# Using Ideas From Hardware To Accelerate Zero-Knowledge Virtual Machines

Celine Zhang[1], Eric Archerman[2] and Simon Langowski[3]

[1] Phillips Exeter Academy, Exeter, USA, chzhang@exeter.edu
[2] Roxbury Latin, West Roxbury, USA, eric.archerman@roxburylatin.org
[3] MIT, Cambridge, USA, slangows@mit.edu

**Abstract.** Zero-knowledge virtual machines (zkVMs) are an up-and-coming solution to the problem of verifiable computation: they allow a prover to generate a proof showing the correct execution of a computer program. A verifier can then quickly verify this proof without knowing certain potentially private details about the program. zkVMs stand out for how they combine the math of traditional verifiable computation schemes with the user-friendly functionality of standard compilers, such as Clang and Rust-C, and widely used programming languages like C++ and Rust. Traditional approaches often require translating programs into low-level domain-specific languages by hand, a process that is both labor-intensive and error-prone. zkVMs solve this issue by accepting programs in the native assembly of the chosen virtual processor (rather than an esoteric DSL). This convenience, however, comes at the cost of additional overhead, particularly in memory emulation, and makes zkVMs generally less performant than traditional techniques.

Using the prominent zkVM Jolt as an example, we seek to reconcile this gap in performance by optimizing Jolt's memory proofs, primarily by mirroring the memory access patterns of physical CPUs in the virtual setting. Just like physical CPUs have been improved by various hardware optimizations, our improvements – multiple in-flight instructions, batched memory reads, caching, and faster registers – seek to improve Jolt's performance. In this report, we outline our implementation plan for these optimizations and give some preliminary predictions of their results.

## 1 Introduction

zkVMs are a new technology in a long line of zkSNARKs, which solve the problem of "Verifiable Computation" [YY]. In the standard scenario, one entity, a "prover" ($\mathcal{P}$), claims that he has correctly executed a computation with a certain result, and another entity, a "verifier" ($\mathcal{V}$), determines whether the claim is true. In essence, the verifier needs to verify the computation done by the prover.

Perhaps most simply, $\mathcal{V}$ can just verify the computation by asking $\mathcal{P}$ for the program inputs and re-running the computation himself. However, this trivial method is not just a waste of computing power but also infeasible and privacy-leaking: in some cases, $\mathcal{V}$ is computationally bounded, and in others, $\mathcal{P}$ does not want to reveal certain details about the program.

Zero-Knowledge Succinct Non-interactive ARguments of Knowledge (zkSNARKs) [Pet19] solve this issue by both greatly reducing verifier overhead while also allowing the program's inputs to stay private. These zkSNARK schemes leverage various number theoretic properties to generate condensed proofs of the program's correct execution. Though these proofs require extra work from the prover, they are succinct, meaning small

in size (reducing communication costs) and fast to verify (reducing $\mathcal{V}$'s costs). Furthermore, if $\mathcal{P}$ desires, these proofs can also be made "zero-knowledge" [BSMP], meaning that verifying the proof will grant $\mathcal{V}$ no knowledge about the program beyond its correct execution. (See 1.1 for more details about zkSNARKs and zkSNARK construction). Since this powerful zero-knowledge property is optional, zkSNARKs are commonly referred to as (zk)SNARKs or just SNARKs.

This combination of succinctness and zero-knowledge lend zkSNARKs to many real-world applications. There's the traditional cloud computing example where $\mathcal{P}$ is running a computation on behalf of its client, $\mathcal{V}$, who does not want to blindly trust the cloud but also does not have the capabilities to re-run the program himself. The zero-knowledge property does not apply here as $\mathcal{V}$ provides $\mathcal{P}$ the inputs. Zero-knowledge becomes useful, however, when $\mathcal{P}$ wants to say something about his private information. This scenario commonly happens on blockchains: Zcash [Zca24] is a fork of the Bitcoin chain with augmented zkSNARK technology that allow for sender, receiver, and transaction size anonymity.

Most notably, these schemes are a powerful paradigm for scaling blockchains. On networks that allow for smart contracts and decentralized applications (e.g., Ethereum, Solana, Avalanche), there are often large computations or datasets that must be verified or stored on the network. These complex actions not only incur high gas fees for users initiating them but also require validation by all participating nodes, consuming the network's bandwidth. zkSNARKs allow this complex task to be taken off-chain and replaced with a succinct proof. This scaling solution saves computational resources for all participating nodes, and the "zero-knowledge" property enables additional use cases involving sensitive information, such as financial transactions or identity verification.

For instance, sidechains like Mina [Min24] and zk-rollups like zkSync [Mat24], StarkNet [Sta24], and Polygon zkEVM [Pol24] allow for proofs of solvency, private trading and auditing, identity verification, and data validation. Though zk-rollups generally focus on batch transactions and smart contracts (swaps, lending, staking), these frameworks are all flexible. Beyond these major blockchain applications, there has also been work in verifiable machine learning and voting systems [Ano24].

For all these great use cases of zkSNARK technology, traditional zkSNARK design levies a huge barrier to entry. As we explain in 1.1, zkSNARKs require programs to be hand-written in esoteric domain-specific languages (DSLs), making the process messy and error-prone. In 1.2, we discuss how zkVMs, a16z's Jolt in particular, solve this accessibility hurdle via generating proofs for CPU abstractions, though this accessibility comes with additional emulation overhead (mainly in regard to memory checking). 1.3 describes this overhead in greater detail. 2 provides technical background needed to understand our optimizations, and 3 covers our optimizations themselves.

## 1.1 Traditional zkSNARK design

zkSNARK proofs generally consist of a few polynomial equality checks that confirm the program's correct execution. This reduction from the computed statements we want to prove to the algebraic statements zkSNARKs actually prove is a process known as arithmetization. Since this reduction from code to algebra is often nontrivial, these programs are traditionally written in an arithmetic circuit format (circuits with either $+$ or $\times$ gates). Niche DSLs such as Circom [BMIMT$^+$] are used to make this translation process easier, but since these toolchains are not very well-developed, much of the compilation is still done by hand and the process remains quite low-level and error-prone.

In addition, these translations into algebra are extremely costly for certain "unfriendly" circuits. Translations can be messy when the programs involve non-algebra-native operations, most notably bitwise operations (AND, XOR, OR), bound checks, and comparisons, as they require breaking the variable into bits. Furthermore, programs must be unrolled

to synthesize the circuit, so complex loops and branches (conditions) often result in lots of work for the translator and larger-than-practical circuits.

