# Neon NTT – (Auto)formalised

## Human-Directed, Agentic Verification of Montgomery and Barrett Modular Arithmetic

Hanno Becker

Amazon Web Services, beckphan@amazon.com

**Abstract.** This document provides a machine-checked Isabelle/HOL formalisation of the modular-arithmetic core of the Neon NTT paper of Becker, Hwang, Kannwischer, Yang, and Yang [6]. We develop parametric theories of Barrett and Montgomery reduction and multiplication; the equivalence of Barrett and Montgomery arithmetic; the doubling- and rounding-Montgomery variants; and correctness and bounds theorems for Neon assembly kernels, against a hand-written model of the word arithmetic underlying the relevant Neon instructions.

The development is a directed auto-formalisation: definitions, theorem statements, and proofs were produced by Claude Opus 4.7 and 4.8 using `AutoCorrode`'s [5] LLM–Isabelle integration layers. The human author set the architecture, chose abstractions and proof strategies, often nudged the model toward shorter or cleaner proofs, and controlled which output entered the development.

This document is auto-generated from the Isabelle sources through Isabelle's document preparation system, eliminating drift between prose and formal artifact.

**Keywords:** Isabelle/HOL · Formal Verification · Auto-formalisation · LLMs · Modular Arithmetic · Barrett Reduction

## Contents

**1 Introduction** **3**

**2 Methodology** **4**

**3 Roadmap** **5**

**4 Integer approximations and modular residues** **6**
4.1 Integer approximations . . . . . 6
4.2 Approximation quality . . . . . 7
4.3 Modular residues . . . . . 8
4.4 Standard locales . . . . . 9

**5 Montgomery reduction and multiplication** **10**
5.1 Montgomery reduction . . . . . 10
5.2 Montgomery multiplication with a known constant . . . . . 12

**6 Barrett reduction and multiplication** **13**
6.1 Barrett reduction . . . . . 13
6.2 Barrett multiplication . . . . . 13
6.3 Refined Barrett reduction . . . . . 13

**7 Barrett ↔ Montgomery bridge** **14**
7.1 Bridge: Barrett reduction and multiplication as Montgomery . . . . . 14
7.2 Bounds for Barrett reduction and multiplication . . . . . 14
7.3 Bounds for refined Barrett reduction . . . . . 16

**8 Quality of the Barrett bounds** **16**
8.1 Empirical maxima and analytic bounds . . . . . 17
8.2 Computable maxima via exhaustive sweep with early termination . . . . . 17
8.3 Example 1: `N = 97`, `n = 8` (prime, illustrative 8-bit modulus) . . . . . 18
8.4 Example 2: `N = 3329`, `n = 16` (ML-KEM modulus) . . . . . 19
8.5 Example 3: `N = 8380417`, `n = 32` (ML-DSA modulus) . . . . . 19

**9 Montgomery multiplication via doubling and rounding** **20**
9.1 Montgomery Multiplication via Doubling . . . . . 20
9.2 Montgomery Multiplication via Rounding . . . . . 20
9.3 Montgomery Multiplication via Rounding, no halving . . . . . 21

**10 A conceptual view of the Barrett–Montgomery bridge** **21**
10.1 Barrett arithmetic via Euclidean rounding . . . . . 22
10.2 Montgomery arithmetic via 2-adic rounding . . . . . 22
10.3 The Barrett–Montgomery bridge . . . . . 24

**11 Barrett division for even moduli** **24**
11.1 Montgomery arithmetic again – for any modulus . . . . . 24
11.2 The Barrett–Montgomery bridge, again . . . . . 25
11.3 Exactness of Barrett division via signed-canonicity . . . . . 26
11.4 ML-DSA decompose specialisations . . . . . 27

**12 Models of common word operations** **27**
12.1 Low-half multiply (`MUL`) and multiply-accumulate (`MLA`, `MLS`) . . . . . 28
12.2 Plain high-half multiply: `MULH` . . . . . 28
12.3 High-half doubling multiplies: `SQDMULH` and `SQRDMULH` . . . . . 29
12.4 Rounding multiply-accumulate high: `SQRDMLAH` . . . . . 30
12.5 Halving subtract: `SHSUB` . . . . . 30
12.6 Rounding right shift: `SRSHR` . . . . . 30
12.7 Assembly-style notation for kernel definitions . . . . . 31
12.8 Conformance testing . . . . . 31

**13 Neon kernels for Barrett-style modular arithmetic** **31**
13.1 Signed Barrett multiplication . . . . . 32
13.2 Unsigned-style Barrett multiplication . . . . . 32
13.3 Single-input Barrett reduction . . . . . 33
13.4 Refined Barrett reduction . . . . . 33
13.5 AArch64 Barrett division for ML-DSA *decompose* . . . . . 35

**14 Neon kernels for Montgomery-style modular arithmetic** **35**
14.1 Signed Montgomery multiplication . . . . . 35
14.2 Signed Montgomery multiplication (rounding variant) . . . . . 36
14.3 Doubled Montgomery multiplication . . . . . 36

**15 Conclusions** **37**

# 1 Introduction

The Number Theoretic Transform (NTT) is a performance-critical component of lattice-based cryptography: ML-KEM [14] and ML-DSA [13] both spend the majority of their arithmetic time in NTTs and the modular arithmetic operations they are comprised of. Fast implementations of ML-KEM and ML-DSA — including `mlkem-native` [16] and `mldsa-native` [15] — handle those modular arithmetic cores through bespoke assembly instruction kernels tailored to the target architecture.

The Neon NTT paper [6] introduced multiple novel such assembly kernels for the Armv8-A Neon vector instruction set, notably signed Barrett multiplication, and presented a bounds reasoning for Barrett arithmetic based on an equivalence between Barrett and Montgomery multiplication and reduction. The correctness arguments for the kernels and the Barrett–Montgomery equivalence are pen-and-paper, and errors at this level are easy to make and easy to miss: an off-by-one in an absolute bound, a mis-stated parity hypothesis, or a silently shared assumption between the Barrett and Montgomery analyses can compromise an otherwise-audited NTT.

This document provides a machine-checked Isabelle/HOL formalisation of this modular-arithmetic core of [6]. First, at the level of abstract algebraic operations: We develop parametric theories of Barrett and Montgomery reduction and multiplication, uniform in modulus, word width, and integer approximation, prove the identity linking Barrett and Montgomery arithmetic, and treat the doubling- and rounding-Montgomery variants. We also offer a conceptual recasting of the Barrett–Montgomery equivalence through the duality between Euclidean and 2-adic rounding. Second, at the level of concrete assembly kernels: we model the relevant Neon instructions — `MUL`, `MLA`, `MLS`, `MULH`, `UMULH`, `SQDMULH`, `SQRDMULH`, `SQRDMLAH`, `SHSUB`, and `SRSHR` — as operations on machine words, and prove correctness and bounds theorems for the signed-word kernels shipped by `mlkem-native` and `mldsa-native`. The kernel-level theorems are stated against this abstract word model, assumed to match the corresponding Armv8-A ISA semantics; NTT- and butterfly-level correctness, and a decoder from the abstract model to concrete binaries, are out of scope.

Beyond its technical content, the development serves as a case study in human-directed, model-generated formalisation. Claude Opus 4.7 and 4.8, connecting to Isabelle through the open-source `AutoCorrode` [5] toolkit, drafted the Isabelle source under the author's direction — the author owned the architecture and proof strategy, while the model supplied the formal text and absorbed the bookkeeping of iteration and refactoring. Section 2 describes the loop in more detail.

Our formalisation uses the Isabelle/HOL interactive proof assistant. Isabelle/HOL is one of the most mature proof assistants available today, with a development history spanning over three decades. It has been used in large-scale verification efforts including the seL4 microkernel [12, 11], the formal proof of the Kepler conjecture [9], recent work on cryptographic library verification at Apple [2], and the ongoing verification of the AWS Nitro Isolation Engine [1]. The Archive of Formal Proofs [3], a refereed collection of Isabelle developments, hosts close to a thousand entries at the time of writing. Beyond maturity, we chose Isabelle/HOL for its general ease of use; its structured proof language Isar, which reads close to natural mathematical exposition; its document preparation system, which makes it possible to share a single source between paper and formal proof; and the author's prior familiarity with the system.

**Contributions.**

1. A machine-checked Isabelle/HOL formalisation of the arithmetic core of [6].

2. A demonstrator for human-directed, model-generated formalisation at the scale of a research paper.

We do not claim novelty on the formal verification of assembly-level modular arithmetic kernels — several prior efforts have established such results (see next section).

**Related work.** [6] is the reference for the algorithms and bounds formalised here; its Lemma 1, Propositions 1–4, Facts 1–3, Corollaries 1 and 2, and Algorithms 2, 5, 7, 8, 9, 10, 11, 12, and 13 each have a named counterpart in this development. Algorithms 1, 3, 4, and 6 are pseudocode renderings of the same operators, captured at the operator level rather than as separate formal artifacts.

Multiple prior efforts have established formal correctness for assembly-level modular arithmetic kernels used in post-quantum cryptography. CryptoLine verifies NTT multiplications for Kyber, SABER, and NTRU at the instruction level for both AVX2 and Cortex-M4 [10, 8]. The Formosa Crypto project [4] uses Jasmin and EasyCrypt to produce verified high-speed implementations with end-to-end guarantees from source to binary. Apple's recent verification of corecrypto [2] covers assembly-level modular arithmetic in a production cryptographic library. And `mlkem-native` and `mldsa-native` ship HOL Light + `s2n-bignum` proofs that work at the binary level against a model of the Armv8-A instruction decoder and semantics for the deployed kernels. Relative to these efforts, the present development does not target a specific deployed binary but isolates the algebraic content shared across kernels — parametrically over modulus, word width, and rounding mode — so that a new kernel or parameter set reuses the bounds rather than reproving them.

**Source code.** The Isabelle sources for this document reside in the `mldsa-native` repository [15] under https://github.com/pq-code-package/mldsa-native/tree/main/proofs/isabelle/neon_ntt.

## 2 Methodology

**AutoCorrode.** Large language models are powerful across many domains, and their efficacy can often be amplified by providing direct interfaces to their target, such as CLIs or MCPs. In the context of interactive theorem proving, a tight feedback loop based on a direct and persistent connection between agent and proof kernel allows the agent to develop proofs faster and more effectively than a repeated edit-and-rebuild loop.

The `AutoCorrode` [5] verification infrastructure includes two agent–prover interfaces for Isabelle. First, Isabelle/Q ("I/Q") is an Isabelle/jEdit plugin exposing proof editing and exploration as an MCP server — the agent edits theory files and observes prover feedback in the same way a human would, without triggering a rebuild after every change. However, it is not conducive to concurrent edits by both the human and the agent. For that, Isabelle/REPL ("I/R") lets the agent talk directly to the Isabelle/ML prover process in a REPL loop, bypassing jEdit and PIDE. I/R is integrated into I/Q: the agent can fork a REPL at an arbitrary point in a theory opened in Isabelle/jEdit (e.g. a `sorry`'ed proof) and start exploring alternative theory texts without yet making the edits in jEdit. I/R is ad-hoc and may be replaced by something more PIDE-native in the future, but it demonstrates what concurrent human/AI proof editing can look like: the agent works out multiple proofs while the human keeps editing the document elsewhere. `AutoCorrode` also includes Isabelle/Proxy ("I/P"), which offloads the Isabelle/ML prover process to a remote server while keeping Isabelle/jEdit and the JVM local — a powerful capability for large-scale developments, though less relevant for the present example.

**The human/AI loop.** The agent provided the initial auto-formalisation of [6] completely autonomously. It lacked clarity, brevity, and generally did not make for a paper-style artifact that a human would like to consume. It was the task of the human author to then gradually nudge the agent to adjust the narrative, compact proofs, tweak definitions,

introduce suitable syntax, and decide on what to abridge and what to include. No proof was ultimately written by the human — even under strong editorial direction, the formal text remained model-generated. Despite this heavy involvement, the development was much faster than had the human formalised the material themselves. In addition to the time-saving of not having to write definitions and proofs manually, the agent was hugely effective at making cross-cutting changes such as syntax adjustments, propagating signature changes through many call sites, and relocating code across theories.

It was noteworthy that the agent had often disregarded existing proofs and even proof strategies from [6], and come up with its own approaches — for better or for worse. Two examples stand out. First, instead of relying on the Barrett–Montgomery equivalence to establish bounds for Barrett arithmetic, the agent proved those bounds directly. This is interesting per se, but directly opposed to the approach of [6], which demonstrates how the obvious Montgomery bound can be carried over to Barrett using the equivalence; in the present development we follow [6] via the equivalence (Section 7), and the agent's direct route was discarded. Second, [6, Algorithm 8] states the correctness of Montgomery multiplication with rounding under a parity hypothesis (`a` odd or `b` odd). The agent correctly noted that this assumption is unnecessary and proved the algorithm unconditionally correct — a genuine improvement over [6] surfaced by the formalisation.[1]

**Unifying paper and proof.** When paper and formal development live in separate artifacts, time and care is needed to ensure that they stay in sync as changes are made on either side. To avoid this overhead and soundness risk, we use Isabelle's document preparation system to auto-generate all documentation from the formal sources. Every chapter of this document is an Isabelle theory file. Refactoring a lemma moves the surrounding prose with it; adding a new abbreviation propagates to every displayed term downstream. Isabelle's mixfix syntax keeps the formal text close to the notation a reader would expect from a paper. The present version is heavily abridged — most auxiliary lemmas and almost all proofs are elided for the sake of highlighting the main arguments. The full theory contents can be included by rebuilding with different tags.

# 3 Roadmap

We provide a brief summary of each chapter:

**Section 4 (`Integer_Approximation`)** develops the parametric notion of integer approximation $[\![_]\!] : \mathbb{Q} \to \mathbb{Z}$ and the residue operator `z mod [|f|] N`.

**Section 5 (`Montgomery_Reduction`)** develops Montgomery reduction and multiplication operators. We distinguish additive vs. subtractive and signed vs. unsigned versions.

**Section 6 (`Barrett_Reduction`)** develops signed and unsigned Barrett reduction and multiplication operators, parametric in a choice of integer approximation $[\![_]\!]$. We also introduce a 'refined' Barrett reduction at an inflated radix. Output bounds are deferred to Section 7, where they fall out of the Barrett–Montgomery equivalence.

**Section 7 (`Barrett_Montgomery`)** proves the Barrett–Montgomery equivalence: Barrett reduction of `z` equals Montgomery reduction of `z * (R mod[|f|] N)`. This collapses the bounds proofs of the two families [6, Corollaries 1 and 2] into a single argument, and yields output bounds for both standard and refined Barrett reduction.

[1] Interestingly, a recent contribution (https://github.com/pq-code-package/mlkem-native/pull/1184) to `mlkem-native` had independently used this algorithm without the parity assumption (https://github.com/pq-code-package/mlkem-native/pull/1184#issuecomment-4485661049).

**Section 8 (`Barrett_Bound_Quality`)** checks the quality of the abstract Barrett bounds against empirical maxima at three concrete moduli: a small illustrative prime ($N = 97$), the ML-KEM modulus ($N = 3329$), and the ML-DSA modulus ($N = 8380417$).

**Section 9 (`Montgomery_Doubling`)** formalises the doubling- and rounding-Montgomery variants [6, Algorithms 7 and 8].

**Section 10 (`Bridge_Conceptual`)** revisits the Barrett–Montgomery equivalence from a conceptual viewpoint, casting it as a consequence of the duality between Euclidean and 2-adic rounding of a rational number.

**Section 11 (`Barrett_Division_Even`)** generalises the bridge to an arbitrary modulus, replacing the use of modular inverses by choices of modular quotients. This is relevant for the `decompose` routine in ML-DSA.

**Section 12 (`Word_Ops`)** models the per-lane behavior of relevant Neon instructions as parametric operations on arbitrary-width words.

**Section 13 (`Asm_Barrett`)** proves correctness of some Barrett Neon kernels — Barrett multiplication, plain Barrett reduction, refined Barrett reduction, and 'even' Barrett reduction — against the models of Section 12.

**Section 14 (`Asm_Montgomery`)** proves correctness of some Montgomery Neon kernels — including the doubling- and rounding-based variants — against the models of Section 12.

