Efficient Implementation of Classic McEliece on ARMv8 processors

https://youtu.be/F7tl64gWfIU

HSU 한성대학교 HANSUNG UNIVERSITY

CryptoCraft LAB

Introduction

- With the development of quantum computers, classic cryptography algorithms are facing threats.
- In preparation for the quantum computer threat, NIST initiated the Post-Quantum Cryptography standardization.
- The **Classic McEliece** was selected as one of the Round 4 candidate algorithms.
- In this paper, we proposed efficient implementation of Classic McEliece.
  - Optimized implementation of multiplier.

Our Contribution

- Multiplication with Parallel Operations on ARMv8 Processors.
  - **Reordering the order of operations** using the **XOR** operation.
  - Utilize vector registers to perform **four values parallel operations.**

- The first implementation of Classic McEliece multiplier on ARMv8 processors using vector registers.
  - It might be helpful to following researchers.

Background: Classic McEliece

- **Code-based cryptography**.
- Round 4 candidate algorithms : BIKE, **Classic McEliece**, HQC, ~~SIKE~~
- Classic McEliece is designed to combine the advantages of McEliece and Niederreiter cryptosystem.
  - Using the Parity Check Matrix(H) used as the public key in Niederreiter cryptosystem.

Background: Classic McEliece

- It has a very short ciphertext length, and encryption and decryption are possible in a short time.
- However, the length of the public key is very large.
  - 256KB to 1.3MB range
- The key generation process is slow.

Algorithm	m	n	t	Security level	Public key	Secret key	Ciphertext
Mceliece 348864	12	3,488	64	1	261,120	6,492	128
Mceliece 460896	13	4,608	86	3	524,160	13,608	188
Mceliece 6688128	13	6,688	128	5	1,044,992	13,932	240
Mceliece 6960119	13	6,960	119	5	1,047,319	13,948	226
Mceliece 8192128	13	8,192	128	5	1,357,824	14,120	240

Table 1. Parameters of Classic McEliece.

Background: 64-bit ARMv8 Processor

- Widely used high-performance processors.
- Support both 32-bit AArch32 and 64-bit AArch64 architectures.
- Provide 64-bit ARM & 128-bit vector registers and instruction sets.
  - 64-bit general-purpose registers from x0 to x30.
  - 128-bit vector registers from v0 to v31.
- **Parallel implementation using vector instructions.**

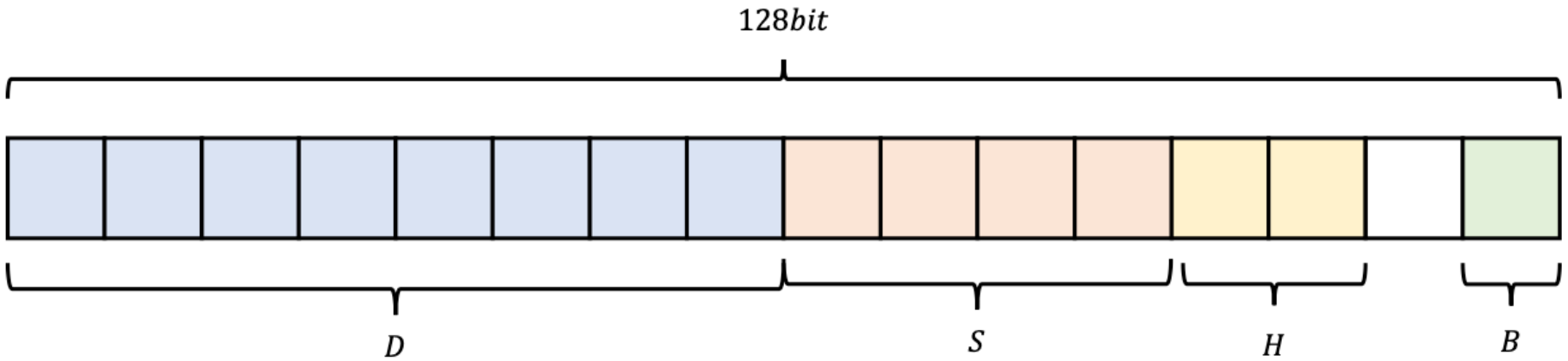
Background: 64-bit ARMv8 Processor

- The vector registers can be processed by dividing stored values into specific units.

