Optimized SIKE Round 2 on 64-bit ARM

Hwajeong Seo[1]*, Amir Jalali[2], and Reza Azarderakhsh[2]

IT Department, Hansung University, Seoul, South Korea, hwajeong84@gmail.com
Department of Computer and Electrical Engineering and Computer Science,
Florida Atlantic University, FL, USA,
{ajalali2016,razarderakhsh}@fau.edu

Abstract. In this work, we present the first highly-optimized implementation of Supersingular Isogeny Key Encapsulation (SIKE) submitted to NIST's second round of post quantum standardization process, on 64-bit ARMv8 processors. To the best of our knowledge, this work is the first optimized implementation of SIKE round 2 on 64-bit ARM over SIKEp434 and SIKEp610. The proposed library is explicitly optimized for these two security levels and provides constant-time implementation of the SIKE mechanism on ARMv8-powered embedded devices. We adapt different optimization techniques to reduce the total number of underlying arithmetic operations on the filed level. In particular, the benchmark results on embedded processors equipped with ARM Cortex-A53@1.536GHz show that the entire SIKE round 2 key encapsulation mechanism takes only 84 ms at NIST's security level 1. Considering SIKE's extremely small key size in comparison to other candidates, our result implies that SIKE is one of the promising candidates for key encapsulation mechanism on embedded devices in the quantum era.

Keywords: Post-quantum cryptography, isogeny-based cryptography, 64-bit ARM processor, ARM assembly, key encapsulation mechanism

1 Introduction

Initiated by the National Institute of Standards and Technology (NIST), Post-Quantum Cryptography (PQC) has been elevated to a standardization process to solicit, evaluate, and standardize one or more quantum-resistant public-key cryptographic algorithms [18]. To prepare for security concerns caused by quantum computers, in 2016, NIST called for the cryptographic algorithms which were assumed to be resistance against high-scale quantum computers. These proposals provided key encapsulation mechanism (KEM) or digital signature algorithms from different arithmetic structures, resulting in different characteristics and parameters. Recently, NIST announced approved candidates for round 2 which are the most promising candidates in terms of security, performance, and compatibility with current technology. For the key encapsulation mechanism, only 17 candidates made it through to the second round for being evaluated and analyzed from different perspectives.

* Corresponding Author

Different PQC candidates are constructed on hard mathematical problems which are assumed to be impossible to solve even for large-scale quantum computers. These problems can be categorized into five main categories: code-based cryptography, lattice-based cryptography, hash-based cryptography, multivariate cryprography, and supersingular isogeny-based cryptography, see, for instance [8].

Supersingular Isogeny Key Encapsulation (SIKE) mechanism is one of the PQC candidates which is constructed on the hardness of solving isogeny maps between supersingular elliptic curves. In fact, SIKE is the only candidate that offers the quantum-resistance cryptographic construction over elliptic curves, resulting in well-known structures in implementation perspective. The proposed key encapsulation mechanism is derived from the original Jao-De Feo's Diffie-Hellman key-exchange and public-key encryption algorithms [15]. However, constructing cryptographic structures from hardness of supersingular isogeny graphs was introduced by Charels-Lauter-Goren [7].

The first round SIKE submission [4] offered three different security levels known as SIKEp503, SIKEp751, and SIKEp964. According to the best known quantum attacks on solving supersingular isogeny problem by that time, the proposed security levels met NIST's level 1, 3, and 5 requirements, respectively.

However, recent studies on the cost of solving isogeny problem on quantum computers by Adj et al. [1] revealed that the security assumptions for SIKE was too conservative. In fact, a set of realistic models of quantum computation on solving Computational Supersingular Isogeny (CSSI) problem in [1] suggests that the Oorschot-Wiener golden collision search is the most powerful attack on the CSSI problem, resulting in significant improvement on the SIKE's classical and quantum security levels.

Accordingly, the second round SIKE [3] offers a new set of security levels which are more realistic and provide significant improvement on the key encapsulation performance. In particular, decreasing the bit-length of SIKE's primes translates to notable performance improvement, making this scheme suitable for many potential applications on low-end and embedded devices.

In this work, we provide a full report on the highly-optimized implementation of SIKE on 64-bit ARM processors over all the proposed security levels. In particular, the reference optimized implementation of SIKE [3] on 64-bit ARM only targets two security levels, i.e., SIKEp503 and SIKEp751. Therefore, in this work, we address this shortcoming by providing the KEM full benchmarks on different security levels which provide a reference for the performance analysis of this scheme for the second round.

Our proposed library takes advantage of state-of-the-art engineering techniques as well as low level assembly optimizations. We studied different approaches for finite field arithmetic implementation over SIKE's new primes. Our benchmark results offer significant improvement in performance compared to portable implementation, suggesting the possible integration of this scheme on mobile devices in the future.

