Three-Layer Formal Verification of Quantized ML Kernels: From Compiler IR to ISA Specification

Yi-De Wu
National Taiwan University

***Abstract*—Quantized neural network inference on RISC-V Vector (RVV) hardware requires correctness guarantees across the compiler and hardware stack. We present a three-layer formal verification pipeline that proves equivalence between a compiler-level specification (MLIR), an intermediate representation (LLVM IR), and the ISA-level semantics derived from the Sail formal model via Isla symbolic execution. A key challenge is that symbolic execution of SIMD vector instructions produces monolithic 256-bit formulas and 2^{VL}-path explosions for masked operations, making naïve verification intractable. We solve this with three scalability techniques: *per-element AST decomposition* splits vector traces into individual lane expressions, reducing solver time from >10 min to <1 s; *mask decomposition* factors masked instructions into an unmasked trace, a spec-level ITE policy, and an algebraic lemma, eliminating exponential path enumeration; and *linear composition* via Z3 `substitute()` scales kernel verification with instruction count, not input width. The pipeline is ISA-agnostic: its parser, decomposer, and composer operate on standard SMT-LIB traces, and the MLIR specification layer provides a cross-ISA golden reference, so porting to a new architecture requires only replacing the Sail model. We verify thirteen kernels covering all ten llama.cpp quantized dot products—Q4_0, Q4_1, Q5_0, Q5_1, Q8_0 (basic and asymmetric formats) and Q2_K, Q3_K, Q4_K, Q5_K, Q6_K (K-quant multi-level quantization)—plus element-wise vector addition, ReLU activation, and 1D convolution, all verified automatically in <30 s. The Q5_0 proof relies on one RVV-specification axiom (the mask-undisturbed policy); all other proofs are fully Isla-backed. We further demonstrate cross-ISA portability by verifying the Q8_0 kernel against WebAssembly SIMD128 semantics (axiomatized from the W3C spec), yielding the first formal proof that two ISAs compute the same quantized dot product for all 2^{256} inputs.**

I. Introduction

To illustrate the risk, we synthesize a realistic bug: in llama.cpp's Q4_0 quantized inference kernel, changing the shift amount from 4 to 3 in a single `vsrl.vx` instruction extracts a 5-bit value where a 4-bit nibble was intended. The kernel still runs and produces plausible output—no crash, no exception—but the dequantized weights are silently corrupted across all 16 SIMD lanes. Our pipeline catches this automatically: Z3 finds the counterexample `0x80` in 0.02 s (Appendix A).

The problem: Quantized LLM kernels—such as the Q4_0 and Q8_0 dot products in llama.cpp [1]—pack 4–8-bit integer weights into SIMD registers and compute through sequences of nibble extraction, widening multiplication, and cross-lane reduction on RISC-V Vector (RVV) hardware [2]. The correctness of this integer datapath spans multiple abstraction layers: the intended algorithm (expressed in a compiler IR), the generated vector assembly, and the hardware's interpretation of each instruction per the ISA specification. A bug at *any* layer—a wrong shift, a missing sign extension, an incorrect widening multiply—produces arbitrary bit-level corruption, not a small floating-point deviation. Testing cannot catch such bugs: the input space of a single Q4_0 block is 2^{384}, and the erroneous outputs are plausible, not crashed.

Why existing tools are insufficient: Formal verification tools exist for individual layers, but none spans the full path. Alive2 [3] verifies LLVM IR optimizations but stops before code generation—it never sees the assembly. Islaris [4] verifies machine code against the authoritative Sail ISA model, but targets systems code (pKVM) and requires interactive Coq proofs. MLIR-TV [5] validates MLIR transformations but does not connect to assembly or hardware semantics. ARM-TV [6] validates LLVM's AArch64 backend but uses a hand-written instruction lifter (not an authoritative ISA spec) and does not handle vector instructions. Table XIV in Section VI shows that no prior work covers more than two of the five key dimensions: ISA specification, compiler IR, vector operations, full automation, and ML kernels.

Why the problem is tractable: The quantized integer datapath is pure bitvector arithmetic: no floating-point rounding, no dynamic memory allocation, no control-flow divergence within the vector pipeline. This places the verification problem in QF_BV—a decidable theory with mature solvers [7]—enabling mathematical certainty for *all* inputs, not bounded model checking or random testing.

Contributions: We present a three-layer formal verification pipeline that, for the first time, spans from compiler IR to ISA formal specification for real-world ML kernels:

1) **Three-layer architecture.** We prove equivalence across three representation boundaries: MLIR ↔ LLVM IR (specification consistency), LLVM IR ↔ RVV assembly (codegen correctness), and RVV assembly ↔ Sail ISA spec (by construction via Isla). The MLIR and LLVM IR serve as hand-written golden references, not compiler output; the pipeline verifies the assembly against these specifications, not the compiler itself.
2) **Scalability techniques for vector ISA traces.** We identify and solve three engineering challenges that arise when applying symbolic ISA execution to SIMD vector instructions (Section III-G): (a) *per-element AST*

https://doi.org/

decomposition splits Isla's monolithic 256-bit formulas into individual SEW-bit expressions, reducing Z3 solver time from >10 min to <1 s; (b) *mask decomposition* avoids 2^{VL} path explosion in masked instructions by factoring into an unmasked trace, a spec-level ITE policy, and an algebraic lemma; (c) *linear composition* via Z3 `substitute()` scales with instruction count, not input width.

3) **End-to-end kernel verification.** We verify thirteen kernels covering all ten llama.cpp quantization formats: five basic/asymmetric dot products (Q4_0, Q4_1, Q5_0, Q5_1, Q8_0) and all five K-quant formats (Q2_K through Q6_K)—plus vector addition, ReLU, and 1D convolution, all fully automated, completing in <30 s total. We additionally prove VLEN portability: per-element formulas are identical across vector lengths (verified at VL=8 and VL=16).
4) **Reusable, ISA-portable pipeline.** The infrastructure (parser, decomposer, composer) is ISA-agnostic: porting to a new ISA requires replacing only the Sail model and regenerating traces. Adding a kernel requires only new specification files. We give concrete recipes for both (Section V-I). All code is open-source.

Paper structure: Section II introduces the technical foundations. Section III presents the three-layer methodology. Section IV describes the implementation. Section V evaluates the approach on three kernels. Section VI discusses related work. Section VII concludes.

II. Background

A. Sail and Isla

Sail [8] is a domain-specific language for defining instruction set architecture (ISA) specifications. The RISC-V community maintains an official Sail model covering the base ISA and standard extensions [9], including the Vector extension (RVV 1.0). The Sail model serves as the *authoritative* specification: it defines, for each instruction encoding, the exact effect on architectural state in terms of register reads, computations, and register writes.

Isla [10] is a symbolic execution engine for Sail. Given an instruction encoding and initial symbolic state, Isla explores all execution paths through the Sail specification and produces an SMT trace in S-expression format. Each trace declares symbolic bitvector inputs (register contents before execution), defines intermediate computations via SMT operations (bitvector arithmetic, extraction, concatenation), and records the final register writes. For a typical RVV arithmetic instruction with VLEN=256, the trace is 100–200 lines of S-expressions. Listing 1 shows an excerpt from the Isla trace for `vand.vx` (vector AND with scalar).

The trace shows that Isla reads vector register `vr1` and scalar register `x13`, computes per-element AND operations, assembles the results via zero-extension and OR into a single 256-bit value, and writes it to `vr4`. Our parser must reverse this assembly pattern to extract individual element formulas.

Listing 1. Isla trace excerpt for `vand.vx` (simplified).

```
(trace
  (read-reg |vr1| nil v332) ; v332: BV(256)
  (read-reg |x13| nil v341) ; v341: BV(64)
  (define v350 ; per-element AND
   (bvor
    ((_ zero_extend 248)
     (bvand ((_ extract 7 0) v332)
            ((_ extract 7 0) v341)))
    (bvshl ((_ zero_extend 248)
     (bvand ((_ extract 15 8) v332)
            ((_ extract 7 0) v341)))
     #x...08) ; shift by 8
    ...)) ; 16 elements total
  (write-reg |vr4| nil v350))
```

B. MLIR and the Arith Dialect

MLIR [11] is a compiler infrastructure framework supporting multiple intermediate representations (*dialects*) at different abstraction levels. The `arith` dialect provides integer arithmetic operations—addition (`arith.addi`), multiplication (`arith.muli`), bitwise AND (`arith.andi`), logical shift (`arith.shrui`), sign extension (`arith.extsi`)—with formally defined modular bitvector semantics. The `vector` dialect provides retargetable vector abstractions including element-wise operations and reductions (`vector.reduction <add>`). Recent work has formalized all 26 `arith` operations in SMT [12], confirming that MLIR's `arith` semantics correspond exactly to SMT bitvector theory.

C. RISC-V Vector Extension

The RISC-V Vector extension (RVV 1.0) [2] provides data-parallel instructions with configurable element width (SEW) and vector length (VLEN). Instructions relevant to quantized inference include: `vand.vx` (bitwise AND with scalar), `vsrl.vx` (logical right shift by scalar), `vsub.vx` (subtract scalar), `vwmul.vv` (widening multiply, SEW→2×SEW), `vwmacc.vv` (widening multiply-accumulate), and `vwredsum.vs` (widening reduction sum). Widening instructions produce results at double the source element width, and the reduction instruction accumulates all vector elements into a single scalar.

