Zero-Knowledge Virtual Machines

John Guibas (Succinct Labs), 5/1/2025

Hello!

- My name is John. I was an undergrad at Stanford from 2019 - 2023.
- I took CS251, CS255, and CS355 and they were my foundation before leaving Stanford my senior year to start Succinct, a startup focused on building infrastructure for zero-knowledge proofs .
- Our team is ~30 people and we work with many leading companies in blockchains and beyond who have complex use cases for SNARKs.
- Our main product is **SP1**, a zero-knowledge virtual machine. That’s why I’m here :)

What you've learned in CS355 (from what I can tell…)

- What zero-knowledge proofs are
- What they are useful for
- Basic ways you can "program" SNARKs, like arithmetic circuits or R1CS
- Proof systems used in production like Groth16
- **The goal of this talk is to get you out of the classroom and show you why zero-knowledge virtual machines are becoming the preferred platform for SNARK development.**

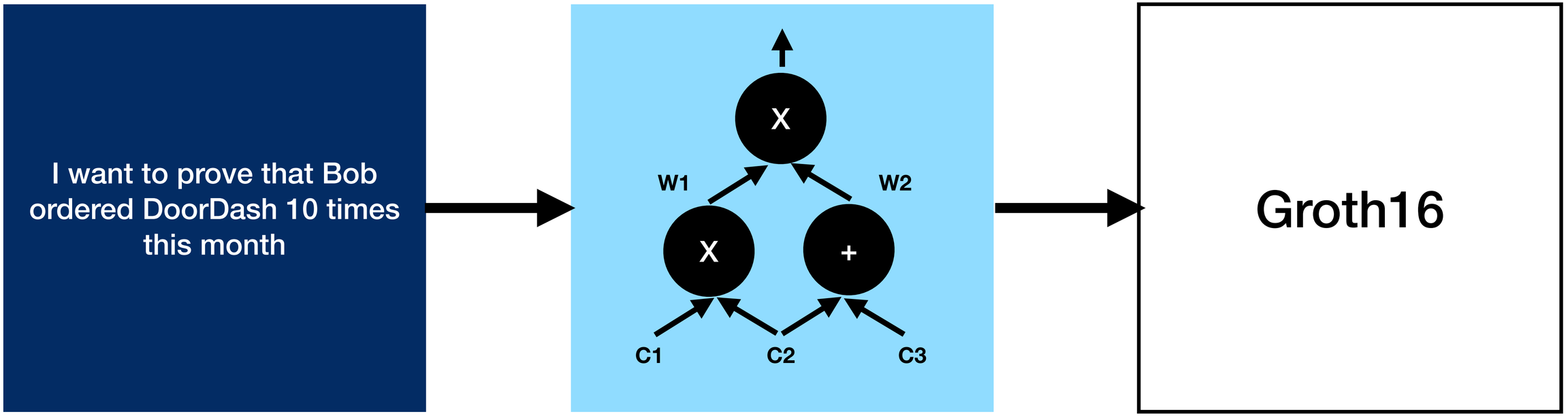
Outline

- Why should you care about zero-knowledge virtual machines?
- General definition of a zero-knowledge virtual machine
- SP1: a zero-knowledge virtual machine that runs RISC-V byte code
- Live demo
- What's next in zero-knowledge virtual machines
- Concluding thoughts

Why do people care about zero-knowledge virtual machines (zkVMs)?

Some historical context: how people use SNARKs

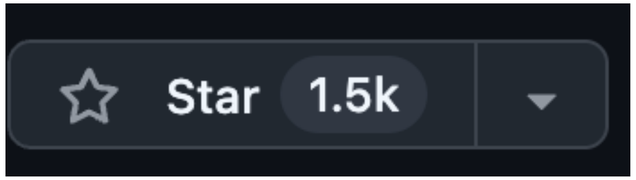
- Choose your favorite off-the-shelf proving protocol (i.e., Groth16)
- Write your application-specific logic as an arithmetic circuit (i.e., R1CS).



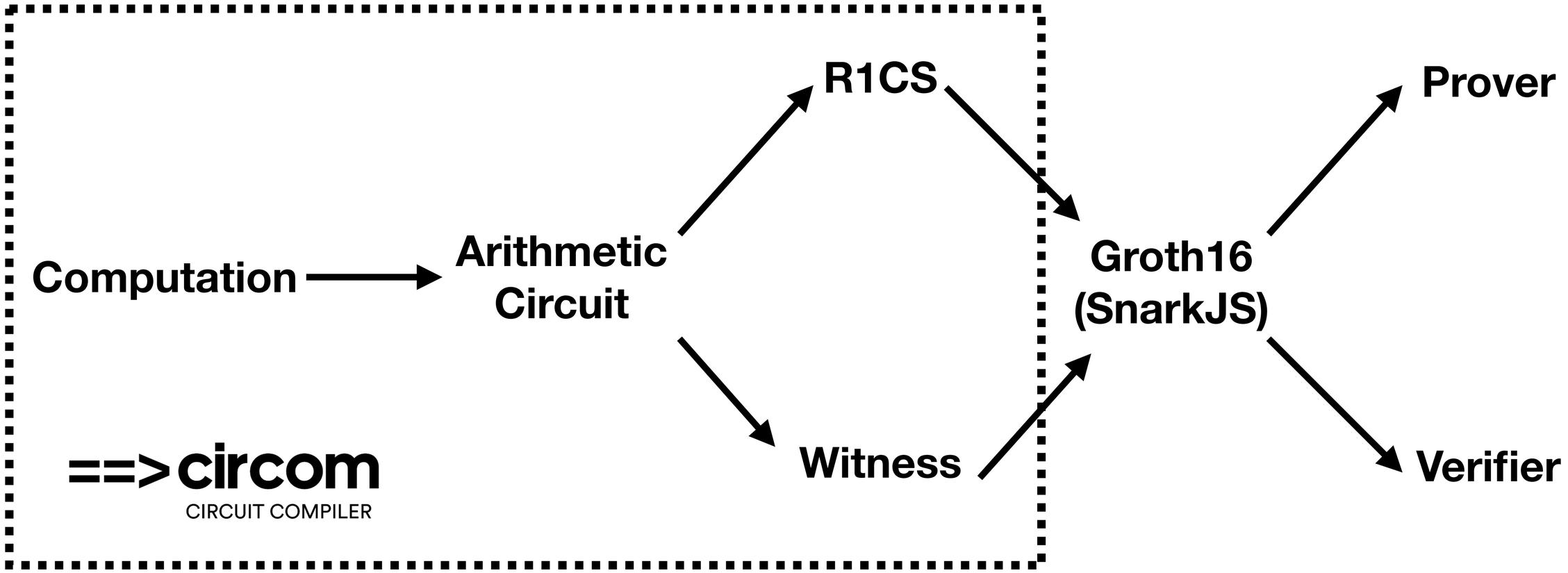
Creating SNARKs with frameworks like Circom

==> circom

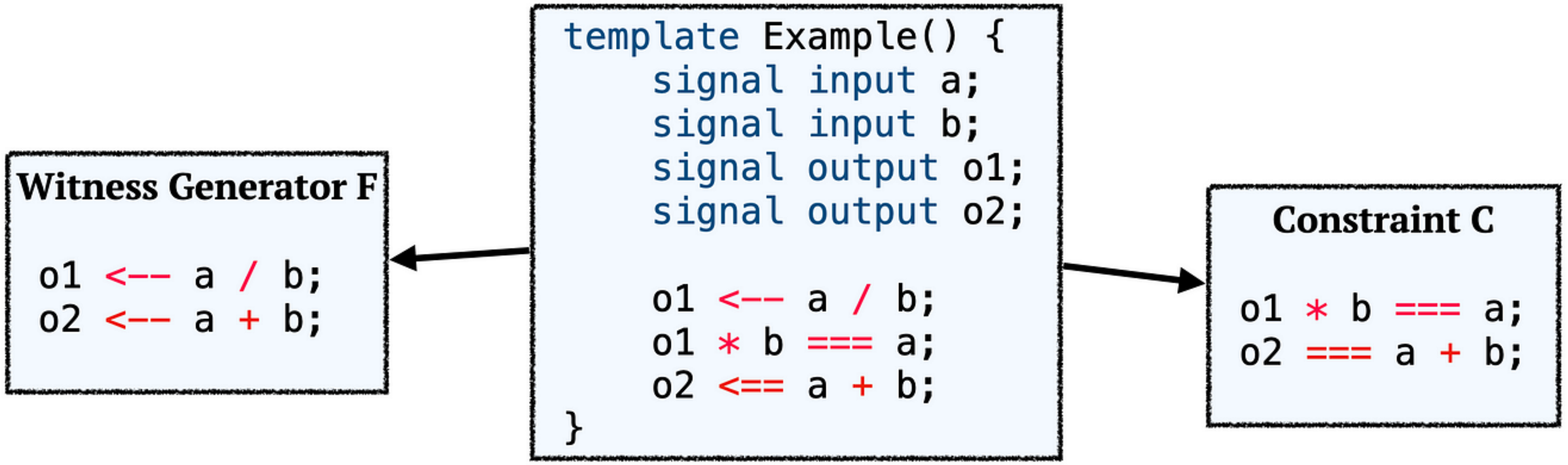
CIRCUIT COMPILER



Creating SNARKs with frameworks like Circom



Creating SNARKs with frameworks like Circom



Developing SNARKs with frameworks like Circom is long, manual, and frustrating process

I want to prove that
Bob ordered DoorDash
10 times this month

Human Developer

Takes months
Prone to error

```
template Example() {
    signal input a;
    signal input b;
    signal output o1;
    signal output o2;

    o1 <-- a / b;
    o1 * b === a;
    o2 <== a + b;
}
```

Decomposing a number to bits in Circom

```
fn num2bits(in: u32, n: u32) {
  let mut bits = Vec::new();

  for i in 0..n {
    let bit = (in >> i) & 1;
    bits.push(bit)
  }

  return bits;
}
```

```
template Num2Bits(n) {
    signal input in;
    signal output out[n];
    var sum = 0;

    var digit = 1;
    for (var i = 0; i < n; i++) {
        out[i] <-- (in >> i) & 1;
        out[i] * (out[i] -1 ) === 0;
        sum += out[i] * digit;
        digit = digit + digit;
    }

    sum === in;
}
```

Basic integer addition in Circom

```
fn modSum(a: u32, b: u32, n: u32) {
  return (a + b) % n;
}
```

```
template ModSum(n) {
    assert(n <= 252);
    signal input a;
    signal input b;
    signal output sum;
    signal output carry;

    component n2b = Num2Bits(n + 1);
    n2b.in <== a + b;
    carry <== n2b.out[n];
    sum <== a + b - carry * (1 << n);
}
```

Simple branching in Circom

```
fn branch(cond: bool) {
  if cond {
    let in0 = expensiveFn();
    return true;
  } else {
    let in1 = cheapFn();
    return false;
  }
}
```

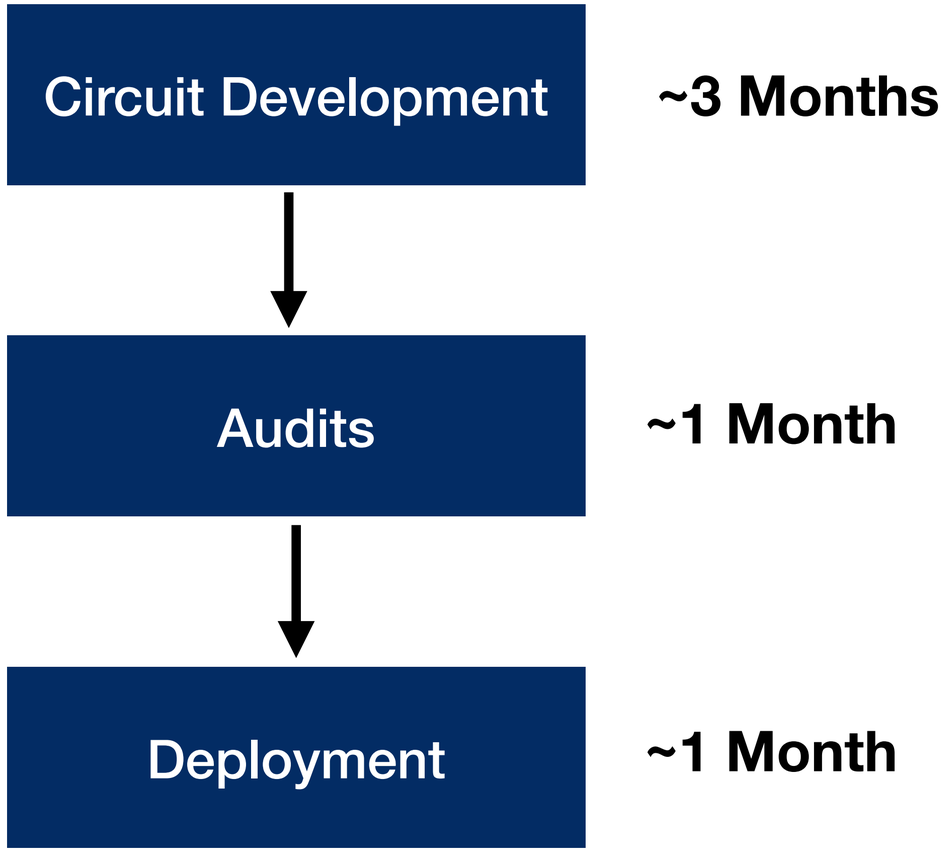
```
template Branch() {

   signal input sel;

   in0 <== expensiveFn()

   in1 <== cheapFn()

   sel * (sel - 1) === 0;

   out <== sel * in1 + (1 - sel) * in0;

}
```

General themes as to why encoding computation into circuits is hard

Aspect	Computation	Circuits / R1CS
Data Types	int, float, string, etc.	Field Elements
Control Flow	for, if, while, recursion	No control flow; must unroll loops, mux branches
Memory	Dynamic Arrays, Pointers, Stack	Everything is fixed-size and statically allocated

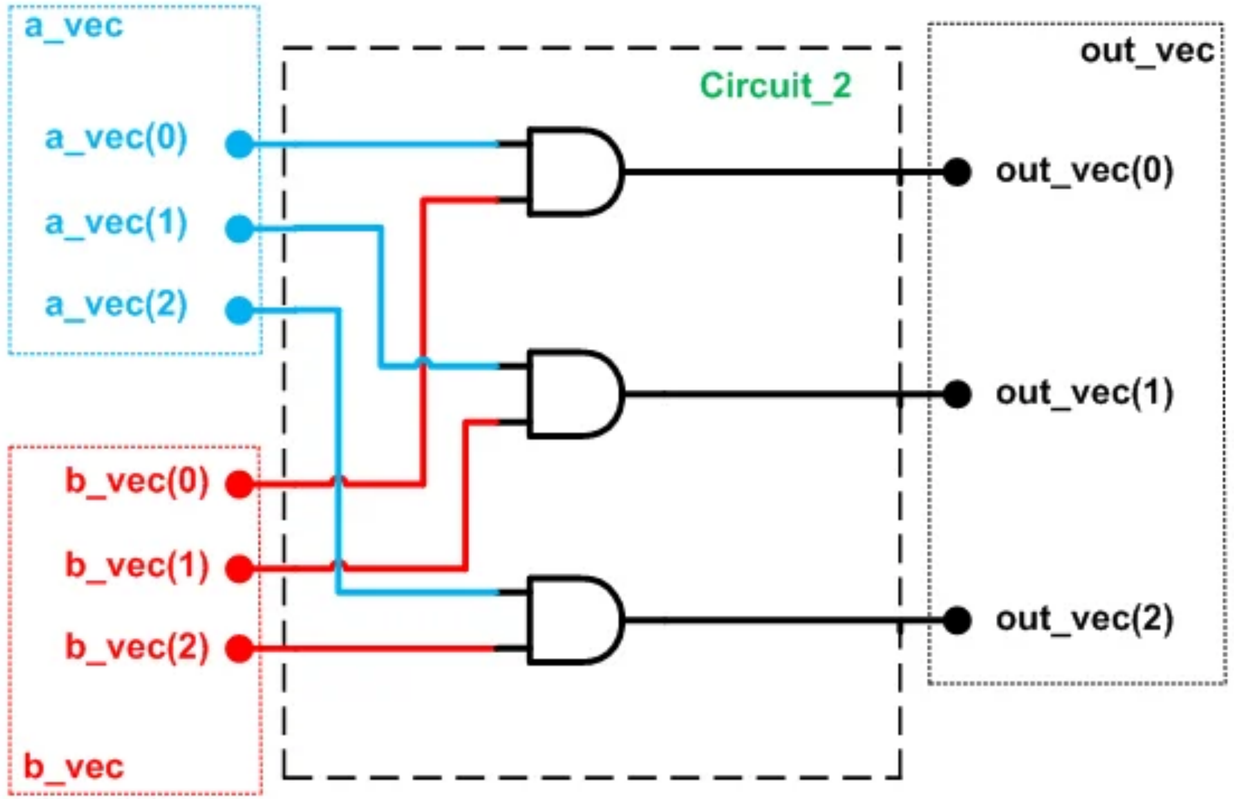
A painful anecdote: BLS12-381 pairing verification SNARKs for Ethereum

