Compact Ring-LWE based Cryptoprocessor

Sujoy Sinha Roy[1], Frederik Vercauteren[1], Nele Mentens[1],
Donald Donglong Chen[2] and Ingrid Verbauwhede[1]

[1]ESAT/SCD-COSIC and iMinds, KU Leuven
Kasteelpark Arenberg 10, B-3001 Leuven-Heverlee, Belgium
Email: {firstname.lastname}@esat.kuleuven.be

[2]Department of Electronic Engineering,
City University of Hong Kong
Tat Chee Avenue, Kowloon, Hong Kong SAR
Email: donald.chen@my.cityu.edu.hk

Abstract. In this paper we propose an efficient and compact processor for a ring-LWE based encryption scheme. We present three optimizations of the Number Theoretic Transform (NTT) used for polynomial multiplication: we avoid pre-processing in the negative wrapped convolution by merging it with the main algorithm, we reduce the fixed computation cost of the twiddle factors and propose an advanced memory access scheme. These optimization techniques reduce both the cycle and memory requirements. Finally, we also propose an optimization of the ring-LWE encryption system that reduces the number of NTT operations from five to four resulting in 20% speed-up. We use these computational optimizations along with several architectural optimizations to implement an instruction-set ring-LWE based cryptoprocessor. For dimension 512, corresponding to a high security level, our processor performs encryption/decryption operations in $53/21\mu s$ on a Virtex 6 FPGA and only requires 1879 LUTs, 1142 FFs and 3 BRAMs. Our processor is therefore three times smaller than the current state of the art hardware implementations, whilst running somewhat faster.

Keywords. Lattice-based cryptography, ring-LWE, Polynomial multiplication, Number Theoretic Transform, Hardware implementation

1 Introduction

Lattice-based cryptography is considered a prime candidate for quantum-secure public key cryptography due to its wide applicability [20] and its security proofs that are based on worst-case hardness of well known lattice problems. The *learning with errors* (LWE) problem [19] and its ring variant known as ring-LWE [11] have been used as a solid foundation for several cryptographic schemes. The significant progress in theoretical lattice-based cryptography [13, 14, 18] has recently been followed by practical implementations [1, 5, 7, 16, 17, 21].

The ring-LWE based cryptosystems operate in a polynomial ring $R_q = \mathbb{Z}_q[\mathbf{x}]/\langle f(x) \rangle$, where one typically chooses $f(x) = x^n + 1$ with n a power of two, and q a prime with $q \equiv 1 \bmod 2n$. An implementation thus requires the basic operations in such a ring R_q, with multiplication taking up the bulk of the resources both in area and time. An efficient polynomial multiplier architecture therefore is a pre-requisite for the deployment of ring-LWE based cryptography in real world systems.

The most important hardware implementations of polynomial multipliers for rings R_q are [1, 7, 16, 17]. In [7], a fully parallel butterfly structure is used for the polynomial multiplier resulting in a huge area consumption. For instance, even for medium security, their ring-LWE cryptoprocessor does not fit on the largest FPGA of the Virtex 6 family. In [16], a sequential polynomial multiplier architecture is designed to use the FPGA resources in an efficient way. The multiplier uses a dedicated ROM to store all the twiddle factors which are required during the NTT computation. In [17] the authors integrated the polynomial multiplier [16] in a complete ring-LWE based encryption system and propose several system level optimizations such as a better message encoding scheme and compression technique for the ciphertext. The work [1] tries to reduce the area of the polynomial multiplier by computing the twiddle factors whenever required, but as we will show, could be improved substantially by re-arranging the loops inside the NTT computation. Furthermore, the paper does not include an implementation of a complete ring-LWE cryptoprocessor.

Our contributions: In this paper we present a complete ring-LWE based encryption processor that uses the Number Theoretic Transform (NTT) algorithm for polynomial multiplication. The architecture is designed to have small area and memory requirement, but is also optimized to keep the number of cycles small. In particular, we make the following contributions:

1. During the NTT computation, the intermediate coefficients are multiplied by the twiddle factors that are computed using repeated multiplications. In [16] a pre-computed table (ROM) is used to avoid this fixed computation cost. The more compact implementation in [1] does not use ROM and computes the twiddle factors by performing repeated multiplications. In this paper we reduce the number of multiplications by re-arranging the nested loops in the NTT computation.
2. The implementations [1, 16] use negative wrapped convolution to reduce the number of evaluations in both the forward and backward NTT computations. However, the use of the negative wrapped convolution has a pre- and post-computation overhead. In this paper we basically avoid the pre-computation which reduces the cost of the forward NTT.
3. The intermediate coefficients are stored in memory (RAM) during the NTT computation. Access to the RAM is a bottleneck for speeding-up the NTT computation. In the implementations [1, 16], FPGA-RAM slices are placed in parallel to avoid this bottleneck. In this paper we propose an efficient memory access scheme which reduces the number of RAM accesses, optimizes the number of block RAMs and still achieves maximum utilization of the computational blocks present in the polynomial multiplier.
4. The proposed optimization techniques are applied to design a compact architecture for the NTT computation suitable for resource constrained platforms. We also implement pipelines in the architecture targeting high-speed applications. The pipeline technique derives an optimal pipeline depth for the architecture to achieve the fastest computation time.
5. Finally, we optimize one of the most popular ring-LWE encryption schemes by reducing the number of NTT computations from five to four, thereby achieving a nearly 20% reduction in the computation cost.

The above optimizations result in a very compact architecture that uses three times less resources than the current state of the art implementation [17] and even achieves somewhat faster computation time.

The remainder of the paper is organized as follows: In Section 2 we provide a brief mathematical background on ring-LWE and the NTT. Section 3 contains our optimization techniques of the NTT and Section 4 presents the actual architecture of our optimized NTT algorithm. A pipelined architecture is given in Section 5. In Section 6, we propose an optimization of an existing ring-LWE encryption scheme and propose an efficient architecture for the complete ring-LWE encryption system. Finally, Section 7 reports on the experimental results of this implementation.

2 Background

In this section we present a brief mathematical overview of the ring-LWE problem, the encryption scheme we will be optimizing and the NTT.

2.1 The LWE and ring-LWE Problem

The *learning with errors* (LWE) problem is a machine learning problem that is equivalent to worst-case lattice problems as shown by Regev [19] in 2005. Since then, the LWE problem has become popular as a basis for developing quantum secure lattice-based cryptosystems.

The LWE problem is parametrized by a dimension $n \geq 1$, an integer modulus $q \geq 2$ and an error distribution, typically a discrete Gaussian distribution with deviation σ and mean 0, $\mathcal{X}$ over the integers. The probability of sampling an integer $z \in \mathbb{Z}$ in the Gaussian distribution $\mathcal{X}_\sigma$ is given by $\rho_\sigma(z)/\rho_\sigma(\mathbb{Z})$ where $\rho_\sigma(z) = \exp\left(\frac{-z^2}{2\sigma^2}\right)$ and $\rho_\sigma(\mathbb{Z}) = \sum_{z=-\infty}^{+\infty} \rho_\sigma(z)$. Note that some authors use the parameter $s = \sqrt{2\pi}\sigma$ to define the Gaussian distribution or even denote the parameter s by σ to add to the confusion.

