Automated Generation of RISC-V Extensions with Formal Correctness Guarantees

Elisavet Lydia Alvanaki* **Jiakun Wang*** **Eugenio Muscinelli** **Luca P. Carloni**
Department of Computer Science - Columbia University in the City of New York
ealvanaki@cs.columbia.edu

***Abstract*—Per-workload custom instructions can improve performance on low-power embedded RISC-V cores, but deploying them requires a full-stack flow: identifying a useful source expression, implementing RTL, integrating the instruction into the core, rewriting the software, and checking that behavior is preserved. LLMs can generate much of this stack, but unconstrained generation gives the agent authority over both the transformation and its justification. This is unsafe for custom instructions: a proposal can compile and pass tests while computing the wrong operation, using the wrong operands, or committing the wrong architectural result.**

We present Janus, an LLM-assisted framework that synthesizes custom instructions integrated into the Ibex RISC-V core while keeping correctness outside the agent. The agent proposes the operation, the RTL module that implements it, and the integration patch that wires it into the core. A deterministic *orchestrator* mechanically rewrites the workload and accepts a proposal only after a three-stage source-to-ISA verification chain succeeds: source-to-operation equivalence, operation-to-RTL equivalence, and ISA-level integration. No agent-authored artifact is trusted by assertion.

Across 13 embedded workloads, the framework accepts 7 single-cycle R/I-type custom instructions with end-to-end speedups from 2.56% to 13.65%. The rejected and unprofitable cases show why verification must be part of the synthesis path: plausible agent proposals can be semantically wrong, incorrectly integrated, or valid but too small to amortize invocation overhead. These results demonstrate a practical path for using LLMs to explore ISA specialization without making the agent part of the trusted correctness boundary.

I. Introduction

Custom instructions are an effective way to specialize embedded RISC-V processors [7]: replacing a frequently-executed arithmetic or bit-manipulation expression with a single instruction reduces dynamic instruction count and improves end-to-end latency without redesigning the rest of the core. In practice, however, per-workload custom instructions remain difficult to deploy, because each one must be selected from the program, assigned an encoding, implemented in RTL, integrated into the pipeline, invoked from software, and checked against the original workload behavior [6]. LLMs offer a promising path to automating this stack: they can search over source expressions, generate RTL, modify integration logic, and repair proposals from tool feedback [18]. However, using an LLM for these tasks creates a circular trust problem. In a naive deployment, the agent does all three jobs at once: it proposes the rewrite, implements the hardware, *and authors the reference the hardware is checked against*. The generated result is unsafe: a proposed instruction can compile and pass tests while computing the wrong operation, using the wrong operands, or committing the wrong architectural result.

Existing work does not close this gap. LLM-for-hardware research has improved RTL code quality [9], [11] but leaves the verification problem to the LLM. Deterministic ISA-extension flows such as CIDRE [14] avoid this issue by keeping the process fully rule-based, but they require an nML-based Synopsys ASIP Designer processor model, which limits portability to existing RISC-V cores outside that ecosystem. The regime that is left uncovered is the one this paper targets: an agent proposing custom instructions for an existing core, with no vendor processor model, where the trust gap must be closed explicitly. The central question is therefore:

> *Can LLMs automate the path from a workload source expression to a processor-integrated custom instruction without becoming part of the trusted correctness boundary?*

We answer the central question positively with Janus, a framework that separates proposal from acceptance. First, an agent authors a three-part *proposal* for one workload: an operation (the C expression the instruction should compute) with the corresponding workload source line (as expressed through an OpSpec struct), an RTL module that implements it, and an integration patch wiring the module into the processor pipeline. Then, a separate deterministic *orchestrator* rewrites the workload, runs a chain of equivalence checks, and accepts the proposal only if every check succeeds. Consequently, the agent sits outside the trust boundary: its proposals affect architectural state only through artifacts the chain has accepted.

