Journal of Cryptographic Engineering
https://doi.org/10.1007/s13389-022-00303-z

REGULAR PAPER

Streamlined NTRU Prime on FPGA

Bo-Yuan Peng[1,2] · Adrian Marotzke[3,4] · Ming-Han Tsai[1] · Bo-Yin Yang[2] · Ho-Lin Chen[1]

Received: 28 October 2021 / Accepted: 4 October 2022

Abstract
We present a novel full hardware implementation of Streamlined NTRU Prime, with two variants: a high-speed, high-area implementation and a slower, low-area implementation. We introduce several new techniques that improve performance, including a batch inversion for key generation, a high-speed schoolbook polynomial multiplier, an NTT polynomial multiplier combined with a CRT map, a new DSP-free modular reduction method, a high-speed radix sorting module, and new encoders and decoders. With the high-speed design, we achieve the to-date fastest speeds for Streamlined NTRU Prime, with speeds of 5007, 10,989, and 64,026 cycles for encapsulation, decapsulation, and key generation, respectively, while running at 285 MHz on a Xilinx Zynq Ultrascale+. The entire design uses 40,060 LUT, 26,384 flip-flops, 36.5 Bram, and 31 DSP.

Keywords NTRU Prime · Hardware Implementation · Lattice Cryptography · Post-Quantum Cryptography · FPGA

1 Introduction

With the advent of quantum computers, many cryptosystems would become insecure. In particular, quantum computers would completely break many public-key cryptosystems, including RSA, DSA, and elliptic curve cryptosystems. Due to this concern, the National Institute of Standards and Technology (NIST) began soliciting proposals for post-quantum cryptosystems [17]. The algorithms solicited are divided into public-key encryption (key exchange) and digital signature. The NIST Post-Quantum Cryptography Standardization Process has entered the third phase, and NTRU Prime [7] is one of the candidates of key encapsulation algorithms, as an alternate candidate. Since hardware implementations will be an important factor in the evaluation, it is important to research hardware implementations for various use cases.

NTRU Prime has two instantiations: Streamlined NTRU Prime and NTRU LPRime. In this paper, we implement the Streamlined NTRU Prime cryptosystem on Xilinx Artix-7 and Xilinx Zynq Ultrascale+ FPGA. We present two different versions: a high-performance, large-area implementation and a slower, compact implementation. Both implement the full cryptosystem, including all encoding and decoding, but without a TRNG. We also present several novel designs to implement the subroutines required by NTRU Prime, such as sorting, modular reduction, polynomial multiplication, and polynomial inversion.

✉ Bo-Yuan Peng
d06921017@ntu.edu.tw; bypeng@crypto.tw

Adrian Marotzke
adrian.marotzke@nxp.com; adrian.marotzke@tuhh.de

Ming-Han Tsai
r08943151@ntu.edu.tw

Bo-Yin Yang
by@crypto.tw

Ho-Lin Chen
holinchen@ntu.edu.tw

1 National Taiwan University, Taipei, Taiwan

2 Institute of Information Science, Academia Sinica, Taipei, Taiwan

3 NXP Semiconductors, Hamburg, Germany

4 Hamburg University of Technology, Hamburg, Germany

2 Background

2.1 Definitions

Streamlined NTRU Prime [7] defines the following polynomial rings:

$$\mathcal{R} = \mathbb{Z}[x]/(x^p - x - 1) \quad (1)$$
$$\mathcal{R}/q = (\mathbb{Z}/q)[x]/(x^p - x - 1) \quad (2)$$
$$\mathcal{R}/3 = (\mathbb{Z}/3)[x]/(x^p - x - 1) \quad (3)$$

The parameters (p, q, w) of Streamlined NTRU Prime satisfy the following:

$$p, q \in \mathbb{P} \tag{4}$$
$$w > 0, w \in \mathbb{Z}, 2p \geq 3w \tag{5}$$
$$q \geq 16w + 1 \tag{6}$$
$$x^p - x - 1 \text{ is irreducible in } \mathcal{R}/q \tag{7}$$

The recommended parameter set for Streamlined NTRU Prime is $sntrup761$:

$$p = 761, w = 286, q = 4591 \tag{8}$$

For this reason, we will focus on this parameter set during this paper. NTRU Prime also uses the following notations:

– Small: A polynomial of $\mathcal{R}$ has all of its coefficients in $(-1, 0, 1)$.
– Weight w: A polynomial of $\mathcal{R}$ has exactly w nonzero coefficients.
– Short: The set of small weight w polynomials of $\mathcal{R}$.
– Round: Rounding all coefficients of a polynomial to the nearest multiple of 3.
– $\text{Hash}_a(x)$: The SHA-512 hash of the byte array x, prepended by the single byte value a. Only the first 256 bits of the output hash are used.
– Encode and Decode: Streamlined NTRU Prime uses an encoding and decoding algorithm to transform polynomials in $\mathcal{R}/3$ and $\mathcal{R}/q$ to and from byte strings.

2.2 Streamlined NTRU Prime

The key generation of Streamlined NTRU Prime is described in Algorithm 1, and the encapsulation and decapsulation in Algorithms 2 and 3, respectively.

Algorithm 1: Streamlined NTRU Prime Key Generation [7]

```
1 Generate a uniform random small $g \in \mathcal{R}/3$ until $g$ is invertible in $\mathcal{R}/3$
2 Generate a uniform random $f \in$ Short
3 Generate a uniform random byte array $\rho$ of length $(p+3)/4$
4 $v \Leftarrow 1/g$ in $\mathcal{R}/3$
5 $K \Leftarrow g/(3f)$ in $\mathcal{R}/q$
6 $\bar{K} \Leftarrow$ Encode $K$
7 $\bar{k} \Leftarrow$ Encode $(f, v)$
8 return $(\bar{K}, (\bar{k}, \bar{K}, \rho, \text{hash}_4(\bar{K})))$ as (public key, secret key)
```

Algorithm 2: Streamlined NTRU Prime Encapsulation [7]

```
Input: public key $\bar{K}$
1 Generate a uniform random $r \in$ Short
2 $K \Leftarrow$ Decode $\bar{K}$
3 $c \Leftarrow$ Round $(Kr)$ in $\mathcal{R}/q$
4 $\bar{c} \Leftarrow$ Encode $c$
5 $\bar{r} \Leftarrow$ Encode $r$
6 $C \Leftarrow (\bar{c}, \text{hash}_2(\text{hash}_3(\bar{r}), \text{hash}_4(\bar{K})))$
7 return $(C, \text{hash}_1(\text{hash}_3(\bar{r}'), C))$ as (ciphertext, session key)
```

Algorithm 3: Streamlined NTRU Prime decapsulation [7]

```
Input: ciphertext $C$ , secret key $(\bar{k}, \bar{K}, \rho, \text{hash}_4(\bar{K}))$
1 $c \Leftarrow$ Decode $\bar{c}$
2 $f, v \Leftarrow$ Decode $\bar{k}$
3 $e \Leftarrow (3fc$ in $\mathcal{R}/q$ ) modulo 3
4 $r' \Leftarrow ev$ in $\mathcal{R}/3$
5 if $r'$ does NOT have weight $w$ then
6     $r' \Leftarrow (1, 1, \ldots, 1, 0, 0, \ldots, 0)$  // The first $w$ elements are 1, the rest 0
7 Redo Encapsulation with $\bar{K}$ and $r'$, compute new ciphertext $C'$
8 $\bar{r}' \Leftarrow$ Encode $r'$
9 if $C' = C$ then
10    return $\text{hash}_1(\text{hash}_3(\bar{r}'), C)$
11 else
12    return $\text{hash}_0(\text{hash}_3(\rho), C)$
```

2.3 Design consideration with FPGAs

Field-programmable gate arrays (FPGAs) are popular hardware implementation platforms as one can easily construct and prototype customized digital logic circuits, without the large cost of manufacturing ASICs. Most FPGAs provide several different general purposed resources which are either common in general logic circuits or are able to simulate or execute Boolean functions, which are constructed from basic logic gates. On hardware implementation with FPGAs, the utilization of these resources is one of the important standards of comparison among similar implementations. To “make an apples-to-apples comparison,” a specified FPGA platform is often assigned in a call-for-proposal project. NIST recommends that “(PQC submission) teams generally focus their hardware implementation efforts on Artix-7” as an FPGA platform [2]. Artix-7™ is a FPGA platform manufactured by Xilinx® . We will focus on Xilinx FPGAs in this paper, in particular Xilinx Zynq® Ultrascale+™ and Artix-7 FPGAs, but note that the philosophy of the design consideration remains the same if the resources are of similar types and structures, even when the FPGA manufacturer differs. Here, we introduce the main resources, of whose properties our design takes advantage, provided in FPGAs.

2.3.1 Look-up tables (LUTs)

Look-up tables are very basic units in popular FPGAs. A LUT is a combinational logic unit with usually 4–6 input bits and 1–2 output bits. Here, we denote an LUT with m input bits and n output bits as $\text{LUT}_{m,n}$. An LUT can be considered as a block of read-only memory. For example, an $\text{LUT}_{5,2}$ can be considered as a block with 32 cells, each of which contains 2 bits. Xilinx Zynq Ultrascale+ and Artix-7 provide LUT units which support both the functions of $\text{LUT}_{5,2}$ and $\text{LUT}_{6,1}$ [21,22].

Usually, LUTs are used to implement combinational digital circuit, but they are also useful to implement read-only memories (ROMs) and random-access memory, call distributed RAM. For example, to construct a 12-bit, 32-cell read-only memory unit, we will need 6 $\text{LUT}_{5,2}$ units. A 13-bit, 64-cell block costs 13 $\text{LUT}_{6,1}$ units.

2.3.2 Digital Signal Processing (DSP) slices

A DSP slice is an arithmetic unit which consists of one multiplier and some accumulators. The multiplier supports signed integer multiplication with many bits, and it costs a lot of LUTs to construct the same multiplier if the DSP slice is not applied. Xilinx Zynq Ultrascale+ provides DSP slices with 27×18-bit signed integer multipliers [26], and Xilinx Artix-7 provides DSP slices with 25×18-bit signed integer multipliers [23].

If we multiply two integers whose bit lengths are more than the limit one DSP slice can offer, we can either apply a pipeline approach, or connect two or more DSP slices in parallel. For example, to multiply a 24-bit signed integer with a 32-bit signed integer, we can connect 2 slices in parallel, or to multiply the multiplicand with the least significant 16 bits of the other integer and then with the most significant bits. If we can control the bit lengths of the integers we want to multiply, however, we are able to limit the bit lengths so that one DSP slice can handle the multiplication.

2.3.3 Block memories (BRAMs)

A block memory unit stores a certain large number of bits. Every BRAM provides several channels with partially customizable data widths during the hardware synthesis stage. We can read and/or write the data stored in one BRAM only via the channels. This fact means that we can access as many words simultaneously as the number of channels in one BRAM, and if we want to access more words at the same clock cycle, we need either to duplicate the data from the BRAM to another in advance, or to partition the data we want to store in two or more BRAMs.

Both Xilinx Zynq Ultrascale+ and Artix-7 provides BRAM units [24,25], each of which contains 36 kbits and two channels. Every BRAM unit can be divided into two blocks with 18 kbits, each of which in turn provides two channels, and the synthesis report records 0.5 BRAMs of utilization as long as a 18 kbits block is utilized. In both FPGAs, the data width of each 18 kbits block can be customized as 1, 2, 4, 9, or 18 bits.

2.4 Multiplication using Good's trick with NTT

Polynomial multiplication is one of the most important operations which needs to be carefully designed in NTRU Prime (the other is the polynomial inversion).

Polynomials in $\mathcal{R}/q$ can be written as

$$f(x) = \sum_{i=0}^{p-1} f_i x^i$$

where $-\frac{q-1}{2} \le f_i \le \frac{q-1}{2}$ for every i satisfying $0 \le i \le p-1$. The polynomial multiplication of two polynomial $f(x)$ and $g(x)$ in $\mathcal{R}/q$ is

$$f(x) \boxtimes g(x) \triangleq (f(x)g(x) \bmod^{\pm} q) \bmod x^p - x - 1$$

where we denote $r = n \bmod^{\pm} q$ (signed modulo) for any integer n and r if $-\frac{q-1}{2} \le r \le \frac{q-1}{2}$ and there exist an integer m such that $n = mq + r$. To reduce modulo $x^p - x - 1$ is easy since we only need to substitute x^j with $x^{j-p+1} + x^{j-p}$ for every $j \ge p$ and reduce the eventual polynomial into the form $\sum_{i=0}^{p-1} f_i x^i$. So the key is to evaluate $f(x)g(x) \bmod^{\pm} q$. For multiplying two polynomials of degree $p-1$, a fast-Fourier-transform-like approach can effectively reduce the number of integer multiplications we need, from $\mathcal{O}(p^2)$ to $\mathcal{O}(p \log p)$. Such an approach operating in a prime field $\mathbb{Z}/q$ but not complex numbers is a number theoretic transform (NTT).

