Compact GF(2) systemizer and optimized constant-time hardware sorters for Key Generation in Classic McEliece

Yihong Zhu[1], Wenping Zhu[1], Chen Chen[1], Min Zhu[2], Zhengdong Li[1], Shaojun Wei[1] and Leibo Liu[1]

[1] Tsinghua University, China, zhuyihon18@mails.tsinghua.edu.cn
[2] Wuxi Micro Innovation Integrated Circuit Design Co.Ltd

Abstract. Classic McEliece is a code-based quantum-resistant public-key scheme characterized with relative high encapsulation/decapsulation speed and small ciphertexts, with an in-depth analysis on its security. However, slow key generation with large public key size make it hard for wider applications. Based on this observation, a high-throughput key generator in hardware, is proposed to accelerate the key generation in Classic McEliece based on algorithm-hardware co-design. Meanwhile the storage overhead caused by large-size keys is also minimized. First, compact large-size GF(2) Gauss elimination is presented by adopting naive processing array, singular matrix detection-based early abort, and memory-friendly scheduling strategy. Second, an optimized constant-time hardware sorter is proposed to support regular memory accesses with less comparators and storage. Third, algorithm-level pipeline is enabled for high-throughput processing, allowing for concurrent key generation based on decoupling between data access and computation

Keywords: Post-quantum cryptography · McEliece · high-throughput · GF(2) · Gauss elimination · constant-time sort · FPGA

1 Introduction

McEliece is based on Goppa code, which is a well-studied construction proposed 40 years ago with thorough security analysis up to now. In July 2022, McEliece has been selected as a key encapsulation mechanism (KEM) finalist in fourth round [GAso22] of National Institute of Standards and Technology (NIST) PQC standardization process. In the 3-round documentation of NIST, it was stated that "Classic McEliece has a very large public key size and fairly slow key generation. This is likely to make Classic McEliece undesirable in many common settings". As a result of its low key generation and large public key, this algorithm is not directly considered to be standardized by NIST, but will continue to be investigated in the fourth round. The focus of our work is to alleviate these problems by hardware acceleration. We hope that our work can provide higher confidence on the performance of Classic McEliece for further evaluation and inspire more insightful work on hardware implementation. For certification authority or public-key infrastructure, high-throughput and massively concurrent key generations are required. These are the target scenarios of this work, where the throughput has the highest priority and memory efficiency (throughput/area) is the second optimization objective.

Based on current performance analysis, more than two magnitudes higher cycles are consumed by key generation than encapsulation or decapsulation [MRAso20]. Therefore, key generation process is becoming the key bottleneck of McEliece, which limits its wider

applications.Large-size GF(2) Gauss elimination process during key generation accounts for the largest part, whose performance directly influence the whole latency. There are multiple McEliece implementations already on various software platforms [BCS13, Cho17, RKK21, CC21] or hardware platforms [SWM+10, WSN17, WSN18, KRGF+21]. [RKK21] presented LUP decomposition method to reduce the memory requirements of large GF(2) Gauss elimination for embedding processors, which was improved further in [CC21]. However, much additional resource usage need to be included if the software method is straightforwardly implemented in hardware, because matrix multiplication and matrix inversion are both involved in the LUP decomposition. Therefore, the memory reduction method in hardware implementation needs to be reconsidered.

Meanwhile, [SWM+10] firstly implemented McEliece in hardware. But multiple modules design was inconsistent with the McEliece specifications proposed later in NIST 3rd-round, including the irreducible polynomial generation, usage of pseudo-random number generator (PRNG) and organization of secret key. A key generator in hardware was designed in [WSN17], including serial hardware sorter, additive FFT hardware, Karatsuba-based polynomial multiplication hardware and so on. Besides, large-size GF(2) Gauss elimination hardware and GF(2^m) matrix inverter were both based on the design [WSN16]. The work [WSN18] extended the design [WSN17] to support the complete schemes including encryption and decryption stages. The performance of encryption and decryption stages achieved only 6k cycles per encryption and 14k cycles per decryption. By contrast, up to 966k cycles are needed to generate the key pairs per attempt, not to mention that at average 3.5 attempts are needed for each successful key-pair generation on average. The work [CCD+22] introduced the early-abort design to optimize the overheads of failed attempts. [KRGF+21] utilized high-level synthesis method to accelerate McEliece in hardware-software co-design way. And the results revealed that the throughput is inferior to full-hardware implementation [WSN17] due to the methodology limitations. In key generation hardware [WSN18], GF(2) Gauss elimination occupies 76.4% cycles of key generation. Besides, constant-time sorter occupies 15.2% cycles, assuming successful operation for each attempt. Therefore, the GF(2) Gauss elimination and constant-time sorter are two dominating functions to accelerate the key generation. This is also the primary motivation of this work.

In this work, a high-throughput key generator for McEliece, is proposed based on the multiple optimized modules and algorithm-level pipeline. The optimized large-size GF(2) matrix systemizer reuses two small data memory during calculations and tailors the instruction memory to half size through improving finite machine in each cell. On the basis of reducing storage, various other strategies are adopted to save the cycle overheads. Many optimization strategies are adopted, including a novel constant-time hardware sorter with less comparators compared to existing work. The pipeline design enables our processor to process multiple key generation through dividing and scheduling flow stages. The design has been verified on the FPGA platforms.

Our Contributions. The key contributions of this work are summarized as follows:

1. **Optimized GF(2) matrix systemizer.** Novel architecture design and efficient memory-reusage scheduling strategy, coupled with other full-pipeline method and early-abort detection hardware, enable the hardware systemizer with only 31.8% memory usage and less cycle overheads.

2. **Efficient constant-time hardware sorter.** Native sorting network design as well as regular memory accesses is adopted to reduce half of comparators and most storage with fewer cycle overhead costs compared to state-of-the-art work.

3. **Algorithm-level pipeline design.** Multiple key generation tasks are enabled to execute in parallel in this work to achieve additional improvement.

Algorithm 1 Key generation algorithm in Classic McEliece [MRAso20]

Input: seed; m,t,n,$f(x)$: parameters; ($mt = m \times t$)
Output: ($g(x)$,$(\alpha_0, \alpha_1, ..., \alpha_{n-1})$):secret key, $\hat{T}$:public key
1: (degree-t polynomial β, 2^m 32-bit random prefixes, newseed) <- PRNG(seed);
2: Generate irreducible polynomial g through calculating minimal polynomial of β;
3: Generate a random full permutation $(\alpha_0, \alpha_1, ..., \alpha_{n-1})$ through sorting random prefixes;
4: Generate the $t \times n$ matrix H calculated by g and $(\alpha_0, \alpha_1, ..., \alpha_{n-1})$;
5: Replace each entry in H with a column of bits to generate $mt \times n$ matrix T;
6: Transform matrix T into its systematic form $[I_{mt}, \hat{T}]$. If matrix T is singular, restart;

2 Background

In this section, we briefly overview the key generation stage in McEliece algorithm. Then, existing works on the large-size GF(2) matrix hardware systemization solutions are analyzed. Finally, current constant-time hardware sorter are discussed.

2.1 Key generation of McEliece

The functions in key generation keep evolving along with the emergence of McEliece's variants [R.M79, BCS13, Cho17, MRAso20]. The latest procedure of key generation in Classic McEliece's specification is demonstrated in Algo. 1.

As Algo. 1 shows, the PRNG in McEliece utilizes SHAKE-256 to generate the random bits for degree-t random polynomial β, random prefixes and 256-bit new seed for possible restart once it fails this time. The irreducible polynomial g is generated by calculating the minimal polynomial of β, which requires t-1 degree-t polynomial multiplications over $f(x)$ and Gauss elimination are involved in the calculation of minimal polynomial, where coefficients belongs to GF(2^m). The random full-permutation $\alpha_0, \alpha_1, ..., \alpha_{n-1}$ is achieved by adding random prefixes and sorting later, because sort process can be implemented in constant-time way. If there are two same values during the process of sorting, or it fails to generate the polynomial g, it will also restart. After generating secret key, $t \times n$ matrix H is constituted through $h_{ij} = {a_j}^i / g(a_j)$, where all the values of $g(a_j)$ are calculated fast through additive FFT methodology. Then the matrix H is transformed into binary parity check matrix T by replacing each entry with a column of m bits. Finally, Gauss elimination process is executed on T to obtain the final $\hat{T}$.

