Jolt

SNARKs for virtual machines via lookups

Arasu Arun[*], Srinath Setty[†], Justin Thaler[§‡]
Michael Zhu[§], Sam Ragsdale[§], Noah Citron[§]

[*]New York University [†]Microsoft Research
[§]a16z crypto [‡]Georgetown University

Important Disclosures

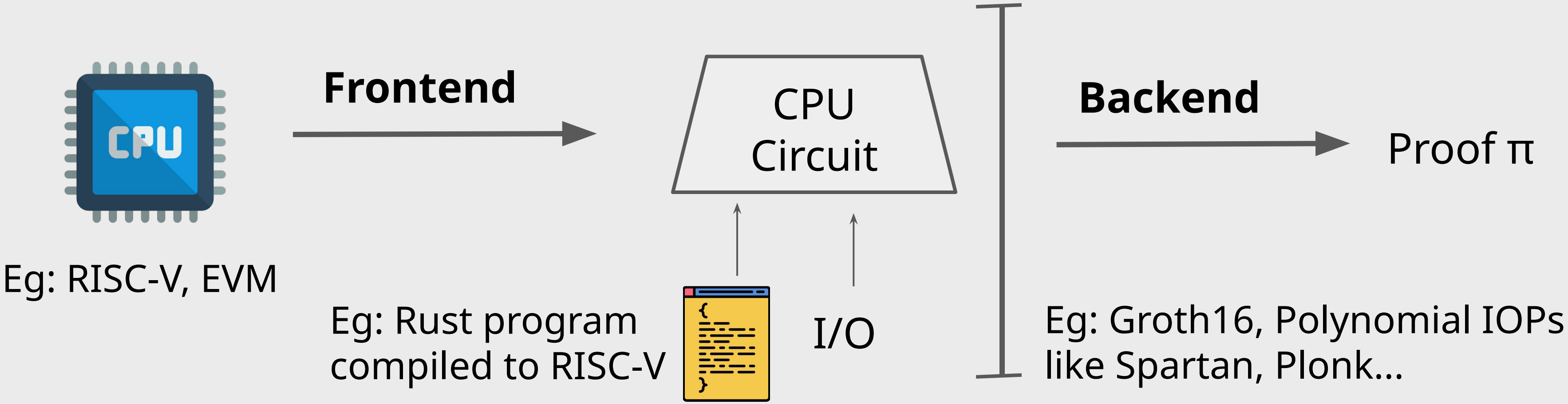
The views expressed here are those of the academic researcher and not necessarily of AH Capital Management, L.L.C. (“a16z”).

This content is provided for academic and informational purposes only, and should not be relied upon as business, investment, tax or legal advice and does not constitute an offer of advisory services.

Charts, graphs or other diagrams provided within are for academic and informational purposes only and should not be relied upon when making any investment decision. Past performance is not indicative of future results. The content speaks only as of the date indicated.

References to any securities, digital assets, and/or tokens (“assets”) are for illustrative purposes only and a16z may maintain holdings in the these assets. A list of investments made by funds managed by Andreessen Horowitz (excluding investments for which the issuer has not provided permission for a16z to disclose publicly as well as unannounced investments in publicly traded digital assets) is available at https://a16z.com/investments/.

zkVM: frontends and backends



Pros:

- One circuit for all programs.
- Re-use infrastructure: existing languages, compilers and tooling.
- Focus audit and optimization efforts on one circuit.

Overheads involved in CPU circuits

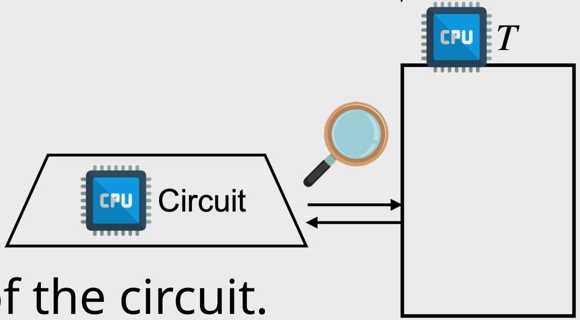
```
switch (instr) {
    case ADD: {..}
    case SUB: {..}
    ...
    case XOR: {..}
}
```

1. To handle arbitrary programs, each step must handle **all possible operations**.

   RISC-V: ~ 50 operations
   EVM: ~ 140 operations

   These incur extra field operation and/or commitment costs.

2. Also, **bitwise operations** involved in primitive ISAs aren't efficiently performed with field elements (such as in a prime-order field).



Jolt: just one lookup table

A zkVM that mostly performs **lookups** to tables outside of the circuit.

- Minimal: Just **60 constraints** and **80 field elements** per step of RISC-V!

How?

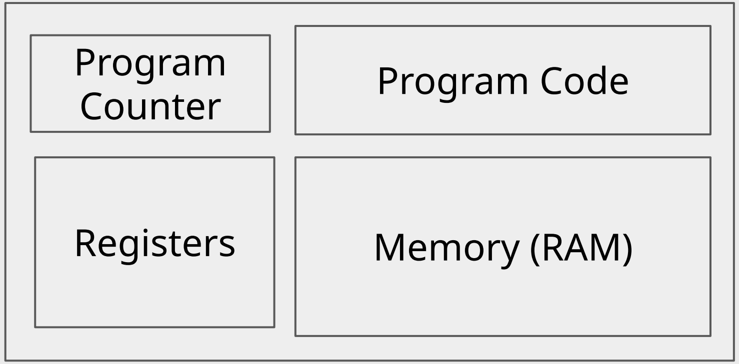
- Primitive assembly instructions have interesting **mathematical structure** (namely, efficient polynomial representations)
- We leverage this to perform structured lookups using **Lasso**.

This leads to a modular and extensible framework.

Lasso (ia.cr/2023/1216) was developed alongside Jolt.

