Verifying Jolt zkVM Lookup Semantics

Carl Kwan[1], Quang Dao[2], and Justin Thaler[3]

[1] The University of Texas at Austin**
carlkwan@cs.utexas.edu
[2] Carnegie Mellon University
qvd@andrew.cmu.edu
[3] Georgetown University & a16z crypto research
justin.r.thaler@gmail.com

Abstract. Lookups are a popular way to express repeated constraints in state-of-the art SNARKs. This is especially the case for *zero-knowledge virtual machines* (zkVMs), which produce succinct proofs of correct execution for programs expressed as bytecode according to a specific instruction set architecture (ISA). The Jolt zkVM (Arun, Setty & Thaler, Eurocrypt 2024) for RISC-V ISA employs Lasso (Setty, Thaler & Wahby, Eurocrypt 2024), an efficient lookup argument for massive structured tables, to prove correct execution of instructions. Internally, Lasso performs multiple lookups into smaller "subtables", then combines the results.
We present an approach to formally verify Lasso-style lookup arguments against the semantics of instruction set architectures. We demonstrate our approach by formalizing and verifying all Jolt 32-bit instructions corresponding to the RISC-V base instruction set (RV32I) using the ACL2 theorem proving system. Our formal ACL2 model has undergone extensive validation against the Rust implementation of Jolt. Due to ACL2's bitblasting, rewriting, and developer-friendly features, our formalization is highly automated.
Through formalization, we also discovered optimizations to the Jolt codebase, leading to improved efficiency without impacting correctness or soundness. In particular, we removed one unnecessary lookup each for four instructions, and reduced the sizes of three subtables by 87.5%.

1 Introduction

Cryptographic proof systems [34] are essential to the scalability and privacy of modern blockchains. Succinct Non-interactive Arguments of Knowledge (SNARKs) [6, 31, 52] allow participants to prove arbitrary NP computations, generating short on-chain proofs that can be efficiently verified. In other words, SNARKs allow untrusted provers to establish that they know a "witness" satisfying some property, such as a correct batch of blockchain transactions that advance the blockchain from one state to another.

** This work was partially performed while the first and second authors were interns at a16z crypto research.

A special case of SNARKs are zero-knowledge virtual machines (zkVMs),[4] which enable succinct proofs of program execution for computer programs compiled to bytecode for some instruction set architecture (ISA). zkVMs enable developers to write programs in high-level programming languages, and for untrusted provers to then prove that they ran the program correctly. This is in contrast to older toolchains for SNARK deployment, which often involve handcrafting circuits or constraint systems, a process that requires domain-specific expertise and is highly error-prone. Because of these benefits, there has been an explosion of zkVMs for various instruction sets such as RISC-V [40, 56, 67, 72], EVM [63, 68, 88], MIPS [86], Cairo [33, 71] and others [48, 49, 77].

zkVMs often rely on *lookup arguments* to efficiently represent high-degree constraints that are needed to implement virtually all instruction sets. These arguments allow for proving the correctness of a sequence of lookup operations into some pre-determined tables. The performance of lookup arguments [10, 26, 29, 30, 35, 61, 64, 84, 85] typically scales with the size of the table;[5] however, a recent work by Setty et al. called Lasso [70] overcomes this limitation (for a large useful class of tables). The key idea of Lasso is that for specific tables that satisfy a form of *decomposability*, a lookup on the large table (say of size 2^{64}) can be performed via a sequence of lookups on much smaller subtables (say of size 2^{16}). This powerful observation forms the basis of the Jolt RISC-V zkVM [2], which represents every RISC-V instruction as one or more decomposable lookups, realizing the "lookup singularity" vision outlined by Whitehat [81]. Jolt's implementation [40] is currently among the fastest zkVMs [65] and, thanks to its lookup-centric approach, is both simpler and more auditable than competitors [75, 76].

While Jolt may be simpler than other zkVMs, it is still a complicated piece of software. The current implementation [40] contains at least 25,000 lines of code (and growing), with at least 2000 lines dedicated to specifying lookups. Given the large surface area, it is crucial to *formally verify* that Jolt lookups are actually performing the right operations. This is not merely a theoretical concern: vulnerabilities in SNARK designs and implementations are quite common, with soundness bugs present in both protocol specifications [28, 62] and implementations [19, 23, 57]. Another class of bugs are those affecting the constraint system (so-called "front-end") used to represent computations [12, 17]. These front-end vulnerabilities are also prevalent in SNARK implementations; for Jolt specifically, a bug in the Lasso lookup front-end could allow a malicious prover to produce an accepting proof for a different program than the one the verifier thinks it is verifying.[6]

In light of these concerns, our goal is to **formally verify the semantic correctness of lookups in the Jolt zkVM**, ensuring that the lookups for each instruction actually produce the expected result for that instruction.

[4] Many zkVMs are only succinct and not zero-knowledge, but in keeping with the common vernacular we will refer to them as zkVMs.

[5] This dependency is present in either the prover cost or the preprocessing cost.

[6] This would be problematic, for instance, if the program does signature verification, and the prover produces a valid proof without knowing the secret key.

1.1 Our Results

We answer this goal with the following contributions:

1. We present a *general methodology* for formally verifying lookup semantics in Jolt (or any proof system that relies on Lasso-style decomposable lookups).
2. We instantiate this methodology with a formalization of all RISC-V base instructions (RV32I) in Jolt using the ACL2 theorem prover [42, 43]. Our formalization is *highly automated* thanks to various features of ACL2.

Our formalization and artifacts are publicly available.[7] Our approach achieves correctness guarantees for Jolt lookups while maintaining the flexibility needed to work with an evolving system like Jolt. We now elaborate on our results.

Jolt Subtables and Instructions. In Jolt, subtables are parametrized by a size parameter $m \in \mathbb{N}$. For each m, a subtable consists of a function

$$T_m : \{0,1\}^m \times \{0,1\}^m \to \mathbb{F},$$

which on input (x, y) returns the (x, y)-th entry of the subtable, and a polynomial

$$P_m \in \mathbb{F}[X_0, \ldots, X_{m-1}, Y_0, \ldots, Y_{m-1}]$$

purported to be the *multilinear extension* (MLE) of T_m (see Section 2.1 for background on MLEs). Here $\mathbb{F}$ is the underlying finite field used in Jolt (of size at least 2^{128}). The statement we want to formally verify is that P_m is indeed the MLE of T_m, i.e. that

$$P_m(x, y) = T_m(x, y) \quad \text{for all} \quad x, y \in \{0,1\}^m. \tag{1}$$

In particular, Equation (1) only needs to hold for parameters m that are used in Jolt; currently, Jolt only uses $m = 8$, giving us subtables of size $2^{16} = 65536$. This number is small enough that we can directly test for correctness between the materialized version and the MLE version. Nevertheless, formalizing the model of MLEs and proving correctness against the subtable functions is an interesting mathematical result in their own right.[8] We present a formal model of MLEs for some subtables and prove Equation (1) for larger bit-widths such as $m \in \{16, 32, \ldots\}$ (see Section 4.1 for an example). In future work, we plan to extend this to arbitrary $m \in \mathbb{N}$.

We next describe Jolt instructions. Each instruction in Jolt is parameterized by a word size $W \in \{32, 64\}$,[9] comes with an expected semantics, and a purported alternative way to achieve the same semantics using lookups into subtables. The alternative way proceeds as follows:

[7] https://github.com/kwancarl/acl2-jolt

[8] Formal modeling of MLEs will also help any future "back-end" verification effort, which establishes security of the argument system using the fact that these are indeed multilinear polynomials.

[9] Currently, Jolt only supports 32-bit instructions, with 64-bit planned for the future.

1. First, the operands of the instruction are split into $C = W/m$ chunks:
$$\mathsf{Chunk}(x, y) = (z_0, \ldots, z_{C-1}) \in \left(\{0,1\}^{2m}\right)^C.$$
2. Second, the chunks are then used to lookup into a list of subtables
$$L = ((\mathsf{ST}_0, \mathsf{idx}_0), \ldots, (\mathsf{ST}_{n-1}, \mathsf{idx}_{n-1})),$$
where for each j, ST_j is a subtable (of a given size m_j) and $\mathsf{idx}_j \in \{0, \ldots, C-1\}$ is the index of the chunk that will be used to query the subtable. In other words, lookup results are computed as
$$\mathsf{Lookup}(L, z_0, \ldots, z_{C-1}) = \left(T_j[z_{\mathsf{idx}_j}]\right)_{j=0}^{n-1}.$$
3. Finally, we produce the final result from the subtable lookups using a function $\mathsf{Combine}(\text{lookup results}) \in \mathbb{F}$, whose range should be in $\{0, \ldots, 2^W - 1\}$.

For a concrete example, the 32-bit AND instruction splits the operands $(x, y) \in \left(\{0,1\}^{32}\right)^2$ into 4 chunks of 16 bits $(x_i, y_i) \in \left(\{0,1\}^8\right)^2$, looks up each chunk in the AND subtable, then concatenates the lookup results, which should be equal to the bit-wise AND of the two operands. See Appendix B for a list of all instructions. The relation we formally verify is that the lookup process (performing steps 1, 2, and 3) gives the expected result of the instruction. In other words, the following equation holds for all $W \in \{32, 64\}$ and $x, y \in \{0,1\}^W$:
$$\mathsf{ExpectedResult}(x, y) = (\mathsf{Combine} \circ \mathsf{Lookup} \circ \mathsf{Chunk})\,(x, y). \tag{2}$$

Theorem 1.1. *Equation (2) holds for every* Jolt *instruction in the base RV32I instruction sets.*

Figure 1 visualizes our results. We further elaborate on our approach in Section 3, and fully work out an example verified instruction in Section 4.

Automation and Extensibility. Our formalization is highly automated, thanks to ACL2's built-in automated rewriting and verified model checking libraries. Our theorems are discharged with minimal user intervention, requiring only simple connecting lemmas. At a minimum, our contribution requires expertise with various formal methods tools (e.g. theorem provers, model checkers, specification languages), the insight to apply the appropriate method for a particular application, and the ability to model real-world and mathematical systems. Our more significant contribution is in developing an extensible theory which can automatically discharge the correctness of *families* of Jolt instructions. Doing this requires finding a pattern of theorems that enables a tool (e.g. ACL2) to automatically rewrite formal statements involving Lasso-style lookups and Jolt instructions into true statements. We illustrate this in Section 3.

In most cases, once we proved correctness of 32-bit instructions, extending to 64-bit versions requires only restatements of definitions and theorems. Initial experiments indicate that our approach scales to formally verifying parts of the RISC-V M-extension [66], which was recently added to Jolt [50]; we leave a full write-up of this effort to future work.

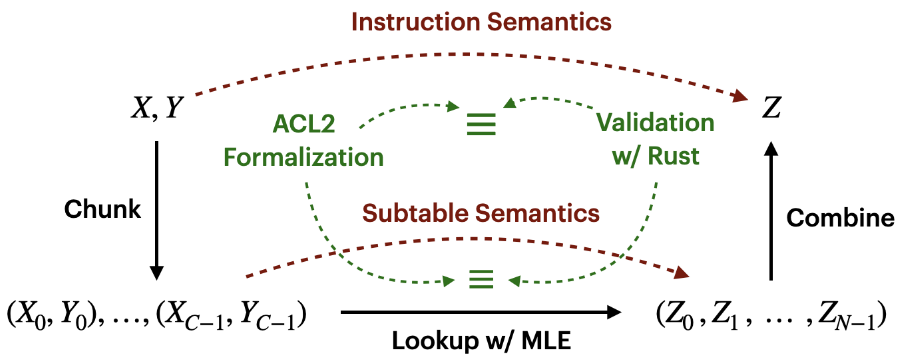

Fig. 1: Overview of our formalization. The main contents of our work are the **green** equivalences, which we prove for all subtables and instructions in the base RV32I & RV64I instruction sets. Our result assumes that Lasso correctly proves the expected decomposable lookups with respect to the black arrows.

Validation with Rust. Since we develop our formal model in ACL2, it is crucial to validate that these models correspond to the Rust implementation [40]. Subtables are small enough (of size at most 2^{16}) that we exhaustively check for correctness between ACL2 and Rust. Since instructions cannot be exhaustively checked, we perform validation on randomly-chosen inputs, and scope our formal model so that the only points of potential differences are in commonly-used subroutine helper functions (such as Chunk and Combine functions). We present these functions in Appendix B for manual inspection.

Efficiency Benefits of Our Formalization. We remove one unused lookup for each of the comparison instructions (SLT, SLTU, BGE, and BGEU), which constitute 4 out of 19 instructions in total. We discover this optimization through our ACL2 formalization, as ACL2 tooling could automatically recognize that a lookup went unused, producing an error message. Furthermore, we recognize that the sizes of shift-related subtables can be significantly smaller than the current implementation. This is because only the last 5 bits of the shift amount operand are relevant to 32-bit shift instructions; thus, we can use $2^8 \times 2^5 = 2^{13}$-sized subtables instead of the 2^{16}-sized subtables that is currently used.

We contacted the Jolt developers to integrate these optimizations. The first (removing unused lookups) was already integrated, and the second (smaller shift subtables) is currently under review. For more details, see Section 5.

Non-Goals. Since our focus is on verifying lookup semantics, it is necessary for us to black-box other parts of the Jolt codebase. In particular, we assume that the underlying "back-end" argument systems (such as Lasso) are secure, and correctly prove all specified relations (such as all lookups). We also do not model other parts of the front-end such as R1CS and offline memory-checking [69]. As Jolt is a large codebase, we believe our approach is a pragmatic compromise that allows for incremental verification. In future work, we plan to incorporate other front-end components into our formal model.

1.2 Related Works

A growing body of work has applied techniques from *formal methods* to rule out bugs in SNARK front-ends, with various tradeoffs between the automation of the tool and the guarantee it provides. Many focus on verifying ZK circuits, specifically in R1CS format [18,21,39,47,60,78,80]. Some of this work involves developing solvers over finite fields [36, 58, 59] or detecting under-constrained circuit bugs [60, 78], aiming for high automation but not necessarily full formal correctness. Some even use ACL2 [18, 20]. On the other hand, there are formalization results that prove full formal correctness of either a VM front-end such as Cairo [3, 4], zkWasm [16], or of R1CS or Plonkish circuits via special DSLs [21,47]. Our work differs from these in the following aspects. First, we target verifying (a particular form of) lookups that are used inside general-purpose zkVMs, whereas other works focus on verifying arithmetic constraints such as R1CS, Plonk-ish, or AIR. Second, we aim for full functional correctness, unlike other works that aim only to detect, e.g. under-constrained circuits. Finally, despite full formal guarantees, we do not compromise on automation, by leveraging desirable features of ACL2.

Cairo. The work most relevant to ours is the formal verification effort for the Cairo ecosystem [33], which consists of a high-level programming language, an ISA, and a compiler from the high-level language to the ISA. The first work [3] verified the correctness of an AIR encoding for the Cairo ISA, while the second work [4] augmented the Cairo compiler with tools that allow for proving correctness of compiled programs, without needing to verify the overall compiler.

There are a few key differences between Cairo and our work with Jolt. Cairo is a particularly simple VM that is specifically designed to be SNARK friendly, and pertains to an ad hoc language and architecture. Users who wish to produce proofs of programs will need to either use the Cairo language, or compile down to the Cairo ISA from another high-level language. This contrasts with the existing mature compiler, tooling, and infrastructure for RISC-V, which is Jolt's target architecture. Finally, Cairo's verification efforts in Lean [53] are highly manual. As Cairo changes, it is unknown how the formal proofs may change. Our efforts in this work aim to enable automatic, extensible, and scalable verification as Jolt matures.

zkWasm. Another recent work [16] by the CertiK team [15] focuses on the zkWasm project [87], verifying the correctness of arithmetic constraints for the zkWasm VM. It is difficult to compare their efforts to ours, since CertiK has not yet put forth a peer-reviewed publication, instead releasing a code preview with all proofs omitted, and several high-level blog posts [11, 13, 14]. Nevertheless, the same comments regarding Cairo's lack of automated verification can also be made about CertiK's formalization in Coq [22]. While Wasm [79] is a more complex ISA than Cairo, the particular form of constraint system used to model Wasm instructions are quite different from Jolt's, and hence the verification effort is orthogonal to ours. In particular, zkWasm relies on Plonkish/Halo2-style constraints [37], while Jolt uses decomposable lookups along with R1CS and memory-checking.

Backend Verification. Finally, some recent works focused on verifying SNARK backends, such as verifying soundness for Linear-PCP-based SNARKs [5] and for the sum-check protocol [7], and functional correctness of SNARK verifier implementations [27, 51]. These are complementary to our efforts, which focus solely on the front-end.

2 Preliminaries

2.1 Jolt & Lasso Basics

In this section, we describe the basics of Lasso and the decomposable lookup relation that the argument proves.

