Towards a Fast and Efficient Hardware Implementation of HQC

Sanjay Deshpande[1], Mamuri Nawan[2], Kashif Nawaz[2], Jakub Szefer[1] and Chuanqi Xu[1]

[1]CASLAB, Department of Electrical Engineering, Yale University, New Haven, USA,
sanjay.deshpande@yale.edu,chuanqi.xu@yale.edu,jakub.szefer@yale.edu
[2]Technology Innovation Institute, Abu Dhabi, UAE,
mamuri@tii.ae,kashif.nawaz@tii.ae

Abstract. This work presents a hardware design for constant-time implementation of the HQC (Hamming Quasi-Cyclic) code-based key encapsulation mechanism. HQC has been selected for the fourth-round of NIST's Post-Quantum Cryptography standardization process and this work presents first, hand-optimized design of HQC key generation, encapsulation, and decapsulation written in Verilog targeting implementation on FPGAs. The three modules further share a common SHAKE256 hash module to reduce area overhead. All the hardware modules are parametrizable at compile time so that designs for the different security levels can be easily generated. The architecture of the hardware modules includes novel, dual clock domain design, allowing the common SHAKE module to run at slower clock speed compared to the rest of the design, while other faster modules run at their optimal clock rate. The design currently outperforms the other hardware designs for HQC, and many of the fourth-round Post-Quantum Cryptography standardization process, with one of the best time-area products as well. For the dual clock design targeting lowest security level, we show that the HQC design can perform key generation in 0.12 ms, encapsulation in 0.30 ms, and decapsulation in 0.43 ms when synthesized for an Xilinx Artix 7 FPGA. The performance can be increased even further at the cost of resources by increasing the level of parallelism, e.g. by having parallel polynomial multiplication modules in the encrypt module, or including even more clock domains, one for each of the main modules. The presented design will further be made available under open-source license.

Keywords: HQC · Hamming Quasi-Cyclic · PQC · Post-Quantum Cryptography · Key Encapsulation Mechanism · Code-Based Cryptography · FPGA · Hardware Implementation

1 Introduction

Since 2016 NIST has been conducting a standardization process with the goal to standardize cryptographic primitives that are secure against attacks aided by quantum computers. There are today five main families of post-quantum cryptographic algorithms: hash-based, code-based, lattice-based, multivariate, and isogeny-based cryptography. Very recently NIST has selected one algorithm for standardization in the key encapsulation mechanism (KEM) category, CRYSTALS-Kyber, and four fourth-round candidates that will continue in the process. One of the four fourth-round candidates is HQC. It is a code-based KEM based on structured codes.

As the standardization process is coming to an end after the fourth round, the performance as well as hardware implementations of the algorithms are becoming very important factor in selection of the algorithms to be standardized. The motivation for our work is to understand how well hand-optimized HQC hardware implementation can be designed and realized on FPGAs. To date, most of the post-quantum cryptographic hardware has focused on lattice-based candidates, with code-based algorithms receiving much less attention. All existing hardware implementations for HQC are based on high-level synthesis (HLS) [AAB+20]. While HLS can be used for rapid prototyping, in our experience it cannot yet outperform Verilog or other hand optimized designs. Indeed, as we show in this work, our design outperforms the existing HQC HLS design.

In addition, our hardware design competes very well with the hardware designs for other candidates currently in the fourth round of NIST's process: BIKE, Classic McEliece, and SIKE. The presented design has best time-area product as well as time for key generation and decapsulation compared to the hardware for these designs. We also achieve similar time-area product for encapsulation when compared to BIKE. Due to limited breakdown of data for SIKE's hardware [MLRB20] comparison to SIKE for all aspects is more difficult, but we believe our design is better since for similar area cost, their combined encapsulation and decapsulation times are two orders of magnitude larger. Detailed comparison to related work is given in Section 4.

As this work aims to show, code-based designs can be competitive with other schemes when optimized hardware is developed. Further our design is constant-time, eliminating timing-based attacks. We believe our work shows that HQC can be a strong contender in the fourth round of NIST's process.

1.1 Open-Source Design

All our hardware designs reported in this paper are fully reproducible, and their source code will be released as open-source after the acceptance of this paper to a journal or a conference with proceedings.

1.2 Paper Outline

The remainder of the paper is structured as follows. Section 2 gives background on the HQC algorithm. Section 3 presents the hardware designs of the HQC modules, as well as, it provides the evaluation results. Section 4 summarizes related work and presents comparison of the HQC design to other existing designs. Section 5 concludes the paper.

2 Preliminaries

In this section, we briefly introduce HQC. We first introduce notations used in this paper. Then concatenated Reed–Muller and Reed–Solomon codes that are used to encode and decode messages in HQC are presented. In the end, HQC public key encryption (PKE) and key encapsulation mechanism (KEM) are described. We refer to the specification of HQC [AAB+20] for more detailed information.

2.1 Notation

In this paper, we denote $\mathbb{F}_2$ the binary finite field, and $\mathcal{R} = \mathbb{F}_2[X]/(X^n - 1)$ the quotient ring on which vectors and operations of HQC are defined. For any field or ring, $\mathbb{F}_2^l$ or $\mathcal{R}^l$ denotes the field or ring of l dimensional vectors over $\mathbb{F}_2$ or $\mathcal{R}$. An element $\mathbf{x} \in \mathcal{R}$ can be represented as either a vector $\mathbf{x} := (x_0, x_1, \ldots, x_{n-1})$ or a polynomial $\mathbf{x} := \sum_{i=0}^{n-1} x_i X^i$. The Hamming weight of $\mathbf{x}$ is defined as $\sum_{i=0}^{n-1} x_i$, i.e., the number of non-zero coefficients in $\mathbf{x}$. $\mathbf{x} \leftarrow \mathcal{R}$ denotes $\mathbf{x}$ is chosen uniformly random from $\mathcal{R}$, while $\mathbf{x} \overset{w}{\leftarrow} \mathcal{R}$ denotes $\mathbf{x}$ is randomly chosen from $\mathcal{R}$ and the Hamming weight of $\mathbf{x}$ is w. Optionally, a random seed can be specified for the sampling process, and the sampling process with random seed θ is denoted as $\mathbf{x} \xleftarrow{w,\theta} \mathcal{R}$. For $\mathbf{x}, \mathbf{y} \in \mathcal{R}$, the addition $\mathbf{a} = \mathbf{x} + \mathbf{y} \in \mathcal{R}$ and is defined as $a_i = x_i + y_i \bmod 2$ or $a_i = x_i$ xor y_i for $i = 0, \ldots, n-1$. The multiplication $\mathbf{a} = \mathbf{x} \cdot \mathbf{y} \in \mathcal{R}$ and is defined as $a_k = \sum_{i+j \equiv k \bmod n} x_i \cdot y_j \bmod 2$ for $k = 0, \ldots, n-1$. Encode($\cdot$) and Decode($\cdot$) are the encode and decode function of concatenated Reed–Muller and Reed–Solomon codes, which will be introduced in Section 2.2.1. $\mathcal{G}(\cdot), \mathcal{H}(\cdot), \mathcal{K}(\cdot)$ are hash functions with domain separation bytes 3, 4, 5 respectively. Parameters n, w, w_r depend on the security level, which can be found on Table 1.

2.2 HQC PKE and KEM Schemes

To introduce HQC KEM, we first introduce how to encode and decode concatenated Reed–Muller and Reed–Solomon codes, which are important parts in HQC PKE to encode and decode messages. Then we introduce HQC PKE. Finally, HQC KEM that achieves IND-CCA2 by performing the Fujisaki–Okamoto [HHK17] transformation is described.

2.2.1 Concatenated Reed–Muller and Reed–Solomon Codes

Concatenated Reed–Muller and Reed–Solomon Codes are used in the encode and decode function of HQC to encode and decode messages. Specifically, the encode of concatenated Reed–Muller and Reed–Solomon Codes is to first encode with Reed–Solomon code, and then the output is encoded with Reed–Muller code. The decode process is executed in the reverse order, i.e., first decode with Reed–Muller code, and then the output is decoded with Reed–Solomon code. The parameter sets for the concatenated code can be found in Table 1. Notice that shortened Reed–Solomon code and duplicated Reed–Muller code instead of the conventional codes are used in HQC.

Encode of shortened Reed–Solomon code. In the polynomial representation, the message can be denoted as $u(x) = u_0 + \cdots u_{k-1}x^{k-1} \in \mathbb{F}_{2^8}[x]/(x^8 + x^4 + x^3 + x^2 + 1)$. The codeword is given by $c(x) = b(x) + x^{n-k}u(x)$, where $b(x) = x^{n-k}u(x) \bmod g(x)$, and $g(x)$ is the generator polynomial[1], which is shown below for different security levels:

$$g_{\text{hqc-128}}(x) = 89 + 69x + 153x^2 + 116x^3 + 176x^4 + 117x^5 + 111x^6 + 75x^7 + 73x^8 + 233x^9 + 242x^{10} + 233x^{11} + 65x^{12} + 210x^{13} + 21x^{14} + 139x^{15} + 103x^{16} + 173x^{17} + 67x^{18} + 118x^{19} + 105x^{20} + 210x^{21} + 174x^{22} + 110x^{23} + 74x^{24} + 69x^{25} + 228x^{26} + 82x^{27} + 255x^{28} + 181x^{29} + x^{30}$$

$$g_{\text{hqc-192}}(x) = 45 + 216x + 239x^2 + 24x^3 + 253x^4 + 104x^5 + 27x^6 + 40x^7 + 107x^8 + 50x^9 + 163x^{10} + 210x^{11} + 227x^{12} + 134x^{13} + 224x^{14} + 158x^{15} + 119x^{16} + 13x^{17} + 158x^{18} + x^{19} + 238x^{20} + 164x^{21} + 82x^{22} + 43x^{23} + 15x^{24} + 232x^{25} + 246x^{26} + 142x^{27} + 50x^{28} + 189x^{29} + 29x^{30} + 232x^{31} + x^{32}$$

[1]Zeroth Coefficient of $g_{\text{hqc-128}}(x)$ is updated based on the software reference implementation given at https://pqc-hqc.org/implementation.html

$$g_{\text{hqc-256}}(x) = 49 + 167x + 49x^2 + 39x^3 + 200x^4 + 121x^5 + 124x^6 + 91x^7 + 240x^8 + 63x^9 + 148x^{10} + 71x^{11} + 150x^{12} + 123x^{13} + 87x^{14} + 101x^{15} + 32x^{16} + 215x^{17} + 159x^{18} + 71x^{19} + 201x^{20} + 115x^{21} + 97x^{22} + 210x^{23} + 186x^{24} + 183x^{25} + 141x^{26} + 217x^{27} + 123x^{28} + 12x^{29} + 31x^{30} + 243x^{31} + 180x^{32} + 219x^{33} + 152x^{34} + 239x^{35} + 99x^{36} + 141x^{37} + 4x^{38} + 246x^{39} + 191x^{40} + 144x^{41} + 8x^{42} + 232x^{43} + 47x^{44} + 27x^{45} + 141x^{46} + 178x^{47} + 130x^{48} + 64x^{49} + 124x^{50} + 47x^{51} + 39x^{52} + 188x^{53} + 216x^{54} + 48x^{55} + 199x^{56} + 187x^{57} + x^{58}$$

The generator polynomial can also be computed by $g(x) = \prod_{i=0}^{n-k-1}(x - \alpha^i)$, where α is the primitive element of the field.

