Streamlined NTRU Prime on FPGA

Bo-Yuan Peng[1,2][0000−0002−8187−7554], Adrian Marotzke[3,4][0000−0002−5253−881X], Ming-Han Tsai[1], Bo-Yin Yang[2][0000−0002−9362−5282], and Ho-Lin Chen[1]

[1] National Taiwan University, Taipei, Taiwan
{d06921017,r08943151,holinchen}@ntu.edu.tw
[2] Institute of Information Science, Academia Sinica, Taipei, Taiwan
{bypeng,by}@crypto.tw
[3] NXP Semiconductors, Hamburg, Germany
adrian.marotzke@nxp.com
[4] Hamburg University of Technology, Hamburg, Germany
adrian.marotzke@tuhh.de

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 en- and decoders. With the high-speed design, we achieve the to-date fastest speeds for Streamlined NTRU Prime, with speeds of 5007, 10989 and 64026 cycles for encapsulation, decapsulation, and key generation respectively, while running at 285 MHz on a Xilinx Zynq Ultrascale+. The entire design uses 40060 LUT, 26384 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 [16]. 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 en- and decoding. We also present several novel designs to implement the subroutines required by NTRU Prime, such as sorting, modular reduction, polynomial multiplication and polynomial inversion.

2 Background

2.1 Definitions

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

$$\mathcal{R} = \mathbb{Z}[x]/(x^p - x - 1) \tag{1}$$

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

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

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 recomended parameter set for Streamlined NTRU Prime is *sntrup*761:

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

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 non-zero 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 & Decode: Streamlined NTRU Prime uses an en- and decoding algorithm to transform polynomials in $\mathcal{R}/3$ amd $\mathcal{R}/q$ 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 Algorithm 2 and 3 respectively.

Algorithm 1: Streamlined NTRU Prime Key Generation

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

Algorithm 2: Streamlined NTRU Prime Encapsulation

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

2.3 Design Consideration with FPGAs

Field Programmable Gate Arrays (FPGAs) are popular hardware implementation platforms so that we can easily construct and simulate customized digital logic circuits, and even make them as products. Most FPGAs provide several different general purposed resources which are either common in general logic circuits or able to simulate or to execute boolean functions, which are constructed by basic logic gates. On the hardware implementation with FPGAs, the utilization of these resources is the most important standard of comparison among similar implementation. To "make an apples-to-apples comparison," a specified FPGA platform is often assigned in a call-for-proposal project. For example, NIST recommands that "(PQC submission) teams generally focus their hardware implementation efforts on Artix-7" as 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

Algorithm 3: Streamlined NTRU Prime Decapsulation

Input: ciphertext C , secret key ($\bar{k}$, $\bar{K}$, ρ, $\text{hash}_4(\bar{K})$)

$c \Leftarrow$ Decode $\bar{c}$
$f, v \Leftarrow$ Decode $\bar{k}$
$e \Leftarrow (3fc$ in $\mathcal{R}/q$) modulo 3
$r' \Leftarrow ev$ in $\mathcal{R}/3$
if r' does NOT have weight w **then**
  $r' \Leftarrow$ (1, 1, ... , 1, 0, 0, ... , 0) // The first w elements are 1, the rest 0
Redo Encapsulation with $\bar{K}$ and r', compute new ciphertext C'
$\bar{r}' \Leftarrow$ Encode r'
if $C' = C$ **then**
  return $\text{hash}_1(\text{hash}_3(\bar{r}'), C)$
else
  return $\text{hash}_0(\text{hash}_3(\rho), C)$

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. The main resources provided in FPGAs are introduced as follows.

Look-Up Tables (LUTs): Look-up tables are very basic units in popular FPGAs. An LUT is a combinational logic unit with usually 4 to 6 input bits and 1 to 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 $\text{LUT}_{6,1}$ [20] [19].

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.

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 [24], and Xilinx Artix-7 provides DSP slices with 25×18-bit signed integer multipliers [21].

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.

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 [23] [22], each of which contains 36kbits and two channels. Every BRAM unit can be divided into two blocks with 18kbits, each of which in turn provides two channels, and the synthesis report records 0.5 BRAMs of utilization as long as a 18kbits block is utilized. In both FPGAs the data width of each 18kbits 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 need 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 $-\frac{q-1}{2} \leq f_i \leq \frac{q-1}{2}$ for every i satisfying $0 \leq i \leq p-1$. 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 $-\frac{q-1}{2} \leq r \leq \frac{q-1}{2}$ 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 $j \geq p$ 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 $O(p^2)$ to $O(p \lg p)$. Such an approach operating in a prime field $\mathbb{Z}/q$ but not complex nubmers is a Number

Theoretic Transform (NTT).

