# Fast Scloud$^+$: A Fast Hardware Implementation for the Unstructured LWE-based KEM – Scloud$^+$

Jing Tian[1*], Yaodong Wei[1*], Dejun Xu[1], Kai Wang[1], Anyu Wang[2,5,6(✉)], Zhiyuan Qiu[3], Fu Yao[4] and Guang Zeng[4]

[1] School of Integrated Circuits, Nanjing University, Suzhou, China
tianjing@nju.edu.cn,{yaodongwei,xudejun,wang_kai}@smail.nju.edu.cn
[2] Institute for Advanced Study, BNRist, Tsinghua University, Beijing, China
anyuwang@tsinghua.edu.cn
[3] Shandong Institute of Blockchain, Jinan, China
qiuzhiyuan@sdibc.cn
[4] Shield Lab, Huawei Technologies, Beijing, China
{yaofu3,zengguang13}@huawei.com
[5] Zhongguancun Laboratory, Beijing, China
[6] National Financial Cryptography Research Center, Beijing, China

**Abstract.** Scloud$^+$ is an unstructured LWE-based key encapsulation mechanism (KEM) with conservative quantum security, in which ternary secrets and lattice coding are incorporated for higher computational and communication efficiency. However, its efficiencies are still much inferior to those of the structured LWE-based KEM, like ML-KEM (standardized by NIST). In this paper, we present a configurable hardware architecture for Scloud$^+$.KEM to improve the computational efficiency. Many algorithmic and architectural co-optimizations are proposed to reduce the complexity and increase the degree of parallelism. Specially, the matrix multiplications are computed by a block in serial and the block is calculated in one cycle, without using any multipliers. In addition, the random bits all are generated by an unfolded Keccak core, well matched with the data flow required by the block matrix multiplier. The proposed design is coded in Verilog and implemented under the SMIC 40nm LP CMOS technology. The synthesized results show that Scloud$^+$.KEM-128 only costs 23.0 *us*, 24.3 *us*, and 24.6 *us* in the *KeyGen*, *Encaps*, and *Decaps* stages, respectively, with an area consumption of 0.69 $mm^2$, significantly narrowing the gap with the state-of-the-art of Kyber hardware implementation.

**Keywords:** post-quantum cryptography · key encapsulation mechanism · learning with errors · lattice code · Hardware Implementation · ASIC

## 1 Introduction

Shor's quantum algorithm [Sho94] makes migrating from traditional public-key cryptography to post-quantum cryptography (PQC) inevitable. Due to the recent development in building quantum circuits, a large-scale quantum computer might become a reality within the next 10 years. These recent developments have, in turn, accelerated the establishment of PQC. In 2016, the national institute of standards and technology (NIST) [CJL+16] initiated the evaluation and standardization of PQC algorithms. Until now, the NIST evaluation is nearing the end and three schemes have been standardized in NIST FIPS standards [oST24].

✉ Corresponding author.
* Contributed equally to this work.

Among the PQC schemes, those based on the *learning with errors* (LWE) problem, called lattice-based PQC (LB-PQC), have gained particular popularity because of their better trade-off between security, performance, and size than other candidates. Two branches of LB-PQC have been developed. One is more efficient in performance and size while the security is based on the variants of the LWE problem with algebraic structure (abbreviated as *structured*-LWE), like the ring-LWE problem [LPR10, PRSD17] and the module-LWE problem [LS15]. Representative schemes include CRYSTALS-Kyber [Sch22], Saber [D'A20], and Aigis [ZYF$^+$20]. Note that CRYSTALS-Kyber belongs to one of the three FIPS standards, named ML-KEM.

The second branch is directly based on the hardness of the LWE problem without any additional algebraic structure (abbreviated as *unstructured*-LWE), which is regarded as more conservative in security than the variants. However, their performance and size both are about one order of magnitude inferior to the *structured*-LWE-based schemes, which largely decreases their superiority in practical applications. The most representative scheme is FrodoKEM [Nae20]. It had been evaluated for three rounds by NIST and was not selected for the aforementioned reason. However, with security being given the first priority, it has still been recommended by many other standards organizations, including the international organization for Standardization (ISO), Germany's Bundesamt für Sicherheit in der Informationstechnik (BSI), and Netherlands national communications security agency (NLNCSA). In order to improve its computational and communication efficiency, Wang **et al.** [WZZ$^+$24] recently presented a more efficient *unstructured*-LWE-based key encapsulation mechanism (KEM) named Scloud$^+$.KEM, which is a variant of FrodoKEM by incorporating ternary secrets and lattice coding to improve both of efficiency. As summarized in Table 1, Scloud$^+$.KEM has significantly improved the time and size over FrodoKEM but it is still much slower and larger than ML-KEM.

In this work, we try to deal with the gap of the computational efficiency between Scloud$^+$.KEM and ML-KEM. As shown in Table 1, our proposed hardware implementation consumes much fewer clock cycles than the software implementation of ML-KEM. When considering the clock frequency, our implementation is still faster for the NIST security levels 1 and 3. As many hardware implementations [XL21, KPM$^+$22, ZZO$^+$24, LLLL24] have been proposed for Kyber, we will provide the detailed results comparison under the same platform in the following comparison section to make it more fair.

## 1.1 Our Contributions

In this paper, we present, for the first time, a fast hardware architecture for Scloud$^+$.KEM by using the algorithmic and architectural co-optimization method to greatly improve the computational efficiency. The main contributions are summarized as follows:

- **High-Parallel and Low-Complexity Hardware Architecture**: We devise a novel configurable architecture for Scloud$^+$.KEM with a high degree of parallelism and low complexity. It mainly includes five modules: Matrix Multiplier, Random Bits Generator, Message Function, Memory, and Control Unit.
  - **Matrix Multiplier**: We present a square block matrix multiplier to iteratively compute all the matrix multiplications, which can perfectly match the left and right matrix multiplications. By utilizing the feature of the ternary values, the block multiplier is designed with only tree adders after a simple preprocessing and is fully parallelized and computed within one cycle.
  - **Random Bits Generator**: Considering the hardware efficiency, we only use the SHA-3 algorithm for generating all the random numbers and devise a completely unfolded Keccak core for this module, where 4 stages of pipeline are inserted, to well match the data flow required in the matrix multiplier. We

**Table 1:** Summary of the Performance of ML-KEM, FrodoKEM, and Scloud$^+$.KEM

| Scheme | Problem | *Platform | Operating Efficiency ($10^3$ cycles) | | | Size (Measured in Bytes) | | |
|---|---|---|---|---|---|---|---|---|
| | | | KenGen | Encaps | Decaps | Public Key | Ciphertext | Shared Secret |
| ML-KEM -512 | M-LWE | software | 134 | 158 | 204 | 800 | 768 | 32 |
| FrodoKEM -640 | LWE | software | 1375 | 1541 | 1474 | 9616 | 9720 | 16 |
| Scloud$^+$.KEM -128 | LWE | software | 1052 | 1115 | 1109 | 7200 | 5456 | 16 |
| | | **hardware** | **6.9** | **7.3** | **7.4** | | | |
| ML-KEM -768 | M-LWE | software | 232 | 258 | 320 | 1184 | 1088 | 32 |
| FrodoKEM -976 | LWE | software | 2786 | 2993 | 2814 | 15632 | 15744 | 24 |
| Scloud$^+$.KEM -192 | LWE | software | 2034 | 2226 | 2262 | 11136 | 10832 | 24 |
| | | **hardware** | **15.0** | **15.5** | **15.5** | | | |
| ML-KEM -1024 | M-LWE | software | 353 | 376 | 453 | 1568 | 1568 | 32 |
| FrodoKEM -1344 | LWE | software | 4906 | 5183 | 4992 | 21520 | 21632 | 32 |
| Scloud$^+$.KEM -256 | LWE | software | 3564 | 3738 | 3884 | 18744 | 16916 | 32 |
| | | **hardware** | **42.9** | **44.6** | **44.7** | | | |

* **Software**: downloading the open-source codes and running on Fedora 33 (Workstation Edition), equipped with an Intel Core-i9 10980XE @3.00GHz, with hyperthreading and TurboBoost disabled. **Hardware**: the proposed hardware architecture implementing under the SMIC 40nm LP CMOS technology, with a frequency of 300MHz.

also carefully regroup the input and output of the Keccak core according to different hash functions and their data access to the memory in each cycle, without any influence on the security. For computing the constant Hamming distribution, we present a new hardware-friendly algorithm without any sorting. For computing the central binomial distribution, we transform the original formula and only use two small adder trees to obtain the right output for all the required parameters.

– **Message Function**: For the message encoder, we design it iteratively for saving area. For the message decoder, we carefully analyze the recursive calculations and make a good trade-off between area and time. Moreover, we simplify the Euclidean distances calculation formula and discard the square root calculations without any accuracy loss.

– **Memory**: We design the memory into three categories with appropriate numbers of banks, widths, and depths, which can well satisfy the data flow to/from the memory. Moreover, most of the storage resources are fully utilized in different parameter settings.

– **Control Unit**: We resolve the computing steps in Scloud$^+$ carefully to make the data flow more smoothly so that the control logic is not so complicated. Note that the control logic of each module is well encapsulated to simplify the higher-level calls.

We code the proposed architecture in Verilog language and synthesize it under the SMIC 40nm CMOS technology. The synthesis results show that the proposed hardware implementation for Scloud$^+$.KEM consumes an area of 0.69 $mm^2$ and Scloud$^+$.KEM-128 costs 23.0 $us$, 24.3 $us$, and 24.6 $us$ in the *KeyGen*, *Encaps*, and *Decaps* stages, respectively. It is almost as efficient as many hardware implementations for Kyber.

### 1.2 Outline

The rest of the paper is organized as follows. Section 2 summarizes the Scloud$^+$.PKE and Scloud$^+$.KEM and presents the related notations and parameters. Section 3 details the proposed hardware architectures for Scloud$^+$.KEM. In Section 4, the ASIC implementation results are provided and compared with previous works. Finally, Section 5 concludes this paper.

## 2 Preliminaries

In this section, we first present the notations and parameters. Then, we review the Scloud$^+$.PKE and Scloud$^+$.KEM algorithms.

### 2.1 Notations and Parameters

The involved notations in Scloud$^+$.PKE and Scloud$^+$.KEM are presented below.

- Vectors are denoted by bold lower-case letters, such as $\mathbf{v}$, while matrices are represented by bold upper-case letters, such as $\mathbf{A}$.
- Sampling from a distribution $\chi$ is denoted by $x \leftarrow \chi$. The uniform discrete distribution over a finite set $\mathcal{S}$ is denoted by $U(\mathcal{S})$.
- Moduli: powers of 2 integers $q > q_1, q_2$;
- Matrix size parameters: positive integers $m, n, \bar{m}, \bar{n}$;
- Secret weight parameters: $h_1, h_2$;
- Error parameters: $\eta_1, \eta_2$;
- Message length: $l_m \in \{128, 192, 256\}$;
- Shared secret length: $l_{\mathbf{ss}} \in \{128, 192, 256\}$;
- Labeling and delabeling parameters: $\mu$ and $\tau$.

The summary of parameters for Scloud$^+$.PKE and Scloud$^+$.KEM is shown in Table 2, where all the three security levels are included.

