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) refers to 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 outlined by Barry Whitehat 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 have structural properties that enable 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

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

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.

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, or even 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 a combination thereof. For example, Cairo-VM is a very simple virtual machine designed specifically for compatibility with SNARK proving [**?**, **?**]. 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. Yet another zkVM project is Polygon Miden.

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. That is, one can directly invoke existing compilers that transform witness-checking programs written in high-level languages down to assembly code for the ISA. 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.

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 discussion in Section 1.5 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 [**?**, **?**, lib] and can be much faster than backends that don't leverage repeated structure [**?**, **?**, **?**].

[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. 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.

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. Ultimately, 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 [**?**]. 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 perhaps 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 extremely expensive for the prover, even for very simple ISAs. For example, the prover for Cairo-VM programs described in [**?**] 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 hardware. 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, and as a concrete example, we instantiate them for the RISC-V instruction set (on 64-bit data types, with multiplication extension [**?**]).

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.

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 $i \in \{1, \ldots, m\}$, 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. A 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 a condition 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.

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 succinct 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 $c_0, \ldots, c_{n-1}$ over an appropriate basis) is

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

for some public generators $g_1, \ldots, g_n$ of a multiplicative group $\mathbb{G}$. Examples include KZG [**?**], Bulletproofs/IPA [**?**, **?**], Hyrax [lib], and Dory [**?**].[2]

The naive MSM algorithm performs n group exponentiations and n group multiplications (note that each group exponentiation can be about 400× 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× 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× relative to applying Pippenger's algorithm to an MSM involving random exponents. This is analogous to how computing $g_i^{2^{16}}$ is 10× 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 $\{0, 1, \ldots, K\} \subset \mathbb{F}$, the number of group operations required to compute the MSM depend only on K and not on the size of $\mathbb{F}$[3]

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 wit a *single* group operation.

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

[3] 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 $\mathbb{F}$

Prover costs of Jolt. 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.

One caveat is that we handle two 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. A counter-caveat is that many instructions (those involving addition, subtraction, shifts, jumps, loads, and stores) can be handled with *fewer than* five committed 256-bit field elements.

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, plus checking one evaluation proof from the chosen polynomial commitment scheme (when applied to a multilinear polynomial over at most $O(\log T)$ variables).

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 [**?**]) 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 [**?**] 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 [**?**], which currently targets the RISC-V ISA on 32-bit data types (with multiply 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.[4] 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 [**?**]. 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 [**?**], 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 [**?**]. 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 [**?**]. 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 are at least equivalent to committing to $50 \cdot 64/256 \approx 13$ 256-bit field elements.[5] 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").

[4] 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.

[5] 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.

1.3 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.

1.4 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 $t_1, \ldots, t_\ell$, 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,[6] then no one needs to cryptographically commit to any of the tables (neither to t itself, nor to $t_1, \ldots, t_\ell$). 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 $t_1, \ldots, t_\ell$, 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.
- 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.

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).

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

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

$$c = \sum_{i=0}^{63} 2^i \cdot \mathsf{AND}(a'_i, b'_i), \tag{1}$$

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.[7]

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

1.5 Other related Work

discuss work on dealing with memory? memory-checking 1994–¿buffet/pantry–¿vRAM

A predecessor of today's zkVM projects is TinyRAM [BSCG+13, BSCTV14], which specified an extremely simple CPU and turned its transition function into an arithmetic circuit with over $1,000$ look up number gates.

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. 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, Bellman, Circom, Zokrates, Noir, etc. We believe that the circuits produced by these approaches can also likely be made much smaller using our techniques, though we leave this to future work.

[7] 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.

2 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 [**?**].[8]

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

Definition 2.1 (Machine State)**.** *The machine state consists of a set of registers, a Program Counter and the contents of Memory. There are 32 registers, each of L bits, where L is 32 or 64. The `PC`, also of L bits, is a separate register that stores the address of the next instruction to be executed. What does address mean? It's the index of a memory cell? So Integer registers is an array of 32 difference 32- or 64-bit values? Memory is what?*

Assembly Instructions: 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`. That is, `PC` is a designed register that at all times holds the address of the current instruction to be executed. 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 (i.e., 4 bytes), so advancing the `PC` to the next memory location entails incrementing `PC` by 4. check this

Each instruction specifies an operation, at most two *source registers* `rs1, rs2` (`rs` stands for source register), a *destination register* `rd` (`rd` standards for destination register), and a constant value `imm` (`imm` stands for immediate, and it is a constant value provided by the definition of the instruction itself). Operations read the source registers, perform some computation, and can do any or all of the following: store a value in `rd`, update the `PC`, read from memory, or write to memory.

Definition 2.2 (5-tuple RISC-V Instruction Format)**.** *RISC-V instructions are specified in the following format: `[opcode, rs1, rs2, rd, imm]`.*

CPU Transition Function: Applying one step of the RISC-V CPU does the following.

1. Read the instruction at location PC.
2. Parse instruction as `[opcode || rs1 || rs2 || rd || imm]`.
   (a) Read the L-bit values stores in registers `rs1, rs2`.
   (b) For load instructions, read the memory cell with address specified by `rs1, rs2, imm`. This memory address is clarify this
   (c) For store instructions, write to an address, where both the the value and location are specified by `rs1, rs2, imm`. clarify this
   (d) Run the appropriate functionality be more precise than run the appropriate functionality... to get output `out`.
3. Update the PC: the PC is typically incremented to the next memory location, but some instructions (such as branches and jumps) update the value based on `out`.
4. Update `rd`: this usually means writing the value `out` to `rd`.

TODO: Talk about flags and status registers. time to actually do this todo...

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

2.1 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

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

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.[9] 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 $L = 3$.

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.

[9] 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.

