Quotient Approximation Modular Reduction

Aurélien Greuet[1], Simon Montoya[1,2], and Clémence Vermeersch[1]

[1]IDEMIA France, Crypto and Security Lab.
firstname.lastname@idemia.com
[2]LIX, INRIA, CNRS, École Polytechnique, Institut Polytechnique de Paris, France.
firstname.lastname@lix.polytechnique.fr

Abstract

Modular reduction is a core operation in public-key cryptography. While a standard modular reduction is often required, a partial reduction limiting the growth of the coefficients is enough for several usecases.

Knowing the quotient of the Euclidean division of an integer by the modulus allows to easily recover the remainder. We propose a way to compute efficiently, without divisions, an approximation of this quotient. From this approximation, both full and partial reductions are deduced. The resulting algorithms are modulus specific: the sequence of operations to perform in order to get a reduction depends on the modulus and the size of the input.

We analyse the cost of our algorithms for a usecase coming from post-quantum cryptography. We show that with this modulus, on a CPU with a slow multiplication, our method gives an algorithm faster than prior art algorithms.

1 Introduction

Modular reduction is used on most of cryptosystems and is an important requirement for an efficient, secure and compact public-key cryptosystem. Depending on the cryptosystem parameters or structure, the modular reduction can be performed in different ways. Generic modular reduction algorithms exist, such as Montgomery [Mon85] or Barrett reduction [Bar86]. However, in some settings, specific modular reduction algorithms are more adapted. For example, Solinas reduction [Sol11] is an efficient method to reduce coefficients with modulus of the form $f\left(2^k\right)$, where f is a polynomial with low degree and small coefficients. Likewise, Crandall introduced a specific method for reduction modulo $2^k - c$ with small c in [Cra27].

In this paper, we introduce a method to compute an approximation of a quotient, leading to a modular reduction algorithm. For some specific application, this algorithm is faster than both generic and specific algorithms. This modular reduction finds applications in Lattice-based Cryptography.

1.1 Notations, definitions

Throughout this paper, all integers are considered as non-negative ones.

Logical operations We denote by:

- ">>" the logical right shift operation: let a be an integer of bit-length k, so that $a = \sum_{i=0}^{k-1} a_i \cdot 2^i$. Then for $0 \leq s \leq k-1$, $(a >> s) = \sum_{i=s}^{k-1} a_i \cdot 2^{i-s}$. If $s \geq k$, then $(a >> s) = 0$,

- "<<" the logical left shift operation.
- " & " the logical bitwise AND.

Integer and fractional parts Given a non-negative real number x, the integer part of x, that is the largest integer $\leq x$, is denoted by $\lfloor x \rfloor$. Its fractional part $x - \lfloor x \rfloor$ is denoted by $\{x\}$.

For the sequel, let $q \geq 2$ be a fixed integer.

Modular Reduction Given $a \in \mathbb{N}$, the modular reduction of a modulo q is the integer $r \in \{0, 1, \ldots, q-1\}$ such that $a = \lfloor a/q \rfloor \cdot q + r$. It is denoted by $a \mod q$.

Partial Modular Reduction Given $a > 2 \cdot q$, a *partial reduction* of a modulo q is an integer a' such that $a' < a$ and $a' \mod q = a \mod q$.

1.2 Problem Statement and Motivations

We consider the following problems:

1. Given a fixed modulus q and a bound k, compute a *modular reduction* of any input of bit-length $\leq k$.
2. Given a fixed modulus q and a bound k, compute a *partial reduction* of any input of bit-length $\leq k$.

Solving efficiently these two problems is motivated by lattice-based post-quantum cryptography. In these cryptosystems, polynomials with coefficients modulo q, where q is fixed and generally fits in a machine word, are multiplied (see e.g. [ABD+20, BDL+19, VSRDK, CDH+19]). Hence, the modular multiplication between coefficients is a core operation.

Most schemes work with a modulus q such that polynomial multiplication is computed using the Number Theoretic Transform (see e.g. [PG12]). After a transformation to the NTT domain, polynomial multiplication is handled by point-wise multiplication. Finally, the result is transformed back from the NTT domain. In this setting, the conversion to and from Montgomery representation can be done for free in the transformations to and from the NTT domain, see e.g. the reference implementations of [ABD+20, BDL+19]. Thus, the point-wise modular multiplications are naturally handled with Montgomery multiplication.

However, some specific devices like smartcards may have a slow CPU multiplication, while having a coprocessor handling large integers multiplication. In this context, it can be faster to transform the polynomial multiplication into a large integer one, thanks to the Kronecker substitution. Then, the large integer arithmetic is handled by the coprocessor, as presented in [AHH+19, BRvV21, GMR21]. Nevertheless, it computes the polynomial multiplication over the integers. Hence, each coefficient must be reduced modulo q. Thus, solving efficiently Problem 1 is of interest in this context.

In general, only the final values have to be fully reduced. Then, it can be sufficient to just control the size of intermediate values, to avoid overflows. In this case, efficient algorithms to solve Problem 2 are adapted.

1.3 Contributions

Let a and q be non negative integers, let k be an upper bound on the bit-length of a. We provide an algorithm that computes an approximation of $\lfloor a/q \rfloor$ as a sum of $(a >> j)$'s in Section 2.2.

A partial reduction modulo q is deduced from this algorithm in Section 2.3.1. Such a partial reduction is a $a \mod q + t \cdot q$, for a non-negative integer t. We prove that $t \leq \sum_{i=0}^{k-1} \left\{ \frac{2^i}{q} \right\}$.

In addition, a *relaxed* version is given in Section 2.3.2. It follows the same idea as the previous partial reduction, with a lower accuracy, leading to a faster algorithm.

In Section 2.4, we show that a full reduction can easily be obtained from the partial ones, e.g. by performing a standard division algorithm. A standard division algorithm leads to $a \mod q$, from $a \mod q + t \cdot q$, with at most $\lfloor \log(t) \rfloor + 1$ subtractions.

Section 3 is devoted to a usecase study. Coming from post-quantum cryptography, the modulus q is such that $\sum_{i=0}^{k-1} \left\{ \frac{2^i}{q} \right\}$ is small. We analyse the cost of our reduction and compare it to prior art algorithms. We show that on a CPU with a slow multiplication, our reduction is faster.

1.4 State of the Art

In this part we present standard modular reduction algorithms. The first two work with any modulus. The two others are designed for modulus with a special shape. For a more complete bibliography on modular reduction, see [GG03, BZ10] and references therein.

Montgomery Reduction Montgomery Reduction is introduced in [Mon85]. Given a modulus q, a radix R coprime to q and an integer $0 \leq a < R \cdot q$ to reduce, it computes $\tilde{a} = a \cdot R^{-1} \mod q$. Montgomery multiplication is a combination of a multiplication and a Montgomery Reduction.

To get $a \mod q$ from the Montgomery reduction $\tilde{a}$, either a pre-computation (input $a \cdot R \mod q$ instead of a) or a post-computation (output $\tilde{a} \cdot R \mod q$ instead of $\tilde{a}$) has to be done.

When several Montgomery multiplications are performed, the relative cost of pre or post-computation becomes negligible. In the context of NTT multiplication, even if only one multiplication by coefficient is done, pre and post-computations can be mixed into the transformations to and from the NTT domain, so that their cost becomes free.

However in our context, we assume that the value to reduce comes from a Kronecker substitution followed by a large integer multiplication. Thus, only one reduction is performed, so that the cost of pre or post-computation, that requires another reduction modulo q, remains significant. Hence, we consider that Montgomery reduction and Montgomery multiplication are out of scope for this paper.

