Jolt: SNARKs for CPU Abstractions 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 targeting a specific instruction-set architecture (ISA).

A *front-end* converts computer programs into a lower-level representation, called an *intermediate representation* (IR), which is typically a generalization of arithmetic circuit satisfiability. 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. The circuits output by Jolt primarily perform lookups into a gigantic lookup table, of size over 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), which is the first lookup argument to avoid costs that grow linearly with the table size.

We describe performance and auditability benefits of Jolt compared to prior zkVMs. As a concrete example, we focus on the popular RISC-V instruction set architecture. The dominant cost for the Jolt prover applied to this ISA, when using an appropriate cryptographic commitment scheme, is cryptographically committing to about four 256-bit field elements per step of the RISC-V CPU.

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 on a witness. In the example above, the computer program might take as input a list of blockchain transactions and associated digital signatures authorizing each of them, and verify that each of the signatures is valid.

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

Many projects today accomplish this via a CPU abstraction (in this context, also often called a *Virtual Machine (VM)* abstraction). 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).

Other so-called zkVM projects take a similar approach but don't target the EVM instruction set. These projects typically choose (or design) ISAs for their "SNARK-friendliness" or for surrounding infrastructure and tooling. For example, Cairo specifies a very simple virtual machine designed specifically for compatibility with SNARK proving. The Cairo 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. 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" 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 typically implement these tasks by carefully devising gates or so-called constraints (often by hand) that implement each instruction. This is tedious, time-intensive, and 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. Generally speaking, SNARKs supporting a virtual machine abstraction lead to develop-friendly tooling at the potential cost of performance overheads. This is loosely analogous to real computer hardware, where CPUs are pervasive general-purpose computing devices, though they may be slower in many contexts than application-specific hardware. But the application-specific hardware takes considerable time and expense to manufacture, for any given application.

Example benefits of zkVMs include the following. Supporting a sufficiently rich ISA such as RISC-V or EVM allows the tooling to leverage existing compilers from existing high-level languages down to the ISA. Relatedly, the benefits (and downsides) of bytecode-level equivalence with the Ethereum Virtual Machine have been discussed (or, more accurately, debated) at length in the blockchain community. 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 1.2 of other front-end approaches).

[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 has write-once memory and does not offer signed integer data types.

On the other hand, circuits implementing a VM can be much larger than circuits that don't. Intuitively, a VM must figure out, at each step of the computation, which primitive instruction to execute, and be prepared to execute any one of them. Alternative approaches can potentially avoid including such logic in the circuit, at least for some programs.

Conversely still, the circuits implementing VMs are uniform, meaning they contain repeated structure (the circuit merely repeatedly executes the Virtual Machine's transition function). SNARKs for uniform circuits avoid the need for an honest party to pre-process the circuit, and require less cryptographic overhead for the prover than do SNARKs targeted at non-uniform circuits of the same size.

AA: But why is this a popular approach in the first place? Why move from circuits? Reasons: uniformity of tools, avoid to re-write computations in a niche DSL.

The conventional wisdom on zkVMs. The general 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 Cairo.

However, this comes at a cost, because primitive operations that are standard in "real-world" CPUs will cost many primitive instructions to implement on the simple VM. To partially compensate for this, many projects have designed restricted domain specific languages (DSLs) that are exposed to the programmer who writes the witness-checking program. The DSL typically exposes to the programmer operations that can be transformed into a relatively small number of primitive instructions in the ISA. This places a burden on the programmer, who must learn the new DSL, which often involves programming in unfamiliar ways. Moreover, the programmer is responsible for writing correct programs in the unfamiliar DSL, with potentially catastrophic security consequences if a program is incorrect.

A new perspective. In this work, we at least partially upend the notion that simpler instruction sets necessarily lead to smaller circuits and associated faster provers. As we explain shortly, we give techniques that transform VMs into equivalent circuits, where the size of the circuit 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, which we call *decomposability*.

A primary contribution of our work is to show that decomposability is satisfied by instruction sets that are substantially more complicated than those currently handled by several existing projects. We describe an associated front-end, Jolt, that outputs circuits that are substantially smaller per step of the computation than prior front-ends for VM abstractions.

