

REIST Division: An Implementation-Oriented Framing of Centered Remainder Arithmetic for Modular Addition

Rudolf Stepan
Independent Researcher
ORCID: 0009-0004-2842-2579

Abstract—Centered remainder representations express signed cyclic corrections using established residue classes. This note defines the integer interval and its bounded modular-addition update, including odd moduli, midpoint conventions and overflow prerequisites. It retains historical ARMv8-A and x86-64 measurements against compiler-remainder baselines. These measurements have not been repeated after correcting boundary handling and do not establish a benefit over bounded non-negative modular addition, which also avoids division. A new controlled evaluation is required before making a representation-specific performance claim. The method concerns addition-dominated updates and does not replace general division or multiplication-oriented Montgomery and Barrett reduction.

Index Terms—REIST, modular addition, centered remainder, balanced arithmetic, performance, ARMv8-A, x86-64.

CORRECTION AND SCOPE (24 OCTOBER 2026)

The benchmark tables below are retained as historical observations, not as a fresh evaluation of the corrected formulas. Earlier versions mishandled some odd-modulus boundaries. Reproducible performance claims require corrected kernels, exact revision identifiers, and a bounded non-negative baseline. For integer $B > 0$, let $l = -\lfloor B/2 \rfloor$ and $h = \lfloor (B-1)/2 \rfloor$. If $a, b \in [l, h]$, compute $s = a + b$ and apply $s \leftarrow s - B$ if $s > h$, or $s \leftarrow s + B$ if $s < l$. One correction suffices. Arbitrary inputs need normalization first. The sum, correction, and shifted construction must be representable without signed overflow. For non-negative $a, b < B$, the alternative $s = a + b$ followed by subtraction of B when $s \geq B$ also avoids division. Either bounded rule can use a runtime modulus. Source-level conditionals do not guarantee branchless machine code or constant-time execution.

I. INTRODUCTION

Integer division fixes a remainder convention. In C and C++, a nonzero remainder has the sign of the dividend. An explicit remainder operator may introduce a division or a compiler-generated replacement; non-negative residue representations themselves do not require division for bounded modular addition. Balanced remainder representations are classical, but they are frequently discussed as a mathematical convention rather than as a concrete implementation pattern for fast modular addition.

This work presents REIST Division as a compact framing: represent modular values in a centered interval and

update them via a signed correction rule. We benchmark the resulting arithmetic against the compiler-provided modulo operator to isolate the architectural effect of eliminating division-based remainder handling in modular addition. The goal is *not* to outperform state-of-the-art modular multiplication schemes; Montgomery and Barrett reduction remain the standard for multiplication-heavy workloads. REIST targets addition-dominated workloads and SIMD-friendly correction rules.

II. BACKGROUND AND NOTATION

Given integers T and $B > 0$, classical division yields unique q, r such that

$$T = qB + r, \quad 0 \leq r < B. \quad (1)$$

A centered remainder convention instead chooses r in a symmetric interval around zero. Such conventions appear in number theory and in practice (e.g., centered coefficient ranges in NTT pipelines). The contribution here is not a new algebraic object, but an implementation-oriented framing and evaluation of a remainder-as-correction rule that maps well to modern CPU instruction mixes.

III. REIST DIVISION

A. Definition

REIST uses a centered, half-open remainder interval:

$$T = qB + r, \quad q \in \mathbb{Z}, \quad -\frac{B}{2} \leq r < \frac{B}{2}. \quad (2)$$

An exact-arithmetic construction, with non-negative mathematical modulo, is

$$r = ((T + \lfloor B/2 \rfloor) \bmod B) - \lfloor B/2 \rfloor. \quad (3)$$

This guarantees $r \in [-B/2, B/2)$ and resolves the even-modulus midpoint by excluding $+B/2$.

B. Relation to Rounding

For every integer $B > 0$, the REIST quotient satisfies

$$q = \left\lfloor \frac{T}{B} + \frac{1}{2} \right\rfloor, \quad (4)$$

i.e., nearest-integer rounding with ties toward positive infinity. REIST differs in that the remainder is treated as a persistent

signed correction variable that can be used in iterative update loops, enabling branchless, SIMD-friendly correction patterns for modular addition.