NTT is usually a 2^k-point transformation method with a pre-determined positive integer k (written as $\text{NTT}_{2^k}(\cdot)$, and the inverse operation $\text{iNTT}_{1^k}(\cdot)$). For polynomials $f(x)$ and $g(x)$ of degree at most $2^k - 1$ and with at least 2^{k-1} zero coefficients, the polynomial multiplication can be implemented as

$$\begin{aligned} f(x)g(x) &= f(x)g(x) \bmod x^{2^k} - 1 \\ &= \text{iNTT}_{2^k}(\text{NTT}_{2^k}(f(x)) \odot \text{NTT}_{2^k}(g(x))) \end{aligned}$$

where $\odot$ is the point-wise multiplication. For NTRU Prime with $p = 761$, we need to pad 263 monomials with zero coefficients to the polynomials, making $\text{NTT}_{2^{11}}(\cdot)$ work.

Good [11] provides another approach, applying $\text{NTT}_{2^9}(\cdot)$ instead and then doing 9 degree-512 polynomial multiplications where the polynomials are with at least 256 zero coefficients. In this approach, we need only to pad 7 zeros to the polynomials. This idea was introduced in NTRU Prime originally in [1,8].

In the case $p = 761$, we regard the polynomial as of degree 767 instead, with the coefficients of the high-degree terms set to 0. Now since $f(x)g(x)$ is of degree of at most 1534, we have $f(x)g(x) = f(x)g(x) \bmod x^{3\cdot 512} - 1$. We set $x = yz$, and it can be shown that

$$f(x)g(x) = f(yz)g(yz) \bmod (y^3-1)(z^{512}-1)$$

In detail, we see that for the set of integer i in $[0, 1535]$, the mapping $i \equiv 513\ell + 512j \pmod{1536}$ to the set of the integer pair (j, ℓ) where $0 \le j \le 3$ and $0 \le \ell \le 511$ is one-to-one and onto. Then, $f(x)$ (and $g(x)$, same as follows) can be expressed as:

$$\begin{aligned}
f(x) &= \sum_{i=0}^{760} f_i x^i + \sum_{i=761}^{1535} 0x^i \equiv \sum_{i=0}^{1535} f_i y^{i \bmod 3} z^{i \bmod 512} \\
&= \sum_{j=0}^{2}\sum_{\ell=0}^{511} f_{(513\ell+512j) \bmod 1536} y^j z^\ell \\
&\equiv \left(\sum_{\ell=0}^{511} f_{513\ell \bmod 1536} \cdot z^\ell\right) \\
&\quad + \left(\sum_{\ell=0}^{511} f_{(513\ell+1024) \bmod 1536} \cdot z^\ell\right) y \\
&\quad + \left(\sum_{\ell=0}^{511} f_{(513\ell+512) \bmod 1536} \cdot z^\ell\right) y^2 \\
&\equiv \left(\sum_{\ell=0}^{511} f_{(\ell \bmod 3)2^9+\ell} \cdot z^\ell\right) \\
&\quad + \left(\sum_{\ell=0}^{511} f_{((\ell-1) \bmod 3)2^9+\ell} \cdot z^\ell\right) y \\
&\quad + \left(\sum_{\ell=0}^{511} f_{((\ell-2) \bmod 3)2^9+\ell} \cdot z^\ell\right) y^2 \\
&\qquad \pmod{(y^3-1)(z^{512}-1)}
\end{aligned}$$

Here, we define $f_{y^j}(z) \triangleq \sum_{\ell=0}^{511} f_{((\ell-j) \bmod 3)2^9+\ell} \cdot z^\ell$ for convenience, and then $f(x) \equiv f_{y^0}(z) + f_{y^1}(z)y + f_{y^2}(z)y^2 \pmod{(y^3-1)(z^{512}-1)}$. We can assert that $f_\cdot(z)$ and $g_\cdot(z)$ are all of z-degree 511 and with at least half of the coefficients being 0, so that $f_\cdot(z)g_\cdot(z)$ can be evaluated as

$$f_\cdot(z)g_\cdot(z) \equiv \mathrm{iNTT}_{2^9}(\mathrm{NTT}_{2^9}(f_\cdot(z)) \odot \mathrm{NTT}_{2^9}(g_\cdot(z)))$$

Then, $f(x)g(x)$ is given by

$$\begin{aligned}
h(x) &= f(x)g(x) \\
&\equiv (f_{y^0}(z) + f_{y^1}(z)y + f_{y^2}(z)y^2)(g_{y^0}(z) \\
&\quad + g_{y^1}(z)y + g_{y^2}(z)y^2) \\
&\equiv (f_{y^0}(z)g_{y^0}(z) + f_{y^1}(z)g_{y^2}(z) + f_{y^2}(z)g_{y^1}(z)) \\
&\quad + (f_{y^0}(z)g_{y^1}(z) + f_{y^1}(z)g_{y^0}(z) + f_{y^2}(z)g_{y^2}(z))y \\
&\quad + (f_{y^0}(z)g_{y^2}(z) + f_{y^1}(z)g_{y^1}(z) + f_{y^2}(z)g_{y^0}(z))y^2 \\
&\triangleq h_{y^0}(z) + h_{y^1}(z)y + h_{y^2}(z)y^2 \\
&\triangleq h(z, y) = \sum_{j=0}^{2}\sum_{\ell=0}^{511} h_{j\ell} z^\ell y^j \pmod{(y^3-1)(z^{512}-1)}
\end{aligned}$$

We can regard the polynomial multiplication of $h(x) = f(x)g(x)$ as a school-book multiplication with respect to y, where the coefficients of the powers of y's are the sum of products of the polynomials in z, which can be computed by NTT. Notice that for every $h_{j\ell}$ the index j directs to the coefficient polynomial of y^j, and the index ℓ directs to the coefficient of z^ℓ in each polynomial. To map back the coefficients of $h(z, y)$ to those of $h(x)$, we can see $h(x)$ is given by

$$h(x) = \sum_{i=0}^{1535} h_{(i \bmod 3),(i \bmod 512)} x^i$$

2.5 Chinese remainder theorem and NTT

To compute $\mathrm{NTT}_{2^k}(\cdot)$, we need to find a 2^k-th root of unity in the field $\mathbb{Z}/q$. Specifically, to apply Good's trick for $p = 761$ and $q = 4591$, we need to find a 512-th root of unity in $\mathbb{Z}/4591$. This is impossible since $4591 - 1 = 2 \cdot 3^3 \cdot 5 \cdot 17$ without the factor 512.

In [8], it is suggested to apply the Chinese remainder theorem (CRT) to resolve this issue. To make it clear how the CRT can be applied, the following two cases are considered: **Case 1** Polynomial multiplications used in the standard of NTRU Prime are multiplications with one *small* polynomial (coefficients are all -1, 0, or 1) and one $\mathcal{R}/q$ polynomial (coefficients are in the range $[-\frac{q-1}{2}, \frac{q-1}{2}]$, or $[-2295, 2295]$). If we use the school-book scheme, we can see that all of the coefficients in the polynomial multiplication without modulo q are ranged in $[-\frac{p(q-1)}{2}, \frac{p(q-1)}{2}]$, which is a 22-bit signed integer. If, instead, we want to apply Good's trick, we can choose two *good* primes (in whose finite fields we can find a 512-th root of unity) q_1 and q_2 such that $q_1q_2 > p(q-1)+1$. Then, we apply Good's trick separately.

For all coefficients of x^i's computed with an NTT in $\mathbb{Z}/q_1$ and $\mathbb{Z}/q_2$, respectively, say $h_{i,1}$ and $h_{i,2}$, we can get the eventual h_i by

$$\begin{aligned}
&h_{i,1}q_2'q_2 + h_{i,2}q_1'q_1 \\
&\quad \equiv ((h_{i,1}q_2') \bmod^{\pm} q_1)q_2 + ((h_{i,2}q_1') \bmod^{\pm} q_2)q_1 \triangleq h_i^{(0)} \\
&h_i \equiv h_i^{(0)} (\bmod^{\pm} q)
\end{aligned}$$

where $q'_1 \equiv q_1^{-1} \pmod{q_2}$ and $q'_2 \equiv q_2^{-1} \pmod{q_1}$. We can see that $h_i^{(0)}$ is in the range $[-q_1q_2 + \frac{q_1+q_2}{2}, q_1q_2 - \frac{q_1+q_2}{2}]$, and we need only to check if it is in $[-\frac{q_1q_2}{2}, \frac{q_1q_2}{2}]$ and tune up or down by q_1q_2. That is,

$$h_i = h_i^{(0)} + kq_1q_2, k \in \{-1, 0, 1\}$$

Notice that we will control the logic such that we always multiply a 25-bit signed integer with a 18-bit signed integer, as the built-in multipliers in Xilinx FPGAs we focused are at least signed 25 × 18-bit multipliers. Controlling the size of the multiplication in this manner provides the portability between high-end and low-end FPGAs and utilizes the built-in multiplier with a better effectiveness. This fact is important in the next case.

Case 2 In our implementation, batch inversion is applied (see Sect. 3.4). This makes multiplication with two $\mathcal{R}/q$ polynomials necessary. In this case, all coefficients in the polynomial multiplication without modulo q are ranged in $[-\frac{p(q-1)^2}{4}, \frac{p(q-1)^2}{4}]$, which is $[-4008206025, 4008206025]$, and then the coefficients are 33-bit signed integers. In this case, three *good* primes are picked. We have that

$$\begin{aligned}
&h_{i,1}q'_{23}q_2q_3 + h_{i,2}q'_{31}q_3q_1 + h_{i,3}q'_{12}q_1q_2 \\
&\quad \equiv ((h_{i,1}q'_{23}) \bmod^{\pm} q_1)q_2q_3 \\
&\qquad + ((h_{i,2}q'_{31}) \bmod^{\pm} q_2)q_3q_1 \\
&\qquad + ((h_{i,3}q'_{12}) \bmod^{\pm} q_3)q_1q_2 \\
&\quad \triangleq h_i^{(0)} \\
&h_i \equiv h_i^{(0)} (\bmod^{\pm} q)
\end{aligned}$$

where $q'_{23} \equiv (q_2q_3)^{-1} \pmod{q_1}, q'_{31} \equiv (q_3q_1)^{-1} \pmod{q_2}$ and $q'_{12} \equiv (q_1q_2)^{-1} \pmod{q_3}$. $h_i^{(0)}$ is (roughly) ranged in $[-\frac{3q_1q_2q_3}{2}, \frac{3q_1q_2q_3}{2}]$, and we can still check if it is in $[-\frac{q_1q_2q_3}{2}, \frac{q_1q_2q_3}{2}]$ and tune up or down by $q_1q_2q_3$. That is,

$$h_i = h_i^{(0)} + kq_1q_2q_3, k \in \{-1, 0, 1\}$$

We choose $q_1 = 7681$, $q_2 = 12{,}289$ and $q_3 = 15{,}361$ here. In this case, $q'_{23} = 2562 = (\text{A02})_{16}$, $q'_{31} = 8182 = 2^{13} - (\text{A})_{16}$ and $q'_{12} = 10 = (\text{A})_{16}$, making all three of $h'_{i,a} = (h_{i,a}q'_{bc}) \bmod^{\pm} q_a$ can be done with simple addition or subtraction only followed by a modulo operation. This makes all $h'_{i,a}$ represented as 14-bit signed integers. Multiplying the remaining q_bq_c can be done also by one 25 × 18-bit multiplier since in this configuration

$$\begin{aligned}
&h'_{i,q_1}q_2q_3 + h'_{i,q_2}q_3q_1 + h'_{i,q_3}q_1q_2 \\
&\quad = h'_{i,q_1} \cdot 188771329 + h'_{i,q_2} \cdot 117987841 + h'_{i,q_3} \\
&\qquad \cdot 94391809 \\
&\quad = h'_{i,q_1}(184347 \cdot 2^{10} + 1) + h'_{i,q_2}(230445 \cdot 2^9 + 1) \\
&\qquad + h'_{i,q_3}(184359 \cdot 2^9 + 1) \\
&\quad = h'_{i,q_1}((\text{2D01B})_{16} \cdot 2^{10} + 1) + h'_{i,q_2}((\text{3842D})_{16} \cdot 2^9 + 1) \\
&\qquad + h'_{i,q_3}((\text{2D027})_{16} \cdot 2^9 + 1)
\end{aligned}$$

2.6 Montgomery's trick

Montgomery's trick is a method to accelerate inversion by doing batch inversion [16]. This allows us to replace n inversions in a ring with a single inversion, together with $3n - 3$ multiplications. Montgomery's trick is described in Algorithm 4. The trick can lead to a significant speedup as long as multiplication is at least 3 times as fast as a single inversion, and one has enough storage space to store the intermediate products. Batch inversion with Montgomery's trick for NTRU Prime was already proposed in the original paper [4]. It was recently implemented for fast key generation in an integration of NTRU Prime into OpenSSL [3]. There, for the parameter set sntrup761 and a batch size of 32, it led to a key generation speed of 156,317 cycles per key, compared to the non-batch 819,332 cycles.