D. Quantized Inference: Q4_0 Format

The Q4_0 quantization format [1] packs two 4-bit integer values per byte. Each block of 32 values stores 16 packed bytes (`qs[0..15]`) and one FP16 scale factor d. The dot product of a Q4_0 block with a Q8_0 block (`y_qs[0..31]`) computes:

$$\texttt{sumi} = \sum_{i=0}^{15} \left[(q_i^{\text{lo}} - 8) \cdot y_i^{\text{lo}} + (q_i^{\text{hi}} - 8) \cdot y_i^{\text{hi}} \right] \tag{1}$$

where $q_i^{\text{lo}} = \texttt{qs}[i]\ \&\ \texttt{0xF}$ and $q_i^{\text{hi}} = \texttt{qs}[i] \gg 4$ are the low and high nibbles, and $y^{\text{lo}} = \texttt{y\_qs}[0{:}15]$, $y^{\text{hi}} = \texttt{y\_qs}[16{:}31]$ are the first and second 16-element halves of the 32-element Q8_0 block. The final result is $d_x \cdot d_y \cdot \texttt{sumi}$. Our verification

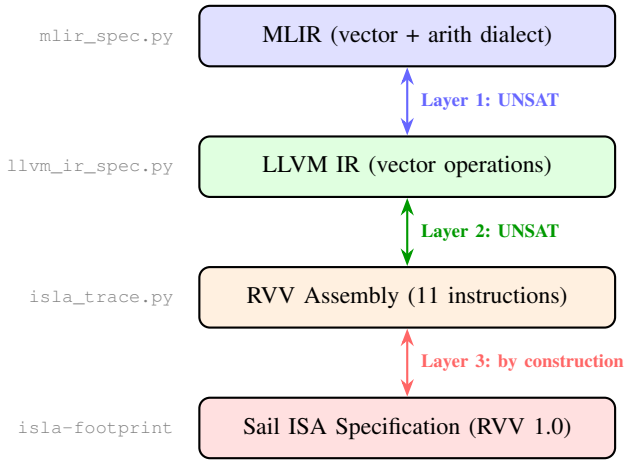

Fig. 1. Three-layer verification architecture. Each double arrow represents a Z3 bitvector equivalence proof. Layer 3 holds by construction: Isla symbolically executes the Sail specification.

targets the integer computation (Eq. 1); the FP16 scale is out of scope (§V-H).

The corresponding C code in llama.cpp's `ggml_vec_dot_q4_0_q8_0` (RVV path) computes this via a sequence of vector operations: load packed bytes, extract nibbles with AND/shift, subtract bias, perform widening multiply-accumulate, and reduce to a scalar sum. The RVV implementation uses 11 instructions per block (Table V), exploiting widening operations to maintain precision without explicit intermediate storage.

III. Methodology

A. Problem Statement

Let $MLIR(\vec{x})$, $LLVM(\vec{x})$, and $ISA(\vec{x})$ denote the functions computed by the kernel at the MLIR, LLVM IR, and ISA levels respectively, where $\vec{x}$ represents the symbolic input vector registers. We wish to prove:

$$\forall\, \vec{x}.\ MLIR(\vec{x}) = LLVM(\vec{x}) = ISA(\vec{x}) \tag{2}$$

We decompose this into three layer proofs. Each is discharged by showing that the negation is UNSAT in the theory of fixed-width bitvectors (QF_BV). Since QF_BV is decidable, the proofs are complete: UNSAT means equivalence holds for *all* inputs, not just a bounded subset.

B. Architecture Overview

Figure 1 shows the three-layer verification architecture. Each layer connects two adjacent representations through a Z3 bitvector equivalence proof. If all layers are UNSAT (no counterexample exists), the entire chain from MLIR to ISA specification is provably correct for all inputs.

C. Layer 3: ISA Semantics via Isla

For each RVV instruction in the kernel, we invoke Isla's `isla-footprint` on the official Sail RISC-V model with the concrete instruction encoding. Isla symbolically executes the Sail specification and outputs an SMT trace. The trace is a faithful rendering of the Sail semantics: it preserves every bitvector operation, sign extension, extraction, and conditional that the Sail model specifies for that instruction and configuration (`vtype`, `vl`).

The key property of Layer 3 is that it holds *by construction*: the trace *is* the Sail specification, symbolically evaluated. There is no separate proof obligation; the trust lies in Isla's symbolic execution engine.

To additionally validate the traces, we prove each auto-parsed Isla formula equivalent to an independently hand-written per-element specification (§V-B). This cross-check catches both parser bugs and potential Isla issues. All 81 element-level checks return UNSAT.

D. Layer 2: Kernel Composition and Codegen Verification

Per-element decomposition: Isla produces a single VLEN-wide (256-bit) formula that assembles all vector elements via nested `bvor`/`zero_extend` operations. Asking Z3 to reason about the full 256-bit formula causes solver timeouts (>10 minutes). We solve this by decomposing at the AST level *before* conversion to Z3: the parser walks the S-expression tree to identify per-element sub-expressions (recognized by the `((_ zero_extend N) elem)` assembly pattern), yielding individual SEW-bit formulas. This reduces solver time from minutes to under one second per element.

Instruction composition: Given per-element formulas for each instruction, we compose the multi-instruction kernel by register chaining. Let $f_i^{(k)}$ denote the formula for element i of the output register after instruction k. Each instruction reads one or more input registers; for each input register r that was written by a previous instruction j, we substitute the prior output:

$$f_i^{(k)} = g_i^{(k)}\left[r[i] \mapsto f_i^{(j)}\right] \tag{3}$$

where $g_i^{(k)}$ is the raw Isla formula with symbolic inputs, and the substitution replaces the symbolic register read with the concrete expression from instruction j.

For widening instructions (e.g., `vwmul.vv`: 8-bit $\rightarrow$ 16-bit), the output element width (16 bits) differs from the input element width (8 bits). The composer must track per-register element widths: when a subsequent instruction reads the widened register, it must extract 16-bit elements rather than 8-bit elements. For the Q4_0 kernel, three distinct element widths coexist: 8-bit (ALU), 16-bit (widening multiply output), and 32-bit (reduction output).

Handling multiple element widths: The widening multiply `vwmul.vv v8,v6,v2` reads 8-bit elements from `v6` and `v2`, and writes 16-bit elements to `v8` (actually occupying register pair `v8:v9` in RVV). The subsequent `vwmacc.vv v8,v7,v3` reads the 16-bit elements from `v8` (as its accumulator) and 8-bit elements from `v7` and `v3`, producing updated 16-bit elements. The final `vwredsum.vs` reads 16-bit elements and produces a single 32-bit result. Our composer handles this by maintaining an element-width tag per register, adjusting the element extraction width during substitution.

Worked example: two-instruction composition: Consider composing `vand.vx v4,v1,a3` followed by `vsub.vx v6,v4,a5`. After parsing, `vand` produces per-element formulas $e_i^{\text{and}} = \texttt{v1}[i]\ \&\ \texttt{a3}[7{:}0]$. The `vsub` trace has input symbols for `v4` and `a5`, producing $e_i^{\text{sub}} = \texttt{v4}[i] - \texttt{a5}[7{:}0]$. Composition substitutes $\texttt{v4}[i] \mapsto e_i^{\text{and}}$, yielding:

$$e_i^{\text{composed}} = (\texttt{v1}[i]\ \&\ \texttt{a3}[7{:}0]) - \texttt{a5}[7{:}0]$$

This extends to the full 11-instruction kernel by chaining all outputs to inputs sequentially. The final composed formula for element i of the kernel output is:

$$\begin{aligned} e_i = \texttt{sext}_{16}\big(((\texttt{x\_qs}[i]\ \&\ \texttt{0xF}) - 8) \cdot \texttt{y\_lo}[i]\big) \\ + \texttt{sext}_{16}\big(((\texttt{x\_qs}[i] \gg 4) - 8) \cdot \texttt{y\_hi}[i]\big) \end{aligned} \quad (4)$$

where all operations are in the appropriate bitvector widths (8-bit inputs, 16-bit intermediate, 32-bit final sum). The kernel result is $\texttt{sumi} = \sum_{i=0}^{15} \texttt{sext}_{32}(e_i)$.

Codegen verification: The composed kernel formula is compared against the LLVM IR specification. Both are expressed over the same symbolic input variables. The Z3 query `composed_isla` $\neq$ `llvm_ir_result` is checked: UNSAT means the LLVM-to-RVV code generation is correct for all inputs.

E. Layer 1: Compiler IR Equivalence

The MLIR and LLVM IR representations of the kernel are parsed into Z3 expressions. Each IR operation maps to a Z3 bitvector operation (Table I). The equivalence proof checks: `mlir_result` $\neq$ `llvm_result` is UNSAT.

Note that Layer 1 proves equivalence between two provided specifications, not that a particular compiler pass preserves semantics. In a production flow, the LLVM IR would be produced by MLIR's lowering passes; our proof verifies that the two representations compute the same function, regardless of how they were generated.

Since MLIR's `arith` dialect semantics are defined as modular bitvector arithmetic [12], and LLVM IR's integer operations have the same semantics, the mapping to Z3 is direct and does not require approximation or abstraction.

The kernels use 8 distinct operation types across the two IRs (Table I). Each maps to exactly one Z3 bitvector operation, preserving bit-exact semantics. The sign extension (`arith.extsi` / `sext`) has no explicit RVV counterpart because the widening instructions (`vwmul`, `vwmacc`, `vwredsum`) implicitly sign-extend their operands. This implicit extension is captured in the Isla trace and verified against the explicit extension in the compiler IR.

F. Pipeline Flow

Figure 2 shows the end-to-end pipeline. Given RVV instruction encodings, `isla-footprint` produces SMT traces. The traces are parsed, decomposed per-element, composed into a kernel formula, and compared against the compiler IR specifications.

TABLE I
MAPPING FROM COMPILER IR OPERATIONS TO Z3 AND RVV.

