High-Performance FPGA Accelerator for SIKE

Rami El Khatib, Reza Azarderakhsh, *Member, IEEE*,
and Mehran Mozaffari-Kermani, *Senior Member, IEEE*

Abstract—In this article, we provide improvements for the architecture of Supersingular Isogeny Key Encapsulation (SIKE), a post-quantum cryptography candidate. We develop a new highly optimized Montgomery multiplication algorithm and architecture for prime fields. The multiplier occupies less area and provide better timing results than the state-of-the-art. We also provide improvements to the scheduling of SIKE in our program ROM. We implement SIKE for all Round 3 NIST security levels (SIKEp434 for NIST security level 1, SIKEp503 for NIST security level 2, SIKEp610 for NIST security level 3, and SIKEp751 for NIST security level 5) on Xilinx Artix 7 and Xilinx Virtex 7 FPGAs. Our best implementation (NIST security level 1) runs 38 percent faster and occupies 30 percent less hardware resources in comparison to the leading counterpart available in the literature and implementations for other security levels achieved similar improvement.

Index Terms—Hardware architectures, isogeny-based cryptography, Montgomery multiplication, post-quantum cryptography, SIKE

1 Introduction

Post-quantum cryptography (PQC) centers on identifying and understanding new mathematical techniques upon which cryptography can be built that are both resistant against quantum attacks and feasible to be implemented on today's widely used computerized devices. In a seminal paper [1], Peter Shor showed that both RSA and ECC would be easily broken by employing a quantum computer. The five main classes of quantum-hard problems are as follows [2]: code-based cryptography, lattice-based cryptography, hash-based cryptography, multivariate cryptography, and isogeny-based cryptography. The second round of the NIST PQC standardization process features a greater emphasis on evaluating the performance of candidates. NIST completed Round 2 evaluation and Supersingular Isogeny Key Encapsulation (SIKE) stayed as an alternate candidate in Round 3 with a strong hope of being standardized in Round 4 [2].

When considering quantum-safe alternatives to ECC, isogeny-based cryptography appears as an attractive replacement. The security of isogeny-based cryptosystems such as SIKE scheme is based on the problem of computing isogenies between elliptic curves. Improving the performance of isogeny-based cryptography is critical to ensuring that it survives into subsequent rounds of standardization. Notably, the SIKE [3] scheme features the smallest public key sizes [4], [5] of known quantum-safe public key exchange algorithms. Small public key sizes are extremely advantageous in many different scenarios as it reduces the communication overhead and storage necessary for secure communications. The only concern is the performance of SIKE towards which this work is another step forward.

SIKE offers four different security levels, as shown in Table 1, with higher security levels utilizing larger primes. The prime is used as the modulus for modular addition and modular multiplication which together form a prime field $\mathbb{F}_p$. The prime field is then used to build the isogenies through a pyramid scheme discussed in Section 2. SIKE's prime has a special form $2^{e_A} \cdot 3^{e_B} - 1$ where the least significant e_A bits are all 1s. This form can be exploited in Montgomery multiplication [6], which is a method for modular multiplication. It is well known that the main drawback of SIKE over other PQC candidates is the high cost of modular multiplication which makes it few orders of magnitude slower than other PQC schemes.

Recently researchers were able to improve the computation time of SIKE by slightly over one order of magnitude [7], [8], reducing the total time to under 20 milliseconds while adding protection against active attacks. In this work, we show that there is still room for improvement of intensive lower level computations. This paper is another step forward in this direction which reduces the computation time to less than 10 milliseconds and cuts the occupied number of hardware resources considerably when implemented in FPGA. The goal of this paper is to develop efficient and high-performance hardware architectures for SIKE. The contributions of this paper is itemized in the following:

1.1 Our Contributions

- We develop a highly optimized Montgomery multiplication algorithm and architecture that further utilizes the special form of SIKE primes. We experimented various configurations for our high-radix design to find the best choice for area-time trade-offs.
- We improve the scheduler mechanism provided in [9] and utilized it in our design. Which resulted in a reduced number of clock cycles.
- We implement SIKE for NIST Round 2 primes; SIKEp434, SIKEp503, SIKEp610, and SIKEp751 with the developed Montgomery multiplier architecture.

- *Rami El Khatib and Reza Azarderakhsh are with the Department of Computer and Electrical Engineering and Computer Science, Florida Atlantic University, Boca Raton, FL 33431 USA. E-mail: {relkhatib2015@fau.edu, razarderakhsh}@fau.edu.*
- *Mehran Mozaffari-Kermani is with the Computer Science and Engineering Department, University of South Florida, Tampa, FL 33620 USA. E-mail: mehran2@usf.edu.*

Manuscript received 23 July 2020; revised 20 Mar. 2021; accepted 1 May 2021. Date of publication 10 May 2021; date of current version 10 May 2022.
(Corresponding author: Reza Azarderakhsh.)
Recommended for acceptance by J. C. Hoe.
Digital Object Identifier no. 10.1109/TC.2021.3078691

TABLE 1
SIKE Primes for Post-Quantum Cryptography Based on NIST Round 2 Standardization Process [3]

Security Level	Prime Form	Public Key Size (Bytes)	Shared Key Size (Bits)
NIST level 1	$p_{434} = 2^{216}3^{137} - 1$	330	128
NIST level 2	$p_{503} = 2^{250}3^{159} - 1$	378	192
NIST level 3	$p_{610} = 2^{305}3^{192} - 1$	462	192
NIST level 5	$p_{751} = 2^{372}3^{239} - 1$	564	256

- We evaluate time and area performance of the proposed hardware architecture benchmarked on FPGA and compare with counterparts.

The organization of the paper is as follows. In Section 2, we give a literature review of SIKE. In Section 3, we discuss the algorithm and architecture of our highly optimized Montgomery multiplication. In Section 4, we discuss the improved scheduler. In Section 5, we propose our SIKE implementation and compare our results with counterparts available in the literature. Finally, in Section 6, we give our final thoughts and discuss future work.

2 Preliminaries: SIKE Protocol

In this section, we provide a brief overview of the SIKE protocol. SIKE mainly requires two operations: Isogeny and Shake256. The latter is part of the NIST standardized hashing algorithm SHA-3 [10]. Isogeny operations are done over Montgomery curve [11], [12] using the efficient projective isogeny formulas [3] for better performance. We point the reader to [3] for a detailed overview of SIKE.

2.1 SIKE Operations

A prime p is chosen of the form $2^{e_A}3^{e_B} - 1$ where $2^{e_A} \approx 3^{e_B}$ (Check Table 1 for standardized primes). For public parameters, we have a starting curve E_0, two points P_A and Q_A of order 2^{e_A} and two points P_B and Q_B of order 3^{e_B} (standardized parameters are in SIKE specs [3]). Each pair of points with the same order must be chosen such that there is Weil pairing so that $P + [s]Q$ also has an order of ℓ^e (the order of P and Q) for any $s < \ell^e$.

Key Generation. In key generation, Bob chooses a random secret key $s_B \in [0, 3^{e_B})$ and computes the isogenous elliptic curve $E_B$using the isogeny ϕ_B with kernel $\langle P_B + [s_B]Q_B \rangle$. The elliptic curve E_B along with $\phi_B(P_A)$ and $\phi_B(Q_A)$ make up Bob's public key pk_B.

Key Encapsulation. In key encapsulation, Alice chooses a secret message $m \in [0, 2^{ss\_size})$ (where ss_size is the shared key size in Table 1) and hashes $\{m, pk_B\}$ using Shake256 to generate her secret key r of size 2^{e_A} bits. She can then compute her emphemeral public key $\{E_A, \phi_A(P_B), \phi_A(Q_B)\}$ using the isogeny $\phi_A : E_0 \rightarrow E_B \cong E_0/\langle P_A + [r]Q_A \rangle$. She also generates a key to encrypt the message m by first computing the elliptic curve E_{AB} under the isogeny $\phi_{AB} : E_B \rightarrow E_{AB} \cong E_B/\langle \phi_B(P_A) + [r]\phi_B(Q_A) \rangle$. Then she computes the j-invariant $j(E_{AB})$ and hashes it with Shake256 to the same size of the message. She encrypts the message m by XORing it with the key to generate c. She shares the ciphertext $ct = \{pk_A, c\}$ publicly and, finally, generates the shared secret ss_A of size ss_size by hashing $\{m, ct\}$ with Shake256.