Algorithm 4: Description of Montgomery's trick for batch inversion

```
Input  : $n$: the batch size, $f_x$: an array of $n$ numbers to be
         inverted
Output: The array of $n$ inverted $f_x^{-1}$
1  $a_1 \leftarrow f_1$ ;
2  for i from 2 to n do
3  |  $a_i \leftarrow a_{i-1} \cdot f_i$ ;
4  end
5  Compute inverse $a_n^{-1}$ ;
6  for i from n to 2 do
7  |  $f_i^{-1} \leftarrow a_i^{-1} \cdot a_{i-1}$ ;
8  |  $a_{i-1}^{-1} \leftarrow a_i \cdot f_i$ ;
9  end
10 $f_1^{-1} \leftarrow a_1^{-1}$ ;
11 return $(f_1^{-1}, \dots, f_n^{-1})$
```

3 Hardware implementation

In this section, we describe the basic functionality and architecture of all core functions and modules of our Streamlined NTRU Prime implementation.

3.1 Parallel schoolbook multiplier

This multiplier use a massively parallel version of the schoolbook multiplication algorithm. It consist of an LFSR, an accumulator register, and a large number of multiply accumulate units.

The use of schoolbook multiplication both for NTRU Prime [15] and for other lattice KEM [10,18] is not new. Two different implementations, based on the same overall design architecture, are presented in this paper: the first is a high-speed, high-area implementation and the second is a much smaller, but also slower implementation. Both are similar with regard to the speed-area product. They also have very simple memory access patterns. The differences between the two is that the faster implementation stores all values in flip-flops, whereas the compact implementation uses distributed RAM. The architecture is shown in Fig. 1.

The high performance and efficiency of this design is based on the fact that in Streamlined NTRU Prime, all multiplications are always with one polynomial in $\mathcal{R}/3$, and the second either also in $\mathcal{R}/3$ or in $\mathcal{R}/q$. Multiplication with both polynomials in $\mathcal{R}/q$ do not normally occur. (The only exception here is during the batch inversion using Montgomery's trick, see Sect. 3.4.) This idea was previously presented in [10,18], and allows a number of optimizations. The fact that one polynomial is always in $\mathcal{R}/3$ allows the individual multiply accumulate (MAC) units to be very simple, as only a very small number of bit operations are needed. This in turn leads to a very small footprint in the FPGA. In addition, we do not perform any modular reduction at this step. Its algorithmic description can be found in Algorithm 5.

Before the multiplication starts, the small $\mathcal{R}/3$ polynomial is loaded into an LFSR of length p, with the tap points set to correspond to the polynomial of the NTRU Prime ring, $\mathcal{R}/3 = (\mathbb{Z}/3)[x]/(x^p - x - 1)$. For this reason, 3 bits are needed per coefficient, as the tap points can lead to coefficients in the range from -2 to 2. Once the $\mathcal{R}/3$ polynomial is fully shifted into the LFSR, the multiplication can begin. During multiplication, one coefficient from the $\mathcal{R}/q$ polynomial is retrieved from BRAM at a time. This coefficient is then multiplied with every single coefficient in the LFSR, and added coefficient-wise to an accumulator register. The LFSR is then shifted once, and the next coefficient from the $\mathcal{R}/q$ polynomial is retrieved. This repeats for every coefficient from the $\mathcal{R}/q$ polynomial. After this, the accumulator register contains the completed polynomial multiplication. The register contents are then sent to the multiplier output, where they are taken modulo q. Because of the LFSR, no additional polynomial modulo reduction is required.

For the high-speed schoolbook multiplier, p MAC units are instantiated, and as a result, one coefficient from the $\mathcal{R}/q$ polynomial can be processed per clock cycle. For the compact implementation using distributed RAM, 24 MAC units are instantiated. This number comes from the value of p, and the size of the smallest distributed RAM blocks. In Xilinx FPGA's, the LUT can be configured as 32-bit dualport RAM, with one read/write port, and one read-only port. With $p = 761$, and $\lceil 761/24 \rceil = 32$, it means that 24 MAC units pack the RAM as densely as possible. This means that every 32 clock cycles, a new coefficient from the $\mathcal{R}/q$ polynomial is processed, and the multiplier thus also takes 32 times as many cycles.

Algorithm 5: Single coefficient multiply accumulate (MAC) algorithm. Note that no modulo calculation is performed here. The 23 bits are large enough so that no overflow can occur.

```
Input : $a$: a 23-bit signed number, $b$: a 13-bit signed number, $c$:
        a 3-bit signed number with $-2 \geq c \geq 2$
Output: The 23-bit result $a + b \cdot c$
1 $r_{-2} \leftarrow -b \ll 1$ ;
2 $r_{-1} \leftarrow -b$ ;
3 $r_0 \leftarrow 0$ ;
4 $r_1 \leftarrow b$ ;
5 $r_2 \leftarrow b \ll 1$ ;
6 return $a + r_c$
```

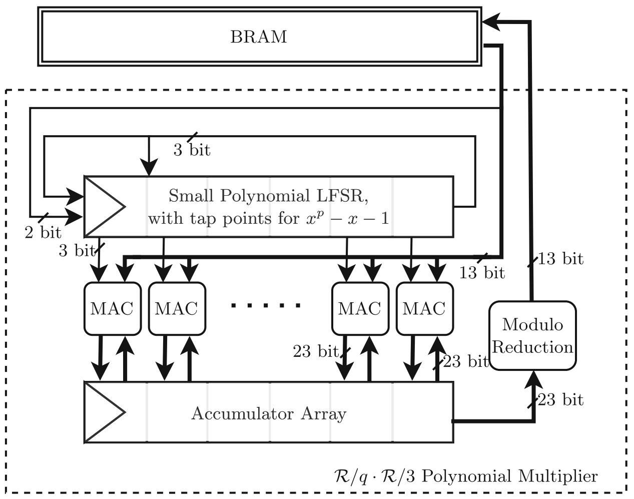


Fig. 1 Architecture of the parallel schoolbook polynomial multiplier for the parameter set sntrup761. The accumulator array has a size of $p \cdot 23$ bits. The blocks with the label MAC are described in Algorithm 5. The difference between the high-speed and the low-area multiplier are in the number of MAC units, and whether the accumulator array and small polynomial LFSR are implemented in flip-flops or in distributed RAM

It takes p clock cycles to shift the $\mathcal{R}/3$ polynomial into the LFSR. It also takes p clock cycles to shift the result out of the accumulator array, during which the accumulator array is also set to 0. Both of these operation can be interleaved to save time, i.e., a new $\mathcal{R}/3$ polynomial can be shifted in, while the accumulator array is shifted out. As a result, for $p = 761$, the high-speed multiplication takes 1522 cycles, otherwise 2283 cycles.

3.2 Architecture of $\mathcal{R}/q \cdot \mathcal{R}/q$ NTT multiplier

The architecture NTT multiplication employing Good's trick and a CRT map is shown in Fig. 2, which is modified from the NTT/INTT architecture from [27].

Coefficients in polynomials $f(x)$ and $g(x)$ are partitioned into those of $f_{y^0}(z)$, $f_{y^1}(z)$, $f_{y^2}(z)$, $g_{y^0}(z)$, $g_{y^1}(z)$, and $g_{y^2}(z)$, as are mentioned in Sect. 2.4. Each z-polynomial is put into bank 0 and 1, and with proper design of the four address generators to the reading and writing channels of bank 0 and 1, the z-polynomials are passed through the 3 Butterfly units (for $\mathbb{Z}/7681$, $\mathbb{Z}/12{,}289$, and $\mathbb{Z}/15{,}361$, respectively), and the corresponding NTT vectors are calculated. Bank 2 is then used to store the result of the summation of the point multiplications. The content in bank 2 contains the NTT vectors of $h_{y^0}(z)$, $h_{y^1}(z)$, and $h_{y^2}(z)$. The NTT vectors are then calculated with three INTT operations, making $h(x) = f(x)g(x)$ ready, where each coefficient is of 3-tuple with each entry representing the coefficient modulo 7681, 12,289, 15,361, respectively. The CRT operation is then done to find each coefficient modulo 4591, and then the reduction of $x^p - x - 1$ is computed, getting the final $h^{(r)}(x) = h(x) \bmod x^p - x - 1$ ready.

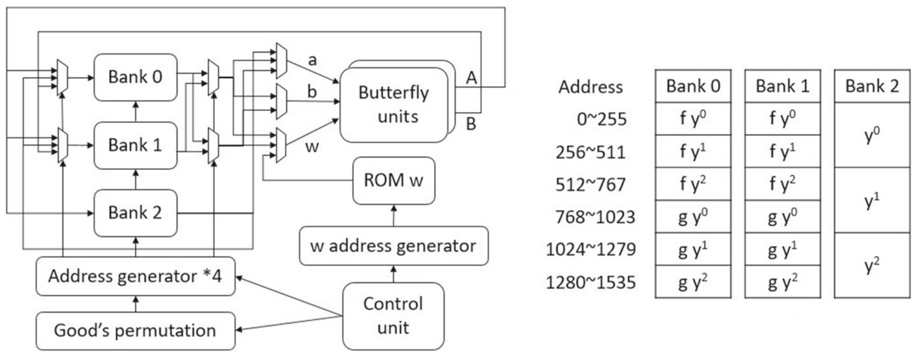
This multiplier is used for the $\mathcal{R}/q \cdot \mathcal{R}/q$ multiplication during batch inversion (see Sect. 3.4), and takes 35,463 clock cycles. The control unit which controls the unit consists of the following stages: *load*, *NTT*, *point_mul*, *reload*, *INTT*, *crt*, and *reduce* stages. When the product of the polynomials is ready, the control unit falls into *finish* stage, and the result can be fetched out of the multiplier.

- In the *load* stage, the coefficients of the polynomials f and g to be multiplied are stored in bank 0 and bank 1, and the address is determined by Good's permutation and address generators. 3090 clock cycles are consumed in this stage.
- In the *NTT* stage, read one data from bank 0 and one data from bank 1, pass them to the butterfly unit, and write the output A and B back to the same address of bank 0 and bank 1. There are six NTT operations with 512 points to be done, and each NTT operation needs 2315 cycles ($256 \times 9 + \epsilon$). It takes 13890 cycles in this stage.
- In the *point_mul* stage, read out two data from the same bank and multiply it and decide which address to write to bank 2 according to the power of y modulo $y^3 - 1$. We carefully write the result into bank 2 for the first time, and for the second and third time, we read the value from bank 2 in advance, add it into the result of the point-multiplication, and save it back to bank 2. With this design, we complete the addition of the 3 sets of point multiplication which influence the value of $h(x)$ in this stage. There are nine sets of 512-point multiplications to be calculated, and it takes 4616 ($512 \times 9 + \epsilon$) cycles in this stage.
- In the *reload* stage, read data from bank 2 and write it back to bank 0 and bank 1 after the nine point multiplications. A total of 1541 cycles are consumed here.
- In the *INTT* stage, the process is similar as in the *NTT* stage. The differences are that the butterfly units and the ω address generator are now operated in the inverse mode. Now, there are three INTT operations to be done, and 6945 cycles are consumed in this stage.
- In the *crt* stage, the DSP slices in the three butterfly units are applied to calculate the partial result $h'_{i,a} q_b q_c$ using the CRT. All of the partial results are then added as one integer, which is the input of the modulo q unit. After this, $f(x)g(x)$ is ready but without modulo $x^p - x - 1$ applied. It takes 3080 cycles to complete this stage.
- In the *reduce* stage, $h^{(r)}(x) = f(x)g(x) \bmod x^p - x - 1$ is evaluated. A register h_d is constructed to help the latter addition. Each coefficient (denoted as h_i) of x^i in $f(x)g(x)$ for $0 \le i \le 760$ is copied into bank 2 sequentially, and $h_i \triangleq 0$ during the duplication of the lower coefficients. h_i for $761 \le i \le 1520$ is then sequentially loaded, and h_{i-761} is loaded from bank 2 simultaneously. $h^{(r)}_{i-761} \triangleq (h_i + h_d + h_{i-761}) \bmod q$ is computed and saved into bank 2 at the next cycle, and $h_d \triangleq h_i$ simultaneously. After h_{1520} is processed, h_{760} is loaded, and $h^{(r)}_{760} \triangleq (h_d + h_{760}) \bmod q$ is computed and saved into bank 2. This stage takes 1530 cycles.
- $h^{(r)}(x)$ is ready in bank 2 at the *finish* stage. To reduce the critical paths while fetching data from bank 2, a pipeline approach in the address assignment integrated with the unit is applied, which results in some overhead. Fetching $h^{(r)}(x)$ takes 770 cycles.