CPU state, transition arithmetization

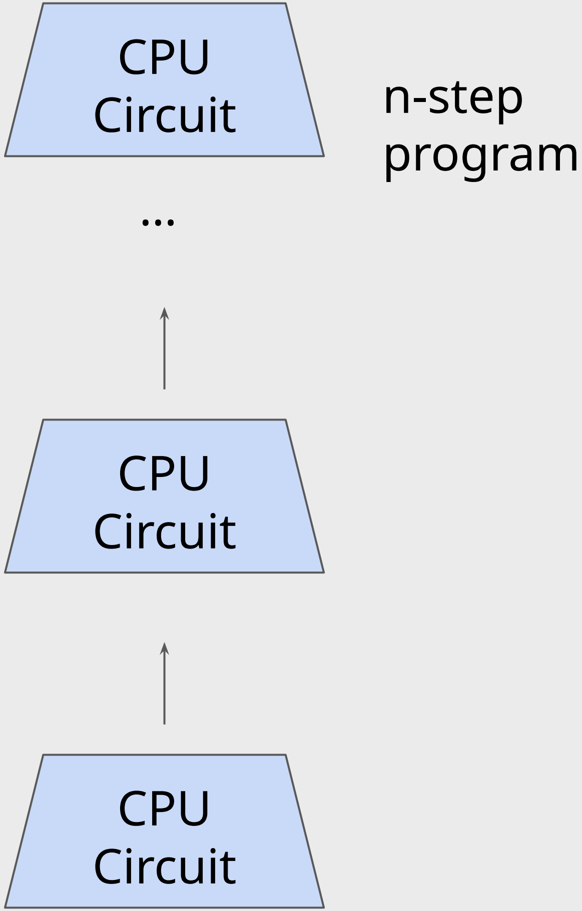
Machine state:

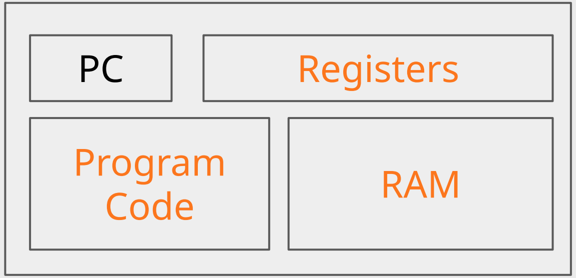


Transition function:

1. **Fetch** instr.
2. **Decode** opcode, operands.
3. **Execute** instruction.
4. **Update** registers.

Frontend ⇒

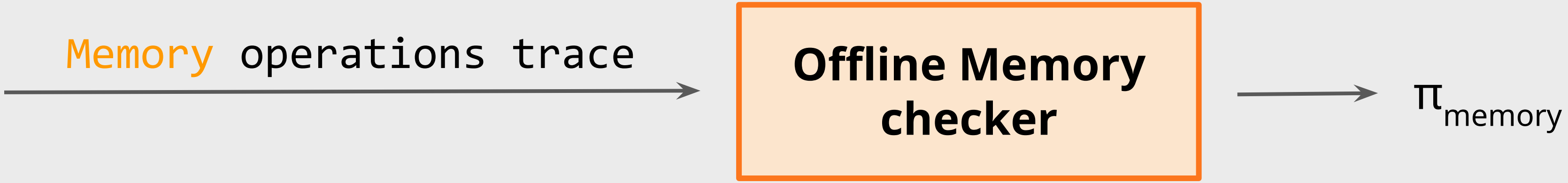




Memory-checking

We use “offline memory checking” (BEGKN91). Adapted to SNARKs in Spice (SAGL18).

Reduces consistency of memory operations to a **multiset equality check**. Jolt performs this using GKR-style grand product arguments.



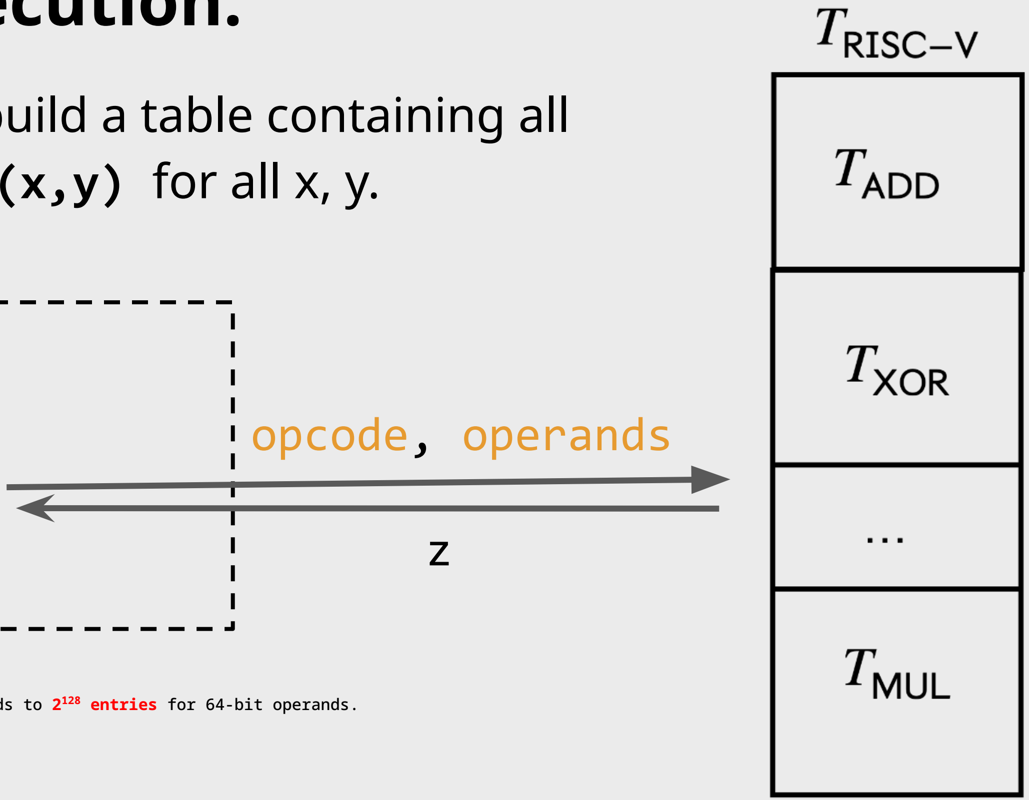
Prover is linear time in the number of memory operations and memory size.

[BEGKN91] - Checking the correctness of memories - Blum et al., 1991
[SAGL18] - Proving the correct execution of concurrent services in zero-knowledge - Setty et al., 2018

Offload instruction execution.

Suppose, for each operation, we build a table containing all possible executions: $\mathbf{T_{OP}(x, y) = OP(x,y)}$ for all x, y.

1. Fetch instr.
2. Decode opcode, operands.
3. Lookup the instruction.
4. Update registers/RAM.



This table would be way too big. Most instructions take two operands, which leads to **2^{128} entries** for 64-bit operands.

But these tables are highly structured

We never have to materialize the tables because they each have a **succinct representation**.

Each operation's output is an efficient-to-evaluate **multilinear polynomial** over its input bits.

