Jolt:

SNARKs for VMs using lookups

Arasu Arun[1] Srinath Setty[2] Justin Thaler[3, 4]

NYU 1 Microsoft Research 2 Georgetown University 3 a16z crypto 4

Proofs of program execution

Prover's claim: Running program $\mathscr{P}$ on input x gives output y.

Verifier could re-execute the claim to check.

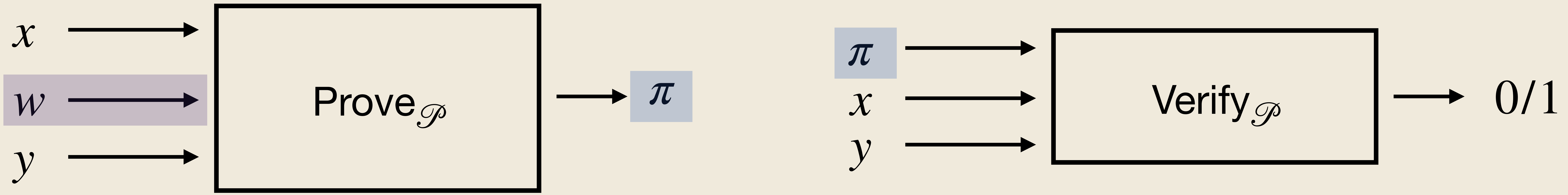
SNARKs convince the verifier far more efficiently.

Proofs of program execution

Prover's claim: Running program $\mathscr{P}$ on input x gives output y.

Verifier could re-execute the claim to check.

SNARKs convince the verifier far more efficiently.



Succinct = short, easy to check; verification often takes seconds or minutes

Non-interactive = just one proof that can be shared with anyone

Argument = computationally-sound

(Optional): Zero-knowledge = the verifier learns nothing about the advice w

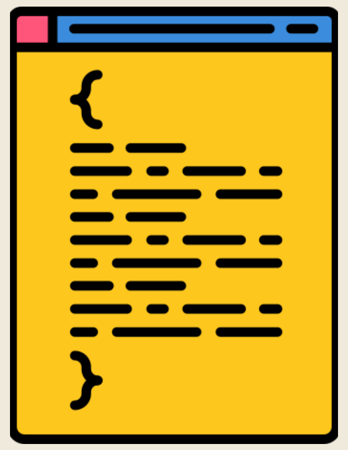
Building SNARKs: frontends and backends

Frontend

Converts program to a mathematical IR

Backend

Proves that the IR is satisfied on the given I/O.



Intermediate Representation

Proof π

Eg: C program

Think of this as an **arithmetic circuit** with wires and $+, \times$ gates over a finite field $\mathbb{F}$.

Think of this as a **Circuit-SAT** proof on the given I/O.

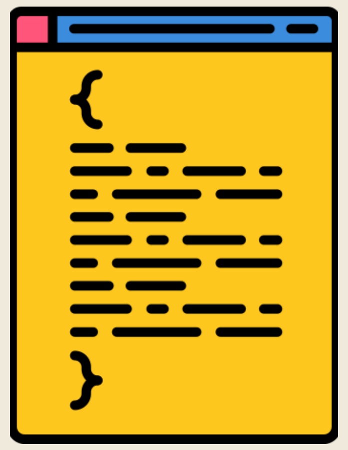
Building SNARKs: frontends and backends

Frontend

Converts program to a mathematical IR

Backend

Proves that the IR is satisfied on the given I/O.



Intermediate Representation

Proof π

Eg: C program

Think of this as an **arithmetic circuit** with wires and $+, \times$ gates over a finite field $\mathbb{F}$.

Think of this as a **Circuit-SAT** proof on the given I/O.

Eg: R1CS, Plonkish, AIR, CCS

Eg: GKR, GGPR, Groth16, Polynomial IOPs like Spartan, Plonk.

A primer on prover costs

Suppose the circuit has g **gates** and w **wires** .

Generally, a two-step process:

Steps	**Type**	**Factor**
1. **Commit** to wires (using a polynomial commitment scheme)	Group operations	$O(w)$
2. Run a probabilistic proof algorithm.	Field operations	$O(g + w)$

A primer on prover costs

Suppose the circuit has g **gates** and w **wires** .

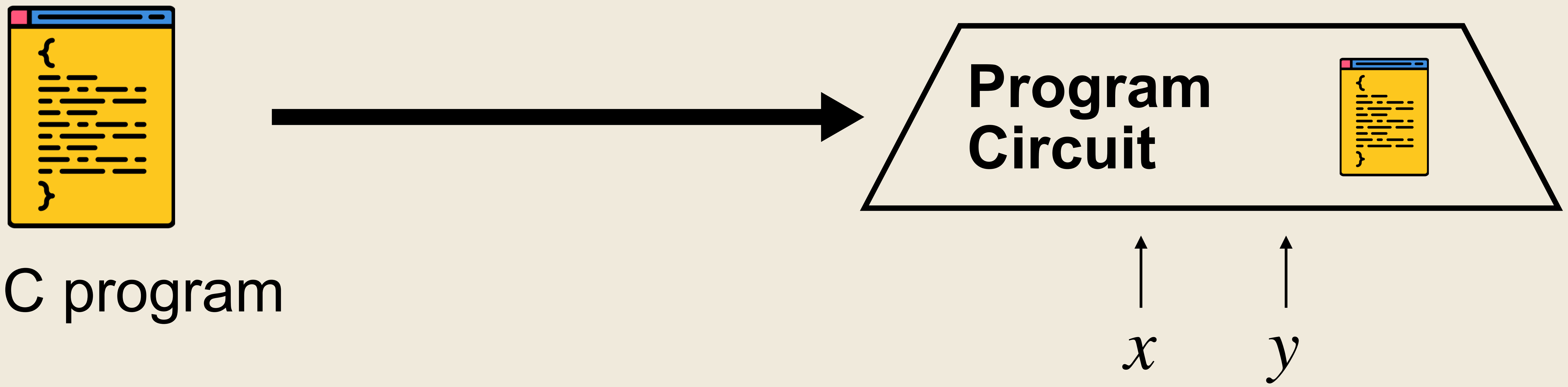
Generally, a two-step process:

Steps	**Type**	**Factor**
1. **Commit** to wires (using a polynomial commitment scheme)	Group operations	$O(w)$
2. Run a probabilistic proof algorithm.	Field operations	$O(g + w)$

The larger the circuit (especially the **wires**) the higher the prover cost.

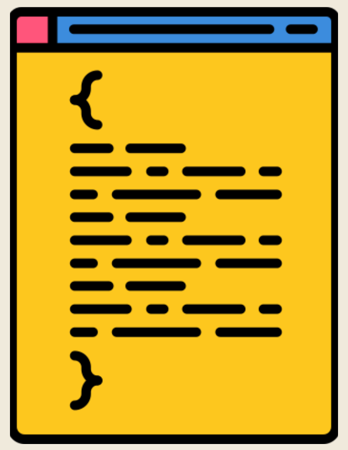
Two frontend approaches

Per-program approach: compiles each program into a new circuit.

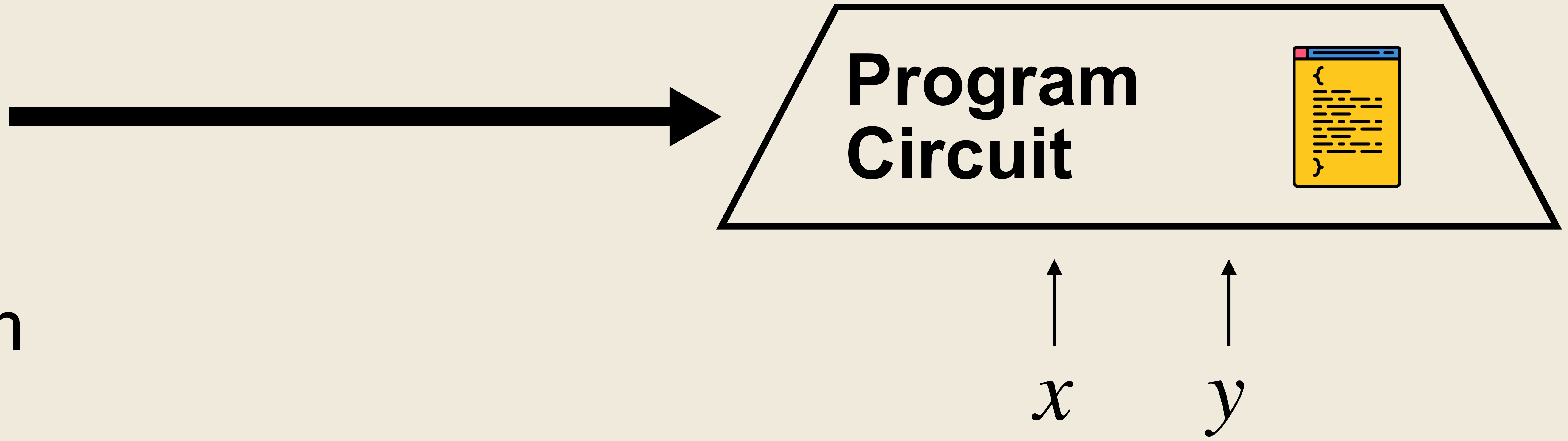


Two frontend approaches

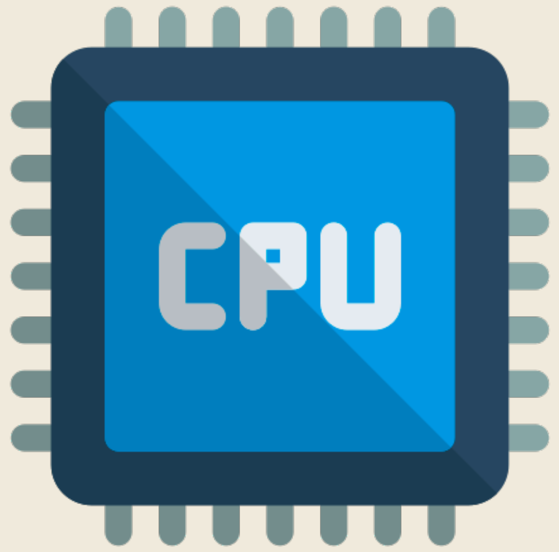
Per-program approach: compiles each program into a new circuit.



