Racing BIKE: Improved Polynomial Multiplication and Inversion in Hardware

Jan Richter-Brockmann[1,2], Ming-Shing Chen[1], Santosh Ghosh[2] and Tim Güneysu[1,3]

[1] Ruhr-Universität Bochum, Horst-Görtz Institute for IT-Security, Bochum, Germany
[2] Security and Privacy Research, Intel Labs, Intel Coorperation, Hillsboro, Oregon
[3] DFKI, Bremen, Germany
jan.richter-brockmann@rub.de,ming-shing.chen@rub.de,santosh.ghosh@intel.com,tim.gueneysu@rub.de

Abstract. BIKE is a Key Encapsulation Mechanism selected as an alternate candidate in NIST's PQC standardization process, in which performance plays a significant role in the third round. This paper presents FPGA implementations of BIKE with the best area-time performance reported in literature. We optimize two key arithmetic operations, which are the sparse polynomial multiplication and the polynomial inversion. Our sparse multiplier achieves time-constancy for sparse polynomials of indefinite Hamming weight used in BIKE's encapsulation. The polynomial inversion is based on the extended Euclidean algorithm, which is unprecedented in current BIKE implementations. Our optimized design results in a 5.5 times faster key generation compared to previous implementations based on Fermat's little theorem.
Besides the arithmetic optimizations, we present a united hardware design of BIKE with shared resources and shared sub-modules among KEM functionalities. On Xilinx Artix-7 FPGAs, our light-weight implementation consumes only 3 777 slices and performs a key generation, encapsulation, and decapsulation in 3 797 µs, 443 µs, and 6 896 µs, respectively. Our high-speed design requires 7 332 slices and performs the three KEM operations in 1 672 µs, 132 µs, and 1 892 µs, respectively.

Keywords: BIKE · QC-MDPC · PQC · Reconfigurable Devices · FPGA.

1 Introduction

Due to extensive research and advanced progress in quantum computation during the last decades [Gam20], in 2017, the National Institute of Standards and Technology (NIST) announced a Post-Quantum Cryptography (PQC) standardization process with the target to find public-key cryptographic algorithms that provide security in the presence of quantum computers [NIS17].

After the call for proposals, the NIST received 69 submissions which were revised with respect to security, efficiency (e.g., key sizes and latency), and implementation costs for software and hardware. Eventually, after the third round, they selected seven finalists and eight alternate candidates [NIS20b]. While the finalists are all considered for standardization, the alternate candidates will be reviewed and may be evaluated in a fourth round such that they potentially could be standardized as well [NIS20b].

The Bit Flipping Key Encapsulation (BIKE) [ABB+20] is one of the NIST's alternate candidates in the Key Encapsulation Mechanism (KEM) category. The security of BIKE relies on the hardness of decoding linear error-correcting codes. More specifically, as underlying linear codes, BIKE utilizes Quasi-Cyclic Moderate-Density Parity-Check (QC-MDPC) codes, which were first presented by Misoczki et *al.* [MTSB13] in 2013.

In this work, we target to improve the efficiency of the KEM functionalities of BIKE by an Field-Programmable Gate Array (FPGA) hardware design. Since NIST announced that performance plays an important role in their PQC standardization efforts [NIS20a], researchers presented several optimization techniques for BIKE on the suggested platforms including the AVX2 instruction set on x86, embedded microprocessors, and FPGAs. For example, Drucker et *al.* [DGK20a] optimized BIKE for x86 Central Processing Units (CPUs). Chen et *al.* [CCK21] presented optimization techniques for x86 and Arm Cortex M4. Richter-Brockmann et *al.* [RBMG21] proposed an optimized scalable hardware implementation for reconfigurable devices. In this paper, we propose new optimization techniques for efficient FPGA implementations of BIKE and report significant improvements compared to previous works.

Related Works on FPGAs. Although there were several early works implementing QC-MDPC codes on hardware devices for variants of the McEliece cryptosystem [VMG14, HVMG13] and for the Niederreiter framework [HC17], the first hardware implementation of BIKE was presented with the round-two submission of NIST's PQC standardization process [ABB+19]. The implementation was designed for an older version of BIKE (called BIKE-1) and only supported the key generation and encapsulation.

In 2020, Reinders et *al.* [RMGS20] proposed a complete hardware design which, however, targets the older parameters of BIKE. Besides, they presented an efficient hardware implementation for a novel constant-time decoder.

Recently, Richter-Brockmann et *al.* [RBMG21] presented the first complete hardware design of the current BIKE version [ABB+20]. They implemented for the first time the Black-Gray-Flip (BGF) decoder on hardware, introduced an optimized polynomial inversion module (based on Fermat's little theorem), and proposed a scalable multiplier.

In further detail, BIKE poses several challenges on the arithmetic level. For improving the polynomial multipliers in code-based schemes, Hu et *al.* [HWCW19] presented two different approaches. While the first design is based on a schoolbook multiplication, the second multiplier improves multiplications by exploiting the sparseness of the polynomials used in QC-MDPC codes. Additionally, they instantiated their designs to create a key generation module based on previous parameter sets of BIKE.

Barenghi et *al.* [BFG+19] presented similar approaches to implement polynomial multiplications for the code-based scheme LEDAcrypt [BBC+19]. They explored different configurations of schoolbook and sparse multipliers for Xilinx FPGAs.

Contribution. In this work, we revise previous concepts and identify significant improvements and systematic explorations of the hardware implementation of BIKE on FPGAs. Specifically, we introduce an optimized polynomial multiplier that exploits the sparseness of QC-MDPC codes while performing all multiplications applied in BIKE in constant time. In addition to that, we present a novel component for polynomial inversion based on the extended Euclidean algorithm (extGCD) accelerating the key generation in hardware. For that we adapt the extGCD from the constant-time algorithm recently proposed by Bernstein and Yang [BY19], and demonstrate that this approach clearly outperforms previous implementations based on Fermat's little theorem in the specific case of BIKE. As a design constraint, our implementation is highly scalable to instantiate specifically tailored cryptographic components for any use-case.

Besides these major arithmetic-oriented optimizations, we also substitute symmetric cryptography from encapsulation and decapsulation implementations presented in [RBMG21] with a single Keccak core to demonstrate the authors' assertion of achieving a lower footprint by applying this modification. Additionally, we present a combined hardware implementation of BIKE that consolidates all three KEM algorithms in one single, united design. This approach enables resource and module sharing between the

KEM algorithms achieving a design that reduces the overall implementation costs.

Our implementations are written in Verilog and are publicly available at https://github.com/Chair-for-Security-Engineering/RacingBIKE.

Outline. In Section 2, we briefly introduce BIKE and cover the background of polynomial arithmetic that is necessary for our hardware implementations. Section 3 starts with an introduction of our design considerations. Afterwards, we introduce our modifications with respect to the random oracles, present our multiplier and inversion modules, and describe the composition of an united hardware design. In Section 4, we evaluate all designs with respect to implementation costs and performance. Before we conclude our work in Section 6, we briefly discuss the resistance against side channels and address the transferability of our approaches to software implementations in Section 5.

2 Preliminaries

In this section, we describe the algorithms and parameters forming BIKE. Then, we summarize important polynomial arithmetic.

2.1 Notations

Throughout this work, we use the following notations:

- $\mathbb{F}_2$: Finite field of two elements $\{0, 1\}$.
- $\mathbb{F}_2[X]$: Polynomials with coefficients in $\mathbb{F}_2$, or bit polynomials.

  In this work, we store a polynomial $a = a_0 + a_1X + \cdots \in \mathbb{F}_2[X]$ as a bit sequence of coefficients $(a_0, a_1, \ldots)$. The 0-th bit corresponds to the coefficient a_0; the first bit corresponds to the coefficient a_1; and so on.
- r: The parameter defining the length of polynomials in BIKE.
- $\mathcal{R} := \mathbb{F}_2[X]/(X^r - 1)$: The cyclic polynomial ring used in BIKE.

  Multiplication in $\mathcal{R}$ is generally implemented as multiplication of bit polynomials in $\mathbb{F}_2[X]$ and followed by a modulo operation by $X^r - 1$ for terms of degrees $\geq r$.
- $|f|$: Hamming weight of a bit polynomial f.
- b: Bandwidth (in bits) for accessing data from memory in our FPGA implementation.
- $f[i]$ is the i-th b-bit chunk of a polynomial f for $0 \leq i < \lceil \frac{r}{b} \rceil$.

2.2 BIKE

We divide this section in three paragraphs describing the KEM functions of BIKE, introducing required hash functions, and summarizing BIKE's parameters.

KEM Functions. The BIKE KEM comprises three algorithms — key generation, encapsulation, and decapsulation. The key generation (see Algorithm 1) outputs a key pair. It randomly samples two sparse polynomials $(h_0, h_1) \in \mathcal{R}^2$ and a random string σ as the private key. By inverting h_0 and multiply the result by h_1, the key generation computes the public key h as shown in line 3.

Algorithm 2 describes the encapsulation which starts by sampling a message m and deriving two error polynomials (e_0, e_1) from $\mathbf{H}(m)$. Afterwards, it computes the first part of the cryptogram c_0 by multiplying e_1 by the public key h and adding (xor) the result to

Algorithm 1: Key Generation.

Input : BIKE parameters n, w, t, ℓ.
Output : Private key (h_0, h_1, σ) and public key h.

1 Generate $(h_0, h_1) \xleftarrow{\$} \mathcal{R}^2$ both of odd weight $|h_0| = |h_1| = w/2$.
2 Generate $\sigma \xleftarrow{\$} \{0,1\}^\ell$ uniformly at random.
3 Compute $h \leftarrow h_1 h_0^{-1}$.
4 Return (h_0, h_1, σ) and h.

Algorithm 2: Encapsulation.

Input : Public key h.
Output : Encapsulated key K and ciphertext $C = (c_0, c_1)$.

1 Generate $m \xleftarrow{\$} \{0,1\}^\ell$ uniformly at random.
2 Compute $(e_0, e_1) \leftarrow \mathbf{H}(m)$.
3 Compute $C = (c_0, c_1) \leftarrow (e_0 + e_1 h, m \oplus \mathbf{L}(e_0, e_1))$.
4 Compute $K \leftarrow \mathbf{K}(m, C)$.
5 Return (C, K).

Algorithm 3: Decapsulation.

Input : Private key (h_0, h_1, σ) and ciphertext $C = (c_0, c_1)$.
Output : Decapsulated key K.

1 Compute syndrome $s \leftarrow c_0 h_0$.
2 Compute $\{(e'_0, e'_1), \perp\} \leftarrow \texttt{decoder}(s, h_0, h_1)$.
3 Compute $m' \leftarrow c_1 \oplus \mathbf{L}(e'_0, e'_1)$.
4 **if** $\mathbf{H}(m') \neq (e'_0, e'_1)$ **then**
5 | Compute $K \leftarrow \mathbf{K}(\sigma, C)$.
6 **else**
7 | Compute $K \leftarrow \mathbf{K}(m', C)$.
8 Return K.

e_0. This step represents the encoding procedure of linear codes but with the difference that the errors are added intentionally. The second part of the cryptogram c_1 is generated by adding the message m to the output of the hash function $\mathbf{L}$. Eventually, the algorithm derives the shared key K by hashing the cryptogram and the message with $\mathbf{K}$.

Algorithm 3 shows the decapsulation that recovers the error polynomials (e_0, e_1) from the cryptogram C. It first computes the syndrome s in line 1 as a common procedure for decoding linear codes. The syndrome and the private key (h_0, h_1) are then fed into a `decoder` to determine the error polynomials (e'_0, e'_1). The BIKE specification [ABB+21] applies a BGF decoder which was extensively investigated in [DGK20b]. If the decoding is successful, the algorithm calculates the message m' from the ciphertext C and the error polynomials (cf. line 3). To ensure that m' and the determined error polynomials match the one generated in the encapsulation, it applies the same sampling algorithm $\mathbf{H}$ to m' and compares the result to the error polynomials returned from the decoder. In case the pair is valid, it computes the shared key $K = \mathbf{K}(m', C)$. Otherwise it computes K using the secret string σ belonging to the private key.

Hash Functions. In BIKE, the encapsulation and decapsulation utilize the three functions $\mathbf{H}$, $\mathbf{K}$, and $\mathbf{L}$ which are modeled as random oracles and defined over the following domains:

Table 1: BIKE parameters.

Security	r	w	t
Level 1	12 323	142	134
Level 3	24 659	206	199
Level 5	40 973	274	264

$$\mathbf{H} : \{0,1\}^{\ell} \to \{0,1\}^{2r}_{[t]} \qquad \mathbf{K} : \{0,1\}^{r+2\ell} \to \{0,1\}^{\ell} \qquad \mathbf{L} : \{0,1\}^{2r} \to \{0,1\}^{\ell}$$

The latest specification [ABB+21] unifies the three random oracles to hash functions based on Keccak [BDPA13]. **H** maps an ℓ-bit string into a $2r$-bit string with Hamming weight t. It is implemented by a SHAKE256-based Pseudo-Random Number Generator (PRNG) while it was realized by AES-256 in previous versions. **K** and **L** uses a SHA3-384 implementation replacing the SHA2-384 of previous versions.

Parameters. Table 1 summarizes the parameters of BIKE for various security levels of NIST's PQC standardization process. As already introduced above, the parameter r represents the length of polynomials used in BIKE. The parameter w specifies the Hamming weight of the private key polynomials (h_0, h_1), satisfying $|h_0| = |h_1| = w/2$. The parameter t defines the decoding radius, i.e., the Hamming weight of the errors randomly sampled in the encapsulations. Eventually, ℓ specifies the length of the shared key of the KEM, which is fixed to 256 bits for all security levels.