**Table 2:** Parameters for Scloud$^+$.PKE and Scloud$^+$.KEM.

| Parameters | $l_{\mathbf{ss}} = l_m$ | $(q, q_1, q_2)$ | $(m, n)$ | $(\bar{m}, \bar{n})$ | $(h_1, h_2)$ | $(\eta_1, \eta_2)$ | $\mu$ | $\tau$ |
|---|---|---|---|---|---|---|---|---|
| Scloud$^+$-128 | 128 | $(2^{12}, 2^9, 2^7)$ | $(600, 600)$ | $(8, 8)$ | $(150, 150)$ | $(7, 7)$ | 64 | 3 |
| Scloud$^+$-192 | 192 | $(2^{12}, 2^{12}, 2^{10})$ | $(928, 896)$ | $(8, 8)$ | $(224, 232)$ | $(2, 1)$ | 96 | 4 |
| Scloud$^+$-256 | 256 | $(2^{12}, 2^{10}, 2^7)$ | $(1136, 1120)$ | $(12, 11)$ | $(280, 284)$ | $(3, 2)$ | 64 | 3 |

### 2.2 Scloud$^+$.PKE and Scloud$^+$.KEM

The Scloud$^+$.PKE algorithm is summarized in Alg. 1, where a tuple of algorithms *(KeyGen, Enc, Dec)* are included. The key generation algorithm Scloud$^+$.PKE.*KeyGen* takes the random seed $\boldsymbol{\alpha} \leftarrow U(\{0,1\}^{256})$ as input and outputs a pair of public key and secret key $(pk, sk)$. The encryption algorithm Scloud$^+$.PKE.*Enc* takes $pk$, a message $\mathbf{m}$, and a random coin $\mathbf{r}$ as input, and outputs a ciphertext $\mathbf{C}$. The decryption algorithm Scloud$^+$.PKE.*Dec* takes $sk$ and $\mathbf{C}$ as input, and outputs a message $m'$. The PKE scheme satisfies IND-CPA

**Algorithm 1:** Scloud$^+$.PKE Algorithm

1: $(pk, sk) = \mathbf{Scloud^+.PKE}.KeyGen(\boldsymbol{\alpha})$

$(\mathbf{seed_A}, \mathbf{r}_1, \mathbf{r}_2) = \mathtt{F}(\boldsymbol{\alpha}) \in \{0,1\}^{128} \times \{0,1\}^{256} \times \{0,1\}^{256}$
$\mathbf{A} = \mathsf{gen}(\mathbf{seed_A}) \in \mathbb{Z}_q^{m \times n}$
$\mathbf{S} = \Psi(\mathbf{r}_1, (n, \bar{n}), h_1) \in \mathbb{Z}^{n \times \bar{n}}$, $\mathbf{E} = \mathtt{CenBinom}(\mathbf{r}_2, (m, \bar{n}), \eta_1) \in \mathbb{Z}^{m \times \bar{n}}$
$\mathbf{B} = \mathbf{A} \cdot \mathbf{S} + \mathbf{E} \in \mathbb{Z}_q^{m \times \bar{n}}$
**return** $pk = (\mathbf{B}, \mathbf{seed_A})$, $sk = \mathbf{S}$

2: $(\mathbf{C}) = \mathbf{Scloud^+.PKE}.Enc(pk, \mathbf{m}, \mathbf{r})$

$\mathbf{A} = \mathsf{gen}(\mathbf{seed_A})$
$(\mathbf{r}'_1, \mathbf{r}'_2) = \mathtt{F}(\mathbf{r}) \in \{0,1\}^{256 \times 2}$
$\mathbf{S}' = \Phi(\mathbf{r}'_1, (\bar{m}, m), h_2) \in \mathbb{Z}^{\bar{m} \times m}$
$\mathbf{E}' = (\mathbf{E_1}, \mathbf{E_2}) = \mathtt{CenBinom}(\mathbf{r}'_2, (\bar{m}, n + \bar{n}), \eta_2)$, where $\mathbf{E}_1 \in \mathbb{Z}^{\bar{m} \times n}$, $\mathbf{E}_2 \in \mathbb{Z}^{\bar{m} \times \bar{n}}$
$\mathbf{M} = \mathtt{MsgEnc}(\mathbf{m}) \in \mathbb{Z}_q^{\bar{m} \times \bar{n}}$
$\mathbf{C}_1 = \mathbf{S}' \cdot \mathbf{A} + \mathbf{E}_1$, $\mathbf{C}_2 = \mathbf{S}' \cdot \mathbf{B} + \mathbf{E}_2 + \mathbf{M}$
$\bar{\mathbf{C}}_1 = \lfloor \frac{q_1}{q} \cdot \mathbf{C}_1 \rceil$, $\bar{\mathbf{C}}_2 = \lfloor \frac{q_2}{q} \cdot \mathbf{C}_2 \rceil_{\text{odd}}$
**return** $\mathbf{C} = (\bar{\mathbf{C}}_1, \bar{\mathbf{C}}_2)$

3: $(\mathbf{m}') = \mathbf{Scloud^+.PKE}.Dec((sk, \mathbf{C})$

$\mathbf{C}'_1 = \frac{q}{q_1} \cdot \bar{\mathbf{C}}_1$, $\mathbf{C}'_2 = \frac{q}{q_2} \cdot \bar{\mathbf{C}}_2$
$\mathbf{D} = \mathbf{C}'_2 - \mathbf{C}'_1 \mathbf{S} \in \mathbb{Z}_q^{\bar{m} \times \bar{n}}$
**return** $\mathbf{m}' = \mathtt{MsgDec}(\mathbf{D}) \in \{0,1\}^{l_m}$

security *(indistinguishability under chosen plaintext attack)*. In our design, considering the hardware efficiency, the random generation functions `gen`, $\Psi/\Phi$, and `CenBinom` all only take the seeds output from the Keccak core instead of an AES core. The message conversion functions `MsgEnc` and `MsgDec` are constructed using the labeling, BDD, and delabeling algorithms. More details for these functions can be referred to as in [WZZ+24] and the following section.

The Scloud$^+$.KEM algorithm is summarized in Alg. 2, where a tuple of algorithms *(KeyGen, Enc, Dec)* are included. The key generation algorithm Scloud$^+$.KEM.*KeyGen* takes the random seed $\boldsymbol{\alpha}$ and a random vector$\mathbf{z} \leftarrow U(\{0,1\}^{256})$ as input, and outputs a pair of public key and secret key $(pk, sk)$. The encapsulation algorithm Scloud$^+$.KEM.*Encaps* takes $pk$ and a message $\mathbf{m}$ as input, and outputs a ciphertext $\mathbf{C}$ and a shared secret $\mathbf{ss}$. The decapsulation algorithm Scloud$^+$.KEM.*Decaps* takes $sk$ and $\mathbf{C}$ as input, and outputs a shared secret $\mathbf{ss}$. The KEM satisfies IND-CCA security *(indistinguishability under chosen ciphertext attack, or IND-CCA2)*. The hash functions `F`, `H`, `G`, and `K` all belong to SHA-3. More details can be referred to in Section 3.3.

# 3 Proposed Hardware Architecture

In this section, we present the proposed hardware architecture for Scloud$^+$.KEM, including the top-level architecture and its sub-modules.

## 3.1 Top-Level Architecture

The top-level architecture for Scloud$^+$.KEM is shown in Fig. 1, mainly consisting of five modules: 1) Matrix Multiplier (MatM), 2)Random Bits Generator (RBitsG), 3) Message Function (MsgFunc), 4) Memory, and 5) Control Unit (CtrU).

**Algorithm 2:** Scloud$^+$.KEM Algorithm

1: $(pk, sk) = \textbf{Scloud}^+\textbf{.KEM}.KeyGen(\boldsymbol{\alpha}, \mathbf{z})$

$(pk, sk') = \text{Scloud}^+.\text{PKE}.KeyGen(\boldsymbol{\alpha})$
$\mathbf{hpk} = \mathtt{H}(pk) \in \{0,1\}^{256}$
$sk = (sk', pk, \mathbf{hpk}, \mathbf{z})$
**return** $(pk, sk)$

2: $(\mathbf{C}, \mathbf{ss}) = \textbf{Scloud}^+\textbf{.KEM}.Encaps(pk, \mathbf{m})$

$(\mathbf{r}, \mathbf{k}) = \mathtt{G}(\mathbf{m}||\mathtt{H}(pk)) \in \{0,1\}^{256\times 2}$
$\mathbf{C} = \text{Scloud}^+.\text{PKE.Enc}(pk, \mathbf{m}, \mathbf{r})$
$\mathbf{ss} = \mathtt{K}(\mathbf{k}||\mathbf{C}) \in \{0,1\}^{l_{\mathbf{ss}}}$
**return** $(\mathbf{C}, \mathbf{ss})$

3: $(\mathbf{ss}) = \textbf{Scloud}^+\textbf{.KEM}.Decaps((sk, \mathbf{C})$

$\mathbf{m}' = \text{Scloud}^+.\text{PKE.Dec}(sk', \mathbf{C})$
$(\mathbf{r}', \mathbf{k}') = \mathtt{G}(\mathbf{m}'||\mathbf{hpk})$
$\mathbf{C}' = \text{Scloud}^+.\text{PKE.Enc}(pk, \mathbf{m}', \mathbf{r}')$
**if** $\mathbf{C} = \mathbf{C}'$
  **return** $\mathbf{ss} = \mathtt{K}(\mathbf{k}', \mathbf{C})$
**else**
  **return** $\mathbf{ss} = \mathtt{K}(\mathbf{z}, \mathbf{C})$
**end if**

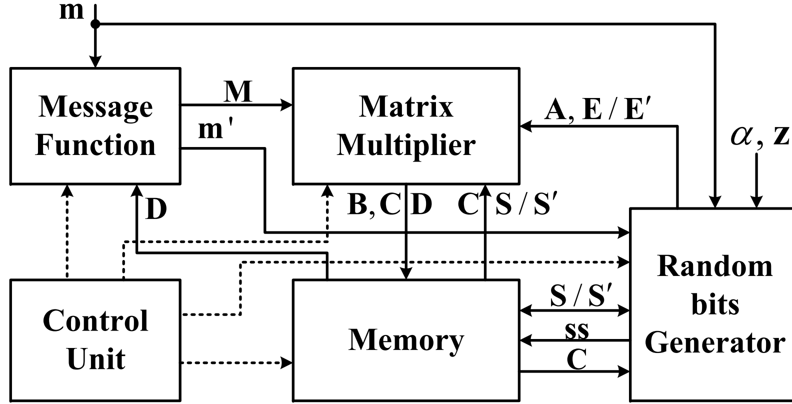


**Figure 1:** Proposed top-level architecture for Scloud$^+$.KEM, where five modules are included and the main data transformed between them are also marked.