Barrett Reduction Barrett reduction, introduced in [Bar86] and described in a more modern way in [MVOV18], is a partial reduction algorithm that does not require any assumption on the modulus. While it is often presented assuming that the value to reduce modulo q is less than q^2, we give here a more general presentation.

Let q and a be integers, let $\ell = \texttt{bitlen}(q)$. Barrett reduction computes a partial reduction of a modulo q as follow: let k be an integer such that $2^\ell < a < 2^k$ and let $m = \lfloor 2^k/q \rfloor$. Then a partial reduction of a is given by $a' = a - ((a \cdot m) >> k) \cdot q$. In addition, a' is either the modular reduction $a \mod q$ or $a \mod q + q$.

During Barrett reduction, the computation $a \cdot m$ can exceed the size of a register. Therefore, additional operation are required to handle this temporary result on 2 registers. However, some variants of Barrett algorithm allow to limit the size of temporary variables [MVOV18, Kon10]. The idea is to perform Barrett reduction using information only on the highest bits of a instead of the whole bits. To do so:

- The precomputed value: $m = \lfloor 2^{\ell+\alpha}/q \rfloor$
- Barrett reduction: $a' = a - q \cdot \texttt{quo}$ where $\texttt{quo} = [(a >> (\ell + \beta)) \cdot m] >> (\alpha - \beta)$ and α, β are integers.

This variant is a trade-off between temporary variables size and final subtraction requirement. Indeed, this partial reduction requires at best one final subtraction at cost of a slight increase in bit size or several final subtractions without increasing the bit size.

Generalized Mersenne Number Reduction Generalized Mersenne numbers are integers of the form $q = f\left(2^k\right)$, where f is a polynomial with small coefficients and low degree. A method to compute their modular reduction is presented in [Sol99, Sol11].

It is not a general modular reduction algorithm. Instead, it allows to find, given a modulus q, a sequence of operations leading to the modular reduction. These operations are logical shifts, logical ands, modular additions and subtractions.

As an illustration, we instantiate the first example in [Sol99] with $k = 5$: the reduction of $a < 2^{30}$ modulo $q = 2^{15} - 2^5 + 1$ is given by $a \mod q = T + S_1 + S_2 - D_1 - D_2$, where $+$ and $-$ are modular operations and $T = a \,\&\, \texttt{0x7FFF}$, $S_1 = (a >> 10) \,\&\, \texttt{0x7FE0}$, $S_2 = (a >> 20) \,\&\, \texttt{0x3E0}$, $D_1 = (a >> 15)$ and $D_2 = (a >> 25)$.

We refer to [Sol99, Sol11] for the general presentation of the method.

Pseudo Mersenne Number Reduction Pseudo Mersenne Numbers are integers of the form $q = 2^\ell - c$, where c is "small". An algorithm to computhe their modular reduction is introduced by Crandall in the patent [Cra27].

Let $q = 2^\ell - c$, with $c < 2^{\ell-1}$. This reduction relies on the identity $2^\ell = c \mod q$: if $a = a_1 2^\ell + a_0$ then $a \mod q = c \times a_1 + a_0 \mod q$. Hence, a recursive computation of $c \times a_1 + a_0$ is done, until the result is fully reduced.

Algorithm 1 CRANDALL

Require: $a, q = 2^\ell - c$
1: **while** $a \geq 2q$ **do**
2: $\quad a_0 = a \,\&\, (2^\ell - 1)$
3: $\quad a_1 = a >> \ell$
4: $\quad a = c \times a_1 + a_0$
5: **end while**
6: **return** a

In the following we present a modular reduction based Quotient Approximation and afterwards we compare it complexity with the previous algorithms in the case of post-quantum cryptography.

2 Quotient Approximation Reduction

2.1 Overview

Let q and a be integers, let $k = \texttt{bitlen}\,(a)$, so that $a = \sum_{i=0}^{k-1} a_i \cdot 2^i$, where a_i is the i-th bit of a.

The Quotient Approximation aims to compute efficiently a value close to $\lfloor a/q \rfloor$. Since $a \mod q = a - \lfloor a/q \rfloor \cdot q$, this is a first step to a partial reduction.

A first approximation is to compute $\sum_{i=0}^{k-1} a_i \cdot \left\lfloor \frac{2^i}{q} \right\rfloor$ rather than $\lfloor a/q \rfloor$. This sum is expected to be "close" to $\lfloor a/q \rfloor$. Indeed,

$$\left\lfloor \frac{a}{q} \right\rfloor = \left\lfloor \sum_{i=0}^{k-1} a_i \cdot \frac{2^i}{q} \right\rfloor = \left\lfloor \sum_{i=0}^{k-1} a_i \cdot \left\lfloor \frac{2^i}{q} \right\rfloor + \sum_{i=0}^{k-1} a_i \cdot \left\{ \frac{2^i}{q} \right\} \right\rfloor = \sum_{i=0}^{k-1} a_i \cdot \left\lfloor \frac{2^i}{q} \right\rfloor + \left\lfloor \sum_{i=0}^{k-1} a_i \cdot \left\{ \frac{2^i}{q} \right\} \right\rfloor,$$

where by definition, for each i, $\left\{2^i/q\right\} < 1$.

The sum $\sum_{i=0}^{k-1} a_i \cdot \left\lfloor \frac{2^i}{q} \right\rfloor$ is then computed replacing the computation of the $\left\lfloor 2^i/q \right\rfloor$'s with the computation of some $(a >> j)$'s, in order to avoid divisions. The idea is that if $\texttt{bitlen}\,(q) = \ell$ and q is "close to" a power of two, then $2^i/q$ is "close to" $2^i >> \ell$.

Section 2.2 is devoted to the computation of $\sum a_i \cdot \left\lfloor 2^i/q \right\rfloor$ as a sum of $(a >> j)$'s. Then, a partial reduction algorithms relying on this quotient approximation are presented in Section 2.3. In Section 2.4, two methods to get a modular reduction from the partial reduction are given. Finally, we present in Section 3 some specific usecases where our method is more efficient than the state of the art.

2.2 Computing $\sum a_i \cdot \lfloor 2^i/q \rfloor$ without divisions

Let q and a be integers, $\ell = \texttt{bitlen}\,(q)$ and $k = \texttt{bitlen}\,(a)$. Since the computation of $\lfloor 2^i/q \rfloor$ without division is straightforward if q is a power of 2, we assume in the sequel that q is not a power of 2. Let a_i be the i-th bit of a, so that $a = \sum_{i=0}^{k-1} a_i \cdot 2^i$. Notice that for each j, $\lfloor 2^j/q \rfloor$ is either $2 \cdot \lfloor 2^{j-1}/q \rfloor$ or $2 \cdot \lfloor 2^{j-1}/q \rfloor + 1$.

Let J_k be the set of integers $J_k = \left\{1 \leq j \leq k-1, \lfloor 2^j/q \rfloor = 2 \cdot \lfloor 2^{j-1}/q \rfloor + 1\right\}$.

Remark 1. *If q is not a power of 2 then $\ell = \texttt{bitlen}\,(q) \in J_k$ and is its smallest element.*

Indeed, let $1 \leq j \leq \ell - 1$. Then $q > 2^{\ell-1} \geq 2^j$, that implies $2^j/q < 1$ and then $\lfloor 2^j/q \rfloor = 0$. In particular, $\lfloor 2^j/q \rfloor = 2 \cdot \lfloor 2^{j-1}/q \rfloor = 0$ for $1 \leq j \leq \ell - 1$.