Key Decapsulation. In key decapsulation, Bob first computes the key used to encrypt c by first computing the elliptic curve E_{BA} under the isogeny $\phi_{BA} : E_A \rightarrow E_{BA} \cong E_A/\langle \phi_A(P_B) + [s_B]\phi_A(Q_A) \rangle$ using Alice's public key pk_A. If he receives Alice's correct ciphertext, E_{BA} should be isomorphic to E_{AB}, a.k.a. equal j-invariant. Therefore, he can compute the key by hashing the j-invariant $j(E_{BA})$ with Shake256. The message m' can then be recovered by XORing c with the key. He can recover Alice's secret key r' by hashing $\{m', pk_B\}$ and then generate Alice's public key $pk'_A = \{E'_A, \phi'_A(P_B), \phi'_A(Q_B)\}$ under the isogeny $\phi'_A = E_0 \rightarrow E'_A \cong E_0/\langle P_A + [r']Q_A \rangle$. He checks that Alice's public key he computed is equal to Alice's actual public key. If they are equal, he outputs the correct shared secret ss_B by hashing $\{m, pk_A, c\}$.

Isogeny Computations. The pyramid in Fig. 1 shows the breakdown of isogeny computations. To compute the Isogeny $E/\langle P + [s]Q \rangle$, the kernel point $R = P + [s]Q$ needs to be computed first using a three point ladder algorithm. The fastest

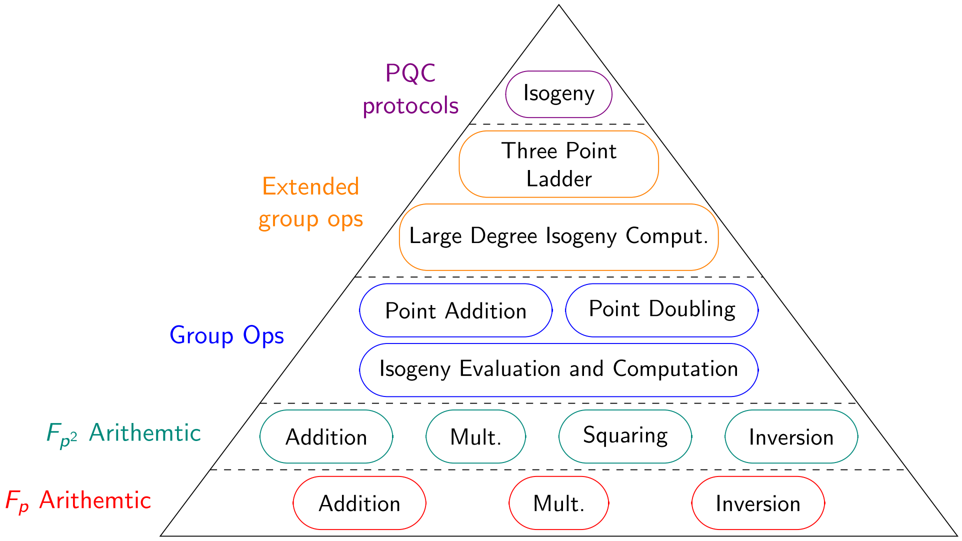

Fig. 1. Breakdown of isogeny computations [8].

TABLE 2
Optimal Modular Adder Parameters

Prime	$a \pm b$	$a \pm b \mp p$
SIKEp434	$L = 23, H = 3$	$L = 21, H = 3$
SIKEp503	$L = 20, H = 3$	$L = 26, H = 3$
SIKEp610	$L = 27, H = 3$	$L = 20, H = 3$
SIKEp751	$L = 25, H = 3$	$L = 20, H = 3$

algorithm is in [13] which requires one point addition and one point doubling per bit of the scalar s. For the large degree isogeny computation $E/\langle R\rangle$, we break it down into point multiplications and small isogeny evaluations and computations following a specific strategy. When the kernel is of order 3^{e_B}, we perform point tripling and 3-isogenies. When the kernel is of order 2^{e_A}, we perform point quadrupling and 4-isogenies as their formulas are more efficient than point doubling and 2-isogenies. Note that for SIKEp610, since e_A is odd, one 2-isogeny is performed at the beginning. The elliptic curve group operations are built using $\mathbb{F}_{p^2}$ arithmetic which in turn is built using $\mathbb{F}_p$ arithmetic.

3 Proposed Efficient Lower Level Arithmetic Operations

In this section, we are going to discuss our low level arithmetic operations. For the modular adder, we reused the modular adder in the leading hardware candidate of SIKE [14], which utilizes the adder in [15], with more efficient parameters. The parameter L indicates length of carry chain before going to the next level compaction while the parameter H indicates the maximum level of compaction. It is near impossible to obtain the optimal parameters for the adder as place and route greatly changes for different parameters. However, going beyond $H = 3$ will add a significant routing delay and roughly $L = \sqrt{p}$ is a good starting point to test. We tested all L around $\sqrt{p}$ for $H = 1, 2, 3$ for $a \pm b$ first and then for $a \pm b \mp p$. Table 2 shows optimal parameters for the modular adder we are using.

For the modular multiplication ($a \times b \bmod p$), Montgomery multiplication is a fast modular multiplication algorithm that transforms the expensive division by p into a cheap division by power of 2 which is a simple shift right in software or hardware. Montgomery multiplication has been used for high-performance ECC applications extensively such as in [16], [17], [18], [19], [20]. Word-by-word Montgomery multiplication algorithms were proposed in [21], [22]. Some Montgomery multiplication architecture for SIKE can be seen in [14], [23].

Finely Integrated Operand Scanning (FIOS) Montgomery multiplication algorithm is a word-by-word algorithm first proposed in [21]. The original implementation was suitable for software. In [23], the FIOS algorithm was re-purposed for hardware implementation suitable for SIKE primes. We had two issues using that implementation directly in SIKE. The first issue is that it was not fully interleaved (a.k.a unused blocks in the multiplier unit can't be used before the multiplication is complete). Since SIKE has a lot of modular multiplication computation that can be parallelized, the extra cycles from non-interleaving slows down SIKE. The issue can be easily resolved by pushing each chunk of the multiplicand (b for example) into the corresponding processing element as soon as it is needed instead of pushing all the chunks in one go. This technique will have no impact on the total number of cycles. The second issue is that when plugged in SIKE, the operating frequency is around 200MHz. This frequency makes the implementation non-competitive.

Algorithm 1: Optimized Montgomery Multiplication for SIKE Primes

```
Input  : $p = 2^{e_A} \cdot 3^{e_B} - 1 < 2^K$, $R = 2^K$, $w$, $s$,
         $K = w \cdot s$, $s_A = \lfloor 2^{e_A}/w \rfloor$, $a, b < 2p - 1$
Output: MontMult$(a, b)$
$T \leftarrow 0$
for $i \leftarrow 0$ to $s - 1$ do
    $(C, S) \leftarrow T[0] + a[i] \cdot b[0]$                    } PE Initial
    $m \leftarrow S$                                              }
    for $j \leftarrow 1$ to $s_A - 1$ do
        $(C, S) \leftarrow T[j] + a[i] \cdot b[j] + C$            } $s_A$-Mult
        $T[j - 1] \leftarrow S$                                   }
    $U[s_A] \leftarrow m + m \cdot p[s_A]$                        } $s_B$-Red0
    for $j \leftarrow s_A + 1$ to $s - 1$ do
        $U[j] \leftarrow m \cdot p[j]$                            } $s_B$-Red
    for $j \leftarrow s_A$ to $s - 1$ do
        $(C, S) \leftarrow T[j] + U[j] + a[i] \cdot b[j] + C$     } $s_B$-Mult
        $T[j - 1] \leftarrow S$                                   }
    if $p < 2^K - 2$ then                                         } PE Final
        $(C, S) \leftarrow C$                                     }
        $T[s - 1] \leftarrow S$                                   }
    else                                                          }
        $(C, S) \leftarrow T[s] + C$                              }
        $T[s - 1] \leftarrow S$                                   }
        $T[s] \leftarrow C$                                       }
return $T$                                                        }
```

3.1 Proposed Montgomery Multiplication Algorithm

We further optimized the Montgomery multiplication algorithm in [23] to minimize the number of operations in the critical path and the total number of operations used specifically for SIKE primes. Our optimized algorithm is provided in Algorithm 1. The algorithm performs the following s (number of words) times: an initial step, $s-1$ multiplication-reduction steps and a final step.

