POLITECNICO DI MILANO

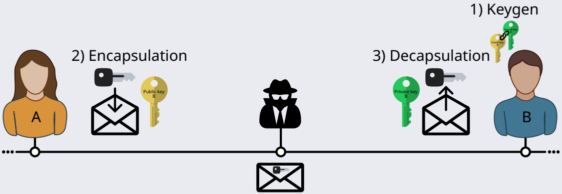
A High Efficiency Hardware Design for the Post-Quantum KEM HQC

Francesco Antognazza[1], Alessandro Barenghi[1], Gerardo Pelosi[1], Ruggero Susella[2]

[1] Department of Electonics, Information, and Bioengineering (DEIB), Politecnico di Milano, Milano, Italy
[2] STMicrolectronics S.r.l., Agrate Brianza, Italy

HOST 2024 · Washington D.C., USA · May 8, 2024

The quantum threat



- Today we assist to continuous advancements in the computational capabilities of quantum computers: $>$1000 of qubits in 2023
- Shor's algorithm speeds up part of the cryptanalysis of **all** currently deployed asymmetric algorithms
- In 2016 NIST started the Post-Quantum standardization process
  - CRYSTALS-Kyber (FIPS 203), CRYSTALS-Dilithium (FIPS 204), SPHINCS+ (FIPS 205), FALCON (WIP)
  - Portfolio variety: standardize also a code-based scheme among Classic McEliece, BIKE, **HQC**, and a new call for digital signatures

Our contribution

We focused on HQC due to its strong security properties, providing a full RTL hardware accelerator:

- having a flexible architecture for the binary polynomial arithmetics
- proposing new approach for the modulo operation during the sample of polynomials: no use of DSP while providing low latency
- using the state-of-the-art algebraic encoders and decoders and adapting them for the HQC public error correction code
- suggesting an optimization to the HQC algorithm improving the overall performance of the scheme

Background

Algebraic structure: binary polynomial ring

- **R**: polynomial ring $\mathbb{F}_2[x]/\langle x^p - 1\rangle$, where p is a prime number

$$a = a_0 + a_1 x + \ldots + a_{p-1} x^{p-1} \in \mathbf{R} \text{ stored as vector } \mathbf{a} = [a_0, a_1, \ldots, a_{p-1}]$$

Background

Algebraic structure: binary polynomial ring

- **R**: polynomial ring $\mathbb{F}_2[x]/\langle x^p - 1\rangle$, where p is a prime number
- $\omega(a)$: Hamming weight of $a \in \mathbf{R}$ (number of non-zero binary coefficients)

$a = a_0 + a_1x + \ldots + a_{p-1}x^{p-1} \in \mathbf{R}$ stored as vector $\mathbf{a} = [a_0, a_1, \ldots, a_{p-1}]$

Background

Algebraic structure: binary polynomial ring

- **R**: polynomial ring $\mathbb{F}_2[x]/\langle x^p - 1\rangle$, where p is a prime number
- $\omega(a)$: Hamming weight of $a \in \mathbf{R}$ (number of non-zero binary coefficients)
- $\mathbf{R}_w$: set containing all polynomials in **R** with Hamming weight w

$a{=}a_0 + a_1x + \ldots + a_{p-1}x^{p-1} \in \mathbf{R}$ stored as vector $\mathbf{a}{=}[a_0, a_1, \ldots, a_{p-1}]$

Moderate Density Parity Check code $\implies w \approx \sqrt{p}$
Since $a_i \in \mathbb{F}_2$, an element $a \in \mathbf{R}_w$ stored as vector of the non-zero i

Background

Algebraic structure: binary polynomial ring

- **R**: polynomial ring $\mathbb{F}_2[x]/\langle x^p - 1\rangle$, where p is a prime number
- $\omega(a)$: Hamming weight of $a \in \mathbf{R}$ (number of non-zero binary coefficients)
- $\mathbf{R}_w$: set containing all polynomials in **R** with Hamming weight w

polynomial addition ($+$)

Coefficient-wise addition: XOR ($\oplus$) boolean operator as coefficients in $\mathbb{F}_2$

$$\begin{array}{rccccccc} a = & a_0 & + & a_1 x & + \ldots + & a_{p-1}x^{p-1} \\ b = & b_0 & + & b_1 x & + \ldots + & b_{p-1}x^{p-1} \\ \hline a + b = & (a_0 \oplus b_0) & + & (a_1 \oplus b_1)x & + \ldots + & (a_{p-1} \oplus b_{p-1})x^{p-1} \end{array}$$

$$[a_0 \oplus b_0, a_1 \oplus b_1, \ldots, a_{p-1} \oplus b_{p-1}]$$

Background

Algebraic structure: binary polynomial ring

- **R**: polynomial ring $\mathbb{F}_2[x]/\langle x^p - 1\rangle$, where p is a prime number
- $\omega(a)$: Hamming weight of $a \in \mathbf{R}$ (number of non-zero binary coefficients)
- $\mathbf{R}_w$: set containing all polynomials in **R** with Hamming weight w

polynomial subtraction ($-$)

Coefficient-wise subtraction: XOR ($\oplus$) boolean operator as coefficients in $\mathbb{F}_2$

$$\begin{array}{rccccccc}
a = & a_0 & + & a_1 x & + \ldots + & a_{p-1}x^{p-1} \\
b = & b_0 & + & b_1 x & + \ldots + & b_{p-1}x^{p-1} \\
\hline
a - b = & (a_0 \oplus b_0) & + & (a_1 \oplus b_1)x & + \ldots + & (a_{p-1} \oplus b_{p-1})x^{p-1}
\end{array}$$

$$[a_0 \oplus b_0, a_1 \oplus b_1, \ldots, a_{p-1} \oplus b_{p-1}]$$

Background

Algebraic structure: binary polynomial ring

- **R**: polynomial ring $\mathbb{F}_2[x]/\langle x^p - 1\rangle$, where p is a prime number
- $\omega(a)$: Hamming weight of $a \in \mathbf{R}$ (number of non-zero binary coefficients)
- $\mathbf{R}_w$: set containing all polynomials in **R** with Hamming weight w

polynomial multiplication (·)

Cyclic convolution: $c_i = \bigoplus_{j+k \equiv i \bmod p} (a_j \otimes b_k),\ i, j, k \in \{0, 1, \ldots, p-1\}$

$$\begin{array}{rccccccc}
a = & a_0 & + & a_1 x & + \ldots + & a_{p-1} x^{p-1} \\
b = & b_0 & + & b_1 x & + \ldots + & b_{p-1} x^{p-1} \\
\hline
\text{acc.} = & (a_0 \oplus b_0) & + & (a_0 \oplus b_1)x & + \ldots + & (a_0 \oplus b_{p-1})x^{p-1}
\end{array}$$

Background

Algebraic structure: binary polynomial ring

- **R**: polynomial ring $\mathbb{F}_2[x]/\langle x^p - 1\rangle$, where p is a prime number
- $\omega(a)$: Hamming weight of $a \in \mathbf{R}$ (number of non-zero binary coefficients)
- $\mathbf{R}_w$: set containing all polynomials in **R** with Hamming weight w

polynomial multiplication (·)