Multilinear Extensions. Given a function $f : \{0,1\}^n \to \mathbb{F}$, its *multilinear extension (MLE)* $\widetilde{f} : \mathbb{F}^n \to \mathbb{F}$ is the unique multilinear polynomial that agrees with f on all inputs in $\{0,1\}^n$. Its formula is given via multilinear Lagrange interpolation:

$$\widetilde{f}(X) = \sum_{y \in \{0,1\}^n} \prod_{i=1}^{n} \mathsf{eq}(X, y) \cdot f(y), \tag{3}$$

where the equality polynomial eq is defined as

$$\mathsf{eq}(X_1, \ldots, X_n, Y_1, \ldots, Y_n) = \prod_{i=1}^{n} \left(X_i \cdot Y_i + (1 - X_i) \cdot (1 - Y_i) \right).$$

Due to their use in the sum-check protocol, efficient evaluation of MLEs are crucial for succinctness of the Lasso verifier. MLE evaluations are not efficient in general, since by (3) we have to sum over 2^n terms. However, some MLEs admit formulas with efficient evaluation, on the order of $O(n)$ field operations instead of $O(2^n)$. This property of fast MLE evaluation is key to the succinctness of Lasso, and in fact all subtables in Jolt (see Appendix A for the full list) are chosen to have fast MLE evaluation.

Decomposable Lookup Relations and Lasso*.* We describe the type of lookup tables that can be proved efficiently by Lasso. In the original paper [70], it is referred to as having a *Surge-only structure* (SOS); in this work, we simply refer to this property as being *decomposable*.

Definition 2.1 (Decomposable Lookup Table). *Let $T : \{0,1\}^n \to \mathbb{F}$ be a lookup table for some $n \geq 1$ and finite field $\mathbb{F}$. Given a divisor $c \geq 1$ of n, some $m \geq 1$, a mapping $\mathsf{i} : [m] \to [c]$, and a low-degree polynomial $g : \mathbb{F}^m \to \mathbb{F}$, we say that T is* (c, m, i, g)-decomposable *if there exist a sequence of* subtables $T_1, \ldots, T_m : \{0,1\}^{n/c} \to \mathbb{F}$ *such that:*

1. *The multilinear extensions $\widetilde{T}_1, \ldots, \widetilde{T}_m$ of $T_1, \ldots, T_m$ can be evaluated in $O(\log(n/c))$ time;*
2. *For all $x \in \{0,1\}^n$, writing $x = (x_1, \ldots, x_c) \in \left(\{0,1\}^{n/c}\right)^c$, we have:*

$$T(x) = g\left(T_1(x_{\mathsf{i}(1)}), \ldots, T_m(x_{\mathsf{i}(m)})\right). \tag{4}$$

Given a (c, m, i, g)-decomposable lookup table T and a commitment scheme cm *for field elements, we define the* (committed) decomposable lookup relation *as:*

$$\mathcal{R}_{T,c,m,\mathsf{i},g,\mathsf{cm}} := \{(c, (y, x)) \mid c = \mathsf{cm}(y) \ \wedge \ y = T(x)\}.$$

Lasso is an argument system for proving correctness of decomposable lookup relations. In this paper, we will assume that Lasso is complete and sound, meaning that it correctly proves Equation (4) for any (c, m, i, g)-decomposable lookup table T. Our focus will be to show that the lookup relations in Jolt that are proved by Lasso in fact represent the intended semantics of RISC-V instructions.

2.2 ACL2 Basics

ACL2 is a highly automated theorem proving system, which also supports programming in Common Lisp. It has seen success in verifying hardware, software, and cyber-physical systems at companies such as Intel, AMD, ARM, Collins Aerospace, and more [38]. Some Jolt-relevant ACL2 successes include highly-efficient formal executable models of x86, Y86, JVM, and RISC-V ISAs [32, 46, 83]. While ACL2 handily supports reasoning about software and general-purpose mathematics, it has seen outsized impact verifying low-level systems, motivating much tool development in this direction. This makes ACL2 well-suited to verifying the front-end of Jolt or other zkVMs.

ACL2's logic is largely first-order, with support for certain higher-order logics in some applications [8, 44, 45]. The ACL2 foundations themselves have also been subjected to extensive formal verification, all the way down to the x86 code running ACL2 itself [24, 25, 54, 55]. Execution in ACL2 is the same as native Common Lisp execution, and included in the meta-verification chain of ACL2 is a verified Common Lisp kernel. The close interaction between ACL2 and Common Lisp enables concurrent programming and theorem proving about programs.

Table 1 lists some common ACL2 functions and macros used in this paper. For brevity, we omit the extensive details of the ACL2 rewriting system and heuristics. For this paper, it is sufficient to know that ACL2 will automatically attempt to rewrite formal statements into known equivalent statements. For example, the following is a theorem which is automatically proven in ACL2:

```
(defthm foo (equal (- a a) 0))
```

Table 1: Common ACL2 functions, macros, and other commands used in this paper.

Command	*Description*
`defun`	Define a function symbol
`define`	Define a function symbol, enforce guard checking, and more
`b*`	Binder for local variables; often used to simplify control flow
`defthm`	Name and prove a theorem
`cons`	Construct a pair or list
`car`	Return the head of a list
`cdr`	Return the second element of a `cons` pair
`logcar`	Return the least significant bit of a number
`logcdr`	Return all but the least significant bit of a number
`part-select`	Return a bitvector part of an integer
`natp`	Recognizer for natural numbers
`bitp`	Recognizer for bits
`unsigned-byte-p`	Recognizer for unsigned numbers fitting a specified bit width
`def-gl-thm`	Name and prove a theorem using GL symbolic simulation

The function `defthm` above introduces a theorem named `foo` stating that $a-a = 0$. When ACL2 proves `foo`, it also automatically introduces a rewrite rule which enables one to replace terms of the form `(- a a)` with `0`. For example, `foo` is sufficient to discharge the following theorem attempt:

```
(defthm bar (equal (- (+ b c) (+ b c)) 0))
```

ACL2 will automatically pattern match the term `(+ b c)` to `a` and apply an instance of theorem `foo`.[10] A typical ACL2 theorem will involve: (1) hypotheses (if any); and (2) the theorem conclusion. More details on the ACL2 rewriting system can be found in the ACL2 documentation [82].

Another instrument in the ACL2 toolkit is the use of *bitblasting*. Bitblasting is the process of reducing theorems on finite domains into decision problems on bitvectors, making them amenable to SAT-like decision procedures. ACL2 supports bitblasting via several frameworks. We use GL, which supports bitblasting via BDDs or external SAT solvers [73, 74]. The internal GL BDD-based model checker and proof procedures are fully written in and verified with ACL2. When external solvers are used, GL can efficiently check the resolution proof produced by the SAT solver to avoid trusting any external tools, thus maintaining soundness. If a proof fails and terminates, then GL produces counterexamples. In addition to a hypothesis and conclusion, a typical GL theorem event will involve bindings which assign bits to represent variables in the theorem statement.

For example, consider the `foo` example from Section 2.2, which stated $a - a = 0$. We can prove this theorem with GL if we restrict the domain of a:

```
(def-gl-thm foo-gl
  :hyp (unsigned-byte-p 32 a)
  :concl (equal (- a a) 0)
  :g-bindings (auto-bindings (:nat a 32)))
```

[10] This example is contrived, as other ACL2 heuristics may discharge `bar` before the `foo` rewrite rule has a chance to fire, but it demonstrates the basics of ACL2 rewriting.

The value provided to the key `:hyp` is the hypothesis of the theorem, which states that `a` is an unsigned 32-bit integer; the value provided to `:concl` is the conclusion, which is the same as `foo`; and `:g-bindings` indicate variable bindings for `a` – in this case `(gl::auto-bindings (:nat a 32))` is simply a convenient way of assigning bits 0 – 32 to `a`.

On the backend, GL is performing symbolic simulation on `a` as a symbolic object. By default, BDDs are used to encode the symbolic objects, but external SAT solvers can be used as well. The internal GL BDD-based model checker and proof procedures are fully written in and verified with ACL2. When external solvers are used, GL can efficiently check the resolution proof produced by the SAT solver to avoid trusting any external tools, thus maintaining soundness. If the proof fails and terminates, then GL produces counterexamples.

Note that `foo-gl` is again a contrived example for two reasons: (a) GL's proof procedures involve clause processing, which enables use of `foo` to discharge `foo-gl`; (b) bitblasting and model checking tends to find better utility in situations which involve detailed, complex, and sometimes tedious operations on many objects. These situations often arise in hardware or low-level systems verification, such as the subject of this paper. To provide an example of such a use case, consider the the following well known bitwiddling hack [1] for the logcount (a.k.a. popcount, Hamming weight, 1s count, etc.) of an int `v`, which is also a common problem in GL tutorials:

```
v = v - ((v >> 1) & 0x55555555);
v = (v & 0x33333333) + ((v >> 2) & 0x33333333);
c = ((v + (v >> 4) & 0xF0F0F0F) * 0x1010101) >> 24;
```

The following ACL2 events verify the above obtuse C-style code is indeed equivalent to a standard definition of logcount:

```
;; Define 32-bit multiplication
(defun 32* (x y)
  (logand (* x y) (1- (expt 2 32))))

;; Define logcount using a bitwiddling hack
(defun fast-logcount-32 (v)
  (b* ((v (- v (logand (ash v -1) #x55555555)))
       (v (+ (logand v #x33333333)
             (logand (ash v -2) #x33333333)))
       (c (ash (32* (logand (+ v (ash v -4)) #xF0F0F0F) #x1010101) -24)))
      c))

;; Verify the correctness of fast-logcount-32 against a standard logcount
(gl::def-gl-thm fast-logcount-32-correct
  :hyp   (unsigned-byte-p 32 v)
  :concl (equal (fast-logcount-32 v) (logcount v))
  :g-bindings (gl::auto-bindings (:nat v 32)))
```

The first event in the above code snippet is a call to `defun` introducing the function `32*`, which performs multiplication modulo 2^{32} by taking the bitwise "and" of a product with $2^{32} - 1$. The second event introduces a function on `v` which performs the same operations as in the bitwiddling hack; notably, `b*` is a binder playing the role of the assignment operator. The final event simply asserts that the bitwiddling version of logcount is the same as the standard ACL2

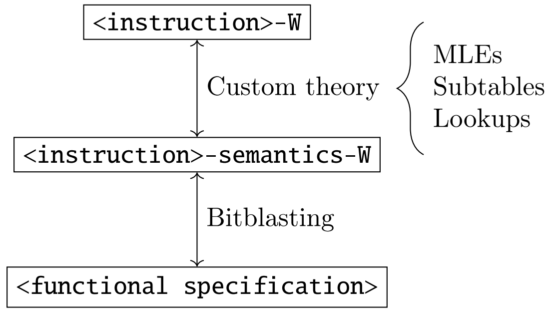

Fig. 2: Approach to verifying Jolt instructions, where $W \in \{32, 64\}$

logcount. This approach – defining different versions (optimized or otherwise) of semantically equivalent functions and proving them equivalent – is archetypal of many verification efforts, and is repeated in this paper.

GL's symbolic simulation of large sets of boolean functions with BDD-based procedures has successfully verified many large-scale industrial designs. However, model checking with SAT or BDDs has its limits, either due to what is expressible in the language or due to scalability issues. For sophisticated verification efforts involving various high-level protocols, low-level machinery, and their interactions – such as Jolt – a unified approach combining bitblasting with light user-guided theorem proving is more effective. Often, the expedient approach to verification with ACL2 is to setup a theory with the appropriate theorems which enable ACL2 heuristics to automatically discharge the final desired theorems. We will describe such an approach in the context of Jolt lookups.

3 General Approach to Lookup Formalization

Formalizing each Jolt instruction in ACL2 broadly involves proving two parts:

(A) the multilinear extensions (MLEs) involved in the lookups are equivalent to their respective intended behavior; and
(B) the composition of chunking the inputs, lookups to materialized subtables, and combining the results is equivalent to the intended instruction semantics.

A particular Jolt instruction may involve multiple lookups, but otherwise part (A) is a straightforward exercise in equational reasoning. In fact, part (A) is amenable to bitblasting if we restrict our attention to Jolt-specific bitwidths.

Part (B) involves combining new definitions, intermediate semantic layers, bitblasting, and theory management to automatically trigger rewrite rules that discharge the desired final theorem with very little user-interaction. It is the part that we focus on in this section. A summary of our approach is visualized in Figure 2, and a full example of an instruction is given in Program 4.

The first step in part (B) is to define and verify the necessary subtables for each Jolt instruction. We make a distinction between subtables which are indexed by one or two parameters, and define analogous functions for each. Since many subtables are intended to be the same size, we have a single function which generates the indices based on upper limits for x and y (for subtables indexed by two parameters). Lookup functions are built on top of existing ACL2 association list and related libraries. Some subtables are further parameterized (e.g. by word sizes or chunk index) but all lookup functions take a subtable and either one or two parameters. A common pattern of theorems and rewrite rules enable us to readily verify that lookups to these subtables return the expected result. These theorems often state the return "type" and correctness of the supporting functions.

For every Jolt instruction, we define an equivalent instruction which replaces the lookup stage with their intended semantics; otherwise, they contain identical chunking and combining stages. We name this intermediary semantics representation `<instruction>-semantics-32` and simply bitblast their correctness. Then we define the Jolt instructions themselves, which we simply name `<instruction>-32`. Because they only differ with respect to lookups, proving the Jolt instructions equivalent to their intermediate versions reduces to the theorems we prove about the subtables and lookups.

We restrict our use of bitblasting to theorems about lower-level functions or intermediate steps. Even though we verify the correctness of each subtable and the associated MLEs, they are simply stored as ACL2 rewrite rules. Clause processing with lookup arguments is not mature in GL or ACL2; a naïve attempt to bitblast or otherwise prove the correctness of `sltu-32` or any direct implementation of a Jolt instruction will involve too many function definitions. Part of this is exacerbated by the concrete values of $W = 32$ and $m = 8$ in the 32-bit version of Jolt instructions. Certain executions in proof attempts with such instructions are no longer symbolic and tables of size 2^{16} can literally be materialized. On their own, such tables can be exhaustively checked for correctness, but layering on top various chunking and combining functions can easily cause the state space to be intractable or misdirect the theorem prover. Instead, for top-level theorems, we control the rules space, preventing the rewriter from expanding the function definition in most proofs, enabling it to focus only on the correctness lemmas, and providing it with a simple chain of equivalences to obtain the final desired theorem.

4 Formalization Example: Set Less Than (Unsigned)

In this section, we describe the process of formalizing and verifying an example Jolt instruction, set less than (unsigned), abbreviated by SLTU. We begin with the modelling and verification of the MLEs, corresponding to part (A) of the approach described in Section 3. Then we verify the top-level Jolt instruction itself, corresponding to part (B).

4.1 Formalizing MLEs for Subtables

Given $x, y \in \{0,1\}^W$, the SLTU instruction returns 1 if $x < y$ and 0 otherwise, treating x and y as unsigned integers. To get a multilinear extension formula for

Program 1: ACL2 Formalization of $\widetilde{\mathsf{Eq}}_m$

```
;; Equality of two bits, computes x * y + (1 - x) * (1 - y)
(define b-eq-w ((x bitp) (y bitp))
  (b-xor (b-and x y) (b-and (b-xor 1 x) (b-xor 1 y))) ...)

;; Equality of two bitvectors, computes: Prod (xi * yi + (1 - xi) * (1 - yi))
(define eq-w ((x :type unsigned-byte) (y :type unsigned-byte))
 (b* (((unless (and (natp x) (natp y)))          0)    ;; Edge cases
      ((if (xor (bitp x) (bitp y)))              0)
      ((if (and (bitp x) (bitp y)))    (b-eq-w x y))    ;; Base case
      (x-0     (logcar x))                              ;; LSB of x
      (y-0     (logcar y))                              ;; LSB of y
      (x-rest  (logcdr x))                              ;; Rest of x
      (y-rest  (logcdr y)))                             ;; Rest of y
     (b-and (b-eq-w x-0 y-0) (eq-w x-rest y-rest)))     ;; Recursive case
 ///
 ;; eq-w correctness theorem for arbitrary unsigned ints
 (defthmd eq-w-equal-equiv
  (implies (and (natp x) (natp y)) (equal (eq-w x y) (if (equal x y) 1 0)))))
```

SLTU, we use two other MLEs, $\widetilde{\mathsf{Eq}}_m$ for "equality" and $\widetilde{\mathsf{Ltu}}_m$ for "less than (unsigned)", a helper polynomial. The construction of the former is straightforward:

$$\widetilde{\mathsf{Eq}}_m(X, Y) = \prod_{i=0}^{m-1} ((1 - X_i)(1 - Y_i) + X_i \cdot Y_i) \tag{5}$$

checks whether bitvector chunks X and Y of length m are equal. The i-th factor in the product of Equation (5) above checks whether the i-th bits in X and Y are the same, returning 1 if so and 0 otherwise. Thus if X and Y mismatch in any position, the entire product is 0. Otherwise, the product is 1.

