REIST-Division

Eine implementierungsorientierte Formulierung zentrierter Restarithmetik:
Theorie, Architekturanalyse und empirische Evaluation auf ARMv8-A, x86-64 und FPGA

Rudolf Stepan
Unabhängiger Forscher, Wien · ORCID: 0009-0004-2842-2579

2026 · DOI: 10.5281/zenodo.21206471

Zusammenfassung

Zentrierte Restdarstellungen sind in der Zahlentheorie klassisch; der Beitrag dieser Arbeit ist ihre implementierungsorientierte Formulierung. Die REIST-Division (*Remainder-Extended Inversion and Subtraction Technique*) behandelt den Rest als vorzeichenbehafteten Korrekturterm in einem zentrierten Intervall und führt ihn als persistente Zustandsvariable durch iterative Aktualisierungsschleifen: Additive modulare Zustände werden ohne Division im Schleifenkörper durch höchstens eine $\pm B$-Korrektur reduziert; die Korrektur ist verzweigungsfrei formulierbar und SIMD-vektorisierbar. Die Arbeit entwickelt die formalen Eigenschaften der zentrierten Division, grenzt sie von Balanced Modulo, Barrett- und Montgomery-Reduktion sowie Compiler-Optimierungen ab und begründet die Gewinne mikroarchitektonisch. Die Evaluation auf ARMv8-A (Apple M2 Pro) und x86-64 (Intel Core i9-14900K) misst für modulare Additionszähler 2,1–4,4× (ARM) bzw. 7,9–8,9× (x86-64), für polynomielle modulare Addition bis zu 17,0× (AVX2); Workloads ohne persistenten modularen Zustand profitieren erwartungsgemäß nicht. Eine FPGA-Implementierung (Gowin GW2A-18) benötigt einen Takt pro Aktualisierung bei 161,8 MHz und 101 Logikzellen, gegenüber vier Takten, 8,1 MHz und 1276 Zellen eines herstellergenerierten Divisions-IP. REIST ist damit ein gezieltes arithmetisches Primitiv für additionsdominierte modulare Rechenkerne und ergänzt multiplikationszentrierte Reduktionsverfahren.

Schlagwörter: ganzzahlige Division, zentrierter Rest, balancierte Arithmetik, modulare Addition, SIMD, branchless, gitterbasierte Kryptographie, ARMv8-A, x86-64.

Inhaltsverzeichnis

1 Einleitung 4
1.1 Beitrag und Einordnung . . . 4
1.2 Aufbau der Arbeit . . . 5

2 Die klassische Division und ihre Asymmetrie 5

3 Die REIST-Division 6
3.1 Definition . . . 6
3.2 Zusammenhang mit der Rundung . . . 6
3.3 Formale Eigenschaften . . . 7
3.4 Algebraische Interpretation . . . 8
3.5 Beispiele . . . 8

4 Abgrenzung zu bestehenden Verfahren 9
4.1 Balanced Modulo . . . 9
4.2 Barrett- und Montgomery-Reduktion . . . 9
4.3 Verhältnis zur Compiler-Optimierung . . . 9

5 Implementierung 10
5.1 Skalare Korrekturregel . . . 10
5.2 Verzweigungsfreie Korrektur . . . 11
5.3 SIMD-Vektorisierung . . . 11

6 Anwendungsdomänen als Modellierungsperspektive 11

7 Experimenteller Aufbau 12
7.1 Benchmark-Suite . . . 12
7.2 Systeme . . . 13
7.3 Übersetzungsstufen . . . 13
7.4 Workloads . . . 13

8 Ergebnisse 14
8.1 Modularer Additionszähler . . . 14
8.2 Polynomielle modulare Addition . . . 15
8.3 Negativ- und Neutralkontrollen . . . 16
8.4 Multicore- und Skalierungsdiagnostik (Apple M2 Pro) . . . 16

9 Mikroarchitektonische Analyse 16
9.1 Latenzen der beteiligten Operationen . . . 16
9.2 Abhängigkeitsketten und ILP . . . 17
9.3 Verzweigungsverhalten . . . 17
9.4 Assembler-Inspektion . . . 17

10 Hardware-Evaluation: FPGA-Implementierung 18
10.1 Der REIST-Rechenpfad in Hardware . . . 18
10.2 Messaufbau . . . 19
10.3 Ergebnisse . . . 19
10.4 Der Latenz-Kompromiss des Divisions-IP . . . 21
10.5 Einordnung . . . 21

11 Limitationen 21

12 Mögliche Erweiterungen **22**

13 Schlussfolgerung **23**

1 Einleitung

Die ganzzahlige Division legt eine Restkonvention fest. In der Programmierpraxis wird meist ein nicht-negativer Rest im Intervall $[0, B)$ erwartet; für Aktualisierungen der Form $(a + b) \bmod m$ liegt damit häufig eine Division — oder ein vom Compiler erzeugter Divisionsersatz — auf dem kritischen Pfad. Balancierte Restdarstellungen sind mathematisch klassisch [2, 3], werden jedoch meist als Konvention diskutiert und selten als konkretes Implementierungsmuster für schnelle modulare Addition behandelt.

Diese Arbeit stellt die REIST-Division als kompakte, implementierungsorientierte Formulierung vor: Modulare Werte werden in einem zentrierten Intervall dargestellt und über eine vorzeichenbehaftete Korrekturregel aktualisiert. Der Rest wird dabei nicht als passives Residuum verstanden, sondern als expliziter Korrekturterm, der Richtung und Betrag einer Abweichung kodiert und als Zustandsvariable durch iterative Schleifen geführt wird. Diese Sichtweise liefert zugleich eine natürliche Modellierungssprache für Fehlerrückkopplung und Phasenausrichtung in Logistik, Scheduling, digitaler Signalverarbeitung, Regelungstechnik und hardwarenahen Korrekturschleifen.

REIST ist kein allgemeiner Divisionsalgorithmus, sondern eine Divisionsformulierung, deren praktischer Nutzen in der wiederholten Reduktion additiver modularer Zustände liegt — im Folgenden *zustandsbehaftete zentrierte modulare Akkumulation* genannt. Wo kein persistenter zentrierter Zustand existiert, ist entsprechend kein Gewinn zu erwarten; die Kontrollmessungen in Abschnitt 8.3 bestätigen das.

1.1 Beitrag und Einordnung

Zentrierte Restsysteme sind seit langem bekannt; der Beitrag dieser Arbeit liegt daher nicht in einem neuen algebraischen Objekt, sondern in fünf Punkten:

1. einer geschlossenen formalen Darstellung der zentrierten Division mit Beweisen der zentralen Eigenschaften (Abschnitt 3);
2. einer systematischen Abgrenzung gegenüber Balanced Modulo, Barrett- und Montgomery-Reduktion sowie gegenüber dem, was moderne Compiler für den Restoperator bereits leisten (Abschnitt 4);
3. einer mikroarchitektonischen Analyse, die erklärt, warum die zentrierte Darstellung auf modernen CPUs schneller ist: Wegfall des vorzeichenabhängigen Korrekturpfads, verzweigungsfreie Maskenkorrektur, SIMD-Vektorisierbarkeit (Abschnitte 5 und 9);
4. einer reproduzierbaren empirischen Evaluation auf zwei Architekturen und drei Übersetzungsstufen anhand einer quelloffenen Benchmark-Suite [13] (Abschnitte 7 und 8);
5. einer Hardware-Evaluation auf einem FPGA, die die REIST-Korrektur gegen ein herstellergeneriertes Divisions-IP in Zyklen, Fläche und Taktfrequenz vermisst [14] (Abschnitt 10).

Viele Beispiele stammen aus der gitterbasierten Kryptographie und der modularen Arithmetik; REIST ist jedoch keine kryptographiespezifische Konstruktion und insbesondere kein kryptographisches Primitiv. Ihr Kernbereich ist die Ganzzahlarithmetik mit vorzeichenbehafteten Resten und die Modellierung zyklischer, rückgekoppelter Prozesse, in denen „zu viel“ und „zu wenig“ symmetrische Korrekturzustände sind. Kryptographische Workloads dienen primär als Belastungstest und als ergiebige Quelle modulo-intensiver Operationen.

In hochoptimierten kryptographischen Bibliotheken wird die modulare *Multiplikation* üblicherweise durch Montgomery- [4] oder Barrett-Reduktion [5] beschleunigt. Die vorliegenden Benchmarks konkurrieren nicht mit diesen Verfahren. Der Vergleich erfolgt gegen den vom

Compiler bereitgestellten Modulo-Operator, um den architektonischen Effekt des Ersatzes divisionsbasierter Restbehandlung durch eine vorzeichenbehaftete Korrekturregel zu isolieren; eine Überlegenheit gegenüber modernen modularen Reduktionsverfahren wird nicht behauptet.

1.2 Aufbau der Arbeit

Abschnitt 2 analysiert die Asymmetrie der klassischen Division. Abschnitt 3 definiert die REIST-Division und beweist ihre formalen Eigenschaften. Abschnitt 4 ordnet REIST in die Landschaft bestehender Verfahren ein. Abschnitt 5 beschreibt die Implementierung; Abschnitt 6 ordnet Anwendungsdomänen jenseits der Hardware als Modellierungsperspektive ein. Die Abschnitte 7 und 8 dokumentieren den experimentellen Aufbau und die Ergebnisse auf CPUs, Abschnitt 9 liefert die mikroarchitektonische Erklärung. Abschnitt 10 ergänzt die Hardware-Evaluation auf einem FPGA. Abschnitt 11 diskutiert Grenzen, Abschnitt 12 zukünftige Arbeiten; Abschnitt 13 fasst zusammen.

2 Die klassische Division und ihre Asymmetrie

Für ganze Zahlen T und $B > 0$ liefert die klassische (euklidische) Division [1] eindeutige ganze Zahlen q und r mit

$$T = qB + r, \qquad 0 \leq r < B. \tag{1}$$

Der Quotient ist $q = \lfloor T/B \rfloor$; der Rest liegt stets im halboffenen Intervall $[0, B)$.

Diese Konvention ist für die elementare Zahlentheorie zweckmäßig, bildet jedoch alle Abweichungen einseitig nicht-negativ ab (Abbildung 1, oben). In zyklischen und rückgekoppelten Systemen ist das suboptimal: Eine kleine negative Korrektur — ein „Zurückspringen" um wenige Einheiten — wäre häufig die effizienteste Handlung, wird durch das Intervall $[0, B)$ jedoch als große positive Korrektur dargestellt. Wer auf einem Zifferblatt von 11 Uhr nach 12 Uhr will, geht eine Stunde vor, nicht elf zurück; die klassische Restkonvention kennt aber nur eine Richtung.

