A Constant Time Full Hardware Implementation of Streamlined NTRU Prime

Adrian Marotzke[1,2][0000−0002−5253−881X]

[1] TUHH adrian.marotzke@tuhh.de
[2] NXP adrian.marotzke@nxp.com

Keywords: NTRU Prime · Post-Quantum Cryptography · FPGA · Hardware · VHDL · Key Encapsulation Mechanism · Lattice-based cryptography

Abstract. This paper presents a constant time hardware implementation of the NIST round 2 post-quantum cryptographic algorithm Streamlined NTRU Prime. We implement the entire KEM algorithm, including all steps for key generation, encapsulation and decapsulation, and all en- and decoding. We focus on optimizing the resources used, as well as applying optimization and parallelism available due to the hardware design. We show the core en- and decapsulation requires only a fraction of the total FPGA fabric resource cost, which is dominated by that of the hash function, and the en- and decoding algorithm. For the NIST Security Level 3, our implementation uses a total of 1841 slices on a Xilinx Zynq Ultrascale+ FPGA, together with 14 BRAMs and 19 DSPs. The maximum achieved frequency is 271 MHz, at which the key generation, encapsulation and decapsulation take 4808 μs, 524 μs and 958 μs respectively. To our knowledge, this work is the first full hardware implementation where the entire algorithm is implemented.

1 Introduction

The NIST Post-Quantum-Cryptography standardization project has resulted in numerous new schemes designed to resist cryptanalysis by quantum computers [18]. Lattice based key encapsulation mechanism (KEM) schemes are the largest group, with nine submissions proceeding to round two. One common feature of all schemes is their complete dissimilarity to traditional public key cryptographic algorithms such as RSA and ECC. This makes existing hardware accelerators often sub optimal, though there has been some research on their use [1]. In addition, key sizes are significantly increased. This calls for fast and efficient hardware implementations, especially for embedded devices. NIST has addressed this by asking for performance benchmarks of the schemes on FPGA platforms. To our knowledge, this paper presents the first full constant time hardware implementation of round 2 Streamlined NTRU Prime. A software/hardware co-design, where parts of the en- and decapsulation were implemented in hardware, was published recently [9]. However, the co-design did not include the key generation. Our implementation, while slower during en- and decapsulation, uses significantly less FPGA fabric resources, even though we implement more functions in hardware.

2 Preliminary

NTRU Prime is a Ring Learning with Rounding based scheme [5, 6]. It has two KEM instantiations: Streamlined NTRU Prime (SNTRUP), and NTRU LPrime. The design goal behind NTRU Prime was to build a robust CCA secure public key scheme based on the original NTRU scheme [12], while minimizing the attack surface by removing the structures of cyclotomic rings. We focus on the Streamlined NTRU Prime variant. The Streamlined variant was the initial instantiation of NTRU Prime [8], has a smaller ciphertext, and faster en- and decapsulation. However, this comes at the cost of slower key generation and a larger public key [6]. However, both use the polynomial ring $\mathcal{R}/q = (\mathbb{Z}/q)[x]/(x^p - x - 1)$, with p and q being prime numbers, together with the ring $\mathcal{R}/3 = (\mathbb{Z}/3)[x]/(x^p - x - 1)$. A polynomial is *small* if all coefficients are in $-1, 0, 1$, e.g. elements of $\mathbb{Z}/3$. A polynomial has *weight* w if exactly w of its coefficients are non-zero. A polynomial is *short* if it is both *small* and of *weight* w. The values for p, q and w are part of the parameter set for SNTRUP, and are shown in Table 1. SHA-512 is used as a hash function internally, though only the first 256 bits of the output hash are used. Since all coefficients are either small or taken modulo a prime q, SNTRUP packs the coefficients together to reduce space. This is done with a decoding and encoding algorithm [5, 6].

Key generation, encapsulation and decapsulation are explained below. We use (a, b) to denote the concatenation of element a and b. Hash(a) denotes the hashing of a using SHA-512. Encode(a) and Decode(b) are the encoding and decoding functions respectively (see Section 4.5 for more details).

Key Generation: Generate a uniform random small element g which is invertible in $\mathcal{R}/3$. Calculate $1/g \in \mathcal{R}/3$. Generate a random short f. Calculate $h = g/(3f) \in \mathcal{R}/q$. Output Encode($h$) as the public key. Generate a random $\rho \in (0, \ldots, 255)^{(p+3)/4}$ and k = (Encode(f), Encode($1/g$)). Output (k, Encode(h), ρ, hash(4, Encode(h))) as the secret key.

