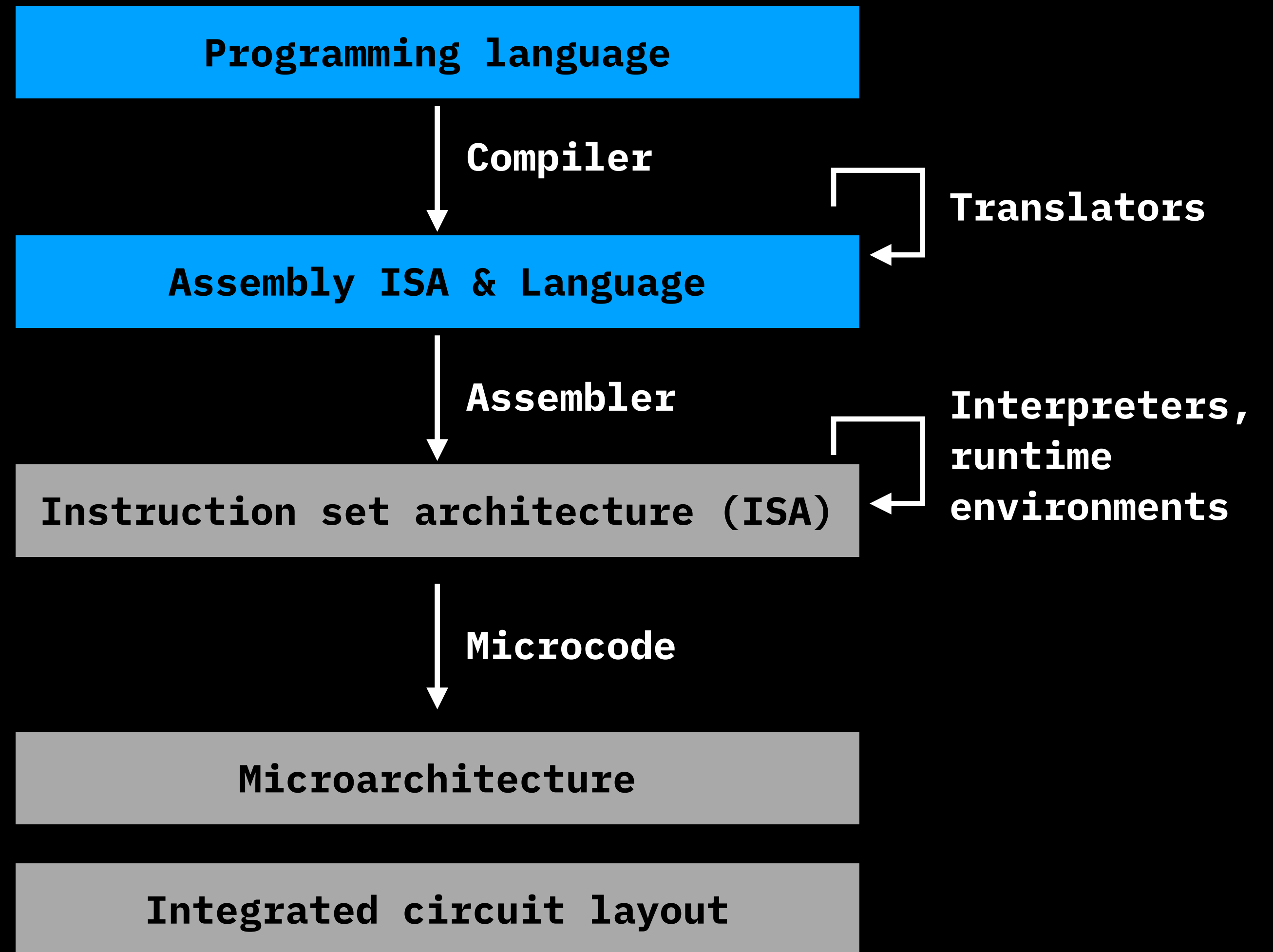
- Constant data for loading into registers
- Bounded by 2^{24} bytes (16MB) per library in total
- Addressable from the code by 24-bit offset and, for variable-sized data, 16-bit length