- BLS12-381 pairings are commonly used primitive to verify aggregated signatures on common blockchains, such as Ethereum.
- **Required team of 2 cryptography engineers for development across ~3 months (~20K LOC).**
- **Spent 6 figures on security audit from S-tier auditing firm.**

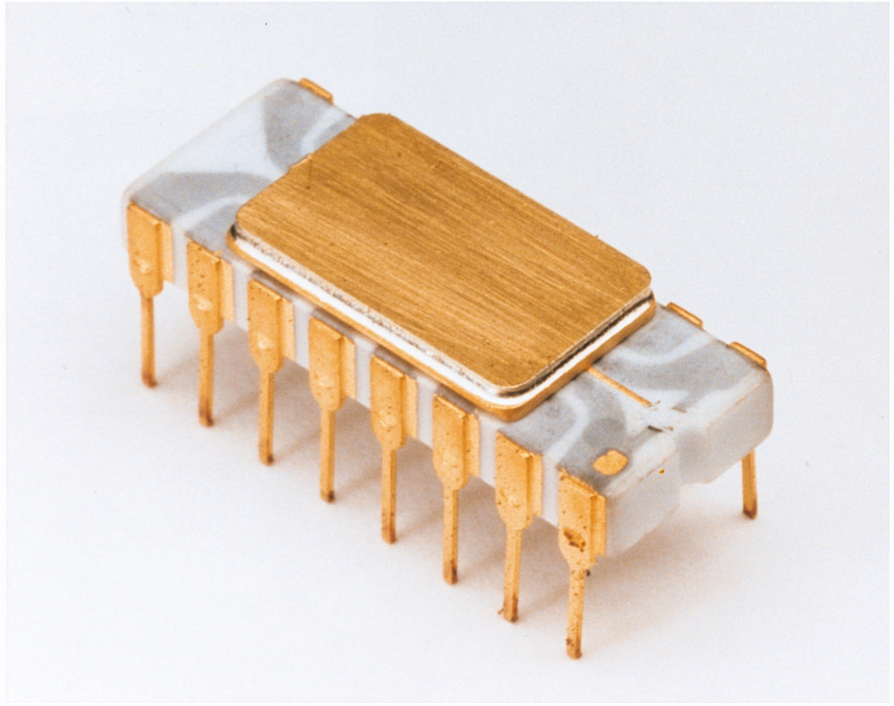


These challenges are extremely familiar to early hardware design: Verilog, ASICs, and microcontrollers

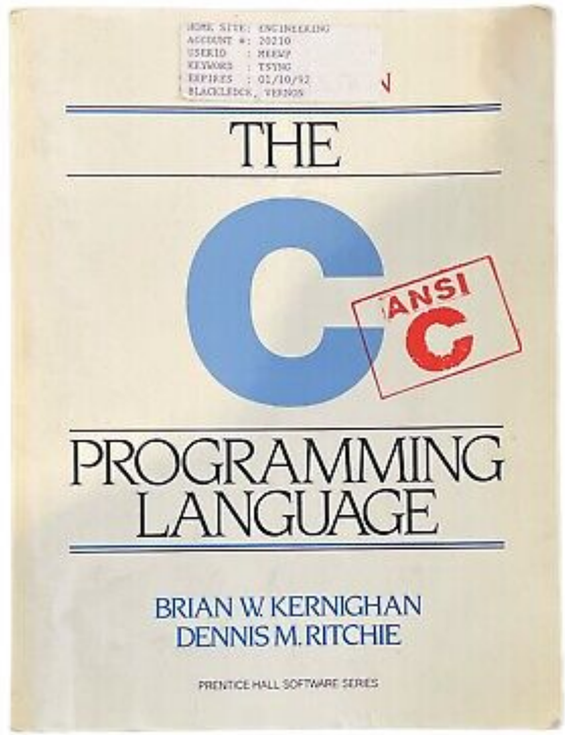
- Hand-crafting every low-level operation
- Minimal abstraction layers
- Small mistakes can lead to costly, fatal failures
- Long feedback loops: slow iteration and testing cycles
- Months to years to ship



1950s–1980s: Higher-level programming and general-purpose CPUs liberated hardware development



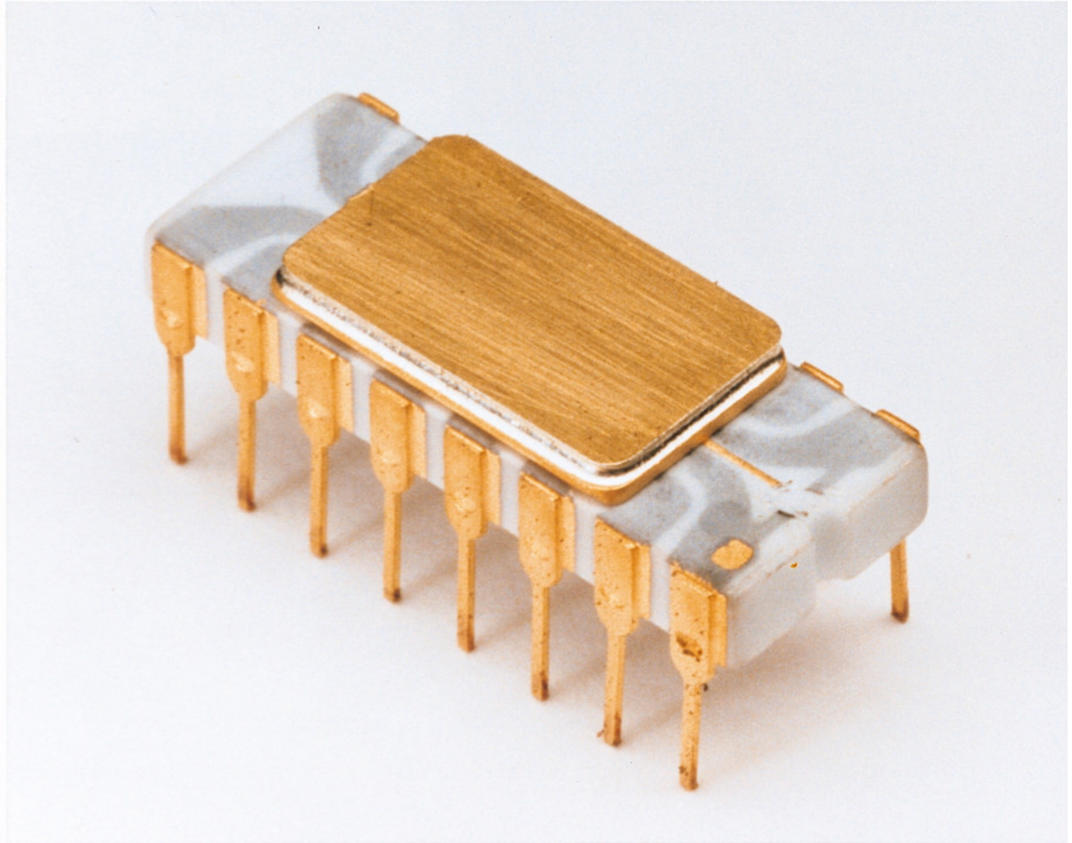
Intel 4004: The First Programmable Microprocessor



C: Programming Language

1950s–1980s: Higher-level programming and general-purpose CPUs liberated hardware development

- **General-purpose reusability:** one chip could run many programs
- **Higher-level thinking:** engineers focus on algorithms and software, not wiring up logic gates
- **Rapid iteration:** Changes meant edited code, not reworking entire circuits



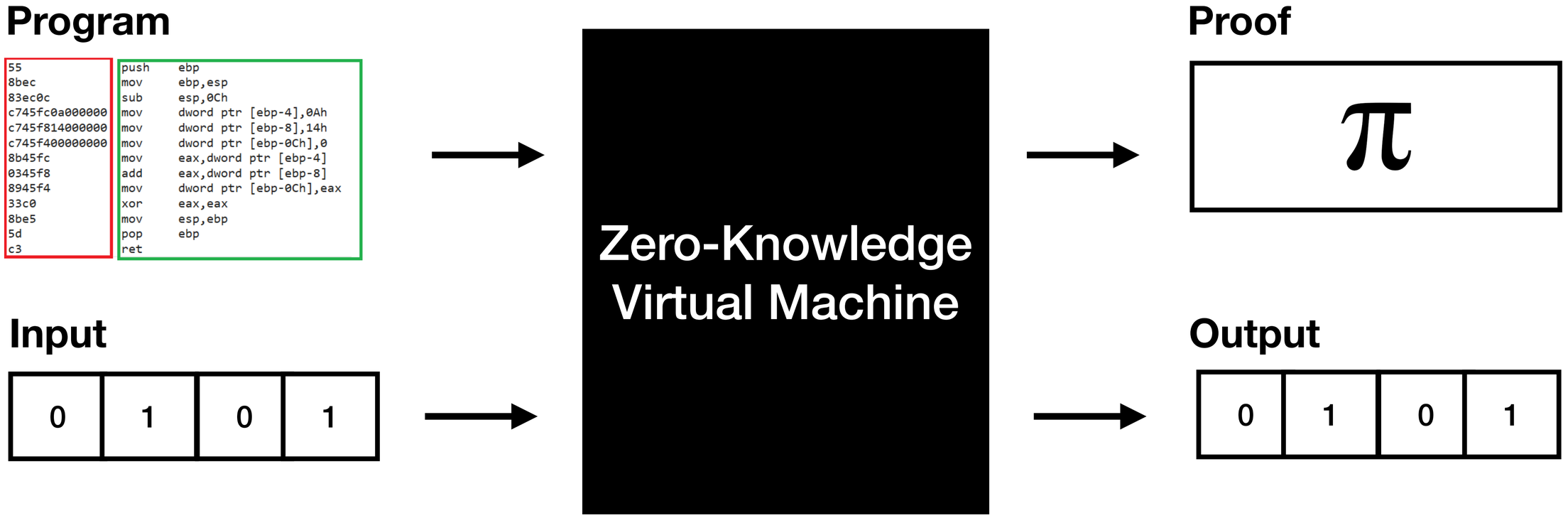
General-purpose computing made software faster to build, more powerful, and vastly more scalable.

Zero-knowledge systems today face the same challenges early computing faced.

Zero-knowledge virtual machines (zkVMs) aim to bring general-purpose computation into the world of SNARKs.

What is a zero-knowledge virtual machine?

- A **zero-knowledge virtual machine (zkVM)** is an arithmetic circuit that can run programs and generate zero-knowledge proofs of their execution.



How programs are compiled for zero-knowledge virtual machines

- Programs can be compiled from a higher-level language or written in raw assembly. Crucially, they allow you to have **stateful execution** and **control flow.**

```
transition add(a: field, b:
field) -> field {

    return a + b;

}
```

Aleo

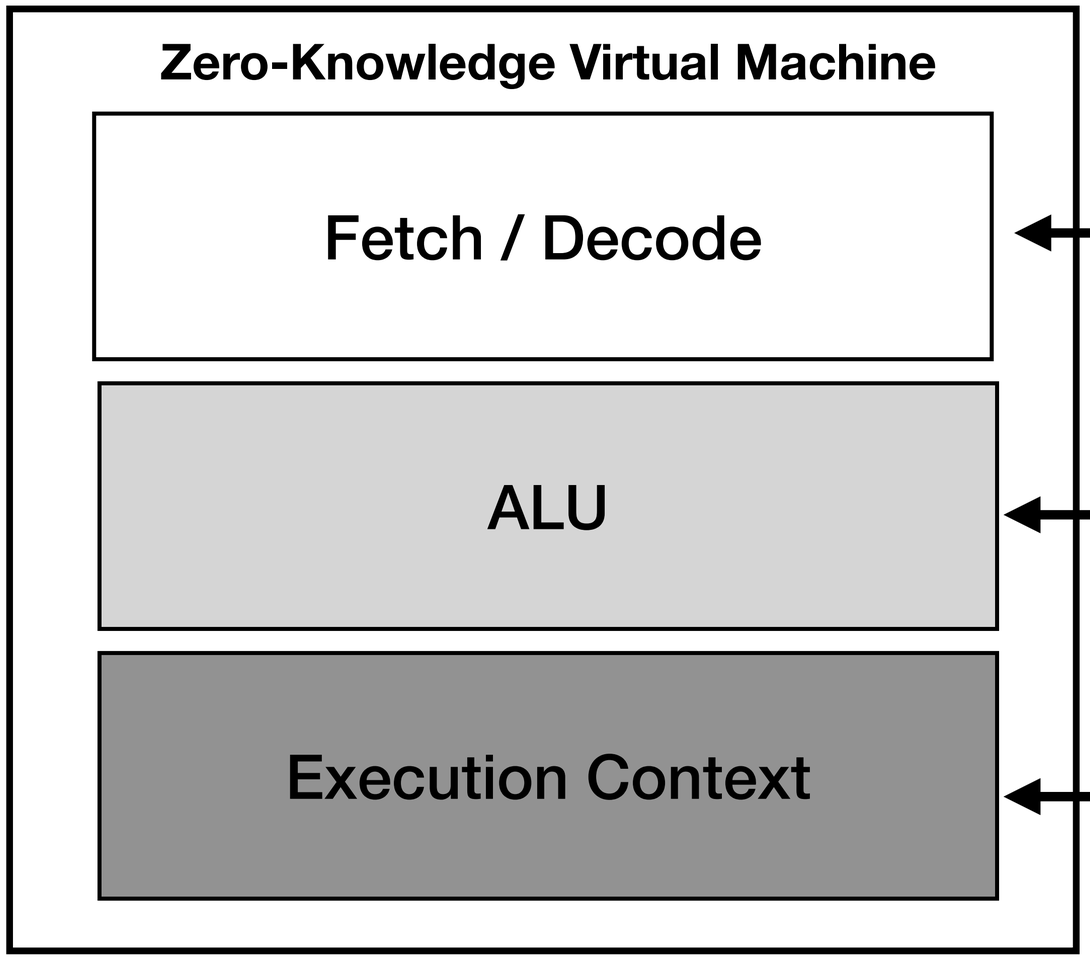
Compile

```
pushq   %rbp
movq    %rsp, %rbp
pushq   %r12
pushq   %rbx
subq    $32, %rsp
movl    %edi, -20(%rbp)
movq    %rsi, -32(%rbp)
movq    %rdx, -40(%rbp)
movq    -40(%rbp), %rdx
movq    -32(%rbp), %rcx
movl    -20(%rbp), %eax
movq    %rcx, %rsi
movl    %eax, %edi
call    _M2_init
```

Aleo IR

How do programs get executed inside the zero-knowledge virtual machine?

(exactly the same as traditional computing!)