2.2 Existing GF(2) Gauss elimination hardware solutions

As the most time-consuming function in key generation, GF(2) Gauss elimination has been implemented in prior works [SWM+10, WSN16]. Two types of Gauss systolic arrays were utilized in [SWM+10] for triangularization and systemization, respectively. That design was optimized in [WSN16] by combining those two arrays into a single hybrid array. And the implementation [WSN17, WSN18] directly adopted the design in [WSN16].

In that array, two types of processing elements are adopted: processor AB and processor B. There is an 1-bit register in each processor cell. The processors AB are located at the diagonal positions of array, which is responsible for finding the pivot row and sending the instructions to the processors B at the same row. Pivot row means the first input vector with a non-zero diagonal bit (pivot bit) for a processing line. There are three types of instructions: 'pass', 'swap' and 'add', which corresponds to operations with assigning output with input, swapping values of the wires and register in cell, and adding the values of the input and output. The processors B receive the instructions from processors AB and execute the corresponding operations. Part of the finite state machine in processor

Table 1: State switching table in processor AB [WSN16]

Start	In	r_{AB}	state/Ins	next r_{AB}
1	d	-	x	d
0	0	0	pass	0
0	0	1	pass	1
0	1	0	swap	1
0	1	1	add	1

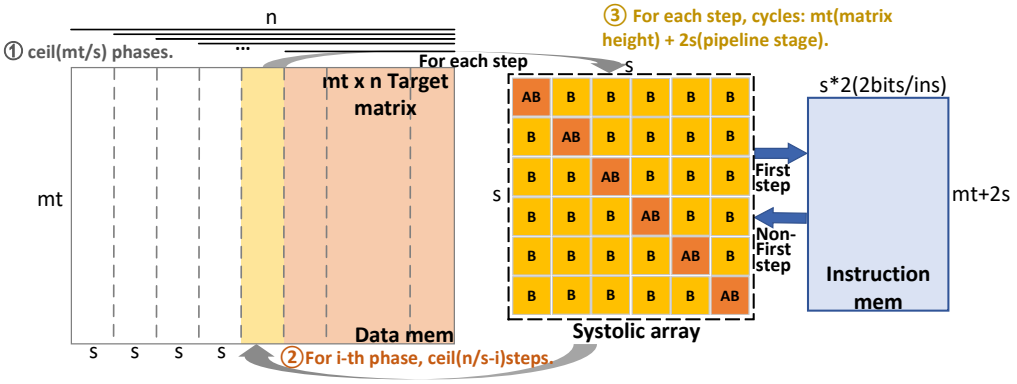

Figure 1: Calculating framework of GF(2) systemizer in [WSN16]

AB is depicted in Table 1. When the 'Start' signal of each step is '1', then the input bit is imported into register cell. When the input bit of processor AB is '0', then 'pass' instruction is distributed. When the register cell in processor AB is '0' and the input bit is '1', then the pivot row is found and 'swap' instruction is distributed. When the register cell and input bit in processor AB are both '1's, then 'add' instruction is distributed. Besides, there are 'start' and 'finish' instructions to initialize the square array and output the register values at the end of each step. During the 'finish' stage, the output of each row still passes through lower rows and elimination effects in these rows are still active.

The calculating framework in [WSN16] is illustrated in Fig. 1. For a $mt \times n$ matrix, coefficients data are buffered in a big data memory, which are divided into n-bit-wide columns. The executions consists of $\lceil mt/s \rceil$ phases, and calculations in each phase are decomposed into multiple steps. And s denotes the size of systolic array. For i-th phase, the i-th column block is reduced to identity matrix form during the first step, while the activities of each processor cell are stored in instruction forms and those are loaded into instruction memory. Then for the following steps in i-th phase, data in $i+1$-th to $\lceil n/s \rceil$-1-th column block are imported into the square array in succession to replay the instructions stored in the first step. Therefore, the Gauss elimination effects are spreaded to the whole row, which guarantees the correctness of results. The calculation flows are summarized into Algo. 2.

For j-th step, each sub-row in j-th column block is imported into the square array per cycle. Finally, the upper triangular matrix is eliminated into the form of identity matrix. In addition to register in each cell, there are register lines between two adjacent processing rows. Thus, $2s$ additional pipeline cycles are needed and there are $mt + 2s$ cycles per step. In sum, for $mt \times n$ input matrix, the total cycles consumed were calculated in [WSN16] as $\sum_{i=0}^{\lceil mt/s-1 \rceil} (mt + 2s)(\lceil n/s - i \rceil)$. The memory consumed was not pointed out directly in [WSN16], but it can be evaluated as the sum of data memory and instruction memory in bits: $mt \times n + 2s \times (mt + 2s)$.

Recently the work [CCD+22] was presented to improve the design [WSN16, WSN17,

Algorithm 2 Calculation flows of GF(2) hardware systemizer in [WSN16]

Input: A: $mt \times n$ input matrix; systolic-array: $s \times s$ square array; ins-mem: (mt+2s) × 2s storage space;
Output: $\hat{A}$: systemize(A);

```
Â = A;
for (phase=0; phase<=⌈mt/s⌉-1; phase++) do
    for (step=phase; step<=⌈n/s⌉-1; step++) do
        j = step;
        if (step == phase) then
            [Â(:, js : js + s − 1),ins-mem] <= systolic-array(Â(:, js : js + s − 1));
        else
            Â(:, js : js + s − 1) <= systolic-array(Â(:, js : js + s − 1),ins-mem);
        end if
    end for
end for
```

//$\hat{A}(:, a : b)$ denotes the column bank of $\hat{A}$ from a-th column to b-th column.

WSN18]. Although certain points, such as early abort and overlapped execution, were proposed to achieve the similar goal, the methods and implementations are substantially different. Furthermore, memory-efficient scheduling strategy is first proposed in this work to enable higher utilization with further reduction on memory consumption. And the work [CCD+22] did not cover the novel memory-efficient scheduling strategy proposed in this paper, which is the most important technique proposed to reduce the memory consumption significantly.

2.3 Existing constant-time hardware sorters

Sorting functions are required in multiple PQC schemes, including NTRU [CCso20] and McEliece [MRAso20], because they help to achieve a random full permutation of a given array in a constant-time way. Therefore, the sorter design is an indispensable part in PQC schemes to introduce the location randomness. Besides high-speed, constant-time property needs to be considered in cryptography hardware to avoid the potential timing attacks.

[WSN18] presented a scalar merge-sorter using only one comparator. However, it was limited by its execution speed. [DVBK21] proposed the FIFO-based parallel merge sorter. In that design, $log_2(l)$ stages are cascaded to sort l-length arrays, where there are two comparators in each stage except the last stage. The inputs of each comparator are two sorted lists and the output is a sorted result list with double input length. And each input list is stored in a separate memory. Because not all storage spaces contain valid data, there is waste of memory. About $3l$ cycles are needed in [DVBK21] to sort a l-element list. The comparators number can be evaluated as $2 \times log_2(l) - 1$. And the memory requirement in elements are evaluated as $4(1 + 2 + 4 + ... + 2^{log_2 l - 2}) + 2(2^{log_2 l - 1}) \approx 3l$. [ZZZ+22] implemented a constant-time design based on two revised Bitonic sorting networks, which was based on feedback mechanism. That design was applicable to ASIC implementation, however it was not applicable on FPGA because of its long critical path.

In traditional sorters design [CO14, SKLG16, MVCK17, SEC+18, PBL18, SQA+20], merge-tree type [SQA+20] design was not constant-time. Among sorting networks design, irregular FIFO and memory accesses design also result in non-constant time executions [SKLG16]. Inspired by the idea in [PBL18], a constant-time sorter is proposed with regular memory accesses and less sorters requirements.

3 Optimized large-size GF(2) systemizer

As mentioned in Section 2.2, the works in [WSN16,WSN17,WSN18] divided the calculations in phases and steps iterations, respectively. Its design has some limitations, including high memory consumption, low processing-cell utilization rate and higher cycles, which are the key issues to address in this work. The advantages of optimized large-scale GF(2) systemizer are listed as followed:

1. Novel architecture design is based on processing lines with auxiliary padding bits, which saves 50% instruction memory for each processing line compared to the previous work.

2. Full-pipeline design is achieved. All processing lines are fully-pipelined once started, which avoids consumed-cycles for pipeline initialization for each step.

3. Dynamic memory-reusage calculating-scheduling framework is proposed, which needs only a quarter to a third memory compared to previous works. The advantage might be more apparent for ASIC implementation due to single-port SRAM utilized.