NTT is usually a 2^k-point transformation method with a pre-determined positive integer k (written as $\mathrm{NTT}_{2^k}(\cdot)$, and the inverse operation $\mathrm{iNTT}_{1^k}(\cdot)$). 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

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

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 $\mathrm{NTT}_{2^{11}}(\cdot)$ work.

Good [11] provides another approach, applying $\mathrm{NTT}_{2^9}(\cdot)$ 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] and [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. Since $f(x)g(x)$ is of degree at most 1535 (1534 actually), 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 $0 \leq j \leq 3$ and $0 \leq \ell \leq 511$ is one-to-one and onto. Then $f(x)$ (and $g(x)$, same as follows) can

be expressed as

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

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

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

We can regard the polynomial multiplication of $h(x) = f(x)g(x)$ as a schoolbook multiplication with respect to y, where the coefficients of the powers of y's are the sum of the z-polynomial products, 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 clear how 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 wan 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 $q_1 q_2 > p(q-1) + 1$. Then we apply Good's trick seperately.

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

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

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 $[-q_1 q_2 + \frac{q_1+q_2}{2}, q_1 q_2 - \frac{q_1+q_2}{2}]$, and we need only to check if it is in $[-\frac{q_1 q_2}{2}, \frac{q_1 q_2}{2}]$ and tune up or down by $q_1 q_2$. That is,

$$h_i = h_i^{(0)} + k q_1 q_2, k \in \{-1, 0, 1\}$$

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 are all signed 25×18-bit multipliers. This fact is important in the next case.

Case 2: In our implementation batch inversion is applied (see Section 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

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

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 $q_1 = 7681$, $q_2 = 12289$ and $q_3 = 15361$ 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

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

3 Hardware Implemenation

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 [14] and for other lattice KEM [17] [10] 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. The second is a much smaller, but also slower implementation. Both are similar with regards 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 Figure 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 Section 3.4). This idea was previously presented in [17] and [10], 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, and we do not perform any modular reduction

at this step. Its algorithmic description can be found in Algorithm 4.

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 to an accumulator register. 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.

Algorithm 4: 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** $a + r_c$

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.

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.

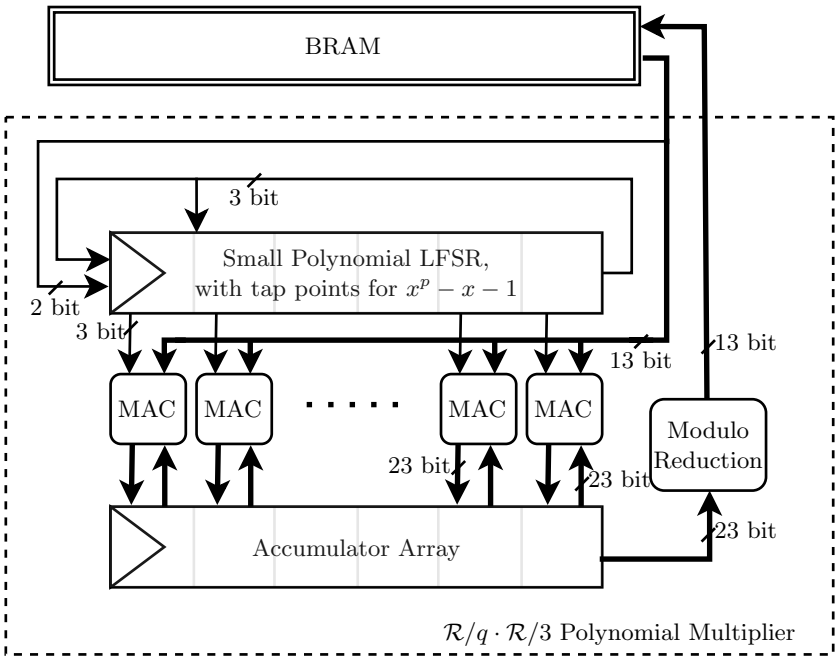

Fig. 1. Architecture of the parallel schoolbook polynomial multiplier for the parameter set sntrup761. The accumulator array has a size of $p \cdot 23$ bits. The blocks with the label MAC are described in Algorithm 4. The difference between the high-speed and the low-area multiplier are in the number of MAC units, and whether the accumulator array and small polynomial LFSR are implemented in flip-flops or in distributed RAM.