# 4 Integer approximations and modular residues

Barrett and Montgomery reduction are usually presented with a fixed rounding mode, but multiple variants exist: the textbook bound proofs are stated for round-to-nearest, the unsigned variant uses flooring, and the `SQRDMULH`-based signed Barrett kernel produces an even-rounded result. To avoid replicating the same proof four times we work uniformly over an arbitrary *integer approximation* $[\![_]\!] : \mathbb{Q} \to \mathbb{Z}$, prove the underlying arithmetic facts once, and specialise $[\![_]\!]$ at each use site.

This chapter develops the predicate `is_int_approx [\_]` defining what counts as an integer approximation, and the residue operator `z mod[f] N`, which generalises both the unsigned (`mod`$^+$) and signed (`mod`$^\pm$) representatives modulo `N`. We write `_Q` for the embedding $\mathbb{Z} \to \mathbb{Q}$ and `_/Q_` for the rational quotient of two integers.

```
notation rat_of_int ("_Q" [1000] 999)
abbreviation quot_of_int (infixl "'/Q" 70) where "x /Q y ≡ xQ / yQ"
```

## 4.1 Integer approximations

Following [6, §2.4], `is_int_approx [\_]` requires only that $|z - [\![z]\!]_\mathbb{Q}| \leq 1$, not the stronger $[\![z]\!] = z$ on integer inputs, which the even-rounding function $\lfloor\cdot\rceil_2$ fails.

```
definition ‹is_int_approx f ⟷ (∀z. |z - [z]Q| ≤ 1)› for f ("[\_]")
```

Standard instances of integer approximations are round-to-nearest $\lfloor\cdot\rceil$, round-half-down $\lfloor\cdot\rceil_\downarrow$, floor $\lfloor\cdot\rfloor$, and ceiling $\lceil\cdot\rceil$. Less canonically, the following is an integer approximation as well:

```
definition round_half_down :: ‹rat ⇒ int› (‹⌊_⌉↓›) where
  ‹round_half_down x = - round (-x)›
definition round_even (‹⌊_⌉2›) where ‹⌊z⌉2 ≡ 2 * ⌊z/2⌉›
```

We quickly confirm that $\lfloor\cdot\rceil$ is round-half-up and $\lfloor\cdot\rceil_\downarrow$ is round-half-down:

```
lemma shows ‹⌊1 /ℚ 2⌉↓ = 0› and ‹⌊1 /ℚ 2⌉ = 1› by eval+
```

We also confirm that indeed all of the above are integer approximations:

```
theorem is_int_approx_instances:
  shows is_int_approx_floor:           ‹is_int_approx ⌊·⌋›
    and is_int_approx_ceiling:         ‹is_int_approx ⌈·⌉›
    and is_int_approx_round:           ‹is_int_approx ⌊·⌉›
    and is_int_approx_round_half_down: ‹is_int_approx ⌊·⌉↓›
    and is_int_approx_round_even:      ‹is_int_approx ⌊·⌉₂›
  ⟨proof⟩
```

## 4.2 Approximation quality

The predicate `is_int_approx ⟦_⟧` only requires the rounding error `|z - ⟦z⟧ℚ|` to be bounded by `1`. Different approximations enjoy tighter uniform bounds: $\lfloor\cdot\rceil$ and $\lfloor\cdot\rceil_\downarrow$ never err by more than `1 / 2`, while $\lfloor\cdot\rfloor$, $\lceil\cdot\rceil$, and $\lfloor\cdot\rceil_2$ can come arbitrarily close to `1`. We capture this as the *approximation quality* `ε(⟦_⟧)`, the smallest rational bound on `|z - ⟦z⟧ℚ|`.

In general, the supremum of the rounding error need not be a rational number (see §4.2.1 for a concrete example), so we characterise `ε(⟦_⟧)` as the unique value (when one exists) that is both a uniform upper bound and least among all upper bounds; this is captured by the predicate `is_int_approx_quality`.

```
definition int_approx_quality (‹ε '(_')›) where
  ‹ε(f) ≡ SOME e. (∀z. |z - ⟦z⟧ℚ| ≤ e) ∧
                  (∀e'. (∀z. |z - ⟦z⟧ℚ| ≤ e') ⟶ e ≤ e')› for f ("⟦_⟧")
```

The uniform quality `ε(f)` dominates the actual rounding error `|x - ⟦x⟧ℚ|` at every input `x`. We name the latter `ε(f, x)`; the relation `ε(f, x) ≤ ε(f)` is immediate.

```
abbreviation int_approx_quality_at :: ‹(rat ⇒ int) ⇒ rat ⇒ rat› (‹ε '(_, _')›)
  where ‹ε(f, x) ≡ |x - ⟦x⟧ℚ|› for f ("⟦_⟧")
```

The five standard approximations admit explicit quality witnesses. Round-to-nearest and round-half-down attain the ideal quality `1 / 2`; the other three only come arbitrarily close to `1` but no rational below `1` is a uniform bound, so their quality is exactly `1`.

```
corollary int_approx_quality_instances:
  shows quality_round:           ‹ε(⌊·⌉) = 1/2›
    and quality_round_half_down: ‹ε(⌊·⌉↓) = 1/2›
    and quality_floor:           ‹ε(⌊·⌋) = 1›
    and quality_ceiling:         ‹ε(⌈·⌉) = 1›
    and quality_round_even:      ‹ε(⌊·⌉₂) = 1›
  ⟨proof⟩
```

For each of the five standard approximations, the worst-case error `ε (f)` is attained only at the supremum locus: for $\lfloor\cdot\rceil$ and $\lfloor\cdot\rceil_\downarrow$ at half-integers, approached but never attained for $\lfloor\cdot\rfloor$ and $\lceil\cdot\rceil$ at integers, and attained at odd integers for $\lfloor\cdot\rceil_2$ (e.g. $\lfloor 1\rceil_2$ `= 2`). Outside these loci the pointwise error is strictly smaller. Note: for $\lfloor\cdot\rfloor$ and $\lceil\cdot\rceil$ the strict inequality holds unconditionally the residue `x - of_int (floor x)` always lies in `[0, 1)`, so no precondition is needed.

```
lemma quality_at_strict:
  shows quality_at_strict_round:           ‹2 * x ∉ ℤ ⟹ ε(⌊·⌉, x) < ε(⌊·⌉)›
    and quality_at_strict_round_half_down: ‹2 * x ∉ ℤ ⟹ ε(⌊·⌉↓, x) < ε(⌊·⌉↓)›
```

```
    and quality_at_strict_round_even:      ‹x ∉ ℤ ⟹ ε(⌊·⌉₂, x) < ε(⌊·⌉₂)›
    and quality_at_strict_floor:           ‹ε(⌊·⌋, x) < ε(⌊·⌋)›
    and quality_at_strict_ceiling:         ‹ε(⌈·⌉, x) < ε(⌈·⌉)›
⟨proof⟩
```

Fun fact: For every $\delta \in [0, 1/2]$, the offset rounding $z \mapsto \mathtt{round}(z + \delta)$ is an integer approximation of quality $1 / 2 + \delta$. Setting $\delta = 0$ recovers $\lfloor\cdot\rceil$; $\delta = 1 / 2$ gives $\lfloor\cdot\rfloor$ (up to a half-integer shift). The map $\delta \mapsto \varepsilon(\ldots)$ is therefore a bijection $[0, 1/2] \to [1/2, 1]$, witnessing that the quality range is fully achieved.

```
lemma int_approx_quality_offset_round:
  assumes ‹0 ≤ δ› and ‹δ ≤ 1/2›
  shows ‹is_int_approx (λz. ⌊·⌉ (z + δ))›
    and ‹ε(λz. ⌊·⌉ (z + δ)) = 1/2 + δ›
⟨proof⟩
```

### 4.2.1 A non-standard example: the golden-ratio approximation

This subsection is for illustration and is not essential for the remainder of the development. We construct an integer approximation whose rounding threshold is the irrational number $(\sqrt{5} - 1)/2$ (the golden ratio minus one), proving that no rational quality witness exists.

```
definition golden_approx (‹⌊_⌉_φ›) where
  "⌊z⌉_φ = (THE n. let r = z - n_ℚ in
     (r ≥ 0 ∧ r*r + r < 1) ∨ (-1 < r ∧ r < 0 ∧ (r+1)*(r+1) + (r+1) ≥ 1))"
```

The golden approximation is indeed an integer approximation:

```
theorem is_int_approx_golden: "is_int_approx ⌊·⌉_φ"
  ⟨proof⟩
```

Its real quality is the golden ratio minus one, where the real quality $\varepsilon_{\mathbb{R}}(\mathtt{f})$ is defined analogously to the rational quality $\varepsilon(\mathtt{f})$.

```
theorem real_quality_golden: ‹ε_ℝ(⌊·⌉_φ) = (√5 - 1) / 2›
  ⟨proof⟩
```

Since the golden ratio is irrational, no rational quality exists:

```
theorem ‹¬ is_int_approx_quality ⌊·⌉_φ e›
⟨proof⟩
```

## 4.3 Modular residues

Each approximation $[\![_]\!]$ induces a residue $\mathtt{z\ mod}[\![\mathtt{f}]\!]\ \mathtt{N} := \mathtt{z} - \mathtt{N} \cdot [\![\mathtt{z} /_{\mathbb{Q}} \mathtt{N}]\!]$. Setting $[\![_]\!] = \lfloor\cdot\rfloor$ recovers the unsigned representative $\mathtt{z\ mod}^{+}\ \mathtt{N}$ in $\{0, \ldots, \mathtt{N}-1\}$; $[\![_]\!] = \lfloor\cdot\rceil$ the signed representative $\mathtt{z\ mod}^{\pm}\ \mathtt{N}$ in $\{-\lfloor \mathtt{N}/2 \rfloor, \ldots, \lfloor (\mathtt{N}-1)/2 \rfloor\}$.

```
definition ‹mod_approx f z N ≡ z - N * [⌊z /_ℚ N⌉]› for f ("[⌊_⌉]")

abbreviation mod_approx_paper (‹(_) mod [⌊_⌉] (_)› [70, 0, 71] 70) where
  ‹z mod[⌊f⌉] N ≡ mod_approx f z N›

abbreviation mod_unsigned (infixl ‹mod⁺› 70) where
  ‹z mod⁺ N ≡ mod_approx ⌊·⌋ z N›

abbreviation mod_signed (infixl ‹mod±› 70) where
  ‹z mod± N ≡ mod_approx ⌊·⌉ z N›
```

A sanity check: the $\lfloor\cdot\rfloor$-induced residue coincides with the standard unsigned reduction, justifying $\mathtt{mod}^{+}$ as the unsigned representative throughout.

```
lemma mod_unsigned_eq_mod:
  assumes ‹N > 0›
  shows ‹z mod⁺ N = z mod N›
  ⟨proof⟩
```

A general absolute bound |z mod ⟦f⟧ N| ≤ N holds for every integer approximation.

```
lemma mod_approx_bound:
  fixes f :: ‹rat ⇒ int› (‹⟦_⟧›)
  assumes ‹is_int_approx f› and ‹N > 0›
  shows ‹|z mod⟦f⟧ N| ≤ N›
⟨proof⟩
```

The following is merely a reformulation of the definition, but useful:

```
lemma magic_const_rat:
  fixes f :: ‹rat ⇒ int› ("⟦_⟧") and N R :: int
  assumes ‹is_int_approx f› and ‹N ≠ 0›
  shows ‹(⟦R /ℚ N⟧)ℚ = (R - R mod⟦f⟧ N) /ℚ N›
⟨proof⟩
```

If a quality witness exists for ⟦_⟧, the residue is bounded by ε (f) * N: the slope shrinks by exactly the approximation quality.

```
lemma mod_approx_bound_eps_at:
  fixes f :: ‹rat ⇒ int› (‹⟦_⟧›)
  assumes ‹N > 0›
  shows ‹|(z mod⟦f⟧ N)ℚ| = Nℚ * ε(f, z /ℚ N)›
⟨proof⟩

lemma mod_approx_bound_eps:
  fixes f :: ‹rat ⇒ int› (‹⟦_⟧›)
  assumes ‹is_int_approx_quality f e› and ‹N > 0›
  shows ‹|(z mod⟦f⟧ N)ℚ| ≤ Nℚ * ε(f)›
  ⟨proof⟩
```

For the signed and unsigned residues, both pinned to canonical intervals, the bound ε (f) * N from mod_approx_bound_eps specialises to the integer-form inequalities below: the signed residue lies in $\{-\lfloor N/2 \rfloor, \ldots, \lfloor (N-1)/2 \rfloor\}$ (matching ε (⌊·⌋) = 1 / 2), and the unsigned residue in $\{0, \ldots, N-1\}$ (matching ε (⌊·⌋) = 1).

```
lemma mod_canonical_bounds:
  assumes ‹N > 0›
  shows mod_signed_lower:   ‹- (N div 2) ≤ z mod± N›
    and mod_signed_upper:   ‹z mod± N ≤ (N - 1) div 2›
    and mod_unsigned_lower: ‹0 ≤ z mod⁺ N›
    and mod_unsigned_upper: ‹z mod⁺ N < N›
⟨proof⟩
```

## 4.4 Standard locales

The Barrett and Montgomery analyses repeatedly need the same side conditions on the modulus N and the bit-width n: N > 1, n > 0, N < 2^n, and — for the parts of the development that need it — N odd. The integer-approximation analyses likewise rely on is_int_approx f. We capture each context in an Isabelle **locale** — AnyModulus for the parity-agnostic modulus side conditions, OddModulus for AnyModulus plus oddness of N, IntegerApproximation for the rounding hypothesis, and BarrettContext combining OddModulus with IntegerApproximation (defined in Section 6). Lemmas that depend on these conditions are introduced with the locale annotation (in AnyModulus), (in OddModulus), or (in BarrettContext) and take no explicit assumes; the side conditions are inherited from the locale, and abbreviations like R ≡ 2^n become visible.

```
locale AnyModulus =
  fixes N :: int and n :: nat
  assumes Ngt1: ‹N > 1› and npos: ‹n > 0›
      and N_lt_R: ‹N < 2^n›
begin

abbreviation R :: int where ‹R ≡ 2^n›
end

locale OddModulus = AnyModulus +
  assumes Nodd: ‹odd N›
```

`OddModulus` is `AnyModulus` with the extra hypothesis that `N` is odd. Parity-agnostic material lives in `AnyModulus`; lemmas that genuinely need oddness stay in `OddModulus`, and can be migrated to `AnyModulus` as the dependence on parity is removed.

```
locale IntegerApproximation =
  fixes f :: ‹rat ⇒ int›
  assumes f_approx: ‹is_int_approx f›
begin
notation f (‹⟦_⟧›)
end
```

# 5 Montgomery reduction and multiplication

In this chapter, we discuss Montgomery reduction and Montgomery multiplication along the lines of [6] and establish their correctness and bounds. The idea of Montgomery reduction is to replace an expensive division by `N` with a division by `R = 2^n`, by efficiently adjusting a representative of `z mod N` to become divisible by $2^n$.

Inside `OddModulus`, the side conditions `0 < n` and `odd N` are inherited from the locale, and the modulus $2^n$ is exposed as the abbreviation `R`.