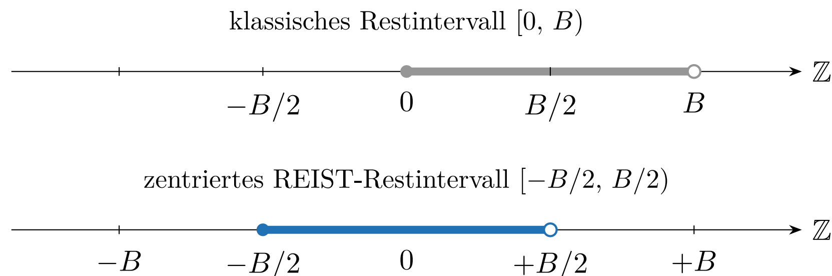

Abbildung 1: Restintervalle im Vergleich. Oben: Die klassische Division bildet alle Abweichungen einseitig in $[0, B)$ ab; die Null liegt am Rand. Unten: REIST zentriert das Intervall um null; positive und negative Korrekturen sind gleichberechtigt (ausgefüllter Punkt: enthalten, offener Punkt: ausgeschlossen).

Neben der Modellierungsasymmetrie hat die Konvention einen konkreten Preis auf der Maschinenebene. Dabei sind drei Ebenen zu trennen: Mathematisch wird häufig der euklidische Rest in $[0, B)$ verwendet; C/C++ definiert den vorzeichenbehafteten `%`-Operator dagegen über die *abgeschnittene* Division, deren Rest das Vorzeichen des Dividenden trägt; soll daraus ein nicht-negativer (oder zentrierter) Rest erzeugt werden, entsteht auf der Maschinenebene zusätzlicher Korrekturaufwand. Selbst wenn der Compiler die Division durch eine bekannte Konstante in eine Multiplikations-Shift-Sequenz übersetzt [6], verbleibt ein vorzeichenabhängiger Korrekturpfad (typisch eine Sign-Mask-Berechnung `a >> 31`) in der Abhängigkeitskette. Abschnitt 9 quantifiziert diese Kosten.

3 Die REIST-Division

3.1 Definition

Definition 3.1 (REIST-Division)**.** Seien $T \in \mathbb{Z}$ und $B \in \mathbb{Z}$, $B > 0$. Die *REIST-Division* von T durch B ist das eindeutige Paar $(q, r) \in \mathbb{Z}^2$ mit

$$T = qB + r, \qquad -\frac{B}{2} \le r < \frac{B}{2}. \tag{2}$$

r heißt *REIST-Rest* (zentrierter Rest), q *REIST-Quotient.*

REIST verwendet also ein zentriertes, halboffenes Restintervall $[-B/2,\ B/2)$ (Abbildung 1, unten). Eine praktische Konstruktion aus dem nicht-negativen Rest ist

$$r = \Big((T + \lfloor B/2 \rfloor) \bmod B\Big) - \lfloor B/2 \rfloor, \tag{3}$$

wobei mod den nicht-negativen Rest bezeichnet. Für gerades B liefert (3) exakt das Intervall $[-B/2,\ B/2)$; der Wert $+B/2$ tritt nie auf, wodurch der Mittelpunktsfall deterministisch aufgelöst ist. Für ungerades B ergibt sich das symmetrische Intervall $[-(B-1)/2,\ (B-1)/2]$, und die Darstellung ist stets eindeutig, da $|r| = B/2$ nicht auftreten kann.

Bemerkung 3.2 (Spiegelvariante)*.* In der Literatur und in Implementierungen findet sich ebenso die gespiegelte Konvention $(-B/2,\ B/2]$, bei der Mittelpunktsfälle auf $+B/2$ statt $-B/2$ abgebildet werden; auch die Benchmark-Suite [13] verwendet intern diese Variante. Für ungerades B stimmen beide Konventionen überein; für gerades B unterscheiden sie sich nur in der Behandlung des Falls $|r| = B/2$. Alle Aussagen dieser Arbeit — insbesondere sämtliche Leistungsmessungen — gelten für beide Varianten gleichermaßen. Kanonisch verwendet diese Arbeit $[-B/2,\ B/2)$, konsistent mit (3).

Die Interpretation des Restes als Korrekturterm lautet:

- $r > 0$: das System liegt vor (Überschuss),
- $r < 0$: das System liegt zurück (Defizit),
- $r = 0$: exakte Ausrichtung.

3.2 Zusammenhang mit der Rundung

Satz 3.3 (Rundungsdarstellung des Quotienten)**.** *Für $T \in \mathbb{Z}$, $B > 0$ ist der REIST-Quotient aus Definition 3.1 gegeben durch*

$$q = \left\lfloor \frac{T}{B} + \frac{1}{2} \right\rfloor. \tag{4}$$

Beweis. Sei $q = \lfloor T/B + 1/2 \rfloor$. Nach Definition der Gaußklammer gilt $q \le T/B + 1/2 < q + 1$, also $-1/2 \le T/B - q < 1/2$. Multiplikation mit $B > 0$ liefert $-B/2 \le T - qB < B/2$, d. h. $r = T - qB$ erfüllt die Intervallbedingung in (2). Da das Intervall $[-B/2, B/2)$ genau B aufeinanderfolgende ganze Zahlen enthält und damit aus jeder Restklasse modulo B genau einen Vertreter, ist das Paar (q, r) eindeutig. □

Der REIST-Quotient ist also die auf die nächste ganze Zahl gerundete rationale Zahl T/B, wobei Mittelpunktsfälle (nur bei geradem B möglich) aufgerundet werden. Für ungerades B ist (4) exakt die eindeutige Rundung zur nächsten ganzen Zahl. Während die klassische Division den Quotienten *abschneidet* bzw. *abrundet*, *rundet* REIST — die gesamte Struktur der zentrierten Restklasse ist eine unmittelbare Folge dieser anderen Quotientenwahl.

3.3 Formale Eigenschaften

Satz 3.4 (Eindeutigkeit)**.** *Für gegebene* $T \in \mathbb{Z}$ *und* $B > 0$ *existiert genau ein Paar* (q, r) *mit* $T = qB + r$ *und* $r \in [-B/2, B/2)$*.*

Beweis. Existenz und Eindeutigkeit folgen aus dem Beweis von Satz 3.3: Das Intervall enthält aus jeder Restklasse modulo B genau einen Vertreter. □

Satz 3.5 (Betragsminimalität des Restes)**.** *Die Quotientenwahl* (4) *minimiert den Betrag des Restes: Für jede alternative Wahl* $q' \in \mathbb{Z}$ *gilt*

$$|T - qB| \leq |T - q'B|. \tag{5}$$

Beweis. Die Funktion $q' \mapsto |T - q'B| = B \cdot |T/B - q'|$ wird minimiert durch die ganze Zahl q', die T/B am nächsten liegt — das ist gerade die Rundung (4). (Im Mittelpunktsfall $T/B = k + 1/2$ sind beide Nachbarn gleich weit entfernt; REIST wählt deterministisch $q = k + 1$.) □

Die klassische Division besitzt diese Eigenschaft nicht: Für $T = 17$, $B = 10$ liefert sie $r = 7$, während der betragsminimale Rest $r = -3$ ist.

Satz 3.6 (Additive Konsistenz)**.** *Seien* $T_1 = q_1B + r_1$ *und* $T_2 = q_2B + r_2$ *REIST-Zerlegungen. Dann gilt*

$$T_1 + T_2 = (q_1 + q_2)B + (r_1 + r_2), \qquad r_1 + r_2 \in [-B, B), \tag{6}$$

und der REIST-Rest von $T_1 + T_2$ *entsteht aus* $r_1 + r_2$ *durch höchstens eine Korrektur um* $\pm B$*.*

Beweis. Aus $r_1, r_2 \in [-B/2, B/2)$ folgt $r_1 + r_2 \in [-B, B)$. Liegt die Summe bereits in $[-B/2, B/2)$, ist nichts zu tun. Liegt sie in $[B/2, B)$, führt eine Subtraktion von B in das Intervall $[-B/2, 0)$; liegt sie in $[-B, -B/2)$, führt eine Addition von B in $[0, B/2)$. In beiden Fällen genügt genau eine Korrektur. □

Aus Satz 3.6 folgt die Eigenschaft, auf der die Implementierung dieser Arbeit beruht; sie sei als eigenständige Aussage festgehalten:

Korollar 3.7 (Implementierungsinvariante)**.** *Seien der Zustand* r *und die Schrittweite* x *zentrierte Vertreter,* $r, x \in [-B/2, B/2)$*. Dann gilt* $r + x \in [-B, B)$*, und höchstens eine Korrektur um* $\pm B$ *führt die Summe in das Intervall* $[-B/2, B/2)$ *zurück. Die Invariante* $r \in [-B/2, B/2)$ *bleibt damit über beliebig viele additive Aktualisierungen erhalten, ohne dass im Schleifenkörper eine Division oder Restoperation ausgeführt wird.*

Beweis. Unmittelbar aus Satz 3.6; die Erhaltung über beliebig viele Schritte folgt durch Induktion, da die korrigierte Summe wieder ein zentrierter Vertreter ist. □

Korollar 3.7 verlangt lediglich, dass B während der Akkumulationsphase fest ist und dass die Eingaben zentriert vorliegen — nicht aber, dass B zur Übersetzungszeit bekannt wäre: Der Modul kann zur Laufzeit gewählt werden, ohne dass im Schleifenkörper eine Division entsteht. Diese Unterscheidung wird in Abschnitt 4.3 wichtig.

Satz 3.8 (Fehlerterm und beste Approximation)**.** *Mit* $e = r/B$ *gilt* $|e| \leq 1/2$*. Der REIST-Quotient ist die beste ganzzahlige Approximation von* T/B*:*

$$\left| \frac{T}{B} - q \right| \;\leq\; \left| \frac{T}{B} - q' \right| \quad \textit{für alle } q' \in \mathbb{Z}. \tag{7}$$

Beweis. Unmittelbar aus Satz 3.5 nach Division durch B. □

Bemerkung 3.9 (Statistik der Reste). Unter der Modellannahme, dass die Reste (näherungsweise) gleichverteilt über die B möglichen Werte des Restintervalls sind, gilt für den REIST-Rest

$$\mathbb{E}[r] \approx 0, \qquad \mathrm{Var}(r) = \frac{B^2-1}{12} \approx \frac{B^2}{12},$$

während das klassische Restintervall $[0, B)$ denselben Streuterm, aber einen systematischen Bias $\mathbb{E}[r] \approx B/2$ aufweist. Genauer: Für ungerades B gilt exakt $\mathbb{E}[r] = 0$; für gerades B erzeugt die halboffene Tie-Break-Konvention einen exakten Offset von $-1/2$ (bzw. $+1/2$ in der Spiegelvariante aus Bemerkung 3.2), der relativ zu B vernachlässigbar ist. In iterativen Akkumulationen gleichen sich positive und negative REIST-Reste daher statistisch aus; die zentrierte Darstellung übernimmt Erwartungswert und Varianz der klassischen Rundungsarithmetik. Abbildung 2 illustriert den Effekt.