To be more precise, our Jolt front-end does not truly spit out "circuits" in the traditional sense, as it makes heavy use of so-called *lookup arguments*, described below.

rewrite the below When we refer to the "size of the circuits" output by Jolt, what we really refer to is the *number of field elements that must be cryptographically committed by the prover*. This is the runtime bottleneck for the prover when a back-end for circuit-satisfiability (more precisely, a generalization called *rank-one constraint satisfaction*, or R1CS) is combined with the Jolt front-end and the particular lookup argument used by Jolt (which is called Lasso and introduced in a companion paper [?]).

In almost all SNARKs for circuit-satisfiability or generalizations thereof (the one exception being those based on the so-called GKR interactive proof [?, ?, lib, ?, ?]), the prover has to cryptographically commit to *at least* one field element per gate in the circuit. In some SNARKs, such as Plonk [?], Marlin [?], and Groth16 [?], it is substantially more than one. For example, in Plonk, the prover commits to 11 field elements per circuit gate. This is why "number of field elements committed" by the Jolt prover is a reasonable proxy for "number of gates in a circuit" output by other front-ends: the prover time in these other front-end/back-end combinations is *at least* that required to commit to one field element per gate in the circuit, and often *substantially* higher than that.

Lookup arguments. TODO: Introduce what lookup arguments are

1.2 The lookup singularity

Barry Whitehat has explicitly articulated a goal of designing front-ends that produce circuits that *only* perform lookups. Whitehat terms this the *lookup singularity* [Whi] 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. In particular, current lookup arguments either require prover time linear in the table size,

The discussion has identified a key bottleneck in existing techniques: the inability to use very large tables, due to the need of the prover (or in some lookup arguments, the verifier as well) to cryptographically commit to the entire vector of table contents. Fortunately, in a companion paper called Lasso, we have introduced a lookup argument that does not suffer from this bottleneck. So long as the lookup table is modestly "structured" (meaning, in particular, that its contents have a simpler description than simply listing all table elements in full), neither the prover nor verifier in Lasso needs to materialize the full table.

The bottleneck for the Lasso prover is cryptographically committing to some number of field elements per lookup. The number of field elements per lookup that need to be committed depends on the size N of the lookup table. For m lookups into a table of size N and some parameter $c > 0$, roughly $3cm + N^{1/c}$ field elements need to be committed in total. This means about $3c$ field elements per lookup, where c has to be chosen large enough that the $N^{1/c}$ term is not a bottleneck.

In this work, we show that Lasso unlocks the lookup singularity in front-end design, or at least is a major step in the right direction. Specifically, we focus on the RISC-V instruction set (described above and featuring prominently in the RISC-Zero project). For each of the RISC-V instructions f_i, our idea is to create a lookup table that contains the entire evaluation table of f_i. So if f_i takes two 32-bit data types as input, the table will have 2^{64} entries. The (x, y)'th entry is $f_i(x, y)$. The tables for multiple instructions can easily be "glued together" into a single giant table, hence the name Jolt (Just One Lookup Table).

We work through each of the RISC-V instructions, including extensions for multiplication, division, and remainder, and show that the resulting lookup table for each has the structure required by Lasso to ensure fast proving and verification.

update this shit based on most of the values being small For 64-bit data types, JOLT uses a lookup table of size about $N = 2^{140}$, which means that we'll choose c to be about 7, so that $N^{1/c}$ is about 1 million. This means the prover has to commit to about $3c = 21$ field elements per lookup. For 32-bit data types, c can be as small as two, meaning only about six field elements per invocation of the instruction need to be committed by the prover. This is because we show that, for some instructions that operate on two b-bit inputs, we can get the lookup table size down to 2^{b+1} rather than 2^{2b}.

The circuit that Jolt produces includes some logic besides lookups, specifically, to implement reading to and writing from memory, and the task of "deducing" which operands should be fed as inputs to the lookup. All told, the number of field elements (i.e., size of the polynomial) that the Jolt prover has to commit to is roughly 45 per step TODO: double-check this in final version of the RISC-V CPU when operating on 64-bit data types. When adding support for 32-bit floating point operations, this increases to 50. This is vastly faster than prior floating point implementations in the literature [?, ?]

1.3 Other comparisons to prior works

RISC-Zero today supports RISC-V instructions on 32-bit data types. While the project's codebase is not fully open source it appears that the prover has to commit to at least 212 field elements per step of the RISC-V CPU. Earlier work of Cairo [?] had the prover commit to well over 50 field elements per step of a much simpler VM.