MatM is used to compute the matrix multiplications $\mathbf{A} \cdot \mathbf{S}$, $\mathbf{S}' \cdot \mathbf{A}$, $\mathbf{S}' \cdot \mathbf{B}$, and $\mathbf{C}_1' \cdot \mathbf{S}$, where the size of matrix $\mathbf{A}$ is much bigger than those of $\mathbf{S}$, $\mathbf{S}'$, $\mathbf{B}$, and $\mathbf{C}_1'$, and no multipliers are adopted. RBitsG is to calculate the random bits including random matrices $\mathbf{A}$, $\mathbf{S}$, $\mathbf{E}$, and other random vectors. The basic core is the Keccak core with fully unfolded rounds, where random bits can be output each cycle after the initialization. MsgFunc is for calculating the MsgEnc and MsgDec functions detailed in [Scloud$^+$ ]. The Memory module is mainly to save the public and secret keys $pk = (\mathbf{B}, \mathbf{seed_A})$ and $sk = \mathbf{S}$, ciphertexts $\mathbf{C} = (\mathbf{C}_1, \mathbf{C}_2)$, and decompressed message $\mathbf{D}$. Note that $\mathbf{A}$ is dynamically generated by $\mathbf{seed_A}$, the memory for $\mathbf{B}$ and $\mathbf{C}_1$ is shared, and the memory for $\mathbf{C}_2$ and $\mathbf{D}$ is also shared. Finally, CtrU is used to generate control signals to combine those modules. More details about them are shown below.

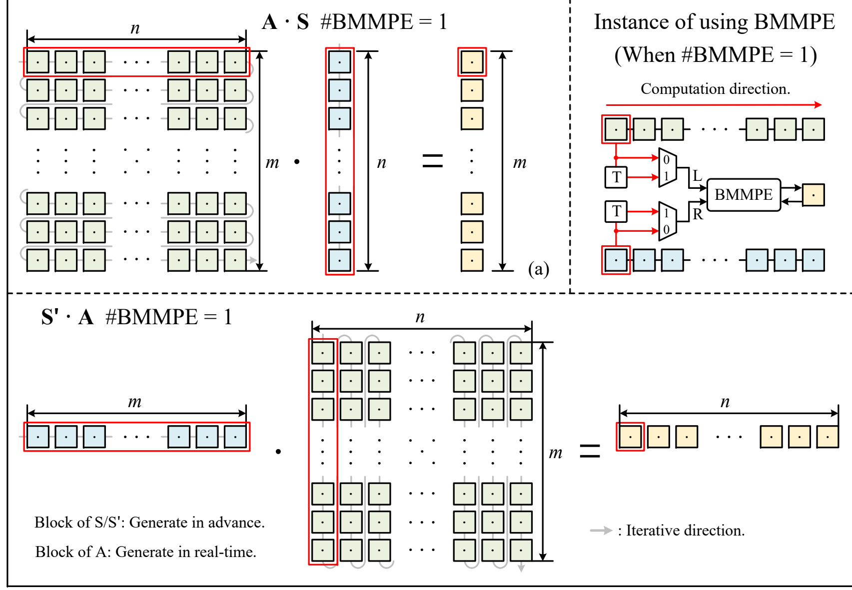


**Figure 2:** Diagrams for matrix multiplications in block, where "$\boxed{\cdot}$" denotes a block matrix multiplier. (a) Diagram of $\mathbf{A}^{m\times n}$ multiplied by $\mathbf{S}^{n\times\bar{n}}$ in block by block. (b) Control logic for transposing block matrices to select left and right matrix multiplications. (c) Diagram of $\mathbf{S}^{\bar{m}\times m}$ multiplied by $\mathbf{A}^{m\times n}$ in block by block.

## 3.2 Matrix Multiplier

The MatM module is designed in block to match the left and right multiplications of matrix $\mathbf{A} \in \mathbb{Z}_q^{m\times n}$ with matrix $\mathbf{S} \in \mathbb{Z}^{n\times\bar{n}}$ and $\mathbf{S}' \in \mathbb{Z}^{\bar{m}\times m}$. Since $(\mathbf{S}'\cdot\mathbf{A})^T = \mathbf{A}^T\cdot\mathbf{S}'^T$, we can directly transpose the two involved block multiplication matrices in this case to match the same control for the data flow and memory addresses as the other case. Fig. 2 shows the diagrams of the two kinds of matrix multiplications in blocks and the corresponding control logic. Assume the size of a block is $b\times b$ and $b \geq \bar{n}$ or $b \geq \bar{m}$. If one block is used, then we can use $\frac{m}{b}\times\frac{n}{b}$ iterations to finish one matrix multiplication. If $\bar{n}/2 \leq b < \bar{n}$ or $\bar{m}/2 \leq b < \bar{m}$, the number of iterations would be doubled. When $b = 8$, the parameters of Scloud$^+$-128 and Scloud$^+$-192 belong to the former and those of Scloud$^+$-256 belong to the latter. When adopting more blocks, the latency would be reduced linearly while the area would be increased sub-linearly as the Memory and MsgFunc modules would basically be unchanged. It should be noted that only one block is considered in the current version.

Fig. 3 shows the diagrams for block matrix multiplications based on processing elements (BMMPE). As shown in Fig. 3(a), the multiplication of two square matrices with a size of $b\times b$ requires $b^2$ PEs. For a PE, a $b$-value vector multiplication is computed. Since the ternary values (*i.e.* -1, 0, and 1) are used in $\mathbf{S}$ and $\mathbf{S}'$, we can use only adders instead of multipliers after a simple preprocessing module, as illustrated in Fig. 3(c).

Assume the values of $\mathbf{A}^{b\times b}$ in a block are $a^{i,j}$ with $q$ bits and $0 \leq i,j < b$, and values of $\mathbf{S}$ or $\mathbf{S}'$ are $s^{j,k}$ or $s'^{j,k}$ with 2 bits and $0 \leq j,k < b$. The preprocessing module can be designed by using a masking, an inverter, and a multiplexer for a bit as shown in Fig. 4(a). It means that we decide $\hat{a}^{i,j,k}$ equal to $a^{i,j}$, 0, or 2's complement of $a^{i,j}$ according to $s^{j,k}$ in advance and then complete the accumulation using an adder tree. Interestingly, it can be found that the computation of 2's complement of $a^{i,j}$ equals $2^q - a^{i,j} = \neg a^{i,j} + 1 \equiv -a^{i,j} \pmod{2^q}$. Namely, we have finished the reduction operation for the negative inputs in the meantime. In addition, computing those 2's complements is not immediately finished

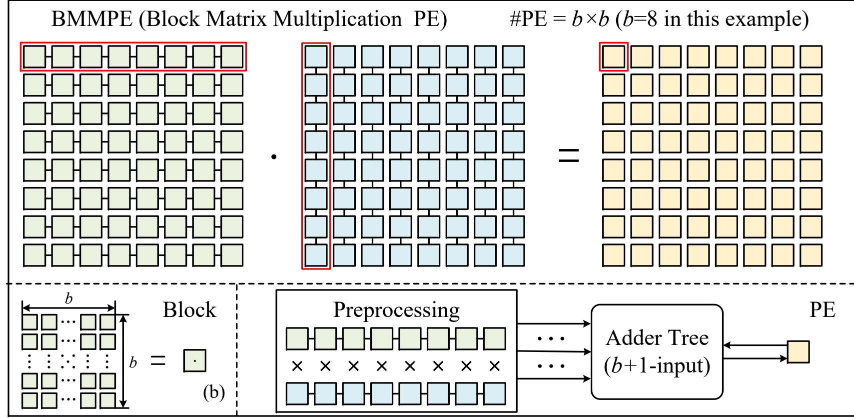


**Figure 3:** Diagrams for block matrix multiplications. (a) Diagram of block matrix multiplication in processing element (PE), where the number of PEs is $b \times b$ and $b = 8$ in this example. (b) Diagram of a block with a size of $b \times b$. (c) Diagram of a PE, consisting of a preprocessing module and an adder tree with $b + 1$ inputs.

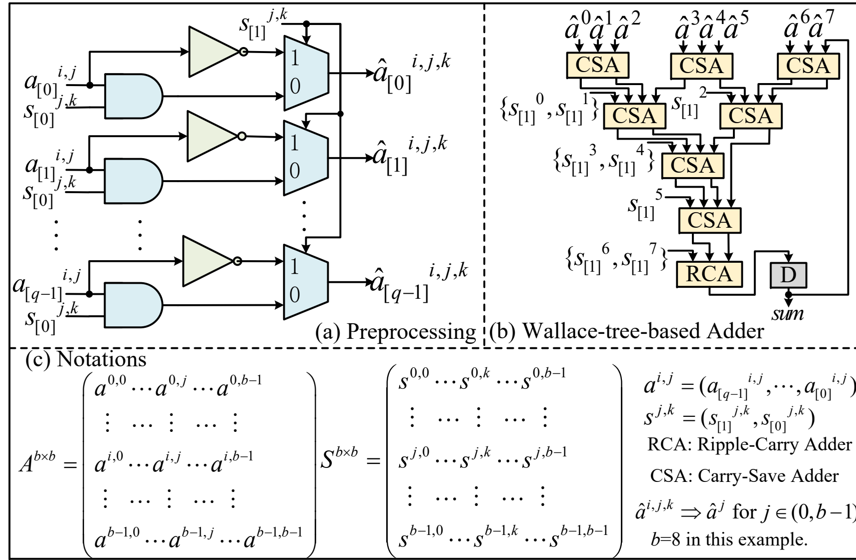


**Figure 4:** Circuits of a PE. (a) Preprocessing. (b) Wallace-tree-based adder for an example of $b = 8$. (c) Notations.

but computed in two steps to reduce the critical path and resource consumption. The first step is to take the negation using inverters as shown in Fig. 4(a). The second step is plus $b$ ones, which are involved in the following adder tree as shown in Fig. 4(b). We refer to the layered Wallace reduction [Wal64] that are made up of $b - 1$ carry-save adders (CSAs) to compress the $b + 1$ inputs to two inputs and then compute the final result using a ripple-carry adder (RCA). Note that the two outputs of a CSA are made up of carries and sums, respectively. For the output with carries, the least-significant bit (LSB) is zero, so we can fill in a sign bit of $s^{j,k}$ there without any extra cost. Hence, for the $b$ sign bits of a PE, $b - 1$ bits are inserted into the CSAs. The last one can also be integrated into the RCA since we can use a full adder instead of a half adder for the LSB of RCA. Meanwhile, since all the inputs are nonnegative integers, we directly omit the most-significant bit

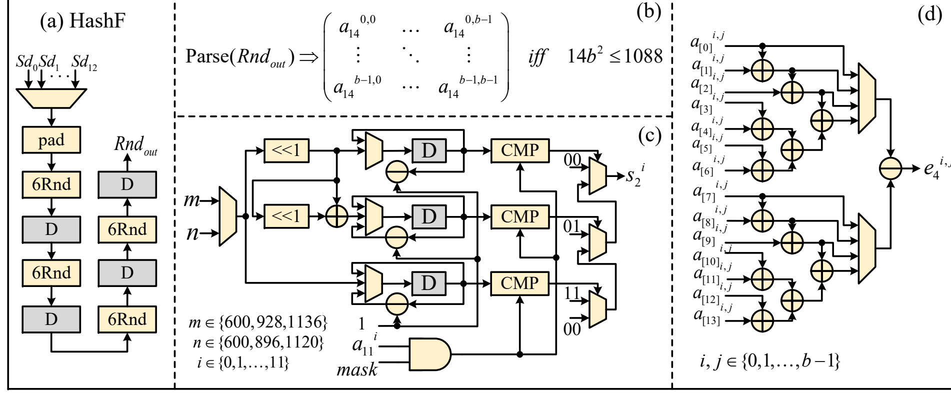


**Figure 5:** The RBitsG module for generating random Matrices $\mathbf{A}$, $\mathbf{S}/\mathbf{S}'$, and $\mathbf{E}/\mathbf{E}'$, and random vectors $\mathbf{seed_A}$, $\mathbf{r}_1$, and $\mathbf{r}_2$. (a) Fully unfolded architecture of HashF, where 6 rounds are merged together and 4 stages of pipelines are inserted. So, continuous random bits for those matrices and vectors can be obtained after 4 cycles of initialization. (b) Demonstration for parsing the output $Rnd_{out}$ into a $b \times b$ block matrix, where each element has 14 bits. The requirement is $14b^2 \leq 1088$ to ensure data for a block is ready in every cycle. (c) Architecture for generating an element of matrix $\mathbf{S}/\mathbf{S}'$. Note that $\max\{\bar{m}, \bar{n}\} = 12$ of such unit are used to reduce the total latency. (d) Architecture for generating an element of matrix $\mathbf{E}/\mathbf{E}'$. Note that $b^2$ of such unit are used to match the size of a block.