2.2 Designing CPU Circuits with Lookups

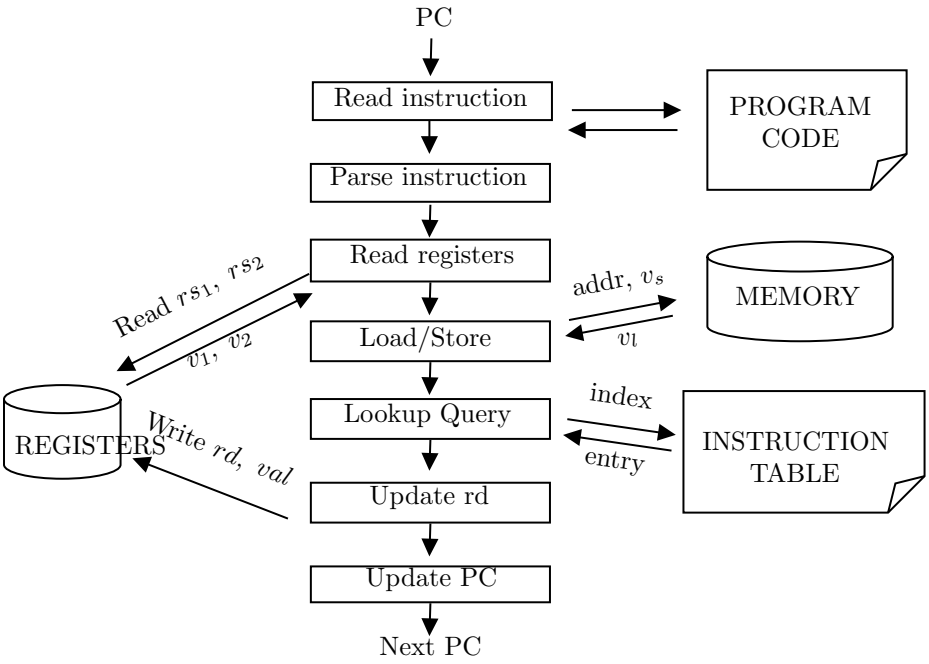

Figure 1: Draft of a diagram

3 Technical Preliminaries

3.1 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[10]: 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.

do we need Lagrange interpolation or sum-check?

SNARKs We adapt the definition provided in [**?**].

Definition 3.1. *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.
- $\mathcal{K}(\mathsf{pp}, \mathsf{s}) \to (pk, \mathsf{vk})$*: On input structure* s*, representing common structure among instances, outputs the prover key pk and verifier key* vk.

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

- $\mathcal{P}(pk, u, w) \to \pi$*: 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 $\mathcal{A}$

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

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

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

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 scheme We adapt the definition from [**?**]. 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 3.2. *A tuple of four protocols* $(\mathsf{Gen}, \mathit{Commit}, \mathit{Open}, \mathit{Eval})$ *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* $\mathcal{G} \in \mathbb{F}[\ell]$*,*

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

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

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

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

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

3.2 Polynomial IOPs and polynomial commitments

Modern SNARKs are constructed by combining a type of interactive protocol called a *polynomial IOP* [**?**] with a cryptographic primitive called a *polynomial commitment scheme* [**?**]. The combination yields a succinct *interactive* argument, which can then be rendered non-interactive via the Fiat-Shamir transformation [**?**], 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.[11] 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.

3.3 Lookup Arguments

Lasso and its costs for indexed lookups, decomposable, LDE/MLE-structured.

say that, to emphasize the interpretation of T as a table, we use square brackets $T[i]$ to denote the i'th entry of T.

MLE-structured. We say that a vector $t_i \in \mathbb{F}^{N^{1/c}}$ is *MLE-structured* if for any input $r \in \mathbb{F}^{\log(N)/c}$, $\widetilde{\mathrm{t}}_i(r)$ can be evaluated with $O(\log(N)/c)$ field operations.

Decomposable tables. Let $T \in \mathbb{F}^N$. We say that T is c-decomposable if there exist $\alpha \leq kc$ tables $T_1, \ldots, T_\alpha$ each of size $N^{1/c}$ and a multilinear α-variate polynomial g such that the following holds. As in Section 3.1, 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 $r \in \{0,1\}^{\log N}$,

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

3.4 Offline Memory Checking

sketch of how this works, maybe copy-paste your slides, say we use it in a now-standard way [cite vRAM?] to check that memory of the RISC-V CPU was maintained correctly? point to Lasso?

formally define decomposable and MLE-structured tables

[11] Any univariate polynomial commitment scheme can be transformed into a multilinear one, though the transformations introduce some overhead (see, e.g., [**?**, **?**, **?**]).

4 Decomposability and LDE-structure of each RISC-V Instruction

We consider each of the RISC-V instructions one at a time, and analyze the decomposability and LDE-structure of its evaluation table. Conceptually, Jolt's "one giant lookup table" is obtained by simply concatenating each the evaluation tables for all instructions (see Section 7). what about 65-bit tables instead of 128 bits.

4.1 Pre-processing Instructions

While assembly instructions can have various formats when stored in memory, they can always be represented precisely as a five-tuple `(opcode, rs1, rs2, rd, imm)`. As described in Section 2, `opcode` is used to denote the specific instruction to be executed, `rs1` and `rs2` are source registers (storing the inputs to the instruction), `rd` is the destination register (which will store the output of the instruction), the memory cell that should be read, and imm stand for immediate.

The "pre-processing" of assembly instructions into this representation is done before the proof starts what does this mean? You mean the circuit output by the Jolt front-end takes opcode, rs1, rs2, rd, imm as non-deterministic advice or something? and Jolt only reads instructions in the above 5-tuple form. Note that this transformation is purely syntactic what does this mean? and deterministic for any given instruction.

Pre-processing immediates: The immediate is a possibly signed constant of up to 20-bits in length that might be provided in the instruction (and thus, fixed in the assembly code of the program). This immediate is either zero-extended or sign-extended to L bits. Remind the reader what L is? When pre-processing the assembly code, Jolt additionally stores in a separate bit the sign of the immediate. Looking ahead, this is a convenience for instructions (namely, jumps and branches) where the immediate is used in arithmetic operations performed by the circuit output by Jolt (as opposed to being performed in a lookup). We note again that the exact type of processing to be done is deterministic given the instruction syntax and independent of the rest of the program. so the verifier in the resulting SNARK would do this pre-processing?? We should say/clarify this??

4.2 The Operands

The actual operation performed by an instruction will take at most two inputs from the set {`value in rs1, value in rs2, imm`}. The output of the operation is either stored in register `rd`, used to modify `PC`, or both. In the remainder of this section, we'll refer to the two operands generically as as x and y as the exact sources of the values do not affect the output of the operation. The inputs x, y will always be L-bits long. `imm` is always at most L bits. I thought imm was at mot 20 bits?

4.3 Showing RISC-V Instructions are Decomposable and LDE-Structured

4.3.1 Notation

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

$$z = \sum_{i=0}^{L-1} 2^i z_i.$$

We use $z_{<i}$ to refer to the subsequence $[z_{i-1}, \ldots, z_0]$. Analogously, $z_{>i}$ refers to the subsequence $[z_{L-1}, \ldots, z_{i+1}]$.