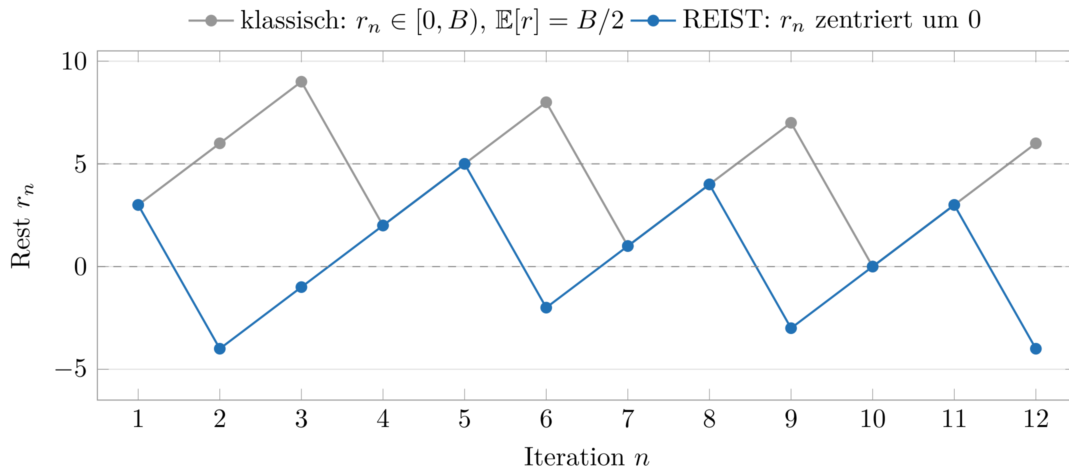

Abbildung 2: Restfolge des Zählers $x_{n+1} = x_n + 3$ modulo $B = 10$. Die klassische Restfolge (grau) verläuft vollständig im positiven Bereich und schwankt um den Bias $B/2 = 5$ (gestrichelt); die REIST-Restfolge (blau) schwankt um null. In rückgekoppelten Systemen bedeutet der Bias eine permanente einseitige Verschiebung, die sich über Iterationen akkumulieren kann.

3.4 Algebraische Interpretation

Als Vertretersystem der Restklassen $\mathbb{Z}/B\mathbb{Z}$ ist das zentrierte Intervall dem klassischen vollständig gleichwertig: Die Abbildung $r \mapsto r \bmod B$ ist eine Bijektion zwischen $[-B/2, B/2) \cap \mathbb{Z}$ und $[0, B) \cap \mathbb{Z}$, die die Ringstruktur respektiert. REIST definiert also keinen neuen Ring, sondern wählt das *betragsminimale Vertretersystem* (*least absolute residues* [2]) und macht diese Wahl zur Grundlage der Implementierung. Der Mehrwert liegt nicht in der Algebra, sondern in zwei Konsequenzen der Wahl:

1. **Numerisch:** Vertreter mit minimalem Betrag halten Zwischenwerte klein und symmetrisch um null — relevant für Fehlerfortpflanzung in RNS-/CRT-Systemen und für Drift in Akkumulationen (Bemerkung 3.9).

2. **Architektonisch:** Die Korrektur in das zentrierte Intervall ist eine symmetrische $\pm B$-Operation, die sich verzweigungsfrei mit Vergleichen und Masken formulieren lässt (Abschnitt 5).

3.5 Beispiele

Beispiel 3.10 (Betragsminimierung)**.** Sei $T = 17$, $B = 10$. Klassisch: $17 = 1 \cdot 10 + 7$, also $r = 7$. REIST: $17 = 2 \cdot 10 - 3$, also $r = -3$ — der betragskleinste Vertreter der Restklasse.

Beispiel 3.11 (Übereinstimmung und Abweichung)**.** Sei $B = 12$. Für $T = 28$ gilt klassisch wie REIST $r = 4$ (da $4 < 6$). Für $T = 34$ wählt REIST $34 = 3 \cdot 12 - 2$, also $r = -2$, während klassisch $r = 10$ resultiert.

Beispiel 3.12 (Mittelpunktsfall bei geradem Modul)**.** Sei $T = 5$, $B = 10$. Beide Kandidaten $r = 5$ ($q = 0$) und $r = -5$ ($q = 1$) haben Betrag $B/2$. Die Konvention $[-B/2, B/2)$ wählt deterministisch $r = -5$, $q = 1$; die Spiegelvariante aus Bemerkung 3.2 wählte $r = +5$, $q = 0$.

4 Abgrenzung zu bestehenden Verfahren

4.1 Balanced Modulo

Zentrierte Restsysteme sind in der Zahlentheorie etabliert [2, 3] und werden in NTT-Pipelines praktisch eingesetzt, etwa als zentrierte Koeffizientenbereiche in gitterbasierten Verfahren [7, 8]. *Balanced Modulo* bezeichnet üblicherweise die nachträgliche Transformation des klassischen Restes in das zentrierte Intervall — eine Umetikettierung des Ergebnisses. REIST erzeugt dasselbe Vertretersystem, unterscheidet sich aber in der Rolle, die der Rest im Programm spielt: Er wird nicht pro Operation neu berechnet und transformiert, sondern als *persistente Zustandsvariable* im zentrierten Intervall gehalten und ausschließlich über $\pm B$-Korrekturen aktualisiert. Erst diese Invariante macht die Restoperation im Schleifenkörper überflüssig und ermöglicht die in Abschnitt 8 gemessenen Gewinne.

4.2 Barrett- und Montgomery-Reduktion

Barrett- [5] und Montgomery-Reduktion [4] sind die Standardverfahren zur Beschleunigung der modularen *Multiplikation*: Sie ersetzen die Division nach einem Produkt durch Multiplikationen und Shifts bzw. durch eine Darstellung im Montgomery-Raum. REIST adressiert ein anderes, komplementäres Problem — die *additionsdominierte* Aktualisierung, bei der gar kein Produkt reduziert werden muss, sondern eine Summe im Zielintervall gehalten wird. In einer NTT-Pipeline können beide koexistieren: Montgomery/Barrett für die Butterfly-Multiplikationen, zentrierte $\pm B$-Korrekturen für die Additions- und Subtraktionsschritte. Tabelle 1 fasst die Einordnung zusammen.

Tabelle 1: Einordnung der REIST-Division gegenüber bestehenden Verfahren.

Verfahren	Restintervall	Kernoperationen	Branchless	Hauptanwendung
REIST	$[-B/2, B/2)$	ADD, CMP, MASK	ja	modulare Addition, Zähler
Klassisches Modulo	$[0, B)$	DIV bzw. Ersatz	teilweise	allgemeine Arithmetik
Balanced Modulo	zentriert	mod + Transf.	teilweise	symmetrische Restklassen
Barrett	$[0, B)$	MUL, SHR	ja	modulare Multiplikation
Montgomery	$[0, B)$	MUL, ADD	ja	modulare Multiplikation
RNS/CRT	je Kanal	Rekonstruktion	nein	Parallelmoduli-Systeme

4.3 Verhältnis zur Compiler-Optimierung

Ein naheliegender Einwand lautet, moderne Compiler optimierten `x % konst` ohnehin. Das trifft zu, grenzt den Beitrag von REIST aber nur genauer ein. Compiler ersetzen die Division durch eine bekannte Konstante automatisch durch Multiplikations-Shift-Sequenzen (*magic multiply*, [6]) und erzeugen für den vorzeichenbehafteten Restoperator maskenbasierte, weitgehend verzweigungsfreie Korrekturen. Ein typisches Muster (x86-64, GCC/Clang, `-O3`) für `a % 2452345`:

```
rax = (a * 1836446189) >> 52   ; magic multiply + shift
edx = a >> 31                  ; Sign-Mask (Vorzeichenpfad)
q   = rax - edx                ; Quotient (trunkiert)
r   = a - q * 2452345          ; Rest gemaess C-Semantik
```

Listing 1: Vom Compiler erzeugtes Muster für `a % konst` (schematisch).

REIST ersetzt diese Optimierung nicht, sondern setzt auf ihr auf: In der zentrierten Darstellung entfällt bei Additionsaktualisierungen die Restberechnung vollständig (Satz 3.6) — und mit ihr der vorzeichenabhängige Korrekturpfad (`a » 31`), den die in C definierte Restsemantik erzwingt. Das verkürzt die Abhängigkeitskette, senkt den Registerdruck und verbessert die Ausnutzung von Instruktionsparallelität (ILP) auf superskalaren Kernen. Die Assembler-Inspektion der Benchmark-Suite (Abschnitt 9.4) bestätigt beide Seiten: Die klassischen Varianten enthalten Magic-Multiply- *und* Sign-Mask-Muster, die REIST-Varianten nur Additions-, Vergleichs- und Maskenoperationen.

Ein Punkt wird dabei leicht verwechselt: Die Übersetzungszeit-Konstanz des Divisors betrifft die *Referenzseite* des Vergleichs. Nur für ein zur Übersetzungszeit bekanntes B kann der Compiler den Restoperator in eine Magic-Multiply-Sequenz übersetzen; bei dynamischem B fällt der klassische Pfad auf die Divisionsinstruktion zurück. Die REIST-Korrektur selbst enthält dagegen *keine* Division im Schleifenkörper: Sie benötigt lediglich, dass B während der Akkumulationsphase bekannt und stabil ist — B kann also auch erst zur Laufzeit gewählt werden, solange die Invariante des zentrierten Zustands erhalten bleibt (Korollar 3.7). Für dynamische, aber phasenweise stabile Moduli ist der Abstand zum klassischen Pfad damit sogar größer als in den hier gemessenen Benchmarks mit konstantem Divisor (vgl. Abschnitt 11).

5 Implementierung

5.1 Skalare Korrekturregel

Verbleiben Zwischenwerte im zentrierten Intervall, lässt sich eine modulare Additionsaktualisierung ohne Restoperation ausdrücken — Korollar 3.7 garantiert, dass eine einzige $\pm B$-Korrektur pro Addition genügt und die Invariante dauerhaft erhalten bleibt. Listing 2 zeigt den Zählerkern aus der Benchmark-Suite [13]:

```
int64_t reist_sym(int64_t B, int64_t N, int64_t step) {
    int64_t halfB = B / 2;
    int64_t r = 0;
    for (int64_t i = 0; i < N; ++i) {
        r += step;
        if (r > halfB)         r -= B;
        else if (r <= -halfB) r += B;
    }
    return r;
}
```

Listing 2: Skalarer REIST-Zähler (aus `bench_modadd_suite.cpp`). Der Zustand r verbleibt dauerhaft im zentrierten Intervall; pro Schritt fällt höchstens eine $\pm B$-Korrektur an.

Die Referenzvariante berechnet stattdessen `r = (r + step) % B` — pro Iteration eine vollständige Restoperation. Die Korrekturbedingungen in Listing 2 (`r > halfB` bzw. `r <= -halfB`) halten den Zustand in der Spiegelvariante $(-\lfloor B/2 \rfloor, \lfloor B/2 \rfloor]$ aus Bemerkung 3.2; für Korrektheit und Leistung ist das der kanonischen Konvention gleichwertig. Auch die bedingte Form in Listing 2 ist für den Compiler gut vorhersagbar bzw. in bedingte Move-Instruktionen (`CMOV`, `CSEL`) übersetzbar; die vollständig verzweigungsfreie Form folgt in Listing 3.

