Highly Optimized Montgomery Multiplier for SIKE Primes on FPGA

Rami Elkhatib
CEECS Department
Florida Atlantic University
Boca Raton, FL
relkhatib2015@fau.edu

Reza Azarderakhsh
CEECS Department
Florida Atlantic University
Boca Raton, FL
razarderakhsh@fau.edu

Mehran Mozaffari-Kermani
Dept. of CSE.
University of South Florida
Tampa, FL
mehran2@usf.edu

***Abstract*—New primes were proposed for Supersingular Isogeny Key Encapsulation (SIKE) in NIST standardization process of Round 2 after further cryptanalysis research showed that the security levels of the initial primes chosen were overestimated [1], [2]. In this paper, we develop a highly optimized $\mathbb{F}_p$ Montgomery multiplication algorithm and architecture that further utilizes the special form of SIKE prime compared to previous implementations available in the literature. We then implement SIKE for all Round 2 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 Virtex 7 using the proposed multiplier. Our best implementation (NIST security level 1) runs 29% faster and occupies 30% less hardware resources in comparison to the leading counterpart available in the literature [3] and implementations for other security levels achieved similar improvement.**

Keywords: hardware architectures, isogeny-based cryptography, Montgomery multiplication, post-quantum cryptography, SIKE.

I. 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 [4], 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 [5]: 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 has anticipated that the second round will conclude by June 2020 and the third round will begin after. During the two final rounds, the PQC candidates will be scrutinized for their security and performance.

When considering quantum-safe alternatives to ECC, isogeny-based cryptography appears as an attractive replacement. The security of isogeny-based cryptosystems such as Supersingular Isogeny Key Encapsulation (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 supersingular isogeny key encapsulation (SIKE) [2] scheme features the smallest public key sizes [6], [7] of known quantum-safe public key exchange algorithms. Although isogeny-based cryptography is among the newest PQC candidates, SIKE offers a conservative security analysis, no possibility of decryption errors, and similar computations to well-established ECC. Small public key sizes are extremely advantageous in many different scenarios as it reduces the communication overhead and storage necessary for secure communications. As an example, low communication overhead is critical to establishing and maintaining secure communications over long distances or in high interference environments. The smallest set of SIKE parameters with key compression features keys of only 196 bytes, which is only around three times larger than 57-byte NIST X448 or 67-byte NIST P-521 public keys. SIKE offers all recommended security levels named SIKEp434, SIKEp503, SIKEp610, and SIKEp751 for NIST level-1, -2, -3, -5, respectively. Unfortunately, the main drawback of SIKE is that it is a few orders of magnitude slower than ECC or other PQC schemes. However, recently researchers were able to improve the computation time of SIKE by over an order of magnitude [8], [9], 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:

Our contributions:

- We develop a highly optimized Montgomery multiplication algorithm and architecture that further utilizes the special form of SIKE prime. We experimented various configurations for our high-radix design to find the best choice for area-time trade-offs.

Table I. SIKE primes for post-quantum cryptography based on NIST Round 2 standardization process [2]

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 implement SIKE for NIST Round 2 primes; SIKEp434, SIKEp503, SIKEp610, and SIKEp751 with the developed Montgomery multiplier architecture.
- We evaluate time and area performance of the proposed hardware architecture benchmarked on an FPGA and compare with counterparts.

The organization of the paper is as follows. In Section II, we give a literature review of SIKE. In Section III, we discuss the algorithm and architecture of our highly optimized Montgomery multiplication. In Section IV, we propose our SIKE architecture and compare our results with counterparts available in the literature. Finally, in Section V, we give our final thoughts and discuss future work.

II. Preliminaries: SIKE Protocol

In this section, we provide an 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 [2] for better performance.

A. 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 I 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 [2]). 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 I) 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.

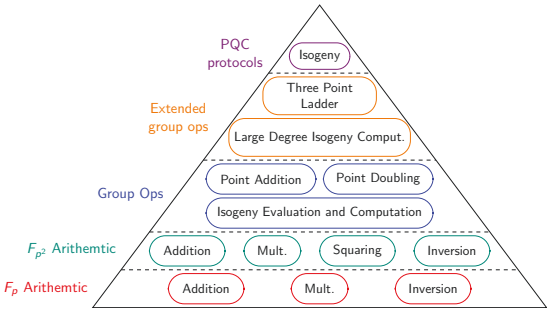

Figure 1. Breakdown of isogeny computations [9]

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 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.