  : Byte (8-bit), Half word (16-bit), Single word (32-bit), and Double word (64-bit)

- Arrangement specifiers belong to instructions.

- **Operational units can be changed by specifiers of instructions.**



Proposed Method

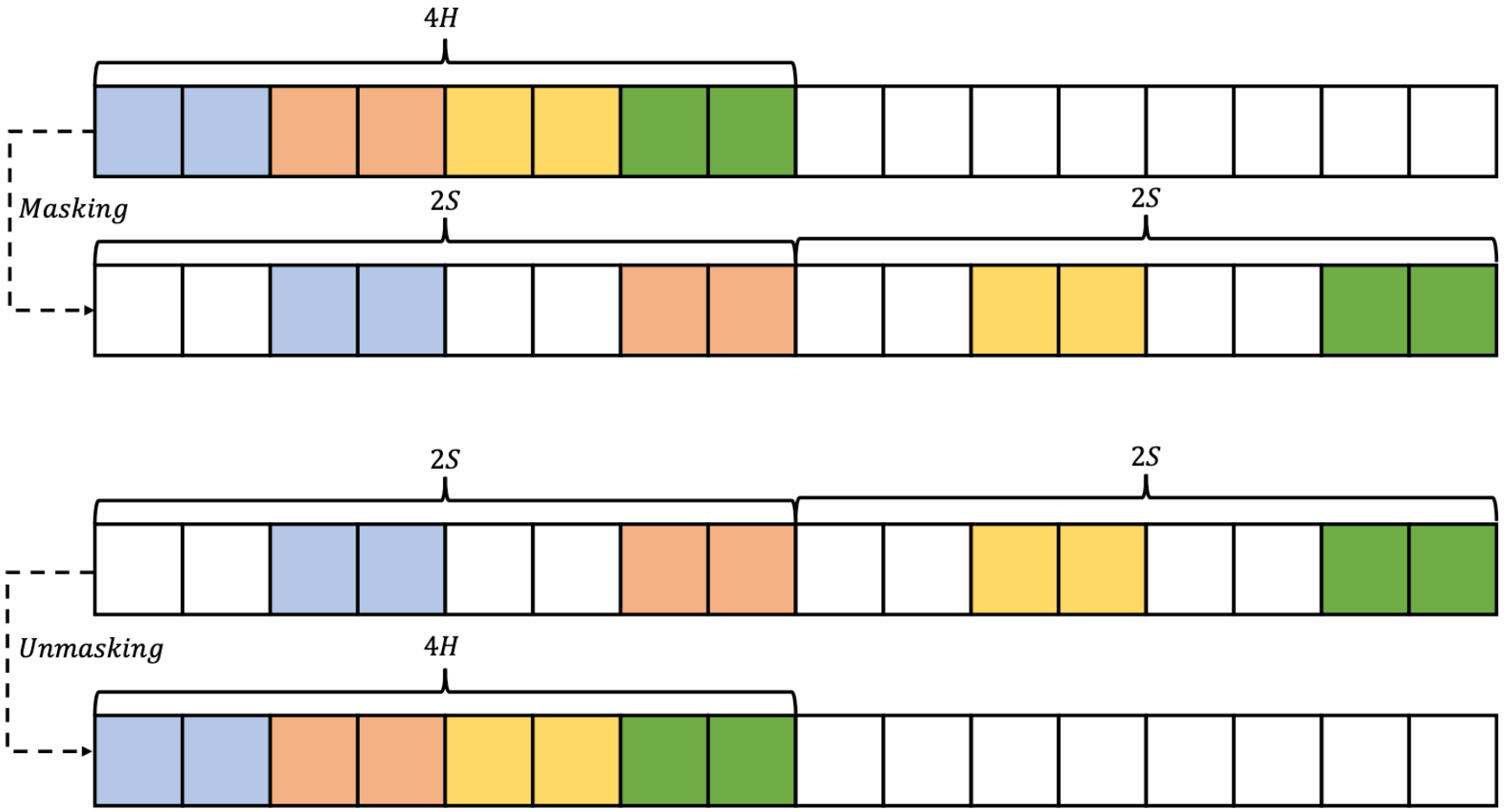
- In Classic McEliece, Multiplication is performed on the extended binary finite-filed $\boldsymbol{F_{2^m}}$ (m is 12 or 13).
  - $\boldsymbol{F_{2^{12}}}$ consists of $\mathbb{F}_2[x]/(x^{12}+x^3+1)$.
  - $\boldsymbol{F_{2^{13}}}$ consists of $\mathbb{F}_2[x]\ /\ (x^{13}+\ x^4+x^3+x+1)$.
  - **The expensive operations are multiplication and inversion on finite-field $\boldsymbol{F_{2^m}}$.**
- **Optimized Multiplication for $\boldsymbol{F_{2^m}}$.**