5.2 Verzweigungsfreie Korrektur

Für vektorisierbare Schleifen wird die Korrektur ohne Kontrollfluss formuliert. Listing 3 zeigt den polynomiellen Additionskern (koeffizientenweise $z_i = a_i + b_i \bmod q$, einen zentralen additiven Schritt gitterbasierter Verfahren [7, 8]):

```
void reist_poly_add(const int32_t* a, const int32_t* b,
                    int32_t* out, int32_t n, int32_t q) {
    const int32_t halfq  = q / 2;
    const int32_t neg_hq = -halfq;
    for (int32_t i = 0; i < n; ++i) {
        int32_t r  = a[i] + b[i];
        int32_t gt  = (r >  halfq);   // 0/1-Flag
        int32_t leq = (r <= neg_hq);  // 0/1-Flag
        r -= gt  * q;                 // branchless
        r += leq * q;                 // branchless
        out[i] = r;
    }
}
```

Listing 3: Verzweigungsfreie polynomielle REIST-Addition (aus `bench_poly_mod.cpp`). Die Vergleiche liefern 0/1-Flags; die Korrektur erfolgt arithmetisch ohne Sprünge.

Äquivalent ist die Maskenform `r -= B & mask`, bei der `mask` aus dem Vergleich als Bitmaske (0 oder -1) gewonnen wird. Beide Formen enthalten keine datenabhängigen Sprünge und keine Division; sie bestehen ausschließlich aus Operationen, die auf allen SIMD-Lanes verfügbar sind.

5.3 SIMD-Vektorisierung

Da die Korrektur nur ADD, CMP und Maskierung verwendet, überträgt sie sich unmittelbar auf Vektorregister. Listing 4 zeigt den NEON-Kern (vier 32-Bit-Lanes):

```
int32x4_t r   = vdupq_n_s32(0);
int32x4_t st  = vdupq_n_s32(step);
int32x4_t hB  = vdupq_n_s32(halfB);
int32x4_t nhB = vdupq_n_s32(-halfB);
int32x4_t Bv  = vdupq_n_s32(B);
for (int64_t i = 0; i < N/4; ++i) {
    r = vaddq_s32(r, st);                // 4 Lanes: r += step
    uint32x4_t mgt = vcgtq_s32(r, hB);  // Maske: r >  B/2
    uint32x4_t mle = vcleq_s32(r, nhB); // Maske: r <= -B/2
    int32x4_t cg = vandq_s32(Bv, vreinterpretq_s32_u32(mgt));
    int32x4_t cl = vandq_s32(Bv, vreinterpretq_s32_u32(mle));
    r = vsubq_s32(r, cg);                // r -= B & maske
    r = vaddq_s32(r, cl);                // r += B & maske
}
```

Listing 4: NEON-Variante des REIST-Zählers: lane-weise Addition und maskenbasierte $\pm B$-Korrektur auf vier Werten gleichzeitig (gekürzt, aus `bench_modadd_suite_neon.cpp`).

Gängige SIMD-Erweiterungen (ARM NEON, x86 AVX2) stellen *keine* ganzzahlige Vektordivision bereit; eine modulo-basierte Schleife kann daher nicht direkt vektorisiert werden, sondern fällt auf skalare Restberechnungen oder aufwendige Multiplikationssequenzen zurück. Die REIST-Korrektur skaliert dagegen mit der Vektorbreite (Abbildung 3).

6 Anwendungsdomänen als Modellierungsperspektive

Der messbare Kern dieser Arbeit liegt in der zentrierten Restarithmetik (Abschnitt 3), der additiven Zustandsinvariante (Korollar 3.7), ihrer verzweigungsfreien SIMD-Implementierung

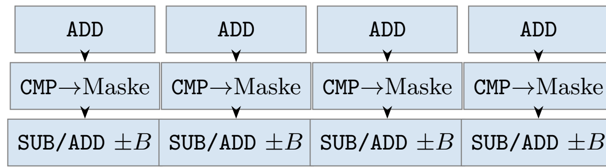

Abbildung 3: Struktureller Unterschied der Rechenpfade. Oben: Die REIST-Korrektur besteht aus lane-fähigen Operationen und verarbeitet vier (NEON) bzw. acht (AVX2) Werte parallel. Unten: Die klassische Restberechnung ist eine serielle Abhängigkeitskette mit vorzeichenabhängigem Korrekturpfad und steht auf gängigen SIMD-Einheiten nicht als Vektorinstruktion zur Verfügung.

(Abschnitt 5) und dem FPGA-Rechenpfad (Abschnitt 10). Der vorliegende Abschnitt gehört nicht zu diesem Kern: Er belegt keine Messergebnisse, sondern skizziert, wo sich die Interpretation des Restes als vorzeichenbehafteter Korrekturterm als Modellierungssicht anbietet. Alle genannten Domänen teilen dasselbe Muster: Ein zyklischer oder rückgekoppelter Prozess weicht von einem Sollraster ab, und die effizienteste Korrektur kann in *beide* Richtungen zeigen.

- **Lieferketten und Logistik:** Positiver Rest $\widehat{=}$ Überbestand, negativer $\widehat{=}$ Unterbestand; Sicherheitsbestände werden zu symmetrischen Puffern um den Sollwert, statt Defizite als große positive „Nachbestellreste“ zu kodieren.
- **Zyklische Systeme und Scheduling:** Negativer Rest kodiert Verzögerung, positiver Verfrühung; die betragskleinste Korrektur (Satz 3.5) minimiert die Drift (Abbildung 2).
- **Digitale Signalverarbeitung:** Phasenfehler in Phasenregelschleifen sind inhärent zyklisch und vorzeichenbehaftet [12]; das zentrierte Intervall ist direkt als Phasenfenster $[-\pi, \pi)$ interpretierbar, die Korrektur wählt automatisch den kürzeren Weg um den Einheitskreis.
- **Regelungstechnik und Robotik:** Rückgekoppelte Systeme arbeiten mit vorzeichenbehafteten Fehlern [11]; für zyklische Größen (Winkelencoder) ist die betragsminimale Korrektur wörtlich der kürzeste physische Weg.
- **Prädiktive Modellierung:** Ein zentriert mitgeführter Fehlerkorrekturterm verhindert einseitige Bias-Akkumulation über viele Iterationen (Bemerkung 3.9).
- **Ressourcenverteilung:** Ein negativer Rest erfasst Unterallokation explizit als Zustand; Über- und Unterlast werden direkt vergleichbare Größen.

7 Experimenteller Aufbau

7.1 Benchmark-Suite

Alle Messungen stammen aus der quelloffenen Benchmark-Suite `reist-crypto-bench` [13], die klassische Modulo-Referenzen und REIST-Varianten nebeneinander implementiert. Die Suite erzeugt zeitgestempelte Rohprotokolle, CSV-Dateien und eine Assembler-Inspektion der

übersetzten Kerne; sämtliche hier berichteten Zahlen sind den generierten Berichten vom 17. Dezember 2025 entnommen und reproduzierbar.

7.2 Systeme

Tabelle 2: Testsysteme.

	ARM-System	x86-System
Prozessor	Apple M2 Pro	Intel Core i9-14900K
Architektur	ARMv8-A (aarch64)	x86-64
SIMD	NEON/ASIMD (128 Bit)	AVX2 (256 Bit)
Speicher	16 GB	64 GB
Betriebssystem	macOS	Windows
Compiler	clang++	GCC/Clang (MinGW)

7.3 Übersetzungsstufen

Jeder Benchmark wird in drei Stufen übersetzt und gemessen:

- **O0** (`-O0 -fno-tree-vectorize`): unoptimierte Referenz. Sie zeigt die *algorithmische* Struktur ohne Compiler-Unterstützung und dient als Kontrolle, welcher Anteil der Gewinne der Formulierung selbst und welcher der Optimierung zuzuschreiben ist.
- **O3** (`-O3 -march=native -mtune=native`): produktionsübliche skalare Optimierung einschließlich automatischer Vektorisierung, wo der Compiler sie findet.
- **SIMD** (`-O3` + explizite Intrinsics): handvektorisierte Varianten (NEON bzw. AVX2), wo anwendbar.

7.4 Workloads

1. **Modularer Additionszähler** (`bench_modadd_suite`): $r \leftarrow (r + s) \bmod B$ über $N = 5 \cdot 10^7$ Iterationen; Moduli 257, 997, 10 007, 10^6+3, 10^7+19, 10^9+7 (Nicht-Zweierpotenzen, konstant zur Übersetzungszeit).
2. **Polynomielle modulare Addition** (`bench_poly_mod`): koeffizientenweise $z_i = (a_i + b_i) \bmod q$ über Polynome mit 1024 Koeffizienten und 50 000 Wiederholungen; Moduli 10^6+3 bis 10^9+7. Dies ist ein zentraler additiver Kern gitterbasierter Verfahren (NTRU, Kyber, Dilithium) — neben Multiplikation, Reduktion und Datenbewegung, die in vollständigen NTT-Pipelines ebenfalls kostenbestimmend sind; die Arithmetikkosten dieser Schemata auf ARM-Plattformen stehen auch im Zentrum systematischer Benchmarks der NIST-Standardisierungskandidaten [22].
3. **Reine Restberechnung** (`bench_modular`): $r = T \bmod B$ für zufällige 64-Bit-Werte — als Negativkontrolle, denn hier existiert kein Zustand, der im Intervall gehalten werden könnte.
4. **ARX-Workloads** (`bench_chacha_stream`): ChaCha20-artige Erzeugung des Keystreams [9] (nur ADD–ROTATE–XOR, keine Modulo-Operationen) — als zweite Negativkontrolle.
5. **Hash-Mixing** (`bench_hashmix`): Mischfunktionen mit multiplikativer Diffusion und Restoperationen — als Beispiel einer Workload-Klasse, in der die zentrierte Darstellung strukturell *nicht* passt.

Angegeben werden Speedups als Quotient $t_{\text{klassisch}}/t_{\text{REIST}}$; Werte über 1 bedeuten einen Vorteil für REIST.

