Cryptographic Engineering a Fast and Efficient SIKE in FPGA

RAMI ELKHATIB, BRIAN KOZIEL, and REZA AZARDERAKHSH, Florida Atlantic University, USA
MEHRAN MOZAFFARI KERMANI, University of South Florida, USA

Recent attacks have shown that SIKE is not secure and should not be used in its current state. However, this work was completed before these attacks were discovered and might be beneficial to other cryptosystems such as SQISign. The primary downside of SIKE is its performance. However, this work achieves new SIKE speed records even using less resources than the state-of-the-art. Our approach entails designing and optimizing a new field multiplier, SIKE-optimized Keccak unit, and high-level controller. On a Xilinx Virtex-7 FPGA, this architecture performs the NIST Level 1 SIKE scheme key encapsulation and key decapsulation functions in 2.23 and 2.39 ms, respectively. The combined key encapsulation and decapsulation time is 4.62 ms, which outperforms the next best Virtex-7 implementation by nearly 2 ms. Our implementation achieves speed records for the NIST Level 1, 2, and 3 parameter sets. Only our NIST Level 5 parameter set was beat by an all-out performance implementation. Our implementations also efficiently utilize the FPGA resources, achieving new records in area-time product metrics for all parameter sets. Overall, this work continues to push the bar for accelerating SIKE computations to make a stronger case for SIKE standardization.

CCS Concepts: • **Security and privacy → Hardware security implementation**; **Embedded systems security**.

Additional Key Words and Phrases: isogeny-based cryptography, Montgomery multiplication, post-quantum cryptography, RISC-V, SIKE

1 INTRODUCTION

In 2016, the United States National Institute of Standards and Technology (NIST) initiated a multiple year process to standardize post-quantum cryptography (PQC) for use by the US government [61]. The fear is that a large-scale quantum computer will soon be available that will completely dismantle our deployed classical cryptography. Post-quantum cryptography includes cryptosystems that are secure against attacks by both classical and quantum computers. Unfortunately, today's commonly deployed public-key cryptosystems such as RSA or elliptic curve cryptography (ECC) are vulnerable to a large-scale quantum computer invoking Shor's algorithm [59]. Shor's algorithm completely breaks the underlying discrete logarithm or factorization problems that support ECC or RSA, respectively. Private-key cryptosystems such as AES or SHA will be weakened by a large-scale quantum computer utilizing Grover's algorithm [32], but their higher security parameter sets may still be used. It is unknown when such a large-scale quantum computer will be available with estimates ranging from a few years to several decades. However, there must be sufficient time to evaluate, implement, and deploy PQC algorithms. Historically, it has taken several years if not decades to completely transition our infrastructure. Thus, at the conclusion of the NIST PQC standardization process, we will begin a huge transition to quantum-safe algorithms over the coming decade.

Authors' addresses: Rami Elkhatib, relkhatib2015@fau.edu; Brian Koziel, bkoziel2017@fau.edu; Reza Azarderakhsh, razarderakhsh@fau.edu, Florida Atlantic University, 777 Glades Rd, Boca Raton, Florida, USA, 33431; Mehran Mozaffari Kermani, mehran2@usf.edu, University of South Florida, 4202 E Fowler Ave, Tampa, Florida, USA, 33620.

1539-9087/2023/2-ART $15.00
https://doi.org/10.1145/3584919

The NIST PQC standardization process is currently at the conclusion of its third round of evaluation. This standardization process allows public submission of algorithms which are then evaluated and scrutinized by various experts across the globe. Starting at 69 full and complete submissions, the third round has cut down to 15 submissions, of which 7 candidates are Round 3 Finalists and the other 8 candidates are Round 3 Alternatives. The finalists are slated to be standardized or discarded at the conclusion of the third round while some alternative candidates may continue to a fourth round of evaluation for eventual standardization. These candidates are further divided into the public-key encryption and key-establishment group as well as the digital signature group, to specify their function in public-key infrastructure. Another separation among these candidates is its hard foundational problem. There are several families of hard problems that are considered to be resistant to quantum computers such as lattices, isogenies, or hashes. For instance, the learning with errors problem is a lattice hard problem that secures several of the NIST lattice submissions. Among the NIST PQC submissions, there are many tradeoffs between foundational problem, performance, bandwidth, implementation profile, and so on. There is no clear winner for each evaluation aspect, but some cryptosystems feature great advantages.