## 5.1 Montgomery reduction

Following [6, §2.4.2, Algorithm 2], we formalise Montgomery reduction along two independent axes: *additive* vs *subtractive* (`T = N^-1 mod R` with `z - k * N`, or `T = - N^-1 mod R` with `z + k * N`), and *signed* ($^\pm$) vs *unsigned* ($^+$) twist for the inner residue `k = z * T mod R`. Each reduction maps `z` to a representative of $z \cdot R^{-1}$ in $\mathbb{Z}_N$, where `R = 2^n`; the canonical inverse $\pm N^{-1}$ is supplied via the notation $N^{-1} \bmod 2^n$.

```
definition mont_sub_signed (‹mont_sub^± ⟦_,_⟧›) where
  ‹mont_sub^±⟦N, n⟧ z ≡ (z - ((z * (N^-1 mod 2^n)) mod^± 2^n) * N) div 2^n› for N

definition mont_add_signed (‹mont_add^± ⟦_,_⟧›) where
  ‹mont_add^±⟦N, n⟧ z ≡ (z + ((z * (- (N^-1 mod 2^n) mod 2^n)) mod^± 2^n) * N) div 2^n›

definition mont_add_unsigned (‹mont_add^+ ⟦_,_⟧›) where
  ‹mont_add^+⟦N, n⟧ z ≡ (z + ((z * (- (N^-1 mod 2^n) mod 2^n)) mod^+ 2^n) * N) div 2^n›

definition mont_sub_unsigned (‹mont_sub^+ ⟦_,_⟧›) where
  ‹mont_sub^+⟦N, n⟧ z ≡ (z - ((z * (N^-1 mod 2^n)) mod^+ 2^n) * N) div 2^n›
```

The signed additive and subtractive Montgomery operators agree except at the boundary case `z * N^-1 mod 2^n mod 2^n = 2^n - 1`, where they differ by exactly `N` [6, §2.4.2].

```
lemma (in OddModulus) mont_sub_eq_add_signed_generic:
  assumes ‹z mod 2^n ≠ 2^(n-1)›
  shows ‹mont_sub^±⟦N, n⟧ z = mont_add^±⟦N, n⟧ z›
⟨proof⟩

lemma (in OddModulus) mont_sub_eq_add_signed_exceptional:
  assumes ‹z mod 2^n = 2^(n-1)›
  shows ‹mont_add^±⟦N, n⟧ z = mont_sub^±⟦N, n⟧ z - N›
⟨proof⟩
```

### 5.1.1 Correctness and Absolute Bounds

Each of the four Montgomery reduction variants is correct and bounded by `|z| / R + N / 2` (signed) or `|z| / R + N` (unsigned).

```
theorem (in OddModulus) mont_correct:
  shows ‹(mont_add^±⟦N, n⟧ z * 2^n) mod N = z mod N›
    and ‹(mont_sub^±⟦N, n⟧ z * 2^n) mod N = z mod N›
    and ‹(mont_add^+⟦N, n⟧ z * 2^n) mod N = z mod N›
    and ‹(mont_sub^+⟦N, n⟧ z * 2^n) mod N = z mod N›
⟨proof⟩
```

The headline output bound: every Montgomery variant of `z` is bounded in absolute value by `|z|/R + N/2` in the signed forms, and by `|z|/R + N - N/R` in the unsigned forms. Both are sharp; they say that Montgomery reduction shrinks any input to *roughly* the canonical range, with a residual error tied to `N`.

```
theorem (in OddModulus) mont_bound:
  shows ‹|(mont_add^±⟦N, n⟧ z)_Q| ≤ |z| /_Q 2^n + N_Q / 2›
    and ‹|(mont_sub^±⟦N, n⟧ z)_Q| ≤ |z| /_Q 2^n + N_Q / 2›
    and ‹|(mont_add^+⟦N, n⟧ z)_Q| ≤ |z| /_Q 2^n + N_Q - N /_Q 2^n›
    and ‹|(mont_sub^+⟦N, n⟧ z)_Q| ≤ |z| /_Q 2^n + N_Q - N /_Q 2^n›
⟨proof⟩
```

The signed bounds are tight: there exist inputs at which the inequality is an equality. As an example, we pick `n = 3`, `N = 3`, so `R = 8`: then $\texttt{N}^{-1} \bmod \texttt{R} = 3$, `k = -4` at `z = -4` (additive) or `z = 4` (subtractive), with `|m| = 2` matching `|z| /_Q R + N_Q / 2`.

```
lemma mont_add_signed_bound_tight:
  shows ‹|(mont_add^±⟦3, 3⟧ (-4))_Q| = |-4| /_Q 2^3 + 3 /_Q 2›
⟨proof⟩

lemma mont_sub_signed_bound_tight:
  shows ‹|(mont_sub^±⟦3, 3⟧ 4)_Q| = |4| /_Q 2^3 + 3 /_Q 2›
⟨proof⟩
```

The unsigned bounds are also tight in the sharper form $|\texttt{m}|_Q \leq |\texttt{z}| /_Q \texttt{R} + \texttt{N}_Q - \texttt{N} /_Q \texttt{R}$. As an example, we pick `n = 3`, `N = 3`, so `R = 8`, with twist $\texttt{-N}^{-1} \bmod \texttt{R} = 5$. At `z = 3` we get `k = 7` and `m = 3`, matching $|\texttt{z}| /_Q \texttt{R} + \texttt{N}_Q - \texttt{N} /_Q \texttt{R} = 3$.

```
lemma mont_add_unsigned_bound_tight:
  shows ‹|(mont_add^+⟦3, 3⟧ 3)_Q| = |3| /_Q 2^3 + 3 - 3 /_Q 2^3›
⟨proof⟩
```

### 5.1.2 Tightness characterisation of the signed Montgomery bound

The signed Montgomery bound $|\texttt{m}| \leq |\texttt{z}| /_Q \texttt{R} + \texttt{N} /_Q 2$ is attained exactly when the input `z` has residue `R/2` modulo `R` and lies on the appropriate side of zero ($\texttt{z} \leq 0$ for the additive variant, $\texttt{z} \geq 0$ for the subtractive).

```
theorem (in OddModulus) mont_signed_bound_iff:
  shows mont_add_signed_bound_iff:
    <|(mont_add^± [N, n] z)_Q| = |z| /_Q 2^n + N_Q / 2 ⟷ z mod^+ 2^n = 2^(n-1) ∧ z ≤ 0>
  and mont_sub_signed_bound_iff:
    <|(mont_sub^± [N, n] z)_Q| = |z| /_Q 2^n + N_Q / 2 ⟷ z mod^+ 2^n = 2^(n-1) ∧ z ≥ 0>
⟨proof⟩
```

## 5.2 Montgomery multiplication with a known constant

Montgomery reduction of the product `a * b` can be simplified in case one operand, say `b`, is known. In this case, one can precompute the 'twist' `b_tw` $\equiv$ `b` $\cdot$ $N^{-1}$ `(mod R)` and compute the Montgomery multiplication as follows:

```
definition mont_mul_sub_signed (<mont_sub^± [_,_]⟨_,_⟩>) where
  <mont_sub^± [N, n]⟨a, b⟩ ≡
     b_tw should be precomputed in practice
     (let b_tw = (b * (N^-1 mod 2^n)) mod 2^n in
        (a * b - ((a * b_tw) mod^± 2^n) * N) div 2^n)>

definition mont_mul_add_signed (<mont_add^± [_,_]⟨_,_⟩>) where
  <mont_add^± [N, n]⟨a, b⟩ ≡
     b_tw should be precomputed in practice
     (let b_tw = (b * ((- (N^-1 mod 2^n)) mod 2^n)) mod 2^n in
        (a * b + ((a * b_tw) mod^± 2^n) * N) div 2^n)>

definition mont_mul_add_unsigned (<mont_add^+ [_,_]⟨_,_⟩>) where
  <mont_add^+ [N, n]⟨a, b⟩ ≡
     b_tw should be precomputed in practice
     (let b_tw = (b * ((- (N^-1 mod 2^n)) mod 2^n)) mod 2^n in
        (a * b + (a * b_tw mod^+ 2^n) * N) div 2^n)>
```

The multiplication operator coincides with the reduction operator applied to `a * b`: by associativity, `a * b_tw mod R = a * b * T mod R`.

```
lemma (in OddModulus) mont_mul_eq_red:
  shows <mont_sub^± [N, n]⟨a, b⟩ = mont_sub^± [N, n] (a * b)>
    and <mont_add^± [N, n]⟨a, b⟩ = mont_add^± [N, n] (a * b)>
    and <mont_add^+ [N, n]⟨a, b⟩ = mont_add^+ [N, n] (a * b)>
⟨proof⟩
```

### 5.2.1 Correctness and bounds

Since Montgomery multiplication is the same as Montgomery reduction of the product, the bounds from the previous section readily carry over to Montgomery multiplication:

```
theorem (in OddModulus) mont_mul_correct:
  shows <(mont_sub^± [N, n]⟨a, b⟩ * 2^n) mod N = (a * b) mod N>
    and <(mont_add^± [N, n]⟨a, b⟩ * 2^n) mod N = (a * b) mod N>
    and <(mont_add^+ [N, n]⟨a, b⟩ * 2^n) mod N = (a * b) mod N>
⟨proof⟩
theorem (in OddModulus) mont_mul_bound:
  shows <|(mont_sub^± [N, n]⟨a, b⟩)_Q| ≤ |a * b| /_Q 2^n + N_Q / 2>
    and <|(mont_add^± [N, n]⟨a, b⟩)_Q| ≤ |a * b| /_Q 2^n + N_Q / 2>
    and <|(mont_add^+ [N, n]⟨a, b⟩)_Q| ≤ |a * b| /_Q 2^n + N_Q - N /_Q 2^n>
⟨proof⟩
```

# 6 Barrett reduction and multiplication

This chapter introduces the abstract Barrett reduction and multiplication operators together with an approximation-error bound. Output absolute bounds are deferred to Section 7, where they follow sharply from the Barrett–Montgomery bridge.

## 6.1 Barrett reduction

Barrett reduction [6, §2.4.1] replaces division by the modulus `N` with a multiply-shift: with `R = 2^n` and `N < R` and precomputed `⟦R/N⟧`, the quotient `z / N` is approximated as `z·⟦R/N⟧/R`, and the residue recovered as `z - N·⌊z·⟦R/N⟧/R⌋`.

There are two variants, differing only in how the outer quotient is rounded: the *signed* variant applies round-to-nearest $\lfloor\cdot\rceil$, recovering `z mod ⟦⌊·⌉⟧ N`; the *unsigned* variant applies floor $\lfloor\cdot\rfloor$, recovering `z mod ⟦⌊·⌋⟧ N`. Both are parameterised by the inner approximation `f` of `R /Q N`.

```
definition barrett_red_signed (‹bar± ⟦_,_,_⟧›) where
  ‹bar±⟦N, n, f⟧ z ≡ z - N * ⌊z * ⟦2^n /Q N⟧ /Q 2^n⌉› for f ("⟦_⟧")

definition barrett_red_unsigned (‹bar+ ⟦_,_,_⟧›) where
  ‹bar+⟦N, n, f⟧ z ≡ z - N * ⌊z * ⟦2^n /Q N⟧ /Q 2^n⌋› for f ("⟦_⟧")
```

We bundle the standing assumptions of this and the next chapter into a locale `BarrettContext = OddModulus` (a positive odd modulus `N` and a splitting exponent `n` with `R = 2^n` and `N < R`) plus `IntegerApproximation` (an integer approximation `⟦_⟧` of the rationals).

```
locale BarrettContext = OddModulus + IntegerApproximation
```

[6, Fact 2] provides a coarse bound for the approximation error between the signed variant and the canonical signed residue:

```
theorem (in BarrettContext) barrett_red_signed_fact2:
  shows ‹|z mod± N - bar±⟦N, n, f⟧ z| ≤ N * ⌈|z| /Q R⌉›
⟨proof⟩
```

## 6.2 Barrett multiplication

In NTTs one operand of each modular multiplication is a fixed twiddle. Barrett multiplication [6, §3.1.2] folds `b` into the magic constant `⟦b·R/N⟧`, collapsing reduction to a single multiply-shift `a·⟦b·R/N⟧/R` with representative `a·b - N·⌊a·⟦b·R/N⟧/R⌋`.

```
definition barrett_mul_signed (‹barM± ⟦_,_,_⟧⟨_,_⟩›) where
  ‹barM± ⟦N, n, f⟧⟨a, b⟩ ≡ a * b - N * ⌊a * ⟦b * 2^n /Q N⟧ /Q 2^n⌉› for f ("⟦_⟧")

definition barrett_mul_unsigned (‹barM+ ⟦_,_,_⟧⟨_,_⟩›) where
  ‹barM+ ⟦N, n, f⟧⟨a, b⟩ ≡ a * b - N * ⌊a * ⟦b * 2^n /Q N⟧ /Q 2^n⌋› for f ("⟦_⟧")
```

Barrett reduction is recovered as Barrett multiplication with `b = 1`:

```
lemma barrett_red_as_mul:
  shows ‹bar±⟦N, n, f⟧ z = barM± ⟦N, n, f⟧⟨z, 1⟩›
    and ‹bar+⟦N, n, f⟧ z = barM+ ⟦N, n, f⟧⟨z, 1⟩›
  ⟨proof⟩
```

## 6.3 Refined Barrett reduction

The output quality of Barrett reduction improves as the radix $2^n$ grows, so one is naturally led to consider radices that fall *between* word boundaries. This section establishes a way to compute such reductions in a two-step fashion: a word-sized truncating high-half multiply, followed by a rounding right shift on the lane-width word.

```
definition barrett_red_refined :: ‹int ⇒ nat ⇒ nat ⇒ int ⇒ int ⇒ int› where
  ‹barrett_red_refined N n α V z ≡
     let t = (2 * z * V) div 2^n;
         q = ⌊t /ℚ 2^α⌋
     in z - q * N›
```

Key identity: for an integer `q`, the value `q /ℚ 2^k + 1 /ℚ 2` has denominator $2^k$, so adding a perturbation less than `1/2^k` cannot cross an integer boundary.

```
lemma floor_then_round_shift:
  fixes x :: int and n k :: nat
  assumes ‹k ≥ 1›
  shows ‹⌊(x div 2^n)ℚ / (2^k)ℚ⌋ = ⌊x /ℚ 2^(n+k)⌋›
```
⟨*proof*⟩

The refined kernel equals Barrett reduction at the effective radix $2^{n + \alpha - 1}$.

```
lemma barrett_red_refined_eq:
  assumes ‹α ≥ 1›
  assumes ‹V = ⌊2^(n+α-1) /ℚ N⌋›
  shows ‹barrett_red_refined N n α V z = bar±⟦N, n+α-1, ⌊·⌋⟧ z›
```
⟨*proof*⟩

As in the standard Barrett section, output bounds are deferred to the Barrett–Montgomery bridge chapter (Section 7); the instantiation as the Neon `SQDMULH`/`SRSHR`/`MLS` sequence ([6, Algorithm 11]) is deferred to Section 13.

# 7 Barrett ↔ Montgomery bridge

This chapter establishes the equivalence of Barrett and Montgomery reduction and multiplication on a suitably twisted input, and derives the corresponding output absolute bounds for Barrett from the Montgomery ones.

Throughout the chapter, we fix an integer approximation `f=⟦_⟧`, an odd modulus `N`, and an exponent `n` with `N < 2^n`. We abbreviate `R = 2^n`. This will be indicated by the `(in BarrettContext)` below.

## 7.1 Bridge: Barrett reduction and multiplication as Montgomery

Barrett reduction returns a new representative of `z mod N`, while Montgomery reduction returns a representative of `z * R⁻¹ mod N`. They cannot literally be equal: Any equivalence must absorb a factor of `R` somewhere. The equivalence identities use a `R mod ⟦f⟧ N`-twist on the Montgomery input. The most direct proof is obtained by carefully unravelling the definitions; Section 10 offers a conceptual explanation.

```
theorem (in BarrettContext) barrett_montgomery_bridge:
  shows ‹bar±⟦N, n, f⟧ z = mont_add±⟦N, n⟧ (z * (R mod⟦f⟧ N))›
    and ‹bar+⟦N, n, f⟧ z = mont_add+⟦N, n⟧ (z * (R mod⟦f⟧ N))›
    and ‹barM± ⟦N, n, f⟧⟨a, b⟩ = mont_add±⟦N, n⟧ (a * ((b * R) mod⟦f⟧ N))›
    and ‹barM+ ⟦N, n, f⟧⟨a, b⟩ = mont_add+⟦N, n⟧ (a * ((b * R) mod⟦f⟧ N))›
```
⟨*proof*⟩

## 7.2 Bounds for Barrett reduction and multiplication

We now derive absolute bounds for Barrett reduction and multiplication by combining the bridge lemmas above with the absolute bounds for additive Montgomery reduction and multiplication.