Encapsulation: Input is the encoded public key, Encode(h). Generate a random short r. Decode the public key h. Compute $hr \in \mathcal{R}/q$. Compute c = Round(hr), where each element is rounded to the nearest multiple of 3. Compute $Confirm$ = hash(2, hash(3, r), hash(4, Encode(h))). Output ciphertext C = (Encode(c), $Confirm$), and hash(1, hash(3, r), C) as the shared secret.

Decapsulation: Input is the encoded secret key (k, Encode(h), ρ, hash(4, Encode(h))) from the key generation step, and the encoded ciphertext C = (Encode(c), $Confirm$) from the encapsulation. Decode polynomials f and v $\in \mathcal{R}/3$ from k (note that $v = 1/g$). Decode public key h. Decode ciphertext C and store as polynomial $c \in \mathcal{R}/q$. Compute $3fc \in \mathcal{R}/q$. While viewing each coefficient x of $3fc \in \mathcal{R}/q$ as an integer $-(q-1)/2 < x < (q-1)/2$, calculate each element modulo 3, and store as the new polynomial $e \in \mathcal{R}/3$. Calculate $r' = ev \in \mathcal{R}/3$.

If r' does not have weight w, then set r' to $(1, 1, ..., 1, 0, 0..., 0)$, with the first w elements being 1. Redo the encapsulation for the Fujisaki-Okamoto transformation [10], setting $r = r'$, obtaining the ciphertext C'. If $C = C'$ then output hash(1,hash(3, r), C) as the shared secret, otherwise output hash(0, hash(3, ρ), C).

Table 1: Streamlined NTRU Prime parameters for the NIST round 2 standardization process, as well as the corresponding public key, secret key and cipher text sizes [6]. Note that the public key is contained in the secret key and does not have to be stored separately. Additional, non-standardized parameter sets can be found in [5].

Security Level	p	q	w	Ciphertext Bytes	Public key Bytes	Secret key Bytes
NIST Level 1	653	4621	250	897	994	1518
NIST Level 3	761	4591	286	1039	1158	1763
NIST Level 5	857	5167	322	1184	1322	1999

3 Design Goals

Our primary goals was to develop a resource optimized Streamlined NTRU Prime implementation, while staying as *generic* as possible. Our implementation, except for the decoding module, is indeed fully generic, with parameters p, q, and w freely customizable, as long as they comply to the SNTRUP specifications. An expanded list of valid parameter sets can be found in [5]. This is useful, as any future parameter changes can be adopted with minimal changes. The encoding and decoding module do currently require some pre-calculated tables based on p and q. In addition, the design currently cannot process different values of p, q, and w at the same time, but needs to be resynthesized for any parameter changes. Though this also means that there is no performance and resource impact of being generic, as all modules and memories are synthesized to exactly match the requirements of the chosen parameter set. In addition, our design is written in platform agnostic VHDL.

4 Core Modules

4.1 $\mathcal{R}/q$ Multiplication

The multiplication in $\mathcal{R}/q$ is the core operation of both the en- and decapsulation. Due to the design of Streamlined NTRU Prime, the method of using the NTT for multiplication is not naturally possible [5]. Instead, conventional means of multiplying polynomials are used.

During a multiplication, a polynomial in $\mathcal{R}/q$ is multiplied with a $\mathcal{R}/3$ polynomial. This means that $\mathcal{R}/q$ polynomial has signed 13 bit coefficients, whereas

the $\mathcal{R}/3$ polynomial has signed two bit coefficients. However, these two bit coefficients can only have the values -1, 0 and 1. This results in the multiplier being almost trivial to implement. Using a pure product scanning schoolbook multiplication[14], our multiplier consumes only a small area (see Table 4). However, to improve performance, we use a combination of Karatsuba multiplication [15] and product scanning schoolbook multiplication. A single layer of Karatsuba multiplication is performed. This splits the multiplication of degree p into three multiplications of degree $\lceil p/2 \rceil$. The three partial multiplications are then done in parallel with schoolbook product scanning multiplication. This speeds up the multiplication by almost a factor of 4. During the schoolbook multiplication, we also make use of the dual port BRAM to read and write two words per clock cycle, doubling the speed of the multiplication. The use of Karatsuba does make the partial multiplication slightly more complicated, as one of the partial multiplications involves the addition of the lower and upper part of the $\mathcal{R}/3$ polynomial. This leads to a polynomial in $\mathcal{R}/q$ with 3 bit coefficients, with values between -2 and +2. The $\mathcal{R}/q$ multiplier also performs the multiplication of two polynomials in $\mathcal{R}/3$ during decapsulation. The single difference is that as a final step, all coefficients are reduced to $\mathbb{Z}/3$.