2.3 Polynomial Multiplication by Sparse Polynomials

In BIKE, all multiplications in $\mathcal{R}$ comprise a sparse operand $f \in \mathcal{R}$ with $|f| \ll r$. For the key generation, h_1 is the sparse polynomial in the multiplication $h_1 \cdot h_0^{-1}$. For the encapsulation, e_1 is sparse in $e_1 \cdot h$. For the decapsulation, h_0 is sparse in $c_0 \cdot h_0$. The `decoder` contains some additional multiplications by the sparse polynomials (h_0, h_1) which are part of the private key.

We represent a sparse polynomial as a set of indexes corresponding to its non-zero terms. For example, the set $I_f = \{i_1, \ldots, i_t\}$ represents the sparse polynomial $f = X^{i_1} + \cdots + X^{i_t}$ with the Hamming weight $|f| = t$. Multiplying a dense polynomial g by the sparse polynomial f simply accumulates t products of multiplications $g \cdot X^i$ for $i \in I_f$. Since g is represented as a bit sequence, multiplication by X^i shifts the bit sequence i-bit to the left and modulo by $X^r - 1$ moves the shifted bit segment exceeding the r-th bit to the empty bit segment starting from the 0-th bit. In other words, the multiplication simply accumulates t rotated g by $i_1, \ldots, i_t$ bits.

2.4 Polynomial Inversion with the Extended Euclidean Algorithm

The key generation (cf. Algorithm 1) computes the multiplicative inverse of a secret polynomial $h_0 \in \mathcal{R}$. Previous works, e.g., [ABB+20, HWCW19, RBMG21], computed the inversion by raising h_0 to the power of $2^r - 2$ (Fermat's little theorem).

In this work, we compute the inversion with the extended Euclidean algorithm (extGCD). The extGCD takes two input polynomials (f, g) and outputs three polynomials $(\gcd(f, g), u, v)$, where $\gcd(f, g)$ is the great common divisor of f and g and $\gcd(f, g) = u \cdot f - v \cdot g$. All polynomials are in $\mathbb{F}_2[X]$ in the context of BIKE.

In a nutshell, we compute $\text{extGCD}(X^r - 1, h_0)$ for the inverse h_0^{-1}. Under the parameters of BIKE, the polynomial $X^r - 1$ has two factors $X^r - 1 = (X - 1)(\sum_{i=0}^{r-1} X^i)$. Since $|h_0| = w/2$ is an odd number, h_0 is not a multiple of $X - 1$. Since $|h_0| \neq r$, h_0 is also

not the polynomial $\sum_{i=0}^{r-1} X^i$. Hence, the extGCD$(X^r - 1, h_0)$ outputs $(1, u, v)$ s.t. $1 = u \cdot (X^r - 1) - v \cdot h_0$, and v is the inverse h_0^{-1} since $v \cdot h_0 \equiv 1 \mod (X^r - 1)$.

However, a traditional extGCD is unsuitable for cryptographic applications because it usually contains branches that depend on the inputs. While the inputs are secret, an attacker can collect the information about the inputs through running-time differences. Hence, we have to apply a constant-time extGCD to prevent the leakage of timing side-channel information.

In this work, we adopt the constant-time version of the extGCD proposed by Bernstein and Yang [BY19]. In contrast to the traditional extGCD that eliminates the head coefficients of polynomials at any degree, the constant-time extGCD in [BY19] always eliminates the 0-th bit of polynomials. This leads to extra coefficient reversal processes for inputs to move its head coefficient to the 0-th bit position and before output for recovering the polynomial to its original coefficient order.[1] Considering for example an input polynomial f, the coefficient process is equivalently to perform the $f' \leftarrow f(1/X) \cdot X^{\deg(f)}$ operation. This operation moves the original head coefficients of f to a new position of degree 0, which is accessed by $f'[0]$. Thus the extGCD always eliminates the head coefficients at the 0-th bit.

Division Steps and Transition Matrix. In this work, we simplify the extGCD in [BY19] regarding $\mathbb{F}_2[X]$ for the BIKE application. The algorithm consists of a constant number of simple *division steps* (`divsteps`) for the two input polynomials. Define `divstep` : $\mathbb{Z} \times \mathbb{F}_2[X] \times \mathbb{F}_2[X] \to \mathbb{Z} \times \mathbb{F}_2[X] \times \mathbb{F}_2[X]$ as

$$\texttt{divstep}(\delta, f, g) = \begin{cases} (1 - \delta, g, (g(0)f - f(0)g)/X) & \text{if } \delta > 0 \text{ and } g(0) \neq 0, \\ (1 + \delta, f, (f(0)g - g(0)f)/X) & \text{otherwise.} \end{cases}$$

Here, δ means the degree difference between f and g. The `divstep` outputs two polynomials. The first polynomial aims for the polynomial of the higher degree among two input polynomials. The other is the result of subtraction of two polynomials for eliminating one head term, and it adjusts the new head term to the degree-0 coefficient by the division of X.

Since the division of X causes negative degrees, we adjust the representation of polynomials to prevent negative degrees. If the polynomial f contains a monomial of negative degree, e.g., $1/X^i$, we will store f as an alternative polynomial f' s.t. $f = f' \cdot (1/X)^i$ and degrees of all monomials of f' are non-negative. For applying `divstep` multiple times, define $(\delta_n, f_n, g_n) = \texttt{divstep}^n(\delta, f, g)$, i.e., applying the `divstep` to inputs (δ, f, g) for n times.

Bernstein and Yang describe the transition of the two polynomials (f, g) under the `divstep` operation as a matrix-vector multiplication. Let $T(\delta, f, g)$ be a 2×2 transition matrix which performs the transition $(f, g) \to (f_1, g_1)$ as matrix multiplication:

$$\begin{pmatrix} f_1 \\ g_1 \end{pmatrix} = T(\delta, f, g) \begin{pmatrix} f \\ g \end{pmatrix}, \text{ where } T(\delta, f, g) = \begin{cases} \begin{pmatrix} 0 & 1 \\ \frac{g(0)}{X} & \frac{-f(0)}{X} \end{pmatrix} & \text{if } \delta > 0 \text{ and } g(0) \neq 0, \\ \begin{pmatrix} 1 & 0 \\ \frac{-g(0)}{X} & \frac{f(0)}{X} \end{pmatrix} & \text{otherwise.} \end{cases}$$

Define the transition matrix of i-th step as $T_i = T(\delta_i, f_i, g_i)$. After n steps, the input polynomials (f, g) become

$$\begin{pmatrix} f_n \\ g_n \end{pmatrix} = T_{n-1} \cdots T_0 \cdot \begin{pmatrix} f \\ g \end{pmatrix} = \begin{pmatrix} u_n & v_n \\ q_n & w_n \end{pmatrix} \begin{pmatrix} f \\ g \end{pmatrix}.$$

[1] See [BY19, Section 6.5] for an alternative method skipping the reversal. It requests a post-process for polynomials before output.

Note that we use w instead of the original r in [BY19] to avoid the symbol conflict.

Since we aim for the polynomial inversion in BIKE, we keep only two vectors (f, g) and (v, w) in our storage space for storing all (f_i, g_i) and (v_i, w_i) for i in $0, \ldots n$, instead of tracking full transition matrices. The polynomials (f, g) and (v, w) are stored in different formats. Since $(v_i, w_i)^T$ is part of the transition matrix, they are polynomials with monomials of negative degrees. Hence, we store the vector (v_i, w_i) in a form of $(v'_i, w'_i) \cdot (1/X)^i$ and i increases with steps to keep the polynomials (v'_i, w'_i) with non-negative degrees. Since $(f_i, g_i)^T$ and $(v_i, w_i)^T$ are multiplied by the same transition matrix, we update the two vectors with similar operations except the degree adjustment. We remove the coefficient of the constant term of g for the division by X but increase the coefficients of v by one degree to keep the correct form of $(v'_i, w'_i) \cdot (1/X)^i$.

Last, we describe the overall algorithm for the polynomial inversion in BIKE. We initialize the two input polynomials $f = X^r - 1$, $g = h_0(1/X) \cdot X^r$, and their degree difference $\delta = 1$. Note that g is initialized as a bit-reversal form. The (v, w) polynomials are initialized to $(0, 1)$ as the right column of an identity matrix. Then we perform $2r - 1$ `divstep`s to update (δ, f, g) and (v, w) as well. After `divstep`s, we reverse the coefficients of the polynomial v and output it as the inverse h_0^{-1}.

3 Optimization Strategies

In this section, we propose several optimization strategies to improve the hardware implementation of BIKE. We start by describing the exchange of the symmetric cryptographic building blocks, i.e., AES-256 and SHA2-384 with a single Keccak core. Then, we introduce a new design of a multiplier exploiting the sparseness of QC-MDPC polynomials. Afterwards, we present an improved inversion module based on the algorithm proposed by Bernstein and Yang [BY19]. We conclude this section with an united hardware design which consolidates all three KEM algorithms of BIKE in one implementation.

3.1 Design Considerations

We start with our design considerations. First, our implementations utilize the framework presented in [RBMG21] while we modify and optimize several hardware modules described in the following sections. Besides these modifications, the main structure is based on the original implementation. However, we translate all modules to Verilog.

Second, we keep the same bandwidth parameter b in our modified modules as proposed in the original implementations from [RBMG21]. Hence, our design is scalable with b as well, and we benchmark our designs with the same instantiations of $b \in \mathcal{B} = \{32, 64, 128\}$. Larger b generally improve the latency of the corresponding computation since b-bit chunks of polynomials can be accessed and processed in parallel.

3.2 Random Oracles

The BIKE team recently updated the random oracles **H**, **K**, and **L** in their latest specification of version 4.2 [ABB+21]. They adapted the core components of these functions from AES256 and SHA2-384 to SHAKE256 and SHA3-384 with an unified Keccak core, respectively. While Richter-Brockmann et *al.* suggested the unified symmetric core would be beneficial for a hardware implementation, they, however, did not test their suggestions in [RBMG21].

In this work, we modify the implementations presented in [RBMG21] to the latest specification of hash functions and report the comparisons in Section 4.1. Therefore, we implement a simple Keccak core which only contains the round function and a controlling

interface. In the following, we describe the implementations of wrappers that are connected to the Keccak core and form the random oracles.

First, for the **H** function, we instantiate a SHAKE256 from the Keccak's round function. As in Algorithm 2, **H** uses a 256-bit message m as seed for SHAKE256 which is requested by an dedicated interface in our implementation. Then, with correct padding and controlling of the Keccak core, the wrapper divides the 1 088 output bits into 32-bit chunks. The integrated sampler uses the chunks to generate the indexes of error polynomials (e_0, e_1) and rejects illegal samplings. If the sampler has consumed all randomness, the wrapper initiates an additional squeezing phase of SHAKE256.

Second, for generating the private key (h_0, h_1) in the key generation (cf. Algorithm 1), our wrapper operates similarly to the **H** function besides different Hamming weights.

Third, for the **L** function, the wrapper uses the error polynomials (e_0, e_1) and provides them in the absorbing phases to the Keccak core. In this case, it performs a SHA3-384 hashing operations. Besides the correct padding, the wrapper ensures to concatenate the error polynomials by eight-bit blocks. Last, it truncates the 384-bit hash value to a 256-bit value and adds it to m.

Fourth, our wrapper for the **K** function is realized similarly to the **L** function. However, the input to the SHA3-384 slightly differs since a 256-bit string needs to be concatenated with an r-bit polynomial and with another 256-bit string. Nevertheless, it truncates the 384-bit output to 256 bits in the same way.

3.3 Sparse Polynomial Multiplier

In this section, we present the hardware design of the sparse polynomial multiplier for BIKE. In 2019, Hu et *al.* [HWCW19] already applied the approach of sparse multiplications to BIKE. However, compared to their design, our optimized implementation achieves a better area-time product and reduces the latency (for detailed information see Section 4.2). Additionally, our design keeps the time-constancy for the encapsulation while computing $e_0 + e_1 \cdot h$ with the indefinite Hamming weight of e_1.

As in Section 2.3, given a multiplication $p_{\text{res}} = p_{\text{sparse}} \cdot p_{\text{arb.}}$ where $p_{\text{sparse}}, p_{\text{arb.}} \in \mathcal{R}$ and $|p_{\text{sparse}}| \ll r$. Further, the polynomial p_{sparse} is represented as a set of indexes of non-zero terms and $p_{\text{arb.}}$ is a r-bit sequence divided into $\lceil \frac{r}{b} \rceil$ chunks. Then, we conduct the multiplication by reading the non-zero indexes of p_{sparse}, rotating $p_{\text{arb.}}$ by the indexes to the left, and accumulating the rotated results to the product p_{res}.

General Sparse Multiplier. Figure 1 shows a simplified architecture of the general sparse multiplier which iterates over the indexes of the sparse polynomial p_{sparse}. Each iteration is initiated by reading a non-zero index from p_{sparse}. Meanwhile, it starts to access the values of the polynomial $p_{\text{arb.}}$ in an ascending order starting at the second uppermost address, proceeding with the uppermost, and then keep going from address zero. This procedure simplifies to deal with the most-significant bits in $p_{\text{arb.}}$ since $r \bmod b \neq 0$ (r is always prime). Figure 1 neglects the hardware to deal with this exception (mostly multiplexers) for clarity.

While processing a particular index from p_{sparse}, the lower $\log b$ bits of the index determine the number of bits to shift the input from $p_{\text{arb.}}$ to the left. The shifted output is added to the current intermediate result depicted by the xor-gates in Figure 1. We instantiate two memories to store the intermediate results of the multiplication. This allows us to read the current intermediate result from one memory and write the new result to the other one in the same clock cycle.