We embed $\widetilde{\mathsf{Eq}}_m$ into ACL2 and prove it is indeed equivalent to the mathematical semantics of an "equal" function. The entire sequence of events is displayed in Program 1. We define two functions: **b-eq-w**, which computes whether two bits are equal; and **eq-w**, which calls **b-eq-w** recursively along the lengths of two bitvectors. The **b*** macro in **eq-w** is a binder that also supports convenient features [9], similar to the Common Lisp **let***. The desired theorem is **eq-w-equal-equiv**, which states that **eq-w** (or Equation (5)) is equivalent to equality when **x** and **y** are unsigned integers.

The MLE for "less than (unsigned)" is defined by

$$\begin{aligned}\widetilde{\mathsf{Ltu}}_m(X, Y) &= \sum_{i=0}^{m-1} (1 - X_i) \cdot Y_i \cdot \widetilde{\mathsf{Eq}}_i(X_{<i}, Y_{<i}) \\ &= \sum_{i=0}^{m-1} (1 - X_i) \cdot Y_i \cdot \prod_{j=0}^{i-1} ((1 - X_j) \cdot (1 - Y_j) + X_j \cdot Y_j).\end{aligned} \tag{6}$$

Intuitively, $\widetilde{\mathsf{Ltu}}_m(X, Y)$ recognizes when $X < Y$ because a summand will be nonzero only if the $(i-1)$-th most significant bits are equal, Y_i is high, and X_i is low, i.e. when i is the most significant bit position which determines $X < Y$. We

Program 2: ACL2 Formalization of $\widetilde{\mathsf{Ltu}}_m$

```
;; Compute the MLE for LTU
(define ltu-w ((x :type unsigned-byte) (y :type unsigned-byte)) ...
  (b* (((unless (and (natp x) (natp y)))                            0) ;; Edge case
       ((if (and (zerop (integer-length x)) (zerop (integer-length y)))) 0) ;; Base case
       (x-0    (logcar x))                                            ;; LSB of x
       (y-0    (logcar y))                                            ;; LSB of y
       (x-rest (logcdr x))                                            ;; Rest of x
       (y-rest (logcdr y))                                            ;; Rest of y
       (ltu-0  (b-and (b-and (b-xor 1 x-0) y-0) (eq-w x-rest y-rest))))     ;; Summand
      (b-xor ltu-0 (ltu-w x-rest y-rest))))                         ;; Recursive case
```

can easily verify Equation (6) is correct for large bitwidths by defining an ACL2 function `ltu-w` which computes $\widetilde{\mathsf{Ltu}}_m$ and using GL to bitblast its correctness theorem `ltu-w-equiv-<-32`.

4.2 Verifying an Instruction involving Subtable Lookups

Now that the MLEs for the SLTU subtables are verified, we proceed with materializing the subtables. We model subtables in ACL2 as association lists, i.e. lists of pairs where the first element in the pair acts as a key and the second element acts as a value. For subtables associated with MLEs, the keys are the operands and the values are the expected results. To make this concrete, consider `materialize-eq-subtable` in Program 3. Evaluating the function on a list of X and Y operands, such as

`((1 . 1) (1 . 0) (0 . 1) (0 . 0))`,

returns the following subtable:

`(((1 . 1) . 1) ((1 . 0) . 0) ((0 . 1) . 0) ((0 . 0) . 1))`.

As expected, the values in the subtable are 1 only when X and Y are equal and 0 otherwise. By proving a small number of intermediate lemmas, we automatically obtain a correctness theorem for $\widetilde{\mathsf{Eq}}_m$'s subtable in the form of `lookup-eq-subtable-correctness` in Program 3. This theorem states that if the operands of interest are indexed in the subtable, then the lookup value is 1 if the operands are equal and 0 otherwise. Very similar functions and theorems are formalized for $\widetilde{\mathsf{Ltu}}_m$; the only substantial difference is that = is replaced with <. The function application

`(create-tuple-indices x-hi y-hi)`

simply creates the list

`((x-hi . y-hi) ... (1 . 1) (1 . 0) (0 . 1) (0 . 0))`,

and

`(tuple-lookup i j subtable)`

returns the value of the subtable at index `(i . j)`.

Program 3: ACL2 Formalization of Eq_m Subtable

```
;; Given a list of (x y) operands, materialize a list of key-value pairs
;;   key: (x y)     value:  (if (= x y) 1 0)
(defun materialize-eq-subtable (idx-lst)
 (b* (((unless (alistp idx-lst))     nil)  ;; Edge case
      ((if (atom idx-lst))           nil)  ;; Base case
      ((cons hd tl)              idx-lst)  ;; Bind head & tail in the index list
      ((unless (consp hd))           nil)  ;; Edge case
      ((cons x y)                     hd)) ;; Bind x & y operands in the head
     ;; Construct a key-value pair and append it to the rest of the eq subtable
     (cons (cons hd (if (= x y) 1 0)) (materialize-eq-subtable tl))))
...
;; Lookup values within the bounds of the subtable are equivalent to "="
(defthm lookup-eq-subtable-correctness
 (implies (and (natp i) (natp j)  (natp x-hi) (natp y-hi) (<= i x-hi) (<= j y-hi))
          (b* ((indices  (create-tuple-indices x-hi y-hi))
               (subtable (materialize-eq-subtable indices)))
              (equal (tuple-lookup i j subtable) (if (= i j) 1 0)))) ... )
```

We are now ready to tackle SLTU itself. Each Jolt instruction has three stages: chunk, lookup, and combine. For SLTU, integers $x,\ y \in \{0,1\}^{W=32}$ are chunked into subvectors of size $m = 8$ so that

$$x = x_0 \| x_1 \| x_2 \| x_3, \qquad\qquad y = y_0 \| y_1 \| y_2 \| y_3 \tag{7}$$

in preparation for the subtable lookups. Lookups are performed using interwoven chunks as indexes, setting

$$\begin{aligned} Z_0 \coloneqq \mathsf{Ltu}_m\left[x_0 \| y_0\right],\ Z_1 \coloneqq \mathsf{Ltu}_m\left[x_1 \| y_1\right],\ Z_2 \coloneqq \mathsf{Ltu}_m\left[x_2 \| y_2\right],\ Z_3 \coloneqq \mathsf{Ltu}_m\left[x_3 \| y_3\right], \\ W_0 \coloneqq \mathsf{Eq}_m\left[x_0 \| y_0\right],\ W_1 \coloneqq \mathsf{Eq}_m\left[x_1 \| y_1\right],\ W_2 \coloneqq \mathsf{Eq}_m\left[x_2 \| y_2\right]. \end{aligned} \tag{8}$$

Then combining the results gives

$$\mathsf{SLTU}(x, y) = \sum_{i=0}^{3} Z_i \prod_{j=0}^{i-1} W_j = Z_0 + Z_1 \cdot W_0 + Z_2 \cdot W_0 \cdot W_1 + Z_3 \cdot W_0 \cdot W_1 \cdot W_2. \tag{9}$$

Verifying that Equation (9) is indeed equivalent to `(if (< x y) 1 0)` is the ultimate objective, which is Program 4.

We first formalize a version of SLTU without subtables, instead directly calling the intended function of the lookups in this intermediary semantics version of the instruction, which we call `sltu-semantics-32`. The bindings which involve `part-select` correspond to Equation (7). One subtle note is that ACL2 indexes bitvectors such that the least significant bit is 0, whereas Jolt indexes the most significant chunks as 0; hence the 8-bit chunk resulting from

`(part-select x :low 0 :width 8)`

corresponds to the 3-rd chunk x_3 of x. The next sequence of bindings is semantically equivalent to the lookup stage in Equation (8). The final expression returned is equivalent to Equation (9). Verification is again a straightforward application of GL. Finally, we define the Jolt version of SLTU, `sltu-32`. The function `sltu-32` is identical to `sltu-semantics-32` except that we actually materialize the subtables and perform lookups. We prove it equivalent to the version without lookups, `sltu-semantics-32`, giving us a chain of equivalences which enable us to conclude that Jolt SLTU is correct.

5 Optimizations & Impact to Jolt's Codebase

An ACL2 analysis of Jolt forces the development of a formal specification for the Jolt components we verify. Formal specification in itself is already an important contribution to any project. In a large technology company, documentation and specification can have its own dedicated stage in a product cycle. However, Jolt was originally announced in the form of an academic paper, eliding some implementation details. Conversely, the Rust implementation of Jolt contains many undocumented engineering efforts, either a result of filling in the gaps of the Jolt paper or complete departures altogether. For example, some MLEs have no mathematical specification beyond their function. The MLE known as $\widetilde{\mathsf{EqAbs}}$ simply checks whether all but the MSBs of two bitvectors are equal. Despite its name, $\widetilde{\mathsf{EqAbs}}$ does **not** check whether two signed integers are equal under the absolute value function. A naïve attempt to verify against the absolute value function resulted in automatically generated counterexamples. Examples such as these, which were not documented in either the original paper or in the Jolt codebase, are now identified and documented.

By the time large projects go to formal verification, the design is often already highly optimized (indeed this is often the point of formal methods: to verify the optimization is correct). In the case of Jolt, we discovered several optimizations during the course of our formalization, some of which were caught automatically by ACL2. Consider again the formalization of SLTU in Program 4. Note that the lookup `w3` is prefixed by a `?`. The `b*` macro enforces that all bindings should be used in the body unless otherwise indicated, such as with `?`. Indeed, `w3` is not used to evaluate SLTU but this lookup is still listed in the Jolt paper, and a previous version of the Jolt codebase. The same also applies for the SLT, BGEU, and BGE instructions.[11] The efficiency gain from this optimization depends on how often these 4 instructions are used in a typical RISC-V program, since the prover only "pays" for instructions that get executed. Nevertheless, instruction overhead is linear in the number of lookups necessary; for an instruction like SLTU which originally required 8 lookups, we achieve a 12.5% speedup in prover time by saving 1 lookup.

We also discovered that all shift subtables in the Jolt codebase can be reduced in size. The maximum meaningful shift for 32-bit integers is $32 = 2^5$. Thus a shift subtable need only be size $2^8 \times 2^5 = 2^{13}$, with 2^8 indices for the chunk to be shifted and 2^5 indices for the shift parameter. However, the Jolt Rust

[11] BLT and BLTU are also RISC-V instructions which benefit from this optimization, but in the Rust implementation they are implemented by referencing the other comparison instructions.

Program 4: ACL2 Formalization of Jolt SLTU Instruction.

```
;; SLTU without subtables, just lookup semantics
(define sltu-semantics-32 ((x (unsigned-byte-p 32 x)) (y (unsigned-byte-p 32 y)))
  (b* (((unless (unsigned-byte-p 32 x)) 0)             ;; Edge cases
       ((unless (unsigned-byte-p 32 y)) 0)
       (x8-3 (part-select x :low  0 :width 8))          ;; Chunk
       (x8-2 (part-select x :low  8 :width 8))
       (x8-1 (part-select x :low 16 :width 8))
       (x8-0 (part-select x :low 24 :width 8))
       (y8-3 (part-select y :low  0 :width 8))
       (y8-2 (part-select y :low  8 :width 8))
       (y8-1 (part-select y :low 16 :width 8))
       (y8-0 (part-select y :low 24 :width 8))
       (z0   (if (< x8-0 y8-0) 1 0))                    ;; Lookup semantics
       (z1   (if (< x8-1 y8-1) 1 0))
       (z2   (if (< x8-2 y8-2) 1 0))
       (z3   (if (< x8-3 y8-3) 1 0))
       (w0   (if (= x8-0 y8-0) 1 0))
       (w1   (if (= x8-1 y8-1) 1 0))
       (w2   (if (= x8-2 y8-2) 1 0))
       (?w3  (if (= x8-3 y8-3) 1 0)))                   ;; ignore w3
      (+ z0 (* z1 w0) (* z2 w0 w1) (* z3 w0 w1 w2))))   ;; Combine

;; Correctness of SLTU intermediate semantics layer
(gl::def-gl-thm sltu-semantics-32-correctness
 :hyp (and (unsigned-byte-p 32 x) (unsigned-byte-p 32 y))
 :concl (equal (sltu-semantics-32 x y) (if (< x y) 1 0))
 :g-bindings (gl::auto-bindings (:mix (:nat x 32) (:nat y 32))))

;; Define SLTU with subtable lookups
(define sltu-32 ((x (unsigned-byte-p 32 x)) (y (unsigned-byte-p 32 y))) ...
  (b* (((unless (unsigned-byte-p 32 x)) 0)                         ;; Edge cases
       ((unless (unsigned-byte-p 32 y)) 0)
       (x8-3 (part-select x :low  0 :width 8))                      ;; Chunk
       (x8-2 (part-select x :low  8 :width 8))
       (x8-1 (part-select x :low 16 :width 8))
       (x8-0 (part-select x :low 24 :width 8))
       (y8-3 (part-select y :low  0 :width 8))
       (y8-2 (part-select y :low  8 :width 8))
       (y8-1 (part-select y :low 16 :width 8))
       (y8-0 (part-select y :low 24 :width 8))
       (indices      (create-tuple-indices (expt 2 8) (expt 2 8)))  ;; Materialize subtables
       (ltu-subtable (materialize-ltu-subtable indices))
       (eq-subtable  (materialize-eq-subtable  indices))
       (z0   (tuple-lookup x8-0 y8-0 ltu-subtable))                 ;; Perform Lookups
       (z1   (tuple-lookup x8-1 y8-1 ltu-subtable))
       (z2   (tuple-lookup x8-2 y8-2 ltu-subtable))
       (z3   (tuple-lookup x8-3 y8-3 ltu-subtable))
       (w0   (tuple-lookup x8-0 y8-0  eq-subtable))
       (w1   (tuple-lookup x8-1 y8-1  eq-subtable))
       (w2   (tuple-lookup x8-2 y8-2  eq-subtable))
       (?w3  (tuple-lookup x8-3 y8-3  eq-subtable)))                ;; ignore w3
      (+ z0 (* z1 w0) (* z2 w0 w1) (* z3 w0 w1 w2)))                ;; Combine
  ///
  ;; Equivalence between sltu-32 & its intermediate semantics version
  (defthm sltu-32-sltu-semantics-32-equiv
   (equal (sltu-32 x y) (sltu-semantics-32 x y))
   :hints (("Goal" :in-theory (enable sltu-semantics-32)))))

;; Correctness of Jolt SLTU
(defthm sltu-32-correctness
 (implies (and (unsigned-byte-p 32 x) (unsigned-byte-p 32 y))
          (equal (sltu-32 x y) (if (< x y) 1 0))))
```

implementation materializes a $2^8 \times 2^8 = 2^{16}$ subtable for shifts. Our formalization exhibits a proof that we can reduce Jolt's three shift subtables by 87.5%. This directly translates to reduced cost for shift operations, though performance gains may vary depending on programs, and integrating this change into Jolt requires additional engineering effort to handle subtables that are now of different sizes.

6 Conclusion & Future Work

We present a formal model of Lasso-style lookup arguments, and verify all 32-bit base Jolt instruction lookups using ACL2 in a highly automated and validated manner. We also demonstrate the utility of ACL2-based formalization via identifying possible optimizations, resulting in performance improvements for Jolt.

Our work takes the first step towards full formal correctness of Jolt's front-end, demonstrating that an incremental approach to front-end verification (starting with instruction lookups) is both feasible and useful. We have already started verifying the M-extension to Jolt, and plan to extend our formalization to include the other two components of Jolt's front-end (R1CS and memory-checking) in future work.

Acknowledgements

The authors would like to thank Daejun Park, Michael Zhu, Sam Ragsdale, and Srinath Setty for their helpful comments and feedback.

Disclosures. Justin Thaler is a Research Partner at a16z crypto and an investor in various blockchain-based platforms, as well as in the crypto ecosystem more broadly (for general a16z disclosures, see https://www.a16z.com/disclosures/.)

