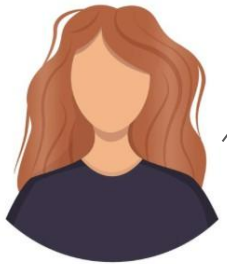
# Architecture-based Modifications for Zero-Knowledge Virtual Machines

Celine Zhang and Eric Archerman

Mentor: Simon Langowski

# Verifying computation

Peggy wants to show Victor that she correctly ran a hash function **without revealing her inputs**

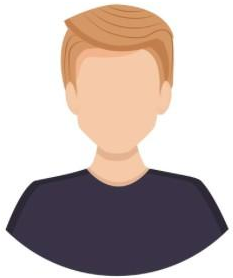


Peggy

Claim: output is 4xtusYz

```
fn sha3_chain(input: [u8; 32], num_iters: u32) -> [u8; 32] {
    let mut hash = input;
    for _ in 0..num_iters {
        let mut hasher = Keccak256::new();
        hasher.update(input);
        let res = &hasher.finalize();
        hash = Into::<[u8; 32]>::into(*res);
    }

    hash
}
```

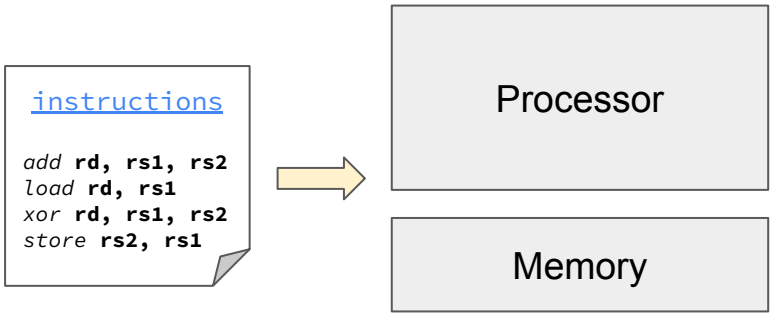


Victor

zkVMs enable this privacy-preserving verification for any computer program!

# Zero-Knowledge Virtual Machines (zkVMs)

zkVMs are cryptographic proof systems for the correct execution of programs on virtual machines.




Virtual machines execute instructions on a virtual processor with virtual memory

$$f(x) = f_0 + f_1 x + f_2 x^2 + \cdots + f_d x^d$$

$$com_f = g^{f(\tau)}$$

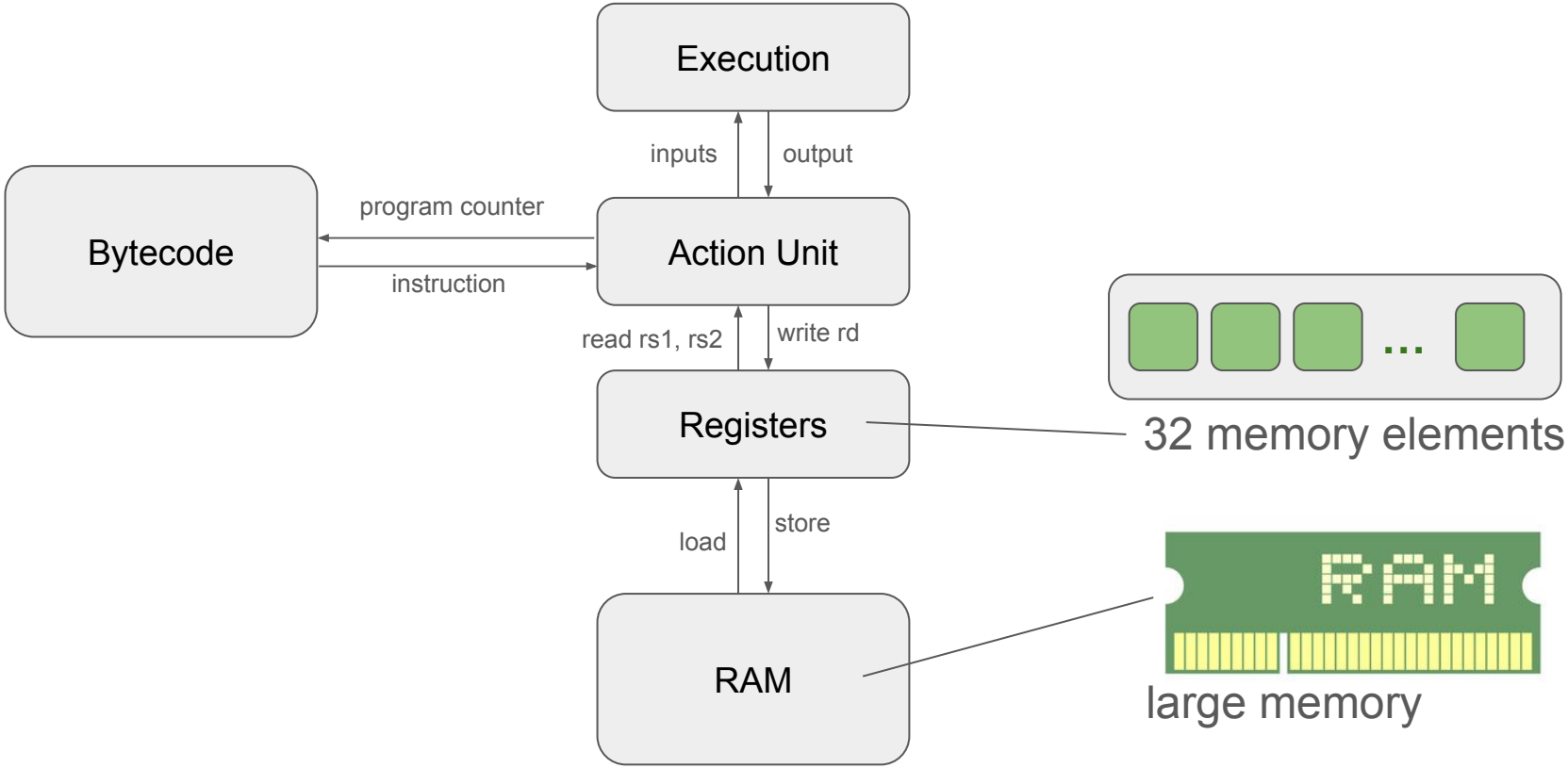
$$= g^{f_0 + f_1 \tau + f_2 \tau^2 + \cdots + f_d \tau^d}$$