It should be noted that the signed Barrett reductions/multiplications satisfy a tighter output bound than the unsigned variants, as for signed vs. unsigned Montgomery reduction.

```
theorem (in BarrettContext) barrett_montgomery_bounds:
  shows <|(bar±⟦N, n, f⟧ z)_Q| ≤ |z| * |R mod⟦f⟧ N| /_Q R + N /_Q 2>
  and <|(bar⁺⟦N, n, f⟧ z)_Q| ≤ |z| * |R mod⟦f⟧ N| /_Q R + N_Q - N /_Q R>
  and <|(barM± ⟦N, n, f⟧⟨a, b⟩)_Q| ≤ |a| * |(b * R) mod⟦f⟧ N| /_Q R + N /_Q 2>
  and <|(barM⁺ ⟦N, n, f⟧⟨a, b⟩)_Q| ≤ |a| * |(b * R) mod⟦f⟧ N| /_Q R + N_Q - N /_Q R>
⟨proof⟩
```

The bounds above are stated in terms of $|2^{n} \bmod ⟦f⟧\ N|$, the magnitude of the residue of $2^{n}$ under $⟦_⟧$-rounding. This residue equals `N_Q * ε (f, R /_Q N)` for reduction and `N_Q * ε (f, b * R /_Q N)` for multiplication, hence giving the following pointwise bounds:

```
theorem (in BarrettContext) barrett_montgomery_bounds_eps:
  shows <|(bar±⟦N, n, f⟧ z)_Q| ≤ |z|_Q * (N_Q * ε(f, R /_Q N)) / R_Q + N_Q / 2>
  and <|(bar⁺⟦N, n, f⟧ z)_Q| ≤ |z|_Q * (N_Q * ε(f, R /_Q N)) / R_Q + N_Q - N /_Q R>
  and <|(barM± ⟦N, n, f⟧⟨a, b⟩)_Q| ≤ |a|_Q * (N_Q * ε(f, (b * R) /_Q N)) / R_Q + N_Q / 2>
  and <|(barM⁺ ⟦N, n, f⟧⟨a, b⟩)_Q| ≤ |a|_Q * (N_Q * ε(f, (b * R) /_Q N)) / R_Q + N_Q - N /_Q
R>
⟨proof⟩
```

When the input magnitude is at most `2^(n-1) = R /_Q 2`, the slope term is at most half of `N_Q * ε (f, x)`, yielding the following narrowed bounds:

```
theorem (in BarrettContext) barrett_bound_eps_narrow:
  shows <|z| ≤ 2^(n-1) ⟹ |(bar±⟦N, n, f⟧ z)_Q| ≤ N_Q * (ε(f, R /_Q N) + 1)/2>
    and <|z| ≤ 2^(n-1) ⟹ |(bar⁺⟦N, n, f⟧ z)_Q| ≤ N_Q * (ε(f, R /_Q N) + 2)/2 - N/_QR>
    and <|a| ≤ 2^(n-1) ⟹ |(barM± ⟦N, n, f⟧⟨a, b⟩)_Q| ≤ N_Q * (ε(f, (b*R)/_Q N) + 1)/2>
    and <|a| ≤ 2^(n-1) ⟹ |(barM⁺ ⟦N, n, f⟧⟨a, b⟩)_Q| ≤ N_Q * (ε(f, (b*R)/_Q N) + 2)/2
- N/_QR>
⟨proof⟩
```

In Section 8 we will demonstrate by example that the right hand sides `N_Q * (ε (f, 2^n /_Q N) + 1) / 2` etc. of `BarrettContext.barrett_bound_eps_narrow` are *tight*.

The pointwise version of `BarrettContext.barrett_bound_eps_narrow` above lets us specialise to the three cases of practical interest — signed Barrett multiplication with round-to-nearest-even and with round-to-nearest, and unsigned Barrett multiplication with round-to-nearest. When the second operand `b` satisfies `b ≠ 0` and `|b| < N`, then `b` is not divisible by `N`, and since `N` is odd, neither is `b * R` nor `2 * b * R`. So the pointwise rounding error at `b * R /_Q N` is strictly below the uniform quality, yielding strict output bounds.

```
theorem (in OddModulus) barrett_mul_narrow:
  assumes <|a| ≤ 2^(n-1)>
      and <|b| < N> and <b ≠ 0>
  shows <|(barM± ⟦N, n, ⌊·⌉_2⟧⟨a, b⟩)_Q| < N_Q>
    and <|(barM± ⟦N, n, ⌊·⌉⟧⟨a, b⟩)_Q|  < 3 * N /_Q 4>
    and <|(barM⁺ ⟦N, n, ⌊·⌉⟧⟨a, b⟩)_Q|  < 5 * N /_Q 4>
⟨proof⟩
```

Specializing at `b = 1`, we obtain the same bounds for Barrett reduction:

```
theorem (in OddModulus) barrett_red_narrow:
  assumes <|z| ≤ 2^(n-1)>
  shows <|(bar±⟦N, n, ⌊·⌉_2⟧ z)_Q| < N_Q>
    and <|(bar±⟦N, n, ⌊·⌉⟧ z)_Q| < 3 * N /_Q 4>
    and <|(bar⁺⟦N, n, ⌊·⌉⟧ z)_Q| < 5 * N /_Q 4>
  ⟨proof⟩
```

### 7.3 Bounds for refined Barrett reduction

We have seen that the output quality of Barrett reduction improves as `n` grows. For refined Barrett reduction, the goal is to choose `n` large enough that the output is in fact *signed canonical*. For practical purposes growing `n` alone is not sufficient: one also needs to take into account the quality of the approximation ⟦R /Q N⟧.

```
lemma (in BarrettContext) barrett_montgomery_bounds_quality_pow:
  assumes ‹ε(f, R /Q N) ≤ 1/2^δ›
  shows ‹|(bar±⟦N, n, f⟧ z)Q| ≤ |z|Q * NQ / (RQ * 2^δ) + NQ / 2›
    and ‹|(bar+⟦N, n, f⟧ z)Q| ≤ |z|Q * NQ / (RQ * 2^δ) + NQ - N /Q R›
⟨proof⟩
```

If we additionally specialise the input `z` to be bounded by $2^{n - 1 - \gamma}$ — that is, we assume $\gamma$ bits of slack for $0 \leq \gamma$ — we obtain the following bound:

```
lemma (in BarrettContext) barrett_red_signed_bound_slack:
  assumes ‹ε(f, R /Q N) ≤ 1/2^δ›
      and ‹|z| ≤ 2^(n-1-γ)›
      and ‹γ ≤ n-1›
  shows ‹|(bar±⟦N, n, f⟧ z)Q| ≤ NQ / 2^(γ+δ+1) + NQ / 2›
⟨proof⟩
```

Finally, we obtain the following statement about canonicity of refined Barrett reduction. Observe how approximation quality and restricted input bounds contribute equally and can be balanced on a case-by-case basis.

```
corollary (in BarrettContext) barrett_red_signed_canonical:
  assumes eps_le: ‹ε(f, R /Q N) ≤ 1/2^δ›
      and z_le:   ‹|z| ≤ 2^(n-1-γ)›
      and γ_le:   ‹γ ≤ n-1›
      and N_lt:   ‹N < 2^(γ+δ)›
  shows ‹|(bar±⟦N, n, f⟧ z)Q| < N /Q 2›
⟨proof⟩
```

Signed-canonicity can also be interpreted as saying the Barrett quotient ⌊z * ⟦R /Q N⟧ /Q R⌉ is *exact*, i.e. equal to ⌊z /Q N⌉:

```
corollary (in BarrettContext) barrett_red_signed_exact_iff:
  shows ‹|(bar±⟦N, n, f⟧ z)Q| < N /Q 2 ⟷ ⌊z * ⟦R /Q N⟧ /Q R⌉ = ⌊z /Q N⌉›
⟨proof⟩
```

Combined with `barrett_red_signed_canonical`, this gives a sufficient condition for the Barrett quotient to be exact, accounting for approximation quality and input bound:

```
corollary (in BarrettContext) barrett_red_signed_exact:
  assumes eps_le: ‹ε(f, R /Q N) ≤ 1/2^δ›
      and z_le:   ‹|z| ≤ 2^(n-1-γ)›
      and γ_le:   ‹γ ≤ n-1›
      and N_lt:   ‹N < 2^(γ+δ)›
  shows ‹⌊z * ⟦R /Q N⟧ /Q R⌉ = ⌊z /Q N⌉›
  ⟨proof⟩
```

## 8 Quality of the Barrett bounds

The purpose of this chapter is to empirically evaluate the quality of the bounds of `barrett_bound_eps_narrow`. In particular, we will exhibit concrete instantiations of `BarrettContext`

demonstrating that barrett_bound_eps_narrow is tight and cannot be improved further (for generic moduli).

Throughout, we work in the context of BarrettContext:

```
context BarrettContext
begin
```

## 8.1 Empirical maxima and analytic bounds

We define, in BarrettContext, the maximum achieved Barrett magnitude over the input range {- $2^{n-1}$..$2^{n-1}$} (signed and unsigned), and the matching analytic upper bound recorded in Section 7.

```
definition ‹bar_signed_max ≡  MAX z∈{-(2^(n-1))..2^(n-1)}. |bar± ⟦N,n,f⟧ z|›
definition ‹bar_unsigned_max ≡ MAX z∈{-(2^(n-1))..2^(n-1)}. |bar+ ⟦N,n,f⟧ z|›
definition ‹bar_signed_bound ≡ ⌊N_Q * (ε(f, R /_Q N) + 1) / 2⌋›
definition ‹bar_unsigned_bound ≡ ⌊N_Q * (ε(f, R /_Q N) + 2) / 2 - N /_Q R⌋›
```

A pointwise integer-form upper bound on |bar± ⟦N,n,f⟧ z|, obtained by floor-applying the rational bound barrett_bound_eps_narrow at the canonical residue ε (f, R /_Q N). Used as the early-termination predicate for the exhaustive sweeps below.

```
definition ‹bar_signed_bnd_at z ≡ ⌊|z|_Q * (N_Q * ε(f, R /_Q N)) / R_Q + N_Q / 2⌋›
definition ‹bar_unsigned_bnd_at z ≡ ⌊|z|_Q * (N_Q * ε(f, R /_Q N)) / R_Q + N_Q - N /_Q R⌋›
```

The following is merely a restatement of barrett_montgomery_bounds_eps:

```
lemma bar_bnd_at_correct_mono:
  shows ‹|bar±⟦N, n, f⟧ z| ≤ bar_signed_bnd_at |z|›
    and ‹|bar+⟦N, n, f⟧ z| ≤ bar_unsigned_bnd_at |z|›
    and ‹0 ≤ a ⟹ a ≤ b ⟹ bar_signed_bnd_at a ≤ bar_signed_bnd_at b›
    and ‹0 ≤ a ⟹ a ≤ b ⟹ bar_unsigned_bnd_at a ≤ bar_unsigned_bnd_at b›
⟨proof⟩
```

## 8.2 Computable maxima via exhaustive sweep with early termination

Naive computation of MAX z∈{- $2^{n-1}$..$2^{n-1}$}. |bar± ⟦N,n,f⟧ z| is impractical for larger values of N and n. Instead, we use the following stand-alone helper computing the maximum absolute value of an integer function f over a symmetric interval {- max_abs..max_abs} with the help of an *early termination* condition: every K inward steps, the recursion checks whether a user-supplied bound f_bnd on the un-swept inner radius is already dominated by the current candidate maximum, and if so returns immediately. The recursion sweeps inward from max_abs toward 0, processing the symmetric pair (g i, g (- i)) per step.

```
end ...  temporarily leaving the context of BarrettContext

function max_abs_with_bound_core ::
  ‹(int ⇒ int) ⇒ (int ⇒ int) ⇒ nat ⇒ nat ⇒ int ⇒ int ⇒ int›
where
  Base case: no steps left; return the running maximum.
  ‹max_abs_with_bound_core f f_bnd 0       K i cur = cur›
| ‹max_abs_with_bound_core f f_bnd (Suc k) K i cur =
   (let cur' = max (max cur |f i|) |f (-i)|;  update with symmetric pair
        tail_radius = i - 1;
        checkpoint = (K = 0 ∨ k mod K = 0);    consult early-exit every K steps
        can_skip = (checkpoint ∧ tail_radius ≥ 0 ∧ f_bnd tail_radius ≤ cur')
    in if can_skip then cur'
       else max_abs_with_bound_core f f_bnd k K tail_radius cur')›
```

⟨*proof*⟩

```
definition ‹max_abs_with_bound f f_bnd max_abs K ≡
  Sweep [-max_abs..max_abs] from the outside in, starting with cur = 0.
  max_abs_with_bound_core f f_bnd (max_abs + 1) K (int max_abs) 0›
```

The following lemma shows that for a correct early-termination function, the function `max_abs_with_bound` indeed computes a maximum absolute value over a symmetric interval:

```
lemma max_abs_with_bound_eq:
  assumes bnd_correct: ‹⋀x. |f x| ≤ f_bnd |x|›
      and bnd_mono:    ‹⋀a b. 0 ≤ a ⟹ a ≤ b ⟹ f_bnd a ≤ f_bnd b›
  shows ‹max_abs_with_bound f f_bnd max_abs K =
           (MAX z∈{-(int max_abs) .. int max_abs}. |f z|)›
```

⟨*proof*⟩

```
context BarrettContext begin ...  re-entering BarrettContext
```

Applying this to the Barrett context, we obtain the following efficient formula for computing the maximum absolute value of a Barrett reduction in the interval `{- 2^n - 1..2^n - 1}`. We will register those lemmas as *code equations* when evaluating `bar_signed_max` and `bar_unsigned_max` in concrete instantiations of `BarrettContext`.

```
corollary bar_signed_max_eq:
  ‹bar_signed_max = max_abs_with_bound bar±⟦N, n, f⟧ bar_signed_bnd_at (2^(n-1)) K›
```

⟨*proof*⟩

```
corollary bar_unsigned_max_eq:
  ‹bar_unsigned_max = max_abs_with_bound bar+⟦N, n, f⟧ bar_unsigned_bnd_at (2^(n-1))
K›
```

⟨*proof*⟩

For each sample we instantiate `BarrettContext` with concrete `(N, n, f)` and tag `bar_unsigned_max_eq` and `bar_signed_max_eq` as code equations so that `value` evaluates `bar_*_max`-constants by an exhaustive sweep with early termination using `max_abs_with_bound`.

```
end Leaving the context of BarrettContext
```

## 8.3 Example 1: `N = 97`, `n = 8` (prime, illustrative 8-bit modulus)

We start with a small prime `N = 97` and `n = 8` to illustrate the methodology on a modulus where the bounds and achieved maxima are easy to inspect by hand.

Here and in the other examples, we use **global_interpretation** to instantiate `BarrettContext` with concrete `(N, n, f)`. The command discharges the locale's assumptions once and propagates every locale conclusion to the global namespace; the qualifier (here `BC_97_8`) prefixes every fact and definition introduced by the locale, preventing accidental shadowing.

The **defines** clause has a special role: each entry binds a global constant to a locale-internal definition, so e.g. `bar_signed_max_97_8` becomes a top-level constant whose definitional equation is the locale definition specialised to the chosen `(N, n, f)`. Without this clause we would still get the qualified name `BC_97_8.bar_signed_max`, but it would not unfold during code generation: the code generator treats locale constants as opaque.

```
global_interpretation BC_97_8: BarrettContext 97 8 round
  defines bar_signed_max_97_8      = ‹BC_97_8.bar_signed_max›
      and bar_unsigned_max_97_8    = ‹BC_97_8.bar_unsigned_max›
      and bar_signed_bound_97_8    = ‹BC_97_8.bar_signed_bound›
      and bar_unsigned_bound_97_8  = ‹BC_97_8.bar_unsigned_bound›
      and bar_signed_bnd_at_97_8   = ‹BC_97_8.bar_signed_bnd_at›
      and bar_unsigned_bnd_at_97_8 = ‹BC_97_8.bar_unsigned_bnd_at›
```

⟨*proof*⟩

We can now evaluate the theoretical bounds against the attained maxima:

```
lemma BC_97_8_bounds:
    Signed theoretical bound vs attained maximum
  shows ‹bar_signed_max_97_8 = 66›
    and ‹bar_signed_bound_97_8 = 66›
    Unsigned theoretical bound vs attained maximum
    and ‹bar_unsigned_max_97_8 = 108›
    and ‹bar_unsigned_bound_97_8 = 114›
  ⟨proof⟩
```

The signed theoretical bound is achieved exactly. The unsigned bound is not attained, but with ≈ 5.2% of slack.

