FPGA-Based Key Generator for Post-Quantum Key Encapsulation Based on Quasi-Cyclic Codes

No Author Given

No Institute Given

Abstract. In this paper, we present a constant-time FPGA-based key generator for a quantum-safe key encapsulation mechanism based on MDPC codes called BIKE, which is among round-2 candidates in the NIST PQC standardization process. Existing hardware implementations of MDPC code-based systems focus on the encryption/decryption (key encapsulation/decapsulation) operations while few of them take into account the key generation operation which is particularly crucial for KEM-based applications. The hardware architecture for the key generator proposed in this work aims at maximizing the metric of timing efficiency while maintaining a low resource consumption. Parallelized polynomial multiplication and squaring algorithms are proposed and implemented to achieve this goal. Our experimental results show that our key generator design requires 9.5×10^5 cycles (0.597 ms), 2.1×10^6 cycles (14.334 ms) and 9.8×10^5 cycles (0.492 ms) respectively to generate a key pair for BIKE-1/2/3. The proposed key generator is very area-efficient, it uses only 544/1272/423 slices for BIKE-1/2/3 on a small-sized low-end Xilinx Spartan-7 FPGA.

Keywords: Post-Quantum Cryptography, Key Encapsulation Mechanism, Key Generator, QC-MDPC Code, FPGA Implementation

1 Introduction

Key encapsulation mechanisms (KEMs) are commonly used encryption techniques to secure the transmission of symmetric cryptographic keys by use of public-key cryptographic algorithms. Therefore, efficient and secure KEM designs have always been an important topic in the cryptographic community. In the past few years, NIST has started the PQC standardization process soliciting the designs of efficient and secure post-quantum KEMs [8] since the construction of a commercial quantum computer emerges as an urgent threat to the commonly used cryptographic primitives nowadays. These primitives rely on the hardness of the integer factorization problem or discrete logarithm problems, e.g., the Diffie-Hellman key exchange, the Rivest-Shamir-Adleman (RSA) and Elliptic Curve Cryptography (ECC). However, in 1994 Shor's algorithm [35] was proposed which can be deployed on a quantum computer and solve both the integer factorization problem and the discrete logarithm problem in polynomial time, and therefore will render the security of the modern cryptographic primitives.

Code-based cryptosystems, which build their security on the hardness of decoding general linear codes, are among the most promising quantum-secure candidates for which no known polynomial time attacks exist by use of a quantum computer. McEliece proposed the first code-based cryptosystem in 1978 [28], which uses binary Goppa codes [18] in the underlying coding system. However, Goppa code-based schemes yield large public keys which limit the deployment of these systems in resource-constrained scenarios. Niederreiter proposed in 1986 a variant of the McEliece cryptosystem [32] which introduced a trick to reduce the size of the public key, namely the Niederreiter cryptosystem. It has been proven that the Niederreiter cryptosystem and the McEliece cryptosystem achieve the same security level [25] when the same family of codes is used.

Active research has been focused on replacing Goppa codes with other families of structured codes for reducing the key size. Nevertheless, many of these attempts lead to the compromise of the the system security. For example, many McEliece variants based on low-density parity-check (LDPC) codes [31], quasi-cyclic low-density parity-check (QC-LDPC) codes [34], quasi-dyadic (QD) codes [29], convolutional codes [27], generalized Reed-Solomon (GRS) codes [4], and rank-metric codes [26,15] have been successfully broken. However, a few variants exploiting the quasi-cyclic form of the codes [3,5,30] have been shown to counteract all the existing cryptanalysis attacks while maintaining small-sized keys. A variety of KEMs built on the Niederreiter cryptosystem and LDPC/MDPC codes such as LEDAkem [2], CAKE [6] and BIKE [1] are proposed and analyzed in the literature. Among these proposals, BIKE is currently among the round-2 candidates for the NIST PQC standardization process. Recently, a new statistical attack, called reaction attack [14,19] has been devised to recover the key by exploiting the information leaked from decoding failures in QC-LDPC and QC-MDPC code-based systems. This attack is further enhanced in [33] where the error pattern chaining method is introduced to generate multiple undecodable error patterns from an initial error pattern, thus improving the performance of such reaction attack in the CPA case. However, the reaction attack only works for CPA systems and long-term keys, therefore, the deployment of a CCA2-secure conversion or ephemeral keys in BIKE can resist the attack.

Contributions. Most of the existing work on hardware implementations of MDPC code-based cryptosystems focus on the encryption and decrption operations. However, KEMs are gradually getting more commonly used in ephemeral key encapsulation scenarios in which cases the key generation is used as often as the encapsulation and decapsulation operations and therefore should also be accelerated by dedicated hardware modules. Therefore, in this work we present the first constant-time, efficient and compact FPGA-based key generator for a promising MDPC code-based post-quantum KEM — BIKE. Our contributions include:

– The proposal of new polynomial arithmetic algorithms (squaring, generic polynomial multiplication and sparse polynomial multiplication) over ring $\mathbb{F}_2[x]/\langle x^r+1\rangle$.

– Constant-time and efficient hardware architectures for the polynomial arithmetics in $\mathbb{F}_2[x]/\langle x^r+1\rangle$ including polynomial inversion, polynomimal squaring, polynomial multiplications and polynomial generation.
– Novel and lightweight hardware architectures for the key generator of three BIKE variants of security ranging between 128-bit and 256-bit.

2 Preliminaries

In the following, we describe the basic notations and definitions in QC-MDPC codes which closely follow the descriptions in [1]. Quasi-cyclic (QC) codes can be defined over either matrices or polynomial rings. We use both the matrix view and the polynomial view to describe the relevant aspects of the BIKE scheme.

2.1 General Definitions

Table 1 shows the notations used in the following discussions.

Polynomial View:	
$\mathbb{F}_2$	Finite field of 2 elements
$\mathcal{R}$	The cyclic polynomial ring $\mathbb{F}_2[x]/\langle x^r+1\rangle$
$\lvert x\rvert$	The Hamming weight of a binary polynomial x
$u \xleftarrow{\$} U$	Variable u is sampled uniformly at random from set U
Matrix View:	
$\mathcal{V}$	The vector space of dimension n over $\mathbb{F}_2$
$\mathbf{m}$	The vector over $\mathbb{F}_2$
$\mathbf{G}$	The matrix over $\mathbb{F}_2$

Table 1: Notations

The basic structure in QC codes is the circulant matrix, which is defined as follows:

Definition 1. *(Circulant Matrix) Let $\mathbf{x} = (x_0, \dots, x_{n-1}) \in \mathbb{F}_2^n$. The circulant matrix induced by $\mathbf{x}$ is defined and denoted as follows:*

$$\mathbf{rot}(\mathbf{x}) = \begin{bmatrix} x_0 & x_{n-1} & \cdots & x_1 \\ x_1 & x_0 & \cdots & x_2 \\ \vdots & \vdots & \ddots & \vdots \\ x_{n-1} & x_{n-2} & \cdots & x_0 \end{bmatrix}$$

Therefore, the product of any two polynomials x and y in $\mathcal{R}$ can be viewed as a vector-matrix (or matrix-vector) product using the operator $\mathbf{rot}(\cdot)$ as:

$$x \cdot y = \mathbf{x} \cdot \mathbf{rot}(\mathbf{y})^T = \mathbf{y} \cdot \mathbf{rot}(\mathbf{x})^T = y \cdot x$$

MDPC codes are a special variant of linear codes and are defined as follows:

Definition 2. *(Linear Code) A Binary Linear Code* $\mathcal{C}$ *of length* n *and dimension* k *(denoted as* $[n, k]$*) is a subspace of* $\mathcal{V}$ *of dimension* k*. Elements of* $\mathcal{C}$ *are referred to as codewords.*

Definition 3. *(Generator and Parity-Check Matrix)* $\mathbf{G} \in \mathbb{F}^{k\times n}$ *is a Generator Matrix for the* $[n, k]$ *code* $\mathcal{C}$ *iff*

$$\mathcal{C} = \{\mathbf{mG}|\mathbf{m} \in \mathbb{F}_2^k\}$$

$\mathbf{H} \in \mathbb{F}^{(n-k)\times n}$ *is called a Parity-Check Matrix of* $\mathcal{C}$ *iff*

$$\mathcal{C} = \{\mathbf{c} \in \mathbb{F}_2^n|\mathbf{Hc}^T = 0\}$$

A linear code can be quasi-cyclic according to the following definition:

Definition 4. *(Quasi-Cyclic Codes) A binary quasi-cyclic (QC) code of index* n_0 *and order* r *is a linear code which admits as generator matrix a block-circulant matrix of order* r *and index* n_0*. A* (n_0, k_0)*-QC code is a quasi-cyclic code of index* n_0*, length* $n_0 r$ *and dimension* $k_0 r$

For instance, the generator matrix of a $(3, 1)$-QC code is formed by three circulant blocks $\mathbf{G_i}$: $\mathbf{G} = [\mathbf{G_0}|\mathbf{G_1}|\mathbf{G_2}]$.

In the polynomial view, each block $\mathbf{G_i}$ can be represented (one-to-one mapping) as a polynomial g_i over $\mathcal{R}$ and thus the generator matrix $\mathbf{G}$ can be viewed as the polynomial matrix $[g_0|g_1|g_2]$ over $\mathcal{R}$. In all aspects, any codeword $\mathbf{c} = \mathbf{m}\cdot\mathbf{G}$ over $\mathbb{F}^n$ can be represented in its polynomial form as $[mg_0|mg_1|mg_2]$ over $\mathcal{R}$. All the above notations are collected in Table 1.

