Jolt: SNARKs for Virtual Machines via Lookups

Arasu Arun[*] Srinath Setty[†] Justin Thaler[‡]

Abstract

Succinct Non-interactive Arguments of Knowledge (SNARKs) allow an untrusted prover to establish that it correctly ran some “witness-checking procedure” on a witness. A zkVM (short for zero-knowledge Virtual Machine) is a SNARK that allows the witness-checking procedure to be specified as a computer program written in the assembly language of a specific instruction set architecture (ISA).

A *front-end* converts computer programs into a lower-level representation such as an arithmetic circuit or generalization thereof. A SNARK for circuit-satisfiability can then be applied to the resulting circuit.

We describe a new front-end technique called Jolt that applies to a variety of ISAs. Jolt arguably realizes a vision called the *lookup singularity*, which seeks to produce circuits that only perform lookups into pre-determined lookup tables. The circuits output by Jolt primarily perform lookups into a gigantic lookup table, of size more than 2^{128}, that depends only on the ISA. The validity of the lookups are proved via a new *lookup argument* called Lasso described in a companion work (Setty, Thaler, and Wahby, e-print 2023). Although size-2^{128} tables are vastly too large to materialize in full, the tables arising in Jolt are structured, avoiding costs that grow linearly with the table size.

We describe performance and auditability benefits of Jolt compared to prior zkVMs, focusing on the popular RISC-V ISA as a concrete example. The dominant cost for the Jolt prover applied to this ISA (on 64-bit data types) is cryptographically committing to about five 256-bit field elements per step of the RISC-V CPU. This compares favorably to prior zkVM provers, even those focused on far simpler VMs.

1 Introduction

A SNARK (Succinct Non-interactive ARgument of Knowledge) is a cryptographic protocol that lets anyone prove to an untrusting verifier that they know a witness w satisfying some property. A trivial proof is for the prover $\mathcal{P}$ to explicitly send the witness to $\mathcal{V}$, who can then directly check on its own that w satisfies the claimed property. We refer to this trivial verification procedure as *direct witness checking.*

A SNARK achieves the same effect, but with better costs to the verifier. Specifically, the term *succinct* roughly means that the proof should be shorter than this trivial proof (i.e., the witness w itself), and verifying the proof should be much faster than direct witness checking.

As an example, the prover could be a cloud service provider running an expensive computation on behalf of its client, the verifier. A SNARK proof gives the verifier confidence that the prover ran the computation honestly. Alternatively, in a blockchain setting, the witness could be a list of valid digital signatures authorizing several blockchain transactions. A SNARK can be used to to prove that one *knows* the signatures, so that the signatures themselves do not have to be stored and verified by all blockchain nodes. Instead, only the SNARK proof needs to be stored and verified on-chain.

1.1 SNARKs for Virtual Machine abstractions

A popular approach to SNARK design today is to prove the correct execution of *computer programs.* This means that the prover proves that it correctly ran a specified computer program Ψ on a witness. In the example above, Ψ might take as input a list of blockchain transactions and associated digital signatures authorizing each of them, and verify that each of the signatures is valid.

[*]New York University
[†]Microsoft Research
[‡]a16 crypto research and Georgetown University

Many projects today accomplish this via a CPU abstraction (in this context, also often called a *Virtual Machine (VM)*). Here, a VM abstraction entails fixing a set of *primitive instructions*, known as an instruction set architecture (ISA), analogous to assembly instructions in processor design. A full specification of the VM also includes the number of registers, and the type of memory that is supported. The computer program that the prover proves it ran correctly must be specified in this assembly language.

To list a few examples, several so-called zkEVM projects seek to achieve "byte-code level compatibility" with the Ethereum Virtual Machine (EVM), which means that the set of primitive instructions is the 141 opcodes available on the EVM. Other zkEVMs do not aim for byte-code level compatibility, instead aiming to offer SNARKs for high-level smart contract languages such as Solidity (without first compiling the solidity to EVM bytecode).

Still other so-called zkVM projects take a similar approach but do not target the EVM instruction set, nor high-level languages like Solidity that are often compiled to EVM bytecode. These projects typically choose (or design) ISAs for their purported "SNARK-friendliness", or for surrounding infrastructure and tooling, or for a combination thereof. For example, Cairo-VM is a very simple virtual machine designed specifically for compatibility with SNARK proving [GPR21, AGL$^+$22]. The VM has 3 registers, memory that is read-only (each cell can only be written to once) and must be "continuous", and the primitive instructions are roughly addition and multiplication over a finite field, jumps, and function calls.[1]

Another example is the RISC Zero project, which uses the RISC-V instruction set. RISC-V is popular in the computer architecture community, and comes with a rich ecosystem of compiler tooling to transform higher-level programs into RISC-V assembly. Other zkVM projects include Polygon Miden[2], Valida[3] and many others.

Front-end, back-end paradigm. SNARKs are built using protocols that perform certain probabilistic checks, so to apply SNARKs to program executions, one must express the execution of a program in a specific form that is amenable to probabilistic checking (e.g., as arithmetic circuits or generalizations thereof). Accordingly, most SNARKs consist of a so-called *front-end* and *back-end*: the front-end transforms a witness-checking computer program Ψ into an equivalent circuit-satisfiability instance, and the back-end allows the prover to establish that it knows a satisfying assignment to the circuit.

Typically, the circuit will "execute" each step of the compute program one at a time (with the help of untrusted "advice inputs"). Executing a step of the CPU conceptually involves two tasks. (1) Identify which primitive instruction should be executed at this step (2) execute the instruction and update the CPU state appropriately.

Existing front-ends implement these tasks by carefully devising gates or so-called constraints that implement each instruction. This is time-intensive and potentially error-prone. As we show in this work, it also leads to circuits that are substantially larger than necessary.

Pros and cons of the zkVM paradigm. One major benefit of zkVMs that use pre-existing ISAs is that they can exploit extant compiler infrastructure and tooling. This applies, for example, to the RISC-V and EVM instruction set, and leads to a developer-friendly toolchain without building the infrastructure from scratch. One can directly invoke existing compilers that transform witness-checking programs written in high-level languages down to assembly code for the ISA, and benefit from prior audits or other verification efforts of these compilers.

Another benefit of zkVMs is that a single circuit can suffice for running all programs up to a certain time bound, whereas alternative approaches may require re-running a front-end for every program (see the

[1]The Cairo toolchain allows programmers to write programs in a higher-level language called Cairo 1.0, and these programs are compiled into primitive instructions for the Cairo-VM. Even the high-level language only exposes write-once (also known as immutable) memory to the programmer and does not offer signed integer data types. See `https://www.cairo-lang.org/` for information on the high-level language and [GPR21, AGL$^+$22] and `https://github.com/lambdaclass/cairo-vm` for information on the Virtual Machine.

[2]`https://polygon.technology/polygon-miden`

[3]`https://github.com/valida-xyz/valida-compiler/issues/2`

discussion in Section 1.6 of other front-end approaches). Finally, frontends for VM abstractions output circuits with repeated structure. For a given circuit size, backends targeting circuits with repeated structure [Set20, BSBHR19, WTS$^+$18] can be much faster than backends that don't leverage repeated structure [CHM$^+$20, GWC19, Gro16].

However, zkVMs also have downsides that render them inappropriate for some applications. First, circuits implementing a VM abstraction are often much larger than circuits that do not. This means that zkVM provers are often much slower end-to-end than SNARK provers that do not impose upon themselves a VM abstraction.

For example, implementing certain important operations in a zkVM (e.g., cryptographic operations such as Keccak hashing or ECDSA signature verification) is extremely expensive—e.g., ECDSA signature verification takes up to 100 microseconds to verify on real CPUs, which translates to millions of RISC-V instructions[4]. This is why zkVM projects contain so-called gadgets or built-ins, which are hand-optimized circuits and lookup tables computing specific functionalities.

A second downside is that, in order to expose a high-level programming language to developers, zkVMs require a compiler that transforms such high-level computer programs into assembly code for the VM. These compilers represent a large attack surface. Any bug in the compiler can render the system insecure: proving that one correctly ran assembly code does not guarantee knowledge of a valid witness if the assembly code fails to correctly implement the intended witness-checking procedure.

The conventional wisdom on zkVMs and known counterpoints. The prevailing viewpoint today is that simpler VMs can be turned into circuits with fewer gates per step of the VM. This is most apparent in the design of particularly simple and ostensibly SNARK-friendly VMs such as the Cairo-VM. However, this comes at a cost, because primitive operations that are standard in real-world CPUs require many primitive instructions to implement on the simple VM.

In part to minimize the overheads in implementing standard operations on such limited VMs, many projects have designed domain specific languages (DSLs) that are exposed to the programmer who writes the witness-checking program. The proliferation of DSLs places a burden on the programmer, who is responsible both for learning the DSL and writing correct programs in it (with catastrophic security consequences if a program is incorrect).

Moreover, existing zkVMs remain expensive for the prover, even for very simple ISAs. For example, the prover for Cairo-VM programs described in [GPR21, AGL$^+$22] cryptographically commits to 51 field elements per step of the Cairo-VM. This means that a single primitive instruction for the Cairo-VM may cause the prover to execute millions of instructions on real CPUs. This severely limits the applicability of SNARKs for VM abstractions, to applications involving only very simple witness-checking procedures.

1.2 Jolt: A new paradigm for zkVM design

In this work, we introduce a new paradigm in zkVM design. The result is zkVMs with much faster provers, as well as substantially improved auditability and extensibility (i.e., a simple workflow for adding additional primitive instructions to the VM). Our techniques are general. As a concrete example, we instantiate them for the RISC-V instruction set (with multiplication extension [And17]), a popular open-source ISA developed by the computer architecture community without SNARKs in mind.

Our results upend the conventional wisdom that simpler instruction sets necessarily lead to smaller circuits and associated faster provers. First, our prover is faster per step of the VM than existing SNARK provers for much simpler VMs. Second, the complexity of our prover primarily depends on the size (i.e., number of bits) of the inputs to each instruction. This holds so long as all of the primitive instructions satisfy a natural notion of structure, called *decomposability.* Roughly speaking, decomposability means that one can evaluate the instruction on a given pair of inputs (x, y) by breaking x and y up into smaller chunks, evaluating a small number of functions of each chunk, and combining the results. A primary contribution of our work is to show that decomposability is satisfied by all instructions in the RISC-V instruction set.

[4]See `https://github.com/risc0/risc0/tree/v0.16.0/examples/ecdsa`.

Lookup arguments and Lasso. In a lookup argument, there is a predetermined "table" T of size N, meaning that $T \in \mathbb{F}^N$. An (*unindexed*) lookup argument allows the prover to commit to any vector $a \in \mathbb{F}^m$ and prove that every entry of a resides somewhere in the table. That is, for every $i \in \{1, \dots, m\}$, there exists some k such that $a_i = T[k]$.

In an *indexed* lookup argument, the prover commits not only to $a \in \mathbb{F}^m$, but also a vector $b \in \mathbb{F}^m$, and the prover proves that for every i, $a_i = t[b_i]$. In this setting, we call a the vector of *lookups* and b the vector of associated *indices*.

In a companion paper, we describe a new lookup argument called Lasso (which applies to both indexed and unindexed lookups). One distinguishing feature of Lasso is that it applies even to tables that are far too large for anyone to materialize in full, so long as the table satisfies the *decomposability* condition mentioned earlier.

Jolt. Say $\mathcal{P}$ claims to have run a certain computer program for m steps, and that the program is written in the assembly language for a VM. Today, front-ends produce a circuit that, for each step of the computation: (1) figures out what instruction to execute at that step and then (2) executes that instruction.

Lasso lets one replace Step 2 with a single lookup. For each instruction, the table stores the entire evaluation table of the instruction. If instruction f operations on two 64-bit inputs, the table stores $f(x, y)$ for every pair of inputs $(x, y) \in \{0,1\}^{64} \times \{0,1\}^{64}$. This table has size 2^{128}. In this work, we show that all RISC-V instructions are decomposable.

1.3 Jolt costs

1.3.1 Background and context

Polynomial commitments and MSMs. A central component of most SNARKs is a cryptographic protocol called a *polynomial commitment scheme*. Such a scheme allows an untrusted prover to succinctly commit to a polynomial p and later reveal an evaluation $p(r)$ for a point r chosen by the verifier (the prover will also return a *proof* that the claimed evaluation is indeed equal to the committed polynomial's evaluation at r). In Jolt, as with most SNARKs, the bottleneck for the prover is the polynomial commitment scheme.

Many popular polynomial commitments are based on multi-exponentiations (also known as multi-scalar multiplications, or MSMs). This means that the commitment to a polynomial p (with n coefficients $c_0, \dots, c_{n-1}$ over an appropriate basis) is

$$\prod_{i=0}^{n-1} g_i^{c_i},$$

for some public generators $g_1, \dots, g_n$ of a multiplicative group $\mathbb{G}$. Examples include KZG [KZG10], Bulletproofs/IPA [BCC+16, BBB+18], Hyrax [WTS+18], and Dory [Lee21].[5]

The naive MSM algorithm performs n group exponentiations and n group multiplications (note that each group exponentiation is about $400\times$ slower than a group multiplication). But Pippenger's MSM algorithm saves a factor of about $\log(n)$ relative to the naive algorithm. This factor can be well over $10\times$ in practice.

Working over large fields, but committing to small elements. If all exponents appearing in the multi-exponentiation are "small", one can save another factor of $10\times$ relative to applying Pippenger's algorithm to an MSM involving random exponents. This is analogous to how computing $g_i^{2^{16}}$ is $10\times$ faster than computing $g_i^{2^{160}}$: the first requires 16 squaring operations, while the second requires 160 such operations.

In other words, if one is promised that all field elements (i.e., exponents) to be committed via an MSM are in the set $\{0, 1, \dots, K\} \subset \mathbb{F}$, the number of group operations required to compute the MSM depend only on K and not on the size of $\mathbb{F}$.[6]

[5]In Hyrax and Dory, the prover does $\sqrt{n}$ MSMs each of size $\sqrt{n}$

[6]Of course, the cost of each group operation depends on the size of the group's base field, which is closely related to that of the scalar field $\mathbb{F}$. However, the *number* of group operations to compute the MSM depends only on K, not on $\mathbb{F}$.

Quantitatively, if all exponents are upper bounded by some value K, with $K \gg n$, then Pippenger's algorithm only needs (about) one group *operation* per term in the multi-exponentiation. More generally, with any MSM-based commitment scheme, Pippenger's algorithm allows the prover to commit to roughly $k \cdot \log(n)$-bit field elements (meaning field elements in $\{0, 1, \ldots, n\}$) with only k group *operations* per committed field element. This means that for size-n MSMs, one can commit to $\log(n)$ bits with a *single* group operation.

1.3.2 Costs of Jolt

Prover costs For RISC-V instructions on 64-bit data types (with the multiply extension), Jolt's $\mathcal{P}$ commits to under 50 field elements per step of the RISC-V CPU. Only seven of those field elements are larger than 2^{25}, and none of them are larger than 2^{64}. With MSM-based polynomial commitment, the Jolt prover costs are roughly that of committing to 5 arbitrary (256-bit) field elements per CPU step.

AA: TODO: insert tiny table showcasing numbers for both RV32IM and RV64IM

One caveat is that we handle six check this number RISC-V instructions via several "pseudoinstructions". For example, we handle the division with remainder instruction by having $\mathcal{P}$ provide the quotient and remainder as untrusted advice, and they are checked for correctness by applying multiplication and addition instructions. Conversely, many instructions (those involving addition, subtraction, shifts, jumps, loads, and stores) can be handled with *fewer than* five committed 256-bit field elements.

Comparison of prover costs to prior works. A detailed experimental comparison of Jolt to existing zkVMs will have to wait until a full implementation is complete, but some crude comparisons to prior works are illustrative. Recall that, when using an MSM-based multilinear polynomial commitment scheme (such as multilinear analogs of KZG, like Zeromorph [KT23]) we estimate the cost of the JOLT prover as being roughly that of committing to five arbitrary 256-bit field elements per step of the RISC-V CPU.

Plonk [GWC19] is a popular backend that can prove statements about certain generalizations of arithmetic circuit satisfiability. When Plonk is applied to an arithmetic circuit (i.e., consisting of addition and multiplication gates of fan-in two), the Plonk prover commits to 11 field elements per gate of the circuit, and 7 of these 11 field elements are random. Thus, the Jolt prover costs are roughly equivalent to applying the Plonk backend to an arithmetic circuit with only about one gate per step of the RISC-V CPU.