Alice		**Bob**
$\mathtt{sk}_A : m_A, n_A \in_R /\ell_A^{e_A}$		$\mathtt{sk}_B : m_B, n_B \in_R /\ell_B^{e_B}$
$\phi_A := E/\langle [m_A]P_A + [n_A]Q_A \rangle$		$\phi_B := E/\langle [m_B]P_B + [n_B]Q_B \rangle$
$\mathtt{pk}_A : \phi_A(P_B), \phi_A(Q_B), E_A$		$\mathtt{pk}_B : \phi_B(P_A), \phi_B(Q_A), E_B$
	$\xrightarrow{\mathtt{pk}_A}$	
	$\xleftarrow{\mathtt{pk}_B}$	
$E_{AB} :=$		$E_{BA} :=$
$E_B/\langle [m_A]\phi_B(P_A) + [n_A]\phi_B(Q_A) \rangle$		$E_A/\langle [m_B]\phi_A(P_B) + [n_B]\phi_A(Q_B) \rangle$
Shared secret: $j(E_{AB})$		**Shared secret:** $j(E_{BA})$

Fig. 1: SIDH key exchange protocol.

2 Background

In this section, we briefly review the SIDH protocol and the required steps for Alice and Bob to generate a shared secret. Furthermore, we describe the SIKE, a post-quantum key encapsulation mechanism from isogenies of supersingular elliptic curves which was submitted to NIST's PQC standardization competition. We refer the readers to [15, 5] for further details.

2.1 SIDH key exchange

In 2011, Jao and De Feo [15] proposed the SIDH, a quantum resistant key exchange protocol from isogenies of supersingular elliptic curves. Similar to classical Diffie-Hellman key exchange, SIDH protocol is constructed over some public parameters which are agreed upon by communication parties prior to key exchange.

Public parameters Fix a prime p of the form $p = \ell_A^{e_A} \cdot \ell_B^{e_B} \cdot f \pm 1$ where ℓ_A and ℓ_B are small primes, e_A and e_B are positive integers, and f is a very small cofactor. We define a based supersingular elliptic curve E over $\mathbb{F}_{p^2}$ with cardinality $\#E = (\ell_A^{e_A} \cdot \ell_B^{e_B} \cdot f \mp 1)^2$, and base points $\{P_A, Q_A\}$ and $\{P_B, Q_B\}$ from the torsion subgroups $E[\ell_A^{e_A}]$ and $E[\ell_B^{e_B}]$ respectively, such that $\langle P_A, Q_A \rangle = E[\ell_A^{e_A}]$ and $\langle P_B, Q_B \rangle = E[\ell_A^{e_B}]$.

Key exchange protocol Alice randomly chooses two integers $m_A, n_A \in \mathbb{Z}/\ell_A^{e_A}\mathbb{Z}$, not both divisible by ℓ_A as her secret key and computes an isogeny $\phi_A : E \to E_A$ using kernel $R_A := \langle [m_A]P_A + [n_A]Q_A \rangle$. Alice also computes the image points $\{\phi_A(P_B), \phi_A(Q_B)\} \subset E_A$ by applying her secret isogeny ϕ_A to the public basis P_B and Q_B. She sends $\phi_A(P_B), \phi_A(Q_B)$ and E_A to Bob as her public key. Bob also selects random elements $m_B, n_B \in \mathbb{Z}/\ell_B^{e_B}\mathbb{Z}$, not both divisible by ℓ_B and computes a secret isogeny $\phi_B : E \to E_B$ from kernel $R_B := \langle [m_B]P_B + [n_B]Q_B \rangle$, along with image points $\{\phi_B(P_A), \phi_B(Q_A)\} \subset E_B$. He sends his public key, i.e., $\phi_B(P_A), \phi_B(Q_A)$ and E_B to Alice.

Alice		**Bob**
Key generation:		
$\mathtt{pk}_A = [E_A, \phi_A(P_B), \phi_A(Q_B)]$		
$s \in_R \{0,1\}^t$		
		Encapsulation:
		$m \in_R \{0,1\}^t$
		$r = H_1(m \parallel \mathtt{pk}_A)$
		$\mathtt{pk}_B(r) = [E_B, \phi_B(P_A), \phi_B(Q_A)]$
		$j = j(E_{BA})$
		$c = (c_0, c_1) = (\mathtt{pk}_B(r), H_2(j) \oplus m)$
		$K = H_3(m \parallel c)$
	$\xleftarrow{(c_0,c_1)}$	
Decapsulation:		
$j = j(E_{AB})$		
$m' = c_1 \oplus H_2(j)$		
$r' = H_1(m' \parallel \mathtt{pk}_A)$		
If $(\mathtt{pk}_B(r') = c_0) \rightarrow K = H_3(m' \parallel c)$		
If $(\mathtt{pk}_B(r') \neq c_0) \rightarrow K = H_3(s \parallel c)$		

Fig. 2: SIKE mechanism.