AluVM requirements for microarch & IC layout

- 162 microinstructions
- Registers:
 - ALU (fast access) – ~77 kB of data
 - ▶ 65 kBit for integer registers
 - ▶ 25 kBit for float registers
 - ▶ 525 kBit for cryptographic registers
 - 16 mB for string data (similar to L2 cache/core size in M1)
 - 192 kB for call stack (similar to L1 cache/core size in M1)

AluVM ecosystem

- Compiler: none
- Assembler: aluasm
 - rust-based
 - takes rust DSL as an input
- Runtime environments
 - AluRE: general sandboxed computing, including Web4 applications or distributed computing tasks
 - RGB: for RGB smart contract state validation
 - LNP: for custom LN channels business logic
 - REBICA: runtime environment for biologically-inspired computing architectures



Alu Assembly Language

Alu Assembly

- Very similar to x86 assembly, but:
 - RISC, i.e. smaller instruction set
 - No memory access
 - No stack operations
 - Has floating point, string & cryptographic operations
- Assembler (assembly compiler) is implemented as Rust DSL

Simple PoW mining with Alu Assembly

```

    read s[1], $input           ; AluRE ISA extension opcode to read
                                ; user/dynamic data with the given $id

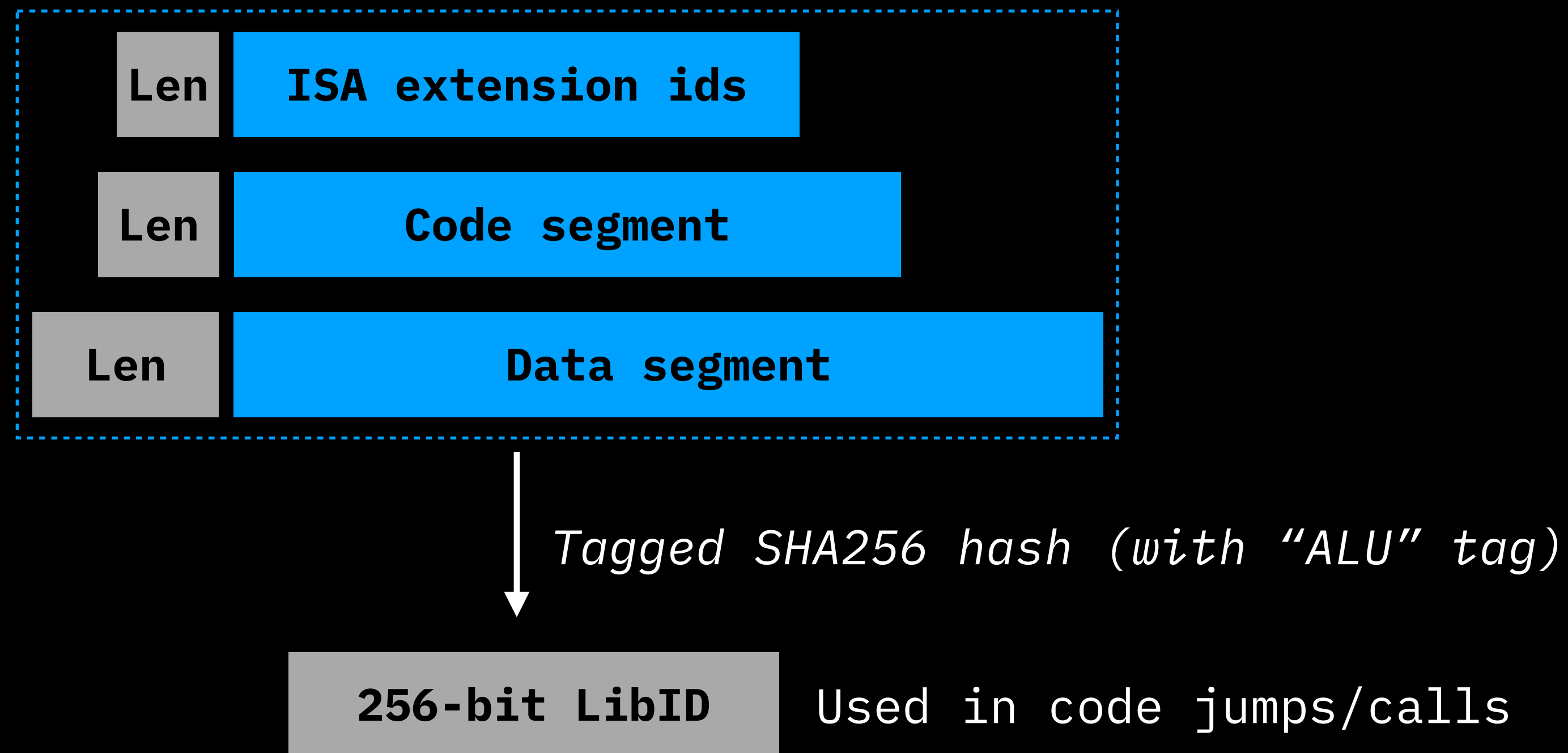
    read r256[1], $difficulty
    read a16[2], $cycle_limit
    mov  a16[1], 0               ; putting a value into register
loop:                               ; label for cycle

    sha2 s[1], r256[2]          ; taking hash of the data
    extr r256[2], s[1]          ; saving hash result
    inc  a16[1]                 ; counting steps
    ge   a16[1], a16[2]         ; making sure we do not exceed $cycle_limit
    jif  exceeded               ; if greater or equal, jump to exceeded
    le   r256[2], r256[1]       ; checking against difficulty
    jif  done                   ; if less or equal, jump to done
    jmp  loop

done:                               ; target difficulty reached!
    succ
exceeded:                          ; failing since we exceeded $cycle_limit
    fail
```