Decode of shortened Reed–Solomon code. We denote the codeword to be $v(x) = v_0 + v_1x + \cdots + v_{n-1}x^{n-1}$, the error polynomial to be $e(x) = e_0 + e_1x + \cdots + e_{n-1}x^{n-1}$, and the received word to be $r(x) = r_0 + r_1x + \cdots + r_{n-1}x^{n-1}$. With these definitions, $r(x) = v(x) + e(x)$. The primitive element α of the field satisfies $v(\alpha^i) = 0$ for $i = 1, \ldots, 2d$ (notice $2d = n - k$), since $g(\alpha^i) = 0$ and $v(x) \bmod g(x) = 0$. If there is no error in the received word, $r(\alpha^i) = v(\alpha^i) = 0$, so $e(\alpha^i) = 0$. Otherwise, we can denote $r(\alpha^i) = e(\alpha^i) = e_{j_1}(\alpha^i)^{j_1} + \cdots + e_{j_t}(\alpha^i)^{j_t}$ where $e(x)$ has t errors at locations $j_1, \ldots, j_t$. Let us define:

$$S_i = r(\alpha^i) = e(\alpha^i) = e_{j_1}(\alpha^{j_1})^i + \cdots + e_{j_t}(\alpha^{j_t})^i, \quad i = 1, \ldots, 2d$$
$$\sigma(x) = (1 + \alpha^{j_1}x)(1 + \alpha^{j_2}x)\cdots(1 + \alpha^{j_t}x) = 1 + \sigma_1x + \sigma_2x^2 + \cdots + \sigma_tx^t$$
$$Z(x) = 1 + (S_1 + \sigma_1)x + (S_2 + \sigma_1S_1 + \sigma_2)x^2 + \cdots + (S_t + \sigma_1S_{t-1} + \sigma_2S_{t-2} + \cdots + \sigma_t)x^t$$

Then the error value at location j_l can be computed by:

$$e_{j_l} = \frac{Z((\alpha^{j_l})^{-1})}{\prod_{i=1,i\neq l}^{t}[1 + \alpha^{j_i} \cdot (\alpha^{j_l})^{-1}]}$$

The decode steps are:

1. Compute $S_i = r(\alpha^i), i = 1, \ldots, 2d$.
2. Because $\sigma((\alpha^{j_i})^{-1}) = 0,\ i = 1, \ldots, t$, the coefficients $\sigma_i,\ i = 1, \ldots, t$ can be calculated from the linear equation set: $0 = \sum_{i=1}^{t} e_{j_i}(\alpha^{j_i})^{l+t}\sigma((\alpha^{j_i})^{-1}) = S_{l+t} + \sigma_1S_{l+t-1} + \sigma_2S_{l+v-2} + \cdots + \sigma_tS_l,\ l = 0, \ldots, t-1$.
3. The roots of $\sigma(x)$ can be calculated, which are $(\alpha^{j_i})^{-1}, i = 1, \ldots, t$.
4. $Z((\alpha^{j_l})^{-1}),\ l = 1, \ldots t$ can be computed, and so do $e_{j_l}, l = 1, \ldots, t$.
5. The codeword can be computed by $v(x) = r(x) - e(x)$.

Encode of duplicated Reed–Muller code. Encode of duplicated Reed–Muller code is to directly perform a matrix vector multiplication. The generator matrix is shown below (note that numbers are big endian and in hexadecimal):

$$\mathbf{G} = \begin{pmatrix} \texttt{aaaaaaaa} & \texttt{aaaaaaaa} & \texttt{aaaaaaaa} & \texttt{aaaaaaaa} \\ \texttt{cccccccc} & \texttt{cccccccc} & \texttt{cccccccc} & \texttt{cccccccc} \\ \texttt{f0f0f0f0} & \texttt{f0f0f0f0} & \texttt{f0f0f0f0} & \texttt{f0f0f0f0} \\ \texttt{ff00ff00} & \texttt{ff00ff00} & \texttt{ff00ff00} & \texttt{ff00ff00} \\ \texttt{ffff0000} & \texttt{ffff0000} & \texttt{ffff0000} & \texttt{ffff0000} \\ \texttt{00000000} & \texttt{ffffffff} & \texttt{00000000} & \texttt{ffffffff} \\ \texttt{00000000} & \texttt{00000000} & \texttt{ffffffff} & \texttt{ffffffff} \\ \texttt{ffffffff} & \texttt{ffffffff} & \texttt{ffffffff} & \texttt{ffffffff} \end{pmatrix}$$

If the message is $\mathbf{m} = (m_0, \dots, m_7) \in \mathbb{F}_{2^8}$, then $\mathbf{c} = \mathbf{mG}$, and the codeword is given by duplicating $\mathbf{c}$ 3 or 5 times, depending on the security level.

Decode of duplicated Reed–Muller code. The decoding of duplicated Reed–Muller codes is done in three steps:

1. The first step is applying the function F on the received codeword. Let v be a duplicated Reed–Muller codeword with multiplicity 3, it can be seen as $v = (a_1b_1c_1, ..., a_{n_1}b_{n_1}c_{n_1})$ where each a_i, b_i, c_i has 128 bits size ($a_i = (a_{i_1}, ..., a_{i_{128}})$, $b_i = (b_{i_1}, ..., b_{i_{128}})$ and $c_i = (c_{i_1}, ..., c_{i_{128}})$). The transformation F is applied to each element in v as $((-1)^{a_{i_1}} + (-1)^{b_{i_1}} + (-1)^{c_{i_1}}, ..., (-1)^{a_{i_{128}}} + (-1)^{b_{i_{128}}} + (-1)^{c_{i_{128}}})$. For multiplicity 5, it follows the same process.

2. The second step is applying Hadamard transform on the output of the previous step.

3. The third step is finding the location of the highest value on the output of Hadamard transform. When the peak is positive, we add all-one-vector. If there are two identical peaks, we take the peak with smallest value in the lowest 7 bits.

2.2.2 HQC PKE

Key generation. First a vector $\mathbf{h}$ is sampled uniformly random, which is viewed as the vector to generate a circulant matrix and further a systematic quasi-cyclic code of index 2. More specifically, let $\mathbf{h} = (h_0, \dots, h_{n-1})$. Then

$$\mathbf{rot}(\mathbf{h}) = \begin{pmatrix} h_0 & h_{n-1} & \cdots & h_1 \\ h_1 & h_0 & \cdots & h_2 \\ \vdots & \vdots & \ddots & \vdots \\ h_{n-1} & h_{n-2} & \cdots & h_0 \end{pmatrix}$$

is a circulant matrix, and $\mathbf{H} = [\mathbf{I}|\mathbf{rot}(\mathbf{h})]$ is the parity-check matrix of a systematic quasi-cyclic code of index 2. The secret key is composed of two vectors $\mathbf{x}, \mathbf{y}$ that are sampled with a specified weight w. $[\mathbf{x}|\mathbf{y}]$ can be viewed as a random codeword with a random error. Its syndrome is $\mathbf{s} = [\mathbf{x}|\mathbf{y}]\mathbf{H}^T = \mathbf{x} + \mathbf{y} \times \mathbf{rot}(\mathbf{h})^T = \mathbf{x} + \mathbf{h} \cdot \mathbf{y}$, where $\mathbf{y} \times \mathbf{rot}(\mathbf{h})^T$ is defined as the general vector-matrix multiplication. The public key is composed of $\mathbf{h}$ and the syndrome $\mathbf{s}$.

Encryption. Similar to key generation, three vectors $\mathbf{r}_1, \mathbf{r}_2, \mathbf{e}$ are sampled with a specified weight w_r. Then the syndrome $\mathbf{u}$ of $[\mathbf{r}_1, \mathbf{r}_2]$ is computed. The message is encoded by concatenated Reed–Muller and Reed–Solomon code introduced previously, as well as added by $\mathbf{s} \cdot \mathbf{r}_2 + \mathbf{e}$ to get $\mathbf{v}$. The final ciphertext comprises of $\mathbf{u}$ and $\mathbf{v}$, i.e., $\mathbf{c} = [\mathbf{u}|\mathbf{v}]$.

Decryption. $\mathbf{v} - \mathbf{u} \cdot \mathbf{y}$ is decoded by concatenated Reed–Muller and Reed–Solomon code. The message can be correctly decoded whenever the Hamming weight of the given element is less than the minimum distance of the code. The probability that the message cannot be decoded from $\mathbf{v} - \mathbf{u} \cdot \mathbf{y}$ is shown to be very small [AAB+20].