References

1. Anderson, S.E.: Bit twiddling hacks – counting bits set, in parallel. https://graphics.stanford.edu/~seander/bithacks.html#CountBitsSetParallel (2005), accessed 2024-11-08
2. Arun, A., Setty, S.T.V., Thaler, J.: Jolt: SNARKs for virtual machines via lookups. In: Joye, M., Leander, G. (eds.) EUROCRYPT 2024, Part VI. LNCS, vol. 14656, pp. 3–33. Springer, Cham (May 2024). https://doi.org/10.1007/978-3-031-58751-1_1
3. Avigad, J., Goldberg, L., Levit, D., Seginer, Y., Titelman, A.: A verified algebraic representation of cairo program execution. In: Proceedings of the 11th ACM SIGPLAN International Conference on Certified Programs and Proofs. pp. 153–165 (2022)
4. Avigad, J., Goldberg, L., Levit, D., Seginer, Y., Titelman, A.: A proof-producing compiler for blockchain applications. In: 14th International Conference on Interactive Theorem Proving (ITP 2023). Schloss Dagstuhl-Leibniz-Zentrum für Informatik (2023)

5. Bailey, B., Miller, A.: Formalizing soundness proofs of linear PCP SNARKs. In: Balzarotti, D., Xu, W. (eds.) USENIX Security 2024. USENIX Association (Aug 2024)
6. Bitansky, N., Canetti, R., Chiesa, A., Tromer, E.: From extractable collision resistance to succinct non-interactive arguments of knowledge, and back again. In: Goldwasser, S. (ed.) ITCS 2012. pp. 326–349. ACM (Jan 2012). https://doi.org/10.1145/2090236.2090263
7. Bosshard, A.G., Bootle, J., Sprenger, C.: Formal verification of the sumcheck protocol. In: 2024 IEEE 37th Computer Security Foundations Symposium (CSF). pp. 205–219. IEEE Computer Society (2024)
8. Boyer, R.S., Goldschlag, D.M., Kaufmann, M., Moore, J.S.: Functional instantiation in first-order logic. In: Artificial and Mathematical Theory of Computation (1991), https://api.semanticscholar.org/CorpusID:53804035
9. ACL2 XDOC: B*. https://www.https://www.cs.utexas.edu/~moore/acl2/manuals/latest/?topic={ACL2}____B_A2 (2024), accessed 2024-10-10
10. Campanelli, M., Faonio, A., Fiore, D., Li, T., Lipmaa, H.: Lookup arguments: Improvements, extensions and applications to zero-knowledge decision trees. In: Tang, Q., Teague, V. (eds.) PKC 2024, Part II. LNCS, vol. 14602, pp. 337–369. Springer, Cham (Apr 2024). https://doi.org/10.1007/978-3-031-57722-2_11
11. CertiK: Advanced formal verification of zero-knowledge proof blockchains. https://www.certik.com/resources/blog/advanced-formal-verification-of-zero-knowledge-proof-blockchains (2024)
12. CertiK: Advanced formal verification of zkp: A tale of two zk-bugs. https://www.certik.com/resources/blog/advanced-formal-verification-of-zkp-a-tale-of-two-zk-bugs (2024)
13. CertiK: Advanced formal verification of zkp: How zk memory was proven. https://www.certik.com/resources/blog/advanced-formal-verification-of-zkp-how-zk-memory-was-proven (2024)
14. CertiK: Advanced formal verification of zkp: Verifying a zk instruction. https://www.certik.com/resources/blog/advanced-formal-verification-of-zkp-verifying-a-zk-instruction (2024)
15. Certik - securing the web3 world. https://www.certik.com/ (2024)
16. CertiK: Verification of zkwasm in coq. https://github.com/CertiKProject/zkwasm-fv (2024)
17. Chaliasos, S., Ernstberger, J., Theodore, D., Wong, D., Jahanara, M., Livshits, B.: SoK: What don't we know? Understanding security vulnerabilities in SNARKs. In: Balzarotti, D., Xu, W. (eds.) USENIX Security 2024. USENIX Association (Aug 2024)
18. Chin, C., Wu, H., Chu, R., Coglio, A., McCarthy, E., Smith, E.: Leo: A programming language for formally verified, zero-knowledge applications. Cryptology ePrint Archive, Report 2021/651 (2021), https://eprint.iacr.org/2021/651
19. Ciobotaru, O., Peter, M., Velichkov, V.: The last challenge attack: Exploiting a vulnerable implementation of the fiat-shamir transform in a KZG-based SNARK. Cryptology ePrint Archive, Report 2024/398 (2024), https://eprint.iacr.org/2024/398
20. Coglio, A.: Ethereum's recursive length prefix in ACL2. arXiv preprint arXiv:2009.13769 (2020)
21. Coglio, A., McCarthy, E., Smith, E., Chin, C., Gaddamadugu, P., Dellepere, M.: Compositional formal verification of zero-knowledge circuits. Cryptology ePrint Archive, Report 2023/1278 (2023), https://eprint.iacr.org/2023/1278

22. The coq proof assistant. https://coq.inria.fr/ (2024)
23. Dao, Q., Miller, J., Wright, O., Grubbs, P.: Weak fiat-shamir attacks on modern proof systems. In: 2023 IEEE Symposium on Security and Privacy. pp. 199–216. IEEE Computer Society Press (May 2023). https://doi.org/10.1109/SP46215.2023.10179408
24. Davis, J., Myreen, M.O.: The reflective milawa theorem prover is sound (down to the machine code that runs it). Journal of Automated Reasoning **55**(2), 117–183 (Jun 2015). https://doi.org/10.1007/s10817-015-9324-6, http://dx.doi.org/10.1007/s10817-015-9324-6
25. Davis, J.C.: A self-verifying theorem prover. Ph.D. thesis, The Univeristy of Texas at Austin (2009)
26. Eagen, L., Fiore, D., Gabizon, A.: cq: Cached quotients for fast lookups. Cryptology ePrint Archive, Report 2022/1763 (2022), https://eprint.iacr.org/2022/1763
27. Firsov, D., Livshits, B.: The ouroboros of ZK: Why verifying the verifier unlocks longer-term ZK innovation. Cryptology ePrint Archive, Report 2024/768 (2024), https://eprint.iacr.org/2024/768
28. Gabizon, A.: On the security of the BCTV pinocchio zk-SNARK variant. Cryptology ePrint Archive, Report 2019/119 (2019), https://eprint.iacr.org/2019/119
29. Gabizon, A., Khovratovich, D.: flookup: Fractional decomposition-based lookups in quasi-linear time independent of table size. Cryptology ePrint Archive, Report 2022/1447 (2022), https://eprint.iacr.org/2022/1447
30. Gabizon, A., Williamson, Z.J.: plookup: A simplified polynomial protocol for lookup tables. Cryptology ePrint Archive, Report 2020/315 (2020), https://eprint.iacr.org/2020/315
31. Gennaro, R., Gentry, C., Parno, B., Raykova, M.: Quadratic span programs and succinct NIZKs without PCPs. In: Johansson, T., Nguyen, P.Q. (eds.) EUROCRYPT 2013. LNCS, vol. 7881, pp. 626–645. Springer, Berlin, Heidelberg (May 2013). https://doi.org/10.1007/978-3-642-38348-9_37
32. Goel, S., Hunt, W.A., Kaufmann, M., Ghosh, S.: Simulation and formal verification of x86 machine-code programs that make system calls. In: 2014 Formal Methods in Computer-Aided Design (FMCAD). pp. 91–98 (2014). https://doi.org/10.1109/FMCAD.2014.6987600
33. Goldberg, L., Papini, S., Riabzev, M.: Cairo – a Turing-complete STARK-friendly CPU architecture. Cryptology ePrint Archive, Report 2021/1063 (2021), https://eprint.iacr.org/2021/1063
34. Goldwasser, S., Micali, S., Rackoff, C.: The knowledge complexity of interactive proof-systems (extended abstract). In: 17th ACM STOC. pp. 291–304. ACM Press (May 1985). https://doi.org/10.1145/22145.22178
35. Haböck, U.: Multivariate lookups based on logarithmic derivatives. Cryptology ePrint Archive, Report 2022/1530 (2022), https://eprint.iacr.org/2022/1530
36. Hader, T., Ozdemir, A.: An smt-lib theory of finite fields. arXiv preprint arXiv:2407.21169 (2024)
37. Plonkish arithmetization - the halo2 book. https://zcash.github.io/halo2/concepts/arithmetization.html
38. Hunt, W.A., Kaufmann, M., Moore, J.S., Slobodova, A.: Industrial hardware and software verification with ACL2. Philosophical Transactions of the Royal Society of London Series A **375**(2104) (September 2017). https://doi.org/10.1098/rsta.2015.0399
39. Isabel, M., Rodríguez-Núñez, C., Rubio, A.: Scalable verification of zero-knowledge protocols. In: 2024 IEEE Symposium on Security and Privacy (SP). pp. 133–133. IEEE Computer Society (2024)

40. Jolt: The simplest and most extensible zkVM. https://github.com/a16z/jolt (2024)
41. Jolt m-extension documentation. https://jolt.a16zcrypto.com/how/m-extension.html
42. Kaufmann, M., Manolios, P., Moore, J.S.: Computer-Aided Reasoning. Springer US (2000). https://doi.org/10.1007/978-1-4615-4449-4, http://dx.doi.org/10.1007/978-1-4615-4449-4
43. Kaufmann, M., Moore, J.S.: ACL2 home page. https://cs.utexas.edu/~moore/acl2/ (1997), accessed 2024-10-10
44. Kaufmann, M., Moore, J.S.: Limited second-order functionality in a first-order setting. Journal of Automated Reasoning **64**(3), 391–422 (Dec 2018). https://doi.org/10.1007/s10817-018-09505-9, http://dx.doi.org/10.1007/s10817-018-09505-9
45. Kaufmann, M., Moore, J.S.: Iteration in ACL2. Electronic Proceedings in Theoretical Computer Science **327**, 16–31 (Sep 2020). https://doi.org/10.4204/eptcs.327.2, http://dx.doi.org/10.4204/EPTCS.327.2
46. Liu, H.: Formal Specification and Verification of a JVM and its Bytecode Verifier. The University of Texas at Austin (2006)
47. Liu, J., Kretz, I., Liu, H., Tan, B., Wang, J., Sun, Y., Pearson, L., Miltner, A., Dillig, I., Feng, Y.: Certifying zero-knowledge circuits with refinement types. In: 2024 IEEE Symposium on Security and Privacy (SP). pp. 1741–1759. IEEE (2024)
48. Liu, T., Zhang, Z., Zhang, Y., Hu, W., Zhang, Y.: Ceno: Non-uniform, segment and parallel zero-knowledge virtual machine. Cryptology ePrint Archive, Report 2024/387 (2024), https://eprint.iacr.org/2024/387
49. Argument computer corporation. https://argument.xyz/ (2024)
50. Addition of the m-extension to jolt. https://github.com/a16z/jolt/commit/5cff00530030b01d21fe89d9720d0edab43c73ee (2024)
51. Formal verification of zksync's on-chain verifier. https://x.com/zksync/status/1835736788385538507 (2024)
52. Micali, S.: CS proofs (extended abstracts). In: 35th FOCS. pp. 436–453. IEEE Computer Society Press (Nov 1994). https://doi.org/10.1109/SFCS.1994.365746
53. Moura, L.d., Ullrich, S.: The Lean 4 theorem prover and programming language. In: Automated Deduction–CADE 28: 28th International Conference on Automated Deduction, Virtual Event, July 12–15, 2021, Proceedings 28. pp. 625–635. Springer (2021)
54. Myreen, M.O., Davis, J.: A verified runtime for a verified theorem prover. In: van Eekelen, M., Geuvers, H., Schmaltz, J., Wiedijk, F. (eds.) Interactive Theorem Proving. pp. 265–280. Springer Berlin Heidelberg, Berlin, Heidelberg (2011)
55. Myreen, M.O., Davis, J.: The reflective milawa theorem prover is sound. In: Klein, G., Gamboa, R. (eds.) Interactive Theorem Proving. pp. 421–436. Springer International Publishing, Cham (2014)
56. Nexus. https://nexus.xyz/ (2024)
57. Nguyen, W.D., Boneh, D., Setty, S.: Revisiting the nova proof system on a cycle of curves. In: 5th Conference on Advances in Financial Technologies (2023)
58. Ozdemir, A., Kremer, G., Tinelli, C., Barrett, C.: Satisfiability modulo finite fields. In: International Conference on Computer Aided Verification. pp. 163–186. Springer (2023)
59. Ozdemir, A., Pailoor, S., Bassa, A., Ferles, K., Barrett, C., Dillig, I.: Split gröbner bases for satisfiability modulo finite fields. In: International Conference on Computer Aided Verification. pp. 3–25. Springer (2024)

60. Pailoor, S., Chen, Y., Wang, F., Rodríguez, C., Van Geffen, J., Morton, J., Chu, M., Gu, B., Feng, Y., Dillig, I.: Automated detection of under-constrained circuits in zero-knowledge proofs. Proceedings of the ACM on Programming Languages **7**(PLDI), 1510–1532 (2023)
61. Papini, S., Haböck, U.: Improving logarithmic derivative lookups using GKR. Cryptology ePrint Archive, Report 2023/1284 (2023), https://eprint.iacr.org/2023/1284
62. Parno, B.: A note on the unsoundness of vnTinyRAM's SNARK. Cryptology ePrint Archive, Report 2015/437 (2015), https://eprint.iacr.org/2015/437
63. Polygon zkevm. https://polygon.technology/polygon-zkevm (2024)
64. Posen, J., Kattis, A.A.: Caulk+: Table-independent lookup arguments. Cryptology ePrint Archive, Report 2022/957 (2022), https://eprint.iacr.org/2022/957
65. Ragsdale, S., Zhu, M., Thaler, J.: Understanding Lasso and Jolt, from theory to code. https://a16zcrypto.com/posts/article/building-on-lasso-and-jolt/ (2024)
66. Risc-v isa manual, version 20240411. https://github.com/riscv/riscv-isa-manual/releases/tag/20240411 (2024)
67. Risc zero. https://risczero.com/ (2024)
68. Scroll - native zkevm layer 2 for ethereum. https://scroll.io/ (2024)
69. Setty, S.: Spartan: Efficient and general-purpose zkSNARKs without trusted setup. In: Micciancio, D., Ristenpart, T. (eds.) CRYPTO 2020, Part III. LNCS, vol. 12172, pp. 704–737. Springer, Cham (Aug 2020). https://doi.org/10.1007/978-3-030-56877-1_25
70. Setty, S.T.V., Thaler, J., Wahby, R.S.: Unlocking the lookup singularity with Lasso. In: Joye, M., Leander, G. (eds.) EUROCRYPT 2024, Part VI. LNCS, vol. 14656, pp. 180–209. Springer, Cham (May 2024). https://doi.org/10.1007/978-3-031-58751-1_7
71. Starkware. https://starkware.co/ (2024)
72. Succinct. https://succinct.xyz/ (2024)
73. Swords, S., Davis, J.: Bit-blasting ACL2 theorems. Electronic Proceedings in Theoretical Computer Science **70**, 84–102 (Oct 2011). https://doi.org/10.4204/eptcs.70.7, http://dx.doi.org/10.4204/EPTCS.70.7
74. Swords, S.O.: A verified framework for symbolic execution in the ACL2 theorem prover. Ph.D. thesis, The Univeristy of Texas at Austin (2010)
75. Thaler, J.: Approaching the 'lookup singularity': Introducing Lasso and Jolt. https://a16zcrypto.com/posts/article/introducing-lasso-and-jolt/ (2024)
76. Thaler, J.: Understanding Jolt: Clarifications and reflections. https://a16zcrypto.com/posts/article/understanding-jolt-clarifications-and-reflections/ (2024)
77. Valida. https://github.com/valida-xyz (2024)
78. Wang, F., the 0xParc Team: Ecne: Automated verification of zk circuits. https://0xparc.org/blog/ecne (2024)
79. Webassembly specification. https://webassembly.github.io/spec/core/ (2024)
80. Wen, H., Stephens, J., Chen, Y., Ferles, K., Pailoor, S., Charbonnet, K., Dillig, I., Feng, Y.: Practical security analysis of zero-knowledge proof circuits. In: Balzarotti, D., Xu, W. (eds.) USENIX Security 2024. USENIX Association (Aug 2024)
81. Whitehat, B.: Lookup singularity. https://zkresear.ch/t/lookup-singularity/65/7 (2024)
82. ACL2 XDOC: User manual for the ACL2 theorem prover. https://www.cs.utexas.edu/~moore/acl2/manuals/current/manual/ (2024), accessed 2024-10-10

83. ACL2 y86 model. https://github.com/acl2/acl2/tree/master/books/models/y86 (2014), accessed 2024-10-10
84. Zapico, A., Buterin, V., Khovratovich, D., Maller, M., Nitulescu, A., Simkin, M.: Caulk: Lookup arguments in sublinear time. In: Yin, H., Stavrou, A., Cremers, C., Shi, E. (eds.) ACM CCS 2022. pp. 3121–3134. ACM Press (Nov 2022). https://doi.org/10.1145/3548606.3560646
85. Zapico, A., Gabizon, A., Khovratovich, D., Maller, M., Ràfols, C.: Baloo: Nearly optimal lookup arguments. Cryptology ePrint Archive, Report 2022/1565 (2022), https://eprint.iacr.org/2022/1565
86. Zkm. https://www.zkm.io/ (2024)
87. Delphinus lab. https://delphinuslab.com/ (2024)
88. Zksync. https://zksync.io/ (2024)

A Subtables Specification

In this section, we give a full specification of all subtables used in the RV32IM version of Jolt.