This paper focuses on the supersingular isogeny key encapsulation (SIKE) [4] candidate which enables key establishment between two parties while also featuring the smallest public key sizes. Small public key sizes mean less bandwidth when sending the public key as well as less space to store a party's public key. SIKE is the only isogeny-based candidate in the NIST PQC process. Protected by the difficulty to compute isogenies between two supersingular elliptic curves, SIKE also features no possibility for decryption errors, no complicated error distributions, and a simple, conservative security analysis when assuming only generic attacks. In its report on Round 2 PQC candidates, NIST praised SIKE for its small key and ciphertext sizes that could enable some applications [2]. However, NIST placed SIKE as a Round 3 Alternative candidate because of its slow performance and because further investigation of its basic security problem was needed. This paper serves as another nice improvement in SIKE's performance. In terms of the basic security problem, recent attacks [11, 46, 53] have shown that the private key can be recovered from the public key. Therefore, SIKE is not secure in its current state.

Among the PQC candidates, SIKE can heavily benefit from an optimized hardware implementation. For instance, consider a small embedded ARM Cortex-M4 implementation of SIKE that can perform key encapsulation and decapsulation (combined for SIKEp434) in 140 million cycles [3]. When running at a low frequency, this latency may be unacceptable. However, if this intense computation was instead offloaded to a SIKE hardware accelerator, this computation could be completed in only a few million cycles. Our FPGA results, for instance, feature a key encapsulation and decapsulation time of 1.264 million cycles for SIKEp434, over 100 times improvement. Since many currently deployed chips include hardware co-processors for ECC or RSA, it is not out of question to consider a hardware accelerator for SIKE. SIKE offers a unique perspective where the computational overhead can be reduced by a hardware co-processor and the communication overhead is smaller as a result of its public key sizes.

Related Work. Over the past several years, SIKE and its predecessor SIDH have enjoyed a myriad of hardware implementation works and improvements that continue to push its performance envelope. Starting in 2016, Koziel *et al.* [41] published the first hardware implementation of SIDH (SIKE's predecessor), achieving a full key exchange in about 33 ms for a 503-bit prime. With improvements to isogeny algorithms, isogeny-optimized multiplication, and high-level control, this current paper achieves a similar set of computations (SIKE key encapsulation and key decapsulation over SIKEp503) in just 5.8 ms. Over the following years, Koziel *et al.* continued to improve their design, integrating techniques for high parallelization during the isogeny computation [40], implementing a scalable architecture with new isogeny formulas [39], and upgrading the architecture from an SIDH implementation to a SIKE implementation [36]. Concurrently, several different authors have proposed their own optimizations to the field multiplication unit for SIDH/SIKE including Barrett reduction [35], Montgomery multiplication [20, 22, 43–45, 63], and even redundant number system (RNS) multiplication [56]. These multiplication algorithms take advantage of special number representations bywhich multiplication can

take advantage of the shape of isogeny-friendly primes. In terms of full implementations, the implementations have primarily focused on high performance by using a mix of replicated multipliers with efficient schedulers including [23, 25, 26, 36, 51, 64]. Based on the number of field multiplication and additions required for these isogenies on elliptic curves, multiple addition and multiplication units are required, which must be efficiently scheduled to achieve good performance. Otherwise, software-hardware co-design implementations of SIKE feature both the flexibility of software with the optimization of complex computations in hardware including [9, 21, 24, 47, 55]. These implementations have generally featured a smaller profile and the flexibility to support multiple parameter sets.