In the second round of key exchange, Alice uses Bob's public key $(\phi_B(P_A), \phi_B(Q_A), E_B)$ and computes an isogeny $\phi'_A : E_B \rightarrow E_{AB}$ from kernel equal to $\langle [m_A]\phi_B(P_A) + [n_A]\phi_B(Q_A)\rangle$; Similarly, Bob computes an isogeny $\phi'_B : E_A \rightarrow E_{BA}$ having kernel $\langle [m_B]\phi_A(P_B) + [n_B]\phi_A(Q_B)\rangle$ using Alice's public key. Since the common j-invariant of E_{AB} and E_{BA} are equal, they use this value to form a secret shared key. The entire SIDH key exchange protocol is illustrated in Figure 1.

2.2 SIKE mechanism

SIKE mechanism is constructed by applying a transformation of Hofheinz, Hövelmanns, and Kiltz [12] to the supersingular isogeny Public Key Encryption (PKE) scheme described in [15]. It is an actively secure key encapsulation mechanism (IND-CCA KEM) which addresses the static key vulnerability of SIDH due to active attacks in [11].

Public parameters Similar to SIDH, SIKE can be defined over a prime of the form $p = \ell_A^{e_A} \cdot \ell_B^{e_B} \cdot f \pm 1$. However, for efficiency reasons, $\ell_A = 2, \ell_B = 3$, and $f = 1$ are fixed, thus the SIKE prime has the form of $p = 2^{e_A} \cdot 3^{e_B} - 1$. The starting supersingular elliptic curve $E_0/\mathbb{F}_{p^2} : y^2 = x^3 + x$ with cardinality equal to $(2^{e_A} \cdot 3^{e_B})^2$, along with base points $\langle P_A, Q_A\rangle = E_0[2^{e_A}]$ and $\langle P_B, Q_B\rangle = E_0[3^{e_B}]$ are defined as public parameters.

Key encapsulation mechanism The key encapsulation mechanism can be divided into three main operations: Alice's key generation, Bob's key encapsulation, and Alice's key decapsulation. We describe each operation in the following. Figure 2 presents the entire key encapsulation mechanism in a nutshell.

Key generation. Alice randomly chooses an integer $\text{sk}_A \in \mathbb{Z}/2^{e_A}\mathbb{Z}$ and by applying an isogeny $\phi_A : E_0 \to E_A$ with kernel $R_A := \langle P_A + [\text{sk}_A]Q_A \rangle$ to the base points $\{P_B, Q_B\}$, computes her public key $\text{pk}_A = [E_A, \phi_A(P_B), \phi_A(Q_B)]$. Moreover, she generates an t-bit[1] random sequence $s \in_R \{0,1\}^t$.

Encapsulation. Bob generates an t-bit random message m $\in_R \{0,1\}^t$, concatenates it with Alice's public key pk_A and computes an e_B-bit hash value r using `cSHAKE256` hash function H_1, taking $m \parallel \text{pk}_A$ as the input. Using r, he applies a secret isogeny $\phi_B : E0 \to EB$ to the base points $\{P_A, Q_A\}$ and forms his public key $\text{pk}_B(r) = [E_B, \phi_B(P_A), \phi_B(Q_A)]$. Bob also computes the common j-invariant of curve E_{BA} by applying another isogeny $\phi'_B : EA \to E_{BA}$ using Alice's public key. Bob forms a ciphertext $c = (c_0, c_1)$, such that:

$$c = (c_0, c_1) = (\text{pk}_B(r), H_2(j(E_{BA})) \oplus m),$$

where H_2 is a `cSHAKE256` hash with a custom length output and a defined initialization parameter. Finally, Bob computes the shared secret as $K = H_3(m \parallel c)$ and sends c to Alice.

Decapsulation. Upon receipt of c, Alice computes the common j-invariant of E_{AB} by applying her secret isogeny to E_B. She computes $m' = c_1 \oplus H_2(j(E_{AB}))$ and $r' = H_1(m \parallel \text{pk}_A)$. Finally, she validates Bob's public key by computing $\text{pk}_B(r')$ and comparing it with c_0. She generates the same shared secret $K = H_3(m' \parallel c)$ if the public key is valid, otherwise she outputs a random value $K = H_3(c \parallel s)$ to be resistant against active attacks.

3 Target Architecture

ARMv8 Cortex-A, or simply ARMv8, is the latest generation of ARM architectures targeted at the "application" profile. It includes the typical 32-bit architecture, called "AArch32", and advanced 64-bit architecture named "AArch64" with its associated instruction set "A64" [2]. AArch32 preserves backwards compatibility with ARMv7 and supports the so-called "A32" and "T2" instructions sets, which correspond to the traditional 32-bit and Thumb instruction sets, respectively. AArch64 comes equipped with 31 general purpose 64-bit registers (i.e. `X0`∼`X30`) and one zero register (i.e. `XZR`), and an instruction set supporting 32-bit and 64-bit operations. The significant register expansion means that with AArch64 the maximum register capacity is expanded to 1,984 bits (i.e. 31×64, a 4x increase with respect to ARMv7.).