(MSB) of each adder (including CSA and RCA) to finish the modulo $2^q$ operation at the same time. Additionally, the RCA unit is also reused to compute the plus operation by $\mathbf{E}/\mathbf{E}'$, $\mathbf{M}$ and $\mathbf{C}_2'$, which does not affect the critical path. Therefore, for $b = 8$ and $q = 12$, the critical path of the proposed adder tree is only $4 + 12 = 16$ full adders.

## 3.3 Random Bits Generator

The RBitsG module mainly includes three parts: 1) hash function for all random bits (HashF), 2) generator for the matrix $\mathbf{S}/\mathbf{S}'$ (Gen_S/S′), and 3) generator for the matrix $\mathbf{E}/\mathbf{E}'$ (Gen_E/E′), as shown in Fig. 5. The details are provided in the following.

HashF: This part is to generate the random bits with different seeds. To increase the overall throughput, we totally unfold the 24-round Keccak core whose implementation is referred to as the work of the Keccak team [BDPVA13]. As shown in Fig. 5(a), six rounds are combined together and four stages of pipelines are inserted. Note that Keccak[512] and Keccak[1024] with rates of 1088 and 576 respectively are involved in Scloud$^+$.KEM, where only the $\mathbf{G}$ hash function adopts the rate of 576. So, we carefully checked all the hash functions used in Scloud$^+$.KEM and designed their valid bits of inputs and outputs more hardware-friendly. The summary of these inputs, outputs, and the corresponding consumed cycles is shown in Table 3, where 13 groups of inputs and outputs are included. Note that the first 7 groups are implemented in a forward manner. Input seeds ($Sd$) are sent in each cycle and continuous random bits are output after 4 cycles. So, the consumed cycles are computed by plus 4. The last 6 groups need to be implemented iteratively because the updated output must be XORed with the next input. So, the consumed cycles are calculated by multiplying 4.

As shown in Fig. 5(b), to generate the random bits for matrices $\mathbf{A}$, $\mathbf{S}/\mathbf{S}'$, and $\mathbf{E}/\mathbf{E}'$, the 1088 random bits of $Rnd_{out}$ are parsed into a form of $b \times b$ matrix with 14 bits for each element. So, the block length $b$ should satisfy the inequality $14b^2 \leq 1088$. Therefore, a block matrix $\mathbf{A}_q^{b\times b}$ of $\mathbf{A} \in \mathbb{Z}_q^{m\times n}$ with $q = 12$ is obtained by simply abandoning the two

**Table 3:** Statistics for Inputs, Outputs, and Consumed Cycles of HashF

| Input | | Output | | Cycle |
|---|---|---|---|---|
| Variable | Valid Bits per Input | Variable | Valid Bits per Output | – |
| $\boldsymbol{\alpha}$ | 256 | $(\mathbf{seed_A}, \mathbf{r}_1, \mathbf{r}_2)$ | $128 + 2 \times 256$ | 4 |
| $\mathbf{r}$ | 256 | $(\mathbf{r}'_1, \mathbf{r}'_2)$ | $2 \times 256$ | 4 |
| [1] $\{\mathbf{seed_A} \vert i \vert j\}$ | $128 + 2\times$ $\max\{\lceil\log \frac{m}{b}\rceil, \lceil\log \frac{n}{b}\rceil\}$ | $\mathbf{A}$ | $qb^2$ | $\lceil \frac{m}{b} \rceil \times \lceil \frac{n}{b} \rceil + 4$ |
| [2] $\{\mathbf{r}_1 \vert i\}$ | $256+$ $\max\{\lceil\log m\rceil, \lceil\log n\rceil\}$ | $\mathbf{S}_{in}$ | $\max\{\lceil\log m\rceil, \lceil\log n\rceil\}$ $\times \max\{\bar{m}, \bar{n}\}$ | [6] $n + \delta_{\mathbf{S}} + 4$ |
| [3] $\{\mathbf{r}'_1 \vert i\}$ | $256+$ $\max\{\lceil\log m\rceil, \lceil\log n\rceil\}$ | $\mathbf{S}'_{in}$ | $\max\{\lceil\log m\rceil, \lceil\log n\rceil\}$ $\times \max\{\bar{m}, \bar{n}\}$ | [6] $m + \delta_{\mathbf{S}'} + 4$ |
| [4] $\{\mathbf{r}_2 \vert i \vert j\}$ | $256 + 2\times$ $\max\{\lceil\log \frac{m}{b}\rceil, \lceil\log \frac{n}{b}\rceil\}$ | $\mathbf{E}_{in}$ | $2 \times \max\{\eta_1, \eta_2\} \times b^2$ | $\lceil \frac{m}{b} \rceil \times \lceil \frac{\bar{n}}{b} \rceil + 4$ |
| [5] $\{\mathbf{r}'_2 \vert i \vert j\}$ | $256 + 2\times$ $\max\{\lceil\log \frac{m}{b}\rceil, \lceil\log \frac{n}{b}\rceil\}$ | $\mathbf{E}'_{in}$ | $2 \times \max\{\eta_1, \eta_2\} \times b^2$ | $\lceil \frac{\bar{m}}{b} \rceil \times \lceil \frac{n+\bar{n}}{b} \rceil + 4$ |
| $pk$ | $qb^2 + 128$ | $\mathbf{hpk}$ | 256 | $\lceil \frac{m}{b} \rceil \times \lceil \frac{\bar{n}}{b} \rceil \times 4$ |
| $\{\mathbf{m} \vert \mathbf{H}(pk)\}$ | $\max\{l_m\} + 256$ | $(\mathbf{r}, \mathbf{k})$ | $2 \times 256$ | 4 |
| $\{\mathbf{m}' \vert \mathbf{hpk}\}$ | $\max\{l_m\} + 256$ | $(\mathbf{r}', \mathbf{k}')$ | $2 \times 256$ | 4 |
| $\{\mathbf{k} \vert \mathbf{C}\}$ | $qb^2 + 256$ | $\mathbf{ss}$ | $\max\{l_{ss}\}$ | $\lceil \frac{\bar{m}}{b} \rceil \times \lceil \frac{n+\bar{n}}{b} \rceil \times 4$ |
| $\{\mathbf{k}' \vert \mathbf{C}\}$ | $qb^2 + 256$ | $\mathbf{ss}$ | $\max\{l_{ss}\}$ | $\lceil \frac{\bar{m}}{b} \rceil \times \lceil \frac{n+\bar{n}}{b} \rceil \times 4$ |
| $\{\mathbf{z} \vert \mathbf{C}\}$ | $qb^2 + 256$ | $\mathbf{ss}$ | $\max\{l_{ss}\}$ | $\lceil \frac{\bar{m}}{b} \rceil \times \lceil \frac{n+\bar{n}}{b} \rceil \times 4$ |

[1] $i \in \{0, 1, ..., \lceil \frac{m}{b} \rceil - 1\}$ and $j \in \{0, 1, ..., \lceil \frac{n}{b} \rceil - 1\}$.
[2] $i \in \{0, 1, ..., n - 1\}$.
[3] $i \in \{0, 1, ..., m - 1\}$.
[4] $i \in \{0, 1, ..., \lceil \frac{m}{b} \rceil - 1\}$ and $j \in \{0, 1, ..., \lceil \frac{\bar{n}}{b} \rceil - 1\}$.
[5] $i \in \{0, 1, ..., \lceil \frac{\bar{m}}{b} \rceil - 1\}$ and $j \in \{0, 1, ..., \lceil \frac{n+\bar{n}}{b} \rceil - 1\}$.
[6] Values of $\delta_{\mathbf{S}}$ and $\delta_{\mathbf{S}'}$ are uncertain due to the rejection sampling.

MSBs of every element of the parsed matrix and then directly sent to the MatM module. The generations for matrices $\mathbf{S}/\mathbf{S}'$ and $\mathbf{E}/\mathbf{E}'$ are more complicated, which are detailed in the following.

Gen_S/S′: As introduced in [WZZ+24], the constant Hamming distribution (CHD) is used to generate the matrix $\mathbf{S} \in \mathbb{Z}_2^{n \times \bar{n}}$ ($\mathbf{S}' \in \mathbb{Z}_2^{\bar{m} \times m}$), where each column (row) contains exactly $\frac{n}{2}$ ($\frac{m}{2}$) zeros, $\frac{n}{4}$ ($\frac{m}{4}$) ones, and $\frac{n}{4}$ ($\frac{m}{4}$) negative ones. To speed up the hardware implementation, we propose a boundary-based sampling method as shown in Alg. 3. We set three decreasing counters $Cnt_0$, $Cnt_1$, and $Cnt_{-1}$, whose initial values are $\frac{l}{2}$, $\frac{3l}{4}$, and $l$ ($l \in \{n, m\}$), respectively. They are updated based on the value of the random number fetched from the parsed matrix. Note that if the fetched number is larger than each of those counters, no counters will be updated and no output will be added. In order to decrease the rejection rate $R_{reject}$, the fetched number is masked according to the size of $Cnt_{-1}$, so we have $R_{reject} < 50\%$. Assuming the length of $Cnt_{-1}$ as $B$, we can compute the average rejection rate as:

$$\begin{aligned} \mu_{Rreject} &= \frac{\frac{2^{B-1}-1}{2^B} + \frac{2^{B-1}-2}{2^B} + ... + \frac{2^{B-1}-2^{B-1}}{2^B}}{2^{B-1}} \\ &= \frac{2^{B-1} - 1}{2^{B+1}}. \end{aligned} \tag{1}$$

When $B$ is relatively large, $\mu_{Rreject}$ is approximate 25%. When $B$ becomes small, $\mu_{Rreject}$ is gradually less than 25%, until to 0%. The corresponding architecture for one element of the sample vector $\mathbf{s}$ is shown in Fig. 5(c). In our design, one vector is generated in serial, and twelve vectors are computed in parallel.