8 Ergebnisse

8.1 Modularer Additionszähler

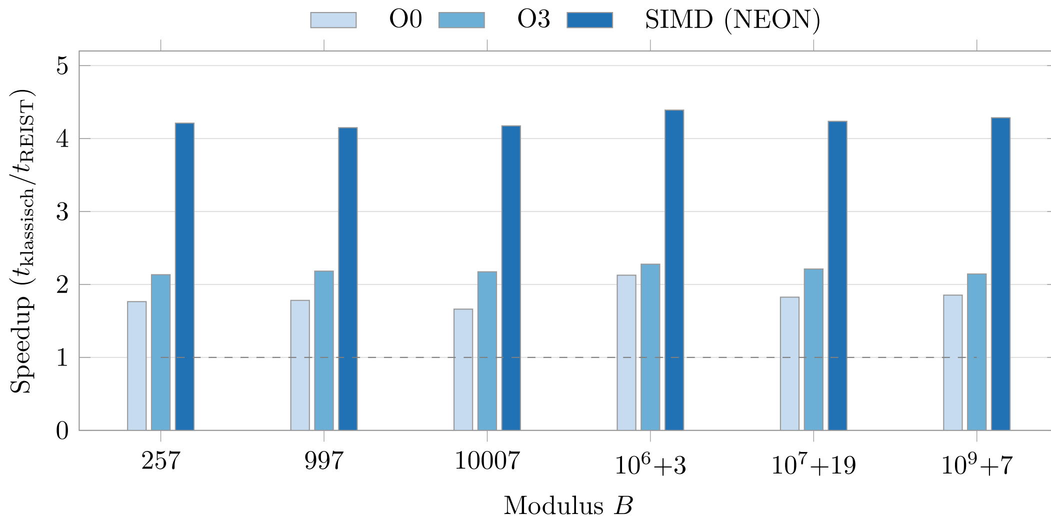

Abbildung 4: Modularer Additionszähler auf ARMv8-A (Apple M2 Pro): Speedup von REIST gegenüber klassischem Modulo je Modulus und Übersetzungsstufe. Die gestrichelte Linie markiert Gleichstand (1,0×). Die Gewinne sind über alle Moduli stabil; die NEON-Variante skaliert nahezu mit der Lane-Zahl.

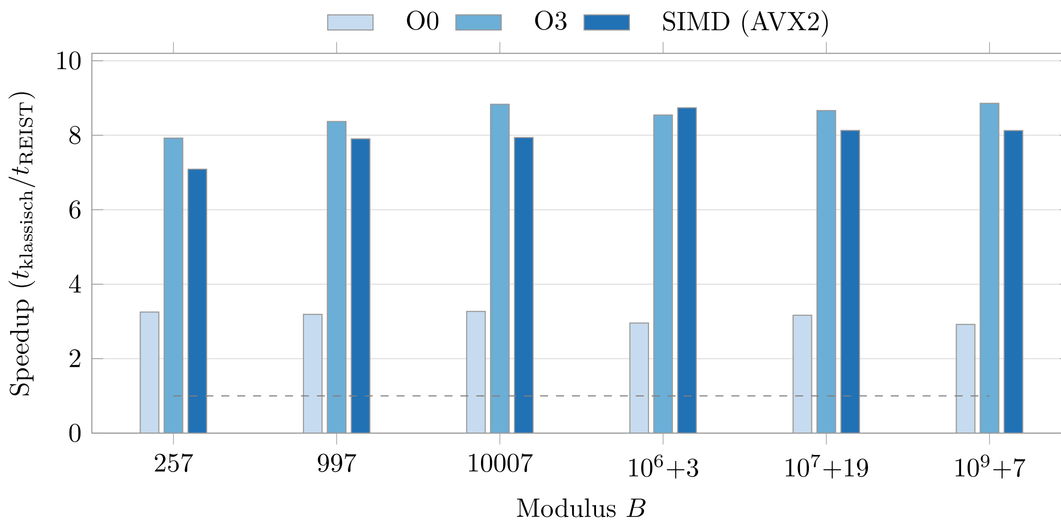

Abbildung 5: Modularer Additionszähler auf x86-64 (Intel Core i9-14900K): Speedup je Modulus und Übersetzungsstufe. Bereits die skalare O3-Variante erreicht 7,9–8,9×; die explizite AVX2-Variante liegt gleichauf, da die serielle Zählerabhängigkeit die Vektorisierung hier begrenzt.

Auf ARM (Abbildung 4) erreicht REIST 1,7–2,1× bereits ohne jede Compiler-Optimierung (O0) — der Gewinn entsteht also aus der Formulierung selbst, nicht aus der Optimierung. Mit O3 steigen die Werte auf 2,1–2,3×, mit expliziter NEON-Vektorisierung auf 4,1–4,4×. Auf x86-64 (Abbildung 5) sind die Effekte größer: 2,9–3,3× (O0) und 7,9–8,9× (O3). Die Gewinne sind über drei Größenordnungen des Modulus stabil — konsistent damit, dass weder

die Magic-Multiply-Sequenz noch die $\pm B$-Korrektur nennenswert vom Zahlenwert des Modulus abhängt.

8.2 Polynomielle modulare Addition

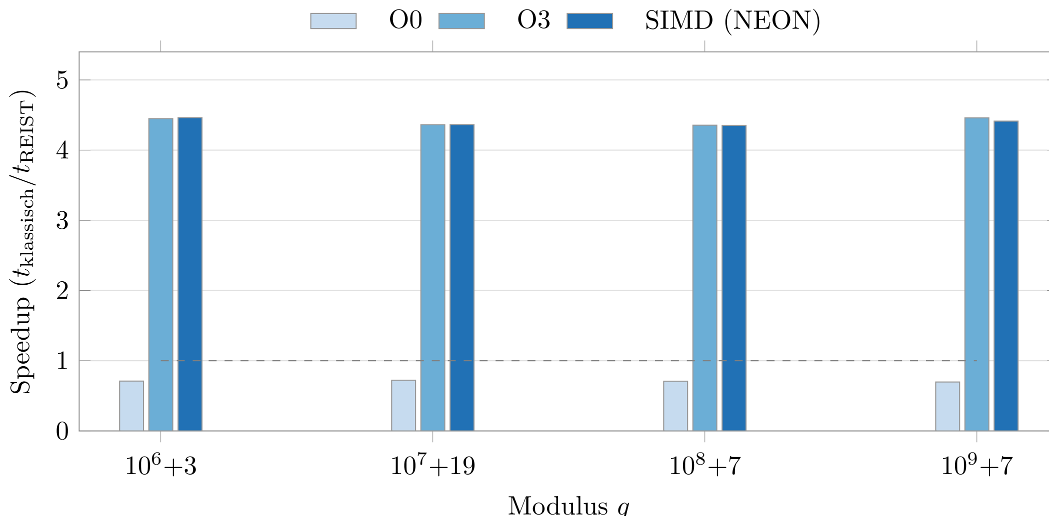

Abbildung 6: Polynomielle modulare Addition auf ARMv8-A: Ohne Optimierung (O0) ist die verzweigungsfreie REIST-Form langsamer ($\approx$ 0,7×), da die arithmetische Korrektur mehr Instruktionen kostet als der unoptimierte Restoperator. Mit O3 vektorisiert der Compiler den REIST-Kern automatisch; O3 und explizites NEON fallen zusammen (4,3–4,5×).

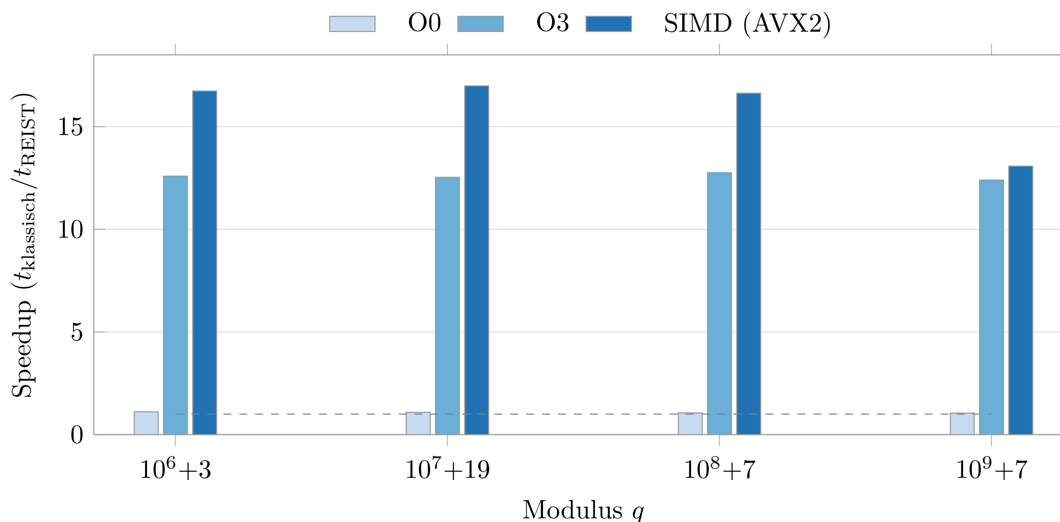

Abbildung 7: Polynomielle modulare Addition auf x86-64: 12,4–12,8× (O3) bzw. bis 17,0× (AVX2). Der datenparallele Kern ohne serielle Abhängigkeit ist der Idealfall der REIST-Korrektur: acht 32-Bit-Lanes werden pro Instruktion korrigiert, während die klassische Variante pro Koeffizient die Restsequenz durchläuft.

Die polynomielle Addition ist der datenparallele Idealfall: kein serieller Zustand, 1024 unabhängige Koeffizienten pro Durchlauf. Auf ARM (Abbildung 6) erreicht REIST 4,3–4,5×; bemerkenswert ist, dass die O3-Autovektorisierung bereits das NEON-Niveau erreicht: Der verzweigungsfreie Kern aus Listing 3 ist so regelmäßig, dass der Compiler ihn selbstständig vektorisiert. Auf x86-64 (Abbildung 7) ergeben sich 12,4–12,8× (O3) und bis zu 17,0× (AVX2).

Der O0-Fall auf ARM fällt aus dem Rahmen: Dort ist REIST mit $\approx 0{,}7\times$ langsamer als die klassische Variante. Ohne Optimierung zahlt die branchless-Form ihre zusätzlichen Instruktionen (zwei Vergleiche, zwei Multiplikationen) voll aus, während der Restoperator von der Hardware-Divisionseinheit profitiert. Die Gewinne der REIST-Formulierung entstehen hier also erst im Zusammenspiel mit Optimierung und Vektorisierung.

8.3 Negativ- und Neutralkontrollen

Tabelle 3: Kontrollmessungen: Workloads ohne strukturellen Ansatzpunkt für REIST. Speedup $t_{\text{klassisch}}/t_{\text{REIST}}$; Werte < 1 bedeuten eine Verlangsamung.