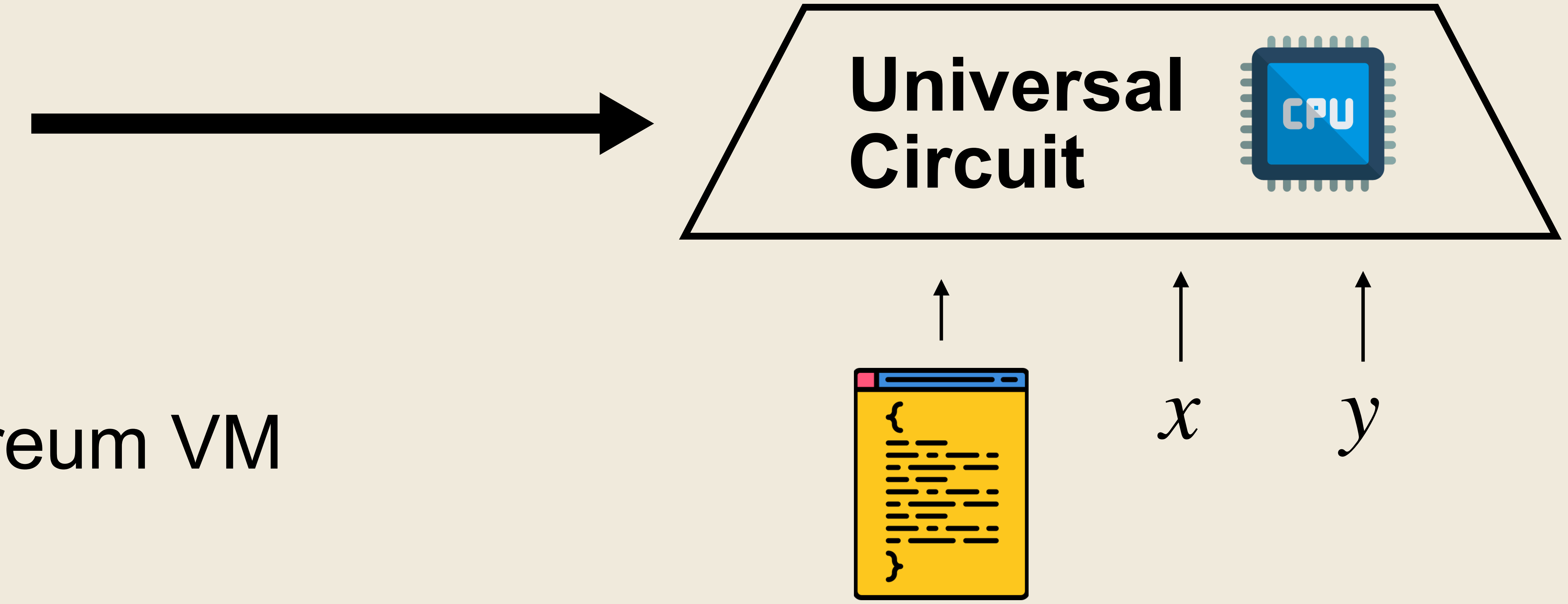
Eg: C program



Per-processor approach: a universal circuit that can take a class of programs as input.



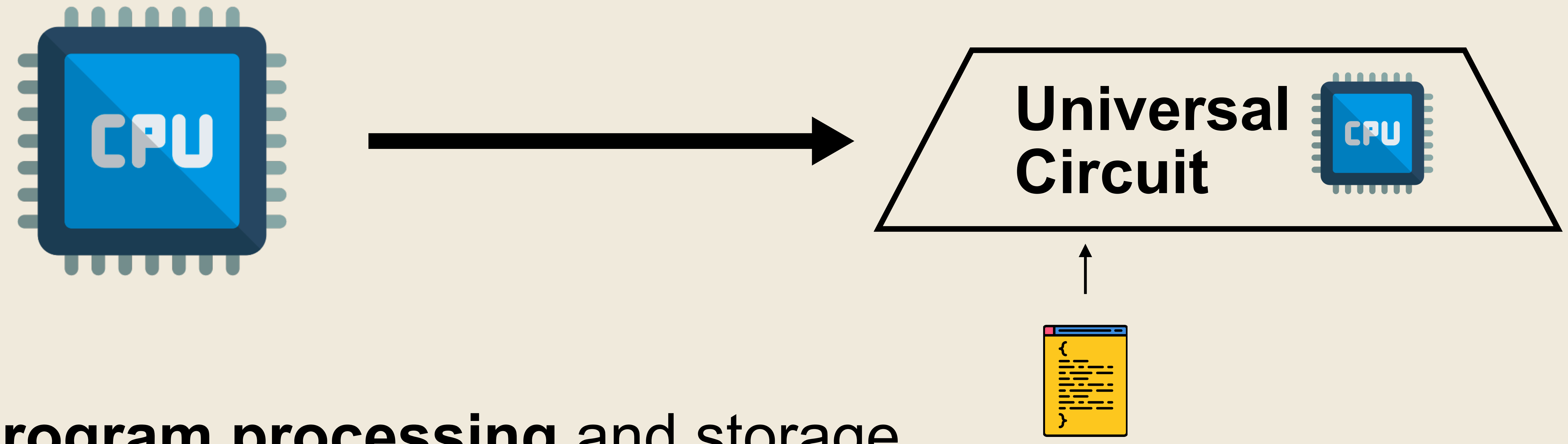
Eg: x86, RISC-V, Ethereum VM



Popularly referred to as “zkVMs”

Eg: RISC-V assembly program

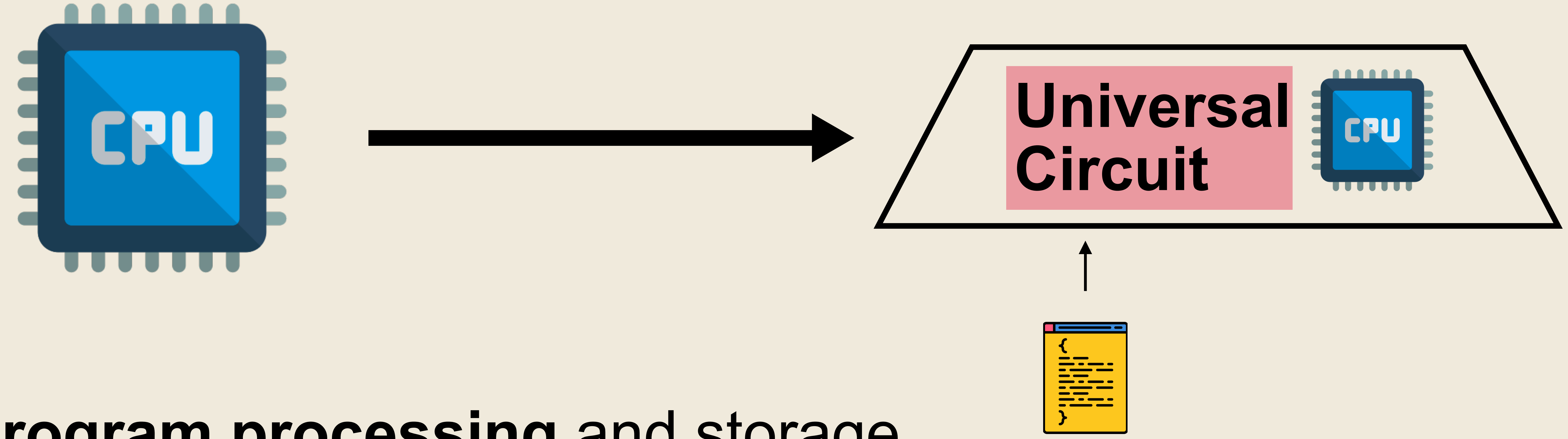
Advantages of the CPU approach



1. Avoids **per-program processing** and storage
2. **Programmability**: re-use existing languages, compilers and tooling.
3. Focus **auditing** and formal verification efforts into one circuit.

Vital for developing and deploying SNARKs.

Advantages of the CPU approach



1. Avoids **per-program processing** and storage
2. **Programmability**: re-use existing languages, compilers and tooling.
3. Focus **auditing** and formal verification efforts into one circuit.

Vital for developing and deploying SNARKs.

However… universal circuits are notoriously **large**, incurring proving time overheads compared to a circuit optimized for a given program.

Why are CPU circuits large?

1. **The cost of generality**: To handle arbitrary programs, CPU circuits must be able to execute any operation at a given step. This leads to a blowup in the gate/wire count.

RISC-V ≈ 50 operations.

Ethereum VM ≈ 140 operations.