Similarly, given a vector $z = [z_{L-1}, \ldots, z_0] \in \{0,1\}^L$, we denote the associated field element as $\mathsf{int}(z) = \sum_{i=0}^{L-1} 2^i \cdot z_i$.

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

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

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

Under this definition, it holds that

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

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

$$x = [x_{L-1} \ldots x_0] = X_{c-1} \parallel \ldots \parallel X_2 \parallel X_0, \tag{3}$$

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

4.4 Three Instructive Functions

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

4.4.1 The Equality Function

LDE-structured. The equality function eq takes as inputs two vectors $x, y \in \{0,1\}^L$ 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_L denotes the equality function defined over domain $\{0,1\}^L \times \{0,1\}^L$. It is easily confirmed that its multilinear extension of eq is as follow:

$$\widetilde{\mathsf{eq}}_L(x, y) = \prod_{j=0}^{L-1} \left(x_j y_j + (1 - x_j)(1 - y_j)\right). \tag{4}$$

Indeed, the right hand side is clearly a multilinear polynomial in x and y, and if $x, y \in \{0,1\}^L$, 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 (4) can be evaluated at any point $(x, y) \in \mathbb{F}^L \times \mathbb{F}^L$ with $O(L)$ field operations.

Decomposability. To determine whether two L-bit inputs $x, y \in \{0,1\}^L$ are equal, one can decompose x and y into c chunks of length L/c, say somewhere we assume for simplicity that c divides L? 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 (3). Let EQ_L denote the "big" table of size $N = 2^{2L}$ indexed by pairs (x, y) with $x, y \in \{0,1\}^L$, such that $t_{\mathsf{eq}}[x, y] = \mathsf{eq}(x, y)$. Let $\mathsf{EQ}_{L/c}$ denote the "small" table of size $N^{1/c}$ indexed by pairs (X, Y) of chunks $X, Y \in \{0,1\}^{L/c}$, 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**
$\mathsf{C}_i = X_i \parallel Y_i$, $X_i, Y_i \in \{0,1\}^{L/c}$	$\mathsf{EQ}_{L/c}[X, Y] = \mathsf{eq}_{L/c}(X, Y)$	$\mathsf{EQ}_L[x, y] = \prod_{i=0}^{c-1} \mathsf{EQ}_{L/c}[X_i, Y_i]$

The (lone) subtable $\mathsf{EQ}_{L/c}$ is LDE-structured by Equation (4).

4.4.2 Set Less Than - Unsigned (SLTU)

LDE-structured. The SLTU function takes as input two unsigned data types $x, y \in \{0, 1, \ldots, 2^{L-1}\}$, 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 $(2L)$-variate multilinear polynomial:

$$\widetilde{\mathsf{LT}}_i(x, y) = (1 - x_i) \cdot y_i \cdot \widetilde{eq}_{L-i}(x_{>i}, y_{>i}). \tag{5}$$

Clearly, this polynomial satisfies the following two properties:

(1) When $x \geq y$, $\widetilde{\mathsf{LT}}_i(x, y) = 0$ 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 $\widetilde{\mathsf{LT}}_k(x, y) = 1$ and $\widetilde{\mathsf{LT}}_i(x, y) = 0$ for all $i \neq k$.

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

$$\widetilde{\mathsf{SLTU}}(x, y) = \sum_{i=0}^{L-1} \widetilde{\mathsf{LT}}_i(x, y). \tag{6}$$

Indeed, the right hand side is clearly multilinear, and by the two properties above, it equals $\mathsf{SLTU}(x, y)$ whenever $x, y \in \{0, 1\}^L$. It is not difficult to see that the right hand side of Equation (6) can be evaluated at any point $(x, y) \in \mathbb{F}^L \times \mathbb{F}^L$ with $O(L)$ field operations. elaborate on this?

Decomposing SLTU. A similar reasoning to the derivation of Equation (6) reveals the following. As usual, break x and y into c chunks, $X_1 \| \cdots \| X_c$ and $Y_1 \| \cdots \| Y_c$. Let $\mathsf{SLTU}_{L/c}(X_i, Y_i)$ denote the function that outputs 1 if $X_i < Y_i$ when interpreted as unsigned (L/c)-bit data types, and 0 otherwise. Then

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

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

CHUNKS	SUBTABLES	FULL TABLE
$\mathtt{C}_i = X_i \| Y_i$ $X_i, Y_i \in \{0, 1\}^{L/c}$	$\mathsf{SLTU}_{L/c}(X, Y)$ and $\mathsf{EQ}_{L/c}(X, Y)$	$\mathtt{SLTU}_L(x, y) = \sum_{i=0}^{c-1} \mathsf{SLTU}_{L/c}(X_i, Y_i) \cdot \prod_{j>i} \mathsf{EQ}_{L/c}(X_j, Y_j)$.

The two subtables SLTU and EQ are LDE-structured by Equations (4) and (6).

4.4.3 Shift Left Logical

Arasu, can you shift left by a full L bits, with the output the being all 0s? If so, y needs to be in $\{0, 1\}^{\log(L+1)}$, not $\{0, 1\}^{\log L}$, right? Is it possible to "input to the instruction" a y that's even larger than L?

LDE-structured. SLL takes an L-bit integer x and a $\log(L)$-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 zeros the vacated lower bits are filled with zeros. For a constant k, let

$$\widetilde{\mathsf{SLL}}_k(x) = \sum_{j=k}^{L-1} 2^j \cdot x_{j-k}. \tag{7}$$

It is straightforward to check that the right hand side of Equation (7) is multilinear (in fact, linear) function in x, and that when evaluated at $x \in \{0,1\}^L$, it outputs the unsigned L-bit data type whose binary representation is given by $\mathsf{SLL}(x,k)$.

Now consider

$$\widetilde{\mathsf{SLL}}(x,y) = \sum_{k \in \{0,1\}^{\log L}} \widetilde{eq}(y,k) \cdot \widetilde{\mathsf{SLL}}_k(x). \tag{8}$$

It is straightforward to check that the right hand side of Equation (8) is multilinear in (x,y), and that, when evaluated at $x \in \{0,1\}^L \times \{0,1\}^{\log L}$, it outputs the unsigned L-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 $L' = L/c$ bits. As explained below, we decompose a lookup into the evaluation table of SLL into a lookup into c different subtables, each of size $2^{L'+\log L}$. For $L = 64$, a reasonable setting of c would be 4, ensuring that $2^{L'+\log L} = 2^{22}$.