Workload	System	O0	O3	SIMD
Reine Restberechnung	ARM	0,86×	1,02×	1,01×
Reine Restberechnung	x86-64	0,68×	0,96×	0,85×
Hash-Mixing ($q = 10^6{+}3$)	ARM	0,47×	0,79×	0,83×
Hash-Mixing ($q = 10^6{+}3$)	x86-64	0,64×	0,81×	0,75×
ChaCha20-Keystream	ARM	Durchsatz unverändert (Abweichung $\leq 1\,\%$)		
ChaCha20-Keystream	x86-64	Durchsatz unverändert (Abweichung $\leq 10\,\%$)		

Tabelle 3 dokumentiert die Fälle, in denen REIST keinen Vorteil bietet:

- **Reine Restberechnung** ($r = T \bmod B$ für zufällige T): REIST und klassisches Modulo sind bei O3 praktisch gleich schnell (Unterschied $\leq 4\,\%$). Erwartungsgemäß, denn ohne persistenten Zustand gibt es kein Intervall, das gehalten werden könnte; beide Varianten müssen dieselbe Reduktionsarbeit leisten. REIST ist ein Optimierer für *additive Aktualisierungen*, nicht für die allgemeine Restberechnung.
- **ARX-Workloads** (ChaCha20): Algorithmen, die ausschließlich Addition, Rotation und XOR auf Zweierpotenz-Ringen verwenden [9], enthalten keine Modulo-Operationen gegen allgemeine Moduli; der Keystream-Durchsatz bleibt unverändert.
- **Hash-Mixing**: In Mischfunktionen mit multiplikativer Diffusion ist REIST durchgängig 15–25 % langsamer. Die statistische Diffusion dieser Funktionen beruht gerade auf den Überlauf- und Wraparound-Eigenschaften des klassischen nicht-negativen Rings; der vorzeichenbehaftete Korrekturterm fügt hier Arbeit hinzu, ohne Struktur zu ersparen.

8.4 Multicore- und Skalierungsdiagnostik (Apple M2 Pro)

Ergänzend zur Kernmessung dokumentiert die Suite eine Multicore-Diagnostik auf dem M2 Pro (10 Threads, 10^8 Iterationen, thread-lokale Daten): Reine Addition skaliert unter NEON mit 3,26× gegenüber skalar, eine Barrett-artige arithmetikdichte Last mit 4,95×, die vollständige Reduktion mit Zentrierung (REIST-Tree-artig) mit 3,33×; der Speicherbandbreiten-Kontrolltest liegt bei 3,49×. Die SIMD-Skalierung der REIST-Korrektur liegt damit im erwarteten Bereich der Plattform und ist kein Messartefakt; zugleich saturiert die Last alle Kerne, d. h. SIMD- und Thread-Parallelität kombinieren sich.

9 Mikroarchitektonische Analyse

9.1 Latenzen der beteiligten Operationen

Moderne CPUs führen ganzzahlige Divisionen in weitgehend seriellen Divisionseinheiten aus; Addition und Multiplikation sind dagegen kurzlatent, mehrfach vorhanden und out-of-order ausführbar [10]. Abbildung 8 stellt die Größenordnungen gegenüber.

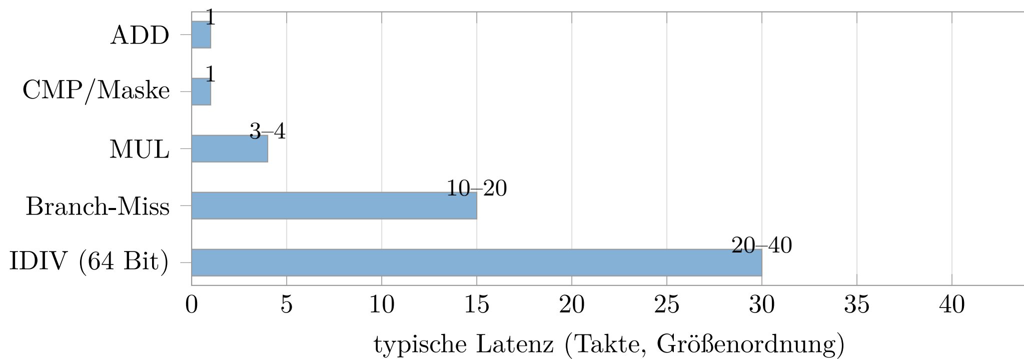

Abbildung 8: Größenordnung der Latenzen beteiligter Operationen auf aktuellen Kernen (zusammengefasst nach [10]; exakte Werte variieren je Mikroarchitektur). Die REIST-Korrektur verwendet ausschließlich die oberen drei Klassen; klassische Restpfade enthalten die unteren beiden zumindest strukturell.

9.2 Abhängigkeitsketten und ILP

Der entscheidende Effekt liegt nicht in einer einzelnen Instruktion, sondern in der Länge und Form der Abhängigkeitskette pro Aktualisierung:

- **Klassisch** (konstanter Divisor, O3): Magic-Multiply → Shift → Sign-Mask → Subtraktion → Rückmultiplikation → Subtraktion (Listing 1). Die Sign-Mask hängt am Eingangswert und verlängert den kritischen Pfad; das Ergebnis wird für die nächste Iteration benötigt.
- **REIST**: Addition → Vergleich → maskierte $\pm B$-Korrektur. Drei kurzlatente Operationen; Vergleich und Maske sind unabhängig voneinander schedulbar.

Der Wegfall des Sign-Mask-Pfades reduziert Registerdruck und Datenabhängigkeiten und gibt superskalaren Kernen mehr Spielraum zur parallelen Ausführung. Als grobes Kostenmodell:

$$C_{\text{REIST}} \approx L_{\text{ADD}} + L_{\text{CMP}} + L_{\text{MASK}} \quad \text{gegenüber} \quad C_{\text{MOD}} \approx L_{\text{Magic}} + L_{\text{SignMask}} + L_{\text{Korrektur}}, \tag{8}$$

mit $C_{\text{REIST}} < C_{\text{MOD}}$ für additionsdominierte Schleifen. Bei *nicht* zur Übersetzungszeit bekanntem Divisor entfällt auf der klassischen Seite die Magic-Multiply-Übersetzung, und die Divisionsinstruktion (20–40 Takte) kehrt in jeden Schleifendurchlauf zurück; die REIST-Korrektur bleibt unverändert divisionsfrei und benötigt lediglich eine einmalige Restoperation zur Erstzentrierung unzentrierter Eingaben (Abschnitt 11).

9.3 Verzweigungsverhalten

Klassischer Modulo-Code für vorzeichenbehaftete Werte enthält häufig bedingte Korrekturen zur Erzwingung von $r \geq 0$. Jede datenabhängige Verzweigung riskiert Fehlvorhersagen mit Pipeline-Flushes von 10–20 Takten. Die REIST-Korrektur ist strikt maskenbasiert (Listings 3 und 4): Es existiert kein datenabhängiger Sprung, die Fehlvorhersagerate des Korrekturpfads ist strukturell null. Dieselbe Eigenschaft ist die Voraussetzung der Vektorisierung, denn SIMD-Lanes kennen keinen divergenten Kontrollfluss.

9.4 Assembler-Inspektion

Die Benchmark-Suite erzeugt zu jedem Kern die übersetzte Assembler-Datei und klassifiziert sie automatisch. Die Befunde (x86-64, O0/O3) bestätigen die Analyse:

- In keiner der übersetzten Varianten mit konstantem Divisor erscheint eine `DIV`-Instruktion — der Compiler ersetzt sie durchgängig durch Magic-Multiply-Sequenzen. Der Vergleich „REIST vs. Modulo“ ist also tatsächlich ein Vergleich „Korrekturregel vs. optimierter Restpfad“, nicht „gegen naive Division“.
- Die klassischen Kerne enthalten bei O3 Sign-Mask-Muster (`sar`/`shr` des Eingangswerts); die REIST-Kerne nicht.
- Die REIST-Kerne bestehen aus ADD/CMP/Masken-Sequenzen, die bei O3 auf Vektorregister abgebildet werden (Autovektorisierung der polynomiellen Addition).

10 Hardware-Evaluation: FPGA-Implementierung

Die CPU-Messungen zeigen, dass die REIST-Korrektur den optimierten Restpfad des Compilers schlägt. Auf einem FPGA lässt sich darüber hinaus fragen, wie die Korrektur gegen dedizierte Divisions-Hardware abschneidet: in Taktzyklen, Chipfläche und erreichbarer Taktfrequenz. Dazu wurde eine eigenständige Benchmark-Engine in VHDL implementiert und auf einem Gowin-GW2A-18-FPGA (Tang Primer 20K) vermessen [14]. Als Referenz dient kein handgeschriebener Divider, sondern das herstellergenerierte *Gowin Integer Division*-Soft-IP [15], also die Komponente, die ein Entwickler für $a \bmod B$ tatsächlich instanziieren würde; hier in der 32-Bit-Konfiguration mit Pipeline-Latenz 2 (Durchsatz 1 Ergebnis/Takt).

Die Messung reiht sich damit in eine aktive Forschungslinie zu Hardwareimplementierungen der gitterbasierten Kryptographie ein. Insbesondere die Arbeiten von Howe et al. zeigen durchgängig, dass Fläche, Taktfrequenz und die Struktur der modularen Arithmetik die bestimmenden Entwurfsgrößen solcher Implementierungen sind — von kompakten gitterbasierten Signaturen [16] und Verschlüsselung über Standard-Gittern [17] über Schlüsselkapselung auf eingebetteten Plattformen [20] bis zur gezielten Ausnutzung von Parallelität in FrodoKEM-Hardware [21]. Die vorliegende Evaluation isoliert aus diesem Kontext einen einzelnen Baustein — die Reduktion in additiven modularen Aktualisierungen — und vermisst ihn unter kontrollierten Bedingungen.

10.1 Der REIST-Rechenpfad in Hardware

Der REIST-Datenpfad setzt die Korrekturregel aus Satz 3.6 unmittelbar in Logik um: zwei Komparatoren, ein Addierer/Subtrahierer, ein Multiplexer. Listing 5 zeigt den kombinatorischen Kern:

```
-- lo = -floor(B/2),  hi = lo + B = ceil(B/2)
half <= shift_right(b, 1);
lo   <= -half;
hi   <= b - half;

r <= sum - b when sum >= hi else
     sum + b when sum <  lo else
     sum;
```

Listing 5: Der kombinatorische REIST-Korrekturschritt (`reist_core.vhd`, gekürzt). Ein Guard-Bit (W downto 0) stellt sicher, dass weder $acc + x$ noch die $\pm B$-Korrektur überlaufen. Die floor/ceil-Aufteilung behandelt gerade und ungerade Moduli einheitlich.

Die Hardware verwendet das Intervall $[-\lfloor B/2 \rfloor, \lceil B/2 \rceil)$ — für gerades B exakt die kanonische Konvention $[-B/2, B/2)$ aus Definition 3.1, für ungerades B das identische symmetrische Vertretersystem. Die Korrektheit des Kerns wird in der Simulation gegen die Software-Referenzimplementierung geprüft (Testbench `tb_reist_core`), bevor auf der Hardware gemessen wird.

Der Modul b ist dabei ein *Laufzeiteingang* des Kerns, keine Synthesekonstante: Dieselbe instanziierte Einheit verarbeitet in der Benchmark-Engine nacheinander alle vier Moduli. Die Hardware belegt damit unmittelbar, dass die REIST-Korrektur keinen zur Übersetzungszeit fixierten Divisor benötigt (vgl. Korollar 3.7 und Abschnitt 4.3).

Vorzeichenbehaftete, um null zentrierte Darstellungen sind der Hardware gitterbasierter Kryptographie dabei nicht fremd: Fehlerterme werden dort aus symmetrischen, um null zentrierten diskreten Gaußverteilungen gezogen, deren flächen- und laufzeiteffiziente (und seitenkanalresistente) Sampler ein eigenes, gut untersuchtes Entwurfsfeld bilden [18, 19]. Die REIST-Einheit setzt dieselbe Symmetrie in der Reduktionsstufe fort, sodass zentriert erzeugte Werte ohne Darstellungswechsel akkumuliert werden können.