Table II. 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 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: $\texttt{MontMult}(a, b)$

1 $T \leftarrow 0$
2 **for** $i \leftarrow 0$ **to** $s - 1$ **do**
3 $\quad (C, S) \leftarrow T[0] + a[i] \cdot b[0]$ } PE Initial
4 $\quad m \leftarrow S$
5 $\quad$ **for** $j \leftarrow 1$ **to** $s_A - 1$ **do**
6 $\quad\quad (C, S) \leftarrow T[j] + a[i] \cdot b[j] + C$ } s_A-Mult
7 $\quad\quad T[j-1] \leftarrow S$
8 $\quad U[s_A] \leftarrow m + m \cdot p[s_A]$ } s_B-Red0
9 $\quad$ **for** $j \leftarrow s_A + 1$ **to** $s - 1$ **do**
10 $\quad\quad U[j] \leftarrow m \cdot p[j]$ } s_B-Red
11 $\quad$ **for** $j \leftarrow s_A$ **to** $s - 1$ **do**
12 $\quad\quad (C, S) \leftarrow T[j] + U[j] + a[i] \cdot b[j] + C$ } s_B-Mult
13 $\quad\quad T[j-1] \leftarrow S$
14 $\quad$ **if** $p < 2^K - 2$ **then**
15 $\quad\quad (C, S) \leftarrow C$
16 $\quad\quad T[s-1] \leftarrow S$
17 $\quad$ **else**
18 $\quad\quad (C, S) \leftarrow T[s] + C$ } PE Final
19 $\quad\quad T[s-1] \leftarrow S$
20 $\quad\quad T[s] \leftarrow C$
21 **return** T

III. 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 [3], which utilizes the adder in [14], 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 II shows optimal parameters for the modular adder we are using.

For the modular multiplication ($a \times b \mod p$), 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. Word-by-word Montgomery multiplication algorithms were proposed in [15], [16]. Some hardware implementations can be found in [3], [17], [18], [19], [20], [21], [22].

Finely Integrated Operand Scanning (FIOS) Montgomery multiplication algorithm is a word-by-word algorithm first proposed in [15]. The original implementation was suitable for software. In [17], 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.

A. Proposed Montgomery Multiplication Algorithm

We further optimized the Montgomery multiplication algorithm in [17] 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 $w < e_A$, $p' = -p^{-1} \mod 2^w = 1$. This leads to $m = S \cdot p' \mod 2^w = S$. Finally, a second carry C_r is propagated to the first multiplication-reduction step. $(C_r, S) = S + m \cdot p[0] = m + m \cdot p[0] = (m, 0) \implies C_r = m$. 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 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 $1 \leq j < s_A = \lfloor 2^{e_A}/w \rfloor$ (s_A-$\texttt{Mult}$), 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 $s_A \leq j < s$ (s_B-$\texttt{Mult}$ and s_B-$\texttt{Red}$), all operations of