2.2 BIKE

BIKE was proposed by Aragon *et. al* in 2017 [1], which is a novel code-based key encapsulation mechanism based on QC-sMDPC codes. The use of a QC-MDPC code $C(n_0, k_0)$ in BIKE brings a big key size reduction compared to unstructured codes. BIKE uses the double-circulant structure for the parity check matrix where two smaller cyclic matrices are included. Nine instances of BIKE (BIKE-1, BIKE-2, BIKE-3) are proposed in their specification. BIKE-1 is a variant which provides fast key generation by using a variation of the McEliece cryptosystem. A preliminary version of BIKE-1 first appeared in [6]. BIKE-2 follows Niederreiter's framework by use of a systematic parity check matrix which can yield a very compact formulation. The BIKE-3 variant follows the work of Ouroboros [11], featuring fast, inversion-less key generation operations. The main difference in BIKE-3 is that the decapsulation invokes the decoding algorithm on a noisy syndrome. This also suggests that BIKE-3 is fundamentally distinct from BIKE-1 and BIKE-2 in terms of security aspects. Parameters for these BIKE instances are shown in Table 2, which covers the 5 security categories suggested by the NIST PQC standardization process. QC construction of the codes brings a significant reduction in memory storage required for public keys, since only

Table 2: Parameters for BIKE, referenced from [1]

Category		n	r	w	t	SL	DFR
1	BIKE-[1,2]	20,326	10,163	142	134	128	$\leq 10^{-7}$
	BIKE-3	22,054	11,027	134	154		
2-3	BIKE-[1,2]	39,706	19,853	206	199	192	$\leq 10^{-7}$
	BIKE-3	43,366	21,683	198	226		
4-5	BIKE-[1,2]	65,498	32,749	274	264	256	$\leq 10^{-7}$
	BIKE-3	72,262	36,131	266	300		

the first row/column of each circulant block needs to be stored and the remaining part can be reconstructed by cyclic rotations of the first row/column, thus largely minimizing the public key size.

The key generation algorithms of BIKE-1/2/3 are described in the following text. Algorithm 1 describes the key generation algorithm of BIKE-1. The private keys comprise of two "sparse" polynomials h_0 and h_1 where $|h_0|$ and $|h_1|$ is restricted to $w/2$. The public keys (f_0, f_1) are exposed for public use and therefore must be scrambled to hide the structure of h_0 and h_1. This is done by multiplying h_0 and h_1 by a random dense polynomial g to get gh_0 and gh_1.

Input: λ, the target quantum security level.
Output: the sparse private key **PK** $= (h_0, h_1)$ and the dense public key **SK** $= (f_0, f_1)$
Given λ, set the parameters r, w as described above
Generate $h_0, h_1 \leftarrow \mathcal{R}$ both of (odd) weight $|h_0| = |h_1| = w/2$
Generate $g \leftarrow \mathcal{R}$ of odd weight (so $|g| \approx r/2$)
Compute $(f_0, f_1) \leftarrow (gh_1, gh_0)$
return *PK and SK*

Algorithm 1: BIKE-1 key generation algorithm in polynomial view [1]

Input: λ, the target quantum security level.
Output: the sparse private key **PK** $= (h_0, h_1)$ and the dense public key **SK** $= h$
Given λ, set the parameters r, w as described above
Generate $h_0, h_1 \leftarrow \mathcal{R}$ both of (odd) weight $|h_0| = |h_1| = w/2$
Compute $h \leftarrow h_1 h_0^{-1}$
return *PK and SK*

Algorithm 2: BIKE-2 key generation algorithm in polynomial view [1]

Algorithm 2 describes the BIKE-2 key generation algorithm. Note that BIKE-2 does not generate any random polynomial to hide the secret keys h_0 and h_1. Instead, it computes $h = h_0^{-1} h_1$ as the public key. Therefore, the public key

Input: λ, the target quantum security level.
Output: the sparse private key $\mathbf{PK} = (h_0, h_1)$ and the dense public key $\mathbf{SK} = (f_0, f_1)$

Given λ, set the parameters r, w as described above
Generate $h_0, h_1 \leftarrow \mathcal{R}$ both of (odd) weight $|h_0| = |h_1| = w/2$
Generate $g \leftarrow \mathcal{R}$ of odd weight (so $|g| \approx r/2$)
Compute $(f_0, f_1) \leftarrow (h_1 + gh_0, g)$
return *PK and SK*

Algorithm 3: BIKE-3 key generation algorithm in polynomial view [1]

size is halved, however, the computational cost increases due to the required polynomial inversion operation.

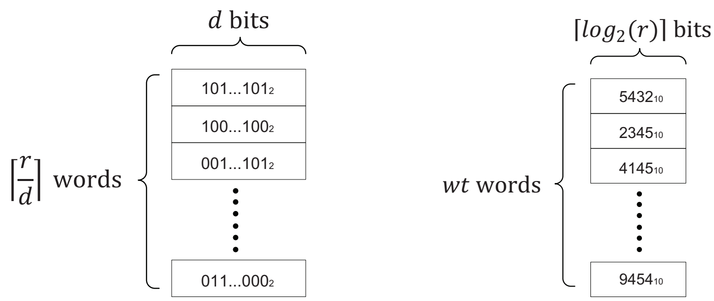
Algorithm 3 describes the key generation algorithm of BIKE-3, which is an improved version of BIKE-1. BIKE-3 uses only one multiplication and one addition to compute the first half of public key $f_0 = h_1 + gh_0$. The second half f_1 equals to the random dense polynomial g.

3 Polynomial Arithmetics

In this section, we introduce the basic polynomial arithmetics in $\mathbb{F}_2[x]/(x^r + 1)$ for building the BIKE key generation architecture. Algorithms and implementations for polynomial squaring and multiplications, polynomial inversion, and generation of sparse/dense polynomials with prescribed Hamming weight are presented in the following sub-sections.

Polynomial Representations. In this paper, two formats are used to represent polynomials in the quotient ring $\mathbb{F}_2[x]/(x^r + 1)$. One is the dense format which stores all the coefficients of a given polynomial $f = f_0 + f_1 x + \ldots + f_{r-1}x^{r-1}$. This format is used when the polynomial is dense, e.g., the public keys (f_0, f_1) are dense polynomials and therefore are described in this format. In our design, the coefficients of the polynomials are stored in block RAMs, in which each address keeps multiple coefficients as a word. Small endian notation is used here, *i.e.*, f_0 is first stored, then f_1, and finally f_{r-1}. We define such data structure for a polynomial f as $\overline{\mathbf{f}}$ where $\overline{\mathbf{f}[i]}$ denotes the word (d bits in total) stored in the i-th address as: $(f_{id}, f_{id+1}, \ldots, f_{id+d-1})_2$. By using this format, in total $\lceil r/d \rceil$ words are needed to represent a polynomial over $\mathbb{F}_2[x]/(x^r + 1)$. For the last word $\overline{\mathbf{f}}[\lceil r/d \rceil - 1]$, we append a few zeros to fill it as a complete word: $(f_{(\lceil r/d \rceil - 1)d}, \ldots, f_r, \ldots, 0)_2$. Figure 1a illustrates the data structure of $\overline{\mathbf{f}}$.

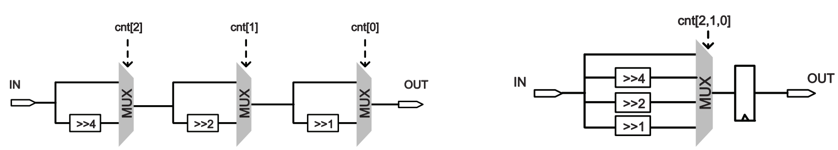
The other format is the sparse format which is used when the polynomial is sparse, e.g., the secret keys (h_0, h_1). This format only saves the bit positions of non-zero coefficients in a polynomial f. We define this sparse form for a polynomial f as $\hat{\mathbf{f}}$ where $\mathbf{f}\hat{[i]}$ denotes the non-zero bit position (belongs to the set $\{i | f_i = 1\}$, each item of this set is $log_2(r)$ bits) stored in the i-th address. This format is memory-efficient especially when the weight of f is small: It takes



(a) Polynomial represented in dense Format

(b) Polynomial represented in sparse format

Fig. 1: Polynomial in $\mathbb{F}_2[x]/(x^r + 1)$ represented in different formats



(a) Barrel shifter constructed by combinational logic

(b) Barrel shifter constructed by sequential logic

Fig. 2: Barrel shifter used in our key generator to ensure constant execution of multiplication and squaring

$|f|$ words and each word is of $log_2(r)$ bits to represent a given polynomial f. Figure 1b illustrates the data structure of $\hat{\mathbf{f}}$.

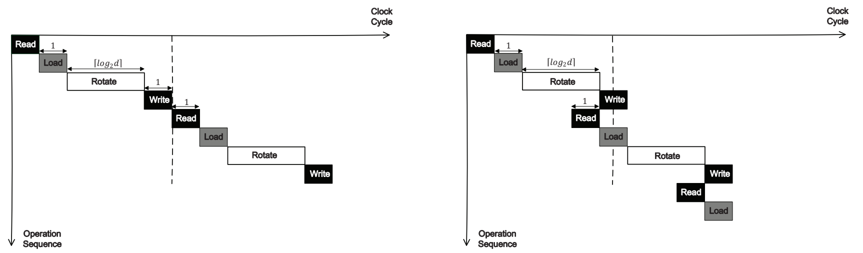
3.1 Polynomial Squaring

In this paper, the general squaring form $(a(x))^{2^n}$ is considered. The following theorem states how to calculate the squaring of a polynomial over $\mathcal{R}$.