A.1 Notation

We denote $[a, b] := \{a, a+1, \ldots, b\}$, and for a vector x, we denote $x_{[a,b]} := (x_a, \ldots, x_b)$.

- M is the size of the subtable, assumed to be of the form $M = 2^{2m}$ for some $m \in \mathbb{N}$. Here m is the number of bits for each operand, and we have $m = 8$ for all use cases of subtables in Jolt instructions. However, for full generality, we will parametrize each subtable by m.
- We write x as the first operand. It is a m-bit natural number, whose binary representation (in big-endian form) is a vector $(x_0, x_1, \ldots, x_{m-1}) \in \{0,1\}^m$. Note that x_0 denotes the *most* significant bit. Similarly, we write $X = (X_0, X_1, \ldots, X_{m-1})$ to be the vector of variables in the multilinear extension (over the finite field $\mathbb{F}$).
- We write y as the second operand. It is a m-bit natural number, whose binary representation is a vector $(y_0, y_1, \ldots, y_{m-1}) \in \{0,1\}^m$. Similarly, we write $Y = (Y_0, Y_1, \ldots, Y_{m-1})$ to be the vector of variables in the multilinear extension.
- For some subtables, it is more convenient to think of a single vector z that is the concatenation of x and y. We write $z = x \| y$ to denote the concatenation of x and y, indexed as $z = (z_{2m-1}, z_{2m-2}, \ldots, z_1, z_0) \in \{0,1\}^{2m}$. Similarly, we write $Z = (Z_0, Z_1, \ldots, Z_{2m-1})$ to be the vector of variables in the multilinear extension.
- The subtable value is a natural number having at most $2m$ bits, embedded in a larger prime field. The indexing is done in the natural way, by interpreting the concatenation $z := x \| y$ as a $2m$-bit natural number.

Some subtables (such as shift) are further parametrized by additional values:

- $W \in \{32, 64\}$ is the word size.
- i: For shift instructions, this is the index of the chunk being shifted.

A.2 List of Subtables

For each subtable, recall that we write $x, y \in \{0,1\}^m$, $z := x \| y \in \{0,1\}^{2m}$ for the input, and similarly write $X = (X_0, \ldots, X_{m-1})$, $Y = (Y_0, \ldots, Y_{m-1})$, $Z = X \| Y = (Z_0, \ldots, Z_{2m-1})$ for the list of variables.

We list the subtables roughly in order of complexity and following the order of their appearance in the instructions (listed in Appendix B). For each subtable, we list the function used in materializing the subtable, the ACL2 implementation of the function, and the multilinear extension of the function. We note that four subtables, namely DivByZero, LeftIsZero, RightIsZero, and ZeroLSB, are used in Jolt only in proving the RISC-V M-extension.

1. **And**:
   - **Function:**

$$\mathsf{And}_m(x,y) = (x_0 \wedge y_0, x_1 \wedge y_1, \ldots, x_{m-1} \wedge y_{m-1})$$

   - **ACL2:** `(logand x y)`
   - **Multilinear Extension:**

$$\widetilde{\mathsf{And}}_m(X,Y) = \sum_{i=0}^{m-1} 2^{m-i-1} \cdot X_i \cdot Y_i$$

2. **Or**:
   - **Function:**

$$\mathsf{Or}_m(x,y) = (x_0 \vee y_0, x_1 \vee y_1, \ldots, x_{m-1} \vee y_{m-1})$$

   - **ACL2:** `(logior x y)`
   - **Multilinear Extension:**

$$\widetilde{\mathsf{Or}}_m(X,Y) = \sum_{i=0}^{m-1} 2^{m-i-1} \cdot (X_i + Y_i - X_i \cdot Y_i)$$

3. **Xor**:
   - **Function:**

$$\mathsf{Xor}_m(x,y) = (x_0 \oplus y_0, x_1 \oplus y_1, \ldots, x_{m-1} \oplus y_{m-1})$$

   - **ACL2:** `(logxor x y)`
   - **Multilinear Extension:**

$$\widetilde{\mathsf{Xor}}_m(X,Y) = \sum_{i=0}^{m-1} 2^{m-i-1} \cdot ((1 - X_i) \cdot Y_i + X_i \cdot (1 - Y_i))$$

4. **DivByZero**:
   - **Function:**

$$\mathsf{DivByZero}_m(x,y) = \begin{cases} 1 & \text{if } (x_i, y_i) = (0,1) \text{ for all } i \\ 0 & \text{otherwise} \end{cases}$$

   - **ACL2:** `(if (and (= x 0) (= y (1- (expt 2 m)))) 1 0)`
   - **Multilinear Extension:**

$$\widetilde{\mathsf{DivByZero}}_m(X,Y) = \prod_{i=0}^{m-1} ((1 - X_i) \cdot Y_i)$$

5. **Eq**:

- **Function:**

$$\mathsf{Eq}_m(x, y) = \begin{cases} 1 & \text{if } x = y \\ 0 & \text{otherwise} \end{cases}$$

- **ACL2:** `(if (= x y) 1 0)`
- **Multilinear Extension:**

$$\widetilde{\mathsf{Eq}}_m(X, Y) = \prod_{i=0}^{m-1} ((1 - X_i) \cdot (1 - Y_i) + X_i \cdot Y_i)$$

6. **EqAbs**:
   - **Function:**

$$\mathsf{EqAbs}_m(x, y) = \begin{cases} 1 & \text{if } x_i = y_i \text{ for all } i > 0 \\ 0 & \text{otherwise} \end{cases}$$

   - **ACL2:** `(if (= (loghead (1- m) x) (loghead (1- m) y)) 1 0)`
   - **Multilinear Extension:**

$$\widetilde{\mathsf{EqAbs}}_m(X, Y) = \prod_{i=1}^{m-1} ((1 - X_i) \cdot (1 - Y_i) + X_i \cdot Y_i)$$

7. **Identity**:
   - **Function:**

$$\mathsf{Id}_m(z) = z$$

   - **ACL2:** `(z)`
   - **Multilinear Extension:**

$$\widetilde{\mathsf{Id}}_m(Z) = \sum_{i=0}^{m-1} 2^{m-i-1} \cdot Z_i$$

8. **LeftIsZero**:
   - **Function:**

$$\mathsf{LeftIsZero}_m(x, y) = \begin{cases} 1 & \text{if } x = 0 \\ 0 & \text{otherwise} \end{cases}$$

   - **ACL2:** `(if (= x 0) 1 0)`
   - **Multilinear Extension:**

$$\widetilde{\mathsf{LeftIsZero}}_m(X, Y) = \prod_{i=0}^{m-1} (1 - X_i)$$

9. **LeftMSB**:
   - **Function:**

$$\mathsf{LeftMSB}_m(x, y) = x_0$$

- **ACL2:** `(logbit (1- m) x)`
- **Multilinear Extension:**

$$\widetilde{\mathsf{LeftMSB}}_m(X,Y) = X_0$$

10. **LtAbs**:
    - **Function:**

$$\mathsf{LtAbs}_m(x,y) = \begin{cases} 1 & \text{if } x_{[1,m-1]} < y_{[1,m-1]} \\ 0 & \text{otherwise} \end{cases}$$

    - **ACL2:** `(if (< (loghead (1- m) x) (loghead (1- m) y)) 1 0)`
    - **Multilinear Extension:**

$$\widetilde{\mathsf{LtAbs}}_m(X,Y) = \sum_{i=1}^{m-1} (1 - X_i) \cdot Y_i \cdot \prod_{j=0}^{i-1} ((1 - X_j) \cdot (1 - Y_j) + X_j \cdot Y_j)$$

11. **Ltu**:
    - **Function:**

$$\mathsf{Ltu}_m(x,y) = \begin{cases} 1 & \text{if } x < y \\ 0 & \text{otherwise} \end{cases}$$

    - **ACL2:** `(if (< x y) 1 0)`
    - **Multilinear Extension:**

$$\widetilde{\mathsf{Ltu}}_m(X,Y) = \sum_{i=0}^{m-1} (1 - X_i) \cdot Y_i \cdot \prod_{j=0}^{i-1} ((1 - X_j) \cdot (1 - Y_j) + X_j \cdot Y_j)$$

12. **RightIsZero**:
    - **Function:**

$$\mathsf{RightIsZero}_m(x,y) = \begin{cases} 1 & \text{if } y = 0 \\ 0 & \text{otherwise} \end{cases}$$

    - **ACL2:** `(if (= y 0) 1 0)`
    - **Multilinear Extension:**

$$\widetilde{\mathsf{RightIsZero}}_m(X,Y) = \prod_{i=0}^{m-1} (1 - Y_i)$$

13. **RightMSB**:
    - **Function:**

$$\mathsf{RightMSB}_m(x,y) = y_0$$

    - **ACL2:** `(logbit (1- m) y)`
    - **Multilinear Extension:**

$$\widetilde{\mathsf{RightMSB}}_m(X,Y) = Y_0$$

14. **ZeroLSB**:
    - **Function:**
    $$\mathsf{ZeroLSB}_m(z) = z - (z \bmod 2)$$
    - **ACL2:** `(- z (mod z 2))`
    - **Multilinear Extension:**
    $$\widetilde{\mathsf{ZeroLSB}}_m(Z) = \sum_{i=1}^{m-1} 2^{m-i} \cdot Z_i$$

15. **TruncateOverflow**: This subtable is further parametrized by a word size W. Informally, this subtable truncates $z = x\|y$ to w bits and then zero-extends it to $2m$ bits, where $w = W \bmod (2m)$ is the number of overflow bits.
    - **Function:**
    $$\mathsf{TruncateOverflow}_{m,W}(z) = z \bmod 2^{W \bmod (2m)}.$$
    - **ACL2:** `(mod z (expt 2 (mod W (* 2 m))))`
    - **Multilinear Extension:**
    $$\widetilde{\mathsf{TruncateOverflow}}_{m,W}(Z) = \sum_{i=0}^{(W \bmod 2m)-1} 2^i \cdot Z_{m-i-1}.$$

16. **SignExtend**: This subtable is further parametrized by a width $w \leq 2m$. Informally, this subtable returns either all-zero or all-one (for w bits) depending on the w-th least significant bit of $z = x\|y$ (interpreted as a sign bit).
    - **Function:**
    $$\mathsf{SignExtend}_{m,w}(z) = \begin{cases} 0 & \text{if } z_{2m-w} = 0 \\ 2^w - 1 & \text{if } z_{2m-w} = 1 \end{cases}$$
    - **ACL2:** `(* (logbit (1- w) z) (1- (expt 2 w)))`
    - **Multilinear Extension:**
    $$\widetilde{\mathsf{SignExtend}}_{m,w}(Z) = (2^w - 1) \cdot Z_{2m-w}.$$

17. **Sll**: This subtable is further parametrized by two values, the chunk index i and the word size W. Informally, this subtable assumes that x is the i-th chunk of some W-bit number[12], shifts it left by $y \bmod W$ bits, then truncates the result to be $W - m \cdot i$ bits.

[12] Each chunk has m bits, and the indexing goes from left-to-right, so the first chunk is the most significant bits of x

- **Function:**

$$\mathsf{Sll}_{m,i,W}(x,y) = (x \ll (y \bmod W)) \bmod 2^{W-m\cdot i}$$

- **ACL2:** `(mod (ash x (mod y W)) (expt 2 (- W (* i m))))`
- **Multilinear Extension:** Let $m' = \min(m, \lceil \log_2(W) \rceil)$ and $n_{\mathsf{out}} = \min(m, \max(0, k + m \cdot (i+1) - W))$ denotes the number of bits that goes out of range. Then:

$$\widetilde{\mathsf{Sll}}_{m,i,W}(X,Y) = \sum_{k=0}^{2^{m'}-1} \widetilde{\mathsf{Eq}}_{m'}(Y,k) \cdot \left(\sum_{j=0}^{m-n_{\mathsf{out}}-1} 2^{k+j} \cdot X_{m-j-1} \right).$$

18. **Srl**: This subtable is further parametrized by two values, the chunk index i and the word size W. Informally, this subtable assumes that x is the i-th chunk of some W-bit number, shifts it left by $m \cdot i$ bits to align with the pre-chunk position, then shifts it right by $y \bmod W$ bits.
    - **Function:**

$$\mathsf{Srl}_{m,i,W}(x,y) = (x \ll m \cdot i) \gg (y \bmod W).$$

    - **ACL2:** `(ash (ash x (* i m)) (- (mod y W)))`
    - **Multilinear Extension:** Let $m' = \min(m, \lceil \log_2(W) \rceil)$, $m'' = \min(m, W - m \cdot i)$, and $n_{\mathsf{out}} = \min(m, \max(0, k + m \cdot (i+1) - W))$ denotes the number of bits that goes out of range. Then:

$$\widetilde{\mathsf{Srl}}_{m,i,W}(X,Y) = \sum_{k=0}^{2^{m'}-1} \widetilde{\mathsf{Eq}}_{m'}(Y,k) \cdot \left(\sum_{j=n_{\mathsf{out}}}^{m''-1} 2^{m\cdot(i-1)-k+j} \cdot X_{m-j-1} \right).$$

19. **SraSign**: This subtable is further parametrized by the word size W. Informally, this subtable returns $x_{(m-1)-((W-1) \bmod m)}$ (interpreted as a sign bit) duplicated $y \bmod W$ times in the most significant bits of the result (which is a W-bit number).
    - **Function:** Let $i_{\mathsf{sign}} = m - 1 - ((W-1) \bmod m)$. Then:

$$\mathsf{SraSign}_{m,W}(x,y) = \begin{cases} 0 & \text{if } x_{i_{\mathsf{sign}}} = 0 \\ \sum_{i=0}^{(y \bmod W)-1} 2^{W-1-i} & \text{if } x_{i_{\mathsf{sign}}} = 1 \end{cases}.$$

    - **ACL2:**

```
(* (logbit (1- m) x)
   (ash (1- (expt 2 (logand (1- W) y)))
        (- W (logand (1- W) y))))
```

    - **Multilinear Extension:** Let $m' = \min(m, \lceil \log_2(W) \rceil)$. Then:

$$\widetilde{\mathsf{SraSign}}_{m,W}(X,Y) = \sum_{k=0}^{2^{m'}-1} \widetilde{\mathsf{Eq}}_{m'}(Y,k) \cdot X_{i_{\mathsf{sign}}} \cdot \left(\sum_{j=0}^{k-1} 2^{W-1-j} \right).$$

B Instructions Specification

In this section, we give a full specification of all (single-cycle) instructions used in the RV32IM version of Jolt. We omit the specification of multi-cycle instructions, which are decomposed into a series of single-cycle instructions. For more details, see [41].

B.1 Notation & Helper Functions

We use the following notation for the instructions:

- $W \in \{32, 64\}$ is the word size.
- $x, y \in \{0,1\}^W$ are the first and second input operands, respectively. For some instructions, there is only one operand, in which case the other operand is set to zero (and not involved in the lookup procedure).
- Signed operands are interpreted in two's complement form, so that addition, subtraction, and multiplication have the same result for both signed and unsigned interpretations of the operands.
- $M \in \mathbb{N}$ is the size of the subtables, so that $m := \log_2(M)/2$ is half of the number of bits in each chunk. For all Jolt instructions, we have $M = 2^{16}$ and $m = 8$.
- $C \in \mathbb{N}$ is the number of chunks that each operand is split into. Except where otherwise specified, we have $C = 4$ for $W = 32$ and $C = 8$ for $W = 64$. Note that $W = C \cdot m$ for all instructions, unless otherwise specified.

Note that in the Rust codebase [40], regardless of whether $W = 32$ or $W = 64$, the Rust code represents operands as `u64`, the type of unsigned 64-bit integers.

We also introduce the following helper functions that are used in instructions. For each function, we give the mathematical definition, followed by the ACL2 and Rust implementations. For our formalization to be correct, the ACL2 and Rust definitions must be equivalent to the mathematical definition. This is currently checked by inspection for each function; future work will include a verified connection.

1. **Truncate:** Given word size $W \in \mathbb{N}$, we define the *truncate* function as:

   $$\mathsf{Truncate}_W(z) = z \bmod 2^W \in \{0,1\}^W. \tag{10}$$

   Note that if z has less than W bits, it will be zero-extended to W bits.

   In ACL2, we represent this function in many different ways: `(loghead W z)`, `(mod z (expt 2 W))`, or `(part-select z :low 0 :width W)`. Given ACL2's automated nature, we can use any of these interchangeably with automatic proofs that they are equivalent.

   In Rust, this function is also implemented differently (and often implicitly) depending on the situation. For instance, in the shift instructions, it is written as `y % 32`, while in the add instructions it is implicitly applied after addition as `x.overflowing_add(y).0`.