MLIR	LLVM IR	Z3	RVV
`arith.andi`	`and`	`bvand`	`vand.vx`
`arith.shrui`	`lshr`	`bvlshr`	`vsrl.vx`
`arith.shli`	`shl`	`bvshl`	(axiomatized)
`arith.subi`	`sub`	`bvsub`	`vsub.vx`
`arith.extsi`	`sext`	sign_ext	(implicit)
`arith.muli`	`mul`	`bvmul`	`vwmul.vv`
`arith.addi`	`add`	`bvadd`	`vwmacc.vv`
`vector.reduction`	`reduce.add`	Σ	`vwredsum.vs`

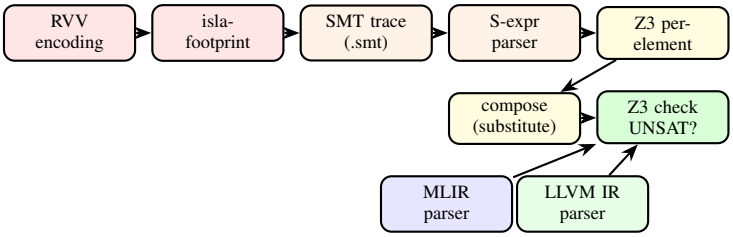

Fig. 2. Pipeline flow. Isla traces are parsed into per-element Z3 expressions, composed via substitution, and checked against MLIR and LLVM IR specifications.

G. Scalability Techniques

Applying symbolic ISA execution to SIMD vector instructions raises three scalability challenges absent in scalar verification.

Challenge 1: Monolithic vector formulas: Isla produces a single VLEN-wide (256-bit) formula per instruction, assembling all vector elements via nested `bvor`/`zero_extend` operations. Z3 cannot efficiently reason about these monolithic formulas: a single `vwmul.vv` at VLEN=256 causes solver timeouts exceeding 10 minutes. We decompose at the S-expression AST level *before* Z3 conversion, recognizing the `((_ zero_extend` N`)` $elem$`)` assembly pattern to extract individual SEW-bit sub-formulas. Result: each element solves in $<$1 s instead of $>$10 min for the monolithic formula. This decomposition is sound because RVV lane-parallel instructions compute each element independently: element i's output depends only on element i of its source operands, with no cross-lane data flow. Reduction instructions (`vredsum`, `vwredsum`) are an exception—they depend on all elements—and are verified as a single formula, not per-element.

Challenge 2: Masked instruction path explosion: Masked vector instructions (e.g., `vsub.vx v4,v4,a5,v0.t`) cause 2^{VL} path explosion in Isla because the Sail model branches on each mask bit independently. We decompose masked operations into three independently verified components: (a) an Isla trace for the mask computation itself (e.g., `vmnand.mm`, which is unmasked and traces normally); (b) the mask-undisturbed ITE policy from the RVV specification; and (c) an algebraic equivalence lemma verified in Z3. This is general: *any* masked RVV arithmetic instruction can be decomposed this way.

TABLE II
SCALABILITY CHALLENGES AND THEIR SOLUTIONS.

Challenge	Naïve	Our solution
256-bit formula	$>$10 min (timeout)	$<$1 s/element
Masked instruction	2^{16} paths (timeout)	3 lemmas, $<$1 s
n-instr. kernel	exponential risk	$O(n)$ substitutions

Challenge 3: Composition without blowup: We compose multi-instruction kernels via Z3 `substitute()`, which performs syntactic replacement without simplification. The final equivalence check operates on small per-element formulas (8–32 bits), not the full 256-bit vector. Total cost: $O(n)$ substitution steps plus one Z3 solve per element.

H. Soundness Argument

The verification is sound under the following trust assumptions:

1) The Sail RISC-V model [9] correctly specifies RVV 1.0 semantics. (Maintained by the RISC-V community.)
2) Isla [10] faithfully performs symbolic execution of Sail. (Tested extensively by Armstrong et al.)
3) The Z3 SMT solver is correct. (Standard assumption in SMT-based verification.)
4) Our parsers correctly translate S-expressions, MLIR, and LLVM IR to Z3 bitvector expressions.

Assumption (4) is the weakest. We validate it through three independent mechanisms:

1) *Per-element cross-checking* (§V-B): all 81 auto-parsed element formulas across 6 instruction types are proved equivalent to independently hand-written specifications (100% coverage of active elements).
2) *Structural simplicity*: the Isla trace parser handles only 14 SMT operations (10 bitvector arithmetic, 3 type conversions, and `ite`). Each maps to exactly one Z3 API call. The parser contains no approximation, abstraction, or case-splitting logic.
3) *End-to-end consistency*: the three-layer proof itself serves as a cross-check—the Isla-derived formulas, MLIR formulas, and LLVM IR formulas are constructed by three independent parsers from three independent inputs. All three agreeing (UNSAT) is evidence that no single parser is silently wrong.

These mechanisms provide strong evidence for parser correctness but do not formally eliminate the parsers from the TCB. A parser bug would need to produce formulas that are simultaneously wrong yet equivalent to both hand-written specs and two independently parsed compiler IR representations—a highly constrained failure mode, though not impossible in principle.

Trusted computing base: The TCB consists of: (a) the Sail RISC-V model (∼30K lines of Sail, maintained by the RISC-V community); (b) the Isla symbolic execution engine (∼18K lines of Rust); (c) the Z3 SMT solver (∼500K lines of C++); and (d) our parsers (1,268 lines of Python for `isla_trace.py`, `mlir_spec.py`, and `llvm_ir_spec.py`). Components (a)–(c) are widely used and independently validated. Our parsers are the smallest component and the least externally validated, but the three cross-checking mechanisms above constrain the space of undetected bugs to those that preserve equivalence across all three representations—a highly unlikely scenario for 14 simple operation mappings.

TABLE III
IMPLEMENTATION COMPONENTS.

Component	File	LOC
Isla trace parser + composer	`isla_trace.py`	908
MLIR spec extractor	`mlir_spec.py`	174
LLVM IR spec extractor	`llvm_ir_spec.py`	186
Per-instruction verifier	`verify_q4_0_auto.py`	215
Three-layer verifiers	`verify_three_layer*.py`	1,079
VLEN portability verifier	`verify_vlen_portable.py`	157
Instruction encoder	`rvv_encode.py`	118
Declarative framework	`rvs.py`	186
Total (Python)		**3,714**
Rust CLI (`tool/`)	`rvs`	2,012

IV. IMPLEMENTATION

The pipeline is implemented in 3,714 lines of Python, with an additional 2,012-line Rust CLI that provides declarative TOML-based verification. Table III summarizes the components.

A. Isla Trace Parser

`isla_trace.py` is the largest component. It implements: (1) an S-expression tokenizer and recursive-descent parser; (2) an Isla trace analyzer that identifies `read-reg` and `write-reg` events, building a map from register names to symbolic expressions; (3) a recursive converter from Isla's SMT-like AST to Z3 Python API calls, supporting all bitvector operations used in RVV traces: `bvadd`, `bvsub`, `bvmul`, `bvand`, `bvor`, `bvxor`, `bvshl`, `bvlshr`, `bvashr`, `extract`, `sign_extend`, `zero_extend`, `concat`, and `ite`; (4) a per-element decomposer (`get_elements()`) that walks the AST to extract individual element sub-formulas.

The per-element decomposition is critical for scalability. Isla's output for a vector arithmetic instruction at VLEN=256, SEW=8 assembles all vector elements into a single 256-bit formula. The assembly pattern is:

$$\textit{result} = \big(\texttt{zext}_{248}(e_0)\big) \,|\, \big(\texttt{zext}_{248}(e_1) \ll 8\big) \,|\, \cdots \,|\, \big(\texttt{zext}_{248}(e_{15}) \ll 120\big)$$

where e_i is the SEW-bit formula for element i. Asking Z3 to simplify $\texttt{Extract}(7, 0, \textit{big\_formula})$ on this expression causes the solver to reason about the entire 256-bit structure, leading to timeouts exceeding 10 minutes.

Our decomposer walks the S-expression AST to locate the `((_ zero_extend N) elem)` nodes. Each such node contains the complete per-element formula as its child. The decomposer collects these in traversal order (which corresponds

Listing 2. Q4_0 kernel in MLIR (excerpt).

```
func.func @q4_dot_block(
   %x_qs: vector<16xi8>,
   %y_lo: vector<16xi8>,
   %y_hi: vector<16xi8>) -> i32 {
  %mask = arith.constant dense<15>
                  : vector<16xi8>
  %lo = arith.andi %x_qs, %mask
                  : vector<16xi8>
  %hi = arith.shrui %x_qs, %shift
                  : vector<16xi8>
  %lo_s = arith.subi %lo, %bias
                  : vector<16xi8>
  // ... sign extend, multiply, accumulate
  %result = vector.reduction <add>, %prod_32
          : vector<16xi32> into i32
  return %result : i32
}
```

to element index order due to Isla's left-to-right assembly) and converts each to a separate Z3 expression. This yields 16 independent SEW-bit formulas, each solvable in under one second.

B. Compiler IR Parsers

`mlir_spec.py` and `llvm_ir_spec.py` parse their respective IR formats using regular expressions over the SSA form. Each recognized operation is converted to a Z3 bitvector call via the mapping in Table I. Both parsers maintain SSA value maps (`%name` → `[Z3_expr]`), track vector element types, and support type conversions (`sext`, `zext`, `trunc`) and horizontal reductions.

Listing 2 shows the MLIR specification of the Q4_0 kernel used in our evaluation.

C. Kernel Composer

The `KernelComposer` class (within `isla_trace.py`) automates multi-instruction composition. It maintains a per-register element-width-aware state map. For each instruction step, it: (1) loads the Isla trace; (2) identifies which input registers overlap with previously computed outputs; (3) substitutes the prior output expressions for the new input symbols via Z3 `substitute()`; (4) stores the resulting expressions as the new register state.

Register remapping allows reusing a single Isla trace for multiple instances of the same instruction with different operands. For example, the Q4_0 kernel contains two `vsub.vx` instructions: `vsub.vx v6,v4,a5` and `vsub.vx v7,v5,a5`. Both use the same Isla trace (same opcode), but the composer remaps the source register from `v4` to `v5` and the destination from `v6` to `v7` for the second instance.