The upper part of the schematic in Figure 1 determines the addresses for both memories. When an index of the sparse polynomial is read, the upper bits are sampled in a register used as initial value for a counter. To handle the jump from the highest address (i.e., $\lfloor \frac{r}{b} \rfloor$) to zero, our final design contains slightly more logic. Again, Figure 1 neglects this logic

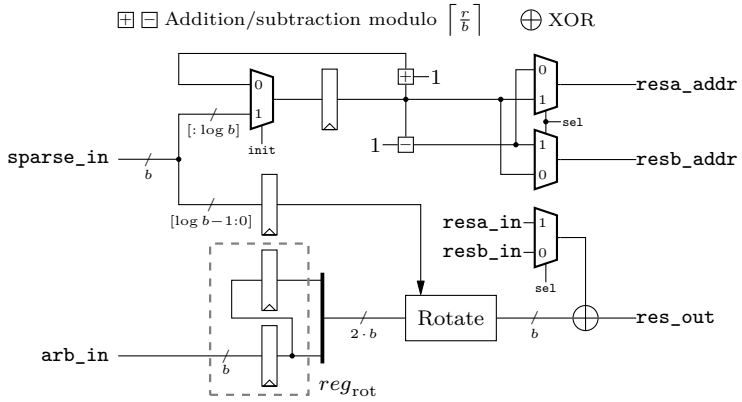


Figure 1: Schematic architecture of the general sparse multiplier.

for the sake of clarity. However, the output of the counter is subtracted by one, and two multiplexers decide which of the address values are used to access which of the memories. The decision signal `sel` is determined based on the LSB from the address counter used to read out the indexes of the sparse polynomial.

For each index of the sparse polynomial, our multiplier spends $\lceil \frac{r}{b} \rceil + 4$ clock cycles for shifting and accumulating the intermediate results. The total latency is given by

$$L_{\text{mult}}(th) = \left(\left\lceil \frac{r}{b} \right\rceil + 4 \right) \cdot th + 1 \tag{1}$$

where th denotes the weight of the sparse polynomial (e.g., for the key generation in BIKE $th = \frac{w}{2}$). The circuit switches to the `DONE` state in the additional clock cycle.

This design iterates over a fixed number of indexes of the sparse polynomial. While this approach is capable of processing the secret polynomials (h_0, h_1), it cannot process the multiplication $e_1 \cdot h$ in the encapsulation with a constant latency since the Hamming weight of e_1 is unknown. Therefore, we modify the design of the general sparse multiplier into a dedicated multiplier for BIKE in the next paragraph.

Tailored Constant-time Multiplier for BIKE. To deal with the indefinite weight of e_1 in the encapsulation, we utilize the relation $|e_0| + |e_1| = t$ defined by BIKE. It allows to rephrase the encoding operation as an addition of two multiplications

$$c_0 = e_0 \cdot 1 + e_1 \cdot h. \tag{2}$$

For computing c_0, we modify the general sparse multiplier introduced above and add a multiplexer choosing h or 1 as input for $p_{\text{arb.}}$ depending on e_0 or e_1. To indicate whether e_0 or e_1 is processed, we add an additional leading bit to the indexes and set the MSB of the indexes belonging to e_0 to '1'. We embed this operation directly into the sampling function **H**. Hence, the multiplexer selects its output according to the MSB of the indexes of the sparse polynomial.

In order to illustrate the two modes of the multiplication engine, we provide a small example for $r = 11$, $b = 4$, and $p_{\text{arb.}} = X^{10} + X^8 + X^7 + X^6 + X^5 + X^4 + X^3 + 1 =$ `101 1111 1001` (corresponds to h in Equation 2). For the error polynomials, we exemplary assume $e_0 = X^5$ and $e_1 = X^7$ and their corresponding indexes $e_{0,\text{idx}} =$ `1 0101` and $e_{1,\text{idx}} =$ `0 0111`, respectively. For both modes, we assume that the current intermediate result is $p_{\text{int}} =$ `010 1001 0110`.

Figure 2 visualizes the multiplication $e_1 \cdot p_{\text{arb.}}$ where each dashed line separates the data flow between the clock cycles. In this case, the expected result is

$$X^7 \cdot \texttt{101 1111 1001} \oplus \texttt{010 1001 0110} = \texttt{100 1101 1111} \oplus \texttt{010 1001 0110} = \texttt{110 0100 1001}.$$

$p_{\text{arb.}}$ in:	1111	0101	1001	1111	0101	
reg_{rot} in:	0001 0000	1011 0000	1001 1011	1111 1001	0101 1111	
reg_{rot} out:	0000 0000	0001 0000	1011 0000	1001 1011	1111 1001	0101 1111
shifted:	0000 0000	1000 0000	1000 0000	1101 1000	1100 1000	1111 1000
p_{int} in:				1001	0010	0110
result out:				0100	0110	1001

index of the sparse polynomial: 0 0111

write to addr 0x01 | write to addr 0x02 | write to addr 0x00

Figure 2: Example for a multiplication with an index from e_1.

p_{one} in:	0000	0000	0001	0000	0000	
reg_{rot} in:	0000 0000	0000 0000	0001 0000	0000 0001	0000 0000	
reg_{rot} out:	0000 0000	0000 0000	0000 0000	0001 0000	0000 0001	0000 0000
shifted:	0000 0000	0000 0000	0000 0000	0010 0000	0000 0010	0000 0000
p_{int} in:				1001	0010	0110
result out:				1011	0010	0110

index of the sparse polynomial: 1 0101

write to addr 0x01 | write to addr 0x02 | write to addr 0x00

Figure 3: Example for a multiplication with an index from e_0.

As described above, the module first reads the second uppermost chunk from the input polynomial which is 1111 in our example. Since $r = 11$ and $b = 4$, only the most significant bit from this chunk is required and stored in the register reg_{rot} (cf. Figure 1). The remaining three bits are taken from the uppermost chunk. Afterwards, the process proceeds in a regular pattern by reading a new chunk and moving the old chunk to the lower part of reg_{rot}. The multiplier determines the starting address to read the first chunk from the intermediate result by the upper bits of the error index, i.e., 0x01 in our example. This describes the required shift on word level. The output of the register is shifted to the left by 3 bit which are the least $\log(b)$ bits from index $e_{1,\text{idx}}$ and describe the required shift on bit level. Hence, the first chunk of the new intermediate result is written to address 0x01. Note, when the multiplier writes the result to address 0x02, the most significant bit is set to 0 since it does not belong to a valid polynomial of size $r = 11$.

The procedure for a multiplication with the index $e_{0,\text{idx}}$ is similar. Instead of providing the polynomial $p_{\text{arb.}}$ to the multiplier, the polynomial $p_{\text{one}} = 1 = 000\ 0000\ 0001$ is selected by the most signification bit of $e_{0,\text{idx}}$. The corresponding data flow is visualized in Figure 3. It is clearly visible that the multiplication with an index from e_0 requires the same amount of clock cycles such that a constant-time operation is guaranteed.

To this end, Figure 4 shows the adjustment for processing the operand $p_{\text{arb.}}$ in the multiplier. Note, the polynomial of one does not require an extra memory but is generated on the fly. While accessing the 0-th chunk of $p_{\text{arb.}}$, the circuit feeds a b-bit chunk of 0...01 to the multiplexer. Otherwise, the multiplexer always gets a zero b-bit chunk. Hence, the multiplier always finishes the multiplication from Equation 2 in $L_{\text{mult}}(t)$ clock cycles.

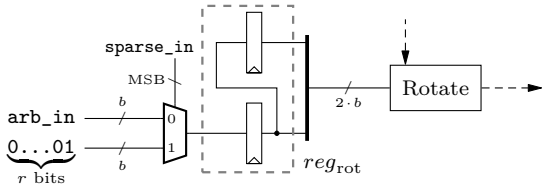


Figure 4: Modifications to the input operand of the tailored multiplier.

Last, we add an additional input to the multiplier design determining the number of non-zero indexes of the sparse polynomial for the two possible weights $(t, w/2)$ of the sparse input polynomials.

3.4 Polynomial Inversion

We present our hardware design and optimization for the polynomial inversion in this section. In 2020, Marotzke [Mar20] had reported an implementation for the polynomial inversion required in NTRU Prime, a post-quantum KEM. The inversion module utilizes Bernstein and Yang's extGCD algorithm [BY19] optimized to perform inversions of polynomials of degree 760 with coefficients in prime fields, where the arithmetic takes place in Digital Signal Processor (DSP) units. Since our design targets to invert polynomials in $\mathcal{R}$ with large degrees (i.e., $\geq 12\,323$), the two implementations pursue different purposes and are not directly comparable.

In the following, we first divide the computation of `divstep` into two subroutines. Then, we introduce the main framework of the inversion and the two subroutines followed by our hardware designs.

Performing the `divstep`. Recalling Section 2.4, an extGCD for polynomial inversion computes $2r - 1$ `divsteps`. In [BY19], based on the shape of the transition matrix, Bernstein and Yang optimized the multiplication by the transition matrix in a single `divstep` as two simple functions:

1. a conditional swap: replacing (δ, f, g) with $(-\delta, g, f)$ if $\delta > 0$ and $g(0) \neq 0$.
2. an elimination: replacing (δ, f, g) with $(1 + \delta, f, (f(0)g - g(0)f)/X)$.

Since the head coefficient $f(0)$ is always one for computing the inversion in BIKE, we need only two information bits deduced from $(\delta, g(0))$ in each `divstep` as instructions for updating (f, g) and (v, w). The first bit indicates the swap operation and the second bit is $g(0)$ used in the elimination operation. We refer to the two information bits as *control bits* of one `divstep` in this paper. Furthermore, we split one `divstep` into two operations:

1. `get_control_bits()`: calculates the control bits based on the values of δ and the necessary coefficients of the polynomials (f, g), and
2. `update_fg_or_vw()`: updates the polynomials (f, g) and (v, w) based on the computed control bits.

Main Framework. Algorithm 4 describes the main framework of the polynomial inversion. As introduced above, the algorithm uses four temporary polynomials f, g, v, and w while g is initialized with the bit-reversed input polynomial g_{in}. The main parts of the algorithm are $2r - 1$ `divsteps`, which are decoupled into series of `get_control_bits()` and `update_fg_or_vw()` subroutines. Last, the algorithm shifts v one bit to the right, reverses its coefficients, and returns v as the inverse of the input polynomial.

Algorithm 4: Main framework for the polynomial inversion.

Input : Input polynomial g_{in} and step size s.
Output : Inverted polynomial $g_{\text{out}} = g_{\text{in}}^{-1}$

1 $N \leftarrow \lceil \frac{r}{b} \rceil$;
2 $f[N], g[N], v[N], w[N] \leftarrow 0$; // Initialize polynomials (arrays)
3 $w[0] = 1$; $f[0] = 1$; $f[N-1] \leftarrow 2^{r \bmod b}$;
4 $g \leftarrow$ `bitreverse` (g_{in}); // Reverse the bits of the input polynomial
5 $\delta \leftarrow 1$; // Degree difference of polynomials f and g
6 $\tau \leftarrow 2r - 1$; // Number of divsteps to be executed
7 **while** $\tau \geq s$ **do**
8 $\quad \delta, c \leftarrow$ `get_control_bits`$(\delta, f[0], g[0], s)$;
9 $\quad$ **for** $j = 0$ **to** N **do**
10 $\quad\quad f_0, f_1 \leftarrow f[j], ((j+1) > N \ ? \ 0 : f[j+1])$;
11 $\quad\quad g_0, g_1 \leftarrow g[j], ((j+1) > N \ ? \ 0 : g[j+1])$;
12 $\quad\quad f[j], g[j] \leftarrow$ `update_fg_or_vw`$(c, f_1, f_0, g_1, g_0, s, 1)$;
13 $\quad$ **end**
14 $\quad$ **for** $j = N$ **to** 0 **do**
15 $\quad\quad v_0, v_1 \leftarrow (j == 0 \ ? \ 0 : v[j-1]), v[j]$;
16 $\quad\quad w_0, w_1 \leftarrow (j == 0 \ ? \ 0 : w[j-1]), w[j]$;
17 $\quad\quad v[j], w[j] \leftarrow$ `update_fg_or_vw`$(c, v_1, v_0, w_1, w_0, s, 0)$;
18 $\quad$ **end**
19 $\quad \tau \leftarrow \tau - s$;
20 **end**
21 **if** $\tau > 0$ **then**
22 $\quad \delta, c \leftarrow$ `get_control_bits`$(\delta, f[0], g[0], \tau)$;
23 $\quad$ **for** $j = N$ **to** 0 **do**
24 $\quad\quad v_0, v_1 \leftarrow (j == 0 \ ? \ 0 : v[j-1]), v[j]$;
25 $\quad\quad w_0, w_1 \leftarrow (j == 0 \ ? \ 0 : w[j-1]), w[j]$;
26 $\quad\quad v[j], w[j] \leftarrow$ `update_fg_or_vw`$(c, v_1, v_0, w_1, w_0, s, 0)$;
27 $\quad$ **end**
28 **end**
29 $v \leftarrow$ `shift_right`(v); // Shift one bit to the right
30 **return** `bitreverse` (v);

In the algorithm, we introduce a parameter s to control the *step size*, allowing to proceed s `divstep`s in each iteration in parallel (cf. line 7). The `get_control_bits()` and the `update_fg_or_vw()` take the parameter as well and proceed s steps accordingly. Therefore, the subroutine `get_control_bits()` determines $2s$ control bits and updates δ based on the state of $(\delta, f[0], g[0])$. Afterwards, a loop iterates over all four polynomials f, g, v, and w and updates them by `update_fg_or_vw()` for s steps in each call. Starting from line 22, the algorithm covers the remaining steps and updates only (v, w) accordingly.