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 $k \in \{0,1\}^{\log L}$ check on max shift length, define:

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

Here, $m_{i,k}$ equals the number of bits from the i'th chunk that go out of range. Let $m'_{i,k} = (L-1) - m_{i,k}$ 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 $T_0, \ldots, T_{c-1}$ as follows.

actually, in the table below, should y be L bits rather than $\log L$, but with all but the low-order $\log L$ bits equal to 0? If so, we should actually replace y in the table with Y_0 and say we are promised that all bits of y other than the low-order $\log L$ bits are zero.

CHUNKS	**SUBTABLES**	**FULL TABLE**
$\mathtt{C}_i = X_i \parallel y$	For $i \in \{0, \ldots, c-1\}$, $T_i(\mathtt{C}_i)$ equals	$\mathtt{SLL}(x,y) = \sum_{i=0}^{L/c-1} 2^i T_i(X_i, y)$
$x = X_{c-1} \parallel \cdots \parallel X_0$, $X_i \in \{0,1\}^{L/c}$, $y \in \{0,1\}^{\log L}$	$\sum_{k \in \{0,1\}^{\log L}} \mathsf{eq}(y,k) \cdot 2^{\mathsf{int}(k)} \cdot \left(\sum_{j=0}^{m'_{i,k}} 2^j \cdot X_{i,j} \right)$	

Note that

$$\widetilde{T}_i(x,y) = \sum_{k \in \{0,1\}^{\log L}} \widetilde{\mathsf{eq}}(y,k) \cdot 2^{L' \cdot (i-1) + \mathsf{int}(k)} \cdot \left(\sum_{j=0}^{m'_{i,k}} X_{i,j} \right)$$

and that $\widetilde{T}_i$ can be evaluated at any input $(x,y) \in \mathbb{F}^L \times \mathbb{F}^{\log L}$ in $O(L)$ field operations. explain this... requires evaluating $\widetilde{\mathsf{eq}}(y,k)$ at all Boolean inputs k, and then exploiting that as k increases, the sum on the right can be updated in constant time. Note that if y were more than $\log L$ bits, evaluating $\widetilde{\mathsf{eq}}(y,k)$ at all Boolean inputs k would take more than L field ops. Also, flag that this means you need to redefine decomposable to not require the divided-by-c factor in verifier time for MLE evaluation...

5 Analyzing the Evaluation Tables of RISC-V Instructions

The lookup tables employed for each instruction in RV64I is presented below. For some instructions, the lookup performs the entire operation and the looked-up entry needs to simply be stored in `rd`. In other cases (such as in branches), the lookup is used to perform a core part of the the operation (such as a comparison) and the entry is processed further in the circuit.

5.1 Logical Instructions

Each instruction performs the corresponding operation bitwise over the L-bits of x and y and stores the L-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	MLE OF FULL TABLE
AND	$x \parallel y \in \{0,1\}^L \times \{0,1\}^L$	$\sum_{i=0}^{L-1} 2^i \cdot (x_i \cdot y_i)$
OR	$x \parallel y \in \{0,1\}^L \times \{0,1\}^L$	$\sum_{i=0}^{L-1} 2^i \cdot (x_i + y_i - x_i \cdot y_i)$
XOR	$x \parallel y \in \{0,1\}^L \times \{0,1\}^L$	$\sum_{i=0}^{L-1} 2^i \cdot (x_i \cdot (1-y_i) + y_i \cdot (1-x_i))$

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

OP	CHUNKS	SUBTABLES	FULL TABLE
$\mathsf{AND}(x,y)$	$\mathtt{C}_i = X_i \parallel Y_i$	$\mathsf{AND}_{L/c}[X,Y]$	$\mathsf{AND}_L[x,y] = \sum_{i=0}^{c-1} 2^{i\cdot L/c} \cdot \mathsf{AND}_{L/c}[X_i, Y_i]$
$\mathsf{OR}(x,y)$	$\mathtt{C}_i = X_i \parallel Y_i$	$\mathsf{OR}_{L/c}[X,Y]$	$\mathsf{OR}_L[x,y] = \sum_{i=0}^{c-1} 2^{i\cdot L/c} \cdot \mathsf{OR}_{L/c}[X_i, Y_i]$
$\mathsf{XOR}(x,y)$	$\mathtt{C}_i = X_i \parallel Y_i$	$\mathsf{XOR}_{L/c}[X,Y]$	$\mathsf{XOR}_L[x,y] = \sum_{i=0}^{c-1} 2^{i\cdot L/c} \cdot \mathsf{XOR}_{L/c}[X_i, Y_i]$

5.1.1 Arithmetic Instructions

Addition. For $x, y \in \{0,1\}^L$, `ADD`(x, y) returns the lowest L bits of (the binary representation of) the sum $\mathsf{int}(x) + \mathsf{int}(y)$. As discussed in Section 2.1, 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 $\{0,1\}^L$, 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 $L+1$ bits long, i.e., z can be any field element in $\{0, \ldots, 2^{L+1} - 2\}$. 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 $(L+1)$-bit vectors $z \in \{0,1\}^{L+1}$, with the z's entry equal to the field element $\sum_{i=0}^{L-1} 2^i \cdot z_i$ equal to $\mathsf{int}(z_{<L})$. Note that this lookup table has size only 2^{L+1}, which is less than the tables of size 2^{2L} 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 L-bit string as

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

In words, subtracting y from x is equivalent to adding the two's complement of y to x.[12] check this. insert citation?

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

Decomposability. 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 [?]. The prover provides as untrusted advice c field elements $Z_1, \ldots, Z_c$ claimed to equal the L/c-bit limbs of $\sum_{i=0}^{L-1} 2^i z_i$, as well as a final field element b that is constrained to satisfy $b \cdot (b-1) = 0$ (i.e., $b \in \{0,1\}$). clarify/rewrite $Z_1, \ldots, Z_c$ are each confirmed to be in the "small" table $T_{L/c}$ of size $2^{L/c}$ such that $T[j] = \sum_{i=0}^{L/c-1} 2^i j_i$ for each $j \in \{0,1\}^{L/c}$.

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 L-bit numbers, respectively.

The table, full MLE and decomposed MLE for `SLTU` were derived in Section 4.4.2. 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	MLE
SLTU	$x \parallel y$	See Section 4.4.2
SLT	$x \parallel y$	$\widetilde{\mathsf{LTS}} := x_s \cdot (1 - y_s) + \widetilde{\mathsf{eq}}(x_s, y_s) \cdot \widetilde{\mathsf{LTU}}(x_{<s}, y_{<s})$

The decomposition of `SLTU` was discussed in Section 4.4.2 and requires two subtables of size $2^{L/c}$. The decomposition of `SLT` uses the same decomposition (applied to $(x_{<s}, y_{<s})$, which has $L-1$ rather than L bits). Additionally, the R1CS instance takes as untrusted advice inputs two field elements x_s, y_s, which are constrained to be in $\{0,1\}$ by requiring that $x_s(1-x_s) = 0$ and $y_s(1-y_s) = 0$. write out more clearly 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. TODO: We can actually have the final function compute those terms right? As they involve just reading two bits? Yes

[12] In a system implementing arithmetic on L-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 $\left(2^L - 1 - y\right) + 1$, 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 $2^L - y$.