Proposed Method

- We proposed Multiplication with Parallel Operations.
- Optimization techniques :
  - Apply **masking operation** for multiplication.
  - **Parallel multiplier using reordering of operations.**
  - Utilize vector registers to perform **four values parallel operations.**

Proposed Method

- **Masking & Unmasking operation**



Proposed Method

- **Optimization implementation code for multiplication on F_{2^m}.**
  - Written by ARM assembly.

```
void PQCLEAN_MCELIECE348864_CLEAN_GF_mul(gf *out, const gf *in0, const gf *in1) {
    int i, j;

    gf prod[ SYS_T * 2 - 1 ];

    for (i = 0; i < SYS_T * 2 - 1; i++) {
        prod[i] = 0;
    }

    for (i = 0; i < SYS_T; i++) { //1
        for (j = 0; j < SYS_T; j++) {
            prod[i + j] ^= PQCLEAN_MCELIECE348864_CLEAN_gf_mul(in0[i], in1[j]);
        }
    }

    //

    for (i = (SYS_T - 1) * 2; i >= SYS_T; i--) {
        prod[i - SYS_T +  9] ^= PQCLEAN_MCELIECE348864_CLEAN_gf_mul(prod[i], (gf)  877);
        prod[i - SYS_T +  7] ^= PQCLEAN_MCELIECE348864_CLEAN_gf_mul(prod[i], (gf) 2888);
        prod[i - SYS_T +  5] ^= PQCLEAN_MCELIECE348864_CLEAN_gf_mul(prod[i], (gf) 1781);
        prod[i - SYS_T +  0] ^= PQCLEAN_MCELIECE348864_CLEAN_gf_mul(prod[i], (gf)  373);
    }

    for (i = 0; i < SYS_T; i++) {
        out[i] = prod[i];
    }
}
```

```
gf PQCLEAN_MCELIECE348864_CLEAN_gf_mul(gf in0, gf in1) {
    int i;

    uint32_t tmp;
    uint32_t t0;
    uint32_t t1;
    uint32_t t;

    t0 = in0;
    t1 = in1;

    tmp = t0 * (t1 & 1);

    for (i = 1; i < GFBITS; i++) {
        tmp ^= (t0 * (t1 & (1 << i)));
    }

    t = tmp & 0x7FC000;
    tmp ^= t >> 9;
    tmp ^= t >> 12;

    t = tmp & 0x3000;
    tmp ^= t >> 9;
    tmp ^= t >> 12;

    return tmp & ((1 << GFBITS) - 1);
}
```