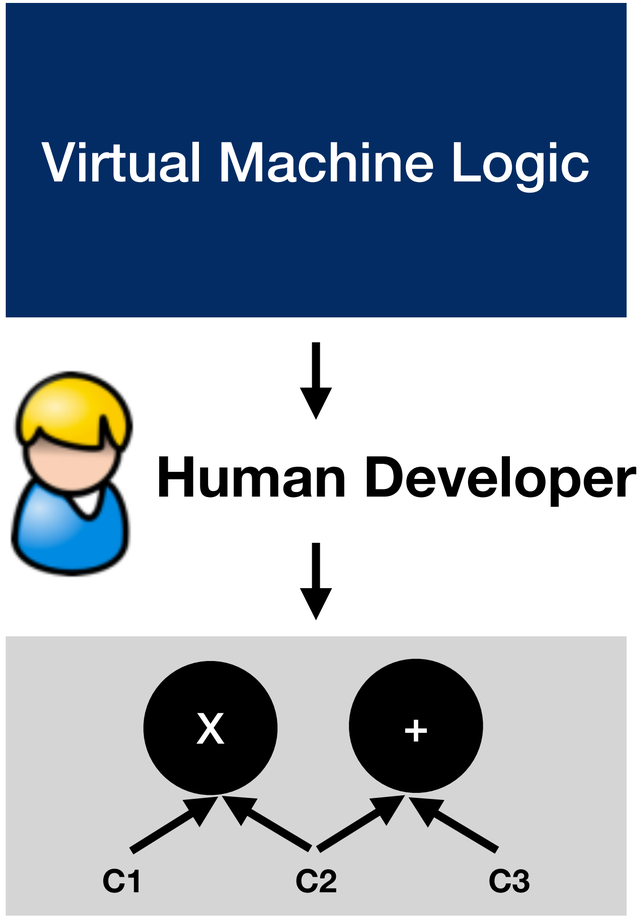
Determine what instruction to run next

Execution unit: performs the operation described any an instruction

Registers/Memory: maintain program state and store variables used as inputs/outputs to operations

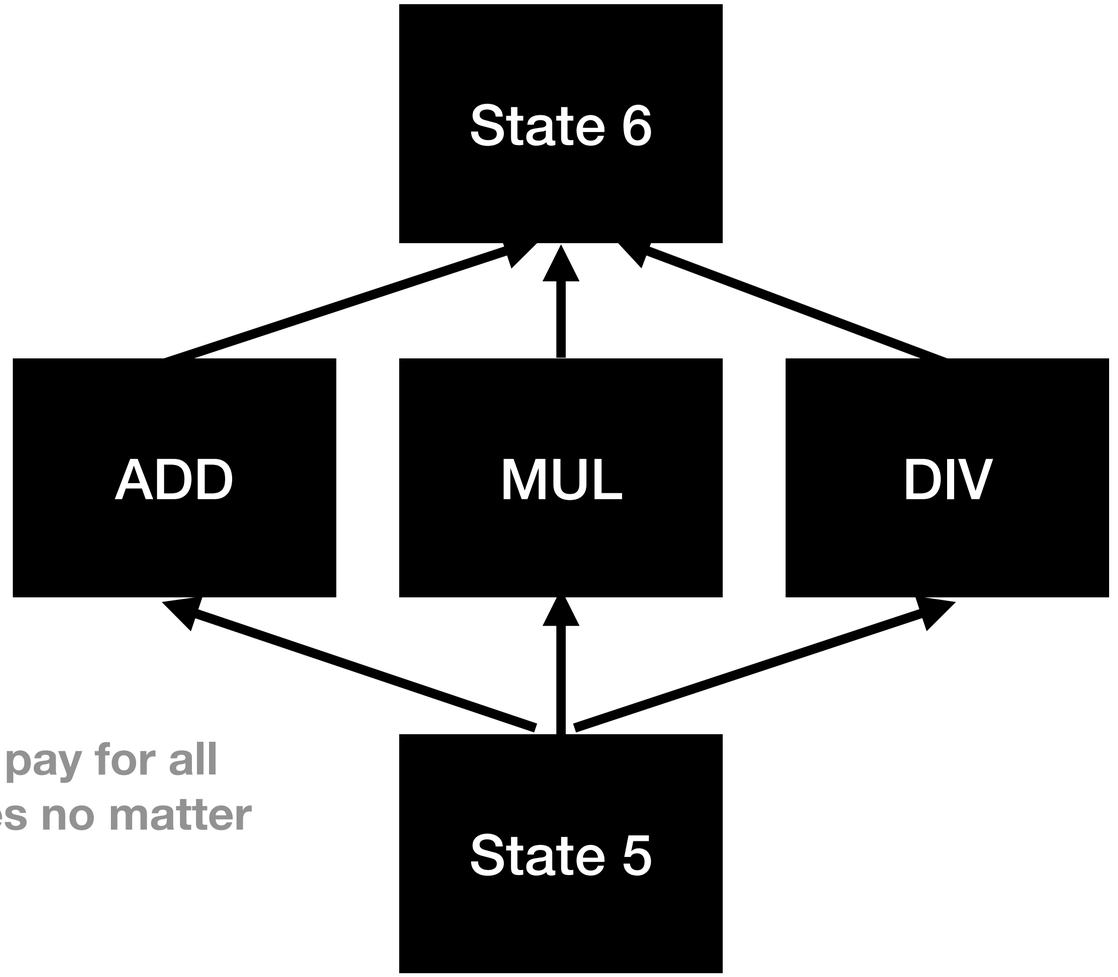
The remaining question: how to implement a zero-knowledge virtual machine

- Once you've decided on your instruction set and state transition function, the key thing is to encode the virtual machine as inside some SNARK protocol
- **What we want:**
  - **Universality.** Should be able to support any program not just a limited set.
  - **Efficiency.** Should be efficient to prove with reasonable verification cost.



Encoding the virtual machine inside SNARKs: Arithmetic Circuits

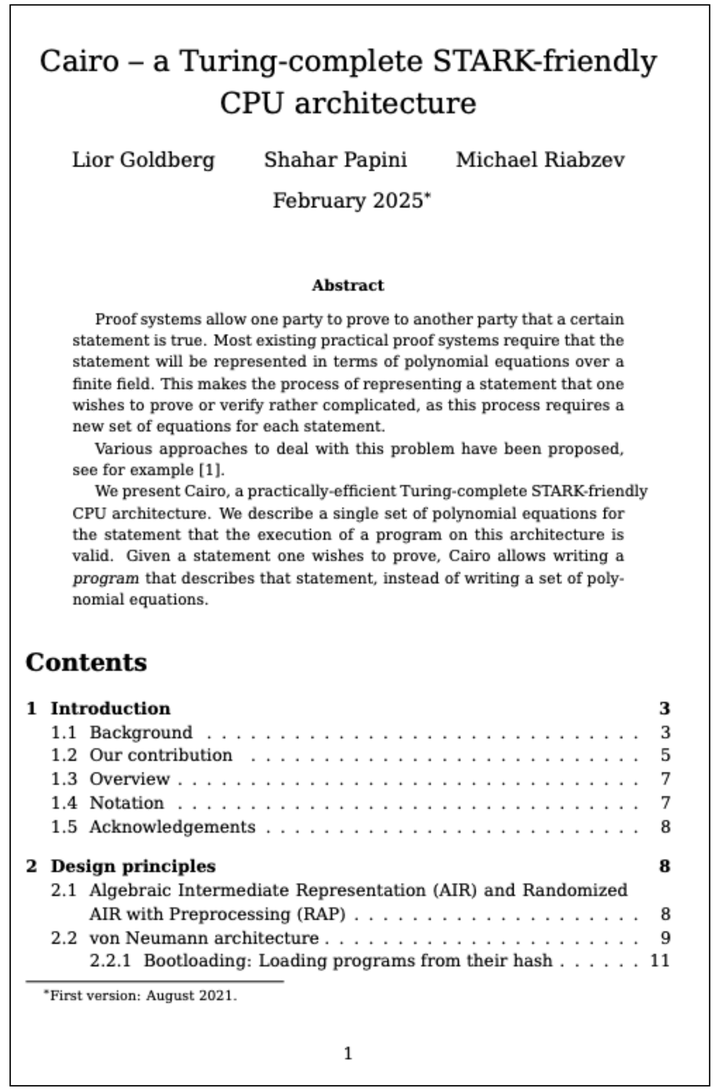
- Naive solution: encode a virtual machine as an arithmetic circuit and run inside your favorite proving protocol (i.e., Groth16).
- **Problem 1:** Because arithmetic circuits don't have branching, you pay the cost of all instructions, not just one.
- **Problem 2:** Memory access is extremely expensive. Selecting from an array of N costs O(log(N)) with a Merkle Tree.



Have to pay for all branches no matter what!

Encoding the virtual machine inside SNARKs: Cairo

- **Cairo (Goldberg et. al, 2021):** one of the first implementations of an efficient Turing-complete CPU / VM inside a SNARK
- Key Insights:
  - Prover-friendly instructions (field elements as the native type) to avoid overhead of supporting many expensive u32/bitwise instructions
  - Custom cryptographic arguments allow for cheap memory reads/writes



Cairo – a Turing-complete STARK-friendly CPU architecture

Lior Goldberg Shahar Papini Michael Riabzev

February 2025*

Abstract

Proof systems allow one party to prove to another party that a certain statement is true. Most existing practical proof systems require that the statement will be represented in terms of polynomial equations over a finite field. This makes the process of representing a statement that one wishes to prove or verify rather complicated, as this process requires a new set of equations for each statement.

Various approaches to deal with this problem have been proposed, see for example [1].

We present Cairo, a practically-efficient Turing-complete STARK-friendly CPU architecture. We describe a single set of polynomial equations for the statement that the execution of a program on this architecture is valid. Given a statement one wishes to prove, Cairo allows writing a *program* that describes that statement, instead of writing a set of polynomial equations.

Contents

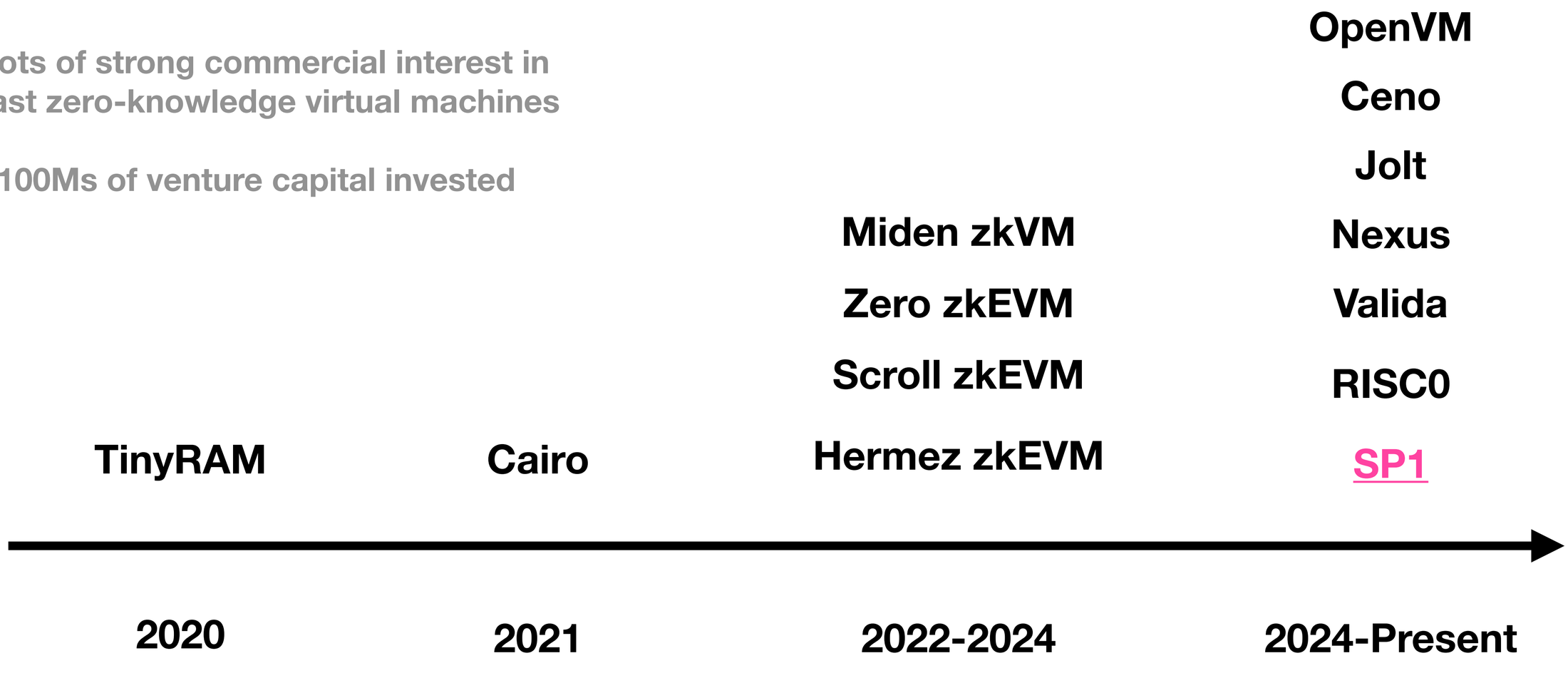
*First version: August 2021.

1

Fast forward to 2025: modern-day zero-knowledge virtual machine architectures

Lots of strong commercial interest in fast zero-knowledge virtual machines

$100Ms of venture capital invested



Fast forward to 2025: modern-day zero-knowledge virtual machine architectures

Aleo

zkVM focused on privacy (i.e., private transfers)
ISA: Aleo IR

polygon zkEVM

zkVM focused on scaling blockchains like Ethereum
ISA: EVM

A deep dive into a state-of-the-art zero-knowledge virtual machine: SP1

(that our team at Succinct built!)

Using zero-knowledge virtual machines still relied on obscure ZK-specific programming languages

- Until last year, mainstream tooling for SNARKs still relied on obscure DSLs like Circom or new programming languages specific to ZK
- Using a new programming language is *awful*
  - Immature IDE support
  - Immature compiler / debugging
  - Need to build all libraries from scratch

```
func main{output_ptr : felt*}() -> ():
    let a = 5
    let b = 7
    let result = a + b


    # Write result to the output
    assert [output_ptr] = result


    return ()
end
```

Cairo DSL

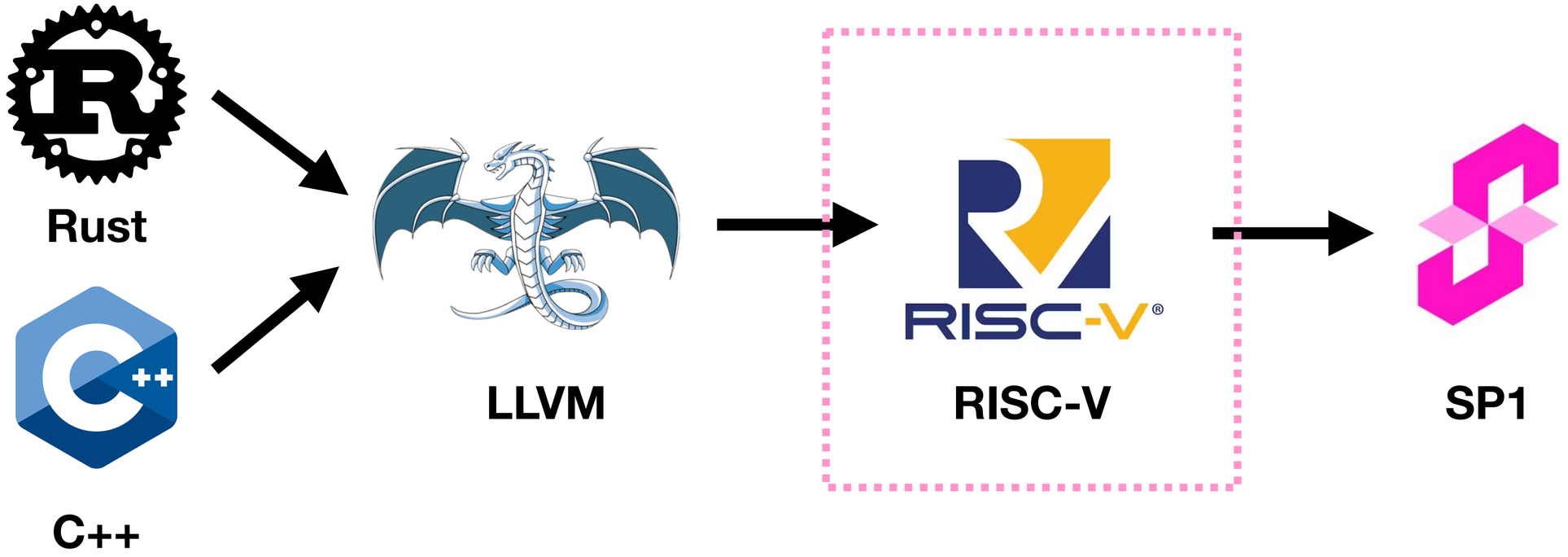
SP1: developing SNARKs should be as simple as writing C++/Rust

- In ~2024 with SP1, we released a zkVM that was up to 27x faster than existing approaches that could run C++/Rust
- zkVM performance still had a lot of room to grow with new algorithms and performance optimization
- **For the first time, could use traditional ISAs without worrying about overhead**
- While systems like RISC0 explored this approach before, they had some performance limitations that limited adoption

```
int main(int arc, char **argv)
{
  int x = 1
  for (int i = 0; i < 10; i++) {
    x = x + x;
  }
  return 0;
}
```