A more apt comparison is to the RISC Zero project[7], which currently targets the RISC-V ISA on 32-bit data types (with the multiplication extension). A direct comparison is complicated, in part because RISC Zero uses FRI as its (univariate) polynomial commitment scheme, which is based on FFTs and Merkle-hashing, avoiding the use of elliptic curve groups. Jolt can use related polynomial commitment schemes (Jolt can use any commitment scheme for multilinear polynomials). However, we choose to focus on elliptic-curve-based schemes, because Jolt's property of having the prover commit only to elements in $\{0, \ldots, b\}$ for some $b \ll |\mathbb{F}|$ benefits those commitment schemes more than hashing-based ones.[8] Still, a crude comparison can be made by comparing how many field elements the RISC Zero prover commits to, vs. the Jolt prover.

The RISC Zero prover commits to at least 275 31-bit field elements per CPU step [Tom23]. This is roughly equivalent to committing to about $275 \cdot 32/256 \approx 34$ different 256-bit field elements per CPU step: at least on small instances, the prover bottleneck is Merkle-hashing the result of various FFTs [Tom23], and one can hash 8 different 31-bit field elements with the same cost as hashing one 256-bit field element.

A final comparison point is to the SNARK for the Cairo-VM described in the Cairo whitepaper [GPR21]. The prover in that SNARK commits to about 50 field elements per step of the Cairo Virtual Machine, using FRI as the polynomial commitment scheme. StarkWare currently works over a 251-bit field.[9] This field size may be larger than necessary (it is chosen to match the field used by certain ECDSA signatures), but the provided arithmetization of Cairo-VM *requires* a field of size at least 2^{63}. So the commitment costs for the prover

[7]`https://www.risczero.com/`

[8]This property would also benefit hashing-based commitment schemes that operate over an extension field of a relatively small base field, owing to all committed elements in Lasso being in the base field.

[9]See, for example, `https://github.com/starkware-libs/starkex-contracts/blob/master/audit/EVM_STARK_Verifier_v4.0_Audit_Report.pdf`.

are at least equivalent to committing to $50 \cdot 64/256 \approx 13$ 256-bit field elements.[10] Jolt's prover costs per CPU compare favorably to this, despite the RISC-V instruction set being vastly more complicated than the Cairo-VM (and with the Cairo-VM instruction set specifically designed to be ostensibly "SNARK-friendly"). AA: STARK-friendly?

Verifier costs of Jolt. For RISC-V programs running for at most T steps, the dominant costs for the Jolt verifier are performing $O(\log(T)\log\log(T))$ hash evaluations and field operations,[11] plus checking one evaluation proof from the chosen polynomial commitment scheme (when applied to a multilinear polynomial over at most $O(\log T)$ variables).

Verifier costs can be further reduced, and the SNARK rendered zero-knowledge, via composition with a zero-knowledge SNARK with smaller proof size. For example, see the recent work Testudo for a related approach (Testudo instantiates Spartan [Set20] with a variant of PST polynomial commitments [PST13] (an analog of KZG commitments [KZG10] for multilinear rather than univariate polynomials) and composes this with Groth16 [Gro16].

1.4 The lookup singularity

In a research forum post in 2022, Barry Whitehat articulated a goal of designing front-ends that produce circuits that *only* perform lookups [Whi]. Whitehat terms this the *lookup singularity* and sketches how achieving this would help address a key issue (the potential for security bugs, and difficulty of auditability) that must be addressed for long-term and large-scale adoption of SNARKs. Circuits that only perform lookups (and the lookup arguments that enable them) should be much simpler to understand and formally verify than circuits consisting of many gates that are often hand-optimized.

Whitehat's post acknowledges that current lookup arguments are expensive, but predicts that lookup arguments will get more performative with time. Arguably, Jolt realizes the vision of the lookup singularity. The bulk of the prover work in Jolt lies in the lookup argument, Lasso. The Jolt front-end does output some constraints that effectively implement the task of the RISC-V CPU figuring out, at each step of the computation, which instruction to execute. These constraints are simple and easily captured in R1CS.

1.5 Technical details: CPU instructions as structured polynomials

talk about figuring out in some cases how to quickly compute the MLE, keeping some tables smaller (add and sub tables), etc.

Lasso is most efficient when applied to lookup tables satisfying a property called *decomposability*. Intuitively, this refers to tables t such that one lookup into t of size N can be answered with a small number (say, about c) of lookups into much smaller tables $t_1, \dots, t_\ell$, each of size $N^{1/c}$. Furthermore, if a certain polynomial $\widetilde{t}_i$ associated with each t_i can be evaluated at any desired point r using, say, $O(\log(N)/c)$ field operations,[12] then no one needs to cryptographically commit to any of the tables (neither to t itself, nor to $t_1, \dots, t_\ell$). Specifically, $\widetilde{t}_i$ can be any so-called *low-degree extension* polynomial of t_i. In Jolt, we will exclusively work with a specific low-degree extension of t_i, called the *multilinear extension*, and denoted $\widetilde{\mathsf{t}}_i$.

Hence, to take full advantage of Lasso, we must show two things:

- The evaluation table t of each RISC-V instruction has is decomposable in the above sense. That is, one lookup into t, which has size N, can be answered with a small number of lookups into much smaller tables $t_1, \dots, t_\ell$, each of size $N^{1/c}$. For most RISC-V instructions, ℓ equals one or two, and about c lookups are performed into each table.

[10] Furthermore, in order to control proof size, StarkWare currently uses a "FRI blowup factor" of 16, compared to RISC Zero's choice of 4. This adds at least an extra factor of 4 to the prover time per field element committed, relative to RISC Zero's.

[11] As described in Appendix C.3, Lasso can use any so-called *grand product argument*. The $O(\log(T)\log\log(T))$ verifier cost are due to the choice of grand product argument from [SL20, Section 6]. Other choices of lookup argument offer different tradeoffs between commitment costs for the prover, versus proof size and verifier time.

[12] The Lasso verifier has to evaluate $\widetilde{t}_i$ at a random point r on its own, so we need this computation to be fast enough that we are satisfied with the resulting verifier runtime. For all tables arising in Jolt, the verifier can compute all necessary $\widetilde{t}_i$ polynomial evaluations in $O(\log(N))$ total field operations.

- For each of the small tables t_i, the multilinear extension $\widetilde{t_i}$ is evalutable at any point, using just $O(\log(N)/c)$ field operations.

Establishing the above is the main technical contribution of our work. It turns out to be quite straightforward for certain instructions (e.g., bitwise AND), but more complicated for others (e.g., bitwise shifts, comparisons).

Decomposable instructions. Suppose that table t contains all evaluations of some primitive instruction $f\colon \{0,1\}^n \to \mathbb{F}$. Decomposability of the table t is equivalent to the following property of f: for any n-bit input x to f, x can be decomposed into c "chunks", $X_0, \ldots, X_{c-1}$, each of size n/c, and such that there following holds. There are ℓ functions $f_0, \ldots, f_{\ell-1}$ such that $f(x)$ can be derived in a relatively simple manner from $f_i(x_j)$ as i ranges over $0, \ldots, \ell-1$ and j ranges over $0, \ldots, c-1$. Then the evaluation table t of f is decomposable: one lookup into t can be answered with c total lookups into $\ell \cdot c$ lookups into the evaluation tables of $f_0, \ldots, f_{\ell-1}$.

Bitwise AND is a clean example by which to convey intuition for why the evaluation tables of RISC-V instructions are decomposable. Suppose we have two field elements a and b in $\mathbb{F}$, both in $\{0, \ldots, 2^{64}-1\}$. We refer to a and b as 64-bit field elements (we clarify here that "64 bits" does *not* refer to the size of the *field* $\mathbb{F}$, which may, for example, be a 256-bit field. Rather to the fact that a and b are both in the much smaller set $\{0, \ldots, 2^{64}-1\} \subset \mathbb{F}$, no matter how large $\mathbb{F}$ may be).

Our goal is to determine the 64-bit field element c whose binary representation is given by the bitwise AND of the binary representations of a and b. That is, if $a = \sum_{i=0}^{63} 2^i \cdot a_i$ and $b = \sum_{i=0}^{63} 2^i \cdot b_i$ for $(a_0, \ldots, a_{63}) \in \{0,1\}^{64}$ and $(b_0, \ldots, b_{63}) \in \{0,1\}^{64}$, then $c = \sum_{i=0}^{63} 2^i \cdot a_i \cdot b_i$.

One way to compute c is as follows. Break a and b into 8 chunks of 8 bits each compute the bitwise AND of each chunk, and concatenate the results to obtain c. Equivalently, we can express

$$c = \sum_{i=0}^{7} 2^{8 \cdot i} \cdot \mathsf{AND}(a'_i, b'_i), \tag{1}$$

where each $a'_i, b'_i \in \{0, \ldots, 2^8 - 1\}$ is such that $a = \sum_{i=0}^{7} 2^{8 \cdot i} \cdot a'_i$ and $b = \sum_{i=0}^{7} 2^{8 \cdot i} \cdot b'_i$. These a'_i's and b'_i's represent the decomposition of a and b into 8-bit limbs.[13]

In this way, one lookup into the evaluation table of bitwise-AND, which has size 2^{128}, can be answered by the prover providing $a'_1, \ldots, a'_8, b'_1, \ldots b'_8 \in \{0, \ldots, 2^8 - 1\}$ as untrusted advice, and performing 8 lookups into the size-2^{16} table t_1 containing all evaluations of bitwise-AND over pairs of 8-bit inputs. The results of these 8 lookups can easily be collated into the result of the original lookup, via Equation (1). No party has to commit to the size-2^{16} table t_1 because for any input $(r'_0, \ldots, r'_7, r''_0, \ldots, r''_7) \in \mathbb{F}^{16}$,

$$\widetilde{t}_1(r'_0, \ldots, r'_7, r''_0, \ldots, r''_7) = \sum_{i=0}^{15} 2^i \cdot r'_i \cdot r''_i,$$

which can be evaluated directly by the verifier with only 32 field operations.

Challenges for other instructions. One may initially expect that correct execution of RISC-V operations capturing 64-bit addition and multiplication would be easy prove, because large prime-order fields come with addition and multiplication operations that behave like integer addition and multiplication until the result of the operation overflows the field characteristic. Unfortunately, the RISC-V instructions capturing addition and multiplication have specified behavior upon overflow that differs from that of field addition and multiplication. Resolving this discrepancy is one key challenge that we overcome.

come back to this

[13]Just as "digits" refers a base-10 decomposition of an integer or field element, "limbs" refer to a decomposition into a different base, in this case base 8.

1.6 Other front-end approaches

As with other zkVM projects, Jolt produces a so-called *universal circuit*, meaning one circuit works for all RISC-V programs running up to some time bound T. AA: We use m in other places to be consistent with number of lookups. This has the benefit that the circuit-generation process only needs to be run once.

Other front-end approaches do not implement a Virtual Machine abstraction (i.e., they do not produce circuits that repeatedly execute the transition function of a specific ISA). These approaches typically output a different circuit for every computer program, such as Buffet [WSR+15], Bellman, Circom, Zokrates, Noir, etc. The circuits produced by these approaches can also be made smaller and proved faster using our techniques, though we leave this to future work.

2 Technical Preliminaries

2.1 Multilinear extensions

An ℓ-variate polynomial $p\colon \mathbb{F}^\ell \to \mathbb{F}$ is said to be *multilinear* if p has degree at most one in each variable. Let $f\colon \{0,1\}^\ell \to \mathbb{F}$ be any function mapping the ℓ-dimensional Boolean hypercube to a field $\mathbb{F}$. A polynomial $g\colon \mathbb{F}^\ell \to \mathbb{F}$ is said to *extend* f if $g(x) = f(x)$ for all $x \in \{0,1\}^\ell$. It is well-known that for any $f\colon \{0,1\}^\ell \to \mathbb{F}$, there is a unique *multilinear* polynomial $\widetilde{f}\colon \mathbb{F} \to \mathbb{F}$ that extends f. The polynomial $\widetilde{f}$ is referred to as the *multilinear extension* (MLE) of f.

Multilinear extensions of vectors. Given a vector $u \in \mathbb{F}^m$, we will often refer to the *multilinear extension of* u and denote this multilinear polynomial by $\widetilde{u}$. $\widetilde{u}$ is obtained by viewing u as a function mapping $\{0,1\}^{\log m} \to \mathbb{F}$ in the natural way[14]: the function interprets its $(\log m)$-bit input $(i_0, \ldots, i_{\log m-1})$ as the binary representation of an integer i between 0 and $m-1$, and outputs u_i. $\widetilde{u}$ is defined to be the multilinear extension of this function.

Lagrange interpolation. An explicit expression for the MLE of any function is given by the following standard lemma (see [Tha22, Lemma 3.6]).

Lemma 1. *Let $f\colon \{0,1\}^\ell \to \mathbb{F}$ be any function. Then the following multilinear polynomial $\widetilde{f}$ extends f:*

$$\widetilde{\mathsf{f}}(x_0, \ldots, x_{\ell-1}) = \sum_{w \in \{0,1\}^\ell} f(w) \cdot \chi_w(x_0, \ldots, x_{\ell-1}), \tag{2}$$

where, for any $w = (w_0, \ldots, w_{\ell-1})$, $\chi_w(x_0, \ldots, x_{\ell-1}) \coloneqq \prod_{i=0}^{\ell} (x_i w_i + (1-x_i)(1-w_i))$. Equivalently,

$$\chi_w(x_0, \ldots, x_{\ell-1}) = \widetilde{\mathsf{eq}}(x_0, \ldots, x_{\ell-1}, w_0, \ldots, w_{\ell-1}).$$

The polynomials $\{\chi_w : w \in \{0,1\}^\ell\}$ are called the *Lagrange basis polynomials* for ℓ-variate multilinear polynomials. The evaluations $\{\widetilde{f}(w) : w \in \{0,1\}^\ell\}$ are sometimes called the coefficients of $\widetilde{f}$ *in the Lagrange basis*, terminology that is justified by Equation (2).

do we need sum-check?AA: I'd guess no.

SNARKs. We adapt the definition provided in [KST22].

Definition 2.1. *Consider a relation $\mathcal{R}$ over public parameters, structure, instance, and witness tuples. A non-interactive argument of knowledge for $\mathcal{R}$ consists of PPT algorithms $(\mathcal{G}, \mathcal{P}, \mathcal{V})$ and deterministic $\mathcal{K}$, denoting the generator, the prover, the verifier and the encoder respectively with the following interface.*

- $\mathcal{G}(1^\lambda) \to \mathsf{pp}$*: On input security parameter λ, samples public parameters* pp.

[14] All logarithms in this paper are to base 2.