Algorithm 1 HQC.PKE.KeyGen() and HQC.KEM.KeyGen()

```
1: $\mathbf{h} \leftarrow \mathcal{R}$
2: $(\mathbf{x}, \mathbf{y}) \overset{w}{\leftarrow} \mathcal{R}^2$
3: $\mathbf{s} := \mathbf{x} + \mathbf{h} \cdot \mathbf{y}$
4: return (pk := (h, s), sk := (x, y))
```

Algorithm 2 HQC.PKE.Encrypt(pk $= (\mathbf{h}, \mathbf{s})$, $\mathbf{m}$, θ)

```
1: $(\mathbf{r}_1, \mathbf{r}_2, \mathbf{e}) \xleftarrow{\omega_r, \theta} \mathcal{R}^3$
2: $\mathbf{u} := \mathbf{r}_1 + \mathbf{h} \cdot \mathbf{r}_2$
3: $\mathbf{t} := \text{Encode}(\mathbf{m})$
4: $\mathbf{v} := \mathbf{t} + \mathbf{s} \cdot \mathbf{r}_2 + \mathbf{e}$
5: return $\mathbf{c} := (\mathbf{u}, \mathbf{v})$
```

Algorithm 3 HQC.PKE.Decrypt(sk $= (\mathbf{x}, \mathbf{y})$, $\mathbf{c} = (\mathbf{u}, \mathbf{v})$)

```
1: $\mathbf{m}' := \text{Decode}(\mathbf{v} - \mathbf{u} \cdot \mathbf{y})$
2: return $\mathbf{m}'$
```

2.2.3 HQC KEM

Key generation. Key generation in KEM is the same as key generation in PKE.
Encapsulation. The message $\mathbf{m}$ is sampled to generate the shared secret. The random seed $\theta = \mathcal{G}(\mathbf{m})$, which will be used in the encryption to control the randomness. $\mathbf{m}$ is then encrypted to generate $\mathbf{c}$. Finally, the shared secret $K = \mathcal{K}(\mathbf{m}, \mathbf{c})$, and the ciphertext is $[\mathbf{c}|\mathcal{H}(\mathbf{m})]$.
Decapsulation. $\mathbf{c}$ is used to retrieve the message $\mathbf{m}'$. The decryption process may not be correct and thus returns a wrong message. Therefore, the same process as encapsulation needs to be done and the ciphertext needs to be checked with the received ciphertext. Finally, whether there are mistakes is returned.

Algorithm 4 HQC.KEM.Encapsulate(pk $= (\mathbf{h}, \mathbf{s})$)

```
1: $\mathbf{m} \leftarrow \mathbb{F}_2^k$
2: $\theta := \mathcal{G}(\mathbf{m})$
3: $\mathbf{c} := (\mathbf{u}, \mathbf{v}) = \text{HQC.PKE.Encrypt}(\text{pk}, \mathbf{m}, \theta)$
4: $K := \mathcal{K}(\mathbf{m}, \mathbf{c})$
5: $\mathbf{d} := \mathcal{H}(\mathbf{m})$
6: return $(K, (\mathbf{c}, \mathbf{d}))$
```

3 Hardware Design of HQC

HQC Key Encapsulation Mechanism (HQC-KEM) consists of three main primitives: Key Generation, Encapsulation, and Decapsulation. The algorithms for each primitive were shown in Algorithm 1, Algorithm 4, and Algorithm 5, respectively. These primitives are built upon the HQC Public Key Encryption (HQC-PKE) primitives shown in Algorithm 1, Algorithm 2, and Algorithm 3. Which in turn are built upon other, more basic building blocks. In this work, we implement optimized and parameterizable hardware designs for all the primitives and the building blocks from scratch. In the following subsections we briefly discuss all the building blocks and provide comparisons with any existing designs. The main building blocks involved for each of the primitives are as follows:

- Key Generation: Fixed weight vector generator, PRNG based random vector generator, polynomial multiplication, modular addition, and SHAKE256
- Encapsulation: Encrypt, SHAKE256
- Decapsulation: Decrypt, Encrypt, SHAKE256

Algorithm 5 HQC.KEM.Decapsulate(sk = (**x**, **y**), **c**, **d**)

```
1: $\mathbf{m}' \coloneqq$ HQC.PKE.Decrypt(sk, $\mathbf{c}$)
2: $\theta' \coloneqq \mathcal{G}(\mathbf{m}')$
3: $\mathbf{c}' \coloneqq (\mathbf{u}', \mathbf{v}') =$ HQC.PKE.Encrypt(pk, $\mathbf{m}', \theta'$)
4: $\mathbf{d}' \coloneqq \mathcal{H}(\mathbf{m}')$
5: $K' \coloneqq \mathcal{K}(\mathbf{m}', \mathbf{c})$
6: if $\mathbf{c} \neq \mathbf{c}'$ or $\mathbf{d} \neq \mathbf{d}'$ then
7:     return $(K', 0)$
8: else
9:     return $(K', 1)$
10: end if
```

Table 1: Parameter sets for HQC. n is the length of the vector (polynomial). n_1 is the length of the Reed–Solomon code. n_2 is the length of the Reed–Muller code. w is the weight of vectors $\mathbf{x}, \mathbf{y}$. w_r is the weight of vectors $\mathbf{r}_1, \mathbf{r}_2, \mathbf{e}$. $[n, k, d]$ of Reed–Solomon and Reed–Muller codes are shown in the last two columns, and they are the length, the dimension, and the minimum distance of the code. In HQC, shortened Reed–Solomon code and duplicated Reed–Muller code are used. The multiplicity for duplicated Reed–Muller code is 3, 5, 5 for hqc-128, hqc-192, hqc-256.

Instance	n	w	w_r	security	p_{fail}	Reed–Solomon	Reed–Muller
hqc128	17,669	66	75	128	$< 2^{-128}$	[46, 16, 15]	[384, 8, 192]
hqc192	35,851	100	114	192	$< 2^{-192}$	[56, 24, 16]	[640, 8, 320]
hqc256	57,637	131	149	256	$< 2^{-256}$	[90, 32, 29]	[640, 8, 320]

3.1 Modules Common Across the Design

In this section we give a high-level overview of hardware designs of the building blocks that are used across the HQC-KEM and HQC-PKE.

3.1.1 SHAKE256

HQC uses SHAKE256 for multiple purposes e.g., as a PRNG for fixed weight vector generation and random vector generation in Key Generation, as a PRNG for fixed weight vector generation in Encryption, and for hashing in encapsulation and decapsulation. We use the SHAKE256 module described in [CCD+22] (which was originally designed based on `Keccak` design from [WTJ+20]) to perform SHAKE256 operations. We further tailor the SHAKE256 hardware module as per the requirement for our hardware design:

- The existing `SHAKE256` module [CCD+22] operates with command based interface where the number of input bytes to be processed and number of output bytes required are specified before starting the hash operation and there is no command to request for additional bytes. We modify the exiting design and add an additional command which provides the capability of requesting additional bytes. The purpose of adding this command is to support the fixed weight vector generation process described in Section 3.1.4.
- Since our modification of `SHAKE256` holds the current state and does not automatically return to its new input loading state, we modify the operation of the existing forced exit signal to return the SHAKE256 module to default state. To support the dual clock domain design described in Section 3.7, we also add a forced exit acknowledgement port.

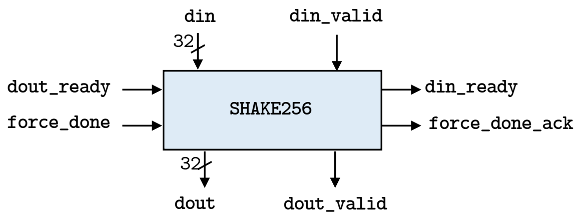


Figure 1: Interface for SHAKE256 module.

Table 2: `SHAKE256` module area and timing information, data based on synthesis results for Artix 7 board with `xc7a200t-3` FPGA chip. Formula for time-area product, T $\times$ A, is (LUT * Time)/10^3.

Parallel Slices	Resources				F	Cycles	Time	T x A
	Logic		Memory					
	(LUT)	(DSP)	(FF)	(BR)	(MHz)	(cyc.)	(us)	
1	1,437	0	498	0	163	5,010	30.74	44
2	1,558	0	466	0	167	2,306	13.81	21
4	1,625	0	370	0	157	1,086	6.92	11
8	1,958	0	280	0	158	542	3.43	6
16	2,819	0	236	0	164	270	1.65	4

We use a similar performance parameter `parallel_slices` as described in the original `keccak` design in [WTJ+20]. The `SHAKE256` module has a fixed 32-bit data ports and data input and output is based on typical ready-valid protocol. The results targeting Xilinx Artix 7 `xc7a200t` FPGA are shown in Table 2. The clock cycle numbers provided in the Table 2 are for processing one block of input and generating one block of output (where each block size is 1088-bits). There are five different options to chose for the `parallel_slices` which provide different time-area trade-offs. We choose `parallel_slices = 16` as it provides the best time-area product. An interface diagram of the `SHAKE256` module is shown in Figure 1. For brevity, we represent all the ports interfacing with the `SHAKE256` module with $\Leftrightarrow$ in all further block diagrams in this paper.

3.1.2 Polynomial Multiplication

HQC uses polynomial multiplication operation in all the primitives of HQC-KEM. The polynomial multiplication operation is multiplication of two polynomials with n components in $\mathbb{F}_2$. Our polynomial multiplication module uses sparse polynomial multiplication technique followed by a specific modular reductions to $X^n - 1$ (values of n can be found in Table 1).

After profiling all the polynomial multiplication operations from the HQC specification document and the reference design [AAB+20], we note that in all the polynomial multiplication operations, one of the inputs is a sparse fixed weight vector (with weight w or w_r in Table 1) of width n-bits. Consequently, we design a position based polynomial multiplication unit where one of the inputs to the module is a non-sparse polynomial ('A') while the other input is a list of the positions of ones from the sparse fixed-weight vector polynomial ('B'). The multiplication is performed by shifting 'A' with each position of 'B' and then performing addition of all the resultant vectors. Since the value of n is large in all parameter sets, we take a sequential approach for performing the multiplication. We design our sequential shifter similar to 'SVS module' described in [DdPM+21]. The polynomial multiplication module works as follows: each time the input at the port `din` is sequentially shifted by position given at `shift_pos_in` and resultant vector is added with

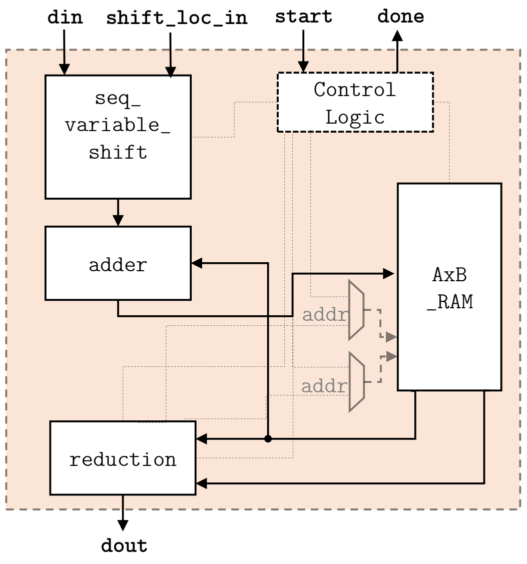


Figure 2: Hardware design of `poly_mult` module.