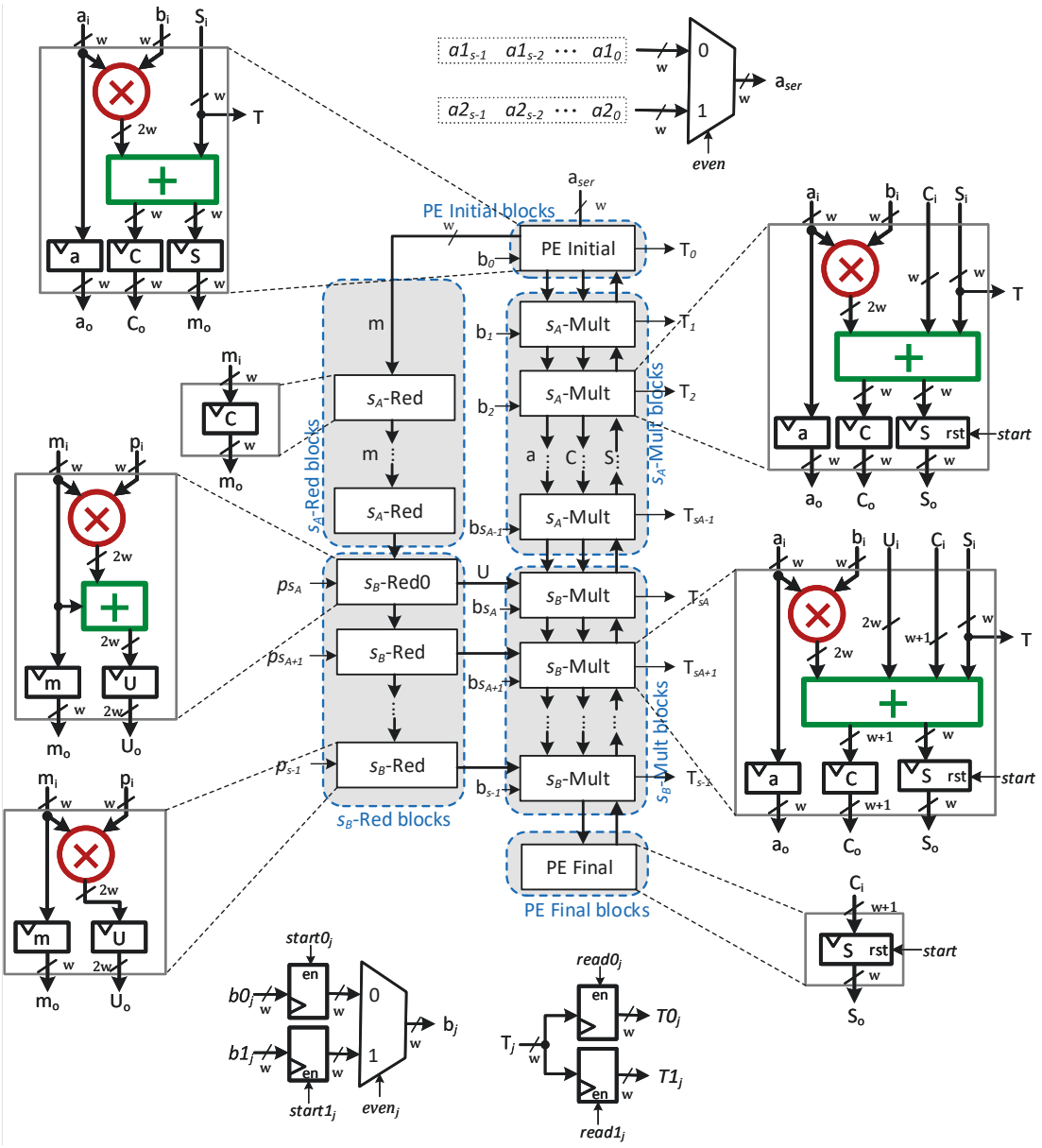

Figure 2. Proposed Montgomery multiplication architecture.

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.

B. 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 carry C is pushed to

Table III. 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.* [17] 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	-	-	$M_w + A_{2w}$	-	$(s + s_B)M_W + (2s + s_B)A_{2w}$

Note: $s_B = s - s_A$

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 III 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 [17] (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.

Table IV. 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 V. Montgomery multiplication DSP and timing analysis

Reference	# DSP	Freq (MHz)	Latency (cc)		Latency (ns)	
			Mult.	Interleave	Mult.	Interleave
SIKEp434						
Liu *et al.* [23]*	36	236	66	54	280	229
This work	**65**	**294.0**	**81**	**52**	**276**	**177**
SIKEp503						
Koziel *et al.* [3]	88	171.2	70	49	409	286
Liu *et al.* [23]*	64	213	66	54	310	254
This work	**75**	**294.0**	**93**	**60**	**316**	**204**
SIKEp610						
Liu *et al.* [23]*	81	191	66	54	346	283
This work	**90**	**294.0**	**111**	**72**	**378**	**245**
SIKEp751						
Koziel *et al.* [3]	128	167.4	100	69	597	412
Liu *et al.* [23]*	144	161	66	54	410	335
This work	**113**	**294.0**	**138**	**90**	**469**	**306**

∗ LUT usage is 5-6× more than our design.

C. Implementation and Results

The FPGA we are using in our SIKE implementation is the Xilinx Virtex-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

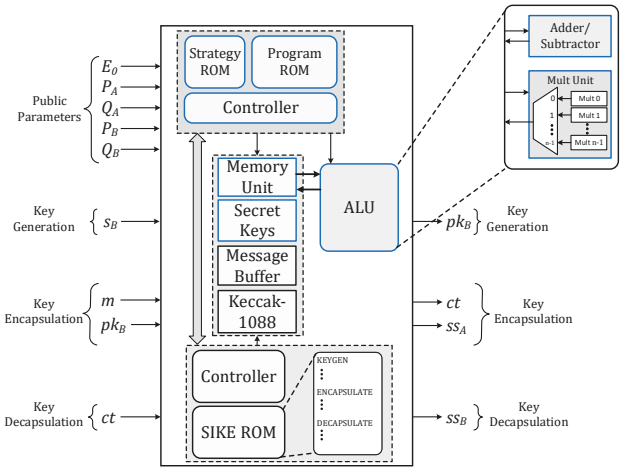

Figure 3. Proposed Hardware Architecture for SIKE protocol.