- $\mathcal{K}(\mathsf{pp}, \mathsf{s}) \to (pk, \mathsf{vk})$*: On input structure* s*, representing common structure among instances, outputs the prover key* pk *and verifier key* vk.
- $\mathcal{P}(pk, u, w) \to \pi$*: On input instance* u *and witness* w*, outputs a proof* π *proving that* $(\mathsf{pp}, \mathsf{s}, u, w) \in \mathcal{R}$.
- $\mathcal{V}(\mathsf{vk}, u, \pi) \to \{0, 1\}$*: On input the verifier key* vk*, instance* u*, and a proof* π*, outputs 1 if the instance is accepting and 0 otherwise.*

A non-interactive argument of knowledge satisfies completeness if for any PPT adversary $\mathcal{A}$

$$\Pr\left[\mathcal{V}(\mathsf{vk}, u, \pi) = 1 \;\middle|\; \begin{array}{l} \mathsf{pp} \leftarrow \mathcal{G}(1^\lambda), \\ (\mathsf{s}, (u, w)) \leftarrow \mathcal{A}(\mathsf{pp}), \\ (\mathsf{pp}, \mathsf{s}, u, w) \in \mathcal{R}, \\ (pk, \mathsf{vk}) \leftarrow \mathcal{K}(\mathsf{pp}, \mathsf{s}), \\ \pi \leftarrow \mathcal{P}(pk, u, w) \end{array} \right] = 1.$$

A non-interactive argument of knowledge satisfies knowledge soundness if for all PPT adversaries $\mathcal{A}$ *there exists a PPT extractor* $\mathcal{E}$ *such that for all randomness* ρ

$$\Pr\left[\begin{array}{l} \mathcal{V}(\mathsf{vk}, u, \pi) = 1, \\ (\mathsf{pp}, \mathsf{s}, u, w) \notin \mathcal{R} \end{array} \;\middle|\; \begin{array}{l} \mathsf{pp} \leftarrow \mathcal{G}(1^\lambda), \\ (\mathsf{s}, u, \pi) \leftarrow \mathcal{A}(\mathsf{pp}; \rho), \\ (pk, \mathsf{vk}) \leftarrow \mathcal{K}(\mathsf{pp}, \mathsf{s}), \\ w \leftarrow \mathcal{E}(\mathsf{pp}, \rho) \end{array} \right] = \mathit{negl}(\lambda).$$

A non-interactive argument of knowledge is succinct if the verifier's time to check the proof π *and the size of the proof* π *are at most polylogarithmic in the size of the statement proven.*

Polynomial commitment schemes. AA: The macros for Commit and Open are footnotesized. Just want to make sure this is intentional? We adapt the definition from [BFS20]. A polynomial commitment scheme for multilinear polynomials is a tuple of four protocols $\mathsf{PC} = (\mathsf{Gen}, \mathsf{Commit}, \mathsf{Open}, \mathsf{Eval})$:

- $\mathsf{pp} \leftarrow \mathsf{Gen}(1^\lambda, \ell)$: takes as input ℓ (the number of variables in a multilinear polynomial); produces public parameters pp.
- $\mathcal{C} \leftarrow \mathsf{Commit}(\mathsf{pp}, \mathcal{G})$: takes as input a ℓ-variate multilinear polynomial over a finite field $\mathcal{G} \in \mathbb{F}[\ell]$; produces a commitment $\mathcal{C}$.
- $b \leftarrow \mathsf{Open}(\mathsf{pp}, \mathcal{C}, \mathcal{G})$: verifies the opening of commitment $\mathcal{C}$ to the ℓ-variate multilinear polynomial $\mathcal{G} \in \mathbb{F}[\ell]$; outputs $b \in \{0, 1\}$.
- $b \leftarrow \mathsf{Eval}(pp, \mathcal{C}, r, v, \ell, \mathcal{G})$ is a protocol between a PPT prover $\mathcal{P}$ and verifier $\mathcal{V}$. Both $\mathcal{V}$ and $\mathcal{P}$ hold a commitment $\mathcal{C}$, the number of variables ℓ, a scalar $v \in \mathbb{F}$, and $r \in \mathbb{F}^\ell$. $\mathcal{P}$ additionally knows a ell-variate multilinear polynomial $\mathcal{G} \in \mathbb{F}[\ell]$. $\mathcal{P}$ attempts to convince $\mathcal{V}$ that $\mathcal{G}(r) = v$. At the end of the protocol, $\mathcal{V}$ outputs $b \in \{0, 1\}$.

Definition 2.2. *A tuple of four protocols* $(\mathsf{Gen}, \mathsf{Commit}, \mathsf{Open}, \mathsf{Eval})$ *is an extractable polynomial commitment scheme for multilinear polynomials over a finite field* $\mathbb{F}$ *if the following conditions hold.*

- ***Completeness.*** *For any* ℓ*-variate multilinear polynomial* $\mathcal{G} \in \mathbb{F}[\ell]$,

$$\Pr\left\{ \begin{array}{c} \mathsf{pp} \leftarrow \mathsf{Gen}(1^\lambda, \ell);\ \mathcal{C} \leftarrow \mathsf{Commit}(\mathsf{pp}, \mathcal{G}): \\ \mathsf{Eval}(\mathsf{pp}, \mathcal{C}, r, v, \ell, \mathcal{G}) = 1 \wedge v = \mathcal{G}(r) \end{array} \right\} \geq 1 - \mathit{negl}(\lambda)$$

- ***Binding.*** *For any PPT adversary* $\mathcal{A}$*, size parameter* $\ell \geq 1$,

$$\Pr\left\{ \begin{array}{c} \mathsf{pp} \leftarrow \mathsf{Gen}(1^\lambda, \ell);\ (\mathcal{C}, \mathcal{G}_0, \mathcal{G}_1) = \mathcal{A}(\mathsf{pp}); \\ b_0 \leftarrow \mathsf{Open}(\mathsf{pp}, \mathcal{C}, \mathcal{G}_0);\ b_1 \leftarrow \mathsf{Open}(\mathsf{pp}, \mathcal{C}, \mathcal{G}_1): \\ b_0 = b_1 \neq 0 \wedge \mathcal{G}_0 \neq \mathcal{G}_1 \end{array} \right\} \leq \mathit{negl}(\lambda)$$

- ***Knowledge soundness.*** Eval *is a succinct argument of knowledge for the following NP relation given* $\mathsf{pp} \leftarrow \mathsf{Gen}(1^\lambda, \ell)$*:*

$$\mathcal{R}_{\mathsf{Eval}}(\mathsf{pp}) = \{\langle (\mathcal{C}, r, v), (\mathcal{G}) \rangle : \mathcal{G} \in \mathbb{F}[\mu] \wedge \mathcal{G}(r) = v \wedge \mathsf{Open}(pp, \mathcal{C}, \mathcal{G}) = 1\}.$$

2.2 Polynomial IOPs and polynomial commitments

Modern SNARKs are constructed by combining a type of interactive protocol called a *polynomial IOP* [BFS20] with a cryptographic primitive called a *polynomial commitment scheme* [KZG10]. The combination yields a succinct *interactive* argument, which can then be rendered non-interactive via the Fiat-Shamir transformation [FS86], yielding a SNARK.

Roughly, a polynomial IOP is an interactive protocol where, in one or more rounds, the prover may "send" to the verifier a very large polynomial g. Because g is so large, one does not wish for the verifier to read a complete description of g. Instead, in any efficient polynomial IOP, the verifier only "queries" g at one point (or a handful of points). This means that the only information the verifier needs about g to check that the prover is behaving honestly is one (or a few) evaluations of g.

In turn, a polynomial commitment scheme enables an untrusted prover to succinctly *commit* to a polynomial g, and later provide to the verifier any evaluation $g(r)$ for a point r chosen by the verifier, along with a proof that the returned value is indeed consistent with the committed polynomial. Essentially, a polynomial commitment scheme is exactly the cryptographic primitive that one needs to obtain a succinct argument from a polynomial IOP. Rather than having the prover send a large polynomial g to the verifier as in the polynomial IOP, the argument system prover instead cryptographically commits to g and later reveals any evaluations of g required by the verifier to perform its checks.

Whether or not a SNARK requires a trusted setup, as well as whether or not it is plausibly post-quantum secure, is determined by the polynomial commitment scheme used. If the polynomial commitment scheme does not require a trusted setup, neither does the resulting SNARK, and similarly if the polynomial commitment scheme is plausibly binding against quantum adversaries, then the SNARK is plausibly post-quantum sound.

Lasso can make use of any commitment schemes for *multilinear* polynomials g.[15] Here an ℓ-variate multilinear polynomial $g\colon \mathbb{F}^\ell \to \mathbb{F}$ is a polynomial of degree at most one in each variable.

2.3 Lookup arguments

Lookup arguments allow a prover to commit to two vectors $a \in \mathbb{F}^m$ and $b \in \mathbb{F}^m$ (with a polynomial commitment scheme) and prove that each entry a_i of vector a resides in index b_i of a pre-determined lookup table $T \in \mathbb{F}^N$. That is, For each $i = 1, \ldots, m$, $a_i = T[b_i]$. Here, to emphasize the interpretation of T as a table, we use square brackets $T[i]$ to denote the i'th entry of T. Here, if $b_i \notin \{1, \ldots, N\}$, then $t[b_i]$ is undefined, and hence $a_i \neq T[b_i]$. We refer to a as the vector of *looked-up values* and b as the vector of *indices.*

Definition 2.3 (Lookup arguments, indexed variant)**.** *Let $PC = (\mathsf{Gen}, \mathit{Commit}, \mathit{Open}, \mathit{Eval})$ be an extractable polynomial commitment scheme for multilinear polynomials over $\mathbb{F}$. A lookup argument (for* indexed lookups*) for table $T \in \mathbb{F}^N$ is a SNARK for the relation*

$$\left\{(\mathsf{pp}, \mathcal{C}_1, \mathcal{C}_2, w = (a, b))\colon a, b \in \mathbb{F}^m \wedge a_i = T[b_i] \textit{ for all } i \in \{1, \ldots, n\} \wedge \mathit{Open}(\mathsf{pp}, \mathcal{C}_1, \widetilde{a}) = 1 \wedge \mathit{Open}(\mathsf{pp}, \mathcal{C}_2, \widetilde{b}) = 1\right\}.$$

Here $w = (a, b) \in \mathbb{F}^m \times \mathbb{F}^m$ is the witness, while pp, $\mathcal{C}_1$, and $\mathcal{C}_2$ are public inputs.

Definition 2.3 captures so-called *indexed* lookup arguments (this terminology was introduced in our companion work [STW23]). Other works consider *unindexed* lookup arguments, in which only the vector vector $a \in \mathbb{F}^m$ of looked-up values is committed, and the prover claims that *there exists* a vector b of indices such that $a_i = T[b_i]$ for all $i = 1, \ldots, m$.

Definition 2.4 (Lookup arguments, unindexed variant)**.** *Let $PC = (\mathsf{Gen}, \mathit{Commit}, \mathit{Open}, \mathit{Eval})$ be an extractable polynomial commitment scheme for multilinear polynomials over $\mathbb{F}$. A lookup argument (for* indexed lookups*) for table $T \in \mathbb{F}^N$ is a SNARK for the relation*

$$\left\{(\mathsf{pp}, \mathcal{C}_1, \mathcal{C}_2, a)\colon a \in \mathbb{F}^m \wedge \textit{ for all } i \in \{1, \ldots, n\}, \textit{ there exists a } b_i \textit{ such that } a_i = T[b_i] \wedge \mathit{Open}(\mathsf{pp}, \mathcal{C}_1, \widetilde{a}) = 1\right\}.$$

Here $a \in \mathbb{F}^m \times \mathbb{F}^m$ is the witness, while pp and $\mathcal{C}_1$ are public inputs.

[15]Any univariate polynomial commitment scheme can be transformed into a multilinear one, though the transformations introduce some overhead (see, e.g., [CBBZ23, BCHO22, ZXZS20]).

Jolt primarily requires indexed lookups. However, a few instructions (namely ADVICE and MOVE) require range checks, which are naturally handled by unordered lookups (to prove that a value is in the range $\{0, \dots 2^L - 1\}$, perform an unordered lookup into the table T with $T[i] = i$ for $i = \{0, \dots, 2^L - 1\}$.

There are natural reductions in both directions, i.e., unindexed lookup arguments can be transformed into index lookup arguments and vice versa. To obtain an unindexed lookup argument from an indexed one, $\mathcal{P}$ separately commits to the index vector b and applies the indexed lookup argument. Obtaining an indexed lookup argument from an unindexed one is slightly more complicated and is detailed in Appendix **??**.

A companion work: Lasso. Our companion work [STW23] introduces a family of lookup arguments called Lasso. The lookup arguments in this family are the first that don't require any party to cryptographically commit to the table vector $T \in \mathbb{F}^N$, so long as T satisfies one of the two structural properties defined below.

Definition 2.5 (MLE-structured tables). *We say that a vector $T \in \mathbb{F}^N$ is* MLE-structured *if for any input $r \in \mathbb{F}^{\log(N)}$, $\widetilde{\mathsf{T}}(r)$ can be evaluated with $O(\log(N))$ field operations.*

Definition 2.6 (Decomposable tables). *Let $T \in \mathbb{F}^N$. We say that T is c-decomposable if there exist a constant k and $\alpha \le kc$ tables $T_1, \dots, T_\alpha$ each of size $N^{1/c}$ and each MLE-structured, as well as a multilinear α-variate polynomial g such that the following holds. As in Section 2.1, let us view T as a function mapping $\{0,1\}^{\log N}$ to $\mathbb{F}$ in the natural way, and view each T_i as a function mapping $\{0,1\}^{\log(N)/c} \to \mathbb{F}$. Then for any $r \in \{0,1\}^{\log N}$, writing $r = (r_1, \dots, r_c) \in \{0,1\}^{\log(N)/c}$,*

$$T[r] = g(T_1[r_1], \dots, T_k[r_1], T_{k+1}[r_2], \dots, T_{2k}[r_2], \dots, T_{\alpha-k+1}[r_c], \dots, T_\alpha[r_c]).$$

We refer to $T_1, \dots, T_\alpha$ as sub-tables.

come back to this

For any constant $c > 0$ and any c-decomposable table, our companion paper gives a lookup argument called Lasso, in which the prover commits to roughly $3cm + cN^{1/c}$ field elements. Moreover, all of these field elements are *small*, meaning that they are all in $\{0, \dots, m\}$ (specifically, they are counts for the number of times each entry of each subtable is read), or are elements of the subtables $T_1, \dots, T_\alpha$. The verifier performs $O(\log(m) \log\log(m))$ hash evaluations and field operations, processes one evaluation proof from the polynomial commitment scheme applied to a multilinear polynomial in $\log m$ variables, and evaluates $\widetilde{T}_1, \dots, \widetilde{T}_\alpha$ each at a single randomly chosen point.

Our companion paper also describes a lookup argument called Generalized-Lasso, which applies to any MLE-structured table, not just decomposable ones.[16] The main disadvantage of Generalized-Lasso relative to Lasso is that cm out of the $3cm + cN^{1/c}$ field elements committed by the Generalized-Lasso prover are random rather than small. As described in Section 1.3.1, such field elements can take an order of magnitude more work to commit to than small field elements.

The relationship between MLE-structured and decomposable tables. For any decomposable table $T \in \mathbb{F}^N$, there is some low-degree extension $\hat{T}$ of T (namely, an extension of degree at most k in each variable) that can be evaluated in $O(\log N)$ time. Specifically, the extension polynomial is

$$\hat{T}(r) = g(\widetilde{T}_1(r_1), \dots, \widetilde{T}_\alpha(r_c)).$$

In general, $\hat{T}$ is not necessarily multilinear, so a table being decomposable does not necessarily imply that it is MLE-structured. But Generalized-Lasso actually applies to any table with a low-degree extension that is evaluable in logarithmic time. In this sense, decomposability (the condition required to apply Lasso) is a strictly stronger condition than what is necessary to apply Generalized-Lasso.

In Jolt, we show *all* lookup tables used are *both* c-decomposable (for any integer $c > 0$) as well as MLE-structured. We choose to apply Lasso rather than Generalized-Lasso due to its superior efficiency (which

[16] In fact, Generalized-Lasso applies to any table with *some* low-degree extension, not necessarily its multilinear one, that is evaluable in logarithmic time.

comes from the prover only committing to small field elements, avoiding the need to commit to random field elements).

discuss pasting together tables for each instruction?

2.4 Offline Memory Checking

Any SNARK for VM execution has to perform *memory-checking*. This means that the prover must be able to commit to an execution trace for the VM (that is, a step-by-step record of what the VM did over the course of its execution), and the verifier has to find a way to confirm that the prover maintained memory correctly throughout the entire execution trace. In other words, the value purportedly returned by any read operation in the execution trace must equal the value most recently written to the appropriate memory cell. We use the term *memory-checking argument* to refer to a SNARK for the above functionality. Note that a lookup table $T \in \mathbb{F}^N$ can be viewed as a read-only memory of size N, with memory cell i initialized to $T[i]$. Hence, lookup argument for indexed lookups (Definition 2.3 is equivalent the a memory-checking argument for read-only memories.

A variety of memory-checking arguments have been described in the research literature [ZGK+18, BCG+18, STW23, BFR+13, BSCGT13] (with the underlying techniques rediscovered multiple times). The most efficient are based on lightweight fingerprinting techniques for the closely related problem of *offline memory checking* [Lip89, BEG+91]. In this work, we use such an argument due to Spice [SAGL18], but optimize it using Lasso.

For completeness, we describe the memory-checking argument of Spice in Appendix **??** and provide a brief description below. We provide an overview of other memory-checking arguments in Appendix C.