Cyclic convolution: $c_i = \bigoplus_{j+k \equiv i \bmod p}(a_j \otimes b_k),\ i, j, k \in \{0, 1, \ldots, p-1\}$

$$
\begin{array}{rcccccll}
a = & a_0 & + & a_1 x & + \ldots + & a_{p-1}x^{p-1} & & \\
b = & b_0 & + & b_1 x & + \ldots + & b_{p-1}x^{p-1} & & \\
\hline
\text{acc.} = & (a_0 \oplus b_0) & + & (a_0 \oplus b_1)x & + \ldots + & (a_0 \oplus b_{p-1})x^{p-1} & & \\
 & & & (a_1 \oplus b_0)x & + \ldots + & (a_1 \oplus b_{p-2})x^{p-1} & + & (a_1 \oplus b_{p-1})x^p
\end{array}
$$

Background

Algebraic structure: binary polynomial ring

- **R**: polynomial ring $\mathbb{F}_2[x]/\langle x^p - 1\rangle$, where p is a prime number
- $\omega(a)$: Hamming weight of $a \in \mathbf{R}$ (number of non-zero binary coefficients)
- $\mathbf{R}_w$: set containing all polynomials in **R** with Hamming weight w

polynomial multiplication ($\cdot$)

Cyclic convolution: $c_i = \bigoplus_{j+k \equiv i \bmod p} (a_j \otimes b_k),\ \ i, j, k \in \{0, 1, \ldots, p-1\}$

$$
\begin{array}{rccccc}
a = & a_0 & + & a_1 x & + \ldots + & a_{p-1} x^{p-1} \\
b = & b_0 & + & b_1 x & + \ldots + & b_{p-1} x^{p-1} \\
\hline
\text{acc.} = & (a_0 \oplus b_0) & + & (a_0 \oplus b_1)x & + \ldots + & (a_0 \oplus b_{p-1})x^{p-1} \\
 & (a_1 \oplus b_{p-1}) & + & (a_1 \oplus b_0)x & + \ldots + & (a_1 \oplus b_{p-2})x^{p-1}
\end{array}
$$

Background

Algebraic structure: binary polynomial ring

- **R**: polynomial ring $\mathbb{F}_2[x]/\langle x^p - 1 \rangle$, where p is a prime number
- $\omega(a)$: Hamming weight of $a \in \mathbf{R}$ (number of non-zero binary coefficients)
- $\mathbf{R}_w$: set containing all polynomials in **R** with Hamming weight w

polynomial multiplication $(\cdot)$

Cyclic convolution: $c_i = \bigoplus_{j+k \equiv i \bmod p} (a_j \otimes b_k),\ \ i, j, k \in \{0, 1, \ldots, p-1\}$

$$
\begin{array}{rccccccc}
a = & a_0 & + & a_1 x & + \ldots + & {\color{red} a_{p-1}} x^{p-1} \\
b = & {\color{blue} b_0} & + & {\color{blue} b_1} x & + \ldots + & {\color{blue} b_{p-1}} x^{p-1} \\
\hline
\text{acc.} = & (a_0 \oplus b_0) & + & (a_0 \oplus b_1)x & + \ldots + & (a_0 \oplus b_{p-1})x^{p-1} \\
 & (a_1 \oplus b_{p-1}) & + & (a_1 \oplus b_0)x & + \ldots + & (a_1 \oplus b_{p-2})x^{p-1} \\
 & \vdots & & \vdots & \vdots & \vdots \\
 & ({\color{red} a_{p-1}} \oplus {\color{blue} b_1}) & + & ({\color{red} a_{p-1}} \oplus {\color{blue} b_2})x & + \ldots + & ({\color{red} a_{p-1}} \oplus {\color{blue} b_0})x^{p-1}
\end{array}
$$

Background

Algebraic structure: binary polynomial ring

- **R**: polynomial ring $\mathbb{F}_2[x]/\langle x^p - 1\rangle$, where p is a prime number
- $\omega(a)$: Hamming weight of $a \in \mathbf{R}$ (number of non-zero binary coefficients)
- $\mathbf{R}_w$: set containing all polynomials in **R** with Hamming weight w

polynomial multiplication (·)

Cyclic convolution: $c_i = \bigoplus_{j+k \equiv i \bmod p}(a_j \otimes b_k),\ i, j, k \in \{0, 1, \ldots, p-1\}$

$$
\begin{array}{rccccccc}
a = & a_0 & + & a_1 x & + \ldots + & a_{p-1} x^{p-1} \\
b = & b_0 & + & b_1 x & + \ldots + & b_{p-1} x^{p-1} \\
\hline
\text{acc.} = & (a_0 \oplus b_0) & + & (a_0 \oplus b_1)x & + \ldots + & (a_0 \oplus b_{p-1})x^{p-1} \\
 & (a_1 \oplus b_{p-1}) & + & (a_1 \oplus b_0)x & + \ldots + & (a_1 \oplus b_{p-2})x^{p-1} \\
 & \vdots & & \vdots & \vdots & \vdots \\
 & (a_{p-1} \oplus b_1) & + & (a_{p-1} \oplus b_2)x & + \ldots + & (a_{p-1} \oplus b_0)x^{p-1}
\end{array}
$$

if $a \in \mathbf{R}_w$ and $b \in \mathbf{R}$, has asymptotic complexity $\Theta(pw) = \Theta(p\sqrt{p}) = \Theta(p^{1.5})$

Background

Error correction code

- quasi-cyclic random $[2p, p, d]$ code with a public parity-check matrix $\mathbf{H} = [\mathbf{I}_p \mid \mathbf{rot}(h)]$

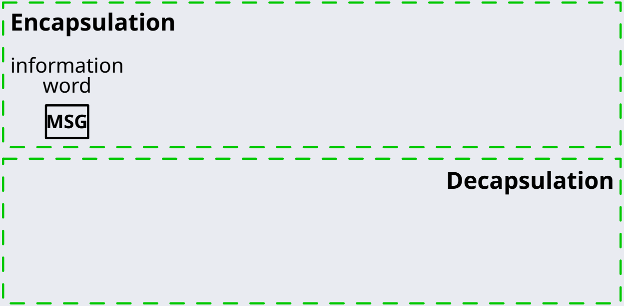
$$\mathbf{H} = \left[\begin{array}{ccccc|ccccc} 1 & 0 & 0 & \cdots & 0 & h_0 & h_{p-1} & h_{p-2} & \cdots & h_1 \\ 0 & 1 & 0 & \cdots & 0 & h_1 & h_0 & h_{p-1} & \cdots & h_2 \\ 0 & 0 & 1 & \cdots & 0 & h_2 & h_1 & h_0 & \cdots & h_3 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & 1 & h_{p-1} & h_{p-2} & h_{p-3} & \cdots & h_0 \end{array}\right]$$

h is a random vector generated from the public key seed

Background

Error correction code(s)

- quasi-cyclic random $[2p, p, d]$ code with a public parity-check matrix $\mathbf{H} = [\mathbf{I}_p \mid \mathbf{rot}(h)]$
- public $[n_e n_i, k_e k_i, d_e d_i]$ fixed code generated by a shortened Reed-Solomon (RS) $[n_e, k_e, d_e]$ (external) code with a duplicated Reed-Muller (RM) $[n_i, k_i, d_i]$ (internal) code such that $n_e n_i \approx p$.