$$= (g)^{f_0} \cdot (g^{\tau})^{f_1} \cdot \left(g^{\tau^2}\right)^{f_2} \cdot \ldots \cdot \left(g^{\tau^d}\right)^{f_d}$$

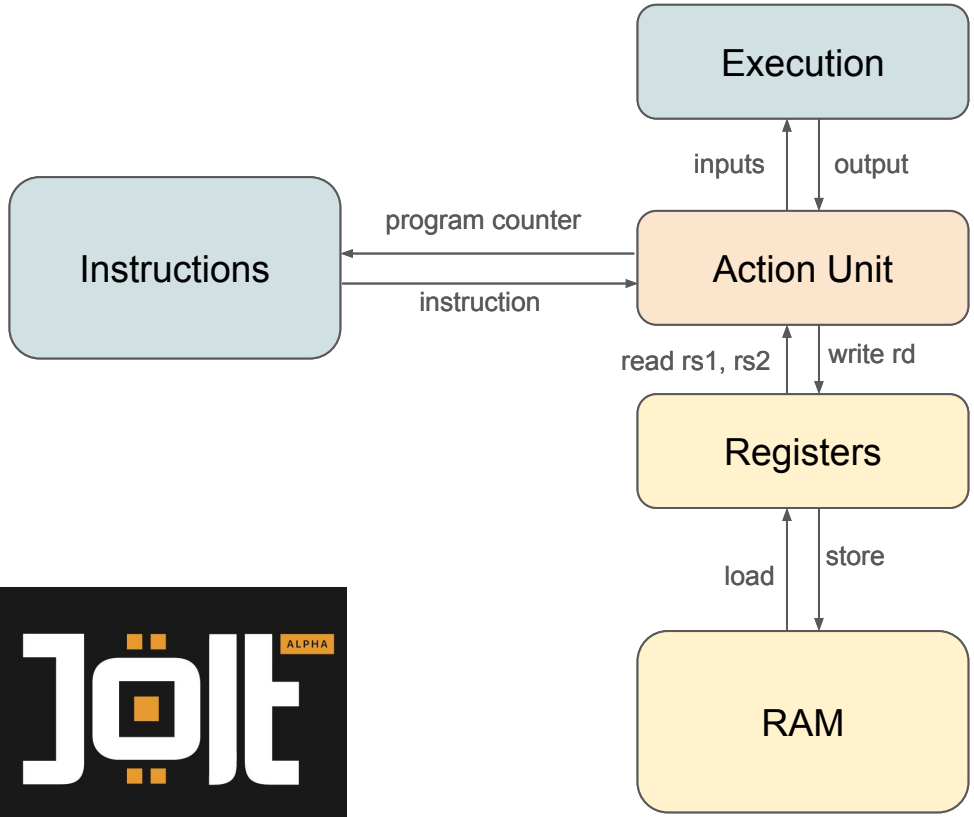
Cryptography gives two important properties:

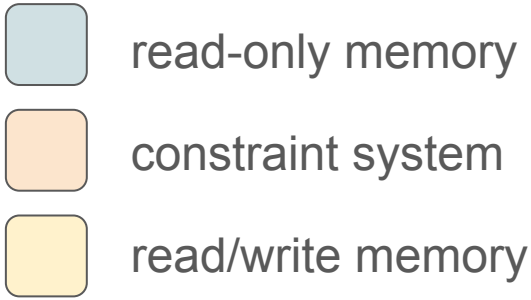
- succinctness (small, fast proofs)
- zero-knowledge (input hiding)