Besides the step size s, the execution time of Algorithm 4 scales with the bandwidth parameter b as well. Enlarging b decreases the number of chunks N and therefore, less numbers of iteration are executed in the inner loop since `update_fg_or_vw()` updates one chunk in each execution. In our design, the choice for s is also limited by $s \leq b$ since `get_control_bits()` takes inputs of one polynomial chunk only. We describe the details of `get_control_bits()` and `update_fg_or_vw()` in the following paragraphs.

Determining Control Bits. Algorithm 5 details the process of `get_control_bits()`. The algorithm takes four inputs, which are the degree difference δ, two b-bit chunks

Algorithm 5: Compute control bits.

Input : Current δ, $f[0]$, $g[0]$, and the step size s.
Output : Updated δ and an array of control bits $c[2s]$

$f, g \leftarrow f[0], g[0]$;
for $i = 0$ **to** $s - 1$ **do**
  $swap \leftarrow ((-\delta < 0)\ ?\ 1 : 0)\ \&\ (g \wedge 1)$;
  $\alpha \leftarrow g \wedge 1$;
  $c[i \cdot 2] \leftarrow swap$;
  $c[i \cdot 2 + 1] \leftarrow \alpha$;
  $\delta \leftarrow swap\ ?\ -\delta + 1 : \delta + 1$;
  $f, g \leftarrow (swap\ ?\ g : f), (g \oplus (f \cdot \alpha))/2$;
end
return δ, c;

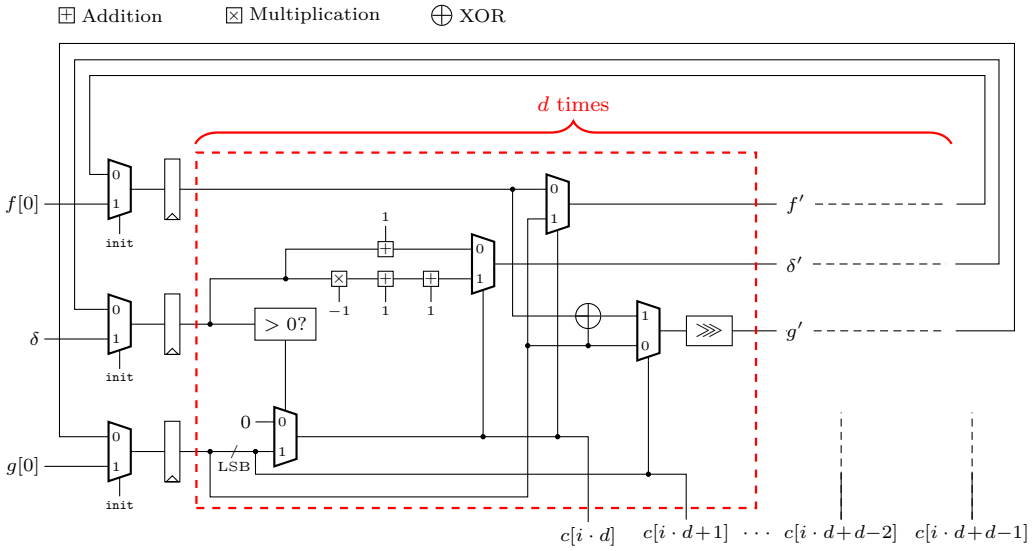


Figure 5: Hardware design for the computation of the control bits.

$(f[0], g[0])$ from the polynomials (f, g), and the step size s. The algorithm outputs the updated δ and $2s$ control bits c for s `divsteps`. For generating the control bits for the s `divsteps`, the algorithm uses only s bits from each input polynomial instead of the full coefficients. Note, however, the algorithm is a sequential process where the control bits of iteration i depends on the results of the previous iterations.

Our hardware design for `get_control_bits()` incorporates this characteristic such that we aim to fully utilize the computational capacity and hence execute d iterations of the loop shown in Algorithm 5 in one clock cycle. Therefore, Figure 5 shows a schematic draft of this approach where one iteration is highlighted by the red dashed border. For larger step sizes s, however, unrolling the whole loop in a hardware implementation would result in a long critical path. Hence, we introduce a round-based circuit that is executed $\lceil \frac{s}{d} \rceil$ times since $d \cdot \lceil \frac{s}{d} \rceil \geq s$. We store the generated control bits in registers to use them immediately for updating the polynomials (f, g) and (v, w) by `update_fg_or_vw()`.

Updating Polynomials. We summarize the details of `update_fg_or_vw()` in Algorithm 6. The algorithm expects as inputs the control bits c, two $2b$-bit chunks of the polynomials (f, g) or (v, w), the step size s, and one bit specifying whether the input chunks originating

Algorithm 6: `update_fg_or_vw()`

Input : Control bits c, two $2b$-bit chunks of f and g, and the step size s .
Output : Updated b-bit chunks r_0 and r_1

for $i = 0$ **to** $s - 1$ **do**
  $f, g \leftarrow (c[i \cdot 2] \ ? \ g : f), (c[i \cdot 2 + 1] \ ? \ g \oplus f : g)$;
  if *is_updating_fg* **then**
    $g \leftarrow g/2$; `// Shift right, i.e., dividing by` X
  else
    $f \leftarrow f \cdot 2$; `// Shift left, i.e., multiplying by` X
  end
end
if *is_updating_fg* **then**
  $r_0, r_1 \leftarrow f[0 : b], g[0 : b]$; `// lower` b `bits`
else
  $r_0, r_1 \leftarrow f[b : 2b], g[b : 2b]$; `// higher` b `bits`
end
return r_0, r_1;

from the pairs (f, g) or (v, w). The algorithm updates the given chunks for s `divsteps` according to the control bits c. Since (f, g) and (v, w) are multiplied by the same transition matrix in the same `divstep`, the arithmetic for updating the polynomials is identical. The different formats of storing polynomials (see Section 2.4) cause the difference of the two operating modes, which shift polynomials in different directions and output different chunks of polynomials.

Figure 6 shows our hardware design for updating the polynomials (f, g). The *basic block* (highlighted by the red dashed border) updates the polynomials for one `divstep`, consisting of simple shifts, an addition (xor), and multiplexing operations. The whole submodule can finish the computation with s consecutive basic blocks which, however, would result in a long critical path without any further modifications. Therefore, to control the length of the critical path, we introduce pipeline registers after u basic blocks. Hence, there are $\lfloor \frac{s}{u} \rfloor$ pipeline stages in the module. Note, we implement a similar module to update (v, w).

Although, Figure 6 depicts two full b-bit chunks for each input associated with the different polynomials, the algorithm actually only requires $b + s$ bits of data from the input polynomials. The algorithm inputs the $2b$-bit chunks because it accesses polynomials in chunks of b bits from the memory. However, the module only instantiates logic for processing $s + b$ data such that no area overhead occurs.

Overall Design of the Polynomial Inversion. The entire polynomial inversion module consists of two counters controlling the reversion of the bits and the final right shift (cf. Algorithm 4). Additionally, we instantiate `get_control_bits()` and two versions of `update_fg_or_vw()` (updating (f, g) and (u, w) in parallel) as described above. Since the algorithm works on four temporary polynomials, the inversion module utilizes eight Block-RAMs (BRAMs) allowing to read and write the intermediate results in the same clock cycle. Nevertheless, the latency of the proposed design depends on several parameters, i.e., r, b, s, d, and u. It is determined by

$$L_{\text{inv}}(s, d, u) = \underbrace{\lambda \cdot \left(3 + \left\lceil \frac{s}{d} \right\rceil + \left\lceil \frac{s}{u} \right\rceil + \left\lceil \frac{r}{b} \right\rceil\right)}_{\text{main computation}} + \underbrace{\rho + \left\lceil \frac{s}{d} \right\rceil + \left\lceil \frac{r}{b} \right\rceil}_{\text{remainder}} + \underbrace{3 \cdot \left\lceil \frac{r}{b} \right\rceil + 13}_{\text{bitreverse \& shift}} \quad (3)$$

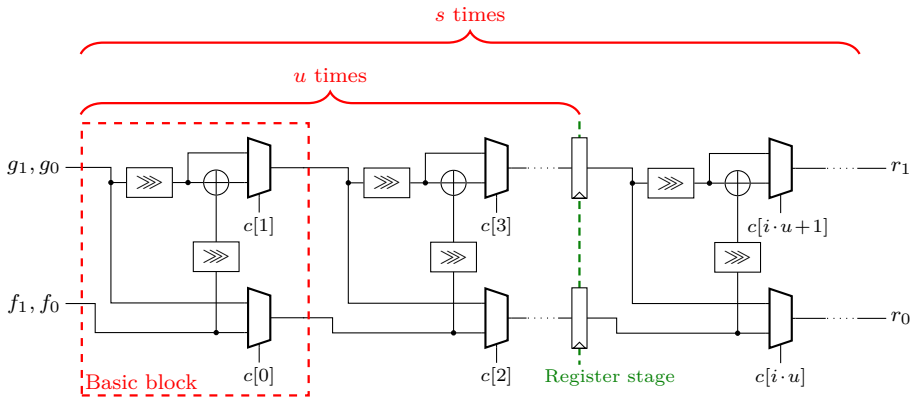


Figure 6: Hardware implementation of the update process for the f and g polynomial.

where $\lambda = \left\lfloor \frac{2 \cdot r - 1}{s} \right\rfloor$ and $\rho = \left\lceil \frac{2 \cdot r - 1 - \lambda \cdot s}{u} \right\rceil$. Note, our design for $s = 1$ does not follow Equation 3 since it is a handcrafted and optimized design which achieves a slightly smaller latency and requires only seven BRAMs instead of eight.

3.5 United Hardware Design

Given the optimized modules for the polynomial arithmetic and the modifications for the random oracles, we now present an *united hardware design* of BIKE consolidating the key generation, encapsulation and decapsulation in one module. Such a design allows to share resources between the different KEM operations. For example, we only instantiate one single multiplier, one KECCAK core with the corresponding wrappers described in Section 3.2, and a limited number of BRAM modules. The number of required BRAMs is given by the decapsulation since its implementation utilizes the most memories (cf. [RBMG21]). However, this design decision implies that only one of the three KEM algorithms of BIKE can be executed at the same time. Therefore, we implement a control interface that allows to enable the desired algorithm by a three bit instruction, load and read data (polynomials and 256-bit strings), and request randomness used as seed for the PRNG. A top-level draft of this implementation is shown in Figure 7. While all building blocks that are used by more than one KEM algorithms are marked by a green border, the black modules are only required for a single KEM operation (the inversion module and sampler are used only in the key generation, and the BFIter module together with the Hamming weight and threshold computation only in the decapsulation).

The Finite-State Machine (FSM) on the right side manages all input/output operations and the control flow of the three KEM algorithms. The input interface expects a six-bit instruction identifying which data should be loaded. For the key generation no initial data is required. The encapsulation requires the public key h which needs to be loaded to a BRAM before the computation can be started. To perform a decapsulation, the implementation assumes that the user load the two parts of the cryptogram (c_0, c_1), the two polynomials of the private key (h_0, h_1), and σ. The output interface returns the same data and additionally the shared key K. After the required data has been accessed, all memories are reset by overwriting the content with zero.

4 Implementation Results

In this chapter, we evaluate the proposed optimizations and modifications for a hardware implementation of BIKE. First, we show that the modifications of the random oracles are

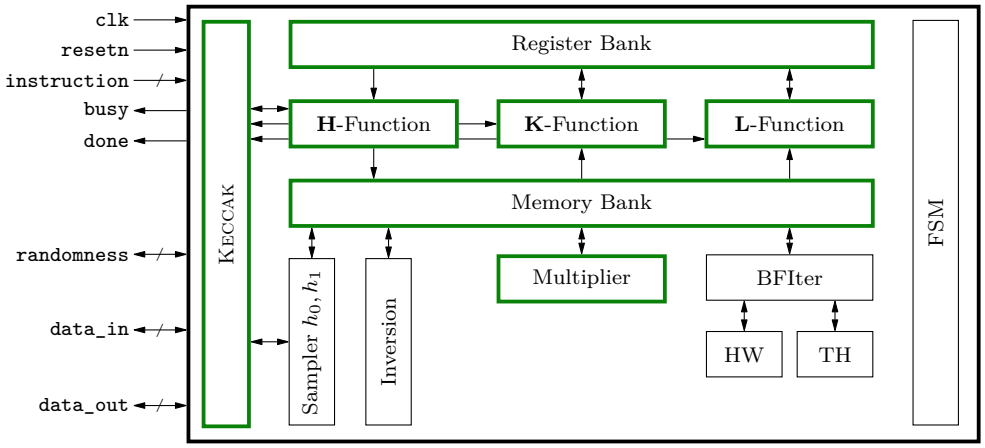


Figure 7: Top-level view of the united hardware design.

beneficial for a hardware design of BIKE. Second, we report implementation results for the proposed sparse multipliers and compare them to designs from the literature. Third, we demonstrate the scalability of our inversion module by presenting implementation results for different configurations. Fourth, since both – the multiplication and inversion – influence the footprint and performance of the key generation, we provide dedicated implementation results for a stand-alone key generation design. Fifth, we present the implementation results of the united hardware design and compare it to other implementations of code-based PQC schemes. We generate all results for an Artix-7 XC7A200T FPGA manufactured by Xilinx.

4.1 New Random Oracles

As described in Section 3.2, BIKE's new specification [ABB+21] updates the random oracles from AES-256 and SHA2 to an unified Keccak core. To test how the design choice of cryptographic primitives effects the performance of hardware implementations, we compare the implementations of the original VHDL code[2] from [RBMG21] with our adapted version applying the new specification with a replaced Keccak core. We performed no other optimizations for a fair comparison.