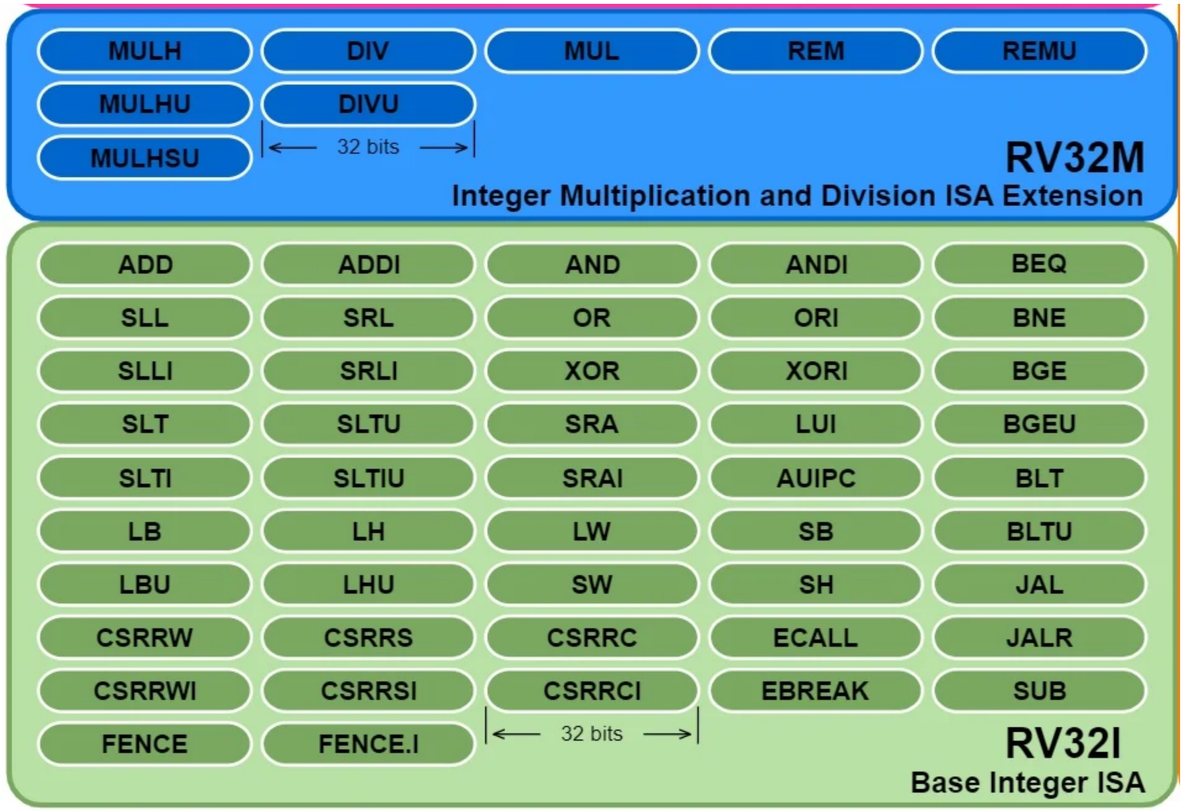
C++

How SP1 compiles C++/Rust programs for a zero-knowledge virtual machine



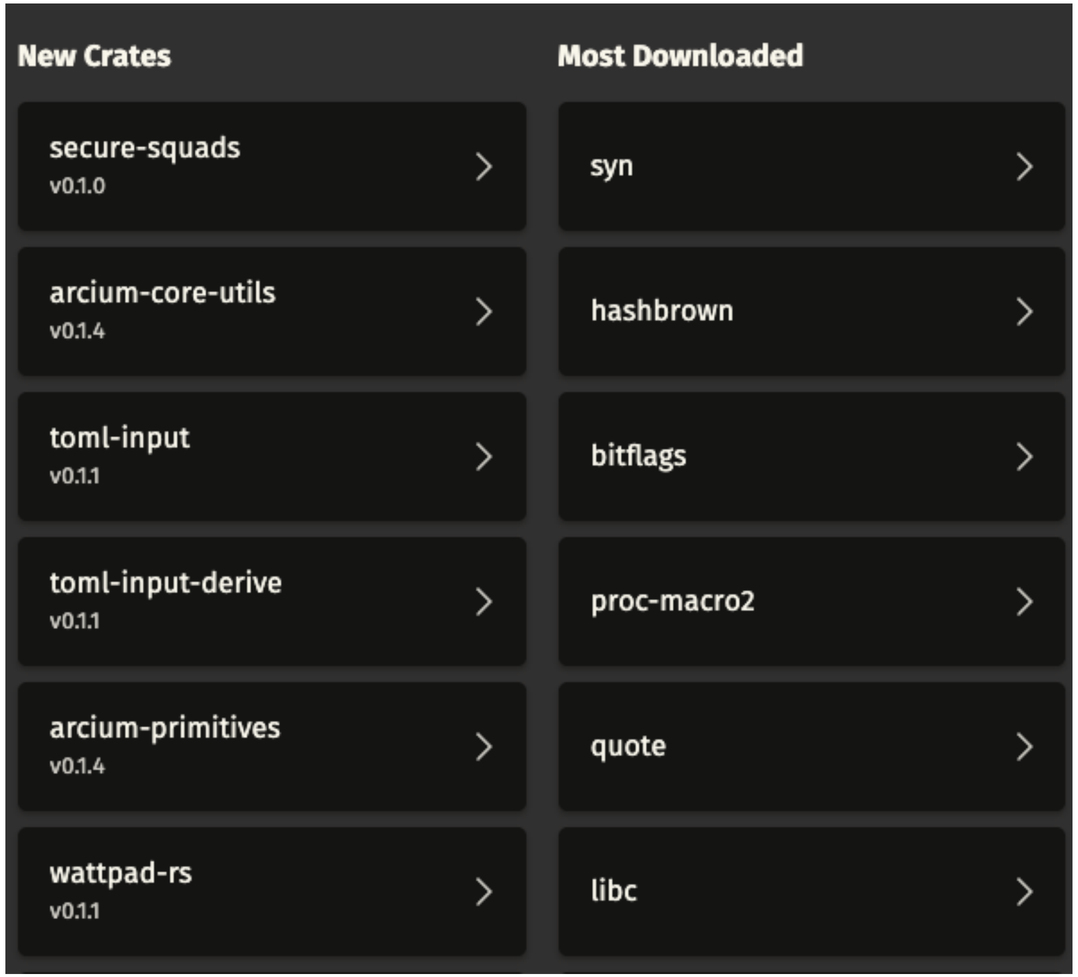
Why RISC-V is increasingly being adopted for zero-knowledge virtual machines

- **Open and free.** Most ISAs are proprietary and require licenses.
- **LLVM Compatible:** Most programming languages compile to LLVM IR, which can compile to RV32IM.
- **Simple:** Less than 60 instructions.
- **Performant:** Despite being so simple is quite performant compared to x86 and ARM.



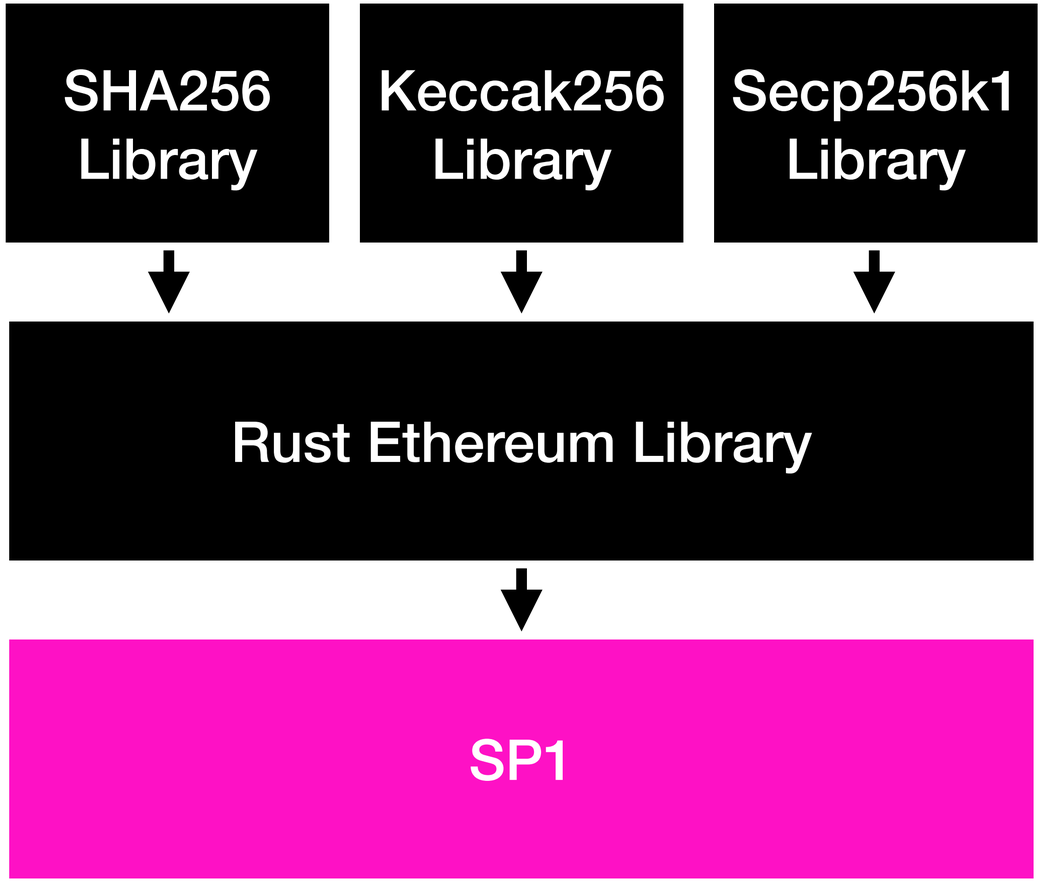
Building SNARKs is now as easy as normal software development with RISC-V zkVMs

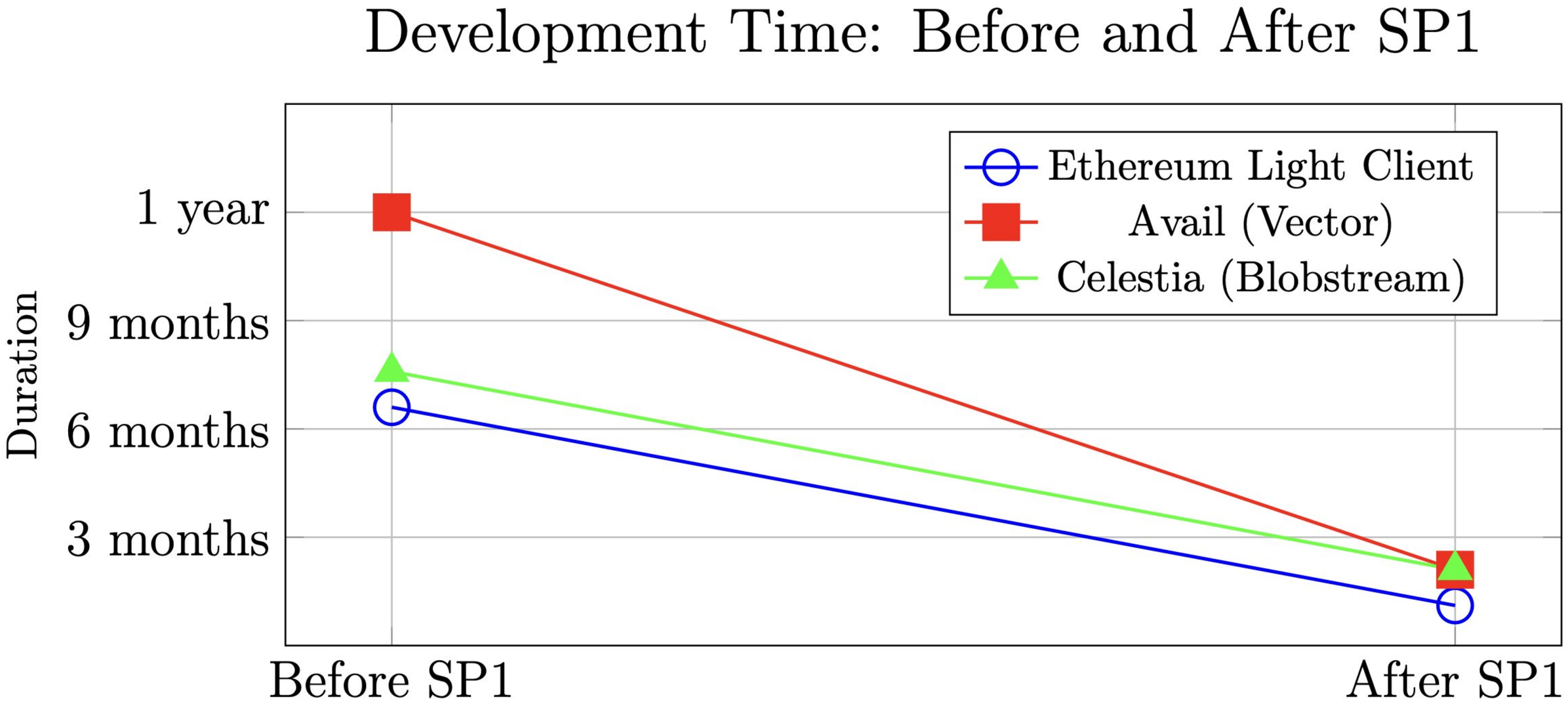
- **Reuse libraries:** cryptography libraries, integer libraries, strings, types, etc.
- **Reuse compiler optimizations:** use the decades of performance passes that have been built
- **Use your favorite IDE:** VSCode, extensions, debuggers, lints, GDB, etc.



An anecdote about SP1: a prover for Ethereum in *a week* instead of years

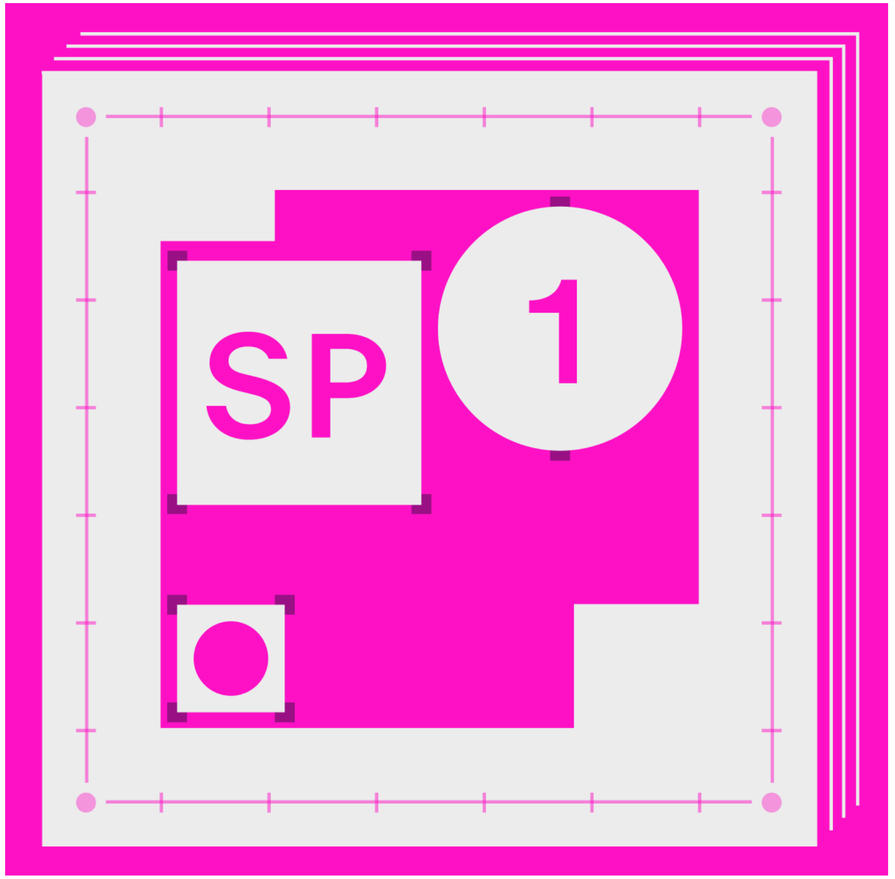
- One of the biggest usecase for SNARKs is scaling blockchains. To do so, many venture-backed teams have spent years building efficient application-specific provers for Ethereum.
- With SP1, you can just take an existing EVM implementation written in Rust, compile it to RISC-V and get a performant prover out of the box.
- **100% code reuse!!**




Development Time: Before and After SP1
Duration
1 year
9 months
6 months
3 months
Before SP1
After SP1
Ethereum Light Client
Avail (Vector)
Celestia (Blobstream)

SP1 has seen tremendous adoption for production SNARK applications in < 1 year

- Released in February 2024, up to 27x faster than SOTA.
- Now has 1.3K+ Github Stars, 86 contributors, 200K+ all time downloads
- Used by 30+ VC-backed startups and teams building in blockchains and beyond.