## 8.4 Example 2: N = 3329, n = 16 (ML-KEM modulus)

In this section, we look at N = 3329, the modulus underlying ML-KEM [14], together with n = 16 (ML-KEM is computed in 16-bit arithmetic). As before, we first instantiate the Barrett context:

```
global_interpretation BC_3329_16: BarrettContext 3329 16 round
  defines bar_signed_max_3329_16      = ‹BC_3329_16.bar_signed_max›
      and bar_unsigned_max_3329_16    = ‹BC_3329_16.bar_unsigned_max›
      and bar_signed_bound_3329_16    = ‹BC_3329_16.bar_signed_bound›
      and bar_unsigned_bound_3329_16  = ‹BC_3329_16.bar_unsigned_bound›
      and bar_signed_bnd_at_3329_16   = ‹BC_3329_16.bar_signed_bnd_at›
      and bar_unsigned_bnd_at_3329_16 = ‹BC_3329_16.bar_unsigned_bnd_at›
  ⟨proof⟩
```

We can now evaluate the theoretical bounds against the attained maxima:

```
lemma BC_3329_16_bounds:
    Signed theoretical bound vs attained maximum
  shows ‹bar_signed_max_3329_16 = 2160›
    and ‹bar_signed_bound_3329_16 = 2186›
    Unsigned theoretical bound vs attained maximum
    and ‹bar_unsigned_max_3329_16 = 3798›
    and ‹bar_unsigned_bound_3329_16 = 3850›
  ⟨proof⟩
```

Neither the signed nor the unsigned theoretical bound is attained, but with ≈ 1.1% (signed) and ≈ 1.3% (unsigned) of slack.

## 8.5 Example 3: N = 8380417, n = 32 (ML-DSA modulus)

In this section, we look at N = 8380417, the modulus underlying ML-DSA [13], together with n = 32 (ML-DSA is computed in 32-bit arithmetic). As before, we first instantiate the Barrett context:

```
global_interpretation BC_8380417_32: BarrettContext 8380417 32 round
  defines bar_signed_max_8380417_32      = ‹BC_8380417_32.bar_signed_max›
      and bar_unsigned_max_8380417_32    = ‹BC_8380417_32.bar_unsigned_max›
      and bar_signed_bound_8380417_32    = ‹BC_8380417_32.bar_signed_bound›
      and bar_unsigned_bound_8380417_32  = ‹BC_8380417_32.bar_unsigned_bound›
      and bar_signed_bnd_at_8380417_32   = ‹BC_8380417_32.bar_signed_bnd_at›
      and bar_unsigned_bnd_at_8380417_32 = ‹BC_8380417_32.bar_unsigned_bnd_at›
```

⟨*proof*⟩

We can now evaluate the theoretical bounds against the attained maxima:

```
lemma BC_8380417_32_bounds:
    Signed theoretical bound vs attained maximum
  shows ‹bar_signed_max_8380417_32 = 6283521›
    and ‹bar_signed_bound_8380417_32 = 6283521›
    Unsigned theoretical bound vs attained maximum
    and ‹bar_unsigned_max_8380417_32 = 10469648›
    and ‹bar_unsigned_bound_8380417_32 = 10473729›
⟨proof⟩
```

The signed theoretical bound is achieved exactly. The unsigned bound is not attained, but with $\approx$ `0.038%` of slack.

# 9 Montgomery multiplication via doubling and rounding

Not every architecture exposes a plain truncating high-half multiply `MULH` of the kind used in standard Montgomery multiplication. Fixed-point arithmetic ISAs commonly provide *doubling* or *rounding* high-half multiplies instead — on AArch64, `SQDMULH` (returns $\lfloor$`2 * a * b /`$_Q$ `R`$\rfloor$ for `a`, `b`, `R`) and `SQRDMULH` (returns $\lfloor$`2 * a * b /`$_Q$ `R`$\rceil$ under the same typing). [6, Algorithms 7 and 8, §3.3] rebuild Montgomery multiplication around these two operations.

## 9.1 Montgomery Multiplication via Doubling

The idea behind Montgomery Multiplication with Doubling is simple: use doubling high-half multiplies, and compensate for the doubling through a halving subtract — which too is common in fixed-point arithmetic ISAs:

```
definition ‹mont_mul_doubling N n T a b ≡
     (let R = (2::int)^n;
          z = (2 * (a * b)) div R;
          k = ((a * b * T) mod± R);
          c = (2 * (k * N)) div R
      in (z - c) div 2)›
```

Algorithm 7 computes $\texttt{mont}_{sub}{}^{\pm}$ `⟦N,n⟧ (a * b)` for arbitrary `a`, `b`:

```
theorem (in OddModulus) mont_mul_doubling_eq:
  assumes T_inv: ‹(N * T) mod 2^n = 1 mod 2^n›
  shows ‹mont_doubling N n T a b = mont_sub±⟦N, n⟧ (a * b)›
⟨proof⟩
```

## 9.2 Montgomery Multiplication via Rounding

The idea behind Montgomery Multiplication via Rounding is to express the carry from the low half of the standard Montgomery step as the joint *rounding carry* of two `SQRDMULH` operations: each contributes a `1 / 2` on its own, and combining them recovers the missing high-bit. The construction is exact for all `a`, `b` except at one boundary case (low half exactly $2^{n-1}$) where both `SQRDMULH`s round up; the trailing `SHSUB` halves the doubled output and absorbs the boundary error.

Note: [6] did not recognise this corrective effect of `SHSUB` and unnecessarily assumed one operand to be odd; we drop that assumption. The boundary case becomes a genuine off-by-one in the doubled variant of the next section (no `SHSUB`), where assuming it away is the correct fix.

```
definition ‹mont_mul_rounding N n T a b ≡
     (let z = ⌊(2 * (a * b)) /ℚ 2^n⌉;
          k = ((a * b * T) mod± 2^n);
          c = ⌊(2 * (k * N)) /ℚ 2^n⌉
      in (z + c) div 2)›
```

Under the standard Montgomery side-conditions, Algorithm 8 computes exactly $\texttt{mont}_{add}^{\pm}$ ⟦N,n⟧ (a * b) for all a, b. The parity assumption of [6, Algorithm 8] that either a or b be odd is unnecessary.

```
theorem (in OddModulus) mont_mul_rounding_eq:
  assumes T_inv: ‹(N * T) mod 2^n = (- 1) mod 2^n›
  shows ‹mont_mul_rounding N n T a b = mont_add^±⟦N, n⟧ (a * b)›
⟨proof⟩
```

## 9.3 Montgomery Multiplication via Rounding, no halving

A variant of Algorithm 8 that drops the trailing SHSUB (the final div 2) and fuses the high-half product with the correction into a single rounding multiply-accumulate SQRDMLAH. Without the halving step there is no longer a 1 / 2 to absorb the {0, 1} ambiguity at the boundary 2 * a * b mod $2^n$ = $2^{n-1}$, so we exclude this case by hypothesis. Under that hypothesis the kernel computes exactly 2 * $\texttt{mont}_{add}^{\pm}$ ⟦N,n⟧ (a * b). In implementation contexts, this hypothesis can for example be discharged by odd b together with the input bound |a| < R / 4, matching the "a odd or b odd" assumption of [6, Algorithm 8]: odd b alone forces 2 * a * b mod $2^n$ ≠ $2^{n-1}$ when |a| < $2^{n-2}$.

```
definition ‹mont_mul_rounding_doubled_int N n T a b ≡
     (let z = ⌊2 * a * b /ℚ 2^n⌉;
          k = a * b * T mod± 2^n;
          c = ⌊2 * k * N /ℚ 2^n⌉
      in z + c)›
```

The doubled-rounding kernel computes exactly twice the additive signed Montgomery reduction. The hypothesis 2 * a * b mod $2^n$ ≠ $2^{n-1}$ rules out the only configuration in which the two SQRDMULH roundings would together contribute 2 rather than 1.

```
theorem (in OddModulus) mont_mul_rounding_doubled_eq:
  assumes T_inv: ‹(N * T) mod 2^n = (- 1) mod 2^n›
      and parity: ‹2 * a * b mod 2^n ≠ 2^(n-1)›
  shows ‹mont_mul_rounding_doubled_int N n T a b = 2 * mont_add^±⟦N, n⟧ (a * b)›
⟨proof⟩
```

# 10 A conceptual view of the Barrett–Montgomery bridge

The Section 7 bridge was proved by direct algebraic manipulation. This chapter recasts it as a corollary of a (trivial) duality between Euclidean and 2-adic *rounding* of a rational number, plus some arithmetic properties of the 2-adic side. The recasting was first sketched in the talk [7] presenting [6] at CHES 2022.

Rounding can be described as the process of expressing a continuum as the sum of something discrete and something small (relatively compact); the choice of absolute value on ℚ determines the roles. The Euclidean absolute value gives $\lfloor\cdot\rceil_{[\infty]}$ : ℚ → ℤ, with error in [-½, ½): this is the rounding underlying Barrett. The 2-adic absolute value gives $\lfloor\cdot\rceil_{[2]}$ : ℚ → ℤ[½] ∩ [-½, ½), with error in the odd-denominator rationals $\mathbb{Z}_{(2)}$ ∩ ℚ: this is the rounding underlying Montgomery. The duality is the swap: each rounding's target sits inside the other's error set. On the dyadic rationals ℤ[½] the two roundings sum to the

identity. This, together with basic arithmetic properties of 2-adic rounding, establishes the Barrett–Montgomery bridge.

We treat only the signed variant; the unsigned variant follows the same recipe with floor-rounding.

## 10.1 Barrett arithmetic via Euclidean rounding

For the Euclidean metric, the rounding target is $\mathbb{Z}$ and the error lies in `[-½, ½)`. The half-open right end is forced by HOL's $\lfloor\cdot\rceil$ (round half-up): at `q = ½` the error is `-½`.

```
definition ‹euclid_target q ≡ snd (quotient_of q) = 1›
definition ‹euclid_error q ≡ - (1/2) ≤ q ∧ q < 1/2›
```

Every rational has a unique decomposition with respect to Euclidean rounding:

```
theorem euclidean_decomposition_unique:
  shows ‹∃!m. euclid_target m ∧ euclid_error (q - m)›
⟨proof⟩

definition round_E (‹⌊_⌉[∞]›) where ‹⌊q⌉[∞] ≡ THE m. euclid_target m ∧ euclid_error
(q - m)›
```

Of course, `[∞]` is just ordinary rounding:

```
lemma round_E_eq_round: ‹⌊q⌉[∞] = ⌊q⌉Q›
  ⟨proof⟩
```

Finally, Barrett reduction is — by definition — based on Euclidean rounding of `z * f (R /Q N) /Q R` scaled by `N` and subtracted from `z`.

```
lemma (in BarrettContext) bar_signed_as_round_E:
  shows ‹(bar±⟦N, n, f⟧ z)Q = zQ - NQ * ⌊z * ⟦2^n /Q N⟧ /Q 2^n⌉[∞]›
  ⟨proof⟩
```

## 10.2 Montgomery arithmetic via 2-adic rounding

For the 2-adic metric, the rounding target is $\mathbb{Z}[½] \cap$ `[-½, ½)` (a transversal of $\mathbb{Q}_2$ / $\mathbb{Z}_2$) and the error set is the odd-denominator rationals $\mathbb{Z}_{(2)} \cap \mathbb{Q}$. Compared to the Euclidean side the targets and errors swap: $\mathbb{Z}$ now sits inside the error set, and `[-½, ½)` now contains the target. We need a little more arithmetic groundwork than the Euclidean side.

```
definition Dyadic ("ℤ[1'/2]")
  where ‹ℤ[1/2] ≡ {q. is_power_of_two (snd (quotient_of q))}›

definition TwoAdicIntegral ("ℤ[2]")
  where ‹ℤ[2] ≡ {q. odd (snd (quotient_of q))}›
```

We will use the following congruence relation to embed modular arithmetic modulo `R` into the intermediate subring $\mathbb{Z}_{[2]}$ between $\mathbb{Z}$ and $\mathbb{Q}$. Note that it does not degenerate because we left `2` non-invertible in $\mathbb{Z}_{[2]}$.

```
abbreviation cong_R_two_adic :: "rat ⇒ rat ⇒ int ⇒ bool"
      ("[_ = _] '( mod _ ℤ[2]')" [50, 50, 50] 50)
      where "[x = y] (mod R ℤ[2]) ≡ ((x - y) / RQ) ∈ ℤ[2]"
```

Our 2-adic rounding target is the (2-adically) discrete set of dyadic numbers in the interval `{- 1 / 2..<1 / 2}`:

```
definition TwoAdicTarget ("'[-½, ½')[1'/2]") where
  ‹[-½, ½)[1/2] ≡ ℤ[1/2] ∩ {-(1/2)..<(1/2)}›
```

A rational with odd denominator (in lowest terms) lies in $\mathbb{Z}_{[2]}$.

```
lemma two_adic_integral_div_odd:
  assumes ‹N > 0› and ‹odd N›
  shows ‹a /ℚ N ∈ ℤ[2]›
⟨proof⟩
```

For every rational number, there exists a unique 2-adic decomposition:

```
theorem twoadic_decomposition_unique:
  shows ‹∃!m. m ∈ [-½, ½)[1/2] ∧ (q - m) ∈ ℤ[2]›
⟨proof⟩
```

The 2-adic rounding $\lfloor\cdot\rceil_{[2]} : \mathbb{Q} \to \mathbb{Q}$ picks the unique decomposition `q = m + e` with `m` $\in \mathbb{Z}$`[½] ∩ [-½, ½)` and `e` two-adic-integral.

```
definition round_2 (‹⌊_⌉[2]›) where
  ‹⌊q⌉[2] ≡ THE m. m ∈ [-½, ½)[1/2] ∧ (q - m) ∈ ℤ[2]›
```

On a dyadic rational, the Euclidean and 2-adic roundings sum to the identity. This, while trivial, is a core step in the transition from Barrett arithmetic (Euclidean domain) to Montgomery arithmetic (2-adic domain).

```
theorem complementarity:
  assumes ‹q ∈ ℤ[1/2]›
  shows ‹⌊q⌉[∞] + ⌊q⌉[2] = q›
⟨proof⟩
```

Next, we gather a few basic arithmetic properties of 2-adic rounding. First, the 2-adic round of an integer divided by $2^n$ is its signed remainder modulo $2^n$, scaled. The candidate is dyadic and lies in `[-½, ½)`; the integer remainder `(x - x mod ⟦⌊·⌉⟧ 2^n) div 2^n` is trivially in $\mathbb{Z}_{[2]}$.

```
lemma round_2_int_div_pow2:
  assumes npos: ‹n > 0›
  shows ‹⌊x /ℚ 2^n⌉[2] = (x mod± 2^n /ℚ 2^n)›
⟨proof⟩
```

Next, `1 /ℚ N` is 2-adically congruent to $(N^{-1} \bmod R)_{\mathbb{Q}}$ modulo `R`. While this looks trivial, it should be noted that the left hand side computes `1 /ℚ N` as a *rational* quotient, while the right hand side computes $N^{-1} \bmod R$ as a quotient in $\mathbb{Z}/R\mathbb{Z}$ and then re-embeds it into $\mathbb{Q}$ via the composition of the unsigned embedding $\mathbb{Z}/R\mathbb{Z} \hookrightarrow \mathbb{Z}$ and the standard embedding $\mathbb{Z} \hookrightarrow \mathbb{Q}$. The point is that $\mathbb{Z}_{[2]}$ is a domain that can hoist both while also keeping `R` non-invertible, and hence allowing arithmetic modulo `R` in it.

```
lemma (in BarrettContext) mod_inverse_cong:
  shows ‹[1 /ℚ N = (N^-1 mod R)ℚ] (mod R ℤ[2])›
    and ‹[zℚ * (1 /ℚ N) = zℚ * (N^-1 mod R)ℚ] (mod R ℤ[2])›
⟨proof⟩
```

Taking both together gives the following concrete formula for the 2-adic round of `zℚ / (Nℚ * Rℚ)`. The reader will already notice the similarity of the right hand side with the definition of Montgomery reduction of `z`.

```
lemma (in BarrettContext) round_2_div_NR:
  shows ‹⌊z /ℚ (N * R)⌉[2] = ((z * (N^-1 mod R)) mod± R) /ℚ R›
⟨proof⟩
```

Finally, this establishes the description of Montgomery reduction through 2-adic rounding:

```
theorem (in BarrettContext) mont_sub_signed_as_round_2:
  shows ‹(mont_sub± ⟦N, n⟧ z)ℚ = z /ℚ R - Nℚ * ⌊z /ℚ (N * R)⌉[2]›
⟨proof⟩
```

## 10.3 The Barrett–Montgomery bridge

To derive the connection between Montgomery and Barrett arithmetic, the last remaining piece is the following identity relating `1 / N` and `⟦R /Q N⟧` modulo `R`:

```
lemma (in BarrettContext) magic_const_cong:
  shows ‹[⟦R /Q N⟧Q = - (R mod⟦f⟧ N) /Q N] (mod R ℤ[2])›
    and ‹[zQ * ⟦R /Q N⟧Q = zQ * (- (R mod⟦f⟧ N) /Q N)] (mod R ℤ[2])›
⟨proof⟩
```