This work presents a new high-performance implementation of SIKE that is also area-efficient. Our contributions can be summarized as follows:

Our contributions:

- We propose and implement a new SIKE-optimized multiplier that efficiently utilizes DSPs and resources for Xilinx 7th generation FPGAs.
- We propose and implement a small hardware Keccak accelerator that is specially finetuned for SIKE's performance profile. This Keccak requires approximately 300 slices and runs at 1500 cycles per permutation.
- We propose and implement an isogeny accelerator controller that utilizes a tiny RISC-V processor.
- We achieve the best area-time products for all SIKE parameter sets.
- We achieve new FPGA speed records for SIKEp434, SIKEp503, and SIKEp610.

The organization of the paper is as follows. In Section 2, we review the fundamentals of SIKE. In Section 3, we propose our finite field accelerator which contains our new field multiplier. In Section 4, we propose our SIKE-optimized Keccak unit. In Section 5, we propose our new SIKE control architecture using a RISC-V processor. In Section 6, we present and discuss our FPGA design's results and compare to the state-of-the-art. In Section 7, we close and discuss more directions for future work.

2 PRELIMINARIES

In this section, we review the necessary background of fundamentals of isogeny-based cryptography needed for SIKE [4]. The reader can refer to [28] for further background on the mathematics of isogenies.

2.1 Isogeny Fundamentals

Isogeny-based cryptography primarily focuses on isogenies, or mappings, between elliptic curves and their use in creating secure cryptosystems. An elliptic curve over a finite field $\mathbb{F}_q$ is the collection of all points (x, y) as well as the point at infinity that satisfy the short Weierstrass form of elliptic curve $E/\mathbb{F}_q : y^2 = x^3 + ax + b$, where $a, b, x, y \in \mathbb{F}_q$. This set creates an abelian group over addition. In standard elliptic curve cryptography, we pick a point $P = (x, y)$ and perform consecutive point additions and point doublings to execute an elliptic curve point multiplication, $Q = kP$ where $k \in \mathbb{Z}$ and $P, Q \in E$. Given P and Q, the elliptic curve discrete logarithm problem states that it is computationally infeasible to find the scalar k. However, a large-scale quantum computer can use Shor's algorithm [59] to compute k.

Isogeny-based cryptography uses isogenies between elliptic curves for which there are cases where it is difficult for a quantum computer to compute the isogenies. An elliptic curve isogeny over $\mathbb{F}_q$, $\phi : E \rightarrow E'$, is a non-constant rational map from $E(\mathbb{F}_q)$ to $E'(\mathbb{F}_q)$ that is a group homomorphism, or preserves the point at infinity. An elliptic curve's j-invariant serves as a unique identifier for the elliptic curve's isomorphism class. An isogeny moves from one elliptic curve to another elliptic curve, changing j-invariants. In SIDH and SIKE, we efficiently compute isogenies by using Vélu's formulas [65] over a kernel point: $\phi : E \rightarrow E/\langle ker \rangle$. The degree of an isogeny

is its degree as a rational map. For efficiency, we compute a large-degree isogeny of the form ℓ^e as a chain of e isogenies of degree ℓ.

2.2 Isogeny-Based Cryptosystems

History. Isogeny-based cryptography has evolved over the past few decades of research. The use of isogenies in cryptography was first proposed in independent works by Couveignes [17] and Rostovtsev and Stolbunov [54] that were first published in 2006. These works proposed utilizing the hardness of computing isogenies between ordinary elliptic curves as a basis for a key exchange. These papers initially claimed to have quantum resistance, until Childs, Jao, and Soukharev [13] proposed a quantum subexponential algorithm that computes isogenies between ordinary elliptic curves. Concurrently, Charles, Lauter, and Goren [12] also proposed a new isogeny-based hash function, this time based on the hardness to compute isogenies between *supersingular* elliptic curves in 2009. A few years later in 2011, Jao and De Feo [33] proposed the supersingular isogeny Diffie-Hellman (SIDH) key exchange protocol that was now also protected by the hardness to compute isogenies between supersingular elliptic curves. Interestingly, the non-commutative nature of the endomorphism ring of supersingular elliptic curves renders the Childs, Jao, and De Feo [13] isogeny attack unusable. In 2017, the supersingular isogeny key encapsulation (SIKE) mechanism was submitted as an IND-CCA2 upgrade of SIDH to the NIST PQC standardization process [5]. In 2022, Castryck and Decru [11] showed that the private key can be recovered from the public key as long as the endomorphism ring of the starting curve is known. Furthermore, Maino and Martindale [46] and Robert [53] extended the attack to include any random starting curve.