In addition, $2^{\ell-1} < q < 2^\ell$ (where $2^{\ell-1} < q$ being strict as q is not a power of 2), so that $1 < 2^\ell/q < 2$. Hence, $1 \leq \lfloor 2^\ell/q \rfloor < 2$. As an integer, $\lfloor 2^\ell/q \rfloor$ is necessarily equal to 1. In particular, $\lfloor 2^\ell/q \rfloor = 2 \cdot \lfloor 2^{\ell-1}/q \rfloor + 1 = 1$ and ℓ is the smallest element in J_k.

Lemma 1. *Let i_0, i_1 such that $]i_0, i_1] \cap J_k = \emptyset$. Then $\lfloor 2^{i_1}/q \rfloor = 2^{i_1 - i_0} \cdot \lfloor 2^{i_0}/q \rfloor$.*

Proof. Since $]i_0, i_1] \cap J_k = \emptyset$, for any $k \in]i_0, i_1]$, $\lfloor 2^k/q \rfloor = 2 \cdot \lfloor 2^{k-1}/q \rfloor$. Hence,

$$\lfloor 2^{i_1}/q \rfloor = 2 \cdot \lfloor 2^{i_1-1}/q \rfloor = 2^2 \cdot \lfloor 2^{i_1-2}/q \rfloor = \cdots = 2^{i_1-i_0} \cdot \left\lfloor 2^{i_1-(i_1-i_0)}/q \right\rfloor = 2^{i_1-i_0} \cdot \lfloor 2^{i_0}/q \rfloor .$$

□

Proposition 1. *Let $J_k = \left\{1 \leq j \leq k-1, \lfloor 2^j/q \rfloor = 2 \cdot \lfloor 2^{j-1}/q \rfloor + 1\right\}$. For all a such that $\texttt{bitlen}\,(a) \leq k$,*

$$\sum_{i=\ell}^{k-1} a_i \lfloor 2^i/q \rfloor = \sum_{j \in J_k} (a >> j) .$$

Proof. Let t be the cardinal of J_k and let $j_1 = \ell \leq j_2 \leq \cdots \leq j_t$ be its elements. We prove the following statement by backward induction: for all $2 \leq s \leq t$,

$$\sum_{i=j_s}^{k-1} a_i \lfloor 2^i/q \rfloor = \sum_{i=j_s}^{k-1} a_i \cdot 2^{i-j_{s-1}} \lfloor 2^{j_{s-1}}/q \rfloor + \sum_{j \in J_k,\, j \geq j_s} (a >> j) . \tag{1}$$

First, let's prove the case $s = t$. Applying Lemma 1 with $i_0 = j_t$ and $i_1 = i$ to $\lfloor 2^i/q \rfloor$ leads to

$$\sum_{i=j_t}^{k-1} a_i \cdot \lfloor 2^i/q \rfloor = \sum_{i=j_t}^{k-1} a_i \cdot 2^{i-j_t} \cdot \lfloor 2^{j_t}/q \rfloor .$$

Since $j_t \in J_k$, $\lfloor 2^{j_t}/q \rfloor = 2 \cdot \lfloor 2^{j_t-1}/q \rfloor + 1$. Replacing in the above equation gives

$$\sum_{i=j_t}^{k-1} a_i \cdot \lfloor 2^i/q \rfloor = \sum_{i=j_t}^{k-1} a_i \cdot 2^{i-j_t} \cdot \left(2 \cdot \lfloor 2^{j_t-1}/q \rfloor + 1\right) = \sum_{i=j_t}^{k-1} a_i \cdot 2^{i-j_t+1} \cdot \lfloor 2^{j_t-1}/q \rfloor + \sum_{i=j_t}^{k-1} a_i \cdot 2^{i-j_t}. \tag{2}$$

In the first term of the right-hand side, $\lfloor 2^{j_t-1}/q \rfloor$ can be replaced with $2^{j_t-j_{t-1}-1} \lfloor 2^{j_{t-1}}/q \rfloor$, using Lemma 1 with $i_0 = j_{t-1}$ and $i_1 = j_t - 1$.

In addition, the second term of the right-hand side is $(a >> j_t)$. Since j_t is the greatest element in J_k, $(a >> j_t)$ can be written $\sum_{j \in J_k,\, j \geq j_t} (a >> j)$.

Reporting these equalities in Equation 2 leads to

$$\begin{aligned}\sum_{i=j_t}^{k-1} a_i \cdot \lfloor 2^i/q \rfloor &= \sum_{i=j_t}^{k-1} a_i \cdot 2^{i-j_t+1} \cdot 2^{j_t-j_{t-1}-1} \lfloor 2^{j_{t-1}}/q \rfloor + \sum_{j \in J_k,\, j \geq j_t} (a >> j) \\ &= \sum_{i=j_t}^{k-1} a_i \cdot 2^{i-j_{t-1}} \lfloor 2^{j_{t-1}}/q \rfloor + \sum_{j \in J_k,\, j \geq j_t} (a >> j) ,\end{aligned}$$

that proves the case $s = t$.

We now assume that the induction hypothesis (Equation 1) is true for a given s and we prove it also holds for $s-1$. We split the sum $\sum_{i=j_{s-1}}^{k-1} a_i \cdot \lfloor 2^i/q \rfloor$ in two parts to get

$$\begin{aligned}\sum_{i=j_{s-1}}^{k-1} a_i \cdot \lfloor 2^i/q \rfloor &= \sum_{i=j_{s-1}}^{j_s-1} a_i \cdot \lfloor 2^i/q \rfloor + \sum_{i=j_s}^{k-1} a_i \cdot \lfloor 2^i/q \rfloor \\ &= \sum_{i=j_{s-1}}^{j_s-1} a_i \cdot \lfloor 2^i/q \rfloor + \left(\sum_{i=j_s}^{k-1} a_i \cdot 2^{i-j_{s-1}} \lfloor 2^{j_{s-1}}/q \rfloor + \sum_{j \in J_k,\, j \geq j_s} (a >> j) \right), \end{aligned} \tag{3}$$

where the second equality comes from the induction hypothesis.

Applying again Lemma 1, with $i_0 = j_{s-1}$ and $i_1 = i$, one gets

$$\sum_{i=j_{s-1}}^{j_s-1} a_i \cdot \lfloor 2^i/q \rfloor = \sum_{i=j_{s-1}}^{j_s-1} a_i \cdot 2^{i-j_{s-1}} \lfloor 2^{j_{s-1}}/q \rfloor ,$$

thus, replacing this expression in Equation 3,

$$\begin{aligned}\sum_{i=j_{s-1}}^{k-1} a_i \cdot \lfloor 2^i/q \rfloor &= \sum_{i=j_{s-1}}^{j_s-1} a_i \cdot 2^{i-j_{s-1}} \lfloor 2^{j_{s-1}}/q \rfloor + \left(\sum_{i=j_s}^{k-1} a_i \cdot 2^{i-j_{s-1}} \lfloor 2^{j_{s-1}}/q \rfloor + \sum_{j \in J_k,\, j \geq j_s} (a >> j) \right) \\ &= \sum_{i=j_{s-1}}^{k-1} a_i \cdot 2^{i-j_{s-1}} \lfloor 2^{j_{s-1}}/q \rfloor + \sum_{j \in J_k,\, j \geq j_s} (a >> j) . \end{aligned} \tag{4}$$