One possible downside of our approach is that we must work over relatively large fields. TODO: Include discussion about significant bits.

For instructions operating on two b-bit inputs and producing one b-bit output, our use of Lasso requires working over a field of characteristic at least 2^{3b}. For example, to handle two 64-bit inputs that produce a 64-bit output, our field size has to be at least 2^{192}. For 32-bit data types, this would fall to about 2^{96}. Still, projects such as RISC-Zero and Polygon Hermez and Miden and Plonky2 seek to work over (extensions of) 64-bit fields[2]

However, big fields are required regardless of whether one wishes to use elliptic-curve-based commitment schemes. huh? preceding sentence is wrong Moreover, many statements that are bottlenecks in applications today, including the earlier example of proving knowledge of (elliptic-curve-based) digital signatures, are statements are large fields, and hence are most naturally and efficiently proven over large fields.[3]

Other front-end approaches. As with Cairo JT: and RISC-Zero? Miden?, 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 approaches produce a different circuit for every computer program, such as Buffet, Bellman, Circom, 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.

1.4 Further discussion: R1CS strikes back, and reductions in developer time?

There has been tremendous attention and effort by SNARK designers devoted to designing compact circuits (typically by hand) for specific applications, in a format called *Plonkish arithmetization* or *custom constraints*. Roughly speaking, this refers to a generalization of arithmetic circuits that allows for "gates" that apply a polynomial of degree more than two to their inputs and insist that the result is zero. An earlier generalization of arithmetic circuits called *rank-1 constraint satisfaction* permits only degree-2 gates (in fact, a subclass of degree-2 constraints, called *rank-one* constraints). See the recent paper by Setty, Thaler, and Wahby [?] for an overview of these intermediate representations, and an explanation of how most SNARKs (with the notable exception of Groth16 [?] and its predecessors) can support all of them (and in fact generalizations thereof).

Interestingly, Jolt produces R1CS. Conceptually, all of the "high-degreeness" in the "checks" that it utilizes are encapsulated in the lookup argument. The checks that remain are all degree-2 (in fact, rank-one). This suggests a possible "comeback" for R1CS, and tooling targeted at that IR.

is the below ready for prime time if there's not a complete implementation?

Besides improved prover performance, a substantial possible benefit of Jolt over prior work is improvements in the human effort required to develop, maintain, and update the system. Our Jolt front-end consists entirely of about 250 lines of Zokrates code written over the course of a couple of days. Zokrates outputs R1CS, which can be fed through any number of back-end implementations that support that IR (we use Spartan [?]), combined with any implementation of the lookup argument Lasso. In contrast, to our knowledge, projects that have taken advantage of "custom constraints" typically write those constraints by hand, with substantial investments of developer time [?, ?, ?].[4]

[2]They do so at the cost of provable security, relying on unproven conjectures about the statistical soundness of FRI [?] when run interactively, before applying the Fiat-Shamir transformation to render it non-interactive.

[3]E.g., StarkEx [?] works over a 251-bit field that matches the base field of the elliptic curve used in its ECDSA signatures `https://docs.starkware.co/starkex/crypto/stark-curve.html`.

[4]For maximum performance in Lasso's application of the sum-check protocol, the Lasso prover implementation should be specialized to the specific lookup tables used by Jolt to represent each RISC-V instruction, and this tailoring does currently require expertise. However, Lasso does describe some more generic prover implementations that are somewhat less performative (specifically, the prover does logarithmically rather than $O(1)$ field operations per lookup). We do not believe that these field operations will be a bottleneck for the prover even if the slower, more generic implementation is used. maybe we can confirm this experimentally

talk about how a major impediment to the widespread adoption of SNARKs in security-critical applications is that no one will trust the 64,000 lines of code implementing them, many of which is implementing circuits, any error in which will bust security

1.5 Technical details: CPU instructions as structured polynomials

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

To take advantage of Lasso, we must show that the evaluation table of each RISC-V instruction has particular structure. In particular, we have to identify a so-called *low-degree extension polynomial* of the evaluation table that the verifier can evaluate in logarithmic time.

The efficiency of the whole framework hinges on this being feasible.