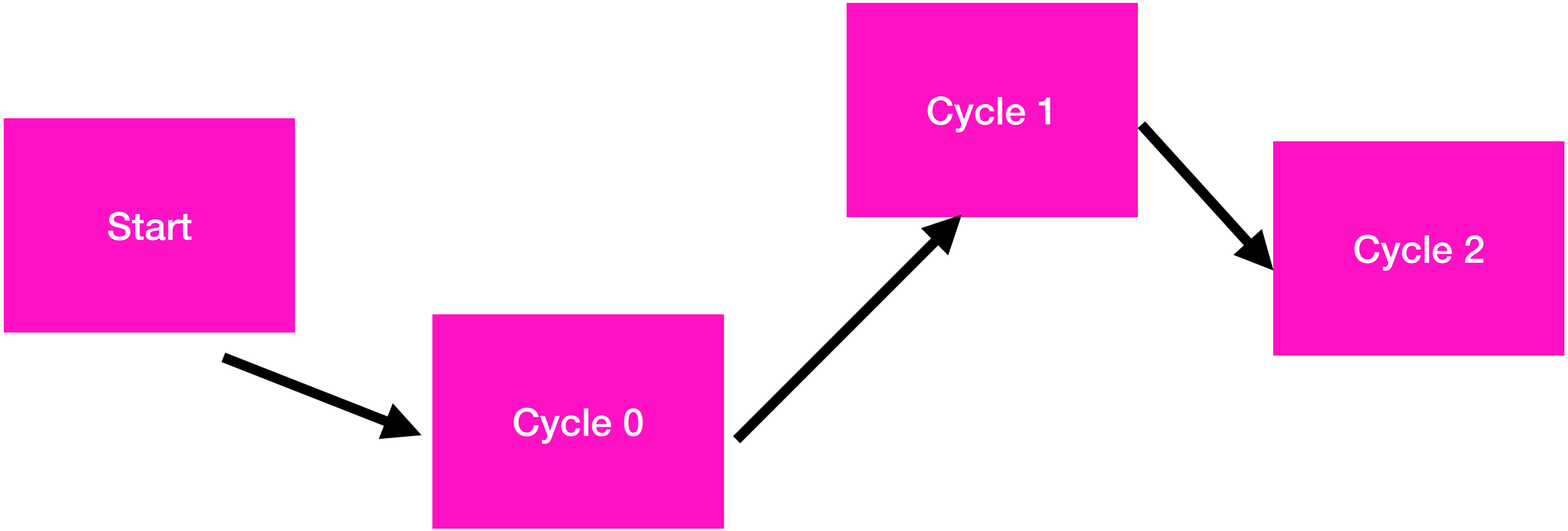


How SP1 Proves RISC-V

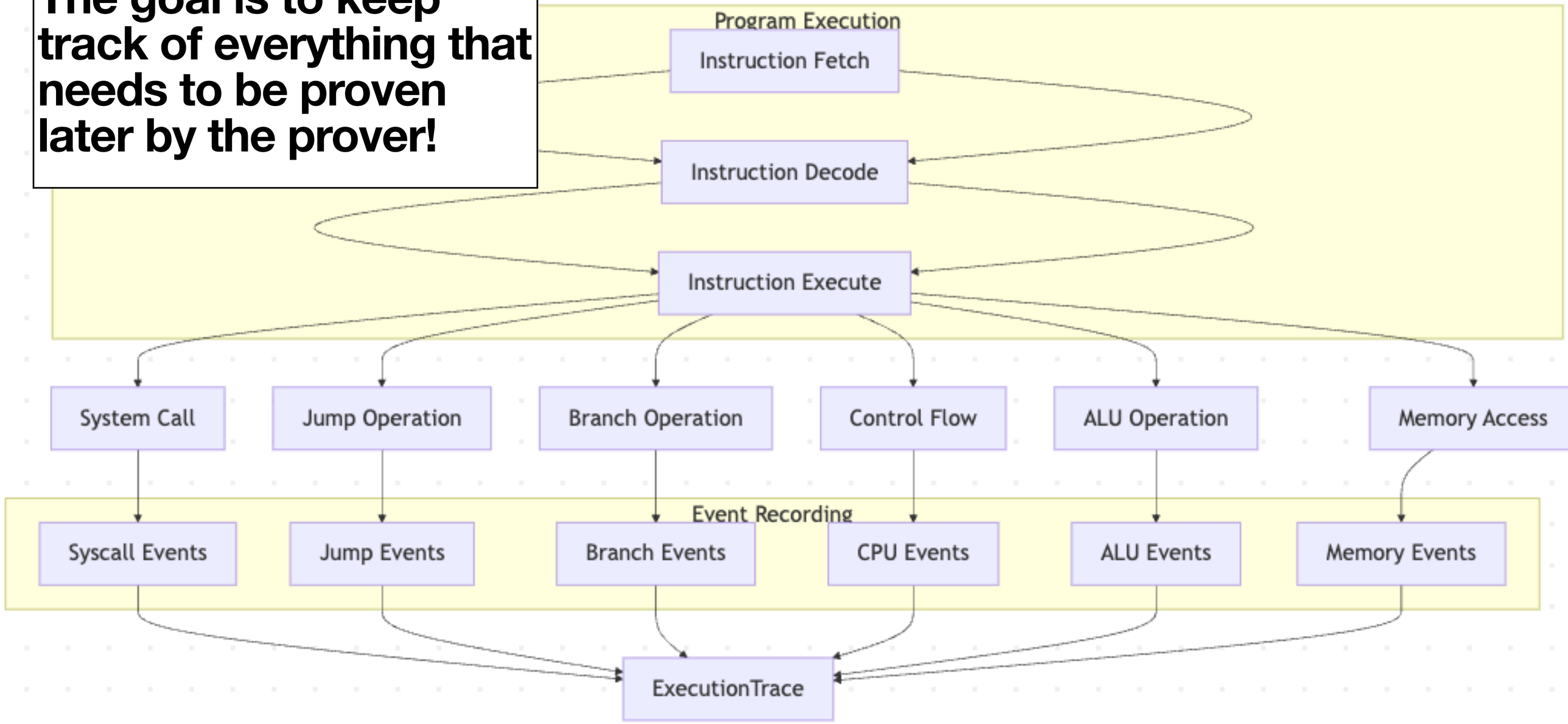
Proving RISC-V is like proving anything using a zero-knowledge proof

- Let R be a polynomial-time relation such that $(x, w) \in R$ means that w is a valid witness of a RISC-V execution with output x.
- A prover P who knows a witness w such that $(x, w) \in R$
- A verifier V who only knows the the output x
- **Completeness:** If the prover knows a valid trace $(x, w) \in R$, then the verifier accepts with probability 1.
- **Soundness:** If no such trace exists, then no (even malicious) prover can convince the verifier to except, except with negligible probability.

We call the witness w the execution trace. It's a comprehensive trail of every step the virtual machine took



The goal is to keep track of everything that needs to be proven later by the prover!



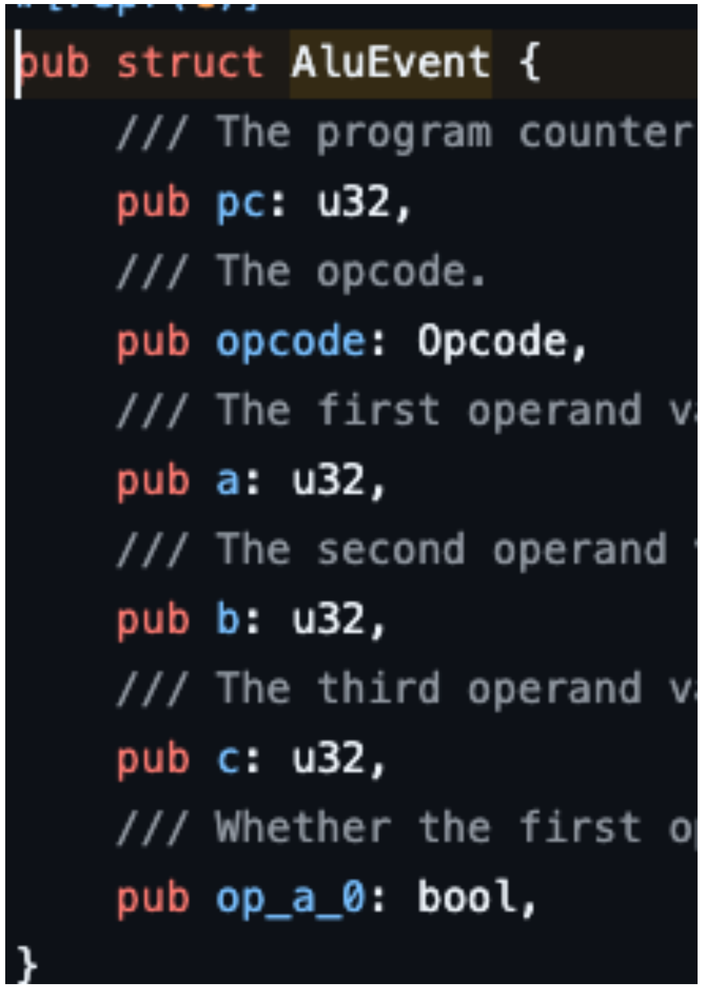
Creating the Execution Trace: CPU Events

- Think of a “CPU Event” as a checkpoint of the state of the virtual machine at every cycle.
- Contains the current clock, program counter, next program counter, the instruction being executed, etc.
- **Prover needs to prove we’re updating the program counter correctly!**

```rust
#[derive(Debug, Copy, Clone, Serialize, Deserialize)]
pub struct CpuEvent {
    /// The clock cycle.
    pub clk: u32,
    /// The program counter.
    pub pc: u32,
    /// The next program counter.
    pub next_pc: u32,
```

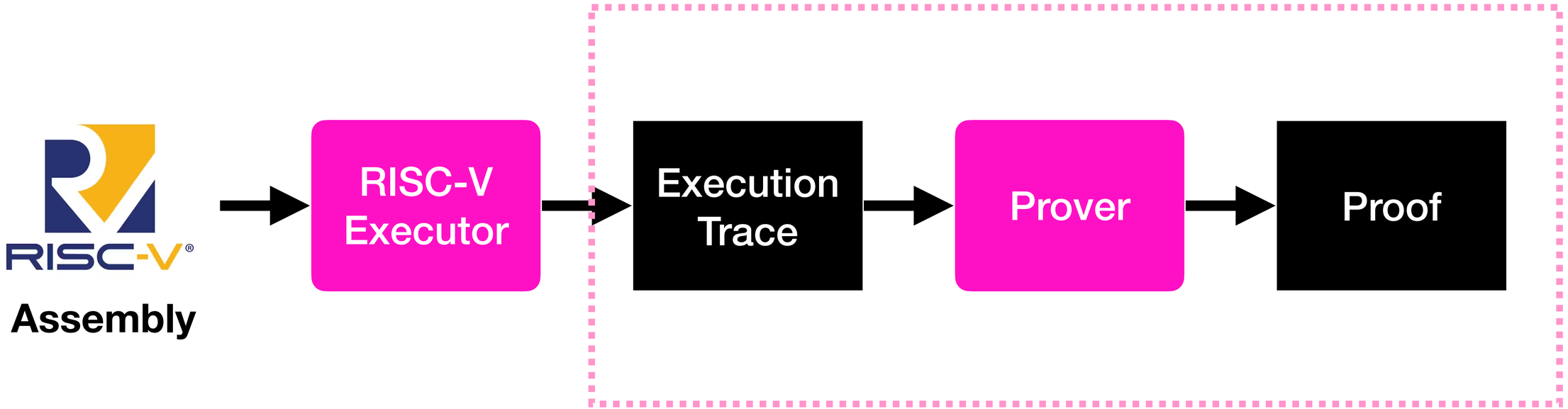
Creating the Execution Trace: ALU Events

- Think of a “ALU Event” as a record of a basic u32 or byte operation that occurred inside the virtual machine.
- Includes opcodes like ADD, SUB, MUL, DIV, REM, AND, XOR, OR, etc.
- **Prover needs to prove that the operation was correct!**



High-level overview of the entire SP1 proving pipeline

How does SP1 turn an execution trace into a proof?



STARK Primer: Arithmetic Intermediate Representation

- An AIR P over a field F has length n and width w.
- P is defined by a set of *constraint polynomials* $\{f_i\}$ *over* $2w$ *variables.*
- A witness W for P consists of n vectors of length w over elements of F. We think of these as rows of width w.
- W is *valid* if submitting the $2w$ values from any two consecutive rows to any constraint polynomial f_i evaluates to zero.

Fibonacci Table

1	1
2	3
5	8
13	21

$$f_1(X_1, X_2, Y_1, Y_2) = Y_1 - X_2 - X_1$$

$$f_2(X_1, X_2, Y_1, Y_2) = Y_2 - Y_1 - X_2$$

STARK Primer: Arithmetic Intermediate Representation

- Why do most zkVMs universally build on STARKs?
- AIRs map far more naturally on the notion of step-by-step execution than arithmetic circuits.
- Columns of AIRs can be thought as registers, while the constraint polynomials constraint a state-transition-function.
- Yields much leaner, more efficient proofs for long sequential computations.

Fibonacci Table

1	1
2	3
5	8
13	21

$$f_1(X_1, X_2, Y_1, Y_2) = Y_1 - X_2 - X_1$$

$$f_2(X_1, X_2, Y_1, Y_2) = Y_2 - Y_1 - X_2$$

STARK Primer: Arithmetic Intermediate Representation

Fibonacci Table

1	1
2	3
5	8
13	21

Start

State 1

State 2

End

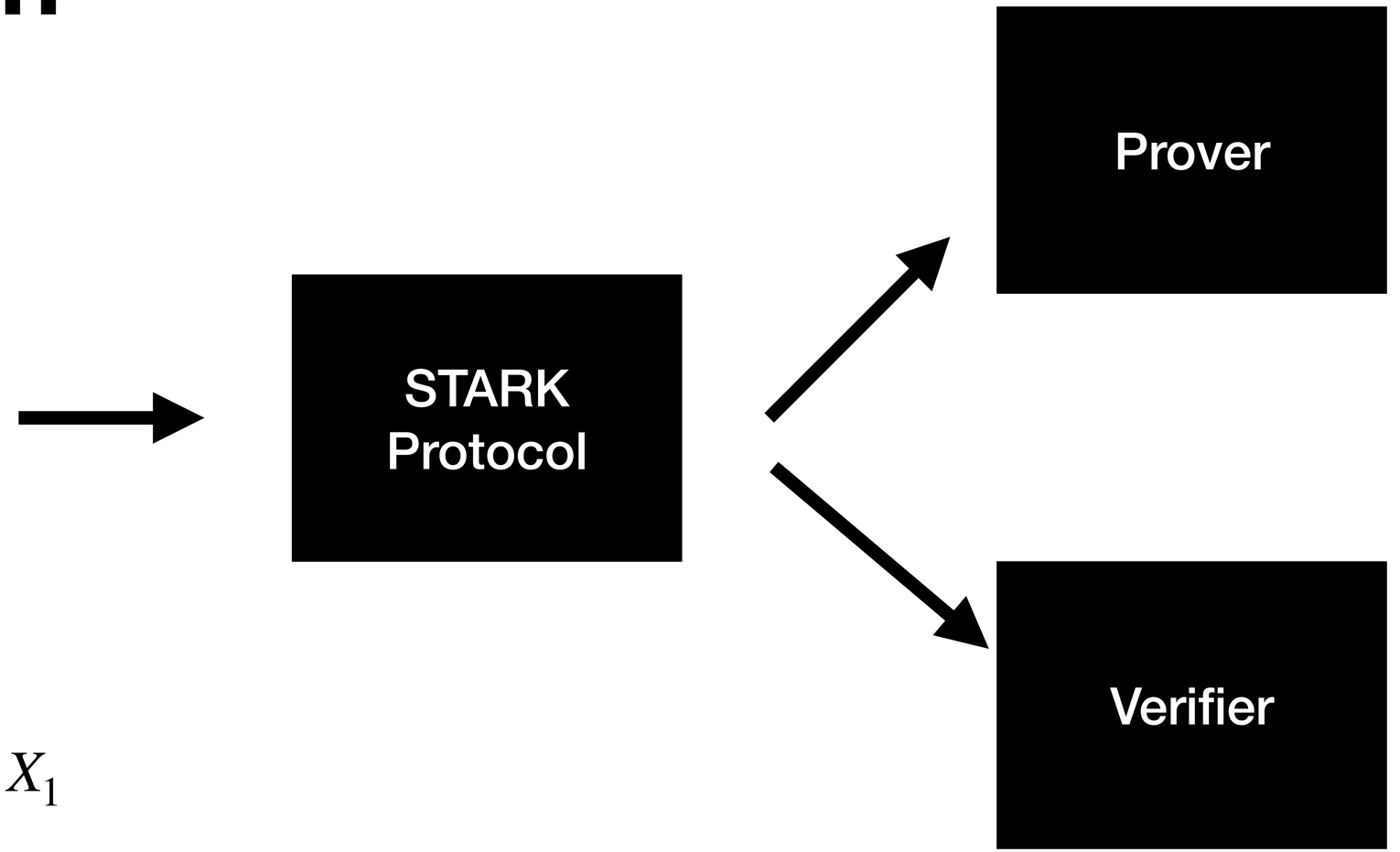
$$f_1(X_1, X_2, Y_1, Y_2) = Y_1 - X_2 - X_1$$