Why is this exciting?

Because polynomials are the language of SNARK backends.

Some examples:

$$T_{\mathsf{XOR}}(x,y) = \sum_{i=0}^{63} 2^i\left(x_i \cdot y_i + (1-x_i)\cdot(1-y_i)\right)$$

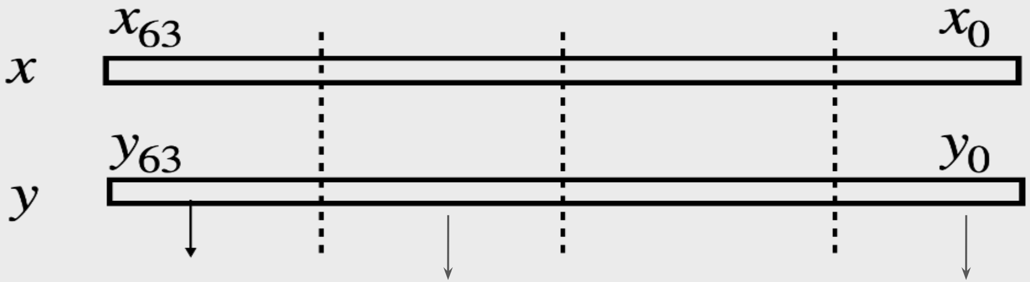
$$T_{\mathsf{LT}}(x,y) = \sum_{i=0}^{63} (1-x_i)\cdot y_i \cdot \widetilde{\mathsf{EQ}}(x_{>i}, y_{>i})$$

$$T_{\mathsf{SLL}}(x,y) = \sum_{k=0}^{63} \widetilde{\mathsf{EQ}}(y,k) \cdot \sum_{j=k}^{63} 2^j x_{j-k}$$

$$\widetilde{\mathsf{EQ}}(x,y) = \prod_{i\in n}(x_i \cdot y_i + (1-x_i)\cdot(1-y_i))$$

The tables can be decomposed even further.

Each table is a simple collation of smaller **subtables**, each represented by an MLE over a chunk of the original inputs.



c = 8 chunks, say

$$T(x, y) = g(\; g_c(\,..\,) \quad g_{c-1}(\,..\,) \quad \cdots \quad g_1(\,..\,) \;)$$

We only need **23 unique subtables** MLEs to represent **all RISC-V base** instructions.

Lasso efficiently looks up decomposed tables

Core tools: sum-checks and memory-checking. Built on Spark from Spartan.

(Setty19: ia.cr/2019/550)

Instruction lookup trace → **Lasso lookup argument** → π_{lookup}

- Operand chunks, subtable outputs.
- Memory-checking timestamps.
- Flags indicating the subtables used.

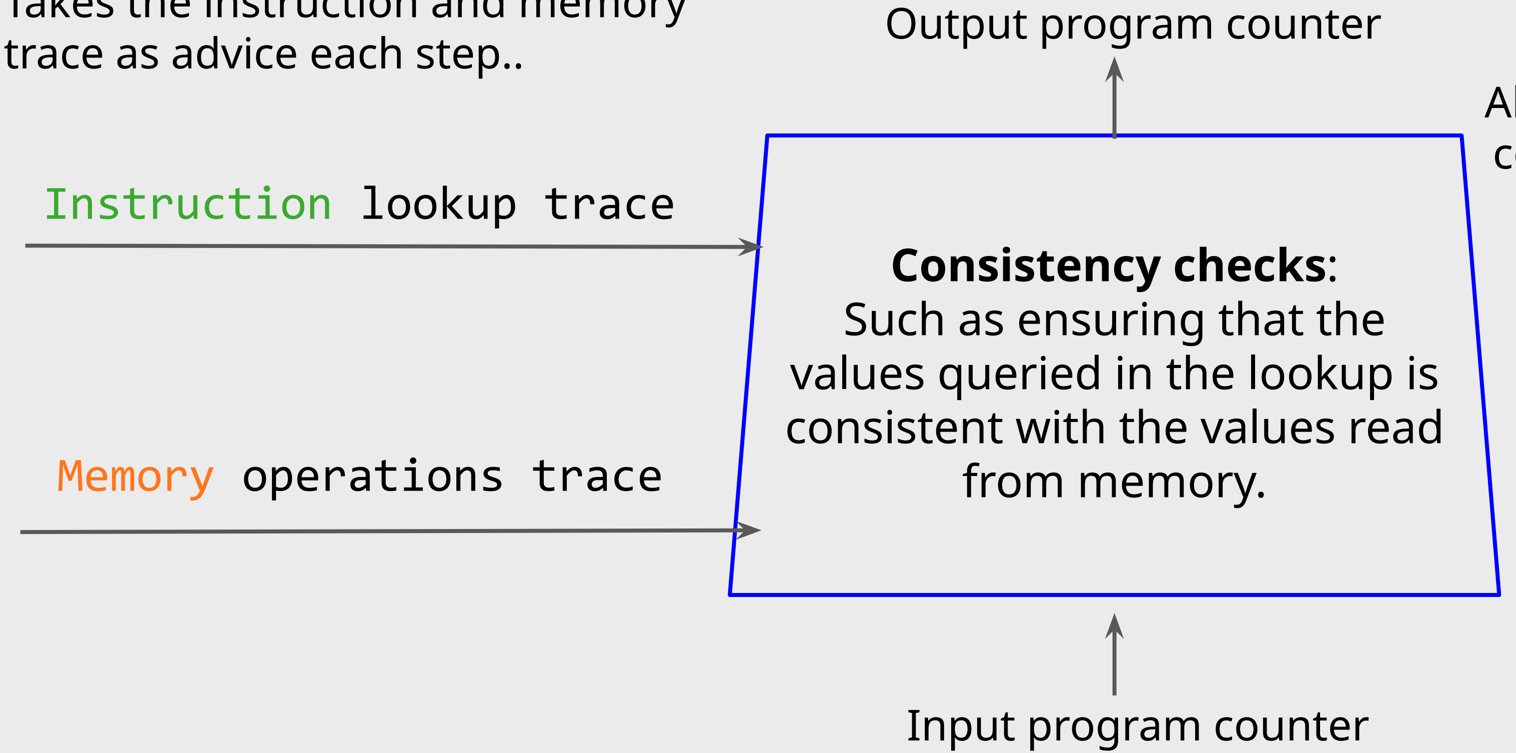
Performing m lookups into an N-sized table with c chunks = $O(cm + N^{1/c})$ prover commitment costs.