Since $j_{s-1} \in J_k$, $\lfloor 2^{j_{s-1}}/q \rfloor = 2 \cdot \lfloor 2^{j_{s-1}-1}/q \rfloor + 1$. Hence,

$$\begin{aligned}\sum_{i=j_{s-1}}^{k-1} a_i \cdot 2^{i-j_{s-1}} \lfloor 2^{j_{s-1}}/q \rfloor &= \sum_{i=j_{s-1}}^{k-1} a_i \cdot 2^{i-j_{s-1}} \left(2 \cdot \lfloor 2^{j_{s-1}-1}/q \rfloor + 1 \right) \\ &= \sum_{i=j_{s-1}}^{k-1} a_i \cdot 2^{i-j_{s-1}+1} \cdot \lfloor 2^{j_{s-1}-1}/q \rfloor + \sum_{i=j_{s-1}}^{k-1} a_i \cdot 2^{i-j_{s-1}} \end{aligned}$$

According to Lemma 1 applied with $i_0 = j_{s-2}$ and $i_1 = j_{s-1} - 1$, $\lfloor 2^{j_{s-1}-1}/q \rfloor = 2^{j_{s-1}-1-j_{s-2}} \lfloor 2^{j_{s-2}}/q \rfloor$, thus above equation becomes

$$\begin{aligned}\sum_{i=j_{s-1}}^{k-1} a_i \cdot 2^{i-j_{s-1}} \lfloor 2^{j_{s-1}}/q \rfloor &= \sum_{i=j_{s-1}}^{k-1} a_i \cdot 2^{i-j_{s-1}+1} \cdot 2^{j_{s-1}-1-j_{s-2}} \lfloor 2^{j_{s-2}}/q \rfloor + \sum_{i=j_{s-1}}^{k-1} a_i \cdot 2^{i-j_{s-1}} \\ &= \sum_{i=j_{s-1}}^{k-1} a_i \cdot 2^{i-j_{s-2}} \cdot \lfloor 2^{j_{s-2}}/q \rfloor + \sum_{i=j_{s-1}}^{k-1} a_i \cdot 2^{i-j_{s-1}} . \end{aligned}$$

Replacing above equation in Equation 4 and noticing that $\sum_{i=j_{s-1}}^{k-1} a_i \cdot 2^{i-j_{s-1}} = a >> j_{s-1}$, it comes

$$\begin{aligned}\sum_{i=j_{s-1}}^{k-1} a_i \cdot \lfloor 2^i/q \rfloor &= \sum_{i=j_{s-1}}^{k-1} a_i \cdot 2^{i-j_{s-2}} \cdot \lfloor 2^{j_{s-2}}/q \rfloor + (a >> j_{s-1}) + \sum_{j \in J_k,\, j \geq j_s} (a >> j) \\ &= \sum_{i=j_{s-1}}^{k-1} a_i \cdot 2^{i-j_{s-2}} \cdot \lfloor 2^{j_{s-2}}/q \rfloor + \sum_{j \in J_k,\, j \geq j_{s-1}} (a >> j) , \end{aligned}$$

that proves the backward induction step.

To conclude, we prove the statement of the proposition, starting from

$$\sum_{i=j_1}^{k-1} a_i \cdot \lfloor 2^i/q \rfloor = \sum_{i=j_1}^{j_2-1} a_i \cdot \lfloor 2^i/q \rfloor + \sum_{i=j_2}^{k-1} a_i \cdot \lfloor 2^i/q \rfloor . \tag{5}$$

According to Lemma 1 applied with $i_0 = j_1 = \ell$ and $i_1 = i$, the first term of the right-hand side can be written

$$\sum_{i=j_1}^{j_2-1} a_i \cdot \lfloor 2^i/q \rfloor = \sum_{i=j_1}^{j_2-1} a_i \cdot 2^{i-j_1} \lfloor 2^{j_1}/q \rfloor .$$

The second term is re-written thanks to Equation 1 for $s = 2$, that is true from the previous backward induction:

$$\sum_{i=j_2}^{k-1} a_i \cdot \lfloor 2^i/q \rfloor = \sum_{i=j_2}^{k-1} a_i \cdot 2^{i-j_1} \lfloor 2^{j_1}/q \rfloor + \sum_{j \in J_k,\, j \geq j_2} (a >> j) .$$

Replacing these two expressions in Equation 5 leads to:

$$\begin{aligned} \sum_{i=j_1}^{k-1} a_i \cdot \lfloor 2^i/q \rfloor &= \sum_{i=j_1}^{j_2-1} a_i \cdot 2^{i-j_1} \lfloor 2^{j_1}/q \rfloor + \sum_{i=j_2}^{k-1} a_i \cdot 2^{i-j_1} \lfloor 2^{j_1}/q \rfloor + \sum_{j \in J_k,\, j \geq j_2} (a >> j) \\ &= \sum_{i=j_1}^{k-1} a_i \cdot 2^{i-j_1} \lfloor 2^{j_1}/q \rfloor + \sum_{j \in J_k,\, j \geq j_2} (a >> j) \end{aligned}$$

Since $j_1 = \ell$, $\lfloor 2^{j_1}/q \rfloor = 1$, so that the first term of above right-hand side is $\sum_{i=j_1}^{k-1} a_i \cdot 2^{i-j_1}$, that is $(a >> j_1)$.

This means that

$$\sum_{i=j_1}^{k-1} a_i \cdot \lfloor 2^i/q \rfloor = (a >> j_1) + \sum_{j \in J_k,\, j \geq j_2} (a >> j) = \sum_{j \in J_k,\, j \geq j_1} (a >> j) = \sum_{j \in J_k} (a >> j)$$

□

Proposition 1 allows to compute, without division, an approximation of the quotient $\lfloor a/q \rfloor$. In the next section, we deduce a partial reduction algorithm, based on this quotient approximation.

2.3 Partial Modular Reductions

Let q be a modulus of bit-length ℓ, let k be the maximum bit-length of the numbers to reduce. Let J_k be the set of integers $J_k = \left\{1 \leq j \leq k-1, \lfloor 2^j/q \rfloor = 2 \cdot \lfloor 2^{j-1}/q \rfloor + 1\right\}$.

2.3.1 Quotient Approximation Partial Reduction

The following algorithm computes a partial reduction modulo q of any input a of bit-length at most k and such that $a \geq 2^\ell$.

Algorithm 2 QAPartialRed(a, q, J_k): Quotient Approximation Partial Reduction

Require: $a = \sum_{i=0}^{k-1} a_i \cdot 2^i \geq 2^\ell$, q, J_k defined as above
Ensure: $r = a - \left(\sum_{i=\ell}^{k-1} a_i \cdot \lfloor 2^i/q \rfloor\right) \cdot q$

```
quo_approx ← 0
for each j ∈ J_k do
    quo_approx ← quo_approx + (a >> j)
end for
r ← a − quo_approx · q
return r
```

Proposition 2. *Algorithm 2 is correct and it outputs a partial reduction of a modulo q, that is $r < a$ and $r \mod q = a \mod q$.*

Proof. It is clear that at the end of the inner loop, $\texttt{quo\_approx} = \sum_{j \in J_k} (a >> j)$. According to Proposition 1, $\sum_{j \in J_k} (a >> j) = \sum_{i=\ell}^{k-1} a_i \cdot \lfloor 2^i/q \rfloor$. Hence, at the end of the algorithm,

$$r = a - \left(\sum_{i=\ell}^{k-1} a_i \cdot \lfloor 2^i/q \rfloor\right) \cdot q,$$

so that it outputs the expected value.