AluVM Libraries

Library identity



- No code change is possible
- No data change is possible
- No mis-interpretation with different ISA is possible

Library naming

- Consists of human-readable org name and lib name pair, separated by ':'

Org : **Name**

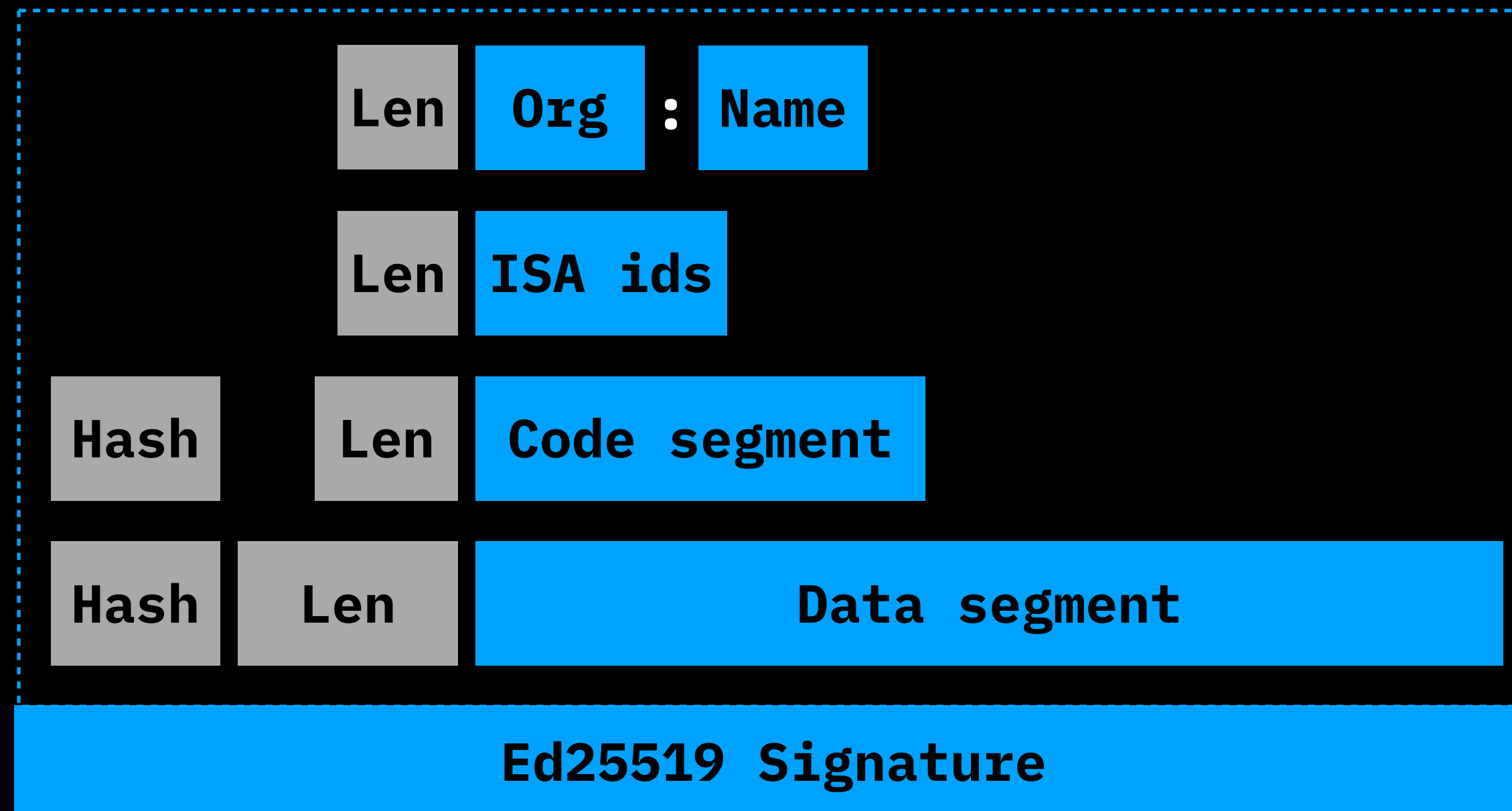
- Both are self-assigned
- No global names;
Organization names are managed via WoT public keys from library signature
(i.e. each dev defines trust list for public key and checks the org name against that list)
- Library name is assigned by the library creator
- NB: the only library id is the LibID
 - Runtime environments use manifests that associates LibIDs with respective names, like

```
zpk = { org: "lnpbp", name: "lnpbp", libid: "22feebf0c875e8b1ff38a17780e8efbfcf2b95ee28405d3d1cd5fc496a0594e7" }
```
 - Assembly language creates manifest using directives/pragmas

```
@pragma name zpk lnpbp:zpk 22feebf0c875e8b1ff38a17780e8efbfcf2b95ee28405d3d1cd5fc496a0594e7
```

Library packaging

.alu file extension



- Bytecode (up to 2^{16} bytes)
- Data segment (up to 2^{24} bytes)
- Name: library URN, which includes producer name
- Stores hashes of code and data segment for detection of modifications
- Library linking ID: hash on combined byte code + data segment + ISA extensions
- Package ID:
signature on $\text{SHA256}(\text{LibName} || \text{LibID})$

Runtime environments

Runtime environment provides:

- Libraries & dynamic linking
- Package management
- Executable format
- Package IDs
- Set of allowed ISA extensions

Runtime Environments

RGB Core AluVM runtime
for client-side state validation

RGB state validation

Programmable validation of state evolution inside RGB state transitions

- Allows creation of custom schema with custom logic;
- Allows algorithmically stable coins, programmable inflation/deflation and many more complex scenarios.