Table 3: `poly_mult` module (with datapath width 128-bits) area and timing information, data based on synthesis results for Artix 7 board with `xc7a200t-3` FPGA chip.

Input Length	Resources							
	Logic		Memory		F	Cycles	Time	T x A
(bits)	(LUT)	(DSP)	(FF)	(BR)	(MHz)	(cyc.)	(us)	
17,669	1,834	0	573	4	228	18,765	82.30	150
35,851	1,821	0	587	6	215	56,969	264.97	482
57,637	1,837	0	606	6	203	119,333	587.85	1,080

the previous result and stored in `AxB_RAM`. Since all the operations take place in $\mathbb{F}_2$ the addition is achieved by performing an `exclusive-OR` operation. The maximum resultant vector size after multiplying two n-bit polynomials is $2n$-bit polynomial.

The $2n$-bit resultant polynomial is reduced using the `reduction` module shown in Figure 2. The polynomial needs to be reduced to back to a n-bit polynomial and to do so we slice the n-bit polynomial into two parts and then perform an addition. And since operations take place in $\mathbb{F}_2$ the addition is achieved by performing an `exclusive-OR` operation. The `reduction` module sequentially performs the slicing and addition. We make the width of sequential shifter as a performance parameter based on which the width of `reduction` module, `adder` module and the width and depth of `AxB RAM` is chosen. A user can choose the width of sequential shifter based on suitable time-area trade off. Results of our polynomial multiplication module for one performance parameter (width = 128) are shown in Table 3.

3.1.3 Polynomial Addition/Subtraction

HQC uses polynomial addition/subtraction in all of its primitives. Since all addition and subtraction operations happen in $\mathbb{F}_2$, the addition and subtraction could be realized as the same operation. We design two variants of constant-time adders namely `xor_based_adder` and `location_based_adder` that could be attached with our polynomial multiplication module described in Section 3.1.2. We design our adder modules as an extension for polynomial multiplication because the addition/subtraction always appears with the

Table 4: Polynomial addition modules (`xor_based_adder` and `loc_based_adder` with datapath width 128-bits) area and timing information, data based on synthesis results for Artix 7 board with `xc7a200t-3` FPGA chip.

Input Length	Resources				F	Cycles	Time	T x A
	Logic		Memory					
(bits)	(LUT)	(DSP)	(FF)	(BR)	(MHz)	(cyc.)	(us)	
			`xor_based_adder`					
17,669	143	0	159	0	330	142	0.43	0.06
35,851	142	0	161	0	318	284	0.89	0.12
57,637	142	0	161	0	311	455	1.46	0.20
			`loc_based_adder`					
17,669	160	0	174	0	316	69	0.22	0.03
35,851	161	0	174	0	300	103	0.34	0.05
57,637	161	0	175	0	300	134	0.45	0.07

polynomial multiplication as shown in Algorithm 1, Algorithm 2, and Algorithm 3. The adders operate on contents of block RAM since the polynomials are stored inside the block RAM. Both of the adder module designs do not use any additional block RAM resources, they load the polynomial multiplication output, perform the addition, and write the value back to the same block RAM inside the polynomial.

The `xor_based_adder` design performs addition in a regular $\mathbb{F}_2$ fashion by performing bit-wise `exclusive-OR` operation. The module performs addition sequentially by generating one block RAM address per clock cycle to load inputs from two block RAMs and then performs addition and writes them back to one of the specified block RAMs at the same block RAM address.

The `location_based_adder` is an optimized adder designed to perform addition when one of the input is a sparse vector. This module is mainly designed to perform operations $\mathbf{x}+\mathbf{h}\cdot\mathbf{y}$ from Algorithm 1 and $\mathbf{r}_1+\mathbf{h}\cdot\mathbf{r}_2$ and $\mathbf{s}\cdot\mathbf{r}_2+\mathbf{e}$ from Algorithm 2. In these operations the values of $\mathbf{x}$, $\mathbf{r}_1$, and $\mathbf{e}$ are sparse, fixed-weight vectors so the addition is optimized by only flipping the bits of the other input in the position of one. The `location_based_adder` module takes location of ones from the sparse vector as input and computes the address to load out the part of non-sparse polynomial from the block RAM and flips the bit on the appropriate location and writes it back to the same location. The process is repeated until all locations with ones are covered. Since there are a fixed, and known number of ones in the fixed-weight vector, there is a fixed number of operations and timing does not reveal any sensitive information. Results of our polynomial addition `location_based_adder` module for one performance parameter (width = 128) are shown in Table 4.

3.1.4 Fixed-Weight Vector Generator

The fixed-weight vector generator function generates a uniform random n-bit fixed-weight vector of a specified input weight (w). The module assumes that there is a random number generator that can be used to generate uniformly random bits. The algorithm for fixed-weight generation as specified in [AAB+20] first generates $24 \times w$ random bits. These random bits are then arranged into w 24-bit integers. These 24-bit integers undergo a threshold check and are rejected if the integer value is beyond the threshold ($949 \times 17,669$, $467 \times 35,851$, $291 \times 57,637$ for hqc-128, hqc-192 and hqc-256 respectively). After the threshold check, these integers are reduced modulo n. After the threshold check and reduction process if the weight is not equal to w then more random bits are drawn from RNG and the process is repeated until w integers are achieved. After the threshold check and reduction then a check for duplicates is performed over all the reduced integers. In

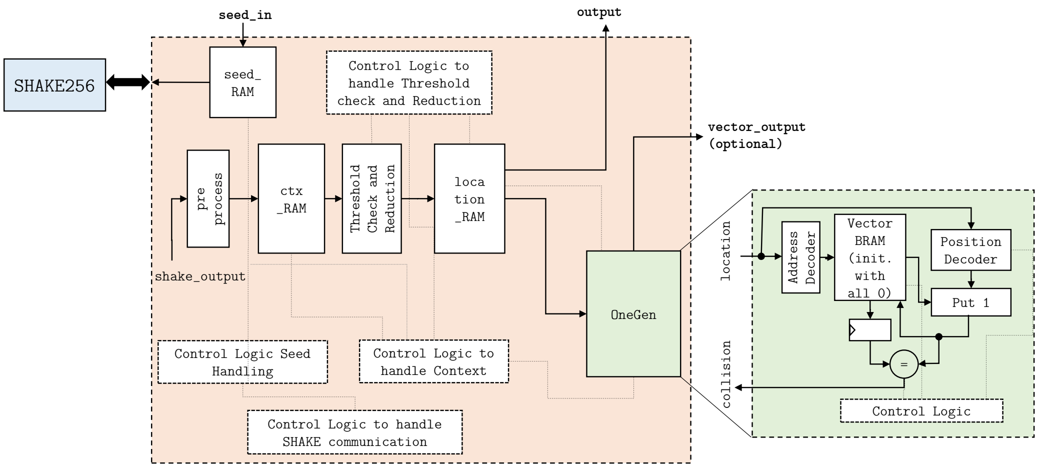

seed_in
output
SHAKE256
seed_
RAM
Control Logic to handle Threshold check and Reduction
vector_output (optional)
pre process
ctx _RAM
Threshold Check and Reduction
loca tion _RAM
shake_output
OneGen
location
collision
Address Decoder
Vector BRAM (init. with all 0)
Position Decoder
Put 1
=
Control Logic
Control Logic Seed Handling
Control Logic to handle Context
Control Logic to handle SHAKE communication

Figure 3: Hardware design of `fixed_weight_vector` module.

Table 5: `fixed_weight_vector` module area and timing information, data based on synthesis results for Artix 7 board with `xc7a200t-3` FPGA chip and probability of failing constant-time behavior of our `fixed_weight_vector` generation module best and worst case values for the parameter, `ACCEPTABLE_REJECTIONS`.

Design	Weight	Resources: Logic (LUT)	Memory (FF)	(BR)	F (MHz)	Cycles (cyc.)	Time (us)	T x A	Failure^{+} Prob.
non constant-time design (`ACCEPTABLE_REJECTIONS` = 0)									
hqc128	75	240	111	2.0	226	709	3.14	0.75	1.1×2^{-11}
hqc192	114	229	112	2.0	220	1,840	8.36	1.92	1.1×2^{-9}
hqc256	149	234	117	2.0	228	2,106	9.24	2.16	1.1×2^{-12}
constant-time design (`ACCEPTABLE_REJECTIONS` = w_r)									
hqc128	75	316	124	2.0	223	3,649	16.36	5.17	2.8×2^{-199}
hqc192	114	295	125	2.0	236	4,200	17.80	5.25	1.1×2^{-280}
hqc256	149	314	192	2.5	242	5,935	24.52	7.70	4.9×2^{-355}

$^{+}$ = Probability of our design failing to behave constant-time.

case any duplicate is found, that integer is discarded and more random bits are requested drawn from the RNG which again undergo threshold check, reduction and duplicate check. This process is repeated until a uniform fixed weight vector is generated.

In our hardware design, we use a PRNG to generate the uniformly random bits required for the fixed weight vector generation from an input seed of length 320-bits. Our hardware design includes this PRNG in the form of SHAKE256. Our design assumes that the seed will be initialized by some other hardware module implementing a true random number generator.

The hardware design of `fixed_weight_vector` generation module is shown in Figure 3. We use `SHAKE256` module described in Section 3.1.1 to expand 320-bit seed to a $24 \times w$-bit string. Since the `SHAKE256` module has 32-bit interface the seed is loaded in 32-bit chunks and the seed is stored in `seed_RAM` as shown in the Figure 3. The 32-bit chunk from SHAKE256 is broken into 24-bit integer by `preprocess` unit and stored in the `ctx_RAM` then threshold check and reduction are performed. For the reduction, we use Barrett reduction. The Barrett reduction is optimized to reduce for specific value (n) since that value is constant. After the reduction, the integer values are stored in the `locations_RAM`.

Once the `locations_RAM` is filled the `OneGen` module is triggered. The `OneGen` module helps in detecting if there are any duplicates in the `locations_RAM`. Our `OneGen` module is inspired from duplicate checking logic described in [CCD+22]. While the `OneGen` module checks for duplicates, the `SHAKE256` module generates the next $24 \times w$-bit string to tackle any potential duplicates and stores them in the `ctx_RAM`. This way we are able to mask any clock cycles taken for seed expansion.