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 IV shows how to utilize a maximum of 2 DSPs per block. In [17], 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 V 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 [3]. However, the higher frequency dominates the increased cycle count and the overall total time to perform an operation is lower. In [23], 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.

IV. FPGA Implementations of SIKE

The implementation is performed in Xilinx Vivado 2019.2 for Xilinx Virtex-7 FPGA xc7vx690tffg1157-3 to be able to fairly compare our proposed scheme with the ones available in the literature. This FPGA includes 108,300 Slices (each 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. Our design is based on the design in the leading literature [3] with a modified ALU based on Section III.

A. Proposed SIKE Architecture

The architecture for SIKE used in our design is illustrated in Fig. 3 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 ($\mod 2p$) as well as modular reduction ($\mod p$) over the specified primes for SIKE. The multiplication unit consists of n Dual-Multipliers based on the design proposed in Section III. 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 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 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. Secret keys are generated in the host CPU. Following the SIKE protocol discussed in Section II-A, key encapsulation and decapsulation are performed and ss_A and ss_B are generated.

B. Implementation Results and Comparison

The proposed SIKE architectures for all NIST security levels were implemented and tested using Xilinx Vivado

Table VI. Area and Timing results of SIKE implementation in Xilinx Virtex-7

Reference	# Mults	Area					Time			Area×Time
		# FFs	# LUTs	# Slices	# DSPs	# BRAMs	Freq (MHz)	Latency ($cc \times 10^6$)	Total time (ms)	AT×10^{-3}
SIKEp434										
Massolino *et al.* [24] (Fast)	-	-	-	7,408	162	38.0	152.2	-	24.3	180
Koziel *et al.* [3]	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**	**2.19**	**8.8**	**48**
SIKEp503										
Koziel *et al.* [9]*	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.* [8]*	6	24,908	18,820	7,491	192	43.5	202.1	3.34	16.5	124
Massolino *et al.* [24] (Fast)	-	-	-	7,408	162	38.0	152.2	-	28.7	212
Koziel *et al.* [3]	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**	**2.88**	**11.8**	**73**
SIKEp610										
Massolino *et al.* [24] (Fast)	-	-	-	7,408	162	38.0	152.2	-	51.8	384
Koziel *et al.* [3]	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**	**4.56**	**19.1**	**142**
SIKEp751										
SIKE Team [2]	8	51,914	44,822	16,756	376	56.5	198.0	6.60	33.4	560
Massolino *et al.* [24] (Fast)	-	-	-	7,408	162	38.0	152.2	-	60.8	450
Koziel *et al.* [3]	8	50,079	39,953	15,834	512	43.5	163.1	4.55	27.8	440
This work	**8**	**39,339**	**20,207**	**11,136**	**452**	**41.5**	**232.7**	**5.93**	**25.5**	**284**

∗ SIDH

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 VI. 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 [3] as well as the Software-Hardware co-design [24] (fast implementation only) and some of the previous Supersingular Isogeny Diffie-Hellman (SIDH) implementations. 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 8.8 ms. The drop in frequency in comparison to the Montgomery multiplier in Table V 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% improved in comparison to the leading counterpart [3]. For the remaining security levels in Virtex-7, a similar improvement can be observed. It is to be noted that the design in [24] 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.

The improvements made in the design makes SIKE a feasible option for small embedded devices. Note that SIKE 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 [3]) 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.

V. Conclusion

Post-quantum crypto accelerator hardware cores offer chip-makers 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 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% less hardware area and is 12-20% 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 bench-marking the whole de-

sign with compressed keys for various security level required by NIST.

VI. Acknowledgment

The authors would like to thank the reviewers for their comments. This work 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." Cryptology ePrint Archive, Report 2018/313, 2018.

[2] R. Azarderakhsh, M. Campagna, C. Costello, L. De 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 Post-Quantum Standardization Project, 2019.

[3] 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 Transactions on Circuits and Systems I: Regular Papers*, pp. 1–13, 2020.

[4] P. W. Shor, "Algorithms for Quantum Computation: Discrete Logarithms and Factoring," in *35th Annual Symposium on Foundations of Computer Science (FOCS 1994)*, pp. 124–134, 1994.

[5] 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.