Algorithm 1 In Classic McEliece-348864, 16-bit value multiplication operation;(x0 : Result of Multiplication Operation, x1, x2 : Input of Multiplication Operation)

```
 1: mov x9, #64
 2: loop_i:
 3: mov x8, #64

 4: loop_j:
 5: ld1R {v0.4h}, [x1]
 6: ld1 {v1.4h}, [x2], #8

    //masking
 7: zip1.8h v0, v0, v3
 8: zip1.8h v1, v1, v3

    //Multiplication
 9: and.16b v8, v1, v4
10: mul.4s v6, v8, v0


11: shl.4s v5, v4, #1
12: and.16b v8, v1, v5
13: mul.4s v8, v0, v8
14: eor.16b v6, v6, v8

15: shl.4s v5, v4, #2
16: and.16b v8, v1, v5
17: mul.4s v8, v0, v8
18: eor.16b v6, v6, v8

    ⋮

19: shl.4s v5, v4, #10
20: and.16b v8, v1, v5
21: mul.4s v8, v0, v8
22: eor.16b v6, v6, v8

23: shl.4s v5, v4, 11

24: and.16b v8, v1, v5
25: mul.4s v8, v0, v8
26: eor.16b v6, v6, v8

    //Reduction
27: and.16b v9, v6, v14
28: sri.4s v10, v9, #9
29: eor.16b v6, v6, v10
30: sri.4s v10, v9, #12
31: eor.16b v6, v6, v10

32: and.16b v9, v6, v15
33: sri.4s v10, v9, #9
34: eor.16b v6, v6, v10
35: sri.4s v10, v9, #12
36: eor.16b v6, v6, v10

37: and.16b v6, v6, v11

    //unmasking
38: uzp1.8h v6, v6, v7

39: ld1.4h {v2}, [x0]
40: eor.16b v2, v2, v6

41: st1.4h {v2}, [x0], #8

42: add x8, x8, #-4
43: cbnz x8, loop_j

44: add x0, x0, #-126
45: add x2, x2, #-128
46: add x1, x1, #2

47: add x9, x9, #-1
48: cbnz x9, loop_i
```

Proposed Method

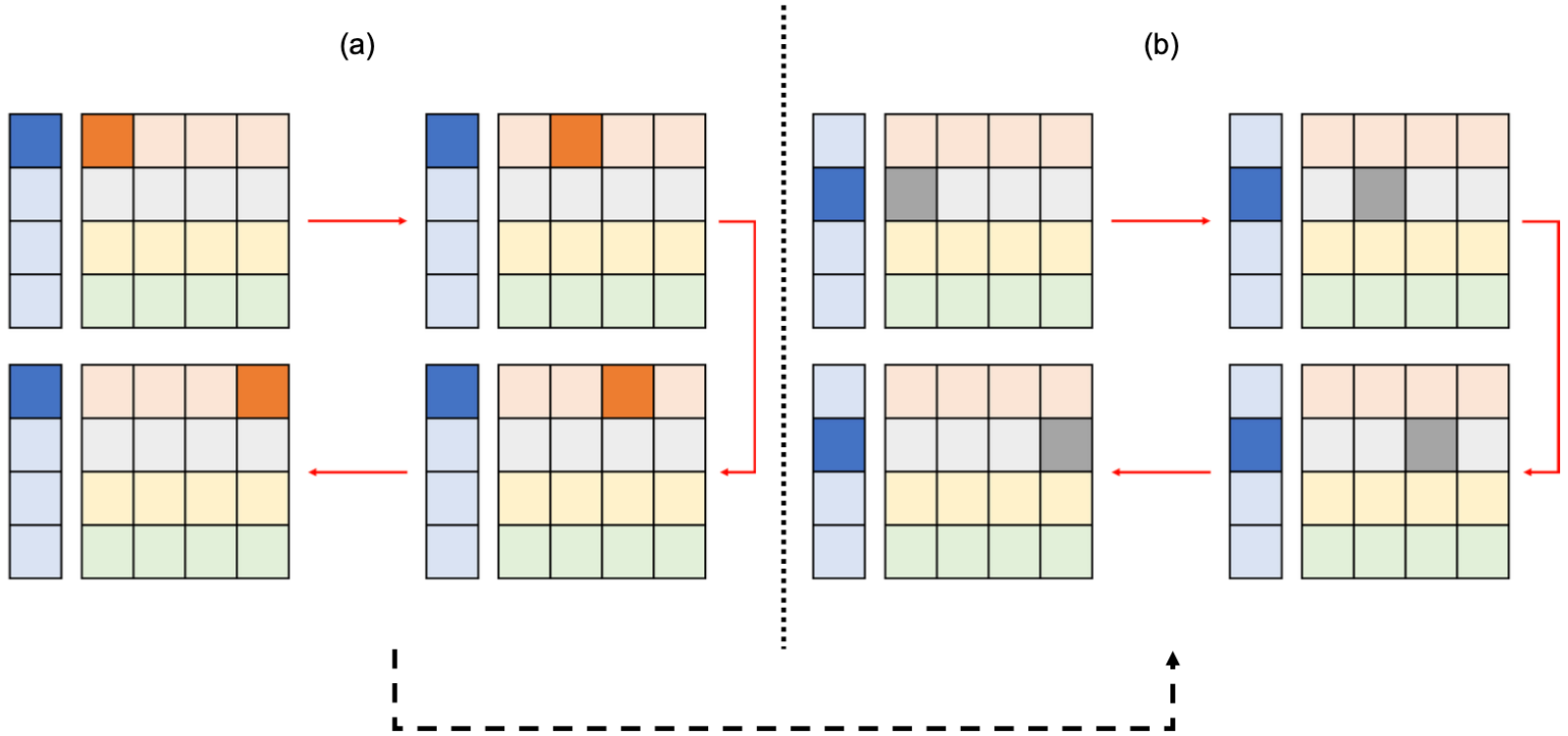
- **Reordering of operations**
  - A commutative operation indicates that the order of operands can be inter changed without affecting the result.