The technical mechanism is a three-stage source-to-ISA chain. First, the orchestrator proves that the proposed operation is equivalent to the source expression selected for replacement. Second, it proves that the RTL module implements that operation. Third, it checks that the modified pipeline decodes, routes, and commits the instruction with the intended ISA-level behavior. A proposal may compile, pass the workload test, and improve cycle count; it is still rejected if any link in this chain fails. We target the low-end of the RISC-V design space: in-order, single-issue, bare-metal cores running embedded workloads without an operating system. As a representative core in this class we use Ibex [15], an open-source 32-bit RISC-V core. While Janus may be extended to

*These authors contributed equally to this work.

different custom instructions, we target the commonly used single-cycle R and I-type instructions. Janus integrates single-cycle, combinational R-type and I-type custom instructions into Ibex's decode and execute stage.

We evaluate Janus on 13 embedded workloads from four benchmark suites. 7 workloads produce accepted custom instructions with end-to-end speedups from 2.56% to 13.65%. The remaining workloads expose where the current scope is too narrow or too costly. Some proposals pass the chain but do not yield speedups: some opportunities require loop-level rather than local expression rewrites, and some accepted custom instructions do not have enough dynamic coverage to amortize invocation overhead.

Our paper makes three contributions:

- We present Janus, a framework that automates the path from a workload source expression to a custom instruction integrated into the Ibex RISC-V core, while keeping correctness decisions outside the agent.
- We introduce a three-stage source-to-ISA verification chain that checks the source rewrite, the RTL module, and the processor-level integration of each accepted instruction.
- We quantify where the current scope succeeds and fails, showing that custom instructions require both semantic validity and enough dynamic coverage to amortize invocation overhead.

II. The Janus Framework

A. Proposal and Trust Boundary

When the agent proposes a custom instruction, it produces a three-part *proposal*: an OpSpec that declares the *operation* (a C expression saying what the instruction should compute) and source lines, an *RTL module* M implementing that operation, an *integration patch* Δ_{Ibex} wiring the module into Ibex's decode stage and execute stage. The orchestrator rewrites the workload mechanically; the agent never edits the source code. The trust boundary is therefore: the orchestrator trusts the C compiler, the SMT solver Z3 [5], the model checker hw-cbmc [12], the RVFI specification [17], and the unmodified Ibex baseline [15]; it does not trust any artifact the agent authors. The OpSpec may misdescribe the source program, the RTL module may implement the wrong operation, and the integration patch may decode, route, or commit the instruction incorrectly. Each agent-authored artifact is treated as a claim to be checked, not a specification to be assumed correct.

B. The OpSpec

To facilitate orchestrator parsing, the agent proposes workload-specific optimizations in a structured block, the OpSpec. The OpSpec declares what the instruction computes and where in the workload it should be used. The grammar is fixed:

```
OP_SPEC ----- <module_name>
encoding:       r | i
operands:       a | a, b | a, imm
workload_expr:  <C expression>
target_line:    <verbatim source line>
target_subexpr: <optional substring>
END_OP_SPEC
```

The `workload_expr` field is a C expression W, in the workload's own variables, that specifies what the instruction computes. The `target_line` field gives a verbatim source line of the form `lhs = rhs;`, whose right-hand side `rhs` will be replaced; the optional `target_subexpr` narrows the rewrite to a sub-expression of `rhs`. We call the matched location the *splice site* ℓ, and write r_ℓ for the source expression at that site after macro expansion. To rewrite the workload, the orchestrator locates each line matching `target_line` (modulo whitespace) and, after Stage 1 of the chain proves $r_\ell \equiv W$, emits an inline-assembly invocation of the custom instruction in place of r_ℓ. Each workload has a *success sentinel*, a completion marker emitted only when the workload reaches its expected end state, which the orchestrator checks alongside the chain to confirm that the rewritten workload still terminates correctly.