- We show that this is indeed possible for all instructions required by RISC-V.
- This means that each instruction should be represented as an MLE of the bits of its inputs. That is, if the inputs are two `XLEN` -bit numbers x, y which are read from the registers, the output of the instruction should be an MLE over the boolean values $(x_0, \ldots, x_{\texttt{XLEN}-1}, y_0, \ldots, y_{\texttt{XLEN}-1})$.
- This turns out to be very convenient for instructions like SHIFTs that require bit manipulation. However, we have to support shifts of variable lengths.
- The bigger challenge turns out to be arithmetic and comparison instructions, which require the output to be a specific number of bits. For example, most ADD instructions take two `XLEN` -bit numbers but must negate the overflow into the `XLEN` th bit. And most processor's MUL instructions take two `XLEN` -bit numbers and choosing either the higher or lowest `XLEN` bits.
- For comparisons, we need to take the appropriate function (say, less than) and represent the final 0/1 value as a MLE of the bits.

1.6 Related Work

The approach of producing proofs over all programs for a given processor has been explored in the TinyRAM works of [BSCG$^+$13] and [BSCTV14], Buffet [WSR$^+$15] and vRAM [ZGK$^+$18]. These works take different approaches to emulating a CPU's instruction and memory. None of these works were done on a real-world processor as they all involve weakening the CPU architecture for efficiency.

risc0 is a popular (?) project also building a ZKVM for the RISC-V processor. Though they have no paper, whitepaper, specifics of their protocol, or performance details, we'll try our best to represent their work faithfully. AA: Considering this version isn't a final conference submission, we could just ignore risc0 now and then if they respond with actual details we can incorporate it. risc0 has a whitepaper now, and ingonyama put out something with concrete costs on small benchmarks

zkEVMs sound similar in approach but they all tailor circuit to specific smart contracts.

Limitations on instruction sets No current approach to a ZKVM works on a real-world RAM.

- Pixar has made four movies about TinyRAM.
- Even on a toy processor, [BSCG$^+$13] and [BSCTV14] achieve really poor performance.
- Need to comment on Buffet's and vRAM's performance.

1.7 Our contributions

- We define a framework for verifying general CPUs efficiently using lookup arguments.

Work	**Per-program Setup**	**Processor**	**Prover cost**	**Verification cost**
[BSCG+13]	None	TinyRAM		
[BSCTV14]				
Buffet				
vRAM				
risc0			240 C	
(This work)	None		46 C + O(100) F	

Table 1: AA: We could have some qualitative comparison like this at the start

- We detail how this framework can be applied to RISC-V, giving a proof system that emualates all basic RISC-V instructions on a 64-bit processsor, along with the M extension for multiplication and F extension for floating points.
- We specify the qualitative costs of this approach, showing that the amount of work done by the prover per step of the computation is extremely small.
- (Coming soon: Implementation of Jolt + SparkPlug)

2 Background

2.1 Lookup Arguments

2.2 Offline Memory Checking

2.3 CPU Architecture

3 An Overview of CPU Architectures

This section provides an overview of the architecture followed in this work. It consists of a CPU and a read-write memory, collectively called the machine.

Definition 3.1 (Machine State)**.** *The machine state consists of: {Integer registers, Program Counter, Memory}. There are 32 integer registers, each of `XLEN` bits, where `XLEN` is 32 or 64. The `PC`, also of `XLEN` bits, is a separate register that stores the address of the next instruction to be executed.*

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`. Unless specified by the instruction, the `PC` is advanced to the next memory location after executing the instruction.

Each instruction specifies an operation, at most two source registers `rs1, rs2`, a destination register `rd` and a constant value `imm` (for immediate). 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, write to memory.

Definition 3.2 (5-tuple RISC-V Instruction Format)**.** *RISC-V instructions are specified in the following format: `(opcode, rs1, rs2, rd, imm]`.*

CPU Transition Function: The CPU step does the following, as depicted in Figure 1.

1. Read the instruction at location PC.
2. Parse instruction as `[opcode || rs1 || rs2 || rd || imm]`.
   (a) Read the values at registers `rs1, rs2`.
   (b) For load instrutions, read the address specified by `rs1, rs2, imm`.

   (c) For store instructions, write to an address, where both the the value and location are specified by `rs1, rs2, imm`.

   (d) Run the appropriate functionality to get output `out`.

3. Update the PC: the PC is usually just incremented to the next memory location, but some instructions (such as branches, jumps) update the value based on `out`.