4. Input and output hiding designs are introduced, which scheduling the data importing, result exporting and elimination processing in parallel to save the cycle count.

5. Singular detection module is proposed to achieve early abortion, reducing unnecessary clock overheads when the targeted matrix is singular.

3.1 Novel processing-line based architecture

Rather than processing cell-targeted finite state machine (FSM) design in [WSN16,WSN17, WSN18], finite machine processing line (Pline)-oriented FSM design is proposed in this work. Besides, auxiliary padding bits are appended to the input vectors for state identification in Plines. The *pad*-bit padding bits are appended before the input vector enters the processing array, and then follow the data flow in the array. The architecture of processing array is shown in Fig. 2. Assuming the input vector is s-bit vector and the processing array consists of s-level Plines. Moreover, the processing array is divided into Q systolic stages and P-level Plines are included in each systolic stage. Between two adjacent systolic stages, a level of pipeline register (Rpipe) is inserted to reduce the critical path in the whole architecture, which is different from previous work [WSN16]. The register vectors like Rline were inserted between any two adjacent systolic lines in [WSN16], rather than systolic stages composed of P Plines. The idea of systolic stage is more general, because the value of P or Q is configurable according to required working frequency.

For each level of Pline, three types of registers are involved: s-bit Rline, *pad*-bit rPads and 1-bit rPivot. Among them, resulted data and auxiliary padding bits are stored in Rline and rPads, respectively. And 1-bit flag bit, which indicates whether the pivot row is stored in this Pline, is stored in rPivot.

As mentioned in Table 1, three types of states and instructions are utilized: 'pass', 'add' and 'swap'. Therefore, 2-bit instruction for each line is needed to encode these three states. The state designs are followed in this work. However, only 1-bit instruction is needed in this work with the help of rPivot. The original switching table in Plines is depicted in Table 2. The start case in each step is encoded through padding bits, which is illustrated further in Section 3.2. The bit in rPivot simulates the r_{AB} in [WSN16] during first step and non-first step in each phase. The switching pattern during first step is the same as [WSN16] except the instructions generated. The instruction only indicates the pivot bit of input vector, which provides the input information during fist step for non-first step calculations. For non-first step, the corresponding states in first step can be replayed through instructions read and rPivot, where rPivot acts as the same as in first step. At

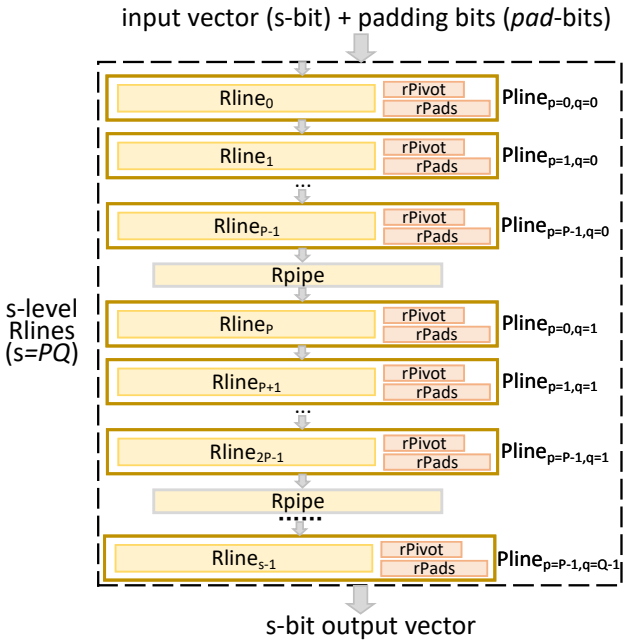

Figure 2: Proposed processing array architecture for GF(2) matrix systemization

Table 2: Proposed switching table in Pline

FirstStep						Non-Firststep				
Start Case	In	rPivot	InsWr	State	next rPivot	Start Case	InsRd	rPivot	State	next rPivot
Yes	d	-	d	swap	d	Yes	d	-	swap	d
No	0	0	0	pass	0	No	0	0	pass	0
No	0	1	0	pass	1	No	0	1	pass	1
No	1	0	1	swap	1	No	1	0	swap	1
No	1	1	1	add	1	No	1	1	add	1

the cost of s 1-bit flag registers, the instruction size is reduced to half. And because more instruction memory are needed in novel calculation scheduling framework proposed in Section 3.3, the reduction effects are augmented further.

3.2 Complete FSM table and full-pipeline design

Table 2 is the core of the finite-machine adopted in this work, the complete finite-machine design needs to cover other corner cases: including start and finish cases, phase or step switching cases, low triangular data rows cases. The padding bits, assisting Plines in covering all states switching involved, are generated after the data vector is read from memory and flow in the processing array with the data vectors. And the critical path of control signals are mitigated by this approach, because the length of control path is the same as the data path.

'firststep' padding bit is needed in Plines to tell whether the first step of one phase or not as stated in Table 2. 'datavalid' padding bit, indicating the data vector is valid, is utilized to start up all the calculations through switching to 'swap' state when 'd_datavalid'='1' and 'r_datavalid'='0', where 'd_' denotes the padding bits from input and 'r_' denotes the corresponding bit in register rPads of Plines. 'finish' padding bit helps to swap the valid data out of the Plines and it is generated after all the data vectors are read. Additional

zeros-vector in $s+Q$ cycles are needed after all the data vectors are read because 'finish' bit is needed to spread all the Plines.

In [WSN16], individual $2s$ cycles are needed to restart the pipeline in each step, which results in a relatively large cycle overheads, especially when s/mt is not negligible. For achieving full-pipeline design, seamless switching is implemented between different steps or phases. In other words, the final output of the previous step/phase and the input of the next step/phase are performed simultaneously. Two padding bits, 'phasecheck' and 'stepcheck', are utilized, where 'phasecheck'=phaseid%2, and 'stepcheck'=stepid%2. And Plines calculate 'psaccord'=('d_phasecheck'=='r_phasecheck')&('d_stepcheck'=='r_stepcheck'). When 'psaccord'='0', it denotes that input data vector is from the next step/phase and state is jumped to 'swap'. Therefore, the old data vector from the last step/phase outputs from Plines and the new vector from the next step/phase is stored into Plines. Therefore, the parallel pipeline is obtained. The whole GF(2) matrix elimination cycles are reduced to $s+Q+\sum_{i=0}^{\lceil mt/s-1\rceil} mt(\lceil n/s-i\rceil)$ benefited from full-pipeline design.

When mt is not divided by s, the data row number from lower triangular matrix is less than the number of Plines. Therefore, data vectors from upper triangular part may be stored in Plines during elimination in the last phase. And one flag bit is needed to denote this case to allow the input vectors to directly 'pass' the Plines. 'lowtriangular' padding bit is set to '1' when the input data vector is from the low triangular matrix.

In sum, the complete state transition formulas are:

$$\begin{aligned} swap\_state =& !psaccord \;||\; (d\_firststep \;\&\; d\_pivot \;\&\; !r\_pivot \;\&\; r\_lowtriangle) \;||... \\ & (!d\_firststep \;\&\; InsRd \;\&\; !r\_pivot \;\&\; !d\_finish) \;||... \\ & (d\_finish \;\&\; r\_valid) \;||\; (d\_datavalid \;\&\; !r\_valid); \end{aligned} \tag{1}$$

$$\begin{aligned} pass\_state =& (((r\_valid \;\&\; !r\_lowtriangle) \;||\; (!d\_pivot \;\&\; d\_firststep) \;||... \\ & (!d\_firststep \;\&\; !InsRd)) \;\&\; psaccord \;\&\; d\_datavalid \;\&\; r\_valid) \;||.... \\ & (d\_finish \;\&\; !r\_valid); \end{aligned} \tag{2}$$

$$\begin{aligned} add\_state =& ((r\_pivot \;\&\; d\_pivot \;\&\; d\_firststep) \;||\; (r\_pivot \;\&\; InsRd \;\&\; !d\_firststep))... \\ & \&\; r\_lowtriangle \;\&\; psaccord \;\&\; d\_datavalid; \end{aligned} \tag{3}$$

$$idle\_state = !d\_datavalid \;\&\; !d\_finish; \tag{4}$$

3.3 Dynamic memory-reusage calculating-scheduling framework

Besides processing array, an original dynamic memory-reusage calculating-scheduling framework is also proposed. The improvement is based on the observation that: the data memory can be re-utilized and the size requirement of $mt \times n$ can be reduced. A novel calculating framework is proposed based on reusage of data memory.