Gen_E/E′: As introduced in [WZZ+24], the central binomial distribution (CBD) is used to generate the matrices $\mathbf{E} \in \mathbb{Z}^{m \times \bar{n}}$ and $\mathbf{E}' \in \mathbb{Z}^{\bar{m} \times (n+\bar{n})}$, whose elements are expressed as $e^{i,j} = \sum_{k=1}^{\eta} (x^{i,j}[k] - y^{i,j}[k])$ where $x^{i,j}[k], y^{i,j}[k] \in \{0, 1\}$ and $\eta \in \{1, 2, 3, 7\}$. It means

**Algorithm 3:** Proposed Boundary-Based Sampling Method of CHD for Gen_S/S′

**Input:** Length of the target vector $l \in \{n, m\}$
**Output:** The sampled vector $\mathbf{s} \in \{0, 1, -1\}^l$

1: $i = 0$
2: $Cnt_0 = \frac{l}{2}, Cnt_1 = \frac{3l}{4}, Cnt_{-1} = l$
3: **while** $Cnt_{-1} > 0$ **do**
4: $mask \leftarrow \{1\}^{\lceil log_2 Cnt_{-1} \rceil}$
5: $idx_{full} \leftarrow \text{Parse}(Rnd_{out})$
6: $idx_{mask} = idx_{full}\ \&\ mask$
7: **if** $idx_{mask} < Cnt_0$ **then**
8: $\mathbf{s}[i] = 0, Cnt_0 = Cnt_0 - 1, Cnt_1 = Cnt_1 - 1, Cnt_{-1} = Cnt_{-1} - 1, i = i + 1$
9: **else if** $idx_{mask} < Cnt_1$ **then**
10: $\mathbf{s}[i] = 1, Cnt_1 = Cnt_1 - 1, Cnt_{-1} = Cnt_{-1} - 1, i = i + 1$
11: **else if** $idx_{mask} < Cnt_{-1}$ **then**
12: $\mathbf{s}[i] = -1, Cnt_{-1} = Cnt_{-1} - 1, i = i + 1$
13: **else**
14: Reject sampling
15: **end if**
16: **end while**
17: **return** $\mathbf{s}$

that $e^{i,j}$ ranges in $\{-\eta, -\eta+1, ..., \eta-1, \eta\}$ and consumes $2\eta$ random bits. Therefore, we set the width of elements of the parsed random matrix to $14 = 2 \times \max\{\eta\}$ to match this requirement and take them directly for the Gen_E/E′ module. In order to make it more hardware-friendly, we transform the original expression of $e^{i,j}$ as follows:

$$e^{i,j} = \sum_{k=1}^{\eta} x^{i,j}[k] - \sum_{k=1}^{\eta} y^{i,j}[k], \tag{2}$$

where two adder trees can be used and only one subtractor is required. The corresponding architecture for an element $e^{i,j}$ is shown in Fig. 5(d). In our design, one $b \times b$ block matrix is generated in parallel and blocks in matrices $\mathbf{E}$, and $\mathbf{E}'$ are computed in serial.

## 3.4 Message Function

The MsgFunc module mainly includes two parts: 1) message encoder for transferring the vector message into a matrix (MsgEnc) and 2) message decoder for recovering the message from the matrix (MsgDec), as shown in Fig. 6. The details are provided below.

MsgEnc: As described in [WZZ+24], the input message vector $\mathbf{m}$ is divided into $\frac{l_m}{\mu}$ groups, and each grouped vector is encoded using the labeling method. During the labeling, a grouped vector is mapped by cutting this data into 16 pieces with different data widths that are defined based on the Hamming weight of their location orders. Next, these data are expressed in complex-number representation and updated by a 32-dimensional Barnes-Wall lattice ($BW_{32}$) expressed as a tensor product. In the tensor product operation, the complex elements $w_j = w_j^{Re} + w_j^{Im}\mathtt{i}$ for $j \in \{2, 4, ..., 16\}$ are iteratively calculated. The main computation is to calculate the formula:

$$w_j^{Re} + w_j^{Im}\mathtt{i} + (\mathtt{i}+1)(w_{j+1}^{Re} + w_{j+1}^{Im}\mathtt{i}) = (w_j^{Re} + w_{j+1}^{Re} - w_{j+1}^{Im}) + (w_j^{Im} + w_{j+1}^{Re} + w_{j+1}^{Im})\mathtt{i}. \tag{3}$$

Note that $l$ is not fixed in each iteration and its maximum number is 16. So, we use 8 plus-minus units (PMUs) and 8 plus-plus units (PPUs) to compute the real parts and imaginary parts, respectively. Five iterations are needed for a grouped vector and each

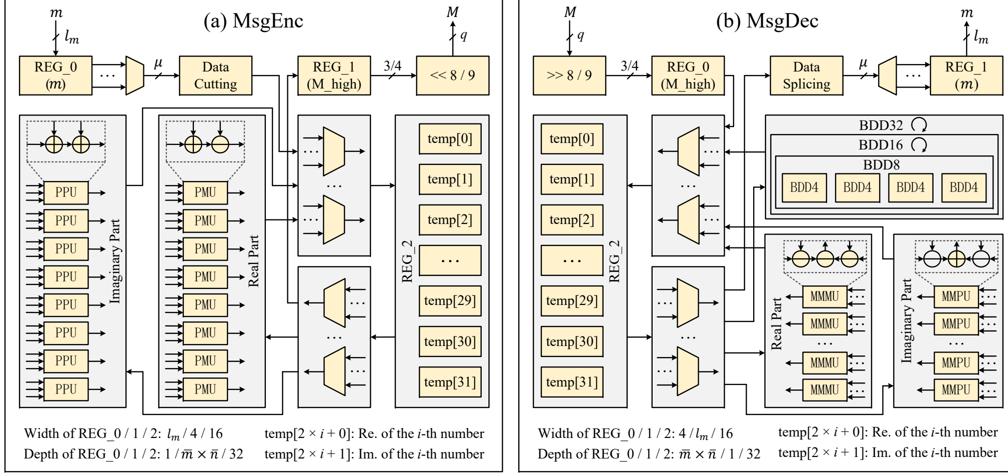


**Figure 6:** The MsgFunc module for decompressing and compressing the message $\mathbf{m}$. (b) The MsgEnc part for decompressing the message vector $\mathbf{m} \in \{0,1\}^{l_m}$ to the matrix $\mathbf{M} \in \mathbb{Z}_q^{\bar{m} \times \bar{n}}$. (c) The MsgDec part for compressing the message vector $\mathbf{m} \in \{0,1\}^{l_m}$ from the matrix $\mathbf{M} \in \mathbb{Z}_q^{\bar{m} \times \bar{n}}$. Note that the circuits in gray are implemented by reusing the existing units.

consumes one cycle. Then, the updated vectors are left-shifted and output. The proposed architecture is shown in Fig. 6(a). Two extra cycles are needed for pre-processing and post-processing the data, respectively. Therefore, this module costs $7 \times \frac{l_m}{\mu}$ cycles. Note that the computations in this module are totally independent of the others and can be computed in advance, so these cycles are implicit for the total time.

MsgDec: This part is the reverse process of MsgEnc, as shown in Fig. 6(b). However, before computing the delabeling, the bounded distance decoding (BDD) problem should be solved, which is much more complicated than the other operations. The $n$-dimensional BDD algorithm ($\text{BDD}_n$) is computed recursively, where four sub-BDDs are included, and each forward recursion contains four BW operations in each forward recursion. In the backward recursions, the Euclidean distances are computed between the updated complex vectors ($\mathbf{x}$ and $\mathbf{x}'$) and the target complex vector ($\mathbf{t}$) and compared for finding the minimum radius (shorted as EdC). Hence, as for $n = 32$, we need to compute 256 $\text{BW}_2$, 64 $\text{BW}_4$, 16 $\text{BW}_8$, 4 $\text{BW}_{16}$, 64 $\text{EdC}_2$, 16 $\text{EdC}_4$, 4 $\text{EdC}_8$, and 1 $\text{EdC}_{16}$.

For the $\text{BDD}_{32}$ computation, we have proposed two optimization techniques. The first one is to make a good trade-off between the area and the speed for the BW and EdC operations. The relationship of iterations and operations for different unfolded factors is summarized in Table 4. We take the unfolded factor equal to 8 in our design where the computation complexity is roughly reduced by 16x and only 16 iterations are needed.

The second optimization technique for BDD is to omit the square root operations for EdCs without any accuracy loss and construct two tree adders. Note that the EdC can be formulated as follows:

$$\mathbf{y} = (\|\mathbf{x} - \mathbf{t}\| < \|\mathbf{x}' - \mathbf{t}\|) \ ? \ \mathbf{x} : \mathbf{x}', \tag{4}$$

where $\|\mathbf{x} - \mathbf{t}\| = \sqrt{(\mathbf{x} - \mathbf{t})^2}$ and $\|\mathbf{x}' - \mathbf{t}\| = \sqrt{(\mathbf{x}' - \mathbf{t})^2}$. It can be found that we do not need the absolute distance values and therefore the square root operations can be totally discarded. Actually, we can still simplify the square operations by using the difference of square formula. But, it would become more complex when adding those signed values together after the difference operations. Additionally, the bit widths of those square operations are only 4 bits. So, we adopt two tree adders similar to those in Gen_E/E′ to

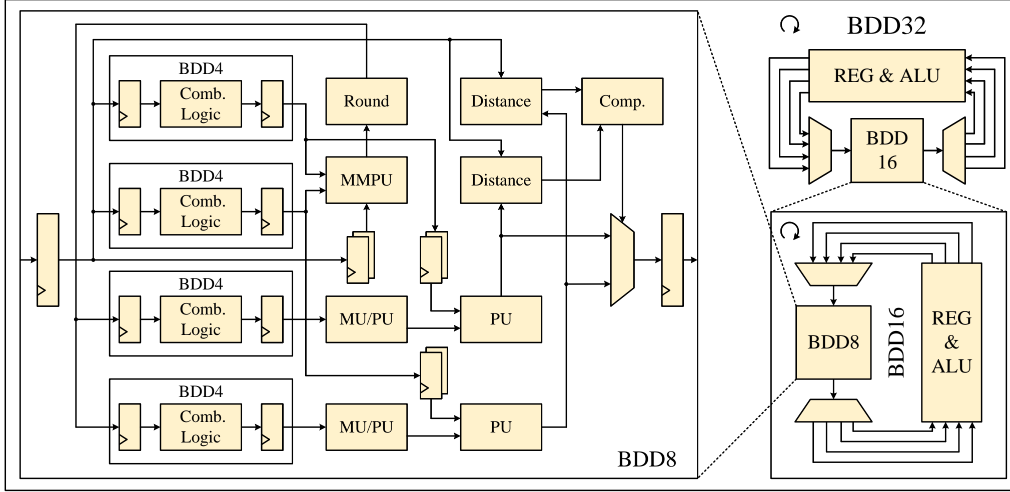


**Figure 7:** Proposed architecture for $\mathrm{BDD}_{32}$.

**Table 4:** Relationship of Iterations and Operations for Different Unfolded Factors in BDD

| Unfolded factor | Iteration | Operation |
|---|---|---|
| 32 | 1 | $256\mathrm{BW}_2 + 64\mathrm{BW}_4 + 16\mathrm{BW}_8 + 4\mathrm{BW}_{16} +$ $64\mathrm{EdC}_2 + 16\mathrm{EdC}_4 + 4\mathrm{EdC}_8 + 1\mathrm{EdC}_{16}$ |
| 16 | 4 | $64\mathrm{BW}_2 + 16\mathrm{BW}_4 + 4\mathrm{BW}_8 + 1\mathrm{BW}_{16} +$ $16\mathrm{EdC}_2 + 4\mathrm{EdC}_4 + 1\mathrm{EdC}_8 + 1\mathrm{EdC}_{16}$ |
| 8 | 16 | $16\mathrm{BW}_2 + 4\mathrm{BW}_4 + 1\mathrm{BW}_8 + 1\mathrm{BW}_{16} +$ $4\mathrm{EdC}_2 + 1\mathrm{EdC}_4 + 1\mathrm{EdC}_8 + 1\mathrm{EdC}_{16}$ |
| 4 | 64 | $4\mathrm{BW}_2 + 1\mathrm{BW}_4 + 1\mathrm{BW}_8 + 1\mathrm{BW}_{16} +$ $1\mathrm{EdC}_2 + 1\mathrm{EdC}_4 + 1\mathrm{EdC}_8 + 1\mathrm{EdC}_{16}$ |
| 2 | 256 | $1\mathrm{BW}_2 + 1\mathrm{BW}_4 + 1\mathrm{BW}_8 + 1\mathrm{BW}_{16} +$ $1\mathrm{EdC}_2 + 1\mathrm{EdC}_4 + 1\mathrm{EdC}_8 + 1\mathrm{EdC}_{16}$ |

reduce the critical path. The details of $\mathrm{BDD}_{32}$ are shown in Fig. 7.