Encapsulation

information word

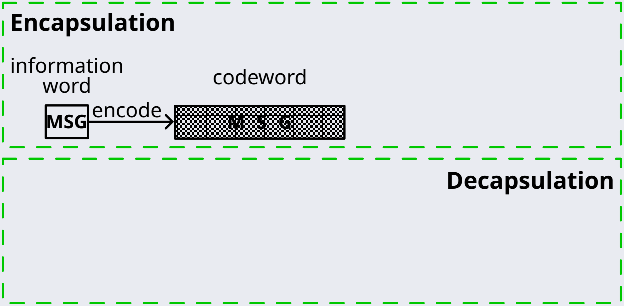
MSG

Decapsulation

Background

Error correction code(s)

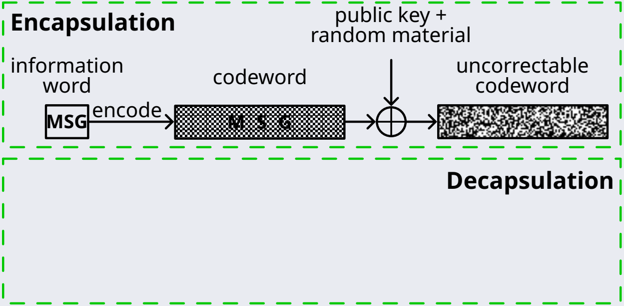
- quasi-cyclic random $[2p, p, d]$ code with a public parity-check matrix $\mathbf{H} = [\mathbf{I}_p \mid \mathbf{rot}(h)]$
- public $[n_e n_i, k_e k_i, d_e d_i]$ fixed code generated by a shortened Reed-Solomon (RS) $[n_e, k_e, d_e]$ (external) code with a duplicated Reed-Muller (RM) $[n_i, k_i, d_i]$ (internal) code such that $n_e n_i \approx p$.



Background

Error correction code(s)

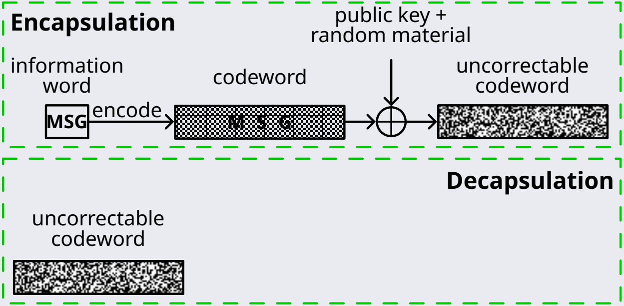
- quasi-cyclic random $[2p, p, d]$ code with a public parity-check matrix $\mathbf{H} = [\mathbf{I}_p \mid \mathbf{rot}(h)]$
- public $[n_e n_i, k_e k_i, d_e d_i]$ fixed code generated by a shortened Reed-Solomon (RS) $[n_e, k_e, d_e]$ (external) code with a duplicated Reed-Muller (RM) $[n_i, k_i, d_i]$ (internal) code such that $n_e n_i \approx p$.



Background

Error correction code(s)

- quasi-cyclic random $[2p, p, d]$ code with a public parity-check matrix $\mathbf{H} = [\mathbf{I}_p \mid \mathbf{rot}(h)]$
- public $[n_e n_i, k_e k_i, d_e d_i]$ fixed code generated by a shortened Reed-Solomon (RS) $[n_e, k_e, d_e]$ (external) code with a duplicated Reed-Muller (RM) $[n_i, k_i, d_i]$ (internal) code such that $n_e n_i \approx p$.



Encapsulation

public key +
random material

information
word

MSG

encode

codeword

M S G

uncorrectable
codeword

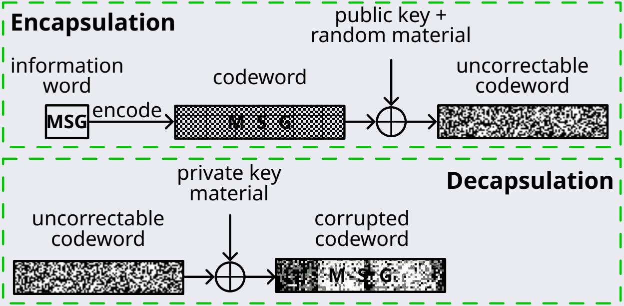
Decapsulation

uncorrectable
codeword

Background

Error correction code(s)

- quasi-cyclic random $[2p, p, d]$ code with a public parity-check matrix $\mathbf{H} = [\mathbf{I}_p \mid \mathbf{rot}(h)]$
- public $[n_e n_i, k_e k_i, d_e d_i]$ fixed code generated by a shortened Reed-Solomon (RS) $[n_e, k_e, d_e]$ (external) code with a duplicated Reed-Muller (RM) $[n_i, k_i, d_i]$ (internal) code such that $n_e n_i \approx p$.



Encapsulation

information word

MSG

encode

codeword

public key + random material

uncorrectable codeword

Decapsulation

uncorrectable codeword

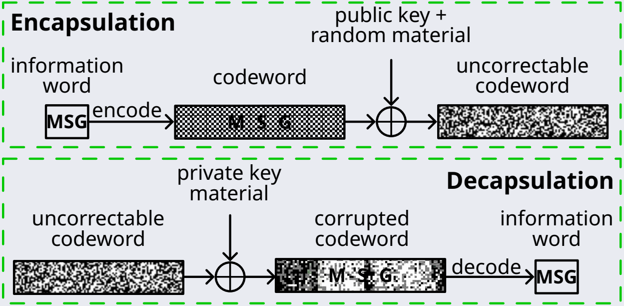
private key material

corrupted codeword

Background

Error correction code(s)

- quasi-cyclic random $[2p, p, d]$ code with a public parity-check matrix $\mathbf{H} = [\mathbf{I}_p \mid \mathbf{rot}(h)]$
- public $[n_e n_i, k_e k_i, d_e d_i]$ fixed code generated by a shortened Reed-Solomon (RS) $[n_e, k_e, d_e]$ (external) code with a duplicated Reed-Muller (RM) $[n_i, k_i, d_i]$ (internal) code such that $n_e n_i \approx p$.