In the nested loops adopted in [WSN16], the outer loop is phase-loop and the inner loop is step-loop. The definitions of phase and step are illustrated earlier. Among different steps in the same phase, the instruction memory is reused but the column blocks of data memory are traversed to the last block. Therefore, it is hard to reuse the data memory in this way because data memories are traversed to the last block in each phase.

The order in nested loop is reversed in this work: the outer loop is step-loop and the inner loop is phase-loop. To avoid the confusion of narration, the "step" in this work is called as "bigstep". The proposed calculating-scheduling framework is shown in Fig. 3. For a certain column block of matrix, elimination processes are first repeated in "non first step" way through the instruction memories recorded earlier, then elimination processes on this column block continue in "first step" way and another instruction memory is recorded. The calculation flow is illustrated in Algo. 3. In this method, only two mt-depth s-width

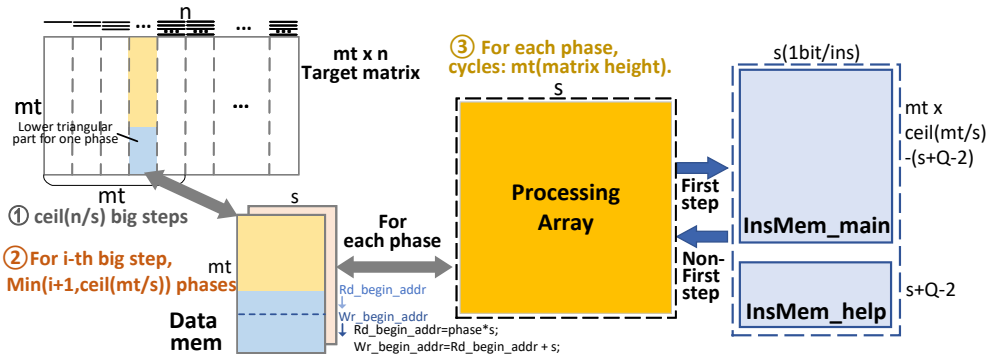

Figure 3: Proposed Calculating-scheduling framework

Algorithm 3 Novel calculation flows of GF(2) hardware systemizer proposed

Input: A: $mt \times n$ input matrix; systolic-array: $s \times s$ square array; $insmem_j$: $mt \times s$ storage space, j=0,1,...$\lceil mt/s \rceil$-1.
Output: $\hat{A}$: systemize(A);

```
Â = A;
for (bigstep=0; bigstep<=⌈n/s⌉-1; bigstep++) do
    j = bigstep;
    Acolblock = Â(:, js : js + s − 1);
    for (phase=0; phase<=min(⌈mt/s⌉-1,bigstep); phase++) do
        step = bigstep - phase;
        if (step == 0) then
            [Acolblock,insmem_j] <= systolic-array(Acolblock);
        else
            Acolblock <= systolic-array(Acolblock,insmem_phase);
        end if
    end for
    Â(:, js : js + s − 1) = Acolblock;
end for
```

data memories storing the column block of matrix in ping-pong way are needed. Two relatively big instruction memories, InsMem_help and InsMem_main, stores instructions from $phase_0$ to $phase_{\lceil mt/s \rceil - 1}$.

For accesses of data memory in j-th phase, the begin address of reading is $j \times s$. First s data rows are stored in Plines, therefore the begin address of writing is $j \times s + s$. The write address is maintained by a dedicated counter, which is activated by the valid signal of output. The counter returns to the first row when the address exceeds mt in each phase. Data are not polluted within and between the phases, because the data at the address written are all read out earlier, and the data required to read have been in the memory. This satisfies the full-pipeline design.

Fig. 4 depicts the scheduling flow in the proposed framework and Read/Write cases for instruction memories. A column block of target matrix corresponds to an individual bigstep, as mentioned in Algo. 3. The arrow in the left half part of Fig. 4 illustrates the execution order of phases and bigsteps. When all phases involved with a data column block are executed done, the next bigstep just begins. Because an individual block of instruction memory is utilized in each phase, the execution in one bigstep involve multiple blocks of instruction memories. The phase/bigstep switching problem of instruction memory needs to considered, especially when full-pipeline technique is adopted. InsMEm_main

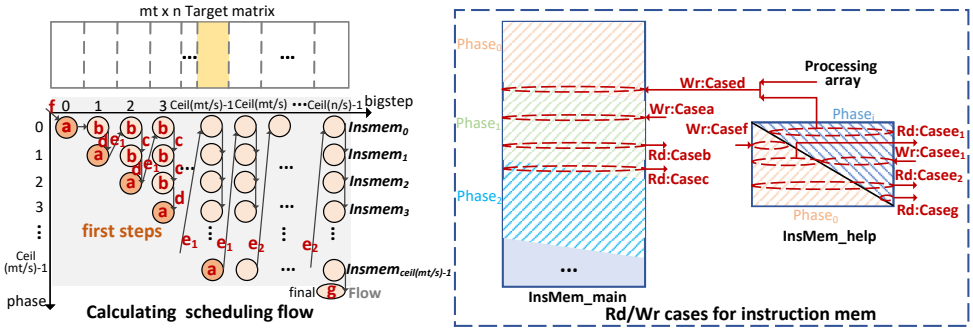

Figure 4: Calculating-scheduling flow and Rd/Wr cases for instruction memories

stores main part of instructions and InsMem_help stores instructions at the switching stage between two adjacent phases/bigsteps. Different cases of phases or phase/bigstep switching are tagged in Fig. 4. Based on Section 3.2, switching stage consists of the first $s+Q-2$ cycles of each phase. Case a and case b denote the first step process and non-first step process without switching, while the instruction vector is written into or read from the InsMem_main. An instruction vector is formed from the instructions from all the Plines in the same cycle, which is stored into one row of InsMem. Case f denotes the start process of the $phase_0$ and Plines pipeline are started, while the instruction are stored in InsMem_help. Due to that only the first few Plines are activated during this start process and the number of activated rows increases gradually, only the lower left corner of InsMem_help is filled. Case e_1 denotes the switching stage from first step to non-first step, during which instructions from the last few Plines belonging to the first step are written into the upper right corner. Meanwhile, instructions at the same row from $phase_0$ are read for the upper Plines belonging to the non-first step. The upper right corner of InsMem_help is read out in case d to lower Plines, which denotes the switching stage from non-first step to first step, and written to InsMem_main merged with the newly-generated instructions from the upper Plines. Case c or case e_2 indicate the switching stage between two non-first steps across bigsteps or not, while the instructions are from InsMem_main or InsMem_help, respectively. Case g indicate the final process, when instructions are from the upper right corner of Insmem_help.

Therefore, InsMem_main has a depth of $mt \times \lceil n/s \rceil - (s+Q-2)$. Benefited from the size reduction of instruction described in Section 3.1, only 1 bit is needed for each instruction. The total memory consumption in bits is assessed as $2 \times mt \times s + s \times mt \times \lceil mt/s \rceil$, which is independent of the width of matrix, s. In Classical McEliece, the width of matrix, n, is multiple times of the the height, mt. Thus, compared to the previous work [WSN16], most memory is reduced. The theoretical comparison results among different parameters are shown in Table 3, where $s = 160$. At least 68% of the memory in bits are saved adopting the proposed scheduling framework. To make the things better, InsMem_main, occupying the biggest portion of storage system, requires only single port for read or write each cycle. This single-port design will amplify the benefits for implementation on ASIC. By comparison, the data RAM in [WSN16], occupying the biggest portion of storage system, needs one-read one-write ports, which consumes more area for the same amount of memory.