5.1.3 Shifts

$\texttt{SLL}(x, y)$ (Shift Left Logical) and $\texttt{SRL}(x, y)$ (Shift Right Logical) perform left/right logical shifts of x by length y, respectively, where $y < L$. $\texttt{SRA}(x, y)$ (Shift Right Arithmetic) shifts the bits of x right by length y, where $y < L$.

All operations return L-bit values, as 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 4.4.3. We provide the analogous information for for SRL and SRA below. For SRA we additionally ask change this language the subtable handling the most significant bit to also perform sign-extension.

For SRL, SRA, we define: (note that if $m' > 15$, then $X_{i,\geq m'} = 0$)

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

OP	CHUNKS	SUBTABLES	FINAL TABLE
SRL	$\texttt{C}_i = X_i \parallel Y$	For $i \in [0,3]$, $T_i(\texttt{C}_i) = \sum_{k=0}^{63} \widetilde{\texttt{eq}}(y, \overline{k}) \cdot 2^{16\cdot(i-1)-k} \cdot X_{i,\geq m'_{i,k}}$	$T_{\texttt{SRL}} = \sum_{i=0}^{3} T_i$
SRA	$\texttt{C}_i = X_i \parallel Y$	For $i \in [0,2]$, $T_i(\texttt{C}_i) = \sum_{k=0}^{63} \widetilde{\texttt{eq}}(y, \overline{k}) \cdot 2^{16\cdot(i-1)-k} \cdot X_{i,\geq m'_{i,k}}$ $T_4(C_4) = \sum_{k=0}^{63} \widetilde{\texttt{eq}}(y, \overline{k}) \cdot 2^{16\cdot(i-1)-k} \cdot X_{i,\geq m'_{i,k}} + \sum_{i=64-k}^{63} 2^i \cdot X_{15}$ I don't underestand X_{15} here	$T_{\texttt{SRA}} = \sum_{i=0}^{3} T_i$

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 4.1) into an L-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 L-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?

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

point out again these are smaller tables, as for ADD and SUB

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 4.1, 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	MLE
JAL	$z = x + y$	$\sum_{i=0}^{L-1} 2^i z_i$ `// identity`
JALR	$z = x + y$	$\sum_{i=1}^{L-1} 2^i z_i$ `// identity, but with LSB set to 0`

Decomposition of these MLEs can be done in the natural way, requiring only one subtable function.

point out again these are smaller tables, as for ADD and SUB work through details of how the extra bit (out of $L + 1$) is specified...

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 4.4.2 and used in `SLT/SLTU`. We use the same MLEs here, along with the $\widetilde{eq}$ MLE.

OP	INDEX	MLE
BEQ	$x \parallel y$	$1 - \widetilde{eq}(x, y)$
BNE	$x \parallel y$	$\widetilde{eq}(x, y)$
BLTU	$x \parallel y$	$\mathsf{LTU}(x, y)$ `// from SLTU`
BLT	$x \parallel y$	$\mathsf{LTS}(x, y)$ `// from SLT`
BGEU	$x \parallel y$	$1 - \mathsf{LTU}(x, y)$
BGE	$x \parallel y$	$1 - \mathsf{LTS}(x, y)$

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

Memory operations are handled in the circuit outside of the lookup. The lookup tables are used to perform the requisite sign-extensions and bit cutting on the values being read or written.

`LD` reads a 64-bit value from memory and stores it into `rd`. The lookup performs no processing here. `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 L bits into `rd`. `L[W/H/B]U` are identical to their signed counterparts but do not perform any sign-extension. The lookup tables for these instructions have a row all possible L-bit values with the entry being the sign- or zero-extension of the lowest 32/16/8 bits of the index.

`SD` takes a 64-bit operand, y, and stores it into a specified memory location. As we don't need any processing here, we just employ the identity lookup. `S[W/H/B]` store only the lowest 32/16/8 bits of the operand (without any sign-extension). The lookup tables for these instructions have a row all possible L-bit values with the entry being the lowest 32/16/8 bits of the index.

OP	**INDEX**	**MLE**
LD	y	$\sum_{i=0}^{L} y_i$ `identity`
L[W/H/B]	y	$\sum_{i=k}^{L} 2^i \cdot y_{k-1} + \sum_{i=0}^{k-1} 2^i \cdot y_i$ `where k = 32/16/8`
L[W/H/B]U	y	$\sum_{i=0}^{k-1} 2^i \cdot y_i$ `where k = 32/16/8`
SD	y	$\sum_{i=0}^{L} y_i$ `identity`
S[W/H/B]	y	$\sum_{i=0}^{k} y_i$ `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. As these instructions are more complex and involve non-deterministic advice, we introduce new techniques to efficiently support them.

6.1 Virtual Instrutions and Virtual Registers

Jolt splits some assembly instructions into a sequence of multiple 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 at the assembly code during pre-processing and is independent of the input.

To avoid jumbling with the base registers, Jolt contains "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, stored to and treated identically. In a sequence of virtual instructions, only the final result is reflected into the true destination register of the original instruction. As instructions are executed sequentially in Jolt, virtual registers never overlap between instructions and no more than five are needed in total. For clarity, we refer to the these registers as v_q indicating that the virtual register v stores variable q. In actual code, these are replaced by a free numbered virtual register.

6.1.1 Virtual Assert Instructions

Asserts are a special type of new virtual instruction that add circuit constraints on the lookup output. 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 uses the following assert instructions:

- `ASSERT_LT_ABS` takes two L-bit 2's complement signed inputs and outputs $|x| < |y|$.
- `ASSERT_EQ_SIGNS` takes two L-bit 2's complement signed inputs and outputs $x_s == y_s$.