The initial step begins by adding the first result chunk $T[0]$ with $a[i] \times p[0]$. The least significant word S is used to compute the quotient m and the carry C is propagated to the first multiplication-reduction step. Because of the special form of SIKE primes where $p[0]$ is all 1s for any word $w < e_A$, $p' = -p^{-1} \bmod 2^w = 1$. This leads to $m = S \cdot p' \bmod 2^w = S$. Finally, a second carry C_r is propagated to the first multiplication-reduction step. $(C_r, S) = S + m \cdot p[0] = m + \ m \cdot p[0] = (m, 0) \Rightarrow C_r = m$. Our first change here is to keep the carries separate instead of merging them together by adding them.

Each of the multiplication-reduction steps consists of addition of current result chunk $T[j]$, two parallel multiplications ($a[i] \cdot b[j]$ and $m \cdot p[j]$), and the carry from the previous step. The least significant word is stored in the previous result

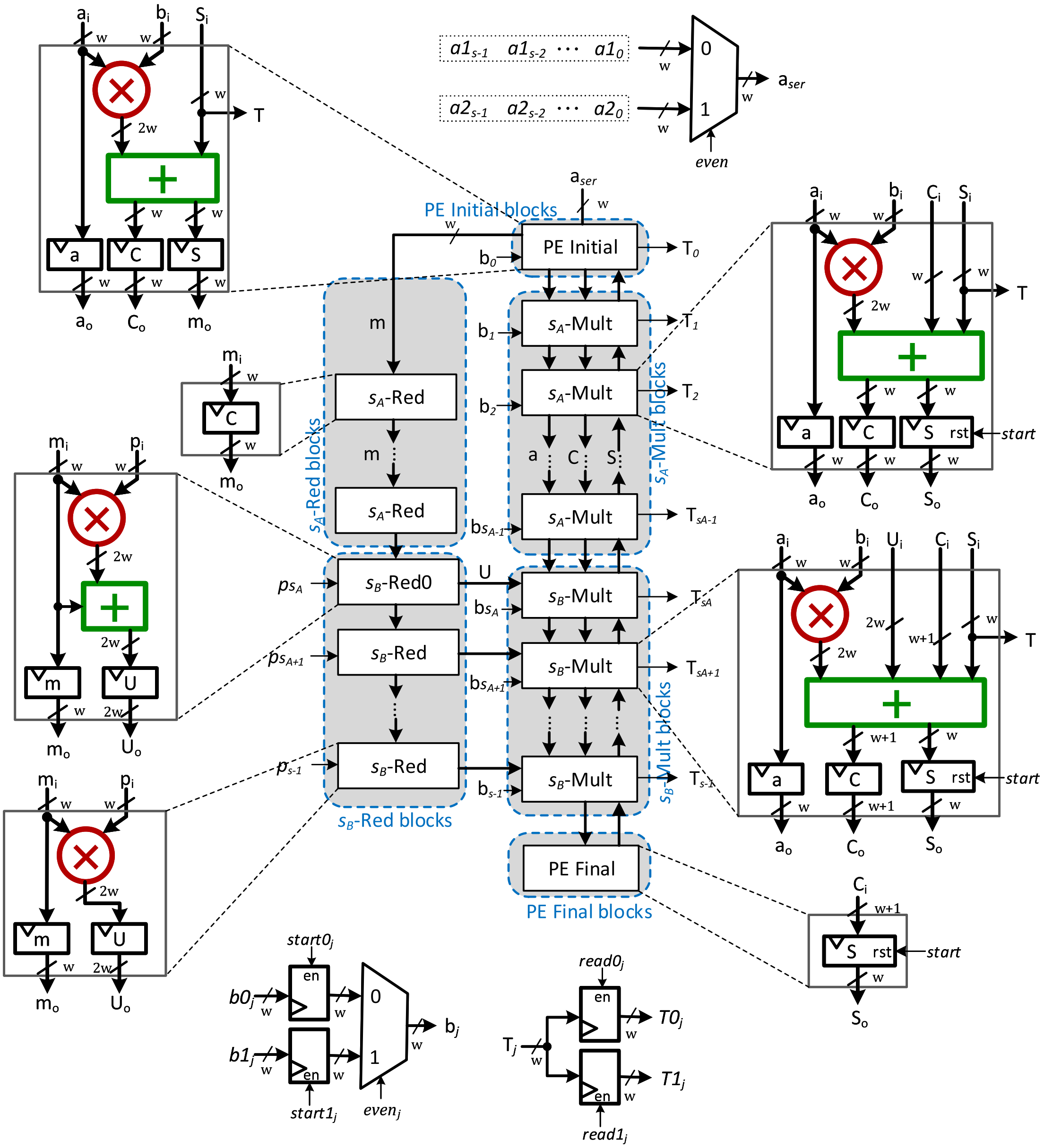

Fig. 2. Proposed Montgomery multiplication architecture.

chunk $T[j-1]$ and the carry is propagated to the next step. Our approach was to split the multiplication-reductions steps into two parts. In the first part where $1 \le j < s_A = \lfloor 2^{e_A}/w \rfloor$ (s_A-Mult), we notice that all the bits of $p[j]$ are 1. The reduction operation $m \times p[j]$ can be skipped completely as $(C_r, S) = C_r + m \times p[j] = (m, 0)$. Therefore, $T[j-1]$ is independent of the reduction operation and we are always propagating m to the next step. In the second part where $s_A \le j < s$ (s_B-Mult and s_B-Red), all operations of the multiplication-reduction step are performed. In the first reduction operation (s_B-Red0), we add the carry $C_r = m$ to the reduction operation $m \times p[s_A]$ which will be added to the first multiplication operation in s_B-Mult and merged with the carry C in subsequent steps. This means that in subsequent reduction operations only $m \times p[j]$ is performed without adding C_r. Note that the carry C is 1 bit larger ($w + 1$ bits total) after the merging.

In the final step, the carry C of the last multiplication-reduction step is pushed into the final result chunk $T[s-1]$. If the radix $R = 2^K = 2^{s \cdot w}$ is chosen such that $p < 2^{K-2}$, then $C < 2^w$ can fit in the result chunk. Otherwise, if $p = 2^{K-1}$, then an additional 1-bit register $T[s]$ is used to process the extra bit of C.

The changes made to the algorithm cut $s_A - 1$ multiplications and $s_A - 2$ additions. Furthermore, s_B-Red operations can be computed ahead of time which will reduce the critical path delay in our architecture.

3.2 Proposed Architecture for Montgomery Multiplication

Fig. 2 shows our proposed architecture. Our design can perform two multiplications in parallel and each block in our design is pipelined and performs one operation in the algorithm. The first block PE initial computes the first multiplication carry C and the quotient m, which is also the reduction carry C_r for Montgomery multiplication with SIKE primes. m is pushed to the reduction path (s_A-Red→ s_B-Red0→ s_B-Red) where the reduction operations in the algorithm are performed. The first multiplication

TABLE 3
Breakdown of Our Proposed Montgomery Multiplication Architecture Compared to Previous Design (Dual Multiplier)

Block	Total Blocks	Operation	Critical Path	Arithmetic Operations	Total Arithmetic Operations
El Khatib *et al.* [23] twice					
PE initial	1	$T[0] + a[i] \cdot b[0]$	$M_w + A_{2w}$	$M_w + A_{2w}$	$M_w + A_{2w}$
Mult-Red	$s-1$	$T[i] + a[i] \cdot b[j] + m \cdot p[j] + C$	$M_w + 2A_{2w}$	$2M_w + 3A_{2w}$	$(2s-2)M_w + (3s-3)A_{2w}$
PE final	1	C	0	0	0
Full design	-	-	$M_w + 2A_{2w}$	-	$(2s-1)M_w + (3s-2)A_{2w}$
Proposed Design					
PE initial	1	$T[0] + a[i] \cdot b[0]$	$M_w + A_{2w}$	$M_w + A_{2w}$	$M_w + A_{2w}$
s_A-Red	$s_A - 2$	C	0	0	0
s_A-Mult	$s_A - 1$	$T[i] + a[i] \cdot b[j] + C$	$M_w + A_{2w}$	$M_w + 2A_{2w}$	$(s_A - 1)M_w + (2s_A - 2)A_{2w}$
s_B-Red0	1	$m + m \cdot p[j]$	$M_w + A_{2w}$	$M_w + A_{2w}$	$M_w + A_{2w}$
s_B-Red	$s_B - 1$	$m \cdot p[j]$	M_w	M_w	$(s_B - 1)M_w$
s_B-Mult	s_B	$T[i] + U[j] + a[i] \cdot b[j] + C$	$M_w + A_{2w}$	$M_w + 3A_{2w}$	$(s_B)M_w + (3s_B)A_{2w}$
PE final	1	C	0	0	0
Full design	-	-	$M_w + A_{2w}$	-	$(s + s_B)M_W + (2s + s_B)A_{2w}$

carry C is pushed to the multiplication path (s_A-Mult → s_B-Mult) where the multiplication operations in the algorithm are performed and the result chunks are collected. Finally, PE final receives the final carry from the multiplication path and is used to process the final result chunk. Inside the main path (PE initial→Multiplication path→PE final), carry C is propagated forward while S is propagated backward as S is stored in previous result chunk $T[j-1]$ in the algorithm.