3.4 Input and output hiding design

Considering the large size of targeted matrix, the importing time of matrix and exporting time of results can not be ignored. the exporting time is limited by the number of output ports. If 8-Gbps output bandwidth is assumed, nearly 50% cycle are consumed to execute importing and exporting compared to one successful elimination execution. Benefited from

Table 3: Required memory size (bit) for different parameter sets. mt/n: the height/width of matrix.

	mt=768; n=3488	mt=1248; n=4608	mt=1664; n=6688	mt=1547; n=6960	mt=1664; n=8192
[WSN16]	3.03×10^6	6.25×10^6	11.66×10^6	11.76×10^6	14.26×10^6
Proposed	0.86×10^6	2.00×10^6	3.46×10^6	2.97×10^6	3.46×10^6
Reduced Ratio	71.6%	68.1%	70.6%	73.9%	75.7%

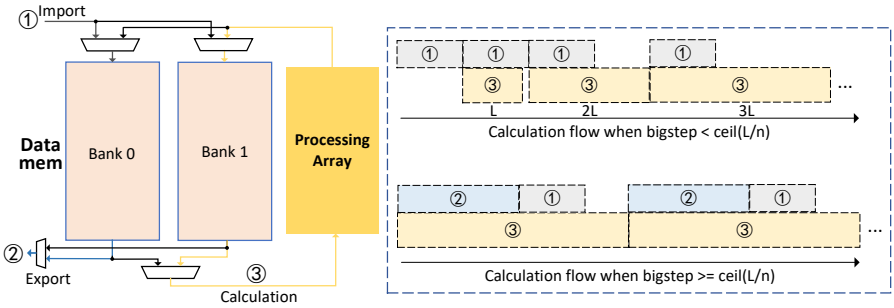

Figure 5: Ping-pong data memory design and hiding calculation flow

the better blocking properties of scheduling framework proposed, it is better and easier to hide the importing and exporting than [WSN16]. As the calculations on one column block of matrix are continuous, the data exporting task can start after these calculations. While one column block of matrix is calculated in iterations, the data exporting task of the last column block if necessary and data importing task of the next column block are executed in the another bank of data memory. It can directly stream out the partial public key as [RKK21] or re-organized in the external memory as [CC21].

The hiding diagram is illustrated in Fig. 5. There are two hiding cases: $bigstep < \lceil L/n \rceil$ and $bigstep >= \lceil L/n \rceil$, because the identity matrix is not included in the final results. In most bigsteps, the calculation latency is able to hide the data importing and data exporting. This hiding design was not considered in [CCD+22], because its scheduling framework follows the design [WSN16]. The cycle overheads of exporting data are not explicitly provided in the final results listed [WSN16, WSN17, WSN18, CCD+22]. For a fair comparison, all the results are involved in this article without taking into account the data exporting if there is no special statement.

3.5 Singularity detection-based early abort

In the specification of McEliece [MRAso20], the key generation should be terminated if the targeted GF(2) matrix is singular. Hence, unnecessary operations could be avoided, and re-execution of key generation could also be started immediately once the matrix is determined as singular.

When the matrix is singular, there are not enough pivot rows in some first step. In the proposed full-pipeline framework, there are too many data rows that are calculated to all-zeros and 'pass' through all the Plines. Therefore, there exists Pline that is not able to find its pivot row in this step while its register "r_pivot" still equals to zero. A padding bit, named 'detect_sing', flows in the Plines as the other padding bits mentioned in Section 3.2 to detect this situation: the data in Rline belongs to the first step and the pivot row has not been found, but the input vector does not belong to the lower triangle part in this first step or the input flag "d_detect_sing" is active. The output bit 'detect_sing' of one

level of Plines is calculated as:

$$\begin{aligned} dout\_detect\_sing =& !r\_pivot \,\&\, r\_firststep \,\&\, r\_lowtriangle \\ & (!d\_lowtriangle \,||\, !psaccord) \,||\, d\_detect\_sing; \end{aligned} \tag{5}$$

And this flag bit spreads through the Plines and output to the system controller, which manipulates the restart of key generation.

Besides, the memory-efficient calculating-scheduling framework proposed is beneficial to the singular detection. The latest detection of singular matrix occurs at $\lceil mt/s \rceil - 1$ bigstep, which is far less than the whole elimination task. In contrast, the corresponding detection happens at the last phase in [WSN16, WSN17], which is near the end of the whole GF(2) elimination.

The advantages relative to [CCD+22] comes from the combination of detecting singularity and elimination process. The work [CCD+22] proposed three working mechanisms to schedule the detection of singularity and the final elimination: HEA, SPEA and DPEA. These three mechanisms all have their limitations. The singularity is first checked through eliminating the leftmost $mt \times mt$ matrix in HEA mode individually, and then the eliminations on the whole matrix are executed when this matrix is invertible. But the detection cycles are wasted because the same tasks are repeated in subsequent elimination. The instructions during checking are stored in SPEA and DPEA modes to replay on the remaining part, which avoids the repeated executions. But it incurs more instruction memories and additional control cycles. And the results show that the elimination cycles of SPEA or DPEA are even higher than HEA when the size of systolic array is large (s $\geq$ 128).

3.6 Performance comparison

Based on the trade-off between area and performance, the size of systolic array n is selected as 160 or 128 for different parameters. Table 4 shows the implementation results of GF(2) systemizers. For cycles data, data importing is included in the results of our work and not in other works.

Compared to the latest work [CCD+22], only 31.8% memory space is needed. When $mt = 768, n = 3488$, there is only one reversible matrix after an average of 3.46 attempts. For [CCD+22], about 169.9k cycles are needed to eliminate a inversible matrix, while 161.7k cycles are consumed in this work. If data exporting is additional included and 32-bit output width is used, at least half of cycles (86k) are extra needed. But benefited from hiding design in Section 3.4, only additional 1.2k cycles are needed in this work(s=128) because of the balanced design. Therefore, the benefits are amplified if data exporting is included, which is not shown in Table 4. And compared to work [WSN18], only a quarter memory space is needed in our design after device conversion.

4 Optimized constant-time sorter

The proposed optimized constant-time sorter is based on merge sorting algorithm, which is the same as prior works. Different from the serial implementation in [WSN18], four-element parallel implementation is utilized to accelerate the sorting process. Compared to the FIFO-based parallel sorter [DVBK21], only small-size FIFOs are utilized in this proposed design with full utilization of comparators. Considering that hundreds of or thousands of need to be sorted in post-quantum cryptography to introduce the location randomness, the proposed sorter is more suitable than the large-scale sorting network in [PBL18]. The advantages of the proposed sorting hardware are listed as follows:

1. Full utilization of comparators. All comparators are active once started, including the iteration switching stage.

Table 4: Comparisons with the state-of-the-art GF(2) systemizers. mt=t×m, n: the width of matrix T. (Implementations are based on Xilinx Ultrascale+ FPGA unless specified)

		Cycles in failed try	Cycles in successful try	BRAM	LUT	Flip-Flops	Freq (MHz)
mt=768 n=3488	[WSN18] [a] (s=128)	157.8k	157.8k	149.5	26393	35267	175
	[CCD+22] [a] (s=128)	14.51k	135.1k	132	19538	35606	181
	This Work [a] (s=128,Q=128)	**17.1k**	**119.8k**	**42**	**21494**	**35694**	**185**
	This Work (s=128,Q=64)	**17.0k**	**119.7k**	**28.5**	**22039**	**26931**	**284**
	This work (s=160,Q=80)	**12.6k**	**79.0k**	**29.5**	**32505**	**41279**	**278**
mt=1248 n=4608	**This Work (s=160,Q=80)**	**46.4k**	**258.1k**	**67**	**32580**	**41448**	**262**
mt=1664 n=6688	**This Work (s=160,Q=80)**	**111.9k**	**682.3k**	**103.5**	**32900**	**41445**	**259**
mt=1547 n=6960	[WSN18] [b] (s=160)	737.86k	737.86k	720(400)	40530	55675	290
	This Work (s=160,Q=80)	**87.0k**	**615.9k**	**100.5**	**32630**	**41314**	**264**
mt=1664 n=8192	**This Work (s=160,Q=80)**	**111.9k**	**865.8k**	**103.5**	**32892**	**41443**	**259**

[a] Implementations are based on Xilinx Artix 7 FPGA.
[b] The implementation results are based on Altera Straix V FPGA. The memory consumed is the same as that of 400 BRAMs.

2. Higher performance with less comparators is achieved compared with the previous work due to a novel sorting mechanism.

3. Only small-size FIFO-based couplers are utilized to balance the memory accesses and sorting consumption, while constant-time property is also guaranteed.

4. Feedback-less mechanism is followed to achieve higher operating frequency.

The merge sorting process of 2^m elements is divided into m-iterations. In i-th iteration, each group of 2^i elements is taken as a sorted sub-array. The sorting processes between each two sorted groups are executed in i-th iteration to obtain a longer sorted array of 2^{i+1} elements. Finally, 2^m-length sorted array is obtained after m-iterations.

For the proposed parallel sorter, four elements, which are defined as a vector in this work, are read from and written to the data memory in parallel. The structure of proposed sorter is shown as Fig. 6. The sorter mainly consists of three parts: ping-pong data memory, FIFO-based couplers and odd-even based sorting network. FIFO-A or FIFO-B, which belongs to coupler, are splitted into four individual FIFOs and each is one-element wide. Four FIFOs in FIFO-A or FIFO-B manage the pop operations and reading addresses individually. However, these four FIFOs share a common writing address because the data vector from sort memory is written into the same address of FIFOs. Besides, registers counting the vector index, groups or iterations and judging logic hardware for enqueue or dequeue signals of FIFOs are also included in the design.