C. Source-to-ISA Verification Chain

The chain proves three equivalences between r_ℓ, W, M, and $\text{ISA}(M)$, where $\text{ISA}(M)$ is the value committed to the destination register when the modified Ibex core executes the custom instruction. An accepted proposal must establish

$$r_\ell \;\equiv\; W \;\equiv\; M \;\equiv\; \text{ISA}(M).$$

Stage 1 (source-to-operation) discharges $\forall \vec{x}.\ r_\ell(\vec{x}) = W(\vec{x})$ by lowering both sides to quantifier-free bit-vector formulas and calling Z3 [5]. This stage ensures the source expression to be replaced is equivalent to the operation declared in OpSpec. **Stage 2 (operation-to-RTL)** wraps W as a C reference function and uses hw-cbmc [12] to prove M equivalent to it. The OpSpec grammar restricts M to a single-cycle combinational function, so the input space is finite (2^{32} for an R-type with one register operand, 2^{64} for two, and $2^{32} \times 2^{12} = 2^{44}$ for an I-type with one register and a sign-extended 12-bit immediate). Bounded model checking [4] at depth one exhaustively covers the legal operand domain; on the rare proposals where the LLM-generated RTL introduces latched logic, the orchestrator wraps the module in a single-clock harness and retries at depth two. The operand domain remains finite in either case, so Stage 2 is sound and complete for the class of operations the grammar admits. **Stage 3 (RTL-to-ISA)** *extracts a verified ISA-level check for the new instruction itself*: the framework instantiates a spec module (verifiable with any formal verification tool) from a fixed template, declaring the custom op's encoding constraints, source/destination register addresses, result $M(\texttt{rs1}, \texttt{rs2/imm})$, and PC advance. The proof certifies that this module is consistent with the integrated Ibex pipeline which includes the agent's RTL edits [17]. The output is a per-op formal specification that, on acceptance, sits alongside the 10 base RV32I R-type entries [16] as part of the trusted ISA. Any future RVFI-equipped CPU implementing the same opcode can be verified against it. The check covers

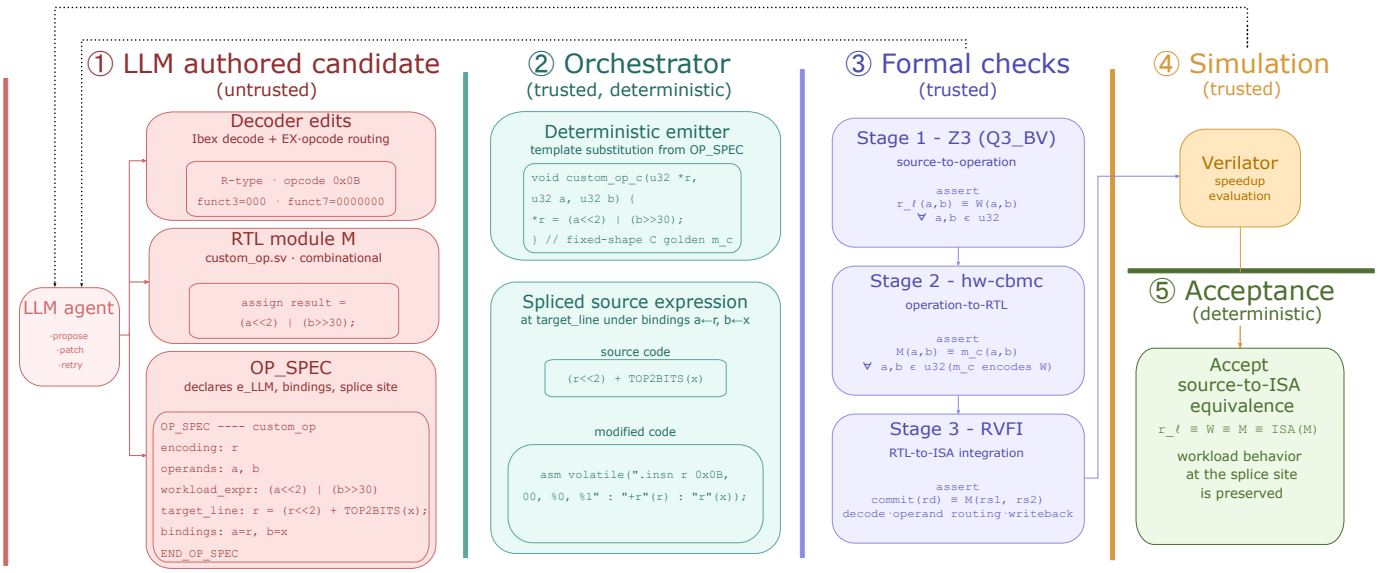

Fig. 1. Overview of the Janus framework.

integration correctness such as decode, operand routing, write-back, and PC advance that properties Stages 1–2 cannot see, because they prove the module is correct in isolation rather than correctly integrated. This stage also *verifies that the ISA remains correct with the added instruction*.

