Review of RISCV (and other) zkVMs

by Eduard Sanou from PSE (EF)

Overview

- RISCV intro
- What is a zkVM
  - A note on privacy
- Comparison Overview
  - Types of zkVM
- Architecture
  - Let’s take a look
  - Continuations
  - Coprocesor
- Benchmarks

RISCV intro

- Open CPU architecture following the Reduced Instruction Set Computer paradigm
  - Simpler than ARM64
- 32 bit and 64 bit flavors
- Multiple standard extensions
  - I: Base Integer ISA
  - M: Integer Multiplication & Division
  - A: Atomic
  - C: Compressed
  - F: Floating Point
- Support from major compilers: gcc LLVM (Rust, clang, …), golang

```
.section .text
.global strlen
strlen:
    # a0 = const char *str
    li      t0, 0          # i = 0
1: # Start of for loop
    add     t1, t0, a0     # Add the byte offset for str[i]
    lb      t1, 0(t1)      # Dereference str[i]
    beqz    t1, 1f         # if str[i] == 0, break for loop
    addi    t0, t0, 1      # Add 1 to our iterator
    j       1b             # Jump back to condition (1 backwards)
1: # End of for loop
    mv      a0, t0         # Move t0 into a0 to return
    ret                    # Return back via the return address register
```

What is a zkVM

- VM: Machine that behaves like a computer. Can do arithmetic ops, bitwise ops, branching, read/write memory, etc.
- zkVM: a system that can generate zk proofs of correct execution under a VM.
  - Execution trace: chronological list of instructions (with input-output) that are executed when running a VM program.
  - A zkVM requires: a circuit that verifies the correctness of an execution trace of the VM.
    - Realized by repeating the following verifying logic on each trace step:
      - Fetch, Decode?, Execute, Memory I/O

A note on privacy

- Is there really zk in these zkVMs?
- No privacy in most of the zkVM projects, so the term zk can be misleading: **there's no zero knowledge**.
- Usually the goal is **succintness** (the "S" from SNARK).

“Verifiable computing (colloquially mis-termed ZK) is an incredibly powerful technology” [1]

[1] https://a16zcrypto.com/posts/article/building-jolt/

Comparison Overview

Project	ISA	Arithmetization	circuit framework	Proof System	Field
RiscZero	RV32IM	AIR+Plonkish	zirgen [10]	STARK	BabyBear
SP1	RV32IM	AIR	Plonky3	STARK	BabyBear
Powdr	RV32IMAC [1]	Plonkish [5]	custom	pluggable [6]	pluggable [7]
Jolt	RV32I	R1CS+lookups	custom	Spartan + Lasso	BN254 scalar
zkm	MIPS32	AIR	Plonky2	STARK	Goldilocks
o1VM	MIPS32	Plonkish	Kimchi	Kimchi (PLONK)	BN254 scalar
zkWASM	WASM	Plonkish	halo2	halo2	BN254 scalar
Nexus zkVM	NexusVM [3][4]	R1CS	custom	(Hyper)Nova	255 bits [9]
CairoVM	Cairo	AIR	custom	STARK	felt252
Valida	Valida [2]	AIR	Plonky3	STARK	31 bits [8]
Triton VM	Triton [11]	AIR+AET	custom	STARK	Oxfoi [12]

[1] https://github.com/powdr-labs/powdr/blob/3f771638467a8bd527805239ac614dd56e65f11c/riscv/tests/instruction_tests/README.md
[2] https://github.com/valida-xyz/valida/blob/898d5ce0581095eabdb5c7219f222cb17cdd46a9/opcodes/src/lib.rs
[3] https://docs.nexus.xyz/Specs/nexus-vm#nexus-virtual-machine-instruction-set
[4] https://github.com/nexus-xyz/nexus-zkvm/blob/a3a2b156b90276fc456a58f15fd46b638d2ca010/vm/src/instructions.rs#L22
[5] Powdr uses a PIL-like language, which allows extesions to Air making it Plonkish
[6] Currently implemented: halo2, eSTARK. Work in progresss: SuperNova.
[7] Goldilocks (63 bits) with eSTARK backend, 255+ bit fields with halo2 backend (for example: BN254 scalar field).
[8] Both BabyBear and Mersenne31 are 31 bit fields.
[9] Uses the BN254 and Grumpkin curve cycle
[10] zirgen is closed source. This tool allows compilation of circuits from a high level description; the output generated for RiscZero is public.
[11] https://triton-vm.org/spec/instructions.html
[12] Oxfoi is a 64 bit prime https://triton-vm.org/spec/isa.html#oxfoi