RGB Core AluVM Runtime Environment (RARE)

- Input comes from state transition:
 - previous state assigned to single-use seals
 - new state data assigned to single-use seals
 - metadata
 - bitcoin transaction data for arbitrary TxId
- Output: value of st0 register
(true/false, meaning validated/invalid)

Runtime Environments

AluRE: generic sandboxed environment

AluRE: Native AluVM Runtime Environment

- I/O via “dynamic data segment”
- Inputs are taken from dynamic data segment, outputs are taken from register values
- Provides ISAE for reading values from a special dynamic data segment, which may be defined by the user
- Declares which registers should be used by output
- Includes ISAE: SIMD, WEB4

Simple PoW mining with Alu Assembly

```

    read s[1], $input           ; AluRE ISA extension opcode to read
                                ; user/dynamic data with the given $id

    read r256[1], $difficulty
    read a16[2], $cycle_limit
    mov  a16[1], 0               ; putting a value into register
loop:                                ; label for cycle

    sha2 s[1], r256[2]          ; taking hash of the data
    extr r256[2], s[1]          ; saving hash result
    inc  a16[1]                  ; counting steps
    ge   a16[1], a16[2]          ; making sure we do not exceed $cycle_limit
    jif  exceeded
    le   r256[2], r256[1]        ; checking against difficulty
    jif  done                    ; target difficulty reached!
    jmp  loop
done:    succ
exceeded: fail                   ; failing since we exceeded $cycle_limit
```