In addition, $\texttt{quo\_approx} \neq 0$: since $a \geq 2^\ell$, at least one index $i \geq \ell$ is such that a's i-th bit a_i is not 0. Thus, the sum $\texttt{quo\_approx} = \sum_{i=\ell}^{k-1} a_i \cdot \lfloor 2^i/q \rfloor$ is necessarily non-zero. Hence, $r = a - \texttt{quo\_approx} \cdot q < a$. As the subtraction of a and a multiple of q, $r \mod q = a \mod q$, and r is a partial reduction of a modulo q. □

The following Proposition gives a bound on the difference between the modular reduction $a \mod q$ and the partial reduction given by Algorithm 2.

Proposition 3. *Let r be the output of Algorithm 2 for an input a of bit-length k. Then*

$$0 \leq r - (a \mod q) \leq \left\lfloor \sum_{i=0}^{k-1} \left\{ \frac{2^i}{q} \right\} \right\rfloor \cdot q.$$

Proof. Let a be a k-bit integer and r the corresponding output of Algorithm 2. According to Proposition 2, denoting by a_i the i-th bit of a,

$$r = a - \left(\sum_{i=\ell}^{k-1} a_i \cdot \lfloor 2^i/q \rfloor\right) \cdot q. \tag{6}$$

Moreover,

$$\left\lfloor \frac{a}{q} \right\rfloor = \left\lfloor \sum_{i=0}^{k-1} a_i \cdot \frac{2^i}{q} \right\rfloor = \left\lfloor \sum_{i=0}^{k-1} a_i \cdot \left\lfloor \frac{2^i}{q} \right\rfloor + \sum_{i=0}^{k-1} a_i \cdot \left\{ \frac{2^i}{q} \right\} \right\rfloor = \sum_{i=0}^{k-1} a_i \cdot \left\lfloor \frac{2^i}{q} \right\rfloor + \left\lfloor \sum_{i=0}^{k-1} a_i \cdot \left\{ \frac{2^i}{q} \right\} \right\rfloor$$

Or $\sum_{i=0}^{l-1} a_i \cdot \left\lfloor \frac{2^i}{q} \right\rfloor = 0$, then

$$\left\lfloor \frac{a}{q} \right\rfloor = \sum_{i=\ell}^{k-1} a_i \cdot \left\lfloor \frac{2^i}{q} \right\rfloor + \left\lfloor \sum_{i=0}^{k-1} a_i \cdot \left\{ \frac{2^i}{q} \right\} \right\rfloor \tag{7}$$

Since $a \mod q = a - \lfloor a/q \rfloor \cdot q$, using it follows from Equations 6 and 7 that

$$\begin{aligned} r - (a \mod q) &= \left(a - \left(\sum_{i=\ell}^{k-1} a_i \cdot \lfloor 2^i/q \rfloor\right) \cdot q\right) - (a - \lfloor a/q \rfloor \cdot q) \\ &= \left(\lfloor a/q \rfloor - \sum_{i=\ell}^{k-1} a_i \cdot \lfloor 2^i/q \rfloor\right) \cdot q \\ &= \left(\sum_{i=\ell}^{k-1} a_i \cdot \lfloor 2^i/q \rfloor + \left\lfloor \sum_{i=0}^{k-1} a_i \cdot \{2^i/q\} \right\rfloor - \sum_{i=\ell}^{k-1} a_i \cdot \lfloor 2^i/q \rfloor\right) \cdot q \\ &= \left\lfloor \sum_{i=0}^{k-1} a_i \cdot \{2^i/q\} \right\rfloor \cdot q \leq \left\lfloor \sum_{i=0}^{k-1} \{2^i/q\} \right\rfloor \cdot q, \end{aligned}$$

the last inequation holding because each a_i is a bit, so that $a_i \leq 1$. It is also clear that $\left\lfloor \sum_{i=0}^{k-1} a_i \cdot \{2^i/q\} \right\rfloor \geq 0$, as a sum of non-negative elements, thus so is $r - (a \mod q)$. □

Remark 2. *Practically, when working with a fixed modulus, it is more efficient to "unroll" the loop in Algorithm 2, as shown in the next toy example.*

Example 1. *Let $q = 2^4 - 2 = 14$, so that $\ell = 4$. Let $k = 10$ be the maximum bit-length of the number to reduce. The table*

i	0	1	2	3	4	5	6	7	8	9
$\lfloor 2^i/q \rfloor$	0	0	0	0	1	2	4	9	18	36
$\{2^i/q\}$	1/14	1/7	2/7	4/7	1/7	2/7	4/7	1/7	2/7	4/7

ensures that $J_8 = \{4, 7\}$. Then, Algorithm 2 can be written:

- `quo_approx` $\leftarrow (a >> 4) + (a >> 7)$
- *return* $a -$ `quo_approx` $\cdot\, q$

In addition, Proposition 3 ensures that for any 10-bit integer a, the corresponding result r is such that:

$$0 \leq r - (a \mod q) \leq \left\lfloor \sum_{i=0}^{10-1} \left\{\frac{2^i}{q}\right\} \right\rfloor \cdot q = \left\lfloor \frac{43}{14} \cdot q \right\rfloor = 3 \cdot q.$$

Finally, we give the number of operations to perform the Quotient Approximation partial reduction.

Proposition 4. *Let q be a modulus of bit-length ℓ, let k be the maximum bit-length of the numbers to reduce. Let J_k be the set of integers $J_k = \left\{1 \leq j \leq k-1, \lfloor 2^j/q \rfloor = 2 \cdot \lfloor 2^{j-1}/q \rfloor + 1\right\}$ and let $n = \#J_k$ be its cardinality. Then for all a of bit-length at most k, QAPartialRed(a, q, J_k) is computed with*

- *n right shifts on k-bit elements,*
- *$n - 1$ additions of $(k - \ell)$-bit elements,*
- *1 multiplication between an element of bit-length at most $k - \ell + 1$ and one ℓ-bit element,*
- *1 subtraction between two elements of size at most k.*

Proof. The for loop computes the sum of n elements, each element being a $(a >> j)$ with $j \geq \ell$. Hence, this can be done with n shifts on k-bit elements and $n-1$ additions on $(k-\ell)$-bit elements. Since `quo_approx` $\leq \lfloor a/q \rfloor$, it has bit-length at most $k - \ell + 1$.

Thus, the final step $a -$ `quo_approx` $\cdot\, q$ requires 1 multiplication between an element of bit-length at most $k - \ell + 1$ and one ℓ-bit element. Since the result is non-negative according to Proposition 3, the result of the multiplication is necessarily of bit-length at most k. The subtraction is then between two elements of size at most k. □

2.3.2 Quotient Approximation Partial Reduction Relaxed

The Algorithm 2 determines an approximation of the quotient by iterating over all the elements of J_k. However in some cases we do not need such a precision for the approximated quotient. Hence, one can perform the Algorithm 2 in a subset of J_k.

Let $J_k = J'_k \cup \bar{J}'_k$ where $J'_k \cap \bar{J}'_k = \emptyset$ and the first element of J_k is included in J'_k. The Algorithm 3 computes a partial reduction modulo of any input a of bit-length at most k such that $a \geq 2^\ell$ and the inner loop iterates in $J'_k \subset J_k$.