2. **Sign extension:** Given parameters $W, n \in \mathbb{N}$, we define the *sign extension* function as:

$$\mathsf{SignExtend}_{W,n}(z) = \begin{cases} (z \bmod 2^W) & \text{if } z_{n-1} = 0 \\ (z \bmod 2^W) + \sum_{i=n}^{W-1} 2^i & \text{if } z_{n-1} = 1 \end{cases} \tag{11}$$

   In other words, we take the n-th bit of z to be the sign bit, and fill it in the remaining bits from n to $W-1$. If $n > W$, this simply truncates z to W bits.
   Similar to $\mathsf{Truncate}$, this function can be implemented in several different ways in both ACL2 and Rust; we assume that the semantics of the function is clear whenever it is used.
3. **Chunk:** Given parameters $m, C \in \mathbb{N}$, we define the *chunking* function as:

$$\mathsf{Chunk}_{m,C}(z) = [z_0, z_1, \ldots, z_{C-1}] \in (\{0,1\}^m)^C . \tag{12}$$

   Here we write $\mathsf{Truncate}_{C\cdot m}(z) = z_0 \| z_1 \| \ldots \| z_{C-1}$, with $z_i \in \{0,1\}^m$ for all i. In ACL2, this function is often invoked with explicit values for m and C. For instance, the chunking for the ADD instruction is implemented as

```
(z8-3 (part-select z :low  0 :width 16))
(z8-2 (part-select z :low 16 :width 16))
(z8-1 (part-select z :low 32 :width 16))
(z8-0 (part-select z :low 48 :width 16))
```

   where `z` is the addition result `(+ x y)`. Another example of chunking is in the SLTU instruction Program 4.
   In Rust, this function is implemented as:

```rust
pub fn chunk_operand(x: u64, C: usize, chunk_len: usize) -> Vec<u64> {
    let bit_mask = (1 << chunk_len) - 1;
    (0..C)
        .map(|i| {
            let shift = ((C - i - 1) * chunk_len) as u32;
            x.checked_shr(shift).unwrap_or(0) & bit_mask
        })
        .collect()
}
```

4. **Chunk & interleave:** Given parameters $m, C \in \mathbb{N}$, we define the *chunk & interleave* function as:

$$\mathsf{ChunkInterleave}_{m,C}(x, y) = [x_0 \| y_0, \ldots, x_{C-1} \| y_{C-1}] \in \left(\{0,1\}^{2m}\right)^C , \tag{13}$$

   where $\mathsf{Chunk}_{m,C}(x) = [x_0, x_1, \ldots, x_{C-1}]$ and $\mathsf{Chunk}_{m,C}(y) = [y_0, y_1, \ldots, y_{C-1}]$.

   In ACL2, this function is implicit in the arranging of input chunks for lookups. In Program 4 for SLTU, for instance, we first apply $\mathsf{Chunk}_{m,C}$ to both operands, and then arrange the chunks in the desired order for the `tuple-lookup` function.
   In Rust, this function is implemented as:

```rust
pub fn chunk_and_concatenate_operands(x: u64, y: u64, C: usize, log_M: usize) ->
    Vec<usize> {
    let operand_bits: usize = log_M / 2;

    #[cfg(test)]
    {
        let max_operand_bits = C * log_M / 2;
        if max_operand_bits != 64 {
            // if 64, handled by normal overflow checking
            let max_operand: u64 = (1 << max_operand_bits) - 1;
            assert!(x <= max_operand);
            assert!(y <= max_operand);
        }
    }

    let operand_bit_mask: usize = (1 << operand_bits) - 1;
    (0..C)
        .map(|i| {
            let shift = ((C - i - 1) * operand_bits) as u32;
            let left = x.checked_shr(shift).unwrap_or(0) as usize & operand_bit_mask;
            let right = y.checked_shr(shift).unwrap_or(0) as usize & operand_bit_mask;
            (left << operand_bits) | right
        })
        .collect()
}
```

5. **Chunk for shift:** Given parameters $m, C \in \mathbb{N}$, we define the *chunk for shift* function as:

$$\mathsf{ChunkForShift}_{m,C}(x, y) = [x_0 \| y_{C-1}, x_1 \| y_{C-1}, \ldots, x_{C-1} \| y_{C-1}] \in \left(\{0,1\}^{2m}\right)^C, \tag{14}$$

where $\mathsf{Chunk}_{m,C}(x) = [x_0, x_1, \ldots, x_{C-1}]$ and $\mathsf{Chunk}_{m,C}(y) = [y_0, y_1, \ldots, y_{C-1}]$.

In ACL2, this function is implicit in the arranging of input chunks for lookups. For the SLL instruction, the chunks are arranged as follows:

```lisp
        (x8-3 (part-select x :low  0 :width 8))
        (x8-2 (part-select x :low  8 :width 8))
        (x8-1 (part-select x :low 16 :width 8))
        (x8-0 (part-select x :low 24 :width 8))
        (y8-3 (part-select y :low 0 :width 8))
        ;; perform lookups
        (u8-3 (tuple-lookup x8-3 y8-3 slli-subtable-3))
        (u8-2 (tuple-lookup x8-2 y8-3 slli-subtable-2))
        (u8-1 (tuple-lookup x8-1 y8-3 slli-subtable-1))
        (u8-0 (tuple-lookup x8-0 y8-3 slli-subtable-0))
```

In Rust, this function is implemented as:

```rust
pub fn chunk_and_concatenate_for_shift(x: u64, y: u64, C: usize, log_M: usize) ->
    Vec<usize> {
    let operand_bits: usize = log_M / 2;
    let operand_bit_mask: usize = (1 << operand_bits) - 1;

    let y_lowest_chunk: usize = y as usize & operand_bit_mask;

    (0..C)
        .map(|i| {
            let shift = ((C - i - 1) * operand_bits) as u32;
            let left = x.checked_shr(shift).unwrap_or(0) as usize & operand_bit_mask;
```

```
            (left << operand_bits) | y_lowest_chunk
        })
        .collect()
}
```

6. **Concatenate:** Given a sequence of words $(x_0, x_1, \ldots, x_{C-1}) \in (\{0,1\}^m)^C$, we define the *concatenate* function as:

$$\begin{aligned} \mathsf{Concatenate}_{m,C}(x_0, x_1, \ldots, x_{C-1}) &= \sum_{i=0}^{C-1} x_{C-1-i} \cdot 2^{i \cdot m} \\ &= x_0 \| x_1 \| \ldots \| x_{C-1} \in \{0,1\}^{C \cdot m}. \end{aligned} \tag{15}$$

ACL2: this depends on particular 'm' and 'C'; an example from the AND instructions is:

```
(merge-4-u16s z0 z1 z2 z3)
```

In Rust, this function is implemented as:

```
pub fn concatenate_lookups<F: JoltField>(vals: &[F], C: usize, operand_bits: usize) ->
     F {
    assert_eq!(vals.len(), C);

    let mut sum = F::zero();
    let mut weight = F::one();
    let shift = F::from_u64(1u64 << operand_bits).unwrap();
    for i in 0..C {
        sum += weight * vals[C - i - 1];
        weight *= shift;
    }
    sum
}
```

B.2 List of Instructions in the base (I) instruction set

We now list all Jolt instructions, roughly by increasing complexity. Unless otherwise specified, all instructions have $m = 8$, $W \in \{32, 64\}$, and $C = W/m \in \{4, 8\}$.

Note that Jolt instructions are not in one-to-one correspondence with RISC-V instructions; for instance, the ADD instruction in Jolt is used internally to prove each of the ADD, ADDI, LUI, and AUIPC instructions in the RV32I instruction set. All these RISC-V instructions have the same addition semantics, with the only difference being which values are added together. The routing of these values are handled by other parts of Jolt's constraint system, which is out of scope for this paper.

1. **AND**:
   - **Operands:** $x, y \in \{0,1\}^W$

- **Expected Output:** Bitwise AND of two unsigned W-bit integers:

$$x \wedge y \in \{0,1\}^W.$$

ACL2: `(logand x y)`. Rust: `x & y`.
- **Chunking:**

$$\mathsf{Chunk}_{\mathsf{AND}}(x,y) = \mathsf{ChunkInterleave}_{m,C}(x,y).$$

- **Subtables:**

$$\mathsf{Subtables}_{\mathsf{AND}} = \left([(\mathsf{And}, i)]_{i=0}^{C-1}\right).$$

- **Lookup Combination:** Let $(Z_0, \ldots, Z_{C-1})$ be the lookup results. Then

$$\mathsf{Combine}_{\mathsf{AND}}(Z_0, \ldots, Z_{C-1}) = \mathsf{Concatenate}_{m,C}(Z_0, \ldots, Z_{C-1}).$$

2. **OR**:
   - **Operands:** $x, y \in \{0,1\}^W$
   - **Expected Output:** Bitwise OR of two unsigned W-bit integers:

$$x \vee y \in \{0,1\}^W.$$

   ACL2: `(logior x y)`. Rust: `x | y`.
   - **Chunking:**

$$\mathsf{Chunk}_{\mathsf{OR}}(x,y) = \mathsf{ChunkInterleave}_{m,C}(x,y).$$

   - **Subtables:**

$$\mathsf{Subtables}_{\mathsf{OR}} = \left([(\mathsf{Or}, i)]_{i=0}^{C-1}\right).$$

   - **Lookup Combination:** Let $(Z_0, \ldots, Z_{C-1})$ be the lookup results. Then

$$\mathsf{Combine}_{\mathsf{OR}}(Z_0, \ldots, Z_{C-1}) = \mathsf{Concatenate}_{m,C}(Z_0, \ldots, Z_{C-1}).$$

3. **XOR**:
   - **Operands:** $x, y \in \{0,1\}^W$
   - **Expected Output:** Bitwise XOR of two unsigned W-bit integers:

$$x \oplus y \in \{0,1\}^W.$$

   ACL2: `(logxor x y)`. Rust: `x ^ y`.
   - **Chunking:**

$$\mathsf{Chunk}_{\mathsf{XOR}}(x,y) = \mathsf{ChunkInterleave}_{m,C}(x,y).$$

   - **Subtables:**

$$\mathsf{Subtables}_{\mathsf{XOR}} = \left([(\mathsf{Xor}, i)]_{i=0}^{C-1}\right).$$

- **Lookup Combination:** Let $(Z_0, \ldots, Z_{C-1})$ be the lookup results. Then

$$\mathsf{Combine}_{\mathsf{XOR}}(Z_0, \ldots, Z_{C-1}) = \mathsf{Concatenate}_{m,C}(Z_0, \ldots, Z_{C-1}).$$

4. **ADD**:
   - **Operands:** $x, y \in \{0,1\}^W$
   - **Expected Output:** Addition of two unsigned W-bit integers, truncated to W bits:

$$\mathsf{Truncate}_W(x+y) \in \{0,1\}^W.$$

ACL2: `(loghead W (+ x y))`. Rust: `x.overflowing_add(y).0`.

   - **Chunking:** Let $z = x + y \in \{0,1\}^{W+1}$ be the addition result without truncation. Then

$$\mathsf{Chunk}_{\mathsf{ADD}}(x, y) = \mathsf{Chunk}_{2m,C}(z).$$

   - **Subtables:** Let $k = C - W/(2m)$, which is 2 for $W = 32$ and 4 for $W = 64$. Return

$$\mathsf{Subtables}_{\mathsf{ADD}} = \left([(\mathsf{TruncateOverflow}_W, i)]_{i=0}^{k-1}, [(\mathsf{Identity}, i)]_{i=k}^{C-1} \right).$$

   - **Lookup Combination:** Let $(Z_0, \ldots, Z_{C-1})$ be the lookup results. Then

$$\mathsf{Combine}_{\mathsf{ADD}}(Z_0, \ldots, Z_{C-1}) = \mathsf{Concatenate}_{m,C}(Z_0, \ldots, Z_{C-1}).$$

5. **SUB**:
   - **Operands:** $x, y \in \{0,1\}^W$
   - **Expected Output:** Subtraction of two unsigned W-bit integers, with the result taken modulo 2^W:

$$\mathsf{Truncate}_W(x-y) \in \{0,1\}^W.$$

ACL2: `(loghead W (- x y))`. Rust: `x.overflowing_sub(y).0`.

   - **Chunking:** Let $z = x + (2^W - y) \in \{0,1\}^{W+1}$ be the two's complement subtraction result. Then

$$\mathsf{Chunk}_{\mathsf{SUB}}(x, y) = \mathsf{Chunk}_{2m,C}(z).$$

   - **Subtables:** Let $k = C - W/(2m)$, which is 2 for $W = 32$ and 4 for $W = 64$. Return

$$\mathsf{Subtables}_{\mathsf{SUB}} = \left([(\mathsf{TruncateOverflow}_W, i)]_{i=0}^{k-1}, [(\mathsf{Identity}, i)]_{i=k}^{C-1} \right).$$

   - **Lookup Combination:** Let $(Z_0, \ldots, Z_{C-1})$ be the lookup results. Then

$$\mathsf{Combine}_{\mathsf{SUB}}(Z_0, \ldots, Z_{C-1}) = \mathsf{Concatenate}_{m,C}(Z_0, \ldots, Z_{C-1}).$$

6. **BEQ**:
   - **Operands:** $x, y \in \{0,1\}^W$
   - **Expected Output:** Whether two operands are equal: $(x \stackrel{?}{=} y) \in \{0,1\}$.[13]
     ACL2: `(if (= x y) 1 0)`. Rust: `x == y`.
   - **Chunking:**
   $$\mathsf{Chunk}_{\mathsf{BEQ}}(x,y) = \mathsf{ChunkInterleave}_{m,C}(x,y).$$
   - **Subtables:**
   $$\mathsf{Subtables}_{\mathsf{BEQ}} = \left([(\mathsf{Eq}, i)]_{i=0}^{C-1}\right).$$
   - **Lookup Combination:** Let $(Z_0, \dots, Z_{C-1})$ be the lookup results. Then
   $$\mathsf{Combine}_{\mathsf{BEQ}}(Z_0, \dots, Z_{C-1}) = \prod_{i=0}^{C-1} Z_i.$$
7. **BNE**:
   - **Operands:** $x, y \in \{0,1\}^W$
   - **Expected Output:** Whether two operands are not equal:
   $$(x \stackrel{?}{\neq} y) \in \{0,1\}.$$
     ACL2: `(if (= x y) 0 1)`. Rust: `x != y`.
   - **Chunking:**
   $$\mathsf{Chunk}_{\mathsf{BNE}}(x,y) = \mathsf{ChunkInterleave}_{m,C}(x,y).$$
   - **Subtables:**
   $$\mathsf{Subtables}_{\mathsf{BNE}} = \left([(\mathsf{Eq}, i)]_{i=0}^{C-1}\right).$$
   - **Lookup Combination:** Let $(Z_0, \dots, Z_{C-1})$ be the lookup results. Then
   $$\mathsf{Combine}_{\mathsf{BNE}}(Z_0, \dots, Z_{C-1}) = 1 - \prod_{i=0}^{C-1} Z_i.$$
8. **SLTU**:
   - **Operands:** $x, y \in \{0,1\}^W$
   - **Expected Output:** Whether the first operand is less than the second operand (both interpreted as unsigned W-bit numbers):[14]
   $$(x \stackrel{?}{<} y) \in \{0,1\}.$$
     ACL2: `(if (< x y) 1 0)`. Rust: `x < y`.

---

[13] The "branch" part of the instruction is handled via non-lookup Jolt constraints (such as R1CS).

[14] The "set" part of the instruction is handled via non-lookup Jolt constraints (such as R1CS)

- **Chunking:**

$$\mathsf{Chunk}_{\mathsf{SLTU}}(x, y) = \mathsf{ChunkInterleave}_{m,C}(x, y).$$

- **Subtables:**

$$\mathsf{Subtables}_{\mathsf{SLTU}} = \left([(\mathsf{Ltu}, i)]_{i=0}^{C-1}, [(\mathsf{Eq}, i)]_{i=0}^{C-2} \right).$$

- **Lookup Combination:** Let $(Z_0, \ldots, Z_{C-1}, W_0, \ldots, W_{C-2})$ be the lookup results. Then

$$\mathsf{Combine}_{\mathsf{SLTU}}(Z_0, \ldots, Z_{C-1}, W_0, \ldots, W_{C-2}) = \sum_{i=0}^{C-1} Z_i \cdot \prod_{j=0}^{i-1} W_j.$$

ACL2, for 32-bit: `(+ z0 (* z1 w0) (* z2 w0 w1) (* z3 w0 w1 w2))`.
ACL2, for 64-bit:

```
( + z0 (* z1 w0) (* z2 w0 w1) (* z3 w0 w1 w2) (* z4 w0 w1 w2 w3)
    (* z5 w0 w1 w2 w3 w4) (* z6 w0 w1 w2 w3 w4 w5)
    (* z7 w0 w1 w2 w3 w4 w5 w6))
```

Rust:

```rust
fn combine_lookups<F: JoltField>(&self, vals: &[F], C: usize, M: usize) -> F {
    let vals_by_subtable = self.slice_values(vals, C, M);
    let ltu = vals_by_subtable[0];
    let eq = vals_by_subtable[1];

    let mut sum = F::zero();
    let mut eq_prod = F::one();

    for i in 0..C - 1 {
        sum += ltu[i] * eq_prod;
        eq_prod *= eq[i];
    }
        // Do not need to update ‘eq_prod‘ for the last iteration
        sum + ltu[C - 1] * eq_prod
}
```

9. **BGEU**:
    - **Operands:** $x, y \in \{0, 1\}^W$
    - **Expected Output:** Whether the first operand is greater than or equal to the second operand (both interpreted as unsigned W-bit numbers):

$$(x \overset{?}{\geq} y) \in \{0, 1\}.$$

ACL2: `(if (< x y) 0 1)`. Rust: `x >= y`.