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

The architecture NTT multiplication employing Good's trick is shown in Figure 2, which is modified from the NTT/INTT architecture from [25]. This multiplier is used for the $\mathcal{R}/q \cdot \mathcal{R}/q$ multiplication during batch inversion (see Section 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.

– 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.

- 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.
- 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 $y^3 - 1$.
- In the *reload* stage, read data from bank 2 and write it back to bank 0 and bank 1 after nine point multiplications.
- In the $INTT$ stage, the process is the same as in the NTT stage.
- In the *crt* stage, the DSP slices in the 3 butterfly units are applied to calculte the partial result $h'_{i,a}q_bq_c$ in 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 $x^p - x - 1$ yet.
- In the *reduce* stage, $f(x)g(x) \bmod x^p - x - 1$ is evaluated.

We inspect in detail how the coeffcients 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 simutaneously into the butterfly units, and an efficient in-place memory addressing is introduced in [13], 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

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

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 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. A total of of p 32-bit random numbers are created. Of the first w, the lowest two bits are set so that the number is always even. For the others, the lowest two bits are set so that the number is always odd. This list of numbers is then sorted, and the upper 30 bits are discarded, and each number is subtracted by one. As a result, exactly w elements are either 1 or -1, and the

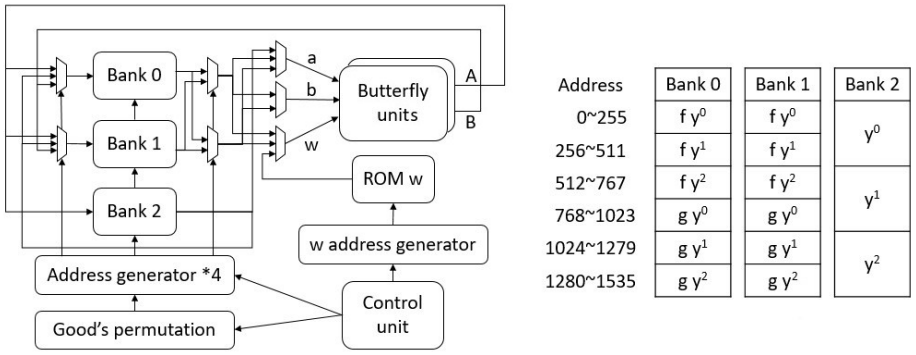

Fig. 2. Architecture of Good's trick NTT multiplication

rest are all zero.

The reference C implementation of NTRU Prime [7], as well as the hardware implementation in [14] 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. 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 implememation 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 4,837 cycles.

A further optimizations we have implemented is the pregeneration of short polynomials. As these can be generated independent of the operation (encapsulation or key generation) or any other input, we can pregenerate a short polynomial, instead of generating it on-demand. Once it has been output at the start of an, e.g. encapsulation, we can use the rest of the time spent on encapsulation 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.

3.4 Batch Inversion using Montgomery's Trick

Montgomery's trick is a method to speed up inversion by doing batch inversion [15]. This allows us to replace n inversions with a single inversion, together with $3n - 3$ multiplications. Montgomery's trick is described in Algorithm 5. The algorithm is faster assuming multiplication is faster than inversion, and that 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 generation speed of 156,317 cycles per key, compared to the non-batch 819,332 cycles.

For the polynomial inversion itself, we use the constant-time extended GCD algorithm from CHES 2019 [6]. This algorithm uses a constant number of "division steps" (or divsteps) to calculate the gcd of the input polynomials. It was already used in a previous implementation [14]. We extend this by allowing a configurable number of divsteps per clock cycle. Increasing the number of divsteps per clock cycle proportionally decreases the number of cycles.