$N = 2^{128}$, $c = 8 \Rightarrow$ second term is 2^{16}

(Lasso - STW23: ia.cr/2023/1216)

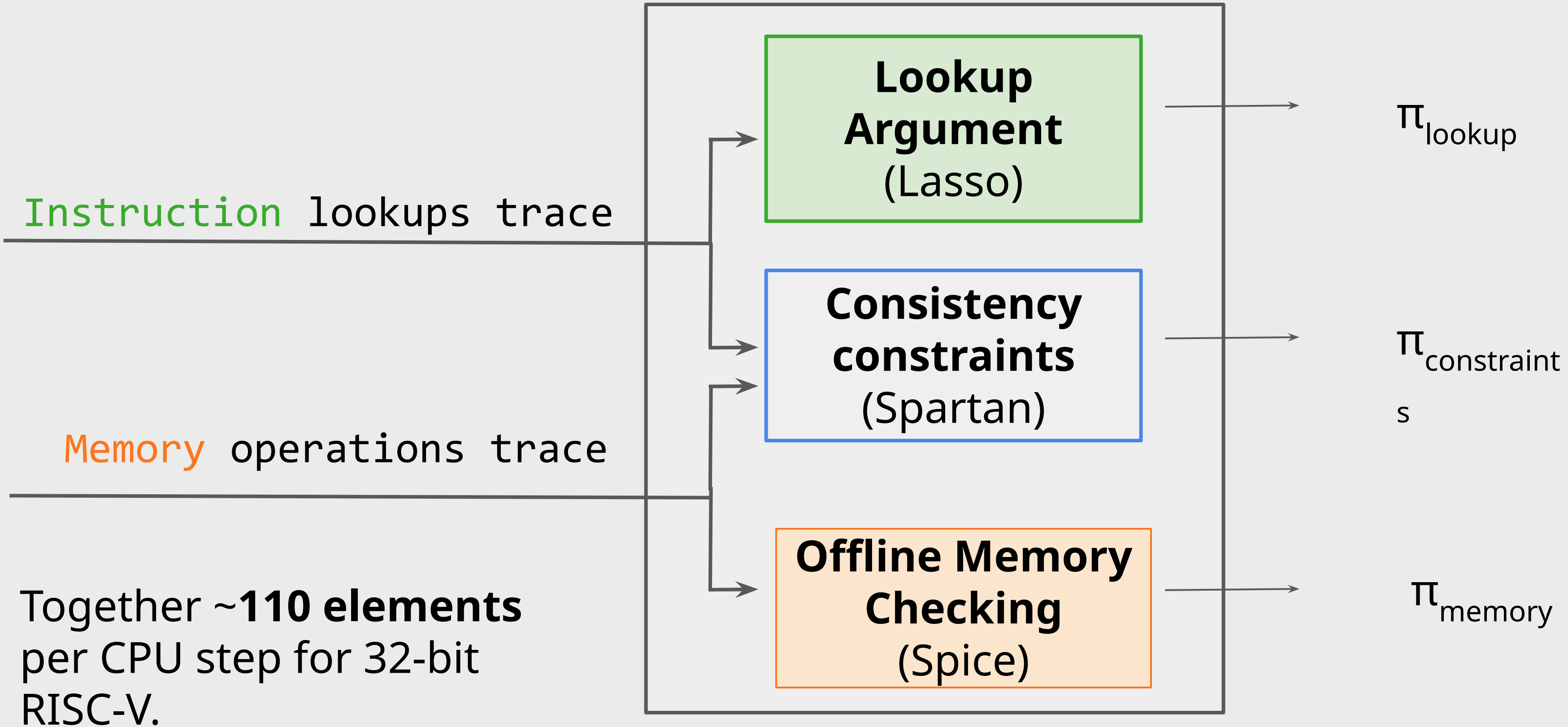
So what does the CPU circuit do now?

Takes the instruction and memory trace as advice each step..



About ~60 R1CS constraints per CPU step

Putting it all together: the Jolt prover modules



Together ~**110 elements** per CPU step for 32-bit RISC-V.

The Jolt prover's costs

Commitment costs: Using Hyrax. As most elements are small, it's equivalent to **11 arbitrary (**256-bit long) field elements per step when using Pippenger's MSM algorithm for 32-bit RISC-V.

Jolt's backend: just **sum-checks** and multilinear polynomial evaluations

Module	**Main operation**	**P cost**
Lookups (Lasso)	1 sum-check, 2 GKRs	$O(c^2 n)$
Constraints (Spartan)	2 sum-checks	$O(n)$
Memory-checking (Spice)	2 GKRs	$O(n + \|memory\|)$

Initial Jolt implementation

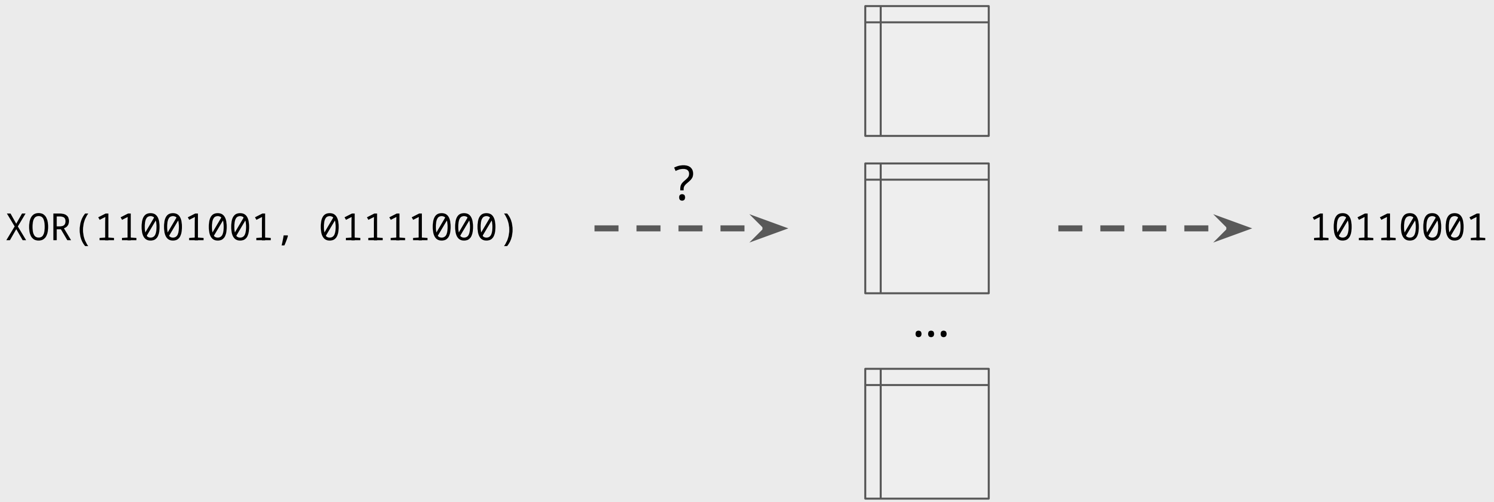
- Open source: https://github.com/a16z/jolt
- Instruction set: RISC-V 32-bit Base Integer Instruction Set (RV32I)
- Polynomial commitment scheme: Hyrax
- Elliptic curve: BN254 (interchangeable)
- Fork of Spartan2 (optimized to leverage uniformity) to prove R1CS