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

   ...
  case SHIFT: {..}
}
```

A **switch-case** over the instruction set is emulated in the CPU circuit.

Why are CPU circuits large?

1. **The cost of generality**: To handle arbitrary programs, CPU circuits must be able to execute any operation at a given step. This leads to a blowup in the gate/wire count.

RISC-V ≈ 50 operations.

Ethereum VM ≈ 140 operations.

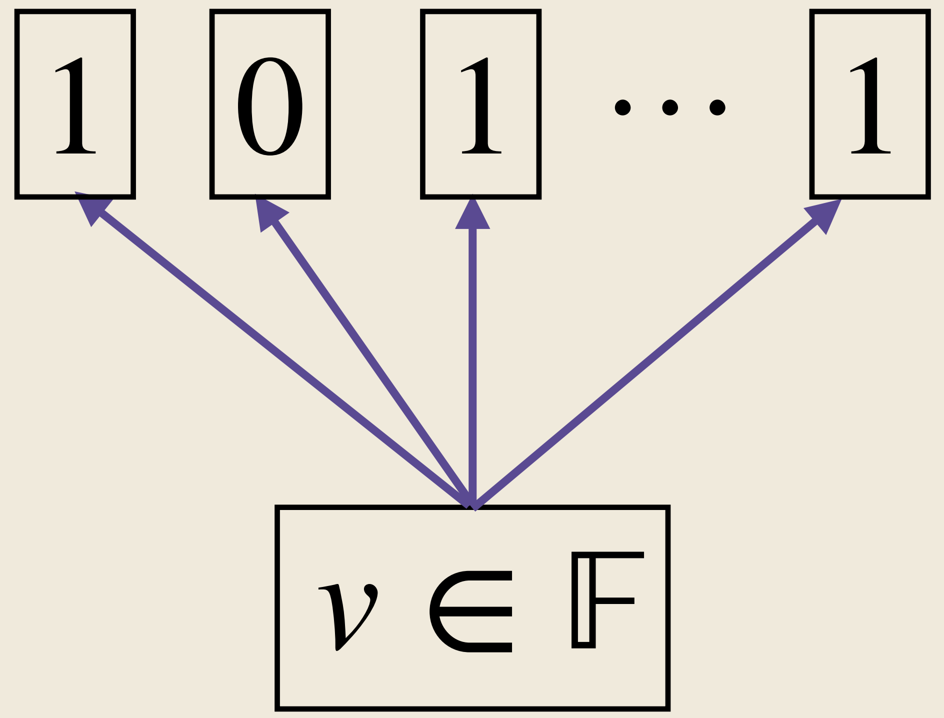
```
switch (instr) {
  case ADD: {..}
  case XOR: {..}
  ...
  (50 more)
   ...
  case SHIFT: {..}
}
```

A **switch-case** over the instruction set is emulated in the CPU circuit.

2. Instruction sets are designed to work with **bitwise operations**, which are costly to perform with field elements.

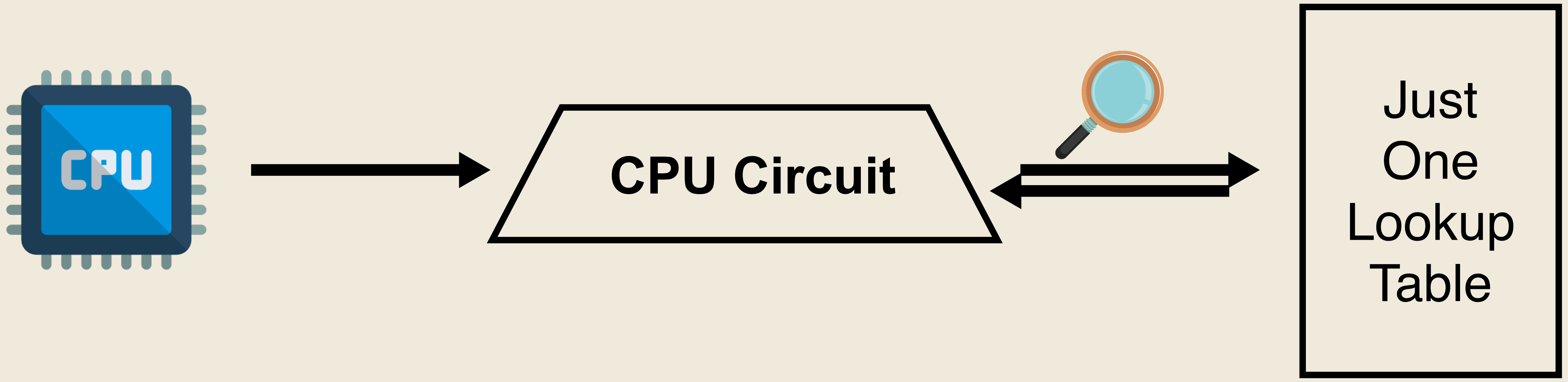
Require bit decompositions: 1 wire per bit of input.

XOR of two 32-bit values takes ≈ 100 gates and wires!

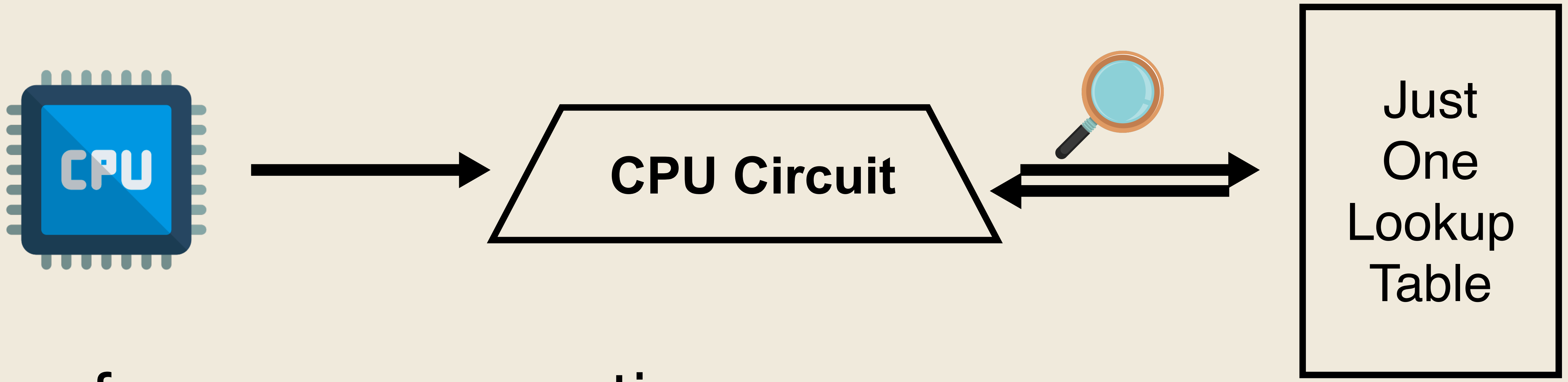


Decomposition of a field element.

This work: Jolt



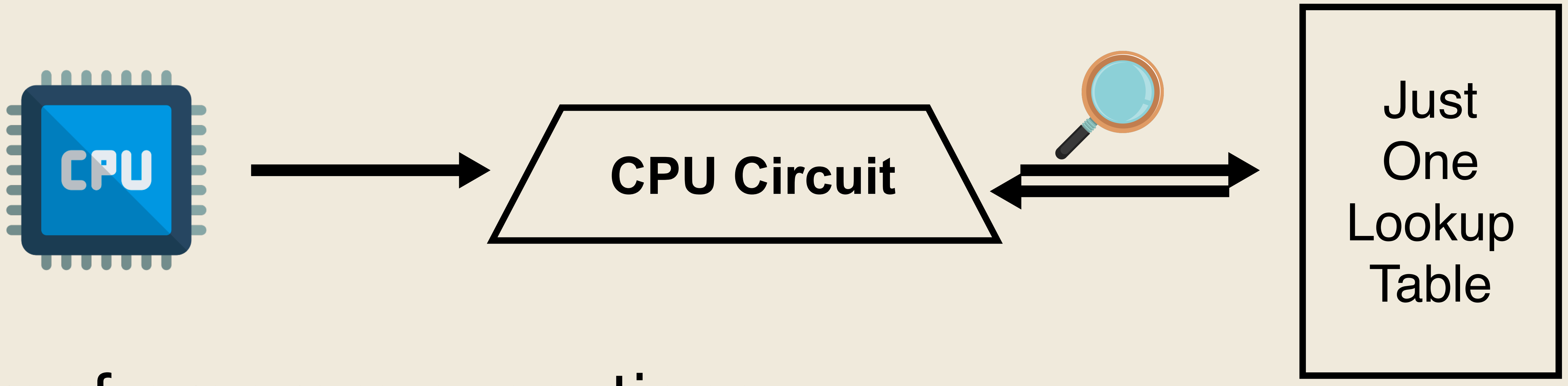
This work: Jolt



We design a new paradigm to efficiently proof program executions.

- Pay for only the instruction that is executed!
- Minimal circuit: just about 60 gates and 100 wires per step of RISC-V

This work: Jolt



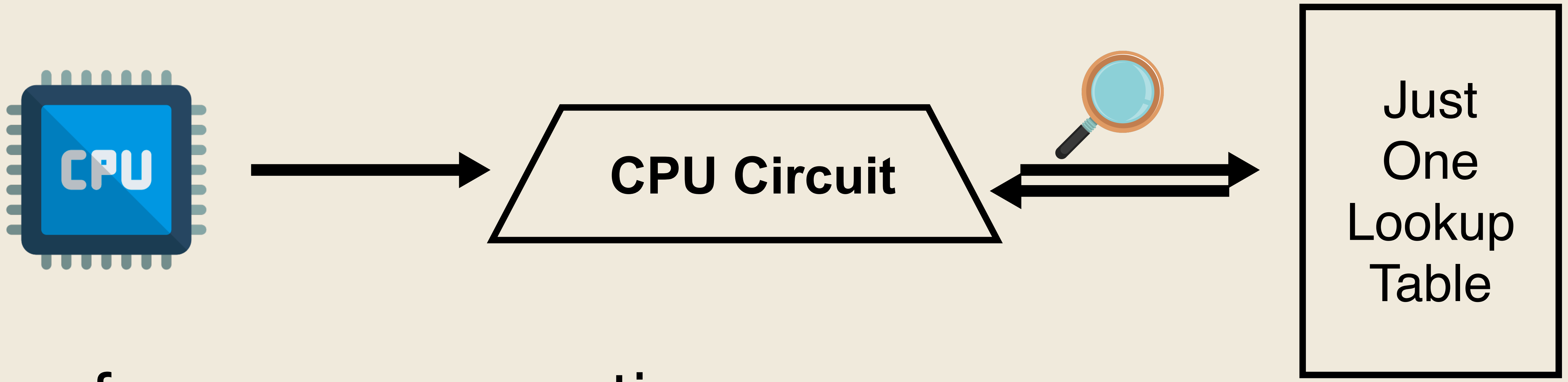
We design a new paradigm to efficiently proof program executions.

- Pay for only the instruction that is executed!
- <u>Minimal circuit</u>: just about 60 gates and 100 wires per step of RISC-V

How? **Offload** work outside of the circuit to more efficient arguments.

- Primitive assembly instructions have interesting **mathematical structure** (namely, efficient polynomial representations).
- We use this to design efficient “lookup arguments” for CPU instructions— namely, structured **Lasso**. Companion work: STW23 - ia.cr/2023/1216

This work: Jolt



We design a new paradigm to efficiently proof program executions.

- Pay for only the instruction that is executed!
- Minimal circuit: just about 60 gates and 100 wires per step of RISC-V

How? **Offload** work outside of the circuit to more efficient arguments.