Background

NIST provided a security level classification to match the security margin of AES symmetric key encryption algorithm:

Table: Security level classification by NIST

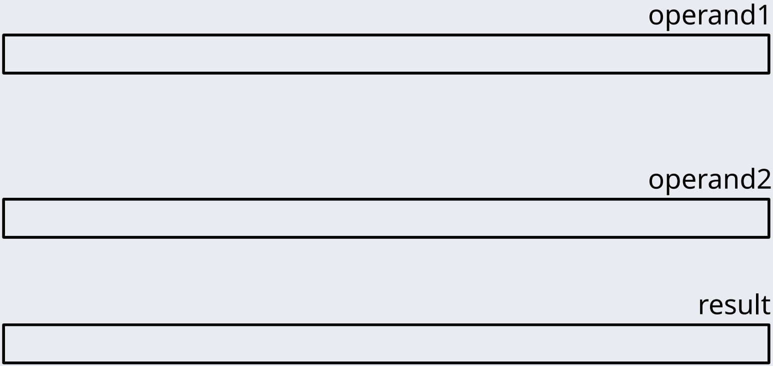
Security level	**AES parameter**	**HQC parameter**
1	AES-128	hqc-128
3	AES-192	hqc-192
5	AES-256	hqc-256

Each HQC parameter set specifies a different algebraic structure and public error correction code.

Polynomial adder

Addition/subtraction $\mathbf{R} \times \mathbf{R} \mapsto \mathbf{R}$

In case both operands are in **R**:

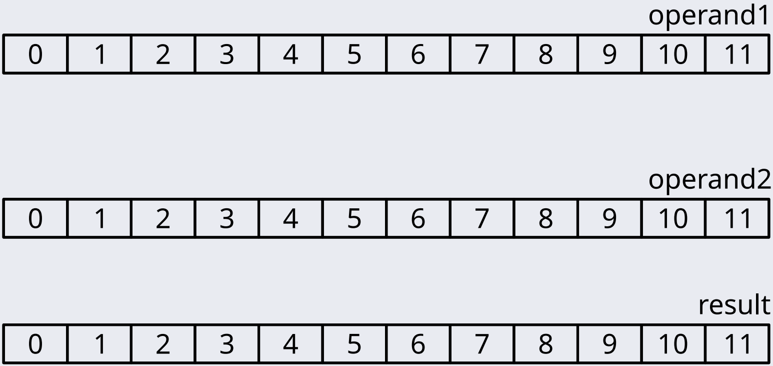


Polynomial adder

Addition/subtraction $\mathbf{R} \times \mathbf{R} \mapsto \mathbf{R}$

In case both operands are in **R**:

- access data in blocks of $B = 128$ bits



operand1

0	1	2	3	4	5	6	7	8	9	10	11

operand2

0	1	2	3	4	5	6	7	8	9	10	11

result

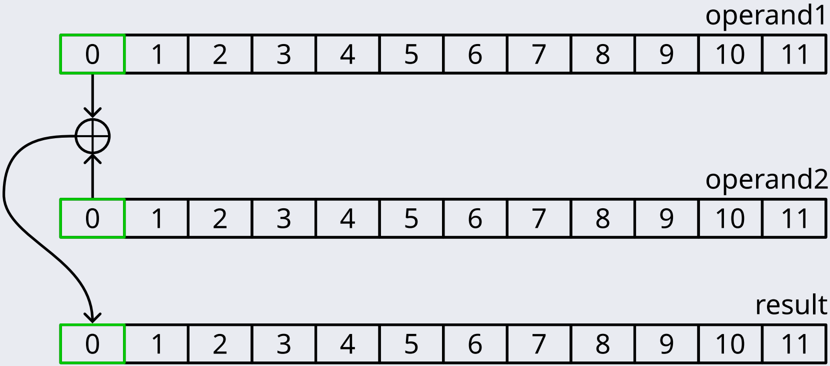
0	1	2	3	4	5	6	7	8	9	10	11

Polynomial adder

Addition/subtraction $\mathbf{R} \times \mathbf{R} \mapsto \mathbf{R}$

In case both operands are in **R**:

- access data in blocks of $B = 128$ bits
- perform the XOR operation block-wise

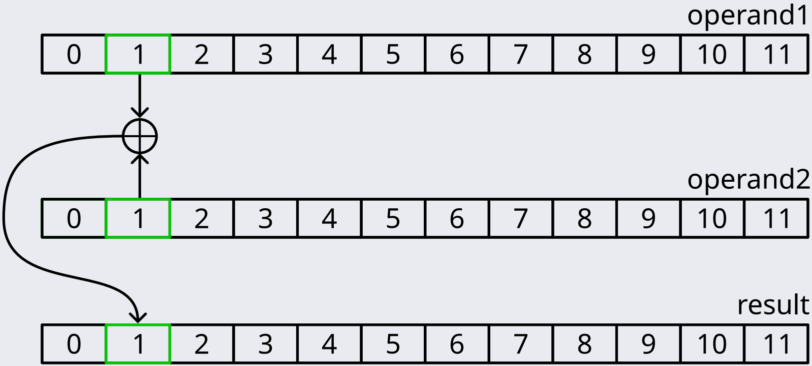


Polynomial adder

Addition/subtraction $\mathbf{R} \times \mathbf{R} \mapsto \mathbf{R}$

In case both operands are in **R**:

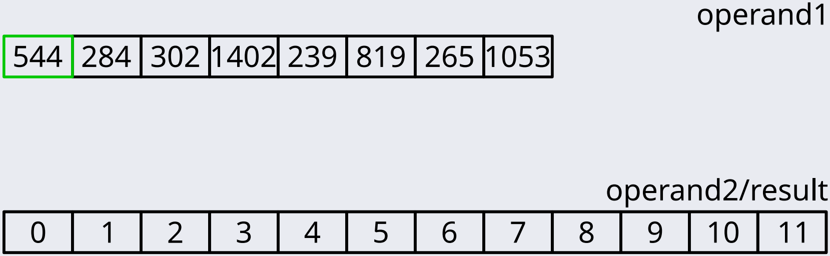
- access data in blocks of $B = 128$ bits
- perform the XOR operation block-wise



Polynomial adder

Addition $\mathbf{R}_w \times \mathbf{R} \mapsto \mathbf{R}$

In case operand1 in $\mathbf{R}_w$ and operand2 is in $\mathbf{R}$:
For each index *i* in the vector of operand1:



operand1

544	284	302	1402	239	819	265	1053

operand2/result

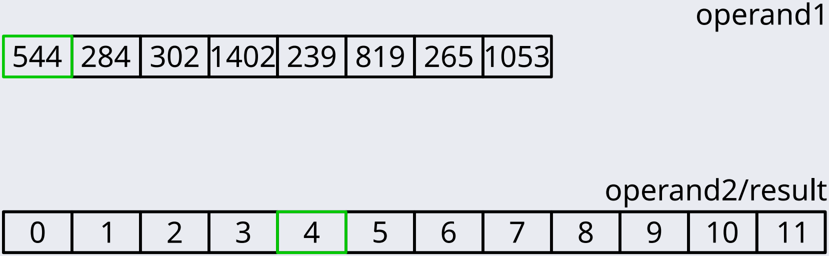
0	1	2	3	4	5	6	7	8	9	10	11

Polynomial adder

Addition $\mathbf{R}_w \times \mathbf{R} \mapsto \mathbf{R}$

In case operand1 in $\mathbf{R}_w$ and operand2 is in $\mathbf{R}$:
For each index *i* in the vector of operand1:

- determine the operand2 block index as $\lfloor i/B \rfloor$



operand1

544	284	302	1402	239	819	265	1053

operand2/result

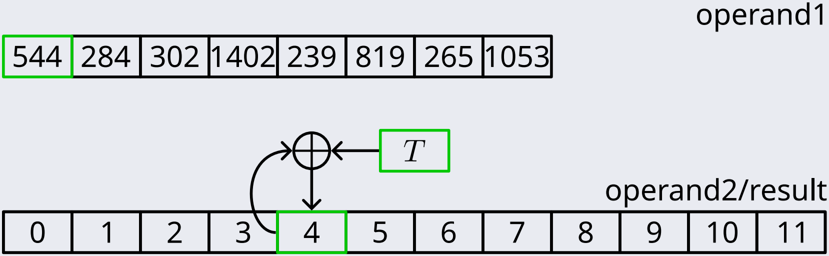
0	1	2	3	4	5	6	7	8	9	10	11

Polynomial adder

Addition $\mathbf{R}_w \times \mathbf{R} \mapsto \mathbf{R}$

In case operand1 in $\mathbf{R}_w$ and operand2 is in $\mathbf{R}$:
For each index i in the vector of operand1:

- determine the operand2 block index as $\lfloor i/B \rfloor$
- flip a single bit of that block by generating $T = 1 \ll (i \bmod B)$

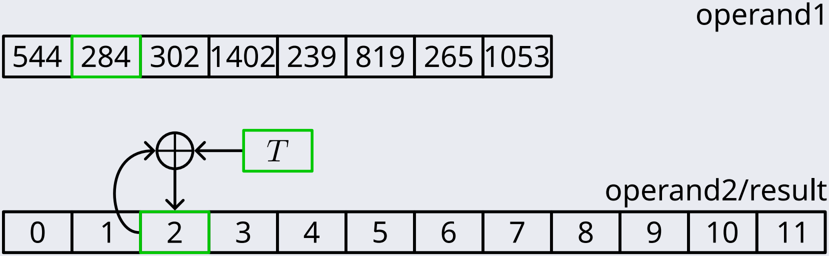


Polynomial adder

Addition $\mathbf{R}_w \times \mathbf{R} \mapsto \mathbf{R}$

In case operand1 in $\mathbf{R}_w$ and operand2 is in $\mathbf{R}$:
For each index i in the vector of operand1:

- determine the operand2 block index as $\lfloor i/B \rfloor$
- flip a single bit of that block by generating $T = 1 \ll (i \bmod B)$

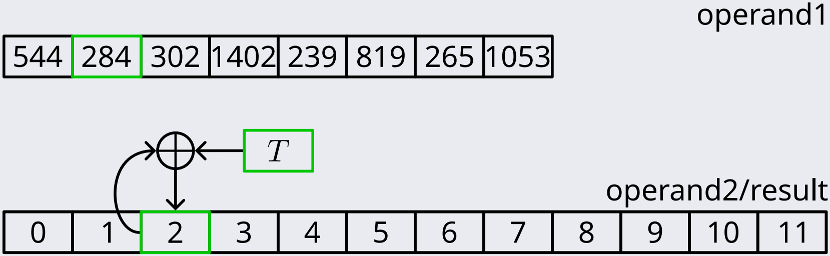


Polynomial adder

Addition $\mathbf{R}_w \times \mathbf{R} \mapsto \mathbf{R}$

In case operand1 in $\mathbf{R}_w$ and operand2 is in $\mathbf{R}$:
For each index i in the vector of operand1:

- determine the operand2 block index as $\lfloor i/B \rfloor$
- flip a single bit of that block by generating $T = 1 \ll (i \bmod B)$
- cannot be easily pipelined due to read-after-write dependency!



Polynomial multiplier: $\mathbf{R} \times \mathbf{R}_w$

- One operand is always in $\mathbf{R}_w$
- The low weight of polynomial ($\approx \sqrt{p}$) makes the schoolbook shift-and-add approach interesting: $\Theta(p^{1.5})$ asymptotic complexity

Polynomial multiplier: $\mathbf{R} \times \mathbf{R}_w$

- One operand is always in $\mathbf{R}_w$
- The low weight of polynomial ($\approx \sqrt{p}$) makes the schoolbook shift-and-add approach interesting: $\Theta(p^{1.5})$ asymptotic complexity

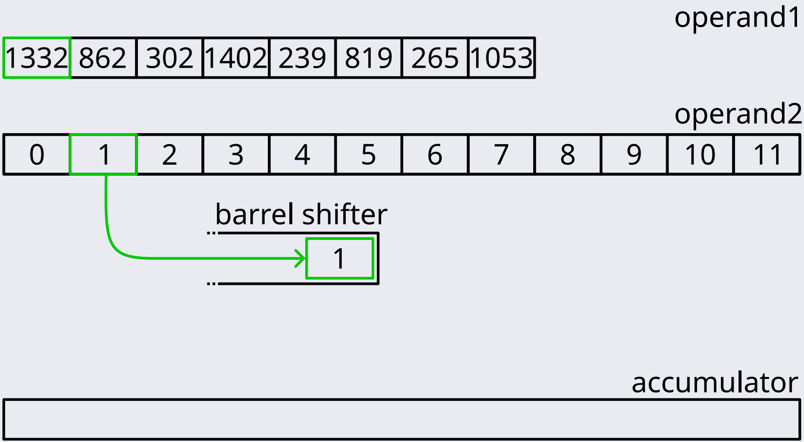
- There are faster algorithms based on the NTT with better asymptotic complexity, but:
    - the polynomial ring is not compatible with any NTT algorithm
    - memory access pattern is challenging to optimize