$a1$ and $a2$, the first operands for the dual multiplier, are pushed serially in odd and even cycles, respectively, into PE initial and then propagated to the next block in the multiplication path. The second operands for the dual multiplier, $b1$ and $b2$, are pushed directly to their respective block. However, to achieve interleaving and increase throughput, $b1$ and $b2$ are pushed in the first s cycles with one cycle delay for the next word. On odd cycles, the odd blocks $(1, 3, 5, \ldots)$ compute chunks for the first pair of operands ($a_1$and b_1) while the even blocks $(2, 4, 6, \ldots)$ compute chunks for the second pair of operands (a_2 and b_2). On even cycles, the blocks switch places where now the odd blocks work on the second pair of operands and the even blocks work on the first pair of operands. A reset is required to the register S that stores the result chunks during the first s cycles. The final result is collected word-by-word over s cycles after $2s$ cycles have passed since the start of the multiplier.

In the reduction path, s_A-Red is completely eliminated in our algorithm and therefore m is simply propagated to s_B-Red0 after a certain delay. To shorten the critical path, s_B-Red blocks are processed one cycle in advance before the result is pushed into their corresponding s_A-Mult block.

Table 3 gives a breakdown of the total number of blocks required as well as the critical path and the number of arithmetic operations used in comparison to [23] (used twice for dual-multiplication). The critical path is shortened by one addition and the design requires $s_A - 1$ less multiplications and $s_A - 2$ less additions.

3.3 Implementation and Results

The FPGAs we are using in our SIKE implementation are the Xilinx Virtex-7 and Xilinx Artix-7. The DSP unit in this series of FPGA can perform fast multiply-and-add ($a \times b + c$) or 3-input addition ($a + b + c$). Chaining the DSPs allow for complex arithmetic operations with a small additional delay per DSP. Furthermore, DSPs support dual input for one of the multiplicand ($a \times b_1 + c$ or $a \times b_2 + c$) by exploiting the pre-adder. This allows us to design a dual multiplier while fully utilizing the DSP unit. Table 4 shows how to utilize a maximum of 2 DSPs per block. In [23], the reduction and multiplication operations are not separated and therefore require 3 chained DSPs to compute them and more DSPs for a dual-multiplier design. Thus, our design requires less number of DSPs in the critical path and less total DSPs.

A few additional optimizations can be exploited by the DSP. The registers to store the second operands $b0$ and $b1$ are used directly in the DSP. The DSP can select whether to add 0 or one of the operands in the addition step. This is used to replace the reset signal of the registers that hold the result chunks S. Another optimization that can be utilized is to store a and b going to the multiplication of each block in the DSP's register. This will add one extra cycle but greatly shorten the critical path. The start control signals and the even control signal for $b1$ and $b2$ are stored one cycle in advance in the DSP's control registers for improved performance. The registers used to store C and S are stored in the fabric outside the DSP as this will give the best performance.

Table 5 shows number of DSPs used and timing results of our implementations for each of the SIKE primes. Our design requires less DSP, has a higher frequency, but require more clock cycles in comparison to [14]. However, the higher

TABLE 4
DSP Breakdown of Our Proposed Montgomery Multiplication Architecture (Dual Multiplier)

Block	DSP 1	DSP 2	Total DSPs
PE initial	$T[0] + a[i] \times b[0]$	-	**1**
s_A-Red	-	-	-
s_A-Mult	$a[i] \times b[j]$	$DSP1 + T[i] + C$	$2(s_A - 1)$
s_B-Red0	$m + m \cdot p[j]$	-	**1**
s_B-Red	$m \cdot p[j]$	-	$s_B - 1$
s_B-Mult	$U[j] + a[i] \times b[j]$	$DSP1 + T[i] + C$	$2s_B$
PE final	-	-	**0**
Full design	-	-	$2s + s_B - 1$

TABLE 5
Montgomery Multiplication DSP and Timing Analysis

Reference	#	Freq	Latency (cc)		Latency (ns)	
	DSP	(MHz)	Mult.	Interleave	Mult.	Interleave
			SIKEp434			
Liu *et al.* [24]*	36	236	66	54	280	229
This work	**65**	**294.0**	**81**	**52**	**276**	**177**
			SIKEp503			
Koziel *et al.* [14]**	88	171.2	70	49	409	286
Liu *et al.* [24]*	64	213	66	54	310	254
This work	**75**	**294.0**	**93**	**60**	**316**	**204**
			SIKEp610			
Liu *et al.* [24]*	81	191	66	54	346	283
This work	**90**	**294.0**	**111**	**72**	**378**	**245**
			SIKEp751			
Koziel *et al.* [14]**	128	167.4	100	69	597	412
Liu *et al.* [24]*	144	161	66	54	410	335
This work	**113**	**294.0**	**138**	**90**	**469**	**306**

$*$ *LUT usage is 5-6 × more than our design.*
$**$ *Interleave cycle is odd number which adds an addition cycle.*

frequency dominates the increased cycle count and the overall total time to perform an operation is lower. In [24], a huge part of the computation is moved from DSP to fabric. Their LUT usage for SIKEp434 is 6724 in comparison to our LUT usage of 1,157. In addition, the design is not very scalable as SIKEp751 uses more DSP and $5\times$ LUT in comparison to our design. We reserve further comment until the design is plugged in SIKE.

4 Scheduling Prime Field Operations

The most expensive operations for performing the isogeny computation are the `double-and-add` to compute the three-point-ladder, `double/triple`, `get-isogeny` and `evaluate-isogeny` to compute the l-degree isogeny, and finally $\mathbb{F}_p$ `inversion` for encoding and decoding the data (generating public key, getting the elliptic curve equation, and computing j-invariant). The inversion formulas has been optimized in [3] for each prime and since it is Fermatt-Based, there is little room for improving the scheduling as the algorithm is mostly sequentional. For the three-point-ladder and the l-degree isogeny computation, a good scheduler can exploit the available resources to reduce the time to compute them. In this section, we look at the scheduler used by Farzam *et al.* [9] and try to improve on it.

4.1 Scheduling Operations

In [9], the authors implement an efficient scheduler by utilizing an optimization programming language (OPL) instead of implementing the optimization using a script as was done in [14] (and its previous iterations in [7], [8], [25]). The main advantage of using OPL is that the optimization techniques are heavily scrutinized and will almost always outperform a hard-coded optimizer without a lot of investment in it. To implement the efficient scheduler using OPL, the authors start by generating a dependency graph after expanding all operations to their $\mathbb{F}_p$ equivalent. For example, Fig. 3 shows the dependency graph of a subroutine that performs an $\mathbb{F}_p^2$ multiplication followed by an $\mathbb{F}_p^2$ addition.

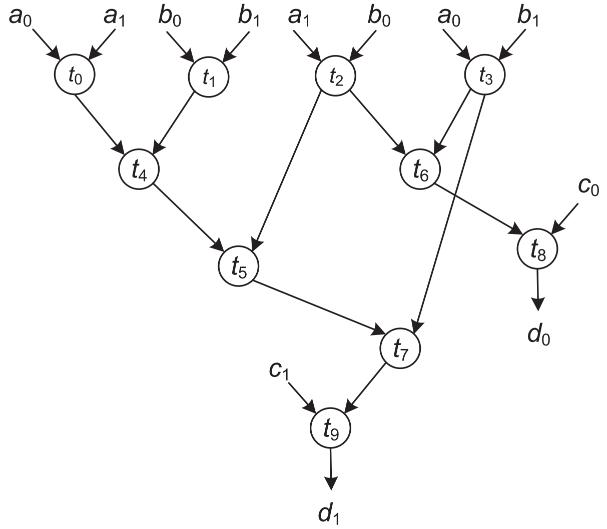

Fig. 3. Dependency graph showing $\mathbb{F}_p^2$ multiplication [9] followed by $\mathbb{F}_p^2$ addition ($d = a \cdot b + c$).