Theorem 1. *For squaring $(a(x))^{2^n} = \sum \widetilde{a_i} x^i$ of a polynomial $a(x) = \sum a_i x^i \in \mathcal{R}$, each coefficient $\widetilde{a_i}$ is updated by a_j in $[a_{r-1}, a_{r-2}, \cdots, a_0]$ as:*

$$\widetilde{a_i} = a_{i2^{-n} \bmod r}$$

Note that the data is typically processed word by word on hardware, *i.e.*, only one or two words (in our design, one word is d-bits) per time are fetched from memory and loaded to core function for further calculation. However, sometimes we want to extract the precise bit in the word. Squaring is such an example where the result is indeed computed bit by bit. To preserve the constant-time



(a) Primitive operation pipelining for continuous square

(b) Optimised operation pipelining for continuous square

Fig. 3: Timing diagram used in continuous square

feature during executions while avoiding the performance degradation in timing, a Barrel shifter is used to rotate the desired bit within $\lceil log_2 d \rceil$ cycles. Barrel shifter can be built by pure combinational logic (shown in Fig. 2a) or sequential logic (shown in Fig. 2b). The sequential logic structure optimizes the critical path delay and is used in our design. By doing this, the computational steps for squaring $(a(x))^{2^n}$ are constant and independent of the value of n.

Figure 3a depicts the pipeline stages for computing the polynomial squaring operations continuously. The basic computation pattern is: First $a_{i2^{-n}}$ is read out from the memory, then it gets loaded to the Barrel shifter and rotated to the correct bit position. Finally the result is written back to the memory. In general, such pattern repeats r times and hence the cycle count for computing the squaring operation continuously is: $r(\lceil log_2 d \rceil + 3)$.

We further optimized the above pipeline based on the observation that read and rotation operations can be parallelized, and load and write can be parallelized as well, the pipeline stages are shown in 3a. After this optimization, for computing the squaring operation continuously, we achieve a cycle count of: $r(\lceil log_2 d \rceil + 1) + 2$.

3.2 Generic Polynomial Multiplication

A generic polynomial multiplication involves two random dense polynomials. For two dense polynomials $a(x)$ and $b(x)$, their multiplication is represented as:

$$a(x) \cdot b(x) = (a_{r-1}x^{r-1} + \cdots + a_1 x + a_0) \cdot (b_{r-1}x^{r-1} + \cdots + b_1 x + b_0) \quad (1)$$

$$= \widetilde{c_{r-1}} x^{r-1} + \widetilde{c_{r-2}} x^{r-2} + \cdots + \widetilde{c_1} x + \widetilde{c_0} \quad (2)$$

For software implementations [10,12], the best practice is to apply a carry-less Karatsuba algorithm to improve the performance for sufficiently high-degree polynomial multiplications, in particular those used for BIKE and other QC-MDPC code related cryptosystems. Carry-less Karatsuba algorithm is efficient on modern general-purpose processors equipped with “carry-less multiplication” instruction PCLMULQDQ. This instruction computes the product of two binary polynomials of degree 63 and thus an appropriate software flow can use PCLMULQDQ to perform the carry-less Karatsuba algorithm for polynomials with any degree. However, FPGAs do not have similar embedded instructions and moreover, a relatively complex control flow of Karatsuba algorithm will degrade the timing performance on FPGAs. These issues drive us to find out a better solution for polynomial multiplications on FPGAs.

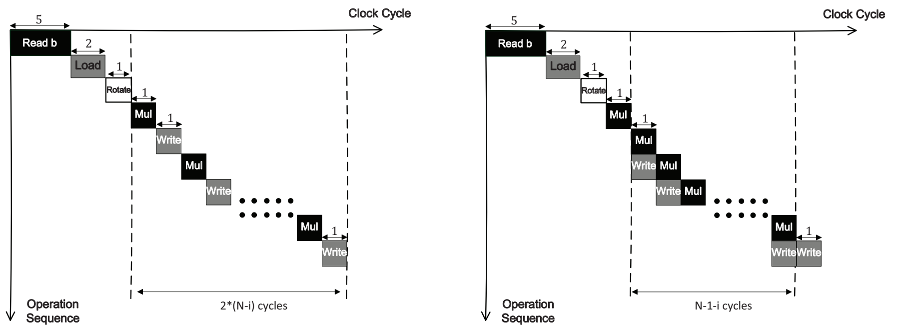
Input: dense polynomials $a(x), b(x) \in \mathbb{F}_2[x]/(x^r + 1)$.
Output: $a(x) \cdot b(x) \in \mathbb{F}_2[x]/(x^r + 1)$

Reformulate $a(x)$ to block matrix view as $A[0], A[1], A[2], \ldots, A[n-1]$ where $n = \lceil r/d \rceil$, d is the number of bits used in the row vector $A[i]$.
Compute partial blocks from the transformation matrix BT_{mul} associated with $b(x)$: $\mathrm{BT}_{mul}[0,0]$, $\mathrm{BT}_{mul}[0,1], \mathrm{BT}_{mul}[0,2], \ldots, \mathrm{BT}_{mul}[0,n-1]$ and $\mathrm{BT}_{mul}[1,0]$, $\mathrm{BT}_{mul}[2,0], \ldots, \mathrm{BT}_{mul}[n-1,0]$
The multiplication result is presented again in block matrix view $C[0], C[1], C[2], \ldots, C[n-1]$
/*Diagonal Computation*/
for $i \leftarrow 0$ ***to*** $n-1$ **do**
  $C[i] \leftarrow A[i] \cdot \mathrm{BT}_{mul}[0,0]$
/*Upper Triangle Computation*/
for $i \leftarrow 1$ ***to*** $n-1$ **do**
  for $j \leftarrow 0$ ***to*** $n-1-i$ **do**
    $C[i+j] \leftarrow C[i+j] + A[j] \cdot \mathrm{BT}_{mul}[0,i]$
/*Lower Triangle Computation*/
for $i \leftarrow 1$ ***to*** $n-1$ **do**
  for $j \leftarrow 0$ ***to*** $n-1-i$ **do**
    $C[j] \leftarrow C[j] + A[i+j] \cdot \mathrm{BT}_{mul}[i,0]$
return *the vector* $C = [C[0], C[1], \ldots, C[n-1]]$ *as the coefficients of* $c(x) = a(x)b(x)$

Algorithm 4: Proposed generic multiplication algorithm

A formal description of the proposed generic multiplication algorithm can be found in Algorithm 4. As we can see, this algorithm is computed in exactly n^2 steps with $n = \lceil r/d \rceil$, and therefore, for fixed BIKE parameters r and d, the computational time is also fixed. The core function in the generic polynomial multiplication is the multiplication of a d-bit vector $A[k]$ of $a(x)$ and a block matrix $\mathrm{BT}_{mul}[i,j]$ from BT_{mul}:

$$[a_j, a_{j+1}, \ldots, a_{j+d-1}] \cdot \begin{bmatrix} b_i & b_{i+1} & b_{i+2} & \cdots & b_{i+d-1} \\ b_{i-1} & b_i & b_{i+1} & \cdots & b_{i+d-2} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ b_{i-d+1} & b_{i-d+2} & b_{i-d+3} & \cdots & b_i \end{bmatrix}_{d \times d}$$



(a) Primitive operation pipelining for generic multiplication

(b) Optimised operation pipelining for generic multiplication

Fig. 4: Timing diagram used in generic multiplication

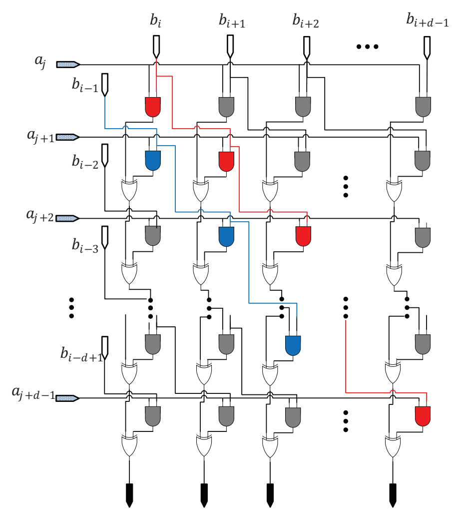
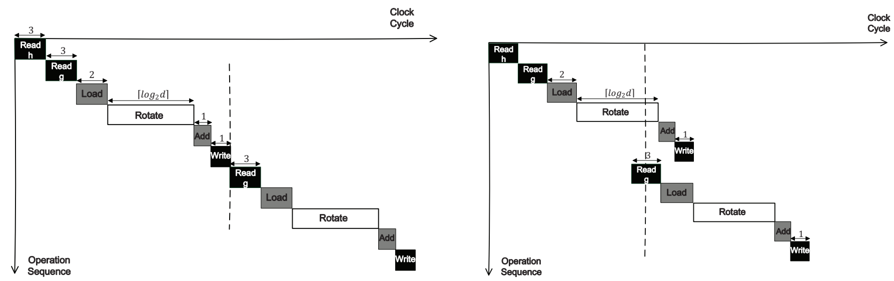


Fig. 5: Architectural overview of the core unit in the generic multiplication

We design a dedicate hardware core for the above multiplication which can be finished in one clock cycle, as depicted in Figure 5. With this module, the proposed generic multiplication algorithm can be scheduled as shown in Figure 4a. Read takes 5 cycles to balance the critical path for reading $B[i, j]$. A dual-port RAM is used such that the target vector $B[i, j]$ can be loaded within two cycles. Here a fixed rotation is performed to move $A[k]$ to the correct bit positions. The MUL module performs the core multiplication $A[k] \cdot B[i, j]$ and later writes the result back.