ARMv8 processors started to dominate the smartphone market soon after their first release in 2011, and nowadays they are widely used in various high-end

[1] The value of t is defined by the implementation parameters.

smartphones (e.g. iPhone, Huawei Mate and Samsung Galaxy series). Since this architecture is used primarily in embedded systems and smartphones, efficient and compact implementations are of special interest.

ARMv8 processor supports powerful 64-bit wise unsigned integer multiplication instructions. Our implementation of modular multiplication uses the AArch64 architecture and makes extensive use of the following multiply instructions:

- `MUL` (unsigned multiplication, low part):
  `MUL X0, X1, X2` computes `X0` $\leftarrow$ (`X1` $\times$ `X2`) mod 2^{64}.
- `UMULH` (unsigned multiplication, high part):
  `UMULH X0, X1, X2` computes `X0` $\leftarrow$ (`X1` $\times$ `X2`)$/2^{64}$.

The two instructions above are required to compute a full 64-bit multiplication of the form 128-bit $\leftarrow 64 \times 64$-bit, namely, the `MUL` instruction computes the lower 64-bit half of the product while `UMULH` computes the higher 64-bit half.

For the addition and subtraction operations, `ADDS` and `SUBS` instructions ensure 64-bit wise results, respectively. The detailed descriptions are as follows:

- `ADDS` (unsigned addition):
  `ADDS X0, X1, X2` computes {`CARRY,X0`} $\leftarrow$ (`X1` + `X2`).
- `SUB` (unsigned subtraction):
  `SUBS X0, X1, X2` computes {`BORROW,X0`} $\leftarrow$ (`X1` − `X2`).

4 Optimized Field Arithmetic Implementation

There is a number of works in the literature that study the ARMv8 instructions to implement multi-precision multiplication or the full Montgomery multiplication for "SIDH friendly" modulus [14, 13, 17]. In [13], Jalali et al. implemented 751-bit and 964-bit finite field multiplication. They utilized the Comba method (i.e. column-wise multiplication) for both cases [9]. In particular, they used 2-level Karatsuba for 964-bit finite field multiplication, which shows 23.9% performance enhancements than conventional Comba method. In [17], Seo et al. optimized the 503-bit finite field multiplication for SIKEp503. They also used the Comba method with 2-level Karatsuba method to enhance the performance of multiplication. Furthermore, they optimized the MAC (Multiplication ACcumulation) routines to avoid the pipeline stalls.

Recently, two novel SIKE protocols (i.e. SIKEp434 and SIKEp610) for NIST Post Quantum Cryptography competition were suggested, which meet NIST security level 1 and 3, respectively [3]. However, previous works do not show the optimized results for both protocols. In this paper, we show the first practical implementations of SIKEp434 and SIKEp610 protocols on 64-bit ARMv8-A processors. In order to achieve high performance, the arithmetic for SIKEp434 and SIKEp610 is optimized to utilize the ARMv8-A ability fully. To describe the multi-precision arithmetic, we used following notations. Let A and B be operands of length m bits each. Each operand is written as $A = (A[n-1], ..., A[1], A[0])$ and $B = (B[n-1], ..., B[1], B[0])$, where $n = \lceil m/w \rceil$ is the number of words to represent operands, m is operand length, and w is the computer word size (i.e. 64-bit). The addition result ($C = A + B$) is represented as $C = (C[n-1], ..., C[1], C[0])$. For the multiplication ($C = A \times B$), the result is represented as $C = (C[2n-1], ..., C[1], C[0])$.

4.1 Finite field addition and subtraction

In the beginning, the finite field addition and subtraction operations need to perform addition and subtraction operations, respectively. Afterward, the intermediate results are reduced, when the carry or borrow bit is set. In order to avoid the timing attack, both reduction routines are performed without conditional statements (i.e. constant timing). Instead, we used the masked modular reduction approach, which always perform regular routines, regardless of the carry or borrow bit. When the carry or borrow bit is set, the mask value is set to $2^{64}-1$. Otherwise, the mask value is set to 0. With the mask value, the modulus is determined whether it is modulus or 0.

For 434-bit addition or subtraction operation, we utilized 14 general purpose registers to store the operands (i.e. $2 \times \lfloor 434/64 \rfloor$) since each operand requires 7 registers. In particular, two limbs of 434-bit modulus are $2^{64}-1$ (i.e. `0xFFFFFFFFFFFFFFFF`). We only set one limb to $2^{64}-1$ and use it twice for computations, which reduces one operand setting overheads.