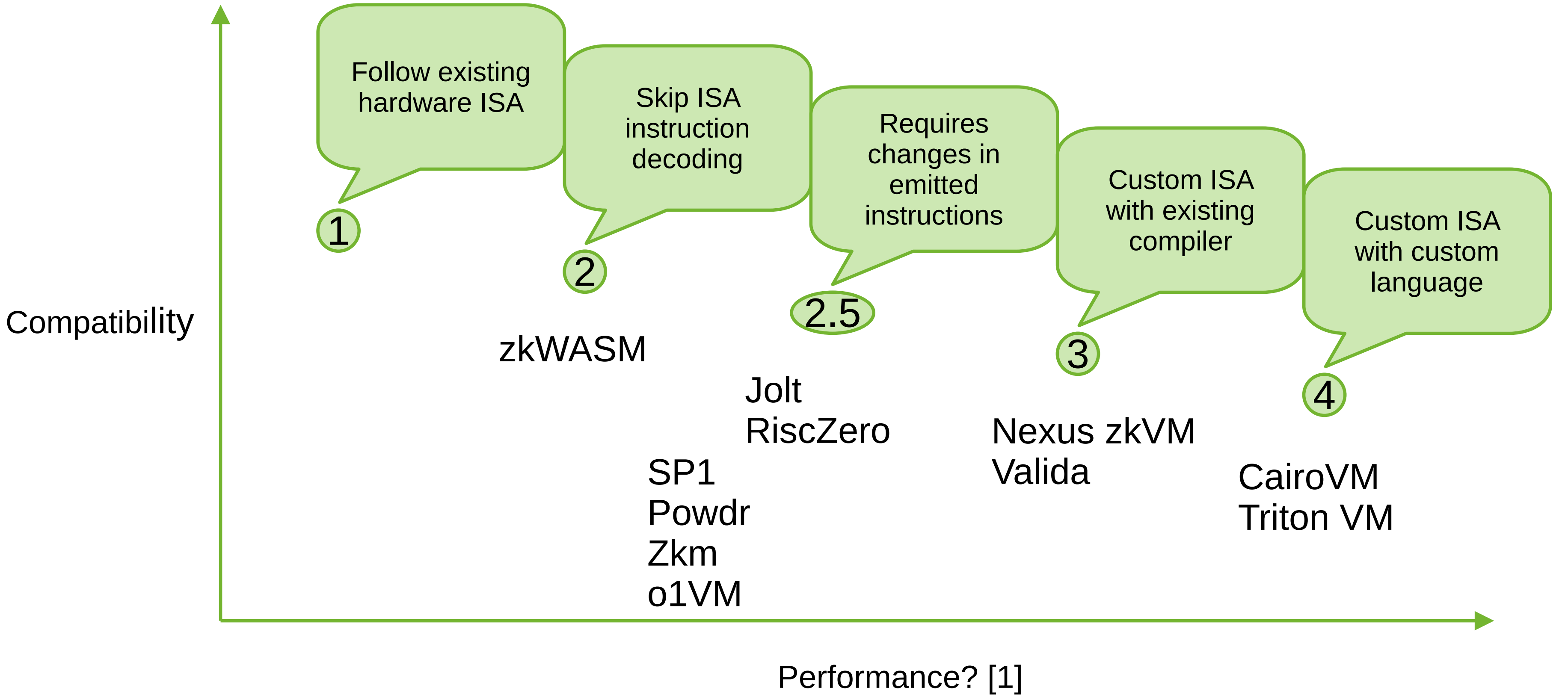
Types of zkVM

Vitalik defined a classification for zkEVMs [1]. This is my proposal for zkVMs.