Overall, though the math behind traditional zkSNARK schemes is quite clean, the process of converting programs into a usable format remains messy and labor-intensive. This barrier to entry has prevented traditional zkSNARK schemes from being employed in many real-world problems of verifiable computation.

## 1.2 zkVMs and Jolt

Zero-Knowledge Virtual Machines (zkVMs) clear this accessibility hurdle by verifying actions for an abstracted processor. Most CPU abstractions (aka Virtual Machines) emulate a normal processor's "fetch-decode-execute" cycle (see 2.2 for details). Zero-Knowledge Virtual Machines generate zkSNARKs that ensure the correctness of this emulation.

As opposed to traditional schemes where the computer program must first be hand-compiled into a low-level DSL, zkVMs allow developers to write programs in standard languages such as Rust and C++. Well-established compiler toolchains then compile the program into the zkVM's native instruction set architecture (ISA) and the verification scheme proves correct fetching, decoding, and executing. Under this new paradigm for zkSNARK design, developers no longer need to learn any additional knowledge pertaining to zkSNARKs, nor do they need to spend additional time with a messy arithmetization process.

Jolt [AST23] by a16z, is currently the most performant zkVM. It generates proofs for the correct execution of a binary file on an emulated RISC-V CPU [WLPA]. This new scheme allows standard compilers such as clang or rustc to take care of the previously messy and labor-intensive translation to a SNARK-friendly input format. Furthermore, by handling all instructions using lookup tables (see 2.7 for details), Jolt can handle bitwise operations much more efficiently. In addition, the VM model handles loops and branches in a streamlined manner, allowing for a wider range of applications than is possible with traditional zkSNARKs.

Though Jolt allows for an excellent developer experience, this CPU emulation adds significant overhead to the prover's computation time, mainly in generating proofs for memory accesses. Jolt abstracts memory as a single address space that data can be read from and written to one index at a time, where each index stores a single byte. This design, covered in more detail in 2.4, is relatively primitive compared to the memory hierarchy system employed in physical computer hardware (the computer uses multiple different memories, some faster than others). As a result, Jolt's memory checking protocols dominate the proof time (see Figure 1 for a breakdown).

## 1.3 Deficiencies in naive implementation

Although Jolt is already heavily optimized in various areas, it does not implement many of the optimizations generally applied to physical CPUs. For example, unlike most processors, which implement some form of instruction pipelining or process multiple in-flight instructions, Jolt currently only handles one instruction at a time. In addition, registers are stored in the same address space as RAM, so storing a value in a register doesn't bring any benefit over storing it in RAM. This also means that every instruction has to perform at least one, and up to seven, loads or stores from memory. On a separate note, Jolt uses a 254-bit field, but all instruction operands and outputs are only 64 bits. This results in some excess capacity that we can capitalize on to optimize the memory-checking protocol.

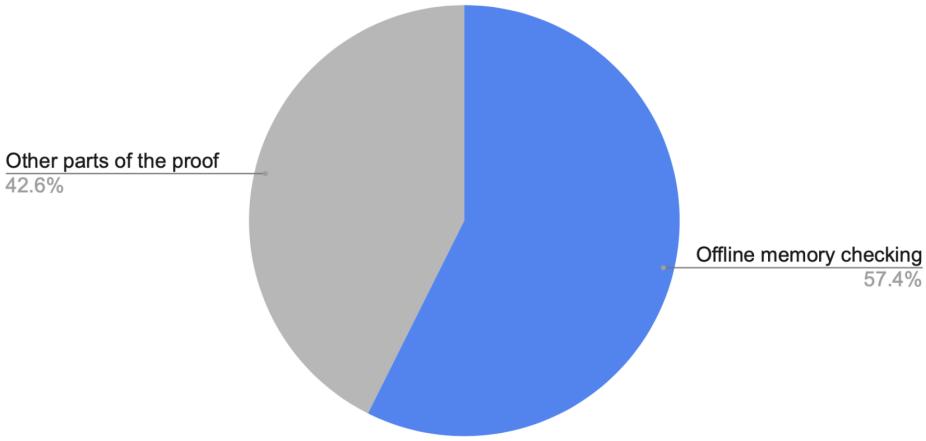


**Figure 1:** Offline memory checking dominates Jolt prover time.

# 2 Technical preliminaries

This section provides much of the context and background knowledge behind our optimizations in Section 3. The first three subsections (2.1, 2.2, and 2.3) focus on various computer hardware and computer architecture systems. 2.4 covers the architecture of Jolt in more detail, and the last three subsections (2.5, 2.6, 2.7) include background relating to the zk-space that is relevant to Jolt's architecture.

## 2.1 Caching

In most programs, the processor operates on certain memory values more than others. Intuitively, this non-uniformity occurs because the inputs of future operations tend to be the outputs of previous ones. On physical CPUs, caching [KLW94] uses this fact to accelerate memory accesses by introducing a smaller, faster auxiliary unit called a cache.

The cache temporarily stores frequently accessed data, and when the processor needs to access certain values, it will first search the cache. A "cache hit" is when the cache does contain the necessary information (resulting in a faster memory access). In the case of a "cache miss" (desired data not in cache), however, the processor moves on to search larger, slower memory units. On modern CPUs there are multiple cache layers, each around an order of magnitude faster than the next. There exist a variety of algorithms for actually managing the cache (i.e. loading and evicting values), the details of which are beyond the scope of this section.

## 2.2 Fetch-decode-execute cycle

The fetch-decode-execute cycle, or the instruction cycle [KMS11], is followed by the CPU to process instructions. When a program is run as a virtual machine (VM) abstraction, the VM interprets the program one instruction at a time, repeatedly running the fetch-decode-execute cycle of a CPU (once for each instruction.) To ensures correct execution of CPU actions, Jolt verifies each step of the fetch-decode-execute cycle.

During the fetch stage, the address of the next instruction to be executed is copied from the program counter (PC) to the memory address register (MAR). Eventually, the instruction itself (not its address) is copied into the current instruction register (CIR). Finally, the PC is incremented to the address of the instruction to be read in the next cycle.