For a uniformly chosen $\mathbf{s} \in \mathbb{Z}_q^n$, the LWE distribution $A_{s,\mathcal{X}}$ over $\mathbb{Z}_q^n \times \mathbb{Z}_q$ consists of tuples $(\mathbf{a}, t)$ where $\mathbf{a}$ is chosen uniformly from $\mathbb{Z}_q^n$ and $t = \langle \mathbf{a}, \mathbf{s} \rangle + e \mod q \in \mathbb{Z}_q$ and e is sampled from the error distribution $\mathcal{X}$. The *search* version of the LWE problem asks to find $\mathbf{s}$ given a polynomial number of pairs $(\mathbf{a}, t)$ sampled from the LWE distribution $A_{s,\mathcal{X}}$. In the *decision* version of the LWE problem, the solver needs to distinguish with non-negligible advantage between a polynomial number of samples drawn from $A_{s,\mathcal{X}}$ and the same number of samples drawn from $\mathbb{Z}_q^n \times \mathbb{Z}_q$. For hardness proofs of the search and decision LWE problems, interested readers are referred to [10].

The initial LWE encryption system in [19] is based on matrix operations which are quite inefficient and result in large key sizes. To achieve computational efficiency and to reduce the key size, an algebraic variant of the LWE called *ring*-LWE [11] uses special structured ideal lattices. Such lattices correspond to ideals in rings $\mathbb{Z}[\mathbf{x}]/\langle f \rangle$, where f is an irreducible polynomial of degree n. For efficiency reasons, the ring is often taken as $R_q = \mathbb{Z}_q[\mathbf{x}]/\langle f \rangle$ with $f(x) = x^n + 1$, where n is a power of two and the prime q is taken as $q \equiv 1 \mod 2n$. The ring-LWE distribution on $R_q \times R_q$ consists of tuples (a, t) with $a \in R_q$ chosen uniformly random and $t = as + e \in R_q$, where $s \in R_q$ is a fixed secret element and e has small coefficients sampled from the discrete Gaussian above. The resulting distribution on R_q will also be denoted $\mathcal{X}_\sigma$.

The ring-LWE based encryption scheme that we will use was introduced in the full version of [11] and uses a global polynomial $a \in R_q$. Key generation, encryption and decryption are as follows:

1. $KeyGen(a)$: Choose two polynomials $r_1, r_2 \in R_q$ from $\mathcal{X}_\sigma$ and compute $p = r_1 - a \cdot r_2 \in R_q$. The public key is (a, p) and the private key is r_2. The polynomial r_1 is simply noise and is no longer required after key generation.
2. $Enc(a, p, m)$: The message m is first encoded to $\bar{m} \in R_q$. Three polynomials $e_1, e_2, e_3 \in R_q$ are sampled from $\mathcal{X}_\sigma$. The ciphertext then consists of two polynomials $c_1 = a \cdot e_1 + e_2$ and $c_2 = p \cdot e_1 + e_3 + \bar{m} \in R_q$.
3. $Dec(c_1, c_2, r_2)$: Compute $m' = c_1 \cdot r_2 + c_2 \in R_q$ and recover the original message m from m' using a decoder.

One of the simplest encoding functions maps a binary message m to the polynomial $\bar{m} \in R_q$ such that its i-th coefficient is $(q-1)/2$ iff the i-th bit of m is 1 and 0 otherwise. The corresponding decoding function then simply reduces the coefficients m'_i of m' in the interval $(-q/2, q/2]$ and decodes to 1 when $|m'_i| > q/4$ and 0 otherwise.

The parameters that we use in our implementation follow the security analyzes of [10, 22, 9]. Since we mainly focus on a high security level implementation, we focus on one parameter set namely: dimension $n = 512$, deviation $\sigma = 3.2$ (to obtain a minimum $s = 8$) and prime $q = 2^{20} + 3 \cdot 2^{10} + 1$. The prime q was chosen such that $q \equiv 1 \bmod 2n$ with the consideration that it is smaller than the maximum q following the analysis in [9] and also large enough such that no decryption errors occur. We limit the Gaussian sampler in our implementation to 6σ, meaning that our samples are bounded to $[-6\sigma, 6\sigma]$. Although one can normally sample the secret $r_2 \in R_q$ also from the distribution $\mathcal{X}_\sigma$, we restrict r_2 to have binary coefficients.

Algorithm 1: *Iterative NTT*

```
    Input: Polynomial $a(x) \in \mathbb{Z}_q[\mathbf{x}]$ of degree $N-1$ and $N$-th primitive root $\omega_N \in \mathbb{Z}_q$ of unity
    Output: Polynomial $A(x) \in \mathbb{Z}_q[\mathbf{x}] = \mathrm{NTT}(a)$
 1  begin
 2      $A \leftarrow BitReverse(a)$;
 3      for $m = 2$ to $n$ by $m = 2m$ do
 4          $\omega_m \leftarrow \omega_N^{N/m}$ ;
 5          $\omega \leftarrow 1$ ;
 6          for $j = 0$ to $m/2 - 1$ do
 7              for $k = 0$ to $n - 1$ by $m$ do
 8                  $t \leftarrow \omega \cdot A[k + j + m/2]$ ;
 9                  $u \leftarrow A[k + j]$ ;
10                  $A[k + j] \leftarrow u + t$ ;
11                  $A[k + j + m/2] \leftarrow u - t$ ;
12              end
13              $\omega \leftarrow \omega \cdot \omega_m$ ;
14          end
15      end
16  end
```

2.2 The Number Theoretic Transform

There are many efficient algorithms in the literature to perform polynomial multiplication and a survey of fast multiplication algorithms can be found in [2]. In this section we review the Number Theoretic Transform (NTT) which corresponds to a Fast Fourier Transform (FTT) where the roots of unity are taken from a finite ring instead of the complex numbers.

The FFT and NTT: Recall that the N-point FFT (with $N = 2^k$) is an efficient method to evaluate a polynomial $a(x) = \sum_{j=0}^{N-1} a_j x^j \in \mathbb{Z}[x]$ in the N-th roots of unity ω_N^i for $i = 0, \ldots, N-1$ where ω_N denotes a primitive N-th root of unity. More precisely, on input the coefficients $[a_0, \ldots, a_{N-1}]$ and ω_N, the FFT computes $FFT([a_j], \omega_N) = [a(\omega_N^0), a(\omega_N^1), \ldots, a(\omega_N^{N-1})]$ in $\theta(N \log N)$ time. Due to the orthogonality relations between the N-th roots of unity, we can compute the inverse FFT simply as $\frac{1}{N} FFT(\cdot, \omega_N^{-1})$.