Random elements are distributed in the input bank of data memory before sorting. The executions begin at the first pass, while the vectors are continuously read from the input bank, pass through the sorting network and written into the output bank. Because sorting network is an odd-even based network to sort any four elements in parallel. Therefore, the output results at the first pass is 4-length sorted vectors. In the following $m-2$ passes, the input bank is divided into groups A and groups B as Fig. 6 shows. During one pass, one group from groups A and another group from groups B are sorted into longer arrays. At the beginning, vectors from groups A or groups B are firstly pre-filled into

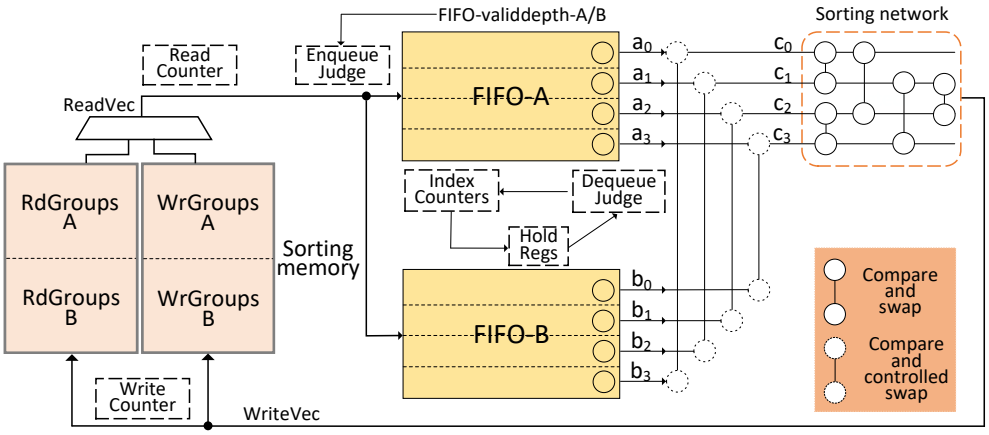

Figure 6: 4-parallel sorter architecture

FIFO-A and FIFO-B through enqueue judging logic. When the actual depth of FIFO-A and FIFO-B exceeds the pre-defined threshold, the sorting network begins to run. During the comparison between two groups, four comparators next to FIFOs compare the top eight elements of FIFO-A and FIFO-B and the connections are as Fig. 6 shows. The comparison results are passed to the dequeue judging logic to determine which four elements should be popped from the FIFOs. And four smallest elements among the remaining elements of these two groups are selected for sorting network. Finally, the sorted groups are written into the output bank. After each iteration, the input bank and output bank are swapped.

The reading addresses are generated by the read counter. The counter manages the addresses of groups A and groups B, which increases by 1 correspondingly while reading. Write counters manage the write indexes of vector, group and iteration individually, which generates the write addresses in groups A and groups B alternately of output bank. The groups to be read in the next iteration have been written in advance, therefore the full-utilization is able to achieve during the iteration switching stage.

The correctness of sorting two sorted groups is guaranteed by the connections of 4 comparators as similar in [PBL18]. Four smaller elements are selected to pop after the comparison of these eight elements from the two groups. Besides, it is also guaranteed that the maximum difference of valid depth among four FIFOs in FIFO-A or FIFO-B is 1. The proofs are shown as follows. Assuming in each cycle four top elements a_0, a_1, a_2, a_3 of FIFO-A and b_0, b_1, b_2, b_3 of FIFO-B satisfy the relation as follows, where i,j are integers ranging from [0,3].

$$\begin{aligned} &a_i \le a_{i+1 \bmod 4} \le a_{i+2 \bmod 4} \le a_{i+3 \bmod 4}; \\ &b_j \le b_{j+1 \bmod 4} \le b_{j+2 \bmod 4} \le b_{j+3 \bmod 4}; \\ &i + j = 0 \bmod 4; \end{aligned} \tag{6}$$

It is naturally established for the first cycle of each group comparisons, because $i, j = 0$. And it can be proved that the output vector c_i consists of four smallest elements among the remaining vectors and Eq. (6) still satisfies in the next cycle. The function $a_{i+k \bmod 4} - b_{j+3-k \bmod 4}$ is constructed to denote connection of comparators, where k ranges from [0,3]. When the value of function is not bigger than zero, then $a_{i+k \bmod 4}$ is popped. It is easy to see that this function is increasing monotonically as k increases. If k_1 elements from FIFO-A and k_2 elements from FIFO-B are popped in this cycle, then a_i, $a_{i+1 \bmod 4}$, ... $a_{i+k_1-1 \bmod 4}$ and b_j, $b_{j+1 \bmod 4}$, ... $b_{j+k_2-1 \bmod 4}$ in continuous locations are popped. And these top pointers are shifted to the next positions. The remaining $a_{i+k_1 \bmod 4}$, $a_{i+1+k_1 \bmod 4}$, ... $a_{i+3+k_1 \bmod 4}$ on the top in the next cycle are also sorted. They are all bigger than the output a_i because of the input sorted list.And the smallest element

Table 5: Comparison of the hardware sorter's resource usage (sorting 8192 elements; random prefixes:32-bit; satellite: 13-bit)

	LUTs	Flip-Flops	BRAMS	Cycles	Frequency
[WSN18]	583	411	20	147505	250M
[DVBK21]	2533	1589	33	26646	250M
Proposed	2510	3887	20	24598	300M

remained in FIFO-A $a_{i+k_1 \bmod 4}$ of next cycle is bigger than $b_{3-(i+k_1) \bmod 4}$=$b_{j+k_2-1 \bmod 4}$ because for the comparison results. It is the same for b. Thus, the four output elements are all smaller than $a_{i+k_1 \bmod 4}$ and $a_{j+k_2 \bmod 4}$. Then the output vector c_i in this cycle are the smallest 4 elements.

There are hold registers and dequeue logic to ensure that elements of next group will not be popped in advance. When elements belonging to this group in some FIFO are emptied, then holding registers are active to prevent continuing to pop. Other sets of counters, indexes of vector, group and iteration, are maintained for holding register. When holding signal of FIFO-B is not active and values in FIFO-A will be emptied in this cycle, then hold register of FIFO-A is triggered. When holding register of FIFO-B is already active and values in FIFO-A will be emptied in this cycle, or FIFO-A/FIFO-B will be both emptied in this cycle, then holding register of FIFO-A is released to continue the comparison in the next group. It is the same for FIFO-B. Consequently, no stall happens during the comparison in two groups to enable full utilization.

The actual depth of FIFO-A and FIFO-B is maintained through the number of valid elements in FIFO-A_3 and FIFO-B_3, which indicates that a column of data are emptied because the elements popped are continuously. Enqueue logic hardware detects whether the valid depth is lower than the threshold and determines read vectors from groups A or groups B. However, the response path, from popping the elements in FIFO-A_3 or FIFO-B_3 to supplementing vectors by enqueuing, includes dequeuing, valid depth updating, enqueue judging, reading and final enqueueing. 4 cycles are needed in this work, therefore FIFO's threshold is 5 with its depth as 10. Before the execution of sorting network, there are already 5 elements in each FIFO-A_i/FIFO-B_i. During executions, it is guaranteed that there is no overflow or excessive consumption in data FIFOs for constant-time sorting.

Only 9 comparators are utilized in this design. The width of element in comparison is 32-bit, besides 12/13-bit constants (satellite) are moved along with the random prefixes. When the comparators detect that two random prefixes are equal, an output flag is activated and passed to the system controller for restarting. The cycle overhead to sort 2^m elements is $(m-2) \times 2^m/4$. And the resource consumption is listed in Table 5. The additional storage space is reduced to 80 coefficients (the total size in FIFOs) from several thousand, which must be stored in BRAMs. In contrast, only flip-flops are used to afford the storage of these elements.

It is noted that the proposed sorter is also applicable to the hardware of other PQC algorithms, such as the hardware implementation of NTRU [DVBK21], as long as that sorting is used to introduce location randomness.