4. Update rd: this usually means writing the value `out`.

TODO: Talk about flags and status registers.

3.1 Designing CPU Circuits with Lookups

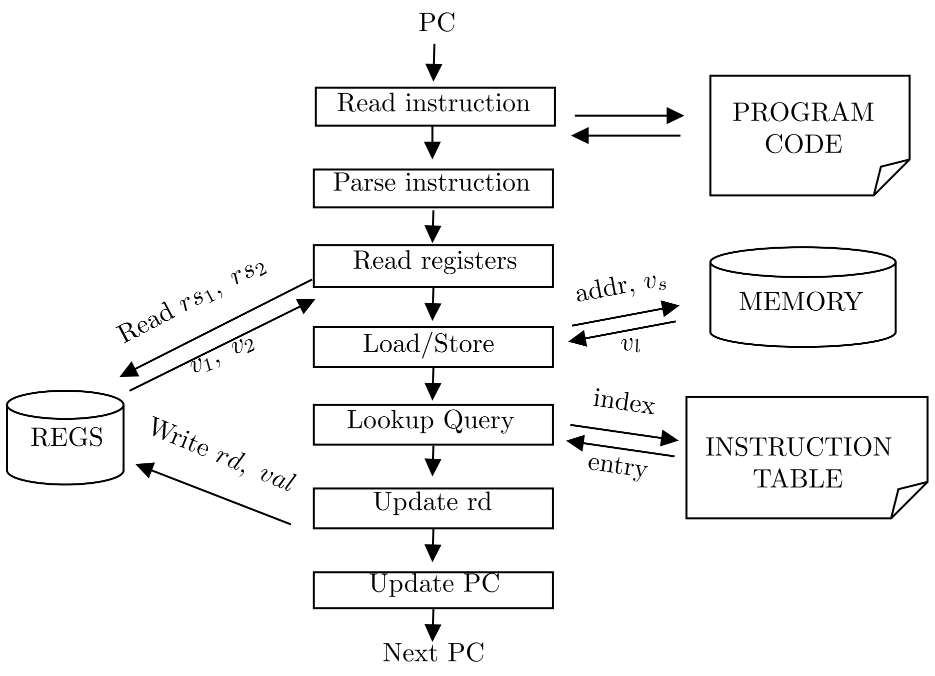

Figure 1: Draft of a diagram

4 Two Tools

4.1 Tool 1: Offline Memory Checking

4.2 Tool 2: Lookups

4.2.1 Decomposition of MLEs

5 RISC-V as one table

5.1 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)`. The "pre-processing" of assembly instructions into this representation is done before the proof starts and Jolt only reads instructions in the above 5-tuple form. Note that this transformation is purely syntactic 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). While pre-processing the assembly code, Jolt performs two types of pre-processing to the immediates depending on the instruction involved: (1) They can be sign-extended or zero-extended into 64-bits, or (2) The absolute immediate value in $[-2^{20}, 2^{20}-1]$ is converted to the corresponding element in the native field, $\mathbb{F}_p$.

The first is straightforward and prescribed by RISC-V if needed for an instruction. The second is a convenience employed by Jolt for instructions (namely, jumps and branches) where the immediate is treated as an address offset added to the PC. Here, an immediate value $-|v|$ is fed to Jolt as $p-|v|$ so arithmetic using the instruction can be represented as constraints in the native field. We note again that the exact type of processing to be done is deterministic for any given instruction independent of the rest of the program.

5.2 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 as `imm` is always at most that length and a core guarantee of Jolt is that the values stored in the ZKVM's registers are in the right range (at most L bits long).

Notation:

- We refer to the binary coefficients of a k-bit number z as $[z_{k-1}, \ldots, z_0]$ from MSB to LSB such that $z = \sum_{i=0}^{k-1} 2^i z_i$.
- We use $x_{<i}$ to refer to the subsequence $[x_{i-1}, \ldots, x_0]$. Analagously, $x_{>i}$ refers to the subsequence $[x_{L-1}, \ldots, x_{i+1}]$.
- For 2's complement signed numbers, we use $x_s = x_{L-1}$ to refer to the sign bit. And use $x_{<s}$ to refer to the subsequence of all coefficients but the sign bit.
- We use $x \parallel y$ to refer to the number whose binary representation is the concatenation $[x_{L-1}, \ldots, x_0 \parallel y_{L-1}, , \ldots, y_0]$. This number is obtained as $x \cdot 2^L + y$.
- **Chunks:** We split the range of bits $[0, \ldots L-1]$ into c segments the natural way and refer to them as $I_1, \ldots, I_c$.