The NTT replaces the complex roots of unity by roots of unity in a finite ring $\mathbb{Z}_q$. Since we require elements of order N, q is chosen to be a prime with $q \equiv 1 \bmod N$. Note furthermore that the NTT immediately leads to a fast multiplication algorithm in the ring $S_q = \mathbb{Z}_q[x]/(x^N - 1)$: indeed, given two polynomials $a, b \in S_q$ we can easily compute their (reduced) product $c = a \cdot b \in S_q$ by computing

$$c = NTT_{\omega_N}^{-1}\big(NTT_{\omega_N}(a) * NTT_{\omega_N}(b)\big)\,, \tag{1}$$

where $*$ denotes point-wise multiplication.

The NTT computation is usually described recursive, but in practice we use an in-place iterative version taken from [3] that is given in Algorithm 1. For the inverse NTT, an additional scaling of the resulting coefficients by N^{-1} is performed. The factors ω used in line 8 are called the *twiddle factors*.

Multiplication in R_q: Recall that we will use $R_q = Z_q[\mathbf{x}]/\langle f \rangle$ with $f = x^n + 1$ and $n = 2^k$. Since $f(x) | x^{2n} - 1$ we could use the $2n$-point NTT to compute the multiplication in R_q at the expense of three $2n$-point NTT computations and a reduction by trivially embedding the ring R_q into S_q, i.e. expanding the coefficient vector of a polynomial $a \in R_q$ by adding n extra zero coefficients. However, we can do much better by exploiting the special relation between the roots of $x^n + 1$ and $x^{2n} - 1$ using a technique known as the *negative wrapped convolution.*

Indeed, using the same evaluation-interpolation strategy used above for the ordinary NTT, we conclude that we can efficiently multiply two polynomials $a, b \in R_q$ if we can quickly evaluate them in the roots of f. These roots are simply ω_{2n}^{2j+1} for $j = 0, \ldots, n-1$ (since the even exponents give the roots of $x^n - 1$) and as such can be written as $\omega_{2n} \cdot \omega_n^j$. These evaluations can thus be computed efficiently using a *classical* n-point NTT (instead of a $2n$-point NTT) on the scaled polynomials $a'(x) = a(\omega_{2n} \cdot x)$ and $b'(x) = a(\omega_{2n} \cdot x)$. The point-wise multiplication gives the evaluations of $c(x) = a(x)b(x) \bmod f(x)$ in the roots of f, and the classical inverse n-point NTT thus results in the coefficients of the scaled polynomial $c'(x) = c(\omega_{2n} \cdot x)$. To recover the coefficients c_i of $c(x)$, we therefore simply have to compute $c_i = c'_i \cdot \omega_{2n}^{-i}$. Note that the scaling operation by n^{-1} can be combined with the multiplications of c'_i by ω_{2n}^{-i}.

3 Optimization in the NTT Computation

In this section we optimize the NTT and compare with the recent hardware implementations of polynomial multipliers [1, 16, 17]. First, the fixed cost involved in computing the powers of ω_n is reduced, then the pre-computation overhead in the forward negative-wrapped convolution is optimized, and finally an efficient memory access scheme is proposed that reduces the number of memory accesses during the NTT and also minimizes the number of block RAMs in the hardware architecture.

3.1 Optimizing the Fixed Computation Cost

In line 13 of Algorithm 1 the computation of the twiddle factor $\omega \leftarrow \omega \cdot \omega_m$ is performed in the j-loop. This computation can be considered as a fixed cost. However in [1, 16] the j-loop and the k-loop are interchanged, such that ω is updated in the innermost loop which is much more frequent than in Algorithm 1. To avoid the computation of the twiddle factors, in [16] all the twiddle factors are kept in a pre-computed look-up table (ROM) and are accessed whenever required. As the twiddle factors are not computed on-the-fly, the order of the two innermost loops does not result in an additional cost. However in [1] a more compact polynomial multiplier architecture is designed without using any look-up table and the twiddle factors are simply computed on-the-fly during the NTT computation. Hence in this implementation, the interchanged loops cause substantial additional computational overhead. In this paper our target is to design a very compact polynomial multiplier. Hence we do not use any look-up table for the twiddle factors and follow Algorithm 1 to avoid the extra computation of [1].

3.2 Optimizing the Forward NTT Computation Cost

Here we revisit the forward negative-wrapped convolution technique used in [1, 16, 17]. Recall that the negative-wrapped convolution corresponds to a classical n-point NTT on the scaled polynomials $a'(x) = a(\omega_{2n} \cdot x)$ and $b'(x) = (\omega_{2n} \cdot x)$. Instead of first pre-computing these scaled polynomials and then performing a classical NTT, it suffices to note that we can integrate the scaling and the NTT computation. Indeed, it suffices to change the initialization of the twiddle factors in line 5 of Algorithm 1: instead of initializing ω to 1, we can simply set $\omega = \omega_{2m}$. The rest of the algorithm remains exactly the same, and no pre-computation is necessary. Note that this optimization only applies to the NTT itself and not to the inverse NTT.

3.3 Optimizing the Memory Access Scheme

The NTT computation requires memory to store the input and intermediate coefficients. When the number of coefficients is large, RAM is most suitable for hardware implementation [1, 16, 17]. In the innermost loop (lines 8-to-11) of Algorithm 1, two coefficients $A[k+j]$ and $A[k+j+m/2]$ are first read from memory and then arithmetic operations (one multiplication, one addition and one subtraction) are performed. The new $A[k+j]$ and $A[k+j+m/2]$ are then written back in memory. During one iteration of the innermost loop, the arithmetic circuits are thus used only once, while the memory is read or written twice. This leads to idle cycles in the arithmetic circuits. The polynomial multiplier in [16] uses two parallel memory blocks to provide a continuous flow of coefficients to the arithmetic circuits. However this approach could result in under-utilization of the RAM blocks if the coefficient size is much smaller than the word size (for example in the ring-LWE cryptosystem [11]). In literature there are many papers on efficient memory management schemes using segmentation and efficient address generation (see [12]) for the classical FFT algorithm. Another well known approach is the constant geometry FFT (or NTT) which always maintains a constant index difference between the processed coefficients [15]. However the constant geometry algorithm is not inplace and hence not suitable for resource constrained platforms. In this paper we propose a memory access scheme which is designed to minimize the number of block RAM slices and to achieve maximum utilization of computational circuits present in the NTT architecture.