OP	INDEX	MLE
`ASSERT_LT_ABS`	$x \parallel y$	$\mathsf{LTU}(x_{<63}, y_{<63})$ `// ignore sign bits`
`ASSERT_EQ_SIGNS`	$x \parallel y$	$\widetilde{eq}(x_s, y_s)$

6.1.2 Virtual Advice and Move Instructions

Advice instructions allow 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 read from the registers or the instruction, but a non-deterministic advice value passed into the circuit. This differs from the immediate as advice isn't present in the assembly code and need not be known at the start of the proof. Thus, advice instructions have no source register or immediate and only specify a destination register. Move instructions copy the contents of one register into another and also use the range check lookup table.

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

6.2 The MUL Tables

6.2.1 Unsigned or Lower MUL

Tthe following instructions take two L-bit operands x and y.

- `MUL` returns the lower L bits of $x \times y$ where the operands are treated as signed 2's complement numbers.
- `MULU` returns the lower L bits of $x \times y$ where the operands are treated as unsigned L-bit numbers.
- `MULHU` returns the higher L bits of $x \times y$ where the operands are treated as unsigned L-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 $2L$-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	MLE
MUL	$z = x \times y$	$\sum_{i=0}^{L-1} z_i$ // lower L bits
MULU	$z = x \times y$	$\sum_{i=0}^{L-1} z_i$ // lower L bits
MULHU	$z = x \times y$	$\sum_{i=L}^{2L-1} z_i$ // higher L bits

6.2.2 Signed and Higher MUL

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

MULHSU returns the higher L 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 $2L$ bits before performing the multiplication. This leads to the result having $4L$ and $3L$ total bits in MULH and MULHSU, respectively. We avoid this by computing the desired bits in stages.

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

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

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:

MOVSIGN takes an L-bit input x and stores the L-bit number with x_s as all of its binary coefficients in the destination register.

OP	INPUT	MLE
MOVSIGN	x	$\sum_{i=0}^{L-1} x_s$ // 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_a to denote the virtual register storing operand a. (In actual code, these are replaced by a free numbered virtual register.)

Original	Virtual Sequence (OP, 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 $\mathrm{v}_0, \mathrm{v}_1, -$, rd 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

6.3 Division and Remainder

In RISC-V, division and remainder operations take two L-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 L-bit numbers.

- `DIVU/REMU` requires $x = q \times y + r$ such that $r < y$ and $q \times y \leq x$.

Original	Virtual Sequence (OP, 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`

6.4 One Giant Table

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.

8 Cost Estimation

We first provide a qualitative evaluation of Jolt. Figure 8.1 depicts the prover costs involved for the RV64IM processor when proving statments about programs that run for m steps on the RV64IM processor.

The table on the left lists the broad steps the prover performs and their associated costs. The bottleneck criteria here is the first step, cryptographically committing to the witness elements involved in the CPU steps. As the monolithic circuit for the program consists of data parallel computations of the smaller CPU circuit, the R1CS constraint count (which is around 100 for RV64IM) is not high enough to matter when using sum-check-based SNARK, such as Spartan. Here we see that Jolt commits to totally 48 witness elements per step, which is 4-5x fewer elements than protocols like risc0 (which commits to at least 270 elements. After this, the prover commits to $c \cdot N^{1/c}$ elements to complete the Lasso proof which, substituting $N = 2^{140}, c = 7$ for RV64IM, comes to about $7 \cdot 2^{20}$ field elements (regardless of the number of CPU steps). TODO: Talk about Spartan and memory checking.

8.1 Significant Elements

- The table on the right dives into this the cost of committing to witness elements (the first step) by detailing the number of significant bits of the elements committed.
- As commitments done using the multi-exponentiation operation cost roughly in proportion to the number of significant 64-bit chunks in the binary representation of field element committed, Jolt's performance becomes even better: every element that the prover commits to is at most 64 bits!
- In particular, for RV32I the largest element the prover commits to is just 50-bits with all other elements being at most 32 bits.
- For RV64I, the prover commits to only 4 non-zero elements between 32 and 64-bits long.
- Of these, 26 elements are witness elements involved in the CPU transition of Jolt and the remaining 18 are involved in Lasso (with parameter $c = 6$).
- Appendix A provides more details about the particular elements committed.

Step	Cost
1. Commit to Witness	$48 \cdot m$ comm
2. Lasso Proof	$7 \cdot 2^{20}$ comm
3. SuperSpartan Proof	
4. Memory Checking	
Total Prover Cost	

Witness Elements per Step

Sig Bits	RV64IM	RV32IM
1 sig. bit	14 elements	14 elements
$\leq$ 16 sig. bits	5 elements	5 elements
$\leq$ 32 sig. bits	3c+4 elements	3c+7 elements
$\leq$ 64 sig. bits	3 elements	1 elements
Total	44 elements	44 elements

Figure 2: The left table shows the prover's operations and costs involved in a Jolt proof of a program involving m steps. For non-trivial programs (i.e, those involving more than a 10 million steps), the bottleneck is usually the cost of committing to witness elements. The right table breaks down this cost in terms of the number significant elements when using Lasso with parameter $c = 6$. The key takeaway is that the Prover doesn't commit to any elements larger than 64 bits.

References

[BSCG+13] Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin, Eran Tromer, and Madars Virza. SNARKs for C: Verifying program executions succinctly and in zero knowledge. In *Proceedings of the International Cryptology Conference (CRYPTO)*, August 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.

[lib] libfennel. Hyrax reference implementation. `https://github.com/hyraxZK/fennel`.

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

[WSR+15] Riad S. Wahby, Srinath Setty, Zuocheng Ren, Andrew J. Blumberg, and Michael Walfish. Efficient RAM and control flow in verifiable outsourced computation. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2015.

[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.

Additional Material

A Quick Reference for Significant bits and elements

1 sig bit: 14 elements

- These are the 14 `op_flags` that are proved to be a part of `instr_packed` and used throughout the circuit.

5 sig bits: 3 elements

- `(rd, rs1, rs2)` are the register values that are proved to be a part of `instr_packed`.

7 sig bits: 1 element

- `op_code`, which is used in constructing the lookup index in `lookup_query` below.
- 7 bits is sufficient to identify 128 unique instructions (and virtual instructions).

14 sig bits: 1 element

- `op_flags_packed`, which is the 14 flags packed into one.
- AA: This can be avoided as it's a combination of other committed bits.

20 sig bits: 1 element

- `imm` can be up to 20 bits long (the maximum is when used as offset in jump instructions).
- Instructions may sign-extend or use in constraints as required.

32 sig bits: 4 elements (3 non-zero)

- These are the four timestamps: `rs1_read_ts, rs2_read_ts, code_read_t, mem_read_ts`.
- But at least one of them will be 0 in a given round.