Polynomial multiplier: R $\times$ R$_w$

Single index processed

$$\text{start block} = \lfloor (p - i)/B \rfloor$$

$$\text{shift amount} = \lfloor (p - i) \bmod B \rfloor$$

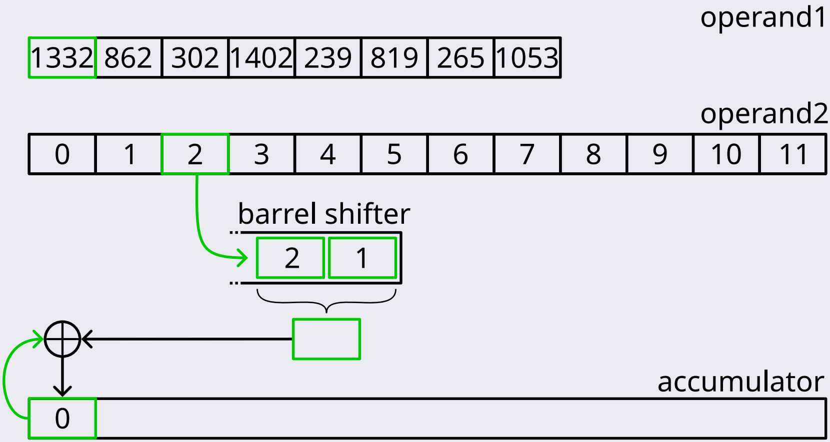


Polynomial multiplier: R × R_w

Single index processed

$$\text{start block} = \lfloor (p - i)/B \rfloor$$

$$\text{shift amount} = \lfloor (p - i) \bmod B \rfloor$$

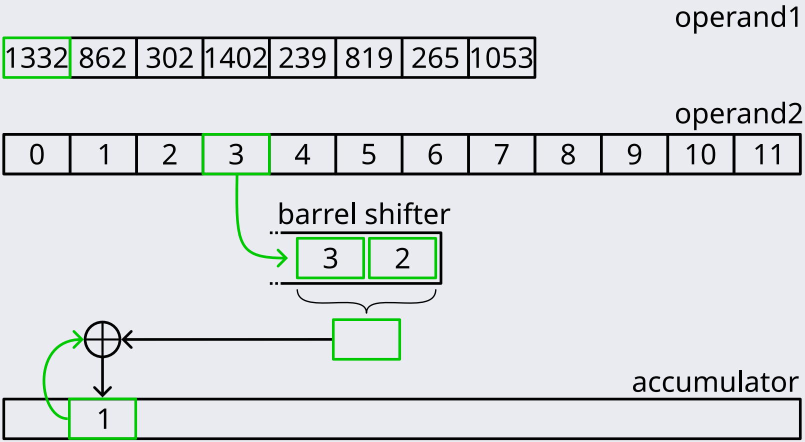


Polynomial multiplier: R × R_w

Single index processed

$$\text{start block} = \lfloor (p - i)/B \rfloor$$

$$\text{shift amount} = \lfloor (p - i) \bmod B \rfloor$$

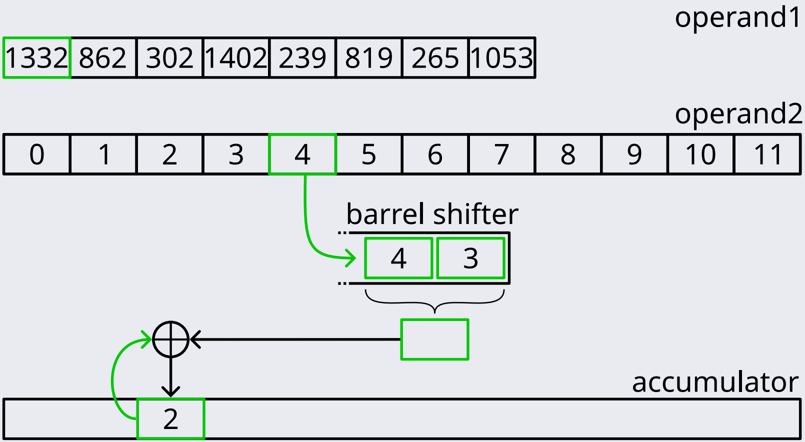


Polynomial multiplier: R × R_w

Single index processed

$$\text{start block} = \lfloor (p - i)/B \rfloor$$

$$\text{shift amount} = \lfloor (p - i) \bmod B \rfloor$$

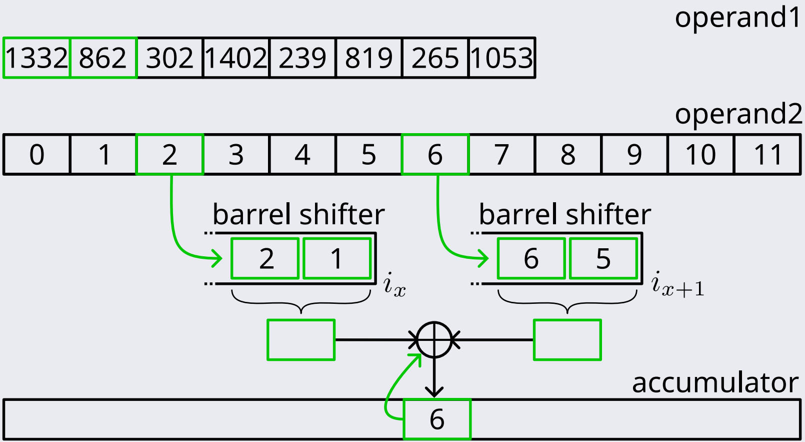


Polynomial multiplier: $\mathrm{R} \times \mathrm{R}_w$

Multiple indexes processed

$$\text{start block} = \lfloor (p - i)/B \rfloor$$

$$\text{shift amount} = \lfloor (p - i) \bmod B \rfloor$$

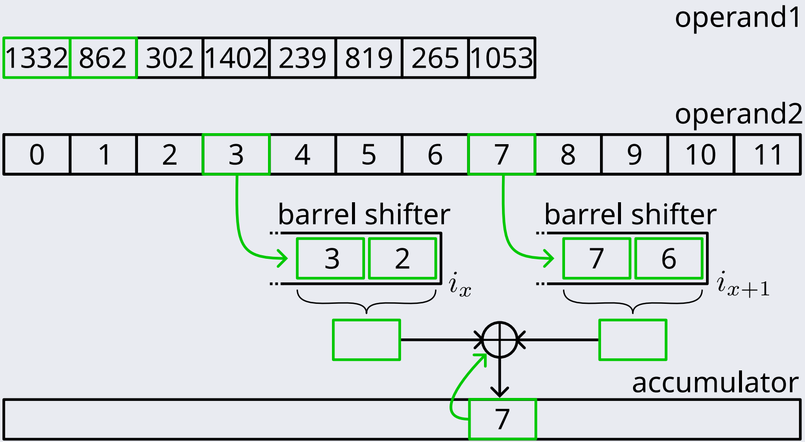


Polynomial multiplier: R × R_w

Multiple indexes processed

$$\text{start block} = \lfloor (p - i)/B \rfloor$$

$$\text{shift amount} = \lfloor (p - i) \bmod B \rfloor$$



Sampling polynomials uniformly

Generating polynomials in **R** and $\mathbf{R}_w$

The component of the vector $\mathbf{h} \in \mathbf{R}$ are generated by the SHAKE-256 algorithm (a SHA-3 eXtensible Output Function) expanding the small 320-bits public seed.

The HQC specification uses the constant-time algorithm from [1]:

- runs in constant-time
- uses of an exact amount of randomness ($32 \cdot w$ bits)
- requires a modulo operations between a 32-bit dividend and a generic 16-bit divisor

We used a straightforward *shift-and-subtract* pipelined algorithm, not requiring DSPs to perform the operation.

At synthesis time the number of pipeline stages can be selected to balance resources usage and timing closure.