(a) Primitive operation pipelining for sparse polynomial multiplication

(b) Optimised operation pipelining for sparse polynomial multiplication

Fig. 6: Timing diagram used in sparse polynomial multiplication

An improved schedule is depicted in Figure 4b where Mul and Write perform simultaneously. To handle the diagonal computation, it takes $5+2+1+2\lceil r/d \rceil$ cycle count where $i=0$. $\lceil r/d \rceil - 1$ iterations are required for the upper triangle computation. For the $i = 1, 2, \ldots, d-1$-th iteration, it consumes $5+2+1+\lceil r/d \rceil - 1 - i + 1$ cycles. Therefore the entire upper triangle computation costs $(\lceil r/d \rceil^2 + 17\lceil r/d \rceil - 18)/2$ cycles. The same cycle count is needed for the lower triangle computation.

In general, the cycle count for the optimised generic polynomial multiplication is: $\lceil r/d \rceil^2 + 18\lceil r/d \rceil - 9$.

3.3 Sparse Polynomial Multiplication

In BIKE, a special form of multiplication called sparse polynomial multiplication, is frequently used. In sparse polynomial multiplication, one of the operands is a sparse polynomial whereas the other one is a dense polynomial. Such multiplications can be efficiently computed as follows: The sparse polynomial, *e.g*, $a(x)$ is first represented by an array of indices in $I = \{i | a_i = 1\}$. Then, the multiplication result can be obtained by extracting $i \in I$ rows of BT_{mul} and then accumulating them, *i.e.*, $\sum_{i \in I} x^i b(x)$ and the i-th row of the matrix BT_{mul} represents the coefficients of $x^i b(x)$. Formally, the coefficient vector of sparse polynomial multiplication can be calculated as: $\sum_{i \in I} [b_{-i}, b_{-i+1}, \cdots, b_{-i-1}]$.

Figure 6a depicts the basic pipeline schedule where the total cycle count for one complete sparse polynomial multiplication is: $|I| \cdot \lceil r/d \rceil \cdot (10 + \lceil log_2(d) \rceil)$.

The cycle count can be further improved if Read_g is computed in parallel with Rotate while Load is computed in parallel with Add+Write. This optimized schedule for pipelining is presented in Figure 6b and the optimized cycle count is: $|I|(\lceil r/d \rceil \cdot (2 + \lceil log_2(d) \rceil) + 8)$.

Table 3 present the cycle count of the two pipeline scheduling schemes. The optimized pipeline can improve the cycle count by about 50%.

Table 3: Cycle count comparison between unoptimized and optimized sparse polynomial multiplication. The pipelined architecture introduces about 50% reduction of cycle count for all BIKE instances

Category		d	r	$\lvert I\rvert = w/2$	unoptimized version	optimized version
128-bit	BIKE-[1,2]	64	10163	71	$180,624$	$90,880$
	BIKE-[3]	64	11027	67	$185,456$	$93,264$
192-bit	BIKE-[1,2]	64	19853	103	$512,528$	$257,088$
	BIKE-[3]	64	21683	99	$536,976$	$269,280$
256-bit	BIKE-[1,2]	64	32749	137	$1,122,304$	$562,248$
	BIKE-[3]	64	36131	133	$2,404,640$	$1,203,384$

Input: $r - 2 = (r_{q-1} \dots r_0)_2$ and $\alpha \in \mathbb{F}_2[x]/(x^r + 1)$.
Output: $\beta = \alpha^{-1}$

$\beta \leftarrow \alpha$
$t \leftarrow 1$
for $i \leftarrow q - 2$ ***to*** 0 **do**
    $\beta \leftarrow (\beta)^{2^t} \cdot \beta$
    $t \leftarrow 2t$
    if $r_i = 1$ **then**
        $\beta \leftarrow (\beta)^2 \cdot \alpha$
        $t \leftarrow t + 1$
$\beta \leftarrow \beta^2$
return β

Algorithm 5: Itoh-Tsujii Inversion Algorithm (ITA) [23]

3.4 Polynomial Inversion

Theorem 2. *Let a be an invertible polynomial in the quotient ring $\mathbb{F}_2[x]/(x^r+1)$ where r is a prime, then the inverse of polynomial a can be computed as $a^{-1} = (a^{2^{r-2}-1})^2$*

According to Theorem 1, we can compute the inverse of a polynomial over ring $\mathbb{F}_2[x]/(x^r+1)$ by exponentiation. To efficiently compute the exponentiation of the polynomial, we propose a method which is described as follows. First, we define a function $\beta_k(a) = a^{2^k-1}$. It is easy to see that $a^{-1} = (\beta_{r-2}(a))^2$. Therefore, this recursive formula holds: $\beta_{k+j}(a) = (\beta_k(a))^{2^j}\beta_j(a)$.

With the above recursion formula, we can generate $\beta_{r-2}(a)$ from $\beta_1(a) = a$ through an addition chain in constant steps, which is known as Itoh-Tsujii Inversion algorithm (ITA), as shown in Algorithm 5. More importantly, the building blocks for ITA are polynomial squaring and multiplication which can be highly optimized in our design. Another common approach for polynomial inversion is through the Extended Euclidean algorithm (EEA). In [16], EEA with constant flow is proposed which can be exploited for secure polynomial inversions. However, the steps in the proposed algorithm are data-dependent and thus is not

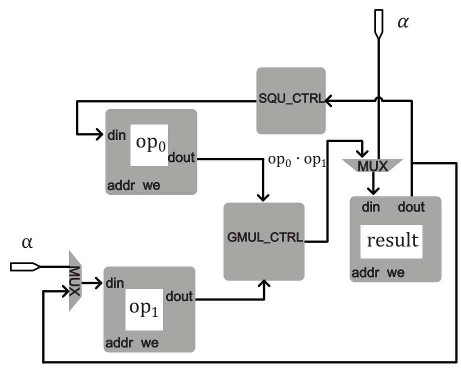


Fig. 7: Diagram of the polynomial inversion architecture

Input: PRNG seed, polynomial length r, data width d
Output: binary vector $\overline{\mathbf{f}}$ representing $f = f_0 + \ldots + f_{r-1}x^{r-1} \in \mathbb{F}_2[x]/(x^r + 1)$, with odd weight $|f| \approx r/2$

```
wt ← 0
for i ← 0 to ⌈r/d⌉ − 1 do
    f̄[i] ← truncate_d(SecureRand(seed))
    wt ← wt + |f̄[i]|
if wt is even then
    flip the first bit in f̄[i]
return f̄
```

Algorithm 6: Generation of a random polynomial with odd weight

suitable for parallel designs on hardware. An illustrative example for computing the polynomial inversion used in BIKE-2 is shown in Table 9 in the Appendix.

By use of ITA, the polynomial inversion operation can be performed in constant time. Figure 7 depicts the diagram of the ITA inverter. The input polynomial α is used to initialize the RAM result. Then the polynomial β stored in the RAM is raised to β^{2^t} by use of the SQU_CTRL module which is the module for polynomial squaring as described in Section 3.1. The result is then stored to RAM op0. The GMUL_CTRL module (Section 3.2) performs a generic polynomial multiplication (step 4,7, Algorithm 5). After the final squaring operations (step 10), the inverted polynomial is stored as the result in RAM op0.

3.5 Generation of Polynomials with Prescribed Weight

Strictly speaking, the execution time of the random polynomial generations proposed in Algorithms 6, 7 is non-constant. Algorithm 6 optionally performs a step if the Hamming weight of the polynomial is an even number; Algorithm 7 keeps

Input: PRNG seed, polynomial length r, data width d, polynomial weight $wt = w/2$

Output: integer vector $\hat{\mathbf{f}}$ contains non-zero bit-positions ($\{i|f_i \neq 0\}$ to represent $f = f_0 + \ldots + f_{r-1}x^{r-1} \in \mathbb{F}_2[x]/(x^r+1)$, with odd weight $|f| = wt$