Algorithm 3 QAPartialRedRelaxed(a, q, J'_k): Quotient Approximation Partial Reduction Relaxed

Require: $a = \sum_{i=0}^{k-1} a_i \cdot 2^i \geq 2^\ell$, q, J'_k defined as above
Ensure: $r = a - \left(\sum_{i=\ell}^{k-1} a_i \cdot \lfloor 2^i/q \rfloor + \sum_{j \in \bar{J}'_k} \lfloor (2^k - 1)/2^j \rfloor\right) \cdot q$
1: quo_approx $\leftarrow 0$
2: **for** each $j \in J'_k$ **do**
3: quo_approx $\leftarrow$ quo_approx $+ (a >> j)$
4: **end for**
5: $r \leftarrow a -$ quo_approx $\cdot q$
6: **return** r

Proposition 5. *Algorithm 3 is correct and it outputs a partial reduction of a modulo q, that is $r < a$ and $r \mod q = a \mod q$.*

Proposition 6. *Let r be the output of Algorithm 3 for an input a of bit-length k. Then*

$$0 \leq r - (a \mod q) \leq \left\lfloor \left\lfloor \sum_{i=0}^{k-1} \left\{ \frac{2^i}{q} \right\} \right\rfloor + \sum_{j \in \bar{J}'_k} \lfloor (2^k - 1)/2^j \rfloor \right\rfloor \cdot q.$$

Proof. By definition $J_k = J'_k \cup \bar{J}'_k$ where $J'_k \cap \bar{J}'_k = \emptyset$ and the first element of J_k is included in J'_k. Then,

$$\sum_{j \in J'_k} (a >> j) = \sum_{j \in J_k} (a >> j) - \sum_{j \in \bar{J}'_k} (a >> j) = \sum_{i=\ell}^{k-1} a_i \cdot \lfloor 2^i/q \rfloor - \sum_{j \in \bar{J}'_k} \lfloor a/2^j \rfloor > 0 \tag{8}$$

Hence at the end of Algorithm 3:

$$r = a - \left(\sum_{i=\ell}^{k-1} a_i \cdot \lfloor 2^i/q \rfloor - \sum_{j \in \bar{J}'_k} \lfloor a/2^j \rfloor \right) \cdot q \leq a - \left(\sum_{i=\ell}^{k-1} a_i \cdot \lfloor 2^i/q \rfloor - \sum_{j \in \bar{J}'_k} \lfloor (2^k - 1)/2^j \rfloor \right) \cdot q \tag{9}$$

Using equations 8 and 9, the proofs of Propositions 5 and 6 can be deduced from the proofs of Propositions 2 and 3. □

The Proposition 4 holds for the set J'_k. A use case of Algorithm 3 is described in Section 3.

2.4 From Partial Reduction to Modular Reduction

In this section, two methods are proposed to get a modular reduction from the Quotient Approximation partial reduction.

2.4.1 Partial Reduction Iterations

A first straightforward way to get a complete modular reduction is to iterate the partial reduction algorithm until the result is $< 2^\ell$. Up to a final subtraction with q, the result is a full modular reduction. This leads to Algorithm 4.

Algorithm 4 QARed: Quotient Approximation Reduction

Require: $a = \sum_{i=0}^{k-1} a_i \cdot 2^i$, q, J_k
Ensure: $r = a \mod q$

```
r ← a
J ← J_k
while r ≥ 2^ℓ do
    r ← QAPartialRed(r, q, J)
    J ← J_bitlen(r)
end while
if r ≥ q then
    r ← r − q
end if
return r
```

Proposition 7. *Algorithm 4 is correct.*

Proof. Given $r \geq 2^\ell$, $\mathsf{QAPartialRed}(r)$ is a partial reduction modulo q according to Proposition 2. In particular, $\mathsf{QAPartialRed}(r) < r$. Thus, the while loop ends and $r \mod q = a \mod q$ always holds.

In addition, if $r < 2^\ell$, then either it is already less than q (and then already $= a \mod q$) or it is between q and $2 \cdot q$. Indeed, q has bit-length equal to ℓ, so that $2 \cdot q \geq 2^\ell > r$. In the latter case, $0 \leq r - q < q$, thus $r - q = a \mod q$. □

Remark 3. *A closed-form formula for the complexity in the general case seems hard to get and outside the scope of this paper. However, for a fixed modulus q, the maximum number of iteration of the while loop can be computed thanks to the bound given in Proposition 3.*

Example 2 (Example 1 continued)**.** *In Example 1, after one iteration, $r - (a \mod q) \leq 3 \cdot q$. Hence, $r \leq (a \mod q) + 3 \cdot q \leq 4 \cdot q = 56 < 2^6$ is at most 6-bit long.*

Then for the second iteration, $\texttt{quo\_approx} = (r >> 4)$ *and* $r \leq (a \mod q) \cdot q + \left\lfloor \sum_{i=0}^{6-1} \{2^i/q\} \right\rfloor \cdot q \leq 2 \cdot q$*, thus it is at most 5-bit long.*

For the third iteration, $\texttt{quo\_approx} = (r >> 4)$*. If r was $\geq 2^4$ before this third step, since it was necessarily $\leq 2 \cdot q < 2^5$, then* $\texttt{quo\_approx} = 1$*. In that case, the new r is $r = r - q \leq q < 2^4$. In the other case,* $\texttt{quo\_approx} = 0$ *but r was $< 2^4$ already.*

Finally, with the last subtraction, the full algorithm can be unrolled as follow:

1. $\texttt{quo\_approx} \leftarrow (a >> 4) + (a >> 7)$
2. $r \leftarrow a - \texttt{quo\_approx} \cdot q$
3. $\texttt{quo\_approx} \leftarrow (r >> 4)$
4. $r \leftarrow r - \texttt{quo\_approx} \cdot q$
5. $\texttt{quo\_approx} \leftarrow (r >> 4)$
6. $r \leftarrow r - \texttt{quo\_approx} \cdot q$
7. *if* $r \geq q$ *then* $r \leftarrow r - q$
8. *return* r

2.4.2 Division Algorithm

A second way to get a modular reduction from a Quotient Approximation partial reduction is to perform a standard division algorithm until the full reduction is reached.

Algorithm 5 QARedDiv: Quotient Approximation Reduction with Divisions

Require: $a = \sum_{i=0}^{k-1} a_i \cdot 2^i$, q, J_k
Ensure: $r = a \mod q$
1: $r \leftarrow \mathsf{QAPartialRed}(a, q, J_k)$
2: $s \leftarrow \texttt{bitlen}(r)$
3: **while** $s \geq \ell$ **do**
4: **if** $r - (q << (s - \ell)) \geq 0$ **then**
5: $r \leftarrow r - (q << (s - \ell))$
6: **end if**
7: $s \leftarrow s - 1$
8: **end while**
9: **return** r

The correctness of Algorithm 5 is straightforward as it is QARedDiv followed by a standard binary division algorithm, that computes the remainder of r in its division by q.

Likewise, according to Proposition 3, $r \leq (a \mod q) + \left\lfloor \sum_{i=0}^{k-1} \{2^i/q\} \right\rfloor \cdot q$, that is $r = (a \mod q) + \delta \cdot q$, where $0 \leq \delta \leq \left\lfloor \sum_{i=0}^{k-1} \{2^i/q\} \right\rfloor$. Then the while loop actually computes $r - \delta \cdot q$, that is done with $\texttt{bitlen}(\delta)$ iterations with a standard binary algorithm. Hence, at most $\texttt{bitlen}\left(\left\lfloor \sum_{i=0}^{k-1} \{2^i/q\} \right\rfloor\right)$ iterations of the while loop are performed.

Remark 4. *Practically, for a fixed modulus, the algorithm is unrolled, as shown in the following example.*

Example 3 (Example 1 continued)**.** *In Example 1, after one iteration,* $r - (a \mod q) \leq 3 \cdot q$. *Hence,* $a \mod q$ *can be recovered from* r *with a most 2 subtractions. The unrolled algorithm is then:*

1. $\texttt{quo\_approx} \leftarrow (a >> 4) + (a >> 7)$
2. $r \leftarrow a - \texttt{quo\_approx} \cdot q$
3. *if* $r - (q << 1) \geq 0$ *then* $r \leftarrow r - (q << 1)$
4. *if* $r - q \geq 0$ *then* $r \leftarrow r - q$
5. *return* r

3 Application: CRYSTALS-Dilithium

Dilithium [BDL+19] is a lattice-based signature, finalist of the NIST call for Post-Quantum Cryptography [MAA+20]. It relies on multiplication of polynomials of degree 256, modulo $q = 8380417 = 2^{23} - 2^{13} + 1$.