5.3 Expressing Instructions as Decomposed MLEs

Using Lasso for a ZKVM requires representing CPU instructions in a lookup table in an amenable format. Jolt does this by designing a "truth table" for each instruction that lists as entries every possible combination of operands and that combination's corresponding output. As explain in Section 4.2.1, this truth table's entries should be expressable as a "decomposition" of MLEs. The remainder of this does exactly this for all RISC-V instructions in the base integer instruction set and the multiplication extension for a 64-bit architecture.

As a first step, we notice that we can express each instruction's lookup table as a single MLE. Using this as a starting point, we show how to decompose the table into a simple function of smaller MLEs, as required. We highlight this process using three particular instructions, which act as helper functions in building the tables of other instructions: equality ($\widetilde{eq}$), Set Less than (SLT) and Left Shift (SHIFT). For each of these instructions, we first represent the lookup tables of the instructions as a single MLE, as required by Lasso. Then, we then break this up to represent the table as a simple combination of MLEs over smaller tables that read different subsets of the input, as required by SuperLasso.

5.4 Three Instructive MLEs

5.4.1 The equality MLE

The equality MLE is used in the construction of tables for many instructions. This MLE takes two vectors of bits of identical length and outputs 1 if they are the same vector and 0 if not.

$$\widetilde{eq}(x_{<i}, y_{<i}) = \prod_{j=0}^{i-1} (x_j y_j + (1 - x_j)(1 - y_j)) = \prod_{j=0}^{i-1} (1 - x_j - y_j + 2x_j y_j)$$

TODO: We can add the note about equality being doable in linear-time here.

Decomposing $\widetilde{eq}$: This can be done in the straightforward way by splitting the input into c sections, having each table store the output $\widetilde{eq}$ on its section, and taking a product of these outputs to get the final table entry. Precisely, table T_j takes as input the j^{th} section of x and y (with set of indices I_j) and has its entries represented by the following MLE: $T_j[x_{I_j} \parallel y_{I_j}] = \prod_{i \in I_j} (1 - x_i - y_i + 2x_i y_i)$.The final table entry is $T[x \parallel y] = \prod_{i=1}^{c} T_j[x_{I_j} \parallel y_{I_j}]$. It can be seen by inspection that this value is indeed $\widetilde{eq}$.

5.4.2 Set Less Than - Unsigned

We first represent the instruction as a complete MLE, before decomposing it into smaller MLEs. The Set Less Than - Unsigned (`SLTU`) instruction takes two L-bit unsigned inputs x and y, and stores 1 in `rd` when $x < y$ and stores 0 otherwise. The lookup table for `SLTU` has a row for each possible index $x \parallel y$ with the entry being the boolean result of the unsigned comparison $x < y$.

We construct the MLE for the table in two stages. First, consider the following MLE taking two L-bit values as input: $\mathsf{LT}_i(x, y) = (1 - x_i) \cdot y_i \cdot \widetilde{eq}(x_{>i}, y_{>i})$. By inspection, we observe:

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

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

The desired MLE representing the strict less than comparison is thus:

$$\mathsf{LTU}(x, y) = \sum_{i=0}^{L-1} \mathsf{LT}_i(x, y)$$

Decompositing LT: Representing the above MLE for LT as smaller MLEs is tricky as each variable x_i is a part of terms containing long ranges of values due to the $\widetilde{eq}$ term. Thus, no term of the MLE can be computed with only a narrow subset of inputs. While keeping the core of the above LT MLE, we design a way to decompose it using a combination of $2c$ smaller MLEs, $\{T_1^l, \ldots, T_c^l, T_1^e, \ldots, T_c^e\}$.

Tables T_j^l and T_j^e take the j^{th} section of x and y as input.

- The entries of T_j^l are represented by the LTU MLE: let $l_j = T_j^l[x_{I_j} \parallel y_{I_j}] = \mathsf{LTU}(x_{I_j}, y_{I_j})$.
- The entries of T_j^e are represented by the $\widetilde{eq}$ MLE: let $e_j = T_j^e[x_{I_j} \parallel y_{I_j}] = \widetilde{eq}(x_{I_j}, y_{I_j})$.