The signed Barrett–Montgomery bridge is now a simple chain of equalities putting together the facts established above:

*Note.* Proof bodies are normally elided in this PDF; we exceptionally retain this one because of its brevity and centrality. See Section 2 for the editorial policy.

```
theorem (in BarrettContext) bridge_signed_conceptual:
  shows ‹bar±⟦N, n, f⟧ z = montadd±⟦N, n⟧ (z * (R mod⟦f⟧ N))›
proof -
  have cong: ‹(z * ⟦R /Q N⟧ /Q R - (- ((z * (R mod⟦f⟧ N)) /Q (N * R)))) ∈ ℤ[2]›
    using magic_const_cong(2)[of z] by (simp add: field_simps of_int_mult)
  have key: ‹NQ * (z * ⟦R /Q N⟧ /Q R) + (z * (R mod⟦f⟧ N)) /Q R = zQ›
    unfolding mod_approx_def by (simp add: field_simps)
  have ‹(bar±⟦N, n, f⟧ z)Q = zQ - NQ * ⌊z * ⟦R /Q N⟧ /Q R⌉[∞]›
    by (rule bar_signed_as_round_E)
  also have ‹... = zQ - NQ * (z * ⟦R /Q N⟧ /Q R - ⌊z * ⟦R /Q N⟧ /Q R⌉[2])›
    using complementarity[OF int_div_pow2_dyadic[of ‹z * ⟦R /Q N⟧› n]] by simp
  also have ‹... = (z * (R mod⟦f⟧ N)) /Q R + NQ * ⌊z * ⟦R /Q N⟧ /Q R⌉[2]›
    using key by (simp add: algebra_simps)
  also have ‹... = (z * (R mod⟦f⟧ N)) /Q R + NQ * ⌊- ((z * (R mod⟦f⟧ N)) /Q (N * R))⌉[2]›
    using round_2_invariance[OF cong] by simp
  also have ‹... = (montadd±⟦N, n⟧ (z * (R mod⟦f⟧ N)))Q›
    using mont_add_signed_as_round_2[of ‹z * (R mod⟦f⟧ N)›] by (simp add: algebra_simps)
  finally show ?thesis
    by (metis of_int_eq_iff)
qed
```

# 11 Barrett division for even moduli

The Montgomery reduction and multiplication of the previous chapters are defined in terms of the inverse `N⁻¹ mod R`, and are hence — *as written* — confined to an odd modulus `N`. Implementations of the ML-DSA *decompose* routine [13, Algorithm 36], however, often use Barrett arithmetic to divide exactly by an *even* modulus. The purpose of this chapter is to explain how the previous chapters can be generalized to even moduli, obtaining exactness of Barrett division as a corollary to (a version of) signed canonicity of Barrett reduction derived from the Barrett-Montgomery bridge.

The main observation is that the Montgomery reduction of `z` does not need `N⁻¹ mod R` to exist, but only the division `z / N` *modulo* `R`. This quotient can exist even if `N⁻¹ mod R` doesn't, though it is also non-unique in this case. We generalize the definition of Montgomery arithmetic to the choice of such modular quotients, and show how to express the Barrett–Montgomery bridge in this setting. Signed canonicity of Barrett reduction follows along the same lines as before, and the exactness of Barrett division is a corollary.

## 11.1 Montgomery arithmetic again – for any modulus

For an even modulus, `N` is not invertible modulo `R = R`, so the definition

$$\texttt{mont}_{sub}^{\pm}\ [\![\texttt{N,n}]\!]\ \texttt{z} \equiv (\texttt{z - z * N}^{-1}\ \texttt{mod}\ 2^{\texttt{n}}\ \texttt{mod}\ [\![\lfloor\cdot\rceil]\!]\ 2^{\texttt{n}}\ \texttt{* N})\ \texttt{div}\ 2^{\texttt{n}}$$

is ill-defined as written. However, it is well-defined as a partial function if we replace `z * N`$^{-1}$ `mod R` by a *choice* of modular quotient `z / N mod R`, assuming it exists. This renders Montgomery reduction well-defined for all inputs whose residue `mod R` is a multiple of `N`:

```
definition div_mod :: ‹int ⇒ int ⇒ int ⇒ int ⇒ bool›
    (‹(_) ≡ (_) '/ (_) '(mod _')› [61, 61, 61, 61] 60) where
  ‹q ≡ a / b (mod m) ≡ (b * q) mod m = a mod m›
```

A modular division by a unit is computed by its inverse:

```
lemma div_mod_inv_iff:
  assumes ‹(b * binv) mod m = 1 mod m›
  shows ‹(q ≡ a / b (mod m)) ⟷ q mod m = (a * binv) mod m›
⟨proof⟩
```

This readily specializes to the case of interest to us:

```
corollary div_mod_odd_iff:
  assumes ‹n > 0› and ‹odd N›
  shows ‹(q ≡ a / N (mod 2^n)) ⟷ q mod 2^n = (a * (N^-1 mod 2^n)) mod 2^n›
  ⟨proof⟩
```

Montgomery reduction for any modulus takes an input `z` and a chosen modular quotient `z_div_N` of `- z` by `N` modulo `R = `$2^{\texttt{n}}$, and returns `(z + z_div_N mod` $[\![\lfloor\cdot\rceil]\!]$ $2^{\texttt{n}}$ `* N) div` $2^{\texttt{n}}$. The shape mirrors the additive signed Montgomery reduction $\texttt{mont}_{add}^{\pm}\ [\![\texttt{N,n}]\!]$ of Section 5, with the chosen inverse replaced by the explicit quotient.

```
definition mont_red (‹mont_add^± [![_,_]!⟨_; _⟩›) where
  ‹mont_add^± [![N, n]!⟨z; z_div_N⟩ ≡ (z + ((z_div_N mod^± 2^n) * N)) div 2^n›
```

We obtain the same bound as before:

```
lemma mont_red_bound:
  assumes ‹N > 0› and ‹z_div_N ≡ - z / N (mod 2^n)›
  shows ‹|mont_add^± [![N, n]!⟨z; z_div_N⟩|_ℚ ≤ |z| /_ℚ 2^n + N /_ℚ 2›
⟨proof⟩
```

Montgomery multiplication also generalizes to potentially even moduli by passing (and assuming the existence of) a chosen modular quotient `b_tw ≡ b / N (mod R)`:

```
definition mont_mul (‹mont_add^± [![_,_]!⟨_, _; _⟩›) where
  ‹mont_add^± [![N, n]!⟨a, b; b_tw⟩ ≡ (a * b + ((a * b_tw) mod^± 2^n) * N) div 2^n›
```

As expected, multiplication is just reduction of the product against the folded twiddle:

```
lemma mont_mul_as_red:
  shows ‹mont_add^± [![N, n]!⟨a, b; b_tw⟩ = mont_add^± [![N, n]!⟨a * b; a * b_tw⟩›
  ⟨proof⟩
```

Correctness for any modulus: as long as the twiddle is a modular division of `- b` by `N` modulo `R`, the Montgomery multiplication is correct:

```
theorem mont_mul_correct:
  assumes ‹b_tw ≡ - b / N (mod 2^n)›
  shows ‹(mont_add^± [![N, n]!⟨a, b; b_tw⟩ * 2^n) mod N = (a * b) mod N›
⟨proof⟩
```

## 11.2 The Barrett–Montgomery bridge, again

Recall the bridge for an odd modulus (Section 7): signed Barrett reduction is signed Montgomery reduction of the input scaled by the correction `R mod` $[\![\texttt{f}]\!]$ `N`:

```
bar± ⟦N,n,f⟧ z = mont_add± ⟦N,n⟧ (z * (R mod ⟦f⟧ N))
```

To make sense of this for even `N`, we exhibit a modular quotient of `R mod ⟦f⟧ N` by `N`.

```
lemma magic_quotient_f:
  fixes N :: int and n :: nat and f :: ‹rat ⇒ int› (‹⟦_⟧›)
  shows ‹⟦2^n /Q N⟧ ≡ - (2^n mod⟦f⟧ N) / N (mod 2^n)›
⟨proof⟩
```

The bridge then carries over to *any* modulus, passing `⟦R /Q N⟧` as the concrete choice of modular quotient:

```
lemma barrett_montgomery_bridge_any:
  fixes N :: int and n :: nat and z :: int and f :: ‹rat ⇒ int› (‹⟦_⟧›)
  shows ‹bar±⟦N, n, f⟧ z = mont_add± ⟦N, n⟧⟨z, 2^n mod⟦f⟧ N; ⟦2^n /Q N⟧⟩›
⟨proof⟩
```

As before, we derive a bound for Barrett multiplication from the bridge:

```
theorem barrett_bound_gen:
  fixes N :: int and n :: nat and z :: int and f :: ‹rat ⇒ int› (‹⟦_⟧›)
  assumes Npos: ‹N > 0›
  shows ‹|(bar±⟦N, n, f⟧ z)Q| ≤ |z * (2^n mod⟦f⟧ N)| /Q 2^n + N /Q 2›
⟨proof⟩
```

## 11.3 Exactness of Barrett division via signed-canonicity

Recall from `barrett_red_signed_exact_iff` (Section 7) that exactness of the Barrett quotient and signed-canonicity of the Barrett reduction are equivalent. In this section we obtain a sufficient condition for the generalized (any-modulus) Barrett reduction to be signed-canonical, and read off exactness of the Barrett-division quotient as a corollary.

**Warning.** The signed-canonical window below is the half-open interval `- N/2 < x ≤ N/2`: it admits `N /Q 2` and *excludes* `- (N /Q 2)`, matching the round-half-down `mod±` of [13, Algorithm 36]. It is the *mirror image* of the `mod±` used here, whose range `- (N div 2) ≤ x, x < N div 2` admits `-N/2` and excludes `N/2`: the two place the half-way point at opposite ends. Our `mod±` matches the two's-complement interpretation of signed machine words, which is why it admits `-N/2` rather than `N/2`.

```
theorem barrettE_signed_canonical:
  fixes N :: int and n :: nat and z c :: int
  assumes Npos: ‹N > 0› and znn: ‹0 ≤ z› and cpos: ‹0 < c›
      and c_def: ‹c = 2^n - N * ⌊2^n /Q N⌋›
      and small: ‹2 * (z * c) < 2^n›
  shows ‹- (N /Q 2) < (bar±⟦N, n, ⌊·⌋⟧ z)Q›
    and ‹(bar±⟦N, n, ⌊·⌋⟧ z)Q ≤ N /Q 2›
⟨proof⟩
```

We obtain the desired exactness of the Barrett division as a corollary. Recall from Section 4 that $\lfloor\cdot\rceil_{\downarrow}$ denotes $\lfloor\cdot\rfloor_{\downarrow}$, the round-half-down operator that FIPS 204 uses for *decompose*:

```
theorem barrettE_division_exact:
  fixes N :: int and n :: nat and z c :: int
  assumes Npos: ‹N > 0› and znn: ‹0 ≤ z› and cpos: ‹0 < c›
      and c_def: ‹c = 2^n - N * ⌊2^n /Q N⌋›
      and small: ‹2 * (z * c) < 2^n›
  shows ‹⌊z * ⌊2^n /Q N⌋ /Q 2^n⌉ = ⌊z /Q N⌉↓›
⟨proof⟩
```

### 11.4 ML-DSA decompose specialisations

The above material can be specialized to the context of the `decompose` routine of ML-DSA [13, Algorithm 36]. That algorithm relies on round-half-down division of integers in the range `{0..<MLDSA_Q}` by the even integer `2 * γ2` [13, Table 1]. We justify two efficient Barrett-division strategies, both used among the backends of `mldsa-native`.

```
definition ‹MLDSA_Q :: int ≡ 2^23 - 2^13 + 1›
The ML-DSA modulus
definition ‹MLDSA_GAMMA2_88 :: int ≡ (MLDSA_Q - 1) div 88›
The value of γ2 in ML-DSA-44
definition ‹MLDSA_GAMMA2_32 :: int ≡ (MLDSA_Q - 1) div 32›
The value of γ2 in ML-DSA-65 and ML-DSA-87
```

For concreteness, we confirm the specific values of the two even moduli:

```
lemma ‹2 * MLDSA_GAMMA2_88 = 190464›
  and ‹2 * MLDSA_GAMMA2_32 = 523776›
  by eval+
```

The following results are at the heart of the correctness of the AArch64 assembly computing [13, Algorithm 36] in `mldsa-native`, proved against the instruction model in Subsection 13.5:

```
corollary barrett_decompose_32_aarch64:
  assumes ‹a ≥ 0› and ‹a < MLDSA_Q›
  shows ‹⌊a /Q (2 * MLDSA_GAMMA2_32)⌉↓ = ⌊a * 1074791425 /Q 2^49⌉›
⟨proof⟩

corollary barrett_decompose_88_aarch64:
  assumes ‹a ≥ 0› and ‹a < MLDSA_Q›
  shows ‹⌊a /Q (2 * MLDSA_GAMMA2_88)⌉↓ = ⌊a * 1477838209 /Q 2^48⌉›
⟨proof⟩
```

The C and AVX2 backends in mldsa-native instead divide by `128` first — rounding *up*, i.e. taking `⌈a /Q 128⌉` — and then Barrett-divide by `2 * γ2 div 128`:

```
corollary barrett_decompose_32_c_avx2:
  assumes ‹a ≥ 0› and ‹a < MLDSA_Q›
  shows ‹⌊a /Q (2 * MLDSA_GAMMA2_32)⌉↓ = ⌊⌈a /Q 128⌉ * 1025 /Q 2^22⌉›
⟨proof⟩

corollary barrett_decompose_88_c_avx2:
  assumes ‹a ≥ 0› and ‹a < MLDSA_Q›
  shows ‹⌊a /Q (2 * MLDSA_GAMMA2_88)⌉↓ = ⌊⌈a /Q 128⌉ * 11275 /Q 2^24⌉›
⟨proof⟩
```

## 12 Models of common word operations

In this chapter, we provide word-level models of common assembly instructions used in modular arithmetic kernels. The names for our models are aligned with the Armv8-A Neon instruction mnemonics. However, we emphasise that our models operate on individual words, while the corresponding Neon operations would — in the spirit of Single Instruction Multiple Data (SIMD) — apply them to each lane of the corresponding vector.

We model fixed-width words using Isabelle's standard *word library* (`HOL-Library.Word`). The type `'a word` represents bitvectors whose length is encoded by the type variable `'a` via the type class `len`; `LENGTH('a)` is the corresponding bit-width. `sint` is the two's-complement signed interpretation. Working at this level of generality lets us state every kernel correctness theorem once, without committing to a specific lane width.

For each instruction — the low-half multiplies `MUL`, `MLA`, `MLS`; the high-half multiplies `MULH` and `UMULH`; the doubling high-half multipliers `SQDMULH` and `SQRDMULH`; the multiply-accumulate `SQRDMLAH`; the halving subtract `SHSUB`; and the rounding right shift `SRSHR` — we give an abstract integer specification, define the corresponding word operation on `'a::len`-bit lanes, and prove that `sint` (or `uint`) of the word operation matches the spec.