Key Generation We first consider multiplications in Key Generation for Dilithium2. In this context, 2 polynomials of degree $n = 256$ are multiplied. The first one, says f, has coefficients in $[0, q[$ while the second one, say g, has coefficients in $[-\eta, \eta]$, with $\eta = 2$. To deal with non-negative coefficients, one can compute $f \cdot g$ as $f \cdot g^+ - f \cdot g^-$, where g^+ has coefficients in $[0, \eta]$, g^- in $]0, \eta]$ and $g = g^+ - g^-$.

Hence, without loss of generality, we consider the polynomial multiplication between f and a polynomial with coefficients in $[0, \eta]$. Each coefficient a of the result is such that $0 \leq a \leq n \cdot q \cdot \eta < 2^{32}$.

To apply Quotient Approximation reduction with $k = 32$, we compute $J_k = \{23\}$ and $\left\lfloor \sum_{i=0}^{k-1} \{2^i/q\} \right\rfloor = \left\lfloor \frac{12574208}{8380417} \right\rfloor = 1$. It follows that at most 1 subtraction by q is needed to get $a \mod q$ from $\mathsf{QAPartialRed}(a, q, J_k)$. Hence, our modular reduction becomes:

1. $\texttt{quo\_approx} \leftarrow (a >> 23)$
2. $r \leftarrow a - \texttt{quo\_approx} \cdot q$
3. if $r - q \geq 0$ then $r \leftarrow r - q$
4. return r

In the following we assess Quotient Approximation, a variant of Barrett, Crandall and Solinas algorithms in the case of Dilithium parameters.

For the Barrett algorithm, we use the parameters $\alpha = 10, \beta = -2$ and $m = \lfloor 2^{33}/q \rfloor$. These parameters ensure that the temporary variable fits in a register and at most one final subtraction is required.

For Crandall algorithm, the modular reduction is done with one iteration in the "while" loop and a final subtraction.

For Solinas algorithm, the reduction of $a < 2^{32}$ modulo $q = 2^{23} - 2^{13} + 1$ is given by $a \mod q = T + S_1 + S_2 - D$, where $+$ and $-$ are modular operations and $T = a \,\&\, \texttt{7FFFFF}$, $S_1 = (a \,\&\, \texttt{0x7FE0}) >> 10$, $S_2 = (a \,\&\, \texttt{0x3E0}) >> 20$, $D = (a \,\&\, \texttt{0x1ff800000}) >> 15$.

The Table 1 describes the operations count for each reduction algorithm. The result of the operations fits in a register.

	Operation on 32 bits				
	Mult.	Add/Sub	Shift	AND	Cond. Sub
$\texttt{Barrett}_{10,-2}$	2	1	2	0	1
Solinas	0	2	2	3	1
Crandall	1	1	1	1	1
QARed	1	1	1	0	1

Table 1: Complexity of the reduction algorithms

In this context Quotient Approximation reduction is more efficient than Barrett and Crandall algorithms. The comparison between Solinas and QA algorithms depends on the component. Indeed, if a multiplication where both the result and each operand fit in a register costs less than 3 AND, 1 add and 1 shift, then Quotient Approximation is faster than Solinas. Otherwise Solinas is faster.

Signature In the Dilithium signature computation, one polynomial has coefficients in $[0, q[$. The other one has coefficients in $[0, 2^{\gamma_1}]$, where $\gamma_1 = 2^{17}$ or 2^{19}. Hence, each coefficient a of the result is such that $0 \leq a \leq n \cdot q \cdot 2^{\gamma_1} < 2^{50}$.

Here, for $k = 50$, we get $J_k = \{23, 33, 44, 45, 46\}$ and $\left\lfloor \sum_{i=0}^{k-1} \{2^i/q\} \right\rfloor = 5$. It follows that at most $\texttt{bitlen}\,(5) = 3$ subtractions by q are needed to get $a \mod q$ from $\mathsf{QAPartialRed}(a, q, J_k)$:

1. $\texttt{quo\_approx} \leftarrow (a >> 23) + (a >> 33) + (a >> 44) + (a >> 45) + (a >> 46)$
2. $r \leftarrow a - \texttt{quo\_approx} \cdot q$
3. if $r - (q << 2) \geq 0$ then $r \leftarrow r - (q << 2)$
4. if $r - (q << 1) \geq 0$ then $r \leftarrow r - (q << 1)$
5. if $r - q \geq 0$ then $r \leftarrow r - q$
6. return r

However, one can perform the modular reduction by firstly use $\mathsf{QAPartialRedRelaxed}$ and afterwards use $\mathsf{QAPartialRed}$. For $k = 50$, we take $J_k = \{23, 33, 44, 45, 46\}$, $J'_k = \{23, 33\}$ and $\bar{J}'_k = \{44, 45, 46\}$. First we apply the algorithm $r \leftarrow \mathsf{QAPartialRedRelaxed}(a, q, J'_k)$:

1. $\texttt{quo\_approx} \leftarrow (a >> 23) + (a >> 33)$
2. $r \leftarrow a - \texttt{quo\_approx} \cdot q$
3. return r

Due to Proposition 6 we got

$$r \leq \left\lfloor \left\lfloor \sum_{i=0}^{k-1} \left\{ \frac{2^i}{q} \right\} \right\rfloor + \sum_{j \in \bar{J}'_k} \lfloor (2^k - 1)/2^j \rfloor \right\rfloor \cdot q = 114 \cdot q < 2^{32}$$

After this partial reduction our result is lower than 2^{32}. Then, we re-define J_k as $J_k = \{23, 33\}$ and by applying, like in the previous key generation example, $\mathsf{QAPartialRed}(r, q, J_k)$ and a final subtraction we got the expected reduction. In the following we denote by $\mathsf{QARedRelaxed}$ the combination of these reductions.

We compare the Quotient Approximation Algorithm with the Barrett modular reduction.

For Crandall algorithm, the modular reduction is done with three iterations in the "while" loop and a final subtraction. After the second iteration, the result fits on 32 bits.

For the Barrett algorithm, we use the parameters $\alpha = 28, \beta = -2$ and $m = \lfloor 2^{51}/q \rfloor$. These parameters ensure that the temporary variables are encoded on 58 bits rather than 50 bits but fits on 2 machine words. The reduction requires at most one final subtraction.

For Solinas algorithm, the reduction of $a < 2^{50}$ modulo $q = 2^{23} - 2^{13} + 1$ is given by $a \mod q = T + S_1 + S_2 + S_3 + S_4 - (D_1 + D_2 + D_3 + D_4)$, where $+$ and $-$ are modular operations and $T = a \,\&\, \texttt{0x7fffff}$, $S_1 = (a \,\&\, \texttt{0x1ff800000}) >> 10$, $S_2 = (a \,\&\, \texttt{0x7fe00000000}) >> 20$, $S_3 = (a \,\&\, \texttt{0x7f80000000000}) >> 30$, $S_4 = (a \,\&\, \texttt{0x7c00000000000}) >> 46$, $D_1 = (a \,\&\, \texttt{0x3fffff800000}) >> 23$, $D_2 = (a \,\&\, \texttt{0x7c00000000000}) >> 32$, $D_3 = (a \,\&\, \texttt{0x3ffe00000000}) >> 33$, $D_4 = (a \,\&\, \texttt{0x7f80000000000}) >> 43$. In practice, S_4 and D_4 are computed as $S_4 = D_2 >> 14, D_4 = S_3 >> 13$. Most of the masks are sparse (especially lower bytes), therefore operations can be done on one machine word instead of two.