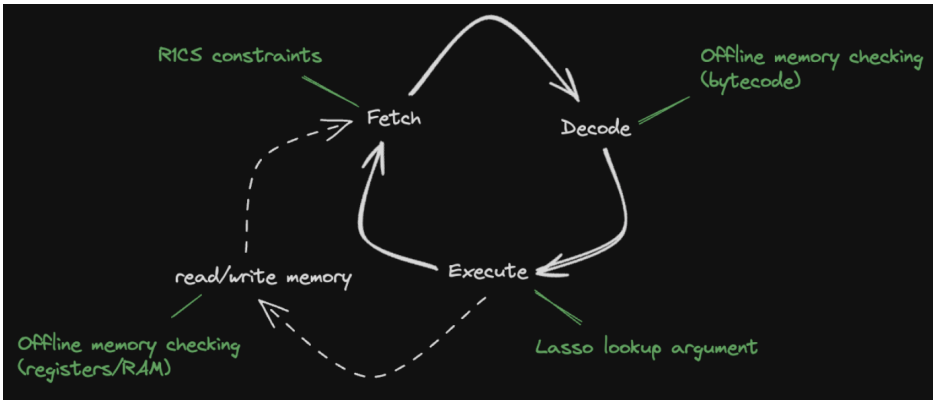


**Figure 2:** The four steps of Jolt

During the decode stage, the opcode of the instruction is decoded by binary decoders in the CPU's control unit. Then, the opcode is moved to the appropriate registers. Note that if the current instruction is a memory instruction, the execute step will occur during the next clock pulse.

During the execute stage, the CPU sends the instruction as a set of control signals to any components required to carry out the instruction, and thus performs the requested computation (e.g. produces the sum of two numbers).

## 2.3 RISC-V

In the RISC-V ISA, the machine consists of a CPU and a read-write memory. This memory is composed of three separate parts: 32 integer registers, program inputs/outputs (including the sequence of instructions), and the byte-addressable RAM. Note that in the zkVM model, the instructions are data and must be parsed from memory, following the von Neumann architecture [VN45]. In addition, since RISC-V supports byte-addressable memory, so must Jolt. However, byte-addressable memory is inefficient in the context of zkVMs, since offline memory checking natively operates on much larger (254-bit) field elements.

A RISC-V instruction can be expressed in the following form: (**opcode**, **rs1**, **rs2**, **rd**, **imm**), where **opcode** uniquely identifies the function of the specific instruction, **rs1** and **rs2** are the source registers (which commonly hold the operands of the instruction), **rd** is the destination register, and **imm** is a constant provided in the bytecode that sometimes acts as an operand. For example, the instruction (**AND**, 3, 8, 10, **imm**) performs the bitwise **AND** operation on the values stored in registers 3 and 8, and stores the result of this operation in register 10. Each RISC-V instruction is 4 bytes long, so for instructions that do not perform branches or jumps, incrementing the **PC** by 4 moves to the next instruction.

Note that RISC-V instructions can come in many forms – some take **rs1** and **imm** as the two operands, some take only **rs1** as an operand, etc. However, all RISC-V instructions share the properties of having at most two operands and writing one output.

## 2.4 Architecture of Jolt

Jolt proves zkVM execution using four cryptographic steps: the bytecode proof, the instruction lookup proof, the read-write memory proof, and the R1CS proof.

**Bytecode proof** ("bytecode" in Figure 2) The first step of Jolt is the bytecode proof, which is done in preprocessing. First, the ELF file [You95] for the program is converted to preprocessed bytecode, which is available in full to both $\mathcal{P}$ and $\mathcal{V}$. Then this preprocessed bytecode is converted to a bytecode trace, which contains decoded information about the sequence of RISC-V instructions to be executed. Taking the preprocessed bytecode as input, the bytecode proof uses read-only offline memory checking (see 2.6) to ensure that the bytecode trace was generated correctly. In other words, the bytecode proof ensures that the sequence of instructions in the program was correctly decoded into a more usable format for the rest of Jolt. This corresponds to the "decode" step of the fetch-decode-execute cycle.

**Instruction lookup proof** ("Lasso lookup argument" in Figure 2) The instruction lookup proof ensures that all instructions in the bytecode were executed correctly. Jolt executes instructions using several large lookup tables (see 2.7), one for each RISC-V instruction. The lookup table for a certain instruction stores the output of that instruction for each possible pair of 64-bit inputs, resulting in a table of size greater than $2^{128}$. The instruction lookup proof makes sure that all lookups into this table were performed correctly, therefore ensuring that the purported "outputs" of each instruction are correct. In addition, the instruction lookup proof employs a read-only offline memory checking argument (see 2.6) to ensure that all reads to the large lookup table were done correctly. The instruction lookup proof corresponds to the "execute" step of the fetch-decode-execute cycle.

**Read-write memory proof** ("registers/RAM" in Figure 2) Like on physical computers, instructions often read or write to a memory address space. The other proofs operate assuming that the values read and written to this memory are correct, so an additional proof is needed for this. In Jolt, this address space contains the 32 RISC-V registers, program inputs and outputs, and RAM, and the read-write memory proof ensures that reads and writes to any of this data is done correctly. This proof employs a read-write variant of offline memory checking (see 2.6), which scales with the total number of memory accesses.

**R1CS proof** ("R1CS constraints" in Figure 2) The R1CS proof uses a small constraint system (see 2.5) to enforce certain rules about instruction execution; for example, it ensures that the two operands of the instruction were fetched correctly based on the instruction type. In addition, the R1CS proof ensures consistency between the other proofs; for example, it checks that corresponding values passed to all proofs are actually equal. There are two types of constraints used in the R1CS proof: uniform and non-uniform constraints. Uniform constraints are checked once per VM step, and take inputs from only one step. Non-uniform constraints are checked between VM steps, and can take inputs from different steps.

As stated previously, the most time-consuming parts of Jolt are offline memory checking protocols, specifically the protocols accompanying the instruction lookup proof and the read-write memory proof.

## 2.5 Constraint systems

As implied by the name, constraint systems "constrain" relationships between variables. In zkSNARKs, Rank-1 Constraint Systems (R1CS) [BSCG+13] help determine linear (rank-1) arithmetic relationships between values of a computation. Mathematically, for a vector of computation variables $\mathbf{z}$, an r1cs enforces the relationships between $\mathbf{z}$'s elements with the equation:

$$\mathbf{A} \cdot \mathbf{z} \odot \mathbf{B} \cdot \mathbf{z} = \mathbf{C} \cdot \mathbf{z}.$$

Here, $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{C}$ are matrices, the $\cdot$ performs matrix-vector multiplications, and $\odot$ denotes element-wise (Hadamard) multiplication.

Jolt's R1CS consists of many small sections (one for each instruction cycle) that ensure correct instruction fetching (see 2.4 for more details). Below is an example R1CS construction constraining the variables of an if-statement.

**Example** Consider encoding a conditional statement where we multiply $u$ and $v$ if the boolean $b$ is 0, and we add $u$ and $v$ if $b$ is 1:

$$b \cdot (u + v) + (1 - b) \cdot (uv) = w.$$

This statement can be broken down into the following equations:

1. $b$ binary: $b \cdot (1 - b) = 0$,
2. $t_1$ first intermediate value: $b \cdot (u + v) = t_1$,
3. $t_2$ second intermediate value: $u \cdot v = t_2$,
4. $t_3$ third intermediate value: $(1 - b) \cdot t_2 = t_3$,
5. $w$ correct output: $t_1 + t_3 = w$.