D. Instruction Encoding

`rvv_encode.py` automates the generation of instruction encodings for Isla. Given an RVV assembly mnemonic (e.g., `vwmul.vv v8,v6,v2`), it invokes the RISC-V GNU assembler (`riscv64-linux-gnu-as`) to assemble the instruction, extracts the 4-byte encoding via `objcopy`, and returns it in the little-endian hex format that `isla-footprint` expects. This eliminates manual encoding lookup and reduces the risk of encoding errors.

TABLE IV
ISLA TRACE SIZES FOR THE Q4_0 KERNEL INSTRUCTIONS.

Trace file	Lines	SEW config
`vand_trace_sew8.smt`	169	SEW=8
`vsrl_trace_sew8.smt`	169	SEW=8
`vsub_trace_sew8.smt`	169	SEW=8
`vwmul_sew8_trace.smt`	209	SEW=8
`vwmacc_sew8_trace.smt`	209	SEW=8
`vwredsum_sew16_trace.smt`	144	SEW=16

E. Multi-SEW Tracing

The Q4_0 kernel uses mixed element widths: SEW=8 for ALU operations (nibble extraction, bias subtraction, widening multiply) and SEW=16 for the widening reduction sum. Since Isla requires a fixed `vtype` per trace, we generate separate Sail IR files for each SEW setting by patching the `init_model` function in the Sail source and rebuilding with Isla's `isla-sail` compiler. Three configurations are used:

- **SEW=8, LMUL=1, VL=16**: For 8-bit ALU operations (`vand`, `vsrl`, `vsub`) and widening operations (`vwmul`, `vwmacc`) that read 8-bit elements.
- **SEW=16, LMUL=2, VL=16**: For the widening reduction (`vwredsum.vs`) that reads 16-bit elements and produces a 32-bit result. LMUL=2 is required because the 16-bit source vector occupies two physical registers.
- **SEW=32, LMUL=1, VL=8**: Used during early development for simpler instruction verification (e.g., `vmacc.vv`) but not required for the Q4_0 kernel.

Each Sail IR file is approximately 14 MB and takes 2–3 minutes to build. Once built, `isla-footprint` produces a trace in 1–5 seconds per instruction. The Isla traces for the Q4_0 kernel range from 144 to 209 lines of S-expressions (Table IV).

V. EVALUATION

A. Verified Kernel

We verify the integer core of llama.cpp's `ggml_vec_dot_q4_0_q8_0`, which computes one block (16 Q4_0 nibble pairs) of the quantized dot product using 11 RVV instructions. Table V lists the instruction sequence with encodings and element widths.

The three vector loads (#1–3) bring the Q4_0 packed nibble bytes (`x_qs`) and Q8_0 values (`y_lo`, `y_hi`) into vector registers. Instructions #4–5 extract low and high nibbles from each packed byte: `vand.vx` masks with `0x0F` to get the low nibble, and `vsrl.vx` shifts right by 4 to get the high nibble. Instructions #6–7 subtract the zero-point bias 8 from each nibble (converting unsigned $[0, 15]$ to signed $[-8, 7]$).

Instructions #8–9 are the computational core: `vwmul.vv` performs widening signed multiply ($8 \rightarrow 16$ bits), computing $\text{sext}(\text{lo}_i) \times \text{sext}(y_i^{\text{lo}})$, and `vwmacc.vv` performs widening

TABLE V
VERIFIED RVV INSTRUCTIONS IN THE Q4_0 KERNEL. SEW INDICATES THE SOURCE ELEMENT WIDTH; WIDENING INSTRUCTIONS PRODUCE 2×SEW OUTPUT.

#	Instruction	SEW	Operation
1–3	`vle8.v` vN,(aM)	8	load ×3
4	`vand.vx v4,v1,a3`	8	nibble mask
5	`vsrl.vx v5,v1,a4`	8	nibble shift
6	`vsub.vx v6,v4,a5`	8	bias subtract
7	`vsub.vx v7,v5,a5`	8	bias subtract
8	`vwmul.vv v8,v6,v2`	8→16	widen multiply
9	`vwmacc.vv v8,v7,v3`	8→16	widen MAC
10	`vmv.v.x v10,x0`	16	zero init
11	`vwredsum.vs v10,v8,v10`	16→32	widen reduce

TABLE VI
PER-INSTRUCTION VERIFICATION RESULTS. EACH INSTRUCTION IS DECOMPOSED INTO VL=16 PER-ELEMENT PROOFS. ALL RETURN UNSAT.

Instruction	SEW	Spec	Elems	Result
`vand.vx`	8	$v1[i]$ & `a3`	16	UNSAT
`vsrl.vx`	8	`LShR`($v1[i]$, `a4`)	16	UNSAT
`vsub.vx`	8	$v4[i] -$ `a5`	16	UNSAT
`vwmul.vv`	8→16	`sext`($v6[i]$) × `sext`($v2[i]$)	16	UNSAT
`vwmacc.vv`	8→16	$vd[i]$ + `sext`($v7[i]$) × `sext`($v3[i]$)	16	UNSAT
`vwredsum.vs`	16→32	$v10[0] + \sum$ `sext`($v8[i]$)	1	UNSAT
Total element proofs			**81**	**all UNSAT**

multiply-accumulate, adding $\text{sext}(\text{hi}_i) \times \text{sext}(y_i^{\text{hi}})$ to the result. After these two instructions, `v8` contains 16 elements of 16-bit partial products.

Instruction #10 initializes the accumulator to zero, and instruction #11 (`vwredsum.vs`) reduces all 16 elements of `v8` into a single 32-bit sum, producing the integer dot product `sumi` from Equation 1.

Note that the vector loads (#1–3) are not verified at the functional level—they are axiomatized as reading fresh symbolic values from memory. The verification proves that *given* the loaded values, the subsequent computation correctly produces the dot product. Address computation and memory consistency are outside our current scope.

B. Per-Instruction Verification

Each of the 6 non-load instructions is verified individually. The Isla trace is auto-parsed and decomposed into per-element formulas (up to `VLEN/SEW` elements per register; we verify the first VL=16 active elements). Each active element is proved equivalent to its hand-written specification. Table VI shows the results.

All 81 element-level proofs return UNSAT, confirming the auto-parser's correctness. The reduction instruction (`vwredsum.vs`) produces a single scalar result rather than per-element outputs, hence 1 proof instead of 16.

The per-element decomposition is essential: without it, the full 256-bit formula for `vwmul.vv` causes Z3 to exceed a 10-minute timeout. With decomposition, each 16-bit element formula solves in <1 second.

TABLE VII
THREE-LAYER VERIFICATION RESULTS. TOTAL WALL-CLOCK TIME INCLUDES PARSING, Z3 FORMULA CONSTRUCTION, AND SOLVER TIME.

Layer	What is proved	Result	Time
Per-instr	Isla == hand-spec (×6)	UNSAT	20.6 s
Layer 1	MLIR == LLVM IR	UNSAT	$<$0.1 s
Layer 2	LLVM IR == Isla composed	UNSAT	0.1 s
Layer 3	Isla traces == Sail spec	by constr.	—
E2E	MLIR == Isla (transitive)	UNSAT	$<$0.1 s

C. Three-Layer Results

Table VII summarizes the verification across all three layers. The end-to-end proof (MLIR == Isla, transitively through LLVM IR) is also checked directly, confirming consistency.

The per-instruction verification (20.6 s) dominates total time because it checks 81 individual element formulas across 6 instructions, each requiring a separate solver invocation. Layers 1 and 2 complete in under 0.1 s because they compare pre-composed expressions that share symbolic variables—Z3 recognizes structural equivalence rapidly.

Discussion: The fast Layer 1 and Layer 2 times deserve explanation. Both the MLIR and LLVM IR parsers produce Z3 expressions using the same symbolic variable names (e.g., `x_qs_0` through `x_qs_15`). The Z3 expressions for the MLIR and LLVM IR specifications are structurally identical because the operations are the same (AND, shift, subtract, sign-extend, multiply, add, reduce); only the syntax differs. Z3 recognizes this structural equivalence without exploring the bitvector search space.

For Layer 2, the composed Isla formula is also structurally similar to the LLVM IR formula after substitution—both express the same sequence of operations over the same inputs. The solver time is dominated by Z3's internal simplification rather than SAT search.

This is a positive result: it means the pipeline scales well. The cost grows linearly with the number of instructions (one trace parse + one composition step per instruction), not exponentially with input width.

Isla trace generation time: The one-time cost of generating Isla traces is not included in the verification time above. Building a Sail IR file for a given SEW configuration takes 2–3 minutes. Running `isla-footprint` for a single instruction takes 1–5 seconds. For the Q4_0 kernel (6 unique instruction types, 2 SEW configurations), the total trace generation time is approximately 6 minutes. These traces are generated once and reused across all verification runs.

D. Second Kernel: Q8_0 × Q8_0

To demonstrate that the pipeline generalizes beyond Q4_0, we verify a second kernel: llama.cpp's `ggml_vec_dot_q8_0_q8_0`. The Q8_0 format stores weights as signed 8-bit integers (no nibble packing), so the integer datapath is simpler: widening multiply followed by widening reduction, using only 2 RVV arithmetic instructions (Table VIII).

TABLE VIII
VERIFIED RVV INSTRUCTIONS IN THE Q8_0 KERNEL.

#	Instruction	SEW	Operation
1–2	`vle8.v vN,(aM)`	8	load ×2
3	`vwmul.vv v4,v1,v2`	8→16	widen multiply
4	`vwredsum.vs v6,v4,v6`	16→32	widen reduce

TABLE IX
THREE-LAYER VERIFICATION RESULTS FOR Q8_0 × Q8_0. THE SAME PIPELINE AND ISLA TRACES ARE REUSED FROM Q4_0.