The rest circuits are mainly to compute the delabeling, which are almost the mirror circuits of the labeling in MsgEnc. It should be noted that the tensor product is slightly different and the multiplication factor $(\mathtt{i}+1)$ becomes $\frac{1}{(\mathtt{i}+1)} = \frac{(1-\mathtt{i})}{2}$. So the computation of the elements can be formulated as:

$$
\begin{aligned}
&(1-\mathtt{i})/2 \cdot (w_{j+1}^{Re} + w_{j+1}^{Im}\mathtt{i} - (w_j^{Re} + w_j^{Im}\mathtt{i})) \\
=&(((w_{j+1}^{Im} - w_j^{Im}) + (w_{j+1}^{Re} - w_j^{Re})) + ((w_{j+1}^{Im} - w_j^{Im}) - (w_{j+1}^{Re} - w_j^{Re}))\mathtt{i})/2,
\end{aligned} \tag{5}
$$

where the real part contains two subtractions and one addition and the imaginary part contains three subtractions. Note that the two subtractors in hardware can be easily shared. As shown in Fig. 6(b), 8 minus-minus-minus units (MMMUs) and 8 minus-minus-plus units (MMPUs) are used, where the two minus operations marked in gray in the MMPU are realized by reusing the operations in the corresponding MMMU. Many stages of pipeline are inserted in different layers to reduce the critical path and the total number of cycles used by MsgDec is $153 \times \frac{l_m}{\mu}$.

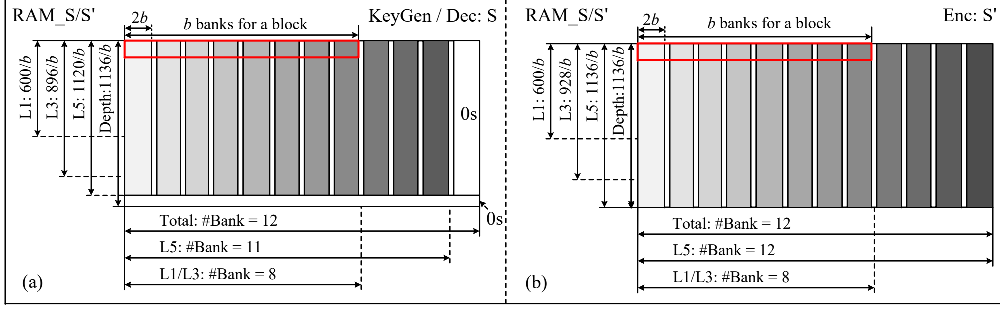


**Figure 8:** The RAM_S/S′ module for saving the matrices $\mathbf{S} \in \mathbb{Z}_2^{n\times\bar{n}}$ and $\mathbf{S}' \in \mathbb{Z}_2^{\bar{m}\times m}$. The total number of bits is $2b \times 1136/b \times 12$, where $b$ is the length of the block matrix. (a) Memory scheduling for the matrix $\mathbf{S} \in \mathbb{Z}_2^{n\times\bar{n}}$, where $b$ is assumed to 8 and $2b \times 1120/b \times 11$ are filled in, activated in the *KeyGen* and *Dec* functions. (b) Memory scheduling for the matrix $\mathbf{S}' \in \mathbb{Z}_2^{\bar{m}\times m}$, where $b$ is assumed to 8 and $2b \times 1136/b \times 12$ are filled in, activated in the *Enc* function.

## 3.5 Memory

According to the designed processing scheduling, the Memory module mainly contains three parts: 1) RAMs for the matrices $\mathbf{S}$ and $\mathbf{S}'$ (RAM_S/S′), 2) RAMs for the matrices $\mathbf{B}$ and $\bar{\mathbf{C}}_1$ (RAM_B/$\bar{\text{C}}_1$), and 3) registers for the matrices $\bar{\mathbf{C}}_2$ and $\mathbf{D}$, and the random-seed vectors $\mathbf{seed_A}$, $\mathbf{r}_1$, and $\mathbf{r}_2$ (REG_$\bar{\text{C}}_2$/D/Seed). Note that the registers used in pipelining are not considered here. The details are provided in the following.

RAM_S/S′: In this part, single-port RAMs are used and the size is $2b \times \max\{m, n\}/b \times \max\{\bar{m}, \bar{n}\} = 2b \times 1136/b \times 12$, as shown in Fig. 8. Matrices $\mathbf{S}$ and $\mathbf{S}'$ share the same memory space in different stages. In the MatM module, $b^2$ 2-bit values of a block from matrices $\mathbf{S}$ or $\mathbf{S}'$ are taken out from the memory, which are generated by RBitsG in advance. To satisfy the requirement of parallelism, we design 12 banks of RAMs, each with a width of $2b$ bits and a depth of $1136/b$. In every cycle, $b$ banks are read simultaneously from the same address and sent to MatM. Additionally, when generating those bits in RBitsG, 12 CHDs are independently worked to write those banks to reduce the total latency.

RAM_B/$\bar{\text{C}}_1$: In this part, dual-port RAMs are used and the size is $qb \times \max\{m, n\}/b \times \max\{\bar{m}, \bar{n}\} = qb \times 1136/b \times 12$, as shown in Fig. 9. It saves the final output data from MatM when $b^2$ $n$-number (or $m$-number) vector multiplications are finished. Matrices $\mathbf{B}$ and $\bar{\mathbf{C}}_1$ share the same memory space in different stages. So, to save matrix $\mathbf{B}$, as shown in Fig. 9(a), the write signals are enabled every $\frac{n}{b}$ cycles. Similarly, to fit with the degree of parallelism, 12 banks of RAMs are used, with a width of $qb$ and a depth of $1136/b$. To save matrix $\bar{\mathbf{C}}_1$, as shown in Fig. 9(b), $q_1 b^2$ bits are write in every $\frac{m}{b}$ cycles, where $(q - q_1)b^2$ are initialized to zeros. It should be noted that the different formats of ciphertexts $\mathbf{C}_1 \in \mathbb{Z}_q^{\bar{m}\times n}$ and $\mathbf{C}_1' \in \mathbb{Z}_q^{\bar{m}\times n}$ are dynamically computed to and from $\bar{\mathbf{C}}_1 \in \mathbb{Z}_{q_1}^{\bar{m}\times n}$, respectively.

REG_$\bar{\text{C}}_2$/D/Seed: This part contains $q \times \bar{m} \times \bar{n} + 128 + 256 + 256 = q\bar{m}\bar{n} + 640$ bits of registers. Matrices $\bar{\mathbf{C}}_2$ and $\mathbf{D}$ share the same memory space in different stages. The 640 random bits for $\mathbf{Seed_A}$, $\mathbf{r}_1$, and $\mathbf{r}_2$ are directly saved in registers when generated by RBitsG at the beginning of the *KeyGen* stage and then used repeatedly. For saving the matrix $\bar{\mathbf{C}}_2 \in \mathbb{Z}_{q_2}^{\bar{m}\times\bar{n}}$, the $q\bar{m}\bar{n}$ registers are enabled after $\frac{m}{b}$ cycles. Similarly, the intermediate ciphertexts $\mathbf{C}_2 \in \mathbb{Z}_q^{\bar{m}\times\bar{n}}$ and $\mathbf{C}_2' \in \mathbb{Z}_q^{\bar{m}\times\bar{n}}$ are dynamically calculated. For saving the matrix $\mathbf{D} \in \mathbb{Z}_q^{\bar{m}\times\bar{n}}$, these registers are enabled after $\frac{n}{b}$ cycles. If $b \geq \bar{m}$ and $b \geq \bar{n}$, those enable signals are only valid once; otherwise, they are valid $\lceil \frac{m}{b} \rceil$ or $\lceil \frac{n}{b} \rceil$ times.

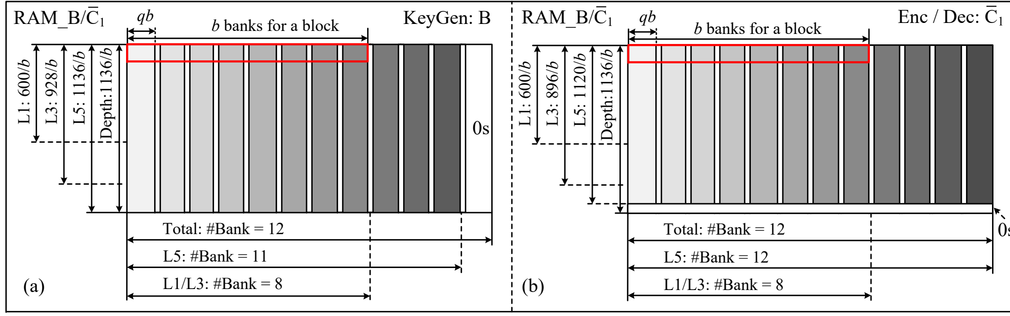


**Figure 9:** The RAM_B/C$_1$ module for saving the matrices $\mathbf{B} \in \mathbb{Z}_q^{m\times\bar{n}}$ and $\bar{\mathbf{C}}_1 \in \mathbb{Z}_{q_1}^{\bar{m}\times n}$ where $q_1 < q$ and $(q-q_1)$ MSBs are set to zeros. The total number of bits is $qb\times 1136/b\times 12$, where $b$ is the length of the block matrix. (a) Memory scheduling for the matrix $\mathbf{B} \in \mathbb{Z}_q^{m\times\bar{n}}$, where $b$ is assumed to 8 and $qb \times 1136/b \times 11$ are filled in, activated in the *KeyGen* function. (b) Memory scheduling for the matrix $\bar{\mathbf{C}}_1 \in \mathbb{Z}_{q_1}^{\bar{m}\times n}$, where $b$ is assumed to 8 and $qb \times 1120/b \times q_1$ are filled in, activated in the *Enc* and *Dec* functions.