4.2 Reciprocal in $\mathcal{R}/q$ and $\mathcal{R}/3$

During key generation, two polynomial inversions must be performed, one in $\mathcal{R}/q$ and one in $\mathcal{R}/3$. We implement the constant time extended GCD algorithm from CHES 2019 [7]. For hardware accelerators for servers and other heavy-duty systems, batch inversion using Montgomery's trick [17] may be beneficial, but we did not consider this for our implementation. During key generation, the two inversions are done in parallel. Even so, the inversion is by far the single slowest operation (see Table 2).

In addition, we follow the reference implementation of SNTRUP by always calculating the inverse of g, rather than checking beforehand if g is invertible. Instead, we check the validity after the inversion has been completed. Should g not have been invertible, then we repeat the inversion with a new random small polynomial. In those cases, the key generation time is almost doubled.

4.3 Generating Short Polynomials and Constant Time Sorting

For the generation of the random short f during key generation and the random short r during encapsulation, a *constant time* sorting algorithm is used to sort a total of p 32 bit random values. In this context, *constant time* refers to being timing independent of the values sorted. Before sorting, the lowest 2 bits from first w 32 bit numbers are set so that they are always even. The lowest 2 bits from the rest are set so that they are odd. After sorting, the upper 30 bits are discarded, and the result is subtracted by 1. As a result, exactly w elements are either 1 or -1, and the rest are all zero, thus the polynomial is short. For the sorting itself, we implement a VHDL version of the algorithm suggested by [5]. The C-code of the sorting algorithm can be found in Listing 1.1.

```
void minmax(uint32 *x,uint32 *y) {
  uint32 xi = *x; uint32 yi = *y;
  uint32 xy = xi ^ yi;
  uint32 c = yi - xi;
  c ^= xy & (c ^ yi ^ 0x80000000);
  c >>= 31;
  c = -c;
  c &= xy;
  *x = xi ^ c; *y = yi ^ c;
}
void uint32_sort(uint32 *x,int n) {
  int top,p,q,i;
  top = 1;
  while (top < n - top) top += top;
  for (p = top;p > 0;p >>= 1) {
    for (i = 0;i < n - p;++i)
      minmax(x + i,x + i + p);
    for (q = top;q > p;q >>= 1)
      for (i = 0;i < n - q;++i)
        minmax(x + i + p,x + i + q);
  }
}
```

Listing 1.1: The C code of the sorting algorithm. The minmax function compares and swaps the two inputs [5].

4.4 Modulo Reduction in $\mathbb{Z}/q$ and $\mathbb{Z}/3$

In many parts during the KEM, we must reduce integers to $\mathbb{Z}/3$ and $\mathbb{Z}/q$, e.g. during the inversion, or after the $\mathcal{R}/3$ multiplication. To quickly reduce coefficients, we used pipelined Barrett reductions [3]. This is different from the reference implementation, which uses a constant time division algorithm [6]. Barrett reduction allows a modulo operation to be replaced with a multiplication and a bit shift. This can be implemented efficiently in the DSP units of the FGPA. For the reduction to $\mathbb{Z}/3$, we slightly modify the constants than those from Barrett's original paper. The modified constants are needed as we need to reduce values from a larger interval than normally allowed: Barrett reduction for a modulus n is only correct for the interval $[0, n^2]$. This obviously does not work if we wish to reduce values from $\mathbb{Z}/q$ to $\mathbb{Z}/3$ if $q > 9$. In addition, the modifications allows us to reuse the reduction during encapsulation to round coefficients to the nearest multiple of three. A Python version of our reduction algorithm with the constants is found in Listing 1.2. To verify the correctness of the modified constants, we mechanically verify in a simulator that all possible elements from $\mathbb{Z}/q$ are both rounded correctly and reduced to the correct values in $\mathbb{Z}/3$. This would mean that for elements outside of $\mathbb{Z}/q$, one would first have to reduce to $\mathbb{Z}/q$, and then again to $\mathbb{Z}/3$. However, this situation does not occur in our design.

```
import math
q=4591; q_half = math.floor(q/2); p=761
k=16; r=math.floor((2**k) / 3)
def reduce_and_round(input):
    rounded_output = 3*(((input+q_half) * r + 2**(k - 1)) >> k)-q_half
    mod_3_result = input - rounded_output
    return [rounded_output, mod_3_result]
```

Listing 1.2: A python version of our combined Barrett reduction to $\mathbb{Z}/3$ and rounding algorithm.