- **Type 1**: Follows the full spec of an existing hardware ISA (RISCV, MIPS, etc).
  - Includes instruction decoding after fetching from memory.
- **Type 2**: Encodes instructions in a custom format.
  - Decoding to custom format done during compilation
  - Can’t write to memory and execute it (no JIT / AOT compilation)
- **Type 2.5**: Requires some changes on emitted instructions
  - Either via transformation in the decoding step or by patching the compiler.
- **Type 3**: Targets custom ISA (potentially zk friendly) but can be connected to existing compiler framework
  - Example: implements an LLVM backend for the custom ISA
- **Type 4**: Targets custom zk friendly ISA and requires custom high level language

[1] https://vitalik.eth.limo/general/2022/08/04/zkevm.html
Note 1: Vitalik’s categories distinguish matching gas cost. We could translate that to CPU cycles, but the RISCV ISA doesn’t specify the cycles per instruction (that’s implementation dependent).

Types of zkVM


Compatibility
Follow existing hardware ISA
1
Skip ISA instruction decoding
2
zkWASM
SP1
Powdr
Zkm
o1VM
Requires changes in emitted instructions
2.5
Jolt
RiscZero
Custom ISA with existing compiler
3
Nexus zkVM
Valida
Custom ISA with custom language
4
CairoVM
Triton VM
Performance? [1]

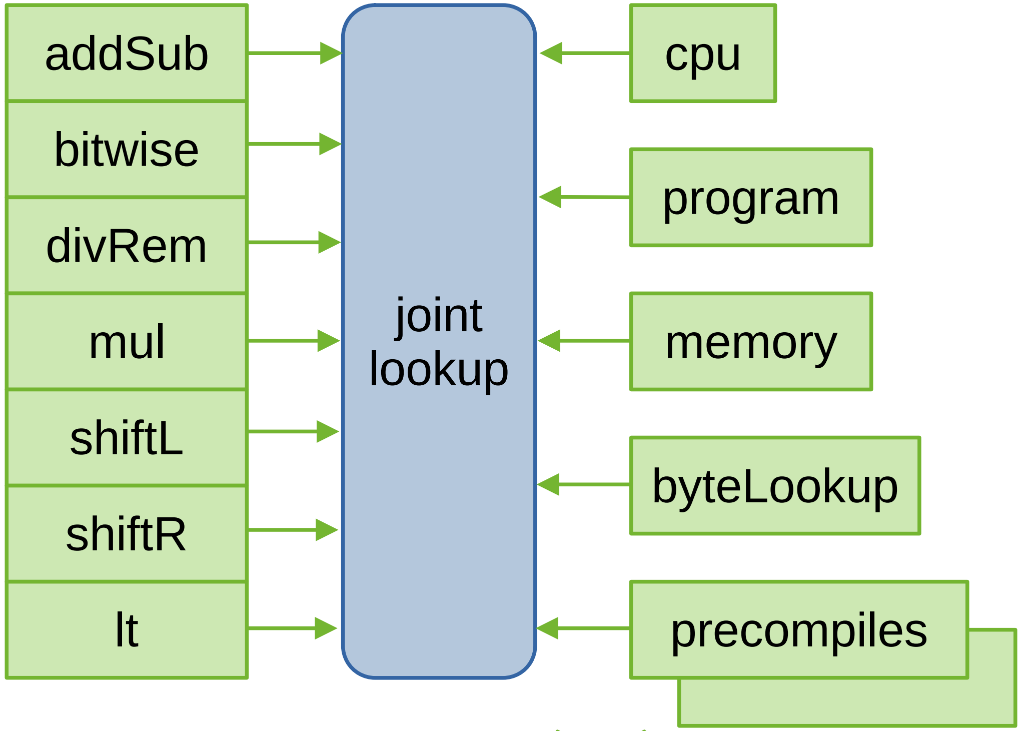
[1] Justin Thaler argues that a zk friendly ISA doesn't bring better performance
https://a16zcrypto.com/posts/article/faqs-on-jolts-initial-implementation/#section--8

Architecture

- Main blocks:
  - CPU: “Loop” logic that performs Fetch, Decode?, Execute, Memory I/O
    - Execute sometimes delegated to other blocks
  - Memory: Verify RAM read/write consistency
- Extras:
  - “std”: heap, stdin, stdout, threads, time, file system, ...
  - Continuations
  - Coprocessors