Since $\mathbf{z}$ is the vector of variables (with 1 as a constant value preserved for additions), it will be of the form:

$$\mathbf{z} = \begin{bmatrix} 1 \\ b \\ u \\ v \\ t_1 \\ t_2 \\ t_3 \\ w \end{bmatrix}.$$

Then the following matrices $\mathbf{A}$, $\mathbf{B}$, $\mathbf{C}$ can be constructed. Each row encodes one of the five non-constant products, and the columns represent variables:

$$\left.\begin{array}{ll} (1) & b \cdot (1 - b) = 0 \\ (2) & b \cdot (u + v) = t_1 \\ (3) & u \cdot v = t_2 \\ (4) & (1 - b) \cdot t_2 = t_3 \\ (5) & t_1 + t_3 = w \end{array}\right\} \quad \longrightarrow \quad \begin{array}{l} \mathbf{A} = \begin{array}{c} \begin{matrix} 1 & b & u & v & t_1 & t_2 & t_3 & w \end{matrix} \\ \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \end{bmatrix} \end{array}, \\ \mathbf{B} = \begin{array}{c} \begin{matrix} 1 & b & u & v & t_1 & t_2 & t_3 & w \end{matrix} \\ \begin{bmatrix} 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \end{array}, \\ \mathbf{C} = \begin{array}{c} \begin{matrix} 1 & b & u & v & t_1 & t_2 & t_3 & w \end{matrix} \\ \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \end{array}. \end{array}$$

Altogether, the R1CS can be written as follows:

$$\underbrace{\begin{array}{c}\begin{matrix} b & u & v & t_1 & t_2 & t_3 & w \end{matrix}\\ \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & -1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 \end{bmatrix}\end{array}}_{\mathbf{A}} \cdot \underbrace{\begin{bmatrix} 1 \\ b \\ u \\ v \\ t_1 \\ t_2 \\ t_3 \\ w \end{bmatrix}}_{\mathbf{z}} \odot \underbrace{\begin{array}{c}\begin{matrix} 1 & b & u & v & t_1 & t_2 & t_3 & w \end{matrix}\\ \begin{bmatrix} 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}\end{array}}_{\mathbf{B}} \cdot \underbrace{\begin{bmatrix} 1 \\ b \\ u \\ v \\ t_1 \\ t_2 \\ t_3 \\ w \end{bmatrix}}_{\mathbf{z}}$$

$$= \underbrace{\begin{array}{c}\begin{matrix} 1 & b & u & v & t_1 & t_2 & t_3 & w \end{matrix}\\ \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}\end{array}}_{\mathbf{C}} \cdot \underbrace{\begin{bmatrix} 1 \\ b \\ u \\ v \\ t_1 \\ t_2 \\ t_3 \\ w \end{bmatrix}}_{\mathbf{z}}.$$

Now, for whatever values the variables in $\mathbf{z}$ take, the constrain system's equality condition holds only if $b$, $t_1$, $t_2$, $t_3$, and $w$ satisfy the five constraint relations above. Jolt's R1CS uses around 60 constraints per instruction cycle, and employs the Spartan zkSNARK scheme [Set19] to prove that $\mathbf{z}$ (the witness vector) does indeed satisfy its R1CS.

## 2.6 Offline memory checking

Offline memory checking [BEGN94] is a protocol that allows a prover to prove to a verifier that a sequence of memory accesses (reads and/or writes) were handled correctly. In offline memory checking, unlike other memory checking protocols, all memory accesses are verified at once, after all memory operations have been performed.

Offline memory checking is usually carried out using four multisets, which contain the initial and final memory states, and all reads and writes done to memory. Each read is accompanied by a write which updates a timestamp associated with the memory address.

Because every memory value read must have been written at some point, we can observe two properties about memory: 1) every value in memory is either from initialization or written into memory. 2) every value in memory is either read or part of the final memory state. These two properties form the basis for the memory checking, as we can check that the unions of these two pairs of multisets (initial $\cup$ writes and reads $\cup$ final) are equivalent.

This check is accomplished by Reed-Solomon fingerprinting each multiset [VSG02]. The fingerprints are computed using an optimized version of a large grand product argument [Tha17].

In Jolt, read-only offline memory checking is used in the bytecode proof and the memory proof accompanying the instruction lookup proof. Additionally, a read-write variant of offline memory checking called Spice [SAGL18] is used in Jolt's read-write memory proof.

## 2.7 Lookup tables and Lasso

A lookup table is a data structure that precomputes and stores the results of certain operations, so they can be quickly retrieved when needed instead of performing the computation each time. In the zk-space, "lookup arguments" generate proofs showing that a correct array indexing operation was performed on a lookup table. In Jolt's case, the Lasso lookup argument [STW24] proves lookups the lookup tables for RISC-V instructions.

This lookup-based design for proving instruction execution offloads verifying complex structures of + and × gates to extensive memory checking procedures (note the parallel to how traditional lookup tables trade computation overhead for memory overhead). For this reason, lookup arguments (Lasso in particular) are much faster than traditional zkSNARK schemes for bit-related operations (and RISC-V instruction executions in general). Lookup tables allow more expressivity in circuit descriptions, resulting in more efficient proofs.

For each RISC-V instruction (**ADD**, **XOR**, etc.), Lasso generates a lookup proof for the following statement:

$$\#\text{ rows} = \#\text{ lookups} \left\{ \overbrace{\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix}}^{\#\text{ columns} = \text{size of } \mathbf{T}} \cdot \begin{bmatrix} 5 \\ 7 \\ 6 \\ 9 \end{bmatrix} \right\} N\ (2^{128}\ \text{ in Jolt}) = \left\{ \begin{bmatrix} 5 \\ 7 \\ 6 \\ 9 \\ 9 \end{bmatrix} \right\} \#\text{ rows} = \#\text{ lookups}$$

$$\mathbf{M} \quad \cdot \quad \mathbf{T} \quad = \quad \mathbf{a}$$

$$\text{lookup indices} \quad \cdot \quad \text{lookup table} \quad = \quad \text{lookup results}$$

**Figure 3:** Looking up indices $\mathbf{M}$ of table $\mathbf{T}$ gives $\mathbf{a}$.

Since $\mathbf{T}$ is so large ($2^{128}$) Lasso does not actually fully materialize the lookup table. Instead, it uses the Sum-Check Protocol [LFKN92] to reduce the statement into many smaller memory checks. This explainer video ([Tha]) breaks the scheme down quite well.