1 **for** $i \leftarrow 0$ ***to*** $wt - 1$ **do**

2   **do**

3         $rand \leftarrow \text{truncate}_d(SecureRand(seed))$

4   **while** $rand \geq r$ *or* $rand \in \hat{\mathbf{f}}$

5     $\hat{\mathbf{f}}[i] \leftarrow rand$

6 **return** $\hat{\mathbf{f}}$

Algorithm 7: Generation of a random sparse polynomial with given (odd) weight

generating new random numbers until a non-repeated number smaller than the system parameter r is generated. Nevertheless, in the following text we validate that these non-constant-time behaviors do not leak any secret information that can be exploited by an adversary: The optional step in Algorithm 6 flips the first bit-position of $\overline{\mathbf{f}}$ and the probabilistic distribution of this bit position does not change as long as the PRNG itself is unbiased (uniformly distributed). An adversary has no advantage in estimating the value of this specific bit even if he observes the timing difference. For Algorithm 7, the timing difference observed by an adversary is actually the total amount of time introduced by the re-generation of random numbers when an overflow ($rand \geq r$) or a repetition ($rand \in \hat{\mathbf{f}}$) occurs. An overflow does not leak any information about $\hat{\mathbf{f}}$ as the information that the generated number is above r does not leak any information on $\hat{\mathbf{f}}$. The probability that the generated random number is smaller than r is $Pr(\text{rand} < r) = r/2^{\lceil log_2(r) \rceil}$. To eliminate the possible timing leakage from the repetition, we introduce a method by changing the while-loop in step 4 of Alg. 7 to while $rand \geq r$. In another word, we generate w_t random numbers each of which is smaller than r without considering whether they collide with each other or not. By doing this, no timing information is leaked on repetition but in this case, a few repetitions might occur which in turn reduces the Hamming weight of the generated sparse polynomial. However, a small loss of Hamming weight w is tolerable in BIKE: To reach λ bits of classical security, $\lambda \approx w - log_2 r$ for BIKE-1 and BIKE-2 and $\lambda \approx w - \frac{1}{2} log_2 r$ for BIKE-3, and 1 or 2 repetitions only decrease the security from 2^{λ} to $2^{\lambda-2}$. More importantly, the probability that a single run of Algorithm 7 without considering collision succeeds if we tolerate

Table 4: Success Rate for generating a random polynomial with odd weight using Algorithm 7

Category		no repetition	1 repetition	2 repetitions	success rate
128-bit	BIKE-[1,2]	0.7826	0.1926	0.0228	0.9981
	BIKE-[3]	0.8179	0.1649	0.0159	0.9989
192-bit	BIKE-[1,2]	0.7671	0.2040	0.0264	0.9976
	BIKE-[3]	0.7992	0.1796	0.0196	0.9985
256-bit	BIKE-[1,2]	0.7521	0.2148	0.0300	0.9970
	BIKE-[3]	0.7840	0.1911	0.0228	0.9981

Table 5: Cycle counts for generating the sparse polynomial h_0/h_1 in BIKE KeyGen

Category		Total	r	w_t	$Pr(\text{rand} < r)$	Pr(Alg. 7 succeeds)
128-bit	BIKE-[1,2]	1279	10163	71	0.62	0.9981
	BIKE-[3]	1062	11027	67	0.67	0.9989
192-bit	BIKE-[1,2]	2120	19853	103	0.60	0.9976
	BIKE-[3]	1847	21683	99	0.66	0.9985
256-bit	BIKE-[1,2]	2801	32749	137	0.99	0.9970
	BIKE-[3]	3350	36131	133	0.55	0.9981

one or two repetitions:

$$Pr(\text{Alg. 7 succeeds}) = \frac{r(r-1)\cdots(r-w_t+1)}{r^{w_t}} + \frac{\binom{w_t}{2}r(r-1)\cdots(r-w_t+2)}{r^{w_t}} \tag{3}$$

$$+ \frac{[\binom{w_t}{3} + \binom{w_t}{2}\binom{w_t-2}{2}/2]r(r-1)\cdots(r-w_t+3)}{r^{w_t}} \tag{4}$$

The probability is very high (at least 0.997, as shown in Table 4):

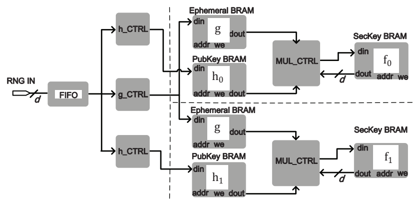
Note that the design uses a simple PRNG based on linear congruence to enable deterministic generation of random numbers (*i.e.*, the function of $SecureRand(\cdot)$ in Algorithm 6, 7). For real-world deployment, such PRNG must be replaced with a cryptographically secure random number generator, *e.g.*, [24,9]. We require at most d random bits per T_{PRNG} clock cycles where $T_{\text{PRNG}} = 13$ in our settings. The average clock cycles for generating a qualified rand number is $T'_{\text{PRNG}} = T_{\text{PRNG}}/Pr(\text{rand} < r)$

Therefore, the total cycle count for generating a random polynomial with odd weight (Algorithm 6) is $\lceil r/d \rceil \cdot T_{\text{PRNG}} + 1$.

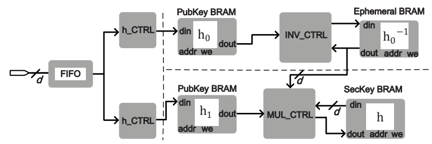
We use the average case for Algorithm 7 to estimate the clock cycle count for generating a random sparse polynomial with given (odd) weight:

$$\left(w_t T'_{\text{PRNG}} + \frac{w_t(w_t+1)}{2} + 5w_t \right) / Pr(\text{Alg. 7 succeeds})$$

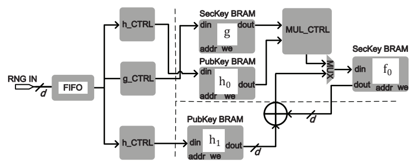
By using the FIFO-based architecture, random number generation and Hamming weight checking are parallelized since FIFO keeps buffering the random



(a) BIKE-1



(b) BIKE-2



(c) BIKE-3

Fig. 8: Overview of the proposed BIKE architectures

numbers regardless if the Hamming weight checking process is idle or not. This optimization brings the cycle count down to:

$$\left({T'}^2_{\mathrm{PRNG}} - 4T'_{\mathrm{PRNG}} + \frac{T'_{\mathrm{PRNG}} + w_t + 6}{2}(w_t - T'_{\mathrm{PRNG}} + 5) \right) / Pr(\text{Alg. 7 succeeds})$$

If a dual-port RAM is used, the above cycle count estimation can be further improved to:

$$\left({T''}^2_{\mathrm{PRNG}} - 4T''_{\mathrm{PRNG}} + \frac{T''_{\mathrm{PRNG}} + \lceil w_t/2 \rceil + 6}{2}(\lceil w_t/2 \rceil - T''_{\mathrm{PRNG}} + 5) \right) / Pr(\text{Alg. 7 succeeds})$$

where $T''_{\mathrm{PRNG}} = T_{\mathrm{PRNG}} / Pr^2(\mathrm{rand} < r)$. The detailed cycle counts for generating the sparse polynomial h_0/h_1 in all BIKE variants are listed in Table 5.

4 Key Generator Architecture

Our key generator performs the computations in BIKE-1/2/3 as shown in Algorithm 1, Algorithm 2, and Algorithm 3. The top-level architectures are depicted in Figure 8a, 8b, 8c respectively. BIKE-1 and BIKE-3 share similar architectures where the sparse polynomial multiplication module (SMUL_CTRL in Section 3.3) is the core building block. The difference within the two architectures is that BIKE-1 requires two SMUL_CTRL modules whereas BIKE-3 requires only one SMUL_CTRL module and one additional addition unit. In our design, the two sparse polynomial multiplications in BIKE-1 are performed in parallel to improve the timing efficiency. Therefore, the total cycle count of BIKE-1 is close

Table 6: Implementation results of BIKE on a Xilinx Virtex-7 XC7V585T FPGA and a Xilinx Spartan-7 XC7S50 FPGA after the place and route process.

Aspect	BIKE-1 KeyGen		BIKE-2 KeyGen		BIKE-3 KeyGen	
	Virtex-7	Spartan-7	Virtex-7	Spartan-7	Virtex-7	Spartan-7
FFs	1055	993	2159	2098	955	911
LUTs	1970	1897	4445	3872	1482	1402
Slices	612	544	1483	1272	472	423
BRAM	7	7	10	10	4	4
Frequency	350 MHz	160 MHz	320 MHz	150 MHz	360 MHz	200 MHz
Time/Op	0.273 ms	0.597 ms	6.719 ms	14.334 ms	0.273 ms	0.492 ms
Compute h_0	1279 cycles		1279 cycles		1062 cycles	
Compute h_1	1279 cycles		1279 cycles		1062 cycles	
Compute g	2068 cycles		—		2250 cycles	
Compute ${h_0}^{-1}$	—		$2,056,683$ cycles		—	
Compute f_0	$90,880$ cycles		—		$94,135$ cycles	
Compute f_1	$90,880$ cycles		—		—	
Compute h	—		$90,880$ cycles		—	
Overall	$95,506$ cycles		$2,150,121$ cycles		$98,509$ cycles	

to that of BIKE-3. Among BIKE-1/2/3, the architecture of BIKE-3 is the most compact while the critical path is the shortest, this is mainly due to the simplicity of the architecture since only one SMUL_CTRL is instantiated in BIKE-3. The modules h_ctrl and g_ctrl in BIKE-3 represents the control logic for generating the sparse polynomials h_0, h_1 and the dense polynomial g, as discussed in Section 3.5.

The key generator architecture for BIKE-2 is significantly more expensive compared to BIKE-1 and BIKE-3. This is mainly due to the requirement of a polynomial inversion step in the key generation process, therefore a INV_CTRL as described in Section 3.4 is needed. After the inversion operation, the inversed polynomial of h_0 is obtained, which is represented as h_0^{-1}. Then, the sparse polynomial multiplication begins by triggering the SMUL_CTRL module with polynomials h_0^{-1} and h_1 as inputs. After the SMUL_CTRL module finishes, the result is sent out as the output, which is the secret key $h = h_0^{-1}h_1$.

Similar optimizations as proposed to schedule pipelines as described in Section 3 are applied to the modules within the BIKE key generator as well: g_ctrl, h_ctrl, SMUL_CTRL, and INV_CTRL. These optimizations improves the timing efficiency for the key generator.