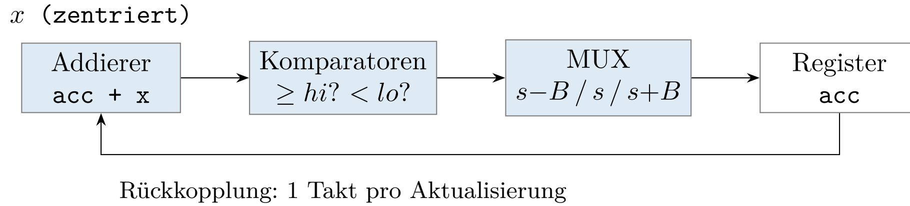

Abbildung 9: REIST-Akkumulator in Hardware: Addierer, Komparatoren und Multiplexer bilden eine kurze kombinatorische Stufe; die Rückkopplung schließt in einem Takt. Ein Divisions-IP an derselben Stelle müsste seine volle Pipeline-Latenz in jeder Iteration der Abhängigkeitskette abwarten.

10.2 Messaufbau

Wie bei den CPU-Messungen ist der entscheidende Punkt, dass modulare Akkumulation eine Abhängigkeitskette bildet: Jeder Schritt benötigt das Ergebnis des vorigen ($acc \leftarrow \text{reduce}(acc + x)$); ein gepipelintes Divisions-IP kann Iterationen daher nicht überlappen und zahlt seine volle Latenz in jedem Schritt. Die Benchmark-Engine misst deshalb drei Pfade über jeweils $N = 1024$ modulare Additionen und die Moduli 251, 256, 1009 und 65 521 (ungerade/gerade, klein/groß):

1. **REIST:** zentrierter Akkumulator in der Abhängigkeitskette — ein Takt pro Schritt;
2. **IP, Abhängigkeitskette:** dieselbe Kette, reduziert durch das Divisions-IP (ausgeben, Latenz abwarten, zurückführen);
3. **IP, unabhängiger Strom:** voneinander unabhängige Reduktionen, die die IP-Pipeline füllen — der Bestfall des IP.

Die Zyklenzähler werden auf der Hardware erfasst und über UART ausgegeben; die Ergebnisse sind über alle vier Moduli identisch, da weder Korrektur noch IP-Latenz vom Zahlenwert abhängen. Für Fläche und Taktfrequenz isolieren zwei Mess-Tops mit *identischem* Rahmen (LFSR-Stimulus → Eingangsregister → Addierer → Reduktionseinheit → Ausgangsregister) den Beitrag der Reduktionseinheit; die beiden Projekte unterscheiden sich ausschließlich in dieser Einheit.

10.3 Ergebnisse

Tabelle 4 fasst die Messungen zusammen. Betrachtet man nur Taktzyklen, gewinnt REIST in der Abhängigkeitskette (1 gegen 4 Takte) und liegt beim unabhängigen Durchsatz gleichauf. Hinzu kommen jedoch Fläche und Taktfrequenz: Das Divisions-IP packt die 32-Bit-Division in zwei Pipeline-Stufen, sodass jede Stufe ein sehr langer kombinatorischer Pfad ist (189 Logikstufen) — es erreicht nur 8,1 MHz. Die REIST-Korrektur ist ein Komparator und ein Addierer (8 Logikstufen) und schließt bei 161,8 MHz. In Echtzeit pro Aktualisierung (Abbildung 10) verschwindet damit auch das scheinbare Unentschieden beim unabhängigen Durchsatz.

Tabelle 4: FPGA-Ergebnisse (Gowin GW2A-18, Tang Primer 20K): Zyklen pro Aktualisierung (auf Hardware gemessen, $N = 1024$, alle Moduli identisch) sowie Fläche und Taktfrequenz der isolierten Reduktionseinheiten (Place-and-Route-Berichte, identischer Messrahmen).

Metrik	REIST	Divisions-IP	Verhältnis
Takte/Schritt, Abhängigkeitskette	1	4	4×
Takte/Ergebnis, unabhängiger Strom	1	≈ 1	$\approx 1\times$
Logikzellen (LUT + ALU)	101	1276	12,6× kleiner
Register (FF)	85	194	2,3× weniger
DSP-Blöcke	0	0	—
Maximale Taktfrequenz	161,8 MHz	8,1 MHz	20× höher
Logikstufen im kritischen Pfad	8	189	23,6× kürzer

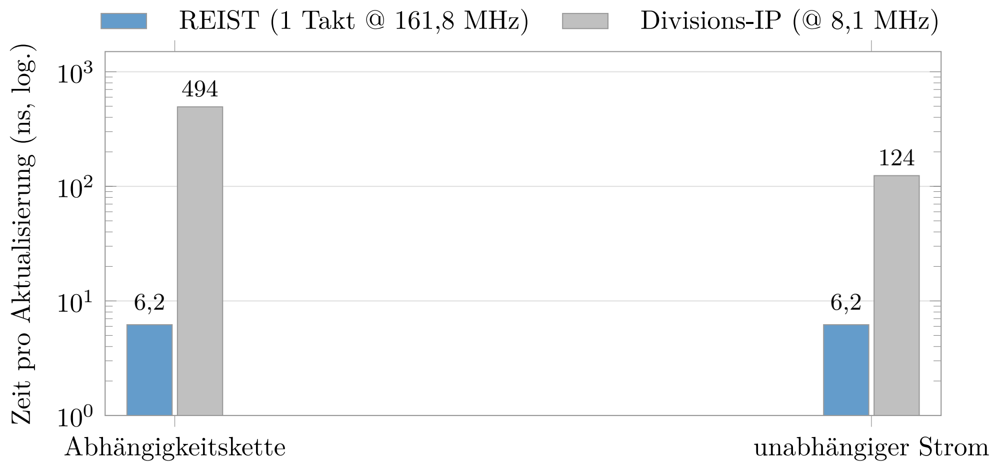

Abbildung 10: Echtzeit pro modularer Aktualisierung auf dem FPGA (logarithmische Skala): Zyklen und Taktfrequenz kombiniert. In der vermessenen akkumulativen Rückkopplungskette ergibt sich unter den gewählten IP-Parametern (32 Bit, Latenz 2) eine rund 80-fach geringere Zeit pro Aktualisierung; selbst im Bestfall des IP (gefüllte Pipeline, unabhängige Eingaben) bleibt ein Faktor von rund 20. Die Aussage gilt für diese Zielstruktur und Messkonfiguration; die Verallgemeinerung über die IP-Parameter hinweg liefert das Schrankenargument in Abschnitt 10.4.

10.4 Der Latenz-Kompromiss des Divisions-IP

Die Pipeline-Latenz des IP ist konfigurierbar, und man könnte einwenden, eine tiefere Pipeline würde die niedrige Taktfrequenz beheben. Dieser Einwand lässt sich allgemein entkräften. Sei L die Pipeline-Tiefe des Dividers, $f_{\mathrm{IP}}(L)$ seine damit erreichbare Taktfrequenz und T_{div} die gesamte kombinatorische Verzögerung des Divisionsnetzes. Auch bei idealer Stufenteilung gilt $f_{\mathrm{IP}}(L) \leq L/T_{\mathrm{div}}$, und für die Abhängigkeitskette folgt

$$t_{\mathrm{Kette}}(L) \;=\; \frac{L}{f_{\mathrm{IP}}(L)} \;\geq\; T_{\mathrm{div}}, \tag{9}$$

d. h. die Zeit pro Kettenschritt ist durch die Gesamtverzögerung der Division nach unten beschränkt, unabhängig von der Pipeline-Tiefe. Mehr Stufen erhöhen die Taktfrequenz, verlängern aber die Latenz im gleichen Maß; in einer Rückkopplungsschleife hebt sich beides bestenfalls auf, während Registerzahl und Fläche mit L wachsen. Die REIST-Korrektur unterliegt dieser Schranke nicht, weil sie die Division gar nicht ausführt: Ihre Stufe ist eine einzelne kurze Vergleichs-Additions-Kaskade ($T_{\mathrm{REIST}} \ll T_{\mathrm{div}}$, hier 8 gegen 189 Logikstufen). Für unabhängige Ströme kann eine hinreichend tiefe IP-Pipeline zwar vergleichbaren *Durchsatz* erreichen — jedoch nur bei weiter wachsender Fläche, und der Flächenvorsprung von bereits 12,6× vergrößert sich mit jeder zusätzlichen Stufe. Es existiert somit keine Konfiguration des Divisions-IP, die die REIST-Korrektur in ihrer Zieldomäne (modulare Akkumulation mit phasenweise stabilem Modul) dominiert.

Zur Einordnung der gemessenen 4 Takte pro Kettenschritt: Sie enthalten neben der reinen IP-Latenz (2 Takte) den Ausgabe- und Rückführungsaufwand der Schleife; der rein architektonische Boden liegt bei 2:1. Beide Lesarten ändern das Ergebnis nicht.

10.5 Einordnung

Die FPGA-Messung bestätigt die These der Arbeit auf einer zweiten, von CPUs unabhängigen Plattform: Der Vorteil der zentrierten Korrektur ist kein Artefakt von Compilern oder Instruktionssätzen, sondern eine Eigenschaft des Rechenwegs. Wie bei den CPU-Messungen gilt die Abgrenzung unverändert: Die REIST-Einheit ersetzt ausschließlich die *Reduktion* in modularer Addition und Akkumulation. Sie ist kein allgemeiner Divider; für echte Division bleibt ein Divisions-IP erforderlich, und für $(a \cdot b) \bmod B$ wäre ein DSP-Multiplizierer mit einer REIST-, Barrett- oder Montgomery-Reduktion zu paaren. Die geringe Größe der Einheit (101 Logikzellen, keine DSP-Blöcke) macht sie zugleich zu einem brauchbaren Baustein für NTT-Beschleuniger mit zentrierten Koeffizientenbereichen und für Koprozessor-Integration, zumal Hardware-Entwürfe gitterbasierter Verfahren ihre Leistung wesentlich über die Vervielfachung paralleler Arithmetikeinheiten skalieren [21] und eine kleine Reduktionseinheit entsprechend oft instanziiert werden kann.

11 Limitationen

Die Ergebnisse aus den Abschnitten 8.3 und 10 sowie die Analyse aus Abschnitt 9 definieren den Geltungsbereich der Methode. REIST ist eine gezielte, keine universelle Optimierung:

1. **Kein Ersatz für Montgomery/Barrett:** Für multiplikationslastige modulare Arithmetik bleiben etablierte Reduktionsverfahren angemessen. Direkte Vergleiche mit handoptimierten Montgomery-/Barrett-Implementierungen liegen außerhalb des Umfangs dieser Arbeit; die Referenz ist der Compiler-Modulo-Operator.