Since the two coefficients $A[k+j]$ and $A[k+j+m/2]$ are processed together in Algorithm 1, we keep the two coefficients as a pair in one memory location. Let us analyze two consecutive iterations of the m-loop (line 3 in Algorithm 1) for $m = m_1$ and $m = m_2$ where $m_2 = 2m_1$. In the m_1-loop, for some j_1 and k_1 (maintaining the loop bounds in Algorithm 1) the coefficients $(A[k_1 + j_1], A[k_1 + j_1 + m_1/2])$ are processed as a pair. Then k increments to $k_1 + m_1$ and the processed coefficient pair is $(A[k_1 + m_1 + j_1], A[k_1 + m_1 + j_1 + m_1/2])$. Now from Algorithm 1 we see that the coefficient $A[k_1 + j_1]$ will again be processed in the m_2-loop with coefficient $A[k_1+j_1+m_2/2]$. Since $m_2 = 2m_1$, the coefficient $A[k_1+j_1+m_2/2]$ is the coefficient $A[k_1+j_1+m_1]$ which is updated in the m_1-loop for $k = k_1 + m_1$. Hence during the m_1-loop if we swap the updated coefficients for $k = k_1$ and $k = k_1 + m_1$ and store $(A[k_1 + j_1], A[k_1 + j_1 + m_1])$ and $(A[k_1 + j_1 + m_1/2], A[k_1 + j_1 + 3m_1/2])$ as the coefficient pairs in memory, then the coefficients in a pair have a difference of $m_2/2$ in their index and thus are ready for the m_2-loop. The operations during the two consecutive iterations $k = k_1$ and $k = k_1 + m_1$ during $m = m_1$ are shown in Algorithm 2 in lines 8-15. During the operations u_1, t_1, u_2 and t_2 are used as temporary storage registers.

A complete description of the efficient memory access scheme is given in Algorithm 2. In this algorithm for all values of $m < n$, two coefficient pairs are processed in the innermost loop and a swap of the updated coefficients is performed before writing back to memory. For $m = n$, no swap operation of the updated coefficients is required as this is the final iteration of the m-loop. The coefficient pairs generated by Algorithm 2 can be re-arranged easily for another (say inverse) NTT operation by performing address-wise bit-reverse-swap operation. Appendix A describes the memory access scheme using an example.

4 The NTT Processor Organization

In this section we present an architecture for performing the forward and backward NTT using the proposed optimization techniques. Our NTT processor (Figure 1) consists of three main components: the arithmetic unit, the memory block and the control-address unit.

The Memory Block is implemented as a simple dual port RAM. To accommodate two coefficients, the word size is $2\lceil \log q \rceil$ where q is the prime modulus. In FPGAs, a RAM can be implemented as a *distributed* or as a *block* RAM. When the amount of data is large, block RAM is the ideal choice. In Virtex 6 FPGAs, a block RAM has word size 36 bits. For our parameter set, the prime modulus q is slightly larger than 18 bits, and the extra bits are accommodated in a narrow distributed RAM instead of using another block RAM.

The Arithmetic Unit (NTT-ALU) is designed to support Algorithm 2 along with other operations such as polynomial addition, point-wise multiplication and rearrangement of the co-

Algorithm 2: *Iterative NTT : Memory Efficient Version*

```
    Input: Polynomial $a(x) \in \mathbb{Z}_q[\mathbf{x}]$ of degree $n-1$ and $n$-th primitive root $\omega_n \in \mathbb{Z}_q$ of unity
    Output: Polynomial $A(x) \in \mathbb{Z}_q[\mathbf{x}] = \mathrm{NTT}(a)$
 1  begin
 2      $A \leftarrow BitReverse(a)$; /* Coefficients are stored in the memory as proper pairs */
 3      for $m = 2$ to $n/2$ by $m = 2m$ do
 4          $\omega_m \leftarrow m$-th primitiveroot(1) ;
 5          $\omega \leftarrow squareroot(\omega_m)$ or 1 /* Depending on forward or backward NTT */ ;
 6          for $j = 0$ to $m/2 - 1$ do
 7              for $k = 0$ to $n/2 - 1$ by $m$ do
 8                  $(t_1, u_1) \leftarrow (A[k+j+m/2], A[k+j])$ /* From MEMORY[k+j] */ ;
 9                  $(t_2, u_2) \leftarrow (A[k+j+m], A[k+j+m/2])$ /* MEMORY[k+j+m/2] */ ;
10                  $t_1 \leftarrow \omega \cdot t_1$ ;
11                  $t_2 \leftarrow \omega \cdot t_2$ ;
12                  $(A[k+j+m/2], A[k+j]) \leftarrow (u_1 - t_1, u_1 + t_1)$ ;
13                  $(A[k+m+j+m/2], A[k+m+j]) \leftarrow (u_2 - t_2, u_2 + t_2)$ ;
14                  $MEMORY[k+j] \leftarrow (A[k+j+m], A[k+j])$ ;
15                  $MEMORY[k+j+m/2] \leftarrow (A[k+j+3m/2], A[k+j+m/2])$ ;
16              end
17              $\omega \leftarrow \omega \cdot \omega_n$ ;
18          end
19      end
20      $m \leftarrow n$ ;
21      $k \leftarrow 0$ ;
22      $\omega \leftarrow squareroot(\omega_m)$ or 1 /* Depending on forward or backward NTT */ ;
23      for $j = 0$ to $m/2 - 1$ do
24          $(t_1, u_1) \leftarrow (A[j+m/2], A[j])$ /* From MEMORY[j] */ ;
25          $t_1 \leftarrow \omega \cdot t_1$ ;
26          $(A[j+m/2], A[j]) \leftarrow (u_1 - t_1, u_1 + t_1)$ ;
27          $MEMORY[j] \leftarrow (A[j+m/2], A[j])$ ;
28          $\omega \leftarrow \omega \cdot \omega_m$ ;
29      end
30  end
```

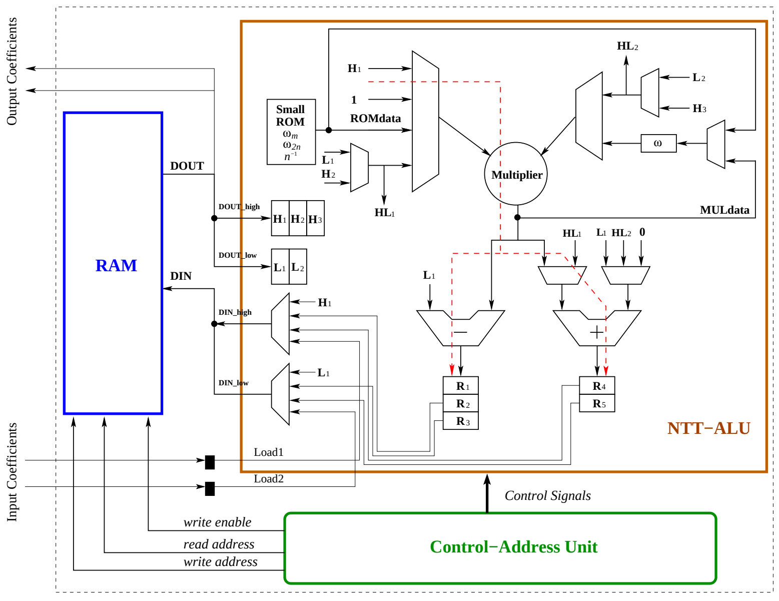

Fig. 1. Hardware Architecture for NTT

efficients. This NTT-ALU is interfaced with the memory block and the control-address unit. The central part of the NTT-ALU consists of a modular multiplier and addition/subtraction circuits.