Throughout the history of SIKE, we have seen many upgrades to the use of isogenies for cryptography. For instance, we have various investigations of foundational isogeny security [1, 16, 29, 34], public key compression [6, 14, 50, 52], digital signatures [30, 66], hybrid key exchange [8, 15], and password-authenticated key exchange [7, 60]. Aside from the hardware implementations we have previously described, there are also a plethora of software implementations targeting microarchitectures including x86-64 [15, 18, 27], ARM A-processors [42, 58], and ARM M-processors [3, 57]. Lastly, there have also been works on implementation security, including side-channel attacks [37, 38] and fault attacks [31, 62]. Overall, these works have greatly strengthened our knowledge of SIKE implementations through faster performance, better implementation protection, and even more applications.

2.3 SIKE

The supersingular isogeny key encapsulation (SIKE) [4] mechanism is a key encapsulation mechanism (KEM) based on the hardness of computing isogenies between supersingular elliptic curves. SIKE is the only isogeny-based candidate in the NIST PQC standardization process, coming with submitters from industry and academia. As a KEM, SIKE allows two parties, Alice and Bob, to securely establish a shared secret. As is shown in Figure 1, there are three phases. In this scenario Bob is initiating a secure session with Alice. Bob performs key generation, by which he generates a secret key and a public key. The public key is then broadcast over a public channel to Alice. Note that key generation only needs to be performed once by a party. Alice retrieves Bob's public key and proceeds by performing key encapsulation, where she generates a ciphertext and a locally stored shared secret. Alice responds to Bob by sending her ciphertext over a public channel. Bob completes the key establishment by performing key decapsulation where Bob uses his secret key and Alice's ciphertext to generate a shared secret. Assuming nothing went wrong, both parties have now separately generated the same shared secret which can be used to generate a symmetric key for encrypted communications.

In the SIKE submission, there are eight parameter sets targeting various NIST security levels from 1 to 5. NIST security level 1 is considered as hard to break as a brute force attack on AES128, NIST security level 2 is considered as hard to break as finding a hash collision on SHA2-256, and so on. There are SIKE parameter sets at NIST security levels 1, 2, 3, and 5. Within each security level there are uncompressed and compressed variants

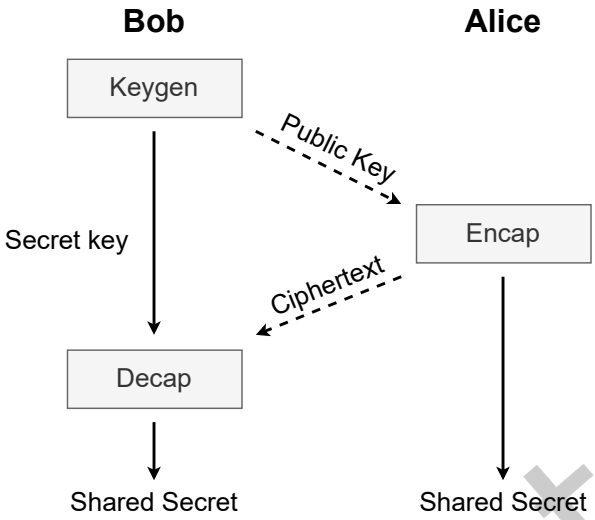


Fig. 1. SIKE key establishment operations

Table 1. Summary of Uncompressed SIKE Parameter Sets from Azarderakhsh et al. [4]