Algorithm 5: 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 $\quad a_i \leftarrow a_{i-1} \cdot f_i$;
4 **end**
5 Compute inverse a_n^{-1} ;
6 **for** i from n to 2 **do**
7 $\quad f_i^{-1} \leftarrow a_i^{-1} \cdot a_{i-1}$;
8 $\quad a_{i-1}^{-1} \leftarrow a_i \cdot f_i$;
9 **end**
10 $f_1^{-1} \leftarrow a_1^{-1}$;
11 **return** $(f_1^{-1}, ..., f_n^{-1})$

In our implementation, we only implement batch inversion for the inversion in $\mathcal{R}/q$. For inversion in $\mathcal{R}/3$, it is very simple and efficient to simply increase the number of parallel divsteps, as the modular multiplication in $\mathcal{R}/3$ is trivial. 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. Currently, 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 5). This is an issue, as the polynomial multiplier for NTRU Prime normally always has one operand in $\mathcal{R}/3$. This mean we cannot use the same multiplier, as the normal multiplier has optimizations that rely on one operand being in $\mathcal{R}/3$. As a result, we include a second multiplier only 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 Figure 3. In practice, we recommend to use batch sizes of 5, 21 and 42. These sizes are found via experimentation, and pack the 36kbit BRAM as densely as possible. Table 4 lists the additional BRAM cost for the different batch sizes, as well as the associated cycles.

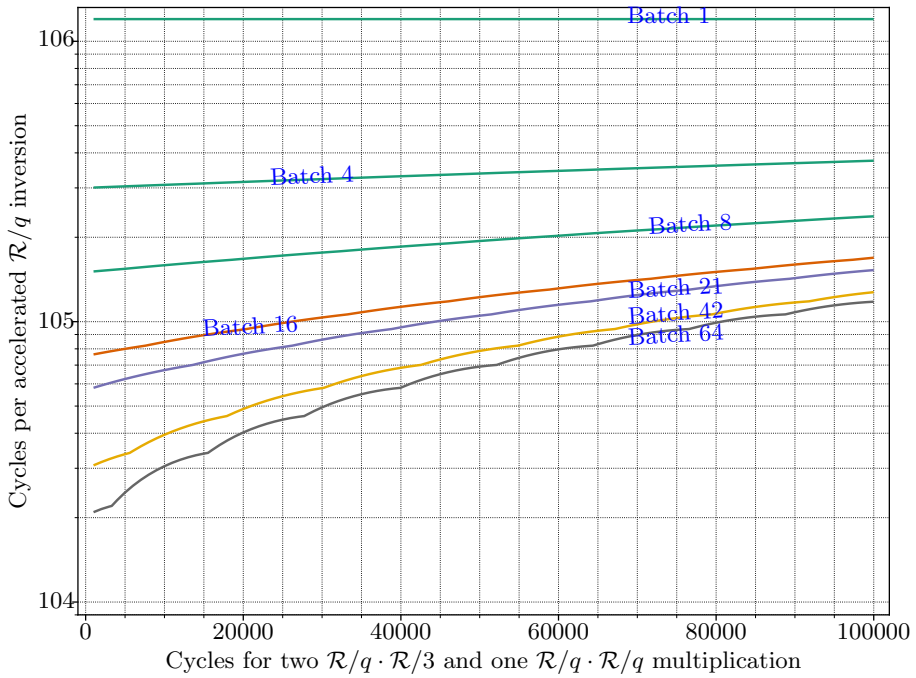

Fig. 3. Minimum batch size when comparing the cycle count for the three multiplications incurred per polynomial inversion when using Montgomery's Trick, to simply accelerating the inversion itself. This assumes a base $\mathcal{R}/q$ inversion speed of 1,200,000 cycles. An example: Assume the three multiplications take 40,000 cycles in total. At the same, assume that with the extra hardware resources, we could alternatively accelerate the inversion by a factor of 4, so that it takes only 300,000 cycles. According to the plot, we would need a batch size of at least just over 4 for Montgomery's trick to be worthwhile.

3.5 Reduction without DSPs

In this section we extend the technique of fast modulo reduction in [25] (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 $q \in \{7681, 12289, 15361\}$. 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.

Fast Signed Modular Multiplication on $q = 12289$ We start with the modification of the unsigned reduction with $q = 12289$ as introduced in [25]. 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

$$(5\text{C}00000)_{16} = -37748736 \leq z = ab \leq 37748736 = (2400000)_{16}$$

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 detivation in [25], z can be re-expressed as

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

We let

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

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

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

which is not greater than $12288 + 2048 + 4095 = 18431 < q + \frac{q-1}{2}$. 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 $[-6074, 18431]$. 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 equivlent logic circuit is given in Figure 4. The thicker blocks and dataflows differ from that in [25] for signed reduction.