Now we describe how the different components of the NTT-ALU are used during the butterfly steps (excluding the last loop for $m = n$). First, the memory location $(k + j)$ is fetched and then the fetched data (t_1, u_1) is stored in the input register pair (H_1, L_1). The same also happens for the memory location $(k+j+m/2)$ in the next cycle. The multiplier computes $\omega \cdot H_1$ and the result is added to or subtracted from L_1 using the adder and subtracter circuits to compute $(u_1 + \omega t_1)$ and $(u_1 - \omega t_1)$ respectively. In the next cycle the register pair (R_1, R_4) is updated with $(u_1 - \omega t_1, u_1 + \omega t_1)$. Another clock transition shifts the contents of (R_1, R_4) to (R_2, R_5). In this cycle the pair (R_1, R_4) is updated with $(u_2 - \omega t_2, u_2 + \omega t_2)$ as the computation involving (u_2, t_2) from the location $(k+j+m/2)$ lags by one cycle. Now the memory location $(k+j)$ is updated with the register pair (R_4, R_5) containing $(u_2 + \omega t_2, u_1 + \omega t_1)$. Finally, in the next cycle the memory location $(k+j+m/2)$ is updated with $(u_2 - \omega t_2, u_1 - \omega t_1)$ using the register pair (R_2, R_3). The execution of the *last* m-loop is similar to the intermediate loops, without any data swap between the output registers. The register pair (R_2, R_5) is used for updating the memory locations. In Figure 1, the additional registers (H_2, H_3 and L_2) and multiplexers are used for supporting operations such as addition, point-wise multiplication and rearrangement of polynomials. The Small-ROM block contains the fixed values ω_m, ω_{2n}, their inverses and n^{-1}. This ROM has depth of order $\log(n)$

The Control-and-Address Unit consists of three counters for m, j and k in Algorithm 2 and comparators to check the terminal conditions during the execution of any loop. The read address is computed from m, j and k and then delayed using registers to generate the write address. The control-and-address unit also generates the write enable signal for the RAM and the control signals for the NTT-ALU.

5 Pipelining the NTT Processor

The maximum frequency of the NTT-ALU is determined by the critical path (red dashed line in Figure 1) that passes through the modular multiplier and the adder (or subtracter) circuits . To

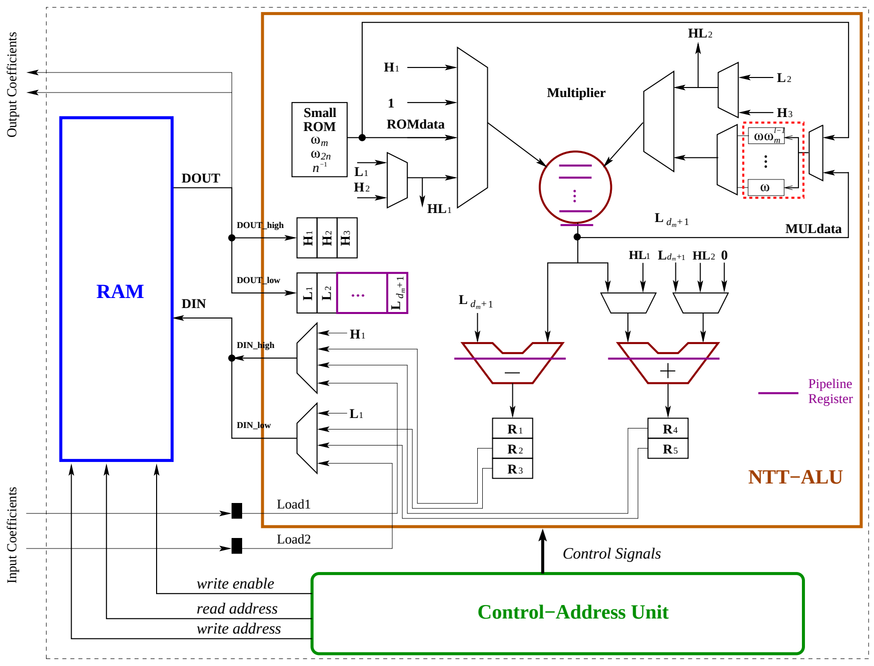

Fig. 2. Pipelined Hardware Architecture for NTT

increase the operating frequency of the processor, we implement efficient pipelines based on the following two observations.

Observation 1: During the execution of any m-loop in Algorithm 2, the computations (multiplication, addition and subtraction) involving a coefficient pair have no data dependency on other coefficient pairs. Such a data-flow structure is suitable for pipeline processing as different computations can be pipelined without inserting bubbles in the datapath.

Assume the modular multiplier has d_m pipeline stages and that the output is latched in a buffer. In the $(d_m + 1)$th cycle after the initiation of $\omega \cdot t_1$, the buffer is updated $\omega \cdot t_1$. Now we need to compute $u_1 + \omega \cdot t_1$ and $u_1 - \omega \cdot t_1$ using the adder and subtracter circuits. Hence we delay the data u_1 by d_m cycles so that it appears as an input to the adder and subtracter circuits in the $(d_m + 1)$th cycle. This delay operation is performed with the help of a shift register $L_1, \ldots, L_{d_m+1}$ as shown in Figure 2.

Observation 2: Every increment of j in Algorithm 2 requires a new ω (line 17). If the multiplier has d_m pipeline stages, then the register-ω in Figure 1 is updated with the new value of ω in the $(d_m + 2)$th cycle. Since this new ω is used by the next butterfly operations, the data dependency results in an interruption in the chain of butterfly operations for $d_m + 1$ cycles. In any m-loop, the total number of such *interruption cycles* is $(m/2 - 1) \cdot (d_m + 1)$.

To reduce the number of interruption cycles, we use a small look-up table to store a few twiddle factors. Let the look-up table (red dashed rectangle in Figure 2) have l registers containing the twiddle factors $(\omega, \ldots \omega\omega_m^{l-1})$. This look-up table is used to provide the twiddle factors during the butterfly operations for say $j = j'$ to $j = j' + l - 1$. The next time j increments, new twiddle factors are required for the butterfly operations. We multiply the look-up table with ω_m^l to compute the next l twiddle factors $(\omega\omega_m^l, \ldots \omega\omega_m^{2l-1})$. The multiplications are independent of each other and hence can be processed in a pipeline. The butterfly operations are resumed after $\omega\omega_m^l$ is loaded in the look-up table. Thus using a small-look-up table of size l we reduce the number of interruption cycles to $(\frac{m}{2l} - 1) \cdot (d_m + 1)$. In our architecture we use $l = 4$; larger value of l will reduce the number of interruption cycles, but will cost additional registers.