For 610-bit addition or subtraction operation, we utilized 20 general purpose registers to retain all operands (i.e. $2 \times \lfloor 610/64 \rfloor$) since each operand requires 10 registers. Similarly, three limbs of 610-bit modulus are set to $2^{64}-1$ (i.e. `0xFFFFFFFFFFFFFFFF`). This limb is used three times with only one memory access, which reduces two operand settting overheads.

4.2 Multiplication

In previous works, they used the Comba method (i.e. column-wise method) to improve the multi-precision multiplication. The Comba method performs the partial products in column-wise, which ensures small number of registers for maintaining the intermediate results. In Figure 3, the part of Multiplication ACculmuation (MAC) routine in column-wise method for 64-bit ARMv8 processors is described. The example performs the three partial products ($A[i] \times B[j]$, $A[i+1] \times B[j-1]$, and $A[i+2] \times B[j-2]$) and accumulates them to the intermediate results. In each MAC routine, two multiplication (`MUL_LOW` and `MUL_HIGH`) and three addition operations (`ACC0`, `ACC1`, and `ACC2`) are required. For one limb multiplication, we need three addition operations. For that reason, n-limb multiplication requires $3 \times n^2$ addition operations.

In this work, we target the relatively shorter modulus (i.e. 434-bit) than previous works (i.e. 503-bit or 751-bit). We decide to use the row-wise multiplication, which requires $2n+2$ registers ($n+1$ for operands and $n+1$ for intermediate results), where n, m, and w are $\lfloor m/w \rfloor$, operand length, and word size, respectively. Under 64-bit processor setting, the n is set to 7 for 434-bit ($\lfloor 434/64 \rfloor$). Considering that ARMv8 supports 31 64-bit registers, the required number of

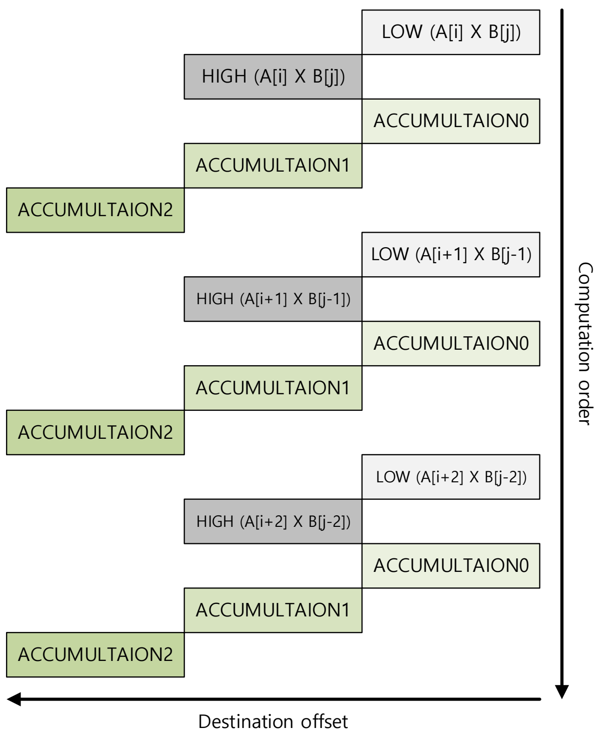

Fig. 3: Part of column-wise multiplication for ARMv8

registers for 434-bit can be retained in the registers. In Figure 4, the part of MAC routine in row-wise method for 64-bit ARMv8 processors is described. The example performs the three partial products ($A[i] \times B[j]$, $A[i] \times B[j+1]$, and $A[i] \times B[j+2]$) and accumulates them to the intermediate results. The number of addition for three partial products in Figure 4 are 8 (i.e. $2 \times (n+1)$ where n is 3.). For the n-limb multiplication, it requires $2 \times n \times (n+1)$ addition operations. The comparison of multiplication methods in terms of the number of addition operations depending on the number of limb are given in Table 1. Compared with the column-wise method (i.e. product-scanning), the row-wise method (i.e. operand-scanning) requires less number of addition operations for accumulation routines. For the 7-limb case (i.e. 434-bit), the row-wise method reduces the number of addition operations by 35 times than the column-wise method. The multiplication is performed in original row-wise multiplication rather than row-wise multiplication with Karatsuba method. The Karatsuba method is also working for 7-limb case but it generates a number of sub-routines to perform and store the intermediate results, which requires additional operations and memory accesses [16].

Table 1: Comparison of multiplication methods, in terms of the number of addition operations depending on the number of limb.

Method	3	4	5	6	7
Operand Scanning	24	40	60	84	112
Product Scanning	27	48	75	108	147

For the 610-bit multiplication, the operands $A = (A[9], \ldots, A[0])$ and $B = (B[9], \ldots, B[0])$ need 20 64-bit registers. Except the operands, we also need registers for intermediate results and temporal storage. Due to the limited number of registers, we only maintain the half number of operands in the registers and load the remaining operands on demand.