When any stage rejects a proposal, the orchestrator returns the counterexample to the agent in a structured form: for Stage 1, the Z3 model showing inputs where r_ℓ and W disagree; for Stage 2, the hw-cbmc trace showing inputs where M and W disagree; for Stage 3, the RVFI assertion failure. The agent may revise any of its artifacts and resubmit; we limit repair to 15 iterations per workload. Since the orchestrator rejects direct agent edits to the workload source, the verified splice sites are the only semantic changes introduced by the rewrite. Therefore, the Janus framework guarantees that, whenever the unmodified workload (the *baseline*) is itself correct, the rewritten workload is observationally equivalent to it at every splice site, conditional on the correctness assumption for the baseline workload.

D. Pipeline overview

Figure 1 shows an overview of the Janus framework. Each workload runs through a deterministic pipeline: the orchestrator invokes the agent with a structured prompt to generate the decode stage modification, RTL module and OpSpec, applies any accepted edits, mechanically splices the verified custom instruction into the workload, invokes the formal checks, rebuilds the modified Ibex simulator, and measures cycle counts under cycle-accurate Verilator simulation. Stages 1–3 of the verification chain run sequentially; failure at any stage returns a structured counterexample to the agent, who may revise any of its proposal artifacts and resubmit. Acceptance requires that: (i) the modified workload reaches its success sentinel, (ii) all three verification stages pass, and (iii) cycle count strictly improves over the baseline.

III. Evaluation

A. Experimental Setup

Consistent with the low-end embedded RISC-V scope set in Section II, we evaluate Janus on 13 bare-metal workloads drawn from bringup-bench [1], Embench-IoT [13], MiBench [8], and monocypher [2] — benchmark suites that target embedded cores rather than application processors. From these suites, we include all non-floating-point workloads that contain at least one R/I-type-shaped expression candidate; floating-point workloads are excluded because Ibex has no floating-point unit. All workloads are compiled for RV32IMC, ILP32 ABI, with GCC 10.2.0 and run on the lowRISC Ibex demo system [10] in a cycle-accurate Verilator [3] simulation. All tools run with a 300-second timeout. The agent is implemented with Gemini 3 Pro using the Vertex AI API.

The agent receives a system prompt describing the OpSpec grammar, the three verification stages, and the structured output format it must produce. Each iteration's user prompt contains the workload source, the current proposal (if any), and either the initial task description or the structured counterexample from the failed stage. Repair prompts include the Z3 model, hw-cbmc trace, or RVFI assertion failure verbatim. Full prompt templates will be released upon acceptance.

Each accepted custom instruction is invoked from C through inline assembly using the RISC-V `.insn` directive; we refer to the resulting wrapper of register moves and `asm` block as the instruction's *invocation sequence* and to its cycle cost as *invocation overhead*. Baseline and modified workloads use the same compiler flags, so measured differences come from the custom instruction and its invocation sequence rather than from compiler optimization differences. Each run uses the same agent, OpSpec grammar, and verification chain. A proposal is accepted only if the modified workload reaches its

TABLE I
ACCEPTED CUSTOM INSTRUCTIONS. SITES CORRESPONDS TO THE NUMBER OF SOURCE CODE EXPRESSIONS REPLACED.