# Architecture of a virtual machine



# Jolt zkVM proof machinery

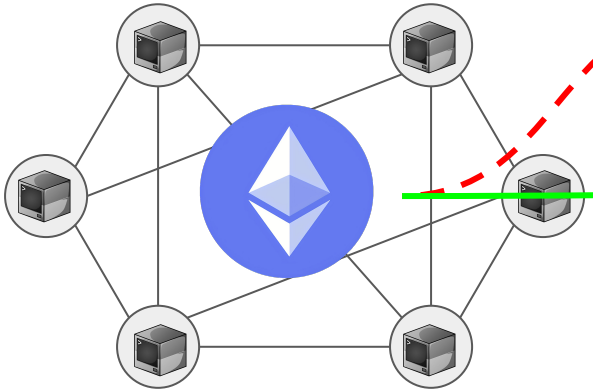


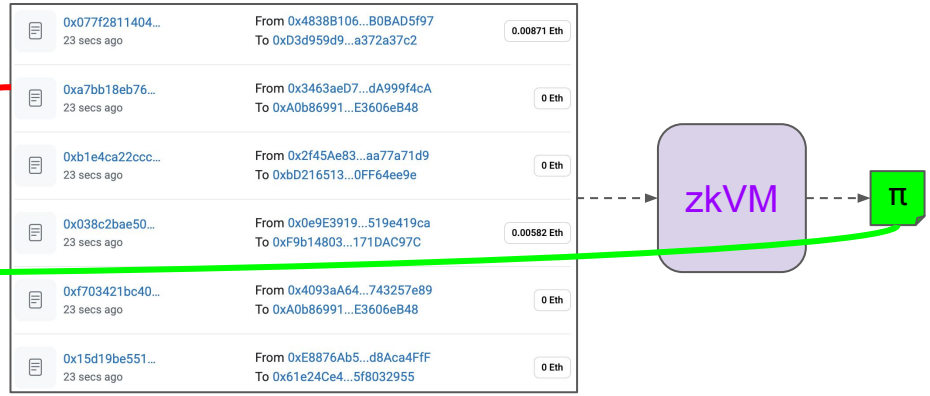




a zkVM uses different schemes to prove each part of the VM

# zkVMs today

blockchain scaling






lots of transactions, smart contracts, etc. → succinct, zk proof!

Ethereum from 19 to **10k** transactions per second

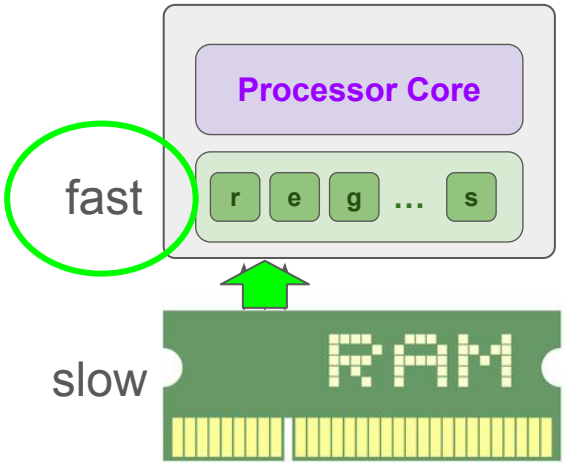
Why not more applications? prover overhead.

~5 orders of magnitude vs unverified computation

Problem: zkVM proof generation overhead

Question: Can insights from computer architecture help accelerate zkVMs?

# Optimizations in hardware




batched memory accesses

```
instructions
…
batchload(3-5)
load(3)

add(a, b) → c
mul(a, c) → d
add(c, d) → e

store(e)
```

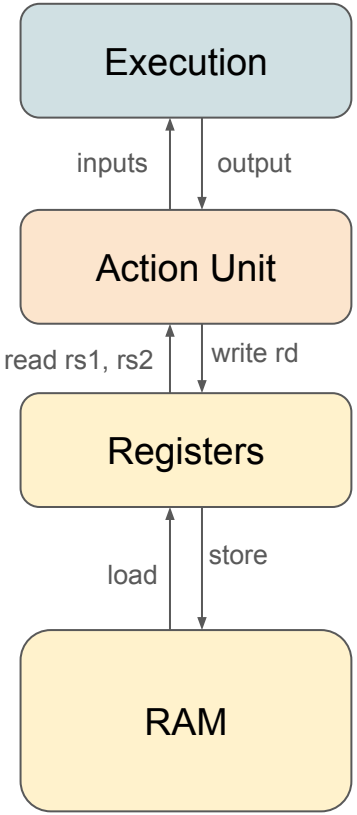
fast registers between instructions

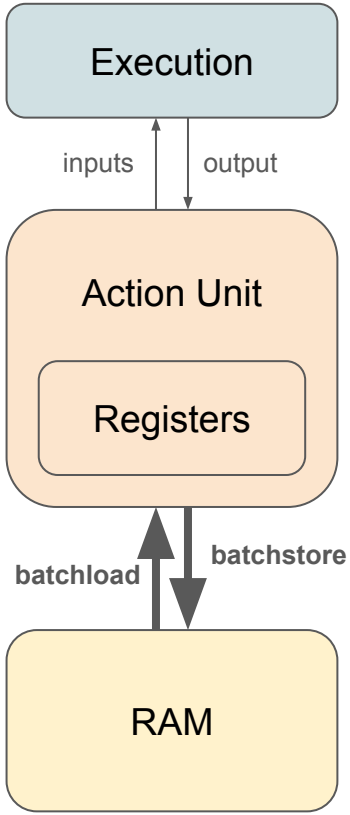
Optimizations take advantage of common programming patterns:

Memory accesses tend to be sequential

Outputs of instructions tend to be inputs to closely following instructions

# How these translate to zkVMs




read-only memory

constraint system

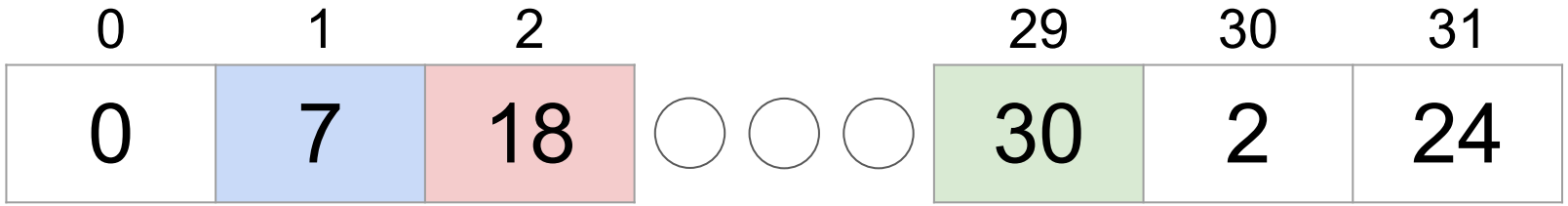
read/write memory

we need:

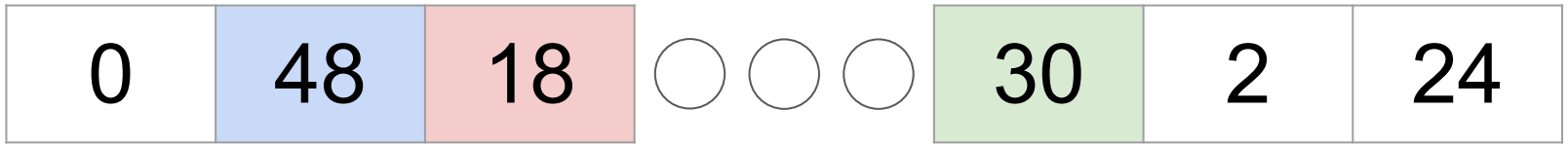
- adapt registers to the constraint system
- adapt constraint system and read/write memory to handle batching

# Register constraints

Before:



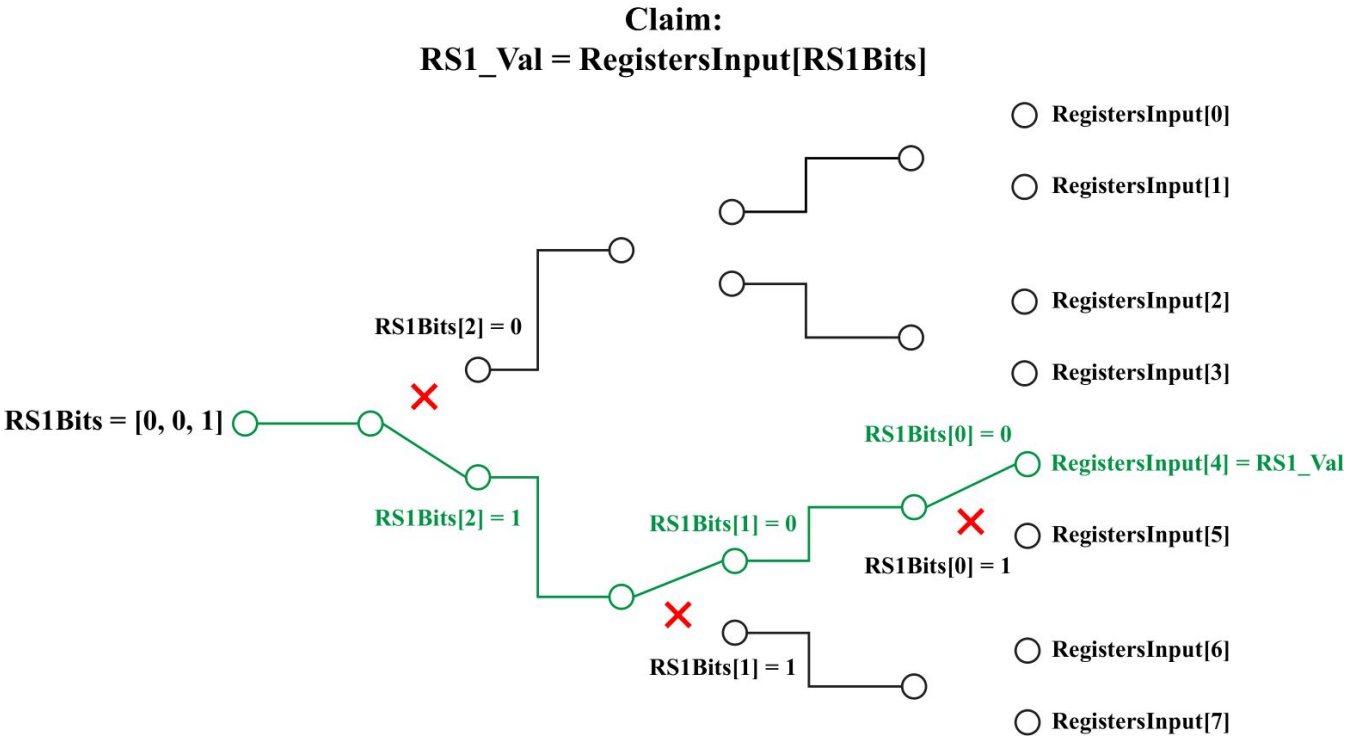

After:




RD RS1 RS2

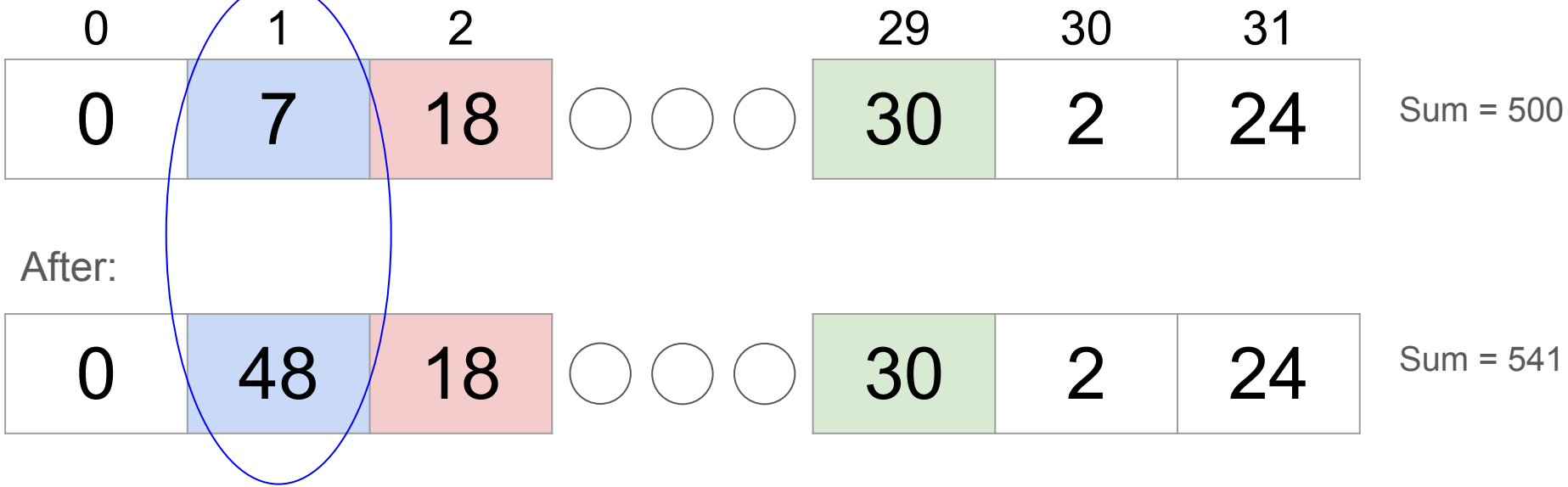
0=0✅ 7≠48 RD✅ 18=18✅ 30=30✅ 2=2✅ 24=24✅

# Register constraints



# Register constraints

Before:



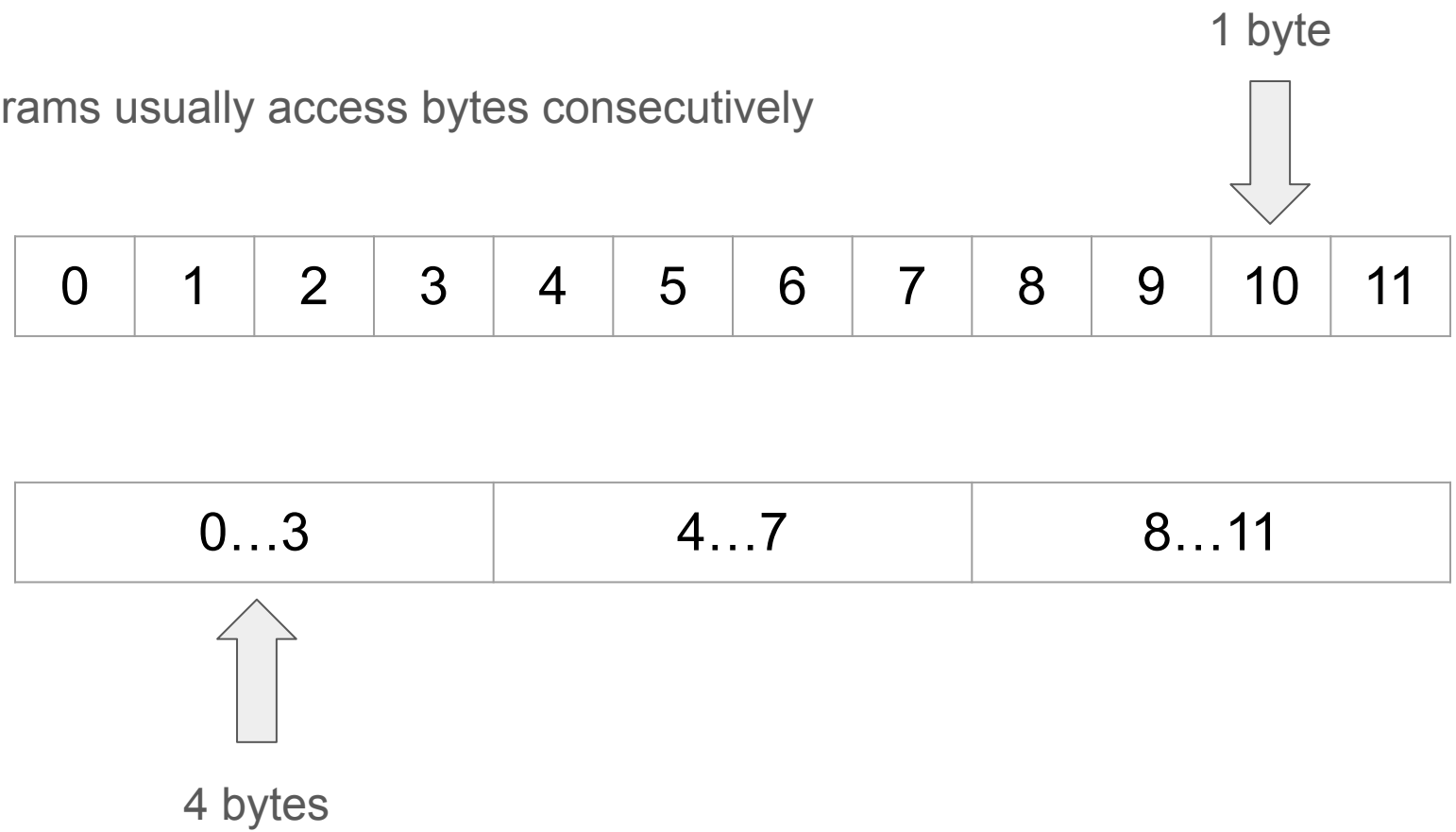

After:




541 - 500 = 48 - 7 ✅

# Memory batching

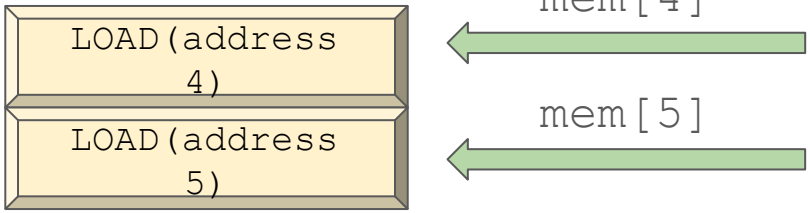
Programs usually access bytes consecutively

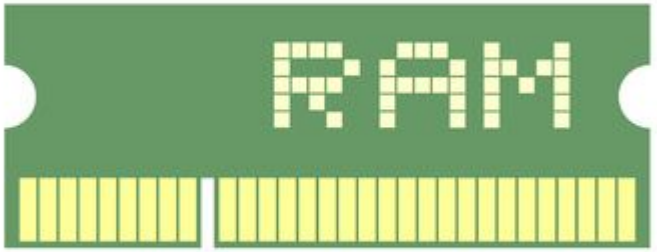


# Memory batching

Instruction: load addresses 4 and 5

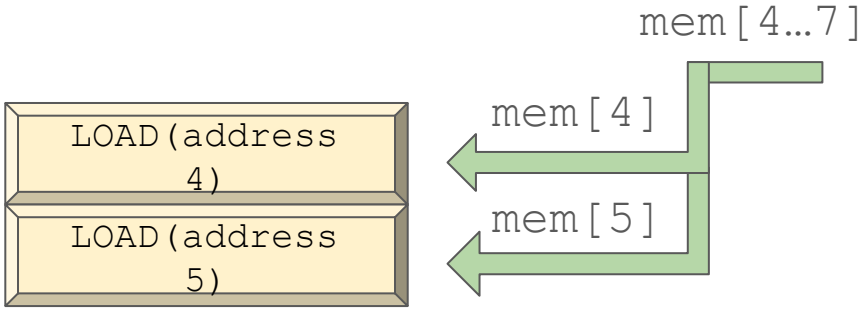
Old:

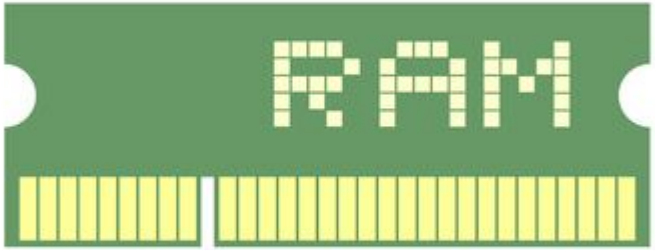



2 memory accesses

New:

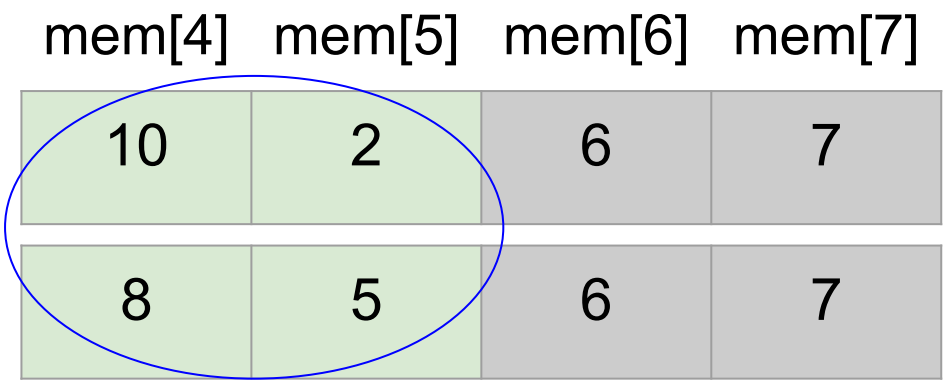



1 memory access

# Memory batching constraints

Instruction: store at indices 4 and 5

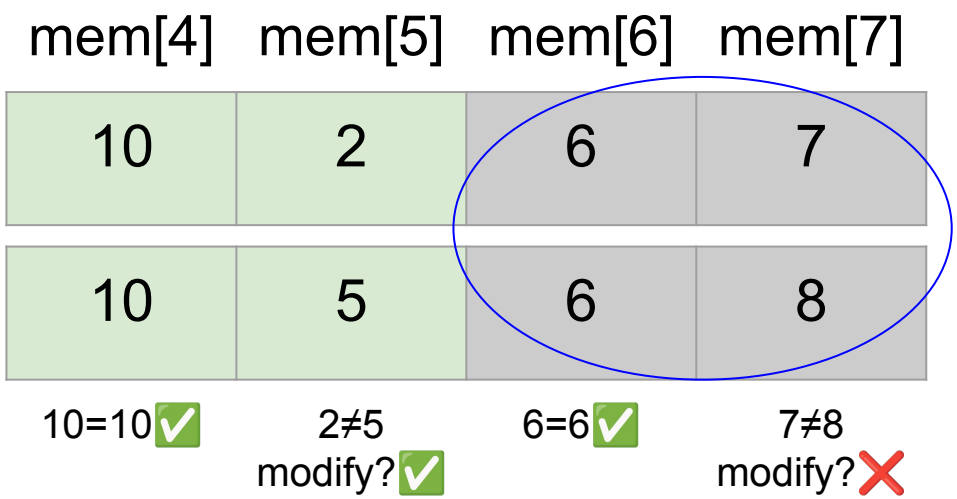



Correct store value = 2053

pack(8, 5) = 2053 ✅

# Memory batching constraints

Instruction: store at indices 4 and 5



# Memory batching constraints

Reported:

| mem[4] | mem[5] | mem[6] | mem[7] |
|---|---|---|---|
| 10 | 2 | 6 | 7 |
| 10 | 2 | 10 | 8 |

6≠10
modify?✅

Actual:

| mem[4] | mem[5] | mem[6] | mem[7] |
|---|---|---|---|
| 10 | 2 | 6 | 7 |
| 10 | 2 | 10 | 8 |