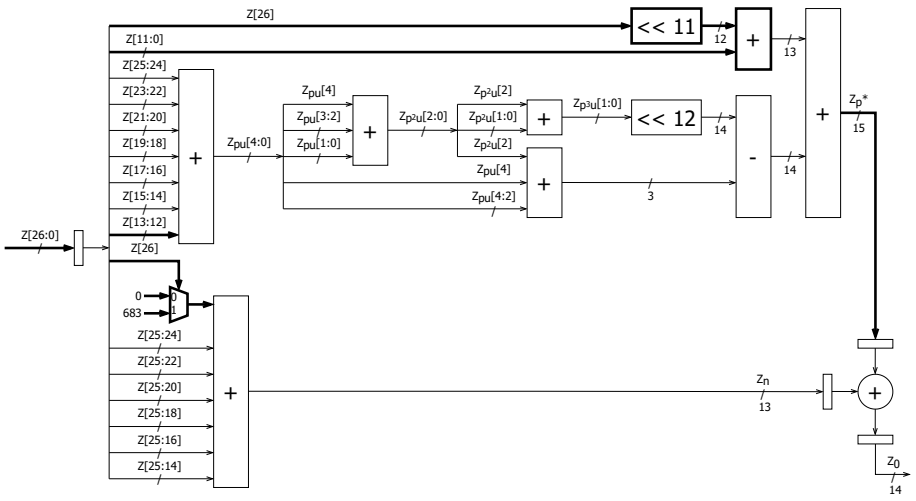

Fig. 4. Modified circuit for signed reduction modulo 12289.

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

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

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

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

We let

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

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

$$3582 \leq 1912 z[24] + z[23:21] + z[23:17] + z[23:13] \leq 4093$$

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,

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

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 equivlent circuit for signed reduction modulo 7681 is shown in Figure 5. We can see that the architecture is very similar to that for modulo 12289. The main difference is the dataflow of the sign bit.

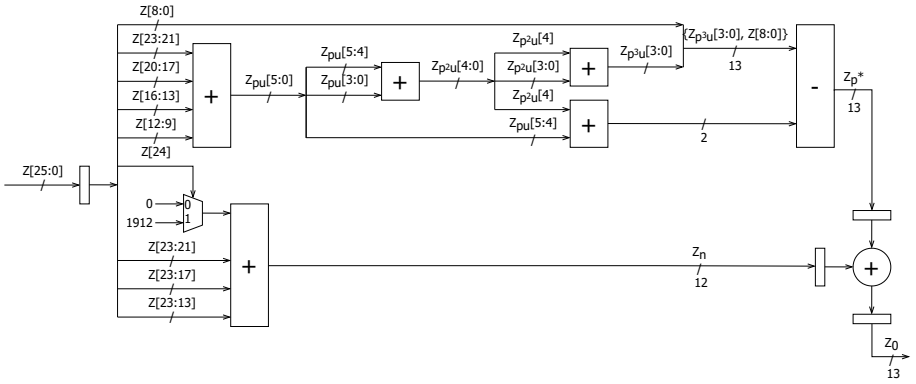

Fig. 5. Equivlent circuit for signed reduction modulo 7681.

Fast Signed Modular Multiplication on $q = 15361$ We still use Shifting Reduction in the case $q = 15361$. Suppose $-7680 \leq a \leq 7680$ and $-7680 \leq b \leq 7680$. Now $z = ab$ is a 27-bit signed number and

$$(47\text{C}0000)_{16} = -58982400 \leq z = ab \leq 58982400 \leq (3840000)_{16}$$

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

$$\begin{aligned} z &= -2^{26}z[26] + 2^{14}z[25:14] + z[13:0] \\ &\equiv 3345z[26] + z[13:0] + 2^{10}(z[25:22] + z[21:18] + z[17:14]) \\ &\quad - (z[25:14] + z[25:18] + z[25:22]) \\ &= 2^{12}z[26] + z[9:0] \\ &\quad + 2^{10}(z[25:22] + z[21:18] + z[17:14] + z[13:10]) \\ &\quad - (751z[26] + z[25:14] + z[25:18] + z[25:22]) \end{aligned}$$

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 $z_{pu}[3:0] \leq 12$ and $z_{p^2u} \leq 19$. So $z_{p^2u} \leq 21$ 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 $[-5116, 16383]$. 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 15361 is omitted as it is similar to those for modulo 7681 and 12289. The main difference is still the dataflow for the sign bit.

Fast Signed Modular Reduction on $q = 4591$ The reduction of 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 $q \in \{7681, 12289, 15361\}$) hard to be analyzed.