$$a * b = b * a$$

- **The XOR operation is also commutative.**
- **We performed operations by modifying the order of multiplicat ion operations by utilizing these properties.**

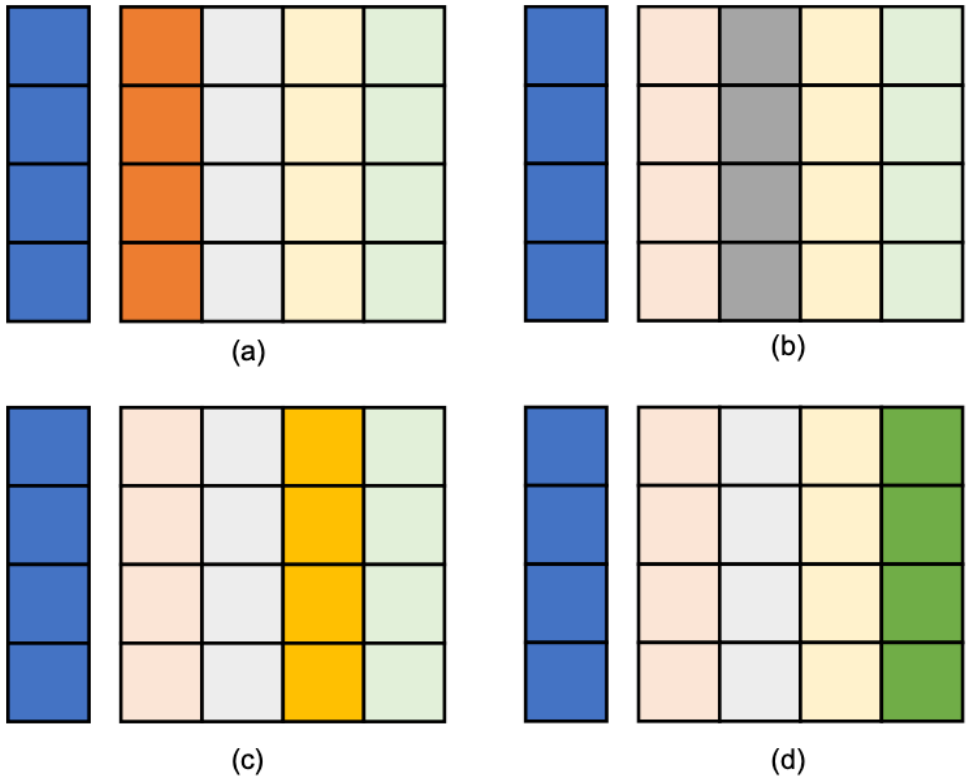
Proposed Method

- Represent the original order of multiplication operations performed.
  - Once the operations depicted in (a) are completed, the multiplication computations proceed in the order presented in (b).