4.5 En- and Decoder for Polynomials $\mathcal{R}/q$ and $\mathcal{R}/3$

In order to save bytes during transmission, Streamlined NTRU Prime has specified an encoding for public and private keys and the ciphertext. The polynomials of the secret key f and $1/g$ are both in $\mathcal{R}/3$, and each polynomial coefficient is in $\mathbb{Z}/3$ and can be represented with 2 bits. Four of these coefficients are simply packed into a byte using a shift register. The encoder for the ciphertext and public key is more complicated, as both polynomials are in $\mathcal{R}/q$, with coefficients in $\mathbb{Z}/q$. Here, the coefficients are all have 13-bit size. However, since values in the interval between q and 2^{13} do not occur, packing can save space by "overlapping" the coefficients. There is a slightly different encoding for the ciphertext, as each coefficient of the ciphertext is rounded to the nearest multiple of 3. This allows us to save further space. See Appendix A for the Python code of the encoder.

Decoding elements in $\mathcal{R}/3$ is similar to the encoding, with a simple shift register. Decoding the elements from $\mathcal{R}/q$ however, is again more complex. In fact, the $\mathcal{R}/q$ decoder is one of the most expensive modules when it comes to resource consumption. One reason for this is that the decoder requires a 32 by 16 bit division. In order to avoid the need to implement a full division circuit, we precalculate all divisors, which are not dependent on any secrets, and store these in a table. For $p = 761$ and $q = 4591$, there are in total 42 different divisors, each fitting in 16 bits. Half of these are for the decoding the rounded coefficients of the ciphertext, the other half are for the public key. Due to the precalculation, we can then use integer division by a constant, allowing us to replace the division with a multiplication and a bit shift [16]. See Appendix A for the Python code of the decoder.

Before decapsulation and encapsulation can begin, the secret key and public key respectively must be decoded first. However, for subsequent en- and decapsulations, the decoding does not have to be repeated. Instead, the decoded public and secret keys are stored in internal memory. This can save time whenever a key is reused for multiple KEM runs.

4.6 SHA-512

Streamlined NTRU Prime uses SHA-512 as a hash function. This is used on the one hand to generate the shared secret after the en- and decapsulation, but also for the ciphertext confirmation hash. The confirmation hash is the hash of the short polynomial r and the public key, and is appended to the ciphertext during encapsulation. We did not write our own SHA-512 implementation, but instead used an open source one [20], with some slight modifications to improve performance and reduce resource consumption. The SHA-512 module consumes nearly half of all LUTs and over half of the flip-flops of the entire implementation (see Table 4), making it by the far the single most expensive component. A note to remember is that only the first 256 bits of the hash output are actually used. As such, usage of other hash functions such as SHA-256 or SHA3-256 could be worth a consideration in order to save resources. On the other hand, one could

argue that in many systems, SHA-512 accelerators are already available, so using SHA-512 does not add any resource cost. From a speed perspective, the time spent on hashing is negligible (see Table 2 and 3).

Table 2: A summary of the cycle count of the core modules of our implementation for the parameter set $p = 761$, $q = 4591$ and $w = 246$. Note that the times for sub-modules are included (sometime multiple times) in upper modules. The cycle counts for key generation, encapsulation and decapsulation are all using the Karatsuba $\mathcal{R}/q$ multiplication.

Module	Clock Cycles
Key generation	1 304 738
Key encapsulation	134 844
Key encapsulation inc. key load	142 229
Key decapsulation	251 425
Key decapsulation inc. key load	259 899
$\mathcal{R}/q$ Schoolbook multiplication	292 232
$\mathcal{R}/q$ Karatsuba multiplication	78 132
Reciprocal in $\mathcal{R}/q$	1 168 960
Reciprocal in $\mathcal{R}/3$	1 168 899
Generating short polynomials	50 927
Sorting algorithm	49 400
SHA-512 (per 1024 bit block)	325
Encode $\mathcal{R}/q$ (public key)	5 348
Encode $\mathcal{R}/q$ (ciphertext)	5 197
Decode $\mathcal{R}/q$ (public key)	7 380
Decode $\mathcal{R}/q$ (ciphertext)	6 721
Encode $\mathcal{R}/3$	761
Decode $\mathcal{R}/3$	761

5 Architecture

The architecture of our implementation is tailored to Streamlined NTRU Prime. After synthesis, the design is specific to a single parameter set. The design has the following inputs:

- **Start Encap**. Begin the encapsulation
- **Start Decap**. Begin the decapsulation
- **Start Key Gen**. Begin the key generation
- **Public Key**. Input a new encoded public key for encapsulation
- **Secret Key**. Input a new encoded secret key for decapsulation
- **Ciphertext**. Input a new encoded ciphertext for encapsulation
- **Random**. Source of randomness

The design has the following outputs:

- **Public Key**. Output encoded public key after key generation

Table 3: A summary of how the how the clock cycles of key generation, encapsulation and decapsulation are distributed among the sub modules for the parameter set $p = 761$, $q = 4591$ and $w = 246$. For en- and decapsulation, both times included the decoding of the public and/or secret key.