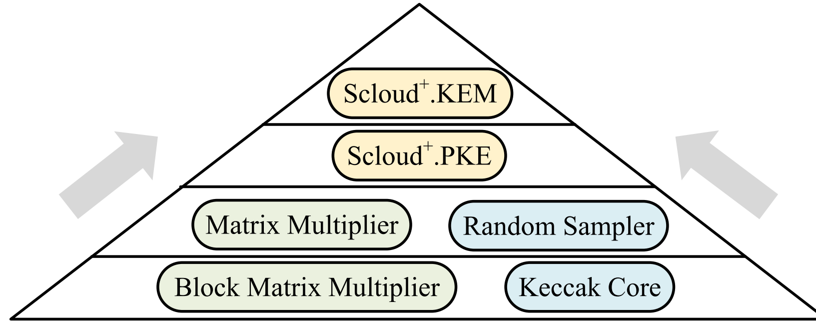


**Figure 10:** Computation breakdown for Scloud$^+$.

## 3.6 Control Unit

This module is to combine the aforementioned modules together to implement different algorithms in Scloud$^+$.KEM. All of those computations in our design can be broken down as shown in Fig. 10. The processing flow is from bottom to top. The basic computation modules are the block matrix multiplier and the Keccak core. Based on them, we obtain different sizes of matrix multipliers and random samplers, respectively. We combine these multipliers and samplers and get the three functions in Scloud$^+$.PKE. Finally, we use the three Scloud$^+$.PKE functions and three hash functions to achieve the three stages of Scloud$^+$.KEM. The operations in order and their required cycles in the formulas of Scloud$^+$.PKE and Scloud$^+$.KEM are summarized as shown in Talbe. 5. Except for the block parameter $b$, all of the other parameters are related to the security levels. When we fix the parameter $b$, the computation cycles can be calculated by substituting numerical values of the parameters for a certain security level into those formulas. Note that the parameters $\delta_{\mathbf{S}}$ and $\delta_{\mathbf{S}'}$ are determined by experiments, which are heavily related to the average rejection rate. It means that about an extra 25% cycles would be consumed in this step.

**Table 5:** Operations in Order and Their Required Cycles in Formula of Scloud$^+$.PKE and Scloud$^+$.KEM

| Scloud$^+$.PKE.*KeyGen* | |
|---|---|
| $(\mathbf{seed_A}, \mathbf{r}_1, \mathbf{r}_2) = \mathtt{F}(\boldsymbol{\alpha}) \in \{0,1\}^{128} \times \{0,1\}^{256} \times \{0,1\}^{256}$ | 4 |
| $\mathbf{S} = \Psi(\mathbf{r}_1, (n, \bar{n}), h_1) \in \mathbb{Z}^{n\times\bar{n}}$ | $^*n + \delta_{\mathbf{S}} + 4$ |
| $\mathbf{B} = \mathbf{A} \cdot \mathbf{S} \in \mathbb{Z}_q^{m\times\bar{n}}$ | $\lceil \frac{m}{b} \rceil \times \lceil \frac{n}{b} \rceil \times \lceil \frac{\bar{n}}{b} \rceil + 4$ |
| $\mathbf{B} = \mathbf{B} + \mathbf{E} \in \mathbb{Z}_q^{m\times\bar{n}}$ | $\lceil \frac{m}{b} \rceil \times \lceil \frac{\bar{n}}{b} \rceil + 4$ |
| Scloud$^+$.PKE.*Enc* | |
| $(\mathbf{r}_1', \mathbf{r}_2') = \mathtt{F}(\mathbf{r}) \in \{0,1\}^{256\times 2}$ | 4 |
| $\mathbf{S}' = \Phi(\mathbf{r}_1', (\bar{m}, m), h_2) \in \mathbb{Z}^{\bar{m}\times m}$ | $^*m + \delta_{\mathbf{S}'} + 4$ |
| $\mathbf{C}_2 = \mathbf{S}' \cdot \mathbf{B} \in \mathbb{Z}_q^{\bar{m}\times\bar{n}}$ | $\lceil \frac{\bar{m}}{b} \rceil \times \lceil \frac{m}{b} \rceil \times \lceil \frac{\bar{n}}{b} \rceil$ |
| $\mathbf{C}_1 = \mathbf{S}' \cdot \mathbf{A} \in \mathbb{Z}^{\bar{m}\times n}$ | $\lceil \frac{m}{b} \rceil \times \lceil \frac{n}{b} \rceil \times \lceil \frac{\bar{m}}{b} \rceil + 4$ |
| $\bar{\mathbf{C}}_1 = \lfloor \frac{q_1}{q} \cdot (\mathbf{C}_1 + \mathbf{E}_1) \rceil \in \mathbb{Z}^{\bar{m}\times n}$ | $\lceil \frac{n}{b} \rceil \times \lceil \frac{\bar{m}}{b} \rceil + 4$ |
| $\mathbf{C}_2 = \mathbf{C}_2 + \mathbf{E}_2 \in \mathbb{Z}_q^{\bar{m}\times\bar{n}}$ | $\lceil \frac{\bar{m}}{b} \rceil \times \lceil \frac{\bar{n}}{b} \rceil$ |
| $\bar{\mathbf{C}}_2 = \lfloor \frac{q_2}{q} \cdot (\mathbf{C}_2 + \mathbf{M}) \rceil_{\text{odd}} \in \mathbb{Z}_q^{\bar{m}\times\bar{n}}$ | $\lceil \frac{\bar{m}}{b} \rceil \times \lceil \frac{\bar{n}}{b} \rceil$ |
| Scloud$^+$.PKE.*Dec* | |
| $\mathbf{D} = (\frac{q}{q_1} \cdot \bar{\mathbf{C}}_1') \cdot \mathbf{S} \in \mathbb{Z}_q^{\bar{m}\times\bar{n}}$ | $\lceil \frac{\bar{m}}{b} \rceil \times \lceil \frac{n}{b} \rceil \times \lceil \frac{\bar{n}}{b} \rceil$ |
| $\mathbf{D} = (\frac{q}{q_2} \cdot \bar{\mathbf{C}}_2) - \mathbf{D} \in \mathbb{Z}_q^{\bar{m}\times\bar{n}}$ | $\lceil \frac{\bar{m}}{b} \rceil \times \lceil \frac{\bar{n}}{b} \rceil$ |
| $\mathbf{m} = \mathtt{MsgDec}(\mathbf{D}) \in \{0,1\}^{l_m}$ | $57 \times \frac{l_m}{\mu}$ |
| Scloud$^+$.KEM.*KeyGen* | |
| $(pk, sk') = $ Scloud$^+$.PKE.*KeyGen*() | $\lceil \frac{m}{b} \rceil \times \lceil \frac{\bar{n}}{b} \rceil \times (\lceil \frac{n}{b} \rceil + 1)$ <br> $+n + \delta_{\mathbf{S}} + 16$ |
| $\mathbf{hpk} = \mathtt{H}(pk) \in \{0,1\}^{256}$ | $\lceil \frac{m}{b} \rceil \times \lceil \frac{\bar{n}}{b} \rceil \times 4$ |
| Scloud$^+$.KEM.*Encaps* | |
| $\mathtt{H}(pk) \in \{0,1\}^{256}$ | $\lceil \frac{m}{b} \rceil \times \lceil \frac{\bar{n}}{b} \rceil \times 4$ |
| $(\mathbf{r}, \mathbf{k}) = \mathtt{G}(\mathbf{m} \| \mathtt{H}(pk)) \in \{0,1\}^{256\times 2}$ | 4 |
| $\mathbf{C} = $ Scloud$^+$.PKE.*Enc*$(pk, \mathbf{m}, \mathbf{r})$ | $\lceil \frac{\bar{m}}{b} \rceil \times \lceil \frac{\bar{n}}{b} \rceil \times (\lceil \frac{m}{b} \rceil + 2)$ <br> $\lceil \frac{\bar{m}}{b} \rceil \times \lceil \frac{n}{b} \rceil \times (\lceil \frac{m}{b} \rceil + 1)$ <br> $+m + \delta_{\mathbf{S}'} + 20$ |
| $\mathbf{ss} = \mathtt{K}(\mathbf{k} \| \mathbf{C})$ | $\lceil \frac{\bar{m}}{b} \rceil \times (\lceil \frac{n}{b} + \lceil \frac{\bar{n}}{b}) \rceil \times 4$ |
| Scloud$^+$.KEM.*Decaps* | |
| $\mathbf{m}' = $ Scloud$^+$.PKE.*Dec*$(sk', \mathbf{C})$ | $\lceil \frac{\bar{m}}{b} \rceil \times \lceil \frac{\bar{n}}{b} \rceil \times (\lceil \frac{n}{b} \rceil + 1)$ <br> $+57 \times \frac{l_m}{\mu}$ |
| $(\mathbf{r}', \mathbf{k}') = \mathtt{G}(\mathbf{m}' \| \mathbf{hpk})$ | 4 |
| $\mathbf{C}' = $ Scloud$^+$.PKE.*Enc*$(pk, \mathbf{m}', \mathbf{r}')$ | $\lceil \frac{\bar{m}}{b} \rceil \times \lceil \frac{\bar{n}}{b} \rceil \times (\lceil \frac{m}{b} \rceil + 2)$ <br> $\lceil \frac{\bar{m}}{b} \rceil \times \lceil \frac{n}{b} \rceil \times (\lceil \frac{m}{b} \rceil + 1)$ <br> $+m + \delta_{\mathbf{S}'} + 20$ |
| $\mathbf{ss} = \mathtt{K}(\mathbf{k}' \| \mathbf{C})$ or $\mathbf{ss} = \mathtt{K}(\mathbf{z}, \mathbf{C})$ | $\lceil \frac{\bar{m}}{b} \rceil \times (\lceil \frac{n}{b} + \lceil \frac{\bar{n}}{b}) \rceil \times 4$ |

* Values of $\delta_{\mathbf{S}}$ and $\delta_{\mathbf{S}'}$ are uncertain due to the rejection sampling, which are heavily related to the average rejection rate.

**Table 6:** Timing Performance of Scloud$^+$.PKE and Scloud$^+$.KEM, with a Frequency of 300MHz

| Stages | | Scloud$^+$.PKE | | | Scloud$^+$.KEM | | |
|---|---|---|---|---|---|---|---|
| | | KeyGen | Enc | Dec | KeyGen | Encaps | Decaps |
| Scloud$^+$-128 | Latency (CCs) | 6602 | 6707 | 379 | 6902 | 7286 | 7370 |
| | Time ($us$) | 22.0 | 23.4 | 1.3 | 23.0 | 24.3 | 24.6 |
| Scloud$^+$-192 | Latency (CCs) | 14468 | 14604 | 479 | 14932 | 15493 | 15483 |
| | Time ($us$) | 48.2 | 48.7 | 1.6 | 49.8 | 51.6 | 51.6 |
| Scloud$^+$-256 | Latency (CCs) | 41730 | 42349 | 1176 | 42866 | 44623 | 44682 |
| | Time ($us$) | 139.1 | 141.2 | 3.9 | 142.9 | 148.7 | 148.9 |

# 4 Implementation Results and Comparison

In this section, the hardware implementation results of Scloud$^+$.KEM are presented first, mainly including area and timing. Then, the comparison with hardware implementations of Kyber is provided.