Security Level	SIKE Parameters	Prime Form	As Strong As	Secret Key Size [B]	Public Key Size [B]	Ciphertext Size [B]	Shared Secret Size [B]
NIST level 1	SIKEp434	$p_{434} = 2^{216}3^{137} - 1$	AES128	374	330	346	16
NIST level 2	SIKEp503	$p_{503} = 2^{250}3^{159} - 1$	SHA256	434	378	402	24
NIST level 3	SIKEp610	$p_{610} = 2^{305}3^{192} - 1$	AES192	524	462	486	24
NIST level 5	SIKEp751	$p_{751} = 2^{372}3^{239} - 1$	AES256	644	564	596	32

of SIKE. Compressed variants reduce the total communication overhead by slightly less than half, but at the cost of many more complex computations. This work primarily focuses on accelerating the uncompressed SIKE parameter sets. We summarize the uncompressed parameter sets of SIKE in Table 1. Each SIKE parameter set's name is based on the bitlength of its underlying prime, such as SIKEp434 for the 434-bit prime $p_{434} = 2^{216}3^{137} - 1$. SIKE primes are of the special form $p = 2^{e_A}3^{e_B} - 1$ for efficiency.

The two primary computations in the uncompressed version of SIKE include the large-degree isogeny as well as the SHAKE256 hash function. The large-degree isogeny, $\phi : E \rightarrow E/\langle R \rangle$ involves first computing a secret kernel point by using the party's secret key, $R = P + nQ$, and then performing a large-degree isogeny over that kernel by chaining together many small-degree isogenies. At the lowest level, kernel point computation and large-degree isogeny computation can be broken down into arithmetic over a finite field, $\mathbb{F}_{p^2}$, which can then be further broken down into prime field arithmetic $\mathbb{F}_p$. Thus, in Section 3, we discuss our hardware-specialized method to accelerate these low-level computations. Next, in Section 4, we present our approach for optimizing the SHAKE256 hash function for use in SIKE. Finally, we present our whole architecture in Section 5, whereby we efficiently control our finite field accelerator and SHAKE256 unit to carry out the whole of SIKE.

Table 2. Summary of $\mathbb{F}_p$ field addition and multiplication latencies

SIKE Parameters	Addition/ Subtraction	Multiplication Interleave	Multiplication Total
SIKEp434	2	10	26
SIKEp503		11	28
SIKEp610		13	32
SIKEp751		16	38

3 PROPOSED FIELD ARITHMETIC UNIT

In this section, we discuss our low-level field arithmetic unit for the SIKE accelerator. At the lowest level, the large-degree isogeny computation can be broken down to modular addition and modular multiplication over a $\mathbb{F}_p$ prime finite field. Thus, we present our modular addition and modular multiplication units. Since these functions are used thousands of times within SIKE, we have carefully optimized them for the Xilinx 7th Generation FPGAs. Table 2 summarizes the total latency for field addition and field multiplication over the SIKE parameter sets.

3.1 Field Addition Unit

Our field addition unit performs prime field addition or subtraction and is specially optimized for the SIKE primes. Given field elements $a, b, c \in \mathbb{F}_p$, finite field addition performs $a + b = c$, where all values are reduced modulo p. In a simple addition scenario, if $c > p$, then a correction must take place to bring c back in the range $[0, p-1]$. Since a and b are already in this range, the reduction $c = c - p$ can be performed. Likewise, for subtraction, $a - b = c$. In a simple subtraction scenario, if $c < 0$, then a reduction must take place by adding p, $c = c + p$. Thus, finite field addition or subtraction both require at most an addition and subtraction.

Our finite field addition unit closely follows that of [24] to perform a large precision addition/subtraction in a single cycle with a high frequency. This methodology specifically targets the Manchester carry chain architectures used in the Xilinx 7th Generation FPGAs. These carry chains are designed for fast addition. As is described in [24], there are three SIKE-specific optimizations in this adder/subtractor design including parallel prefix carry-look ahead simplification, final propagated carry simplification, and simultaneous addition/subtraction. With two pipeline stages, we can achieve a high frequency of around 300 MHz on the Xilinx Virtex-7 FPGA while performing a field addition or subtraction in only two cycles.