In the signed version modification doing modulo 12289, 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

$$(11117\text{A}137)_{16} \leq z \leq (0\text{EEE}85\text{EC}9)_{16} = 4008206025 = (2295)^2 \cdot 761$$

And then z can be represented as

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

Of course we can do 22 signed modular additions, but this approach will make the critical path be 5 signed modular additions. With 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 $\text{LUT}_{5,2}$ and $\text{LUT}_{6,1}$. We can divide the most significant 21 bits into 5 groups, each containing 3 to 5 specified bits, and collect $z[11:0]$ as one group. Specifically, we define

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

Of course 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:

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

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 $\text{LUT}_{5,2}$ 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 [14], which required 304 LUT, 107 FF and one DSP.

3.6 General Purpose Encode/Decode

On the implemenation 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 $\text{len}(M) = 1$, 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 $\text{len}(M) = 2$, 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 $R = \langle r_0, ...r_{2n-1}, r_{2n} \rangle$, $M = \langle m_0, ..., m_0, m_1 \rangle$ and $\text{len}(M) = 2n + 1$ where n is a positive integer, we can compute that $r'_i = r_{2i} + m_0 r_{2i+1}$ for each $0 \leq i \leq n - 1$. 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 $r_i'^{(\text{replaced})}$, which satisfies

$$r_i'^{(\text{replaced})} \in \{r'_i, \lfloor \frac{r'_i}{256} \rfloor, \lfloor \frac{r'_i}{65536} \rfloor\}$$

and then we have $R_2 = \langle r_0'^{(\text{replaced})}, ..., r_{n-1}'^{(\text{replaced})}, r_{2n} \}$ and $M_2 = \langle m'_0, ...m'_0, m_1 \rangle$, with $\text{len}(M_2) = n + 1$. Note that the structure of M' and M are similar: a sequence of specified integers m_0s or m'_0s in $[1, 16383]$ followed by an integer m_1, which is either distinct from or the same as m_0 or m'_0.

Case 4: When $R = \langle r_0, ...r_{2n}, r_{2n+1} \rangle$, $M = \langle m_0, ..., m_0, m_1 \rangle$ and $\text{len}(M) = 2n + 2$ where n is a positive integer, we can compute that $r'_i = r_{2i} + m_0 r_{2i+1}$ for each $0 \leq i \leq n$. If $0 \leq i \leq n - 1$, 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 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 $R_2 = \langle r_0'^{(\text{replaced})}, ..., r_{n-1}'^{(\text{replaced})}, r_n'^{(\text{replaced})} \}$, $M_2 = \langle m'_0, ..., m'_0, m'_1 \}$, 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, $M = \langle q, ..., q \rangle$ 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 interme-

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	11550	2
6	24	10265	2	22	286	1
7	12	1608	1	5	11468	2
8	6	10101	2	4	282	1
9	3	1557	1	1	11127	N/A
10	2	9740	N/A	N/A	11127	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

diate R.

The block diagrams of the encoder and decoder are shown in Figure 6 and 7, 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.

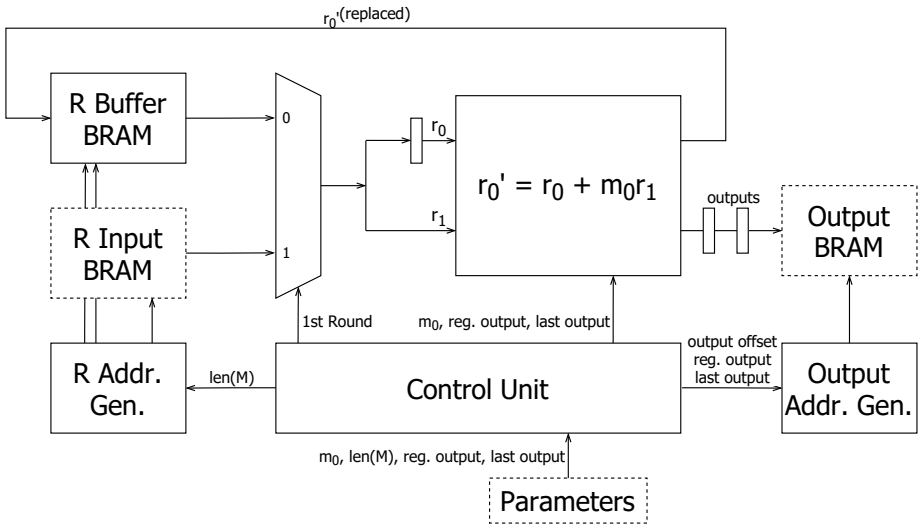

Fig. 6. The block diagram of the encoder.