## 12.1 Low-half multiply (`MUL`) and multiply-accumulate (`MLA`, `MLS`)

`MUL` computes the low half of the product, i.e. the signed reduction of `a * b` modulo the word size. `MLA`/`MLS` combine that product with an accumulator.

```
definition mul_word :: ‹'a::len word ⇒ 'a word ⇒ 'a word› where
  ‹mul_word a b ≡ a * b›

definition mla_word :: ‹'a::len word ⇒ 'a word ⇒ 'a word ⇒ 'a word› where
  ‹mla_word acc a b ≡ acc + a * b›

definition mls_word :: ‹'a::len word ⇒ 'a word ⇒ 'a word ⇒ 'a word› where
  ‹mls_word acc a b ≡ acc - a * b›
```

The signed correctness theorems are uniform: `sint` of each operation equals the corresponding integer expression reduced by symmetric modulo $2^{\mathtt{n}}$. There is no precondition — the low-half family is unconditionally well-behaved on signed inputs.

```
lemma sint_mul_ops:
  fixes a b acc :: ‹'a::len word›
  assumes n_def: ‹n = LENGTH('a)›
  shows ‹sint (mul_word a b) = (sint a * sint b) mod± (2 ^ n)›
    and ‹sint (mla_word acc a b) = (sint acc + sint a * sint b) mod± (2 ^ n)›
    and ‹sint (mls_word acc a b) = (sint acc - sint a * sint b) mod± (2 ^ n)›
  ⟨proof⟩
lemma sint_uint_low_mul:
  fixes a b :: ‹'a::len word›
  shows ‹(sint a * sint b) mod 2^LENGTH('a) = (uint a * uint b) mod 2^LENGTH('a)›
⟨proof⟩
```

By the nature of two's complement, low-half multiplication is the same operation regardless of whether words are interpreted as signed or unsigned: `sint a * sint b mod` $2^{\mathtt{LENGTH('a)}}$ `= uint a * uint b mod` $2^{\mathtt{LENGTH('a)}}$. There is therefore no distinction between signed and unsigned low-half multiply.

```
lemma uint_mul_ops:
  fixes a b acc :: ‹'a::len word›
  shows ‹uint (mul_word a b) = (uint a * uint b) mod 2^LENGTH('a)›
    and ‹uint (mla_word acc a b) = (uint acc + uint a * uint b) mod 2^LENGTH('a)›
    and ‹uint (mls_word acc a b) = (uint acc - uint a * uint b) mod 2^LENGTH('a)›
  ⟨proof⟩
```

## 12.2 Plain high-half multiply: `MULH`

This models *signed* high multiplication (`SMULH` in some ISAs): `mulh_word` interprets both operands via `sint` and returns the high `n` bits of the full `2n`-bit signed product. With no doubling, signed-word inputs yield `|sint a * sint b|` $\leq 2^{2 * \mathtt{n} - 2}$, so the magnitude after dividing by $2^{\mathtt{n}}$ is at most $2^{\mathtt{n} - 2} < 2^{\mathtt{n} - 1}$: `sint_mulh_word` carries no saturation precondition.

Unlike the low-half operations where signed and unsigned interpretations agree (since taking the low `n` bits of `a * b` is the same regardless of sign interpretation), high multipli-

cation is *not* sign-agnostic: the high half of the product depends on whether operands are treated as signed or unsigned. The unsigned variant `UMULH` is given separately below.

```
definition ‹mulh_int n a b ≡ (a * b) div 2^n›

definition mulh_word :: ‹'a::len word ⇒ 'a word ⇒ 'a word› where
  ‹mulh_word a b ≡ word_of_int (mulh_int (LENGTH('a)) (sint a) (sint b))›
```

The signed product of two `n`-bit operands has magnitude at most $2^{2 * n - 2}$, so its top `n` bits never leave the signed range and the specification holds unconditionally:

```
lemma sint_mulh_word:
  fixes a b :: ‹'a::len word›
  assumes n_def: ‹n = LENGTH('a)›
  shows ‹sint (mulh_word a b) = mulh_int n (sint a) (sint b)›
  ⟨proof⟩
```

### 12.2.1 Unsigned high-half multiply: `UMULH`

The unsigned variant interprets both operands via `uint` instead of `sint`. The integer-level formula is the same as `mulh_int`; only the word-level wrapper differs.

```
definition umulh_word :: ‹'a::len word ⇒ 'a word ⇒ 'a word› where
  ‹umulh_word a b ≡ word_of_int (mulh_int (LENGTH('a)) (uint a) (uint b))›

lemma uint_umulh_word:
  shows ‹uint (umulh_word a b :: 'a::len word) = (uint a * uint b) div 2^LENGTH('a)›
⟨proof⟩
```

## 12.3 High-half doubling multiplies: `SQDMULH` and `SQRDMULH`

Next we model saturating high multiplications. To extract the high part, we use Isabelle's `div` operation on `int`. We note that `div` on `int` is a floor division, matching the arithmetic right shift used by signed two's-complement high-half instructions like ARM Neon `SQDMULH` and PowerPC `vmhraddshs`. This is *not* C's `/` on `int32_t`, which truncates toward zero and would diverge on negative dividends.

```
value ‹(-1::int) div 2 = -1›  Sanity check

definition ‹sqdmulh_int n a b ≡ (2 * a * b) div 2^n›
```

The rounding variant `SQRDMULH` uses Isabelle's $\lfloor x \rceil$ `= ⌊x + 1 / 2⌋` round-to-nearest with ties broken toward $+\infty$.

```
definition ‹sqrdmulh_int n a b ≡ ⌊(2 * a * b) /Q 2^n⌉›
```

The word-level operations saturate: when both operands equal the extreme negative $-2^{n-1}$, the doubled product overflows the signed range and the hardware clamps to $2^{n-1} - 1$. This is the only input pair that triggers saturation. We capture this with a signed saturation function that clamps an integer to the `n`-bit signed range.

```
definition signed_saturate :: ‹int ⇒ 'a::len word› where
  ‹signed_saturate x ≡
     word_of_int (max (-(2^(LENGTH('a)-1))) (min x (2^(LENGTH('a)-1) - 1)))›

definition sqdmulh_word :: ‹'a::len word ⇒ 'a word ⇒ 'a word› where
  ‹sqdmulh_word a b ≡ signed_saturate (sqdmulh_int (LENGTH('a)) (sint a) (sint b))›

definition sqrdmulh_word :: ‹'a::len word ⇒ 'a word ⇒ 'a word› where
  ‹sqrdmulh_word a b ≡ signed_saturate (sqrdmulh_int (LENGTH('a)) (sint a) (sint b))›
```

The word-level lifts: as long as the inputs are not both at the extreme $-2^{n-1}$ (the only saturating case), the signed-integer specifications transfer to fixed-width word operands.

```
lemma sint_sqdmulh_word:
  fixes a b :: ‹'a::len word›
  assumes n_def: ‹n = LENGTH('a)›
  assumes not_extreme: ‹¬ (sint a = -(2^(n-1)) ∧ sint b = -(2^(n-1)))›
  shows ‹sint (sqdmulh_word a b) = sqdmulh_int n (sint a) (sint b)›
⟨proof⟩

lemma sint_sqrdmulh_word:
  fixes a b :: ‹'a::len word›
  assumes n_def: ‹n = LENGTH('a)›
  assumes not_extreme: ‹¬ (sint a = -(2^(n-1)) ∧ sint b = -(2^(n-1)))›
  shows ‹sint (sqrdmulh_word a b) = sqrdmulh_int n (sint a) (sint b)›
⟨proof⟩
```

## 12.4 Rounding multiply-accumulate high: SQRDMLAH

SQRDMLAH is the rounding multiply-accumulate variant: it computes `acc + ⌊2 * a * b /Q 2^n⌉`, fusing the high-half rounding multiplication of SQRDMULH with an accumulator add. It is used in the doubled rounding Montgomery kernel of [6, Algorithm 13], where the final SHSUB is dropped and the correction is folded into the multiply-accumulate.

```
definition ‹sqrdmlah_int n acc a b ≡ acc + sqrdmulh_int n a b›

definition sqrdmlah_word :: ‹'a::len word ⇒ 'a word ⇒ 'a word ⇒ 'a word› where
  ‹sqrdmlah_word acc a b ≡
     signed_saturate (sqrdmlah_int (LENGTH('a)) (sint acc) (sint a) (sint b))›
```

With the integer result inside the canonical signed range, `sint` of the word operation matches the integer specification.

```
lemma sint_sqrdmlah_word:
  fixes acc a b :: ‹'a::len word›
  assumes n_def: ‹n = LENGTH('a)›
  assumes lo: ‹-(2^(n-1)) ≤ sqrdmlah_int n (sint acc) (sint a) (sint b)›
      and hi: ‹sqrdmlah_int n (sint acc) (sint a) (sint b) < 2 ^ (n - 1)›
  shows ‹sint (sqrdmlah_word acc a b)
           = sqrdmlah_int n (sint acc) (sint a) (sint b)›
  ⟨proof⟩
```

## 12.5 Halving subtract: SHSUB

SHSUB absorbs a factor of two inside the Barrett-reduction kernel without losing the high bit; with no doubling, saturation cannot occur, so the `sint` specification is unconditional.

```
definition ‹shsub_int a b ≡ (a - b) div 2›

definition shsub_word :: ‹'a::len word ⇒ 'a word ⇒ 'a word› where
  ‹shsub_word a b ≡ word_of_int (shsub_int (sint a) (sint b))›

lemma sint_shsub_word:
  shows ‹sint (shsub_word a b :: 'a::len word) = shsub_int (sint a) (sint b)›
⟨proof⟩
```

## 12.6 Rounding right shift: SRSHR

SRSHR shifts a signed lane right by a compile-time constant `k`, rounding to nearest. We use it in refined Barrett reduction, where it follows a SQDMULH to realise an approximation at an inflated effective radix.

```
definition ‹srshr_int k x ≡ ⌊x /Q 2^k⌋›

definition srshr_word :: ‹nat ⇒ 'a::len word ⇒ 'a word› where
  ‹srshr_word k x ≡ word_of_int (srshr_int k (sint x))›

lemma sint_srshr_word:
  fixes x :: ‹'a::len word›
  assumes n_def: ‹n = LENGTH('a)›
  shows ‹sint (srshr_word k x) = srshr_int k (sint x)›
⟨proof⟩
```

## 12.7 Assembly-style notation for kernel definitions

We use simple syntax sugar around HOL `let` bindings to provide a shallow embedding of symbolic assembly sequences using the instructions above:

```
let ASM « MUL z a b; SQRDMULH t a halfBT; MLS r z t N » in r
```

desugars to nested `let`-bindings over the `*_word` operations above. An `ASM «...»` block must sit at the tail of its enclosing `let`.

Let's look at a simple example:

```
definition asm_test :: ‹'a::len word ⇒ 'a word ⇒ 'a word ⇒ 'a word› where
  ‹asm_test a b N ≡
     let ASM «
       MUL      z a b;
       SQRDMULH t a b;
       MLS      r z t N
     » in r›
```

This definitionally desugars to the following:

```
lemma asm_test_eq:
  ‹asm_test a b N
     ≡ (let z = mul_word a b;
            t = sqrdmulh_word a b
        in mls_word z t N)›
  ⟨proof⟩
```

## 12.8 Conformance testing

The instruction models above are pen-and-paper. To gain confidence that they faithfully reflect the AArch64 ISA, the development ships a small conformance harness under `model/`. Each modelled operation is exposed through Isabelle's code extractor as a string-keyed dispatcher `model_exec MNEMONIC BITWIDTH ARG ...`, exported to SML; a sibling C program `hw_exec` with the same CLI executes the corresponding instruction through inline assembly on real silicon (Neon for 8/16/32-bit lanes; scalar `SMULH`/`UMULH` at 64 bits). A Python driver pipes identical hex inputs into both and compares outputs bit-for-bit. The default `all` mode runs the 8-bit operations exhaustively (33M cases) and one million random plus all edge-value cases per 16/32/64-bit operation; current results are zero divergences across roughly 50M cases.

# 13 Neon kernels for Barrett-style modular arithmetic

In this chapter we analyse four Neon ASM kernel families for Barrett-style modular arithmetic and connect them to the corresponding abstract operators. In consequence, we obtain correctness and bounds statements for the kernel output.

## 13.1 Signed Barrett multiplication

We verify [6, Algorithm 10, §3.2]: the three-instruction MUL/SQRDMULH/MLS sequence computes barM$^{\pm}$ ⟦N,n,⌊·⌉$_2$⟧⟨a,b⟩ on signed lanes. The precomputed constant bt carries half the round-to-even magic constant (halving is exact since round-to-even produces an even integer). The bound |sint b| < sint N ensures bt avoids the extreme negative value, excluding SQRDMULH saturation for any a. We sharpen the published coarse bound |m| ≤ 3 * N / 2 of [6, Algorithm 10] to the strict |m| < N, available from the round-to-nearest-even narrowed bound (Section 7).

```
theorem barrett_mul_neon_word_correct:
  fixes N a b bt :: ‹'a::len word› and n :: nat
  assumes n_def: ‹n = LENGTH('a)›
  assumes bt_eq: ‹sint bt = ⌊sint b * 2^n /Q sint N⌉2 div 2›
      and N_std: ‹OddModulus (sint N) (n-1)›
      and b_bound: ‹|sint b|Q < (sint N)Q›
  defines ‹out ≡ let ASM «
                    MUL      z1 a b;
                    SQRDMULH t  a bt;
                    MLS      r  z1 t N
                  » in r›
  shows ‹sint out = barM± ⟦sint N, n, ⌊·⌉2⟧⟨sint a, sint b⟩›
        abstract description
    and ‹sint out mod sint N = (sint a * sint b) mod sint N›
        correctness
    and ‹|(sint out)Q| ≤ |sint a| * sint N /Q 2^n + sint N /Q 2›
        fine output bound
    and ‹|(sint out)Q| < (sint N)Q›
        coarse output bound
⟨proof⟩
```

## 13.2 Unsigned-style Barrett multiplication

Replacing SQRDMULH with truncating MULH, the three-instruction MUL/MULH/MLS sequence computes barM$^{+}$ ⟦N,n,⌊·⌋⟧⟨a,b⟩ on signed lanes. The coarse output bound is 5 * N /$_Q$ 4 rather than N, requiring a correspondingly tighter modulus constraint OddModulus N (n - 2).

```
theorem barrett_mul_unsigned_neon_word_correct:
  fixes N a b m :: ‹'a::len word› and n :: nat
  assumes n_def: ‹n = LENGTH('a)›
  assumes m_eq: ‹sint m = ⌊sint b * 2^n /Q sint N⌋›
      and N_std: ‹OddModulus (sint N) (n-2)›
      and b_bound: ‹|sint b|Q < (sint N)Q›
  defines ‹out ≡ let ASM «
                    MUL  z1 a b;
                    MULH t  a m;
                    MLS  r  z1 t N
                  » in r›
  shows ‹sint out = barM+ ⟦sint N, n, ⌊·⌋⟧⟨sint a, sint b⟩›
        functional description
    and ‹sint out mod sint N = (sint a * sint b) mod sint N›
        correctness
    and ‹|(sint out)Q| ≤ |sint a|Q * (sint N)Q / (2^n)Q + (sint N)Q›
        fine output bound
    and ‹|(sint out)Q| < 5 * (sint N)Q / 4›
        coarse output bound

⟨proof⟩
```

## 13.3 Single-input Barrett reduction

We verify [6, Algorithm 9, §3.2]: the two-instruction SQRDMULH/MLS sequence computes bar$^{\pm}$ ⟦N,n,⌊·⌉$_2$⟧ z on signed lanes. As for [6, Algorithm 10] above, we sharpen the published coarse bound |m| ≤ 3 * N / 2 of [6, Algorithm 9] to the strict |m| < N.

```
theorem barrett_red_neon_word_correct:
  fixes N Nt z :: ‹'a::len word›
    and n :: nat
  assumes n_def: ‹n = LENGTH('a)›
  assumes Nt_eq: ‹sint Nt = ⌊2^n /ℚ sint N⌉₂ div 2›
      and N_std: ‹OddModulus (sint N) (n-2)›
  defines ‹out ≡ let ASM «
                    SQRDMULH t z Nt;
                    MLS      r z t N
                  » in r›
  shows ‹sint out = bar±⟦sint N, n, ⌊·⌉₂⟧ (sint z)›
        abstract description
    and ‹sint out mod sint N = sint z mod sint N›
        correctness
    and ‹|(sint out)ℚ| ≤ |sint z|ℚ * (sint N)ℚ / (2^n)ℚ + (sint N)ℚ / 2›
        fine output bound
    and ‹|(sint out)ℚ| < (sint N)ℚ›
        coarse output bound
```