Operation	Function	Relative share of clock cycles (%)
Key Generation	$\mathcal{R}/q$ and $\mathcal{R}/3$ reciprocal	89.6%
	$\mathcal{R}/q$ multiplication	6%
	Generating short polynomials	3.9%
	Other	0.5%
Encapsulation	$\mathcal{R}/q$ multiplication	54.9%
	Generating short polynomials	35.8%
	Decoding public key	5.2%
	Encoding cipher text	3.6%
	Other	0.5%
Decapsulation	$\mathcal{R}/q$ and $\mathcal{R}/3$ multiplication	60.3%
	Re-encapsulation	32.4%
	Decoding cipher text	2.6%
	Decoding secret key	2.7%
	Other	2%

- **Secret Key**. Output encoded secret key after key generation
- **Ciphertext**. Output encoded ciphertext after encapsulation
- **Shared secret**. Output shared secret after en- and decapasulation

The design has a main finite state machine (FSM), which controls the access to the different shared modules, as well as starting the separate main operations such as key generation. The design can only process a single key generation, encapsulation or decapsulation at a time. The shared modules are operations that are needed across key generation, de- and encapsulation. This includes the $\mathcal{R}/q$ multiplication, the en- and decoding, hashing, the reduction to $\mathbb{Z}/3$, the generation of short polynomials, as well as the parts of the encapsulation that are needed for the Fujisaki-Okamoto transformation [10] during decapsulation.

6 Implementation Results on an FPGA

We implemented the design on a Xilinx Zynq Ultrascale+ ZCU102 FPGA, using the parameter set for level 3 as an example. This means $p = 761$, $q = 4591$ and $w = 246$. We achieve a maximum clock frequency of 271 MHz. In total, the design uses 9538 LUT, 7802 flip-flops, 14 BRAMs, and 19 DSP units. An encapsulation takes 142 229 clock cycles. A decapsulation takes 251 425 clock cycles, excluding the time it takes to load and decode the secret key, and 259 899 clock cycles if it is included. A key generation takes 1 304 738 clock cycles. We did not implement any kind of RNG into our design, instead it simply has an input where cryptographically secure random bits are expected. During both encapsulation and key generation, 24 352 random bits are needed for the generation of the short polynomials, for which 761 (i.e. p) random 32 bit values are needed. During key

Table 4: A summary of the resources used for the parameter set $p = 761$, $q = 4591$ and $w = 246$. We also included a subset of non-shared sub-modules that are of special interest. Note that the resources for non-shared sub-modules are included (sometime multiple times) in upper modules. The row "total" is using the Karatsuba $\mathcal{R}/q$ multiplication. The numbers for the schoolbook multiplication are for comparison and are not included in the total.

	Module	Logic Slices	LUT	Flip-flops	BRAM	DSP
Main operations	Key generation	458	2499	1082	3	11
	Key encapsulation	77	157	94	0.5	0
	Key decapsulation	204	739	263	1.5	0
Shared modules	Key encapsulation shared core	25	113	22	0.5	1
	$\mathcal{R}/q$ schoolbook multiplication	94	418	281	1	0
	$\mathcal{R}/q$ Karatsuba multiplication	298	1463	817	4	0
	Generating short polynomials	45	231	87	1	0
	SHA-512	716	3174	4710	1	0
	Encode $\mathcal{R}/q$	47	215	131	0.5	1
	Decode $\mathcal{R}/q$	128	676	571	2	5
	Reduce to $\mathbb{Z}/3$ & rounding	10	23	19	0	1
	Top level leaf cells	0	248	6	0	0
	Total	1841	9538	7802	14	19
Non-shared sub-modules	Sorting algorithm	33	159	56	0	0
	Reciprocal in $\mathcal{R}/q$	278	1642	726	2	11
	Reciprocal in $\mathcal{R}/3$	92	518	216	0	0
	Reduce to $\mathbb{Z}/q$	54	304	107	0	1
	Encode $\mathcal{R}/3$	8	18	20	0	0
	Decode $\mathcal{R}/3$	7	24	22	0	0
	Constant integer division	50	232	188	0	5

generation, a further 24 352 bits are needed for the random g, and 1528 bits for the random value ρ, for a total of 50 232 bits. No randomness is needed during decapsulation. A detailed list of the runtimes of all core modules can be found in Table 2. A summary of the resources used can be found in Table 4.