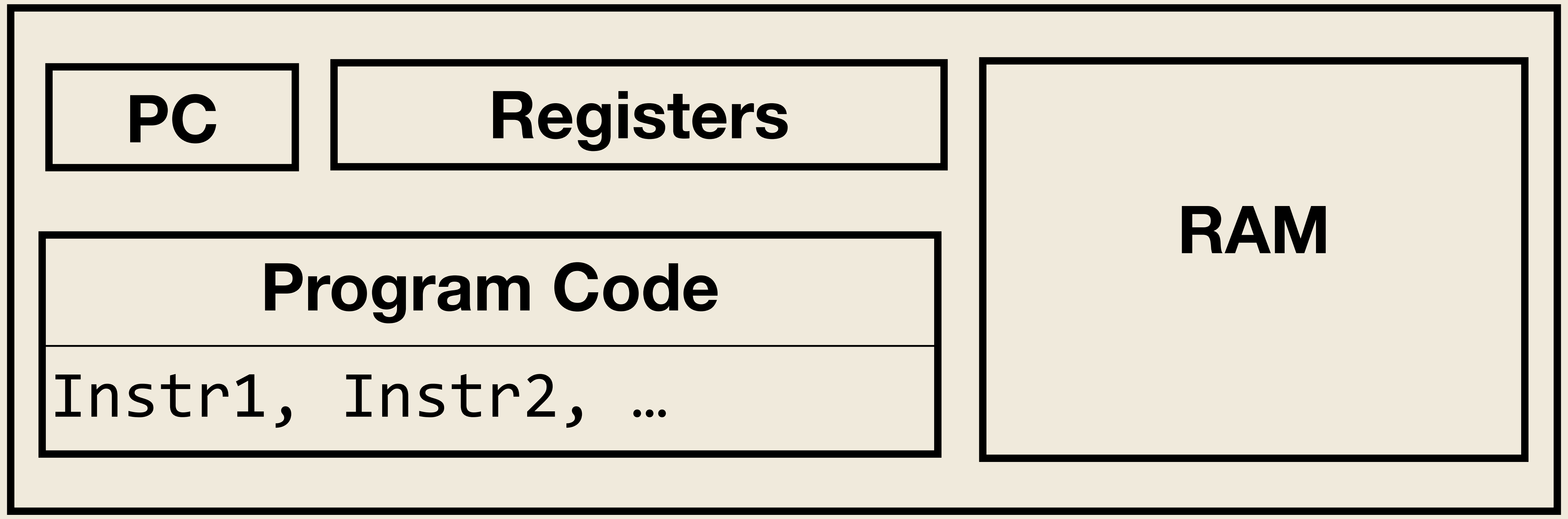
- Primitive assembly instructions have interesting **mathematical structure** (namely, efficient polynomial representations).
- We use this to design efficient "lookup arguments" for CPU instructions— namely, structured **Lasso**. Companion work: STW23 - ia.cr/2023/1216

Implemented this on the RISC-V processor.

- Achieve proving speeds of about **100 kHz** instrs/second on a MacBook.

Machine state and Transitions

Machine State

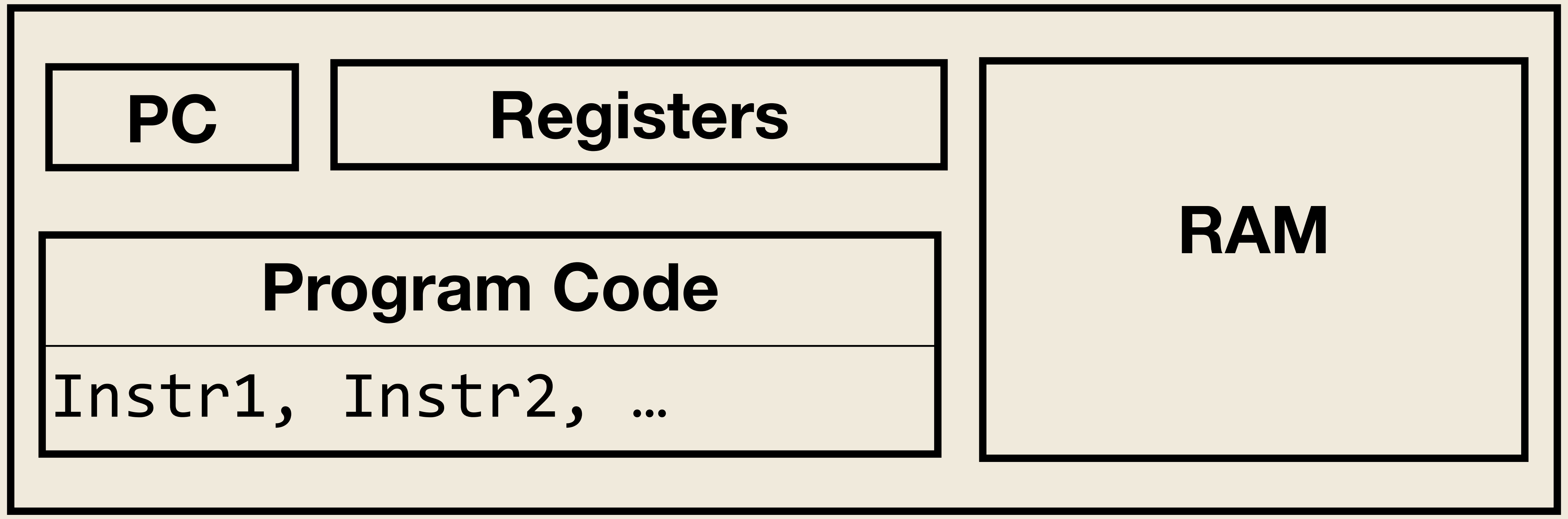


(Deterministic) Transition function

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

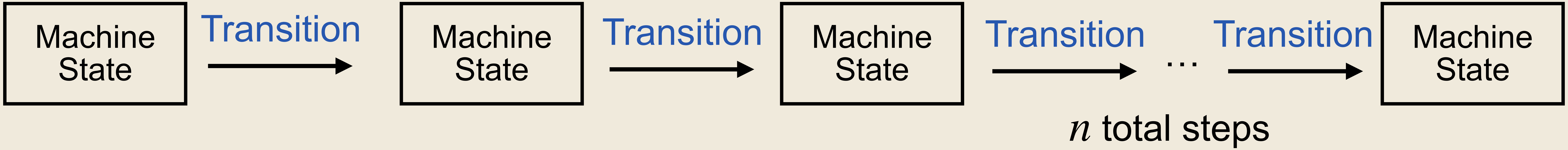
Machine state and Transitions

Machine State



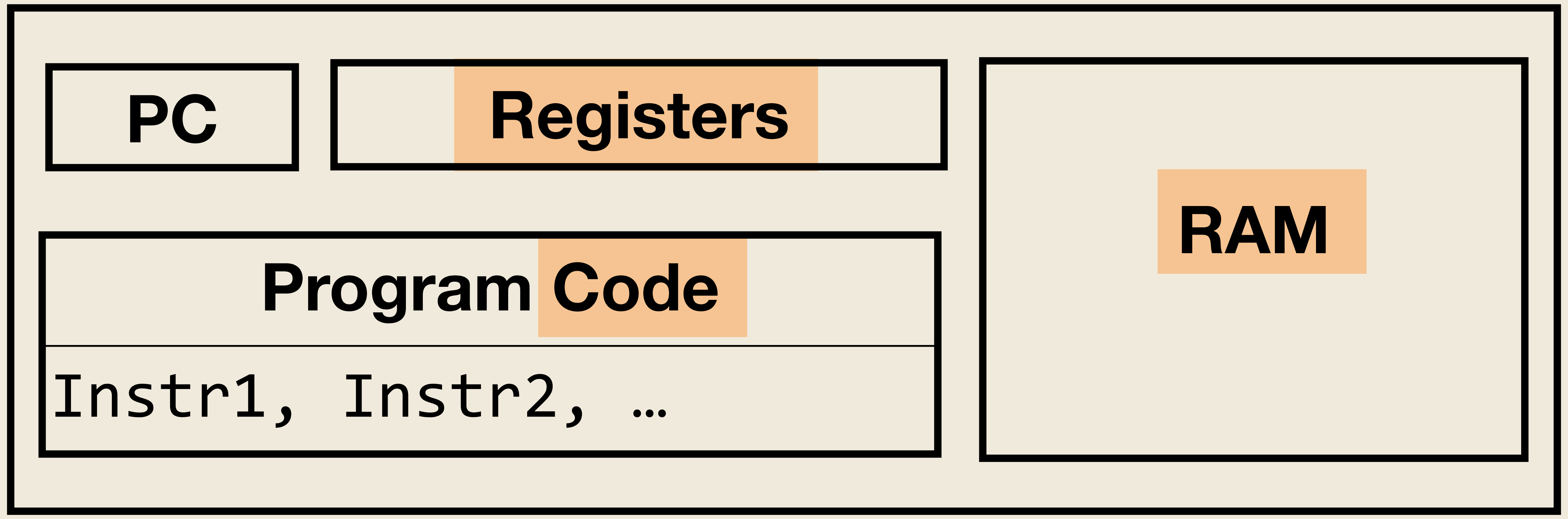
(Deterministic) Transition function

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



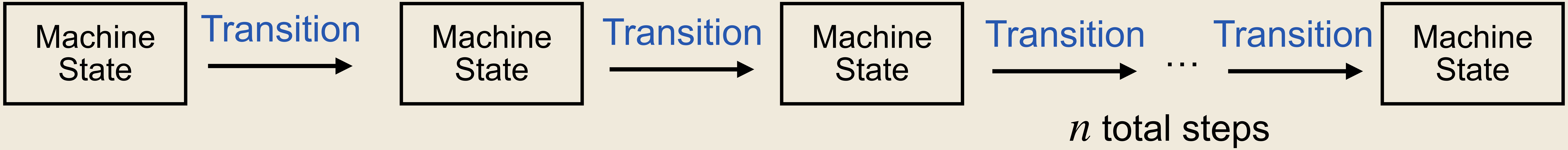
Machine state and Transitions

Machine State



(Deterministic) Transition function

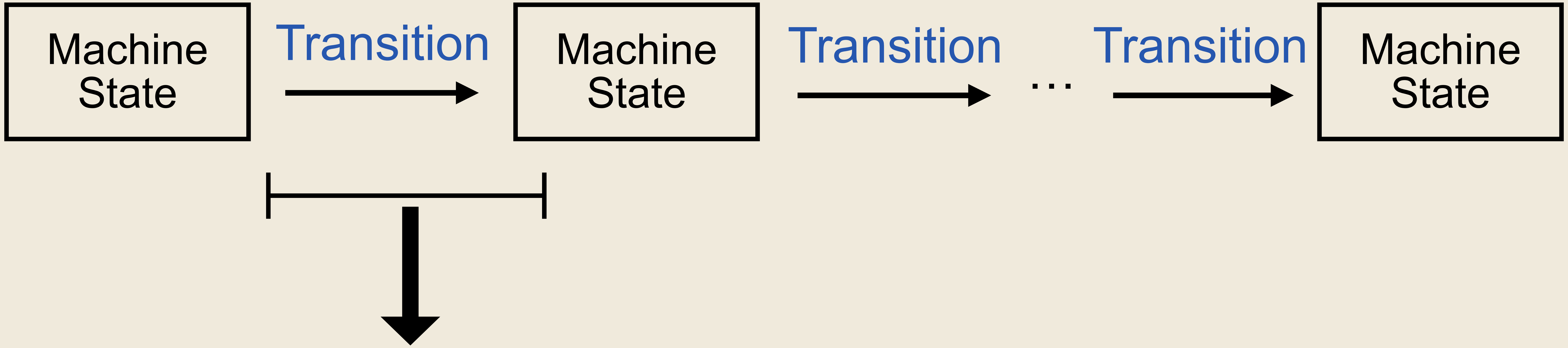
1. **Fetch** instr.
2. **Decode** opcode, operands.
3. **Execute** instruction.
4. **Update** registers



Each transition step consists of **memory accesses** and **instruction executions.**

Obtaining the execution trace

Prover executes the program and records the execution trace trace.



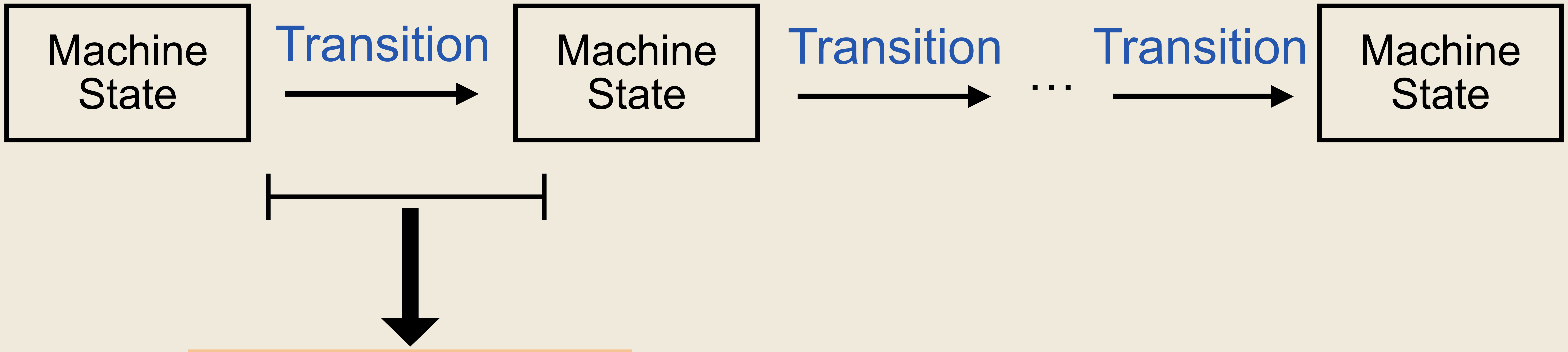
Each step consists of **memory operations** and **instruction logic**:

memory accesses : a vector of `(R/W, address, value)`

instruction exec. : `(operation, operands, output)`

Obtaining the execution trace

Prover executes the program and records the execution trace trace.



Each step consists of **memory operations** and **instruction logic**:

memory accesses : a vector of `(R/W, address, value)`

instruction exec. : `(operation, operands, output)`

Concatenate

After executing the whole program:

Trace of memory accesses.

Trace of instruction execs.

The Jolt proof modules

The Jolt proof modules

The Jolt proof modules

Memory-checking frontend

Online memory-checking: Design a circuit that maintains a commitment to the memory (e.g. **Merkle tree**) in a circuit. Verify reads and verifiably update after writes.