Optimal Pipeline Strategy for Speed : During the execution of any m-loop in Algorithm 2, the number of butterfly operations is $n/2$. In the pipelined NTT-ALU, the cycle requirement for the $n/2$ butterfly operations is slightly larger than $n/2$ due to an initial overhead. The state machine jumps to the ω calculation state $\frac{m}{2l} - 1$ times resulting in $(\frac{m}{2l} - 1) \cdot (d_m + 1)$ interruption cycles. Hence the total number of cycles spent in executing any m-loop can be approximated as shown below:

$$Cycles_m \approx \frac{n}{2} + (\frac{m}{2l} - 1) \cdot (d_m + 1)$$

Let us assume the delay of the critical path with no pipeline stages is D_{comb}. When the critical path is split in balanced-delay stages using pipelines, the resulting delay (D_s) can be approximated as $\frac{D_{comb}}{(d_m+d_a)}$, where d_m and d_a are the number of pipeline stages in the modular multiplier and the modular adder (subtracter) respectively. Since the delay of the modular adder is small compared to the modular multiplier, we have $d_a \ll d_m$. Now the computation time for the m-loop is approximated as

$$T_m \approx \frac{D_{comb}}{(d_m + d_a)} \big[\frac{n}{2} + (\frac{m}{2l} - 1) \cdot (d_m + 1)\big] \approx D_s \frac{n}{2} + C_m \,.$$

Here C_m is constant (assuming $d_a \ll d_m$) for a fixed value of m. From the above equation we find that the minimum computation time can be achieved when D_s is minimum. Hence we pipeline the datapath to achieve minimum D_s. The DSP based coefficient multiplier is optimally pipelined using the Xilinx IPCore tool, while the modular reduction block is suitably pipelined by placing registers between the cascaded adder and subtracter circuits.

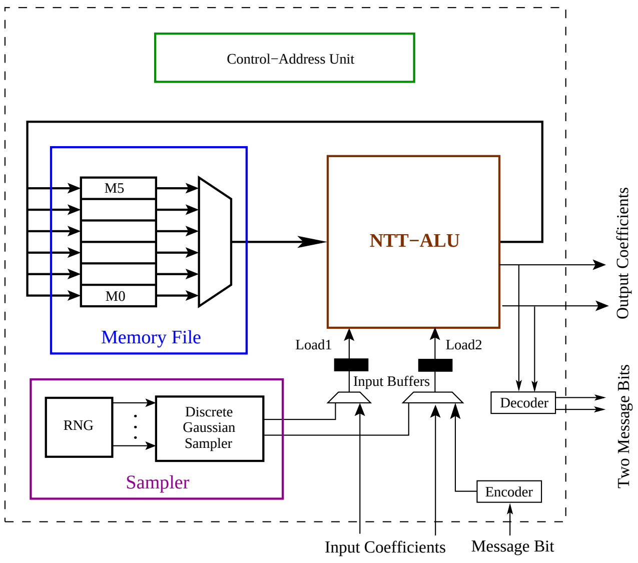

Fig. 3. Ring-LWE Cryptoprocessor

6 The ring-LWE Encryption Scheme

The ring-LWE encryption scheme in [17] optimizes computation cost by keeping the fixed polynomials in the NTT domain. The message encryption and decryption operations require three and two NTT computations respectively. In this paper we reduce the number of NTT operations for decryption from two to *one*. The proposed ring-LWE encryption scheme is described below:

1. $KeyGen(a)$: Choose a polynomial $r_1 \in R_q$ from $\mathcal{X}_\sigma$, choose another polynomial r_2 with binary coefficients and then compute $p = r_1 - a \cdot r_2 \in R_q$. The NTT is performed on the three polynomials a, p and r_2 to generate $\tilde{a}$, $\tilde{p}$ and $\tilde{r_2}$. The public key is $(\tilde{a}, \tilde{p})$ and the private key is $\tilde{r_2}$.
2. $Enc(\tilde{a}, \tilde{p}, m)$: The message m is first encoded to $\bar{m} \in R_q$. Three polynomials $e_1, e_2, e_3 \in R_q$ are sampled from $\mathcal{X}_\sigma$. The ciphertext is then computed as:

$$\tilde{e_1} \leftarrow NTT(e_1); \qquad \tilde{e_2} \leftarrow NTT(e_2)$$
$$(\tilde{c_1}, \tilde{c_2}) \leftarrow \left(\tilde{a} * \tilde{e_1} + \tilde{e_2};\ \tilde{p} * \tilde{e_1} + NTT(e_3 + \bar{m})\right)$$

3. $Dec(\tilde{c_1}, \tilde{c_2}, \tilde{r_2})$: Compute m' as $m' = INTT(\tilde{c_1} * \tilde{r_2} + \tilde{c_2}) \in R_q$ and recover the original message m from m' using a decoder.

The scheme requires both encryption and decryption to use a common primitive root of unity.

6.1 Hardware Architecture for the Ring-LWE Encryption Scheme

Figure 3 shows a hardware architecture for the ring-LWE encryption system. The basic building blocks used in the architecture are: the memory file, the arithmetic unit, the discrete Gaussian sampler and the control-address generation unit. The arithmetic unit is the NTT-ALU that we described in the previous section. Here we briefly describe the memory file and the discrete Gaussian sampler.

The Memory File is designed to support the maximum memory requirement that occurs during the encryption of the message. Six memory blocks M_0 to M_5 are available in the memory file and are used to store $\bar{a}$, $\bar{p}$, e_1, e_2, e_3 and $\bar{m}$ respectively. The memory blocks have width $2\lceil \log q \rceil$ bits and depth $n/2$. All six memory blocks share a common read and a write address and have a common data-input line, while their data-output lines are selected through a multiplexer. Any of the memory blocks in the memory file can be chosen for read and write operation. Due to the common addressing of the memory blocks, the memory file supports one read and one write operation in every cycle.

The Discrete Gaussian Sampler is based on the compact Knuth-Yao sampler [8] architecture proposed in [21]. Though the sampler is very compact it is also quite slow due to sequential scanning of the probability bits. We improve the cycle requirement of the sampler using a look-up table that maps eight random bits into a sample value or an intermediate distance in the 8th column of the probability matrix [21]. A successful look-up operation returns a sample and the sign of the sample is determined by the 9th random bit. In case the look-up operation fails, the Knuth-Yao random walk [21] is performed with the initial distance obtained from the look-up operation. Recall that for our parameter set, we use standard deviation $\sigma = 3.2$ (or $s = 8.02$) and sample upto a tail-bound of 6σ.

The Cycle Count for the encryption and decryption operations can be minimized in the following way. During the encryption operation, first the three error polynomials e_1, e_2 and e_3 are generated by invoking the discrete Gaussian sampler $3n$ times. Next the encoded message $\bar{m}$ is added to e_3 and then three consecutive forward NTT operations are performed on e_1, e_2 and $(e_3 + \bar{m})$. Finally the ciphertext $\tilde{c_1}$, $\tilde{c_2}$ is obtained using two point-wise multiplications followed by two polynomial additions and two rearrangement operations. The decryption operation requires one point-wise multiplication, one polynomial addition and finally one inverse NTT operation.