We inspect in detail how the coefficients in the polynomial $f_{y^i}(z)$ and $g_{y^i}(z)$ are stored in the memory banks. One z-polynomial requires 512 cells as the storage of coefficients, and we save half of the coefficients in 256 cells of bank 0 and the other half in 256 cells of bank 1. This design is to feed the inputs simultaneously into the butterfly units, and an efficient in-place memory addressing is introduced in [14], which provides the formula of bank index $B(\cdot)$ and the lower bits of the address $A_l(\cdot)$. The higher bits of the address $A_h(\cdot)$ just indicate which polynomial it is. The bank index and address are given by

$$\begin{aligned} B(z^i, h_{y^j}(z)) &= i[8] \oplus i[7] \oplus \cdots \oplus i[0] \\ A_l(z^i, h_{y^j}(z)) &= i[8:1] \\ A_h(z^i, h_{y^j}(z)) &= \begin{cases} j, & h = f \\ j+3, & h = g \end{cases} \\ A(z^i, h_{y^j}(z)) &= 2^8 A_h(.) + A_l(.) \end{aligned}$$

It should be noted that in the *reload* stage and at the end of multiplication, as NTT itself re-arranges the order of the coefficients such that the address in one polynomial is *bit*



Address	Bank 0	Bank 1	Bank 2
0~255	f y^0	f y^0	y^0
256~511	f y^1	f y^1	
512~767	f y^2	f y^2	y^1
768~1023	g y^0	g y^0	
1024~1279	g y^1	g y^1	y^2
1280~1535	g y^2	g y^2	

Fig. 2 Architecture of Good's trick NTT multiplication

reversed, the lower 9 bits of the address need to be reversed. The higher 3 bits do not join the bit reversal.

3.3 Generation of short polynomials

During the encapsulation and key generation in NTRU Prime, a so-called short polynomial has to be created. For this, the original NTRU Prime paper suggests using a sorting network [4], and using a sorting algorithm is a well-established method to randomly shuffle a list in constant-time [10,20]. In our case, a list of p 32-bit random numbers is created. Of the first w, the least significant bit is set to 0 so that the number is always even. For the others, the lowest two bits are set to $(0, 1)$. This list of numbers is then sorted, after which the upper 30 bits are discarded. The remaining two-bit numbers are then subtracted by one. As a result, exactly w elements are either 1 or -1, and the rest are all zero. An alternative method for generating short polynomials would be using a shuffling algorithm such as Fisher–Yates, as used by a Dilithium hardware implementation [13]. However, in Dilithium, a *public* polynomial is sorted, wherase in NTRU Prime, a *secret* polynomial is sorted, and thus requires a constant-time algorithm. As Fisher–Yates shuffle is difficult to implement in constant-time [10,20], we do not consider it an option.

The reference C implementation of NTRU Prime [7], as well as the hardware implementation in [15], use a constant-time sorting network. However, on an FPGA, we can use a faster method in the form of the radix sorting algorithm [12]. Radix sort is an extremely fast sorting algorithm, offering $\mathcal{O}(n)$ speed compared to the $\mathcal{O}(n \log n)$ of the sorting network used in [7,15]. But radix sort has the drawback of having input-dependent addressing, which would disqualify it for memory architectures that have a cache due to side-channel leakages. As the BRAMs on an FPGA do not have any sort of cache, we can safely implement the algorithm. Our implementation is based on the radix sorting algorithm found in the SUPERCOP benchmark suite [5]. As a result, we can generate a new short polynomial in 4837 cycles. A comparison of different sorting algorithms is in Table 3.

A further optimization we have implemented is the pregeneration of short polynomials. As short polynomials can be generated independently of the operation (encapsulation or key generation) or any other input (e.g. the public key), we can pregenerate a short polynomial, instead of generating it on-demand. This pregenerated short polynomial is then cached, and is immediately output when the encapsulation or key generation starts. Once it has been output, we can use the rest of the time spent on encapsulation or key generation to pregenerate a new short polynomial for the next operation. This in particular speeds up encapsulation, as the rest of the modules do not have to wait until the sorting has completed. Note that this pregeneration is also possible for NTRU, but not Saber or Kyber, and would allow for a similar speed up.

The one case where this pregeneration would not be possible is if an encapsulation starts immediately after power on. In that case, the encapsulation would have to wait until a short polynomial is generated. Further encapsulations however would be able to use a cached pregenerated short polynomial, so only the very first encapsulation would be delayed. However, the describes scenario is unlikely to occur in the real word, as it disregards aspects such as the loading of the public key from a flash storage, which will likely take longer than the 4837 cycles that the sorting takes.

3.4 Batch inversion using Montgomery's trick

To accelerate the inversion during key generation, we employ batch inversion using Montgomery's trick. For the polynomial inversion itself, we use the constant-time extended GCD algorithm from [6]. This algorithm uses a constant number of "division steps" (or divsteps) to calculate the inverse of the input polynomial. This algorithm is used by the reference implementation of NTRU Prime, and was also used in a previous hardware implementation [15]. We extend it by allowing a configurable number of divsteps per clock cycle. Increasing the number of divsteps per clock cycle proportionally decreases the number of cycles. The architecture of the $\mathcal{R}/q$ inversion is shown in Fig. 3. We do not consider alternative inversion methods, such as Fermat's method or

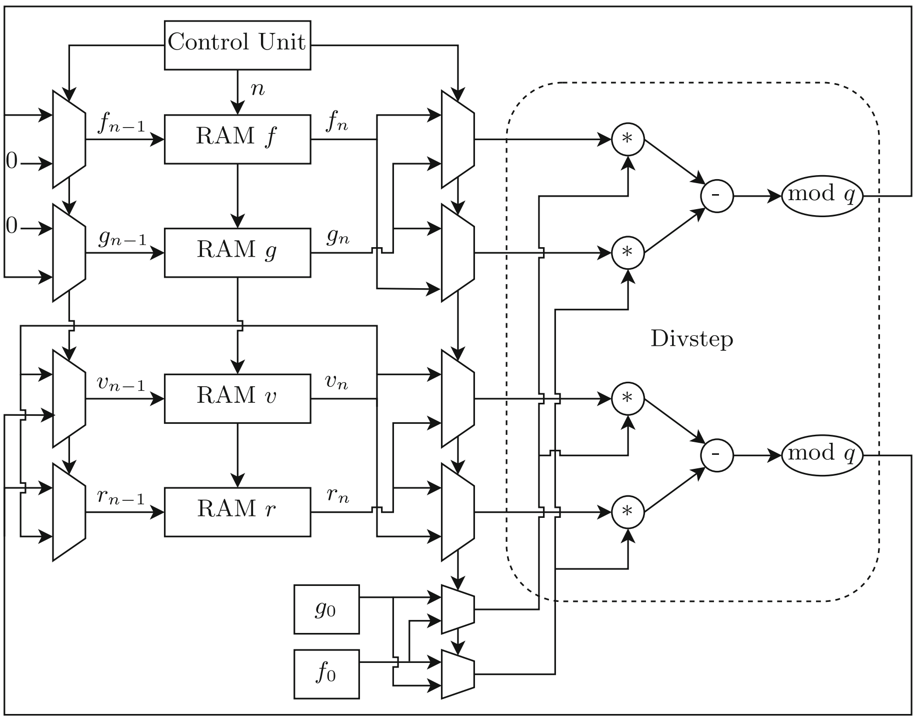


Fig. 3 Architecture of the $\mathcal{R}/q$ inversion module using the extended GCD algorithm. The to-be-inverted polynomial is stored in RAM g. At the start of the algorithm, RAM v stores an all-zero polynomial, RAM r the polynomial $(3^{-1} \bmod q, 0, \ldots, 0)$, and RAM f the polynomial $(1, 0, \ldots, 0, -1, -1)$. The final result is stored in RAM v. The section marked "Divstep" is the part that is replicated when multiple divsteps are performed per clock cycle. This also requires wider read/write ports to the RAM. The architecture of the $\mathcal{R}/3$ inversion is identical, other that all arithmetic operations are performed in $\mathcal{R}/3$

Hensel lifting, as they are either slower, not constant-time or are not applicable to the rings used in NTRU Prime [3,6].

In our implementation, we only implement batch inversion for the inversion in $\mathcal{R}/q$. For inversion in $\mathcal{R}/3$, it is more efficient to simply increase the number of parallel divsteps, as the divstep operation in $\mathcal{R}/3$ is trivial (see Table 6). With, e.g., 32 parallel divsteps, an inversion in $\mathcal{R}/3$ takes 47,166 cycles. The inversion in $\mathcal{R}/3$ also has the potential of having non-invertable polynomials. We skip the invertability check, and simple redo the inversion with a new polynomial in case of a non-invertable polynomial. However, for batch inversion, we would have to check every polynomial for invertability, as a single non-invertable polynomial would force us to redo the entire batch.

Doing batch inversion has an additional caveat: it requires n multiplications where both polynomials are in $\mathcal{R}/q$ (line 7 in Algorithm 4). This is an issue, as the polynomial multiplier for NTRU Prime normally always has one operand in $\mathcal{R}/3$. This means we cannot use the our schoolbook multiplier, as the multiplier has optimizations that rely on one operand being in $\mathcal{R}/3$. As a result, we add a second multiplier to our design, namely the NTT multiplier with a CRT map for the $\mathcal{R}/q \cdot \mathcal{R}/q$ multiplication.

Due to the additional $\mathcal{R}/q \cdot \mathcal{R}/q$ multiplier, batch inversion is not automatically the optimal way of inverting polynomials in $\mathcal{R}/q$. This is because the additional multiplier consumes hardware resources that could otherwise be used to implement more parallel divsteps for the $\mathcal{R}/q$ inversion. In addition, larger batch sizes require more BRAM to store intermediate results. Depending on the speed and hardware consumption of non-batch inversion, batch inversion and multiplication, respectively, together with the available hardware resources and batch size, the optimal solution varies. A contour plot that shows the minimum batch size needed for Montgomery's trick to be worthwhile for different inversion and multiplication speeds is shown in Fig. 4. In practice, we recommend to use batch sizes of 5, 21, and 42. These sizes are found via experimentation, and pack the 36kbit BRAM available in Xilinx FPGAs as densely as possible. Table 10 lists the additional BRAM cost for the different batch sizes, as well as the associated cycles.

3.5 Reduction without DSPs

In this section, we extend the technique of fast modulo reduction in [27] (called *Shifting Reduction* in this paper) without using additional DSP slices which are often necessary in a Barrett reduction unit or a Montgomery reduction unit. We apply this technique in the cases $q \in \{7681, 12{,}289, 15{,}361\}$. Moreover, in the case $q = 4591$, another reduction technique (called *Linear Reduction* in this paper) will be introduced. All four modular reductions are fully pipelined, and can process one new operand per clock cycle.

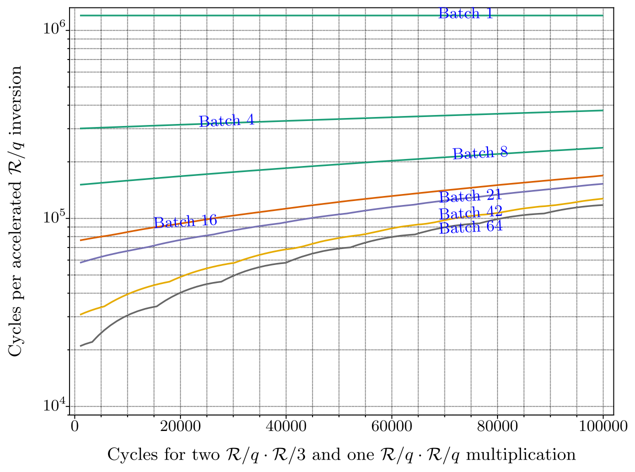


Fig. 4 Minimum batch size when comparing the cycle count for the three multiplications incurred per polynomial inversion when using Montgomery's trick, to simply accelerating the inversion itself. This assumes a base $\mathcal{R}/q$ inversion speed of 1,200,000 cycles, which is the rough number of cycles an $\mathcal{R}/q$ inversion takes with a single divstep per clock cycle. An example: Assume the three multiplications take 40,000 cycles in total, which is roughly how long two $\mathcal{R}/q \cdot \mathcal{R}/3$ and one $\mathcal{R}/q \cdot \mathcal{R}/q$ multiplication take in our design. At the same, assume that with the extra hardware resources, we could alternatively accelerate the inversion by a factor of 2, so that it takes only 600,000 cycles. According to the plot, a batch size of 4 would be sufficient for Montgomery's trick to be worthwhile

3.5.1 Fast signed modular multiplication on $q = 12{,}289$

We start with the modification of the unsigned reduction with $q = 12{,}289$ as introduced in [27]. In the signed case, the reduction is slightly different.