In addition to the data, a constraint set must be provided to the scheduler depending on the available resources. We have noticed one issue and one inefficient utilization of a resource with the constraint the authors provided in [9]. Before discussing the issues, we will briefly mention the constraints which are also the same constraints used in the architecture of [14]

- No simulatenous RAM read, RAM write, or both.
- RAM read is 2 CCs.
- RAM write is 1 CCs
- The field adder/subtractor latency is 2 CCs. Consecutive addition/subtraction are allowed as the second cycle (reduction) utilizes a different unit from the first cycle (addition).

For the field multiplier, our multiplier is designed differently but the constraints are exactly the same as in [14]. If the first multiplication is performed on an even cycle, then the second multiplication is performed on an odd cycle, then the third multiplication is performed on an even cycle and so on. We have noticed an issue in the authors' constraints as can be seen in their paper's Fig. 5 where they scheduled the third multiplication in their multiplier in an odd cycle which means that the third multiplication is overwriting the second multiplication. We are not sure how they got correct results with such a scheduling without any modification to the field multiplier which is not mentioned in the paper.

Each multiplier also has 2 stages; the interleave stage and writing stage. During the interleave stage, the multiplier is locked and doesn't accept new multiplication. Once the interleave stage is complete, a new multiplication can be processed while simulatenously the current multiplier writes the result chunk-by-chunk into a register. The authors here delayed the next multiplication until after the RAM read and interleave cycles. That is actually not necessary. The multiplication can be scheduled 2 cycles before the interleave stage of the multiplier finishes since it takes two cycles to start the multiplication due to the RAM read latency. One downside to the design in [14] is that the interleave stage is an odd number which means that the multiplication need to be scheduled one cycle after that. In our multiplier design, the interleave stage requires an even number of clock cycles and therefore no additional cycle is required.

TABLE 6
Comparison of Major Subroutine for NIST Level 1 (SIKEp434) Between Our Design and [14]

Subroutine	Work	Multipliers						Improvement (%)					
		2	4	6	8	10	12	2	4	6	8	10	12
Ladder Step	[14]	838	569	429	376	371	371	2	19	14	5	9	9
	our	824	460	368	356	338	338						
QUAD	[14]	955	718	659	659	659	659	3	4	5	5	5	5
	our	930	692	627	626	626	626						
GET 4 ISO	[14]	247	207	207	207	207	207	0	5	5	5	5	5
	our	247	196	196	196	196	196						
EVAL 4 ISOx1	[14]	830	580	539	539	539	539	25	23	32	34	37	37
	our	619	445	368	354	337	337						
EVAL 4 ISOx2	[14]	1,219	706	661	635	640	626	2	11	30	34	38	38
	our	1,191	625	463	418	397	388						
EVAL 4 ISOx3	[14]	1,795	974	789	725	713	705	2	4	19	27	35	9
	our	1,763	931	638	527	464	644						
EVAL 4 ISOx4	[14]	2,377	1,258	914	850	805	770	2	4	9	22	28	26
	our	2,335	1,203	834	661	581	566						
EVAL 4 ISOx5	[14]	2,959	1,563	1,127	1,002	927	879	2	3	7	18	22	23
	our	2,908	1,509	1,047	817	720	674						
EVAL 4 ISOx6	[14]	3,539	1,841	1,313	1,158	1,023	1,020	2	3	7	17	18	19
	our	3,484	1,781	1,219	956	834	823						
EVAL 4 ISOx7	[14]	4,092	2,157	1,623	1,483	1,441	1,430	1	3	12	24	32	33
	our	4,059	2,101	1,433	1,121	986	965						
EVAL 4 ISOx8	[14]	4,674	2,447	1,754	1,618	1,542	1,512	0	4	8	20	25	28
	our	4,665	2,357	1,614	1,299	1,154	1,091						
EVAL 4 ISOx9	[14]	5,248	2,754	1,965	1,705	1,648	1,600	0	3	7	16	21	23
	our	5,236	2,684	1,820	1,429	1,297	1,225						
EVAL 4 ISOx10	[14]	5,830	3,014	2,137	1,869	1,787	1,709	1	1	7	15	22	23
	our	5,785	2,973	1,997	1,595	1,390	1,320						
TRIPLE	[14]	950	669	588	584	560	560	2	6	7	8	7	7
	our	927	627	548	536	523	523						
GET 3 ISO	[14]	460	354	277	277	277	277	18	21	8	8	8	8
	our	375	281	254	254	254	254						
EVAL 3 ISOx1	[14]	531	415	331	331	331	331	3	5	4	4	4	4
	our	513	393	319	319	319	319						
EVAL 3 ISOx2	[14]	907	559	452	453	390	392	3	15	16	19	7	9
	our	879	476	379	366	361	355						
EVAL 3 ISOx3	[14]	1,324	753	564	514	496	469	2	10	14	17	19	16
	our	1,295	680	486	429	402	392						
EVAL 3 ISOx4	[14]	1,751	943	691	604	546	544	2	6	9	15	16	19
	our	1,711	886	628	515	460	441						
EVAL 3 ISOx5	[14]	2,172	1,149	843	718	651	619	2	5	8	16	15	15
	our	2,128	1,096	775	601	555	525						
EVAL 3 ISOx6	[14]	2,598	1,364	974	821	754	703	2	4	7	12	14	13
	our	2,543	1,308	902	719	645	611						
EVAL 3 ISOx7	[14]	3,000	1,607	1,179	1,031	1,061	1,030	1	5	11	19	29	32
	our	2,959	1,520	1,055	839	749	705						
EVAL 3 ISOx8	[14]	3,424	1,809	1,317	1,171	1,118	1,159	1	3	9	20	24	31
	our	3,381	1,746	1,204	931	853	794						
EVAL 4 ISOx9	[14]	3,840	2,010	1,469	1,257	1,177	1,208	1	3	10	14	22	27
	our	3,801	1,954	1,317	1,083	915	879						
EVAL 4 ISOx10	[14]	4,264	2,221	1,577	1,419	1,267	1,290	1	2	6	16	17	25
	our	4,225	2,172	1,479	1,187	1,052	968						

Once a dependency graph and constraint are set, the OPL model is ready to be fed into the OPL scheduler. We feed the OPL model to the constraint programming (CP) engine of IBM's CPLEX Studio and use the result to generate the program ROM.

4.2 Scheduler Results

We provide our results in Table 6 for NIST level 1 (SIKE p434) with a multiplier that has an interleave cost of 52 cycles and multiplication cost of 81 cycles as was obtained in Section 3. The table also compares our current results to the results

TABLE 7
Comparison of Scheduling Major Subroutines Used in Computing the Isogeny With 69 Interleave CC and 100 Multiplication CC

Subroutine	Work	Multipliers						Improvement (%)					
		2	4	6	8	10	12	2	4	6	8	10	12
Ladder Step	[14]	1,099	613	523	479	479	476	0	0	3	4	10	10
	[9]	1,097	607	469	453	441	440						
	our	1,095	605	455	433	399	395						
Quad	[14]	1,236	887	827	821	821	821	1	2	5	5	5	5
	[9]	1,232	854	780	780	780	780						
	our	1,220	840	741	741	741	741						
GET 4 ISO	[14]	360	261	261	261	261	261	1	6	6	6	6	6
	[9]	322	248	248	248	248	248						
	our	320	234	234	234	234	234						
EVAL 4 ISOx1	[14]	876	596	498	471	457	457	0	1	5	5	11	10
	[9]	822	562	466	455	441	440						
	our	818	554	441	431	394	394						
EVAL 4 ISOx3	[14]	2,369	1,260	908	814	794	790	0	0	2	7	23	29
	[9]	2,362	1,242	846	731	722	725						
	our	2,358	1,238	832	679	553	516						
EVAL 4 ISOx5	[14]	3,916	2,047	1,452	1,280	1,276	1,270	0	0	0	12	27	36
	[9]	3,902	2,012	1,382	1,199	1,180	1,174						
	our	3,898	2,008	1,380	1,053	858	754						
EVAL 4 ISOx7	[14]	5,462	2,834	1,962	1,762	1,762	1,750	0	0	0	14	28	38
	[9]	5,443	2,783	1,885	1,676	1,650	1,643						
	our	5,438	2,780	1,881	1,448	1,188	1,024						
EVAL 4 ISOx9	[14]	7,007	3,627	2,565	2,243	2,241	2,230	0	0	1	15	32	40
	[9]	6,983	3,552	2,408	2,163	2,170	2,146						
	our	6,978	3,552	2,378	1,831	1,481	1,288						
EVAL 4 ISOx11	[14]	8,557	4,407	3,094	2,779	2,760	2,740	0	0	0	17	32	40
	[9]	8,522	4,322	2,941	2,689	2,668	2,640						
	our	8,520	4,322	2,928	2,226	1,826	1,581						
Triple	[14]	1,224	812	706	693	687	687	0	1	3	2	6	6
	[9]	1,222	783	683	664	654	654						
	our	1,217	774	663	649	617	617						
GET 3 ISO	[14]	579	439	375	374	373	373	9	11	10	10	10	10
	[9]	518	377	326	326	326	326						
	our	472	336	293	293	293	293						
EVAL 3 ISOx1	[14]	676	501	418	418	418	418	1	2	7	7	7	7
	[9]	668	492	405	405	405	405						
	our	659	484	376	376	376	376						
EVAL 3 ISOx3	[14]	1,740	975	734	639	613	613	0	1	4	5	10	16
	[9]	1,732	903	646	562	542	533						
	our	1,728	895	622	532	487	448						
EVAL 3 ISOx5	[14]	2,867	1,497	1,100	981	930	903	0	1	0	11	22	30
	[9]	2,853	1,463	1,032	869	843	839						
	our	2,848	1,454	1,028	774	654	584						
EVAL 3 ISOx7	[14]	3,988	2,066	1,494	1,329	1,263	1,253	0	0	1	13	23	33
	[9]	3,973	2,024	1,406	1,216	1,174	1,179						
	our	3,968	2,014	1,395	1,053	899	788						
EVAL 3 ISOx9	[14]	5,116	2,635	1,902	1,697	1,653	1,613	0	0	1	14	26	35
	[9]	5,093	2,581	1,769	1,566	1,521	1,525						
	our	5,088	2,575	1,752	1,342	1,121	992						
EVAL 3 ISOx11	[14]	6,237	3,212	2,353	2,060	2,000	1,982	0	0	1	16	28	36
	[9]	6,213	3,144	2,172	1,932	1,888	1,859						
	our	6,208	3,140	2,159	1,616	1,350	1,185						