Workload	Suite	Custom op	C expression replaced	Sites	Base cycles	Mod. cycles	Speedup
nettle_sha256	Embench-IoT [13]	SHA-256 σ/Σ dispatch	$\Sigma_0(a)$ and $\Sigma_1(e)$ in ROUND macro	2	4 780 781	4 128 281	**13.65%**
chacha20	monocypher [2]	ChaCha20 const rotate	$\mathrm{ROTL32}(x, n)$, $n \in \{16, 12, 8, 7\}$	4	806 458	765 498	**5.08%**
crc32	bringup-bench [1]	CRC-32 polynomial	$(\mathrm{crc} \gg 1) \oplus ((\mathrm{crc} \,\&\, 1)\,?\,\mathrm{POLY} : 0)$	1	18 761 698	17 779 682	**4.93%**
murmur-hash	bringup-bench [1]	Var. rotate	$(k \ll r_1) \mid (k \gg (32 - r_1))$	3	1 098 481	1 047 281	**4.66%**
mibench_bitcount	MiBench [8]	SWAR/Kernighan	$((x \,\&\, m_{\mathrm{hi}}) \gg k) + (x \,\&\, m_{\mathrm{lo}})$, $x \,\&\, (x-1)$	6	3 573 775	3 433 260	**3.93%**
mibench_isqrt	MiBench [8]	isqrt shift-pack	$(r \ll 2) + \mathrm{TOP2BITS}(x)$	1	14 467 396	13 943 108	**3.62%**
aes	bringup-bench [1]	AES GF(256) doubling	$(x \ll 1) \oplus ((x \,\&\, 0\mathrm{x}80)\,?\,0\mathrm{x}1\mathrm{b} : 0)$	1	7 889 681	7 688 081	**2.56%**

success sentinel, Stage 1 passes Z3 equivalence, Stage 2 passes hw-cbmc equivalence, and Stage 3 passes the RVFI check.

B. End-to-End Accepted Custom Instructions

Janus produces accepted, processor-integrated custom instructions for 7 of the 13 workloads. Table I reports the results. Speedups range from 2.56% to 13.65%, showing that the framework can find useful source expressions to specialize when they fit the current single-cycle combinational scope.

Where speedups come from: Table I illustrates the accepted instructions. Each row passes Stages 1–3 of the verification chain. We group the accepted custom instructions by the shape of the source expression they replace. The greatest gains occur when a compact source expression appears frequently enough to amortize the invocation sequence. texttttnettle_sha256 is the clearest case: a single I-type instruction packs all four SHA-256 σ and Σ functions into one opcode, with the immediate selecting which function. The smaller wins come from ARX-style (add-rotate-XOR) kernels such as `murmur-hash` and `chacha20`. Their custom rotate instructions remove a few instructions per splice site, but the surrounding adds, XORs, and invocation overhead remain.

Rejected and unprofitable proposals: Table II summarizes the workloads that did not produce accepted speedups. These cases define the current boundary of Janus, which is most effective when the target is a single-cycle stateless expression that appears frequently enough to amortize invocation overhead. The remaining workloads fall outside this boundary for four reasons: the proposed operation is too small to offset invocation overhead; the source expression contains an array index that the Stage 1 lowering cannot resolve; the opportunity is a loop-level transformation rather than a local expression replacement; or hw-cbmc cannot discharge Stage 2 within the configured timeout.

C. Why the Formal Chain Matters

The verification chain is not simply a final certificate for the accepted instructions in Table I. It also provides the feedback that drives the synthesis: when a proposal fails, the counterexample from Z3, hw-cbmc, or RVFI identifies the semantic mismatch that the agent must repair. During repair, the chain rejects proposals that compile and simulate, but break one of its three links. The case study (Appendix V-A) walks through one such proposal in detail: a dispatch-table custom instruction whose RTL and reference disagreed on inputs the workload's runtime input distribution never produces. A correctness-only test harness would have accepted it; hw-cbmc rejected it and surfaced a counterexample that drove the repair.

TABLE II
WORKLOADS WHERE JANUS DID NOT PRODUCE AN ACCEPTED SPEEDUP.

Workload	**Outcome**	**Cause**
`cipher`	0.00%	Sub-expression rewrite overhead dominates.
`mersenne`	0.01%	Array-indexed expression exceeds lowerer.
`bit-kernels`	0.00%	Hot expression is a loop recurrence.
`fft-int`	timeout	hw-cbmc timeout on signed multiply.
`md5sum`	0.00%	Argument assembly dominates local saving.
`mibench_sha`	−4.44%	Rewrite regresses under invocation overhead.

The negative cases in Table II show why correctness alone is insufficient. Some proposals pass every link in the chain, but do not improve the cycle count: invocation overhead exceeds the local instruction saving. The evaluation therefore requires both the verification chain and end-to-end cycle-accurate simulation. The chain establishes that a rewrite is valid; the simulation determines whether it is useful.