Fingerprinting reads and writes. Consider a array of memory cells addressed by their index. At the start of the proof, the prover commits to the trace of all reads (and writes) in the form of sets of tuples $\mathsf{Comm}_{RS} = \mathsf{Commit}(\{(k_i, v_i, t_i)\}_{i=0}^{\#reads})$, $\mathsf{Comm}_{WS} = \mathsf{Commit}(\{(k_i, v_i, t_i)\}_{i=0}^{\#writes})$, indicating that the t_i^{th} operation involved index k_i having value v_i either read from (or written to) it. Given this commitment, the Verifier samples randomness ρ. During the proof, a pair of succinct digests to the log of reads and writes, respectively, is maintained, along with a timestamp $t^{(s)}$ used track the operation count. Let "fingerprinting" function $\mathsf{fprint}_\rho(.)$ that maps strings to $\mathbb{F}_p$.

- **Writes:** At time t, if value v is to be written to location k, $\mathsf{WS}^{(s)}$ is updated using the tuple (k, v, t) as $\mathsf{WS}^{(s)} \leftarrow \mathsf{WS}^{(s)} \cdot \mathsf{fprint}_\rho(k, v, t)$.
- **Reads:** If location k is read from at time t, the prover passes as non-deterministic advice the purported value v along with the timestamp t' of the latest write to location k. The read digest is accordingly updated as $\mathsf{RS} \leftarrow \mathsf{RS} \cdot \mathsf{fprint}_\rho(k, v, t')$ and the value is "re-written" to the same location and the write digest is updated as $\mathsf{WS}' \leftarrow \mathsf{WS} * \mathsf{fprint}_\rho(k, v, t' + 1)$.

AA: Double check what needs to be written back. At the end of the execution, the prover employs the memory checking algorithm to show that the final digests are consistent. This proof will accept if and only if all operations were all consistent: that is, every read operation returned the latest value written to that location. See ?? of ?? for a proof.

3 An Overview of the RISC-V Instruction Set Architecture

This section provides a brief overview of the RISC-V instruction set architecture considered in this work. Our goal is to convey enough about the architecture that readers who have not previously encountered it can follow this paper. However, a complete specification is beyond the scope of this work, and can be found at [And17].[17] We also stick to regular control flow and do not support external events and other unusual run-time conditions like exceptions, traps, interrupts and CSR registers.

[17]Another helpful resource for interested readers is Lectures 5-8 at `https://inst.eecs.berkeley.edu/~cs61c/resources/su18_lec/`.

Informally, the RISC-V ISA consists of a CPU and a read-write memory, collectively called the *machine*.

Definition 3.1 (Machine State)**.** *The machine state consists of* (PC, $\mathcal{R}$, $\mathcal{M}$)*.* $\mathcal{R}$ *denotes the 32 integer registers, each of W bits, where W is 32 or 64.* $\mathcal{M}$ *is a linear read-write byte-addressable array consisting of a fixed number of total locations (such as* 2^{20}*) with each location storing 1 byte. The* `PC`*, also of W bits, is a separate register that stores the memory location of the instruction to be executed.*

Assembly programs consist of a sequence of instructions, each of which operate on the machine state. The instruction to be executed at a step is the one stored at the address pointed to by the `PC`. Unless specified by the instruction, the `PC` is advanced to the next memory location after executing the instruction. The RISC-V ISA specifies that all instructions are 32 bits long (i.e., 4 bytes), so advancing the `PC` to the next memory location entails incrementing `PC` by 4.

While RISC-V uses multiple formats to store instructions in memory, we can abstract away the details and represent all instructions in the following 5-tuple format.

Definition 3.2 (5-tuple RISC-V Instruction Format)**.** *Any RISC-V instruction can be written in the following format:* *`[opcode, rs1, rs2, rd, imm]`. ach instruction specifies an operation code that uniquely identifies it, at most two* source registers *`rs1`, `rs2`, a* destination register *`rd`, and a constant value `imm` (standing for "immediate") provided in the program code itself.*

Figure 1 provides a brief schematic of the CPU state change and instruction format. Operations read the source registers, perform some computation, and can do any or all of the following: read from memory, write to memory, store a value in `rd`, or update the `PC`. For example, the logical left-shift instruction "(SLL, `r5, r8, r2, -`)" takes the value stored in register #5, performs a logical left shift of the length stored in register #8, and stores the result in register #2 (and does not involve any immediates).

As another example, the branch instruction "(BEQ, `r5, r8, -, imm`)" sets `PC` to be `PC` + `imm` if the values stored in registers #5 and #8 are equal, or increments `PC` by 4, otherwise. (The destination register is not involved).

3.1 A brief overview of two's complement representation

An *unsigned L-bit data type* refers to a value $z \in \{0, 1, \ldots, 2^L - 1\}$. A *signed L-bit data type* (in twos-complement format) refers to a value $z \in \{-2^{L-1}, \ldots, 2^{L-1} - 1\}$. The twos-complement representation $[z_{L-1}, \ldots, z_0] \in \{0, 1\}^L$ of z is the unique vector such that

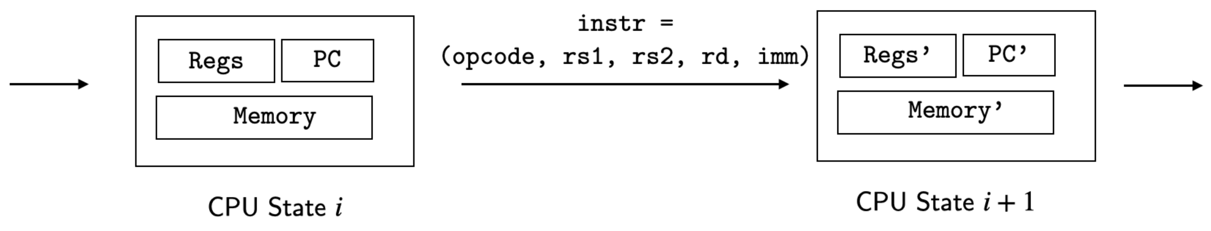
$$z = -z_{L-1} \cdot 2^{L-1} + \sum_{i=0}^{L-2} 2^i z_i. \tag{3}$$

For clarity, when discussing instructions interpreting their inputs as signed data types represented in twos-complement format (e.g., Section 5.1.2), we refer to z_{L-1} as the sign bit of z, and denote this by z_s. We use $z_{<s}$ to refer to $[z_{L-2}, \ldots, z_0] \in \{0, 1\}^{L-1}$.

Unsigned and signed data types. For the RISC-V ISA, data in registers has no type. A register simply stores L bits. However, different instructions can be conceptualized as interpreting register values in different ways.

Specifically, some instructions conceptually operate upon unsigned data types, while others operate over signed data types. All RISC-V instructions involving signed data types interpret the bits in a register as an integer via two's complement representation.[18] For many instructions, the use of two's complement has the consequence that the instruction operates identical regardless of whether or not the inputs are interpreted as signed or unsigned. For example, consider the ADD instruction when $L = 3$.

[18]See `https://en.wikipedia.org/wiki/Two%27s_complement` for an overview of how two's complement maps bit vectors in $\{0, 1\}^L$ to integers in $\{-2^L, \ldots, 2^L - 1\}$ and vice versa.



(a) The CPU state and instrution formats.

CPU Step Transition:

1. Read the instruction at location `PC` in Program Code.

   Parse instruction as `[opcode || rs1 || rs2 || rd || imm]`.
2. Read the W-bit values stored in registers `rs1`, `rs2`.
3. If required, write to or read from memory.

   The value written and memory location accessed is derived from the values stored in `rs1`, `imm`.
4. Perform the instruction's function on the values read from registers, memory and `imm` to get `result`.

   Examples of functions are arithmetic, logical and comparison operations.
5. Store `result` to register `rd`.

   Only a few instructions, like STOREs, do not involve `rd`.
6. Update `PC`.

   `PC` is usually incremented by 4, but instructions like jumps and branches update `PC` in other ways.

(b) The broad stages of a CPU step transition.

Figure 1: A model of RISC-V's CPU state and transition function. Note that the transition function is deterministic and all information required, such as the location of memory accessed, is derived from the CPU state and `instr`.

When adding three-bit unsigned integers 3 and 4, the addition operation proceeds as follows:

$$3 \text{ (i.e., 011) } + 4 \text{ (i.e., 100) } = 7 \text{ (i.e., 111)}.$$

Here, in parenthesis we have provided the binary representations of 3, 4, and 7 when interpreted as unsigned data types in two's-complement format.

When adding three-bit signed integers 3 and -4, the addition operation proceeds as follows:

$$3 \text{ (i.e., 011) } + -4 \text{ (i.e., 100) } = -1 \text{ (i.e., 111)}.$$

Again, in parentheses we have provided the binary representations of 3 and -4 when interpreted as signed data types in two's complement format.

The above example demonstrates that, when using two's complement binary representations, the input/output behavior of the addition operation is independent of whether the inputs are interpreted as signed or unsigned.

For some instructions, like multiplication MUL, and integer comparison, the desired input/output behavior differs depending on whether the inputs are interpreted as signed or unsigned. In these cases, there will be two different RISC-V instructions, one for each interpretation. For example, there are MUL and MULU instructions, with the former interpreting its inputs as signed, and the latter interpreting its inputs as unsigned. Similarly, there are two integer comparison operations, SLT and SLTU.

3.2 The Two Tools: Memory-Checking and Lookups

These two techniques put together gives us the proof system shown in Figure 2.

3.2.1 Memory Checking

The machine state transition involves reading from and writing to three conceptually separate part of memory: (1) the program code, (2) the registers and (3) the random access memory. As discussed in Section 2.4, the most efficient way to handle this is using the offline memory checking techniques. “Offline” here means the memory reads and writes required by the CPU execution are provided by the prover and the proof proceeds assuming these operations are correct. Finally, the correctness of these operations is proven after the execution is completed. Jolt thus maintains three pairs of running digests $(\mathsf{RS}^{(s)}, \mathsf{WS}^{(s)})$ for $s = \texttt{ProgramCode}, \texttt{Regs}, \texttt{Memory}$. These are updated with every memory operation and are finally proved consistent at the end. See Appendix ?? for more details about the digest and memory checking algorithm.

3.2.2 Instruction Logic using Lookups

As described in Section 2.3, the Jolt paradigm avoids the complexity of implementing each instruction’s logic as constraints by abstracting them all away into a lookup table. For example, ADD requires adding two W-bit operands x, y and storing the lowest W bits of the result r into the specified destination register (that is, any overflowing bit must be ignored). We can thus construct a “truth table” for each operation as $T_{\mathsf{opcode}}[x \parallel y] = r$ that contains the required result for all possible inputs x, y. Moreover, Jolt combined the tables for all instructions into one table and thus makes only one lookup query per step to this table as $T_{RISC-V}(\texttt{opcode} \parallel x \parallel y) = r$. Given a processor and instruction set, this table is constant and independent of the program or inputs.

This means that the main responsiblity of the constraint system is to prepare the right operands x, y at each step before the lookup. This is efficient to do as the operands only come from the set {value in rs1, value in rs2, imm}.

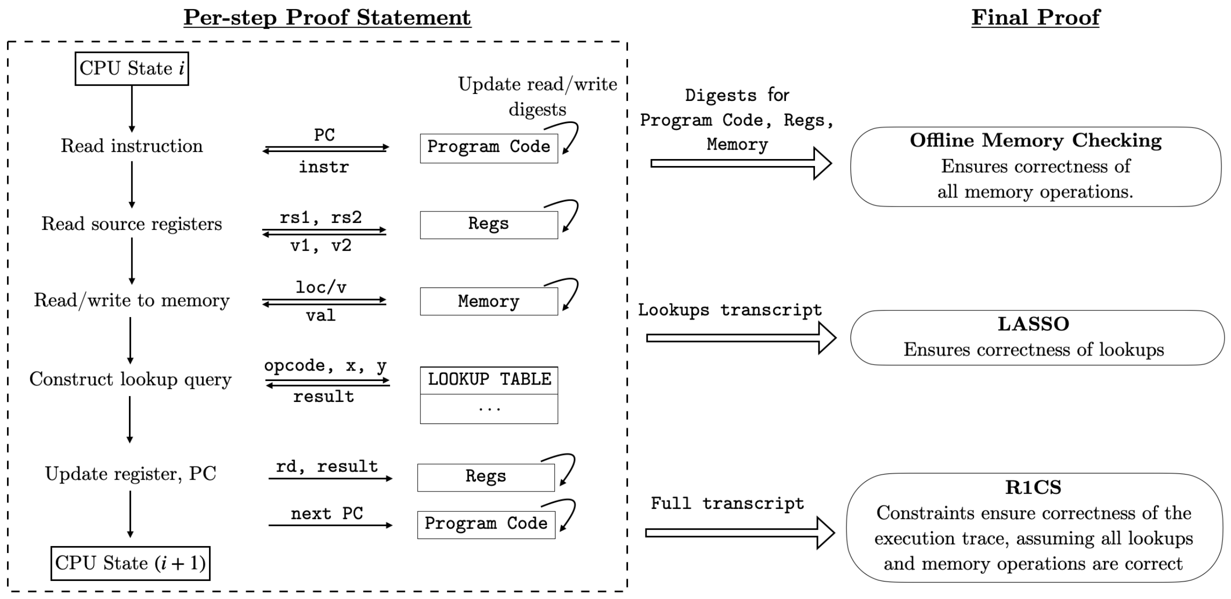


Figure 2: Proving the correctness of CPU execution using offline memory checking and lookups.

3.3 Supporting Byte-Addressable Loads and Stores

One of the challenges with RISC-V is supporting memory that is byte-addressable. To do this, Jolt performs up to $W/8$ memory operations per load/store, one for each byte written or read.

Stores. Each store instruction reads the lower k byte-suffix from `rs2` ($k = 8, 4, 2, 1$ for instructions SD, SW, SH, SB, respectively), sign-extends the suffix to W bits, and stores the result into memory location `loc` = `rs1` + `imm`. There are three steps involved in stores.

1. Jolt first employs a lookup argument on the value in `rs2`: these tables have an index for each possible W-bit z and return the sign-extension of the required suffix.
2. The prover provides as advice the bytes-decomposition (enforced using range checks) of the lookup result.
3. Memory-checking then stores theese bytes in the right locations (which starts at `rs1` + `imm`).

Loads. The load operations take k bytes of memory ($k = 8, 4, 2, 1$ for instructions LD, LW, LH, LB, respectively) from location `loc` = `rs1` + `imm`, sign-extends it to W bits and then stores the result in `rd`.

1. The k bytes are first read using memory-checking. Note that range checks are not required here as they were enforced during the stores.
2. Jolt concatenates these and employs a lookup to get the sign-extension. This lookup's result is stored in `rd` using memory-checking.

Range checks to enforce 8-bit values are performed using unindexed lookups and are efficient with Lasso, costling about ??? per element.

3.4 Formatting Assembly Code

Before the proof starts, the assembly code is formatted into the 5-tuple form of Definition 3.2: `(opcode, rs1, rs2, rd, imm)`. Additionally, each instruction also comes with 14 1-bit "flags" `opflags[14]` that guide the constraint system. For example, `opflag[5]` is 1 for only Jump instructions, and `opflags[7]` is 1 if and only if the lookup's result is to be stored in `rd`. Note that these flags are fixed for any given instruction. See Appendix A.1 for a list of all the flags.

In RISC-V, instructions maybe need to sign-extend or zero-extend `imm` to W bits. This is a deterministic choice made with just the instruction (and independent of the rest of the program or inputs). Thus, when formatting the instruction to the 5-tuple format, the immediate is sign-extended to W bits.

Putting this together, before the proof starts, the Prover and Verifier format the RISC-V assembly code into the required format. For the purposes of memory-checking, program code is a different body of memory from regular random-access memory and is maintained using separate read and write digests. It is read only and is initialized as follows: for the instruction at location PC, `(opflags[14], opcode, rs1, rs2, rd)` is stored at location $2 \cdot$ PC and the extended W-bit `imm` at location $2 \cdot$ PC $+ 1$.

4 Analyzing MLE-structure and Decomposability

This section illustrated the process of designing MLE-structured tables and decomposing them as per Definition 2.6 required by Lasso. We first establish notation and then design the tables for three important functions that are used as building blocks for the tables of many RISC-V instructions: equality, less than, and shifts.