Table 3 details the cycles count for key generation, en - and decapsulation as percentages. For the key generation, the cycle count is dominated by the polynomial inversion, which take 89.6% of the total time. Both the $\mathcal{R}/q$ multiplication and the generation of short polynomials only take 6% and 3.9% respectively. Encoding and hashing are minuscule in comparison, and are grouped together under other. For encapsulation, the time for $\mathcal{R}/q$ multiplication and the generation of short polynomials is 54.9% and 35.8% respectively. Encoding and decoding only take a small single digit percentage of the cycles. The hashing is again negligible, and is grouped under other. For decapsulation, the time is dominated by the $\mathcal{R}/q$ and $\mathcal{R}/3$ multiplication. One each is performed during the core decapsulation itself, and a further $\mathcal{R}/q$ multiplication is done during the re-encapsulation for the Fujisaki-Okamoto transformation [10]. The cycles spent for the re-encapsulation differ from a normal encapsulation in that no short polynomials have to be generated, and the decoding of the public key is

not needed, as it was already decoded as part of the secret key. Once again, decoding, encoding and hashing are almost negligible. As such, improving the $\mathcal{R}/q$ multiplication speed can significantly improve decapsulation speed. Encapsulation can also be improved, but not to such an extent, as the time taken for the generation of the short polynomials will quickly dominate. The cycles for the generation of the short polynomials are almost entirely due to the sorting algorithm, which takes 97.7% of the cycles. Thus, a faster sorting method would also lead to a speed improvement of the encapsulation. Key generation cannot be meaningfully improved by either faster $\mathcal{R}/q$ multiplication or faster sorting, as both only take a small percentage of the total cycles.

In the Xilinx Zynq FPGA, each Block RAM is 36 kilobits. They can be split into two 18 kb BRAMs. The majority of the BRAM are used by the en- and decoding, the $\mathcal{R}/q$ polynomial inversion, and the $\mathcal{R}/q$ multiplication. In several instances, we explicitly use disturbed RAM instead, in order to save BRAM resources from being consumed for very small memories. Examples for this is the storage of small polynomials, which only require 1522 bits. Note that all memory is instantiated where needed, i.e. there is no central memory. This means that there is no sharing or overlapping of memory.

The Xilinx FPGA has numerous Digital Signal Processor (DSP) cells, which can be used for fast addition and multiplication. Of the 23 DSP cells, 11 are used for the $\mathcal{R}/q$ polynomial inversion during key generation. More specifically, they are used for the modular multiplication and modular addition during the inversion (see also Section 4.4 on Barrett reduction). A further five DSP cells are used for the constant integer division during decoding. Only two DSPs are used during the core en- and decapsulation.

To test our implementation for correctness we use the known answer test (KAT) also used by the reference code. In total, we run 50 KATs, for key generation, encapsulation and decapsulation. In all cases, the outputs of our design match that of the reference code. We used the default synthesis and implementation settings of Vivado (version 2018.3.1), with two exceptions: We turn on register retiming, and set the synthesis hierarchy parameter to full.

6.1 Side Channels

The implementation is fully constant time, i.e. all operations are timing independent with regards to secret input. The case during key generation where the inversion of g fails is not relevant here, as the polynomial is discarded, and a new one generated. That being said, we did not employ any protections against more advanced side channels, such as DPA, nor invasive attacks such as fault attacks. There have been side channel attacks on the polynomial multiplication part of lattice schemes [19, 13]. Many of these attacks were applied on software implementations, some were also applied to hardware implementations [2]. There have also been side channel attacks that exploit decryption failures of lattice schemes

[4]. As Streamlined NTRU Prime does not have any decryption failures, these attacks do not apply. The authors from [11] show an attack on the re-encapsulation (based on the Fujisaki-Okamoto transformation [10]) of the FrodoKEM scheme, due to the ciphertext comparison being non constant time. This attack also does not apply to our implementation, as the comparison is completely constant time. We plan on investigating further side channel resistance in future work.

6.2 Comparison With Other Implementations

To our knowledge, there are no other complete pure hardware implementations of Streamlined NTRU Prime. The authors of [9] have a hardware software co/design, where some parts of the algorithm are implemented in hardware. Specifically, they have not implemented the key generation and decoding. Overall, our design uses significantly fewer LUTs and flip-flops, even when considering that our design implements more functionality in hardware (see Table 5). [9] however does not use any DSP units. Our design has an initially noticeable higher usage of block RAMs, but this is due to the inclusion of the key generation and decoding. When these are counted separately, our design uses the same amount of BRAMs. Our design runs at a slightly faster clock speed, though the total encapsulation and decapsulation time of [9] is shorter, as can be seen in Table 5. The main reason for this difference is in the implementation of the $\mathcal{R}/q$ multiplication. Our design uses a combination of Karatsuba and schoolbook multiplication, whereas [9] uses an linear feedback shift register (LFSR). While using an LFSR is faster, the resource cost is significantly higher. In addition, our multiplication design has the additional advantage that it can be easily tweaked, as the numbers of Karatsuba layers can be modified. Using more Karatsuba layers would increase performance, though also increase resource cost. See Table 5 for a full comparison of all metrics.