Suppose $-6144 \leq a \leq 6144$ and $-6144 \leq b \leq 6144$. We know that $z = ab$ is a 27-bit signed number (not 28-bit, which is in the unsigned case) and

$$\begin{aligned}(5C00000)_{16} = -37748736 &\leq z \\ = ab &\leq 37748736 = (2400000)_{16}\end{aligned}$$

We have $q = 2^{14} - 2^{12} + 1$, so $2^{14} \equiv 2^{12} - 1 \pmod q$. The sign bit $z[26]$ contributes $-2^{26} \equiv 1365 = 2^{11} - 683 \pmod q$. With a similar derivation in [27], z can be re-expressed as

$$\begin{aligned}z = &-2^{26}z[26] + 2^{14}z[25:14] + z[13:0] \\ \equiv\ &1365z[26] + z[13:0] \\ &+ 2^{12}(z[25:24] + z[23:22] + z[21:20] + z[19:18] \\ &+ z[17:16] + z[15:14]) \\ &- (z[25:14] + z[25:16] + z[25:18] + z[25:20] \\ &+ z[25:22] + z[25:24]) \\ \equiv\ &2^{11}z[26] + z[11:0] \\ &+ 2^{12}(z[25:24] + z[23:22] + z[21:20] \\ &+ z[19:18] + z[17:16] + z[15:14] + z[13:12]) \\ &- (683z[26] + z[25:14] + z[25:16] \\ &+ z[25:18] + z[25:20] + z[25:22] + z[25:24])\end{aligned}$$

We let

$$\begin{aligned}z_{pu} &\triangleq z[25:24] + z[23:22] + z[21:20] + z[19:18] \\ &\quad + z[17:16] + z[15:14] + z[13:12] \\ z_{p^2u} &\triangleq z_{pu}[4] + z_{pu}[3:2] + z_{pu}[1:0] \\ z_{p^3u} &\triangleq z_{p^2u}[2] + z_{p^2u}[1:0] \\ z_p &\triangleq 2^{12}z_{pu} + 2^{11}z[26] + z[11:0] \\ z_n &\triangleq 683z[26] + z[25:14] + z[25:16] + z[25:18] \\ &\quad + z[25:20] + z[25:22] + z[25:24]\end{aligned}$$

Clearly, $z_p - z_n \equiv z \pmod q$. z_{pu} is not greater than 21, z_{p^2u} is not greater than 6, and z_{p^3u} not greater than 3. z_p can be represented as

$$\begin{aligned}z_p \equiv\ &2^{12}z_{p^3u} + 2^{11}z[26] + Z[11:0] - (z_{p^2u}[2] + z_{pu}[4] \\ &+ z_{pu}[4:2]) \triangleq z_p^* \pmod q\end{aligned}$$

which is not greater than $12{,}288 + 2048 + 4095 = 18{,}431 < q + \frac{q-1}{2}$. We also have $z_n \leq 683 + (3 + 15 + 63 + 255 + 1023 + 4095) = 6074 < \frac{q-1}{2}$. Now, $z_0 \triangleq z_p^* - z_n \equiv z \pmod q$ and is an integer in $[-6074, 18{,}431]$. We need only to check if z_0 is greater than $\frac{q-1}{2} = 6144$, and perform a subtraction of q if this is the case.

The equivalent logic circuit is given in Fig. 5. The thicker blocks and dataflows differ from that in [27] for signed reduction.

3.5.2 Fast Signed Modular Multiplication on $q = 7681$

Shifting Reduction can be easily applied in the case $q = 7681$ since q is of the form $q = 2^h - 2^l + 1$.

Suppose $-3840 \leq a \leq 3840$ and $-3840 \leq b \leq 3840$. Now, $z = ab$ is a 25-bit signed number and

$$\begin{aligned}(11F0000)_{16} = -14745600 \leq z = ab &\leq 14745600 \\ &= (0E10000)_{16}\end{aligned}$$

Since $q = 2^{13} - 2^9 + 1$, we have $2^{13} \equiv 2^9 - 1 \pmod q$. The sign bit $z[24]$ contributes $-2^{24} \equiv -1912 \pmod q$. Now, z can be re-expressed as

$$\begin{aligned}z = &-2^{24}z[24] + 2^{13}z[23:13] + z[12:0] \\ \equiv\ &z[12:0] + 2^9(z[23:21] + z[20:17] + z[16:13]) \\ &- (1912z[24] + z[23:21] + z[23:17] + z[23:13])\end{aligned}$$

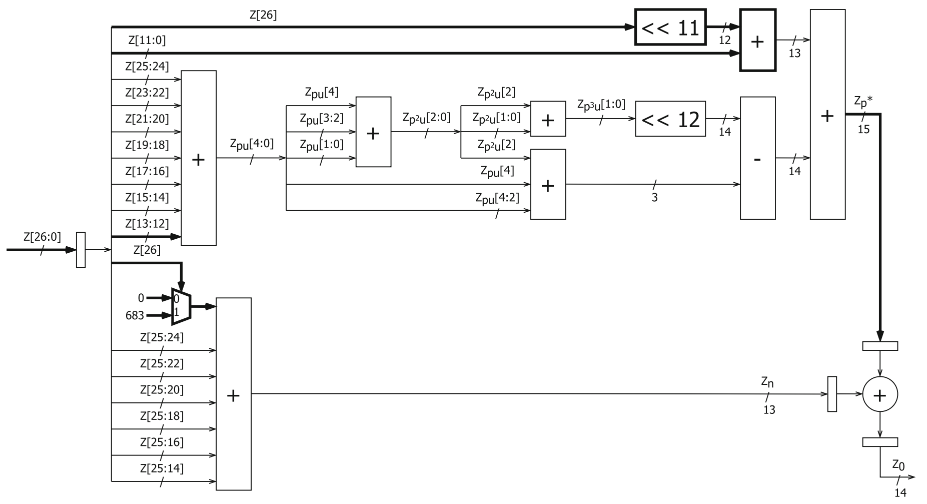


Fig. 5 Modified circuit for signed reduction modulo 12,289

$$
\begin{aligned}
&\equiv z[8:0] + 2^9(z[23:21] + z[20:17] + z[16:13] \\
&\quad + z[12:9]) - (1912z[24] + z[23:21] \\
&\quad + z[23:17] + z[23:13]) \pmod q
\end{aligned}
$$

We let

$$
\begin{aligned}
z_{pu} &\triangleq z[23:21] + z[20:17] + z[16:13] + z[12:9] \\
z_{p^2u} &\triangleq z_{pu}[5:4] + z_{pu}[3:0] \\
z_{p^3u} &\triangleq z_{p^2u}[4] + z_{p^2u}[3:0] \\
z_p &\triangleq z[8:0] + 2^9 z_{pu} \\
z_n &\triangleq 1912z[24] + z[23:21] + z[23:17] + z[23:13]
\end{aligned}
$$

z_{pu} is a 6-bit unsigned integer, and if $z_{pu}[5:4] = 3$, then $z_{pu}[3:0] \leq 4$ and $z_{p^2u} \leq 7$. So $z_{p^2u} \leq 17$ and is a 5-bit integer and then $z_{p^3u} \leq 15$, and $z_{p^2u}[4] + z_{pu}[5:4] \leq 3$. Now,

$$
\begin{aligned}
z_p &= z[8:0] + 2^9 z_{pu} \\
&\equiv z[8:0] + 2^9 z_{p^2u} - z_{pu}[5:4] \\
&\equiv z[8:0] + 2^9 z_{p^3u} - (z_{p^2u}[4] + z_{pu}[5:4]) \triangleq z_p^* \pmod q
\end{aligned}
$$

and is bounded by 8191. On the other hand, z_n is bounded by $1912 + 7 + 127 + 2047 = 4093$. Therefore, $z_0 \triangleq z_p^* - z_n \equiv z_p - z_n = z \pmod q$ and is an integer in $[-4093, 8191]$. Actually, we can tighten the possible values to $[-3581, 8191]$ because of this lemma:

Lemma 1 $z_0 \geq -3581$, *which is larger than* $-(q-1)/2 = -3840$.

Proof We need to consider the case $3582 \leq z_n \leq 4093$ only, since for the case $z_n < 3582$ the inequality always holds. Then,

$$
\begin{aligned}
3582 &\leq 1912z[24] + z[23:21] + z[23:17] + z[23:13] \\
&\leq 4093
\end{aligned}
$$

We know that the bound of each term is 1912, 7, 127, 2047, respectively. To make z_n not less than 3582, we need $z[24] = 1$. Therefore,

$$
\begin{aligned}
3582 - 1912 - 7 - 127 = 1539 = (603)_{16} &\leq z[23:13] \\
\leq 2047 = (7FF)_{16}
\end{aligned}
$$

Then, $z[23:21] \geq 6$, implying $z_{pu} \geq 6$, and further $z_{p^2u} \geq 1$ and $z_{p^3u} \geq 1$. Therefore, $z_p^* \geq 512$, and then

$$
z_p^* - z_n \geq 512 - 4093 = -3581.
$$

□

We now only need to determine if $z_0 > 3840$ where another signed subtraction by q with reduction is necessary to bound the eventual value in $[-3840, 3840]$. The equivalent circuit for signed reduction modulo 7681 is shown in Fig. 6. We can see that the architecture is very similar to that for modulo 12,289. The main difference is the dataflow of the sign bit.

3.5.3 Fast signed modular multiplication on $q = 15{,}361$

We still use Shifting Reduction in the case $q = 15{,}361$. Suppose $-7680 \leq a \leq 7680$ and $-7680 \leq b \leq 7680$. Now, $z = ab$ is a 27-bit signed number and

$$
\begin{aligned}
(47C0000)_{16} = -58{,}982{,}400 \leq z = ab \leq 58{,}982{,}400 \\
\leq (3840000)_{16}
\end{aligned}
$$

We have $q = 2^{14} - 2^{10} + 1$, and the sign bit $z[26]$ contributes $-2^{26} \equiv 3345 = 2^{12} - 751 \pmod q$. z can be re-expressed as

$$
\begin{aligned}
z &= -2^{26}z[26] + 2^{14}z[25:14] + z[13:0] \\
&\equiv 3345z[26] + z[13:0] + 2^{10}(z[25:22] + z[21:18]
\end{aligned}
$$

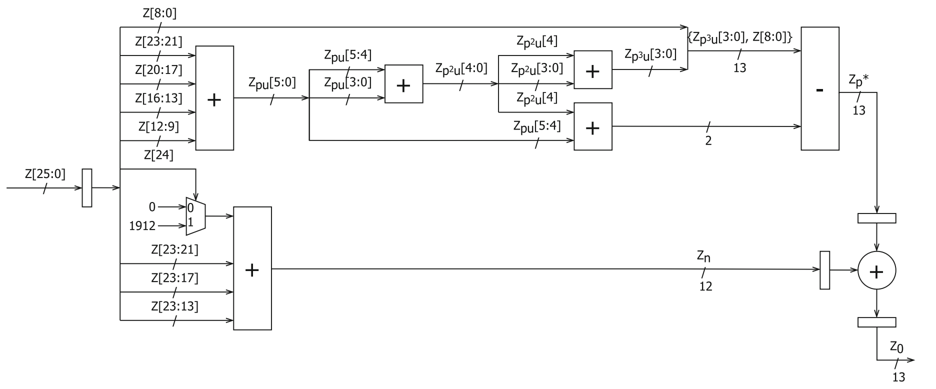


Fig. 6 Equivalent circuit for signed reduction modulo 7681

$$
\begin{aligned}
&\quad + z[17:14]) \\
&\quad - (z[25:14] + z[25:18] + z[25:22]) \\
&= 2^{12}z[26] + z[9:0] \\
&\quad + 2^{10}(z[25:22] + z[21:18] + z[17:14] \\
&\quad + z[13:10]) \\
&\quad - (751z[26] + z[25:14] + z[25:18] + z[25:22])
\end{aligned}
$$

We let

$$
\begin{aligned}
z_{pu} &\triangleq z[13:10] + z[25:22] + z[21:18] + z[17:14] \\
z_{p^2u} &\triangleq z_{pu}[5:4] + z_{pu}[3:0] + 4z[26] \\
z_{p^3u} &\triangleq z_{p^2u}[4] + z_{p^2u}[3:0] \\
z_p &\triangleq 2^{10}z_{pu} + 2^{12}z[26] + z[9:0] \\
z_n &\triangleq 751z[26] + z[25:14] + z[25:18] + z[25:22]
\end{aligned}
$$

Note that the definition of z_{p^2u} is slightly different from the other cases. We can see that z_{pu} is a 6-bit unsigned integer. If $z_{pu}[5:4] = 3$, then $z_{pu}[3:0] \le 12$ and $z_{p^2u} \le 19$. So $z_{p^2u} \le 21$ and is a 5-bit integer. Now,

$$
\begin{aligned}
z_p &= 2^{10}z_{pu} + 2^{12}z[26] + z[9:0] \\
&= 2^{10}z_{pu}[3:0] + 2^{14}z_{pu}[5:4] + 2^{10}\cdot 4z[26] + z[9:0] \\
&\equiv z[9:0] + 2^{10}z_{p^2u} - z_{pu}[5:4] \\
&\equiv z[9:0] + 2^{10}z_{p^3u} - (z_{p^2u}[4] + z_{pu}[5:4]) \\
&\triangleq z_p^* \pmod{q}
\end{aligned}
$$

and is bounded by 16383. z_n is bounded by $751 + 4095 + 255 + 15 = 5116$. Therefore, $z_0 = z_p^* - z_n \equiv z_p - z_n = z \pmod{q}$ and is an integer in $[-5116, 16{,}383]$. We need only to check if the value of z_0 is greater than 7680, and perform a subtraction of q if this is the case.