4.1 Notation

Associating field elements with bit-vectors and vice versa. Let z be a field element in $\{0, 1, \ldots, 2^W - 1\} \subset \mathbb{F}$. We denote the binary representation of z as $\mathsf{bin}(z) = [z_{W-1}, \ldots, z_0] \in \{0,1\}^W$. Here, z_0 is the least significant bit (LSB), while z_{W-1} is the most significant bit (MSB). That is, $z = \sum_{i=0}^{W-1} 2^i z_i$.

We use $z_{<i}$ to refer to the subsequence $[z_{i-1}, \ldots, z_0]$. Analogously, $z_{>i}$ refers to the subsequence $[z_{W-1}, \ldots, z_{i+1}]$. Similarly, given a vector $z = [z_{W-1}, \ldots, z_0] \in \{0,1\}^W$, we denote the associated field element as $\mathsf{int}(z) = \sum_{i=0}^{W-1} 2^i \cdot z_i$.

Remark 1. *In the above paragraphs, we used an italicized z to denote both a field element in $\{0, \ldots, 2^W - 1\}$ and a vector in $\{0,1\}^W$. Throughout the paper, which of the two sets any variable z resides in will be clear from context.*

Concatenation of bit vectors. Given two bit vectors $x, y \in \{0,1\}^W$, we use $x \parallel y$ to refer to the number whose binary representation is the concatenation

$$[x_{W-1}, \ldots, x_0 \parallel y_{W-1}, , \ldots, y_0].$$

Under this definition, it holds that

$$\mathsf{int}(x \parallel y) = \mathsf{int}(x) \cdot 2^W + \mathsf{int}(y).$$

Decomposing bit vectors into chunks. For a constant c, and any $x \in \{0,1\}^L$, we divide the bits of input x naturally into chunks

$$x = [x_{W-1} \ldots x_0] = X_{c-1} \parallel \ldots \parallel X_2 \parallel X_0, \tag{4}$$

with each $X_i \in \{0,1\}^{W/c}$. Throughout, we assume c divides W for simplicity.

4.2 Three instructive functions and associated lookup tables

Let field $\mathbb{F}$ be a prime order field of size at least 2^W (for concreteness, let us fix W to be 64). Let x and y denote field elements that are guaranteed to be in the set $\{0, 1, \ldots, 2^W - 1\}$. We associate x and y with their binary decompositions in $\{0,1\}^W$, i.e., we let $(x_0, \ldots, x_{W-1}) \in \{0,1\}^W$ denote the vector such that $x = \sum_{i=0}^{W-1} 2^i x_i$. Abusing notation, we will use x to denote both the field element in $\{0, 1, \ldots, 2^W - 1\}$ and the vector $(x_0, \ldots, x_{W-1})$.

4.2.1 The Equality function

MLE-structured. The equality function eq takes as inputs two vectors $x, y \in \{0,1\}^W$ of identical length and outputs 1 if they are equal, and 0 otherwise. We will use a subscript to clarify the number of bits in each input to eq, e.g., eq_W denotes the equality function defined over domain $\{0,1\}^W \times \{0,1\}^W$. It is easily confirmed that its multilinear extension of eq is as follow:

$$\widetilde{\mathsf{eq}}_L(x, y) = \prod_{j=0}^{W-1} \left(x_j y_j + (1 - x_j)(1 - y_j) \right). \tag{5}$$

Indeed, the right hand side is clearly a multilinear polynomial in x and y, and if $x, y \in \{0,1\}^W$, the right hand side equals 1 if and only if $x = y$. Hence, the right hand side must equal the unique multilinear extension of the equality function. Clearly, the right hand side of Equation (5) can be evaluated at any point $(x, y) \in \mathbb{F}^W \times \mathbb{F}^W$ with $O(W)$ field operations.

Decomposability. To determine whether two W-bit inputs $x, y \in \{0,1\}^W$ are equal, one can decompose x and y into c chunks of length W/c, compute equality of each chunk, and multiply the results together.

Let $x = [X_1, \ldots, X_c]$ and $y = [Y_1, \ldots, Y_c]$ denote the decomposition of x and y into c chunks each, as per Equation (4). Let EQ_L denote the "big" table of size $N = 2^{2W}$ indexed by pairs (x, y) with $x, y \in \{0,1\}^W$, such that $t_{\mathsf{eq}}[x, y] = \mathsf{eq}(x, y)$. Let $\mathsf{EQ}_{W/c}$ denote the "small" table of size $N^{1/c}$ indexed by pairs (X, Y) of chunks $X, Y \in \{0,1\}^{W/c}$, such that $T_{\mathsf{eq}}[X,Y] = 1$ if $X = Y$ and $T_{\mathsf{eq}}[X,Y] = 0$ otherwise. The table below asserts that evaluating the equality function on x and y is equivalent to evaluating the equality function on each chunk $X_i \parallel Y_I$ and multiplying the results.

CHUNKS	**SUBTABLES**	**FULL TABLE**
$\mathsf{C}_i = X_i \parallel Y_i$	$\mathsf{EQ}_{W/c}[X,Y] = \widetilde{\mathsf{eq}}_{W/c}(X,Y)$	$\mathsf{EQ}_W[x,y] = \prod_{i=0}^{c-1} \mathsf{EQ}_{W/c}[X_i, Y_i]$

The (lone) subtable $\mathsf{EQ}_{W/c}$ is MLE-structured by Equation (5).

4.2.2 Less Than comparision

MLE-structured. The SLTU instruction takes as input two unsigned data types $x, y \in \{0, 1, \ldots, 2^{W-1}\}$, and outputs 1 if $x < y$ and 0 otherwise, where here the inequality interprets x and y as integers in the natural way. Note that the inequality computed here is strict. Consider the following $(2W)$-variate multilinear polynomial (standing for "less than unsigned"):

$$\widetilde{\mathsf{ltu}}_i(x,y) = (1 - x_i) \cdot y_i \cdot \widetilde{\mathsf{eq}}_{W-i}(x_{>i}, y_{>i}). \tag{6}$$

Clearly, this polynomial satisfies the following two properties:

(1) When $x \geq y$, $\widetilde{\mathsf{ltu}}_i(x,y) = 0$ for all i.

(2) Suppose $x < y$. Let k be the first index (starting from the MSB of x and y) such that $x_k = 0$ and $y_k = 1$. Then $\widetilde{\mathsf{ltu}}_k(x,y) = 1$ and $\widetilde{\mathsf{ltu}}_i(x,y) = 0$ for all $i \neq k$.

Based on the above properties, it is easy to check that

$$\widetilde{\mathsf{ltu}}(x,y) = \sum_{i=0}^{W-1} \widetilde{\mathsf{ltu}}_i(x,y). \tag{7}$$

Indeed, the right hand side is clearly multilinear, and by the two properties above, it equals $\widetilde{\mathsf{ltu}}(x,y)$ whenever $x, y \in \{0,1\}^W$. It is not difficult to see that the right hand side of Equation (7) can be evaluated at any point $(x,y) \in \mathbb{F}^W \times \mathbb{F}^W$ with $O(W)$ field operations as the set $\{\widetilde{\mathsf{eq}}_{W-i}(x_{>i}, y_{>i})\}_{i=0}^{W-1}$ can be computed in $O(W)$ total steps using the recurrence relation

$$\widetilde{\mathsf{eq}}_{W-i}(x_{>i}, y_{>i}) = \widetilde{\mathsf{eq}}_{W-i}(x_{>(i+1)}, y_{>(i+1)}) \cdot \widetilde{\mathsf{eq}}(x_i, y_i). \tag{8}$$

See [Tha22, Figure 3.3] for a depiction of this procedure.

Decomposing $\widetilde{\mathsf{ltu}}$. A similar reasoning to the derivation of Equation (7) reveals the following. As usual, break x and y into c chunks, $X_1 \parallel \cdots \parallel X_c$ and $Y_1 \parallel \cdots \parallel Y_c$. Let $\mathsf{LTU}_{W/c}(X_i, Y_i) = \widetilde{\mathsf{ltu}}_{W/c}(X_i, Y_i)$ denote the subtable with entry 1 if $X_i < Y_i$ when interpreted as unsigned (W/c)-bit data types, and 0 otherwise. Then

$$\mathsf{LTU}(x,y) = \sum_{i=0}^{c-1} \mathsf{LTU}_{W/c}(X_i, Y_i) \cdot \mathsf{EQ}_{W/c}(X_{>i}, Y_{>i}) = \sum_{i=0}^{c-1} \big(\mathsf{LTU}(X_i, Y_i) \cdot \prod_{j<i} \mathsf{EQ}_{W/c}(X_j, Y_j)\big).$$

Thus, evaluating $\mathsf{LTU}(x,y)$ can be done by evaluating $\mathsf{LTU}_{W/c}$ and $\mathsf{EQ}_{W/c}$ on each chunk (X_i, Y_i) ($\mathsf{EQ}_{W/c}$ need not be evaluated on the lowest-order chunk (X_c, Y_c)). This is summarized in the table below.

CHUNKS	**SUBTABLES**	**FULL TABLE**
$\mathsf{C}_i = X_i \parallel Y_i$	$\mathsf{LTU}_{W/c}[X_i, Y_i]$, $\mathsf{EQ}_{W/c}[X_i, Y_i]$	$\mathsf{LTU}_W[x,y] = \sum_{i=0}^{c-1} \mathsf{LTU}_{W/c}[X_i, Y_i] \cdot \prod_{j<i} \mathsf{EQ}_{W/c}[X_j, Y_j]$

The two subtables LTU and EQ are MLE-structured by Equations (5) and (7).

4.2.3 Shift Left Logical

MLE-structured. SLL takes an W-bit integer x and a $\log(W)$-bit integer y, and shifts the binary representation of x to the left by length y. Bits shifted beyond the MSB of x are ignored, and the vacated lower bits are filled with zeros.[19] For a constant k, let

$$\widetilde{\mathsf{SLL}}_k(x) = \sum_{j=k}^{W-1} 2^j \cdot x_{j-k}. \tag{9}$$

It is straightforward to check that the right hand side of Equation (9) is multilinear (in fact, linear) function in x, and that when evaluated at $x \in \{0,1\}^W$, it outputs the unsigned W-bit data type whose binary representation is given by $\mathsf{SLL}(x, k)$.

Now consider

$$\widetilde{\mathsf{SLL}}(x, y) = \sum_{k \in \{0,1\}^{\log W}} \widetilde{eq}(y, k) \cdot \widetilde{\mathsf{SLL}}_k(x). \tag{10}$$

It is straightforward to check that the right hand side of Equation (10) is multilinear in (x, y), and that, when evaluated at $x \in \{0,1\}^W \times \{0,1\}^{\log W}$, it outputs the unsigned W-bit data type $\mathsf{SLL}(x, y)$.

Decomposability. We split the value to be shifted, x, into c chunks, $X_1, \ldots, X_c$, each consisting of $W' = W/c$ bits. (y has only one chunk, Y_c, consisting of the lowest order $\log W$ bits). As explained below, we decompose a lookup into the evaluation table of SLL into a lookup into c different subtables, each of size $2^{W'+\log W}$. For $W = 64$, a reasonable setting of c would be 4, ensuring that $2^{W'+\log W} = 2^{22}$.

Conceptually, each chunk X_i of X needs to determine how many of its input bits goes "out of range" after the shift of length y. By out of range, we mean that shifting x left by y bits causes those bits to overflow the MSB of x and hence not contribute to the output of the instruction.

For chunks $i = 1, \ldots, c$ and shift length $k \in \{0,1\}^{\log W}$, define:

$$m_{i,k} = \max\{0, (\mathsf{int}(k) + W' \cdot i) - (W - 1)\}.$$

Here, $m_{i,k}$ equals the number of bits from the i'th chunk that go out of range. Let $m'_{i,k} = (W-1) - m_{i,k}$ denote the highest-order bit within the i'th chunk that does *not* go out of range. Then the evaluation table of SLL decomposes into smaller tables $\mathsf{SLL}_0, \ldots, \mathsf{SLL}_{c-1}$ as follows.

CHUNKS	**SUBTABLES**	**FULL TABLE MLE**
$\mathsf{C}_i = X_i \parallel Y_c$	For $i \in \{0, \ldots, c-1\}$, $\mathsf{SLL}_i[X_i \parallel Y_c] = \sum_{k \in \{0,1\}^{\log L}} \mathsf{eq}(Y_c, k) \cdot 2^{\mathsf{int}(k)} \cdot \left(\sum_{j=0}^{m'_{i,k}} 2^j \cdot X_{i,j}\right)$	$\mathsf{SLL}[x, y] = \sum_{i=0}^{W/c-1} 2^i \cdot \mathsf{SLL}_i[X_i, Y_*]$

Note that

$$\mathsf{SLL}_i[x, y] = \sum_{k \in \{0,1\}^{\log W}} \widetilde{\mathsf{eq}}(Y_c, k) \cdot 2^{W' \cdot (i-1) + \mathsf{int}(k)} \cdot \left(\sum_{j=0}^{m'_{i,k}} X_{i,j}\right), \tag{11}$$

and that SLL_i can be evaluated at any input $(x, y) \in \mathbb{F}^W \times \mathbb{F}^W$ in $O(W)$ field operations. Indeed, the set $\{\widetilde{\mathsf{eq}}(Y_c, k)\}_{k \in \{0,1\}^{\log W}}$ can be computed in $O(W)$ field operations via the recurrence in Equation (8). Similarly, the set $\{2^{W' \cdot (i-1) + \mathsf{int}(k)}\}_{i \in \{0, \ldots, c-1\}, k \in \{0,1\}^{\log W}}$ can be computed with $O(W)$ field operations. It follows that $\mathsf{SLL}_0(x, y), \ldots, \mathsf{SLL}_{c-1}(x, y)$ can be evaluated in $O(W)$ field operations in total.

[19]For $L = 32$-bit data types, the RISC-V manual says that the "shift amount is encoded in the lower $5 = \log(L)$ bits". This guarantees y is a

5 Analyzing the Evaluation Tables of the Base Instruction Set

We now consider each of the RISC-V instructions one at a time, and analyze the MLE-structure and the decomposability of their evaluation tables. Conceptually, Jolt's "one giant lookup table" is obtained by simply concatenating each the evaluation tables for all instructions (see Section 7). Address here that concatenation of MLE-structured tables is MLE-structured. More complicated for decomposable...

For most instruction tables, we present the entry as one "full" MLE and then describe how it can be decomposed into subtables, as required by Lasso.

5.1 Logical instructions

Each instruction performs the corresponding operation bitwise over the W-bits of x and y and stores the W-bit result in `rd`. The lookup tables here have a row for each possible $x \parallel y$ with the entry being the desired output to be stored in `rd`.

OP	**INDEX**	**FULL MLE**
AND	$x \parallel y \in \{0,1\}^W \times \{0,1\}^W$	$\sum_{i=0}^{W-1} 2^i \cdot (x_i \cdot y_i)$
OR	$x \parallel y \in \{0,1\}^W \times \{0,1\}^W$	$\sum_{i=0}^{W-1} 2^i \cdot (x_i + y_i - x_i \cdot y_i)$
XOR	$x \parallel y \in \{0,1\}^W \times \{0,1\}^W$	$\sum_{i=0}^{W-1} 2^i \cdot (x_i \cdot (1-y_i) + y_i \cdot (1-x_i))$

Decomposition. These MLEs can further be decomposed in the natural way, requiring only one subtable per instruction. For example, a bitwise AND on two W-bit inputs can be decomposed into c bitwise ANDs, each on two (W/c)-bit inputs. That is, if $D_0, \ldots, D_{c-1} \in \{0,1\}^{W/c}$ denote the results of these "smaller" bitwise AND operations, with $C_i = \mathsf{AND}_{\mathsf{W/c}}(\mathsf{X_i}, \mathsf{Y_i})$ then the result of the W-bit operation is simpler $\sum_{i=0}^{c-1} 2^{(W/c)\cdot i} D_i$. The decompositions for OR and XOR follow analagously.

5.1.1 Arithmetic instructions

Addition. For $x, y \in \{0,1\}^W$, $\mathsf{ADD}(x,y)$ returns the lowest W bits of (the binary representation of) the sum $\mathsf{int}(x) + \mathsf{int}(y)$. As discussed in Section 3.1, we need not specify whether the inputs to these instructions are signed versus unsigned. This is because in RISC-V signed data types are represented via two's complement, and when the inputs and outputs are viewed as strings in $\{0,1\}^W$, the input/output behavior of ADD and SUB is identical for signed data types as for unsigned ones.