VM Instructions in Jolt

VM Instructions in Jolt

```
XOR(11001001, 01111000)
```

VM Instructions in Jolt

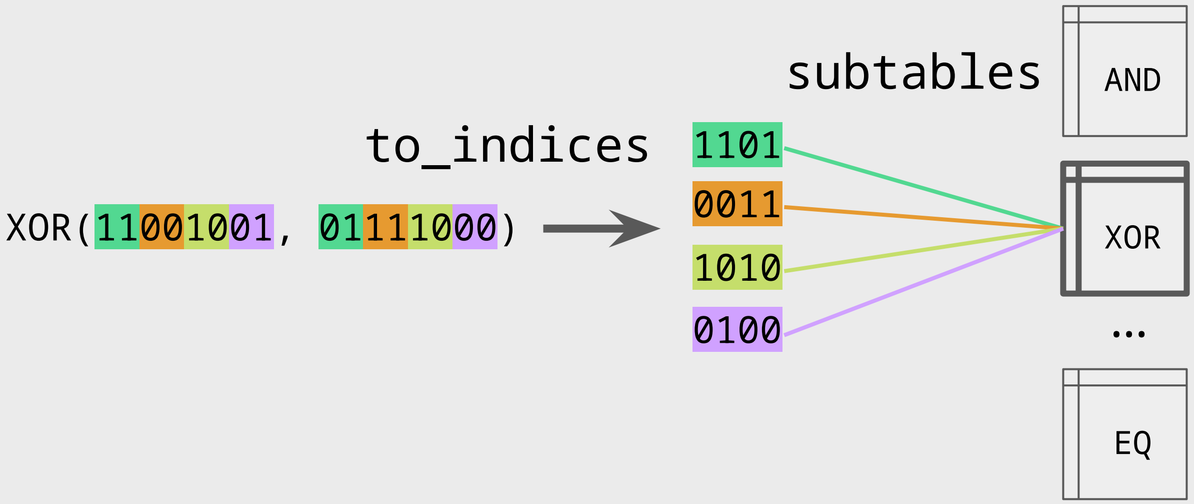


```
fn to_indices(&self, C: usize, log_M: usize) -> Vec<usize> {
    chunk_and_concatenate_operands(self.0, self.1, C, log_M)
}
```