3.2 Proposed Field Multiplication Unit

The finite field multiplication unit architecture has most likely the largest impact on the resulting SIKE performance. Our multiplier design was specifically optimized for Xilinx 7th Generation FPGAs with the field adder, Keccak, and top-level designs in mind to achieve high throughput, high performance, and high frequency, all while using FPGA resources efficiently.

Given field elements $a, b, c \in \mathbb{F}_p$, finite field multiplication performs $a \times b = c$, where all values are reduced modulo p. If using standard multiplication, the resulting value for c may be twice the bitlength of the modulus p, requiring an expensive reduction operation to complete the field multiplication. This paper focuses on a new architecture using Montgomery multiplication [48] that includes the multiplication and reduction steps. Montgomery reduction is very efficient in hardware as it converts expensive division operations to shift operations, which are essentially free in hardware. Similar to existing multipliers, our proposed multiplier can support two simultaneous multiplications in its pipelines.

Algorithm 1: Simple explanation of the Montgomery multiplication hardware through an algorithm.

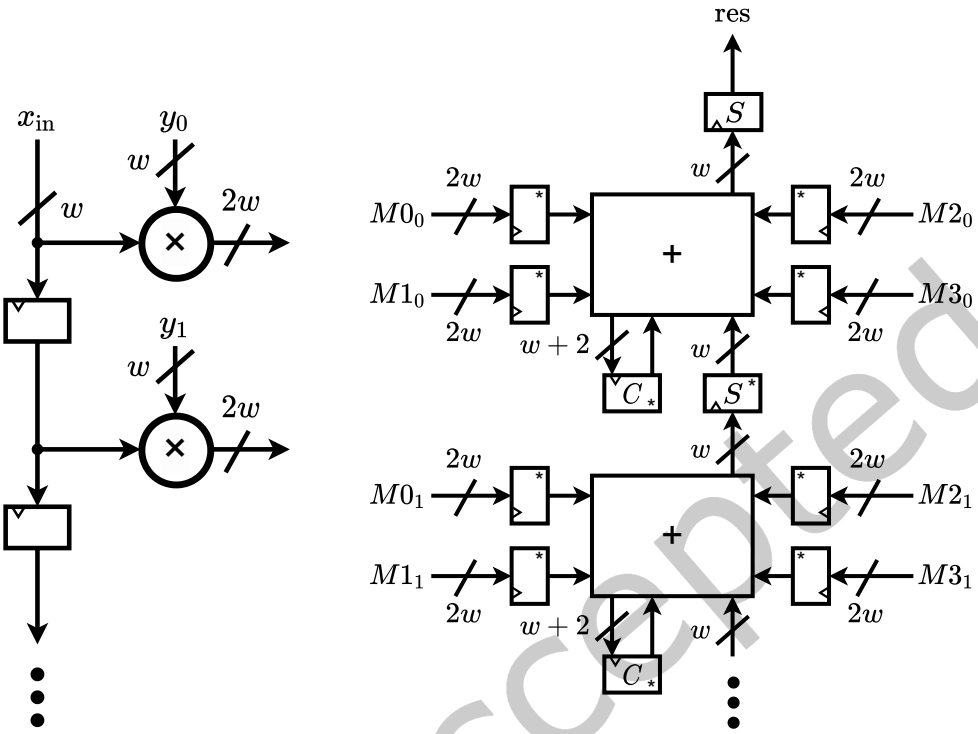
function `systolic_Montgomery_multiplication`
- **Parameters:** w : digit, s : # of digits, $k = w \times s$, $m = 2^{e_A}3^{e_B} - 1 < 2^{k-2}$, $n = m + 1$, $s_A = \lfloor e_A/w \rfloor$, $s_B = s - s_A$, $k_A = w \times s_A$, $k_B = w \times s_B$
- **Input:** $a < 2m$, $b < 2m$
- **Output:** res = MulMont(a, b)
- **Temporary:** $q < 2^{k_B}$, S, C, acc, Mult0, Mult1, Red0, Red1
- **Note:** All out of bound registers are 0
- **Note:** All registers are w bits except C which can be w to $w + 2$ bits
- **Note:** Cycle indicates current cycle and i indicates current block