As finite field addition costs just one constraint in R1CS, we cheaply compute $z = x + y$ in the circuit, where here, addition is performed over finite field $\mathbb{F}$. However, this sum can be $W+1$ bits long, i.e., z can be any field element in $\{0, \ldots, 2^{W+1} - 2\}$. The prescribed behavior for the RISC-V instruction ADD in this event is for the "overflow bit" to be ignored.

To this end, the lookup table for the ADD instructions contains an entry for all possible $(W+1)$-bit vectors $z \in \{0,1\}^{W+1}$, with the z's entry equal to the field element $\sum_{i=0}^{W-1} 2^i \cdot z_i$ equal to $\mathsf{int}(z_{<W})$. Note that this lookup table has size only 2^{W+1}, which is less than the tables of size 2^{2W} that we identify for most RISC-V instructions.

Subtraction. Due to RISC-V's use of two's complement representation of signed data types, subtraction can be performed using addition. Specifically, $\mathsf{SUB}(x,y)$ outputs the same W-bit string as

$$\mathsf{ADD}\left(x, \mathsf{bin}\left(2^W - \mathsf{int}(y)\right)\right).$$

In words, subtracting y from x is equivalent to adding the two's complement of y to x.[20]

OP	**INDEX**	**FULL MLE**
$\mathsf{ADD}(x, y)$	$z = \mathsf{bin}(x+y) \in \{0,1\}^{W+1}$	$\sum_{i=0}^{W-1} 2^i z_i$ `//return value represented by lowest W bits of` z
$\mathsf{SUB}(x, y)$	$z = \mathsf{bin}\left(x + \left(2^W - y\right)\right) \in \{0,1\}^{W+1}$	$\sum_{i=0}^{W-1} 2^i z_i$

Decomposition of the above lookup table is extremely simple and essentially equivalent to the case of range checks considered in our companion paper on Lasso [STW23].

AA: I would stop the paragraph here and cut the sentences below. The workings of the range proof and such can be put in some other section as an example.

5.1.2 Set Less Than

SLTU and SLT return 1 if $x < y$ and 0 otherwise, where x, y are unsigned and 2's complement signed W-bit numbers, respectively.

The table, full MLE and decomposed MLE for SLTU is equivalent to LTU derived in Section 4.2.2. The MLE for SLT additionally takes into consideration the sign bits of the two numbers and resorts to a comparison of the remaining bits only when the sign bits are the same.

OP	**INDEX**	**FULL TABLE MLE**
SLTU	$x \parallel y$	See Section 4.2.2
SLT	$x \parallel y$	$x_s \cdot (1 - y_s) + \widetilde{\mathsf{eq}}(x_s, y_s) \cdot \widetilde{\mathsf{ltu}}(x_{<s}, y_{<s})$

The decomposition of SLTU was discussed in Section 4.2.2 and requires two subtables of size $2^{W/c}$. The decomposition of SLT uses the same decomposition (applied to $x_{<s}, y_{<s}$, which has $W - 1$ rather than W bits), but additionally takes more two subtables to compute the $x_s(1 - y_s)$ and $\widetilde{\mathsf{eq}}(x_s, y_s)$ terms respectively. This brings the total to four types of subtable functions and $2c + 1$ total subtables. AA: In the commented out notes above, you said that we need to take x_s, y_s as advice? We don't need to do that cause the subtables can read into individual bits right?

5.1.3 Shifts

$\mathsf{SLL}(x, y)$ (Shift Left Logical), $\mathsf{SRL}(x, y)$ (Shift Right Logical) and $\mathsf{SRA}(x, y)$ (Shift Right Arithmetic) are the three shift operations. All shift operations take a W-bit input x, shift it by a length defined by the lowest $\log W$ bits of y and return W-bit values. Bits shifted beyond the MSB or LSB are ignored. In logical shifts, vacated bits are filled by zeros, and in arithmetic shifts, the vacated bits are filled by the sign bit of the original input x.

The MLE and decomposition of the evaluation table of SLL was presented in Section 4.2.3. We provide the analogous information for for SRL and SRA below, with the main difference being the direction and that, in SRA, the subtable handling the most significant bits also performs sign-extension.

[20] In a system implementing arithmetic on W-bit unsigned data types, the quantity 2^L cannot be represented. Hence, the two's complement of y needs to be computed in two steps, as $\left(2^W - 1 - y\right) + 1$, with the expression in parenthesis computed first, and then one added to the result. See `https://en.wikipedia.org/wiki/Two%27s_complement` for details. In Jolt, the quantity $2^L - \mathsf{int}(y)$ will be computed directly in the field $\mathbb{F}$, which we assume to have characteristic more than 2^L. Hence, the quantity 2^L can be represented, and the two's complement of y is (the binary representation of) the field element $2^L - y$.

Let $Y_0 = Y_{<\log W}$ and let X_{MSB} denote the MSB of a given chunk. For SRL, SRA, we define: (note that if $m' > 15$, then $X_{i,\geq m'} = 0$)

$$m'_{i,k} = \begin{cases} k - 16 \cdot i & \text{if } k > 16 \cdot (i-1) \\ 0 & \text{otherwise} \end{cases}$$

For these instructions, we provide the subtable decomposition, as done in Section 4.2. For SRL:

CHUNKS	**SUBTABLES**	**FINAL TABLE**
$\mathsf{C}_i = X_i \parallel Y_0$	For $i \in [0,3]$, $\mathsf{SRL}_i(\mathsf{C}_i) = \sum_{k=0}^{63} \widetilde{\mathsf{eq}}(y, \mathsf{bin}(k)) \cdot 2^{16 \cdot (i-1) - k} \cdot X_{i,\geq m'_{i,k}}$	$\mathsf{SRL}[x,y] = \sum_{i=0}^{3} T_i[X_i, Y_0]$

For the SRA instruction:

CHUNKS	**SUBTABLES**	**FINAL TABLE**
$\mathsf{C}_i = X_i \parallel Y_0$	For $i \in [0,2]$, $\mathsf{SRA}_i(\mathsf{C}_i) = \sum_{k=0}^{\log W - 1} \widetilde{\mathsf{eq}}(y,k) \cdot 2^{16 \cdot (i-1) - k} \cdot X_{i,\geq m'_{i,k}}$ $\mathsf{SRA}_4(C_4) = \sum_{k=0}^{\log W - 1} \widetilde{\mathsf{eq}}(y,k) \cdot 2^{16 \cdot (i-1) - \mathsf{int}(k)} \cdot X_{i,\geq m'_{i,k}} + \sum_{i=W-\mathsf{int}(k)}^{W-1} 2^i \cdot X_{\mathsf{MSB}}$	$\mathsf{SRA}[x,y] = \sum_{i=0}^{3} T_i[X_i, Y_0]$

I don't underestand X_{15} here AA: Yeah it should really be the MSB which is the sign bit.

5.1.4 Immediate Loads

AUIPC(x,y) takes the 20-bit immediate (operand y here), adds it to PC (operand x here) and stores the output in the destination register `rd`, but does *not* change the PC. LUI takes the 20-bit immediate (operand y, here) and loads it into the upper 20 bits of the destination register `rd`.

In both of these instructions, the 20-bit immediate is pre-processed (see Section ??) into an W-bit value with the 20 significant bits stored in the higher positions. With this pre-processing, AUIPC is treated identically to ADD with the two operands being PC and `imm`. As pre-processing does most of the work, the remaining task for LUI in the circuit is to store this pre-processed `imm` as provided into `rd`. The corresponding lookup table is simply the identity table taking an W-bit input and returning it as is. not sure I followed this. Seems like we don't even need a lookup here, just a copy (equality) constraint between the final value of `rd` and y? AA: True, but I just wrote it as the identity lookup for uniformity.

OP	**INDEX**	**FULL TABLE MLE**
AUIPC	$z = x + y$	$\sum_{i=0}^{W-1} 2^i z_i$ `// identical to ADD`
LUI	y	$\sum_{i=0}^{W-1} 2^i \cdot y_i$ `// identity`

Decomposition of these MLEs can be done in the natural way. AUIPC requires two subtable functions (as in ADD) and LUI requires only one subtable function. Note that these tables are smaller with only 2^{L+1} entries each.

5.1.5 Jumps

JAL sets $\texttt{PC} \leftarrow \texttt{PC} + \texttt{imm}$. JALR is similar but sets PC to be the same sum but with LSB set to 0. The memory address of the instruction following the jump (that is, (new PC) + 4) is stored in rd.

As discussed in Section ??, we pre-process imm to be the corresponding value in $\mathbb{F}_p$, allowing us to perform the calculation of the new PC as a single constraint. Addition in $\mathbb{F}_p$ avoids the overflow issue caused when performing 2's complement addition with a negative imm (which would require a lookup to correct, like in ADD). In dishonest executions, overflows may still occur with the PC being set to an illogical value. However, this will be caught in the next step as such a memory address will fail when reading the new PC.

For both jump instructions, we first calculate $z \leftarrow \texttt{PC} + \texttt{imm}$ in the circuit. For JAL, no further processing is needed and we update the PC with z and rd with $(z + 4)$. Thus, the lookup table for JAL is the identity table taking and returning $(L + 1)$-bit z as is. JALR is similar but the table takes $L + 1$-bit z and returns it with the LSB set to 0. Note that these instructions return $(L + 1)$-bit values to detect dishonest executions in the next step, as noted above.

OP	**INDEX**	**FULL TABLE MLE**
JAL	$z = x + y$	$\sum_{i=0}^{W-1} 2^i z_i$ `// identity`
JALR	$z = x + y$	$\sum_{i=1}^{W-1} 2^i z_i$ `// identity, but with LSB set to 0`

Decomposition of these MLEs can be done in the natural way, requiring only one subtable function. As with ADD and SUB, these tables have only 2^{L+1} entries.

work through details of how the extra bit (out of $L + 1$) is specified... AA: I didn't get this comment.

5.1.6 Branches

The B[COND] instructions set $\texttt{PC} \leftarrow \texttt{PC} + \texttt{imm}$ if $\mathsf{COND}(x, y) = \texttt{true}$. If false, they resort to the default change in PC.

As in the Jump instructions, imm is pre-processed to the corresponding field value when negative allowing the shifted PC to be obtained in the circuit as $z \leftarrow \texttt{PC} + \texttt{imm}$. Now, the lookup is used to perform the comparisons to decide whether to use the shifted value or not. The MLE for doing signed and unsigned strict "less than" comparisons were discussed in Section 4.2.2 and used in SLT/SLTU. We use the same MLEs here, along with the $\widetilde{\mathsf{eq}}$ MLE.

OP	**INDEX**	**FULL TABLE MLE**
BEQ	$x \parallel y$	$1 - \widetilde{\mathsf{eq}}(x, y)$
BNE	$x \parallel y$	$\widetilde{\mathsf{eq}}(x, y)$
BLTU	$x \parallel y$	$\widetilde{\mathsf{LTU}}(x, y)$ `// from SLTU`
BLT	$x \parallel y$	$\widetilde{\mathsf{LTS}}(x, y)$ `// from SLT` check what we're calling this
BGEU	$x \parallel y$	$1 - \widetilde{\mathsf{LTU}}(x, y)$
BGE	$x \parallel y$	$1 - \widetilde{\mathsf{LTS}}(x, y)$ check what we're calling this

Decomposition of these MLEs follows that of the constituent MLE $\widetilde{\mathsf{eq}}$ or $\widetilde{\mathsf{LTU}}, \widetilde{\mathsf{LTS}}$ MLEs.

5.1.7 Memory Loads and Stores

RISC-V uses a byte-addressable memory system that is access using only the following variants of the load and store instructions:

- LD reads a 64-bit value from memory and stores it into `rd`.
- L[W/H/B] are similar, but they read only the lowest 32/16/8 bits of the value in memory and store it sign-extended to W bits into `rd`.
- L[W/H/B]U are identical to their signed counterparts but do not perform any sign-extension.
- SD takes a 64-bit operand, y, and stores it into a specified memory location.
- S[W/H/B] store only the lowest 32/16/8 bits of the operand (without any sign-extension).

The lookup tables for all these operations are nearly identical. As discussed in Section 3.3, these tables have an index for all possible W-bit z and the entry is the required zero/sign-extension of the desired length suffix.

OP	INDEX	FULL TABLE MLE
LD	z	$\sum_{i=0}^{L} z_i$ `identity`
L[W/H/B]	z	$\sum_{i=k}^{W} 2^i \cdot z_{k-1} + \sum_{i=0}^{k-1} 2^i \cdot z_i$ `where` k `= 32/16/8`
L[W/H/B]U	z	$\sum_{i=0}^{k-1} 2^i \cdot z_i$ `where` k `= 32/16/8`
SD	z	$\sum_{i=0}^{L} 2^i \cdot z_i$ `identity`
S[W/H/B]	z	$\sum_{i=0}^{k} 2^i \cdot z_i$ `where` k `= 32/16/8`

All of these MLEs can be decomposed in the natural way with just one subtable function.

6 The Multiplication Extension

The M extension adds multiplication, division and remainder operations. These instructions are generally more complex than the base integer instruction set and involve new techniques in Jolt to handle - namely the addition of "virtual" registers and instructions.

6.1 Virtual Instrutions and Virtual Registers

Jolt splits certain complex assembly instructions (such as MULH) into a sequence of instructions that are executed in the ZKVM in place of the original instruction. The CPU state transition guarantee that should hold for the original assembly instruction now holds after the entire sequence is executed. Note that the splitting of instructions is done in the assembly code during pre-processing and is independent of the input or execution.

To avoid jumbling with the base registers, Jolt introduces new "virtual" registers that virtual instructions use to store intermediate values. These registers have addresses outside the standard set of base registers but are otherwise read from and stored to identically. To ensure safety, the only time the "real" CPU state is changed is when the last virtual instruction of the sequence stores the final result in the "real" destination register of the original instruction.

6.1.1 ASSERT Instructions

Asserts are a type of virtual instruction that add circuit constraints on an instruction's result. For example, an ASSERT-[COND] constraint uses the lookup table for the branch instruction B[COND] but additionally adds a constraint that the lookup must return 1. Assert instructions do not have a destination register. On top of the conditional checks seen in the Set-Less-Than and Branch instructions, Jolt supports the following assert instructions:

ASSERT-LT-ABS takes two W-bit 2's complement signed inputs and outputs $|x| < |y|$.

ASSERT-EQ-SIGNS takes two W-bit 2's complement signed inputs and outputs $x_s == y_s$.

OP	INDEX	FULL MLE
ASSERT-LT-ABS	$x \parallel y$	$\mathsf{LTU}(x_{<(W-1)}, y_{<(W-1)})$ `// ignore sign bits`
ASSERT-EQ-SIGNS	$x \parallel y$	$\widetilde{\mathsf{eq}}(x_s, y_s)$

6.1.2 ADVICE and MOVE Instructions

ADVICE v: stores a special W-bit non-determistic circuit input into virtual register v.

MOVE v_1, v_2: copies the value in register v_1 into register v_2 (either could be virtual).

The advice instruction allows the prover to store non-deterministic advice into virtual registers. The lookup query's function here is to act as a range check on the advice and thus, uses the range check table. The "non-deterministic" part of these instructions is that their lookup's query isn't derived in the circuit (such as through registers, memory or `imm`) but comes from advice passed into the CPU step circuit. Thus, unlike the immediate `imm`, these values aren't fixed in the assembly code and can be set by the prover at proving time. ADVICE has no source register or immediate and only specifies a destination register.

The MLEs of these instructions are identical to the that of the identity MLE (and thus, **decomposition** can be performed accordingly).

range checks need unindexed lookups!

OP	INDEX	MLE
ADVICE	x	$\sum_{i=0}^{W-1} 2^i \cdot x_i$ `// range check`
MOVE	x	$\sum_{i=0}^{W-1} 2^i \cdot x_i$ `// range check`

6.2 The M-Extension Tables

As before, for some new instructions, we give the full MLE associated with each instruction and describe how they can be decomposed. For other new instructions, we provide the sequence of previously-defined instructions that result in the same CPU state change.

6.2.1 Unsigned or Lower Multiplication

The following instructions take two W-bit operands x and y.

MUL returns the lower W bits of $x \times y$ where the operands are treated as signed 2's complement numbers.