- **Chunking:**

$$\mathsf{Chunk}_{\mathsf{BGEU}}(x, y) = \mathsf{ChunkInterleave}_{m,C}(x, y).$$

- **Subtables:**

$$\mathsf{Subtables}_{\mathsf{BGEU}} = \left([(\mathsf{Ltu}, i)]_{i=0}^{C-1}, [(\mathsf{Eq}, i)]_{i=0}^{C-2} \right).$$

- **Lookup Combination:** Let $(Z_0, \ldots, Z_{C-1}, W_0, \ldots, W_{C-2})$ be the lookup results. Then

$$\begin{aligned}&\mathsf{Combine}_{\mathsf{BGEU}}(Z_0, \ldots, Z_{C-1}, W_0, \ldots, W_{C-2}) \\ &= 1 - \mathsf{Combine}_{\mathsf{SLTU}}(Z_0, \ldots, Z_{C-1}, W_0, \ldots, W_{C-2}).\end{aligned}$$

10. SLT:

- **Operands:** $x, y \in \{0,1\}^W$
- **Expected Output:** Whether the first operand is less than the second operand (both interpreted as **signed** W-bit numbers):

$$(x \overset{?}{<} y) \in \{0,1\}.$$

ACL2: `(if (< (logext 32 x) (logext 32 y)) 1 0)`.
Rust: `(x as i32) < (y as i32)`.
The 64-bit version simply replaces `32` with `64`.

- **Chunking:**

$$\mathsf{Chunk}_{\mathsf{SLT}}(x, y) = \mathsf{ChunkInterleave}_{m,C}(x, y).$$

- **Subtables:**

$$\mathsf{Subtables}_{\mathsf{SLT}} = \begin{pmatrix} (\mathsf{LeftMSB}, 0), (\mathsf{RightMSB}, 0), \\ [(\mathsf{Ltu}, i)]_{i=1}^{C-1}, [(\mathsf{Eq}, i)]_{i=1}^{C-2}, \\ (\mathsf{LtAbs}, 0), (\mathsf{EqAbs}, 0) \end{pmatrix}.$$

- **Lookup Combination:** Let $(L, R, Z_1, \ldots, Z_{C-1}, W_1, \ldots, W_{C-2}, Z_0, W_0)$ be the lookup results. Then

$$\begin{aligned}&\mathsf{Combine}_{\mathsf{SLT}}(L, R, Z_1, \ldots, Z_{C-1}, W_1, \ldots, W_{C-2}, Z_0, W_0) \\ &= L \cdot (1 - R) + ((1 - L) \cdot (1 - R) + L \cdot R) \cdot \left(\sum_{i=0}^{C-1} Z_i \cdot \prod_{j=0}^{i-1} W_j \right).\end{aligned}$$

ACL2, for 32-bit:

```
(+ (* L (- 1 R))
   (* (+ (* (- 1 L) (- 1 R)) (* L R))
      (+ z0 (* z1 w0) (* z2 w0 w1) (* z3 w0 w1 w2))))
```

The 64-bit version is modified similarly to SLTU.
Rust:

```rust
fn combine_lookups<F: JoltField>(&self, vals: &[F], C: usize, M: usize) -> F {
    let vals_by_subtable = self.slice_values(vals, C, M);

    let left_msb = vals_by_subtable[0];
    let right_msb = vals_by_subtable[1];
    let ltu = vals_by_subtable[2];
```

```
    let eq = vals_by_subtable[3];
    let lt_abs = vals_by_subtable[4];
    let eq_abs = vals_by_subtable[5];

    // Accumulator for LTU(x_{<s}, y_{<s})
    let mut ltu_sum = lt_abs[0];
    // Accumulator for EQ(x_{<s}, y_{<s})
    let mut eq_prod = eq_abs[0];

    for i in 0..C - 2 {
        ltu_sum += ltu[i] * eq_prod;
        eq_prod *= eq[i];
    }
    // Do not need to update `eq_prod` for the last iteration
    ltu_sum += ltu[C - 2] * eq_prod;

    // x_s * (1 - y_s) + EQ(x_s, y_s) * LTU(x_{<s}, y_{<s})
    left_msb[0] * (F::one() - right_msb[0]) + (left_msb[0] * right_msb[0]
        + (F::one() - left_msb[0]) * (F::one() - right_msb[0])) * ltu_sum
}
```

11. BGE:

- **Operands:** $x, y \in \{0,1\}^W$
- **Expected Output:** Whether the first operand is greater than or equal to the second operand (both interpreted as **signed** W-bit numbers):

$$(x \overset{?}{\geq} y) \in \{0,1\}.$$

ACL2: `(if (>= (logext 32 x) (logext 32 y)) 1 0)`.
Rust: `(x as i32) >= (y as i32)`.
The 64-bit version simply replaces `32` with `64`.

- **Chunking:**

$$\mathsf{Chunk}_{\mathsf{BGE}}(x, y) = \mathsf{ChunkInterleave}_{m,C}(x, y).$$

- **Subtables:**

$$\mathsf{Subtables}_{\mathsf{BGE}} = \begin{pmatrix} (\mathsf{LeftMSB}, 0), (\mathsf{RightMSB}, 0), \\ [(\mathsf{Ltu}, i)]_{i=1}^{C-1}, [(\mathsf{Eq}, i)]_{i=1}^{C-2}, \\ (\mathsf{LtAbs}, 0), (\mathsf{EqAbs}, 0) \end{pmatrix}.$$

- **Lookup Combination:** Let $(L, R, Z_1, \ldots, Z_{C-1}, W_1, \ldots, W_{C-2}, Z_0, W_0)$ be the lookup results. Then

$$\begin{aligned} &\mathsf{Combine}_{\mathsf{BGE}}(L, R, Z_1, \ldots, Z_{C-1}, W_1, \ldots, W_{C-2}, Z_0, W_0) \\ &= 1 - \mathsf{Combine}_{\mathsf{SLT}}(L, R, Z_1, \ldots, Z_{C-1}, W_1, \ldots, W_{C-2}, Z_0, W_0). \end{aligned}$$

12. LB:

- **Operand:** $x \in \{0,1\}^W$
- **Expected Output:** the lower 8 bits of x sign-extended to W bits:

$$\mathsf{SignExtend}_W(x \bmod 2^8) \in \{0,1\}^W.$$

ACL2: `(logextu 32 8 (logand x #xff))`.

Rust: `(x & 0xff) as i8 as i32`.

The 64-bit version simply replaces `32` with `64`.

- **Chunking:**

$$\mathsf{Chunk}_{\mathsf{LB}}(x) = \mathsf{Chunk}_{m,C}(x).$$

- **Subtables:** Assume $M \geq 2^8$. Then

$$\mathsf{Subtables}_{\mathsf{LB}} = \begin{pmatrix} (\mathsf{TruncateOverflow}_8, C-1), (\mathsf{SignExtend}_8, C-1), \\ [(\mathsf{Identity}, i)]_{i=0}^{C-1} \end{pmatrix}.$$

- **Lookup Combination:** Let $(Z, S, I_0, \ldots, I_{C-1})$ be the lookup results. Then

$$\mathsf{Combine}_{\mathsf{LB}}(Z, S, I_0, \ldots, I_{C-1}) = Z + \sum_{i=1}^{C-1} 2^{8 \cdot i} \cdot S.$$

Note that for all load and store instructions (such as **LB**, **LH**, **SB**, **SH**, and **SW**), the identity subtables are *not* used for computing final result, but are used to range-check the chunks (which is necessary for other parts of the Jolt constraint system).

13. LH:

- **Operand:** $x \in \{0,1\}^W$
- **Expected Output:** the lower 16 bits of x sign-extended to W bits:

$$\mathsf{SignExtend}_W(x \bmod 2^{16}) \in \{0,1\}^W.$$

ACL2: `(logextu 32 16 (logand x #xffff))`.

Rust: `(x & 0xffff) as i16 as i32`.

The 64-bit version simply replaces `32` with `64`.

- **Chunking:**

$$\mathsf{Chunk}_{\mathsf{LH}}(x) = \mathsf{Chunk}_{m,C}(x).$$

- **Subtables:** Assume $M \geq 2^{16}$. Then

$$\mathsf{Subtables}_{\mathsf{LH}} = \left((\mathsf{Identity}, C-1), (\mathsf{SignExtend}, C-1), [(\mathsf{Identity}, i)]_{i=0}^{C-1} \right).$$

- **Lookup Combination:** Let $(Z, S, I_0, \ldots, I_{C-1})$ be the lookup results. Then

$$\mathsf{Combine}_{\mathsf{LH}}(Z, S, I_0, \ldots, I_{C-1}) = Z + \sum_{i=0}^{C-1} 2^{16 \cdot i} \cdot S.$$

14. SB:

- **Operand:** $x \in \{0,1\}^W$

- **Expected Output:** The lower 8 bits of x zero-extended to W bits:

$$\mathsf{ZeroExtend}_W(x \bmod 2^8) \in \{0,1\}^W.$$

ACL2: `(logand x #xff)`. Rust: `(x & 0xff)`.

- **Chunking:**

$$\mathsf{Chunk}_{\mathsf{SB}}(x) = \mathsf{Chunk}_{m,C}(x).$$

- **Subtables:** Assume $M \geq 2^8$. Then

$$\mathsf{Subtables}_{\mathsf{SB}} = \left((\mathsf{TruncateOverflow}_8, C-1), [(\mathsf{Identity}, i)]_{i=0}^{C-1}\right).$$

- **Lookup Combination:** Let $(Z, I_0, \ldots, I_{C-1})$ be the lookup results. Then

$$\mathsf{Combine}_{\mathsf{SB}}(Z, I_0, \ldots, I_{C-1}) = Z.$$

15. SH:

- **Operand:** $x \in \{0,1\}^W$
- **Expected Output:** The lower 16 bits of x zero-extended to W bits:

$$\mathsf{ZeroExtend}_W(x \bmod 2^{16}) \in \{0,1\}^W.$$

ACL2: `(logand x #xffff)`. Rust: `(x & 0xffff)`.

- **Chunking:**

$$\mathsf{Chunk}_{\mathsf{SH}}(x) = \mathsf{Chunk}_{m,C}(x).$$

- **Subtables:** Assume $M \geq 2^{16}$. Then

$$\mathsf{Subtables}_{\mathsf{SH}} = \left((\mathsf{Identity}, C-1), [(\mathsf{Identity}, i)]_{i=0}^{C-1}\right).$$

- **Lookup Combination:** Let $(I_0, \ldots, I_{C-1})$ be the lookup results. Then

$$\mathsf{Combine}_{\mathsf{SH}}(I_0, \ldots, I_{C-1}) = I_0.$$

16. SW:

- **Operand:** $x \in \{0,1\}^W$
- **Expected Output:** The lower 32 bits of x zero-extended to W bits:

$$\mathsf{ZeroExtend}_W(x \bmod 2^{32}) \in \{0,1\}^W.$$

ACL2: `(logand x #xffffffff)`. Rust: `(x & 0xffffffff)`.

- **Chunking:**

$$\mathsf{Chunk}_{\mathsf{SW}}(x) = \mathsf{Chunk}_{m,C}(x).$$

- **Subtables:** Assume $M = 2^{16}$. Then

$$\mathsf{Subtables}_{\mathsf{SW}} = \left((\mathsf{Identity}, C-2), (\mathsf{Identity}, C-1)\right).$$

- **Lookup Combination:** Let (I_{C-2}, I_{C-1}) be the lookup results. Then

$$\mathsf{Combine}_{\mathsf{SW}}(I_{C-2}, I_{C-1}) = \mathsf{Concatenate}_{m,2}(I_{C-2}, I_{C-1}).$$

17. SLL:

- **Operands:** $x, y \in \{0,1\}^W$
- **Expected Output:** The left shift of an unsigned W-bit integer x by $(y \bmod W)$ bits, truncated to W bits:

$$\mathsf{Truncate}_W(x \ll (y \bmod W)) \in \{0,1\}^W.$$

  ACL2: `(loghead 32 (ash x (mod y 32)))`.

  Rust: `x.checked_shl(y % W as u32).unwrap_or()`.

  The 64-bit version simply replaces `32` with `64`.
- **Chunking:**

$$\mathsf{Chunk}_{\mathsf{SLL}}(x, y) = \mathsf{ChunkForShift}_{m,C}(x, y).$$

- **Subtables:**

$$\mathsf{Subtables}_{\mathsf{SLL}} = \left([(\mathsf{Sll}_{C-i,W}, i)]_{i=0}^{C-1}\right).$$

- **Lookup Combination:** Let $(Z_0, \ldots, Z_{C-1})$ be the lookup results. Then

$$\mathsf{Combine}_{\mathsf{SLL}}(Z_0, \ldots, Z_{C-1}) = \mathsf{Concatenate}_{2m,C}(Z_0, \ldots, Z_{C-1}).$$

18. SRL:

- **Operands:** $x, y \in \{0,1\}^W$
- **Expected Output:** The right shift of an unsigned W-bit integer x by $(y \bmod W)$ bits, zero-extended to W bits:

$$\mathsf{Truncate}_W(x \gg (y \bmod W)) \in \{0,1\}^W.$$

  ACL2: `(ash x (- (mod y W)))`.

  Rust: `x.wrapping_shr(y % W as u32)`.
- **Chunking:**

$$\mathsf{Chunk}_{\mathsf{SRL}}(x, y) = \mathsf{ChunkForShift}_{m,C}(x, y).$$

- **Subtables:**

$$\mathsf{Subtables}_{\mathsf{SRL}} = \left([(\mathsf{Srl}_{C-i,W}, i)]_{i=0}^{C-1}\right).$$

- **Lookup Combination:** Let $(Z_0, \ldots, Z_{C-1})$ be the lookup results. Then

$$\mathsf{Combine}_{\mathsf{SRL}}(Z_0, \ldots, Z_{C-1}) = Z_0 + \cdots + Z_{C-1}.$$

19. **SRA**:
   - **Operands:** $x, y \in \{0,1\}^W$
   - **Expected Output:** The right shift of a signed W-bit integer x by $(y \bmod W)$ bits, sign-extended to W bits:

     $$\mathsf{SignExtend}_W(\mathsf{SignExtend}_W(x) \gg (y \bmod W)) \in \{0,1\}^W.$$

     ACL2: `(ashu 32 x (- (mod y 32)))`.

     Rust: `(x as i32).wrapping_shr(y % 32 as u32) as u32`.