50 sig bits: 1 element

- This is the field element representing the instruction that is read from the read-only program code.
- It stores the `opcode || op_flags || rs1 || rs2 || rd || imm`, which are 7 + 14 + 3 x 5 + 12 bits long.
- AA: This cannot be avoided as we effectively range-check the constituent elements by performing lookups on the corresponding bits to get each element.

64 sig bits: 4 elements (3 non-zero)

- Three are the register and memory values: `rs1_val, rs2_val, mem_read_val`.
- One of the above will be zero for any given step.
- And `lookup_output` is the entry stored in the table at the lookup index.

From SuperLasso: all 3c elements are under 32 bits

- c are counts, so between 0 and $\log m$.
- c are identifiers of cells being read, so 0 and $\log(\texttt{memory\_size} - 1)$.
- c are actual values read by read operations which is under 32 bits for the decomposed tables.

B Floating Point Instructions

RISC-V follows the IEEE 754 floating point standard. This introduces 32 new registers dedicated for floating point values. These registers are 32 bits wide (even when the other integer registers are 64 bits wide). The IEEE 754 format stores floats as follows: `[1 sign bit || 8 exponent bits || 23 mantissa bits]` where the binary representation of the number is $(-1)^s \cdot (1.m) \times 2^{e-127}$.

In our ZKVM, we tweak this format slightly:

```
[1 sign bit || 8 exponent bits || 25 zeros || 23 mantissa bits]
```

This representation is of 48 bits in length. Why? Looking ahead, this lets us perform FMUL without the need for pseudo-instructions. Note that this tweak is only cosmetic and handled within the ZKVM and does not actually change the true processor.

TODO: Add support for flag bits which are easy

There are 32 float instructions, which we classify as follows:

- Load/Store (2): FLW, FSW
- Move (4): FMV.[X/W].[W/X]
- Sign Injection (3): FSGNJX, FSGNJ, JSGNJN
- Comparison (5): FMIN, FMAX, FEQ, FLT, FLE
- Conversion int to float (4): FCVT.[W/L[U]].S
- Conversion float to int (4): FCVT.S.[W/L[U]]
- Arithmetic (5): FADD, FSUB, FMUL, FDIV, FSQRT
- Compound arithmetic (4): FMADD, FMSUB, FNMSUB, FMNADD
- Classify (1): FCLASS.S

Doubles But wait there's more. We can do doubles with the exact same methods but one change: FMUL.D costs 3 pseudo-instructions now.

B.1 MLE Tables

Given two inputs x, y (which are of the 48-bit format above), we'll construct the output floating point z by specifying z_s, z_{exp}, z_{mant}, which are routed to the corresponding bit ranges.

Floating Point Loads, Stores and Moves These load/store floating point values from/to memory. These are easier than the integer loads/stores as we do not need to perform any bit cutting or sign extensions. However, when using the tweaked format, we'll have to re-route the bits to insert 25 zeros. Moves just copy bits as is and are implemented using the identity lookup.

Sign Injection These store the exponent and mantissa of x into rd but with the sign bit being:

- FSGNJ: the sign of y
- FSGNJN: the negation of the sign of y
- FSGNJX: the xor of the signs of x and y

OP	INPUT	MLE
FSGNJ	$x \parallel y$	$z_{exp} = x_{exp}; \quad z_{mant} = x_{mant}; \quad z_s = y_s;$
FSGNJN	$x \parallel y$	$z_{exp} = x_{exp}; \quad z_{mant} = x_{mant}; \quad z_s = 1 - y_s;$
FSGNJX	$x \parallel y$	$z_s = \mathsf{XOR}(x_s, y_s); z_{exp} = x_{exp}; \quad z_{mant} = x_{mant}; \quad z_s = \mathsf{XOR}(x_s, y_s);$

Comparisons The operations are self-explanatory. There are no 2s complement involved in floating points. To compare, we (1) test the sign bit, (2) if signs are equal, then test the exponents, and (3) if that's also equal, test the mantissa.

For `FMIN`, `FMAX`, the lookup outputs the binary value denoting which operand to select. The selection is made in the circuit efficiently.

OP	INPUT	MLE
FLT	$x \parallel y$	$x_s \cdot (1 - y_s)$ $+\widetilde{eq}(x_s, y_s) \cdot \mathsf{LTU}(x_{exp}, y_{exp})$ $+\widetilde{eq}(x_{x\parallel exp}, y_{x\parallel exp}) \cdot \mathsf{LTU}(x_{mant}, y_{mant})$
FLE	$x \parallel y$	$\widetilde{eq}(x, y) + \mathsf{FLT}(x, y)$
FMIN	$x \parallel y$	$\mathsf{FLT}(x, y)$ `// Binary choice made in circuit`
FMAX	$x \parallel y$	$1 - \mathsf{FLT}(x, y)$ `// Binary choice made in circuit`

Conversion from Float to Int Converting floats to integers are effectively like shifts as we just need to move the mantissa into the right bit position. Although there are 127 possible exponents we only care about floatings having exponents in the range $[0, 63 + 23]$ as the rest either cannot be represented as a 64-bit integer or we do not have enough significant bits of information to care. TODO: in which case we flag

- `FCVT.LU.S` converts a float to a 64-bit unsigned number.
- `FCVT.L.S` converts converts to signed number and thus required taking a 2s complement of the result. Taking 2s complement might result in an overflow into bit 65 which needs to be taken care of using a second pseudo-instruction.
- There are W versions of these that convert to 32-bit numbers. These are handled analagously by replacing 63 with 31. AA: Or we just give a generic L formula.

AA: We gotta explain this earlier in a list of "commonly used MLEs". In the following, $\mathsf{SHIFT}_{\mathsf{R}}(x, len)$ denotes taking input $\sum_i 2^i \cdot x_i$ and outputting $\sum_i 2^i \cdot x_{i+len}$ (respecting the edges).

OP	INPUT	MLE
FCVT.LU.S	x	$\sum_{i=0}^{23} \widetilde{eq}(i, x_{exp}) \cdot \mathsf{SHIFT}_{\mathsf{R}}(1 \parallel x_{mant}, 23 - i)$ $+ \sum_{i=24}^{23+63} \widetilde{eq}(i, x_{exp}) \cdot \mathsf{SHIFT}_{\mathsf{L}}(1 \parallel x_{mant}, i - 23)$
FCVT.L.S_OF	$x \parallel y$	$\sum_{i=0}^{63} 2^i - \mathsf{FCVT.LU.S}(x, y) + 1$
FCVT.L.S	$x \parallel y$	1. FCVT.L.S_OF $rs1, rs2, rd$ 2. NEGATE_OF rd

Conversion from Int to Float These involve figuring out the exponents and then shifting the mantissa to keep just 23 significant digits. The exponents are formed by finding the prefix of integer x is all 0s. Then, from that index, the next 23 significant digits (except the leading 1) become the mantissa.