We first compute the lower 320-bit multiplication $R_L \leftarrow A[4 \sim 0] \cdot B[4 \sim 0])$ using the row-wise method that requires 25 `MUL`, 25 `UMULH` and 52 addition instructions for accumulating the partial products. Second, we compute the higher 310-bit multiplication $R_H \leftarrow A[9 \sim 5] \cdot B[9 \sim 5]$ similarly. Third, we compute the subtractions and absolute values $|A[4 \sim 0] - A[9 \sim 5]|$ and $|B[4 \sim 0] - B[9 \sim 5]|$ and proceed to the last 310-bit multiplication $R_M \leftarrow |A[4 \sim 0] - A[9 \sim 5]| \cdot |B[4 \sim 0] - B[9 \sim 5]|$. Finally, we obtain the result by performing the accumulation step $R_H \cdot 2^{610} + (R_L + R_H - R_M) \cdot 2^{310} + R_L$. Since the multiplication uses all available registers, 12 callee-saved registers ($X19 \sim X30$) are stored into the stack. The multiplication is also designed to reduce the pipeline stalls. The multiplication and addition/subtraction operations use different instruction group. They can hide each others costs. Based on the above observation, we engineer a multi-precision multiplication to hide the addition costs into the multiplication. At the lowest level, we implement multi-precision multiplication using the row-wise method based on the following multiplication/addition instruction sequence:

```
  ⋮
MUL  X7,X6,X2
ADCS X18,X18,X13
MUL  X8,X6,X3
ADCS X19,X19,X14
MUL  X9,X6,X4
ADCS X20,X20,X15
MUL  X10,X6,X5
ADCS X21,X21,X16
  ⋮
```

We ensure that the destination of `MUL` instruction is not used for the source of following `ADCS` instructions. This approach avoids the pipeline stalls. Second, MUL and `ADCS` instructions are performed one by one to hide the each costs. As will be shown in Section 5, the proposed implementation achieved the high performance (see Table 2).

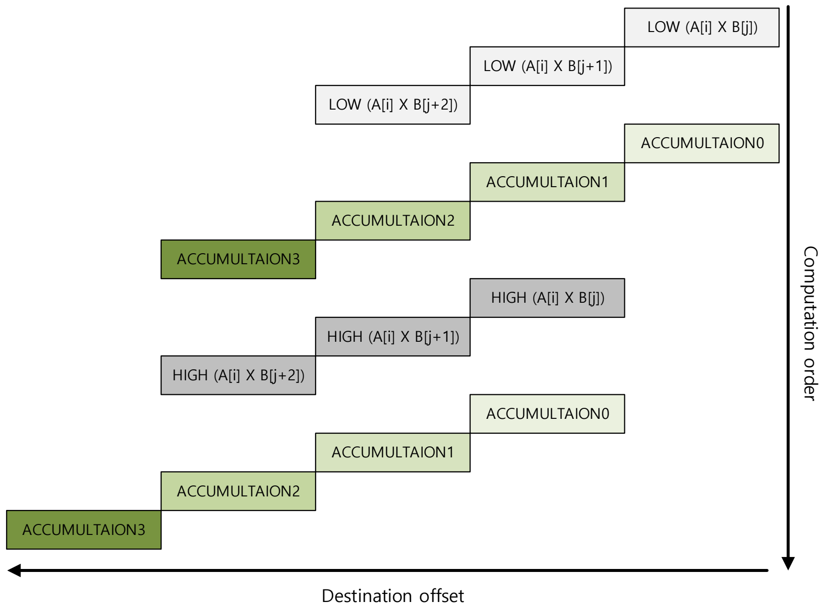

Fig. 4: Part of row-wise multiplication for ARMv8

4.3 Reduction

In this section, we adapt the techniques described in previous sections to implement modular multiplication for the supersingular isogeny-based protocols SIDH and SIKE. Specifically, we target the parameter sets based on the primes p434 and p610 [3].

Multi-precision modular multiplication is the most expensive operation for the implementation of SIKE [15, 10]. In particular, Montgomery multiplication for SIKE can be efficiently exploited and further simplified by taking advantage of so-called "Montgomery-friendly" modulus. The advantage of using Montgomery multiplication for "SIDH-friendly" primes was recently confirmed by Bos and Friedberger [6], who studied and compared different approaches, including Barrett reduction. Recent works by Seo et al also utilized the Montgomery multiplication for SIKEp503 protocols [17].

Based on the observation above, we choose the Montgomery multiplication to implement SIDH-friendly modular arithmetic for SIKEp434 and SIKEp610 protocols. The approach reduces almost half of partial products since the lower part is set to 0. In order to reduce the memory accesses, we keep as many results as possible in the registers. Since the Montgomery multiplication performs the partial products with modulus and quotient (Quotient is intermediate results multiplied by constant m'), we maintained all quotients in the registers and used them directly. The technique reduces the $2 \times (n+1)$ number of memory accesses for $n+1$ load and $n+1$ store operations.