MULU returns the lower W bits of $x \times y$ where the operands are treated as unsigned W-bit numbers.

MULHU returns the higher W bits of $x \times y$ where the operands are treated as unsigned W-bit numbers.

Similar to ADD, Jolt performs the core multiplication operation in the circuit efficiently as computing $z = x \times y$ costs just one constraint. The circuit then queries z in the lookup tables of the instructions, which have a row for every possible $2W$-bit z with the entry being the desired bits. Note that while MUL is a signed operation, performing unsigned multiplication returns the same lower bits.

OP	INDEX	FULL TABLE MLE
MUL	$z = x \times y$	$\sum_{i=0}^{W-1} 2^i \cdot z_i$ `// lower L bits`
MULU	$z = x \times y$	$\sum_{i=0}^{W-1} 2^i \cdot z_i$ `// lower L bits`
MULHU	$z = x \times y$	$\sum_{i=L}^{2W-1} 2^i \cdot z_i$ `// higher L bits`

Decomposition of these MLEs can be done in the natural way, similar to the decomposition of the identity table.

6.2.2 Signed and Higher MUL

MULH returns the higher W bits of $x \times y$ where the operands are treated as signed 2's complement numbers.

MULHSU returns the higher W bits of $x \times y$ where only x signed but y is unsigned.

These instructions are more complicated than the others as they require signed multiplication where signed operands are sign-extended to $2W$ bits before performing the multiplication. This leads to the result having $4W$ and $3W$ total bits in MULH and MULHSU, respectively. As this is too large to handle with a lookup query, we instead compute the desired bits in stages.

For a number x, let s_x be $\sum_{i=0}^{W} 2^i x_s$ such that $[s_x \parallel x]$ is the sign-extension of x to $2W$ bits. The signed multiplication algorithm performs the following $2W \times 2W$-bit multiplication and returns the highest $2W$ bits: $[s_x \parallel x] \times [s_y \parallel y]$. As the instructions above are only interested in the higher W bits of this result, we can represent the required bits as the *lower* W bits as the sum of the following three values each computed using only unsigned multiplication:

$$[\text{higher } W \text{ bits of } x \times y] + [\text{lower } W \text{ bits of } s_x \times y] + [\text{lower } W \text{ bits of } s_y \times x]$$

Given s_x, s_y, the above terms can be obtained using MULH, MULU instructions and the sum computed using ADD. To get s_x, s_y, we define a new instruction, MOVSIGN, which takes an W-bit input x and stores the W-bit number with x_s as all of its binary coefficients in the destination register. (The MLE below can be **decomposed** naturally using one subtable function.)

OP	INPUT	FULL MLE
MOVSIGN	x	$\sum_{i=0}^{W-1} 2^i \cdot x_s$ `// place sign bit in all positions`

We can now split MULH, MULHSU into virtual instructions following the above procedure. We use $\mathrm{r}_x, \mathrm{r}_y$ to denote the two operand registers. We use "v" to name virtual registers. (In actual re-formatted assembly code, these are replaced by a free numbered virtual register.)

Original	**Virtual Sequence** (`OPCODE, rs1, rs2, imm, rd`)
MULH $\mathrm{r}_x, \mathrm{r}_y$, `rd`	1. MOVSIGN $\mathrm{r}_x, -, -, \mathrm{v}_{s_x}$ `// store` s_x `in a virtual register`
	2. MOVSIGN $\mathrm{r}_y, -, -, \mathrm{v}_{s_y}$ `// store` s_y
	3. MULHU $\mathrm{r}_x, \mathrm{r}_y, -, \mathrm{v}_0$ `// get higher bits of` $x \times y$
	4. MULU $\mathrm{v}_{s_x}, \mathrm{r}_y, -, \mathrm{v}_1$ `// get lower bits of` $s_x \times y$
	5. MULU $\mathrm{v}_{s_y}, \mathrm{r}_x, -, \mathrm{v}_2$ `// get lower bits of` $s_y \times x$
	6. ADD $\mathrm{v}_0, \mathrm{v}_1, -, \mathrm{v}_3$
	7. ADD `rd`$, \mathrm{v}_2, -,$ `rd`
MULH r_x, r_y, `rd`	1. MOVSIGN $\mathrm{r}_x, -, -, \mathrm{v}_{s_x}$
	2. MULHU $\mathrm{r}_x, \mathrm{r}_y, -, \mathrm{v}_1$
	3. MULU $\mathrm{v}_{s_x}, \mathrm{v}_y, -, \mathrm{v}_2$
	4. ADD $\mathrm{v}_1, \mathrm{v}_2, -,$ `rd`

The correctness of the output can be seen by inspection as the steps follow the natural binary multiplication algorithm. It can also be seen that the "real" CPU state is only modifed in the final steps of each sequence, when the result is stored into `rd`.

6.3 Division and Remainder

In RISC-V, division and remainder operations take two W-bit values read from registers. For both operations, the prover provides as non-deterministic advice the quotient q and remainder r using the advice instructions introductions in Section 6.1.2. The correctness of this advice is verified using a sequence of virtual instructions. As both DIV and REM instructions perform the same checks, they have nearly identical virtual instructions with only the last instruction differing based on the desired value (q or r).

Unsigned versions. In unsigned division, both operands x, y and quotient q and remainder r are all treated as unsigned W-bit numbers. DIVU/REMU require x to be equal to $q \times y + r$ such that $r < y$ and $q \times y \leq x$.

Original	**Virtual Sequence** (`OPCODE, rs1, rs2, imm, rd`)
DIVU $\mathrm{r}_x, \mathrm{r}_y$, `rd`	1. ADVICE $-, -, -, \mathrm{v}_q$ `// store non-deterministic advice` q `into` v_q
	2. ADVICE $-, -, -, \mathrm{v}_r$ `// store non-deterministic advice` r `into` v_r
	3. MULU $\mathrm{v}_q, \mathrm{r}_y, -, \mathrm{v}_{qy}$ `// compute` $q \times y$
	4. ASSERT_LTU $\mathrm{v}_r, \mathrm{r}_y, -, -$ `// verify that` $r < y$
	5. ASSERT_LTE $\mathrm{v}_{qy}, \mathrm{r}_x, -, -$ `// assert` $q \times y \leq x$
	6. ADD $\mathrm{v}_{qy}, \mathrm{v}_r, -, \mathrm{v}_0$ `// compute` $q \times y + r$
	7. ASSERT_EQ $\mathrm{v}_0, \mathrm{r}_x, -, -$
	8. MOVE $v_q, -, -,$ `rd` `// store` q `in rd`
REMU $\mathrm{r}_x, \mathrm{r}_y$, `rd`	1-7. same as above
	8.MOVE $v_r, -, -,$ `rd` `// store` r `in rd`

Signed versions. In signed division, both operands x, y and quotient q and remainder r are all treated as signed 2's complement L-bit numbers.

DIVU/REMU requires $x = q \times y + r$ such that $|r| < |y|$ and r, y have the same sign.

Original	Virtual Sequence
DIV $\mathrm{r}_x, \mathrm{r}_y$, `rd`	1. ADVICE $-,-,-,\mathrm{v}_q$ `// store non-deterministic advice` q `into` v_q 2. ADVICE $-,-,-,\mathrm{v}_r$ `// store non-deterministic advice` r `into` v_r 3. ASSERT_LT_ABS $\mathrm{v}_r, \mathrm{r}_y, -, -$ `// verify that` $\lvert r\rvert < \lvert y\rvert$ 4. ASSERT_EQ_SIGNS $\mathrm{v}_r, \mathrm{r}_y, -, -$ `// require` r `to have the sign of` y 5. MUL $\mathrm{v}_q, \mathrm{r}_y, -, \mathrm{v}_{qy}$ `// compute` $q \times y$ 6. ADD $\mathrm{v}_{qy}, \mathrm{v}_r, -, v_0$ `// compute` $q \times y + r$ 7. ASSERT_EQ $v_0, x, -, -$ 8. MOVE $v_q, -, -,$ `rd` `// store` q `in rd`
REM $\mathrm{r}_x, \mathrm{r}_y$, `rd`	1-7. same as above 8. MOVE $v_r, -, -,$ `rd` `// store` r `in rd`

As with the splitting of the multiplication instructions, the correctness of the output can be seen by inspection as the steps follow the straightforward verification of division. Also, the real CPU state is only modified in the final steps.

7 Putting It all Together: a SNARK for RISC-V Emulation

A uniform R1CS system whose witness vector's entries (well, a part of the vector) are assumed to all be in the giant lookup table. Separately, Lasso used to confirm all entries of the witness vector are indeed in the lookup table.

Point to Zokrates code.

7.1 Cost Estimation

give top-level commitment costs for RV64IM and RV32IM and point to Appendix A for details

Acknowledgements. Justin Thaler was supported in part by NSF CAREER award CCF-1845125 and by DARPA under Agreement No. HR00112020022. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the United States Government or DARPA.

References

[AGL+22] Jeremy Avigad, Lior Goldberg, David Levit, Yoav Seginer, and Alon Titelman. A verified algebraic representation of cairo program execution. In *Proceedings of the 11th ACM SIGPLAN International Conference on Certified Programs and Proofs*, pages 153–165, 2022.

[And17] Andrew Waterman1, Krste Asanovic. The RISC-V instruction set manual. `https://riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.pdf`, 2017.

[BBB+18] Benedikt Bünz, Jonathan Bootle, Dan Boneh, Andrew Poelstra, Pieter Wuille, and Greg Maxwell. Bulletproofs: Short proofs for confidential transactions and more. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2018.

[BCC+16] Jonathan Bootle, Andrea Cerulli, Pyrros Chaidos, Jens Groth, and Christophe Petit. Efficient zero-knowledge arguments for arithmetic circuits in the discrete log setting. In *Proceedings of the International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, 2016.

[BCG$^+$18] Jonathan Bootle, Andrea Cerulli, Jens Groth, Sune Jakobsen, and Mary Maller. Arya: Nearly linear-time zero-knowledge proofs for correct program execution. In *Proceedings of the International Conference on the Theory and Application of Cryptology and Information Security (ASIACRYPT)*, 2018.

[BCHO22] Jonathan Bootle, Alessandro Chiesa, Yuncong Hu, and Michele Orru. Gemini: Elastic snarks for diverse environments. In *Proceedings of the International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, 2022.

[BEG$^+$91] Manuel Blum, Will Evans, Peter Gemmell, Sampath Kannan, and Moni Naor. Checking the correctness of memories. In *Proceedings of the IEEE Symposium on Foundations of Computer Science (FOCS)*, 1991.

[BFR$^+$13] Benjamin Braun, Ariel J. Feldman, Zuocheng Ren, Srinath Setty, Andrew J. Blumberg, and Michael Walfish. Verifying computations with state. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2013.

[BFS20] Benedikt Bünz, Ben Fisch, and Alan Szepieniec. Transparent SNARKs from DARK compilers. In *Proceedings of the International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, 2020.

[BSBHR19] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. Scalable zero knowledge with no trusted setup. In *Advances in Cryptology–CRYPTO 2019: 39th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 18–22, 2019, Proceedings, Part III 39*, pages 701–732. Springer, 2019.

[BSCGT13] Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin, and Eran Tromer. Fast reductions from rams to delegatable succinct constraint satisfaction problems. In *Proceedings of the 4th conference on Innovations in Theoretical Computer Science*, pages 401–414, 2013.

[BSCTV14] Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. Scalable zero knowledge via cycles of elliptic curves. In *Proceedings of the International Cryptology Conference (CRYPTO)*, 2014.

[CBBZ23] Binyi Chen, Benedikt Bünz, Dan Boneh, and Zhenfei Zhang. HyperPlonk: Plonk with linear-time prover and high-degree custom gates. In *Proceedings of the International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, 2023.

[CHM$^+$20] Alessandro Chiesa, Yuncong Hu, Mary Maller, Pratyush Mishra, Noah Vesely, and Nicholas Ward. Marlin: Preprocessing zkSNARKs with universal and updatable SRS. In *Proceedings of the International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, 2020.

[FS86] Amos Fiat and Adi Shamir. How to prove yourself: Practical solutions to identification and signature problems. In *Proceedings of the International Cryptology Conference (CRYPTO)*, pages 186–194, 1986.

[GKR08] Shafi Goldwasser, Yael Tauman Kalai, and Guy N. Rothblum. Delegating computation: Interactive proofs for muggles. In *Proceedings of the ACM Symposium on Theory of Computing (STOC)*, 2008.

[GPR21] Lior Goldberg, Shahar Papini, and Michael Riabzev. Cairo–a turing-complete stark-friendly cpu architecture. *Cryptology ePrint Archive*, 2021.

[Gro16] Jens Groth. On the size of pairing-based non-interactive arguments. In *Proceedings of the International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, 2016.

[GWC19] Ariel Gabizon, Zachary J. Williamson, and Oana Ciobotaru. PLONK: Permutations over Lagrange-bases for oecumenical noninteractive arguments of knowledge. ePrint Report 2019/953, 2019.

[KST22] Abhiram Kothapalli, Srinath Setty, and Ioanna Tzialla. Nova: Recursive Zero-Knowledge Arguments from Folding Schemes. In *Proceedings of the International Cryptology Conference (CRYPTO)*, 2022.

[KT23] Tohru Kohrita and Patrick Towa. Zeromorph: Zero-knowledge multilinear-evaluation proofs from homomorphic univariate commitments. *Cryptology ePrint Archive*, 2023.

[KZG10] Aniket Kate, Gregory M. Zaverucha, and Ian Goldberg. Constant-size commitments to polynomials and their applications. In *Proceedings of the International Conference on the Theory and Application of Cryptology and Information Security (ASIACRYPT)*, pages 177–194, 2010.

[Lee21] Jonathan Lee. Dory: Efficient, transparent arguments for generalised inner products and polynomial commitments. In *Theory of Cryptography Conference*, pages 1–34. Springer, 2021.

[Lip89] Richard J Lipton. *Fingerprinting sets.* Princeton University, Department of Computer Science, 1989.

[PST13] Charalampos Papamanthou, Elaine Shi, and Roberto Tamassia. Signatures of correct computation. In *Theory of Cryptography Conference*, pages 222–242. Springer, 2013.

[SAGL18] Srinath Setty, Sebastian Angel, Trinabh Gupta, and Jonathan Lee. Proving the correct execution of concurrent services in zero-knowledge. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, October 2018.

[Set20] Srinath Setty. Spartan: Efficient and general-purpose zkSNARKs without trusted setup. In *Proceedings of the International Cryptology Conference (CRYPTO)*, 2020.

[SL20] Srinath Setty and Jonathan Lee. Quarks: Quadruple-efficient transparent zkSNARKs. Cryptology ePrint Archive, Report 2020/1275, 2020.

[STW23] Srinath Setty, Justin Thaler, and Riad S. Wahby. Lasso: Unlocking the lookup singularity, 2023.

[Tha13] Justin Thaler. Time-optimal interactive proofs for circuit evaluation. In *Proceedings of the International Cryptology Conference (CRYPTO)*, 2013.

[Tha22] Justin Thaler. Proofs, arguments, and zero-knowledge. *Foundations and Trends in Privacy and Security*, 4(2–4):117–660, 2022.

[Tom23] Tomer Solberg. RISC Zero prover protocol & analysis. `https://github.com/ingonyama-zk/papers/blob/main/risc0_protocol_analysis.pdf`, 2023.

[Whi] Barry Whitehat. Lookup singularity. `https://zkresear.ch/t/lookup-singularity/65/7`.

[WSR$^+$15] Riad S. Wahby, Srinath Setty, Zuocheng (Andy) Ren, Andrew J. Blumberg, and Michael Walfish. Efficient RAM and control flow in verifiable outsourced computation. In *NDSS*, 2015.

[WTS$^+$18] Riad S. Wahby, Ioanna Tzialla, Abhi Shelat, Justin Thaler, and Michael Walfish. Doubly-efficient zkSNARKs without trusted setup. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2018.

[ZGK$^+$18] Yupeng Zhang, Daniel Genkin, Jonathan Katz, Dimitrios Papadopoulos, and Charalampos Papamanthou. vRAM: Faster verifiable RAM with program-independent preprocessing. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2018.

[ZXZS20] Jiaheng Zhang, Tiancheng Xie, Yupeng Zhang, and Dawn Song. Transparent polynomial delegation and its applications to zero knowledge proof. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2020.