Produce a SNARK proof for this circuit and the given **memory access trace**.

Memory-checking frontend

Online memory-checking: Design a circuit that maintains a commitment to the memory (e.g. **Merkle tree**) in a circuit. Verify reads and verifiably update after writes.

Produce a SNARK proof for this circuit and the given **memory access trace**.

Expensive! Each cryptographic hash costs 100s of wires and gates.

Memory-checking frontend

[BEGKN91] - Checking the correctness of memories - Blum et al., 1991

[SAGL18] - Spice: Proving the correct execution of concurrent services in zero-knowledge - Setty et al., 2018

Online memory-checking: Design a circuit that maintains a commitment to the memory (e.g. **Merkle tree**) in a circuit. Verify reads and verifiably update after writes.

Produce a SNARK proof for this circuit and the given **memory access trace**.

Expensive! Each cryptographic hash costs 100s of wires and gates.

Offline memory checking [BEGKN91]. Adapted to SNARKs in Spice [SAGL18].

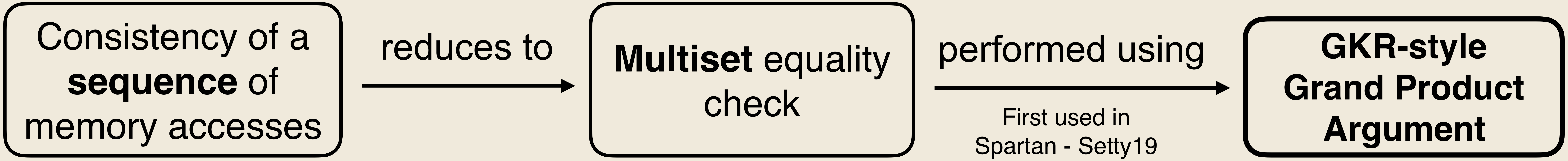
Consistency of a **sequence** of memory accesses → reduces to → **Multiset** equality check

Multiset hash algorithm:

1. Convert each memory access to a scalar with a **Reed-Solomon** fingerprint
2. Product of these scalars produces the multiset hash

With this method, each multiset hash costs only **3 gates** per memory access!

Memory-checking backend



Memory-checking backend

Consistency of a **sequence** of memory accesses

reduces to

Multiset equality check

performed using

First used in Spartan - Setty19

GKR-style Grand Product Argument

Trace of memory accesses

- vector of `(R/W, address, value)` tuples

Offline Memory Checking

π_{mem}

m accesses with memory size M:

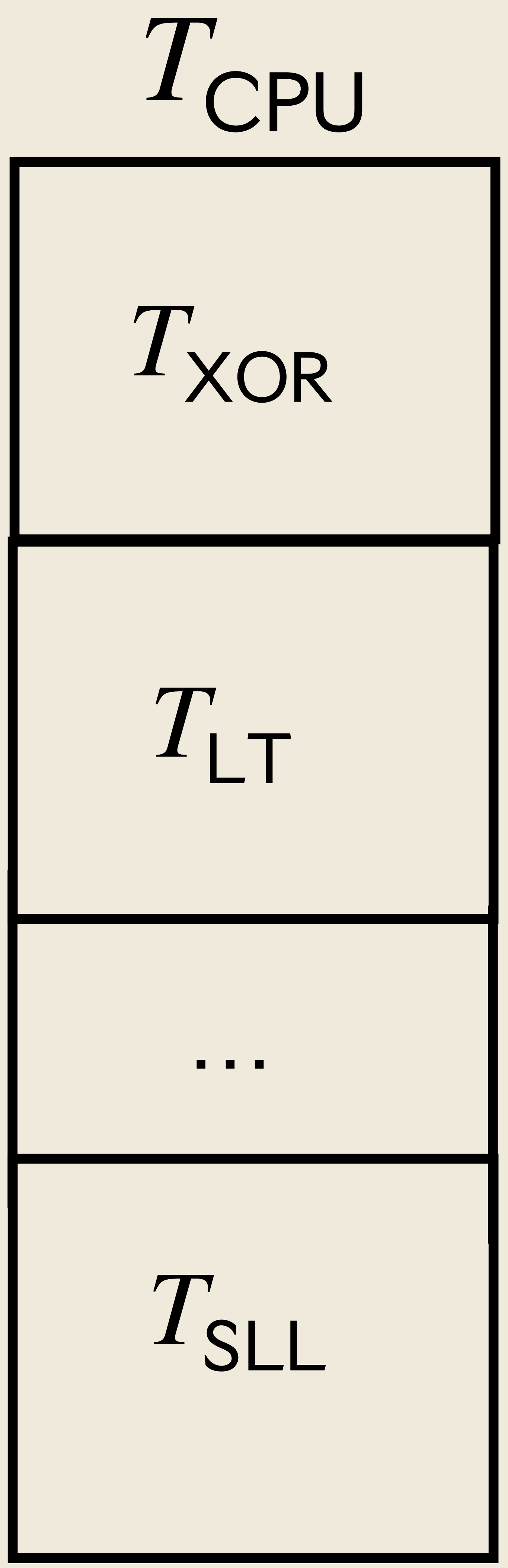
Prover complexity:
$\tilde{O}(m + M)$ field operations

Prover commits to the trace.

Up next: instruction execution

Instruction execution without circuits?

What if we had a **pre-processed table** with a list of all valid `(operation, operands, output)` combinations?



Instruction execution without circuits?