5 Performance Result

In this section, we evaluate the performance of the proposed algorithms for 64-bit ARMv8-A processors. All our implementations were written in assembly language and complied with optimization level `-O3`.

We implemented the multi-precision multiplication algorithm described in Section 4.2 and Montgomery reduction in Section 4.3. We integrated our implementation of the Montgomery multiplication for ARMv8-A into the SIKE round 2 library [3].

Table 2 summarizes the results of different software implementations of the SIKEp434 and SIKEp610 arithmetic on ARMv8-A processor: a 1.536GHz ARM Cortex-A53 processor. Since this is first work for SIKEp434 and SIKEp610 on ARMv8-A processors, we compare the results with the SIKE round 2 reference code. The *unoptimized* reference implementation is written in C using the SIKE round 2 library [3]. In this case, the proposed arithmetic implementations show much higher performance than reference work. In particular, finite field multiplication and inversion operations show performance enhancements by 4.96x and 4.98x, respectively.

Table 3 summarizes the results of different software implementations of the SIKEp434 and SIKEp610 protocols on ARMv8-A processor. Compared with reference work, the proposed implementation is between 3.83 and 3.42 times faster for the computation of the SIKE full protocols. Considering that the target processor is 1.536 GHZ, the SIKEp434 and SIKEp610 requires only 0.084 and 0.30 seconds, respectively.

Compared with the other security levels, the performance depends on the length of modulus. The SIKEp434 shows the highest performance and the SIKEp751 shows the lowest performance as we expected.

Table 2: Comparison of implementations of the SIKEp434 and SIKEp610 arithmetic on ARMv8 Cortex-A53 based processors. Timings are reported in terms of clock cycles.

Implementation	Language	Protocol	Timings [cc]			
			$\mathbb{F}_p$ add	$\mathbb{F}_p$ sub	$\mathbb{F}_p$ mul	$\mathbb{F}_p$ inv
SIKE R2 [3]	C	SIKEp434	172	129	3,110	1,648,372
This work	ASM		71	63	691	380,711
SIKE R2 [3]	C	SIKEp610	257	187	6,599	4,800,694
This work	ASM		100	91	1,329	963,064

Table 3: Comparison of implementations of the SIKE protocols on ARMv8 Cortex-A53 based processors. Timings are reported in terms of clock cycles.

Implementation	Language	Protocol	Timings [cc]	Timings [$cc \times 10^6$]			
			$\mathbb{F}_p$ mul	KeyGen	Encaps	Decaps	Total
SIKE R2 [3]	C	SIKEp434	3,110	114	186	199	499
This work	ASM		691	30	49	52	130
Seo et al. [17]	ASM	SIKEp503	849	38	63	67	168
SIKE R2 [3]	C	SIKEp610	6,599	344	634	615	1,593
This work	ASM		1,329	99	183	183	465
Seo et al. [17]	ASM	SIKEp751	2,450	164	265	284	713

6 Conclusion

This paper presented high-speed implementation of SIKE Round 2 on high-end 64-bit ARMv8 Cortex-A53 processors. A combination of several optimization methods yields very efficient modular multiplications for SIKEp434 and SIKEp610 protocols that are shown, for example, to be approximately 4.96x faster than the normal modular multiplication implementations for “SIDH-friendly” modulus on a 64-bit ARMv8 Cortex-A53 processors. The optimized implementation, which push further the performance of post-quantum supersingular isogeny-based protocols, are 3.42x faster than the previously implementations of SIDHp610 on the same processors. Furthermore, we integrated our fast modular arithmetic implementations, compact prime SIDHp434, and optimal strategy for isogeny computations into Microsoft's SIDH library. A 128-bit full key-exchange execution over optimal prime SIDHp434 is performed in about 0.084 seconds on a 1.536GHz ARMv8 Cortex-A53 processors, which shows the practicality of isogeny based post-quantum cryptography over mobile devices.

7 Acknowledgement

This work of Hwajeong Seo was supported by Institute for Information communications Technology Planning Evaluation (IITP) grant funded by the Korea government(MSIT) (<Q|Crypton>, No.2019-0-00033, Study on Quantum Security Evaluation of Cryptography based on Computational Quantum Complexity).

This work of Reza Azarderakhsh and Amir Jalali is supported in parts by NSF CNS-1801341 and NIST-60NANB16D246.

References

1. G. Adj, D. Cervantes-Vázquez, J. Chi-Domínguez, A. Menezes, and F. Rodríguez-Henríquez. On the cost of computing isogenies between supersingular elliptic

curves. In *Selected Areas in Cryptography - SAC 2018 - 25th International Conference*, pages 322–343, 2018.