1 **for** $cycle \leftarrow 0$ **to** $2s$ **do**
2 # Multiplication lanes
3 **for** $i \leftarrow 0$ **to** $\lceil s/2 \rceil$ **do**
4 Mult0[cycle][i] = a[cycle $- i$]b[$2i$]
5 Mult1[cycle][i] = a[cycle $- i - 1$]b[$2i + 1$]
6 # Reduction lanes
7 **for** $i \leftarrow 0$ **to** $\lceil s_B/2 \rceil$ **do**
8 Red0[cycle][i] = q[cycle $- i - s_A$]n[$2i + s_A$]
9 Red1[cycle][i] = q[cycle $- i - s_A - 1$]n[$2i + s_A + 1$]
10 S[cycle + 1] = 0
11 C[cycle] = 0
12 **for** $i \leftarrow 0$ **to** $s/2$ **do**
13 acc[i] = Mult0[cycle][i] + Mult1[cycle][i] + Red0[cycle][i] + Red1[cycle][i] + S[$i + 1$] + C[i]
14 S[i] = acc[i] % 2^w
15 C[i] = $\lfloor$acc[i] / $2^w\rfloor$
16 q[cycle] = S[0]
17 res[cycle $- s$] = S[0]
18 **return** res

3.2.1 Low Level Multiplication Components. Similar to existing Montgomery multiplication approaches, we use a systolic architecture. The high-level algorithm to explain our Montgomery multiplication operation is shown in Algorithm 1. We will use a number of the variables listed in this algorithm in our description. For clarity, we provide a brief description of our variables in Table 3. Most importantly, w is the digit size of the processing element, s is the number of digits in the systolic architecture, k is the total length of the systolic architecture, and m is the modulus.

In describing our Montgomery multiplication architecture, we will use a bottom-up approach. At the lowest level, we have a multiplier lane unit as depicted in Figure 2a that acts as the processing element within our systolic array. Each multiplier lane performs multiplication of x (w bits) with a large integer y ($m \times w$ bits). This is similar to the systolic multiplication lanes used in [21, 24].

At the next level up, we have an accumulator unit that is shown in Figure 2b. Each cell of the accumulator receives the results of one to four multiplication results (each of size $2w$ bits) and accumulates the results in two registers: sum S (w bits) and carry C (w to $w+2$ bits). The accumulator is also a systolic architecture. The majority of cells will take four multiplications, but the more significant cells in the array can take 3, 2, or 1 multipliers where the carry register C is $w + 2$, $w + 1$, or w bits, respectively. This is again similar to the accumulators used in



(a) Multiplier lane that acts as processing unit in our systolic architecture

(b) Accumulator unit within our multiplier

Fig. 2. Base cells used in proposed systolic architecture

[21, 24], but we can point out some key differences. First, the accumulator supports up to four multiplications instead of two multiplications as the prior art does. Second, the carry C is propagated in place as opposed to forward propagation. Third, the result (w bits) is retrieved from the first sum S instead of retrieving from a different sum S. Lastly, we include a star (*) on the registers in Figure 2b to indicate that they can send the register's value or a 0. This simply means that there is an additional multiplexer on the output to select between these values.

3.2.2 Core Systolic Multiplication Architecture. Next, we describe the core functionality of our systolic multiplication architecture which is shown in Figure 3. As is shown in this figure, there are four lanes Mult0, Mult1, Red0, and Red1 along with two accumulators Acc0 and Acc1. In the SIKE scenario, we are performing Montgomery multiplication on inputs $a, b < 2m$.