5 Performance Evaluations

In this section, we present the hardware implementation results of the key generator of BIKE on FPGAs. All the results are obtained post place-and-route for a Xilinx high-end FPGA device — Virtex XC7V585T and a low-end FPGA device — Spartan XC7S50 FPGA using the Xilinx Vivado v2018.1 synthesis tool. To the best of our knowledge, this work is the first that reports the performance results of the BIKE key generator on FPGA platforms.

As shown in Table 6, for our BIKE-1 key generator, when the design is synthesized on the Virtex-7 device, the synthesis tool reports a maximum frequency of 350 MHz. Our simulation result shows that it takes $95,506$ cycles to generate one key pair for BIKE-1. Therefore, to generate a complete key pair for BIKE-1, only 0.273 ms are needed. When synthesized for a lower-end Spartan-7 device, our design is reported to be able to run at 160 MHz which leads to a key pair generation time of 0.597 ms. The area consumption of the key generator for BIKE-1 is very low: Only 612 slices and 544 slices are needed on Virtex-7 and Spartan-7 respectively. Compared to BIKE-1, the key generator for BIKE-3 is more compact since only one sparse polynomial multiplication module is needed instead of two. However, BIKE-3 takes slightly more cycles compared to BIKE-1 to generate a key pair due to the additional polynomial addition operation needed for computing the secret key f_0 in BIKE-3. In terms of run time, the performance of the BIKE-3 key generator is slightly better than BIKE-1: The key pair generation take 0.273 ms and 0.492 ms respectively on Virtex-7 and Spartan-7 FPGAs.

For the BIKE-2 key generator, the synthesis tool reports frequencies as 320 MHz and 150 MHz on the Virtex-7 and Spartan-7 devices respectively. A full key generation in BIKE-2 takes $2,150,121$ cycles which leads to run time of 6.719 ms and 14.334 ms respectively on Virtex-7 and Spartan-7 devices. As we can see from Table 6, BIKE-2 is around two orders of magnitude slower compared to BIKE-1 and BIKE-3. This performance gap between BIKE-2 and BIKE-1/3 is mainly due to the requirement of an expensive polynomial inversion operation in BIKE-2 while in BIKE-1/3 it is not needed. Due to the existance of the polynomial inversion module, the area utilization of BIKE-2 key generator is more than $2\times$ bigger compared to BIKE-1/3 as well. Moreover, the memory overhead in BIKE-2 key generator is also larger than that of BIKE-1/3 since the polynomial inversion module requests to save a few ephemeral variables.

6 Related Work

In the following, we compare our work with previous designs. Hardware architectures for the classical McEliece/Niederreiter schemes based on binary Goppa codes have been extensively studied in the last decade [13,36,17,20,38,39]. In 2018, Wang, Niederhagen and Szefer presented the fastest-to-date FPGA-based design for the Goppa code-based Niederreiter cryptosystem [39] which includes all the operations, i.e., key generation [38], encryption and decryption. As we can see from Table 8, with the same security level, cryptosystems based on binary Goppa codes require much more time, slices and memory to generate a key pair. Such memory overhead is inevitable in Goppa code cryptosystems due to its large public key size which does not exisit in cryptosystems based on quasi-cyclic codes. Therefore, compared to binary Goppa code-based key generatos, our design based on QC MDPC codes is a much more lightweight design which is also much more efficient and has a much smaller memory overhead.

Table 7: Performance comparison of our FPGA implementation with dedicate software implementations.

Scheme	SL [bit]	Implementation	Platform	Performance [in millions of cycles]
BIKE-1	128	Ours, constant-time	FPGA	0.095
	128	[1], non-constant-time	Intel Core-i5	≈ 0.73
	128	[12], non-constant-time	Intel Xeon-8124M	0.09
BIKE-2	128	Ours, constant-time	FPGA	2.15
	128	[1], non-constant-time	Intel Core-i5	≈ 6.3
	128	[12], non-constant-time	Intel Xeon-8124M	4.38
BIKE-3	128	Ours, constant-time	FPGA	0.098
	128	[1], non-constant-time	Intel Core-i5	≈ 0.43

A few hardware designs have been proposed on MDPC code-based cryptosystems [21,37,22]. However, all of these designs only focus on the encryption/decryption operations in the cryptosystem. Therefore, a direct comparison of our work with other key generator architectures of MDPC code-based cryptosystems are not feasible.

To get a better comparison with other work on MDPC code-based cryptosystems, we also compared our work with the state-of-the-art software implementations of BIKE. Note that a fair comparison between our hardware design and other software implementations of BIKE is not feasible since the platforms are fully different. Comparison results are provided in Table 8. We first compare our design with the implementation which is included in the BIKE specification submitted to the NIST 2nd round PQC standardization process [1]. As we can see from Table 8, with the same security levels, for BIKE-1, our design is around 7.7× faster than their implementation in terms of cycle count. Moreover, the software implementation provided in [1] is not constant-time and therefore is vulerable to timing side-channel attacks. An optimized software implementation for BIKE is reported in [12]. In this implementation, the core functionality in BIKE is written in assembly language, moreover, the PCLMULQDQ instruction extension is exploited to enable carry-less multiplication to speed up the computations in BIKE. The performance results of [12] is provided in Table 8. In terms of cycle count, for BIKE-1, the performance of our hardware design is comparable to [12] while an over 2× speedup is achieved in BIKE-2.

7 Conclusions

This paper presented a constant-time hardware-based key generator for BIKE which is among the 2nd round candidate in the NIST PQC standardization process. Novel multiplication and squaring algorithms were proposed to accelerate the key generation with low logic overhead. Experimental results show that after a dedicate optimized pipeline scheduling for the proposed algorithms, our work outperforms other code-based hardware key generators in terms of timing performance, logic utilization and memory consumption. This work even beat-

Table 8: Performance comparison of our FPGA implementation with other code-based key generators.

Scheme	SL [bit]	Platform	f[MHz]	Time/Op	Cycles	Slices/LUTs/FFs	BRAM
MDPC code:							
Ours (BIKE-1)			**350**	**0.273ms**	9.55×10^4	**612/1970/1055**	**7**
Ours (BIKE-2)	**128**	Virtex-7	**320**	**6.719ms**	2.15×10^6	**1483/4445/2159**	**10**
Ours (BIKE-3)			**360**	**0.273ms**	9.85×10^4	**472/1482/955**	**4**
Goppa code:							
[39]	256	Virtex Ultrascale+	225	3.98ms	$\approx 1.01 \times 10^{11}$	—/112845/—	375
[39]	103	Virtex-5	168	16ms	$\approx 2.72 \times 10^6$	8171/19257/20622	89
[7]	128	Artix-7	38.1	41.9ms	$\approx 1.59 \times 10^6$	—/25327/49383	168

s software implementations with powerful CPUs in terms of cycle count. The methods proposed in this paper are generic and applicable to other code-based schemes if the underlying code is quasi-cyclic.

References

1. N. Aragon, P. Barreto, S. Bettaieb, L. Bidoux, O. Blazy, J.-C. Deneuville, P. Gaborit, S. Gueron, T. Guneysu, C. A. Melchor, et al. Bike: Bit flipping key encapsulation. 2017.
2. M. Baldi, A. Barenghi, F. Chiaraluce, G. Pelosi, and P. Santini. Ledakem: a post-quantum key encapsulation mechanism based on qc-ldpc codes. In *International Conference on Post-Quantum Cryptography*, pages 3–24. Springer, 2018.
3. M. Baldi, M. Bianchi, and F. Chiaraluce. Optimization of the parity-check matrix density in qc-ldpc code-based mceliece cryptosystems. In *Communications Workshops (ICC), 2013 IEEE International Conference on*, pages 707–711. IEEE, 2013.
4. M. Baldi, M. Bianchi, F. Chiaraluce, J. Rosenthal, and D. Schipani. Enhanced public key security for the McEliece cryptosystem. *Journal of Cryptology*, 29(1):1–27, 2016.
5. M. Baldi, M. Bodrato, and F. Chiaraluce. A new analysis of the Mceliece cryptosystem based on QC-LDPC codes. In *Security and Cryptography for Networks*, pages 246–262. Springer, 2008.
6. P. S. Barreto, S. Gueron, T. Gueneysu, R. Misoczki, E. Persichetti, N. Sendrier, and J.-P. Tillich. Cake: code-based algorithm for key encapsulation. In *IMA International Conference on Cryptography and Coding*, pages 207–226. Springer, 2017.
7. D. J. Bernstein, T. Chou, T. Lange, R. Misoczki, R. Niederhagen, E. Persichetti, P. Schwabe, J. Szefer, and W. Wang. Classic mceliece: conservative code-based cryptography 30 march 2019. 2019.
8. L. Chen, S. Jordan, Y.-K. Liu, D. Moody, R. Peralta, R. Perlner, and D. Smith-Tone. Report on post-quantum cryptography. *National Institute of Standards and Technology Internal Report*, 8105, 2016.
9. A. Cherkaoui, V. Fischer, L. Fesquet, and A. Aubert. A very high speed true random number generator with entropy assessment. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 179–196. Springer, 2013.