Layer	What is proved	Result	Time
Layer 1	MLIR == LLVM IR	UNSAT	<0.1 s
Layer 2	LLVM IR == Isla composed	UNSAT	<0.1 s
Layer 3	Isla traces == Sail spec	by constr.	—
E2E	MLIR == Isla (transitive)	UNSAT	<0.1 s

Table IX shows the three-layer verification results. Both the MLIR and LLVM IR specifications are straightforward: sign-extend both input vectors to 16 bits, multiply element-wise, sign-extend to 32 bits, and reduce. The Isla composition reuses the same `vwmul` and `vwredsum` traces from the Q4_0 verification—no new trace generation is required.

The Q8_0 kernel completes in under 0.1 s total—faster than Q4_0 because the composition involves only 2 arithmetic instructions instead of 7. This result validates the "Generalization" claim: adding a new kernel required writing two specification files (32 and 24 lines of MLIR and LLVM IR, respectively) and a verification script, with zero changes to the parser, composer, or infrastructure code. The Isla traces, which represent the most expensive artifact to generate, are fully reused.

E. Third Kernel: Q5_0 × Q8_0

Q5_0 uses 5-bit weights: each value is encoded as a 4-bit nibble plus a 5th bit from a separate bitmask (`qh`). The decoded weight is:

$$w_i = (\texttt{nibble}_i + (\texttt{qh\_bit}_i \ll 4)) - 16$$

Since $\texttt{nibble}_i \in [0, 15]$ and $\texttt{qh\_bit}_i \ll 4 \in \{0, 16\}$ have disjoint bit positions, addition is equivalent to bitwise OR. The result equals $\texttt{nibble}_i$ when $\texttt{qh\_bit}_i$=1 and $\texttt{nibble}_i - 16$ when $\texttt{qh\_bit}_i$=0, covering the 5-bit signed range $[-16, 15]$.

The actual RVV implementation uses masked operations (`vlm.v`, `vmnand.mm`, masked `vsub.vx .mu`) to implement this conditional subtract. Isla cannot directly trace masked arithmetic instructions: the Sail model branches per mask bit, causing 2^{VL} path explosion. We decompose the masked subtract into three independently justified components: (a) a `vmnand.mm` Isla trace proving mask inversion (∼v0); (b) the mask-undisturbed ITE policy from the RVV specification ($\texttt{result}[i] = \texttt{ite}(\texttt{mask}[i], \texttt{op}, \texttt{keep})$); and (c) an algebraic lemma proving $\texttt{ite}(\neg\texttt{qh}, n-16, n) = n + \texttt{qh} \ll 4$ 16, verified in Z3. All six Q4_0 arithmetic traces are reused, plus the new `vmnand.mm` trace (7 total).

TABLE X
THREE-LAYER VERIFICATION RESULTS FOR Q5_0 × Q8_0. SEVEN ISLA TRACES (6 REUSED + VMNAND.MM); MASKED SUBTRACT DECOMPOSED VIA ISLA TRACE + RVV SPEC + ALGEBRAIC LEMMA.

Layer	What is proved	Result	Time
Layer 1	MLIR == LLVM IR	UNSAT	<0.1 s
Layer 2	LLVM IR == Isla composed	UNSAT	<0.1 s
Layer 3	Isla traces == Sail spec	by constr.	—
E2E	MLIR == Isla (transitive)	UNSAT	<0.1 s

TABLE XI
TEN ADDITIONAL VERIFIED KERNELS. ALL REUSE EXISTING ISLA TRACES; ONLY Q2_K REQUIRED ONE NEW TRACE (`VWMULU.VV`).

Kernel	Steps	Inputs	Time	Result
Q4_1×Q8_1	5 trace	2^{384}	<0.1 s	UNSAT
Q5_1×Q8_1	5 trace + 4 axiom	2^{640}	<0.1 s	UNSAT
Q4_K×Q8_K	5 trace	2^{384}	<0.1 s	UNSAT
Q5_K×Q8_K	5 trace + 4 axiom	2^{640}	<0.1 s	UNSAT
Q6_K×Q8_K	4 trace + 2 axiom	2^{384}	<0.1 s	UNSAT
Q3_K×Q8_K	3 trace + 2 axiom	2^{384}	<0.1 s	UNSAT
Q2_K×Q8_K	1 trace + axiom	2^{384}	<0.1 s	UNSAT
Vector add (i8)	1 trace	2^{256}	<0.1 s	UNSAT
ReLU (i8)	1 trace	2^{128}	<0.01 s	UNSAT
Conv1D (i8, k=16)	2 trace	2^{256}	<0.1 s	UNSAT

Table X shows the verification results. The Q5_0 kernel verifies in under 0.1 s, demonstrating that increasing quantization complexity (from 4-bit to 5-bit with bitmask) does not significantly increase verification cost. The input space is 2^{640} (5 input vectors × 16 elements × 8 bits). The only remaining axiom is the mask-undisturbed ITE policy, which is a definitional property of the RVV specification rather than an algebraic claim about the computation.

F. Additional Kernels

To validate the "free lunch" reuse property, we verify ten additional kernels using existing Isla traces with at most one new trace (`vwmulu.vv` for Q2_K).

Q4_1×Q8_1 and **Q5_1×Q8_1**: asymmetric-quantization variants of Q4_0 and Q5_0. The integer datapath is identical except the zero-point subtraction (`vsub.vx`) is omitted—the bias is handled by a floating-point minimum term outside the integer scope.

K-quant kernels (Q2_K through Q6_K): llama.cpp's K-quant formats use multi-level quantization with per-subblock scales and varying bit widths (2–6 bits). We verify the integer core of each sub-block: *Q4_K* is structurally identical to Q4_1 (same 5 traces). *Q5_K* is structurally identical to Q5_1 (5 traces + 4 axiom steps). *Q6_K* uses 6-bit values: $val = (lo \mid (qh \ll 4)) - 32$ (4 traces + 2 axiom). *Q3_K* uses 2-bit values with a 1-bit mask: $val = q_3 + 4 \cdot hmask - 4$ (range -4 to 3; 3 traces + 2 axiom). *Q2_K* is the most complex: a three-term unsigned product $\sum \text{zext}(q_2) \cdot \text{zext}(scale) \cdot \text{sext}(q_8)$. It requires a new `vwmulu.vv` (unsigned widening multiply) Isla trace,

which we verify separately against the ZeroExt × ZeroExt axiom.

Vector addition (`vadd.vv`): element-wise $c_i = a_i + b_i$. **ReLU** (`vmax.vx`): element-wise $\max(x_i, 0)$. **1D convolution**: reuses `vwmul` and `vwredsum` traces from Q8_0.

All ten verify in all three layers (UNSAT). Table XI summarizes the results. Each kernel required only new MLIR (∼15 lines) and LLVM IR (∼15 lines) specification files plus a TOML manifest. The entire K-quant sweep required zero new Isla trace generation (except the single `vwmulu.vv` trace for Q2_K), demonstrating the framework's composability.

G. Compiler Variant Equivalence

A direct application of the pipeline is proving that two different assembly sequences—emitted by different compilers or optimization levels—compute the same result. In the Q4_0 kernel, our verified sequence uses `vsub.vx v,v,8` for bias subtraction, while Clang 18.1 at `-O3` emits `vadd.vi v,v,-8` (a strength reduction exploiting two's complement: $-8 = \texttt{0xF8}$ in 8 bits). We prove in Z3 that (1) $x - 8 = x + \texttt{0xF8}$ for all 8-bit x, (2) both nibble pipelines (AND+sub vs. AND+add, shift+sub vs. shift+add) are equivalent, and (3) the full 16-element composed dot product—with 2^{384} inputs—is identical under either encoding. We additionally verify a second class of strength reduction: `vmul.vx v,v,2` (multiply by 2) versus `vadd.vv v,v,v` (add to self), proving equivalence for both plain and widening (8→16-bit) variants. All 7 proofs return UNSAT in <0.1 s total (`experiments/verify_vadd_vsub_equiv.py`).

This result is significant because it requires *no Isla traces*: the Z3 formulas are constructed directly from the instruction semantics at the bitvector level. Given any two candidate instruction sequences whose per-element semantics can be expressed in SMT, the pipeline can verify their equivalence without re-running symbolic execution. This enables a lightweight form of *translation validation*: given a compiler that emits sequence A and a reference that uses sequence B, our pipeline can certify $A \equiv B$ over all inputs.

H. Scope and Limitations

What is verified: Thirteen kernels covering all ten llama.cpp quantization formats. (1) Q4_0: nibble extraction, bias subtraction, widening multiply, widening MAC, and widening reduction (11 instructions, 2^{384} inputs). (2) Q4_1: same as Q4_0 without zero-point subtraction (5 instructions). (3) Q5_0: same as Q4_0 plus conditional 5th-bit bitmask (2^{640} inputs). (4) Q5_1: same as Q5_0 without zero-point subtraction. (5) Q8_0: widening multiply and reduction (4 instructions, 2^{256} inputs). (6) Q4_K: structurally identical to Q4_1 (per-subblock scale). (7) Q5_K: structurally identical to Q5_1. (8) Q6_K: 6-bit values with zero-point subtraction (4 traces + 2 axiom). (9) Q3_K: 2-bit values with 1-bit hmask (3 traces + 2 axiom). (10) Q2_K: three-term unsigned product (1 new trace + axiom). (11–13) Vector addition, ReLU, 1D convolution. All are verified end-to-end from MLIR to Sail ISA spec.