The main pitfall where the fixed-weight vector generation process may show non constant-time behavior is the rejection sampling process (i.e., the threshold check and duplicate detection as discussed earlier). A timing attack on existing software reference implementation of HQC [AAB+20] was performed in [GHJ+22]. The authors use the information of rejection sampling routine (that is part of fixed-weight generation) being invoked during the deterministic re-encryption process in decapsulation and show that this leaks secret-dependent timing information. The timing of the rejection sampling routine depends upon the given seed. This seed is derived for the encrypt function in encapsulation and decapsulation procedures using the message. The decapsulation operation is dependent on the decoded message and this dependency allows to construct a plaintext distinguisher (described in detail in [GHJ+22]) which is then used to mount the timing attack.

In our hardware module, we make the constant time behavior parameterizable (parameter name is `ACCEPTABLE_REJECTIONS`). We can specify how many indices could be rejected and still the design will behave constant time (at the cost of extra area for more storage and extra cycles). The extra area is because we generate additional (based on parameter value) uniformly random bits in advance and store them in the `ctx_RAM` (shown in Figure 3). The extra clock cycles are because even after we found the required number of indices under the threshold value, we still go over all the `ctx_RAM` locations and for the duplicate detection logic inside `OneGen` module (shown in Figure 3), the control logic is programmed to take the same cycles in case of duplicate is detected or not. The parameter can be set based on user's target failure probability. If the actual failures are within the failure probability set by the selected parameter value, then the timing side channel given in [GHJ+22] is not possible.

The right most column in Table 5 shows probability of non constant-time design (`ACCEPTABLE_REJECTIONS` $= 0$) versus constant-time design (`ACCEPTABLE_REJECTIONS` $= w_r$) failing to behave in constant-time manner. The choice of the demonstrated parameter is made based on the values of w_r given in Table 1. We choose w_r as the parameter value for demonstration because when the rejection sampling procedure rejects the first index the *seedexapander* function in software reference implementation [AAB+20] generates w_r new set of indices and first index from the new set is used to replace the old rejected index and for the second rejection next index from the new set is chosen to replace the old rejected index. This process is repeated until specified weight for the vector is achieved. To compute the failure probability (given in Table 5) for each parameter set, we take in to account both threshold check failure and duplicate detection probabilities for the respective parameter sets.

Table 5 shows the results of the fixed weight hardware module targeting Artix 7 `xc7a200t` FPGA. The area excludes `SHAKE256` module as the SHAKE256 is shared among all primitives. The reported frequency in Table 5 is the core frequency. We discuss later our dual clock domain design (in Section 3.7) that allows the modules to run at their core frequency while `SHAKE256` runs on a slower clock.

3.2 Encode and Decode Modules

The encode and decode modules are building blocks of the encrypt and decrypt modules, respectively. We describe the encode and decode modules here, before describing the bigger encrypt and decrypt modules in Section 3.3.

3.2.1 Encode Module

As specified in Section 2.2.1, HQC Encode uses concatenation of two codes namely Reed–Muller and Reed–Solomon codes. The Encode function takes K-bit input and first encodes it with the Reed–Solomon code. The Reed–Solomon encoding process involves systematic encoding using a linear feedback shift register (LFSR) with a feedback connection based on the generator polynomial (shown in Section 2.2.1). The Reed–Solomon code generates a n_1-bit output (as given in [AAB+20] the value for n_1 is 368, 448, and 720 for `hqc128`, `hqc192`, and `hqc256` respectively). We design a Galois field multiplication unit for the field $\mathbb{F}_2[x]/(x^8 + x^4 + x^3 + x^2 + 1)$ which takes one byte of input per clock cycle and multiplies it with the generator polynomial (g_x). The number of Galois field multiplication units we run in parallel is equal to the degree of the generator polynomial. The outputs from Galois field multipliers are fed in to a LFSR after each cycle. At the end of encoding process the module generates a n_1-bit output.

The n_1-bit output from Reed–Solomon code is then encoded by Reed–Muller code. The Reed–Muller encoding is achieved by performing vector-matrix multiplication where each byte from input is the vector and the matrix is generator matrix ($\mathbf{G}$) given in Section 2.2.1. In our design we store the generator matrix rows (each row is of length 128-bits) in ROM and we select the matrix rows based on each input byte. We store the output after multiplying input byte into a block RAM in chunks of 128-bits. Based on the security parameter set the code word output from Reed–Muller code has a multiplicity value (i.e., number of times a code word or in our case number of times each block RAM location is repeated). As per the specification [AAB+20], `hqc128` has multiplicity value of 3 and `hqc192` and `hqc256` have multiplicity value of 5. To optimize the storage, we only store one copy of code word, and while accessing the code word we compute the block RAM address in a way that the multiplicity is achieved.

3.2.2 Decode Module

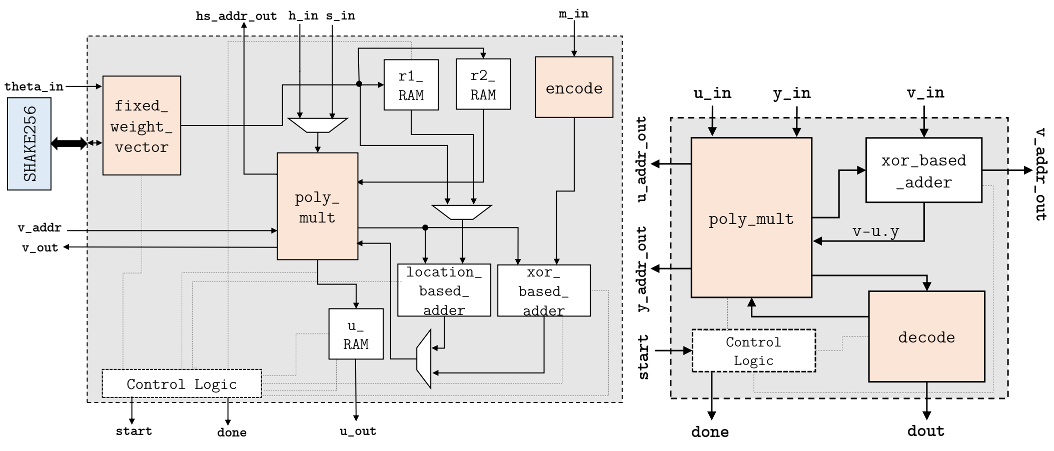
As introduced in Section 2.2.1, the ciphertext is first decoded with duplicated Reed-Muller code and then shortened Reed-Solomon code. To decode duplicated Reed-Muller code, the `Transformation` module expands and adds multiple code words into expanded code word, and then `Hadamard_Transformation` module applies Hadamard transformation to the expanded code word. Finally, `Find_Peak` finds the location of the highest absolute value of the `Hadamard_Transformation` output. To decode Reed-Solomon code, we need to sequentially compute syndromes S_i, coefficients σ_i of error location polynomial $\sigma(x)$, roots of error location polynomial $(\alpha^i)^{-1}$, pre-defined helper polynomial $Z((\alpha^i)^{-1})$, errors e_i, and finally correct the output of decode of Reed-Muller code based on the errors.

3.3 Encrypt and Decrypt Modules

The encrypt and decrypt modules are building blocks of the encapsulation and decapsulation modules, respectively. We describe the encrypt and decrypt modules here, before describing the bigger encapsulation and decapsulation modules later.

3.3.1 Encrypt Module

The `encrypt` module (shown in Algorithm 2) takes public key ($\mathbf{h}$, $\mathbf{s}$), message $\mathbf{m}$, and seed (θ) and generates a ciphertext ($\mathbf{u}$,$\mathbf{v}$) as the output. The hardware design for the `encrypt` module is shown in Figure 4a. We use `fixed_weight_vector` hardware module described in Section 3.1.4 to generate $\mathbf{r}_1$, $\mathbf{r}_2$, and $\mathbf{e}$ fixed-weight vectors of weight w_r by expanding `theta_in` and in parallel we run `encode` module (described in Section 3.2.1). After the generation of $\mathbf{r}_2$ we start the polynomial multiplication of $\mathbf{h}.\mathbf{r}_2$ in parallel to the $\mathbf{e}$ generation. For polynomial multiplication we use the `poly_mult` with the datapath



(a) `encrypt` module. **(b)** `decrypt` module.

Figure 4: Hardware design of `encrypt` and `decrypt` modules.

Table 6: `encrypt` module area and timing information, data based on synthesis results for Artix 7 board with `xc7a200t-3` FPGA chip.

	Resources[†]							
Design	**Logic**		**Memory**		**F**	**Cycles**	**Time**	**T x A**
	(LUT)	(DSP)	(FF)	(BR)	(MHz)	(cyc.)	(us)	
`hqc128`	2,245	0	1,667	13	245	44,982	183	411
`hqc192`	2,437	0	1,855	13	245	133,586	543	1,324
`hqc256`	2,581	0	2,978	13	247	277,083	1,119	2,888

[†] = Given resources does not include the area for `SHAKE256` module.

width of 128-bits module described in Section 3.1.2. The addition of $\mathbf{r}_1$ in $\mathbf{u}$ computation and $\mathbf{e}$ in $\mathbf{v}$ computation is performed by our `location_based_adder` and addition with $\mathbf{t}$ is performed by `xor_based_adder` (described in Section 3.1.3).

Table 6 shows our hardware implementation results targeting Xilinx Artix 7 `xc7a200t` FPGA. The area results do not include the `SHAKE256` module for the same reason as described in Section 3.1.4. We note that, although in the Figure 4a we only show one `poly_mult` module, we provide a choice of using two `poly_mult` modules running in parallel this would reduce the number of clock cycles by approximately half but with some increase in the area.

3.3.2 Decrypt Module

The `decrypt` module (shown in Algorithm 3) takes secret key ($\mathbf{x}$, $\mathbf{y}$), ciphertext ($\mathbf{u}$,$\mathbf{v}$), and generates the message ($\mathbf{m'}$). Figure 4b shows our hardware design for `decrypt` module. The module accepts part of the secret key ($\mathbf{y}$) as locations with ones (since it is a sparse fixed weight vector). We use our `poly_mult` module described in Section 3.1.2 to compute $\mathbf{u.y}$ and use `xor_based_adder` module (described in Section 3.1.3) to compute $\mathbf{v} - \mathbf{u.y}$. We then use the `decode` module (described in Section 3.2.2) to decode $\mathbf{v} - \mathbf{u.y}$ and retrieve the message. Table 7 shows our hardware implementation results for `decrypt` module targeting Xilinx Artix 7 `xc7a200t` FPGA.