Lookup tables can be combined to improve efficiency. For this reason, Jolt combines all of the lookups for the RISC-V instructions into one large lookup table, which gives it its name, Just One Lookup Table.

# 3 Our Work

In this section, we outline the various directions we are taking to optimize Jolt's memory checking, and provide support for the idea that these changes will increase Jolt's performance.

## 3.1 Multiple in-flight instructions

Implementing multiple in-flight instructions means that the Jolt prover will handle multiple instructions at once. Specifically, this means that the Jolt prover will prove multiple instructions in the same proof cycle. This should bring an increase in efficiency, because it takes advantages of any shared memory accesses that these instructions may have. This is especially helpful due to the large volume of memory accesses performed with each instruction. In fact, there is no significant difference in time between proving a load/store instruction and a non-load/store instruction (see Figure 4), and this is likely because registers and RAM are stored in the same address space. Because of this, each instruction performs anywhere from one to seven memory accesses, so it is likely that for consecutive instructions, there will be overlap in these accesses.

However, there are some challenges associated with implementing this optimization, and many of them are similar to problems that occur in instruction pipelining in physical CPUs.


**Figure 4:** For a fixed trace length, having more load/store instructions does not introduce any additional overhead.

**Pipeline stall** In some cases, consecutive instructions are "dependent" on each other and cannot be executed together. For example, if one instruction adds $x + y$ and stores the result in $z$, and the next instruction adds $x + z$ and stores the result in $w$, these two instructions cannot be handled at the same time. Some other examples of this are branches and jumps – we do not know ahead of time what the "next" instruction will be, so we cannot make a block of instructions in this case. In both of these cases, we must split the block of instructions so that no instructions in the same block are dependent. We must also create a set of constraints to determine whether a block of instructions is dependent and handle the set accordingly.

**Nop insertion** Additionally, oftentimes we cannot make a valid block of $k$ instructions to handle together. This may happen as a result of dependent instructions, since that involves splitting a block of $k$ instructions into multiple smaller blocks. In physical CPUs, this is handled by inserting "nop" (no operation) instructions into both instruction blocks to make up the difference. Nop insertion is also sometimes necessary at the end of the trace, when there may not be enough instructions left to create a full block.

**Inter-proof dependency** Since many of Jolt's proofs are structured by the instruction, combining multiple instructions into one proof cycle requires changing all parts of the proof system (except the bytecode proof) in tandem. In particular, the read-write memory proof and the R1CS proof both need to be modified in an internally consistent manner.

## 3.2 Larger memory reads

An observation critical to this optimization is that Jolt uses a 254-bit prime field, while the operands of the instructions (and therefore the inputs to the memory proof) are only 64 bits. This indicates that multiple memory tuples can be packed together and then fed into the memory proof. However, it is not immediately obvious whether or not this packing will result in a speedup. Although packing memory tuples together would result in fewer total memory tuples and a smaller input to the memory proof, therefore speeding up the memory proof, there may be overhead introduced with each nonzero bit per field element.

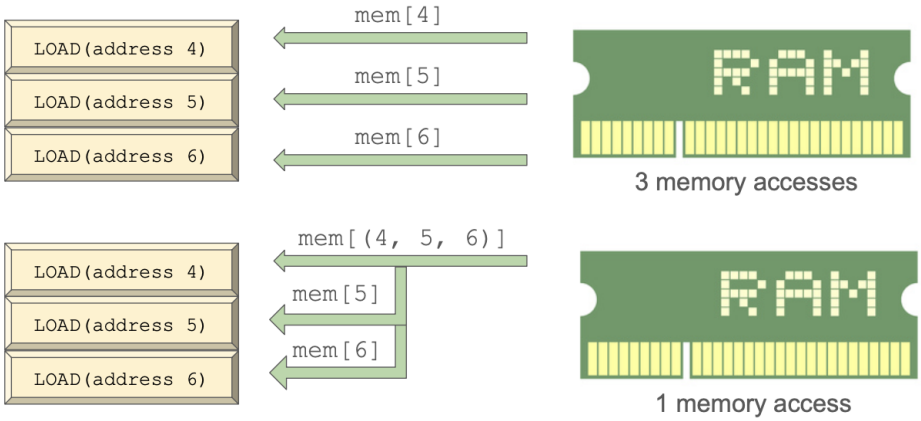


**Figure 5:** An example of the benefits of larger memory reads

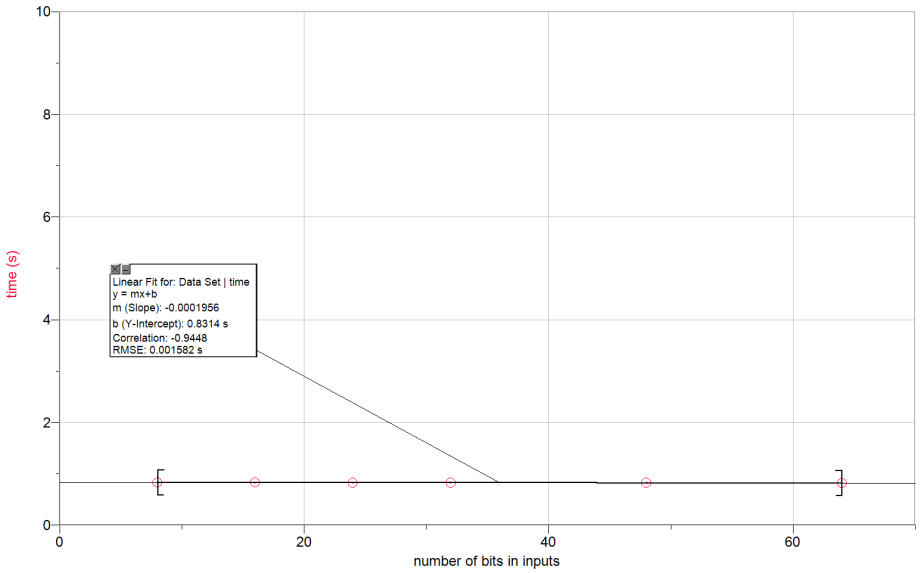


**Figure 6:** The effect of increasing input size on memory checking proof is negligible.

If such overhead does exist, it may overshadow the benefits of packing. To test whether this is the case, we created a unit test to test only the memory checking proof, and fed increasingly large field elements as inputs into the proof. Since this unit test only ran offline memory checking on the inputs, we were not concerned with implementing a valid program. Fortunately, as shown in Figure 6, we found that there was no relationship between the speed of the memory proof and the size of each input (up to a field element). Therefore, the effect of decreasing the number of inputs dominates over the effect of increasing the size of the field elements passed as input. Because of this, packing should result in a faster memory proof, since performing the packing itself should take negligible time.