     The 64-bit version simply replaces `32` with `64` (except for `as u32` right after `%`).
   - **Chunking:**

     $$\mathsf{Chunk}_{\mathsf{SRA}}(x, y) = \mathsf{ChunkForShift}_{m,C}(x, y).$$

   - **Subtables:**

     $$\mathsf{Subtables}_{\mathsf{SRA}} = \left([(\mathsf{Srl}_{C-i,W}, i)]_{i=0}^{C-1}, (\mathsf{SraSign}_W, 0)\right).$$

   - **Lookup Combination:** Let $(Z_0, \ldots, Z_{C-1}, S)$ be the lookup results. Then

     $$\mathsf{Combine}_{\mathsf{SRA}}(Z_0, \ldots, Z_{C-1}, S) = Z_0 + \cdots + Z_{C-1} + S.$$

B.3 List of Instructions in the M-extension

1. **MUL**:
   - **Operands:** $x, y \in \{0,1\}^W$
   - **Expected Output:** Multiplication of two **signed** W-bit integers, truncated to W bits:

     $$\mathsf{Truncate}_W(x \cdot y) \in \{0,1\}^W.$$

     ACL2: `(loghead W (* (logext W x) (logext W y)))`.
     Rust: `(x as i32).wrapping_mul(y as i32) as u32 as u64` for 32-bit words, and
     `(x as i64).wrapping_mul(y as i64) as u64` for 64-bit words.
   - **Chunking:** Let $z = x \cdot y \in \{0,1\}^{2W}$ be the multiplication result without truncation. Then

     $$\mathsf{Chunk}_{\mathsf{MUL}}(x, y) = \mathsf{Chunk}_{2m,C}(z).$$

   - **Subtables:** Let $k = C - W/(2m)$. Return

     $$\mathsf{Subtables}_{\mathsf{MUL}} = \left([(\mathsf{TruncateOverflow}_W, i)]_{i=0}^{k-1}, [(\mathsf{Identity}, i)]_{i=k}^{C-1}\right).$$

   - **Lookup Combination:** Let $(Z_0, \ldots, Z_{C-1})$ be the lookup results. Then

     $$\mathsf{Combine}_{\mathsf{MUL}}(Z_0, \ldots, Z_{C-1}) = \mathsf{Concatenate}_{2m,C}(Z_0, \ldots, Z_{C-1}).$$

2. **MULU**:
   - **Operands:** $x, y \in \{0,1\}^W$
   - **Expected Output:** $\mathsf{Truncate}_W(x \cdot y) \in \{0,1\}^W$, as multiplication of two **unsigned** W-bit integers, truncated to W bits.
     ACL2: `(loghead W (* x y))`.
     Rust: `x.wrapping_mul(y) as u32 as u64` for 32-bit words, and `x.wrapping_mul(y)` for 64-bit words.
   - **Chunking:** Let $z = x \cdot y \in \{0,1\}^{2W}$ be the multiplication result without truncation. Then

     $$\mathsf{Chunk}_{\mathsf{MULU}}(x, y) = \mathsf{Chunk}_{2m,C}(z).$$

   - **Subtables:** Let $k = C - W/(2m)$. Return

     $$\mathsf{Subtables}_{\mathsf{MULU}} = \left([(\mathsf{TruncateOverflow}_W, i)]_{i=0}^{k-1}, [(\mathsf{Identity}, i)]_{i=k}^{C-1}\right).$$

   - **Lookup Combination:** Let $(Z_0, \ldots, Z_{C-1})$ be the lookup results. Then

     $$\mathsf{Combine}_{\mathsf{MULU}}(Z_0, \ldots, Z_{C-1}) = \mathsf{Concatenate}_{2m,C}(Z_0, \ldots, Z_{C-1}).$$

3. **MULHU**:
   - **Operands:** $x, y \in \{0,1\}^W$
   - **Expected Output:** $\mathsf{Truncate}_W((x \cdot y) \gg W) \in \{0,1\}^W$, as high W-bit half of the multiplication of two **unsigned** W-bit integers.
     ACL2: `(logtail W (* x y))`.
     Rust: `x.wrapping_mul(y) >> 32` for 32-bit words, and `((x as u128).wrapping_mul(y as u128) >> 64) as u64` for 64-bit words.
   - **Chunking:** Let $z = x \cdot y \in \{0,1\}^{2W}$ be the multiplication result. Then

     $$\mathsf{Chunk}_{\mathsf{MULHU}}(x, y) = \mathsf{Chunk}_{2m,C}(z).$$

   - **Subtables:**

     $$\mathsf{Subtables}_{\mathsf{MULHU}} = \left([(\mathsf{Identity}, i)]_{i=0}^{C/2-1}\right).$$

   - **Lookup Combination:** Let $(Z_0, \ldots, Z_{C/2-1})$ be the lookup results. Then

     $$\mathsf{Combine}_{\mathsf{MULHU}}(Z_0, \ldots, Z_{C/2-1}) = \mathsf{Concatenate}_{2m,C/2}(Z_0, \ldots, Z_{C/2-1}).$$

4. **ADVICE**:
   - **Operands:** $x \in \{0,1\}^W$
   - **Expected Output:** x.[15]

[15] The ADVICE instruction is used to range-check an advice value x which may be an arbitrary field element. Since range-checking is only important for other parts of Jolt's constraint system (such as memory-checking), we will idealize this aspect in our work, and assume that x is already a W-bit unsigned integer.

- **Chunking:** $\mathsf{Chunk}_{\mathsf{ADVICE}}(x) = \mathsf{Chunk}_{2m,C}(x)$.
- **Subtables:** Let $k = C - W/(2m)$. Return

$$\mathsf{Subtables}_{\mathsf{ADVICE}} = \left([(\mathsf{TruncateOverflow}_W, i)]_{i=0}^{k-1}, [(\mathsf{Identity}, i)]_{i=k}^{C-1}\right).$$

- **Lookup Combination:** Let $(Z_0, \ldots, Z_{C-1})$ be the lookup results. Then

$$\mathsf{Combine}_{\mathsf{ADVICE}}(Z_0, \ldots, Z_{C-1}) = \mathsf{Concatenate}_{2m,C}(Z_0, \ldots, Z_{C-1}).$$

5. **ASSERTLTE**:
   - **Operands:** $x, y \in \{0,1\}^W$
   - **Expected Output:** Whether the first operand is less than or equal to the second operand (both interpreted as **unsigned** W-bit numbers):

$$(x \overset{?}{\leq} y) \in \{0, 1\}.$$

   ACL2: `(if (<= x y) 1 0)`.
   Rust: `x <= y`.
   - **Chunking:**

$$\mathsf{Chunk}_{\mathsf{ASSERTLTE}}(x, y) = \mathsf{ChunkInterleave}_{m,C}(x, y).$$

   - **Subtables:**

$$\mathsf{Subtables}_{\mathsf{ASSERTLTE}} = \left([(\mathsf{Ltu}, i)]_{i=0}^{C-1}, [(\mathsf{Eq}, i)]_{i=0}^{C-1}\right).$$

   - **Lookup Combination:** Let $(Z_0, \ldots, Z_{C-1}, W_0, \ldots, W_{C-1})$ be the lookup results. Then

$$\begin{aligned} &\mathsf{Combine}_{\mathsf{ASSERTLTE}}(Z_0, \ldots, Z_{C-1}, W_0, \ldots, W_{C-1}) \\ &= \mathsf{Combine}_{\mathsf{SLTU}}(Z_0, \ldots, Z_{C-1}, W_0, \ldots, W_{C-2}) \\ &\quad + \mathsf{Combine}_{\mathsf{BEQ}}(W_0, \ldots, W_{C-1}) \\ &= \sum_{i=0}^{C-1} Z_i \cdot \prod_{j=0}^{i-1} W_j + \prod_{i=0}^{C-1} W_i. \end{aligned}$$

6. **AssertValidDiv0**:
   - **Operands:** $x, y \in \{0,1\}^W$
   - **Expected Output:** Return 1 if either the first operand (considered as the divisor) is non-zero, or the first operand is zero and the second operand (considered as the quotient) is $11\ldots1$ (interpreted as an unsigned W-bit number); otherwise return 0:

$$(\neg(x \overset{?}{=} 0) \vee (y \overset{?}{=} 2^W - 1)) \in \{0, 1\}.$$

   ACL2: `(if (or (not (= x 0)) (= y (1- (expt 2 W)))) 1 0)`.
   In Rust, the function is implemented as follows:

```rust
fn lookup_entry(&self) -> u64 {
    let divisor = self.0; // ‘x‘ operand
    let quotient = self.1; // ‘y‘ operand
    if divisor == 0 {
        match WORD_SIZE {
            32 => (quotient == u32::MAX as u64).into(),
            64 => (quotient == u64::MAX).into(),
            _ => panic!("Unsupported WORD_SIZE: {}", WORD_SIZE),
        }
    } else {
        1
    }
}
```

- **Chunking:**

$$\mathsf{Chunk}_{\mathsf{AssertValidDiv0}}(x, y) = \mathsf{ChunkInterleave}_{m,C}(x, y).$$

- **Subtables:**

$$\mathsf{Subtables}_{\mathsf{AssertValidDiv0}} = \left([(\mathsf{LeftIsZero}, i)]_{i=0}^{C-1}, [(\mathsf{DivByZero}, i)]_{i=0}^{C-1}\right).$$

- **Lookup Combination:** Let $(Z_0, \ldots, Z_{C-1}, W_0, \ldots, W_{C-1})$ be the lookup results. Then

$$\mathsf{Combine}_{\mathsf{AssertValidDiv0}}(Z_0, \ldots, Z_{C-1}, W_0, \ldots, W_{C-1}) = 1 - \prod_{i=0}^{C-1} Z_i + \prod_{i=0}^{C-1} W_i.$$

7. **AssertValidSignedRemainder**:

- **Operands:** $x, y \in \{0,1\}^W$
- **Expected Output:** Interpret the operands as **signed** W-bit numbers, with the first operand being the remainder and the second being the divisor. Return 1 if either the remainder or the divisor is zero, or the sign of the remainder is equal to the sign of the divisor and also that the absolute value of the remainder is less than the absolute value of the divisor; otherwise return 0:

$$(x \stackrel{?}{=} 0) \vee (y \stackrel{?}{=} 0) \vee ((x_0 \stackrel{?}{=} y_0) \wedge (|x_{[1:W-1]}| \stackrel{?}{<} |y_{[1:W-1]}|)) \in \{0,1\}.$$

In ACL2, the function is implemented as follows:

```lisp
(if (or (= x 0) (= y 0)
        (and (= (logbit (1- W) x) (logbit (1- W) y))
             (< (abs (logext W x)) (abs (logext W y)))))
    1 0)
```

In Rust, the function is implemented as follows:

```
fn lookup_entry(&self) -> u64 {
    match WORD_SIZE {
        32 => {
            let remainder = self.0 as u32 as i32;
            let divisor = self.1 as u32 as i32;
            let is_remainder_zero = remainder == 0;
            let is_divisor_zero = divisor == 0;

            if is_remainder_zero || is_divisor_zero {
                1
            } else {
                let remainder_sign = remainder >> 31;
                let divisor_sign = divisor >> 31;
                (remainder.abs() < divisor.abs() &&
    remainder_sign == divisor_sign).into()
            }
        }
        64 => {
            let remainder = self.0 as i64;
            let divisor = self.1 as i64;
            let is_remainder_zero = remainder == 0;
            let is_divisor_zero = divisor == 0;

            if is_remainder_zero || is_divisor_zero {
                1
            } else {
                let remainder_sign = remainder >> 63;
                let divisor_sign = divisor >> 63;
                (remainder.abs() < divisor.abs() &&
    remainder_sign == divisor_sign).into()
            }
        }
        _ => panic!("Unsupported WORD_SIZE: {}", WORD_SIZE),
    }
}
```

- **Chunking:**

$$\mathsf{Chunk}_{\mathsf{AssertValidSignedRemainder}}(x, y) = \mathsf{ChunkInterleave}_{m,C}(x, y).$$

- **Subtables:**

$$\mathsf{Subtables}_{\mathsf{AssertValidSignedRemainder}} = \begin{pmatrix} (\mathsf{LeftMSB}, 0), (\mathsf{RightMSB}, 0), \\ [(\mathsf{Eq}, i)]_{i=1}^{C-1}, [(\mathsf{Ltu}, i)]_{i=1}^{C-1}, \\ (\mathsf{EqAbs}, 0), (\mathsf{LtAbs}, 0), \\ [(\mathsf{LeftIsZero}, i)]_{i=0}^{C-1}, \\ [(\mathsf{RightIsZero}, i)]_{i=0}^{C-1} \end{pmatrix}.$$

- **Lookup Combination:** Let the lookup results be

$$(L, R, Z_1, \ldots, Z_{C-1}, W_1, \ldots, W_{C-2}, Z_0, W_0, L'_0, \ldots, L'_{C-1}, R'_0, \ldots, R'_{C-1}).$$

Then

$$\begin{aligned}
&\mathsf{Combine}_{\mathsf{AssertValidSignedRemainder}}(Z_0, \ldots, Z_{C-1}, W_0, \ldots, W_{C-1}) \\
&= (1 - L - R) \cdot \mathsf{Combine}_{\mathsf{SLTU}}(Z_0, \ldots, Z_{C-1}, W_0, \ldots, W_{C-2}) \\
&\quad + L \cdot R \cdot (1 - \mathsf{Combine}_{\mathsf{BEQ}}(W_0, \ldots, W_{C-1})) \\
&\quad + (1 - L) \cdot R \cdot \prod_{i=0}^{C-1} L'_i + \prod_{i=0}^{C-1} R'_i
\end{aligned}$$

8. **AssertValidUnsignedRemainder**:
   - **Operands:** $x, y \in \{0,1\}^W$
   - **Expected Output:** Interpret the operands as **unsigned** W-bit numbers, with the first operand being the remainder and the second being the divisor. Return 1 if either the divisor is zero, or the remainder is less than the divisor; otherwise return 0:

   $$(y \stackrel{?}{=} 0) \vee (x \stackrel{?}{<} y) \in \{0, 1\}.$$

   ACL2: `(if (or (= y 0) (< x y)) 1 0)`.
   Rust: `(y == 0 || x < y)`.
   - **Chunking:**

   $$\mathsf{Chunk}_{\mathsf{AssertValidUnsignedRemainder}}(x, y) = \mathsf{ChunkInterleave}_{m,C}(x, y).$$

   - **Subtables:**

   $$\mathsf{Subtables}_{\mathsf{AssertValidUnsignedRemainder}} = \begin{pmatrix} [(\mathsf{Ltu}, i)]_{i=0}^{C-1}, [(\mathsf{Eq}, i)]_{i=0}^{C-2}, \\ [(\mathsf{RightIsZero}, i)]_{i=0}^{C-1} \end{pmatrix}.$$

   - **Lookup Combination:** Let the lookup results be

   $$(Z_0, \ldots, Z_{C-1}, W_0, \ldots, W_{C-2}, R_0, \ldots, R_{C-1}).$$

   Then

   $$\begin{aligned}
   &\mathsf{Combine}_{\mathsf{AssertValidUnsignedRemainder}}(Z_0, \ldots, Z_{C-1}, W_0, \ldots, W_{C-2}, R_0, \ldots, R_{C-1}) \\
   &= \mathsf{Combine}_{\mathsf{SLTU}}(Z_0, \ldots, Z_{C-1}, W_0, \ldots, W_{C-2}) + \prod_{i=0}^{C-1} R_i.
   \end{aligned}$$

9. **MOVE**:
   - **Operands:** $x \in \{0,1\}^W$

- **Expected Output:** x.[16]
- **Chunking:** $\mathsf{Chunk}_{\mathsf{MOVE}}(x) = \mathsf{Chunk}_{2m,C}(x)$.
- **Subtables:** Return

$$\mathsf{Subtables}_{\mathsf{MOVE}} = \left([(\mathsf{Identity}, i)]_{i=0}^{C-1}\right).$$

- **Lookup Combination:** Let $(Z_0, \ldots, Z_{C-1})$ be the lookup results. Then

$$\mathsf{Combine}_{\mathsf{MOVE}}(Z_0, \ldots, Z_{C-1}) = \mathsf{Concatenate}_{2m,C}(Z_0, \ldots, Z_{C-1}).$$

10. **MOVSIGN**:
   - **Operands:** $x \in \{0,1\}^W$
   - **Expected Output:** Interpret x as a **signed** W-bit number, and return an unsigned W-bit number with the sign bit of x extended to all bits of the result:[17]

$$x_0 \cdot (2^W - 1), \qquad \text{where } x_0 \text{ is the sign bit of } x.$$

   ACL2: `(* (logbit 31 x) (1- (expt 2 32)))`.
   In Rust, the function is implemented as follows:

```rust
const ALL_ONES_32: u64 = 0xFFFF_FFFF;
const ALL_ONES_64: u64 = 0xFFFF_FFFF_FFFF_FFFF;
const SIGN_BIT_32: u64 = 0x8000_0000;
const SIGN_BIT_64: u64 = 0x8000_0000_0000_0000;

fn lookup_entry(&self) -> u64 {
    match WORD_SIZE {
        32 => {
            if self.0 & SIGN_BIT_32 != 0 {
                ALL_ONES_32
            } else {
                0
            }
        }
        64 => {
            if self.0 & SIGN_BIT_64 != 0 {
                ALL_ONES_64
            } else {
                0
            }
        }
        _ => panic!("only implemented for u32 / u64"),
    }
}
```

[16] The MOVE instruction is used to move an operand from one operand to another, but otherwise does not change the operand's value. Since the moving part is handled by other parts of Jolt's constraint system, and thus is out of scope for our work.

[17] Similar to the MOVE instruction, we do not model the moving aspect of the instruction.

- **Chunking:** $\mathsf{Chunk}_{\mathsf{MOVSIGN}}(x) = \mathsf{Chunk}_{2m,C}(x)$.
- **Subtables:** Let $k = C - W/(2m)$. Return

$$\mathsf{Subtables}_{\mathsf{MOVSIGN}} = \left([(\mathsf{SignExtend}_{16}, k)] \, , [(\mathsf{Identity}, i)]_{i=0}^{C-1} \right).$$

- **Lookup Combination:** Let $(S, Z_0, \ldots, Z_{W/(2m)-1})$ be the lookup results. Then

$$\begin{aligned} &\mathsf{Combine}_{\mathsf{MOVSIGN}}(S, Z_0, \ldots, Z_{W/(2m)-1}) \\ &= \mathsf{Concatenate}_{2m, W/(2m)}(S, S, \ldots, S). \end{aligned}$$