Table 7: `decrypt` module area and timing information, data based on synthesis results for Artix 7 board with `xc7a200t-3` FPGA chip.

Design	Resources†							
	Logic		Memory		F	Cycles	Time	T x A
	(LUT)	(DSP)	(FF)	(BR)	(MHz)	(cyc.)	(us)	
`hqc128`	5,747	0	4,801	12.5	204	24,889	0.12	701
`hqc192`	6,242	0	5,705	12.5	148	64,099	0.43	2,703
`hqc256`	7,488	0	7,248	15	156	130,313	0.84	6,255

† = Given resources does not include the area for `SHAKE256` module.

3.4 Key Generation

We now begin to describe the top-level modules, starting with the key generation, following in later sections with encapsulation and decapsulation. The discussion here focuses on single clock domain design. The novel dual clock domain design which allows the top-level modules run at different frequencies from the `SHAKE256` module that they depend on is described in Section 3.7.

The key generation (shown in Algorithm 1) takes secret key seed and public key seed as an input and generates secret key (**x, y**) and public key (**h, s**) respectively as output. Figure 5 shows the hardware design of our `keygen` module. Our `keygen` module assumes that the public key seed and the secret key seed are generated by some other hardware module implementing a true random number generator. We use `fixed_weight_vector` module described in Section 3.1.4 to generate (**x, y**) from the secret key seed. **x** and **y** are fixed weight vectors of weight w and length n-bits. To optimize the storage, rather than storing full n-bit sparse vector we only output locations of ones. There is also an optional provision to output the full vector as described in Section 3.1.4. The `vector_set_random` uses the `SHAKE256` module to expand the public key seed and generates **h**. We then use `poly_mult` module (described in Section 3.1.2) to compute (**h.y** and finally use `location_based_adder` module (described in Section 3.1.3) to compute **s**. We note that in the Figure 5 only a block RAM for **x** storage (`X_RAM`) is visible because the $\mathbf{y}, \mathbf{h}, \mathbf{s}$ are stored in the block RAMs which are inside `fixed_weight_vector`, `poly_mult`, and `location_based_adder` modules respectively.

Table 8 shows the results for the `keygen` module. We note that the maximum clock frequency of `keygen` module alone (without `SHAKE256`) module is in range 241-248 MHz based on the parameter set selected, but since the critical path lies inside the `SHAKE256` module we report SHAKE256's frequency in the Table 8. The area results do not include the `SHAKE256` module for the same reason as described in Section 3.1.4. Our dual clock domain design can be applied to the key generation to run the key generation and SHAKE256 at two different frequencies (discussed in Section 3.7).

3.5 Encapsulation Module

The encapsulate operation (shown in Algorithm 4) takes public key (**h, s**) and message **m** as an input and generates shared secret (K) and ciphertext (**c** = (**u**,**v**)) and **d**. The hardware design of the `encap` module is shown in Figure 6a. Our `encap` module assumes that **m** is generated by some other hardware module implementing a true random number generator and provided as an input to our module. Since the `SHAKE256` module is extensively used in encapsulate operation we design a `HASH_processor` module which handles all the communication with the `SHAKE256` module. `HASH_processor` modules reduces the multiplexing logic of inputs to the `SHAKE256` module significantly.

The `Hash_processor` modules helps in expanding **m** to generate θ. We then use

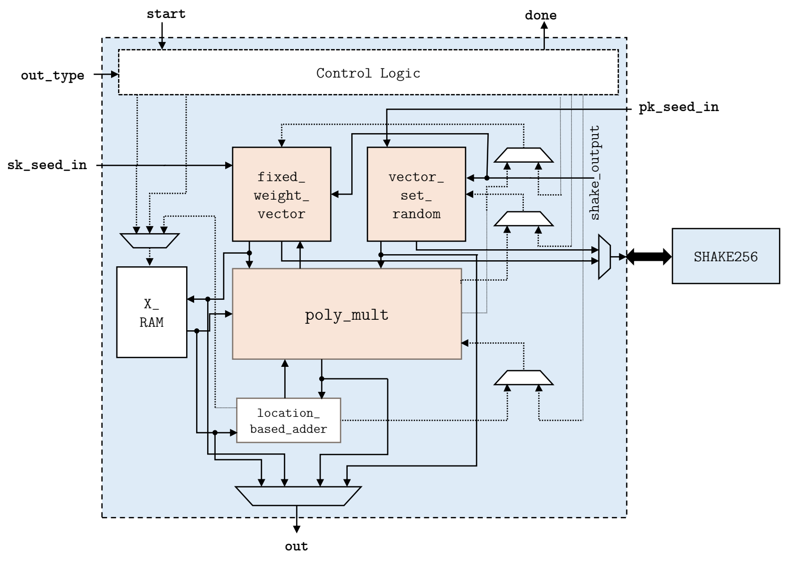


Figure 5: Hardware design of the `keygen` module.

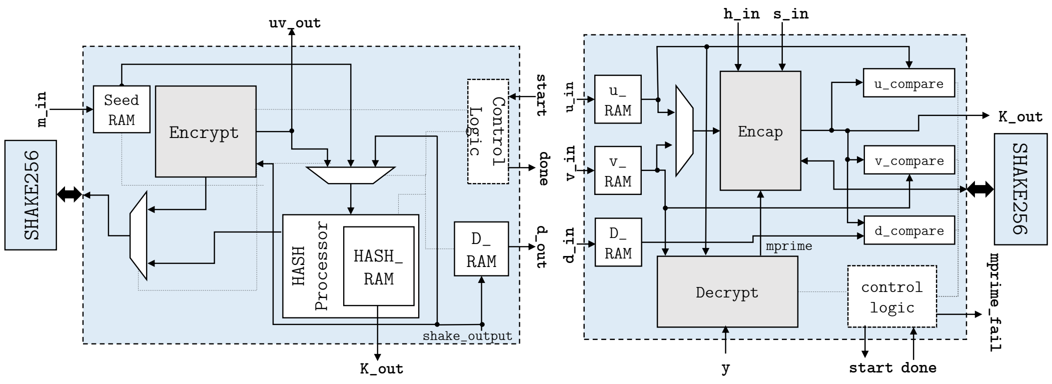
Table 8: `keygen` module area and timing information, data based on synthesis results for Artix 7 board with `xc7a200t-3` FPGA chip. Data for single clock design, using slow clock due to hash module critical path.

Design	Resources†								
	Logic		Memory		F	Cycles	Time	T x A
	(LUT)	(DSP)	(FF)	(BR)	(MHz)	(cyc.)	(ms)	
`hqc128` OUR	2,350	0	1,106	9.5	164	23,480	0.14	336
`hqc192` OUR	1,221	0	834	13.5	164	65,446	0.39	476
`hqc256` OUR	1,348	0	876	13.5	164	132,720	0.81	1,091
`hqc128-perf HLS`* [AAB+20]	12,000	0	9,000	3	150	40,000	0.27	3,240
`hqc128-comp. HLS`* [AAB+20]	4,700	0	2,700	3	129	630,000	4.80	22,560
`hqc128-pure HLS`* [Ong2x]	53,513	0	21,158	7	171	50,662	0.33	17,873
`hqc128-opt. HLS`* [Ong2x]	53,534	0	21,155	6	171	50,658	0.33	17,880

† = Given resources does not include the area for `SHAKE256` module, * = Target FGPA is Artix-7 `xc7a100t-1`

our `encrypt` module (described in Section 3.3.1) to encrypt $\mathbf{m}$ using θ and the public key as inputs and generates ciphertext. After the generation of $\mathbf{r}_1$, $\mathbf{r}_2$, and $\mathbf{e}$ inside the `encrypt` module (described in Section 3.3.1) we then run `HASH_processor` module in parallel to `encrypt` module to generate $\mathbf{d}$. After the encryption of $\mathbf{m}$ we then use the `HASH_processor` to compute $\mathcal{K}(\mathbf{m}, \mathbf{c})$ to generate the shared secret K.

Table 9 shows the results for the `encap` module. We note that the maximum clock frequency of `encap` module alone (without `SHAKE256`) module is in range 208-218 MHz based on the parameter set selected but since the critical path lies inside the `SHAKE256` module we report that frequency in the Table 9. The area results do not include the `SHAKE256` module for the same reason as described in Section 3.1.4. The design can be further improved using our dual clock domain design discussed later, where module and SHAKE256 run on different clocks.



(a) `encap` module. **(b)** `decap` module.

Figure 6: Hardware design of `encap` and `decap` modules.

Table 9: `encap` module area and timing information, data based on synthesis results for Artix 7 board with `xc7a200t-3` FPGA chip. Design with single clock, using the slow clock due to hash module critical path.

Design	Resources† Logic (LUT)	(DSP)	Memory (FF)	(BR)	F (MHz)	Cycles (cyc.)	Time (ms)	T x A
`hqc128` OUR	2,725	0	2,060	15.5	164	52,757	0.32	872
`hqc192` OUR	2,784	0	2,379	17.5	164	150,265	0.91	2,533
`hqc256` OUR	3,231	0	2,972	17.0	164	302,845	1.84	5,945
`hqc128 perf HLS`*[AAB+20]	16,000	0	13,000	5.0	151	89,000	0.59	9,440
`hqc128 comp. HLS`*[AAB+20]	6,400	0	4,100	5.0	127	1,500,000	12.00	76,800
`hqc128 pure HLS`*[Ong2x]	67,494	0	26,223	11.0	169	112,351	0.74	50,080
`hqc128 opt. HLS`*[Ong2x]	67,534	0	26,219	10.0	168	112,326	0.74	50,042

† = Given resources does not include the area for `SHAKE256` module, * = Target FGPA is Artix-7 `xc7a100t-1`

3.6 Decapsulation Module

The decapsulate operation (shown in Algorithm 5) takes secret key $(\mathbf{x}, \mathbf{y})$, public key $(\mathbf{h}, \mathbf{s})$, ciphertext $(\mathbf{c} = (\mathbf{u}, \mathbf{v}))$, $\mathbf{d}$ as an input and generates shared secret (K). Figure 6b shows hardware design the `decap` module. We use our `decrypt` module (described in Section 3.3.2) to decrypt the input ciphertext using secret key $(\mathbf{y})$ and generate the $\mathbf{m}'$. We then use `encap` module to perform re-encryption of $\mathbf{m}'$ and generate $\mathbf{u}'$, $\mathbf{v}'$ and $\mathbf{d}'$. We then pause the `encap` module to verify the $\mathbf{u}'$, $\mathbf{v}'$ and $\mathbf{d}'$ against $\mathbf{u}$, $\mathbf{v}$ and $\mathbf{d}$. After the verification we set a signal (optional port `mprime_fail`) if the verification fails. Irrespective of verification result we still continue with the generation of the shared secret `K` to maintain the constant-time behavior.