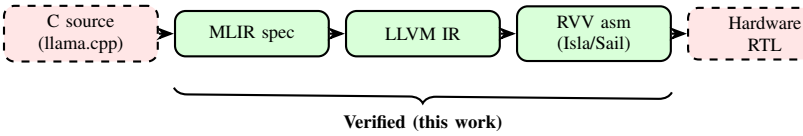

Fig. 3. Verification scope within the full inference pipeline. Solid boxes are verified; dashed boxes are out of scope.

Per-element formulas are additionally proved VLEN-portable (identical at VL=8 and VL=16).

What is not verified: (1) *Floating-point scale multiplication* ($d_x \cdot d_y$ applied after the integer sum). This requires FP theory in Z3, which is decidable but computationally expensive. (2) *Memory semantics*: vector loads are axiomatized as fresh symbolic values. We do not verify that addresses are correct. (2b) *Vector type configuration*: `vsetvli` sets SEW, LMUL, and VL at runtime. We verify that the Sail model's `vsetvli` correctly sets vtype (SEW=8, LMUL=1, vta=1, vma=0) and VL $= \min(\text{AVL}, \text{VLMAX})$ for all 2^{64} AVL values, covering all three execution paths (`experiments/verify_vsetvli.py`). (3) *Loop structure*: we verify one block iteration. The outer loop accumulates across blocks: $\texttt{sumf} \mathrel{+}= d_x \cdot d_y \cdot \texttt{sumi}_k$ for $k = 0, \ldots, n_b - 1$. Given that the single-block integer result $\texttt{sumi}_k$ is proved correct for all inputs, the loop is correct by induction: the base case ($k = 0$) is our single-block proof, and the inductive step is a scalar FP accumulation independent of the datapath. Verifying the branch and loop-counter instructions (`bne`, `addi`) at the Isla trace level is straightforward future work. (4) *Compiler codegen from C*: the mapping from llama.cpp's C source to the RVV assembly sequence is not in our verification chain. We verify the assembly against the Sail ISA spec, not against the C source. However, we validate that our verified sequence matches real compiler output: both GCC 13.3 and Clang 18.1 at `-O3` produce the same 11 RVV operations in the same dataflow order, differing only in register allocation, immediate encoding, and strength reductions whose equivalence we prove formally (Section V-G). (5) *Hardware RTL*: correctness of the hardware implementation against the Sail ISA spec is out of scope (addressed by complementary work such as riscv-formal [13]).

Figure 3 summarizes the verified and unverified portions of the full inference pipeline.

I. Portability and Reuse

A key design goal is that verifying a new kernel or porting to a new ISA should require *minimal per-task effort*. The pipeline cleanly separates ISA-agnostic components (S-expression parser, per-element decomposer, kernel composer, MLIR/LLVM IR parsers) from ISA-specific artifacts (Sail model, Isla traces, instruction encodings).

Recipe: adding a new kernel: Given a new quantized integer kernel (e.g., GEMM, depthwise convolution): (1) write an MLIR specification (∼20 lines); (2) write a matching LLVM IR specification (∼30 lines); (3) encode instructions

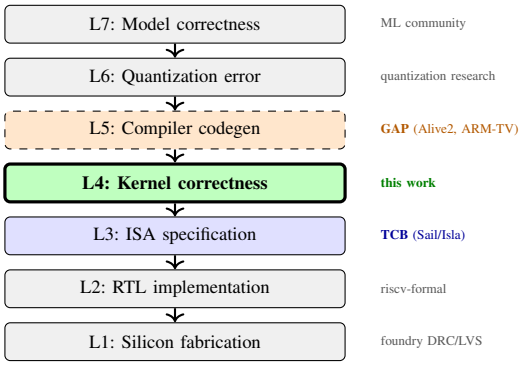

Fig. 4. Seven-layer verification landscape for quantized ML inference. L4 (green) is verified by this work. L3 (blue) is the trusted computing base. L5 (dashed) is the primary open gap.

and run `isla-footprint` (automated); (4) run verification (automated). Steps 3–4 are mechanical; steps 1–2 are the only per-kernel effort. The Q5_0 and Q8_0 kernels validate this recipe: both reuse Q4_0's traces and infrastructure without modification.

Recipe: porting to a new ISA: The pipeline requires a Sail model and Isla for the target ISA. Sail models exist for ARMv8-A (NEON), RISC-V, CHERI-MIPS, and x86 fragments [8]. To port to ARM: (1) replace the Sail IR with the ARM model; (2) re-run Isla for ARM traces; (3) adjust the LLVM IR spec for ARM instruction selection; (4) MLIR spec, parser, decomposer, and composer are *unchanged*. The MLIR specification is ISA-independent by design: it describes *what* the kernel computes, not *how*. The same MLIR spec serves as a cross-ISA golden reference, enabling *transitive cross-ISA equivalence*: if both backends verify against the same spec, they are equivalent to each other.

Cross-ISA proof: RVV × Wasm SIMD128: We demonstrate this concretely by verifying the Q8_0 dot product against WebAssembly SIMD128 semantics, axiomatized directly from the W3C specification [14]. The Wasm backend uses `i16x8.extend_*_i8x16_s` (sign-extend), `i32x4.dot_i16x8_s` (widening dot product), and `i32x4.add` (accumulate). We prove three results: (1) an *overflow lemma* showing that for all `i8` inputs, multiplying in `i16` width (MLIR) equals multiplying in `i32` width (Wasm), since $|a \cdot b| \leq 128^2 = 16384 < 2^{15}$; (2) MLIR spec = Wasm SIMD128 composition (UNSAT); and (3) combined with the existing RVV proof, transitivity gives RVV = MLIR = Wasm for all 2^{256} inputs. The Wasm verification adds 167 lines of Python and completes in $<$1 s; no Isla trace is needed since Wasm semantics are simple enough to axiomatize directly.

Component reuse summary: Table XII shows reuse scope. The top four rows are fully reusable across kernels and ISAs; only specs and traces need updating.

Scope decomposition: Figure 4 and Table XIII decompose the full verification space. The contribution is the *pipeline*—each additional kernel, VLEN, or ISA is engineering effort, not new research.

TABLE XII
COMPONENT REUSE. • = AS-IS, ○ = PARTIAL, — = NEW.

Component	New kernel	New ISA
S-expr parser + decomposer	•	•
Kernel composer	•	•
MLIR / LLVM IR parsers	•	•
MLIR specification	—	•
LLVM IR specification	—	—
Isla traces	○ (may reuse)	—
Sail IR file	•	—

VLEN portability: The per-element decomposition implies VLEN independence: each element's formula depends only on its index and SEW, not on VL or VLEN. We confirm this by generating traces at VL=8 and VL=16 for three instruction types (`vand.vx`, `vsrl.vx`, `vsub.vx`) and proving all 24 per-element formulas identical (UNSAT). This generalizes our proofs to any VLEN $\geq$ 128 (`verify_vlen_portable.py`).

VI. Related Work

Table XIV compares our approach with the most closely related work across five dimensions.

ISA-level verification: Islaris [4] is the closest prior work. It verifies machine code against Sail ISA semantics using Isla traces and Iris separation logic in Coq, demonstrating verification of the pKVM hypervisor on Arm. However, it targets systems code with explicit memory reasoning, requires interactive theorem proving (Coq proofs), and does not handle vector instructions or ML kernels. Our work extends Isla's use from systems code to computational kernel verification, achieves full automation via Z3, and handles mixed-SEW RVV instructions.

VeriISLE [15] verifies Cranelift's instruction selector rules against Sail ISA semantics, but operates on individual lowering rules rather than composed multi-instruction kernels.

Compiler translation validation: Alive2 [3] validates LLVM IR optimizations via bounded translation validation. It operates entirely within LLVM IR and does not extend to machine code or ISA specifications. ARM-TV [6] validates LLVM's AArch64 backend, finding 46 miscompilation bugs. It uses a hand-written instruction lifter rather than an authoritative ISA specification (ASL/Sail) and does not handle SIMD/vector instructions or connect to MLIR.

MLIR verification: MLIR-TV [5] validates MLIR-to-MLIR transformations. The First-Class Verification Dialects work [12] formalizes all 26 `arith` operations in SMT and found 5 upstream MLIR bugs. Bhat et al. [17] verify SSA peephole rewrites in Lean 4. All three stay within the MLIR ecosystem without connecting to assembly or ISA specifications.

ML kernel verification: Dubey et al. [16] check equivalence of GPU kernels at the PTX level but targets floating-point operations and does not use authoritative ISA specifications. QVIP [18] verifies robustness properties of quantized neural

TABLE XIII
SYSTEMATIC DECOMPOSITION OF THE VERIFICATION SPACE. ROWS SHOW VERTICAL LAYERS (MODEL TO SILICON) WITH CONCRETE LLAMA.CPP ARTIFACTS; COLUMNS SHOW HORIZONTAL CONCERNS WITHIN OUR LAYER (L4). • = DONE, ○ = AXIOMATIZED.

Dimension	Concrete artifact	Done	Remaining
Vertical layers			
L7: Model	`transformer_forward()`	Not ours (ML theory)	
L6: Quantization	`quantize_row_q4_0()`	Not ours (error bounds)	
L5: Compiler	`clang -march=rv64gcv`	Gap (future work)	
L4: Kernel	**`ggml_vec_dot_q4_0_q8_0()`**	**This work**	
L3: ISA spec	`sail-riscv` Sail model	Trusted (TCB)	
L2: RTL	`riscv-formal`	Others	
L1: Silicon	Fab DRC/LVS	Others	
L4 horizontal concerns			
Integer datapath		• (all 10 quant formats + 3 utility)	GEMM, softmax
Floating point		—	Hard (FP theory)
Memory		○ axiomatized	Addr., alignment
Control flow		1 iteration	Loops, masks
Concurrency		—	Very hard
Coverage width			
Kernels		13 (all 10 quant fmts + vecadd, ReLU, conv1d)	GEMM, softmax
VLEN		• portable (VL=8,16 proved identical)	—
LMUL		1	2, 4, 8
ISA		RISC-V (Sail) + Wasm SIMD128	ARM (ASL), x86

TABLE XIV
COMPARISON WITH RELATED WORK. COLUMNS INDICATE COVERAGE OF KEY DIMENSIONS. • = YES, ○ = PARTIAL, — = NO.