5 System architecture and algorithm-level pipeline design

5.1 Overall architecture

The overall architecture is shown in Fig. 7. First, random prefixes and random polynomial generated by Keccak module are imported to constant-time sorter and irreducible polynomial generation module, respectively. Then the evaluation process of irreducible polynomial on 2^m points is executed in additive FFT. And the generation of H module collects the data from sorter module and additive FFT module for further processing in GF(2) Gauss

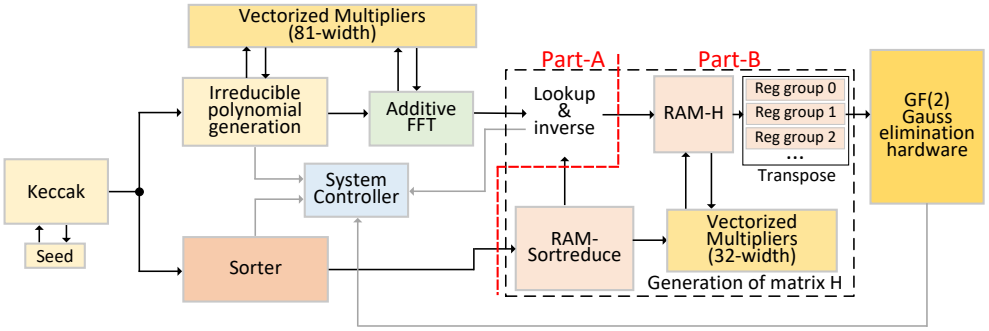

Figure 7: The overall architecture

elimination module. Data from sorting memory are first loaded in memory $Ram_{Sortreduce}$, where only satellite bits need to be stored and prefix bits do not need. Then elements are read from $Ram_{Sortreduce}$ as addresses to look up the memory in additive FFT module. After reading, inversion operation and writing to Ram_H are executed subsequently. The depths of Ram_H and $Ram_{Sortreduce}$ are $(n/32)$ and $2^m/32$ respectively, while the data width of both is 32bit.

During the look-up processes, an additional optimization technique is adopted to achieve earlier abort. If the elements indexed in first mt elements are detected as zeros, then the final matrix can be judged as singular. Thus, the earlier abort can be achieved by just checking the results of the first mt look-ups. The probability of this case is around 17% based on experimental analysis. At average 0.6 attempts for successful key generation are reduced.

During the interaction with GF(2) systemizer, one s-bit width column block is written per execution. The $s/32$ 32-coefficients vectors in Ram_H are read to register groups, and then are transposed to s-bit vectors through shifting. Meanwhile, the vectors that read from Ram_H multiply with the vectors from $Ram_{Sortreduce}$ in vectorized multipliers, and then the results are written back to Ram_H for next iterations. When the input bank of GF(2) Gauss elimination module is empty again, then the filling logic is re-activated to fill the next data column block.

5.2 Other modules design

Besides GF(2) Gauss systemizer and constant-time sorter, some other modules involved are also optimized, including irreducible polynomial generation module and additive FFT module.

The principle of irreducible polynomial generation is based on calculating the minimal polynomial of the random polynomial as specified in [MRAso20]. The calculation of minimal polynomial is divided into two steps: first, continuous polynomial multiplications with modular polynomial $f(x)$ are executed to generate the $t \times (t+1)$ $GF(2^m)$ matrix. Second, $t \times (t+1)$ $GF(2^m)$ matrix is eliminated to solve the linear equations. The reductions on modular polynomial $f(x)$ are needed to realize by a series of shift and XOR operations. Besides, the matrix transpose operation is executed to change the form in column-first order, which is applicable to linear equations solving on $GF(2^m)$.

The diagram of irreducible polynomial generation hardware is depicted in Fig. 8. The input polynomial is imported into ram_A, ram_B and $ram_{results}$ in a coefficient-wise manner from the PRNG(Keccak). Two input polynomials for polynomial multiplication are stored in ram_A and ram_B. Then polynomial multiplication is executed and the results are loaded to $ram_{results}$. The corresponding polynomial multiplication hardware is based on 6-level Karatsuba recursive design, which followed the design [ZZY+21]. The transform mechanism,

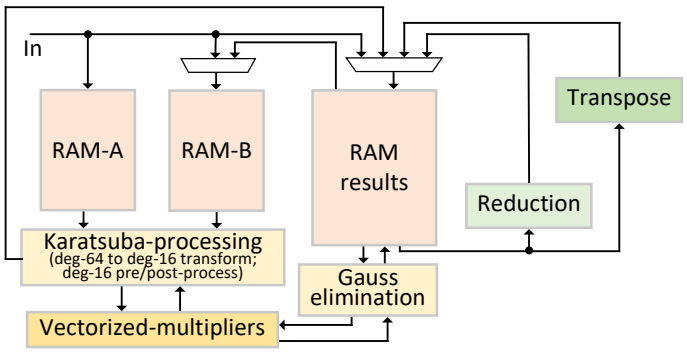

Figure 8: Diagram of random irreducible polynomial generation

including circuits and finite-state machine, from degree-64 to degree-16 sub-polynomials are similar in this work, which adopts 2-level Karatsuba algorithm. And degree-16 sub-polynomial multiplication is executed in parallel each cycle, which adopts four-level Karatsuba algorithm. Through pre-/post-processing of four-level Karatsuba algorithm, degree-16 polynomial multiplication is reduced to 81 coefficient multiplications, which are executed in the vectorized multipliers. Furthermore, plenty of adders in [ZZY+21] are replaced by XOR gates, because all coefficients in McEliece belong to GF(2^m). As a result, the critical path has been greatly shortened. The degree-t polynomial multiplications are disassembled into $\lceil t/64 \rceil^2$ degree-64 polynomial multiplications. The results and immediate values are stored into $ram_{results}$.

After each polynomial multiplication, the modular or reduction operations are executed. The reduction is done in a customized module, which supports continuous shift and XOR operations required by the modular polynomial in the specification. Then, polynomial multiplication and reductions in the next iteration are started. Matrix transpose module inverts the row/column order of matrix after t polynomial multiplications.

Gauss elimination hardware is based on vectorized operations, which is different from [WSN17]. The principle is to simplify the elimination task to vector operations by reusing the vectorized multipliers. The matrix is divided into multiple 32-element width column blocks. For each column block, all the rows are traversed 32 times to eliminate 32 element-column sequentially. For a certain traversal, the pivot row is selected through finding the first row with the corresponding element non-zero. And it is stored in the additional registers. Then the scaling and XOR operations are executed between the pivot row in registers and other rows from memory to eliminate. The scaling factors are from the inversion of pivot element, element in pivot location of input row or scaling factors stored during elimination on previous column blocks. When a column block is eliminated, then the following column blocks repeat the processes through reading the scaling factors stored and the row index of pivot row marked. Therefore, the elimination effects spread to the whole rows. The elimination task is done when all column blocks belonging to the left square matrix are eliminated and the effects spreads to the whole rows of target matrix.

Additive FFT module follows the design in [WSN18], which is depicted in Fig. 9. Differently, the twisting operations in Taylor expansion and additive-FFT butterflies share the vectorized multipliers in Fig. 8.

5.3 Algorithm-level pipeline design

The utilization of Ram_H and $Ram_{Sortreduce}$ divides the whole key generation into two parts: Part-A and Part-B, which are scheduled independently by the system controller. When GF(2) elimination task in Part-B is calculated, tasks in Part-A are able to restart to

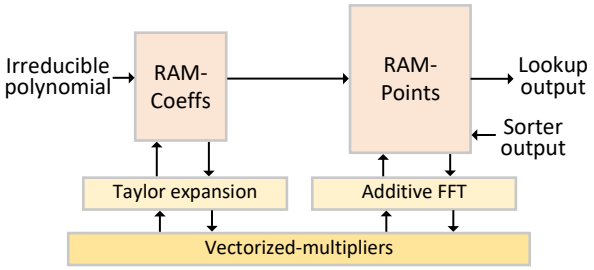

Figure 9: Hardware of additive FFT

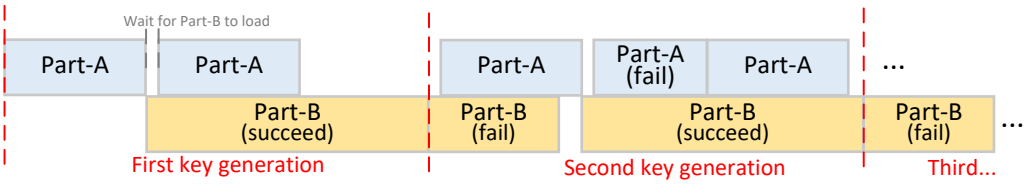

Figure 10: Algorithm-level pipeline design (Batching mode)

execute in parallel with Part-B to save the cycle overheads. There are two cases in general: first if the matrix is detected singular later, starts of Part-A in advance can reduce the cycles for retrying. Second, if the matrix is detected invertible later, starts in advance can reduce Part-A cycles for next key generation.