What if we had a **pre-processed table** with a list of all valid `(operation, operands, output)` combinations?

Trace of instruction execs

- Vector of `(operation, operands, output)` tuples

→ Prove that each exec. is in this pre-processed table T_{CPU} → π_{instr}

T_{CPU}

T_{CPU}
T_{XOR}
T_{LT}
…
T_{SLL}

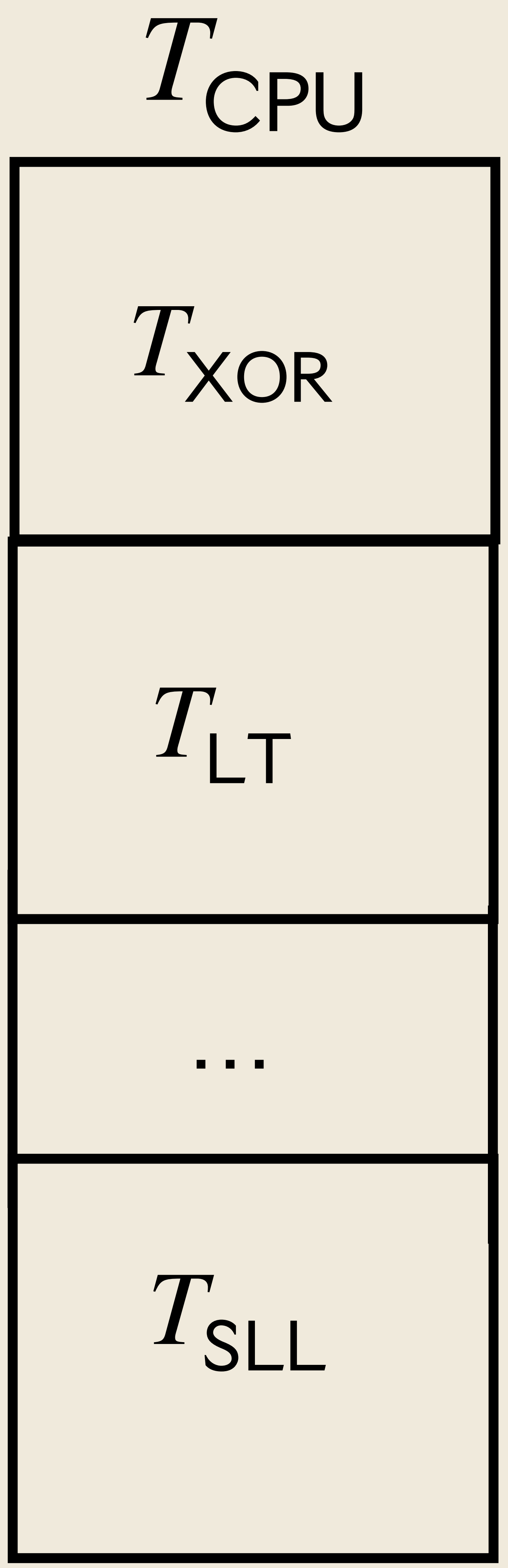
Instruction execution without circuits?

What if we had a **pre-processed table** with a list of all valid `(operation, operands, output)` combinations?

Trace of instruction execs

- Vector of `(operation, operands, output)` tuples

→ Prove that each exec. is in this pre-processed table T_{CPU} → π_{instr}



Lookup arguments are a class of protocols that do this.

Pre-processing + prover costs: usually between linear/quadratic in #ops, $|T|$.

But this table is **HUGE**, making these protocols infeasible.

Two W-bit operands $\Longrightarrow 2^{2W} = 2^{64}$ (32-bit) entries per instruction!

But these tables are highly **structured**.

We never have to materialize these tables because they each have some **succinct representation**.

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

** can be evaluated at a random point $r \in \mathbb{F}$ in $O(|\texttt{vars}|)$

But these tables are highly **structured**.

We never have to materialize these tables because they each have some **succinct representation**.

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

Let the operands be $x, y \in \{0,1\}^W$.

Some example tables are:

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

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

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

** can be evaluated at a random point $r \in \mathbb{F}$ in $O(|\texttt{vars}|)$

But these tables are highly **structured**.

We never have to materialize these tables because they each have some **succinct representation**.

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

Let the operands be $x, y \in \{0,1\}^W$.

Some example tables are:

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

Shift Left Logical: $T_{\text{SLL}}(x, y) = \sum_{k=0}^{W} \widetilde{\text{EQ}}(y, k) \cdot \sum_{j=k}^{W} 2^j x_{j-k}$