$$f_2(X_1, X_2, Y_1, Y_2) = Y_2 - Y_1 - X_2$$

STARK Primer: Arithmetic Intermediate Representation

Fibonacci AIR

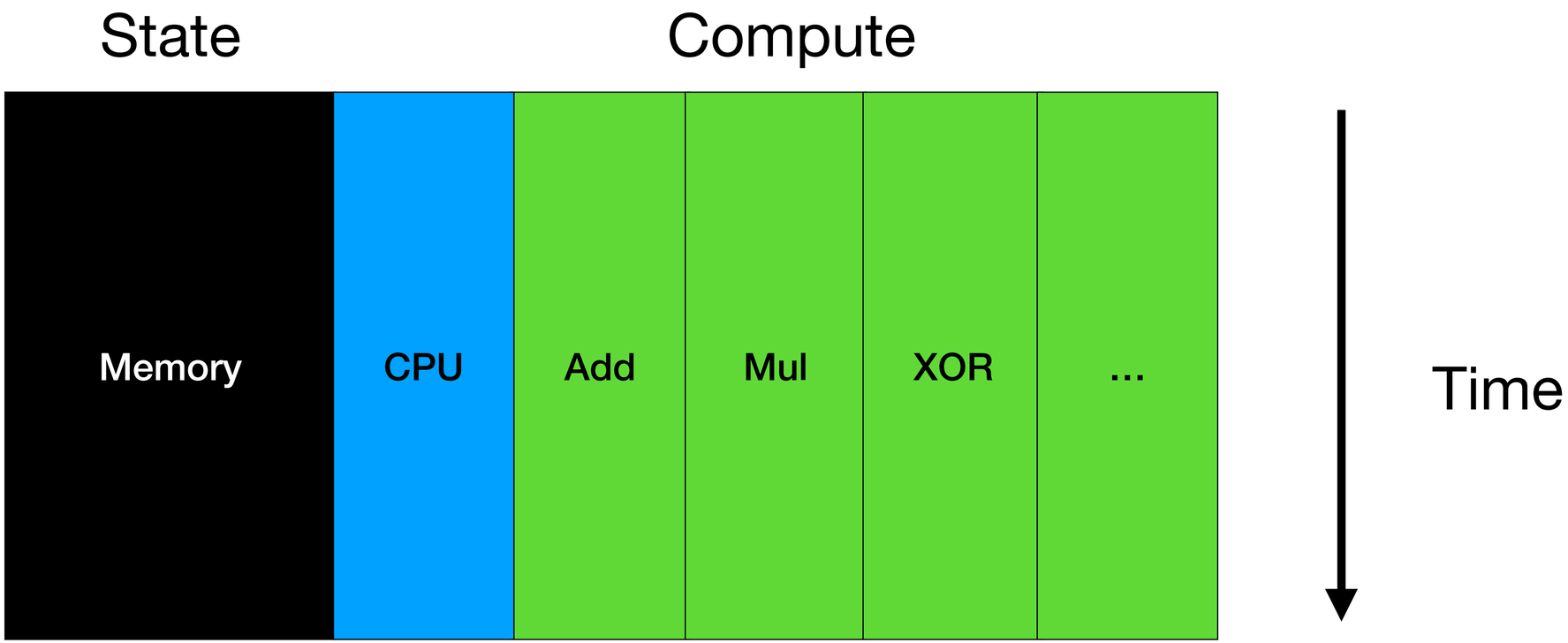
1	1
2	3
5	8
13	21



$$f_1(X_1, X_2, Y_1, Y_2) = Y_1 - X_2 - X_1$$

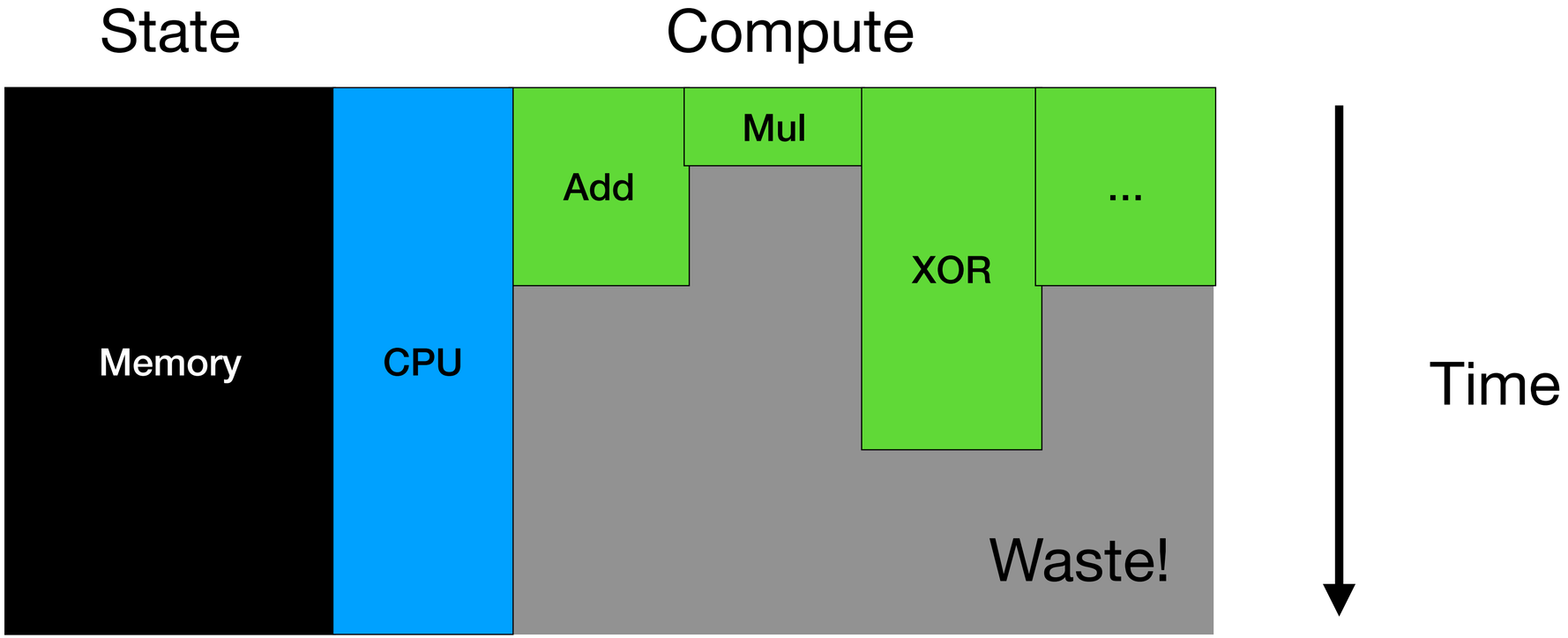
$$f_2(X_1, X_2, Y_1, Y_2) = Y_2 - Y_1 - X_2$$

Naive way to implement a RISC-V zero-knowledge virtual machine as a STARK



Proving Cost: $O(H \times W)$

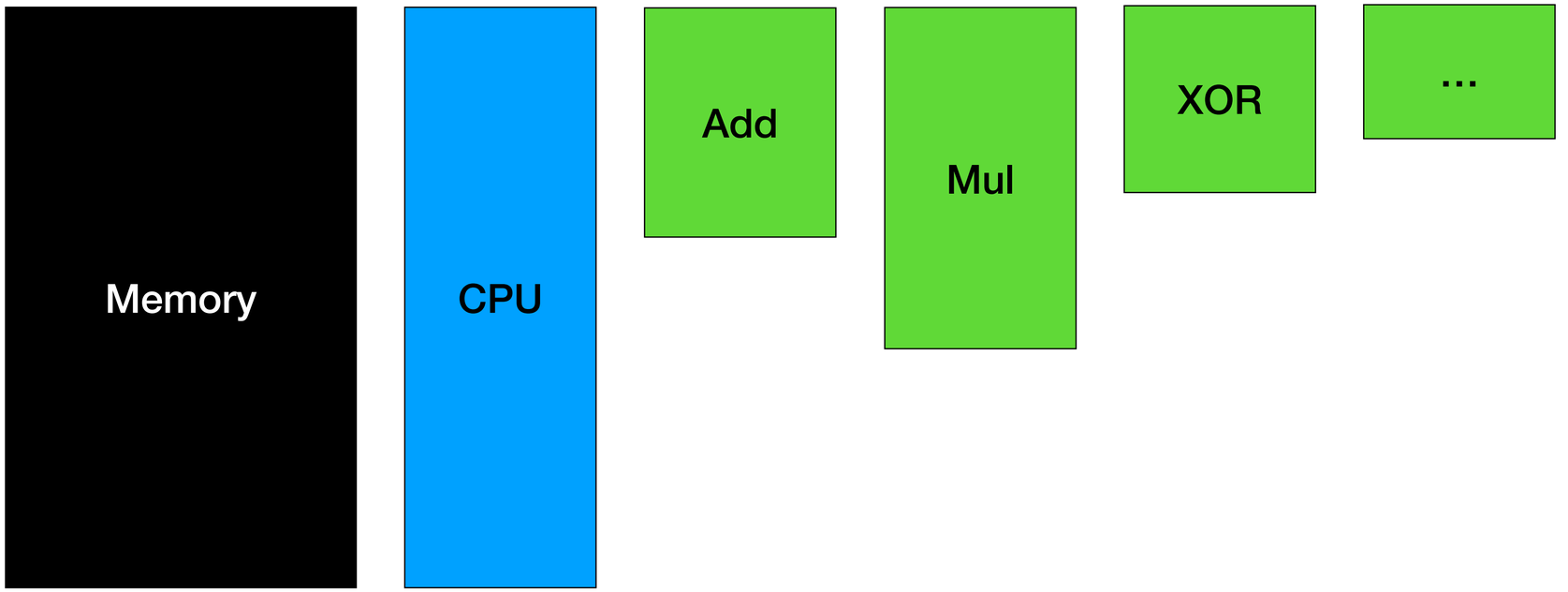
Naive way to implement a RISC-V zero-knowledge virtual machine as a STARK



Proving Cost: $O(H \times W)$

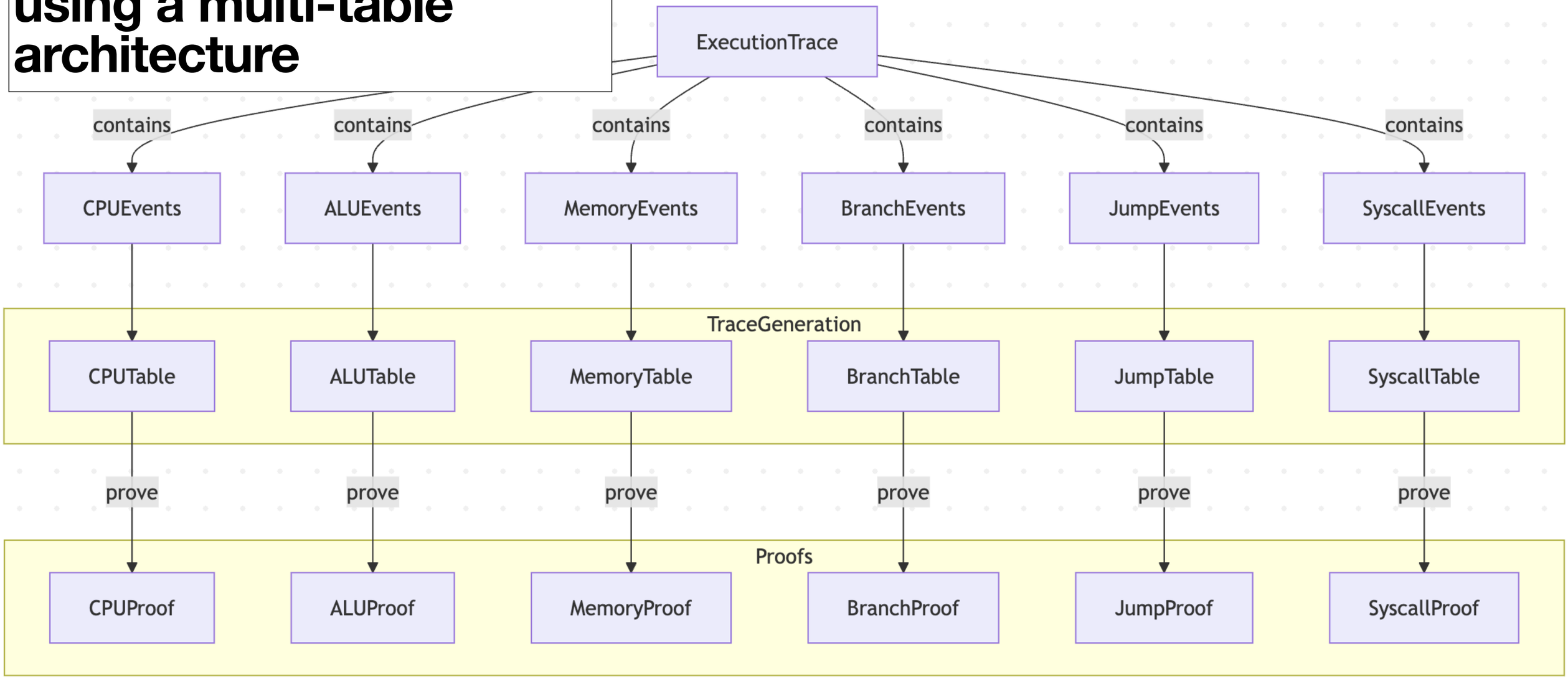
Extremely expensive!!!