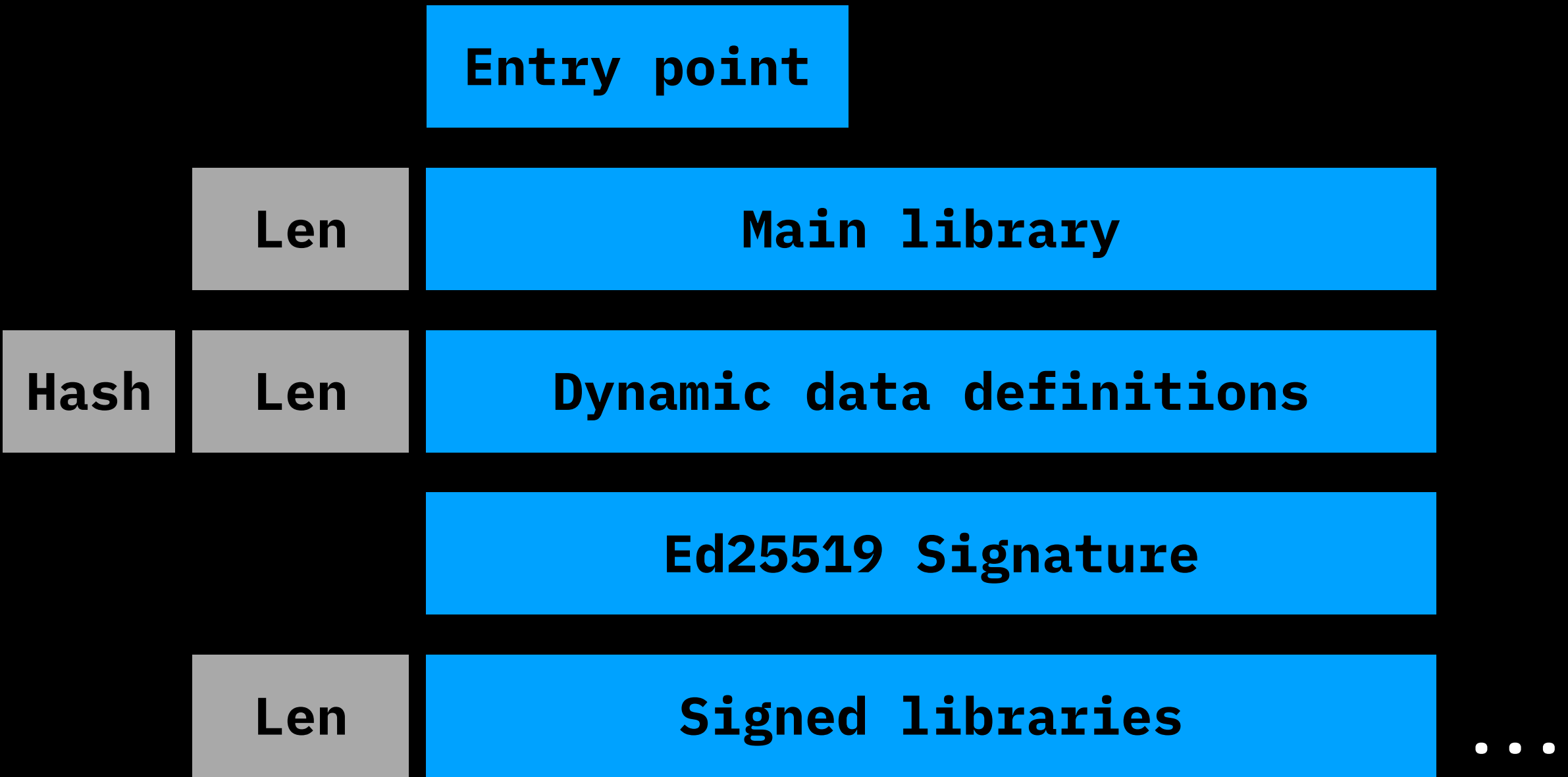
Dynamic data definitions

- User-friendly name (may be displayed for user input)
- Data type (matching one of the registry format)
 - While it can be derived from the code, it can't be deterministically derived (i.e. two instructions may use the same id for different registers), so we need this commitment to the data type*
 - Includes limits on string length
- Program-defined 16-bit ID (used inside the instruction)
- List of output registers with their user-friendly names