The signed case is harder because we may need to take the 2s complement of the input when it is negative. We thus do this in 3 steps: the first takes the 2s complement if necessary (forming a 65-bit integer with overflow), the second does the unsigned conversion, and the third injects the sign from the original integer.

OP	INPUT	MLE
FCVT.S.LU	x	• $z_s = 0$; • $z_{exp} = \sum_{i=0}^{64} \widetilde{eq}(x_{>i}, 0) \cdot x_i \cdot \mathsf{binary}(63 - i)$ `find largest prefix of 0s` • $z_{mant} = \sum_{i=0}^{64} \widetilde{eq}(x_{>i}, 0) \cdot x_i \cdot \left(\sum_{j=0}^{23} 2^j x_{i+j} \right)$
FCVT.S.L	x	1. NEGATE_IF_NEG $x, v1$ `// take 2s complement if negative` 2. FCVT.S.LU $v1, rd$ 3. FSGNJ_INT rd, x, rd `// sign inject from original integer`

B.2 Floating Point Arithmetic

We'll perform MUL in one step but will require pseudo-instructions for the others.

FADD The core condition here is that if the two exponents differ by more than 23, then we can simply ignore the smaller of the two operands as it will simply not affect the 23 significant bits of the larger operand.

We do this in 5 steps, each having a pseudo-instruction. The first two steps move the larger and smaller of the two values into two separate virtual registers. We then "denormalize" the smaller value to match the exponent of the larger value, by running over all exponent values in the range $[e_1, e_1 - 23]$. If the exponent is even smaller, than it is simply ignored and the denormalized value is 0.

We then add the denormalized value to the larger operand. We add or subtract based on the signs. We may overflow or underflow so we'll adjust for that in the 5th step, FNORM. As we know that either the 25th, 24th or 23rd bit has to be the MSB, we use it to figure out the next 23 significant bits of the mantissa.

OP	INPUT	MLE
FADD	$x \parallel y$	1. FMAX_ABS x, y, v_{max} 2. FMIN_ABS x, y, v_{min} 3. FDENORM $v_{max}, v_{min}, v1$ 4. FADD_DENORM $v_{max}, v1, v2$ 5. FNORM $v2$
FDENORM	$x \parallel y$	$\sum_{i=0}^{23} \widetilde{eq}(e_1, \mathsf{ADD}(e_2, i)) \cdot \mathsf{SHIFTR}(2^{23} + y_{mantissa}, i)$
FADD_DENORM	$x \parallel y$	$z_s = x_z$ $z_{exp} = x_{exp}$ $z_{mantissa} = 2^{24} + x_{mantissa} + (2 \cdot EQ(x_{sign}, y_{sign}) - 1) \cdot y_{mantissa}$
FNORM	$x \parallel y$	$z_s = x_s$ $z_{exp} = x_{exp} + (1 - x_{24}) \cdot x_{23} - (1 - x_{24}) \cdot (1 - x_{23})$ $z_{mantissa} = x_{24} \cdot x_{[23...1]} + (1 - x_{24}) \cdot x_{23} \cdot x_{[22...0]} + (1 - x_{24}) \cdot (1 - x_{23}) \cdot x_{[21...0]}$

FSUB Perform FADD but negate the sign of y when copying to the destination reigsters in the the first two FMIN, FMAX steps. Call these instructions FMAX_ABS_N, FMIN_ABS_N, respectively.

FMUL Like in integer multiplication, we'll perform the actual multiplication in the circuit and use the lookup to select the desired bits. Here, in the circuit we compute:

- $w = (2^{23} + x) \times (2^{23} + y)$
- $u = x \times 2^9 + y$

The first puts the 48 bit value that is the product of the mantissas (with the implied 1 bit prepended) as the lower bits of w, which is now totally $48 + 9 * 2 = 66$ bits long. The second puts the 9 sign and exp bits of x in front of the corresponding bits of y, forming a $57 + 9 = 66$ bit value. Send $[u \parallel w]$ to the lookup. After the opcode, this is $66 + 66 = 132$ bits.

There are two cases depending on the value of w_{47} (the 48th bit of the mantissa of w):

If $w_{47} = 0$ then:

- $z_{exp} = x_{exp} + y_{exp}$
- $z_{mantissa}$ = 23 bits of w starting from w_{45}

If $w_{47} = 1$ then: that means we bump up the exponent:

- $z_{exp} = x_{exp} + y_{exp} + 1$
- $z_{mantissa}$ = 23 bits of w starting from w_{46}

AA: need to talk about flags here.

OP	INPUT	MLE
FMUL_NOFLAG	$w \parallel u$	$z_s = \mathsf{XOR}(x_s, y_s)$ `// obtained from w` $z_{exp} = x_{exp} + y_{exp} + w_{47}$ `//` w_{47} `determines the overflow` $z_{mant} = (1 - w_{47}) \times w_{45\ldots(45-23)} + w_{47} \times w_{46\ldots(46-23)}$
FMUL	$x \parallel y$	1. FMUL_NOFLAG $rs1, rs2, rd$ 2. F_CHECK_FLAG rd

FDIV This is derived from FMUL by storing the quotient q and remainder r as advice into a virtual register and then performing FMUL and FADD to verify correctness.

OP	INPUT	MLE
FMUL	$x \parallel y$	1. FADVICE v_q 2. FADVICE v_r 3. FASSERT_LT_ABS v_r, v_y 2. FMUL y, v_q, v_3 2. FADD v_3, r, rd

FSQRT Similar to division, we provide the root r and an error value e because x might not have a perfect square root or at least, one representable in single-precision float.

We use a simple pseudo-instruction `FADD_ERROR` that ignores the sign and exp bits and just adds the last $(23 - K)$ bits of the mantissas, where K is the desired precision. (If $K = 10$ then we're saying that the closest square to x is $x(1 \pm 1/2^{10})$). Note that this addition may overflow and bump up the exponent (as with FADD). And if x is negative then the FASSERT_EQ step will fail.

OP	**INPUT**	**MLE**
FMUL	$x \parallel y$	1. FADVICE v_r 2. FADVICE v_e 2. FMUL v_r, v_r, v_1 2. FADD_ERROR v_1, v_e, v_2 2. ASSERT_FEQ v_2, x

TODO: Might need another check flag.