⟨*proof*⟩

## 13.4 Refined Barrett reduction

[6, Algorithm 11, §3.2] discusses a three-instruction SQDMULH/SRSHR/MLS sequence implementing refined Barrett reduction at effective radix $2^{n+\alpha-1}$. The output is canonical if we assume ε (⌊·⌉, $2^{n+\alpha-1}$ /ℚ sint N) ≤ 1 / 4 instead of the guaranteed ε (⌊·⌉, $2^{n+\alpha-1}$ /ℚ sint N) ≤ 1 / 2. Otherwise, the output is only guaranteed to be canonical for inputs with |sint z| ≤ $2^{n-2}$. The statement of [6, Algorithm 11] as published misses this and is wrong as stated. However, we note — and show below — that the assumption ε (⌊·⌉, $2^{n+\alpha-1}$ /ℚ sint N) ≤ 1 / 4 does hold in the context of ML-KEM [14] and ML-DSA [13], so for those applications the error is of no consequence.

```
theorem barrett_red_refined_neon_word_correct:
  fixes N V z :: ‹'a::len word› and n α δ :: nat
  assumes n_def: ‹n = LENGTH('a)›
  assumes V_eq: ‹sint V = ⌊2^(n+α-1) /ℚ sint N⌉›
        magic constant; the RHS fits in a signed n-bit word because 2^α < sint N
      and N_std: ‹OddModulus (sint N) n›
      and N_alpha: ‹2^α < sint N› ‹sint N < 2^(α+1)›
        α = ⌊log₂ (sint N)⌋
      and δ_quality: ‹ε(⌊·⌉, 2^(n+α-1) /ℚ sint N) ≤ 1/2^δ›
      and z_bound: ‹|sint z| ≤ 2^(n-1-(2-δ))›
        δ = 1 requires 1 bit of slack, 2 ≤ δ covers full signed n-bit range
  defines ‹out ≡ let ASM «
                    SQDMULH t z V;
                    SRSHR   q t #α;
                    MLS     r z q N
                  » in r›
  shows ‹sint out = bar±⟦sint N, n+α-1, ⌊·⌉⟧ (sint z)›
        abstract description: refined Barrett at radix 2^(n+α-1)
    and ‹sint out = sint z mod± sint N›
        output is the signed canonical residue
```

⟨*proof*⟩

Specialisation to ML-KEM: `N = 3329`, `n = 16`, $\alpha$ `= 11`, $\delta$ `= 2`. For this modulus, the rounding-quality factor satisfies $\varepsilon$ `(⌊·⌉, 2^26 /Q 3329) ≤ 1 / 4`, so the canonical-output theorem applies for the full signed 16-bit input range.

```
theorem barrett_red_refined_neon_word_correct_ml_kem:
  fixes z :: ‹16 word›
  defines ‹out ≡ let ASM «
                    SQDMULH t z 20159; (20159 = ⌊2^16 + 11 - 1 /Q 3329⌉)
                    SRSHR   q t #11;
                    MLS     r z q 3329
                  » in r›

  shows ‹sint out = bar±⟦3329, 26, ⌊·⌉⟧ (sint z)›
        abstract description: refined Barrett at radix 2^26
    and ‹sint out = sint z mod± 3329›
        output is the signed canonical residue
```

⟨*proof*⟩

Specialisation to ML-DSA: `N = 8380417`, `n = 32`, $\alpha$ `= 22`, $\delta$ `= 2`. For this modulus, the rounding-quality factor satisfies $\varepsilon$ `(⌊·⌉, 2^53 /Q 8380417) ≤ 1 / 4`, so the canonical-output theorem applies for the full signed 32-bit input range.

```
theorem barrett_red_refined_neon_word_correct_ml_dsa:
  fixes z :: ‹32 word›
  defines ‹out ≡ let ASM «
                    SQDMULH t z 1074791297; (1074791297 = ⌊2^32 + 22 - 1 /Q 8380417⌉)
                    SRSHR   q t #22;
                    MLS     r z q 8380417
                  » in r›

  shows ‹sint out = bar±⟦8380417, 53, ⌊·⌉⟧ (sint z)›
        abstract description: refined Barrett at radix 2^53
    and ‹sint out = sint z mod± 8380417›
        output is the signed canonical residue
```

⟨*proof*⟩

We conclude with a counterexample showing that without the hypothesis $\varepsilon$ `(⌊·⌉, 2^n + α - 1 /Q sint N) ≤ 1 / 4`, refined Barrett reduction only yields canonical outputs for `|sint z| ≤ 2^n - 2`:

```
lemma refined_barrett_example_3947:
  defines ‹n ≡ (16 :: nat)›
      and ‹α ≡ (11 :: nat)›
      and ‹δ ≡ (1 :: nat)›
      and ‹N ≡ (3947 :: 16 word)›
      and ‹V ≡ (17002 :: 16 word)›
      and ‹z ≡ (17762 :: 16 word)›
  shows ‹sint V = ⌊2^(n+α-1) /Q sint N⌉›
    and ‹OddModulus (sint N) n›
    and ‹2^α < sint N› ‹sint N < 2^(α+1)›
        Assumptions of barrett_red_refined_neon_word_correct are satisfied, except
for |sint z| ≤ 2^n - 1 - (2 - δ): We only have |sint z| ≤ 2^n - 1.
    and ‹ε(⌊·⌉, 2^(n+α-1) /Q sint N) ≤ 1/2^δ›
        ...  and only the guaranteed quality bound is available
    and ‹|sint (let ASM «
                    SQDMULH t z V;
                    SRSHR   q t #α;
```

```
                MLS     r z q N
              » in r)|Q ≥ sint N /Q 2›
       output is not signed canonical
```
⟨*proof*⟩

## 13.5 AArch64 Barrett division for ML-DSA *decompose*

The AArch64 *decompose* kernel of `mldsa-native` computes the high part of [13, Algorithm 36] with a two-instruction `SQDMULH/SRSHR` sequence — a doubling high-multiply by the magic constant, then a rounding shift. We show it equals the round-half-down division proved exact in Section 11.

The following AArch64 kernel can be used to compute ML-DSA-65/87's `decompose` routine:

```
theorem decompose_high_neon_word_correct_32:
  fixes z :: ‹32 word›
  assumes z_nn: ‹0 ≤ sint z› and z_lt: ‹sint z < MLDSA_Q›
  defines ‹out ≡ let ASM «
                    SQDMULH t z 1074791425;
                    SRSHR   q t #18
                  » in q›
  shows ‹sint out = ⌊sint z /Q (2 * MLDSA_GAMMA2_32)⌉↓›
```
⟨*proof*⟩

The following AArch64 kernel can be used to compute ML-DSA-44's `decompose` routine:

```
theorem decompose_high_neon_word_correct_88:
  fixes z :: ‹32 word›
  assumes z_nn: ‹0 ≤ sint z› and z_lt: ‹sint z < MLDSA_Q›
  defines ‹out ≡ let ASM «
                    SQDMULH t z 1477838209;
                    SRSHR   q t #17
                  » in q›
  shows ‹sint out = ⌊sint z /Q (2 * MLDSA_GAMMA2_88)⌉↓›
```
⟨*proof*⟩

# 14 Neon kernels for Montgomery-style modular arithmetic

In this chapter we analyse three Neon ASM kernels for Montgomery-style modular multiplication and connect them to the corresponding abstract operators. In consequence, we obtain correctness and bounds statements for the kernel output.

## 14.1 Signed Montgomery multiplication

The first `SQDMULH` needs `b` away from `- 2^n - 1` (one operand non-extreme suffices); the second needs the same for `N`, granted by the standard precondition, and `|k| ≤ R div 2`, automatic from `MUL`. `SHSUB` is non-saturating. The strict input bound on `b` is exactly the non-saturation condition.

The four-instruction `SQDMULH/MUL/SQDMULH/SHSUB` sequence computes $\texttt{mont}_{sub}^{\pm}$ `⟦N,n⟧ (a * b)` on signed lanes.

```
theorem mont_mul_neon_word_correct:
  fixes N bT a b :: ‹'a::len word› and n :: nat
  assumes n_def: ‹n = LENGTH('a)›
```

```
  assumes N_std: ‹OddModulus (sint N) (n-1)›
      and bT_eq: ‹sint bT = (sint b * ((sint N)⁻¹ mod 2^n)) mod± 2^n›
      and nondeg_b: ‹|sint b| < 2^(n-1)›
  defines ‹out ≡ let ASM «
                    SQDMULH z1 a b;
                    MUL     k  a bT;
                    SQDMULH c  k N;
                    SHSUB   r  z1 c
                  » in r›
  shows ‹sint out = mont_sub±⟦sint N, n⟧ (sint a * sint b)›
        abstract description
    and ‹(sint out * 2^n) mod sint N = (sint a * sint b) mod sint N›
        correctness
    and ‹|(sint out)_Q| ≤ |sint a * sint b|_Q / (2^n)_Q + (sint N)_Q / 2›
        fine output bound
    and ‹|(sint out)_Q| < (2^(n-1))_Q›

        coarse output bound
```

⟨*proof*⟩

## 14.2 Signed Montgomery multiplication (rounding variant)

Replacing the two `SQDMULH` instructions with their rounding counterparts `SQRDMULH` — and keeping the trailing `SHSUB` — yields a four-instruction kernel not explicitly discussed in [6].

```
theorem mont_mul_neon_rounding_word_correct:
  fixes N bT a b :: ‹'a::len word› and n :: nat
  assumes n_def: ‹n = LENGTH('a)›
  assumes N_std: ‹OddModulus (sint N) (n-1)›
      and bT_eq: ‹sint bT = (sint b * ((sint N)⁻¹ mod 2^n)) mod± 2^n›
      and nondeg_b: ‹|sint b| < 2^(n-1)›
  defines ‹out ≡ let ASM «
                    SQRDMULH z1 a b;
                    MUL      k  a bT;
                    SQRDMULH c  k N;
                    SHSUB    r  z1 c
                  » in r›
  shows ‹sint out = mont_sub±⟦sint N, n⟧ (sint a * sint b)›
        abstract description
    and ‹(sint out * 2^n) mod sint N = (sint a * sint b) mod sint N›
        correctness
    and ‹|(sint out)_Q| ≤ |sint a * sint b|_Q / (2^n)_Q + (sint N)_Q / 2›
        fine output bound
    and ‹|(sint out)_Q| < (2^(n-1))_Q›

        coarse output bound
```

⟨*proof*⟩

## 14.3 Doubled Montgomery multiplication

[6, Algorithm 13, §3.2]: the three-instruction `SQRDMULH`/`MUL`/`SQRDMLAH` sequence doubles the result and computes `2 * mont_add± ⟦N,n⟧ (a * b)` on signed lanes.

```
theorem mont_mul_neon_rounding_doubled_word_correct:
  fixes N bT a b :: ‹'a::len word› and n :: nat
  assumes n_def: ‹n = LENGTH('a)›
  assumes N_std: ‹OddModulus (sint N) (n-2)›
      and bT_eq: ‹sint bT = (sint b * ((- ((sint N)⁻¹ mod 2^n)) mod 2^n)) mod± 2^n›
```

```
      and small_a: ‹|sint a| < 2^(n-2)›
      and odd_b: ‹odd (sint b)›
  defines ‹out ≡ let ASM «
                    SQRDMULH z a b;
                    MUL      k a bT;
                    SQRDMLAH r z k N
                  » in r›
  shows ‹sint out = 2 * mont_add^±⟦sint N, n⟧ (sint a * sint b)›
        abstract description
    and ‹(sint out * 2^n) mod sint N = (2 * sint a * sint b) mod sint N›
        correctness
    and ‹|(sint out)_Q| ≤ (2^(n-1))_Q›
        coarse output bound
⟨proof⟩
```

# 15 Conclusions

We have given a machine-checked Isabelle/HOL account of the modular-arithmetic core of [6]: parametric Barrett and Montgomery reduction and multiplication uniform in modulus, word width, and integer approximation; the Barrett–Montgomery equivalence and its recasting through Euclidean–2-adic rounding duality; an empirical bound-quality study at the ML-KEM [14] and ML-DSA [13] moduli; an extension of the Barrett–Montgomery bridge to even moduli, underpinning ML-DSA's `decompose`; and correctness theorems for the signed-word Neon kernels shipped by `mlkem-native` and `mldsa-native`. A small algorithmic improvement surfaced in the process: the parity hypothesis on [6, Algorithm 8] is unnecessary, as independently discovered by the PPC64 contributors to `mlkem-native`.

The development also functions as a demonstrator for human-directed, model-generated formalisation at the scale of a research paper. With a tight agent–prover feedback loop — here, `AutoCorrode`'s I/Q and I/R interfaces — the agent drafted definitions and proofs while the human author owned the architecture, the narrative, and the editorial choices. The result was both faster than writing the formal text by hand and arguably tighter: the discipline of mechanization sharpens definitions and surfaces silent assumptions, and cross-cutting refactors that a human author would otherwise defer become cheap enough to perform routinely. We expect this style of work to become common in cryptographic engineering research.

Two pieces of infrastructure made this practical and warrant continued investment. Agent–prover interfaces such as those of `AutoCorrode` [5] give the agent direct access to prover state and a feedback channel without rebuilds. Isabelle's document preparation system keeps paper and proof in a single source: every chapter of this document is a theory file, and the synchronization overhead of maintaining separate paper and proof artifacts is removed by construction.

# References

[1] Amazon Web Services. Isabelle/HOL: The proof assistant behind the Nitro Isolation Engine. https://www.amazon.science/blog/isabelle-hol-the-proof-assistant-behind-the-nitro-isolation-engine, 2025.

[2] Apple Security Engineering and Architecture. Formal verification of corecrypto. https://security.apple.com/blog/formal-verification-corecrypto/, 2025.

[3] Archive of Formal Proofs. Archive of formal proofs. https://www.isa-afp.org.

[4] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Benjamin Grégoire, Vincent Laporte, Jean-Christophe Léchenet, Tiago Oliveira, Hugo Pacheco, Miguel Quaresma, Peter Schwabe, Antoine Séré, and Pierre-Yves Strub. Formally verifying Kyber: Episode IV: Implementation correctness. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2023(3):164–193, 2023.

[5] Hanno Becker, Nathan Chong, Robert Dockins, Jim Grundy, Jason Z. S. Hu, Ike Mulder, Dominic P. Mulligan, Paul Mure, Lawrence C. Paulson, and Konrad Slind. AutoCorrode software verification framework for Isabelle/HOL. https://github.com/awslabs/autocorrode, 2025.

[6] Hanno Becker, Vincent Hwang, Matthias J. Kannwischer, Bo-Yin Yang, and Shang-Yi Yang. Neon NTT: Faster Dilithium, Kyber, and Saber on Cortex-A72 and Apple M1. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2022(1):221–244, 2022. Available at https://eprint.iacr.org/2021/986.

[7] Hanno Becker, Vincent Hwang, Matthias J. Kannwischer, Bo-Yin Yang, and Shang-Yi Yang. Neon NTT: Faster Dilithium, Kyber, and Saber on Cortex-A72 and Apple M1. Talk at CHES 2022. Video: https://youtu.be/TSUtA5hmrtk?t=1483. Slides: https://iacr.org/submit/files/slides/2022/tches/ches2022/1_21/slides.pptx, 2022.

[8] Chun-Ming Chiu, Jiaxiang Liu, Ming-Hsien Tsai, Xiaomu Shi, Bow-Yaw Wang, and Bo-Yin Yang. Algebraic linear analysis for number theoretic transform in lattice-based cryptography. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2025(3):668–692, 2025.

[9] Thomas Hales, Mark Adams, Gertrud Bauer, Tat Dat Dang, John Harrison, Le Truong Hoang, Cezary Kaliszyk, Victor Magron, Sean McLaughlin, Tat Thang Nguyen, et al. A formal proof of the Kepler conjecture. *Forum of Mathematics, Pi*, 5, 2017.

[10] Vincent Hwang, Jiaxiang Liu, Gregor Seiler, Xiaomu Shi, Ming-Hsien Tsai, Bow-Yaw Wang, and Bo-Yin Yang. Verified NTT multiplications for NISTPQC KEM lattice finalists: Kyber, SABER, and NTRU. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2022(4):718–750, 2022.

[11] Gerwin Klein, June Andronick, Kevin Elphinstone, Toby Murray, Thomas Sewell, Rafal Kolanski, and Gernot Heiser. Comprehensive formal verification of an OS microkernel. *ACM Transactions on Computer Systems*, 32(1), 2014.

[12] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal verification of an OS kernel. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP)*, 2009.

[13] National Institute of Standards and Technology. Module-Lattice-Based Digital Signature Standard. Technical Report Federal Information Processing Standards Publication (FIPS) 204, U.S. Department of Commerce, August 2024.

[14] National Institute of Standards and Technology. Module-Lattice-Based Key-Encapsulation Mechanism Standard. Technical Report Federal Information Processing Standards Publication (FIPS) 203, U.S. Department of Commerce, August 2024.

[15] pq-code-package contributors. `mldsa-native`: Secure, fast, and portable C90 implementation of ML-DSA / FIPS 204. https://github.com/pq-code-package/mldsa-native, 2025.

[16] pq-code-package contributors. `mlkem-native`: Secure, fast, and portable C90 implementation of ML-KEM / FIPS 203. https://github.com/pq-code-package/mlkem-native, 2025.