Work	ISA spec	Compiler IR	Vector	Automated	ML kernel
Islaris [4] (PLDI'22)	• Sail	—	—	○	—
Alive2 [3] (PLDI'21)	—	• LLVM	—	•	—
ARM-TV [6] (OOPSLA'25)	○ hand-written	• LLVM	—	•	—
VeriISLE [15] (ASPLOS'24)	• Sail	—	—	•	—
MLIR-TV [5] (CAV'22)	—	• MLIR	○	•	—
Verif. Dialects [12] (PLDI'25)	—	• MLIR	—	•	—
Volta [16] (arXiv'25)	—	—	• GPU	•	• FP
This work	• Sail	• MLIR+LLVM	• RVV	•	• int

networks at the model level, not at the implementation level. No prior work formally verifies a quantized ML kernel implementation against an ISA specification.

Verified compilers: CompCert [19] provides an end-to-end verified C compiler in Coq but does not support RVV, MLIR, or quantized ML workloads. Vellvm [20] formalizes LLVM IR semantics in Coq without extending to assembly or ISA models.

Hardware verification: ISA-Formal verifies ARM processor RTL against the ASL specification using bounded model checking. The complementary riscv-formal project performs similar RTL-vs-spec checking for RISC-V cores using Sail models as the reference [13]. These efforts verify *hardware* correctness: that the processor implements the ISA specification. Our work verifies *software* correctness: that the kernel program computes the intended function per the ISA specification. Together, the two directions would provide full-stack assurance from compiler IR to silicon.

Symbolic execution scalability: Path explosion is a fundamental challenge in symbolic execution [10]. SIMD and vector instructions exacerbate the problem: each vector element can introduce independent branches, and masked operations create 2^{VL} execution paths. Isla [10] addresses path explosion for scalar instructions through careful SMT encoding but does not discuss vector-specific mitigations. Islaris [4] tames Isla trace complexity via Iris separation logic in Coq, an effective but manual approach unsuited to the repetitive per-element structure of vector computations. QVIP [18] verifies quantized neural networks but operates at the model level, avoiding ISA-level path explosion entirely. To our knowledge, no prior work has specifically addressed the scalability of symbolic execution for SIMD or vector ISA instructions. Our per-element AST decomposition and mask decomposition (Section III-G) are domain-specific techniques that exploit the regular, lane-parallel structure of vector operations to reduce solver complexity from minutes to under one second.

Summary: Our work is the first to span compiler IR to ISA formal specification for computational kernels. It is the first to use Isla traces for kernel verification (as opposed to systems code), the first to formally verify quantized ML kernel

implementations, and the first to perform kernel-level formal verification against the Sail RVV model. Additionally, the MLIR specification layer provides a path toward cross-ISA portability: the same `vector+arith` dialect spec can serve as the reference for verifying implementations on different ISAs, requiring only new Isla traces per target (Section V-I). The comparison in Table XIV shows that no prior work covers all five dimensions simultaneously.

VII. Conclusion

We presented a three-layer formal verification pipeline that proves equivalence from MLIR compiler IR through LLVM IR to ISA-level semantics derived from the Sail RISC-V formal model. Applied to thirteen kernels covering all ten llama.cpp quantization formats (Q4_0, Q4_1, Q5_0, Q5_1, Q8_0, Q2_K–Q6_K) plus vector addition, ReLU, and 1D convolution—all layers are verified as UNSAT for all possible inputs, completing in under 30 seconds total. We additionally proved VLEN portability: per-element formulas are identical across vector lengths.

Beyond the pipeline itself, we solved three scalability challenges inherent to vector ISA symbolic execution: monolithic 256-bit formulas (per-element AST decomposition, $>10\,\text{min} \rightarrow <1\,\text{s}$), masked instruction path explosion (2^{VL} paths $\rightarrow$ 3 independent lemmas), and multi-instruction composition ($O(n)$ via Z3 substitution). These techniques are not RISC-V-specific and apply to any SIMD ISA with a Sail model.

The pipeline is designed for reuse: adding a new kernel requires only MLIR and LLVM IR specification files ($\sim$50 lines total); porting to a new ISA requires replacing the Sail model and regenerating traces. The same MLIR specification serves as a cross-ISA golden reference, enabling transitive equivalence between backends verified against it.

The key insight is that quantized integer ML inference is a sweet spot for formal verification: the computation is pure bitvector arithmetic with no floating-point, no dynamic memory allocation, and no control-flow divergence within the vector datapath. This makes SMT-based verification both tractable and complete—a property that does not hold for floating-point ML workloads.

Future work: Three directions are particularly promising. First, closing the compiler codegen gap (L5): proving that LLVM's RVV backend correctly translates LLVM IR to the assembly sequence we verify. This is closely related to ARM-TV's approach [6], but using Sail rather than a hand-written lifter; our Isla traces could serve as the ISA-side specification for such a translation validator. Second, extending beyond integer datapaths to floating-point scale factors and memory address verification. Third, extending cross-ISA verification (demonstrated here for Wasm SIMD128) to ARM SVE and other SIMD ISAs with Sail models.

Artifact: All source code, Isla traces, and verification scripts (6,028 lines of Python + 2,012 lines of Rust) are available at: https://github.com/yiidtw/rvs

References

[1] G. Gerganov, "ggml: Tensor library for machine learning," https://github.com/ggerganov/ggml, 2024.

[2] RISC-V International, *RISC-V "V" Vector Extension, Version 1.0*, 2021, https://github.com/riscv/riscv-v-spec.

[3] N. P. Lopes, J. Lee, C.-K. Hur, Z. Liu, and J. Regehr, "Alive2: Bounded translation validation for LLVM," in *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation (PLDI)*, 2021, pp. 65–79.

[4] M. Sammler, A. Hammond, R. Lepigre, B. Campbell, J. Pichon-Pharabod, D. Dreyer, D. Garg, and P. Sewell, "Islaris: Verification of machine code against authoritative ISA semantics," in *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation (PLDI)*, 2022, pp. 825–840.

[5] S. Bang, S. Nam, I. Chun, H. Y. Jhoo, and J. Lee, "SMT-based translation validation for machine learning compiler," in *Proc. Computer Aided Verification (CAV)*. Springer, 2022, pp. 386–405.

[6] R. Berger, M. Briles, N. Bushehri, N. Coughlin, K. Lam, N. P. Lopes, S. Mada, T. Tirpankar, and J. Regehr, "Translation validation for LLVM's AArch64 backend," *Proc. ACM Program. Lang.*, vol. 9, no. OOPSLA, 2025.

[7] L. de Moura and N. Bjørner, "Z3: An efficient SMT solver," in *Proc. International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Springer, 2008, pp. 337–340.

[8] A. Armstrong, T. Bauereiss, B. Campbell, A. Reid, K. E. Gray, R. M. Norton, P. Mundkur, M. Sherwood, L. Maranget, and P. Sewell, "ISA semantics for ARMv8-A, RISC-V, and CHERI-MIPS," *Proc. ACM Program. Lang.*, vol. 3, no. POPL, pp. 71:1–71:31, 2019.

[9] RISC-V Foundation, "Sail RISC-V model," https://github.com/riscv/sail-riscv, 2024.

[10] A. Armstrong, B. Campbell, B. Simner, C. Sherlock, and P. Sewell, "Isla: Integrating full-scale ISA semantics and axiomatic concurrency models," in *Proc. Computer Aided Verification (CAV)*. Springer, 2021, pp. 303–316.

[11] C. Lattner, M. Amini, U. Bondhugula, A. Cohen, A. Davis, J. Pienaar, R. Riddle, T. Shpeisman, N. Vasilache, and O. Zinenko, "MLIR: Scaling compiler infrastructure for domain specific computation," in *Proc. IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2021, pp. 2–14.

[12] M. Fehr, Y. Fan, H. Pompougnac, J. Regehr, and T. Grosser, "First-class verification dialects for MLIR," in *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation (PLDI)*, 2025, pp. 1466–1490.

[13] C. Wolf, "riscv-formal: RISC-V formal verification framework," https://github.com/YosysHQ/riscv-formal, 2024.

[14] W3C WebAssembly Community Group, "WebAssembly SIMD proposal," https://github.com/WebAssembly/simd, 2024, fixed-width 128-bit SIMD extension.

[15] A. VanHattum, M. Pardeshi, C. Fallin, A. Sampson, and F. Brown, "Lightweight, modular verification for WebAssembly-to-native instruction selection," in *Proc. International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2024.

[16] K. Dubey, B. Driscoll, A. Wei, N. Kayal, R. Sharma, and A. Aiken, "Equivalence checking of ML GPU kernels," arXiv:2511.12638, 2025.

[17] S. Bhat, A. Keizer, C. Hughes, A. Goens, and T. Grosser, "Verifying peephole rewriting in SSA compiler IRs," in *Proc. International Conference on Interactive Theorem Proving (ITP)*, 2024, pp. 9:1–9:20.

[18] Y. Zhang, Z. Zhao, G. Chen, F. Song, M. Zhang, T. Chen, and J. Sun, "QVIP: An ILP-based formal verification approach for quantized neural networks," in *Proc. IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2022.

[19] X. Leroy, "A formally verified compiler back-end," *J. Automated Reasoning*, vol. 43, no. 4, pp. 363–446, 2009.

[20] J. Zhao, S. Nagarakatte, M. M. Martin, and S. Zdancewic, "Formalizing the LLVM intermediate representation for verified program transformations," in *Proc. ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2012, pp. 427–440.