2. **Stabiler Modul und Erstzentrierung:** Die REIST-Korrektur enthält keine Division im Schleifenkörper; der Modul B kann auch erst zur Laufzeit gewählt werden, solange er

während der Akkumulationsphase stabil bleibt (Korollar 3.7). Zwei Kosten bleiben dennoch: Die Erstzentrierung eines unzentrierten Werts erfordert eine gewöhnliche Restoperation, und ein pro Iteration wechselnder Modul entwertet die Zustandsinvariante; dann kommt keine der beiden Seiten ohne Division aus. Die Übersetzungszeit-Konstanz des Divisors ist dagegen nur für die Magic-Multiply-Optimierung der klassischen Vergleichsbaseline relevant.

3. **Kein persistenter Zustand, kein Gewinn:** Die reine Restberechnung profitiert nicht (Tabelle 3); der Vorteil entsteht ausschließlich, wenn ein Wert über viele Aktualisierungen im Intervall gehalten wird.

4. **ARX-basierte Verfahren:** Algorithmen ohne Modulo-Operationen (ChaCha20, BLAKE, SipHash u. a.) bieten keinen Ansatzpunkt.

5. **Diffusionslastige Workloads:** Hash-Mixing-Funktionen, deren Statistik auf den Wraparound-Eigenschaften des nicht-negativen Rings beruht, verlangsamen sich um 15–25 %.

6. **Memory-bound-Schleifen:** In speicherbandbreitenlimitierten Schleifen sind arithmetische Einsparungen zweitrangig; die Speicherlatenz bestimmt die Gesamtleistung.

7. **Unoptimierte Übersetzung:** Ohne Compiler-Optimierung kann die branchless-Form langsamer sein als der Restoperator (polynomielle Addition, ARM, O0: $\approx 0{,}7\times$); die Methode entfaltet sich im Zusammenspiel mit O3/SIMD.

8. **Kompatibilität:** REIST erzeugt andere Reste als die klassische Division. Bestehende Systeme, die explizit $r \in [0, B)$ erwarten, müssen angepasst werden; Anwendungen mit bewusst asymmetrischen Fehlertermen (z. B. Clipping, Kompression) können andere Ergebnismuster zeigen.

9. **Seitenkanäle:** Die Korrekturregeln sind verzweigungsfrei implementierbar; Konstantzeiteigenschaften hängen jedoch von der konkreten Implementierung (und dem Compiler) ab und werden hier nicht beansprucht.

12 Mögliche Erweiterungen

Compiler-Integration. Die automatische Erkennung REIST-geeigneter Muster — modulare Zähler und Akkumulatoren mit compile-time-konstantem oder zur Laufzeit schleifeninvariantem Modul — in LLVM oder GCC könnte verzweigungsfreie, SIMD-fähige Signed-Remainder-Arithmetik ohne Quelltextänderung erzeugen. Die Assembler-Befunde aus Abschnitt 9.4 zeigen, dass die nötigen Bausteine (Magic Multiply, Maskenkorrektur) bereits im Repertoire der Codegeneratoren liegen.

Hardware-Ausbau. Aufbauend auf der FPGA-Evaluation aus Abschnitt 10: ein modularer Multiplikationspfad (DSP-Multiplizierer gefolgt von REIST-Reduktion) im Vergleich zur IP-Divisionsreduktion, eine Skalierungsstudie über Wortbreiten (16/64 Bit), bei der die Divider-Fläche wächst, während die Korrektureinheit flach bleibt, sowie die Integration als speicherabgebildeter Koprozessor, der Software die zentrierte Reduktion direkt zugänglich macht.

RNS- und CRT-Systeme. In Residue-Number-Systemen reduzieren betragskleine Kanalreste die Fehlerfortpflanzung bei der Rekombination; eine vollständige theoretische Integration steht aus.

NTT-Varianten. Eine durchgängig zentrierte NTT — Montgomery/Barrett für die Multiplikationen, $\pm B$-Korrekturen für die Add/Sub-Schichten — könnte die hier gemessenen Additionsgewinne in reale Transformationspipelines übertragen.

Formale Driftanalyse. Die Fehlerstatistik aus Bemerkung 3.9 legt eine formale Analyse des Langzeitverhaltens in Regelschleifen und Akkumulatoren nahe, einschließlich nichtgleichverteilter Eingaben.

13 Schlussfolgerung

Die REIST-Division fasst den Rest der ganzzahligen Division als bidirektionalen Korrekturterm statt als einseitiges Residuum auf. Mathematisch ist sie die konsequente Wahl des betragsminimalen Vertretersystems, verbunden mit der Quotientenwahl durch Rundung (Satz 3.3); als solche ist sie klassisch. Ihr praktischer Wert beruht auf drei zusammenhängenden Eigenschaften: Das zentrierte Intervall macht die additive Aktualisierung zu einer einzigen $\pm B$-Korrektur (Satz 3.6); diese Korrektur ist verzweigungsfrei formulierbar; und die verzweigungsfreie Form ist vektorisierbar.

Die empirische Evaluation auf zwei CPU-Architekturen und drei Übersetzungsstufen zeigt, dass diese Kette auf moderner Hardware trägt: 2–4,4× (ARM) und 8–17× (x86-64) für additionsdominierte modulare Workloads, bei stabilen Werten über drei Größenordnungen des Modulus; zugleich aber keinen Gewinn ohne persistenten Zustand, keine Wirkung auf ARX-Code und messbare Verluste bei diffusionslastigen Mischfunktionen. Die FPGA-Evaluation bestätigt dasselbe Muster unabhängig von Compilern und Instruktionssätzen auf der Ebene der Logik: ein Takt pro Aktualisierung bei — in der vermessenen Konfiguration — 12,6× geringerer Fläche und 20× höherer Taktfrequenz als ein herstellergeneriertes Divisions-IP, mit einer beweisbaren unteren Schranke, die kein Pipeline-Kompromiss des Dividers unterlaufen kann (Abschnitt 10.4). REIST ist damit am treffendsten als gezieltes arithmetisches Primitiv für modulo-additionslastige Rechenkerne zu verstehen: eine SIMD- und hardwarefreundliche Darstellung, die größere modulare Arithmetik-Pipelines ergänzt, nicht ersetzt.

Literatur

[1] D. E. Knuth, *The Art of Computer Programming, Vol. 2: Seminumerical Algorithms*, 3. Aufl. Reading, MA: Addison-Wesley, 1997.

[2] I. Niven, H. S. Zuckerman und H. L. Montgomery, *An Introduction to the Theory of Numbers*, 5. Aufl. New York: Wiley, 1991.

[3] Y. Zhang und L. Qi, „Balanced modular arithmetic and its applications“, *Journal of Number Theory*, Bd. 142, S. 1–12, 2014, doi: 10.1016/j.jnt.2014.03.012.

[4] P. L. Montgomery, „Modular multiplication without trial division“, *Mathematics of Computation*, Bd. 44, Nr. 170, S. 519–521, 1985.

[5] P. Barrett, „Implementing the Rivest Shamir and Adleman public key encryption algorithm on a standard digital signal processor“, in *Advances in Cryptology — CRYPTO ’86*, LNCS 263, Springer, 1987, S. 311–323.

[6] T. Granlund und P. L. Montgomery, „Division by invariant integers using multiplication“, in *Proc. ACM SIGPLAN PLDI*, 1994, S. 61–72.

[7] R. Avanzi et al., „CRYSTALS-Kyber: Algorithm Specifications and Supporting Documentation“, NIST Post-Quantum Cryptography Standardization, 2021.

[8] L. Ducas et al., „CRYSTALS-Dilithium: A lattice-based digital signature scheme“, *IACR Trans. CHES*, Bd. 2018, Nr. 1, S. 238–268, 2018.

[9] D. J. Bernstein, „ChaCha, a variant of Salsa20“, Workshop Record of SASC, 2008.

[10] A. Fog, *Instruction Tables: Lists of Instruction Latencies, Throughputs and Micro-Operation Breakdowns*, Technical University of Denmark, fortlaufend aktualisiert.

[11] K. J. Åström und R. M. Murray, *Feedback Systems: An Introduction for Scientists and Engineers*. Princeton: Princeton University Press, 2012.

[12] A. V. Oppenheim und R. W. Schafer, *Discrete-Time Signal Processing*, 2. Aufl. Upper Saddle River, NJ: Prentice Hall, 1999.

[13] R. Stepan, *reist-crypto-bench: Cryptographic Benchmark Suite for REIST Division* (Quellcode, Messprotokolle, Assembler-Inspektion), `https://github.com/rudolfstepan/reist-crypto-bench`, 2025.

[14] R. Stepan, *REIST FPGA Benchmark Engine* (VHDL-Quellen, Testbenches, Gowin-Projekte und Messberichte, Verzeichnisse `rtl/reist` und `boards/tang_primer_20k/reist`), `https://github.com/rudolfstepan/6502-sbc-fpga`, 2025.

[15] Gowin Semiconductor, *Gowin Integer Division IP: User Guide*, Dokumentation zum Soft-IP-Generator der GowinEDA-Toolchain.

[16] J. Howe, T. Pöppelmann, M. O’Neill, E. O’Sullivan und T. Güneysu, „Practical lattice-based digital signature schemes“, *ACM Transactions on Embedded Computing Systems*, Bd. 14, Nr. 3, Art. 41, 2015.

[17] J. Howe, C. Moore, M. O’Neill, F. Regazzoni, T. Güneysu und K. Beeden, „Lattice-based encryption over standard lattices in hardware“, in *Proc. 53rd Annual Design Automation Conference (DAC)*, 2016, Art. 162.

[18] J. Howe, A. Khalid, C. Rafferty, F. Regazzoni und M. O’Neill, „On practical discrete Gaussian samplers for lattice-based cryptography“, *IEEE Transactions on Computers*, Bd. 67, Nr. 3, S. 322–334, 2018.

[19] A. Khalid, J. Howe, C. Rafferty, F. Regazzoni und M. O’Neill, „Compact, scalable, and efficient discrete Gaussian samplers for lattice-based cryptography“, in *Proc. IEEE International Symposium on Circuits and Systems (ISCAS)*, 2018, S. 1–5.

[20] J. Howe, T. Oder, M. Krausz und T. Güneysu, „Standard lattice-based key encapsulation on embedded devices“, *IACR Transactions on Cryptographic Hardware and Embedded Systems*, Bd. 2018, Nr. 3, S. 372–393, 2018.

[21] J. Howe, M. Martinoli, E. Oswald und F. Regazzoni, „Exploring parallelism to improve the performance of FrodoKEM in hardware“, *Journal of Cryptographic Engineering*, Bd. 11, 2021, doi: 10.1007/s13389-021-00258-7.

[22] J. Howe und B. Westerbaan, „Benchmarking and analysing the NIST PQC lattice-based signature scheme standards on the ARM Cortex M7“, in *Progress in Cryptology — AFRICACRYPT 2023* (zuerst vorgestellt auf der NIST Fourth PQC Standardization Conference, 2022), LNCS, Springer, 2023.