SP1 proves RISC-V using a multi-table architecture

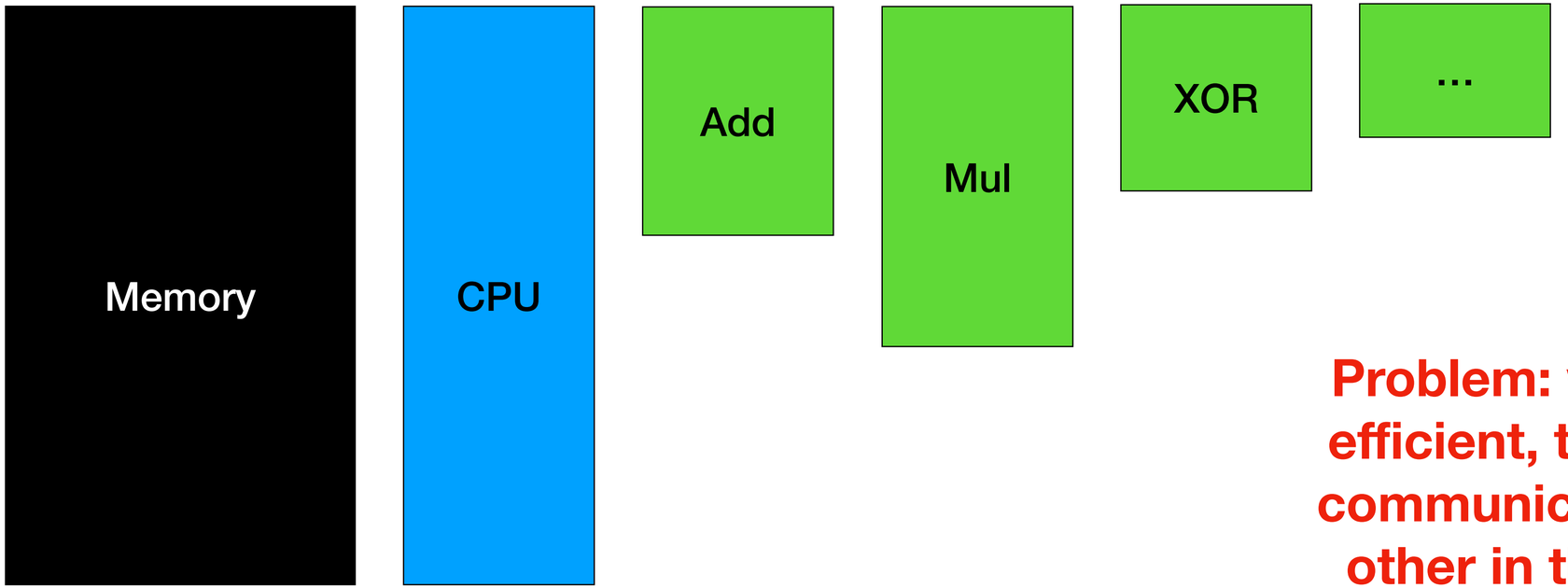


Proving Cost: $\sum_i O(H_i \times W_i)$

SP1 proves RISC-V using a multi-table architecture



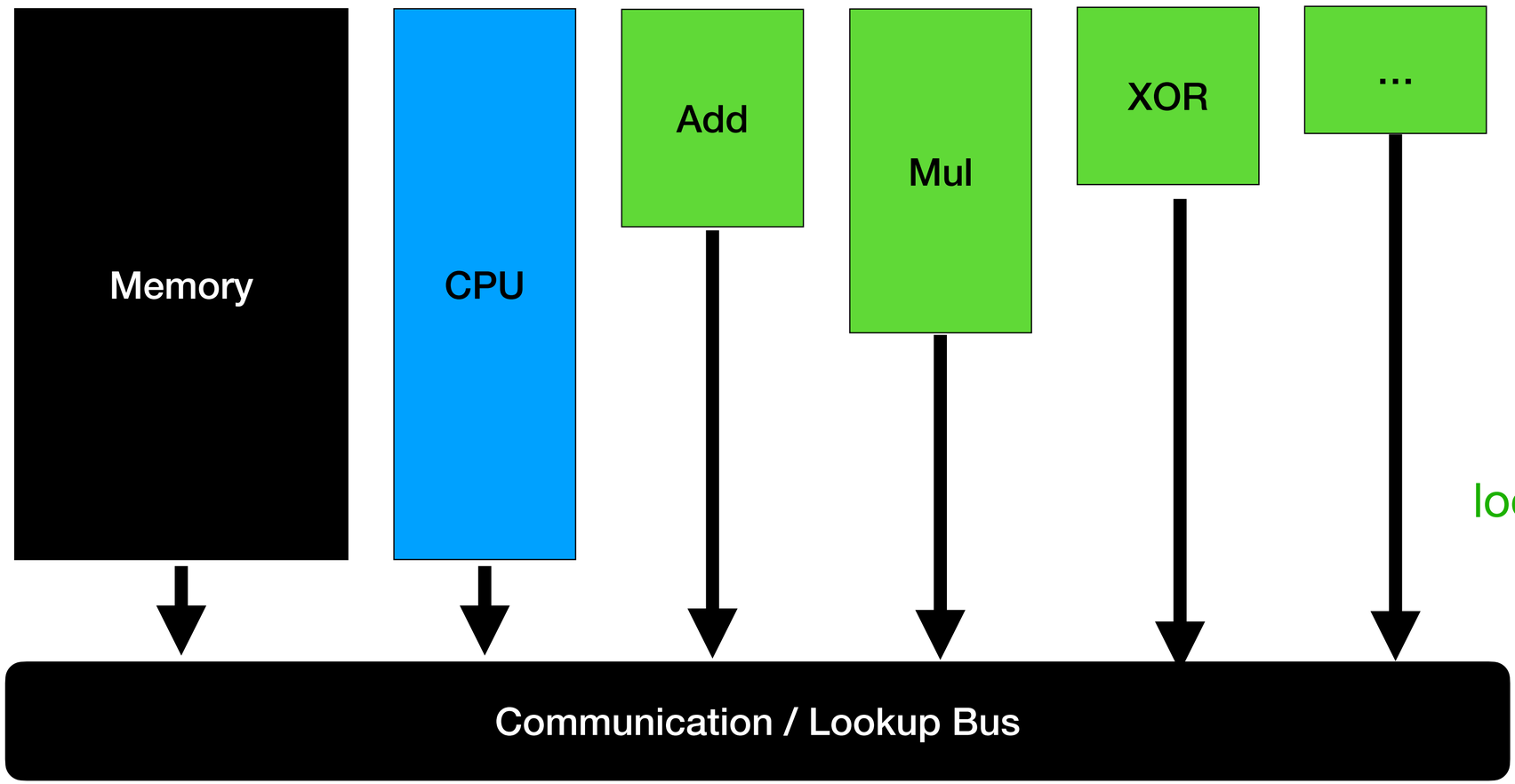
SP1 proves RISC-V using a multi-table architecture



Problem: while more efficient, tables can't communicate to each other in this setting

Proving Cost: $\sum_i O(H_i \times W_i)$

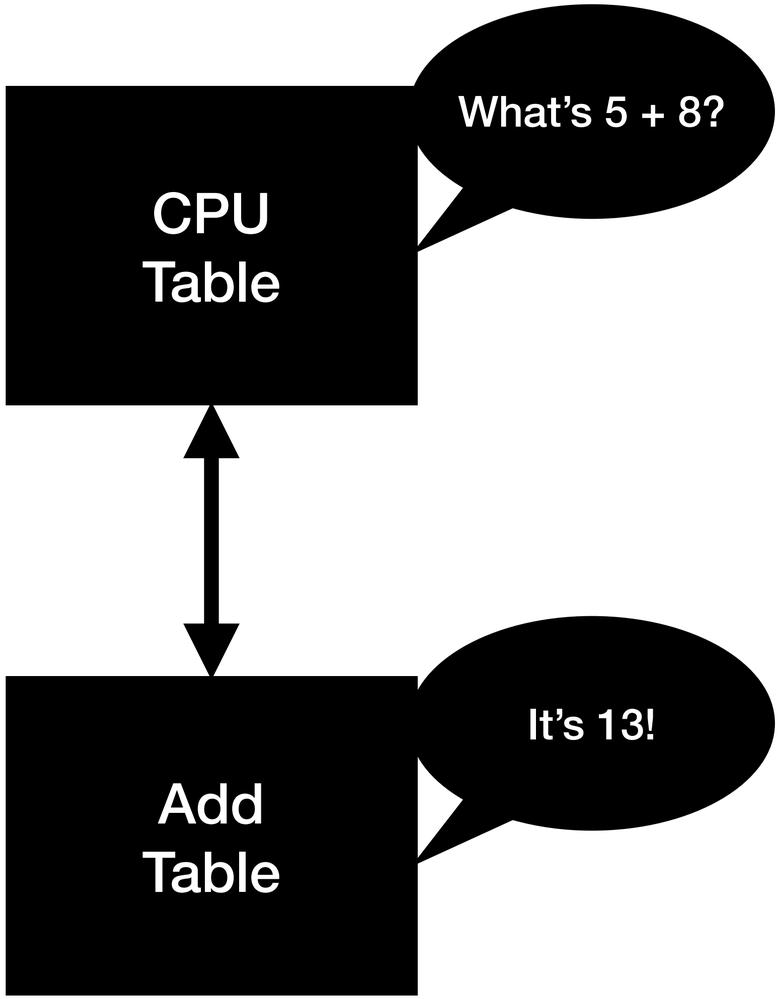
SP1 proves RISC-V using a multi-table architecture



Solution: use permutation / lookup arguments

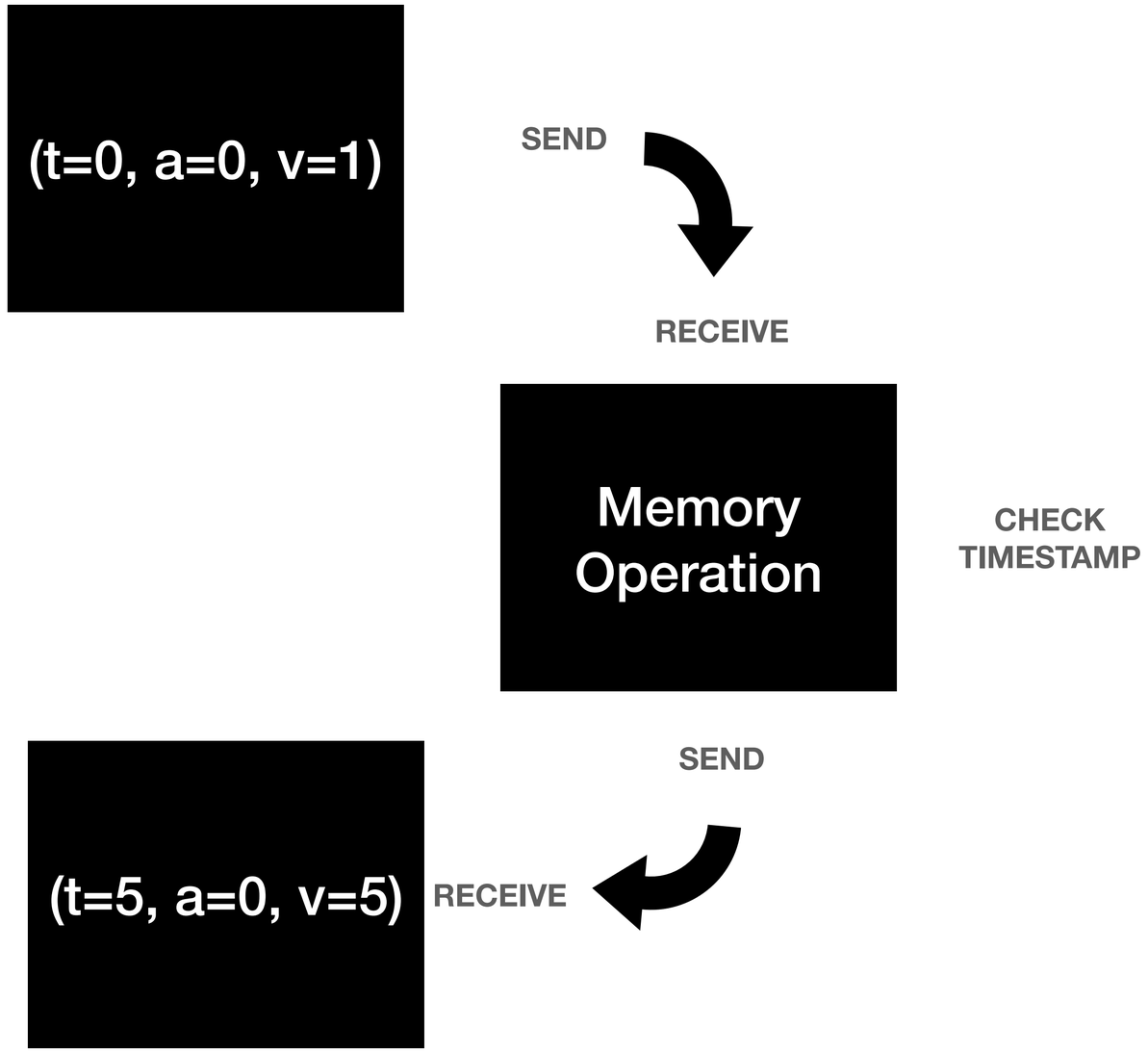
Communicating between tables using lookup arguments

- **Purpose:** A lookup argument enables a prover to convince a verifier that a value exists in some set efficiently.
- **In the context of zero-knowledge virtual machines, think of it as a way to “send” and “receive” messages.**
- Typically these arguments are implemented through grand product arguments or logarithmic derivatives.
- They typically add some amount of fixed overhead per table, but it’s negligible compared to the savings of a multi-table architecture.



Lookup arguments also give us “memory”

- Instead of tracking the full memory state at every step, SP1 models memory using *read-write* logs
- Every memory access is recording as a triple (t, a, v): the timestamp, address, and value
- On every memory access, receive a message of the last entry, check that the timestamp was earlier, and then send a new entry into the lookup bus

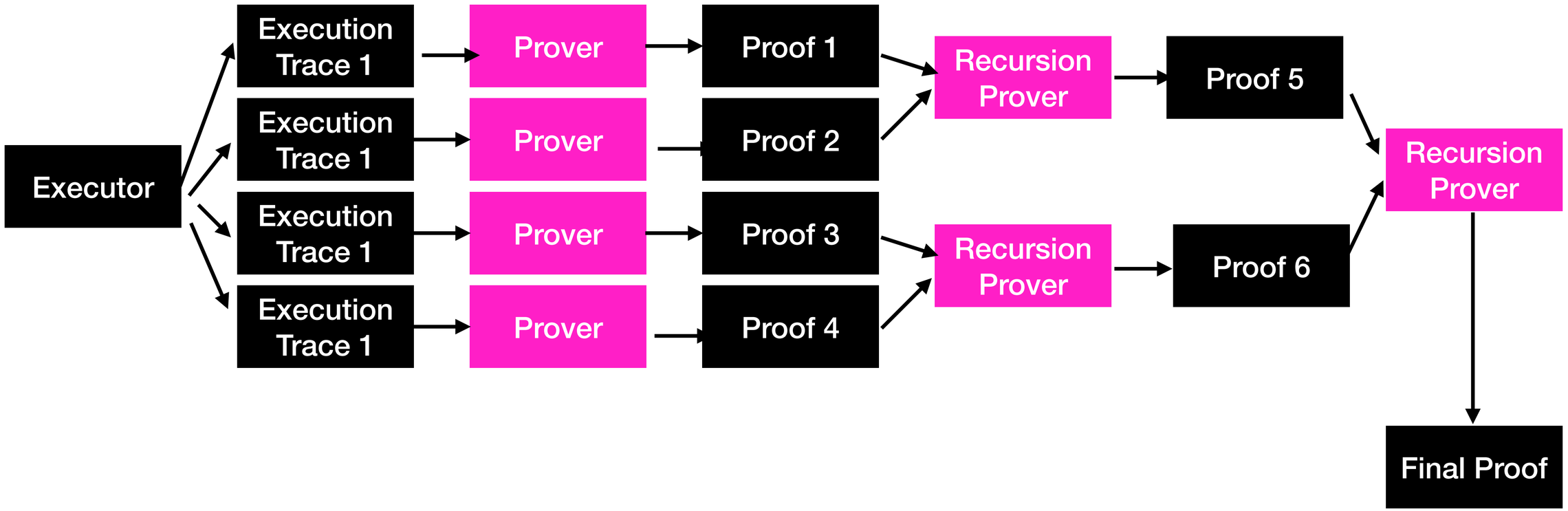


Putting it all together: we can now prove a small RISC-V program ~2M instructions

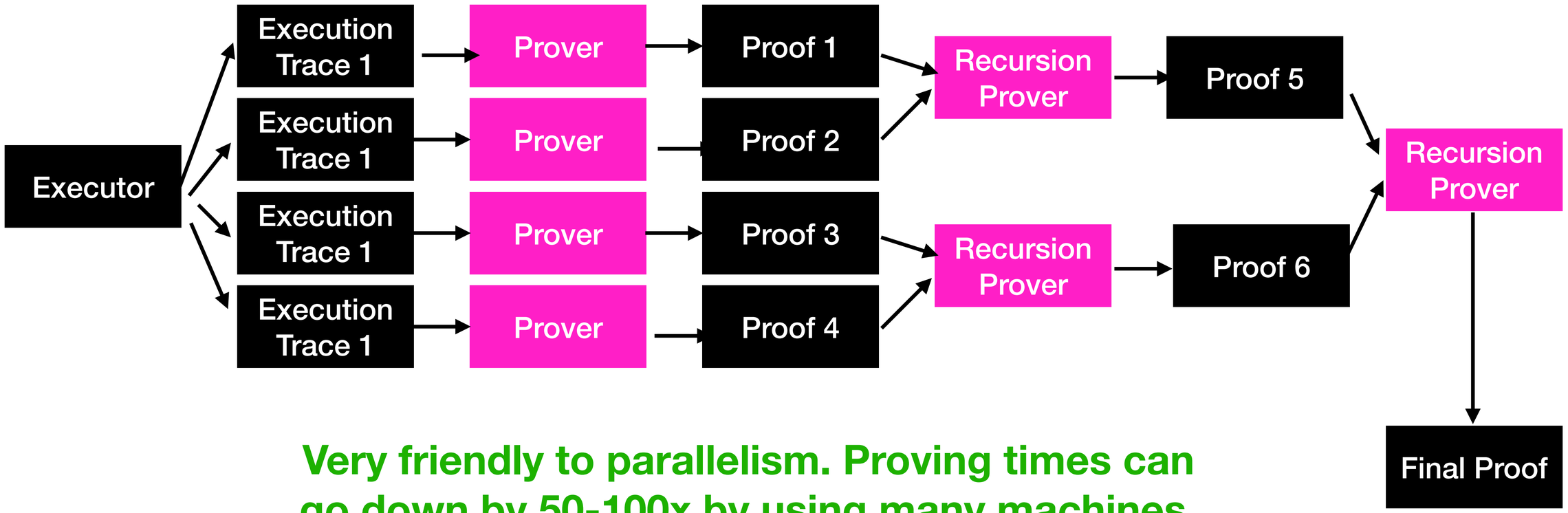
Putting it all together: we can now prove a small RISC-V program ~2M instructions

Problem: prover has linear memory requirements in # of instructions being proven

A prover for “small RISC-V programs” can be composed into a prover for arbitrarily long RISC-V programs



A prover for “small RISC-V programs” can be composed into a prover for arbitrarily long RISC-V programs



Very friendly to parallelism. Proving times can go down by 50-100x by using many machines.

Live Demo: SP1

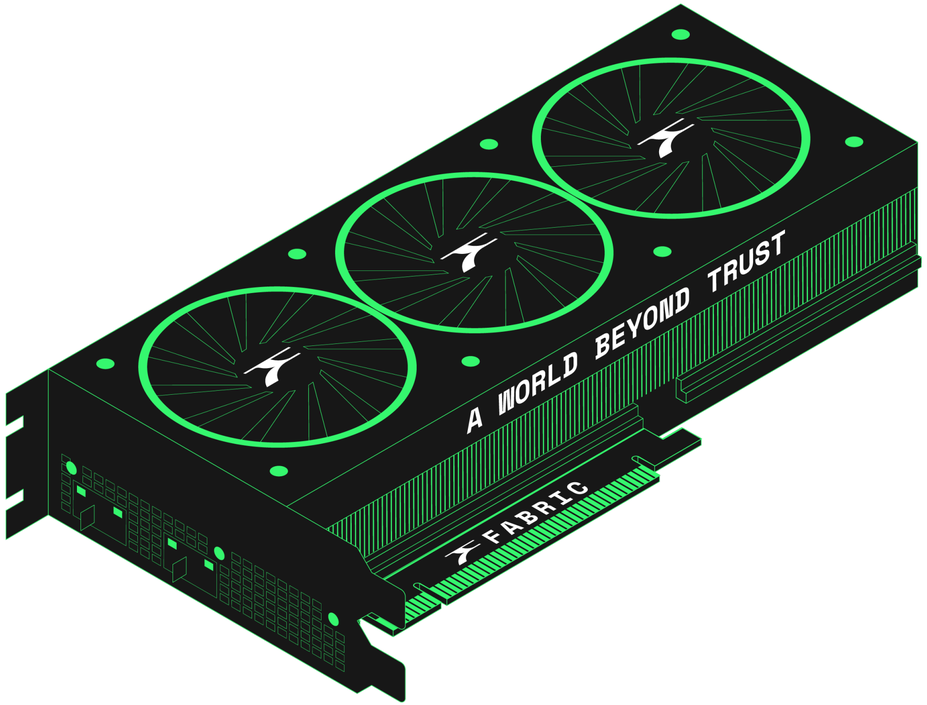
What’s next for zkVMs?