Less Than: $T_{\text{LT}}(x, y) = \sum_{i=0}^{W} (1 - x_i) \cdot y_i \cdot \widetilde{\text{EQ}}(x_{>i}, y_{>i})$

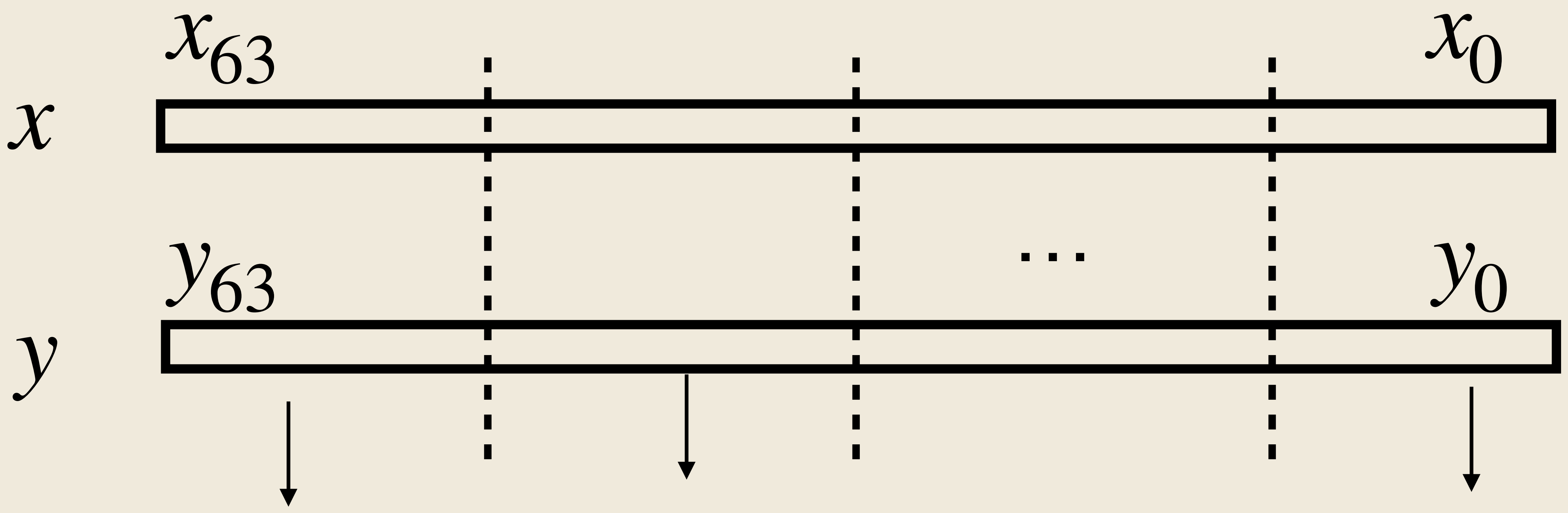
Why is this interesting?

Because polynomials are the **language** of SNARK backends!

** can be evaluated at a random point $r \in \mathbb{F}$ in $O(|\texttt{vars}|)$

The tables can be “decomposed” further

Each table’s output is a simple collation of **smaller subtable MLEs**, each over a chunk of the original inputs.

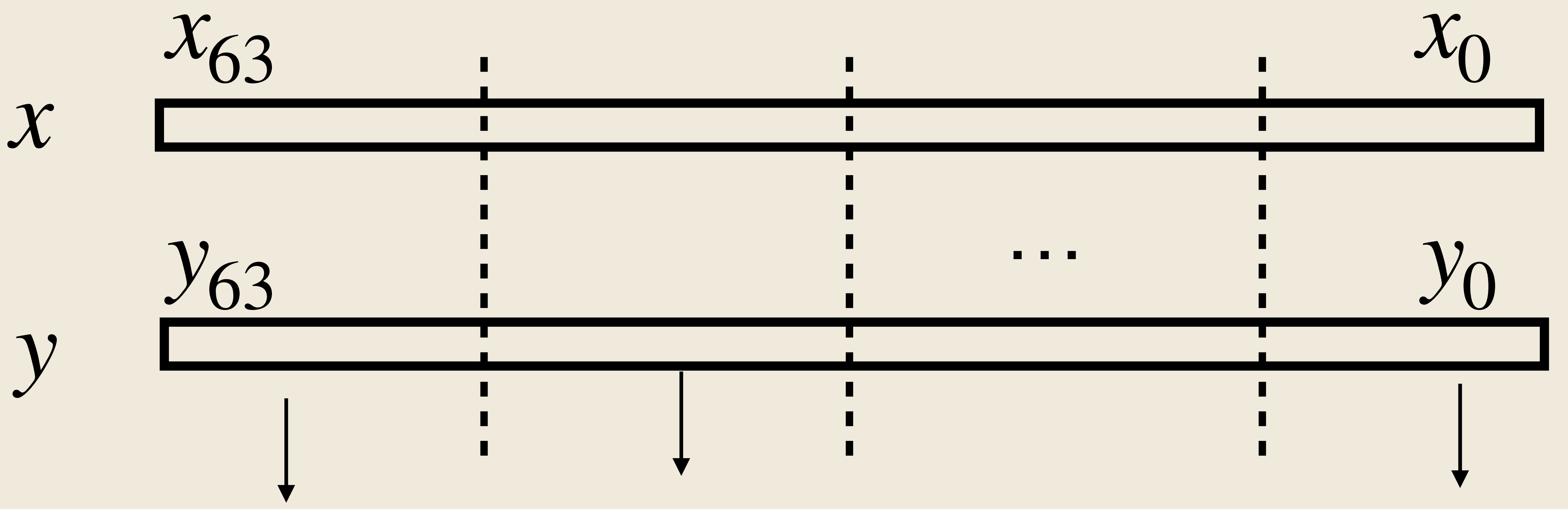


$c = 8$ chunks, say

$$T(x, y) = g(\; g_c(\;) \;, \;\; \cdots \;\;, \; g_2(\;) \;, \; g_1(\;)\;)$$

The tables can be “decomposed” further

Each table’s output is a simple collation of **smaller subtable MLEs**, each over a chunk of the original inputs.



$c = 8$ chunks, say

$$T(x, y) = g(\ g_c(\ )\ ,\ \cdots\ ,\ g_2(\ )\ ,\ g_1(\ )\ )$$

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

`AND, EQ, GT, LTU, OR, SIGN-EXTEND, SLL, SRL, TRUNCATE, ZERO-LSB, …`

Lasso efficiently looks up decomposed tables

[STW23] - Lasso: ia.cr/2023/1216

Core tools: sumchecks, offline memory-checking. Built on Spark from Spartan.

Setty19: ia.cr/2019/550

Trace of **instruction** executions →

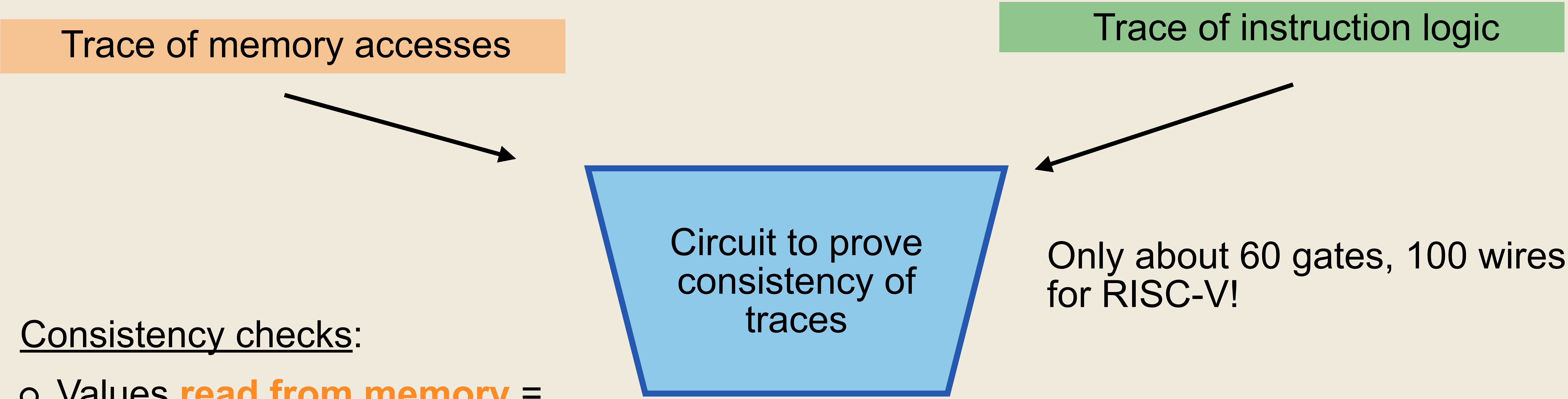
- Operand chunks, subtable outputs.
- Memory-checking advice.
- Flags indicating the subtables used.

Lasso Lookup Argument → π_{instr}

m lookups, c decomposed chunks $\Longrightarrow$ Prover cost is $3c \cdot (m + |T|^{1/c})$

$|T| = 2^{128}, c = 8 \Longrightarrow$ second term is 2^{16}

Proving consistency of traces



Consistency checks:

- Values **read from memory** = operands **looked up**.
- PC = address of instruction fetched in memory
- Check lookup query format (we have four types)
- …

Proving consistency of traces

Trace of memory accesses

Trace of instruction logic

Circuit to prove consistency of traces

Only about 60 gates, 100 wires for RISC-V!

SNARK backend

$\pi_{\text{consistency}}$

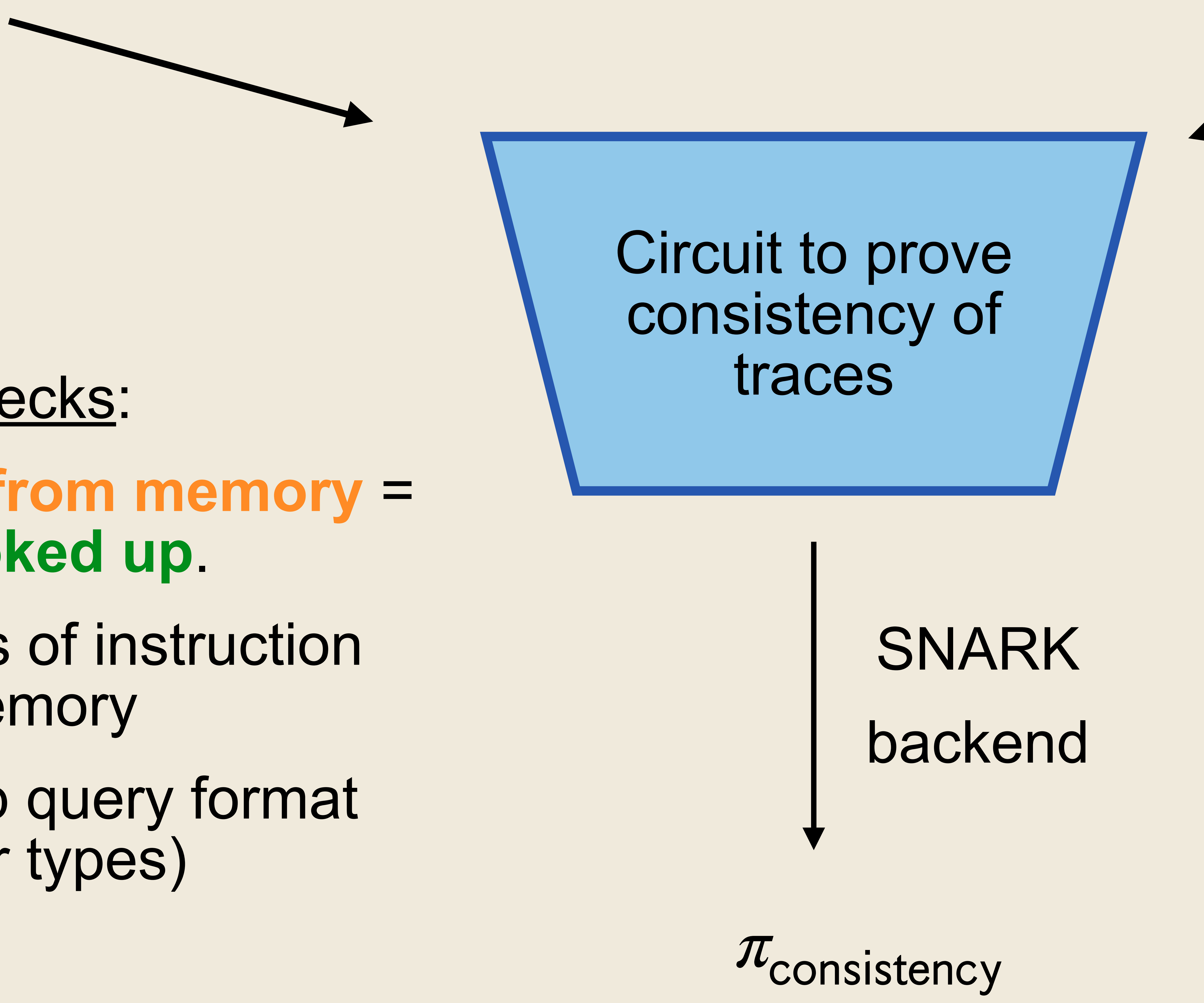
Consistency checks:

- Values **read from memory** = operands **looked up**.
- PC = address of instruction fetched in memory