Table 2 reports the comparisons for the encapsulation and decapsulation. For both KEM algorithms and all hardware configurations, the adapted versions achieve slightly better results in terms of area and latency. Especially the number of required registers decreases by roughly 880 in the adapted implementation for all designs. To this end, these implementation results show that the modifications of the random oracles are indeed beneficial for hardware implementations of BIKE.

4.2 Multiplier

Table 3 shows the implementation results for our two multiplier designs configured for the lowest security level of BIKE, i.e., for $r = 12\,323$. The first design is the general sparse multiplier where the sparse polynomial always has a fixed Hamming weight, i.e., the Hamming weight is determined before synthesis. In BIKE, such cases occur in the key generation and decapsulation where $|p_{\text{sparse}}| = w/2$. The second design reads the Hamming weight of the sparse polynomial via an input interface. Hence, it can be used for all multiplications required in BIKE. Additionally, the design performs the encoding

[2]The authors published their code at https://github.com/Chair-for-Security-Engineering/BIKE/

Table 2: Comparison of KEM functions w.r.t. different random oracle settings ($r = 12\,323$).

	Resources					**Performance**		
	Logic		Memory		Area	Cycles	Freq.	Latency
b	LUT	DSP	FF	BRAM	Slices	Cycles	MHz	ms
Encapsulation with adapted random oracles								
32 bit	6 604	0	2 409	3	1 906	151 587	121.95	1.24
64 bit	8 388	0	2 444	5	2 408	39 264	121.95	0.32
128 bit	15 135	0	2 625	10	4 268	11 136	119.05	0.094
Encapsulation of the previous specification from [RBMG21]								
32 bit	6 730	0	3 298	3	2 143	152 694	121.95	1.25
64 bit	8 253	0	3 327	5	2 538	40 368	121.95	0.33
128 bit	14 829	0	3 471	10	4 540	12 240	121.95	0.10
Decapsulation with adapted random oracles								
32 bit	9 070	7	3 055	10	2 570	1 624 402	125	13
64 bit	14 011	9	3 415	15	3 933	515 823	116.28	4.44
128 bit	29 697	13	4 170	29	8 234	186 364	100	1.86
Decapsulation of the previous specification from [RBMG21]								
32 bit	9 380	7	3 943	10	2 971	1 626 674	125	13.01
64 bit	16 140	9	4 307	15	4 942	518 105	116.28	4.46
128 bit	30 430	13	5 063	29	8 785	188 646	100	1.89

in the encapsulation in constant time. To this end, the hardware utilization is slightly higher as for the general sparse multiplier. Note, for the second multiplier design, we report performance numbers for the multiplication performed in the encapsulation, i.e., $|p_{\text{sparse}}| = t = 134$. The number of clock cycles for different Hamming weights follows Equation 1.

Table 3 also lists the results of the schoolbook-based (dense) multiplier from [RBMG21] and of the sparse multiplier design from [HWCW19]. Since the authors of [RBMG21] only reported implementation results for $r = 10\,163$, we extracted the multiplier from their code and synthesized it for $r = 12\,323$. As expected, the sparse multiplier clearly outperforms the schoolbook-based design with respect to area. For a fixed Hamming weight of 71, the sparse multiplier also achieves better performance results. However, for $b = 128$ the schoolbook multiplier achieves slightly better performance results than the tailored sparse multiplier which it trades with a huge area footprint. Therefore, the sparse multiplier is clearly superior with respect to the Area-Time (AT) product.

Compared to the multiplier from [HWCW19], our design achieves a considerably lower latency albeit our results were generated for a larger parameter set. Our design mainly differentiates from their implementation in two parts. First, we decided to instantiate two memories to store the intermediate results of the multiplication's product. This allows us to perform a read and write access in the same clock cycle while the implementation by Hu et *al.* requires two clock cycles. Note, for Xilinx FPGAs one could exploit the *read-then-write* option allowing to perform a read and write access in the same clock cycle to the same address reducing the amount of required BRAM modules. However, we decided not to use this option but rather instantiate two memories since it is a more generic approach which is universally applicable to other hardware devices as well. Second, our rotation unit performs the whole rotation within one clock cycle while the design by [HWCW19] requires $\lceil \log b \rceil$ clock cycles. Even though our multiplier architectures consume slightly more slices, it clearly improves the AT product.

We also tried to compare our results to the design proposed in [BFG+19] but we were

Table 3: Comparison of sparse polynomial multipliers for $r = 12\,323$.

	Resources				Performance		
	Logic	Memory		Area	Cycles	Frequency	Latency
b	LUT	FF	BRAM	Slices	Cycles	MHz	μs
General sparse multiplier ($\|p_{sparse}\| = w/2 = 71$)							
32	319	127	2	132	27 691	234.36	118.16
64	549	190	4	197	13 988	222.22	62.94
128	1 136	381	8	378	7 172	184.95	38.78
Tailored sparse multiplier for BIKE ($\|p_{sparse}\| = t = 134$)							
32	349	135	2	151	52 261	238.15	219.5
64	629	204	4	245	26 399	222.52	118.8
128	1 249	386	8	437	13 535	184.3	73.44
Sparse multiplier from [HWCW19] ($r = 10\,163$, $\|p_{sparse}\| = 71$)							
32	–	–	2	100	158 614	240	660.89
64	–	–	3	157	90 880	220	413.09
128	–	–	5	292	51 688	210	246.13
Dense polynomial multiplier from [RBMG21]							
32	697	105	1.5	220	150 155	201.37	745.67
64	2 595	137	3	864	37 829	173.82	230.91
128	9 539	293	6	3 332	9 701	183.66	52.82

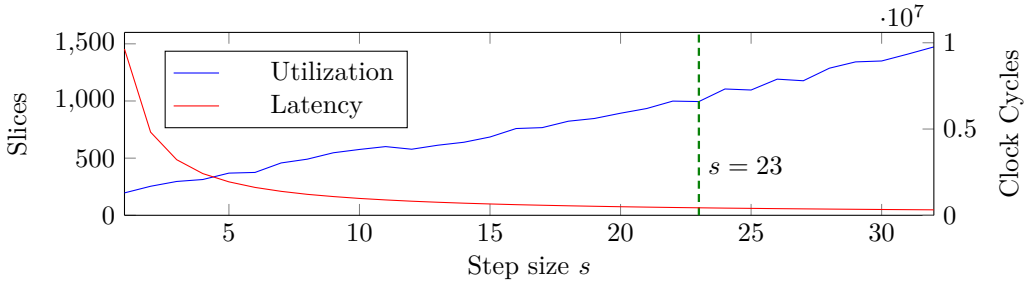
not able to figure out which value the authors applied for the parameter BW (corresponds to our bandwidth parameter b) so that a fair comparison is difficult. However, we assume that their design is similar to our multiplier design which uses fixed Hamming weights.

4.3 Inversion Module

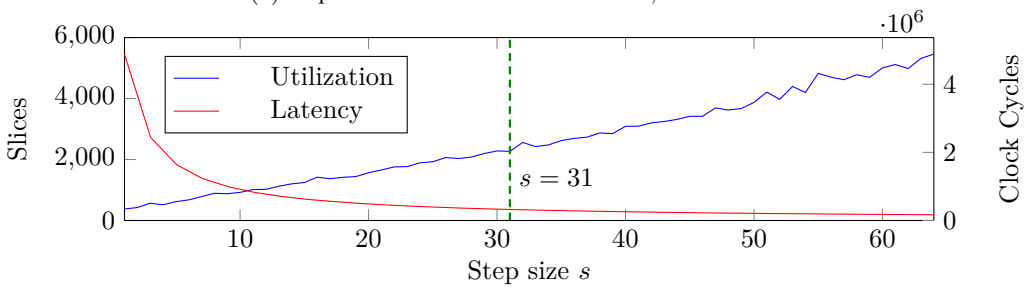
In this section, we first evaluate the polynomial inversion module described in Section 3.4 for $b \in \mathcal{B}$ and for $r = 12\,323$ and compare our approach afterwards to the design from [RBMG21] which is based on Fermat's little theorem. Note, in all experiments we fix the maximum number of basic blocks instantiated between two register stages for the updating process of (f, g), and (v, w) to $u = 8$ achieving a critical path that is smaller than 10 ns. Additionally, we generate all results in this subsection for a target frequency of 100 MHz.

Detailed Evaluation of the Inversion Module Figure 8a shows the number of required slices and the latency in clock cycles for $b = 32$, $1 \leq s \leq 32$, and $d = 2$. The area footprint linearly increases with the step size parameter s while the number of clock cycles follows Equation 3. Moreover, we include the configuration for the best AT product (slices $\times$ cycles$/10^6$) visualized by the green dashed line. The configuration for $s = 23$ achieves the best result with an AT product of 432. A more detailed evaluation of the implementations can be found in the appendix in Table 7.

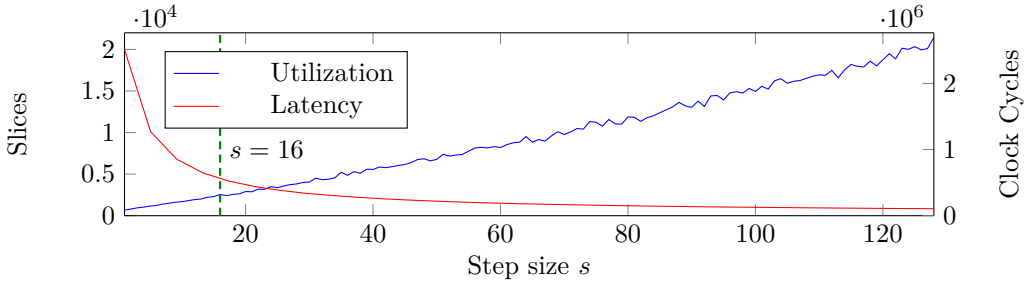
Figure 8b shows the implementations results for different step sizes s for $b = 64$. The trends for the required clock cycles and for the area utilization are very similar to the configurations for $b = 32$. The smallest configuration requires 4 880 299 clock cycles but only consumes 377 slices while the fastest design performs one inversion within 91 678 clock cycles by consuming 5 457 slices. The design with the best AT product is obtained for $s = 31$ (a detailed evaluation can be found in the appendix in Table 8).



(a) Implementation results for $b = 32$, and $d = 2$.



(b) Implementation results for $b = 64$ and $d = 2$.



(c) Implementation results for $b = 128$ and $d = 1$.

Figure 8: Implementation results for the polynomial inversion for a Xilinx Artix-7 FPGA and a target frequency of 100 MHz setting $r = 12\,323$. The green dashed lines indicate the configurations with the best area-time product.

The implementation results for $b = 128$ are plotted in Figure 8c where the best AT product is obtained for $s = 16$. To achieve reasonable critical paths (maximum possible frequency larger than 100 MHz), we reduce the number of unrolled rounds to compute the control bits c to $d = 1$. With $s = 128$ we can instantiate our fastest inversion module which finishes one polynomial inversion in only 47 386 clock cycles. However, the implementation costs drastically increase to 21 435 slices. Again, a detailed evaluation is given in the appendix in Table 9 and Table 10.

Comparison to Related Work. We compare our inversion module to the approach presented in [RBMG21] which is based on Fermat's little theorem in Table 4. The corresponding numbers are extracted from their implementation of the key generation.

With Fermat's little theorem, given a $g \in \mathcal{R}$, [RBMG21] computes the inverse as $g^{2^{r-1}-1}$. To efficiently raise the degree of g, they used a square-and-multiply chain from the Itoh-Tsujii Algorithm (ITA) [IT88] achieving a latency of

$$L_{\text{inv-Fermat}} \approx \log(r) \cdot (r + L_{\text{school}}) + |r_{\text{bin}}| \cdot \left(\left\lceil \frac{r}{b} \right\rceil + L_{\text{school}} \right) \tag{4}$$

where $r_{\text{bin}} = r - 2$ and $L_{\text{school}} = \lceil \frac{r}{b} \rceil \cdot (\lceil \frac{r}{b} \rceil + 3) + 1$. Note, Equation 4 describes just an approximation of the required clock cycles since the implementation from [RBMG21] is highly optimized to the use-case of BIKE. However, compared to the dominant term $\lfloor \frac{2 \cdot r - 1}{s} \rfloor \cdot \lceil \frac{r}{b} \rceil$ from Equation 3, our inversion module has an extra parameter s, allowing to achieve more optimized configurations.

In Table 4, we present results for the light-weight ($s = 1$) and high-speed ($s = b$) configuration as well as the design with the best area-time product. For comparison with the area cost, we report a configuration targeting the number of clock cycles of the approach from [RBMG21]. While finishing the inversion with the same amount of clock cycles, Table 4 shows that the inversion module based on the extGCD achieves a smaller footprint. This implies that the extGCD implementation results in a better area-time product. We note that the inversion based on Fermat's little theorem always requires a dense polynomial multiplier, which increases the area cost notably. For the design with the best area-time product, our approach consumes roughly twice the amount of logic but finishes the inversion with only one sixth clock cycles setting $b = 32$.

While writing this article, Deshpande et *al.* [DdPM+21] presented a hardware implementation of Bernstein and Yang's inversion algorithm for computing the modular inverse for integers. Their implementation targets integer sizes of 255 bits to 2 048 bits which requires units for integer additions with carry logic. Since we compute the inverse of bit polynomials of at least 12 323 bits and perform carry-less additions, i.e., the `XOR` operation, the two implementations target different applications, and a comparison of performance numbers would be misleading.

Additionally, referring to the sequential design of [DdPM+21], they always compute the control bits for only one `divstep` and update the integers with one `divstep`. This corresponds to the configuration of $s = 1$ in our design introduced in Section 3.4. Hence, our inversion module provides more configurations allowing to finely adapt to various circumstances.