Table 5: A comparison of our design (both the full version, and once without the key generation and decoding) and existing Streamlined NTRU Prime implementations for the parameter set $p = 761$, $q = 4591$ and $w = 246$. For the implementation from [9], we only consider the timing from the hardware design.

Design	Slices	LUT	Flip-flops	BRAM	DSP	Clock Speed	Encap Time	Decap Time
Ours	1841	9538	7803	14	19	271 MHz	524 μs	958 μs
Ours, without key gen or decoding	1261	6240	6223	9	3	279 MHz	483 μs	901 μs
[9], no key gen or decoding	10 319	70 066	38 144	9	0	263 MHz	56.3 μs	53.3 μs

6.3 Potential Improvements

In this section, we describe a number of potential improvements that we have not yet implemented or tested. We plan on investigating these improvements in future work.

An option would be to use an LFSR polynomial multiplier for the $\mathcal{R}/3$ multiplier only. As polynomials in $\mathcal{R}/3$ only require 2 bits of storage per coefficient, the resource cost is not as high as with a full $\mathcal{R}/q$ multiplier. Doing so would speed up decapsulation. In a similar vein, the $\mathcal{R}/q$ Karatsuba multiplication currently uses only a single layer of Karatsuba. Implementing more layers would bring further speedups, though also increase resource cost. In particular, the amount of memory needed to store intermediate results would increase, leading to an increase of BRAMs usage. In addition, the partial multiplier would increase in complexity: As the partial factors are added together, previously small factors become more and more complex, requiring an increasingly complex modular multiplication circuit.

During the key generation, the reciprocal algorithms work on a single coefficient per clock cycle. However, there is no interdependence between the coefficients. As such, it would be possible to operate on batches of coefficients. On Xilinx FPGAs, the BRAMs can have word width of up to 72 bits. For the $\mathcal{R}/q$ inversion, this would be enough to read five coefficients per clock cycle. This would speed up the inversion by roughly a factor of five. However, this would also significantly increase the resource consumption of the inversion, as we would need to duplicate the modular multiplication circuits five times. Since the modular multiplication during the $\mathcal{R}/q$ inversion currently requires six DSP units, the duplication would increase this to 30 DSP units.

Currently, the design has a single global clock. However, not all parts are required to run at the same speed. Parts such as the hashing, en- and decoding, which take only a small duration of the overall time, could be clocked at a slower speed, saving power, as well as reducing the number of pipeline steps needed. This is especially true for the hash function. We believe that there is still a potential for resource saving there, though finding the optimal SHA-512 implementation was considered out of scope for this work.

As mentioned previously, there is no central memory, instead block and distributed RAMs are inferred where needed. This aids performance and simplicity, but causes a larger memory footprint, as memory reuse is not possible. However, as many functions are executed sequentially, and not parallelly, sharing memory could save significant resources. At a higher level, this includes the fact that key generation, encapsulation and decapsulation cannot happen at the same time, thus memory space could be shared. At a lower level, this includes functions like the de- and encoding, which also never occur at the same time.

Currently, the design is also fixed to a single parameter set after synthesis. For greater flexibility, it would be useful if the design could switch between parameter sets during runtime.

7 Conclusion

We present the first full constant-time hardware implementation of the round 2 scheme Streamlined NTRU Prime. We implement the entire KEM in VHDL, including key generation, and all en- and decoding. Compared to existing partial implementations of Streamlined NTRU Prime, our design is slower, but uses significantly less resources. The source code of our implementation is available at `https://github.com/AdrianMarotzke/SNTRUP`.

Acknowledgments

I would like to thank Joppe Bos, Christine Van Vredendal, Björn Fey, Thomas Wille, Dieter Gollmann for their help. This work was supported by the Federal Ministry of Education and Research (BMBF) of the Federal Republic of Germany (grant 16KIS0658K, SysKit HW). This work was labelled by the EUREKA cluster PENTA and funded by German authorities under grant agreement PENTA-2018e-17004-SunRISE.

References

[1] Martin R Albrecht, Christian Hanser, Andrea Hoeller, Thomas Pöppelmann, Fernando Virdia, and Andreas Wallner. "Implementing RLWE-based Schemes Using an RSA Co-Processor". In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* (2019), pp. 169–208.