The final table entry is recursively defined as $T(x \parallel y) = l_1 + e_1 \cdot T_2'$ where $T_k' = l_k + e_k \cdot T_{k+1}'$ for all $k = 1, \ldots, c-1$ and $T_c' = l_c$. TODO: Add quick proof.

5.4.3 Shift Left Logical

SLL takes L-bit integer x and shifts it to the left by length y where $y < L$. Bits shifted beyond the MSB are ignored and zeros are filled into the vacated lower bits.

The lookup table for SLL has a row for all possible $x \parallel y$ with the entry being the shifted output $x << y$ introduce this shift notation somewhere (preliminaries), point to C. We again construct this table's MLE in two stages. For a constant k, let $\texttt{SHIFT-L}_k(x) = \sum_{j=k}^{L-1} 2^j \cdot x_{j-k}$.

The desired MLE is

$$\texttt{SLL}(x, y) = \sum_{k=0}^{63} \widetilde{eq}(k, y) \cdot \texttt{SHIFT-L}_k(x).$$

Analagously, we can define

$$\texttt{SHIFT-R}_k(x) = \sum_{j=0}^{L-k-1} 2^j \cdot x_{j+k}.$$

Decomposability: We can perform shifts using smaller MLEs by having each subtable shift its respective input by a length depending on the value of y and k. Precisely, the tables $T_1, \ldots, T_{c-1}$ here each take their respective chunk of the input x and ALL of y, and have their as follows:

- For $j < c$, the tables shift their entire inputs to the left: $T_j[x_{I_j} \parallel y] = \sum_{k=0}^{63} \widetilde{eq}(y, k) \cdot 2^k \cdot x_{I_j}$
- And the last table shifts only the inputs that would remain within the range of the output: $T_c[x_{I_j} \parallel y] = \sum_{k=0}^{63} \widetilde{eq}(y, k) \cdot 2^k \cdot (\sum_{i=L-c}^{L-k} x_i)$.

The final table entry is obtained by effectively contatenating the outputs of the smaller table:]

$T[x \parallel y] = \sum_{j=0}^{c} 2^{j \cdot (L/c)} \cdot T_j[x_{I_j} \parallel y]$.

6 The Table

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.

6.1 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
AND	$x \parallel y$	$\sum_{i=0}^{L-1} 2^i \cdot (x_i \cdot y_i)$
OR	$x \parallel y$	$\sum_{i=0}^{L-1} 2^i \cdot (x_i + y_i - x_i \cdot y_i)$
XOR	$x \parallel y$	$\sum_{i=0}^{L-1} 2^i \cdot (x_i \cdot (1 - y_i) + y_i \cdot (1 - x_i))$

6.1.1 Arithmetic Instructions

ADD returns the lowest L bits of the sum $x + y$. SUB returns the lowest L bits of $x - y$, where the operands are represented in 2's complement.

As addition costs just one constraint in the circuit, we cheaply compute $z = x + y$ in the circuit. However, this sum can be $L + 1$ bits long. We use lookups to obtain the result with the overflow bit ignored. To this end, the lookup table for AND contains a row for all possible $(L + 1)$-bit z and the entry is the lower L bits. Since 2's complement subtraction can be performed using addition as $x + (\sum_{i=0}^{L-1} 2^i - y + 1)$, the lookup table for SUB is identical to AND.

OP	INDEX	MLE
ADD	$z = x + y$	$\sum_{i=0}^{L-1} 2^i z_i$ `// return lowest L bits`
SUB	$z = x + (\sum_{i=0}^{63} 2^i - y + 1)$	$\sum_{i=0}^{L-1} 2^i z_i$ `// return lowest L bits`

6.1.2 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 and MLE for SLTU were derived in Section 5.4.2. 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	$x \parallel y$	$\mathsf{LTU} := \sum_{i=0}^{L-1} \mathsf{LT}_i(x, y))$
SLT	$x \parallel y$	$\mathsf{LTS} := x_s \cdot (1 - y_s) + \widetilde{eq}(x_s, y_s) \cdot \mathsf{LTU}(x_{<s}, y_{<s})$

6.1.3 Shifts

SLL/SRL perform left/right logical shifts of x by length y, respectively, where $y < L$. SRA performs an arithmetic right of x by length y, where $y < L$. Both operations return L-bit values as bits shifted beyond the MSB and below the 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. AA: Need to check what the convention is when shift lenghts are too long

The MLEs for SHIFT were discussed in Section 5.4.3. The table's have a row for each possible $x \parallel y$ with the entry being the desired shifted value. The MLE for arithmetic shift additionally places the sign bit x_s of the input into the vacated upper bits.