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 share secret after encapsualtion 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 [14] and [18], but has been optimized to increase performance. The hashing of a 1024 bit block takes 117 cycles.

4 Evaluation & Comparison with other Implementations

In this section, we will compare our implementation with existing Streamlined NTRU Prime implementations [14] and [9], 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 for the Zynq Ultrascale+ and the Artix-7 are listed in Table 3 and 5 respectively. Benchmark numbers of the full implementation are listed in Table 6. 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 [14], our implementation is significantly faster, with 2.05, 4.08 and 3.04 speedup respec-

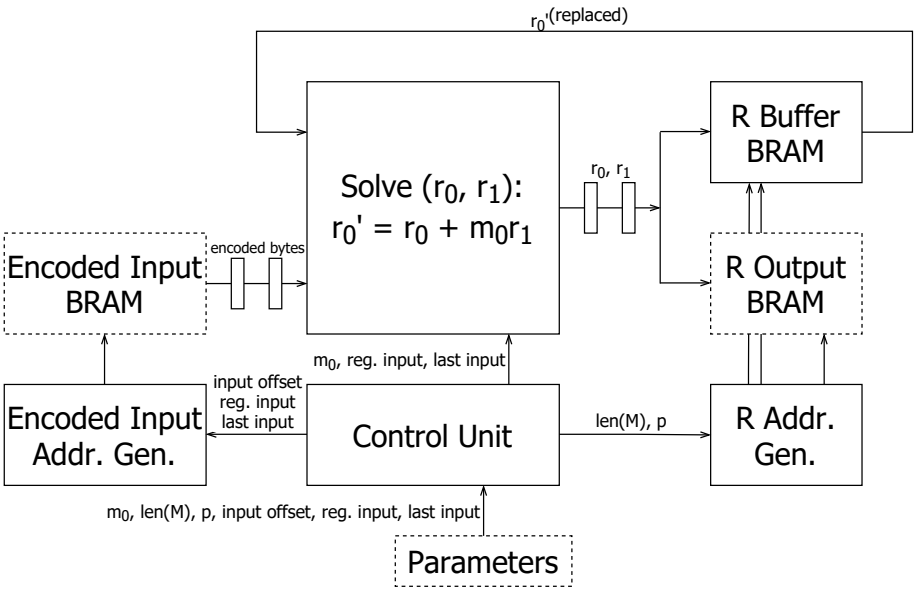

Fig. 7. The block diagram of the decoder.

tively 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 execution 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. While the cycle count is higher and the frequency lower for decapsulation, 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. Batch inversion also has the downside of an initial large latency as the whole batch is calculated. Table 4 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.

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

Table 3. A comparison of different Streamlined NTRU Prime implementations for the parameter set sntrup761 and NTRU-HPS821. The entries for this work and from [14] include all en- and decoding of ciphertext, public and secret keys. 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.

Design	Module	Slices	LUT	FF	BRAM	DSP	Freq (MHz)	Cycles	Time
Streamlined NTRU Prime, this work, high speed	Key Gen	6,038	37,813	25,368	33	23	285	64,026	224.7 μs
	Encap	5,381	31,996	22,425	4.5	6	289	5,007	17.3 μs
	Decap	5,432	32,301	22,724	3.5	9	285	10,989	38.6 μs
Streamlined NTRU Prime, this work, low area	Key Gen	1,232	7,216	3,726	5,5	12	285	629,367	2,208 μs
	Encap	1,074	6,030	3,211	4.5	7	290	29,245	100.8 μs
	Decap	1,051	6,016	3,194	3	7	283	85,628	302.6 μs
Streamlined NTRU Prime [14]	Key Gen	1,068	5,935	4,144	11.5	12	271	1,289,959	4,748 μs
	Encap	844	4,570	2,843	7.5	8	271	119,250	439 μs
	Decap	902	5,117	2,958	7	8	271	260,307	958 μs
SNTRUP [9], no key gen or decoding	Encap	10,319	70,066	38,144	9	0	263	-	56.3 μs
	Decap	10,319	70,066	38,144	9	0	263	-	53.3 μs
NTRU-HPS821, [10]	Key Gen	10,127	50,347	44,281	6.5	45	250	67,157	268.6 μs
	Encap	7,370	33,698	30,551	5.5	0	250	4,576	18.3 μs
	Decap	7,785	38,642	33,003	2.5	45	300	10,211	34.0 μs

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 4. The effect of the different batch sizes on the speed of key generation, with 4 divsteps per clock cycle for the $\mathcal{R}/q$ inversion. The clock frequency and other FPGA resources are only minimally affected by increasing the batch size.