10. T. Chou. QcBits: constant-time small-key code-based cryptography. In *International Conference on Cryptographic Hardware and Embedded Systems*, pages 280–300. Springer, 2016.
11. J.-C. Deneuville, P. Gaborit, and G. Zémor. Ouroboros: A simple, secure and efficient key exchange protocol based on coding theory. In *International Workshop on Post-Quantum Cryptography*, pages 18–34. Springer, 2017.
12. N. Drucker and S. Gueron. A toolbox for software optimization of qc-mdpc code-based cryptosystems. *Journal of Cryptographic Engineering*, pages 1–17, 2017.
13. T. Eisenbarth, T. Güneysu, S. Heyse, and C. Paar. Microeliece: Mceliece for embedded devices. In *Cryptographic Hardware and Embedded Systems-CHES 2009*, pages 49–64. Springer, 2009.
14. T. Fabšič, V. Hromada, P. Stankovski, P. Zajac, Q. Guo, and T. Johansson. A reaction attack on the qc-ldpc mceliece cryptosystem. In *International Workshop on Post-Quantum Cryptography*, pages 51–68. Springer, 2017.
15. P. Gaborit, A. Otmani, and H. T. Kalachi. Polynomial-time key recovery attack on the faure–loidreau scheme based on gabidulin codes. *Designs, Codes and Cryptography*, 86(7):1391–1403, 2018.
16. M. Georgieva and F. de Portzamparc. Toward secure implementation of mceliece decryption. In *International Workshop on Constructive Side-Channel Analysis and Secure Design*, pages 141–156. Springer, 2015.
17. S. Ghosh, J. Delvaux, L. Uhsadel, and I. Verbauwhede. A speed area optimized embedded co-processor for mceliece cryptosystem. In *Application-Specific Systems, Architectures and Processors (ASAP), 2012 IEEE 23rd International Conference on*, pages 102–108. IEEE, 2012.
18. V. D. Goppa. A new class of linear correcting codes. *Problemy Peredachi Informatsii*, 6(3):24–30, 1970.
19. Q. Guo, T. Johansson, and P. Stankovski. A key recovery attack on mdpc with cca security using decoding errors. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 789–815. Springer, 2016.
20. S. Heyse and T. Güneysu. Towards one cycle per bit asymmetric encryption: code-based cryptography on reconfigurable hardware. In *Cryptographic Hardware and Embedded Systems–CHES 2012*, pages 340–355. Springer, 2012.
21. S. Heyse, I. Von Maurich, and T. Güneysu. Smaller keys for code-based cryptography: QC-MDPC Mceliece implementations on embedded devices. In *Cryptographic Hardware and Embedded Systems–CHES 2013*, pages 273–292. Springer, 2013.
22. J. Hu and R. C. Cheung. Area-time efficient computation of Niederreiter encryption on QC-MDPC codes for embedded hardware. *IEEE Transactions on Computers*, 2017.
23. J. Hu, W. Guo, J. Wei, and R. C. Cheung. Fast and generic inversion architectures over gf(2m) using modified itoh–tsujii algorithms. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 62(4):367–371, 2015.
24. R. Laue, O. Kelm, S. Schipp, A. Shoufan, and S. A. Huss. Compact aes-based architecture for symmetric encryption, hash function, and random number generation. In *2007 International Conference on Field Programmable Logic and Applications*, pages 480–484. IEEE, 2007.
25. Y. X. Li, R. H. Deng, and X. M. Wang. On the equivalence of mceliece's and niederreiter's public-key cryptosystems. *IEEE Transactions on Information Theory*, 40(1):271–273, 1994.
26. P. Loidreau. A new rank metric codes based encryption scheme. In *International Workshop on Post-Quantum Cryptography*, pages 3–17. Springer, 2017.

27. C. Löndahl and T. Johansson. A new version of mceliece pkc based on convolutional codes. In *International Conference on Information and Communications Security*, pages 461–470. Springer, 2012.
28. R. J. McEliece. A public-key cryptosystem based on algebraic coding theory. *DSN progress report*, 42(44):114–116, 1978.
29. R. Misoczki and P. S. Barreto. Compact mceliece keys from goppa codes. In *International Workshop on Selected Areas in Cryptography*, pages 376–392. Springer, 2009.
30. R. Misoczki, J.-P. Tillich, N. Sendrier, and P. S. Barreto. MDPC-McEliece: New McEliece variants from moderate density parity-check codes. In *IEEE International Symposium on Information Theory Proceedings (ISIT), 2013*, pages 2069–2073. IEEE, 2013.
31. C. Monico, J. Rosenthal, and A. Shokrollahi. Using low density parity check codes in the mceliece cryptosystem. In *Information Theory, 2000. Proceedings. IEEE International Symposium on*, page 215. IEEE, 2000.
32. H. Niederreiter. Knapsack-type cryptosystems and algebraic coding theory. *Problems Of Control and Information Theory-Problemy Upravleniya I Thorii Informatsii*, 15(2):159–166, 1986.
33. A. Nilsson, T. Johansson, and P. S. Wagner. Error amplification in code-based cryptography. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 238–258, 2019.
34. A. Otmani, J.-P. Tillich, and L. Dallot. Cryptanalysis of two mceliece cryptosystems based on quasi-cyclic codes. *Mathematics in Computer Science*, 3(2):129–140, 2010.
35. P. W. Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM journal on computing*, 26(5):1484–1509, 1997.
36. A. Shoufan, T. Wink, H. G. Molter, S. Huss, E. Kohnert, et al. A novel cryptoprocessor architecture for the Mceliece public-key cryptosystem. *Computers, IEEE Transactions on*, 59(11):1533–1546, 2010.
37. I. von Maurich and T. Güneysu. Lightweight code-based cryptography: QC-MDPC McEliece encryption on reconfigurable devices. In *Proceedings of the conference on Design, Automation & Test in Europe*, page 38. European Design and Automation Association, 2014.
38. W. Wang, J. Szefer, and R. Niederhagen. Fpga-based key generator for the niederreiter cryptosystem using binary goppa codes. In *International Conference on Cryptographic Hardware and Embedded Systems*, pages 253–274. Springer, 2017.
39. W. Wang, J. Szefer, and R. Niederhagen. Fpga-based niederreiter cryptosystem using binary goppa codes. In *International Conference on Post-Quantum Cryptography*, pages 77–98. Springer, 2018.

Table 9: An illustrative example for computing polynomial inverse a^{-1} in $\mathbb{F}_2[x]/(x^r+1)$ with $r = 10163$ used in BIKE-2. This table explicitly demonstrates step by step a constat-time execution for inverse.

i	u_i	rule	$[\beta_{u_{i_1}}(a)]^{2^{u_{i_2}}} \cdot \beta_{u_{i_2}}(a)$	$\beta_{u_{i_i}}(a) = a^{2^{u_i}-1}$
0	1	—	—	$\beta_{u_{i_0}}(a) = a^{2^1-1}$
1	2	$2u_{i-1}$	$[\beta_{u_{i_0}}(a)]^{2^{u_{i_0}}} \cdot \beta_{u_{i_0}}(a)$	$\beta_{u_{i_1}}(a) = a^{2^2-1}$
2	4	$2u_{i-1}$	$[\beta_{u_{i_1}}(a)]^{2^{u_{i_1}}} \cdot \beta_{u_{i_1}}(a)$	$\beta_{u_{i_2}}(a) = a^{2^4-1}$
3	8	$2u_{i-1}$	$[\beta_{u_{i_2}}(a)]^{2^{u_{i_2}}} \cdot \beta_{u_{i_2}}(a)$	$\beta_{u_{i_3}}(a) = a^{2^8-1}$
4	9	$u_{i-1}+u_0$	$[\beta_{u_{i_3}}(a)]^{2^{u_{i_0}}} \cdot \beta_{u_{i_0}}(a)$	$\beta_{u_{i_4}}(a) = a^{2^9-1}$
5	18	$2u_{i-1}$	$[\beta_{u_{i_4}}(a)]^{2^{u_{i_4}}} \cdot \beta_{u_{i_4}}(a)$	$\beta_{u_{i_5}}(a) = a^{2^{18}-1}$
6	19	$u_{i-1}+u_0$	$[\beta_{u_{i_5}}(a)]^{2^{u_{i_0}}} \cdot \beta_{u_{i_0}}(a)$	$\beta_{u_{i_4}}(a) = a^{2^{19}-1}$
7	38	$2u_{i-1}$	$[\beta_{u_{i_6}}(a)]^{2^{u_{i_6}}} \cdot \beta_{u_{i_6}}(a)$	$\beta_{u_{i_7}}(a) = a^{2^{38}-1}$
8	39	$u_{i-1}+u_0$	$[\beta_{u_{i_7}}(a)]^{2^{u_{i_0}}} \cdot \beta_{u_{i_0}}(a)$	$\beta_{u_{i_8}}(a) = a^{2^{39}-1}$
9	78	$2u_{i-1}$	$[\beta_{u_{i_8}}(a)]^{2^{u_{i_8}}} \cdot \beta_{u_{i_8}}(a)$	$\beta_{u_{i_9}}(a) = a^{2^{78}-1}$
10	79	$u_{i-1}+u_0$	$[\beta_{u_{i_9}}(a)]^{2^{u_{i_0}}} \cdot \beta_{u_{i_0}}(a)$	$\beta_{u_{i_{10}}}(a) = a^{2^{79}-1}$
11	158	$2u_{i-1}$	$[\beta_{u_{i_{10}}}(a)]^{2^{u_{i_{10}}}} \cdot \beta_{u_{i_{10}}}(a)$	$\beta_{u_{i_{11}}}(a) = a^{2^{158}-1}$
12	316	$2u_{i-1}$	$[\beta_{u_{i_{11}}}(a)]^{2^{u_{i_{11}}}} \cdot \beta_{u_{i_{11}}}(a)$	$\beta_{u_{i_{12}}}(a) = a^{2^{316}-1}$
13	317	$u_{i-1}+u_0$	$[\beta_{u_{i_{12}}}(a)]^{2^{u_{i_0}}} \cdot \beta_{u_{i_0}}(a)$	$\beta_{u_{i_{13}}}(a) = a^{2^{317}-1}$
14	634	$2u_{i-1}$	$[\beta_{u_{i_{13}}}(a)]^{2^{u_{i_{13}}}} \cdot \beta_{u_{i_{13}}}(a)$	$\beta_{u_{i_{14}}}(a) = a^{2^{634}-1}$
15	635	$u_{i-1}+u_0$	$[\beta_{u_{i_{14}}}(a)]^{2^{u_{i_0}}} \cdot \beta_{u_{i_0}}(a)$	$\beta_{u_{i_{15}}}(a) = a^{2^{635}-1}$
16	1270	$2u_{i-1}$	$[\beta_{u_{i_{15}}}(a)]^{2^{u_{i_{15}}}} \cdot \beta_{u_{i_{15}}}(a)$	$\beta_{u_{i_{16}}}(a) = a^{2^{1270}-1}$
17	2540	$2u_{i-1}$	$[\beta_{u_{i_{16}}}(a)]^{2^{u_{i_{16}}}} \cdot \beta_{u_{i_{16}}}(a)$	$\beta_{u_{i_{17}}}(a) = a^{2^{2540}-1}$
18	5080	$2u_{i-1}$	$[\beta_{u_{i_{17}}}(a)]^{2^{u_{i_{17}}}} \cdot \beta_{u_{i_{17}}}(a)$	$\beta_{u_{i_{18}}}(a) = a^{2^{5080}-1}$
19	10160	$2u_{i-1}$	$[\beta_{u_{i_{18}}}(a)]^{2^{u_{i_{18}}}} \cdot \beta_{u_{i_{18}}}(a)$	$\beta_{u_{i_{19}}}(a) = a^{2^{10160}-1}$
20	10161	$u_{i-1}+u_0$	$[\beta_{u_{i_{19}}}(a)]^{2^{u_{i_0}}} \cdot \beta_{u_{i_0}}(a)$	$\beta_{u_{i_{20}}}(a) = a^{2^{10161}-1}$