## 4.1 Implementation Results

The block size is set to 8, which is optimal because all the matrix sizes are exactly divided by 8 and the 1088 output bits of the Keccak core are basically utilized. We coded the proposed architecture in Verilog and ran the simulation over the Xilinx Vivado 2018.3 EDA platform. We have also tried to implement the design based on an FPGA core but found that the most cost module was the Keccak core. Note that Keccak is mainly made up of logic operations, which are very unfriendly to implement on FPGA. The basic unit of most FPGAs is CLB which contains two slices, each of which mainly includes 4 LUT6s, 8 FFs, 3 MUXs, and 1 CARRY4. When implementing logic operations, the utilization of a slice is very low and the place-and-route is very complicated. So the FPGA implementation results are not as good as expected. However, when we implemented our design in ASIC, we got the expected results. So, we only provide and compare the ASIC results in the following. Our design is synthesized under the SMIC 40nm CMOS technology and the sweet point of clock frequency is 300MHz.

Table 6 shows the timing performance of Scloud$^+$.PKE and Scloud$^+$.KEM, where the latency and time of three sets of parameters all are provided. It should be noted that the computation cycles are almost consistent with our theoretical analysis in Table 5. Several cycles are added because of the wait for outputs of adjacent operations. Additionally, the parameters of $\delta_{\mathbf{S}}$ and $\delta_{\mathbf{S}'}$ both are equal to about half of $n$ and $m$, respectively. It means that the rejection rate is about 50%. The reason is that we make the stop sign enabled until all of the 12 banks of RAM are finished. So, the worst case decides the time. As the increased time is negligible to the total time, we let it alone to simplify the control logic. Another phenomenon can be observed: the time of Scloud$^+$-192 is about double that of Scloud$^+$-128, while the time of Scloud$^+$-256 is about triple that of Scloud$^+$-192. The reason is that the size of block $b = 8$ is smaller than $\bar{m} = 12$ and $\bar{n} = 11$ in Scloud$^+$-256 so an extra iteration is needed in matrix multiplications with a relatively low area utilization. How to reduce the gap and improve resource utilization could be explored in the future.

**Table 7:** Area Performance of Scloud$^+$.KEM under 40nm LP CMOS Technology

| | Area ($um^2$) | Logic Gates ($10^3$ NAND2 equiv.) | Percentage |
|---|---|---|---|
| Top-Level | 387268 | 606.6 | 100% |
| MatM | 41645 | 65.3 | 10.7% |
| RBitsG | 279799 | 438.3 | 72.3% |
| MsgFunc | 24960 | 39.1 | 6.4% |
| CtrU | 40864 | 64.0 | 10.6% |
| *RAM | 302414 | / | / |
| Core area = 0.69 $mm^2$ | | | |

* **Single port RAM**: $3.375 + 20.25 = 23.625$KB, where the 20.25KB are for saving the received ciphertext **C** in Scloud$^+$.KEM.*Decaps*;
**Simple dual port RAM**: 20.25KB.

Table 7 shows the area performance for Scloud$^+$.KEM, where the absolute area and logic gates both are provided. It can be seen that RBitsG occupies about 70% of the total logic area, where the Keccak core takes the most part. The other three modules are almost equally dividing the rest of the logic area. For the memory of RAM, they are generated by RAM IPs and should be computed individually. Note that the received ciphertext **C** in Scloud$^+$.KEM.*Decaps* is also saved in an extra single-port RAM since it has to be frequently accessed twice, which in deed nearly doubles the required RAM resources. How to reduce this consumption could be an interesting issue in the future exploring.

## 4.2 Comparison

Note that Scloud$^+$.KEM has been proposed recently and no hardware implementation was presented before. So, we only compare our design with hardware implementations of Kyber, which has been standardized by NIST in August, 2024. Many hardware designs for Kyber have been presented before, including on FPGA and ASIC. As explained above, we only focus on the implementation on ASIC. So, we pick up three representative works on ASIC for comparison, as shown in Table 8. Since the area of the proposed design is mainly made up of RAM IPs and logic operations, we assume the area utilization after layout as 70% for a fair comparison. The first work in [KPM+22] proposes a configurable energy-efficient *structured*-LWE-based LB-PQC processor. It can be configured with large ranges of polynomial dimensions and modulo sizes and shows the best performance in terms of area, speed, and energy compared to similar works in LB-PQC. Compared with this work, our design is about 6x faster and has comparable area efficiency though the their work adopts a more advanced CMOS technology. The second work in [ZZO+24] presents a high-performance PQC processor under the 28nm HPC CMOS technology. It covers 7 PQC schemes, involving three mathematical problems, namely the *structured*-LWE, hash, and code. This work achieves the best timing performance among existing published works for Kyber512. Meanwhile, this design consumes the most area, about 3x more than ours even though the adopted technology is better. If our design is merged into their processor as the fourth mathematical problem, the added area will be not significant. The third work in [LLLL24] proposes a specially-optimized design for Kyber and implements it under the 40nm LP CMOS technology. It acheives the best area efficiency. Compared to this work, our design consumes about 35.7% more time and 130% more area. Overall, the gap is relatively small and it could be called ′comparable′ to Kyber.

**Table 8:** Performance Comparison with ASIC Implementations of Kyber

| | ESSCIRC'22 [KPM+22] | JSSC'24 [ZZO+24] | CICC'24 [LLLL24] | **This Work** |
|---|---|---|---|---|
| Scheme | Kyber, Dilithum | Kyber, Dilithium, Falcon, Sphincs+, BIKE, McEliece, HQC | Kyber | Scloud$^+$.KEM |
| Technology | 28nm LP | 28nm HPC | 40nm LP | 40nm LP |
| Report | Layout | Layout | Layout | Synthesis |
| Frequency (MHz) | 190 | 500 | 180 | 300 |
| Core Area ($mm^2$) | 0.18 | 3.2 | 0.43 | *0.69 |
| Logic Gates (NAND2 equiv.) | 123k | 2.1M | 302k | *607k |
| On-chip Memory (kB) | 31 | 228.5 | 9 | 43.9 |
| Kyber512 and Scloud$^+$.KEM-128 (*KeyGen*+*Encaps*+*Decaps*) | | | | |
| Latency (CCs) | 83148 | 7197 | 9506 | 21558 |
| Time ($us$) | 438 | 14 | 53 | 71.9 |
| Area Efficiency (Time×GE) | 53.9 | 29.4 | 16.0 | *43.6 |

* Assume the area utilization after layout as 70%. The equivalent core area, logic gates, and area efficiency should be **0.99** $mm^2$, **867k**, and **62.3**, respectively.

# 5 Conclusion

In this paper, we have presented a fast hardware implementation for Scloud$^+$.KEM based on the proposed high-speed and low-complexity block matrix multiplier and low-latency unfolded Keccak core. The experimental results show that the proposed design is almost as fast as the optimal customized Kyber design with an about doubled area consumption. It demonstrates that we have almost removed the computational gap between the *unstructured*-LWE-based KEM and the *structured*-LWE-based KEM. We hope that these achievements would greatly contribute to the Scloud$^+$.KEM's competitiveness over other PQC candidates.

In the future, we would like to further reduce the RAM consumption and increase the speed of Scloud$^+$.KEM-256 as mentioned in Section 4.1. Additionally, the side-channel attack would also be explored.

# References

[BDPVA13] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. Keccak. In *Annual international conference on the theory and applications of cryptographic techniques*, pages 313–314. Springer, 2013.

[CJL+16] Lily Chen, Stephen Jordan, Yi-Kai Liu, Dustin Moody, Rene Peralta, Ray Perlner, and Daniel Smith-Tone. *Report on post-quantum cryptography*, volume 12. US Department of Commerce, National Institute of Standards and Technology, 2016.

[D'A20] Jan-Pieter D'Anvers. SABER. Technical report, National Institute of Standards and Technology, 2020.

[KPM$^+$22] ByungJun Kim, Jaehan Park, Seunghyun Moon, Kiseo Kang, and Jae-Yoon Sim. Configurable energy-efficient lattice-based post-quantum cryptography processor for IoT devices. In *ESSCIRC 2022-IEEE 48th European Solid State Circuits Conference (ESSCIRC)*, pages 525–528. IEEE, 2022.

[LLLL24] Aobo Li, Jiahao Lu, Dongsheng Liu, and Xiang Li. A 40nm 1.26 $\mu$j/Op energy-efficient CRYSTALS-KYBER post-quantum crypto-processor with comprehensive side channel security analysis and countermeasures. In *2024 IEEE Custom Integrated Circuits Conference (CICC)*, pages 1–2. IEEE, 2024.

[LPR10] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On ideal lattices and learning with errors over rings. In *Advances in Cryptology–EUROCRYPT 2010: 29th Annual International Conference on the Theory and Applications of Cryptographic Techniques, French Riviera, May 30–June 3, 2010. Proceedings 29*, pages 1–23. Springer, 2010.

[LS15] Adeline Langlois and Damien Stehlé. Worst-case to average-case reductions for module lattices. *Des. Codes Cryptogr.*, 75(3):565–599, 2015.

[Nae20] Michael Naehrig. FrodoKEM. Technical report, National Institute of Standards and Technology, 2020.

[oST24] National Institute of Standards and Technology. NIST FIPS Standards. In *https://csrc.nist.gov/publications/fips*, 2024.

[PRSD17] Chris Peikert, Oded Regev, and Noah Stephens-Davidowitz. Pseudorandomness of ring-LWE for any ring and modulus. In *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing*, pages 461–473, 2017.

[Sch22] Peter Schwabe. CRYSTALS-KYBER. Technical report, National Institute of Standards and Technology, 2022.

[Sho94] Peter W Shor. Algorithms for quantum computation: discrete logarithms and factoring. In *Proceedings 35th annual symposium on foundations of computer science*, pages 124–134. Ieee, 1994.

[Wal64] Christopher S Wallace. A suggestion for a fast multiplier. *IEEE Transactions on electronic Computers*, (1):14–17, 1964.

[WZZ$^+$24] Anyu Wang, Zhongxiang Zheng, Chunhuan Zhao, Zhiyuan Qiu, Guang Zeng, Ye Yuan, Changchun Mu, and Xiaoyun Wang. Scloud+: a lightweight LWE-based KEM without Ring/Module structure. *Cryptology ePrint Archive, Paper 2024/1306, https://eprint.iacr.org/2024/1306*, accepted by Security Standardisation Research (SSR) Conference 2024.

[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*, pages 328–356, 2021.

[ZYF$^+$20] Jiang Zhang, Yu Yu, Shuqin Fan, Zhenfeng Zhang, and Kang Yang. Tweaking the asymmetry of asymmetric-key cryptography on lattices: KEMs and signatures of smaller sizes. pages 37–65, 2020.

[ZZO$^+$24] Yihong Zhu, Wenping Zhu, Yi Ouyang, Junwen Sun, Qi Zhao, Min Zhu, Jinjiang Yang, Chen Chen, Qichao Tao, Hanning Wang, et al. PQPU: A 4.4-$mu$j/Op 69.4-kOPs agile post-quantum crypto-processor across multiple mathematical problems. *IEEE Journal of Solid-State Circuits*, 2024.