4.4 Key Generation

We report implementation results for stand-alone key generation modules in Table 5 and compare them to the key generation module from [RBMG21]. We evaluate our designs only on the key generation because the polynomial inversion module is used solely in this KEM operation. Because our design is based on the extGCD instead on Fermat's little theorem, we do not install a dense polynomial multiplier that is required for the inversion with Fermat's little theorem. Instead, we use a sparse multiplier which is far more efficient (in both area and latency) than the dense multiplier in the key generation (cf. Table 3). Although the module of key generation consists of various components, including the PRNG based on SHAKE256, the main operations occur in the inversion module and the multiplier.

As described before, both designs perfectly scale with the bandwidth parameter b while the inversion module provides an additional configuration via the step size s. Nevertheless, for each $b \in \mathcal{B}$, we only pick two configurations for the inversion: (1) setting $s = b$ which results in the fastest configurations we can achieve, and (2) instantiating the inversion module with the lowest AT product determined in Section 4.3.

The fastest key generation, that we can implement with our approaches, is obtained for $b = s = 128$. The key generation only takes 484 µs but requires over 25 000 slices. The maximum frequencies for the designs with $b = 128$ are slightly higher than for $b = 64$ because the parameter d is decreased to $d = 1$. We decided to synthesize these designs for $d = 1$ since otherwise the critical path for the computation of the control bits would drastically increase. Note, the results for $b = 64$ and $b = 128$ for the designs adjusted to the best AT product achieves roughly the same performance because b is doubled while s is halved. Therefore, the design for $b = 64$ is more efficient due to the lower footprint.

Table 4: Comparison of our inversion module to related work for $r = 12\,323$.

	Resources				Performance		
	Logic	Memory		Area	Cycles	Frequency	Latency
b	LUT	FF	BRAM	Slices	Cycles	MHz	μs
Our light-weight designs ($s = 1$)							
32	580	117	7	196	9 637 363	100	96 637
64	1 020	183	7	377	4 880 299	100	48 803
128	1 805	247	14	671	2 514 091	100	25 141
Our high-speed design ($s = b$)							
32	5 038	943	8	1 473	316 504	100	3 165
64	18 610	3 563	8	5 457	91 678	100	917
128	75 269	14 028	16	21 435	47 386	100	474
Our design with the best area-time product ($s = 23$, $s = 31$, $s = 16$)							
32	3 359	643	8	995	434 255	100	4 343
64	7 801	1 473	8	2 269	172 522	100	1 725
128	8 322	1 245	16	2 560	182 138	100	1 821
Our design targeting the clock cycles of [RBMG21] ($s = 4$, $s = 7$, $s = 11$)							
32	905	179	8	313	2 416 672	100	24 167
64	2 391	334	8	786	708 310	100	7 083
128	5 615	1 157	16	1 807	253 533	100	2 535
Inversion Module used in [RBMG21]							
32	1 721	343	5	495	2 670 881	131.58	20 299
64	3 597	419	5	994	748 769	113.64	6 589
128	11 878	722	10	3 352	258 555	96.15	2 689

Since our proposed inversion module is highly scalable, there are many other possible configurations. An estimation of the expected footprint and clock cycles can be obtained by using the results provided in the appendix.

Unfortunately, the authors of [RBMG21] did not implement a PRNG to provide randomness to the sampler which makes a comparison more difficult. Therefore, we determined the hardware utilization of our Keccak core which consumes roughly 800 slices. Considering these additional costs, our design adjusted to the AT products of the inversion modules is roughly 5.5 times faster while it only consumes 3.6 more number of slices for $b = 32$.

4.5 United Design

We present the implementation results of the united hardware design of BIKE, introduced in Section 3.5, in Table 6 for the lowest security level. Results for Level 3 and Level 5 can be found in the appendix in Table 11. We created three different implementations where the first one is a light-weight design ($b = 32$), the second one is a design with a trade-off between hardware resources and performance ($b = 64$), and the last one is a high-speed design with $b = 128$. The instantiations of the inversion module are the designs with the best AT product identified in Section 4.3.

Table 6 also contains the estimated implementation results for a united hardware design of BIKE from [RBMG21]. For the light-weight configuration, our design clearly outperforms the previous design with respect to the hardware resources and performance. This improvement is mainly due to the new multiplier design and inversion module.

For the high-speed design, our proposed implementation consumes only half the amount

Table 5: Comparison of stand-alone key generation modules for $r = 12\,323$.

Configuration				Utilization				Performance		
				Logic	Memory		Area	Cycles	Frequency	Latency
b	s	d	PRNG*	LUT	FF	BRAM	Slices	Cycles	MHz	ms
This work – High Speed ($s = b$)										
32	32	2	✓	9 880	3 321	5	3 070	344 777	130.91	2.63
64	64	2	✓	24 564	6 255	10	7 776	106 243	104.65	1.02
128	128	1	✓	82 457	17 510	19	25 009	55 135	113.95	0.484
This work – Best AT product for inversion										
32	23	2	✓	7 791	3 004	5	2 179	462 533	125	3.7
64	31	2	✓	12 741	4 169	10	3 694	187 097	98.04	1.91
128	16	1	✓	14 705	4 709	19	4 121	189 897	113.64	1.67
KeyGen from [RBMG21]										
32	–	–	✗	2 074	659	4	649	2 671 076	131.58	20.30
64	–	–	✗	4 432	1 285	5	1 285	748 964	113.64	6.59
128	–	–	✗	12 654	3 554	10	3 554	258 750	96.15	2.69

* The PRNG (KECCAK) is used to sample (h_0, h_1) (the core consumes roughly 800 slices).

of slices while achieving comparable performance results. Particularly, the latency of the key generation is significantly improved due to the inversion module. However, the number of clock cycles for the encapsulation and decapsulation slightly increased. This slight increase is due to the sparse polynomial multiplier.

Since the latency of the sparse multiplier is proportional to the Hamming weight of the sparse polynomial (cf. Equation 1), the schoolbook multiplier achieves a better performance when the Hamming weight of the sparse polynomial exceeds a certain value. More precisely, the latency of the schoolbook multiplier from [RBMG21] is defined by

$$L_{\text{school}} = \left\lceil \frac{r}{b} \right\rceil^2 + 3 \cdot \left\lceil \frac{r}{b} \right\rceil + 1. \tag{5}$$

In case $L_{\text{mult}}(th)$ results in a larger latency than L_{school} for a Hamming weight th and a fixed $\lceil r/b \rceil$, the schoolbook multiplier finishes the corresponding multiplication in less clock cycles. In BIKE, this phenomena only appears for $b = 128$ and for the parameter sets of the security levels 1 and 3. However, especially for $b = 128$ the sparse multiplier achieves a considerably better AT product as shown in Table 3.

Besides implementation results for BIKE, Table 6 also provides implementation costs and performance values for other code-based cryptographic schemes submitted to the NIST standardization process. As already pointed out in [RBMG21], the comparison to the Classic McEliece implementation is difficult. On the one hand, the reported numbers are only for the Public-Key Encryption (PKE) scheme and not for the KEM. On the other hand, the Classic McEliece design consumes a huge amount of BRAMs which requires to use larger and more expensive FPGAs.

The hardware design for HQC was recently presented in the latest specification [MAB+21] and is based on a high-level synthesis. While our hardware design of BIKE achieves similar performance results for the encapsulation and decapsulation, HQC has a faster key generation since no polynomial inversion is required.

Eventually, the last part of Table 6 reports recent hardware implementation results from other post-quantum schemes which were selected as finalists in the NIST standardization process. We list the corresponding implementation costs and performance numbers from lattice-based schemes including CRYSTALS-KYBER, LightSaber, and NTRU Prime. In general, the comparison shows that lattice-based schemes cost less area and achieve lower latencies than the code-based KEM operations.

Table 6: Comparison of hardware implementations of post-quantum schemes.

	Utilization					Performance					
	Logic		Memory		Area	Frequency	Key Gen		Encaps		Decaps	
Design	LUT	DSP	FF	BRAM	Slices	MHz	cycles†	μs	cycles†	μs	cycles†	μs
This work, united design												
Light weight	12 319	7	3 896	9	3 777	121	463	3 797	54	443	841	6 896
Trade-off	19 607	9	5 008	17	5 617	100	187	1 870	28	280	421	4 210
High speed	25 549	13	5 462	34	7 332	113	190	1 672	15	132	215	1 892
BIKE [RBMG21]												
Light weight	12 868	7	5 354	17	4 078	121	2 671	21 903	153	1 252	1 628	13 349
High speed	52 967	13	7 035	49	15 187	96	259	2 691	12	127	189	1 972
HQC [MAB+21]												
Light weight	8 900	0	6 400	14	3 100	132	630	4 773	1 500	11 364	2 100	15 909
High speed	20 000	0	16 000	12.5	6 600	148	40	270	89	601	190	1 284
mceliece348864[pke] [WSN18]												
Light weight	25 327	0	49 383	168	–	108	1 600	14 800	2.7	25.2	18.3	169.8
High speed	81 339	0	132 190	236	–	106	203	1 920	2.7	25.8	12.7	120.7
CRYSTALS-KYBER												
[XL21]	7 412	2	4 644	3	2 126	161	3.8	23.4	5.1	30.5	6.7	41.3
[DMG21]	9 457	4	8 543	4.5	–	220	2.2	10	3.2	14.7	4.5	20.5
LightSaber [DMG21]												
Light weight	24 688	0	14 785	1.5	–	370	1.6	4.3	2.2	5.8	2.8	7.6
High speed	65 890	0	28 230	1.5	–	310	0.9	2.9	1	3.3	1.3	4.2
NTRU Prime [Mar20]												
–	9 538	19	7 803	14	1 841	271	1 305	4 815	142	524	260	958

[pke] Results are only for the PKE and not for the KEM. † in thousand.

5 Discussion

In this section, we briefly discuss the resistance of our implementations against side-channel attacks and address the transferability of our optimization approaches to software implementations.

5.1 Resistance against Side Channels

In this work, we present a constant-time hardware implementation of BIKE which prevents the timing side-channel leakage. However, we did not apply any specific countermeasure against power Side-Channel Analysis (SCA). In [RMGS20], the authors briefly discussed the resistance of their BIKE hardware implementation against power side channels. They suggested that a parallel processing of $b = 128$ bit chunks makes it hard to identify single bit dependencies in the power trace. Since our implementation also supports a 128 bit bandwidth, it follows the same argumentation. Additionally, using BIKE with ephemeral keys (suggested as one operation mode in the BIKE specification [ABB+21]), makes a side-channel attack even harder since the attacker can only use single traces.

Nevertheless, this is not a guarantee for resisting power side-channel attacks. For example, analyzing a power trace of our proposed multiplication engine from Section 3.3 would probably reveal if an index of e_0 or e_1 is processed due to the Hamming weight difference of $|1|$ and $|h|$. The multiplication with an index from e_0 probably generates different power traces than a multiplication with e_1 such that the Hamming weights of $|e_0|$ and $|e_1|$ are leaked. It requires further research to investigate the effect with respect to security from leaking $|e_0|$ and $|e_1|$. The leakage can be avoid by using two sparse multipliers, where one is dedicated to $e_1 \cdot 1$ and the other is dedicated to $e_2 \cdot h$ running in parallel.

5.2 Transferability to Software

In this section we discuss the possibility of transferring the presented approaches to software implementations for polynomial inversions and spare polynomial multiplications targeting various platforms.

When considering the inversion algorithms for the key generation, given the latency of extGCD inversion (Equation 3) and Fermat's inversion (Equation 4), the key issue is the latency of the exponentiation and multiplication (L_{school}) operations in the ITA algorithm on the target platforms. Although the multiplication involves complicated hardware circuits, it is a sunk cost in software when the underlying platform supports related instructions. Therefore, for platforms with native instructions of bit-polynomial multiplication, e.g., the `pclmulqdq` instruction in `x86`, we believe $L_{\text{inv-Fermat}}$ is smaller than L_{inv}. For platforms without instructions for bit-polynomial multiplication, L_{inv} is likely to be smaller than $L_{\text{inv-Fermat}}$. However, besides the platform, the latency of the multiplication also depends on the implemented algorithms. Recently, Chen et *al.* [CCK21] reported an efficient FFT-based bit-polynomial multiplication on the 32-bit Arm Cortex-M4 platform. Hence, we expect extGCD based inversion outperforms Fermat's inversion in even smaller platforms without efficient multiplication implementations, e.g., 8-bit AVR microcontrollers.

Regarding the sparse polynomial multiplication in BIKE, we mainly consider the side-channel leakage of the degrees of sparse terms. If a software implements the sparse-dense multiplication by accumulating the shifted dense polynomial with the degrees of sparse terms, then it might leak the degrees of sparse terms through a cache-time attack. This is a reason that recent software implementations, e.g., [CCK21, DGK20a], implemented the multiplication with algorithms for dense polynomial multiplication. Thus, we believe that the spare polynomial multiplication will be useful for small microcontrollers without data cache.

6 Conclusion

In this work, we propose various optimization strategies and present an improved hardware design for BIKE, one of the NIST's alternate KEM candidates.

For arithmetic optimizations, we implement a constant-time sparse polynomial multiplier for all three KEM algorithms of BIKE. Compared to a schoolbook implementation, our design improves the area-time product of at least five times for all design parameters. Our implementation also achieves a better latency except for the high-speed design (i.e., $b = 128$) for the encapsulation and the decapsulation. Additionally, we propose a hardware implementation of the polynomial inversion based on the extended Euclidean algorithm. Compared to previous results based on Fermat's little theorem, our new design not only achieves better latency but also provides smaller area-time products for the key generation in BIKE. Moreover, due to its scalable design, the instantiation of the inversion module can be tailored to various circumstances providing higher throughputs or smaller area footprints.