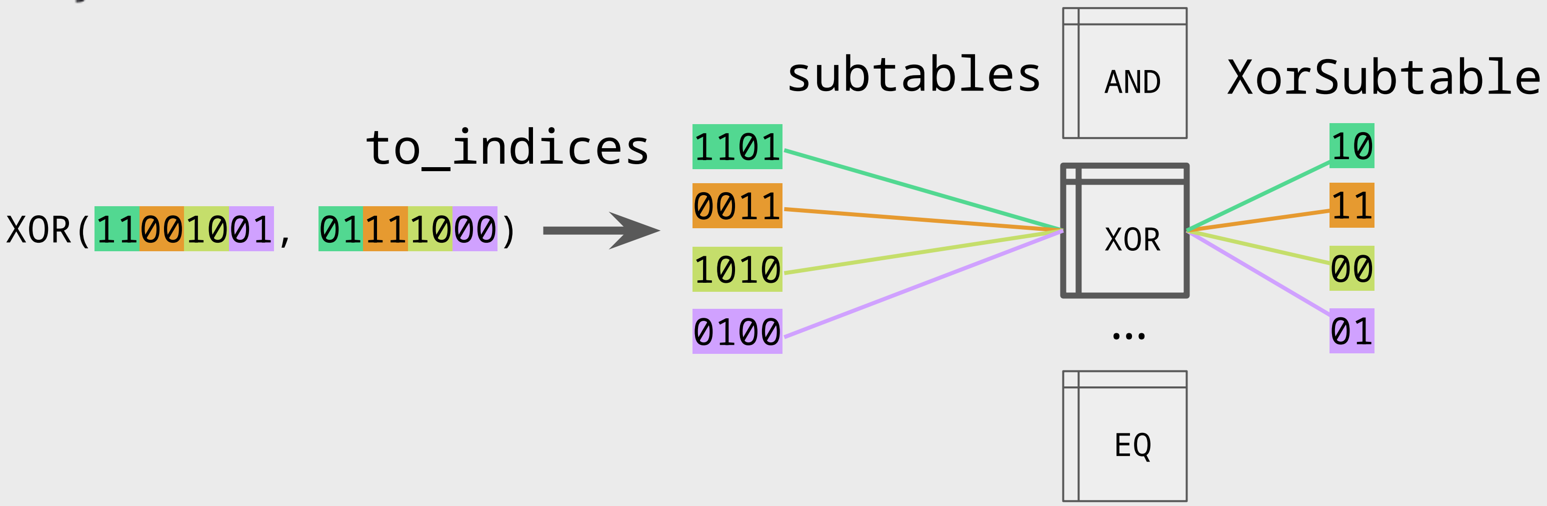
to_indices

XOR(11001001, 01111000) → 1101, 0011, 1010, 0100

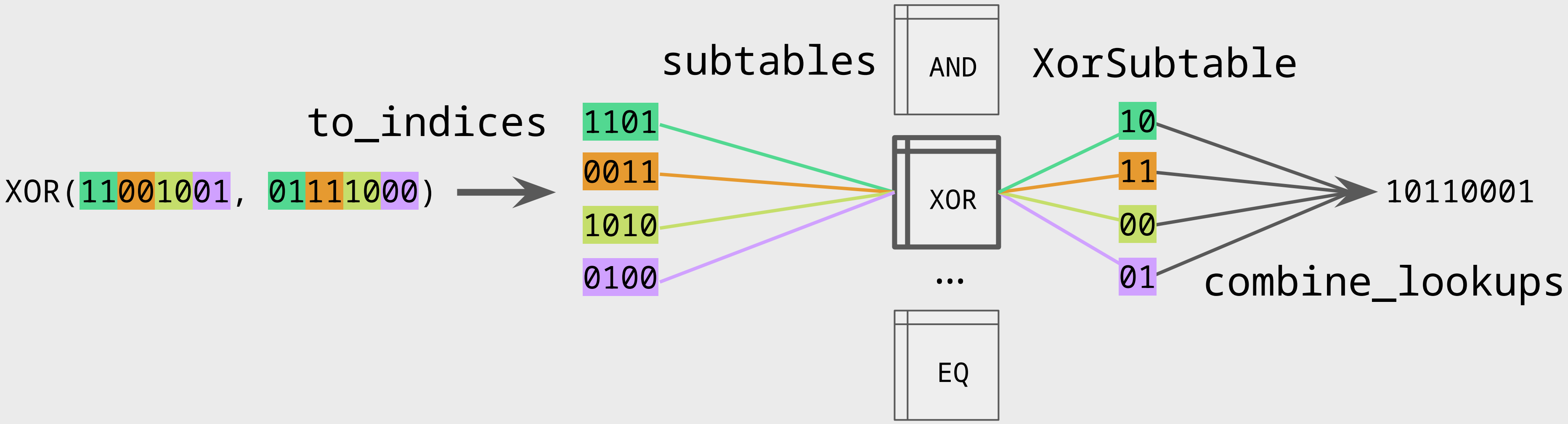
```rust
fn subtables<F: JoltField>(
    &self,
    C: usize,
    _: usize,
) -> Vec<(Box<dyn LassoSubtable<F>>, SubtableIndices)> {
    vec![(Box::new(XorSubtable::new()), SubtableIndices::from(0..C))]
}
```



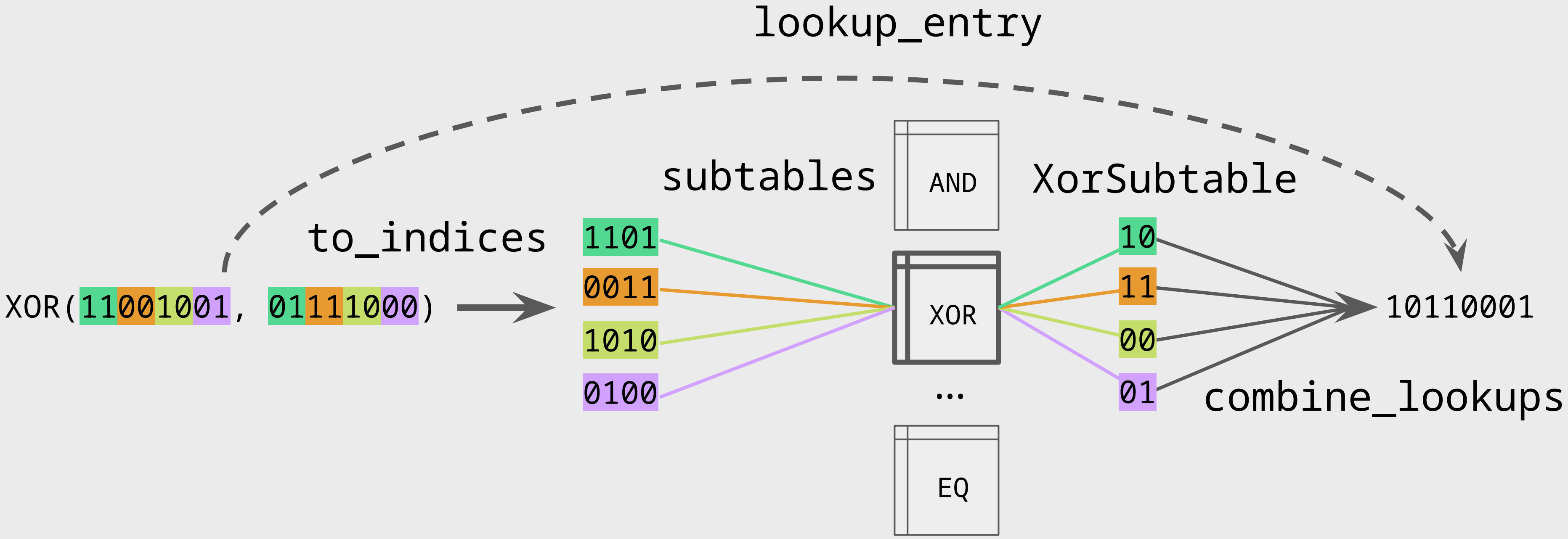
```rust
pub trait LassoSubtable<F: JoltField>: 'static + Sync {
    fn subtable_id(&self) -> SubtableId {
        TypeId::of::<Self>()
    }
    fn materialize(&self, M: usize) -> Vec<F>;
    fn evaluate_mle(&self, point: &[F]) -> F;
}
```