In addition, since registers and RAM are in the same memory space, simply fetching $k$ values from memory at once and placing them in $k$ separate registers will not bring a speedup, because all values would still need to be fetched from memory. Instead, the benefit would come from packing multiple values into the same register with each memory access. If one instruction (or multiple in-flight instructions) happened to use consecutive memory values stored in the same register, as is often the case, packing memory values into the same register would reduce the number of register accesses for that instruction or

group of instructions.

## 3.3 Caching

In a physical computer, cache memory is faster because of both a difference in physical distance to the CPU and a difference in material compared to standard memory. However, in a virtual machine there is no literal concept of "distance" or "material" to be taken advantage of, so instead we can perform the read-write memory proof for the cache in a different fashion. Instead of storing values in the address space used for registers and RAM, we can store a small number of cached values in the constraint system as (address, value) pairs.

In our model, the cache algorithm will be transparent to the processor. Instead, for each cache load or store, the prover will provide as advice which cache index to access. However, the prover cannot be automatically trusted to provide accurate cache indices, so constraints will be required to ensure accuracy and consistency of cache handling. Specifically, constraints would be needed to ensure that stale values are not read from the cache and that cache values are not ignored.

Similarly to in physical CPUs, using a cache reduces the number of accesses to RAM. In the context of Jolt, each memory access is either done to registers/RAM or to the cache. Since the cache is stored in the constraint system, each access to the cache would reduce the size of the read-write memory proof. Since the memory proof brings a much larger overhead than the R1CS proof does, this would therefore bring an overall speedup.

## 3.4 Faster registers

As stated previously, in Jolt the registers and RAM occupy the same memory space. Therefore, storing a value in a register does not bring any actual benefit in terms of the total number of memory accesses verified by the read-write memory proof. Figure 4 supports this assessment, as it shows that non-load/store instructions (that read from registers instead of RAM) do not eliminate memory checking overhead as compared to load/store instructions (that read from RAM).

One way to improve on this model of one unified memory space is to "move" the registers from the RAM space to the constraint system. Similarly to in caching, each memory access is either handled through RAM or registers, so each access to registers stored in the constraint system reduces the total number of memory accesses handled through the memory proof. This register model may be simpler in some ways than caching, because caching requires compiler assistance and advice from the prover, while registers do not. Unlike caching, the "logic" for registers is already implemented: it just needs to be moved from RAM to the constraint system. In addition, the register model does not require proving access consistency, because there isn't the same notion of a "stale" value in registers that there is in a cache – this issue is handled automatically during program compilation.

Because Jolt stores registers in memory, it does not implement any specific proof machinery for registers. Specifically, Jolt has no constraints to handle values read from registers and no ready-made way of ensuring consistency of the registers between proof steps. Ensuring consistency of the registers involves adding additional inputs to the constraint system that are maintained between different iterations of the proof system. The only value currently maintained in this manner is the program counter register, and we can partially model our handling of the registers off of Jolt's handling of the program counter.

Because of this, like the program counter, registers need a combination of non-uniform and uniform constraints to ensure security. Specifically, about 32 extra non-uniform constraints and some small number (at most 20) of uniform constraints are needed per

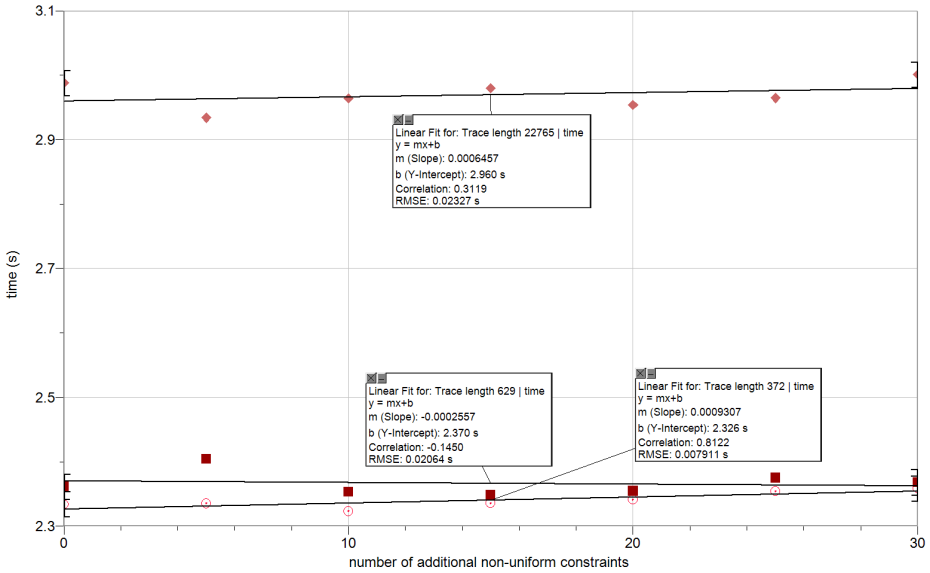


**Figure 7:** Adding a small number of non-uniform constraints has negligible effect on proof time.

step to implement 32 registers. To get a preliminary measure of the overhead of adding these constraints, we tested the time it would take to add a varying number of additional uniform and non-uniform constraints. We found that adding 32 non-uniform constraints has a negligible effect on the total proof time (see Figure 7), and we found that the time to prove uniform constraints scales quadratically (see Figure 8). This makes sense because adding one constraint per step increases the size of the constraint matrix for each step, resulting in superlinear growth. This quadratic growth means that adding a small number of uniform constraints per step has a small effect on total proof time. Specifically, for two sample programs of trace length less than 700, adding 20 extra uniform constraints per step results in an increased proof time of less than 0.000005%.

Since uniform constraints are checked once per trace step, it is natural that the slowdown of adding some number of extra uniform constraints per step is more significant in larger programs (with a longer trace). For example, for a sample program of trace length 23000, adding 20 extra uniform constraints per step results in an increased proof time of about 0.007%. However, this should not affect the effectiveness of our register optimizations (and memory optimizations in general), because the benefits from these optimizations also scale with trace length. For example, a longer program uses registers more than a shorter program does, and therefore would benefit more from this optimization.

# 4 Conclusion and future work

We have analyzed the performance bottlenecks of Jolt and determined that it is most efficient to focus optimizations on the memory checking protocol. In hardware, memory is optimized through strategies such as instruction pipelining, memory batching, and some form of memory hierarchy, so we hypothesize that mirroring these optimizations in the zkVM setting would bring similar benefits.