C. Why this helps for modular addition

If intermediate values remain in a centered range, a modular addition update can often be expressed using additions and conditional corrections instead of a full remainder operation. This replaces expensive division-based remainder handling by a small sequence of ALU operations (and, in vector form, lane-wise corrections). The approach is therefore expected to help when modular addition dominates and modulus-dependent division costs are on the critical path.

IV. EXPERIMENTAL SETUP

We report results from an open-source benchmark suite that implements a classical modulo baseline and REIST-style update rules side-by-side. Measurements were taken on two distinct CPU architectures:

- ARMv8-A (aarch64) with SIMD (NEON) variants where applicable.
- x86-64 (Intel Core i9-14900K) with SIMD variants where applicable.

Benchmarks include: (i) modular-add counters, (ii) polynomial modular addition (coefficient-wise $(a_i + b_i) \bmod q$), (iii) pure remainder computation $x \bmod m$, (iv) ARX-only workloads (e.g., ChaCha-like), and (v) hash-mixing workloads that combine multiplication and modulo operations.

V. RESULTS

Tables I and II summarize speedup ranges (Classic / REIST runtime) across tested moduli.

TABLE I
ARMv8-A SPEEDUP RANGES ACROSS TESTED MODULI.

Workload	O3	SIMD
Modular addition	2.1–2.3×	4.1–4.4×
Polynomial modular addition	4.3–4.5×	4.3–4.5×
Remainder only	$\approx 1.0\times$	$\approx 1.0\times$
Hash mixing	0.79–0.80×	0.79–0.83×

TABLE II
X86-64 SPEEDUP RANGES ACROSS TESTED MODULI.

Workload	O3	SIMD
Modular addition	7.9–8.9×	7.1–8.7×
Polynomial modular addition	12.4–12.8×	13.1–17.0×
Remainder only	$\approx 1.0\times$	$\approx 0.9\times$
Hash mixing	0.80–0.81×	0.73–0.75×

A. Interpretation

The results show a consistent pattern across both architectures:

- **Strong gains** when modular addition (or polynomial modular addition) dominates. This is consistent with

eliminating division-based remainder handling and enabling SIMD lane-wise corrections.

- **No benefit** for pure remainder computation, as expected, since REIST is not intended to replace division for computing $x \bmod m$ from scratch.
- **Slowdowns** for hash-mixing workloads in the tested form, indicating REIST is a targeted optimization rather than a universal speedup.

VI. DISCUSSION AND LIMITATIONS

REIST should be viewed as a practical framing for centered remainder arithmetic with an implementation strategy optimized for modular addition. It is complementary to established multiplication-centric reductions:

- **Not a replacement for Montgomery/Barrett:** those remain appropriate for modular multiplication-heavy code.
- **Baseline choice:** this paper benchmarks against the compiler-provided modulo operator to isolate the effect of replacing division-based remainder handling. Head-to-head comparisons with hand-tuned Montgomery/Barrett implementations are out of scope.
- **Security/side-channels:** correction rules can be implemented branchlessly; constant-time properties depend on the concrete implementation and are not claimed here.

VII. CONCLUSION

Centered representatives provide a precise language for signed cyclic corrections. Their bounded update rule is exact under the stated interval and overflow assumptions. The historical benchmark tables require renewed measurement with corrected kernels and a bounded non-negative baseline. They do not establish an intrinsic speed advantage of the centered convention.

REFERENCES

- [1] D. E. Knuth, *The Art of Computer Programming, Vol. 2: Seminumerical Algorithms*, 3rd ed. Reading, MA, USA: Addison-Wesley, 1997.
- [2] I. Niven, H. S. Zuckerman, and H. L. Montgomery, *An Introduction to the Theory of Numbers*, 5th ed. New York, NY, USA: Wiley, 1991.
- [3] K. J. Åström and R. M. Murray, *Feedback Systems: An Introduction for Scientists and Engineers*. Princeton, NJ, USA: Princeton University Press, 2012.
- [4] Repository (benchmark sources): <https://github.com/rudolfstefan/reist-crypto-bench>