For $(n, \sigma) = (512, 3.2)$, the sampling operation during the encryption uses the look-up table on average 1500 times requiring 1500 cycles. An additional 173 cycles are spent on average for the remaining 36 sampling operations using bit-scanning. The polynomial addition and point-wise multiplication operations require n cycles each with a small overhead. The consecutive processing of I forward or inverse NTTs can share a fixed computation cost fc and requires in total $fc + I \times \frac{n}{2}\log(n)$ cycles. For inverse NTT, the fixed cost fc_{inv} is larger than the fixed cost fc_{fwd} of forward NTT due to the final scaling operation by ω_{2n}/N (Section 2.2). The rearrangement of polynomial coefficients after an NTT operation requires less than n cycles. From the above cycle counts for each primitive operations, we see that the encryption and decryption operations require total $fc_{fwd} + \frac{3}{2}n\log(n) + 10n$ and $fc_{inv} + \frac{n}{2}\log(n) + 2n$ cycles respectively along with additional overhead. Our hardware architecture for $n = 512$ has the fixed computation costs $fc_{fwd} = 1143$ and $fc_{inv} = 1959$ cycles.

7 Experimental Results

We have implemented the proposed ring-LWE cryptosystem on the Xilinx Virtex 6 FPGA for dimension 512, corresponding to a high security level. The area and performance results for the overall architecture are obtained from the Xilinx ISE12.2 tool after place and route analysis and are shown in Table 1. In the table we also compare our results with other reported hardware implementations of the ring-LWE encryption scheme. Our implementation is both fast and small due to its computational optimization and resource efficient design style. The discrete Gaussian sampler consumes only 93 LUTs and has a delay of $2.73ns$. Such a small delay makes the sampler suitable for integration in the pipelined ring-LWE processor. We use nine parallel true random bit generators [6, 4] to generate the random bits for the sampler. The set of true random bit generators consume 378 LUTs and 9 FFs.

The first hardware implementation of the ring-LWE encryption scheme in [7] uses a heavily parallel architecture to minimize the number of clock cycles for the NTT computation. Due to the many parallel computational blocks, the architecture is very large (0.29 million LUTs and 0.14 million FFs for $n = 256$) and does not even fit on the largest FPGA of the Virtex 6 family.

Implementation	Parameters	Device	LUTs	FFs	Freq	DSP	BRAMs	Cycles/Time	
	(n, q, s)				(MHz)		(18K)	Encryption	Decryption
Our	(512,1051649,8.02)	LX75T	1879	1142	250	1	3	13287/53.1 μs	5320/21.3 μs
Pöppelmann[17]	(512,12289,12.18)	LX75T	5595	4760	251	1	14	13769/54.8 μs	8883/35.4 μs

Table 1. Performance of the ring-LWE Cryptoprocessors on Virtex 6 FPGAs

Performance results such as cycle count and frequency are not reported in their paper. The architecture uses a Gaussian distributed array (indexed by an LFSR) for sampling of the error coefficients up to a tail-bound of $2s$.

The implementation in [17] is small and fast due to its resource-efficient design style. A high operating frequency is achieved using pipelines in the architecture. However the paper does not provide details of the actual pipeline strategy. The architecture uses a ROM that keeps all the twiddle factors required during the NTT operation. This approach reduces the fixed computation cost (fc) but consumes block RAM slices in FPGAs. Additionally, the parallel RAM blocks in the NTT processor result in a larger memory requirement compared to our design. The discrete Gaussian sampler uses the inversion sampling method and thus requires many random bits to output a sample value. To supply many random bits, an AES core is used and hence the overall area and delay are larger compared to our sampler architecture.

Although our architecture does not use a dedicated ROM for storing the twiddle factors, it still achieves smaller cycle count and faster computation time compared to [17]. The encryption scheme in [17] computes one forward and two inverse NTTs; while our encryption scheme computes only forward NTTs and hence does not require the $4n$ cycles for the scaling operation. Additionally our negative convolution method is free from the precomputation that takes n cycles in [17]. Hence we save $5n$ cycles in total during the NTT operations in an encryption operation. Since the fixed computation cost fc_{fwd} is smaller than $5n$, we gain in cycle count for the encryption operation. The decryption operation in our case is trivially faster than [17] as only one NTT is performed. We also reduce the area and memory requirement significantly compared to [7, 17]. This reduction is achieved by our resource-efficient design decisions such as 1) absence of a dedicated ROM for the twiddle factors, 2) an efficient RAM access and storage scheme, 3) use of one modular multiplier, 4) use of a smaller and faster (low-delay) discrete Gaussian sampler, and finally 5) the resource sharing between different computations.

8 Conclusion

This paper proposed several optimizations for implementing a ring-LWE based encryption system. The first set of optimizations improved the NTT by reducing the computation cost of the twiddle factors, avoiding the pre-computation during the forward NTT, and deriving an efficient memory access scheme that increases the utilization of the arithmetic components and the memory blocks. A further optimization reduced the number of NTTs required in the encryption scheme from five to four. The proposed optimizations are implemented in an efficient cryptoprocessor for the ring-LWE encryption system that not only is three times smaller in area and memory than any other reported implementations, but also even faster. These features make the architecture suitable for resource constrained platforms. Furthermore, the paper investigated architectural acceleration to meet the high speed requirement for real-time applications and proposes an optimal pipeline strategy that results in a very fast computation time whilst using minimum area and memory. Although the paper focuses on implementation of the ring-LWE based encryption system, we finally remark that the proposed optimization techniques for the NTT computation are applicable for other lattice based cryptosystems where similar polynomial multiplications are performed.

Acknowledgment

This work was supported in part by the Research Council KU Leuven: TENSE (GOA/11/007), by iMinds, by the Flemish Government, FWO G.0550.12N, by the Hercules Foundation AKUL/11/19 and by the Erasmus Mundus Scholarship. We are thankful to Bohan Yang and Vladimir Rozic for useful technical discussions related to memory and random number generation.