So far, in terms of multiple in-flight instructions, we've gained familiarity with manipulating constraints, and we have implemented logic to verify two copies of a single instruction in one R1CS proof cycle. Although for now this simply introduces (minimal) overhead, it is an intermediate step towards verifying two consecutive instructions in the

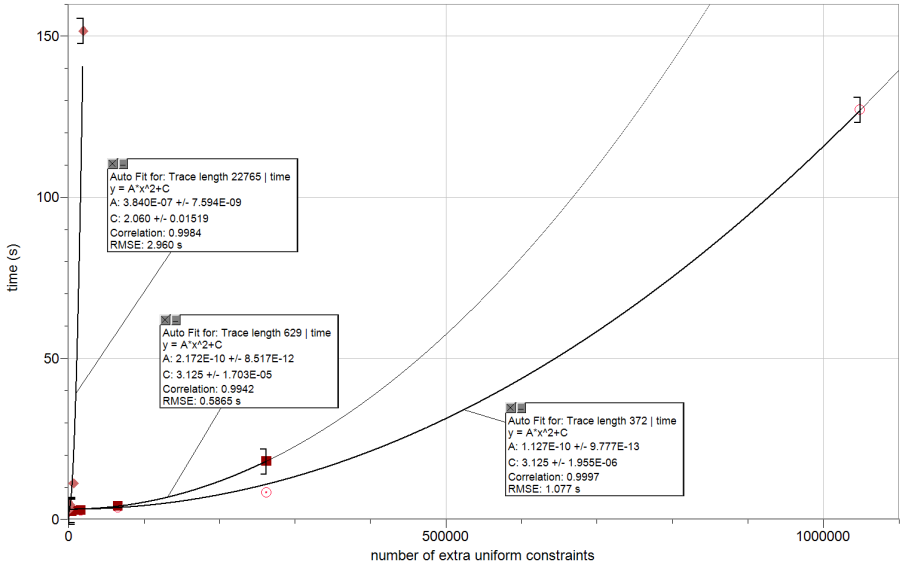


**Figure 8:** The time to prove uniform constraints scales quadratically.

same cycle. In addition, we have begun to manipulate the instruction lookup proof's witness generation function to handle larger chunks of instructions.

In terms of faster registers, we've generated variables from the bytecode that hold the register state at each step, and we've passed these variables as inputs to the R1CS proof. Since in most programs, 50-97% of memory accesses involve registers, and the overhead of adding a small number of constraints is minimal, we expect that this optimization speeds up the read-write instance of offline memory checking by about 50-97%.

By improving the performance of zkVMs like Jolt to make these technologies competitive with traditional zkSNARKs, we improve accessibility and usability for those wishing to use zkSNARKs and enable new performant applications for use with circuits with lots of bitwise operations, loops, and branches.

After implementing our optimizations, we expect that the read-write memory timestamp validity proof (an additional step necessary for read-write offline memory checking) will be the performance bottleneck for Jolt. To address this new bottleneck, we will continue to investigate more complicated hardware optimizations like caching and out-of-order execution, which involves rearranging the instruction execution ordering to reduce the overhead of the timestamp validity proof.

## References

[Ano24] Anonymous. Applications of zksnarks: Blockchain, machine learning, and beyond. https://eprint.iacr.org/2024/1003.pdf, 2024. Cryptology ePrint Archive, Report 2024/1003. Accessed: 2024-12-30.

[AST23] Arasu Arun, Srinath Setty, and Justin Thaler. Jolt: SNARKs for virtual machines via lookups. Cryptology ePrint Archive, Paper 2023/1217, 2023.

[BEGN94] Manuel Blum, W. Evans, Peter Gemmell, and M. Naor. Checking the correctness of memories. *Algorithmica*, 12:225–244, 09 1994.

[BMIMT+] Marta Bellés-Muñoz, Miguel Isabel, Jose Luis Muñoz-Tapia, Albert Rubio, and Jordi Baylina. Circom: A circuit description language for building zero-knowledge applications.

[BSCG+13] Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin, Eran Tromer, and Madars Virza. SNARKs for c: Verifying program executions succinctly and in zero knowledge. Cryptology ePrint Archive, Paper 2013/507, 2013.

[BSMP] Manuel Blum, Alfredo De Santis, Silvio Micali, and Giuseppe Persiano. Non-interactive zero-knowledge.

[KLW94] R. Karedla, J.S. Love, and B.G. Wherry. Caching strategies to improve disk system performance. *Computer*, 27(3):38–46, 1994.

[KMS11] Andrzej Kwiecień, Michał Maćkowski, and Krzysztof Skoroniak. The analysis of microprocessor instruction cycle. In Andrzej Kwiecień, Piotr Gaj, and Piotr Stera, editors, *Computer Networks*, pages 417–426, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[LFKN92] Carsten Lund, Lance Fortnow, Howard J. Karloff, and Noam Nisan. Algebraic methods for interactive proof systems. *Journal of the ACM (JACM)*, 39(4):859–868, 1992.

[Mat24] Matter Labs. zksync: Scaling ethereum with zero-knowledge proofs. https://zksync.io, 2024. Accessed: 2024-12-30.

[Min24] Mina Developers. Mina protocol. https://minaprotocol.com, 2024. Accessed: 2024-12-30.

[Pet19] Maksym Petkus. Why and how zk-snark works. *CoRR*, abs/1906.07221, 2019.

[Pol24] Polygon Labs. Polygon zkevm: Ethereum scaling via zksnarks. https://polygon.technology/solutions/polygon-zkevm, 2024. Accessed: 2024-12-30.

[SAGL18] Srinath Setty, Sebastian Angel, Trinabh Gupta, and Jonathan Lee. Proving the correct execution of concurrent services in zero-knowledge. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 339–356, Carlsbad, CA, October 2018. USENIX Association.

[Set19] Srinath Setty. *Spartan: Efficient and general-purpose zkSNARKs without trusted setup*. Cryptology ePrint Archive, Paper 2019/550, 2019.

[Sta24] StarkWare Industries. Starknet: A decentralized validity-rollup. https://starknet.io, 2024. Accessed: 2024-12-30.

[STW24] Srinath Setty, Justin Thaler, and Riad Wahby. Unlocking the lookup singularity with lasso. In Marc Joye and Gregor Leander, editors, *Advances in Cryptology – EUROCRYPT 2024*, pages 180–209, Cham, 2024. Springer Nature Switzerland.