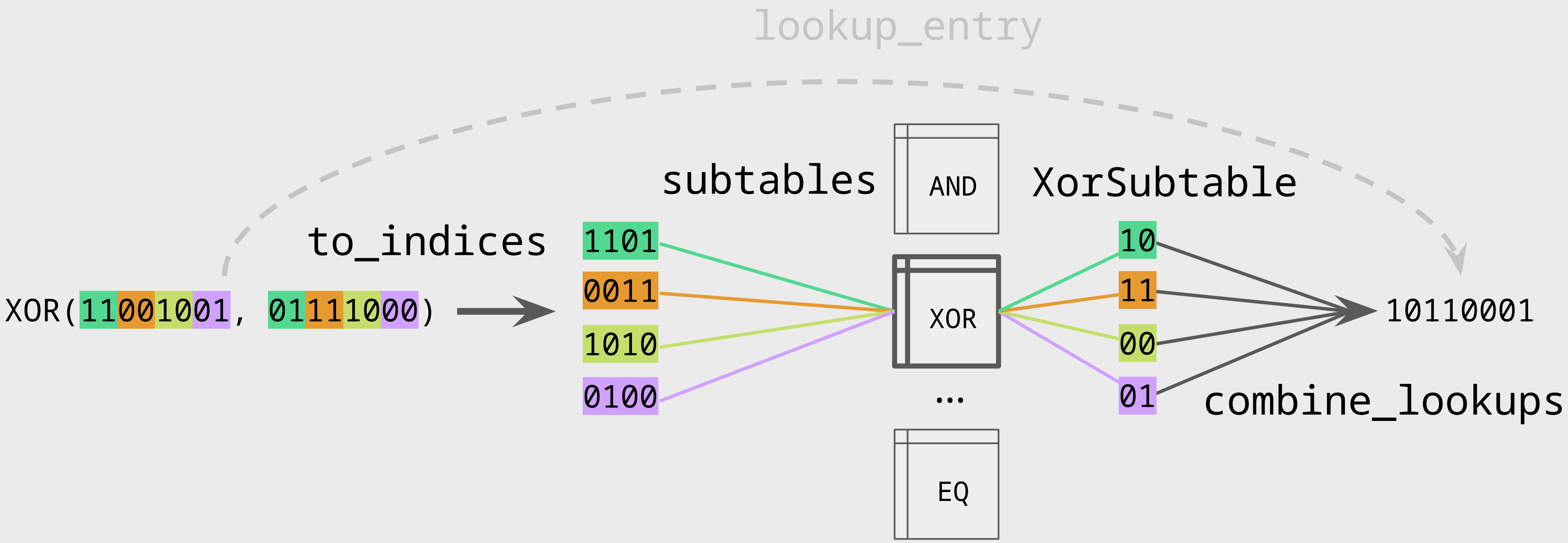


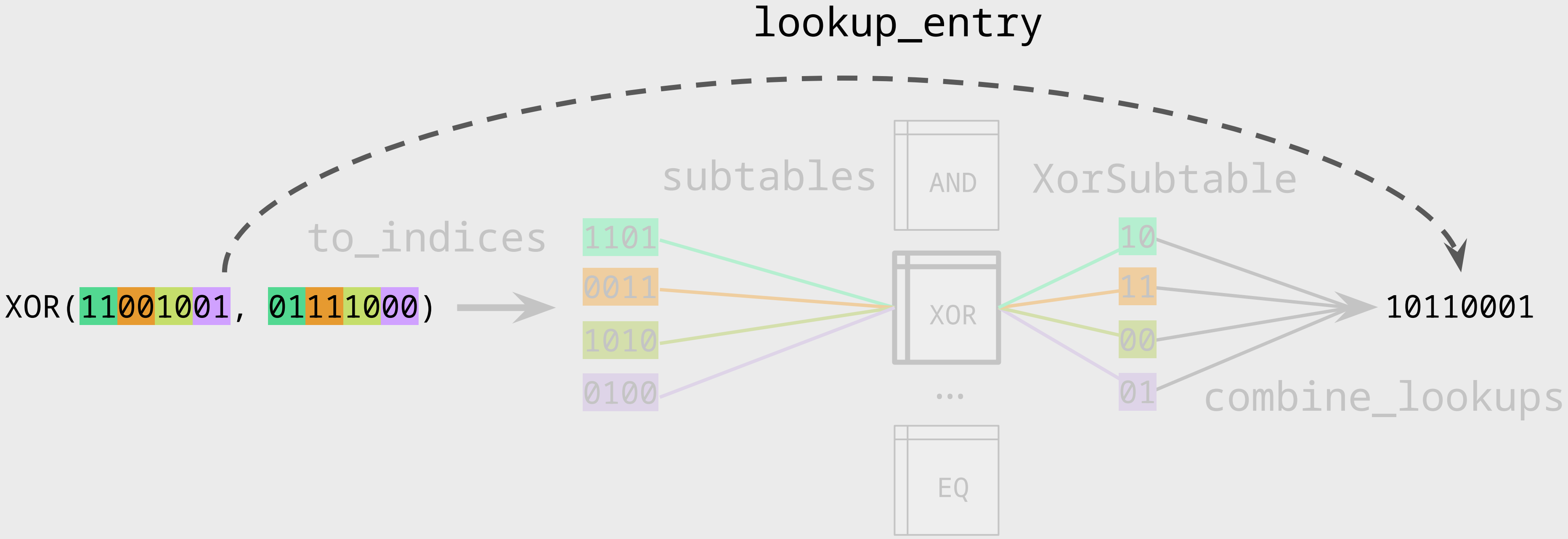
```rust
fn combine_lookups<F: JoltField>(&self, vals: &[F], C: usize, M: usize) -> F {
    concatenate_lookups(vals, C, log2(M) as usize / 2)
}
```



```
fn lookup_entry(&self) -> u64 {
    self.0 ^ self.1
}
```




lookup_entry
subtables
XorSubtable
to_indices
XOR(11001001, 01111000)
1101
0011
1010
0100
AND
XOR
...
EQ
10
11
00
01
10110001
combine_lookups


lookup_entry
subtables
AND
XorSubtable
to_indices
1101
0011
1010
0100
XOR(11001001, 01111000)
XOR
...
EQ
10
11
00
01
10110001
combine_lookups

Performance

Jolt proves ~100,000 RV32I instructions per second (**100 kHz**) on M3 Max Macbook Pro (16-core CPU, 128GB RAM)

M3 clock rate from Wikipedia

Performance

Jolt proves ~100,000 RV32I instructions per second (**100 kHz**) on M3 Max Macbook Pro (16-core CPU, 128GB RAM)

$$\text{Proving overhead} = \frac{\text{CPU clock speed}}{\text{Proving speed}}$$

M3 clock rate from Wikipedia

Performance

Jolt proves ~100,000 RV32I instructions per second (**100 kHz**) on M3 Max Macbook Pro (16-core CPU, 128GB RAM)

$$\text{Proving overhead} = \frac{\text{CPU clock speed}}{\text{Proving speed}} = \frac{12 \cdot 4.05 \text{ GHz} + 4 \cdot 2.75 \text{ GHz}}{100 \text{ kHz}}$$

M3 clock rate from Wikipedia

Performance

Jolt proves ~100,000 RV32I instructions per second (**100 kHz**) on M3 Max Macbook Pro (16-core CPU, 128GB RAM)

M3 Max “performance” cores M3 Max “efficiency” cores

$$\text{Proving overhead} = \frac{\text{CPU clock speed}}{\text{Proving speed}} = \frac{\boxed{12 \cdot 4.05\text{ GHz}} + \boxed{4 \cdot 2.75\text{ GHz}}}{100\text{ kHz}}$$