Architecture example: SP1

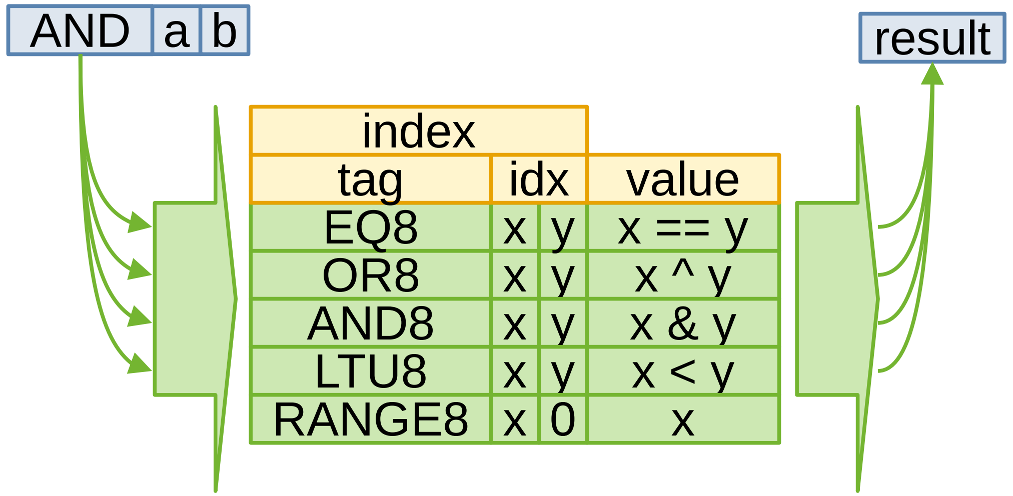
- Each chip defined with AIR constraints
- Chips do IO via joint lookup argument
- Cpu [1] is the entry point. AIR constraints verify 1 step per row:
  - Fetch instruction from program
  - Prepare instruction arguments
    - Maybe read from memory
  - Apply instruction
    - Possibly via lookup to another chip
  - Update Cpu state (registers, pc, timestamps, clk)
  - Maybe write result to memory



[1] https://github.com/succinctlabs/sp1/blob/5db203c55647c30618431822d33f419614f9fab6/core/src/cpu/air/mod.rs#L32

Architecture example: Jolt [1]

- Built with Lasso lookups
- Each RISCV instruction is defined with a decomposable lookup:
  - Decompose arguments into bit chunks
  - Use chunks for indexed lookups to small subtables
    - Tables need to be MLE [3]
  - Combine lookups results
- Merge all lookups, leading to a single decomposable lookup per instruction.
  - Merge all subtables into one
  - Merge all combine expressions into one
- Use R1CS for glue and some instructions



index			
tag	idx		value
EQ8	x	y	x == y
OR8	x	y	x ^ y
AND8	x	y	x & y
LTU8	x	y	x < y
RANGE8	x	0	x

[1] https://people.cs.georgetown.edu/jthaler/Jolt-paper.pdf
[2] https://people.cs.georgetown.edu/jthaler/Lasso-paper.pdf
[3] Table must be the evaluation of a Multi Linear Expression: an expression that takes n bits and can be evaluated in O(n)

Continuations

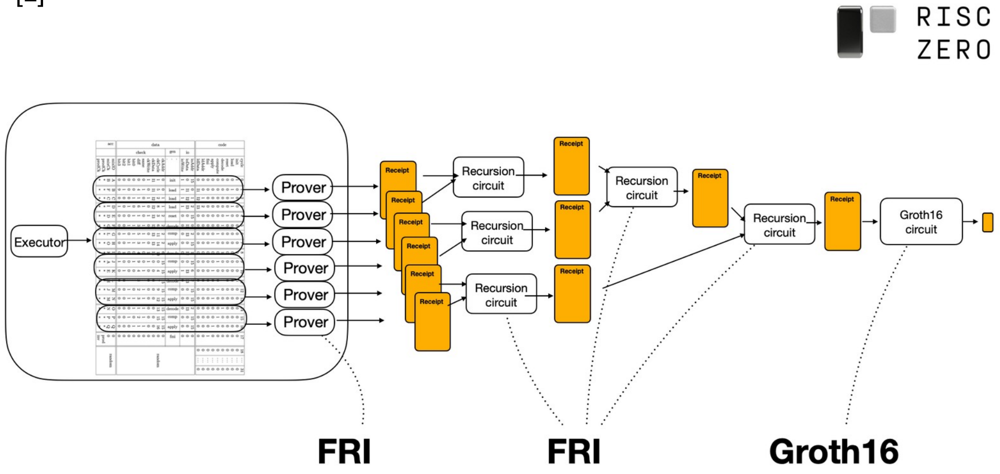
- Long computation may not be feasible in a single proof
  - High memory requirements
  - Proof system limitations
- Split the computation trace into chunks and generate a proof for each one
  - Possibly aggregating the proofs later for constant final proof size
  - Requires connecting the execution context between continuations
- Supported by: RiscZero, zkm, zkWASM, SP1, Powdr, zkWASM, Triton VM, Nexus (via folding) [1]