The circuit for signed reduction modulo 15,361 is omitted as it is similar to those for modulo 7681 and 12,289. The main difference is still the dataflow for the sign bit.

3.5.4 Fast signed modular reduction on $q = 4591$

The reduction in integers modulo $q = 4591$ (or other q's in the parameter set of NTRU Prime) using shifting reduction is not easily obtained since all of these primes are not of the form $q = 2^h - 2^l + 1$. Specifically, $q = 4591 = 2^{12} + 2^9 - 2^4 - 1$ is of effective Hamming weight 4. Shifting reduction will make the bits spread into the lower bits, making the positive and the negative parts of the partial results (as z_p and z_n defined in the case $q \in \{7681, 12{,}289, 15{,}361\}$) hard to be analyzed.

In the signed version modification doing modulo 12,289, we separate the sign bit from other bits and consider it independently. Actually, every bit can be considered independently, especially in the case $q = 4591$. We may transform the reduction problem into several signed additions. Here, we will call this technique *linear reduction*.

In the implementation we are considering, the integer z that will be reduced is a 33-bit signed integer and bounded by

$$
\begin{aligned}
(11117A137)_{16} \le z \le (0EEE85EC9)_{16} &= 4{,}008{,}206{,}025 \\
&= (2295)^2 \cdot 761
\end{aligned}
$$

And then, z can be represented as

$$
\begin{aligned}
z = &- 2^{32}z[32] + \sum_{i=0}^{31} 2^i z[i] \equiv 433z[32] \\
&+ 2079z[31] + 3335z[30] + 3963z[29] \\
&+ 4277z[28] + 4434z[27] \\
&+ 2217z[26] + 3404z[25] + 1702z[24] \\
&+ 851z[23] + 2721z[22] \\
&+ 3656z[21] + 1828z[20] + 914z[19] \\
&+ 457z[18] + 2524z[17] \\
&+ 1262z[16] + 631z[15] + 2611z[14] \\
&+ 3601z[13] + 4096z[12] \\
&+ z[11:0] \pmod{q}
\end{aligned}
$$

We can do 22 signed modular additions, but this approach will make the critical path be 5 signed modular additions. With an implementation in hardware, we can actually pre-combine some of the additions.

The basic idea is to utilize the power of look-up tables (LUTs). Xilinx FPGAs provide LUT units supporting the functions of both $\mathrm{LUT}_{5,2}$ and $\mathrm{LUT}_{6,1}$. We can divide the most significant 21 bits into five groups, each containing 3 to 5 specified bits, and collect $z[11:0]$ as one group. Specifically, we define

$$\begin{aligned}
p_0 &\triangleq z[11:0]\\
p_1 &\triangleq (3335z[30] + 2721z[22]\\
&\quad + 2524z[17] + 2611z[14]) \bmod q\\
p_2 &\triangleq (433z[32] + 851z[23] + 914z[19]\\
&\quad + 457z[18] + 631z[15]) \bmod q\\
n_0 &\triangleq -(3963z[29] + 4277z[28] + 3404z[25]\\
&\quad + 3656z[21] + 3601z[13]) \bmod q\\
n_1 &\triangleq -(4434z[27] + 1262z[16]\\
&\quad + 4096z[12]) \bmod q\\
n_2 &\triangleq -(2079z[31] + 2217z[26]\\
&\quad + 1702z[24] + 1828z[20]) \bmod q
\end{aligned}$$

We can apply any other partition. The partition we decide is good for the latter calculation because we can see the the bound of each group:

$$\begin{aligned}
&p_0 \le 4095\\
&p_1 \le 4076 < 4095\\
&p_2 \le 3286,\ p_2 + 4591 \le 7877 < 8191\\
&n_0 \le 4054 < 4095\\
&n_1 \le 3981 < 4095\\
&n_2 \le 3573, n_2 + 4591 \le 8164 < 8191
\end{aligned}$$

All possible values of p_1, p_2, n_0, n_1, and n_2 are pre-calculated and stored in the distributed memory constructed by $\mathrm{LUT}_{5,2}$ units. The values of p_1, p_2, n_0, n_1, and n_2 are then determined at the outputs of the LUTs, according to the inputs $z[33:12]$.

Now, we can easily implement the reduction with the modular additions:

$$\begin{aligned}
p_{01} &\triangleq p_0 + p_1\\
n_{01} &\triangleq n_0 + n_1\\
z_p &\triangleq (p_{01} \bmod q) + p_2\\
z_n &\triangleq (n_{01} \bmod q) + n_2\\
z \equiv z^* &\triangleq z_p - z_n
\end{aligned}$$

We can see p_{01}, n_{01}, z_p, z_n are all 13-bit unsigned integers. and z^* is bounded by $[-8191, 8191]$, which is a 14-bit signed integer. $z \bmod^{\pm} q$ can be found by

$$z \bmod^{\pm} q = z^* + kq, k \in \{-2, -1, 0, 1, 2\}$$

The modular reduction for $q = 4591$ uses just 117 LUT, 53 FF and no DSP. This is significantly better than the Barrett-based modular reduction from [15], which required 304 LUT, 107 FF and one DSP.

3.6 General purpose encode/decode

On the implementation of the general purpose encoder and decoder used in NTRU Prime (for the algorithms, see Appendix A), we inspect inductively the details of the process of the encoder, especially how R_2 and M_2 (denoted as R2 and M2 in the algorithm) change with respect to R and M, how many output bytes there are in each round, and what exactly M_2 is during each recursive call.

Case 1 When $\mathrm{len}(M) = 1$, that is, $R = \langle r_0 \rangle$ and $M = \langle m_0 \rangle$, there is no recursive call. We know that $r_0 < 16384$, so all the bytes of r_0 are dumped as output bytes to the encoded sequence. If $m_0 > 255$, the output is of 2 bytes and otherwise 1 byte.

Case 2 When $\mathrm{len}(M) = 2$, that is, $R = \langle r_0, r_1 \rangle$ and $M = \langle m_0, m_1 \rangle$, we compute $r'_0 = r_0 + m_0 r_1$. The upper bound for r'_0 and the new m'_0 can actually be pre-determined just from m_0 and m_1. Whether 0, 1, or 2 bytes are sent as output can also pre-determined from m_0 and m_1.

Case 3 When $R = \langle r_0, \ldots r_{2n-1}, r_{2n} \rangle, M = \langle m_0, \ldots, m_0, m_1 \rangle$ and $\mathrm{len}(M) = 2n + 1$ where n is a positive integer, we can compute that $r'_i = r_{2i} + m_0 r_{2i+1}$ for each $0 \le i \le n-1$. The upper bound of each r'_i and the new m'_0 can be pre-determined just from m_0. Whether 0, 1, or 2 bytes are sent as output can also pre-determined by m_0. We denote the "replaced" r'_i appended into R_2 as $r_i'^{(\mathrm{replaced})}$, which satisfies

$$r_i'^{(\mathrm{replaced})} \in \left\{ r'_i, \left\lfloor \frac{r'_i}{256} \right\rfloor, \left\lfloor \frac{r'_i}{65{,}536} \right\rfloor \right\}$$

and then we have $R_2 = \langle r_0'^{(\mathrm{replaced})}, \ldots, r_{n-1}'^{(\mathrm{replaced})}, r_{2n} \}$ and $M_2 = \langle m'_0, \ldots m'_0, m_1 \rangle$, with $\mathrm{len}(M_2) = n+1$. Note that the structure of M' and M are similar: a sequence of specified integers m_0s or m'_0s in [1, 16,383] followed by an integer m_1, which is either distinct from or the same as m_0 or m'_0.

Case 4 When $R = \langle r_0, \ldots, r_{2n}, r_{2n+1} \rangle, M = \langle m_0, \ldots, m_0, m_1 \rangle$ and $\mathrm{len}(M) = 2n + 2$ where n is a positive integer, we can compute that $r'_i = r_{2i} + m_0 r_{2i+1}$ for each $0 \le i \le n$. If $0 \le i \le n - 1$, the upper bound of r'_i and the new m'_0 can be pre-determined only by m_0. The upper bound of r'_n, which is the last element in R_2, and the new m'_1, which is

Table 1 Round infomation doing $\mathcal{R}/q$-encode

Round	len(M)	m_0	Regular output	Subtotal	m_1	Last output
1	761	4591	2	760	4591	N/A
2	381	322	1	190	4591	N/A
3	191	406	1	95	4591	N/A
4	96	644	1	47	4591	1
5	48	1621	1	23	11,550	2
6	24	10,265	2	22	286	1
7	12	1608	1	5	11,468	2
8	6	10101	2	4	282	1
9	3	1557	1	1	11,127	N/A
10	2	9740	N/A	N/A	11,127	2
11	1	N/A	N/A	N/A	1608	2

Table 2 Round infomation doing Round-encode

Round	len(M)	m_0	Regular output	Subtotal	m_1	Last output
1	761	1531	1	380	1531	N/A
2	381	9157	2	380	1531	N/A
3	191	1280	1	95	1531	N/A
4	96	6400	2	94	1531	2
5	48	625	1	23	150	1
6	24	1526	1	11	367	1
7	12	9097	2	10	2188	2
8	6	1263	1	2	304	1
9	3	6232	2	2	1500	N/A
10	2	593	N/A	N/A	1500	1
11	1	N/A	N/A	N/A	3475	2

the last element in M_2, is pre-determined by both m_0 and m_1. For $0 \leq i \leq n-1$, whether 0, 1, or 2 bytes are sent as output when computing r'_i is also pre-determined by m_0. Whether 0, 1, 2 bytes are sent as output when computing r'_n is pre-determined by m_0 and m_1. In this case, the resulting $R_2 = \langle r_0^{\prime(\text{replaced})}, \ldots, r_{n-1}^{\prime(\text{replaced})}, r_n^{\prime(\text{replaced})}\}$, $M_2 = \langle m'_0, \ldots, m'_0, m'_1\}$, and $\text{len}(M) = n+1$. The structure of M_2 and M are still similar: a sequence of specified integers m_0s or m'_0s followed by an integer m'_1, which is either distinct from or the same as m_0 or m'_0.

We know that when the encode starts, $M = \langle q, \ldots, q\rangle$ and $\text{len}(M)$ is odd. This implies that we need only to track m_0, m_1 and the output bytes for each regular pair of r's and for the last r. Table 1 show the values of m_0, m_1, and the output bytes. We can see the total encoded bytes are of length 1158.

With $q' = 1531 = q/3$, which is applied in *Round-encode*, a similar tracking info can also easily be pre-determined, shown in Table 2. The total encoded bytes are of length 1007.

All of the tracked info are provided outside the encoder and the decoder, making the circuit able to do the encode/decode for any case of Q. Both of the encoder and the decoder needs an internal memory buffer to save the intermediate R.

The block diagrams of the encoder and decoder are shown in Figs. 7 and 8 , where the dashed blocks are outside of the module. The parameter module is a look-up table of either Table 1 or Table 2, making the encoder/decoder flexible to do/recover either $\mathcal{R}/q$-encode or Round-encode. The encoder needs a DSP slice to evaluate $r'_0 = r_0 + m_0 r_1$. And the decoder needs 4 DSP slices to apply Barrett's reduction to evaluate $r_0 = r'_0 \mod m_0$.

The encoding of a public key and cipherext takes 2297 and 2296 cycles, respectively. The decoding of a public key and ciphertext takes 1550 and 1541 cycles, respectively.

3.7 SHA-512 hash function

Streamlined NTRU Prime uses SHA-512 internally as a hash function. It is used on the one hand to generate the

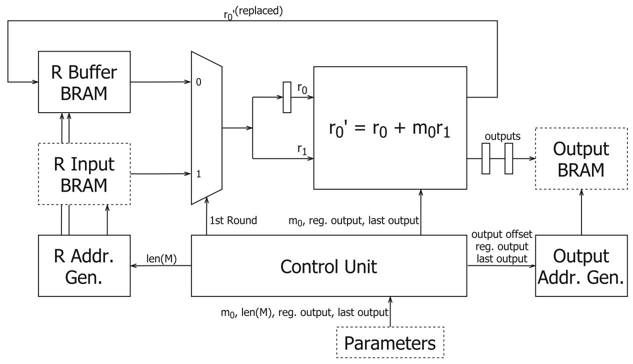


Fig. 7 Block diagram of the encoder

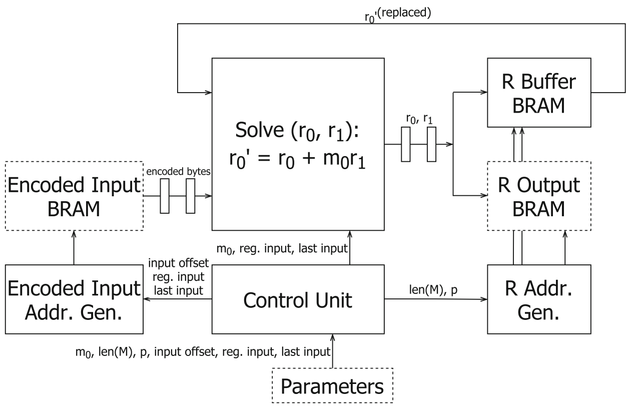


Fig. 8 Block diagram of the decoder

shared secret after encapsulation and decapsulation, but also to create the ciphertext confirmation hash. The ciphertext confirmation hash is a hash of the public key and the short polynomial r and is appended to the ciphertext. Our SHA-512 implementation is based on the implementation used in [15,19], but has been optimized to increase performance. The hashing of a 1024 bit block takes 117 cycles.