References

1. A. Aysu, C. Patterson, and P. Schaumont. Low-cost and Area-efficient FPGA Implementations of Lattice-based Cryptography. In *HOST*, pages 81–86. IEEE, 2013.
2. D. Bernstein. Fast Multiplication and its Applications. *Algorithmic Number Theory*, 44:325–384, 2008.
3. T. Cormen, C. Leiserson, and R. Rivest. *Introduction To Algorithms.* http://staff.ustc.edu.cn/∼csli/graduate/algorithms/book6/toc.htm.
4. M. Dichtl and J. D. Golic. High-Speed True Random Number Generation with Logic Gates Only. In *Cryptographic Hardware and Embedded Systems - CHES 2007*, volume 4727 of *LNCS*, pages 45–62. Springer Berlin, 2007.
5. T. Frederiksen. A Practical Implementation of Regev's LWE-based Cryptosystem. In *http://daimi.au.dk/ jot2re/lwe/resources/*, 2010.
6. J. D. Golic. New Methods for Digital Generation and Postprocessing of Random Data. *IEEE Transactions on Computers*, 55(10):1217–1229, 2006.
7. N. Göttert, T. Feller, M. Schneider, J. Buchmann, and S. Huss. On the Design of Hardware Building Blocks for Modern Lattice-Based Encryption Schemes. In *Cryptographic Hardware and Embedded Systems CHES 2012*, volume 7428 of *LNCS*, pages 512–529. Springer Berlin, 2012.
8. D. E. Knuth and A. C. Yao. The Complexity of Non-Uniform Random Number Generation. *Algorithms and Complexity*, pages 357–428, 1976.
9. T. Lepoint and M. Naehrig. A comparison of the homomorphic encryption schemes FV and YASHE. *IACR Cryptology ePrint Archive*, 2014:62, 2014.
10. R. Lindner and C. Peikert. Better Key Sizes (and Attacks) for LWE-based Encryption. *CT-RSA 2011*, pages 319–339, 2011.
11. V. Lyubashevsky, C. Peikert, and O. Regev. On Ideal Lattices and Learning with Errors over Rings. In *Advances in Cryptology EUROCRYPT 2010*, volume 6110 of *Lecture Notes in Computer Science*, pages 1–23. Springer Berlin Heidelberg, 2010.
12. Y. Ma and L. Wanhammar. A Hardware Efficient Control of Memory Addressing for High-performance FFT Processors. *Signal Processing, IEEE Transactions on*, 48(3):917–921, Mar 2000.
13. D. Micciancio. *Lattices in Cryptography and Cryptanalysis.* 2002.
14. P. Q. Nguyen and J. Stern. The Two Faces of Lattices in Cryptology. In *Cryptography and Lattices, International Conference (CaLC 2001)*, volume 2146 of *LNCS*, pages 146–180. Springer-Verlag, Berlin, 2001.
15. J. Pollard. The Fast Fourier Transform in a Finite Field. *Mathematics of Computation*, 25:365374, 1971.
16. T. Pöppelmann and T. Güneysu. Towards Efficient Arithmetic for Lattice-Based Cryptography on Reconfigurable Hardware. In A. Hevia and G. Neven, editors, *Progress in Cryptology LATINCRYPT 2012*, volume 7533 of *LNCS*, pages 139–158. Springer Berlin, 2012.
17. T. Pöppelmann and T. Güneysu. Towards Practical Lattice-Based Public-Key Encryption on Reconfigurable Hardware. In *Selected Areas in Cryptography SAC 2013*, LNCS. Springer-Verlag, Burnaby, Canada, 2013, Preprint.
18. O. Regev. Quantum Computation and Lattice Problems. *SIAM J. Comput.*, 33(3):738–760, Mar. 2004.
19. O. Regev. On Lattices, Learning with Errors, Random Linear Codes, and Cryptography. In *Proceedings of the thirty-seventh annual ACM symposium on Theory of computing*, STOC '05, pages 84–93, New York, NY, USA, 2005. ACM.
20. O. Regev. Lattice-Based Cryptography. In C. Dwork, editor, *Advances in Cryptology - CRYPTO 2006*, volume 4117 of *LNCS*, pages 131–141. Springer Berlin, 2006.

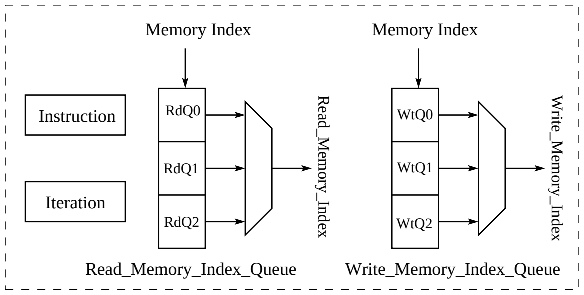

Fig. 4. Instruction Execution Hardware

21. S. S. Roy, F. Vercauteren, and I. Verbauwhede. High precision discrete gaussian sampling on fpgas. *proceedings of SAC*, 2013.
22. J. van de Pol and N. P. Smart. Estimating key sizes for high dimensional lattice-based systems. In M. Stam, editor, *IMA Int. Conf.*, volume 8308 of *Lecture Notes in Computer Science*, pages 290–303. Springer, 2013.

Appendix A

Table 2 shows the memory contents during the execution of Algorithm 2 for $n = 16$. The column-heading represents (m, j, k) during the iterations. The end loop in line 19 of Algorithm 2 for $m = 16$ performs no swap and is shown in the table using $\star$ symbol.

Address	Initial	(2,0,0)	(2,0,6)	(4,0,0)	(4,0,4)	(4,1,4)	(8,3,0)	(16,7,0)$\star$
0	A_1 A_0	A_2 A_0	A_2 A_0	A_4 A_0	A_4 A_0	A_4 A_0	A_8 A_0	A_8 A_0
1	A_3 A_2	A_3 A_1	A_3 A_1			A_5 A_1	A_9 A_1	A_9 A_1
2	A_5 A_4		A_6 A_4	A_6 A_2	A_6 A_2	A_6 A_2	A_{10} A_2	A_{10} A_2
3	A_7 A_6		A_7 A_5			A_7 A_3	A_{11} A_3	A_{11} A_3
4	A_9 A_8		A_{10} A_8		A_{12} A_8	A_{12} A_8	A_{12} A_4	A_{12} A_4
5	A_{11} A_{10}		A_{11} A_9			A_{13} A_9	A_{13} A_5	A_{13} A_5
6	A_{13} A_{12}		A_{14} A_{12}		A_{14} A_{10}	A_{14} A_{10}	A_{14} A_6	A_{14} A_6
7	A_{15} A_{14}		A_{15} A_{13}			A_{15} A_{11}	A_{15} A_7	A_{15} A_7

Table 2. Memory content during the steps in a 16-point NTT

Appendix B

Our ring-LWE cryptoprocessor has one instruction-register, one iteration-register, one read-memory-index-queue and one write-memory-index-queue (Figure 4). The read and write memory-index-queues are loaded with the memory indexes. Since our ring-LWE cryptoprocessor has six memory blocks $M0$ to $M5$, the indexes are in the range 0 to 5. The instruction is stored in the *Instruction* register and the number (I) of consecutive NTT operations is kept in the *Iteration* register. The following instructions are supported by the processor.

1. LOAD : A memory block indexed by $WtQ0$ is loaded with n coefficients. Since two coefficients are processed in a cycle, the instruction takes $n/2 + \epsilon$ cycles.

2. ENCODE-LOAD : A memory block indexed by $WtQ0$ is loaded with an encoded message. The input message bits are first encoded using the encoder and then loaded in the memory block as proper coefficient-pairs. This instruction requires $n + \epsilon$ cycles.
3. GAUSSIAN-LOAD : A memory block indexed by $WtQ0$ is loaded with n samples. The cycle count for this operation depends on the standard deviation and n.
4. FNTT/INTT : Is used to perform inplace forward or inverse NTT. The number of consecutive NTTs is stored in the iteration-register and the indexes of the memory blocks are kept in the read-memory-index-queue
5. ADD/CMULT : Two memory blocks indexed by $RdQ0$ and $RdQ1$ are added or coefficient-wise multiplied. The result is stored in the memory block indexed by $WtQ0$. These two instructions require $n + \epsilon$ cycles.
6. REARRANGE : Performs rearrangement of coefficient pairs in a memory block indexed by $RdQ0$. This instruction requires less than n cycles.
7. READ : The contents of a memory block indexed by $RdQ0$ are read. This instruction requires $n/2 + \epsilon$ cycles.