obtained by using the scheduler in [14] with the same multiplier costs. We observed 5-32 percent improvement across the board with an overall improvement of 10 percent. Similar percentage improvement were observed across all SIKE primes.

Since our multiplier operates at a higher clock frequency at the cost of more clock cycles, a direct comparison between our scheduler and the one provided in [9] is not possible. However, we ran our scheduler with an interleave cost of 69

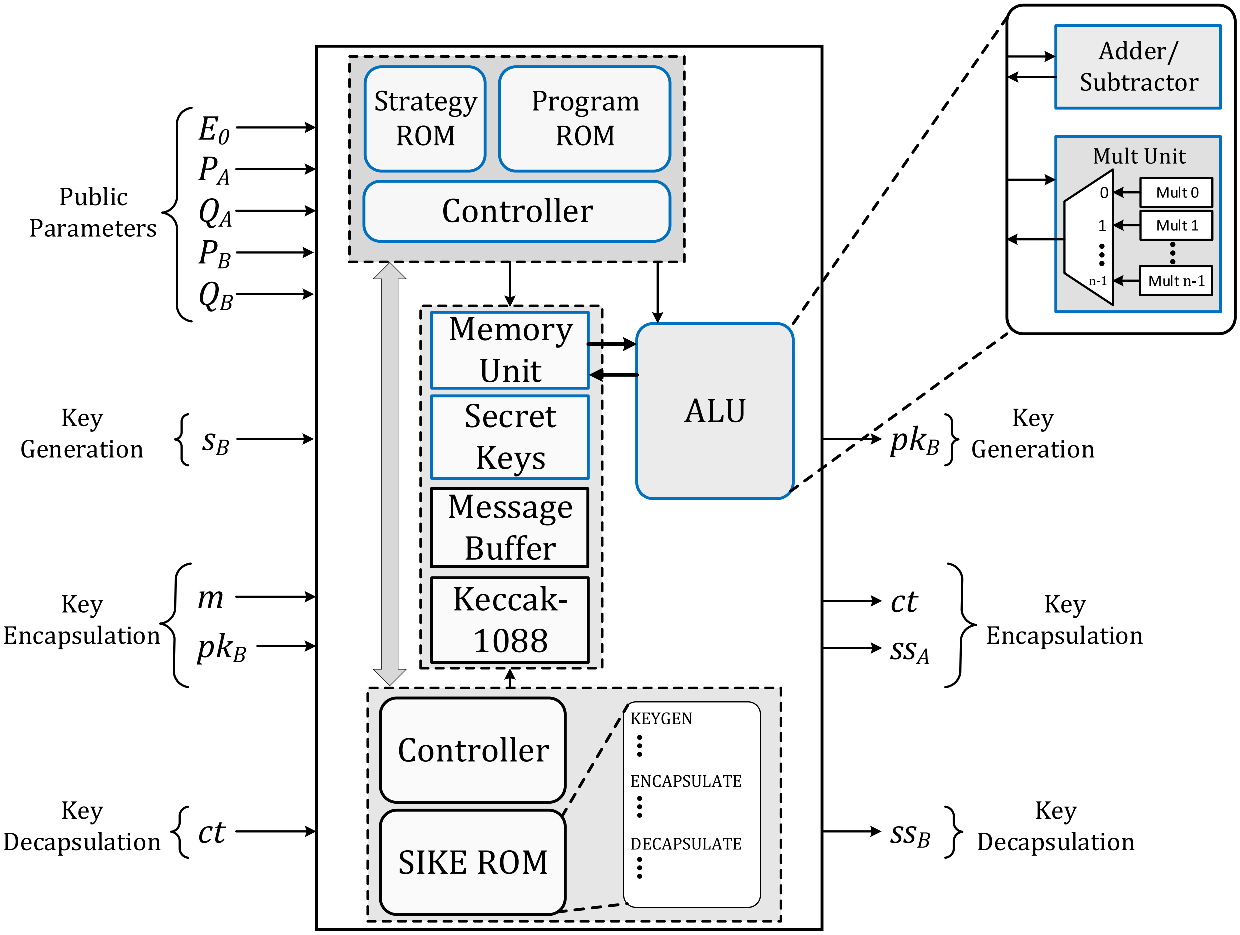

Fig. 4. Proposed hardware architecture for SIKE protocol.

and multiplication cost of 100 for SIKEp751. The results can be observed in Table 7. At 2 multipliers, a very minor improvement can be observed. However, when going to 8 multipliers, our improvements reaches 17 percent in some places. The number increases at 12 multiplier reaching 40 percent improvement in some areas.

5 FPGA Implementations of SIKE

The implementation is performed in Xilinx Vivado 2019.2 for Xilinx Virtex-7 FPGA xc7vx690tffg1157-3 and Xilinx Artix-7 xc7a200tfbg676-2 to be able to fairly compare our proposed scheme with the ones available in the literature. The Virtex-7 FPGA includes 108,300 Slices (most with four LUTs and eight flip-flops), 3,600 DSP blocks and 1,470 36kb BlockRAMs. Each DSP slice contains a pre-adder, a 25×18 multiplier, an adder, and an accumulator. The Artix-7 FPGA includes similar resources but less of available resources for each. Our design is based on the design in the leading literature [14] with a modified ALU based on Section 3 and an improved program generated from the scheduler from Section 4.

5.1 Proposed SIKE Architecture

The architecture for SIKE used in our design is illustrated in Fig. 4 which is composed of field arithmetic logic unit (ALU), main SIKE controller/ROM, program and strategy controller/ROM, memory unit, message buffer to hold Alice's message and ciphertext and Bob's message, secret key buffer to hold Alice's secret key and Bob's secret key, and hash unit based on Keccak-1088.

The ALU is the main core and performs operations in $\mathbb{F}_p$ while interacting with the memory unit. $\mathbb{F}_{p^2}$ arithmetic is done using $\mathbb{F}_p$ architectures. For instance, a $\mathbb{F}_{p^2}$ multiplication requires three $\mathbb{F}_p$ multiplications, two $\mathbb{F}_p$ additions and three $\mathbb{F}_p$ subtractions, whereas a $\mathbb{F}_{p^2}$ squaring requires only two $\mathbb{F}_p$ multiplications, two $\mathbb{F}_p$ additions and one $\mathbb{F}_p$ subtraction. The ALU consists of a Multiplication unit and adder/subtractor unit. The adder/subtractor unit computes modular addition and subtraction $(\bmod 2p)$ as well as modular reduction $(\bmod p)$ over the specified primes for SIKE. The multiplication unit consists of n Dual-Multipliers based on the design proposed in Section 3. Since the multiplication unit is the critical resource, we use as many Dual-Multipliers as is allowed for parallelization while trying to minimize Time-Area cost. The cycle counts for our design is reported in Table 8.