[1] Probably all zkVMs will end up supporting continuations

Continuations example: RiscZero

- Split execution trace into segments of equal number of cycles (steps)
- Between segments, memory is passed via a Merkle Tree in pages
- Inside segment:
  - Beginnig: used pages are “decommited” from MT
  - During: memory ops are verified via permutation check + sorted table [2]
  - End: used pages are “commited” to MT
- Each segment is proved independently (STARK w/ RISC-V Circuit)
- All segment proofs are aggregated recursively (STARK w/ Recursion Circuit)
- Recursion proof can be verified via Groth16 for on-chain verification (Groth16 w/ Groth16 Circuit)

[1]



[1] https://dev.risczero.com/proof-system/
[2] “sorting paradigm” in https://eprint.iacr.org/2023/1555.pdf

Coprocessor

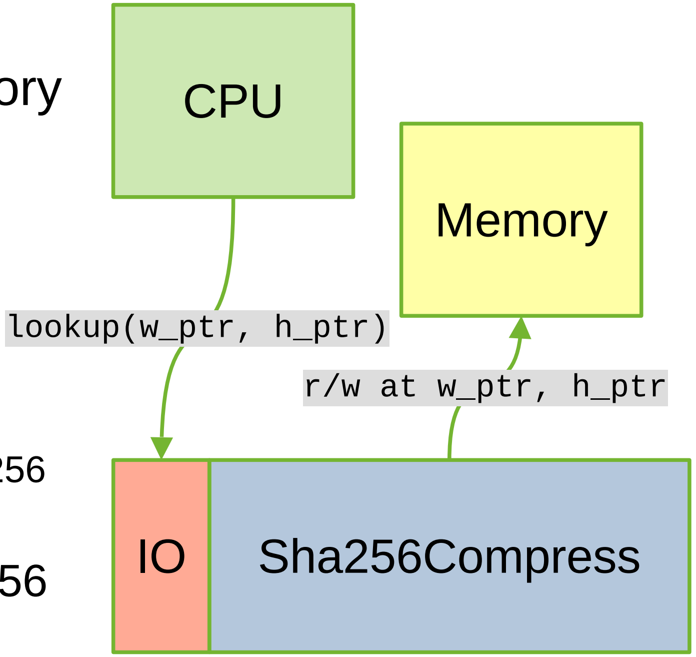
Also called “precompiles”

- Some programs can take a lot of steps when broken down to RISCV instructions
  - Can we accelerate the bottlenecks with custom circuits?
- Examples:
  - Cryptographic operations that require emulated fields (EC signatures, pairings, etc.)
  - Hashing (Keccak, Sha256, Poseidon, etc.)
- Supported by [1]: SP1, Nexus, Powdr,

[1] Probably all zkVMs will end up supporting coprocessors

Coprocessor example: SP1 [1]

- ISA level: accessed via system calls
- Circuit level: implemented with new AIR that shares memory with main VM. Accessed via lookup [2] with IO table.
- Example for sha256 compress [3]:
  - Cpu chip:
    - Lookup with *input word memory pointer*, *sha256 state memory pointer.*
    - Bumps memory timestamp by the number of memory ops the sha256 compress will do.
  - Sha256Compress chip: receives lookup and performs sha256 compression:
    - absorb word (read at *input word memory pointer)*
    - do hash operations
    - update state (read + write at *sha256 state memory pointer)*