Besides these arithmetic optimizations, we show that the random oracles of a unified Keccak core in the new specification of BIKE indeed result in a more efficient hardware design compared to the design using versions of both AES256 and SHA2. Based on our improvements, we developed a united hardware design with shared resources and sub-modules, achieving a better latency with less area compared to previous BIKE implementations. All together, our high-speed implementation performs a key generation in 1 672 µs, an encapsulation in 132 µs, and a decapsulation in 1 802 µs on Xilinx Artix-7 FPGAs.

Acknowledgments

The work described in this paper has been funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) as part of the Excellence Strategy of the German Federal and State Governments – EXC 2092 CASA - 390781972 and by the projects QuantumRISC (16KIS1038) and PQC4Med (16KIS1044) supported by the German Federal Ministry of Education and Research BMBF.

A Additional Implementation Results

Table 7: Implementation results for the polynomial inversion for $r = 12\,323$, $b = 32$, and $d = 2$. We fixed the frequency to 100 MHz and selected an Artix-7 XC7A200T FPGA as target platform.

	Utilization			**Performance**		
Step Size s	LUT	FF	Slices	Clock Cycles	Latency [ms]	Area-Time
$s = 1$	580	117	196	9 637 363	96.37	1 888.92
$s = 2$	732	168	254	4 819 461	48.19	1 224.14
$s = 3$	852	175	296	3 221 840	32.22	953.66
$s = 4$	905	179	313	2 416 672	24.17	756.42
$s = 5$	1 131	187	369	1 938 658	19.39	715.36
$s = 6$	1 166	193	375	1 615 612	16.16	605.85
$s = 7$	1 389	199	458	1 388 442	13.88	635.91
$s = 8$	1 493	206	491	1 215 082	12.15	596.61
$s = 9$	1 697	346	547	1 085 811	10.86	593.94
$s = 10$	1 788	355	576	977 307	9.77	562.93
$s = 11$	1 929	366	601	890 844	8.91	535.40
$s = 12$	1 808	373	578	816 606	8.17	472.00
$s = 13$	1 977	383	613	755 776	7.56	463.29
$s = 14$	2 063	393	639	702 045	7.02	448.61
$s = 15$	2 245	403	685	657 123	6.57	450.13
$s = 16$	2 479	414	759	616 026	6.16	467.56
$s = 17$	2 526	557	767	582 617	5.83	446.87
$s = 18$	2 619	570	823	550 536	5.51	453.09
$s = 19$	2 765	585	847	522 962	5.23	442.95
$s = 20$	2 905	601	893	496 832	4.97	443.67
$s = 21$	3 068	613	934	474 289	4.74	442.99
$s = 22$	3 232	631	999	452 929	4.53	452.48
$s = 23$	3 359	643	995	434 255	4.34	432.08
$s = 24$	3 679	655	1 105	416 076	4.16	459.76
$s = 25$	3 705	802	1 096	401 483	4.01	440.03
$s = 26$	3 869	819	1 191	386 055	3.86	459.79
$s = 27$	3 998	840	1 177	372 758	3.73	438.74
$s = 28$	4 149	865	1 287	359 732	3.60	462.98
$s = 29$	4 411	877	1 342	347 967	3.48	466.97
$s = 30$	4 549	900	1 350	336 542	3.37	454.33
$s = 31$	4 735	921	1 410	326 729	3.27	460.69
$s = 32$	5 038	943	1 473	316 504	3.17	466.21

Table 8: Implementation results for the polynomial inversion for $r = 12\,323$, $b = 64$, and $d = 2$. We fixed the frequency to 100 MHz and selected an Artix-7 XC7A200T FPGA as target platform.

	Utilization			**Performance**		
Step Size s	LUT	FF	Slices	Clock Cycles	Latency [ms]	Area-Time
$s = 1$	1 020	183	377	4 880 299	48.80	3 679.75
$s = 2$	1 245	296	425	2 440 543	24.41	1 037.23
$s = 3$	1 662	306	566	1 635 573	16.36	925.73
$s = 4$	1 540	312	515	1 226 827	12.27	631.82
$s = 5$	1 890	322	618	986 589	9.87	609.71
$s = 6$	1 947	327	676	822 189	8.22	555.80
$s = 7$	2 391	334	786	708 310	7.08	556.73
$s = 8$	2 637	348	893	619 870	6.20	553.54
$s = 9$	2 633	613	878	556 605	5.57	488.70
$s = 10$	2 865	629	922	500 983	5.01	461.91
$s = 11$	3 045	632	1 015	457 752	4.58	464.62
$s = 12$	3 208	645	1 021	419 605	4.20	428.42
$s = 13$	3 444	658	1 122	389 269	3.89	436.76
$s = 14$	3 623	665	1 201	361 593	3.62	434.27
$s = 15$	3 964	685	1 245	339 252	3.39	422.37
$s = 16$	4 506	704	1 422	318 034	3.18	452.24
$s = 17$	4 429	969	1 369	302 188	3.02	413.70
$s = 18$	4 537	985	1 417	285 547	2.86	404.62
$s = 19$	4 697	992	1 440	271 869	2.72	391.49
$s = 20$	4 975	1 010	1 566	258 284	2.58	404.47
$s = 21$	5 439	1 030	1 657	247 128	2.47	409.49
$s = 22$	5 476	1 046	1 758	235 997	2.36	414.88
$s = 23$	5 804	1 064	1 766	226 780	2.27	400.49
$s = 24$	6 323	1 095	1 892	217 286	2.17	411.11
$s = 25$	6 280	1 353	1 928	210 606	2.11	406.05
$s = 26$	6 724	1 383	2 064	202 512	2.03	417.98
$s = 27$	6 769	1 390	2 034	195 970	1.96	398.60
$s = 28$	6 862	1 407	2 080	189 120	1.89	393.37
$s = 29$	7 517	1 442	2 197	183 338	1.83	402.79
$s = 30$	7 733	1 462	2 281	177 317	1.77	404.46
$s = 31$	7 801	1 473	2 269	172 522	1.73	391.45
$s = 32$	8 379	1 500	2 560	167 122	1.67	427.83
$s = 33$	8 324	1 772	2 425	163 434	1.63	396.33
$s = 34$	8 339	1 799	2 480	158 638	1.59	393.42
$s = 35$	8 687	1 814	2 621	154 980	1.55	406.20
$s = 36$	9 016	1 836	2 692	150 602	1.51	405.42
$s = 37$	9 288	1 860	2 735	147 325	1.47	402.93
$s = 38$	9 552	1 882	2 871	143 367	1.43	411.61
$s = 39$	9 909	1 911	2 851	140 261	1.40	399.88
$s = 40$	10 426	1 945	3 090	136 942	1.37	423.15
$s = 41$	10 374	2 227	3 092	134 830	1.35	416.89
$s = 42$	10 751	2 260	3 202	131 489	1.31	421.03
$s = 43$	10 989	2 282	3 247	129 160	1.29	419.38
$s = 44$	11 233	2 310	3 315	126 248	1.26	418.51
$s = 45$	11 737	2 357	3 417	123 887	1.24	423.32
$s = 46$	11 853	2 379	3 419	121 188	1.21	414.34
$s = 47$	12 516	2 430	3 694	119 236	1.19	440.46
$s = 48$	12 758	2 459	3 623	116 750	1.17	422.99
$s = 49$	12 960	2 740	3 673	115 272	1.15	423.39
$s = 50$	13 305	2 777	3 873	112 992	1.13	437.62
$s = 51$	14 176	2 818	4 211	111 420	1.11	469.19
$s = 52$	13 762	2 837	3 973	109 135	1.09	433.59
$s = 53$	15 295	2 882	4 397	107 763	1.08	473.83
$s = 54$	14 616	2 903	4 200	105 695	1.06	443.92
$s = 55$	16 613	2 975	4 825	104 302	1.04	503.26
$s = 56$	16 031	3 002	4 701	102 454	1.02	481.64
$s = 57$	15 994	3 289	4 618	101 473	1.01	468.60
$s = 58$	16 445	3 326	4 780	99 613	1.00	476.15
$s = 59$	16 491	3 345	4 697	98 399	0.98	462.18
$s = 60$	17 232	3 397	5 005	96 761	0.97	484.29
$s = 61$	17 711	3 435	5 116	95 757	0.96	489.89
$s = 62$	17 171	3 462	4 984	94 115	0.94	469.07
$s = 63$	18 103	3 510	5 318	93 095	0.93	494.15
$s = 64$	18 610	3 563	5 457	91 678	0.92	500.29

Table 9: Implementation results for the polynomial inversion for $r = 12\,323$, $b = 128$, and $d = 1$. We fixed the frequency to 100 MHz and selected an Artix-7 XC7A200T FPGA as target platform.

	Utilization			**Performance**		
Step Size s	LUT	FF	Slices	Clock Cycles	Latency [ms]	Area-Time
$s = 1$	1 805	247	671	2 514 091	25.14	16 869.55
$s = 2$	2 134	517	801	1 269 570	12.70	1 016.93
$s = 3$	2 519	527	944	854 765	8.55	806.90
$s = 4$	2 880	542	1 030	647 311	6.47	666.73
$s = 5$	3 257	553	1 158	522 881	5.23	605.50
$s = 6$	3 616	565	1 224	439 857	4.40	538.38
$s = 7$	4 239	578	1 393	380 569	3.81	530.13
$s = 8$	4 496	587	1 498	336 130	3.36	503.52
$s = 9$	4 857	1 117	1 618	304 329	3.04	492.40
$s = 10$	5 315	1 139	1 689	276 380	2.76	466.81
$s = 11$	5 615	1 157	1 807	253 533	2.54	458.13
$s = 12$	6 083	1 170	1 936	234 457	2.34	453.91
$s = 13$	6 286	1 183	2 001	218 341	2.18	436.90
$s = 14$	6 866	1 202	2 211	204 576	2.05	452.32
$s = 15$	7 284	1 215	2 299	192 648	1.93	442.90
$s = 16$	8 322	1 245	2 560	182 138	1.82	466.27
$s = 17$	7 860	1 758	2 407	174 300	1.74	419.54
$s = 18$	8 398	1 787	2 561	166 069	1.66	425.30
$s = 19$	8 553	1 792	2 623	158 655	1.59	416.15
$s = 20$	9 161	1 824	2 903	151 958	1.52	441.13
$s = 21$	9 392	1 834	2 877	145 876	1.46	419.69
$s = 22$	10 000	1 866	3 197	140 424	1.40	448.94
$s = 23$	10 510	1 881	3 174	135 372	1.35	429.67
$s = 24$	11 576	1 930	3 487	130 730	1.31	455.86
$s = 25$	11 088	2 427	3 370	127 494	1.27	429.65
$s = 26$	11 836	2 471	3 569	123 540	1.24	440.91
$s = 27$	12 205	2 488	3 742	119 903	1.20	448.68
$s = 28$	12 346	2 507	3 816	116 590	1.17	444.91
$s = 29$	13 172	2 550	3 999	113 350	1.13	453.29
$s = 30$	13 565	2 567	4 046	110 447	1.10	446.87
$s = 31$	14 739	2 612	4 512	107 758	1.08	486.20
$s = 32$	14 632	2 624	4 334	105 154	1.05	455.74
$s = 33$	14 854	3 162	4 382	103 386	1.03	453.04
$s = 34$	15 394	3 201	4 547	101 075	1.01	459.59
$s = 35$	17 161	3 249	5 225	98 997	0.99	517.26
$s = 36$	16 025	3 238	4 864	96 884	0.97	471.24
$s = 37$	17 789	3 314	5 295	95 011	0.95	503.08
$s = 38$	17 007	3 295	5 113	93 106	0.93	476.05
$s = 39$	18 682	3 365	5 590	91 309	0.91	510.42
$s = 40$	18 997	3 386	5 560	89 762	0.90	499.08
$s = 41$	20 059	3 956	5 853	88 790	0.89	519.69
$s = 42$	19 581	3 953	5 776	87 176	0.87	503.53
$s = 43$	20 206	4 000	5 901	85 822	0.86	506.44
$s = 44$	20 562	4 037	6 044	84 446	0.84	510.39
$s = 45$	21 092	4 052	6 158	83 047	0.83	511.40
$s = 46$	21 587	4 076	6 390	81 772	0.82	522.52
$s = 47$	23 411	4 157	6 754	80 623	0.81	544.53
$s = 48$	23 283	4 189	6 851	79 454	0.79	544.34
$s = 49$	22 459	4 701	6 597	78 768	0.79	519.63
$s = 50$	23 571	4 742	6 772	77 701	0.78	526.19
$s = 51$	24 722	4 799	7 379	76 768	0.77	566.47
$s = 52$	24 364	4 821	7 173	75 667	0.76	542.76
$s = 53$	25 137	4 864	7 296	74 855	0.75	546.14
$s = 54$	25 516	4 887	7 366	73 874	0.74	544.16
$s = 55$	26 330	4 932	7 765	73 033	0.73	567.10
$s = 56$	27 413	4 995	8 144	72 178	0.72	587.82
$s = 57$	28 143	5 522	8 223	71 741	0.72	589.93
$s = 58$	28 594	5 594	8 151	70 850	0.71	577.50
$s = 59$	28 171	5 587	8 310	70 105	0.70	582.57
$s = 60$	28 393	5 640	8 209	69 347	0.69	569.27
$s = 61$	29 925	5 696	8 545	68 739	0.69	587.37
$s = 62$	30 319	5 734	8 769	67 957	0.68	595.91
$s = 63$	30 852	5 775	8 847	67 327	0.67	595.64
$s = 64$	32 695	5 864	9 527	66 686	0.67	635.32

Table 10: Implementation results for the polynomial inversion for $r = 12\,323$, $b = 128$, and $d = 1$. We fixed the frequency to 100 MHz and selected an Artix-7 XC7A200T FPGA as target platform.