Table 10 shows the results for the `decap` module. We note that the maximum clock frequency of `decap` module alone (without `SHAKE256`) module is in range 204 MHz for parameter set `hqc128` but since the critical path lies inside the `SHAKE256` module so we report that frequency in the Table 10. For parameter sets `hqc192` and `hqc256` the critical path lies inside `decode` module so we report the frequency accordingly in Table 10. The area results do not include the `SHAKE256` module for the same reason as described in Section 3.1.4.

Table 10: `decap` module area and timing information, data based on synthesis results for Artix 7 board with `xc7a200t-3` FPGA chip. Design with single clock, using slow clock due to hash module critical path.

Design	Resources† Logic (LUT)	(DSP)	Memory (FF)	(BR)	F (MHz)	Cycles (cyc.)	Time (ms)	T x A
`hqc128` OUR	8,426	0	6,642	36.0	164	78,223	0.48	4,044
`hqc192` OUR	10,021	0	8,045	37.5	148	215,511	1.46	14,630
`hqc256` OUR	12,470	0	9,962	40.0	156	434,987	2.78	34,666
`hqc128 perf HLS`*[AAB+20]	19,000	0	15,000	9.0	152	190,000	1.20	22,800
`hqc128 comp. HLS`*[AAB+20]	7,700	0	5,600	10.5	130	2,100,000	16.00	123,200
`hqc128 pure HLS`*[Ong2x]	58,638	0	21,787	18.0	116	224,430	1.92	112,526
`hqc128 opt. HLS`*[Ong2x]	58,645	0	21,800	18.0	153	222,515	1.47	86,149

† = Given resources does not include the area for `SHAKE256` module, * = Target FGPA is Artix-7 `xc7a100t-1`

3.7 Dual Clock Design

By profiling our `keygen`, `encap`, and `decap` modules we observe that the maximum number of clock cycles in all the operations are taken by our `poly_mult` module (described in Section 3.1.2) and we note that (as specified in Section 3.1.4) the maximum clock frequency of `keygen`, `encap`, `decap` modules is limited by the `SHAKE256` module since the critical path lies inside the round function of SHAKE256. To optimize the time taken by the time consuming modules such as `poly_mult` whose frequency is higher than that of `SHAKE256` module, we implement a `core_wrapper` module (shown in Figure 7) that has a capability to support two asynchronous clocks.

The `core_wrapper` block represented in Figure 7 can be any of the our modules that needs to interface with the `SHAKE256` module, i.e. key generation, encapsulation or decapsulation module. In the `core_wrapper` design we use two FIFOs (generated using the Xilinx IP generator) one in each direction (`Core_to_SHAKE256_FIFO`, `SHAKE256_to_Core_FIFO`). We compute the depth of the FIFO considering the worst case scenario and `CoreClock` to be of higher frequency than `SHAKE256Clock`. For `Core_to_SHAKE_FIFO` we note the depth be 36 (with width of each location to be 32). Out of 36 locations first two block are the `SHAKE256` module commands describing input and output width and rest of $34 \times 32 = 1088$-bits represents the block size of the SHAKE256. We select FIFO width to be 32 since the `SHAKE256` module has a 32-bit interface (as described in Section 3.1.1). For `SHAKE_to_CORE_FIFO` we compute the depth of the FIFO to be 1. For the `force_done` and `force_done_ack` signals we use a `Dual_Flop_Synchronizer`.

Table 11 shows the results for our dual clock hardware designs of `keygen`, `encap`, `decap` modules. We note that there is some overhead in terms of additional clock cycles when moving the data through FIFOs but the overhead is overcome through the higher clock frequency of the core. We also note that using `core_wrapper` was possible, but currently not used, for `decap` for `hqc192` and `hqc256` because the critical path moves inside the `decode` module. The area results do not include the `SHAKE256` module for the same reason as described in Section 3.1.4. Overall using `core_wrapper` gives a benefit of about 10% performance improvement for the modules. Time-area product also actually improves when `core_wrapper` is used, since the area overhead of adding the FIFOs and control logic is outweighted by the performance improvement.

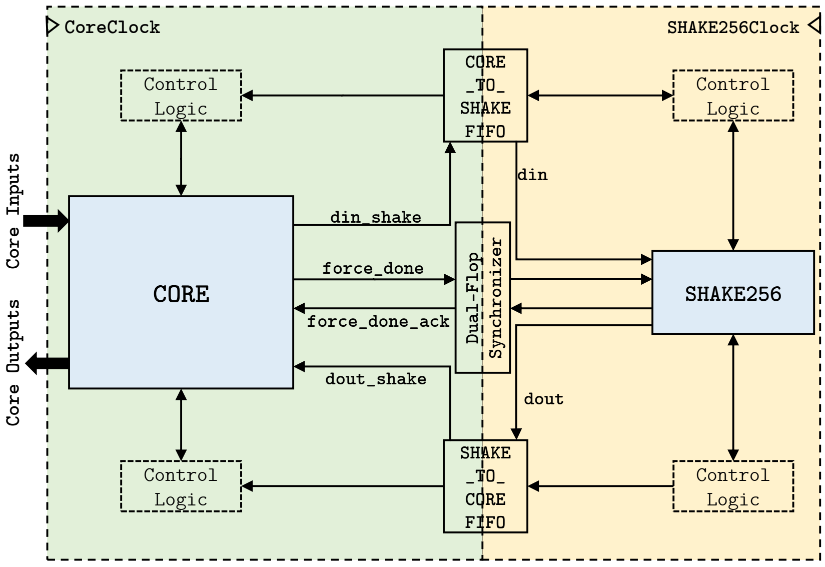


Figure 7: Hardware design of `core_wrapper` supporting dual clock.

4 Related Work

This section presents related work, focusing on full hardware designs of the four fourth-round public-key encryption and key-establishment algorithms in NIST's standardization process: BIKE, Classic McEliece, HQC, and SIKE. We also include the CRYSTALS-Kyber which is a public-key encryption and key-establishment algorithm selected for standardization at the end of the prior third-round. Due to limited space, related work on software implementations is omitted.

4.1 Related Hardware Designs

A hardware design for BIKE has been presented in [RBMG22]. The work investigated different strategies to efficiently implement the BIKE algorithm on FPGAs. The authors improved already existing polynomial multipliers, proposed efficient designs to realize polynomial inversions, and implement the Black-Gray-Flip (BGF) decoder. The authors provided VHDL designs for key generation, encapsulation, and decapsulation. For the fastest design, the authors showed 2.7ms for the key generation, 0.1ms for the encapsulation, and 1.9ms for the decapsulation, the times correspond to the high-speed implementation for the lowest security level. The authors also provide data for light-weight implementation for the lowest security level. Their paper further discusses Level 3 parameters for BIKE, but does not give final hardware data for the that security level. The authors provide free, non-commercial license for the hardware code.[2]

Classic McEliece has been most recently implemented in [CCD+22]. This is the first complete implementation of Classic McEliece KEM. The design provided Verilog code for encapsulation and decapsulation modules as well as key generation module with seed expansion. The authors presented three new algorithms that can be used for systemization of the public key matrix during key generation. The authors showed that the complete Classic McEliece design can perform key generation in 8.6ms, encapsulation in 0.3ms, and decapsulation in 0.9ms, the times correspond to the high-speed implementation for the lowest security level. The authors also provide hardware implementation for other security

[2]https://github.com/Chair-for-Security-Engineering/BIKE

Table 11: `keygen`, `encap`, and `decap` modules area and timing information, data based on synthesis results for Artix 7 board with `xc7a200t-3` FPGA chip. Designs with dual clocks, using slow clock for hash module, and fast clock for remainder of the design.

Design	Resources†				F_C	F_S	Cyc.$_C$	Cyc.$_S$	Time	T x A
	Logic		Memory							
	(LUT)	(DSP)	(FF)	(BR)	(MHz)	(MHz)	(cyc.)	(cyc.)	(ms)	
					keygen					
`hqc128`	3,094	0	879	14.5	242	164	27,013	738	0.12	359
`hqc192`	3,148	0	890	15.5	235	164	70,425	738	0.30	958
`hqc256`	3,001	0	911	15.5	248	164	140,993	738	0.57	1,720
					encap					
`hqc128`	2,609	0	2,070	15.5	218	164	45,739	14,528	0.30	779
`hqc192`	4,500	0	2,410	20	208	164	136,405	28,384	0.83	3,730
`hqc256`	4,961	0	3,045	20	218	164	280,213	44,706	1.56	7,729
					decap					
`hqc128`	8,434	0	6,652	36	204	164	71,199	14,528	0.43	3,691

F_C = Core Frequency, F_S = SHAKE256 Frequency, Cyc$_C$ = Core Cycles, Cyc$_S$ = SHAKE256 Cycles,
† = Given resources does not include the area for `SHAKE256` module

levels, and light-weight and high-speed versions for all the levels. The authors provide open-source code for the hardware.[3]

A hardware designs for HQC has been previously reported in [AAB+20]. The design was generated using high-level synthesis (HLS) as opposed to hand-written HLD code. The code can be generated to obtain performance numbers: 0.3ms for key generation, 0.6ms for encapsulation, and 1.2ms for decapsulation, the times correspond to the high-speed implementation of the lowest security level. Authors also provide light-weight version for the lowest security level, but did not provide hardware designs for other levels. Our implementation covers all three security levels. The authors provide code to generate VHDL implementation, for an Artix-7, from the HLS-compatible sources.[4]

Hardware implementation of SIKE has been provided in [MLRB20]. The authors created VHDL implementation of SIKE as a hardware co-processor. Their design can realize any of the SIKE security levels. For the high-speed design for the lowest security level, authors report the time for encapsulation, decapsulation, and keygen as 15.3ms, 16.3ms, and 9.1ms respectively . The authors make the code available under Creative Commons public domain license.[5]

Different hardware implementations of CRYSTALS-Kyber are available in [JGCS21, DMG21, XL21]. The authors presented design configurable for different performance and area requirements. The high-speed design provdied [DMG21] outperforms all other algorithms in terms of time for key generation, encapsulation, and decapsulation. For the lowest security level, the authors reported 0.02ms for key generation, 0.03ms for encapsulation, and 0.04ms for decapsulation. The authors did not provide access to the code for their hardware design.