2. ARM Limited. ARM architecture reference manual ARMv8, for ARMv8-A architecture profile. `https://static.docs.arm.com/ddi0487/ca/DDI0487C_a_armv8_arm.pdf`, 2013–2017.
3. R. Azarderakhsh, M. Campagna, C. Costello, L. D. Feo, B. Hess, A. Jalali, D. Jao, B. Koziel, B. LaMacchia, P. Longa, M. Naehrig, G. Pereira, J. Renes, V. Soukharev, and D. Urbanik. Supersingular Isogeny Key Encapsulation – Submission to the NIST's post-quantum cryptography standardization process, round 2, 2019. Available at `https://csrc.nist.gov/projects/post-quantum-cryptography/round-2-submissions/SIKE.zip`.
4. R. Azarderakhsh, M. Campagna, C. Costello, L. D. Feo, B. Hess, A. Jalali, D. Jao, B. Koziel, B. LaMacchia, P. Longa, M. Naehrig, J. Renes, V. Soukharev, and D. Urbanik. Supersingular Isogeny Key Encapsulation – Submission to the NIST's post-quantum cryptography standardization process, 2017. Available at `https://csrc.nist.gov/CSRC/media/Projects/Post-Quantum-Cryptography/documents/round-1/submissions/SIKE.zip`.
5. R. Azarderakhsh, M. Campagna, C. Costello, L. D. Feo, B. Hess, A. Jalali, D. Jao, B. Koziel, B. LaMacchia, P. Longa, M. Naehrig, J. Renes, V. Soukharev, and D. Urbanik. Supersingular Isogeny Key Encapsulation – Submission to the NIST's post-quantum cryptography standardization process, 2017. Available at `https://csrc.nist.gov/CSRC/media/Projects/Post-Quantum-Cryptography/documents/round-1/submissions/SIKE.zip`.
6. J. W. Bos and S. Friedberger. Fast arithmetic modulo $2^x p^y \pm 1$. In *IEEE Symposium on Computer Arithmetic (ARITH'17)*, pages 148–155. IEEE, 2017.
7. D. X. Charles, K. E. Lauter, and E. Z. Goren. Cryptographic hash functions from expander graphs. *J. Cryptology*, 22(1):93–113, 2009.
8. L. Chen, L. Chen, S. Jordan, Y.-K. Liu, D. Moody, R. Peralta, R. Perlner, and D. Smith-Tone. *Report on post-quantum cryptography*. US Department of Commerce, National Institute of Standards and Technology, 2016.
9. P. G. Comba. Exponentiation cryptosystems on the IBM PC. *IBM systems journal*, 29(4):526–538, 1990.
10. C. Costello, P. Longa, and M. Naehrig. Efficient algorithms for supersingular isogeny Diffie-Hellman. In M. Robshaw and J. Katz, editors, *Advances in Cryptology - CRYPTO 2016*, volume 9814 of *Lecture Notes in Computer Science*, pages 572–601. Springer, 2016.
11. S. D. Galbraith, C. Petit, B. Shani, and Y. B. Ti. On the security of supersingular isogeny cryptosystems. In *Advances in Cryptology - ASIACRYPT 2016 - 22nd International Conference on the Theory and Application of Cryptology and Information Security,*, pages 63–91, 2016.
12. D. Hofheinz, K. Hövelmanns, and E. Kiltz. A modular analysis of the fujisaki-okamoto transformation. In *Theory of Cryptography - 15th International Conference, TCC 2017,*, pages 341–371, 2017.
13. A. Jalali, R. Azarderakhsh, M. M. Kermani, and D. Jao. Supersingular isogeny Diffie-Hellman key exchange on 64-bit ARM. *IEEE Transactions on Dependable and Secure Computing*, 2017.
14. A. Jalali, R. Azarderakhsh, and M. Mozaffari-Kermani. Efficient post-quantum undeniable signature on 64-bit ARM. In *International Conference on Selected Areas in Cryptography*, pages 281–298. Springer, 2017.

15. D. Jao and L. D. Feo. Towards quantum-resistant cryptosystems from supersingular elliptic curve isogenies. In B. Yang, editor, *Post-Quantum Cryptography (PQCrypto 2011)*, volume 7071 of *Lecture Notes in Computer Science*, pages 19–34. Springer, 2011.
16. P. L. Montgomery. Five, six, and seven-term Karatsuba-like formulae. *IEEE Transactions on Computers*, 54(3):362–369, 2005.
17. H. Seo, Z. Liu, P. Longa, and Z. Hu. SIDH on ARM: faster modular multiplications for faster post-quantum supersingular isogeny key exchange. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 1–20, 2018.
18. The National Institute of Standards and Technology (NIST). Post-quantum cryptography standardization, 2017–2018. `https://csrc.nist.gov/projects/post-quantum-cryptography/post-quantum-cryptography-standardization`.