4 Evaluation and comparison with other implementations

In this section, we compare and evaluate individual sub-modules, as well as the full design, and provide area utilization and performance results.

4.1 Comparison of sub-modules

Table 3 shows a comparison of different sorting algorithm for arrays of size 761. The radix sort from this work is significantly faster than the sorting network from [15]. While not quite as fast as the FIFO merge sort from [10], our radix sort does use less LUT, FF and BRAM, and runs at a higher frequency. Due to the pregeneration of short polynomials, the cycle count of the sorting does not factor into the cycle count of the encapsulation, as long as the encapsulation operation takes longer than the sorting, which is the case. As such, any additional speed-up in sorting would not lead to a speed-up in encapsulation. This is not the case for batch key generation, as multiple short polynomials are needed. There, the key generation must wait until the sorting algorithm is executed a number of times equal to the batch size.

Table 4 shows a comparison of different multiplication algorithms for NTRU Prime. This includes the Karatsuba multiplier from [15], the high-speed and low-area schoolbook multiplier from this work, as well as our new NTT and CRT multiplier. Our new high-speed schoolbook multiplier is by far the fastest, by over an order of magnitude. At the same, it is also by far the most resource intensive. The Karatsuba-based multiplier from [15] is the most compact with regard to LUT, but it also is the slowest, and has a comparatively high BRAM usage. Our new low-area schoolbook multiplier uses no BRAM, and only slightly more LUT, but is more than three time faster with regard to cycle count than the Karatsuba-based multiplier. The NTT and CRT multiplier has the benefit of being extendable to perform $\mathcal{R}/q \cdot \mathcal{R}/q$ multiplication, with no increase in cycle count and only a moderate increase in resource consumption. Otherwise, the low-area schoolbook multiplier is better both in terms of resource consumed and cycle count, and requires no BRAM or DSPs. As a result, we only use the NTT multiplier for the $\mathcal{R}/q \cdot \mathcal{R}/q$ multiplication during batch inversion.

Table 5 show a comparison of our new encoder and decoder compared with the encoder and decoder from [15]. Our new encoder and decoder have either the same or lower resource consumption, while at the same time significantly reducing the cycle counts, as well as increasing the max clock frequency.

Table 6 shows a comparison of different inversion modules for NTRU Prime and NTRU-HPS. All implement the extended GCD algorithm. Our $\mathcal{R}/q$ inversion improves on the $\mathcal{R}/q$ inversion from [15], as due to the two divsteps per clock cycles, we gain a nearly two-times speed up. At the same, the amount of LUT and FF is reduced, and the DSP count remains the same. This is due to the improved modular reduction algorithm of our work. Increasing the number of divsteps to four gives another speedup of almost two, but also increases DSP, FF and LUT consumption. In addition, distributed RAM is used instead BRAM. For the inversion in $\mathcal{R}/3$, it is clearly visible that increasing the number of divsteps only leads to a comparative small increase in resource consumption, in exchange for a significant increase in performance. While the inversion from [10] is over an order of magnitude faster, it also uses significantly more LUT and FF than our design with 32 divsteps per cycle. However, since the $\mathcal{R}/3$ is not the bottleneck during key generation, further increasing the speed of $\mathcal{R}/3$ is unnecessary.

Table 3 A comparison of different sorting algorithms for generating fixed-weight polynomials of degree 761 for NTRU and NTRU Prime

Design	LUT	FF	BRAM	DSP	Freq (MHz)	Cycles	Time (µs)
Radix sort, this work	823	220	2	0	285	4837	16.97
Batcher sorting network [15]	231	87	1.0	0	279	49,400	177.1
FIFO merge sort [10]	1441	940	3.5	0	250	2762	11.05

The target FPGA is a Xilinx Zynq Ultrascale+

Table 4 A comparison of different multiplication algorithms for NTRU Prime

Design	LUT	FF	BRAM	DSP	Freq (MHz)	Cycles	Time (µs)
NTT and CRT, $\mathcal{R}/q \cdot \mathcal{R}/q$	2004	1395	7.5	3	285	35,463	124.4
NTT and CRT, $\mathcal{R}/q \cdot \mathcal{R}/3$	1888	1258	6.5	2	285	35,463	124.4
Schoolbook, High-Speed	27,594	20,078	0	0	289	1522	5.27
Schoolbook, Low-Area	1775	818	0	0	290	25,881	89.25
Karatsuba [15]	1463	817	4	0	279	78,132	280.0

The target FPGA is a Xilinx Zynq Ultrascale+. All multipliers except the NTT with the label $\mathcal{R}/q \cdot \mathcal{R}/q$ are for multiplying one polynomial in $\mathcal{R}/3$ with a second polynomial in $\mathcal{R}/q$. Note that for the high-speed schoolbook multiplier, we assume that the loading of the small polynomial is interleaved with the output of the result. Otherwise a multiplication takes 2283 cycles

Table 5 A comparison of different encoding and decoding algorithms for NTRU Prime

Design	LUT	FF	BRAM	DSP	Freq (MHz)	Cycles	Time (µs)
Encode, this work	201	154	0.5	1	290	2297	7.92
Encode [15]	215	131	0.5	1	279	5348	19.2
Decode, this work	334	273	1	4	290	1550	5.34
Decode [15]	676	571	2	5	279	7380	26.4

All cycle counts are for public keys. The target FPGA is a Xilinx Zynq Ultrascale+

4.2 Comparison of the full design

In this section, we will compare our implementation with existing Streamlined NTRU Prime implementations [9,15], as well as with NTRU-HPS821 [10]. NTRU-HPS821 is a round 3 finalist key-encapsulation algorithm. All Streamlined NTRU Prime implementations employ the parameter set *sntrup761*, and NTRU-HPS821 has comparable security strength. All benchmark numbers of individual operations of our implementation for the Zynq Ultrascale+ and the Artix-7 are listed in Tables 7 and 8, respectively. A comparison with existing Streamlined NTRU Prime implementations, as well as with NTRU-HPS821 is shown in Table 9. Benchmark numbers of our full implementation are listed in Table 11. Note that the cycles counts are not the simple addition of the cycles counts of sub-modules, as there is a certain level of overlapping of operations. Also note that, like the design in [15], both of our implementations do not contain a random number generator. This mirrors the reference design of NTRU Prime [7], and allows us to directly use the inputs of the known-answer-test to verify the correctness of our design. However, particularly for the high-speed encapsulation, this does somewhat skew the comparison with other KEMs. This does not apply to the decapsulation, as it does not require any randomness. In addition, for the low-area design, the SHA-512 hash function could be used to generate the randomness from a seed, as the hash function is both fast enough and has enough idle time between its normal usage, though we did not implement it for this work.

Our high-speed implementation has the fastest cycle count and execution times of all Streamlined NTRU Prime implementations for all 3 operations. At the same time, while our low-area implementation does require slightly more LUTs (at most 31% more) then the lightweight implementation from [15], our implementation is significantly faster, with 2.05, 4.08 and 3.04 speedup respectively for key generation, encapsulation and decapsulation.

When comparing our high-speed implementation with that of NTRU-HPS821 [10], one can see that our encapsulation uses fewer LUT, flip-flops, BRAMs, but more DSP. While our cycle count is slightly higher, this is compensated by the higher frequency, leading to a slightly faster execu-

Table 6 A comparison of different inversion modules for NTRU Prime and NTRU-HPS

Design	LUT	FF	BRAM	DSP	Freq (MHz)	Cycles	Time (μs)
$\mathcal{R}/q$, $p = 761$, 4 divsteps/cycle	3076	1099	0	19	285	311,918	1094
$\mathcal{R}/q$, $p = 761$, 2 divsteps/cycle	1122	593	2	11	285	600,906	2108
$\mathcal{R}/q$, $p = 761$ [15]	1642	726	2	11	279	1,168,960	4190
$\mathcal{R}/3$, $p = 761$, 32 divsteps/cycle	1040	517	0	0	285	47,166	165.5
$\mathcal{R}/3$, $p = 761$, 2 divsteps/cycle	607	211	0	0	285	590,158	2071
$\mathcal{R}/3$, $p = 761$ [15]	518	216	0	0	271	1,168,899	4101
$\mathcal{R}/3$, $p = 821$ [10]	8534	5479	0	0	250	1846	7.38

The target FPGA is a Xilinx Zynq Ultrascale+. The upper rows are for inversion of polynomials in $\mathcal{R}/q$, the lower in $\mathcal{R}/3$. All use the extended GCD algorithm

Table 7 Our work implemented on a Xilinx Zynq Ultrascale+ FPGA

Design	Module	Slices	LUT	FF	BRAM	DSP	Freq (MHz)	Cycles	Time (μs)
SNTRUP, TW, high speed	Key Gen	6038	37,813	25,368	33	23	285	64,026	224.7
	Encap	5381	31,996	22,425	4.5	6	289	5007	17.3
	Decap	5432	32,301	22,724	3.5	9	285	10,989	38.6
SNTRUP, TW, low area	Key Gen	1232	7216	3726	5.5	12	285	629,367	2208
	Encap	1074	6030	3211	4.5	7	290	29,245	100.8
	Decap	1051	6016	3194	3	7	283	85,628	302.6

Encapsulation and key generation assume a single short polynomial has been pregenerated. The key generation cycle counts for our high-speed implementation assume a batch size of 21, and list the amortized per-key cycles

Table 8 Our work implemented on a Xilinx Artix-7 FPGA

Design	Module	Slices	LUT	FF	BRAM	DSP	Freq (MHz)	Cycles	Time (μs)
SNTRUP, TW, high speed	Key Gen	10,827	39,200	25,536	33.5	23	143	64,026	447.7
	Encap	11,218	40,879	22,382	4.5	6	144	5007	34.8
	Decap	10,169	36,789	22,700	3.5	9	137	10,989	80.2
SNTRUP, TW, low area	Key Gen	2376	7579	3824	5.5	12	159	629,367	3958
	Encap	1945	6379	3069	4.5	6	147	29,245	198.9
	Decap	1842	6279	3086	3	7	131	85,628	653.6

As to be expected of the lower-end platform, the design uses more LUT and has a lower maximum clock frequency when compared to the Zynq Ultrascale+

tion time. For decapsulation, our design uses less of every resource except BRAM. In particular, our design uses 31% fewer flip-flops and 78% less DSP. Although our cycle count is higher and our frequency is lower, the total execution time is only 11% slower. For key generation, our design uses fewer LUT, flip-flops, and DSP, while also having a lower cycle count and faster clock speeds. However, our design does use significantly more BRAM due to the batch inversion. Batch inversion also has the downside of an initial large latency as the whole batch is calculated. Table 10 compares the cycle counts for different batch sizes. Larger batches increase the total number of cycles to complete the batch, but dramatically decrease the amortized cycles per key. However, the speedup from increasing the batch size from 21 to 42 is relatively low.

4.3 High-speed vs. low-area design

The difference between our high-speed and low-area implementation lie in a number of different sub modules. For one, the low-area version does not use batch inversion for key generation, and uses only 2 divsteps per clock cycles instead of 4 during $\mathcal{R}/q$ inversion, and 2 divsteps instead of 32 for the $\mathcal{R}/3$ inversion. The low-area implementation also uses the compact version of the parallel schoolbook multiplier. Finally, the high-speed implementation uses two separate decoders, one for public keys, and one for ciphertexts. This allows the secret key (which also contains the public key) and the ciphertext to be decoded in parallel during decapsulation. In the low-area implementation, only one decoder is present, and the decoding occurs sequentially.