The Table 2 describes the number of operations required for each reduction algorithm. The operations are performed on a machine word (32 bits) or 2 machine words (64 bits). For the multiplication on 64 bits we consider that we multiply two integers encoded on a word and the result is encoded on two machine words.

	Operation on 64 bits				Operation on 32 bits				
	Mult.	Add/Sub	Shift	And	Mult.	Add/Sub	Shift	And	Cond. Sub
$\texttt{Barrett}_{28,-2}$	2	1	2	0	0	0	0	0	1
`Solinas`	0	0	2	2	0	8	6	5	2
`Crandall`	1	1	2	0	2	2	1	3	1
`QARed`	1	1	1	0	0	4	4	0	3
`QARedRelaxed`	1	1	1	0	1	2	2	0	1

Table 2: Complexity of the reduction algorithms

In order to compare the reduction algorithms we convert all operations on 64 bits to ones on 32 bits.

- A multiplication with result on 64 bits requires 4 multiplications and 3 additions on 32 bits.
- Add/sub on 64 bits requires 2 add/sub on 32 bits (addition/subtraction of the carry/borrow and the two most/least significant words can generally be performed with a single instruction, at the same cost as a simple add/sub)
- A shift on 64 bits requires 2 shifts and 1 addition on 32 bits. Generally, a shift on 64 bits requires 2 additions on 32 bits rather than 1. However, in our context the shift operation ensures that the result fits on a machine word. Therefore, only one addition is required.
- An AND on 64 bits requires 2 AND on 32 bits.

The complexities on 32 bits are described in Table 3.

	Operation on 32 bits				
	Mult.	Add/Sub	Shift	And	Cond. Sub
$\texttt{Barrett}_{28,-2}$	8	10	4	0	1
`Solinas`	0	10	10	9	2
`Crandall`	6	9	5	3	1
`QARed`	4	10	6	0	3
`QARedRelaxed`	5	8	4	0	1

Table 3: Complexity with machine word operations of the reduction algorithms

In this context Quotient Approximation reduction (relaxed version) is more efficient than Barrett and Crandall algorithms. The comparison between Solinas and QA algorithms depends on the component. Indeed, if 5 multiplications where both the result and each operand fit in a register costs less than 9 AND, 2 add, 1 conditional subtraction and 6 shifts, then Quotient Approximation is faster than Solinas. Otherwise Solinas is faster.

4 Conclusion

In this paper we introduced the modular reduction Quotient Approximation. This reduction is modulo-dependent and does not use temporary variable larger than the coefficient to reduce. In addition, Quotient Approximation is quite flexible allowing an optimized reduction regarding the context. Indeed, the targeted quotient can be computed more or less precisely to obtain the most efficient reduction. In our study, an application of this flexibility has been shown on post-quantum cryptosystems such as Kyber and Dilithium. Quotient Approximation can find other applications on cryptography like elliptic curve cryptosystems.

References

[ABD+20] Roberto Avanzi, Joppe Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M. Schanck, Peter Schwabe, Gregor Seiler, and Damien Stehlé. CRYSTALS-Kyber, 2020. Available at https://csrc.nist.gov/Projects/Post-Quantum-Cryptography/Round-3-Submissions.

[AHH+19] Martin R Albrecht, Christian Hanser, Andrea Hoeller, Thomas Pöppelmann, Fernando Virdia, and Andreas Wallner. Implementing RLWE-based Schemes Using an RSA Co-Processor. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 169–208, 2019.

[Bar86] Paul Barrett. Implementing The Rivest Shamir And Adleman Public Key Encryption On A Standard Digital Signal Processor. *CRYPTO' 86. CRYPTO 1986. Lecture Notes in Computer Science, vol 263. Springer, Berlin, Heidelberg.*, pages 1156–1158, 1986.

[BDL+19] Shi Bai, Léo Ducas, Tancrède Lepoint, Vadim Lyubashevsky, Peter Schwabe, Gregor Seiler, and Damien Stehlé. CRYSTALS-Dilithium, 2019. Available at https://csrc.nist.gov/Projects/Post-Quantum-Cryptography/Round-3-Submissions.

[BRvV21] Joppe W Bos, Joost Renes, and Christine van Vredendaal. Post-quantum cryptography with contemporary co-processors. *USENIX*, 2021.

[BZ10] Richard P. Brent and Paul Zimmermann. Modern computer arithmetic (version 0.5.1). *CoRR*, abs/1004.4710, 2010.

[CDH+19] Cong Chen, Oussama Danba, Jeffrey Hoffstein, Andreas Hülsing, Joost Rijneveld, John M Schanck, Peter Schwabe, William Whyte, and Zhenfei Zhang. NTRU - Algorithm Specifications and Supporting Documentation, 2019. Available at https://csrc.nist.gov/Projects/Post-Quantum-Cryptography/Round-3-Submissions.

[Cra27] Richard E. Crandall. Method and Apparatus for Public Key Exchange in a Cryptographic System, 1992-10-27. https://patentimages.storage.googleapis.com/11/9b/b8/75aa2cab01785d/US5159632.pdf.

[GG03] Joachim Von Zur Gathen and Jurgen Gerhard. *Modern Computer Algebra*. Cambridge University Press, USA, 2 edition, 2003.

[GMR21] Aurélien Greuet, Simon Montoya, and Guénaël Renault. On Using RSA/ECC Coprocessor for Ideal Lattice-Based Key Exchange. In *COSADE 2021*, 2021.

[Kon10] Yinan Kong. Optimizing the improved barrett modular multipliers for public-key cryptography. 12 2010.

[MAA+20] Dustin Moody, Gorjan Alagic, Daniel C Apon, David A Cooper, Quynh H Dang, John M Kelsey, Yi-Kai Liu, Carl A Miller, Rene C Peralta, Ray A Perlner, Angela Y Robinson, Daniel C Smith Tone, and Jacob Alperin Sheriff. Status Report on the Second Round of the NIST Post-Quantum Cryptography Standardization Process. Technical report, National Institute of Standards and Technology, July 2020.

[Mon85] Peter L Montgomery. Modular Multiplication Without Trial Division, 1985.

[MVOV18] Alfred J Menezes, Paul C Van Oorschot, and Scott A Vanstone. *Handbook of Applied Cryptography*. CRC press, 2018.

[PG12] Thomas Pöppelmann and Tim Güneysu. Towards Efficient Arithmetic for Lattice-Based Cryptography on Reconfigurable Hardware. In Alejandro Hevia and Gregory Neven, editors, *Progress in Cryptology – LATINCRYPT 2012*, pages 139–158, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[Sol99] Jerome A. Solinas. Generalized Mersenne Numbers. Technical report, Dept. of C&O, University of Waterloo, 1999. Available at https://cacr.uwaterloo.ca/techreports/1999/corr99-39.pdf.

[Sol11] Jerome A. Solinas. *Generalized Mersenne Prime*, pages 509–510. Springer US, Boston, MA, 2011.

[VSRDK] Frederik Vercauteren, Sujoy Sinha Roy, Jan-Pieter D'Anvers, and Angshuman Karmakar. SABER: Mod-LWR based KEM. Available at https://csrc.nist.gov/Projects/Post-Quantum-Cryptography/Round-3-Submissions.