Additional Material

A Quick Reference for Significant bits and elements

ELEMENT	**PURPOSE**	**#SIG. BITS**
`opflags[14]`	These are 1-bit elements used to guide the constraint system on if-else branches. (List given below.)	1 bit each
`memflags[W/8]`	These indicate the number of memory loads and stores performed.	1 bit each
`opcode[8]`	These bits constitute the 8-bit `opcode` for the instruction.	1 bit each
`rs1, rs2, rd`	The step instruction's source and desintation registers.	5 bits each
`memory_v_bytes[W/8]`	The bytes-decomposition of the value lookup's result in a load/store operation.	8 bits each
`read_ts_code`	The timestamp passed as advice for memory-checking when reading the program code.	< log(`#steps`) bits
`read_ts_rs1`, `read_ts_rs2`	The timestamps passed as advice for memory-checking when reading the values at registers `rs1, rs2`.	< log(`#steps`) bits
`memory_ts[W/8]`	The timestamps passed as advice for memory-checking when reading the bytes in memory for loads.	< log(`#steps`) bits
`imm`	The step instruction's immediate, appropriate sign or zero extended.	W bits
Values read at `rs1, rs2, code`	The actual value read at the locations in `rs1, rs2, code`.	W bits each
Lookup `result`	The lookup entry passed in an advice.	W bits

A.1 List of Operation Flags employed:

1. This flag is 0 if the first operand is `rs1` and 1 if it is `rs2`.
2. This flag is 0 if the second operand is `rs2` and 1 if it is `imm`.
3. Is this a load instruction?
4. Is this a store instruction?
5. Is this a jump instruction?
6. Is this a branch instruction?
7. Does this instruction update `rd`?
8. Does this instruction involve adding the operands?
9. Does this instruction involve subtracting the operands?
10. Does this instruction involve multiplying the operands?
11. Does this instruction involve non-deterministic advice?
12. Does this instruction assert the lookup output to be false?
13. Does this instruction assert the lookup output to be true?
14. This flag is the sign bit of the immediate.

The memory flags are as follows: if the instruction is a load/store operation that reads/writes k bytes, then the memory flags will be of the form $1^k \parallel 0^{W/8-k}$.

A.2 Summary of CPU Step Constraints

Here, we go through the CPU steps outlined in Figure 1 and add more context in terms of the committed elements used and constriants involved.

1. Read the program code to get the instructions details: `opflags[14], opcode, rs1, rs2, imm`.
   - As discussed in Section 3.4, this involves two reads from program memory: one for the section involving `opflags[14], opcode, rs1, rs2`, and the second for just `imm`. The reads involve the same timestamp `read_ts_code` and locations $(2 \cdot \texttt{PC})$ and $(2 \cdot \texttt{PC} + 1)$, respectively.
   - The value read for the first one is obtained by concatenating `opflags[14]` $\parallel$ `opcode` $\parallel$ `rs1` $\parallel$ `rs2`. We are guaranteed that all values read in this step are in the right range as:
     - For each element e in `opflag[14]` and `opcode[8]`, we add the constraint $e \cdot (1 - e) = 0$.
     - `rs1, rs2` are used as locations in future memory-checking reads. Thus, if they were out of range, those memory checks will fails.
     - The value `imm` read from the second read is guaranteed to be in range as it is fixed in the formatted program code.
2. Read the source registers `rs1`, `rs2`. And then set operands x and y
   - Reading the source registers involves two memory-checking updates. The locations are the registers themselves.
   - The values and timestamps involved are committed advice elements: (`read_val_rs1`, `read_ts_rs1`) and (`read_val_rs2`, `read_ts_rs2`), respectively.
   - The setting of operands x and y involves using `opflags[0], opflags[1]` described earlier.
3. Perform loads and stores using memory-checking.
   - For both loads and stores, the memory location involved is `rs1` + `imm`. This sum is calculated using constraints and also involves `opflag`[13] which holds the sign of `imm`.
   - As discussed in Section 3.3, the lookup argument involves $W/8$ memory operations as RISC-V uses a byte-addressable memory. The required bytes-decomposition are the elements in `memory_v_bytes[W/8]`. The lookup tables are described in Section 5.1.7.
   - The range checks for stores involve elements ???
4. Construct the lookup element.
   - This is structured as `opcode` $\parallel$ `z` where z could be $x \parallel y$, $x + y$, $x - y$ or $x * y$.
   - The exact format chosen is guided by `opflags[8-11]`
5. Updating the destination register.
   - If the corresponding `opflag` if 1, the lookup result is stored in `rd`.
   - The memory-checking write operation here involves the location `rd`, value `result` and timestamp being the current global step count.
6. Updating the `PC`.
   - For Jump instructions, `PC` $\leftarrow$ (lookup `result`). For Branch instructions, `PC` $\leftarrow$ `PC` + `imm` (sum computed similar to Step 3 above) if and only if the lookup `result` was true. For other instructions, `PC` $\leftarrow$ `PC` + 4. The right choice is guided by the corresponding `opcodes`.

B Obtaining an indexed lookup argument from an unindexed one

Let $T \in \mathbb{F}^N$ be a lookup table, and let R be such that all table elements are in $\{0, 1, \ldots, R-1\}$, and assume that $m \cdot N$ is less than the field characteristic. Replace each table element $T[i]$ with $i \cdot R + T[i]$ to obtain a modified table $T' \in \mathbb{F}^N$, and replace each lookup pair (b_j, a_j) with the field element $a'_j = b_j \cdot R + a_j$. Apply a range check to confirm that $a_j \in \{0, 1, \ldots, R-1\}$ for all lookups. Then $T[b_j] = a_j$ implies that $a'_j = T'[b_j]$. Conversely, under the guarantee that each a_j is in $\{0, \ldots, R-1\}$, if $T[b_j] \neq a_j$ then no entry of T' is equal to a'_j. Hence, one can apply a lookup argument for unindexed lookups (Definition 2.4) to a' and T' to confirm that $T[b_j] = a_j$ for all pairs (b_j, a_j) in the list of lookups.

If R is chosen to be the smallest power of 2 bounding all the table elements, then the range check can be implemented via a lookup into the (MLE-structured and c-decomposable) table $\{0, 1, \ldots, R-1\}$, and moreover T' is decomposable or MLE-structured if and only if t is decomposable or MLE-structured.

The range check on a_j can be omitted if the value a_j is provided by a party that is guaranteed to be honest. This is the case in Jolt, where the indices (i.e., entries of the vector $b \in \mathbb{F}^m$) are in fact inputs to a computer instruction, and those inputs are computed "by the circuit" output by the front-end. This circuit is "trusted", i.e., correctness of the front-end *means* that the circuit applies the appropriate instruction to the correct input(s) at each step of the computation.

C Overview of Memory-Checking Arguments

C.1 Merkle trees

One way to implement a memory-checking argument is via Merkle trees. This approach was first introduced to the literature on SNARKs in Pantry [BFR+13] and is still used today in so-called *incrementally verifiable computing* (IVC) schemes (see [BSCTV14] for an early example). Conceptually, the VM whose execution is being proved is modified as follows, so as to authenticate its own memory. The VM at all times tracks the root hash of a Merkle tree that has the contents of each of its memory cells at the leaves (let us say that the number of memory cells is N). Every read operation returns not only the value stored in the appropriate memory cell, but also a Merkle authentication path that the machine checks for consistency with the stored root hash. Write operations are preceded by a read operation, which allows the VM to update the root hash in an authenticated manner.

The downside of the Merkle tree approach is that each read and write requires proving knowledge of a Merkle authentication path, which involves $\log N$ many hash evaluations. This can be an effective approach if there are not many reads and writes, but is very expensive for the prover for executions involving many reads and writes.

C.2 Re-ordering the execution trace

The rough idea of this approach to designing a memory-checking argument is to have the prover "re-order" the execution trace so that, rather than entries appearing in increasing order of timestep, they instead are grouped by the memory location read or written at that step, and within each group, entries are sorted by time. This is called a "memory-ordered" execution trace (see [Tha22, Section] for an exposition). It is straightforward to design a quasilinear-size circuit that takes as input a memory-ordered execution trace, and confirms that the value returned by every read operation is indeed the value last written to the appropriate memory cell.

The core of the memory-checking argument amounts to confirming that the memory-ordered execution trace is indeed a re-ordering (i.e., permutation) of the time-ordered execution trace. Some early works [BSCGT13] proposed to accomplish this using so-called *routing networks*, a complicated and expensive technique from the PCP literature. Later works [BCG+18, ZGK+18] proposed to instead use lightweight *permutation-invariant fingerprinting* techniques dating to the memory-checking literature from the late 1980s and early 1990s [Lip89, BEG+91]. The key idea in these techniques is to check whether two vectors $a, b \in \mathbb{F}^N$ are permutations of each other (i.e., whether there exists a permutation $\pi\colon \{1, \ldots N\} \to \{1, \ldots, N\}$ such that $a_i = b_{\pi(i)}$ for all

1. Input: vectors $a, b \in \mathbb{F}^N$.
2. Goal: determine whether a and b are permutations of each other.
3. Pick a random $r \in \mathbb{F}^N$, and checks that

$$\prod_{i=1}^{N}(r - a_i) = \prod_{i=1}^{N}(r - b_i).$$

Figure 3: Permutation-invariant fingerprinting.

1. Input: vectors $a, b \in \mathbb{F}^N$.
2. Goal: determine whether $a = b$.
3. Pick a random $r \in \mathbb{F}^N$, and checks that

$$\sum_{i=1}^{N} a_i r^{i-1} = \prod_{i=1}^{N} b_i r^{i-1}.$$

Figure 4: Reed-Solomon fingerprinting. If $a = b$ then the check passes with probability 1. If $a \neq b$, then the check passes with probability at most $(N-1)/|\mathbb{F}$.

i) by having the verifier pick a random $r \in \mathbb{F}$, and checking that

$$\prod_{i=1}^{N}(r - a_i) = \prod_{i=1}^{N}(r - b_i).$$

If a and b are permutations, this check will pass with probability 1, while if they are not permutations, the check will pass with probability only $m/|\mathbb{F}|$. This technique is now pervasive in SNARK design, appearing in works such as Plonk [GWC19]. We refer to this as *permutation-invariant fingerprinting.*

We have described the checking procedure as interactive, since the verifier picks the random field element r, but the procedure can be rendered non-interactive via the Fiat-Shamir transformation. In any SNARK that uses permutation-checking, the vectors a and b will be committed by the prover, and r will be chosen by hashing those commitments (and any other messages sent by the prover earlier in the interactive protocol).

C.3 Memory-checking via permutation-checking, without re-ordering

Spartan [Set20], Spice [SAGL18], and descendants including our companion paper Lasso [STW23] build on the memory-checking work of Blum et al. [BEG$^+$91] to obtain memory-checking arguments that do not reorder the execution trace.

Overview of the Lasso lookup argument. To illustrate the approach, we begin with a brief overview of (the simplest variant of) the Lasso lookup argument from our companion paper [STW23]. Here, let us consider a VM that is given read-only access to lookup table T. To render the VM's reads more easily checkable, let us modify the VM's reading procedure as follows.

- For each memory cell $j = 1, \ldots, N$ maintain a counter c_j, which is supposed to track how many times cell j has been read.
- Every time a read operation to cell b_i returns a (value, count) pair (a, c), have the VM write the tuple $(a, c+1)$ to cell b_i. That is, the VM follows every read operation by writing the returned value back to the cell that was just read, incrementing the returned counter value by 1.

- When all the reads are done, the VM makes one final pass over memory. This final set of N reads are not paired with write operations.

Let RS (short for *read-set*) be the vector of all (cell, value, count) tuples returned by the memory across all read operations. Let WS (short for *write-set*) be the vector of all (cell, value, count) tuples across all write operations. WS includes the N writes to initialize memory, i.e., the tuples $(j, T[j], 0)\colon j = 1, \ldots, N$. Prior works [BEG+91, Set20, STW23] establish the following lemma, whose proof we omit for brevity.

Lemma 2. RS *and* WS *are permutations of each other if and only if the result of every read operation to each cell j indeed returns the (value, count) pair last written to that cell. Here,* RS *and* WS *are permutations of each other if they are equal as multisets of (cell, value, count) tuples.*

With this lemma in hand, one can obtain an (indexed) lookup argument as follows. Recall that in an indexed lookup argument, the prover has committed to two vectors $a, b \in \mathbb{F}^m$, and wishes to prove that $a_i = T[b_i]$ for all i. View a as the vector of all values returned by the read operations of the modified VM above, and b as the vector specifying the cell targeted by each read operation. The prover next commits to the vector $c \in \mathbb{F}^{m+N}$, whose i'th entry is purported to be the count returned by the i'th read operation.

If the prover chooses c honestly, then by Lemma 2, RS and WS are permutations of each other. Conversely, by the same lemma, if RS and WS are permutations of each other, then each read operation returned the correct value, and hence $a_i = T[b_i]$ for all $i = 1, \ldots, m$. So it suffices for the lookup argument prover to prove that RS and WS are permutations of other.

To this end, first Reed-Solomon fingerprint each (cell, value, count) tuple. That is, the verifier picks a random $\gamma \in \mathbb{F}$ and sends γ to the prover. The prover then replaces all tuples (b, a, c) in RS or WS with $b + \gamma a + \gamma^2 c$.

This reduces RS and WS from vectors of $N + m$ *triples* of field elements, to vectors RS' and WS' in $\mathbb{F}^{N+m}$. If there are no "collisions" (two distinct tuples with matching fingerprints) then RS' and WS' are permutations of each other if and only if RS and WS are. Clearly, the probability of a collision is at most $(N + m)^2/|\mathbb{F}|$ (a more careful analysis can bound the soundness error by at most $2(N + m)/|\mathbb{F}|$).

Hence, up to the above soundness error, checking whether RS and WS are permutations of each other is equivalent to checking whether RS' and WS' are permutations.

This latter check is done via permutation-invariant fingerprint (Figure 3).

In summary, in the indexed lookup argument, the prover first commits to the vector $c \in \mathbb{F}^{m+N}$ purported to be the counts returned by the read operations specified by the vectors $a, b \in \mathbb{F}^m$. (More precisely, the prover commits to the multilinear extension $\widetilde{c}$ of c, using any multilinear polynomial commitment scheme).

The verifier then confirms that RS is a permutation of WS by picking two random field elements γ, α, and ensuring that the following two quantities are equal:

$$\prod_{i=0}^{m-1} \left(\alpha - \left(b_i + \gamma \cdot a_i + \gamma^2 \cdot c_i\right)\right) \prod_{i=0}^{N-1} \left(\alpha - \left(i + \gamma \cdot T[i] + \gamma^2 \cdot c_{m+i}\right)\right) \tag{12}$$

$$\left(\prod_{i=0}^{N-1} \left(\alpha - (i + \gamma \cdot T[i])\right)\right) \cdot \left(\prod_{i=0}^{m-1} \left(\alpha - \left(b_i + \gamma \cdot a_i + \gamma^2 \cdot (c_i + 1)\right)\right)\right). \tag{13}$$

Here, Expressions (12) and (13) are the permutation-invariant fingerprints of RS' and WS' respectively.

Lasso forces the prover to correctly compute these two expressions using any grand product argument. Specifically, it Lasso suggests to use either a highly optimized variant of the GKR-protocol [GKR08] due to Thaler [Tha13], which avoids any additional commitment costs for the prover, or a variant with shorter proofs but slightly higher commitment costs [SL20]. At the end of these grand product arguments, the verifier needs to evaluate each of $\widetilde{T}$, $\widetilde{a}$, $\widetilde{b}$, and $\widetilde{c}$ at a randomly chosen point. The evaluations of $\widetilde{a}$, $\widetilde{b}$, and $\widetilde{c}$ can be obtained from the polynomial commitment scheme used to commit to each of these polynomials. For MLE-structured tables, the evaluation of $\widetilde{T}$ can be computed by the verifier with only $O(\log N)$ field operations.

The above argument protocol is implicit in Spark, the sparse polynomial commitment scheme given in Spartan [?]. However, Spark's security analysis assumed that (the commitment to the) vector c of purported counts is computed by an honest party. This sufficed for Spartan's application, but not for giving a lookup argument. Our companion paper Lasso shows that the lookup argument is secure even if c is committed by a malicious party.

Overview of Spice's memory-checking argument. In Lasso, having