[Tha] Justin Thaler. Jolt, zkvms, and speeding up blockchains.

[Tha17] Justin Thaler. Time-optimal interactive proofs for circuit evaluation, 2017.

[VN45] John Von Neumann. Von neumann architecture. *Online http://en. wikipedia. org/wiki/Von_Neumann_architecture*, 8, 1945.

[VSG02] Ravi Sankar Veerubhotla, Ashutosh Saxena, and Ved Prakash Gulati. Reed solomon codes for digital fingerprinting. In Alfred Menezes and Palash Sarkar, editors, *Progress in Cryptology — INDOCRYPT 2002*, pages 163–175, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.

[WLPA] Andrew Waterman, Yunsup Lee, David Patterson, and Krste Asanović. The risc-v instruction set manual.

[You95] Eric Youngdale. The elf object file format by dissection. *Linux Journal*, 13, 1995.

[YY] Xixun Yu and Zheng Yan. A survey of verifiable computation.

[Zca24] Zcash Developers. Zcash: Privacy-protecting digital currency. https://github.com/zcash/zcash, 2024. Accessed: 2024-12-30.

# A Register frequencies in several sample programs

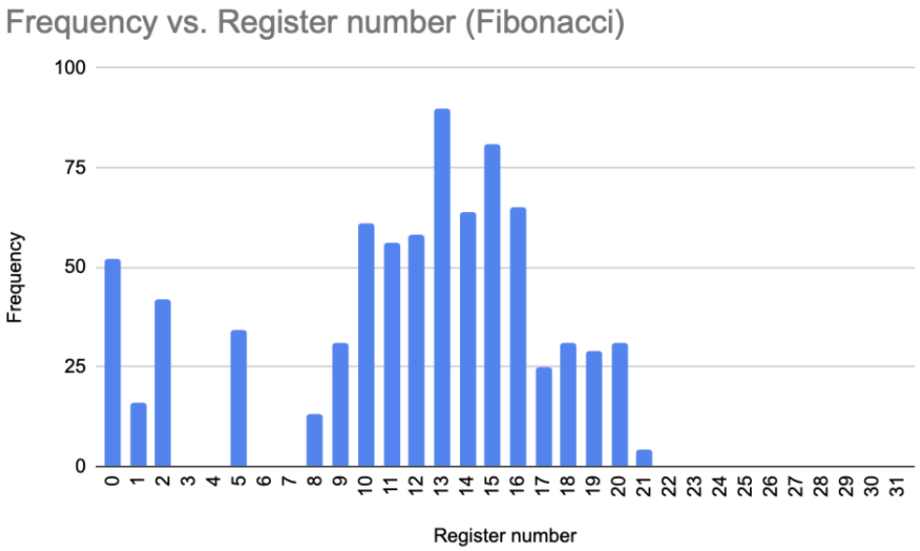


**Figure 9:** How much each register is used in the Fibonacci sample program

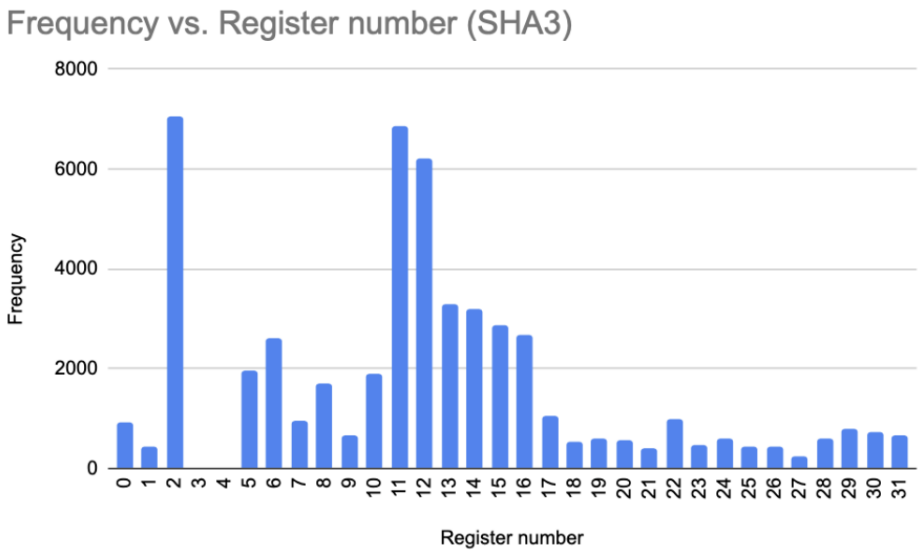


**Figure 10:** How much each register is used in the SHA3 sample program

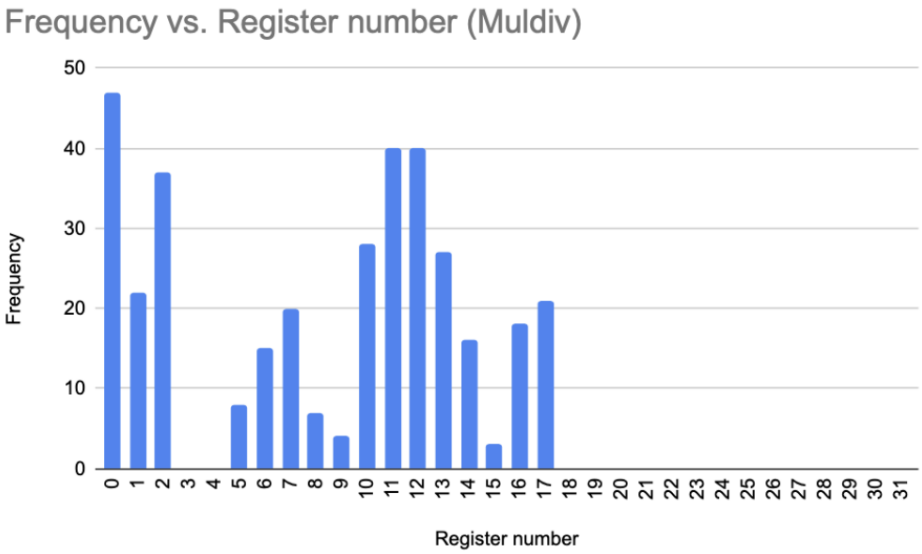


**Figure 11:** How much each register is used in the sample program consisting of a single multiplication followed by a single division

Of the three sample programs, SHA3 is the longest (with trace length about 23000). SHA3 is also the only sample program of the three to use all registers except registers 3 and 4. This indicates that for larger programs, all registers may be used, so since the overhead of adding registers is small, we plan to move to the constraint system all registers except registers 3 and 4, which are not used in any of the three sample programs.