OP	INDEX	MLE
SLL	$x \parallel y$	$\sum_{k=0}^{63} \widetilde{eq}(k, y_{<6}) \cdot \texttt{SHIFT-L}_k(x)$
SRL	$x \parallel y$	$\sum_{k=0}^{63} \widetilde{eq}(k, y_{<6}) \cdot \texttt{SHIFT-R}_k(x)$
SRA	$x \parallel y$	$\sum_{k=0}^{63} \widetilde{eq}(k, y_{<6}) \cdot \big(\texttt{SHIFT-L}_k(x) + \sum_{i=63-k+1}^{63} 2^i \cdot x_s\big)$ `// place sign bit`

6.1.4 Immediate Loads

`AUIPC` takes the 20-bit immediate, adds it to `PC` and stores the output in `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.

In both of these instructions, the 20-bit immediate is pre-processed (see Section 5.1) 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.

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`

6.1.5 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 5.1, 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`

6.1.6 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 5.4.2 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	$x \parallel y$	$1 - \mathsf{LTS}(x, y)$

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 not 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`

7 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.

7.1 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.

7.1.1 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`	$x \parallel y$	$\widetilde{eq}(x_s, y_s)$

7.1.2 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`

7.2 The MUL Tables

7.2.1 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`

7.2.2 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`

7.3 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 7.1.2. 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 r_x, r_y, rd	1. ADVICE $-,-,-,v_q$ // store non-deterministic advice q into v_q 2. ADVICE $-,-,-,v_r$ // store non-deterministic advice r into v_r 3. MULU $v_q, r_y, -, v_{qy}$ // compute $q \times y$ 4. ASSERT_LTU $v_r, r_y, -, -$ // verify that $r < y$ 5. ASSERT_LTE $v_{qy}, r_x, -, -$ // assert $q \times y \leq x$ 6. ADD $v_{qy}, v_r, -, v_0$ // compute $q \times y + r$ 7. ASSERT_EQ $v_0, r_x, -, -$ 8. MOVE $v_q, -, -,$ rd // store q in rd
REMU r_x, 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 r_x, r_y, rd	1. ADVICE $-,-,-,v_q$ // store non-deterministic advice q into v_q 2. ADVICE $-,-,-,v_r$ // store non-deterministic advice r into v_r 3. ASSERT_LT_ABS $v_r, r_y, -, -$ // verify that $\|r\| < \|y\|$ 4. ASSERT_EQ_SIGNS $v_r, r_y, -, -$ // require r to have the sign of y 5. MUL v_q, r_y, v_{qy} // compute $q \times y$ 6. ADD v_{qy}, 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 r_x, r_y, rd	1-7. same as above 8. MOVE $v_r, -, -,$ rd // store r in rd

8 Evaluation

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, $\mathsf{SHIFT_R}(x, len)$ 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	$\sum_{i=0}^{23} \widetilde{eq}(i, x_{exp}) \cdot \mathsf{SHIFT_R}(1 \parallel x_{mant}, 23 - i)$ $+ \sum_{i=24}^{23+63} \widetilde{eq}(i, x_{exp}) \cdot \mathsf{SHIFT_L}(1 \parallel x_{mant}, i - 23)$
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` • $z_{mant} = \sum_{i=0}^{64} \widetilde{eq}(x_{>i}, 0) \cdot x_i \cdot \big(\sum_{j=0}^{23} 2^j x_{i+j} \big)$
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	$x \parallel y$	$z_s = x_s$ $z_{exp} = x_{exp} + (1 - x_{24}) \cdot x_{23} - (1 - x_{24}) \cdot (1 - x_{23})$ $z_{mantissa} = x_{24} \cdot x_{[23 \ldots 1]} + (1 - x_{24}) \cdot x_{23} \cdot x_{[22 \ldots 0]} + (1 - x_{24}) \cdot (1 - x_{23}) \cdot x_{[21 \ldots 0]}$

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.