APPENDIX

SUPPLEMENTARY MATERIAL

The following appendices provide hand-traced walk-throughs and reproducibility instructions. They are not part of the 9-page body and are included as supplementary material.

We trace a concrete example through all three layers to validate the pipeline's claims.

Setup: Two Q8_0 input vectors, 4 elements each (simplified from VL=16):

$$a = [2, -3, 1, -1] \quad \text{(int8)}$$
$$b = [4, 2, -5, 3] \quad \text{(int8)}$$

MLIR spec (Layer 1): Sign-extend to i16, multiply, reduce:

$$\begin{aligned} \text{prod} &= [2 \times 4,\ (-3) \times 2,\ 1 \times (-5),\ (-1) \times 3] \\ &= [8, -6, -5, -3] \quad \text{(int16)} \\ \text{result} &= 8 + (-6) + (-5) + (-3) = -6 \quad \text{(int32)} \end{aligned}$$

LLVM IR (Layer 1 check): Identical operations (`sext`, `mul`, `reduce.add`) $\Rightarrow$ same result: -6. ✓

RVV assembly (Layer 2):

1) `vwmul.vv v4,v1,v2`: widening multiply $8 \rightarrow 16$ bits. Isla trace sign-extends each 8-bit operand to 16 bits and multiplies: $\text{sext}_{16}(a_i) \times \text{sext}_{16}(b_i) = [8, -6, -5, -3]$.
2) `vwredsum.vs v6,v4,v6`: widening reduction $16 \rightarrow 32$ bits. Isla trace sums all elements with sign extension: $\text{sext}_{32}(8)+\text{sext}_{32}(-6)+\text{sext}_{32}(-5)+\text{sext}_{32}(-3) = -6$.

Result: -6. Matches MLIR/LLVM IR. ✓

Verification: Z3 asserts MLIR $\neq$ Isla, gets UNSAT $\Rightarrow$ equivalent for *all* inputs, not just this example.

Artifact package: Download the self-contained artifact from the GitHub release or Zenodo. Extract and `cd rvs-artifact/`. The package includes pre-generated Isla traces, all Python scripts, MLIR/LLVM IR specs, and the Isla binary. All commands in Appendices A–A can be executed directly.

Prerequisites: Python 3.10+, `z3-solver` (`pip install z3-solver`).

One-command verification:

```
cd rvs/
# Basic + asymmetric formats
python3 verify_three_layer.py # Q4_0
python3 verify_three_layer_q4_1.py # Q4_1
python3 verify_three_layer_q5.py # Q5_0
python3 verify_three_layer_q5_1.py # Q5_1
python3 verify_three_layer_q8.py # Q8_0
# K-quant formats
python3 verify_three_layer_q4_k.py # Q4_K
python3 verify_three_layer_q5_k.py # Q5_K
python3 verify_three_layer_q6_k.py # Q6_K
python3 verify_three_layer_q3_k.py # Q3_K
python3 verify_three_layer_q2_k.py # Q2_K
# Utility kernels
python3 verify_three_layer_conv1d.py # Conv1D
python3 verify_vecadd.py # VecAdd
python3 verify_vlen_portable.py # VLEN portable
python3 verify_q4_0_auto.py # per-instr
# Or via Rust CLI:
cd tool && cargo run --release -- ../kernels/*.toml
```

Expected output:

```
VERDICT: ALL THREE LAYERS VERIFIED
  Q4_0: 0.13s, Q8_0: 0.10s, per-instr: 20.6s
  All proofs: UNSAT
```

LOC breakdown:

Module	Lin
`isla_trace.py` (parser + composer)	94
`mlir_spec.py` (MLIR parser)	17
`llvm_ir_spec.py` (LLVM IR parser)	18
`verify_three_layer*.py` (13 kernels)	2,78
`verify_vlen_portable.py`	15
`verify_q4_0_auto.py` (per-instr)	21
`rvv_encode.py` (encoder)	11
`rvs.py` (declarative framework)	18
Total (Python)	**3,71**
`tool/` (Rust CLI)	2,01

We trace a concrete Q4_0 example through all 6 computational instructions to validate the pipeline and provide a hand-checkable reference. The kernel under verification is `ggml_vec_dot_q4_0_q8_0` from llama.cpp (commit `05728db`).

Setup: Consider a single packed Q4_0 byte `x_qs[0] = 0xA7` and two Q8_0 activation values `y_lo[0] = 3`, `y_hi[0] = -2` (both int8). The packed byte contains two 4-bit nibbles: low nibble $=$ `0xA7` & `0x0F` $= 7$, high nibble $=$ `0xA7` $\gg 4 = 10$.

Step-by-step trace (element 0 only):

1) **vand.vx v4,v1,a3** (nibble mask, `a3=0x0F`):
   `v4`$[0] =$ `0xA7` & `0x0F` $= 7$ (uint8)
2) **vsrl.vx v5,v1,a4** (nibble shift, `a4=4`):
   `v5`$[0] =$ `0xA7` $\gg 4 = 10$ (uint8)
3) **vsub.vx v6,v4,a5** (bias subtract, `a5=8`):
   `v6`$[0] = 7 - 8 = -1$ (int8, two's complement: `0xFF`)
4) **vsub.vx v7,v5,a5** (bias subtract):
   `v7`$[0] = 10 - 8 = 2$ (int8)
5) **vwmul.vv v8,v6,v2** (widening multiply, $8 \rightarrow 16$ bits):
   `v8`$[0] = \text{sext}_{16}(-1) \times \text{sext}_{16}(3) = -3$ (int16)
6) **vwmacc.vv v8,v7,v3** (widening MAC, accumulates into v8):
   `v8`$[0] = -3 + \text{sext}_{16}(2) \times \text{sext}_{16}(-2) = -3 + (-4) = -7$ (int16)

After processing all 16 elements, `vwredsum.vs` reduces: `sumi` $= \sum_{i=0}^{15} \text{sext}_{32}($`v8`$[i])$. For this single element: $\text{sext}_{32}(-7) = -7$.

Isla trace formula: The auto-parsed Isla formula for element 0 after composition is:

$$\text{sext}_{16}\big((\texttt{0xA7}\,\&\,\texttt{0x0F} - 8) \cdot 3\big) + \text{sext}_{16}\big((\texttt{0xA7} \gg 4 - 8) \cdot (-2)\big) = -3 + (-4)$$

This matches the hand calculation. ✓

Verification: Z3 asserts MLIR $\neq$ Isla, gets UNSAT $\Rightarrow$ equivalent for *all* inputs, not just this example. The full proof covers 2^{384} possible input combinations (16 packed bytes $\times$ 32 activation bytes).

Scope of this proof: Table XV summarizes what is and is not verified.

TABLE XV
VERIFICATION SCOPE FOR THE Q4_0 KERNEL.

Concern	Status	Note
Integer datapath	• Verified	All 2^{384} inputs
FP16 scale multiply	Not verified	Applied once per block
Memory loads	Axiomatized	Fresh symbolic values
Loop iteration	1 block only	Full loop = future
`vsetvli`	• Verified	All 2^{64} AVL values
Compiler codegen	Not verified	L5 gap

To demonstrate pipeline generality, we verify vector ReLU—a single-instruction kernel. Reproduce: `python3 experiments/appendix_d_relu.py`.

Kernel: Element-wise ReLU on 16 signed 8-bit integers: $\text{ReLU}(x)_i = \max(x_i, 0)$. Assembly: `vmax.vx v4, v1, x0` (SEW=8, VL=16).

Step 1: Specification:

```
x_i = Extract(i*8+7, i*8, vr1)
spec_i = If(x_i >= 0, x_i, BitVecVal(0, 8))
```

Step 2: Isla trace: `isla-footprint -A rv64v.ir -C riscv64v.toml -x -i "5742101e" -s --verbose` produces a 182-line trace. Inputs: `vr1`, `vr4`. Output: `vr4`.

Step 3–4: Decompose and verify: 16 independent 8-bit elements, all UNSAT in 0.11 s. Covers all 2^{128} inputs.

Higher-level specs: MLIR: `arith.maxsi`. LLVM IR: `icmp sgt + select`. Both close the three-layer chain.

Recipe: (1) Encode assembly. (2) Run Isla. (3) Write 5-line Z3 spec. (4) Auto-verify. $<$1 minute human effort, 0.1 s solver time.

We demonstrate bug *catching*, not just correctness confirmation. Reproduce: `python3 experiments/appendix_e_bugfind.py`.

Setup: Q4_0 extracts high nibble via `vsrl.vx v5,v1,a4` with a4=4. Bug: compiler emits shift-by-3.

Correct spec: With a4=4: 16/16 elements UNSAT.

Buggy spec: With a4=3 against the same correct Isla trace:

```
Element 0: SAT -- v1[0]=0x80, correct=0x08, buggy=0x10
Result: 16/16 SAT -- bug detected in ALL elements
```

Impact: Shift-by-3 extracts bits [7:3] (5-bit, 0–31) instead of [7:4] (4-bit, 0–15). After bias subtraction: range $[-8, 23]$ instead of $[-8, 7]$—silent precision corruption.

An ISA designer verifies a new instruction end-to-end in $<$1 s. Reproduce: `python3 experiments/appendix_f_new_insn.py`.

Instruction: `vor.vx v4,v1,a3` (bitwise OR with scalar). Encoding: `57c2162a`.

Trace and decomposition: 169-line trace. Per-element: $\texttt{trace}[i] = \texttt{Extract}(8i{+}7, 8i, \texttt{vr1}) | \texttt{Extract}(7, 0, \texttt{x13})$.

RVV spec: $\texttt{spec}[i] = \texttt{vs2}[i] \mid \texttt{rs1}[7{:}0]$.

Result: 16/16 UNSAT in 0.01 s. Covers all 2^{320} inputs. No manual SMT translation.