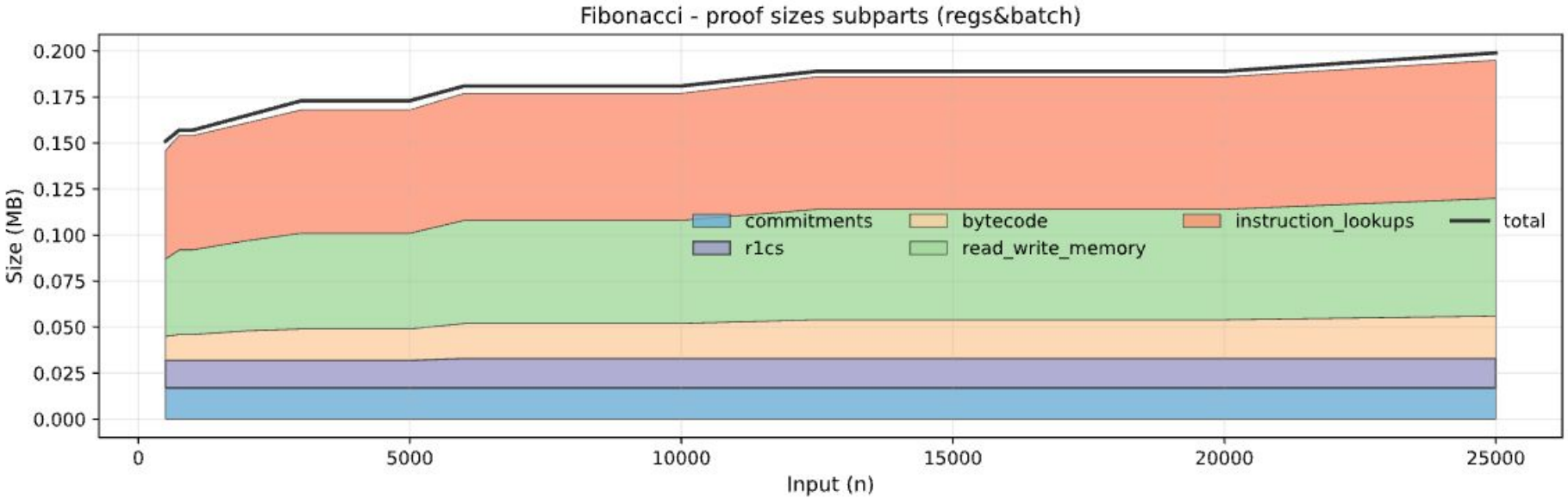
6≠10
modify?❌

Instruction: store at indices 4 and 5

# Evaluation

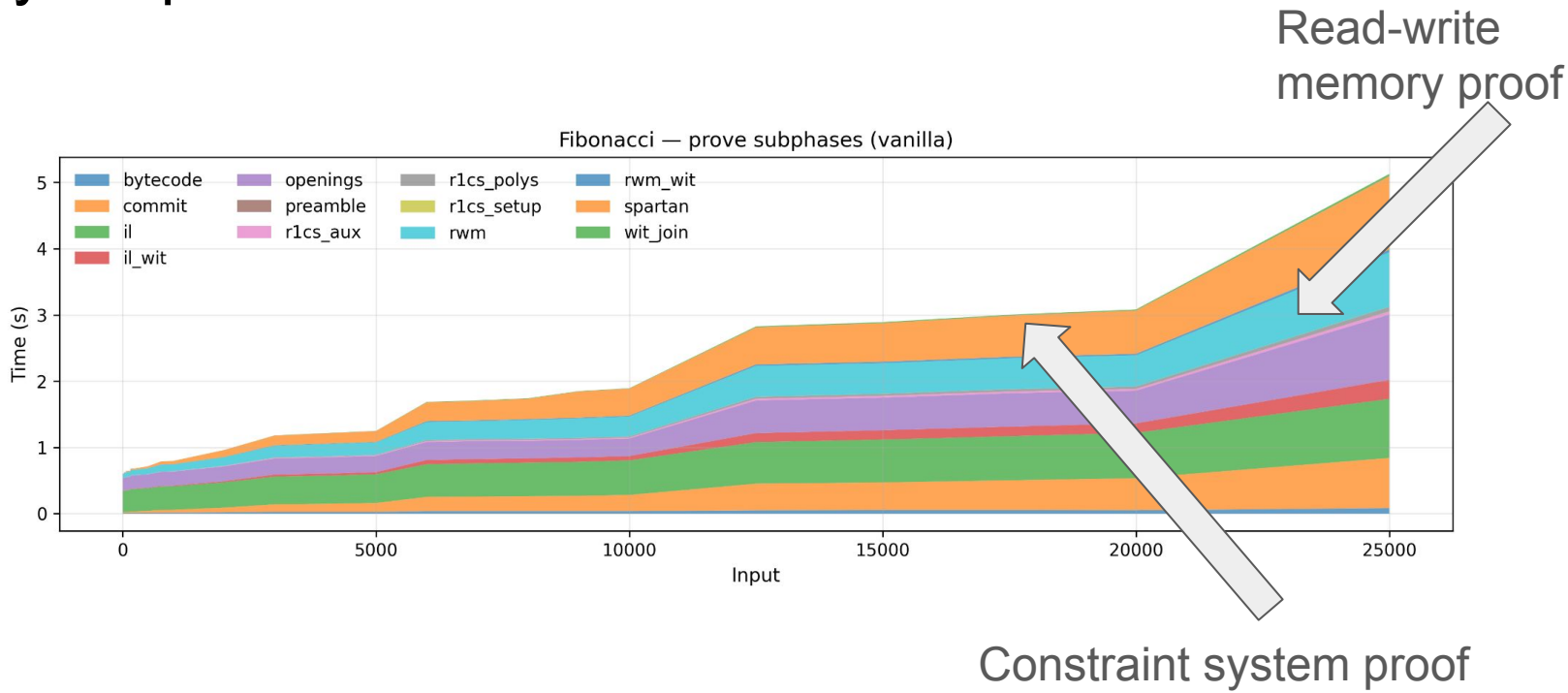
| | Constraints added | Memory proof speedup % | Memory proof size decrease % | Constraint proof slowdown % | Constraint proof size increase % |
|---|---|---|---|---|---|
| Moving registers | 207 | 20-30 | 10 | 350-400 | 90 |
| Moving registers + memory batching | 207 + 85 | 25-35 | 10 | 350-400 | 100-125 |

# Analysis: proof size




Proof size doesn’t change much: the memory proof size is much bigger, so a 10% decrease is as large as a 90% increase in constraint proof size

# Analysis: proof time




Tradeoff wasn’t worth it for Jolt because the constraint proof time increases more than the memory proof decreases

# Next steps

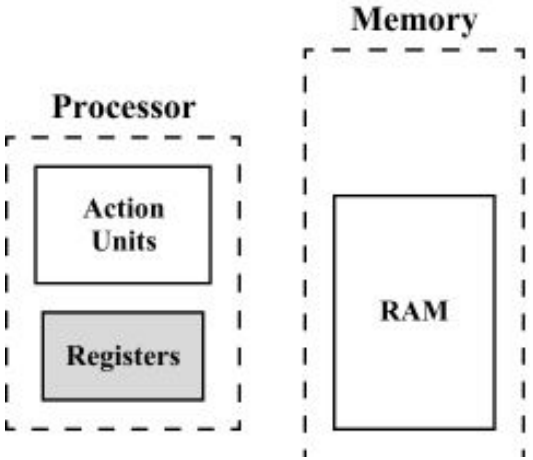
- More efficient register constraints: decreases added constraint overhead
  - Smaller number of registers in the constraint system (keep some in memory checking)
- Other zkVMs might have different tradeoffs

# Next steps

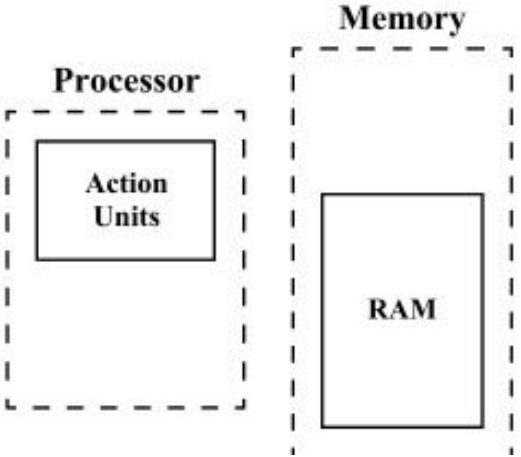
Remove load/store instructions by operating entirely over RAM instead of partially over registers

- This could be done in a preprocessing step: prover must show that they have correctly done the preprocessing

Option 1:




Option 2:



# Acknowledgments

Thank you Simon (very much) and PRIMES

And Prof. Srini Devadas and Dr. Slava Gerovitch

And parents

Questions?