D. Limitations

The Janus framework's guarantees have explicit boundaries. The OPSPEC grammar admits only single-cycle, combinational R/I-type operations over 32-bit integers; this is what makes Stage 2 sound and complete, but it excludes the multi-cycle, stateful, multi-output, and floating-point operations that realistic accelerators (matrix multiply, DSP kernels, stateful crypto) require. Within that grammar, Stage 2's bit-blasting via hw-cbmc [12] discharges bitwise, shift, and add/XOR ops in seconds, but 32-bit multiply, divide, and modulo are adversarial for CDCL solvers and can exceed our timeout (as in fft-int); timeouts are conservative rejections, so soundness is unaffected, only practical reach. Since Stage 3's RVFI check is likewise bounded by the configured riscv-formal instruction horizon, so a custom instruction whose pipeline-state interaction manifests only beyond that horizon could pass Stage 3 and misbehave in longer execution. On the integration side, the orchestrator splices custom instructions inline, which introduces invocation overhead and blocks compiler optimizations across the splice; more extensive compiler toolchain integration is left to future work. Finally, the accepted workloads are crypto, hash, and compression benchmarks – regimes where bit-manipulation custom instructions are known to help

– so effectiveness on control-flow, memory, or floating-point-dominated workloads is not extensively tested. Added R/I-type logic is small relative to Ibex's execute stage, so we do not report area overhead separately.

IV. Conclusions

We presented Janus, a framework for LLM-driven generation of RISC-V custom instructions in which correctness decisions sit outside the agent. The agent proposes an OpSpec, an RTL module, and an integration patch; a deterministic orchestrator accepts the proposal only if a three-stage chain establishes $r_\ell \equiv W \equiv M \equiv \mathrm{ISA}(M)$ through Z3, hw-cbmc, and RVFI. Every accepted instruction is therefore backed by a machine-checkable proof of source-to-ISA equivalence rather than by test-suite agreement.

Across 13 embedded workloads, Janus produces 7 processor-integrated single-cycle R/I-type custom instructions with end-to-end speedups of 2.56% to 13.65% under cycle-accurate Verilator simulation. The remaining workloads characterize the boundary of the current scope: some proposals fail the chain, and some pass it but do not amortize invocation overhead. The `mibench_bitcount` case study shows the value of bit-precise checking: a dispatch-table mismatch that random differential testing would not distinguish was rejected by Stage 2 and repaired in subsequent iterations.

Future work includes extending Janus to support multi-cycle and stateful operations, integrating accepted instructions into the compiler toolchain to remove invocation overhead, and adding formal checks for more complex generated extensions.

Acknowledgments. This work was supported in part by the U.S. DOE DeCoDe Project No. 84245 at PNNL and in part by Amazon.com, Inc.

V. Appendix

A. Case Study: `mibench_bitcount`

We illustrate the verification chain on `mibench_bitcount`, the MiBench bit-counting benchmark. This workload exercises five popcount methods on the same input vector; one is a fixed five-line cascade of bitwise reductions:

```
i = ((i & 0xAAAAAAAAL) >> 1) + (i & 0x55555555L);
i = ((i & 0xCCCCCCCCL) >> 2) + (i & 0x33333333L);
i = ((i & 0xF0F0F0F0L) >> 4) + (i & 0x0F0F0F0FL);
i = ((i & 0xFF00FF00L) >> 8) + (i & 0x00FF00FFL);
i = ((i & 0xFFFF0000L) >> 16) + (i & 0x0000FFFFL);
```

Each line has the same shape: $((i \,\&\, M_j) \gg s_j) + (i \,\&\, \overline{M_j})$. The agent recognized that a single I-type instruction whose immediate selects the (M_j, s_j) pair could replace all five lines with five splice sites against one shared opcode. It emitted an OpSpec declaring a six-way ternary on `imm & 7`—five arms for the SWAR cascade, one for the Kernighan path used elsewhere—together with an RTL module implementing the same dispatch.