[6] R. Azarderakhsh, D. Jao, K. Kalach, B. Koziel, and C. Leonardi, "Key Compression for Isogeny-Based Cryptosystems," in *Proceedings of the 3rd ACM International Workshop on ASIA Public-Key Cryptography*, pp. 1–10, 2016.

[7] C. Costello, D. Jao, P. Longa, M. Naehrig, J. Renes, and D. Urbanik, "Efficient compression of sidh public keys," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pp. 679–706, Springer, 2017.

[8] B. Koziel, R. Azarderakhsh, and M. Mozaffari-Kermani, "A High-Performance and Scalable Hardware Architecture for Isogeny-Based Cryptography," *IEEE Transactions on Computers*, vol. 67, pp. 1594–1609, Nov 2018.

[9] B. Koziel, R. Azarderakhsh, M. Mozaffari-Kermani, and D. Jao, "Post-Quantum Cryptography on FPGA Based on Isogenies on Elliptic Curves," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 64, pp. 86–99, Jan 2017.

[10] The National Institute of Standards and Technology (NIST), "Sha-3 standard: Permutation-based hash and extendable-output functions," tech. rep., 2015.

[11] L. De Feo, D. Jao, and J. Plût, "Towards Quantum-Resistant Cryptosystems from Supersingular Elliptic Curve Isogenies," *Journal of Mathematical Cryptology*, vol. 8, pp. 209–247, Sep. 2014.

[12] P. L. Montgomery, "Speeding the Pollard and Elliptic Curve Methods of Factorization," *Mathematics of Computation*, 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 Transactions on Computers*, vol. 67, no. 11, pp. 1622–1636, 2017.

[14] T. B. Preußer, M. Zabel, and R. G. Spallek, "Accelerating computations on fpga carry chains by operand compaction," in *2011 IEEE 20th Symposium on Computer Arithmetic*, pp. 95–102, IEEE, 2011.

[15] C. Kaya Koc, T. Acar, and B. S. Kaliski, "Analyzing and comparing montgomery multiplication algorithms," *IEEE Micro*, vol. 16, pp. 26–33, June 1996.

[16] H. Orup, "Simplifying Quotient Determination in High-Radix Modular Multiplication," in *Proceedings of the 12th Symposium on Computer Arithmetic*, ARITH '95, (Washington, DC, USA), pp. 193–9, IEEE Computer Society, 1995.

[17] R. El Khatib, R. Azarderakhsh, and M. Mozaffari-Kermani, "Optimized algorithms and architectures for montgomery multiplication for post-quantum cryptography," in *International Conference on Cryptology and Network Security*, pp. 83–98, Springer, 2019.

[18] A. Mrabet, N. E. Mrabet, R. Lashermes, J. Rigaud, B. Bouallegue, S. Mesnager, and M. Machhout, "High-performance elliptic curve cryptography by using the CIOS method for modular multiplication," in *Risks and Security of Internet and Systems - 11th International Conference, CRiSIS 2016, Roscoff, France, September 5-7, 2016, Revised Selected Papers*, pp. 185–198, 2016.

[19] H. Alrimeih and D. Rakhmatov, "Fast and flexible hardware support for ecc over multiple standard prime fields," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 22, pp. 2661–2674, Dec 2014.

[20] T. Blum and C. Paar, "High-radix montgomery modular exponentiation on reconfigurable hardware," *IEEE Transactions on Computers*, vol. 50, pp. 759–764, July 2001.

[21] G. Chen, G. Bai, and H. Chen, "A high-performance elliptic curve cryptographic processor for general curves over $\mathrm{gf}(p)$ based on a systolic arithmetic unit," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 54, pp. 412–416, May 2007.

[22] H. Eberle, N. Gura, S. C. Shantz, V. Gupta, L. Rarick, and S. Sundaram, "A public-key cryptographic processor for rsa and ecc," in *Proceedings. 15th IEEE International Conference on Application-Specific Systems, Architectures and Processors, 2004.*, pp. 98–110, Sep. 2004.

[23] W. Liu, Z. Ni, J. Ni, C. Rafferty, and M. O'Neill, "High performance modular multiplication for sidh," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2019.

[24] P. M. C. Massolino, P. Longa, J. Renes, and L. Batina, "A compact and scalable hardware/software co-design of sike," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pp. 245–271, 2020.

[25] B. Koziel, R. Azarderakhsh, and M. Mozaffari-Kermani, "Fast Hardware Architectures for Supersingular Isogeny Diffie-Hellman Key Exchange on FPGA," in *Progress in Cryptology – INDOCRYPT 2016: 17th International Conference on Cryptology in India*, pp. 191–206, 2016.