Proposed Method

- Sequence of **proposed order of multiplication operation** using **four 16-bit loaded vector registers.**



Proposed Method

- **Part of the optimization implementation code on $F_{2^{13t}}$.**
  - Written by ARM assembly.

```
void PQCLEAN_MCELIECE348864_CLEAN_GF_mul(gf *out, const gf *in0, const gf *in1) {
    int i, j;

    gf prod[ SYS_T * 2 - 1 ];

    for (i = 0; i < SYS_T * 2 - 1; i++) {
        prod[i] = 0;
    }

    for (i = 0; i < SYS_T; i++) { //1
        for (j = 0; j < SYS_T; j++) {
            prod[i + j] ^= PQCLEAN_MCELIECE348864_CLEAN_gf_mul(in0[i], in1[j]);
        }
    }

    //

    for (i = (SYS_T - 1) * 2; i >= SYS_T; i--) {
        prod[i - SYS_T +  9] ^= PQCLEAN_MCELIECE348864_CLEAN_gf_mul(prod[i], (gf)  877);
        prod[i - SYS_T +  7] ^= PQCLEAN_MCELIECE348864_CLEAN_gf_mul(prod[i], (gf) 2888);
        prod[i - SYS_T +  5] ^= PQCLEAN_MCELIECE348864_CLEAN_gf_mul(prod[i], (gf) 1781);
        prod[i - SYS_T +  0] ^= PQCLEAN_MCELIECE348864_CLEAN_gf_mul(prod[i], (gf)  373);
    }

    for (i = 0; i < SYS_T; i++) {
        out[i] = prod[i];
    }
}
```

[i] [i-SYS_T+9], [i-SYS_T +7], [i-SYS_T+5] [i-SYS_T

[126]	71,	69,	67, 62
[125]	70,	68,	66, 61
[124]	69,	67,	65, 60
[123]	68,	66,	64, 59
[122]	67,	65,	63, 58
[121]	66,	64,	62, 57
[120]	65,	63,	61, 56
[119]	64,	62,	60, 55
[118]	63,	61,	59, 54
[117]	62,	60,	58, 53
[116]	61,	59,	57, 52
[115]	60,	58,	56, 51
[114]	59,	57,	55, 50
[113]	58,	56,	54, 49
[112]	57,	55,	53, 48
[111]	56,	54,	52, 47
[110]	55,	53,	51, 46
[109]	54,	52,	50, 45
[108]	53,	51,	49, 44
[107]	52,	50,	48, 43
[106]	51,	49,	47, 42
[105]	50,	48,	46, 41
[104]	49,	47,	45, 40
[103]	48,	46,	44, 39
[102]	47,	45,	43, 38
[101]	46,	44,	42, 37
[100]	45,	43,	41, 36
[99]	44,	42,	40, 35
[98]	43,	41,	39, 34
[97]	42,	40,	38, 33
[96]	41,	39,	37, 32
[95]	40,	38,	36, 31
[94]	39,	37,	35, 30
[93]	38,	36,	34, 29
[92]	37,	35,	33, 28
[91]	36,	34,	32, 27
[90]	35,	33,	31, 26
[89]	34,	32,	30, 25
[88]	33,	31,	29, 24
[87]	32,	30,	28, 23
[86]	31,	29,	27, 22
[85]	30,	28,	26, 21
[84]	29,	27,	25, 20
[83]	28,	26,	24, 19
[82]	27,	25,	23, 18
[81]	26,	24,	22, 17
[80]	25,	23,	21, 16
[79]	24,	22,	20, 15
[78]	23,	21,	19, 14
[77]	22,	20,	18, 13
[76]	21,	19,	17, 12
[75]	20,	18,	16, 11
[74]	19,	17,	15, 10
[73]	18,	16,	14, 9
[72]	17,	15,	13, 8
[71]	16,	14,	12, 7
[70]	15,	13,	11, 6
[69]	14,	12,	10, 5
[68]	13,	11,	9, 4
[67]	12,	10,	8, 3
[66]	11,	9,	7, 2
[65]	10,	8,	6, 1
[64]	9,	7,	5, 0