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

4.1 Side Channels

Both the high-speed and the low-area implementation are fully constant-time with regards to secret input. The radix sorting used in the generation of short

Table 5. Our work implemented on a Xilinx Artix-7 FPGA. 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+.

Design	Module	Slices	LUT	FF	BRAM	DSP	Freq (MHz)	Cycles	Time
Streamlined NTRU Prime, this work, high speed	Key Gen	10,827	39,200	25,536	33.5	23	143	64,026	447.7 μs
	Encap	11,218	40,879	22,382	4.5	6	144	5,007	34.8 μs
	Decap	10,169	36,789	22,700	3.5	9	137	10,989	80.2 μs
Streamlined NTRU Prime, this work, low area	Key Gen	2,376	7,579	3,824	5.5	12	159	629,367	3,958 μs
	Encap	1,945	6,379	3,069	4.5	6	147	29,245	198.9 μs
	Decap	1,842	6,279	3,086	3	7	131	85,628	653.6 μs

Table 6. 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+	1,539	9,154	4,423	8.5	18	285
	Artix-7	2,968	9,574	4,399	8.5	18	128

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.

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. We plan on publishing the source code of our implementation after the review is complete.

Acknowledgments

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).

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 Transactions on Cryptographic Hardware and Embedded Systems pp. 217–238 (2021)
2. Apon, Daniel: NIST assignments of platforms on implementation efforts to PQC teams. https://groups.google.com/a/list.nist.gov/g/pqc-forum/c/cJxMq0_90gU/m/qbGEs3TXGwAJ, [Online 7-February-2019; accessed 15-October-2021]
3. Bernstein, D.J., Brumley, B.B., Chen, M.S., Tuveri, N.: OpenSSLNTRU: Faster post-quantum TLS key exchange (2021)
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 (2017)
5. Bernstein, D.J., Lange, T.: SUPERCOP, the System for Unified Performance Evaluation Related to Cryptographic Operations and Primitive (2021), https://bench.cr.yp.to/supercop.html [Accessed 07.10.2021]
6. Bernstein, D.J., Yang, B.Y.: Fast constant-time gcd computation and modular inversion. IACR Transactions on Cryptographic Hardware and Embedded Systems pp. 340–398 (2019)
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. IACR Transactions on Cryptographic Hardware and Embedded Systems pp. 159–188 (2021)
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
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 (1951)
12. Knuth, D.E.: The Art of Computer Programming: Volume 3: Sorting and Searching. Addison-Wesley Professional (1998)
13. 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 vol. 4 (2001). https://doi.org/10.1109/ISCAS.2001.922322
14. 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 (2020)
15. 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 (2004)
16. NIST: Nist post-quantum cryptography standardization. https://csrc.nist.gov/Projects/Post-Quantum-Cryptography/Post-Quantum-Cryptography-Standardization, [Online; accessed 11-June-2020]
17. Roy, S.S., Basso, A.: High-speed instruction-set coprocessor for lattice-based key encapsulation mechanism: Saber in hardware. IACR Transactions on Cryptographic Hardware and Embedded Systems pp. 443–466 (2020)

18. Savory, D.: SHA-512 hardware implementation in VHDL. Based on NIST FIPS 180-4. . https://github.com/dsaves/SHA-512, [Online; accessed 11-June-2020]
19. Xilinx, Inc.: UG474: 7 Series FPGAs Configurable Logic Block, 1.8 edn. (September 2016)
20. Xilinx, Inc.: UG574: UltraScale Architecture Configurable Logic Block, 1.5 edn. (February 2017)
21. Xilinx, Inc.: UG479: 7 Series DSP48E1 Slice, 1.10 edn. (March 2018)
22. Xilinx, Inc.: UG473: 7 Series FPGAs Memory Resources, 1.14 edn. (July 2019)
23. Xilinx, Inc.: UG573: UltraScale Architecture Memory Resources, 1.13 edn. (September 2021)
24. Xilinx, Inc.: UG579: UltraScale Architecture DSP Slice, 1.11 edn. (August 2021)
25. 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 Transactions on Cryptographic Hardware and Embedded Systems pp. 49–72 (2020)

A Encode and Decode Algorithm

```
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.

```
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].