	Utilization			**Performance**		
Step Size s	LUT	FF	Slices	Clock Cycles	Latency [ms]	Area-Time
$s = 65$	31 221	6 352	8 847	66 414	0.66	587.56
$s = 66$	31 436	6 389	9 158	65 746	0.66	602.10
$s = 67$	30 577	6 397	8 972	65 067	0.65	583.78
$s = 68$	32 594	6 503	9 607	64 547	0.65	620.10
$s = 69$	35 630	6 620	10 104	64 018	0.64	646.84
$s = 70$	33 656	6 572	9 761	63 480	0.63	619.63
$s = 71$	35 061	6 638	10 077	62 933	0.63	634.18
$s = 72$	36 133	6 719	10 485	62 378	0.62	654.03
$s = 73$	36 613	7 276	10 416	62 151	0.62	647.36
$s = 74$	38 766	7 388	11 314	61 748	0.62	698.62
$s = 75$	38 813	7 410	11 239	61 162	0.61	687.40
$s = 76$	37 402	7 400	10 770	60 744	0.61	654.21
$s = 77$	39 655	7 480	11 572	60 319	0.60	698.01
$s = 78$	38 221	7 473	11 045	59 709	0.60	659.49
$s = 79$	39 225	7 561	11 016	59 269	0.59	652.91
$s = 80$	41 260	7 653	11 871	59 002	0.59	700.41
$s = 81$	41 334	8 206	11 838	58 853	0.59	696.70
$s = 82$	40 519	8 219	11 344	58 389	0.58	662.36
$s = 83$	41 921	8 315	11 794	57 918	0.58	683.08
$s = 84$	42 303	8 364	12 025	57 625	0.58	692.94
$s = 85$	43 112	8 427	12 378	57 140	0.57	707.28
$s = 86$	44 150	8 471	12 741	56 836	0.57	724.15
$s = 87$	45 026	8 552	13 084	56 525	0.57	739.57
$s = 88$	46 817	8 654	13 632	56 210	0.56	766.25
$s = 89$	46 236	9 187	13 206	55 977	0.56	739.23
$s = 90$	46 659	9 248	13 038	55 647	0.56	725.53
$s = 91$	48 035	9 326	13 776	55 312	0.55	761.98
$s = 92$	46 928	9 348	13 154	54 972	0.55	723.10
$s = 93$	49 552	9 461	14 406	54 820	0.55	789.74
$s = 94$	51 236	9 514	14 461	54 470	0.54	787.69
$s = 95$	48 958	9 518	13 927	54 114	0.54	753.65
$s = 96$	51 575	9 644	14 776	53 754	0.54	794.27
$s = 97$	52 032	10 221	14 911	53 839	0.54	802.79
$s = 98$	52 507	10 290	14 760	53 466	0.53	789.16
$s = 99$	54 072	10 373	15 308	53 088	0.53	812.67
$s = 100$	52 470	10 375	14 946	52 905	0.53	790.72
$s = 101$	54 968	10 517	15 569	52 719	0.53	820.78
$s = 102$	54 428	10 506	15 225	52 326	0.52	796.66
$s = 103$	57 494	10 651	16 235	52 132	0.52	846.36
$s = 104$	58 278	10 710	16 475	51 730	0.52	852.25
$s = 105$	57 104	11 241	15 935	51 762	0.52	824.83
$s = 106$	57 155	11 314	16 171	51 554	0.52	833.68
$s = 107$	58 343	11 416	16 257	51 343	0.51	834.68
$s = 108$	58 391	11 443	16 525	51 128	0.51	844.89
$s = 109$	59 464	11 547	16 778	50 910	0.51	854.17
$s = 110$	59 938	11 563	16 937	50 688	0.51	858.50
$s = 111$	60 228	11 677	16 895	50 463	0.50	852.57
$s = 112$	63 192	11 787	17 473	50 234	0.50	877.74
$s = 113$	60 735	12 262	16 590	50 220	0.50	833.15
$s = 114$	61 661	12 364	17 550	49 982	0.50	877.18
$s = 115$	64 408	12 476	18 190	49 741	0.50	904.79
$s = 116$	63 069	12 475	17 966	49 496	0.49	889.25
$s = 117$	63 678	12 579	17 887	49 248	0.49	880.90
$s = 118$	67 589	12 727	18 588	48 996	0.49	910.74
$s = 119$	64 989	12 706	18 025	48 960	0.49	882.50
$s = 120$	66 836	12 784	18 754	48 702	0.49	913.36
$s = 121$	69 566	13 467	19 483	48 644	0.49	947.73
$s = 122$	67 738	13 479	18 871	48 600	0.49	917.13
$s = 123$	72 170	13 641	20 145	48 330	0.48	973.61
$s = 124$	71 966	13 736	20 013	48 057	0.48	961.76
$s = 125$	73 447	13 795	20 338	48 006	0.48	976.35
$s = 126$	72 494	13 814	19 953	47 727	0.48	952.30
$s = 127$	72 596	13 900	20 096	47 671	0.48	958.00
$s = 128$	75 269	14 028	21 435	47 386	0.47	1 015.72

Table 11: Implementation results of the united design for Level 3 and Level 5.

	Utilization					**Performance**					
	Logic		Memory		Area	Frequency	Key Gen		Encaps		Decaps	
Design	LUT	DSP	FF	BRAM	Slices	MHz	cycles†	μs	cycles†	μs	cycles†	μs
United design for $r = 24\,659$												
Low weight	13 850	7	4 010	15	4 152	116	1 775	15 268	157	1 348	2 381	20 479
Trade-off	20 049	9	5 039	17	5 688	100	693	6 929	80	801	1 198	11 982
High speed	25 811	13	5 460	34	7 242	113	681	5 997	42	367	605	5 325
United design for $r = 40\,973$												
Low weight	13 973	7	4 002	34	4 192	113	4 809	42 324	343	3 020	5 217	45 911
Trade-off	21 373	9	5 160	34	6 145	94	1 847	19 580	174	1 847	2 620	27 770
High speed	26 441	13	5 601	34	7 288	111	1 798	16 186	90	808	1 321	11 885

† in thousand.

References

[ABB+19] Nicolas Aragon, Paulo SLM Barreto, Slim Bettaieb, France Worldline, Loïc Bidoux, Olivier Blazy, Philippe Gaborit, Tim Güneysu, Carlos Aguilar Melchor, Rafael Misoczki, et al. BIKE: Bit Flipping Key Encapsulation - Round 2 Submission. 2019. https://bikesuite.org/files/round2/spec/BIKE-Spec-Round2.2019.03.30.pdf.

[ABB+20] Nicolas Aragon, Paulo SLM Barreto, Slim Bettaieb, France Worldline, Loïc Bidoux, Olivier Blazy, Philippe Gaborit, Tim Güneysu, Carlos Aguilar Melchor, Rafael Misoczki, et al. BIKE: Bit Flipping Key Encapsulation - Round 3 Submission. 2020. https://bikesuite.org/files/round2/spec/BIKE-Spec-2020.02.07.1.pdf.

[ABB+21] Nicolas Aragon, Paulo SLM Barreto, Slim Bettaieb, France Worldline, Loïc Bidoux, Olivier Blazy, Philippe Gaborit, Tim Güneysu, Carlos Aguilar Melchor, Rafael Misoczki, et al. BIKE: Bit Flipping Key Encapsulation. 2021. https://bikesuite.org/files/v4.2/BIKE_Spec.2021.07.26.1.pdf.

[BBC+19] Marco Baldi, Alessandro Barenghi, Franco Chiaraluce, Gerardo Pelosi, and Paolo Santini. LEDAcrypt: QC-LDPC Code-Based Cryptosystems with Bounded Decryption Failure Rate. In *Code-Based Cryptography - 7th International Workshop*, volume 11666 of *Lecture Notes in Computer Science*, pages 11–43. Springer, 2019.

[BDPA13] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. Keccak. In *EUROCRYPT*, volume 7881 of *Lecture Notes in Computer Science*, pages 313–314. Springer, 2013.

[BFG+19] Alessandro Barenghi, William Fornaciari, Andrea Galimberti, Gerardo Pelosi, and Davide Zoni. Evaluating the Trade-offs in the Hardware Design of the LEDAcrypt Encryption Functions. In *26th IEEE International Conference on Electronics, Circuits and Systems*, pages 739–742. IEEE, 2019.

[BY19] Daniel J. Bernstein and Bo-Yin Yang. Fast constant-time gcd computation and modular inversion. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2019(3):340–398, 2019.

[CCK21] Ming-Shing Chen, Tung Chou, and Markus Krausz. Optimizing BIKE for the Intel Haswell and ARM Cortex-M4. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2021(3):97–124, Jul. 2021.

[DdPM+21] Sanjay Deshpande, Santos Merino del Pozo, Victor Mateu, Marc Manzano, Najwa Aaraj, and Jakub Szefer. Modular Inverse for Integers using Fast Constant Time GCD Algorithm and its Applications. *International Conference on Field-Programmable Logic and Applications (FPL)*, 2021.

[DGK20a] Nir Drucker, Shay Gueron, and Dusan Kostic. Additional Implementation of BIKE (Bit Flipping Key Encapsulation). github, 2020. https://github.com/awslabs/bike-kem.

[DGK20b] Nir Drucker, Shay Gueron, and Dusan Kostic. QC-MDPC Decoders with Several Shades of Gray. In *International Conference on Post-Quantum Cryptography*, pages 35–50. Springer, 2020.

[DMG21] Viet Ba Dang, Kamyar Mohajerani, and Kris Gaj. High-Speed Hardware Architectures and Fair FPGA Benchmarking of CRYSTALS-Kyber, NTRU, and Saber. 2021.

[Gam20] Jay Gambetta. IBM's Roadmap For Scaling Quantum Technology. IBM Research Blog, 2020. https://www.ibm.com/blogs/research/2020/09/ibm-quantum-roadmap/.

[HC17] Jingwei Hu and Ray CC Cheung. Area-Time Efficient Computation of Niederreiter Encryption on QC-MDPC Codes for Embedded Hardware. *IEEE Transactions on Computers*, 66(8):1313–1325, 2017.

[HVMG13] Stefan Heyse, Ingo Von Maurich, and Tim Güneysu. Smaller Keys for Code-Based Cryptography: QC-MDPC McEliece Implementations on Embedded Devices. In *CHES 2013*, pages 273–292. Springer, 2013.

[HWCW19] Jingwei Hu, Wen Wang, Ray CC Cheung, and Huaxiong Wang. Optimized Polynomial Multiplier Over Commutative Rings on FPGAs: A Case Study on BIKE. In *2019 International Conference on Field-Programmable Technology (ICFPT)*, pages 231–234. IEEE, 2019.

[IT88] Toshiya Itoh and Shigeo Tsujii. A Fast Algorithm for Computing Multiplicative Inverses in GF (2m) Using Normal Bases. *Information and computation*, 78(3):171–177, 1988.

[MAB+21] Carlos Aguilar Melchor, Nicolas Aragon, Slim Bettaieb, Loıc Bidoux, Olivier Blazy, Jean-Christophe Deneuville, Philippe Gaborit, Edoardo Persichetti, Gilles Zémor, and IC Bourges. Hamming Quasi-Cyclic (HQC) – third round version, 2021.

[Mar20] Adrian Marotzke. A Constant Time Full Hardware Implementation of Streamlined NTRU Prime. In *CARDIS 2020*, volume 12609 of *Lecture Notes in Computer Science*, pages 3–17. Springer, 2020.

[MTSB13] Rafael Misoczki, Jean-Pierre Tillich, Nicolas Sendrier, and Paulo SLM Barreto. MDPC-McEliece: New McEliece Variants from Moderate Density Parity-Check Codes. In *IEEE International Symposium on Information Theory*, pages 2069–2073. IEEE, 2013.

[NIS17] NIST. Call for Proposals – Post-Quantum Cryptography. Technical Report, CSRC, 2017. https://csrc.nist.gov/Projects/Post-Quantum-Cryptography/Post-Quantum-Cryptography-Standardization/Call-for-Proposals.

[NIS20a] NIST. Guidelines for submitting tweaks for Third Round Finalists and Candidates, 2020. https://groups.google.com/a/list.nist.gov/g/pqc-forum/c/LPuZKGNyQJ0/m/O6UBanYbDAAJ.

[NIS20b] NIST. PQC Standardization Process: Third Round Candidate Announcement. Information Technology Laboratory - Computer Security Resource Center, July 2020. https://csrc.nist.gov/News/2020/pqc-third-round-candidate-announcement.

[RBMG21] Jan Richter-Brockmann, Johannes Mono, and Tim Guneysu. Folding BIKE: Scalable Hardware Implementation for Reconfigurable Devices. *IEEE Transactions on Computers*, pages 1–1, 2021.

[RMGS20] Andrew H. Reinders, Rafael Misoczki, Santosh Ghosh, and Manoj R. Sastry. Efficient BIKE Hardware Design with Constant-Time Decoder. In *IEEE International Conference on Quantum Computing and Engineering, QCE 2020*, pages 197–204. IEEE, 2020.

[VMG14] Ingo Von Maurich and Tim Güneysu. Lightweight Code-Based Cryptography: QC-MDPC McEliece Encryption on Reconfigurable Devices. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1–6. IEEE, 2014.

[WSN18] Wen Wang, Jakub Szefer, and Ruben Niederhagen. FPGA-based Niederreiter Cryptosystem using Binary Goppa codes. In *International Conference on Post-Quantum Cryptography*. Springer, 2018.

[XL21] Yufei Xing and Shuguo Li. A Compact Hardware Implementation of CCA-Secure Key Exchange Mechanism CRYSTALS-KYBER on FPGA. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2021(2):328–356, 2021.