The bug. The Stage 2 reference returned `0` when `imm & 7 > 5`, the default arm of the agent's ternary cascade. The agent's RTL module fell back to returning `a`, the input register value, in the same case. Both choices are sensible in isolation and unreachable from any splice site the orchestrator would generate, since the agent only emits five distinct immediates. On the workload's actual inputs, the buggy RTL and the reference returned identical values. Random testing on 10^6 samples from the workload's input distribution would not have distinguished them.

What the chain caught. hw-cbmc's bit-precise check is exhaustive over the full 2^{32}-input port space, not the distribution the workload exercises. At Iteration 2, it returned a counterexample at `imm = 0x6` with `a = 0x12345678`: the RTL produced `0x12345678`, while the reference produced `0`. The orchestrator surfaced this in the next prompt; six iterations later, the agent aligned the RTL's default arm to return `0`, hw-cbmc discharged Stage 2, and Z3 discharged Stage 1 over the now-uniform default. The modified ELF reached the `libmin_success` sentinel after 3,433,260 cycles—a $3.93\,\%$ reduction over the 3,573,775-cycle baseline, achieved by replacing 30 dynamic SWAR instructions per hot-loop iteration with five custom-instruction invocations.

References

[1] "Bringup-Bench: A benchmark suite for bringing up new CPUs, accelerators, compilers and operating systems," https://github.com/toddmaustin/bringup-bench.

[2] "Monocypher: An easy to use, easy to deploy crypto library," https://github.com/LoupVaillant/Monocypher.

[3] "Verilator," https://verilator.org, 2024.

[4] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu, "Symbolic model checking without BDDs," in *Proc. of the Intl Conf. on Tools and Algorithms for Construction and Analysis of Systems*, 1999, p. 193–207.

[5] L. De Moura and N. Bjørner, "Z3: an efficient SMT solver," in *Proc. of the Theory and Practice of Software, Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, 2008, p. 337–340.

[6] C. Galuzzi and K. Bertels, "The instruction-set extension problem: A survey," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 4, 2011.

[7] Y. Gao, W. Qian, and E. Cui, "RISC-V ISA extension toolchain supports: A survey," in *Proc. of the Intl. Conf. on Computing, Networks and Internet of Things*, 2023, p. 924–929.

[8] M. R. Guthaus *et al.*, "MiBench: A free, commercially representative embedded benchmark suite," in *Proc. of the Annual IEEE Intl. Workshop on Workload Characterization (WWC-4)*, 2001.

[9] M. Liu *et al.*, "ChipNeMo: domain-adapted LLMs for chip design," 2024. [Online]. Available: https://arxiv.org/abs/2311.00176

[10] lowRISC, "Ibex Demo System," https://github.com/lowRISC/ibex-demo-system, 2024.

[11] Y. Lu *et al.*, "RTLLM: An open-source benchmark for design RTL generation with large language model," in *Proc. of the Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2024, pp. 722–727.

[12] R. Mukherjee, M. Purandare, R. Polig, and D. Kroening, "Formal techniques for effective co-verification of hardware/software co-designs," in *Proc. of the Design Automation Conference (DAC)*, 2017.

[13] D. Patterson *et al.*, "Embench: A free benchmark suite for embedded computing from an academic-industry cooperative," Presented at RISC-V Workshop Zürich. https://github.com/embench/embench-iot, 2019.

[14] E. Rezunov, N. Zurstrassen, L. M. Reimann, and R. Leupers, "Automatic microarchitecture-aware custom instruction design for RISC-V processors," in *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, 2025.

[15] P. D. Schiavone *et al.*, "Slow and steady wins the race? a comparison of ultra-low-power RISC-V cores for Internet-of-Things applications," in *Intl. Symp. on Power and Timing Modeling, Optimization and Simulation (PATMOS)*, 2017, pp. 1–8.

[16] A. Waterman *et al.*, "The RISC-V Instruction set manual, Volume I: User-level ISA, version 2.1," Technical report UCB/EECS-2014-54. [Online]. Available: https://apps.dtic.mil/sti/pdfs/ADA605735.pdf

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

[18] Y. Zhao *et al.*, "Mage: A multi-agent engine for automated RTL code generation," in *Proc. of the Design Automation Conference (DAC)*, 2025.