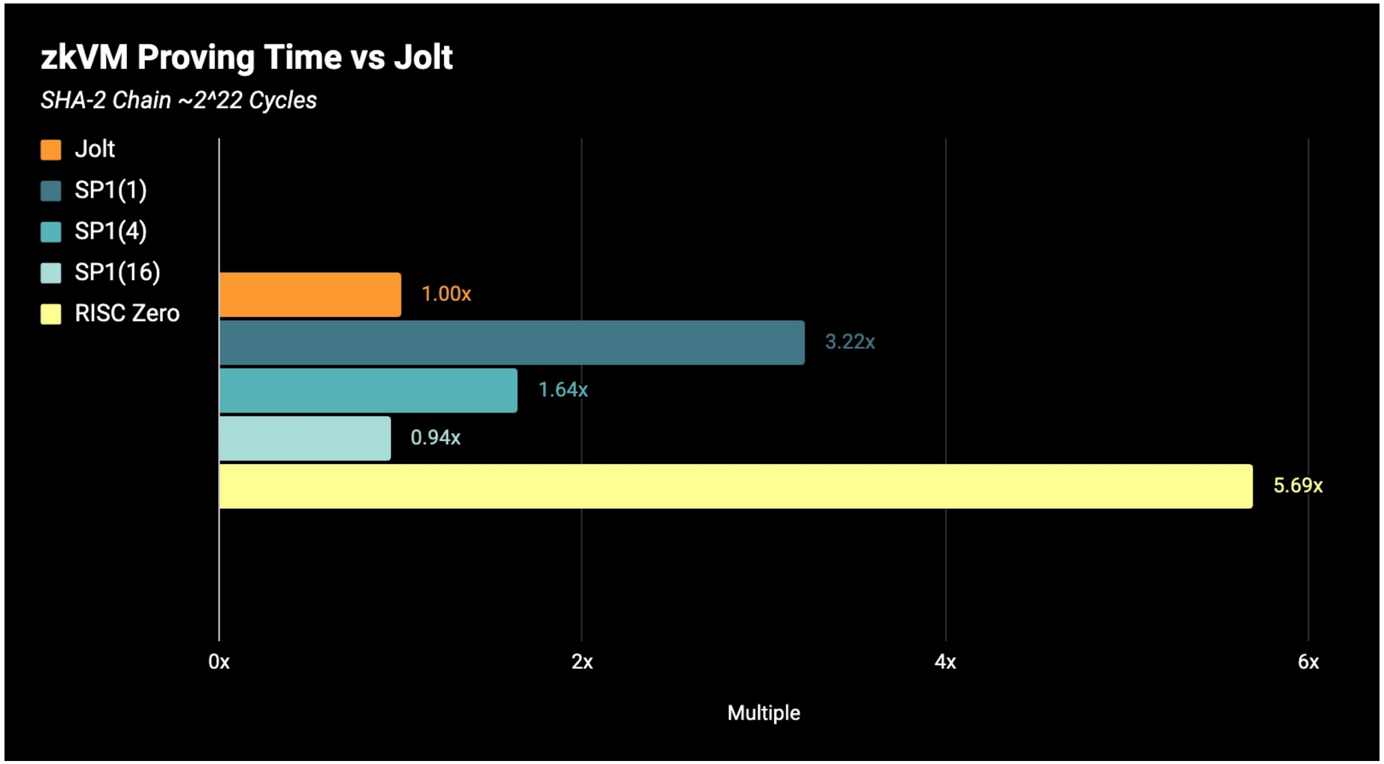


[1] https://succinctlabs.github.io/sp1/writing-programs/precompiles.html
[2] Technically it's a permutation check
[3] https://github.com/succinctlabs/sp1/blob/5db203c55647c30618431822d33f419614f9fab6/core/src/syscall/precompiles/sha256/compress/air.rs#L27

Benchmarks

- WARNING: Benchmarking is tricky because there are many subtleties:
  - CPU vs GPU implementations
  - Better/worse parallelism
  - Some zkVMs support continuations and other don’t
  - Some zkVMs support recursion and others don’t
  - Each zkVM may be in a different stage of optimization

(Partial) Benchmarks



Note: Value in parenthesis is number of shards

Results from https://a16zcrypto.com/posts/article/building-jolt/ at 2024-04-09 obtained via https://github.com/a16z/zkvm-benchmarks

Benchmarks

- Jolt can prove 100k instr/sec.
  - “[…] over twice the on-board computing power of the Apollo 11 mission.”
  - About 30x slower than a TI-84 graphing calculator.

Quote and numbers from Jolt authors at https://a16zcrypto.com/posts/article/building-jolt/ based on benchmarks run on an Apple M3 Max
[1] Jolt article says “150x slower”. But TI-84 uses a z80 @ 15Mhz; and in z80 each instruction takes 4+ cycles.

Questions?