Algorithm 2 In Classic McEliece-348864, 16-bit value multiplication on $\mathbb{F}_{2^{13t}}$ operation.

```
 1: mov x9, #15
 2: add x0, x0, #246

 3: loop:
 4: 4_byte_gf_mul_877
 5: 4_byte_gf_mul_2888
 6: 4_byte_gf_mul_1781
 7: 4_byte_gf_mul_373

 8: add x0, x0, #-110
 9: ld1.4h {v2}, [x0]
10: eor.16b v2, v2, v20
11: st1.4h {v2}, [x0]

12: add x0, x0, #-4
13: ld1.4h {v2}, [x0]
14: eor.16b v2, v2, v21
15: st1.4h {v2}, [x0]

16: add x0, x0, #-4
17: ld1.4h {v2}, [x0]
18: eor.16b v2, v2, v22
19: st1.4h {v2}, [x0]

20: add x0, x0, #-10
21: ld1.4h {v2}, [x0]
22: eor.16b v2, v2, v23
23: st1.4h v2, [x0]

24: add x0, x0, #120

25: add x9, x9, #-1
26: cbnz x9, loop


27: add x0, x0, #2

28: 4_byte_gf_mul_877
29: 4_byte_gf_mul_2888
30: 4_byte_gf_mul_1781
31: 4_byte_gf_mul_373

32: add x0, x0, #-110
33: ld1.4h {v2}, [x0]
34: eor.16b v2, v2, v20

35: st1 v2.h[0], [x0], #2
36: st1 v2.h[1], [x0], #2
37: st1 v2.h[2], [x0]
38: add x0, x0, #-4

39: add x0, x0, #-4
40: ld1.4h {v2}, [x0]
41: eor.16b v2, v2, v21

42: st1 v2.h[0], [x0], #2
43: st1 v2.h[1], [x0], #2
44: st1 v2.h[2], [x0]
45: add x0, x0, #-4

46: add x0, x0, #-4
47: ld1.4h {v2}, [x0]
48: eor.16b v2, v2, v22

49: st1 v2.h[0], [x0], #2
50: st1 v2.h[1], [x0], #2
51: st1 v2.h[2], [x0]
52: add x0, x0, #-4

53: add x0, x0, #-10
54: ld1.4h {v2}, [x0]
55: eor.16b v2, v2, v23

56: st1 v2.h[0], [x0], #2
57: st1 v2.h[1], [x0], #2
58: st1 v2.h[2], [x0]
59: add x0, x0, #-4
```

Proposed Method

- **One of the macros used by Algorithm 2.**
  - Perform the load for four 16 bits and performs one masking process over one whole.

```
void PQCLEAN_MCELIECE348864_CLEAN_GF_mul(gf *out, const gf *in0, const gf *in1) {
    int i, j;

    gf prod[ SYS_T * 2 - 1 ];

    for (i = 0; i < SYS_T * 2 - 1; i++) {
        prod[i] = 0;
    }

    for (i = 0; i < SYS_T; i++) { //1
        for (j = 0; j < SYS_T; j++) {
            prod[i + j] ^= PQCLEAN_MCELIECE348864_CLEAN_gf_mul(in0[i], in1[j]);
        }
    }

    //

    for (i = (SYS_T - 1) * 2; i >= SYS_T; i--) {
        prod[i - SYS_T + 9] ^= PQCLEAN_MCELIECE348864_CLEAN_gf_mul(prod[i], (gf) 877);
        prod[i - SYS_T + 7] ^= PQCLEAN_MCELIECE348864_CLEAN_gf_mul(prod[i], (gf) 2888);
        prod[i - SYS_T + 5] ^= PQCLEAN_MCELIECE348864_CLEAN_gf_mul(prod[i], (gf) 1781);
        prod[i - SYS_T + 0] ^= PQCLEAN_MCELIECE348864_CLEAN_gf_mul(prod[i], (gf) 373);
    }

    for (i = 0; i < SYS_T; i++) {
        out[i] = prod[i];
    }
}
```

```
gf PQCLEAN_MCELIECE348864_CLEAN_gf_mul(gf in0, gf in1) {
    int i;

    uint32_t tmp;
    uint32_t t0;
    uint32_t t1;
    uint32_t t;

    t0 = in0;
    t1 = in1;

    tmp = t0 * (t1 & 1);

    for (i = 1; i < GFBITS; i++) {
        tmp ^= (t0 * (t1 & (1 << i)));
    }

    t = tmp & 0x7FC000;
    tmp ^= t >> 9;
    tmp ^= t >> 12;

    t = tmp & 0x3000;
    tmp ^= t >> 9;
    tmp ^= t >> 12;

    return tmp & ((1 << GFBITS) - 1);
}
```