4.2 Comparison to Related Work

The area and timing of the different hardware designs is listed in Table 12. The table compares our HQC design to other existing HQC HLS designs from the literature. We also provide Table 13 where we tabulate latest hardware implementations of all post-quantum

[3] `https://caslab.csl.yale.edu/code/pqc-classic-mceliece/`
[4] `https://pqc-hqc.org/implementation.html`
[5] `https://github.com/pmassolino/hw-sike`

Table 12: Comparison of the time and area for our HQC hardware design with the related work.

	Resources										
Design	**Logic**		**Memory**		**F**	**Encap**		**Decap**		**KeyGen**	
	(LUT)	(DSP)	(FF)	(BR)	(MHz)	(Mcyc.)	(ms)	(Mcyc.)	(ms)	(Mcyc.)	(ms)
Security Level 1 — Classical 128-bit Security											
HQC – Our Work, HDL design, Artix 7 (`xc7a200t`)											
SC	16,320	0	10,044	61.0	164	0.05	0.32	0.08	0.48	0.02	0.14
DC	16,956	0	9,837	66.0	204	0.06	0.30	0.08	0.43	0.03	0.12
HQC – [AAB+20], HLS design, Artix 7 (`xc7a100t`)											
LightWeight	8,900	0	6,400	14.0	132	1.50	12	2.10	16.0	0.63	4.8
HighSpeed	20,000	0	16,000	12.5	148	0.09	0.6	0.19	1.2	0.04	0.3
HQC – [Ong2x], HLS design, Artix 7 (`xc7a100t`)											
Pure	179,645	0	69,168	36.0	116	0.11	0.96	0.22	1.92	0.05	0.43
Optimized	179,713	0	69,174	34.0	153	0.11	0.73	0.22	1.47	0.05	0.33

SC = SingleClockDomain, *DC* = DualClockDomain, FF = flip-flop, F = $F_{\max}$, BR = BRAM

cryptographic algorithm hardware implementations from the fourth round of NIST's standardization process, plus the to-be standardized Kyber algorithm. Our data is from synthesis reports, while data for the other algorithms is from the cited papers. We focus on comparison of the hardware designs for lowest level of security, Level 1, as all publications give clear time and area numbers. Majority of related work provides hardware designs for more than the lowest security level, but the timing and area numbers are not clearly broken down in the respective publications, so we focus only on comparing among the lowest security level designs.

For most other designs there is a light-weight and high-speed version. For our design we present data for single clock domain (SC) and dual clock domain (DC) designs, both of these target high performance while keeping good time-area product. For our SC design, the resources are the sum of all the resources used by the key generation, encapsulation, decapsulation and shared hash module. The SC frequency is limited by the hash module frequency. For our DC design, we also sum the resources. The resource usage increases due to the asynchronous FIFOs used to bridge the two clock domains and associated control logic. The cycles increase due to the extra cycles waiting for FIFOs to be filled with data before being read. However, since most of the cycles are spent in the faster clock domain, the overall times are reduced.

Compared to fourth-round candidates, our DC design achieves faster key generation and decapsulation that all candidates except for the following high-speed version of Kyber designs [DMG21] and [XL21]. Our DC design beats prior HQC HLS design's encapsulation, and the other designs, except only BIKE's and Kyber's encapsulation is faster. Our design achieves better time-area product than all other candidates except for Kyber.

5 Conclusion

This work presented hardware design for constant-time implementation of the HQC code-based key encapsulation mechanism. This work presented first, hand-optimized design of HQC key generation, encapsulation, and decapsulation written in Verilog targeting implementation on FPGAs. The three modules further share a common SHAKE256 hash module to reduce area overhead. The architecture of the hardware modules included novel,

Table 13: Comparison of the time and area of hardware designs of other (NIST PQC competition) round 4 KEM candidates.

Design	**Resources**				**F**	**Encap**		**Decap**		**KeyGen**	
	Logic		**Memory**								
	(LUT)	(DSP)	(FF)	(BR)	(MHz)	(Mcyc.)	(ms)	(Mcyc.)	(ms)	(Mcyc.)	(ms)
Security Level 1 — Classical 128-bit Security											
BIKE – [RBMG22], HDL design, Artix 7 (`xc7a35t`)											
LightWeight	12,868	7	5,354	17.0	121	0.20	1.2	1.62	13.3	2.67	21.9
HighSpeed	52,967	13	7,035	49.0	96	0.01	0.1	0.19	1.9	0.26	2.7
BIKE – [RBCGG21], HDL design, Artix 7 (`xc7a200t`)											
LightWeight	12,319	7	3,896	9.0	121	0.05	0.4	0.84		0.46	3.8
TradeOff	19,607	9	5,008	17.0	100	0.03	0.3	0.42	4.2	0.18	1.9
HighSpeed	25,549	13	5,462	34.0	113	0.01	0.1	0.21	1.9	0.19	1.7
Classic McEliece – [CCD+22], HDL design, Artix 7 (`xc7a200t`)											
LightWeight	23,890	5	45,658	138.5	112	0.13	1.1	0.17	1.5	8.88	79.2
HighSpeed	40,018	4	61,881	177.5	113	0.03	0.3	0.10	0.9	0.97	8.6
SIKE – [MLRB20], HDL design, Artix 7 (`xc7a100t`)											
LightWeight	11,943	57	7,202	21	145	—	25.6	—	27.2	—	15.1
HighSpeed	22,673	162	11,661	37	109	—	15.3	—	16.3	—	9.1
Kyber – [JGCS21], HDL design, (`xc7a35t-2`)											
CB	5,269	2	2,422	6	—	0.67	2.67	0.73	2.93	0.69	2.75
RB	7,151	2	2,422	5	—	0.03	0.10	0.03	0.12	0.04	0.15
Kyber – [DMG21], HDL design, (`xc7a200t`)											
HighSpeed	9,457	4	8,543	4.5	220	0.003	0.01	0.004	0.02	0.002	0.01
Kyber – [XL21], HDL design, (`xa7a12`)											
HighSpeed	7,412	2	4,644	3	161	0.005	0.03	0.006	0.04	0.003	0.02

CB = CoProcessorBased, *RB* = RoundBased, FF = flip-flop, F = $F_{\max}$, BR = BRAM

dual clock domain design, allowing the common SHAKE module to run at slower clock speed compared to the rest of the design, while other faster modules run at their optimal clock rate. The design currently outperforms the other hardware designs for HQC, and many of the fourth-round Post-Quantum Cryptography standardization process. As this work showed, code-based designs can be competitive with other schemes when optimized hardware is developed.

References

[AAB+20] Carlos Aguilar Melchor, Nicolas Aragon, Slim Bettaieb, Loïc Bidoux, Olivier Blazy, Jean-Christophe Deneuville, Philippe Gaborit, Edoardo Persichetti, Gilles Zémor, and Jurjen Bos. HQC. Technical report, National Institute of Standards and Technology, 2020. available at https://pqc-hqc.org/doc/hqc-specification_2021-06-06.pdf.

[CCD+22] Po-Jen Chen, Tung Chou, Sanjay Deshpande, Norman Lahr, Ruben Niederhagen, Jakub Szefer, and Wen Wang. Complete and improved FPGA implementation of Classic Mceliece. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2022(3):71–113, 2022.

[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. In *Proceedings of the International Conference on Field Programmable Logic and Applications*, FPL, August 2021.

[DMG21] Viet Ba Dang, Kamyar Mohajerani, and Kris Gaj. High-speed hardware architectures and fpga benchmarking of crystals-kyber, ntru, and saber. Cryptology ePrint Archive, Paper 2021/1508, 2021. https://eprint.iacr.org/2021/1508.

[GHJ+22] Qian Guo, Clemens Hlauschek, Thomas Johansson, Norman Lahr, Alexander Nilsson, and Robin Leander Schröder. Don't reject this: Key-recovery timing attacks due to rejection-sampling in hqc and bike. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2022, Issue 3:223–263, 2022.

[HHK17] Dennis Hofheinz, Kathrin Hövelmanns, and Eike Kiltz. A modular analysis of the Fujisaki–Okamoto transformation. In Yael Kalai and Leonid Reyzin, editors, *Theory of Cryptography*, pages 341–371, Cham, 2017. Springer International Publishing.

[JGCS21] Arpan Jati, Naina Gupta, Anupam Chattopadhyay, and Somitra Kumar Sanadhya. A configurable CRYSTALS-Kyber hardware implementation with side-channel protection. *Cryptology ePrint Archive*, 2021.

[MLRB20] Pedro Maat C. Massolino, Patrick Longa, Joost Renes, and Lejla Batina. A compact and scalable hardware/software co-design of SIKE. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2020(2):245–271, Mar. 2020.

[Ong2x] Ongoing. Undecided. *Ongoing*, 202x.

[RBCGG21] Jan Richter-Brockmann, Ming-Shing Chen, Santosh Ghosh, and Tim Güneysu. Racing bike: Improved polynomial multiplication and inversion in hardware. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2022(1):557–588, Nov. 2021.

[RBMG22] Jan Richter-Brockmann, Johannes Mono, and Tim Guneysu. Folding bike: Scalable hardware implementation for reconfigurable devices. *IEEE Transactions on Computers*, 71(5):1204–1215, 2022.

[WTJ$^+$20] Wen Wang, Shanquan Tian, Bernhard Jungk, Nina Bindel, Patrick Longa, and Jakub Szefer. Parameterized hardware accelerators for lattice-based cryptography and their application to the hw/sw co-design of qTESLA. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2020(3):269–306, Jun. 2020.

[XL21] Yufei Xing and Shuguo Li. A compact hardware implementation of cca-secure key exchange mechanism crystals-kyber on fpga. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2021(2):328–356, Feb. 2021.