And there are two ways. The first way (Batching mode) is to set the total number (N) of key generation and supply every seed from outside. This is also benefited from parallel execution between two adjacent key generation, which maximizes the effect of hiding. The disadvantage is that more seeds are needed. This method is applicable to batching application, such as on the server. The second way (Single mode) is to execute only one key generation, and subsequent seeds after failed attempts in this key generation come from Keccak. Within each key generation, parallel execution can still reduce the delay because Part-A after failed attempts is hidden. This method is applicable to single application, such as in edge side. The diagram of parallel execution in batching way is illustrated in Fig. 10. For each successful key generation, the latency of Part-B is much longer than that of Part-A. Therefore, the latency of Part-A is able to be effectively hidden to improve the overall throughput.

The system controller manages the start signals of Part-A and Part-B. There are two sets of flag registers, including ready flag and request flag. A_ready flag denotes that the Part-A module is already idle, which might have finished or failed the previous calculation. A_request flag denotes that the results of Part-A are not valid for Part-B, due to failure in Part-A or previous results have been loaded into Part-B. Similarly, A_out_ready flag denotes that the results of Part-A is valid for Part-B to load. A_out_request flag indicates that the calculations of Part-B have done, including a successful attempt or a failed attempt. When ready flags and out flags are both valid, start signals are then activated.

$$\begin{aligned} A\_start &= A\_ready \& A\_request \\ B\_start &= A\_out\_ready \& A\_out\_request \end{aligned} \tag{7}$$

Whether the calculation in Part-A fails or not is decided by elim_singu bit from irreducible polynomial generation module and sort_equal bit from constant-time sorter. The success of calculation in Part-B is decided by the singularity of matrix. Singular bit comes from the check_zero bit in H generation module and detect_sing bit in GF(2) Gauss module. If GF(2) Gauss module ends successfully, then the number of successful key generation is increased once. When the number reaches N, then state returns to the

initial idle state.

6 Conclusion

This paper proposes a high-performance key generation accelerator for Classic McEliece, which is specifically designed for high-throughput concurrent key generation on high-end platforms, such as data centers or servers. Based on the performance profile, GF(2) Gauss systematization and constant-time sorter, which are the most expensive functions are optimized primarily based on algorithm-hardware co-design. Meanwhile, algorithm-specific pipeline coupled with dedicated implementations of other auxiliary function blocks is also presented to achieve further improvement on the throughput with higher resource utilization. We hope that this work could improve Classic McEliece's competitiveness in high-end application scenarios, and provide solid reference for further performance evaluation in 4-th round of PQC standardization.

References

[BCS13] Daniel J. Bernstein, Tung Chou, and Peter Schwabe. Mcbits: Fast constant-time code-based cryptography. In Guido Bertoni and Jean-Sébastien Coron, editors, *Cryptographic Hardware and Embedded Systems - CHES 2013*, pages 250–272, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[CC21] Ming-Shing Chen and Tung Chou. Classic mceliece on the arm cortex-m4. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2021(3):125–148, Jul. 2021.

[CCD+22] Po-Jen Chen, Tung Chou, Sanjay Deshpande, Norman Lahr, Ruben Niederhagen, Jakub Szefer, and Wen Wang. Complete and improved fpga implementation of classic mceliece. In *International Workshop on Cryptographic Hardware and Embedded Systems*. Springer, 2022.

[CCso20] Jeffrey Hoffstein Cong Chen, Oussama Danba and so on. Ntru: Algorithm specifications and supporting documentation. Technical report, NIST, 2020. 'https://ntru.org/'.

[Cho17] Tung Chou. Mcbits revisited. In Wieland Fischer and Naofumi Homma, editors, *Cryptographic Hardware and Embedded Systems – CHES 2017*, pages 213–231, Cham, 2017. Springer International Publishing.

[CO14] Jared Casper and Kunle Olukotun. Hardware acceleration of database operations. In *Proceedings of the 2014 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '14, page 151–160, New York, NY, USA, 2014. Association for Computing Machinery.

[DVBK21] Mohajerani Kamyar Dang Viet Ba and Gaj Kris. High-speed hardware architectures and fair fpga benchmarking of crystals-kyber ntru and saber. In *Third PQC Standardization Conference*, 2021.

[GAso22] David Cooper Gorjan Alagic, Daniel Apon and so on. Status report on the second round of the nist post-quantum cryptography standardization process. Technical report, NIST, 2022.

[KRGF+21] Vatistas Kostalabros, Jordi Ribes-González, Oriol Farràs, Miquel Moretó, and Carles Hernandez. Hls-based hw/sw co-design of the post-quantum

classic mceliece cryptosystem. In *2021 31st International Conference on Field-Programmable Logic and Applications (FPL)*, pages 52–59, 2021.

[MRAso20] Daniel J. Bernstein Martin R. Albrecht and so on. Classic mceliece: conservative code-based cryptography. Technical report, NIST, 2020. 'https://classic.mceliece.org/nist/mceliece-20201010.pdf'.

[MVCK17] Susumu Mashimo, Thiem Van Chu, and Kenji Kise. High-performance hardware merge sorter. In *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 1–8, 2017.

[PBL18] Philippos Papaphilippou, Chris Brooks, and Wayne Luk. Flims: Fast lightweight merge sorter. In *2018 International Conference on Field-Programmable Technology (FPT)*, pages 78–85, 2018.

[RKK21] Johannes Roth, Evangelos Karatsiolis, and Juliane Krämer. Classic mceliece implementation with low memory footprint. In Pierre-Yvan Liardet and Nele Mentens, editors, *Smart Card Research and Advanced Applications*, pages 34–49, Cham, 2021. Springer International Publishing.

[R.M79] R.McEliece. A public-key cryptosystem based on algebraic coding theory. pages 42–44, 1979.

[SEC+18] Makoto Saitoh, Elsayed A. Elsayed, Thiem Van Chu, Susumu Mashimo, and Kenji Kise. A high-performance and cost-effective hardware merge sorter without feedback datapath. In *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 197–204, 2018.

[SKLG16] Wei Song, Dirk Koch, Mikel Luján, and Jim Garside. Parallel hardware merge sorter. In *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 95–102, 2016.

[SQA+20] Nikola Samardzic, Weikang Qiao, Vaibhav Aggarwal, Mau-Chung Frank Chang, and Jason Cong. Bonsai: High-performance adaptive merge tree sorting. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 282–294, 2020.

[SWM+10] Abdulhadi Shoufan, Thorsten Wink, H. Gregor Molter, Sorin A. Huss, and Eike Kohnert. A novel cryptoprocessor architecture for the mceliece public-key cryptosystem. *IEEE Transactions on Computers*, 59(11):1533–1546, 2010.

[WSN16] Wen Wang, Jakub Szefer, and Ruben Niederhagen. Solving large systems of linear equations over gf(2) on fpgas. In *2016 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, pages 1–7, 2016.

[WSN17] Wen Wang, Jakub Szefer, and Ruben Niederhagen. Fpga-based key generator for the niederreiter cryptosystem using binary goppa codes. In Wieland Fischer and Naofumi Homma, editors, *Cryptographic Hardware and Embedded Systems – CHES 2017*, pages 253–274, Cham, 2017. Springer International Publishing.

[WSN18] Wen Wang, Jakub Szefer, and Ruben Niederhagen. Fpga-based niederreiter cryptosystem using binary goppa codes. In *PQCrypto*, volume 10786, 2018.

[ZZY+21] Yihong Zhu, Min Zhu, Bohan Yang, Wenping Zhu, Chenchen Deng, Chen Chen, Shaojun Wei, and Leibo Liu. Lwrpro: An energy-efficient configurable crypto-processor for module-lwr. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 68(3):1146–1159, 2021.

[ZZZ+22] Yihong Zhu, Wenping Zhu, Min Zhu, Chongyang Li, Chenchen Deng, Chen Chen, Shuying Yin, Shouyi Yin, Shaojun Wei, and Leibo Liu. A 28nm 48kops 3.4µj/op agile crypto-processor for post-quantum cryptography on multi-mathematical problems. In *2022 IEEE International Solid- State Circuits Conference (ISSCC)*, volume 65, pages 514–516, 2022.