Algorithm 1 In Classic McEliece-348864, 16-bit value multiplication operation;(x0 : Result of Multiplication Operation, x1, x2 : Input of Multiplication Operation)

```
1: mov x9, #64
2: loop_i:
3: mov x8, #64

4: loop_j:
5: ld1R {v0.4h}, [x1]
6: ld1 {v1.4h}, [x2], #8

24: and.16b v8, v1, v5
25: mul.4s v8, v0, v8
26: eor.16b v6, v6, v8

   //Reduction
27: and.16b v9, v6, v14
28: sri.4s v10, v9, #9
29: eor.16b v6, v6, v10
```

Algorithm 3 In Classic McEliece-348864, multiplication $\mathbb{F}_{2^{13}}$ macro for 16-bit value multiplication for $\mathbb{F}_{2^{13t}}$.

```
.macro 4_byte_gf_mul_877
1: ld1 {v0.4h}, [x0]
2: mov.8h v1, v12

   //masking
3: zip1.8h v0, v0, v3

   //Multiplication
4: and.16b v8, v1, v4
5: mul.4s v6, v8, v0

6: shl.4s v5, v4, #1
7: and.16b v8, v1, v5
8: mul.4s v8, v0, v8
9: eor.16b v6, v6, v8

10: shl.4s v5, v4, #2
11: and.16b v8, v1, v5
12: mul.4s v8, v0, v8
13: eor.16b v6, v6, v8

    ⋮

14: shl.4s v5, v4, #10
15: and.16b v8, v1, v5

16: mul.4s v8, v0, v8
17: eor.16b v6, v6, v8

18: shl.4s v5, v4, 11
19: and.16b v8, v1, v5
20: mul.4s v8, v0, v8
21: eor.16b v6, v6, v8

22: and.16b v9, v6, v14
23: sri.4s v10, v9, #9
24: eor.16b v6, v6, v10
25: sri.4s v10, v9, #12
26: eor.16b v6, v6, v10

27: and.16b v9, v6, v15
28: sri.4s v10, v9, #9
29: eor.16b v6, v6, v10
30: sri.4s v10, v9, #12
31: eor.16b v6, v6, v10

32: and.16b v6, v6, v11

   //unmasking
33: uzp1.8h v20, v6, v7
.endm
```

Evaluation

- Implementation Environment
  - Target Processor: Apple Macbook Pro 13 with **Apple M1** chip (3.2GHz)
  - Framework: Xcode Integrated Development Environment
  - Compiled with -O2 option(i.e. faster)

- Assessment Methods
  - **Since ARMv8 does not have a Classic McEliece implementation**,

    the performance is compared with the PQ-Clean project reference code.
  - **Multiplier** was repeated by **1,000,000** times to measure the operation time.

Evaluation

- Performance enhancement of up to 2.82× compared to the reference implementation.

Scheme	Unit	PQ-Clean	**This Work**	**Improvement**
Classic McEliece 348864	ms	4,294	**5,429**	**0.79x**
	cc	13,740	**17,372**	
Classic McEliece 460896	ms	40,579	**14,563**	**2.79x**
	cc	129,852	**46,601**	
Classic McEliece 6699128	ms	71,185	**25,343**	**2.80x**
	cc	227,792	**81,097**	
Classic McEliece 6960119	ms	61,690	**21,972**	**2.81x**
	cc	197,408	**70,310**	
Classic McEliece 819212	ms	71,193	**25,222**	**2.82x**
	cc	227,817	**80,710**	

Conclusion

- We proposed **parallel multiplier** of Classic McEliece.
  - Optimized implementation version on ARM processors.
- Method of proposed parallel multiplication.
  - **Reordering the order of operations**.
  - Utilize vector registers to perform **four values parallel operations.**
- It showed a performance improvement of up to **2.84 times**.

Q & A