Public code: Reed Solomon

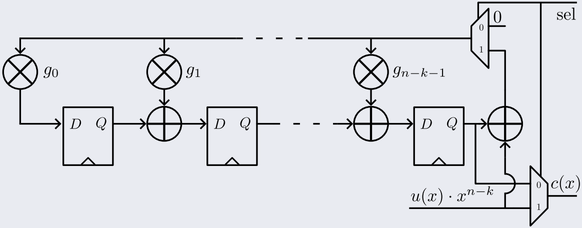
Encoder

The code treats a block of data as a set of $\mathbb{F}_{2^8}$ elements (symbols).

In a systematic encoding procedure the sequence of symbols of the message polynomial $u(x)$ are the prefix of the codeword, and the error correcting symbols are the suffix:

$$c(x) = x^{n_e-k_e}u(x) - \left(x^{n_e-k_e}u(x) \bmod g(x)\right)$$

A simple way to produce such special encoding is through a Linear Feedback Shift Register [2]:



Public code: Reed Solomon

Decoder

Consider a valid codeword $c(x)$ affected by an unknown error $e(x)$ which has up to t terms:

$$r(x) = c(x) + e(x)$$

Decoding algorith overview

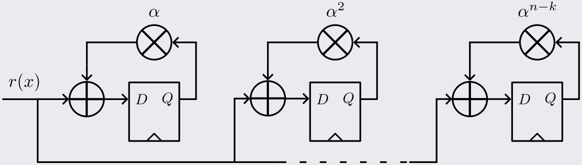
The decoder computes:

- the polynomial associated to the syndrome of the received word $r(x)$
- both positions and values of the coefficients of $e(x)$
- the error-free codeword is derived as $c(x) = r(x) - e(x)$.

Public code: Reed Solomon

Decoder

First, the received polynomial $r(x)$ is evaluated at each root α^i of the generator polynomial $g(x)$ using the Horner's method, determining the *syndrome polynomial* $S(x)$

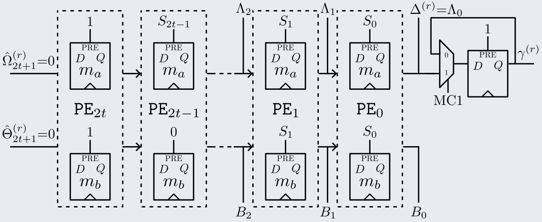


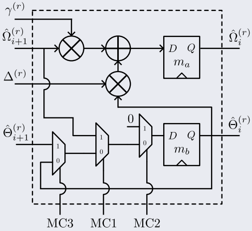
Public code: Reed Solomon

Decoder

We employed the design of the Enhanced Parallel Inversionless Berlekamp-Massey Algorithm (ePIBMA) introduced in [3].

The *error locator polynomial* $\Lambda(x)$ and the *auxiliary polynomial* $B(x)$ are derived from the syndrome polynomial $S(x)$

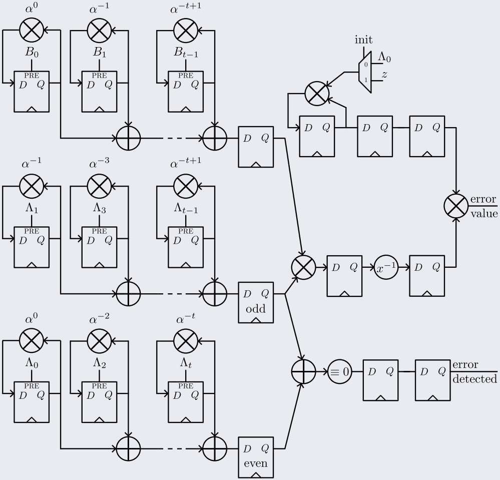




Public code: Reed Solomon

Decoder

Similarly, we used the Enhanced Chien Search and Error Evaluator design from [3].
The *error evaluator polynomial* $\Omega(x)$ is computed from $\Lambda(x)$ and $B(x)$.



Public code: Reed Muller

Encoder

To derive the 128-bit codewords corresponding to each 8-bit input message, we follow the traditional message vector multiplied by the generator matrix G.

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

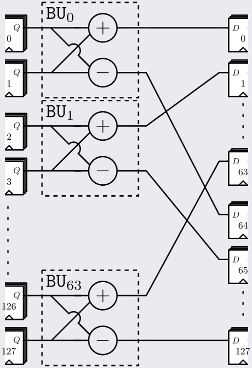
Working with 32-bits words, the presence of repeated words in G yields some identical intermediate values during the multiplication.

Consequently, the size of multiplexers and the number of XOR gates were decreased substantially.

Public code: Reed Muller

Decoder

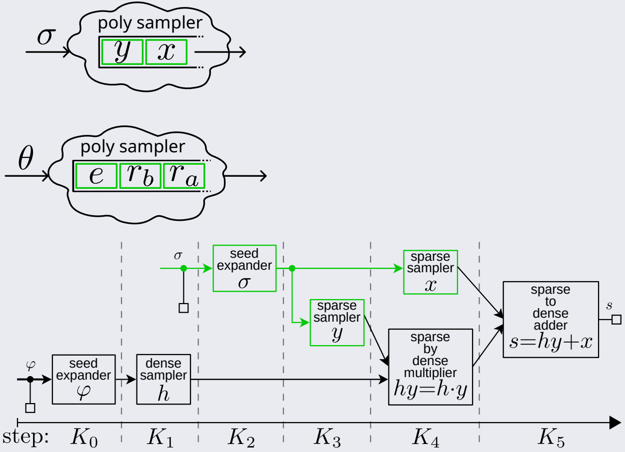
The operation is carried out by a Maximum Likelihood (ML) decoder computing a fast Hadamard transform [4]



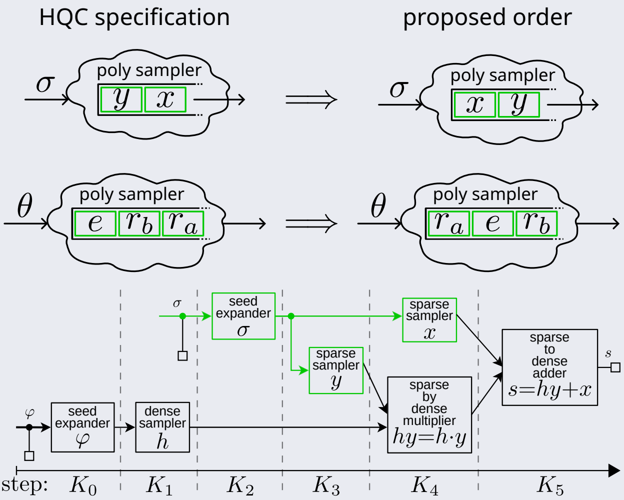
We find the maximum absolute value with a pipelined comparator tree computing pairwise maxima, acting on a tunable-sized input vector.

HQC schedule

HQC specification



HQC schedule



Performance gains from 13% to 32% over the entire cryptographic primitive **without any cost or security implications**

Experimental results

Designed in SystemVerilog, tested with CocoTB following the Universal Verification Methodology (UVM).