TABLE 8
Number of Clock Cycles (in 10^6 CC) for the Key Encapsulation Mechanism (KEM) in Our Design

Prime	# Mults	Keygen	Key Encap	Key Decap	Total (E+D)
SIKEp434	6	0.541	0.974	1.019	1.994
SIKEp503	6	0.729	1.291	1.363	2.654
SIKEp503	6	1.056	2.144	2.112	4.256
SIKEp751	8	1.343	2.554	2.683	5.237

TABLE 9
Area and Timing Results of SIKE Implementation in Xilinx Virtex-7

		Area					Time			Area × Time
Reference	# Mults	# FFs	# LUTs	# Slices	# DSPs	# BRAMs	Freq (MHz)	Latency ($cc \times 10^6$)	Total time (ms)	AT $\times 10^{-3}$
				SIKEp434						
Massolino *et al.*[26] (Fast)	-	-	-	7,408	162	38.0	152.2	-	24.3	180
Koziel *et al.* [14]	6	23,819	21,059	8,121	240	26.5	168.4	1.91	11.3	92
This work	**6**	**18,271**	**12,818**	**5,527**	**195**	**32.0**	**249.6**	**1.99**	**8.0**	**44**
				SIKEp503						
Koziel *et al.* [8]*	6	30,031	24,499	10,298	192	27	177	5.97	33.7	347
Koziel *et al.* [25]*	6	26,659	19,882	8,918	192	40	181.4	3.80	20.9	186
Koziel *et al.* [7]*	6	24,908	18,820	7,491	192	43.5	202.1	3.34	16.5	124
Massolino *et al.*[26] (Fast)	-	-	-	7,408	162	38.0	152.2	-	28.7	212
Koziel *et al.* [14]	6	27,609	23,746	8,907	264	33.5	165.9	2.35	14.1	126
This work	**6**	**19,935**	**13,963**	**6,163**	**225**	**34.0**	**243.7**	**2.65**	**10.9**	**67**
				SIKEp610						
Massolino *et al.*[26] (Fast)	-	-	-	7,408	162	38.0	152.2	-	51.8	384
Koziel *et al.* [14]	6	33,297	28,217	10,675	312	39.5	165.8	3.59	21.6	231
This work	**6**	**26,757**	**16,226**	**7,461**	**270**	**38.5**	**239.0**	**4.26**	**17.8**	**133**
				SIKEp751						
SIKE Team [3]**	8	51,914	44,822	16,756	376	56.5	198.0	6.60	33.4	560
Massolino *et al.*[26] (Fast)	-	-	-	7,408	162	38.0	152.2	-	60.8	450
Koziel *et al.* [14]	8	50,079	39,953	15,834	512	43.5	163.1	4.55	27.8	440
Farzam *et al.* [9]**	8	-	-	15336	512	45	160.9	3.877	24.10	369
This work	**8**	**39,339**	**20,207**	**11,136**	**452**	**41.5**	**232.7**	**5.24**	**22.5**	**251**

$*$ *SIDH.*
$**$ *SIKE Round 1 Parameters.*

The memory unit is implemented using BlockRAM resources from the FPGA device. The memory unit, secret key buffer, message buffer, and the hash unit can share data with each other and can be accessed directly 64-bit at a time. The SIKE controller/ROM includes main routines (fixed sequence of instructions) for key generation, key encapsulation, and key decapsulation. On the other hand, The strategy and program controller/ROM includes hand-optimized routines for all the operations required for computing an isogeny (three-point ladder and large-degree isogeny). The program ROM includes the new subroutines discussed in Section 4. The ROM units, similar to the memory unit, are implemented using the BlockRam resources. Our design requires 32 BlockRAMs for SIKEp434.

The sizes for various component of the SIKE architecture are different based on the required security level. For the whole operation, first we pre-load public parameters into the Memory unit. For the secret key and message, Random values are generated in the host CPU since they have negligible impact on performance. Following the SIKE protocol discussed in Section 2.1, key encapsulation and decapsulation are performed and ss_A and ss_B are generated.

5.2 Implementation Results and Comparison

The proposed SIKE architectures for all NIST security levels were implemented and tested using Xilinx Vivado 2019.2 and all the results were obtained after place-and-route. We report area, timing and area-time trade-off (number of slices×time in ms) results of the design in Table 9 for Virtex-7 and Table 10 for Artix-7. For the best performance, we chose 3 Dual-Multipliers (6 multipliers total) for SIKEp434, SIKEp503 and SIKEp610 and 4 Dual-Multipliers for SIKEp751. We tested the functionality of the design using known answers tests (KATs) available in SIKE submission to NIST.

We compare our architecture results to the previous leading one [14] as well as the Software-Hardware co-design [26] (fast implementation only) and some of the previous Supersingular Isogeny Diffie-Hellman (SIDH) implementations. In addition, we compare our results with [9]. However, they used the old public parameters from Round 1 where the three-point-ladder operation for Alice Round 1 (Alice's public key isogeny) can be heavily optimized since $x_{Q_1} = 0$. The total latency is the summation of key encapsulation and key decapsulation as key generation can be done offline. As one can see, for NIST level 1 security (SIKEp434) in Virtex-7, our design requires 5,458 Slices (17,557 flip flops, 12,999 LUTs), 195 DSPs, and 32 BlockRAMs. It also runs 249.6 MHz and performs the whole SIKE protocol in 8.0 ms. The drop in frequency in comparison to the Montgomery multiplier in Table 5 is caused by the strategy and program controller. Our design is smaller (except for the BlockRAMs) and faster with area-time trade-off being about 92 percent improved in comparison to the leading counterpart [14]. For the remaining security levels in Virtex-7, a similar improvement can be observed. It is to be noted that the design in [26] is one design for all SIKE security levels. In addition, the design targets smaller area/lower performance device so a direct comparison is not fair. As for Artix-7, we can observe that the results are better across the board.

The improvements made in the design makes SIKE a feasible option for small embedded devices. Note that SIKE

TABLE 10
Area and Timing Results of SIKE Implementation in Xilinx Artix-7

Reference	# Mults	Area					Time			Area × Time
		# FFs	# LUTs	# Slices	# DSPs	# BRAMs	Freq (MHz)	Latency ($cc \times 10^6$)	Total time (ms)	AT $\times 10^{-3}$
					SIKEp434					
Koziel *et al.* [14]	6	24,328	21,946	8,006	240	26.5	132.2	1.91	14.4	115
This work	**6**	**17,557**	**12,999**	**5,458**	**195**	**32.0**	**184.8**	**2.04**	**11.0**	**60**
					SIKEp503					
Koziel *et al.* [14]	6	27,759	24,610	9,186	264	33.5	129.9	2.35	18.1	166
This work	**6**	**19,952**	**13,552**	**5,985**	**225**	**34.0**	**172.3**	**2.71**	**15.7**	**94**
					SIKEp610					
Koziel *et al.* [14]	6	33,198	29,447	10,843	312	39.5	125.3	3.59	28.6	310
This work	**6**	**25,004**	**16,502**	**7,525**	**270**	**38.5**	**168.7**	**4.26**	**25.2**	**190**
					SIKEp751					
Koziel *et al.* [14]	8	49,982	40,792	15,794	512	43.5	127.0	4.55	35.8	565
This work	**8**	**38,950**	**20,154**	**11,114**	**452**	**41.5**	**155.0**	**5.24**	**33.8**	**375**

already offers smallest key sizes which reduces communication overhead in comparison to the other PQC submissions. Although all of our computations and implementations in this paper are secure (based on [14]) and constant-time, it is worth mentioning that this work mainly focuses on the high-performance implementations of the isogeny-based candidate SIKE in FPGA and investigating side-channel analysis attacks will be in our future work.

6 Conclusion

Post-quantum crypto accelerator hardware cores offer chipmakers an easy-to-integrate technology-independent solution, offering various NIST recommended security levels. In this paper, we optimized the Montgomery multiplication algorithm and architecture targeting SIKE primes. We also improved the scheduler for SIKE subroutines. We also presented FPGA implementations of supersingular isogeny key encapsulation (SIKE) for all NIST Round 2 security levels. The designs are the fastest FPGA implementations of SIKE over large prime characteristic fields for various NIST security levels. More specifically, our design utilizes 36 percent less hardware area and is 12-20 percent faster than the leading FPGA implementations. For NIST level 1, our proposed hardware accelerator performs the SIKE protocol in 8.8 ms. We verified our architectures by using the Known Answer Tests (KATs) from the SIKE submission and our code will be available online for further improvements and evaluations.