- Check lookup query format (we have four types)
- …

Proving consistency of traces

Trace of memory accesses

Trace of instruction logic



Only about 60 gates, 100 wires for RISC-V!

<u>Consistency checks</u>:

- Values **read from memory** = operands **looked up**.
- PC = address of instruction fetched in memory
- Check lookup query format (we have four types)
- …

Highly **uniform** computation: repeated copies of the same circuit. Significantly improves proving and verification times.

- We use R1CS and Spartan

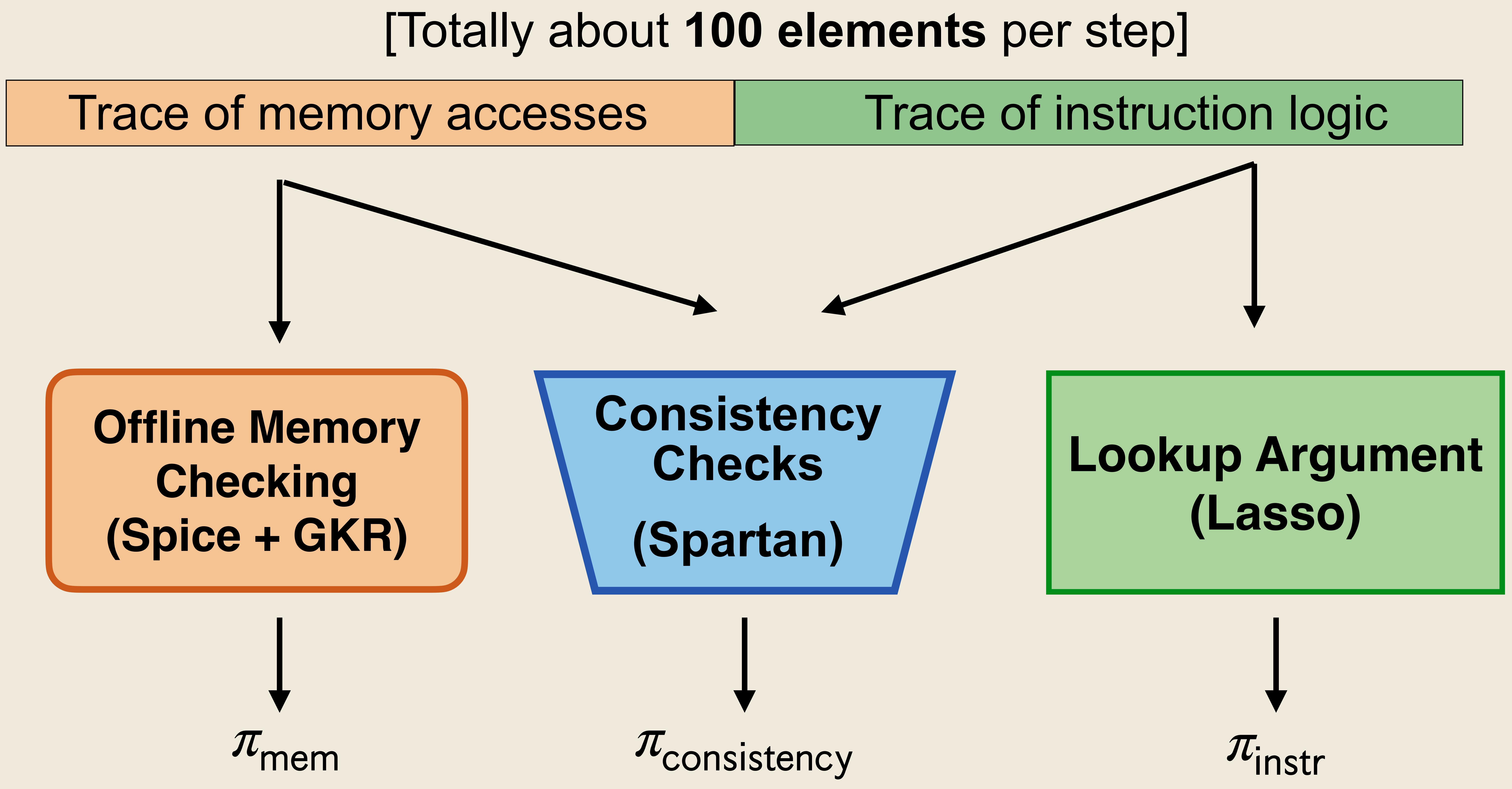
Setty19: ia.cr/2019/550

The final Jolt prover

CPU = RISC-V 32-bit Integer ISA

1. **Commit** to the traces.
   - We use the Hyrax scheme.

2. **Prover backend:**
   - Linear in the number of steps.
   - Entirely sumcheck + multi-linear polynomial evaluations.



Prover backend is **linear** in the number of CPU steps.

The Jolt prover's costs

CPU = RISC-V 32-bit Integer ISA

1. **Commitment costs** As most of the 100 elements are small, when using Hyrax with Pippenger's MSM algorithm, this is equivalent to committing to about **8 arbitrary (256-bit) $\mathbb{F}$ elems**.

2. **Prover backend** Just sumchecks and multi-linear polynomial evaluations.

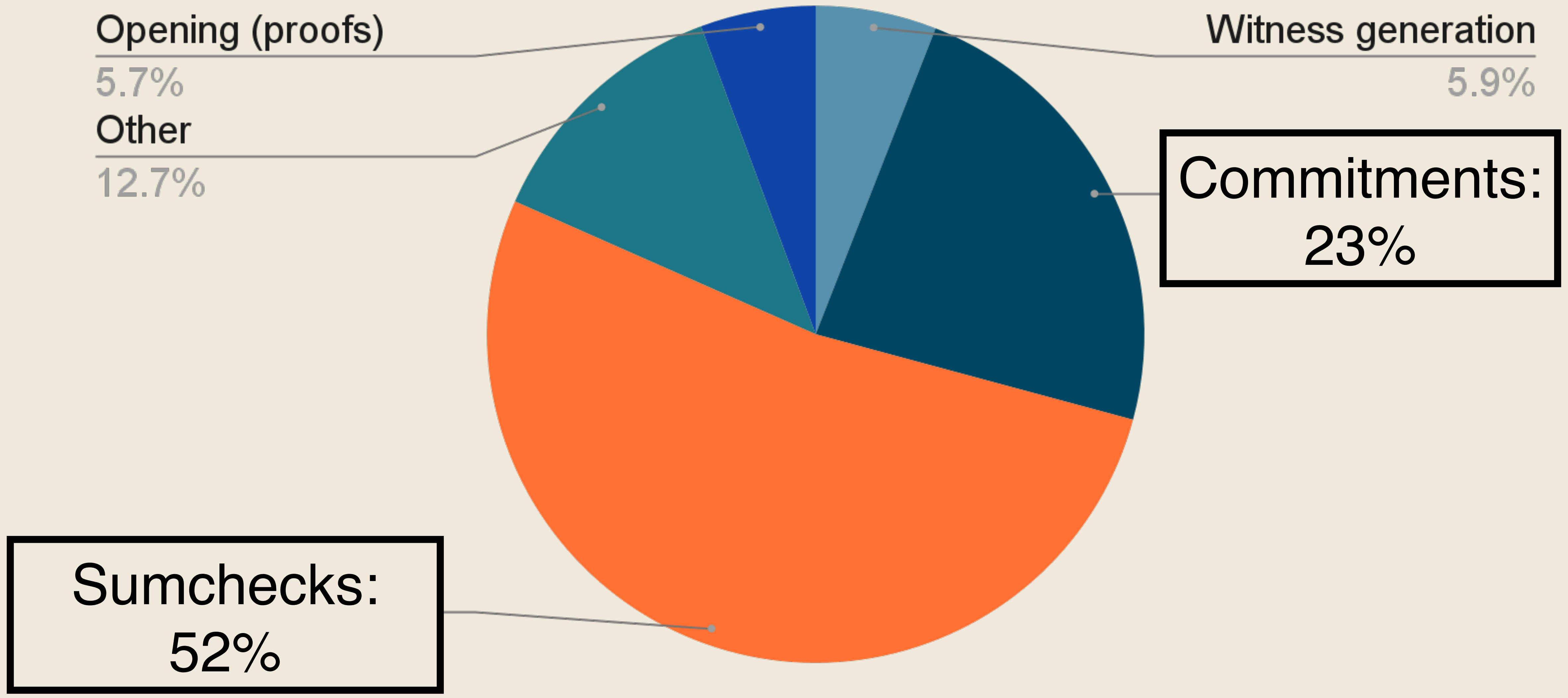
For an n-step program with memory size $|M|$:

Module	**Main steps**	**P cost**
Memory-checking (**Spice**)	2 GKRs	O(n + |memory|)
Constraints (**Spartan**)	2 sumchecks	O(n)
Lookups (**Lasso**)	1 sumcheck, 2 GKRs	$O(c^2 n)$

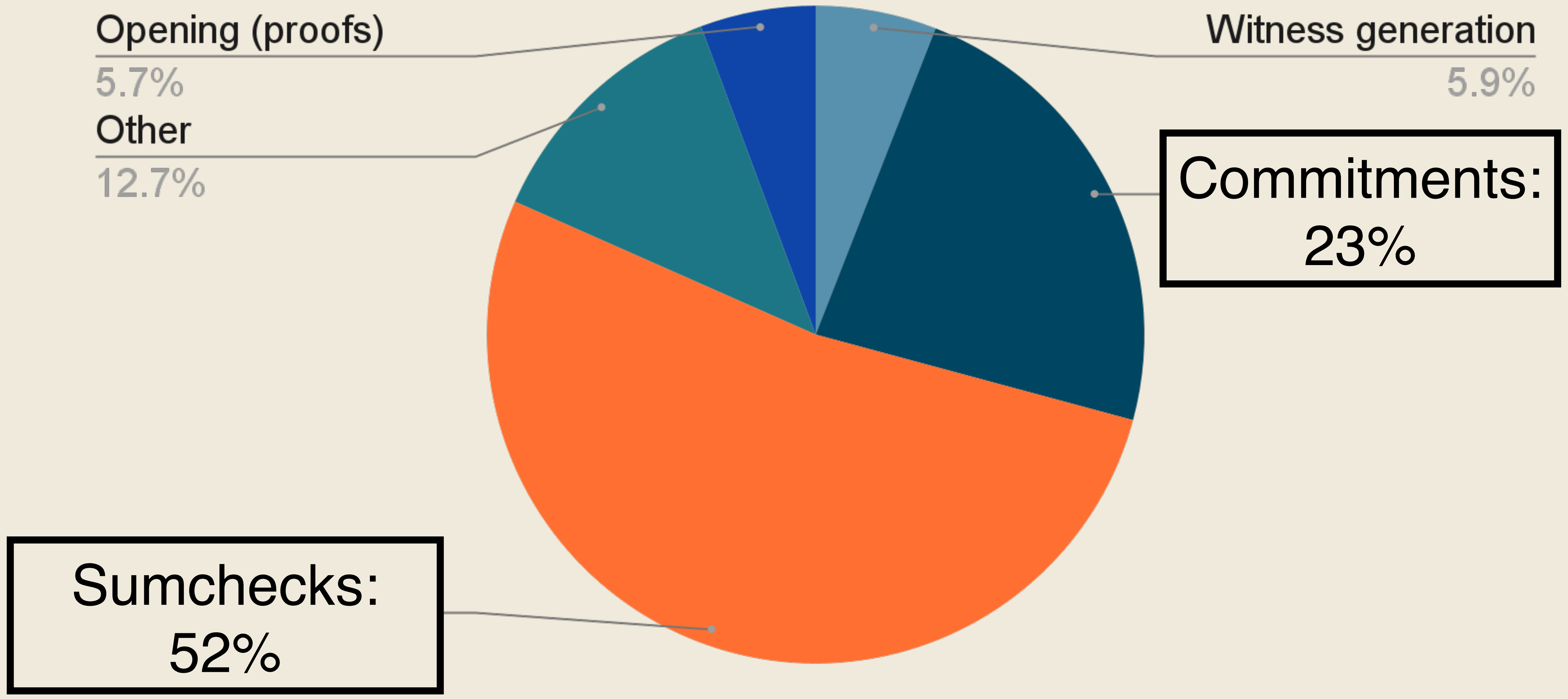
Proof size: Depends on the poly comm scheme. With Hyrax, it's $O(\sqrt{n})$ group elements.

Conclusion

Open-source implementation:
https://github.com/a16z/jolt



Conclusion



Open-source implementation:
https://github.com/a16z/jolt

Instructions proven per second: (on a MacBook)

50 kHz

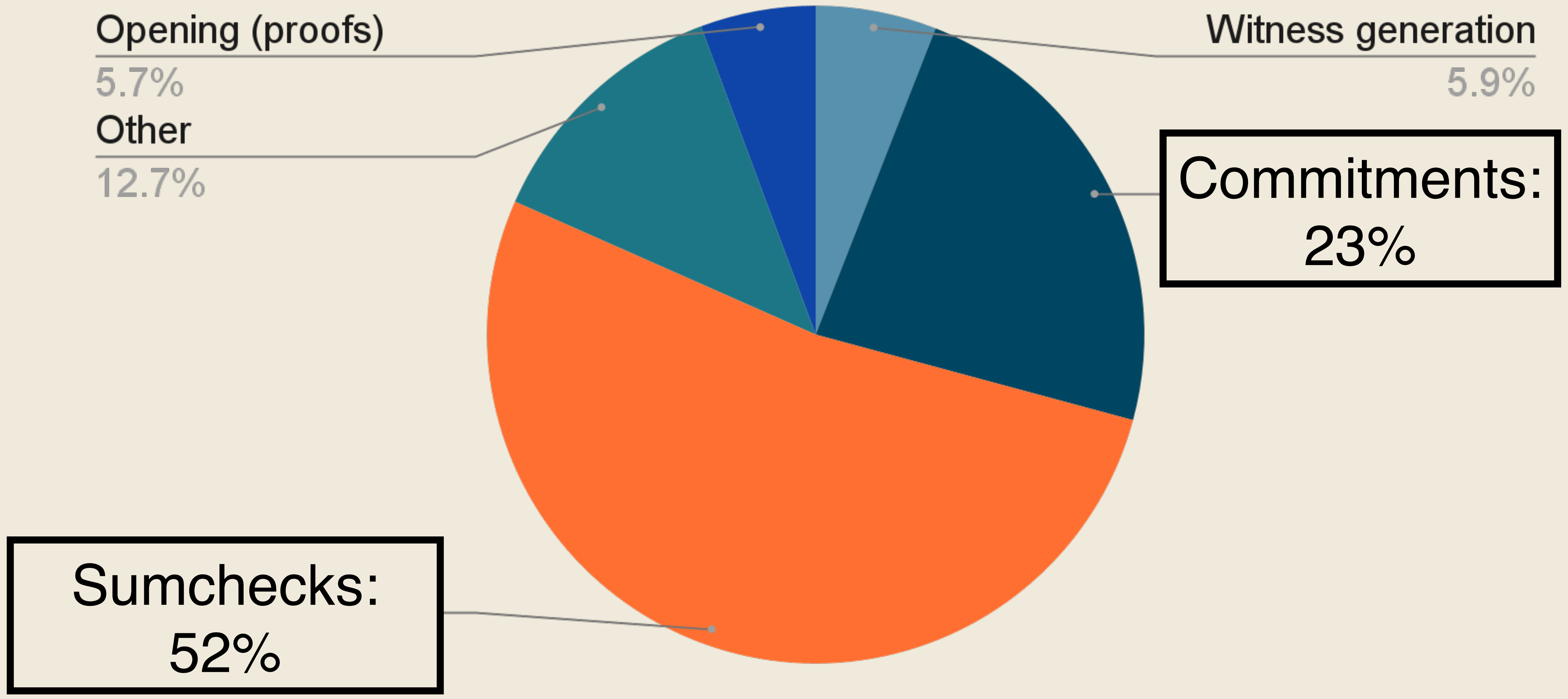
The Apollo 11 computer

100 kHz

Jolt

15 mHz

Conclusion



Open-source implementation:
https://github.com/a16z/jolt

Instructions proven per second: (on a MacBook)

50 kHz

The Apollo 11 computer

100 kHz

Jolt

15 mHz

A lot more (exciting) work to do!

Thanks for listening!