[2] Aydin Aysu, Youssef Tobah, Mohit Tiwari, Andreas Gerstlauer, and Michael Orshansky. "Horizontal side-channel vulnerabilities of post-quantum key exchange protocols". In: *2018 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*. IEEE. 2018, pp. 81–88.

[3] Paul Barrett. "Implementing the Rivest Shamir and Adleman public key encryption algorithm on a standard digital signal processor". In: *Conference on the Theory and Application of Cryptographic Techniques*. Springer. 1986, pp. 311–323.

[4] Daniel J Bernstein, Leon Groot Bruinderink, Tanja Lange, and Lorenz Panny. "HILA5 pindakaas: on the CCA security of lattice-based encryption with error correction". In: *International Conference on Cryptology in Africa*. Springer. 2018, pp. 203–216.

[5] Daniel J Bernstein, Chitchanok Chuengsatiansup, Tanja Lange, and Christine van Vredendaal. "NTRU Prime: reducing attack surface at low cost". In: *International Conference on Selected Areas in Cryptography*. Springer. 2017, pp. 235–260.

[6] Daniel J Bernstein, Chitchanok Chuengsatiansup, Tanja Lange, and Christine van Vredendaal. "NTRU Prime: round 2". In: *Post-Quantum Cryptography Standardization Project*. NIST. 2019.

[7] Daniel J Bernstein and Bo-Yin Yang. "Fast constant-time gcd computation and modular inversion". In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* (2019), pp. 340–398.

[8] Daniel J. Bernstein, Chitchanok Chuengsatiansup, Tanja Lange, and Christine van Vredendaal. *NTRU Prime: reducing attack surface at low cost*. Cryptology ePrint Archive, Report 2016/461. `https://eprint.iacr.org/2016/461`. 2016.

[9] Viet Ba Dang, Farnoud Farahmand, Michal Andrzejczak, Kamyar Mohajerani, Duc Tri Nguyen, and Kris Gaj. *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. `https://eprint.iacr.org/2020/795`. 2020.

[10] Eiichiro Fujisaki and Tatsuaki Okamoto. "Secure integration of asymmetric and symmetric encryption schemes". In: *Annual International Cryptology Conference*. Springer. 1999, pp. 537–554.

[11] Qian Guo, Thomas Johansson, and Alexander Nilsson. *A key-recovery timing attack on post-quantum primitives using the Fujisaki-Okamoto transformation and its application on FrodoKEM*. Cryptology ePrint Archive, Report 2020/743. `https://eprint.iacr.org/2020/743`. 2020.

[12] Jeffrey Hoffstein, Jill Pipher, and Joseph H Silverman. "NTRU: A ring-based public key cryptosystem". In: *International Algorithmic Number Theory Symposium*. Springer. 1998, pp. 267–288.

[13] Wei-Lun Huang, Jiun-Peng Chen, and Bo-Yin Yang. "Power Analysis on NTRU Prime". In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* (2020), pp. 123–151.

[14] Michael Hutter and Erich Wenger. "Fast multi-precision multiplication for public-key cryptography on embedded microprocessors". In: *International Workshop on Cryptographic Hardware and Embedded Systems*. Springer. 2011, pp. 459–474.

[15] Anatolii Alekseevich Karatsuba and Yu P Ofman. "Multiplication of many-digital numbers by automatic computers". In: *Doklady Akademii Nauk*. Vol. 145. 2. Russian Academy of Sciences. 1962, pp. 293–294.

[16] Daniel Lemire, Owen Kaser, and Nathan Kurz. "Faster remainder by direct computation: Applications to compilers and software libraries". In: *Software: Practice and Experience* 49.6 (2019), pp. 953–970.

[17] Pradeep Kumar Mishra and Palash Sarkar. "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*. Springer. 2004, pp. 41–54.

[18] NIST. *Nist post-quantum cryptography standardization*. `https://csrc.nist.gov/Projects/Post-Quantum-Cryptography/Post-Quantum-Cryptography-Standardization`. [Online; accessed 11-June-2020].

[19] Robert Primas, Peter Pessl, and Stefan Mangard. "Single-trace side-channel attacks on masked lattice-based encryption". In: *International Confer-*

ence on Cryptographic Hardware and Embedded Systems. Springer. 2017, pp. 513–533.

[20] Danny Savory. *SHA-512 hardware implementation in VHDL. Based on NIST FIPS 180-4.* `https://github.com/dsaves/SHA-512`. [Online; accessed 11-June-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 1.3: The Python code of the encoder [6]. 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 1.4: The Python code of the decoder [6].