Minimizing public key sizes are critical for reducing transmission and storage requirements for internet applications as well as IoTs. Our future work will involve implementing the key compression mechanism and benchmarking the whole design with compressed keys for various security level required by NIST.

Acknowledgments

This work was supported in part by the U.S. National Science Foundation under Grant CNS-1801341 and in part by the U.S. National Institute of Standards and Technology under Grant 60NANB16D246.

References

[1] P. W. Shor, "Algorithms for quantum computation: Discrete logarithms and factoring," in *Proc. 35th Annu. Symp. Foundations Comput. Sci.*, 1994, pp. 124–134.

[2] The National Institute of Standards and Technology, "Post-quantum cryptography standardization," 2017. [Online]. Available: https://csrc.nist.gov/projects/post-quantum-cryptography/post-quantum-cryptography-standardization

[3] R. Azarderakhsh *et al.*, "Supersingular isogeny key encapsulation," 2019. [Online]. Available: https://sike.org/

[4] R. Azarderakhsh, D. Jao, K. Kalach, B. Koziel, and C. Leonardi, "Key compression for isogeny-based cryptosystems," in *Proc. 3rd ACM Int. Workshop ASIA Public-Key Cryptogr.*, 2016, pp. 1–10.

[5] C. Costello, D. Jao, P. Longa, M. Naehrig, J. Renes, and D. Urbanik, "Efficient compression of SIDH public keys," in *Proc. Annu. Int. Conf. Theory Appl. Cryptogr. Techn.*, 2017, pp. 679–706.

[6] P. L. Montgomery, "Modular multiplication without trial division," *Math. Comput.*, vol. 44, no. 170, pp. 519–521, 1985.

[7] B. Koziel, R. Azarderakhsh, and M. Mozaffari-Kermani, "A high-performance and scalable hardware architecture for isogeny-based cryptography," *IEEE Trans. Comput.*, vol. 67, pp. 1594–1609, Nov. 2018.

[8] B. Koziel, R. Azarderakhsh, M. Mozaffari-Kermani, and D. Jao, "Post-quantum cryptography on FPGA based on isogenies on elliptic curves," *IEEE Trans. Circuits Syst. I: Regular Papers*, vol. 64, pp. 86–99, Jan. 2017.

[9] M.-H. Farzam, S. Bayat-Sarmadi, and H. Mosanaei-Boorani, "Implementation of supersingular isogeny-based Diffie-Hellman and key encapsulationusing an efficient scheduling," *IEEE Trans. Circuits Syst. I: Regular Papers*, vol. 67, no. 12, pp. 4895–4903, Dec. 2020.

[10] The National Institute of Standards and Technology, "SHA-3 standard: Permutation-based hash and extendable-output functions," Inf. Technol. Lab, Comput. Secur. Resour. Center, Nat. Inst. Standards Technol, Gaithersburg, MD, USA, Tech. Rep., TR-FIPS.202, 2015.

[11] L. De Feo, D. Jao, and J. Plût, "Towards quantum-resistant cryptosystems from supersingular elliptic curve isogenies," *J.Math. Cryptol.*, vol. 8, pp. 209–247, 2014.

[12] P. L. Montgomery, "Speeding the pollard and elliptic curve methods of factorization," *Math. Comput.*, vol. 48, pp. 243–264, 1987.

[13] A. Faz-Hernández, J. López, E. Ochoa-Jiménez, and F. Rodríguez-Henríquez, "A faster software implementation of the supersingular isogeny Diffie-Hellman key exchange protocol," *IEEE Trans. Comput.*, vol. 67, no. 11, pp. 1622–1636, Nov. 2018.

[14] B. Koziel, A. Ackie, R. El Khatib, R. Azarderakhsh, and M. M. Kermani, "SIKE'd up: Fast hardware architectures for supersingular isogeny key encapsulation," *IEEE Trans. Circuits Syst. I: Regular Papers*, vol. 67, no. 12, pp. 4842–4854, Dec. 2020.

[15] T. B. Preußer, M. Zabel, and R. G. Spallek, "Accelerating computations on FPGA carry chains by operand compaction," in *Proc. IEEE 20th Symp. Comput. Arith.*, 2011, pp. 95–102.

[16] A. Mrabet *et al.*, "High-performance elliptic curve cryptography by using the CIOS method for modular multiplication," in International *Proc. Conf. Risks Secur. Internet Syst.*, 2016, pp. 185–198.

[17] H. Alrimeih and D. Rakhmatov, "Fast and flexible hardware support for ECC over multiple standard prime fields," *IEEE Trans. Very Large Scale Integration Syst.*, vol. 22, no. 12, pp. 2661–2674, Dec. 2014.

[18] H. Eberle, N. Gura, S. C. Shantz, V. Gupta, L. Rarick, and S. Sundaram, "A public-key cryptographic processor for RSA and ECC," in *Proc. 15th IEEE Int. Conf. Appl.-Specific Syst., Architectures Process.*, 2004, pp. 98–110.

[19] M. Imran, M. Rashid, A. R. Jafri, andM. Kashif, "Throughput/area optimised pipelined architecture for elliptic curve crypto processor," *IET Comput. Digit. Techn.*, vol. 13, no. 5, pp. 361–368, 2019.

[20] M. Imran, S. Pagliarini, and M. Rashid, "An area aware accelerator for elliptic curve point multiplication," in *Proc. 27th IEEE Int. Conf. Electron., Circuits Syst.*, 2020, pp. 1–4.

[21] C. KayaKoc, T. Acar, and B. S. Kaliski, "Analyzing and comparing montgomery multiplication algorithms," *IEEE Micro*, vol. 16, no. 3, pp. 26–33, Jun. 1996.

[22] H. Orup, "Simplifying quotient determination in high-radix modular multiplication," in *Proc. 12th Symp. Comput. Arith.*, 1995, pp. 193–199.

[23] R. El Khatib, R. Azarderakhsh, and M. Mozaffari-Kermani, "Optimized algorithms and architectures for montgomery multiplication for post-quantum cryptography," in *Proc. Int. Conf. Cryptology Netw. Secur.*, 2019, pp. 83–98.

[24] W. Liu, Z. Ni, J. Ni, C. Rafferty, and M. O'Neill, "High performance modular multiplication for SIDH," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 39, no. 10, pp. 3118–3122, Oct. 2020.

[25] B. Koziel, R. Azarderakhsh, and M. Mozaffari-Kermani, "Fast hardware architectures for supersingular isogeny diffie-hellman key exchange on FPGA," in *Proc. Int. Conf. Cryptol. India*, 2016, pp. 191–206.

[26] P. M. C. Massolino, P. Longa, J. Renes, and L. Batina, "A compact and scalable hardware/software co-design of SIKE," *IACR Trans. Cryptographic Hardware Embedded Syst.*, vol. 2020, pp. 245–271, 2020.

Rami El Khatib received the bachelor' s degree in electrical and computer engineering from the American University of Beirut, the MSc degree with a focus on implementations of post-quantum cryptography from Florida Atlantic University, where he is currently working toward the PhD degree in computer engineering. He has authored or coauthored several published papers in the areas of cryptography and hardware engineering. His research interests include diverse background in cryptography, programming, and mathematics.

Reza Azarderakhsh (Member, IEEE) received the PhD degree in electrical and computer engineering from Western University in 2011. He was the recipient of the NSERC postdoctoral research fellowship while working with the Center for Applied Cryptographic Research and the Department of Combinatorics and Optimization, University of Waterloo. He is currently an associate professor with the Department of Electrical and Computer Engineering, Florida Atlantic University. His research interests include finite field and its applications, elliptic curve cryptography, pairing-based cryptography, lattice-based cryptography, and post-quantum cryptography. He is an associate editor for the *IEEE Transactions on Circuits and Systems*.

Mehran Mozaffari-Kermani (Senior Member, IEEE)) received the BSc degree from the University of Tehran, Iran, and the MESc and PhD degrees from the University of Western Ontario, London, Canada, in 2007 and 2011, respectively. In 2012, he joined the Electrical Engineering Department, Princeton University, NJ, as an NSERC postdoctoral research fellow. From 2013 to 2017, he was an assistant professor with the Rochester Institute of Technology, and starting 2017 he joined the Department of Computer Science and Engineering, University of South Florida, where he is currently an associate professor. He is currently an associate editor for the *IEEE Transactions on Very Large Scale Integration Systems*, the *ACM Transactions on Embedded Computing Systems*, and the *IEEE Transactions on Circuits and Systems* - Part I: Regular Papers. He has been the guest editor for the *IEEE Transactions on Dependable and Secure Computing*.