Appendix

7.1 Illustrative example for constant-time ITA used in BIKE

We illustrate a concrete instance for computing the polynomial inversion used in BIKE-2 as shown in Table 9.

7.2 Proof of Theorem 1

We briefly proof Theorem 1 as follows: Consider the straighforward squaring operation for a random polynomial $a(x) \in \mathcal{R}$:

$$
\begin{aligned}
a^2(x) &= (a_{r-1}x^{r-1} + a_{r-2}x^{r-2} + \cdots + a_1x + a_0)^2 && (5)\\
&= a_{r-1}x^{2(r-1)} + a_{r-2}x^{2(r-2)} + \cdots + a_1x^2 + a_0 && (6)\\
&= \widetilde{a_{r-1}}x^{r-1} + \widetilde{a_{r-2}}x^{r-2} + \cdots + \widetilde{a_1}x + \widetilde{a_0} && (7)
\end{aligned}
$$

The computation of the coefficients $\widetilde{a_i}$ can be realized by a matrix multiplication operation, which relies on the base transformation matrix BT, as shown follows:

$$a^2(x) = [a_0, a_1, \cdots, a_{r-1}] \begin{bmatrix} x^0 \\ x^2 \\ \vdots \\ x^{2(r-2)} \\ x^{2(r-1)} \end{bmatrix} = [a_0, a_1, \cdots, a_{r-1}] \cdot \mathrm{BT}_{r\times r} \cdot \begin{bmatrix} x^0 \\ x^1 \\ \vdots \\ x^{r-2} \\ x^{r-1} \end{bmatrix}$$

where $[\widetilde{a_0}, \widetilde{a_1}, \cdots, \widetilde{a_{r-1}}] = [a_0, a_1, \cdots, a_{r-1}] \cdot \mathrm{BT}$ and BT is a $r \times r$ permutation matrix defined as:

$$\mathrm{BT} = \begin{bmatrix} \mathbf{1}_0 & 0 & \cdots & \cdots & \cdots \\ \cdots & \mathbf{1}_2 & 0 & \cdots & \cdots \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ \vdots & \vdots & \mathbf{1}_{2r-4} & 0 & \vdots \\ \vdots & \vdots & \vdots & \mathbf{1}_{2r-2} & 0 \end{bmatrix}$$

$\mathbf{1}_i$ represents the ($i \bmod r$)-th (counting from zero) non-zero entry in the corresponding row of the matrix BT. More generally, we derive the base transformation matrix $\mathrm{BT}_n = \mathrm{BT}^n$ for computing polynomial squaring continuously: $a(x)^{2^n} = \widetilde{a_{r-1}} x^{r-1} + \widetilde{a_{r-2}} x^{r-2} + \cdots + \widetilde{a_1} x + \widetilde{a_0}$. As we will see in Section 3.4, this is particularly useful in computing the inverse of a polynomial.

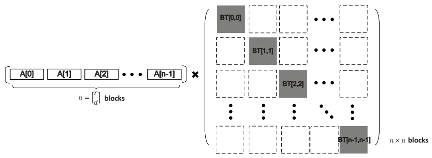
$$\mathrm{BT}^n = \begin{bmatrix} \mathbf{1}_0 & 0 & \cdots & \cdots & \cdots \\ \cdots & \mathbf{1}_{2^n} & 0 & \cdots & \cdots \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ \vdots & \vdots & \mathbf{1}_{(r-2)2^n} & 0 & \vdots \\ \vdots & \vdots & \vdots & \mathbf{1}_{(r-1)2^n} & 0 \end{bmatrix}$$

In order to compute the coefficient vector $[\widetilde{a_0}, \widetilde{a_1}, \cdots, \widetilde{a_{r-1}}] = [a_0, a_1, \cdots, a_{r-1}] \cdot \mathrm{BT}^n$, BT^n must be represented in column view format, *i.e.*, the row number of the non-zero entry for every column of BT^n. In this form, the squaring of $a(x)$ is essentially the substitutions of the coefficients of $a(x)$:

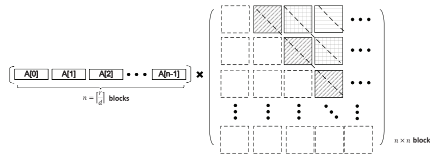
$$\mathrm{BT}^n = \begin{bmatrix} \mathbf{1}_0 & \vdots & \vdots & \vdots & \vdots \\ \vdots & \mathbf{1}_{2^{-n}} & \vdots & \vdots & \vdots \\ \vdots & \vdots & \vdots & \mathbf{1}_{(r-2)2^{-n}} & \vdots \\ \vdots & \vdots & \vdots & \vdots & \mathbf{1}_{(r-1)2^{-n}} \end{bmatrix}$$

7.3 Remarks on generic multiplication

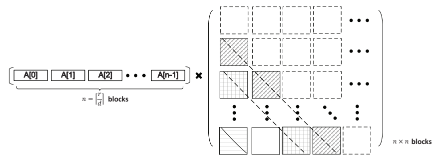
We describe the proposed generic multiplication algorithm (Alg. 4) in more details here. Similar to the proposed squaring algorithm, another base transforma-



(a) Diagonal Computation



(b) Upper Triangle Computation



(c) Lower Triangle Computation

Fig. 9: Illustration for the proposed generic multiplication algorithm

tion matrix BT_{mul} is used to compute the coefficients $\widetilde{c_i}$, shown as follows:

$$a(x)b(x) = [\widetilde{c_0}, \widetilde{c_1}, \cdots, \widetilde{c_{r-1}}] \begin{bmatrix} x^0 \\ x^1 \\ \vdots \\ x^{r-2} \\ x^{r-1} \end{bmatrix} = [a_0, a_1, \cdots, a_{r-1}] \cdot \mathrm{BT}_{mul, r\times r} \cdot \begin{bmatrix} x^0 \\ x^1 \\ \vdots \\ x^{r-2} \\ x^{r-1} \end{bmatrix}$$

BT_{mul} is a $r \times r$ matrix where each column/row is a cyclic form of the vector $[b_{r-1}, b_{r-2}, \ldots, b_1, b_0]$:

$$\mathrm{BT}_{mul} = \begin{bmatrix} b_0 & b_1 & \cdots & b_{d-1} & \cdots & b_{r-1} \\ b_{r-1} & b_0 & \cdots & b_{d-2} & \cdots & b_{r-2} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ b_2 & b_3 & \cdots & b_{d+1} & \cdots & b_1 \\ b_1 & b_2 & \cdots & b_d & \cdots & b_0 \end{bmatrix}$$

Storing the full BT_{mul} matrix is redundant since the blocks along each diagonal are identical (for example, $\mathrm{BT}_{mul}[0 : d-1, 0 : d-1] = \mathrm{BT}_{mul}[d : 2d-1, d : 2d-1] = \ldots$). This way, BT_{mul} is split into three parts: the central diagonal, the upper triangle, and the lower triangle. Figure 9a, 9b, and 9c illustrate how to compute these parts. By eliminating such redundancy, the communication overhead for loading BT_{mul} is minimized which enables a fast pipeline scheduling in the generic polynomial multiplication module.