The frontier for R&D in zero-knowledge virtual machines in terms of cost

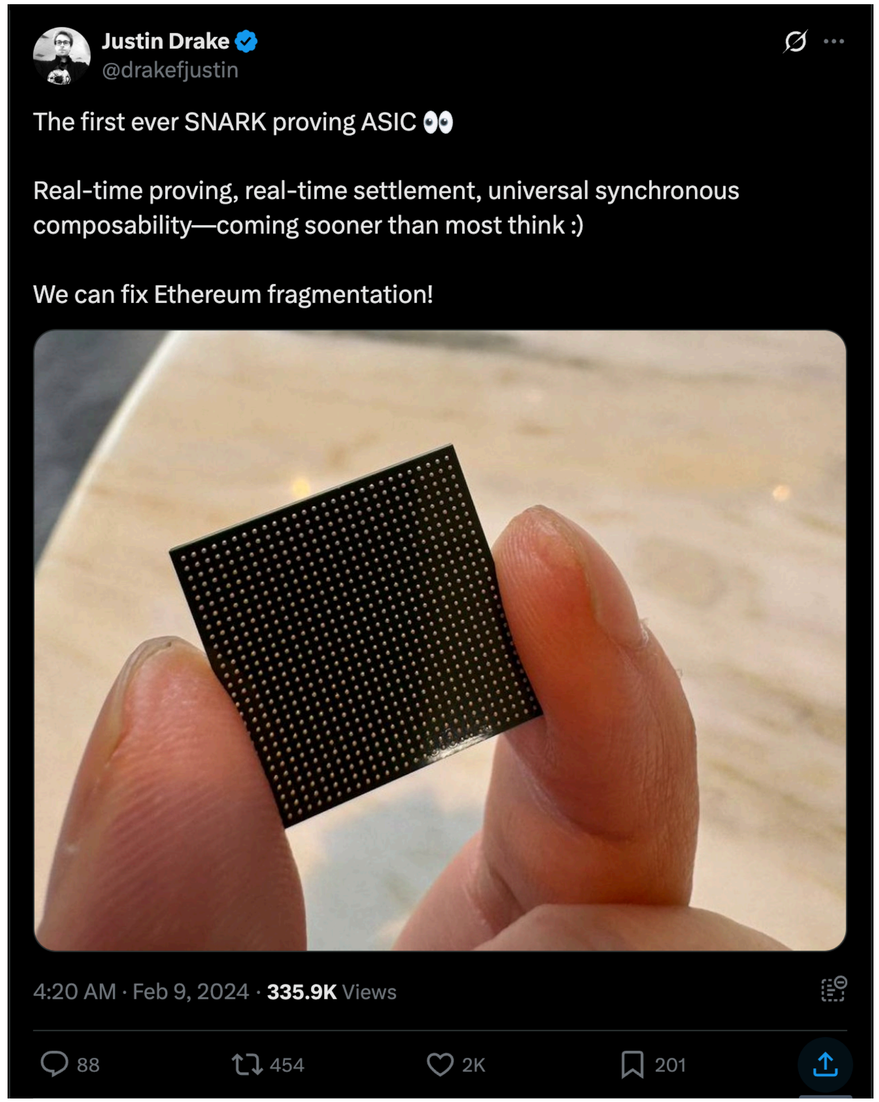
- **Algorithms.** New proof system techniques and tricks are constantly cutting costs.
- **Precompiles.** Enshrining certain circuits for commonly used operations such as hashing, elliptic curve operations, etc. can make certain workloads 10x faster. No “RISC-V overhead”.
- **Hardware Acceleration.** Most SOTA provers are now written to support NVIDIA GPUs, which can provide a 10x improvement in cost.

SHA256-Table

The frontier for R&D in zero-knowledge virtual machines in terms of cost



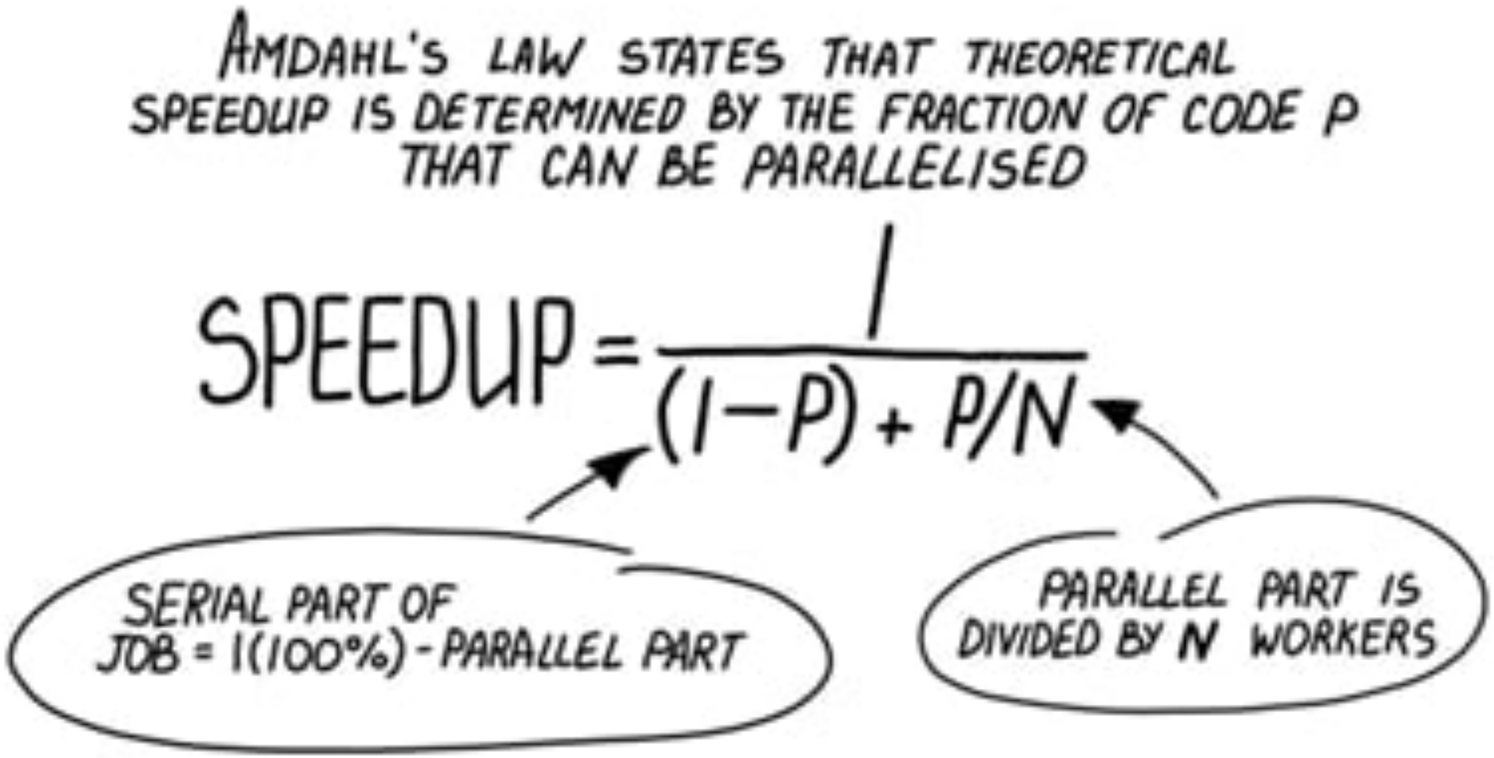
Fabric VPU



The frontier for R&D in zero-knowledge virtual machines in terms of latency

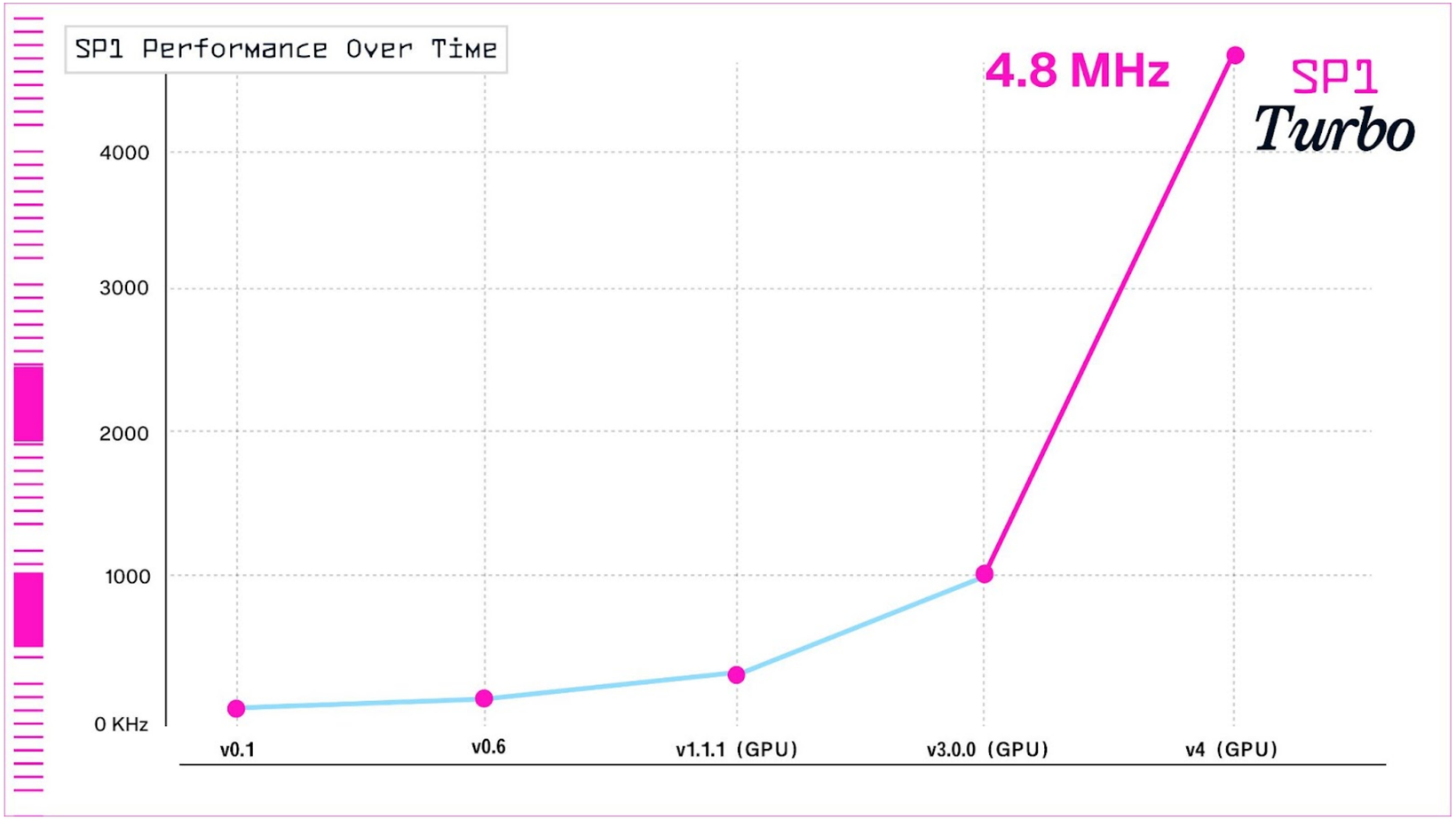
- **Lots of open interest in “real-time proving”.** Or generating proofs as fast as a normal computer can run code.
- **Provers that run in the cloud.** Coordinate thousands of GPUs to generate proofs as fast as possible.

One lesson: Amdahl's law is very real.



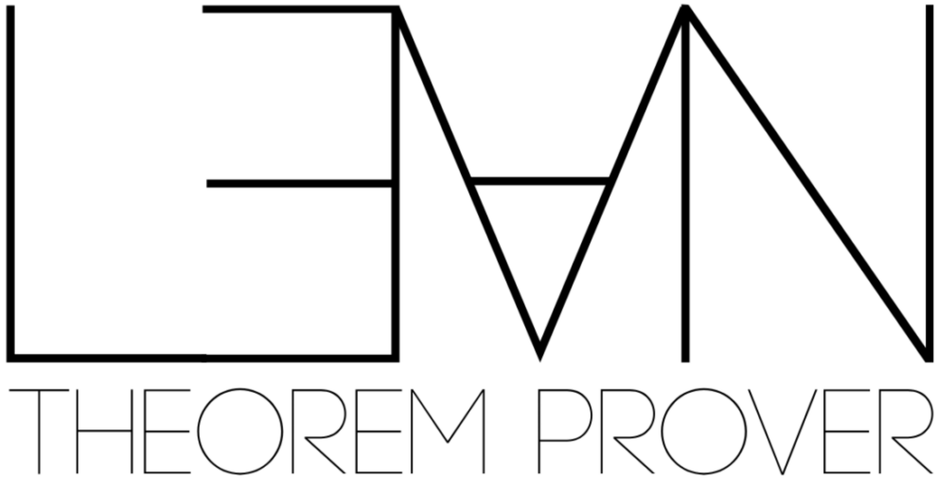
Performance engineering for real-world ZKP systems is somewhat detached from what's studied in academia. Constants on Big O analysis matter!

Another lesson: general-purpose systems allow you to invest heavily in performance



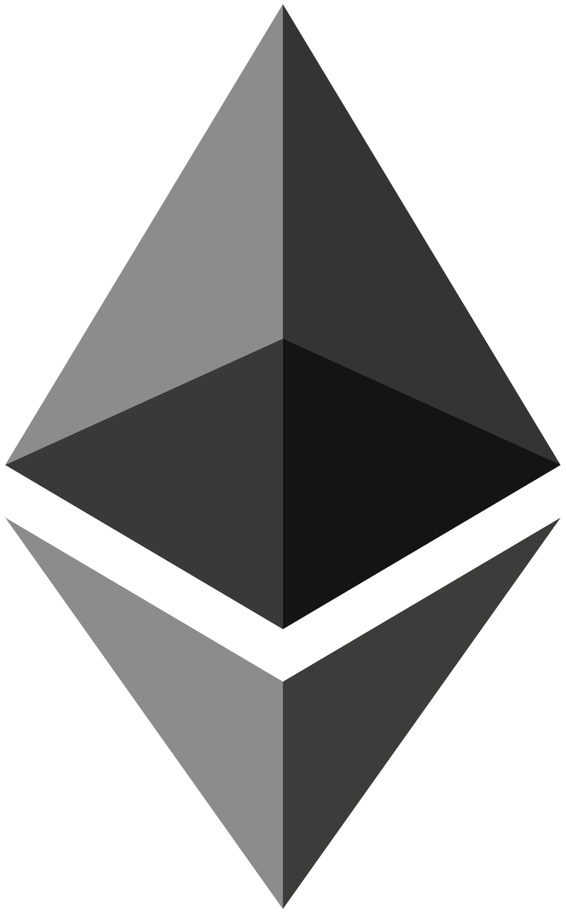
Ensuring that zero-knowledge virtual machines are secure

- Zero-knowledge virtual machines are some of the most complex cryptographic systems deployed in production
- Very prone to bugs that affect security and soundness
- Currently many systems are deployed with extra layers of security such as “approved provers” or “TEE 2FA”
- Lots of interest in “formal verification” as way to secure zkVMs



Emerging interest around prover markets

- Proof generation is becoming a critical dependency of many production systems securing billions of value
- In decentralized financial systems, censorship resistance is as important as safety (i.e., prevent users from repaying loans)
- **New idea:** decentralized infrastructure networks for proving with zkVMs
  - Competitive marketplace to drive proving costs down
  - Decentralized to ensure censorship resistance
- Lots of companies are building this, including us and many other players in the zkVM space



That’s it! Concluding thoughts

1. Traditional SNARK development is hard, error-prone, and similar to hardware design due to low-level circuit programming.

2. zkVMs abstract away that complexity, allowing developers to write SNARKs like normal programs in C++ or Rust.

3. Lots of interesting engineering and research problems to solve in this space, maybe some of you can help work on them!

Questions or comments?