Synthesized on an Artix-7 `xc7a200tfbg484-3` FPGA, and validated it on a Digilent's Arty A7-100T employing the (modified) official Known Answer Tests (KAT) via a UART module.

The source code is available on Zenodo:

Experimental results

Public code encoder/decoder

Table: Performance of the public error correction code decoder. Area-Time product in $\mathrm{eSlices} \cdot \mathrm{ns}$

Parameter set	**Design**	**Resources eSlice**	**Frequency MHz**	**Latency** μs	**Area-Time product**
hqc128	our	1794	212	**6.10**	**10.94**
	[5]	1025	205	22.49	23.06
	[6]	2923	-	-	-
hqc192	our	2125	219	**7.69**	**16.35**
	[5]	1135	212	25.87	29.37
hqc256	our	2843	225	**12.02**	**34.18**
	[5]	1240	206	44.66	55.37

Experimental results

Fixed weight polynomial sampler

Table: Performance of fixed-weight polynomial samplers. Area-Time product [1] in $\mathrm{eSlices} \cdot \mu\mathrm{s}$

Parameter set	Design	Resources DSP	Resources eSlice	Frequency MHz	Latency μs	Area-Time product
hqc128	our	0	520	230	**10.15**	5.28
	[5] CWW	**4**	179	201	15.23	2.73⋆
	[5] FNB	0	335	223	6.63	2.22
	[7]	0	646	170	5.74	3.71
hqc192	our	0	509	237	**21.97**	11.18
	[5] CWW	**5**	181	200	34.08	6.17⋆
	[5] FNB	0	330	236	9.43	3.11
	[7]	0	773	185	8.84	6.84
hqc256	our	0	513	225	**39.26**	20.14
	[5] CWW	**5**	182	204	56.31	10.25⋆
	[5] FNB	0	399	242	13.42	5.36
	[7]	0	777	181	12.53	9.74

---

[1] ⋆ Contribution of DSP units not present in the AT product

Experimental results

Top-level: Key Generation

Table: Performance of HQC keygen top-module w/o SHAKE256 (5520 LUTs and 2810 FFs).
AT product in $\mathrm{eSlices} \cdot \mathrm{ns}$

Parameter set	**Design**	**Resources eSlice**	**Frequency MHz**	**Latency μs**	**Area-Time product**
hqc128	[5]	1879	179	88	165
	[6] (HLS, perf.)	2849	150	270	768
	our	4267	208	**30**	**127**
hqc192	[5]	1866	189	222	415
	our	4348	207	**72**	**314**
hqc256	[5]	1866	188	437	817
	our	4272	201	**138**	**591**

Experimental results

Top-level: Encapsulation

Table: Performance of HQC encapsulation top-modules w/o SHAKE256 (5520 LUTs and 2810 FFs). AT product in $\mathrm{eSlices} \cdot \mathrm{ns}$

Parameter set	Design	Resources eSlice	Frequency MHz	Latency μs	Area-Time product
hqc128	[5] (balanced)	2701	179	186	504
	[5] (high speed)	3377	179	125	423
	[6] (HLS, perf.)	4575	152	586	2682
	our	4326	168	**79**	**343**
hqc192	[5] (balanced)	2990	182	496	1484
	[5] (high speed)	3785	196	292	1106
	our	4468	175	**180**	**803**
hqc256	[5] (balanced)	3123	182	973	3039
	[5] (high speed)	3901	196	553	2160
	our	4412	187	**313**	**1382**

Experimental results

Top-level: Decapsulation

Table: Performance of HQC decapsulation top-modules w/o SHAKE256 (5520 LUTs and 2810 FFs). AT product in $\mathrm{eSlices} \cdot \mathrm{ns}$

Parameter set	Design	Resources eSlice	Frequency MHz	Latency μs	Area-Time product
hqc128	[5] (balanced)	4806	192	251	1206
	[5] (high speed)	5556	179	207	1154
	[6] (HLS, perf.)	6130	152	1270	7787
	our	5956	167	**119**	**709**
hqc192	[5] (balanced)	5309	186	676	3590
	[5] (high speed)	6051	186	498	3018
	our	7068	161	**287**	**2026**
hqc256	[5] (balanced)	5549	186	1335	7408
	[5] (high speed)	6289	186	966	6076
	our	8098	151	**570**	**4614**

Conclusions

Our work contributes to the current state-of-the-art:

- improving both latency and efficiency of HQC Key Encapsulation Mechanism RTL designs
- detailing an efficient implementation for the public error correction code in use by HQC
- providing an optimization for the HQC algorithm significantly improving the performance of the algorithm

Thanks for your attention

Francesco Antognazza

PhD student - Politecnico di Milano
email: `francesco.antognazza@polimi.it`
website: `https://antognazza.faculty.polimi.it/`

References I

[1] Nicolas Sendrier. “Secure Sampling of Constant-Weight Words - Application to BIKE”. In: *IACR Cryptol. ePrint Arch.* (2021), p. 1631. URL: `https://eprint.iacr.org/2021/1631`.

[2] Shu Lin and Daniel J. Costello Jr. *Error control coding - fundamentals and applications*. Prentice Hall computer applications in electrical engineering series. Prentice Hall, 1983. ISBN: 978-0-13-283796-5.

[3] Yingquan Wu. “New Scalable Decoder Architectures for Reed-Solomon Codes”. In: *IEEE Trans. Commun.* 63.8 (2015), pp. 2741–2761. DOI: `10.1109/TCOMM.2015.2445759`. URL: `https://doi.org/10.1109/TCOMM.2015.2445759`.

[4] Yair Be’ery and Jakov Snyders. “Optimal soft decision block decoders based on fast Hadamard transform”. In: *IEEE Trans. Inf. Theory* 32.3 (1986), pp. 355–364. DOI: `10.1109/TIT.1986.1057189`. URL: `https://doi.org/10.1109/TIT.1986.1057189`.

References II

[5] Sanjay Deshpande et al. “Fast and Efficient Hardware Implementation of HQC”. In: *IACR Cryptol. ePrint Arch.* (2023). URL: `https://eprint.iacr.org/2022/1183`.

[6] Carlos Aguilar Melchor et al. “Towards Automating Cryptographic Hardware Implementations: A Case Study of HQC”. In: *Code-Based Cryptography - 10th International Workshop, CBCrypto 2022, Trondheim, Norway, May 29-30, 2022, Revised Selected Papers*. Ed. by Jean-Christophe Deneuville. Vol. 13839. Lecture Notes in Computer Science. Springer, 2022, pp. 62–76. DOI: `10.1007/978-3-031-29689-5\_4`. URL: `https://doi.org/10.1007/978-3-031-29689-5\_4`.

[7] Pengzhou He, Yazheng Tu, and Jiafeng Xie. “LOCS: LOw-Latency and ConStant-Timing Implementation of Fixed-Weight Sampler for HQC”. In: *2023 IEEE International Symposium on Circuits and Systems (ISCAS)* (2023), pp. 1–5. URL: `https://api.semanticscholar.org/CorpusID:260004183`.