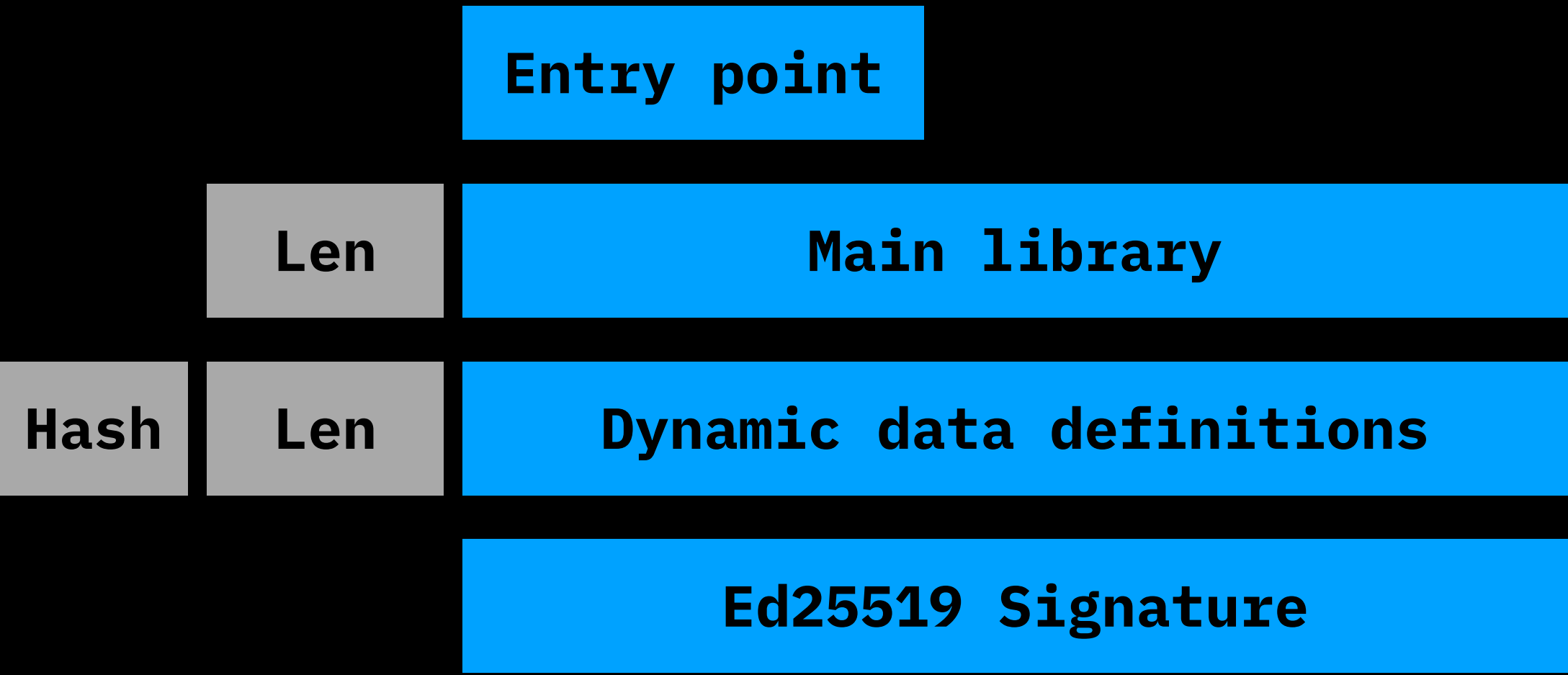
AluRE executable packaging

For distribution

For execution



.rpk file extension



.rex file extension

AluRE for writing Node extensions

- In LNP it will be possible to program custom channel structures without recompiling LN node code
- In RGB it will be possible to construct user interaction for custom schemata without requiring to re-compile/update wallet software for each new schema type.

This is different from state RGB validation with RARE:

- *not used for state or ownership client-side-validation;*
- *instead, constructs a new valid state with user input without the need to modify the host wallet software.*

Runtime Environments

Future use cases & runtimes for AluVM

Blockchain & client-side-validation scripting

For systems that lack native script.

For instance, if there will be a blockchain-free single-use-seal commitment medium (like in “sigchain” proposal), an RGB-like client-side-validation system will need scripting to control ownership rights (since there is no “bitcoin script” anymore).

Also, one may add “additional scripting” to RGB for extended ownership control, further restricting conditions specified by Bitcoin script.

NB: This is different from the previous use case, since this is not about state validation, but about ownership validation.

Multiparty computing and deterministic ML