Table 9 A comparison of different Streamlined NTRU Prime implementations for the parameter set sntrup761, as well as NTRU-HPS821

Module	Design	Slices	LUT	FF	BRAM	DSP	Freq (MHz)	Cycles	Time (μs)
Keygen	SNTRUP, TW, HS	6038	37,813	25,368	33	23	285	64,026	224.7
	SNTRUP, TW, LA	1232	7216	3726	5.5	12	285	629,367	2208
	SNTRUP [15]	1068	5935	4144	11.5	12	271	1,289,959	4748
	NTRU-HPS821, [10]	10,127	50,347	44,281	6.5	45	250	67,157	268.6
	Kyber L3, [10]	–	10,590	10,458	6.5	6	450	ca. 2600	5.9
	Saber L3, [10]	3634	20,496	13,939	1.5	0	370	2709	7.3
Encap	SNTRUP, TW, HS	5381	31,996	22,425	4.5	6	289	5007	17.3
	SNTRUP, TW, LA	1074	6030	3211	4.5	7	290	29,245	100.8
	SNTRUP [15]	844	4570	2843	7.5	8	271	119,250	439
	SNTRUP [9]	10,319	70,066	38,144	9	0	263	–	56.3
	NTRU-HPS821, [10]	7370	33,698	30,551	5.5	0	250	4576	18.3
	Kyber L3, [10]	–	10,590	10,458	6.5	6	450	ca. 3700	8.3
	Saber L3, [10]	3321	21,069	14,074	1.5	0	370	3735	10.1
Decap	SNTRUP, TW, HS	5432	32,301	22,724	3.5	9	285	10,989	38.6
	SNTRUP, TW, LA	1051	6016	3194	3	7	283	85,628	302.6
	SNTRUP [15]	902	5117	2958	7	8	271	260,307	958
	SNTRUP [9]	10,319	70,066	38,144	9	0	263	–	53.3
	NTRU-HPS821, [10]	7785	38,642	33,003	2.5	45	300	10,211	34.0
	Kyber L3, [10]	–	10,590	10,458	6.5	6	450	ca. 4900	10.9
	Saber L3, [10]	3816	21,342	14,233	1.5	0	370	4682	12.7

The entries for this work are marked with TW, and HS denotes the high-speed version, and LA the low-area version. The implementation from [9] does not implement decoding or key generation. Encapsulation and key generation for our work assume a single short polynomial has been pregenerated. The key generation cycle counts for our high-speed implementation assume a batch size of 21, and list the amortized per-key cycles. All entries are implemented on a Xilinx Zynq Ultrascale+ FPGA. Implementations for Kyber and Saber (level 3 parameter set) are also included for reference

Table 10 Effect of the different batch sizes on the speed of key generation, with 4 divsteps per clock cycle for the $\mathcal{R}/q$ inversion

Batch size	Total cycles	Amortized cycles	BRAM
1	316,785	316,785	3.5
5	524,174	104,835	16.5
21	1,344,558	64,026	33
42	2,447,759	58,280	55.5

The clock frequency and other FPGA resources are only minimally affected by increasing the batch size

Table 11 Full implementation of our work, with all operations merged

Design	Platform	Slices	LUT	FF	BRAM	DSP	Freq (MHz)
High speed	Zynq Ultrascale+	7051	40,060	26,384	36.5	31	285
	Artix-7	11,745	41,428	26,381	36.5	31	140
Low area	Zynq Ultrascale+	1539	9154	4423	8.5	18	285
	Artix-7	2968	9574	4399	8.5	18	128

4.4 Timing side channels

Both the high-speed and the low-area implementation are fully constant-time with regard to secret input. The radix sorting used in the generation of short polynomials does include secret-dependant memory indexing. However, as the BRAMs on modern Xilinx FPGA have no cache, this does not expose a side channel. At the same time, we did not implement any advanced protections against more advanced attacks such as DPA.

4.5 Applicability to NTRU LPRime

As mentioned earlier, the NIST submission of NTRU Prime describes the KEM Streamlined NTRU Prime and NTRU LPrime [7]. Both share many components, and many parts of our design can be reused to implement NTRU LPrime, namely the multipliers, the sorting module, the encoders and decoders, the hash module and the modular reduction modules. In addition, one would require the AES-based XOF used in NTRU LPrime, as well new state machines for the control flow and operation scheduling.

5 Conclusion

We present a novel and complete constant-time hardware implementation of Streamlined NTRU Prime, with two variants: A high-speed implementation and a low-area one. Both compare favorably to existing Streamlined NTRU Prime implementations, as well as to the round 3 finalist NTRU-HPS821. The full source code of our implementation in mixed Verilog and VHDL can be found on Github at https://github.com/AdrianMarotzke/SNTRUP_on_FPGA.

Acknowledgements We would like to thank Joppe Bos and Dieter Gollmann for their help. This work was labelled by the EUREKA cluster PENTA and funded by German authorities under grant agreement PENTA-2018e-17004-SunRISE. Taiwanese authors would like to thank Ministry of Science and Technology Grant 109-2221-E-001-009-MY3, Sinica Investigator Award (AS-IA-109-M01), Executive Yuan Data Safety and Talent Cultivation Project (AS-KPQ-109-DSTCP).

A Appendix: Encode and decode algorithm

```python
limit = 16384
def Encode(R,M):
   if len(M) == 0: return []
   S = []
   if len(M) == 1:
      r,m = R[0],M[0]
      while m > 1:
         S += [r%256]
         r,m = r//256,(m+255)//256
      return S
   R2,M2 = [],[]
   for i in range(0,len(M)-1,2):
      m,r = M[i]*M[i+1],R[i]+M[i]*R[i+1]
      while m >= limit:
         S += [r%256]
         r,m = r//256,(m+255)//256
      R2 += [r]
      M2 += [m]
   if len(M)&1:
      R2 += [R[-1]]; M2 += [M[-1]]
   return S+Encode(R2,M2)
```

Listing 1.1 The Python code of the encoder [7]. The lists R and M must have the same length, and $\forall i : 0 \leq R[i] \leq M[i] \leq 2^{14}$. Then, Decode(Encode($R$; M); M) $= R$.

```python
limit = 16384
def Decode(S,M):
   if len(M) == 0: return []
   if len(M) == 1: return [sum(S[i]*256**i for i in range(len(S)))%M[0]]
   k = 0; bottom,M2 = [],[]
   for i in range(0,len(M)-1,2):
      m,r,t = M[i]*M[i+1],0,1
      while m >= limit:
         r,t,k,m = r+S[k]*t,t*256,k+1,(m+255)//256
      bottom += [(r,t)]
      M2 += [m]
   if len(M)&1:
      M2 += [M[-1]]
   R2 = Decode(S[k:],M2)
   R = []
   for i in range(0,len(M)-1,2):
      r,t = bottom[i//2]; r += t*R2[i//2];
      R += [r%M[i]]; R += [(r//M[i])%M[i+1]]
   if len(M)&1:
      R += [R2[-1]]
   return R
```

Listing 1.2 The Python code of the decoder [7].

References

1. Alkim, E., Cheng, D.Y.L., Chung, C.M.M., Evkan, H., Huang, L.W.L., Hwang, V., Li, C.L.T., Niederhagen, R., Shih, C.J., Wälde, J., Yang, B.Y.: Polynomial multiplication in NTRU prime. IACR Trans. Cryptogr. Hardw. Embed. Syst. (2021). https://doi.org/10.46586/tches.v2021.i1.217-238
2. Apon, D.: NIST assignments of platforms on implementation efforts to PQC teams (**online**) https://groups.google.com/a/list.nist.gov/g/pqc-forum/c/cJxMq0_90gU/m/qbGEs3TXGwAJ. 7 Feb. 2019; Accessed 15 Oct. 2021
3. Bernstein, D.J., Brumley, B.B., Chen, M-S., Tuveri, N.: OpenSSLNTRU: faster post-quantum TLS key exchange. In: 31st USENIX Security Symposium (USENIX Security 22). pp. 845–862. Boston, MA (2022). https://www.usenix.org/conference/usenixsecurity22/presentation/bernstein
4. Bernstein, D.J., Chuengsatiansup, C., Lange, T., van Vredendaal, C.: NTRU prime: reducing attack surface at low cost. In: International Conference on Selected Areas in Cryptography. pp. 235–260. Springer, Berlin (2017)
5. Bernstein, D.J., Lange, T.: SUPERCOP, the system for unified performance evaluation related to cryptographic operations and

primitive. https://bench.cr.yp.to/supercop.html (2021). Accessed 07 Sept 2021

6. Bernstein, D.J., Yang, B.Y.: Fast constant-time GCD computation and modular inversion. IACR Trans. Cryptogr. Hardw. Embed. Syst. (2019). https://doi.org/10.13154/tches.v2019.i3.340-398
7. Brumley, B.B., Chen, M.S., Chuengsatiansup, C., Lange, T., Marotzke, A., Tuveri, N., van Vredendaal, C., Yang, B.Y.: NTRU prime: round 3. In: Post-Quantum Cryptography Standardization Project. NIST (2020)
8. Chung, C-M.M., Hwang, V., Kannwischer, M.J., Seiler, G., Shih, C-J., Yang, B-Y.: NTT Multiplication for NTT-unfriendly Rings: New Speed Records for Saber and NTRU on Cortex-M4 and AVX2. IACR Trans. Cryptogr. Hardw. Embed. Syst. 2021:159–188 (2021). https://tches.iacr.org/index.php/TCHES/article/view/879. https://doi.org/10.46586/tches.v2021.i2.159-188
9. Dang, V.B., Farahmand, F., Andrzejczak, M., Mohajerani, K., Nguyen, D.T., Gaj, K.: Implementation and Benchmarking of Round 2 Candidates in the NIST Post-Quantum Cryptography Standardization Process Using Hardware and Software/Hardware Co-design Approaches. Cryptology ePrint Archive, Report 2020/795 (2020), https://eprint.iacr.org/2020/795
10. Dang, V.B., Mohajerani, K., Gaj, K.: High-Speed Hardware Architectures and Fair FPGA Benchmarking of CRYSTALS-Kyber, NTRU, and Saber. Cryptology ePrint Archive, Report 2021/1508 (2021). https://eprint.iacr.org/2021/1508
11. Good, I.J.: Random motion on a finite abelian group. In: Mathematical Proceedings of the Cambridge Philosophical Society, vol. 47, pp. 756–762. Cambridge University Press, Cambridge (1951)
12. Knuth, D.E.: The Art of Computer Programming: Volume 3: Sorting and Searching. Addison-Wesley Professional, Boston (1998)
13. Land, G., Sasdrich, P., Güneysu, T.: A hard crystal—implementing dilithium on reconfigurable hardware. In: Grosso, V., Pöppelmann, T. (eds.) Smart Card Research and Advanced Applications, pp. 210–230. Springer, Cham (2022)
14. Lo, H.F., Shieh, M.D., Wu, C.M.: Design of an efficient FFT processor for DAB system. In: ISCAS 2001. The 2001 IEEE International Symposium on Circuits and Systems (Cat. No. 01CH37196). vol. 4, pp. 654–657 (2001). https://doi.org/10.1109/ISCAS.2001.922322
15. Marotzke, A.: A constant time full hardware implementation of streamlined NTRU prime. In: International Conference on Smart Card Research and Advanced Applications, pp. 3–17. Springer, Berlin (2020)
16. Mishra, P.K., Sarkar, P.: Application of Montgomery's trick to scalar multiplication for elliptic and hyperelliptic curves using a fixed base point. In: International Workshop on Public Key Cryptography. pp. 41–54. Springer, Heidelberg (2004)
17. NIST: Nist post-quantum cryptography standardization (**online**). https://csrc.nist.gov/Projects/Post-Quantum-Cryptography/Post-Quantum-Cryptography-Standardization. Accessed 11 June 2020
18. Roy, S.S., Basso, A.: High-speed instruction-set coprocessor for lattice-based key encapsulation mechanism: Saber in hardware. IACR Trans. Cryptogr. Hardw. Embed. Syst. (2020). https://doi.org/10.13154/tches.v2020.i4.443-466
19. Savory, D.: SHA-512 hardware implementation in VHDL. Based on NIST FIPS 180-4 (**online**). https://github.com/dsaves/SHA-512. Accessed 11 June 2020
20. Wang, W., Szefer, J., Niederhagen, R.: FPGA-based Niederreiter cryptosystem using binary Goppa codes. In: International Conference on Post-Quantum Cryptography, pp. 77–98. Springer, Cham (2018)
21. Xilinx, Inc.: UG474: 7 Series FPGAs Configurable Logic Block, 1.8 edn. Sept. 2016
22. Xilinx, Inc.: UG574: UltraScale Architecture Configurable Logic Block, 1.5 edn. Feb. 2017
23. Xilinx, Inc.: UG479: 7 Series DSP48E1 Slice, 1.10 edn. Mar. 2018
24. Xilinx, Inc.: UG473: 7 Series FPGAs Memory Resources, 1.14 edn. July 2019
25. Xilinx, Inc.: UG573: UltraScale Architecture Memory Resources, 1.13 edn. Sept. 2021
26. Xilinx, Inc.: UG579: UltraScale Architecture DSP Slice, 1.11 edn. Aug. 2021
27. Zhang, N., Yang, B., Chen, C., Yin, S., Wei, S., Liu, L.: Highly efficient architecture of NewHope-NIST on FPGA using low-complexity NTT/INTT. IACR Trans. Cryptogr. Hardw. Embed. Syst. (2020). https://doi.org/10.13154/tches.v2020.i2.49-72