M3 clock rate from Wikipedia

Performance

Jolt proves ~100,000 RV32I instructions per second (**100 kHz**) on M3 Max Macbook Pro (16-core CPU, 128GB RAM)

M3 Max "performance" cores   M3 Max "efficiency" cores

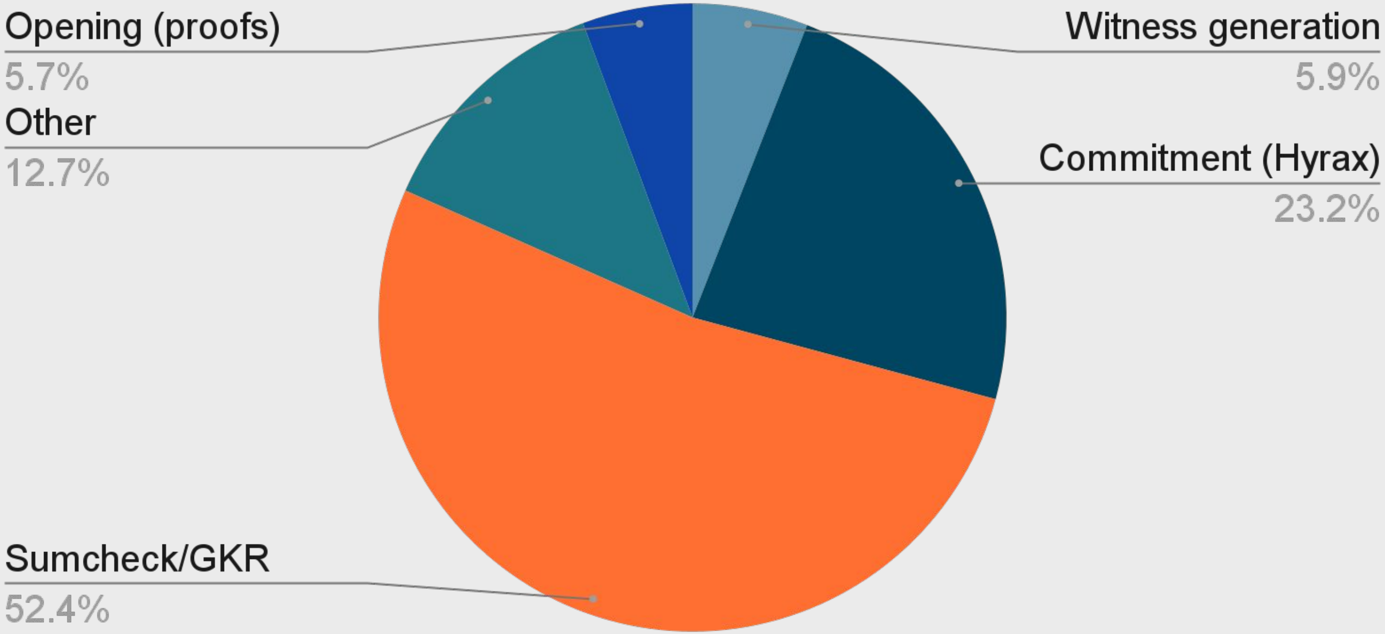
$$\text{Proving overhead} = \frac{\text{CPU clock speed}}{\text{Proving speed}} = \frac{\boxed{12 \cdot 4.05\text{ GHz}} + \boxed{4 \cdot 2.75\text{ GHz}}}{100\text{ kHz}}$$

= **596,000x** overhead

vs native CPU execution

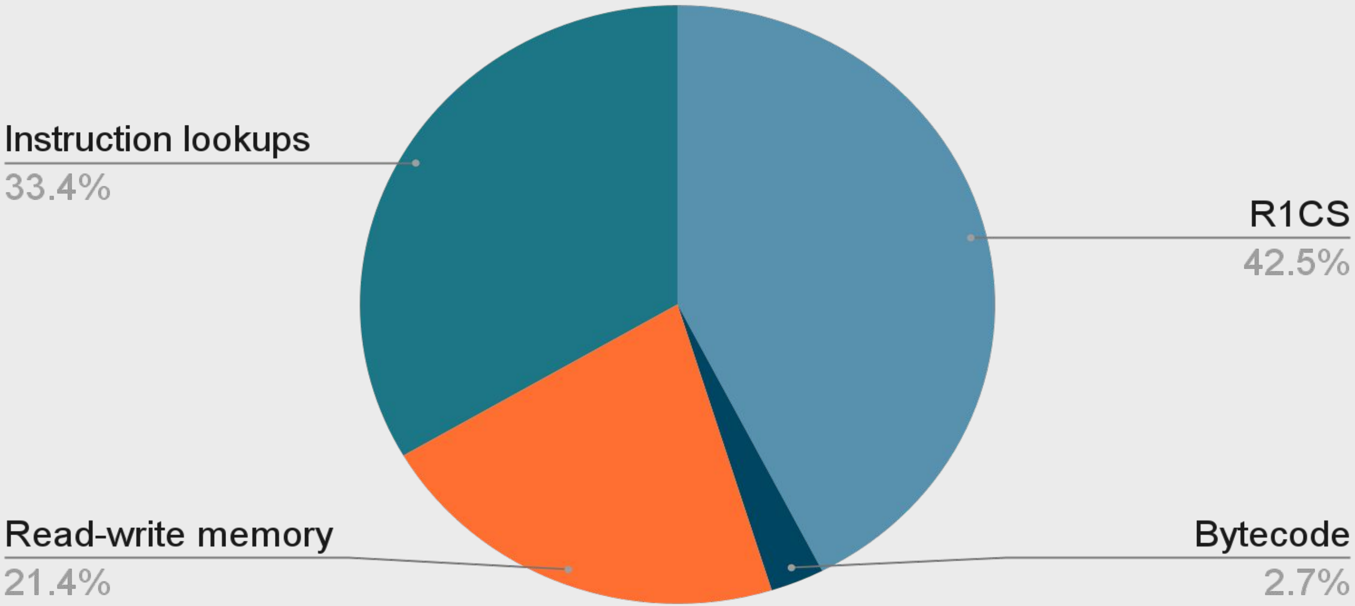
M3 clock rate from Wikipedia

Performance



Numbers from sha2-chain benchmark (3,632,556 RV32I cycles)

Performance



Numbers from sha2-chain benchmark (3,632,556 RV32I cycles)

What's next?

Jolt today

elliptic curves

good for EC cryptography,
e.g. ECDSA

Zeromorph

Folding

binary extension fields

good for general purpose

Binius

Self-recursion

Thank you!