- Multiparty computing, including trustless distributed computations, require deterministic VM without I/O. Existing VMs are either I/O based and non-deterministic (like JVM or WASM, having random memory access) or poorly suited for general computing providing unnecessary blockchain-specific functionality (EVM, IELE).
- Today's machine learning computing is non-deterministic which prevent its use in trustless distributed environments since two parties can't reach agreement on the computation result. AluVM solves this problem.

Remote code execution: Internet2 / Web4

“Better JVM/JavaScript/.NET/WASM”

Address the same problem for which these VMs were created –
but better in doing so, since it is

- deterministic and can't generate exceptions;
- natively isolated / purely sandboxed:
has no side effects, memory access etc.;
- has immutable library version commitments to the actual library code
(no “dependency hell” is possible as in all other systems);
- bounded by size & memory use, can run on even low-profile IoT hardware;
- formally verifiable.

Genetic algorithms & computational lifeforms

- Perfect for program self-polymorphism (as in genetic algorithms) since
 - any byte sequence represents a valid code which can be successfully executed;
 - and does not have an external state/memory access.
- Use cases range from simple genetic algorithms to cellular automata modelling and experimental computational lifeforms development.

Read more

- Specification:
github.com/internet2-org/aluvvm-spec
- ISA & API documentation:
docs.rs/aluvvm/0.2.1/aluvvm/
- Rust codebase for AluVM:
github.com/internet2-org/rust-aluvvm
- First concept presentation:
youtu.be/Mma0oyiVbSE