First, we explain the purpose of each of the four lanes. As a systolic architecture, each processing element performs multiplications or reductions over the inputs. For the ith cycle in a Montgomery multiplication, the Mult0 lane receives $x_{in} = a[i]$ sequentially and $y = b[0], b[2], b[4], \ldots$ in parallel. Similarly, the Mult1 lane receives $x_{in} = a[i-1]$ sequentially and $y = b[1], b[3], b[5], \ldots$ in parallel. There are ceil$(s/2)$ processing elements in Mult0 lane and floor$(s/2)$ processing elements in Mult1 lane. Note that Mult1 lane receives a given a input one cycle

Table 3. Summary of multiplication-related variables

Variable	Value	Description
e_A		Number of 2-isogenies Alice performs (SIKE parameter)
e_B		Number of 3-isogenies Bob performs (SIKE parameter)
m	$2^{e_A}3^{e_B} - 1$	Modulus which is the SIKE prime
w		Digit size in the systolic architecture
s	$\lceil \text{len}(m) + 2/w \rceil$	Number of digits in the systolic architecture
k	$w \times s$	Total number of bits in the multiplication part
R	2^k	Radix for Montgomery multiplication
s_A	$\lfloor e_A/w \rfloor$	Number of digits that have all 1's for modulus
s_B	$s - s_A$	Number of digits that do not have all 1's for modulus
k_A	$w \times s_A$	The number of bits eliminated from the reduction part
k_B	$w \times s_B$	The number of bits used in the reduction part
a		Array of operand a extended to k bits
b		Array of operand b extended to k bits
q		Array storing the quotient values for Montgomery multiplicaiton
n	$m + 1$	Adjusted Montgomery modulus to save an adder
S		Array stores the low w bits of the sum in the accumulator
C		Array stores remaining bits of the sum in the accumulator
Mult0		Array stores Mult0's lane results
Mult1		Array stores Mult1's lane results
Red0		Array stores Red0's lane results
Red1		Array stores Red1's lane results

delayed from Mult0. Each cell in these two lanes operates for s cycles with each cycle processing one digit of array a. Once the cells have processed all digits of a, the set of operands for the next $\mathbb{F}_p$ multiplication can be pushed to achieve a multiplication interleaving of s cycles.

On the reduction side, Red0 receives $x_{in} = q$ sequentially and $y = n[s_A], n[s_A + 2], n[s_A + 4]$,... in parallel. Similarly Red1 receives $x_{in} = q$ sequentially (but delayed 1 cycle from Red0) and $y = n[s_A{+}1], n[s_A{+}3], n[s_A{+}5]$, ... in parallel. The size of Red0 lane is ceil$(s_B/2)$, while the size of Red1 lane is floor$(s_B/2)$.

Next, we have the two accumulators Acc0 and Acc1. Both accumulators perform the same functionality, but slightly offset to achieve the multiplication interleaving. Each accumulator is ceil$(s/2)$, or the same size as the largest lane Mult0. Each processing element of the accumulator receives one multiplication result from each processing element of the four lanes Mult0, Mult1, Red0, and Red1. Specifically, cell 0 (the least significant processing element) of each accumulator receives the products from cell 0 of the lanes (Mult0, Mult1, Red0, Red1). Likewise, cell 1 of each accumulator takes from cells 1 of the lanes. This tiling continues until the end where we will have special cases for the final cell. For instance, if s_B is odd, then cell floor$(s_B/2)$ where Red1 ends will take 3 multiplier results from 3 lanes (Mult0, Mult1, Red0) and cell ceil$(s_B/2)$ where Red0 ends will take 2 multiplier results from (Mult0, Mult1). If s_B is even, then cell $s_B/2$ where Red0 and Red1 end will take 2 multiplier results from (Mult0, Mult1). These 2 multiplier result cells will continue until the very end. If s is odd, then cell floor$(s/2)$ will only have one multiplier result from lane Mult0, whereas if s is even, then there will be no cells with only one multiplier result cell. Lastly, the last accumulator cell ceil$(s/2) - 1$ does not have a next cell, so it does not have an S as input.

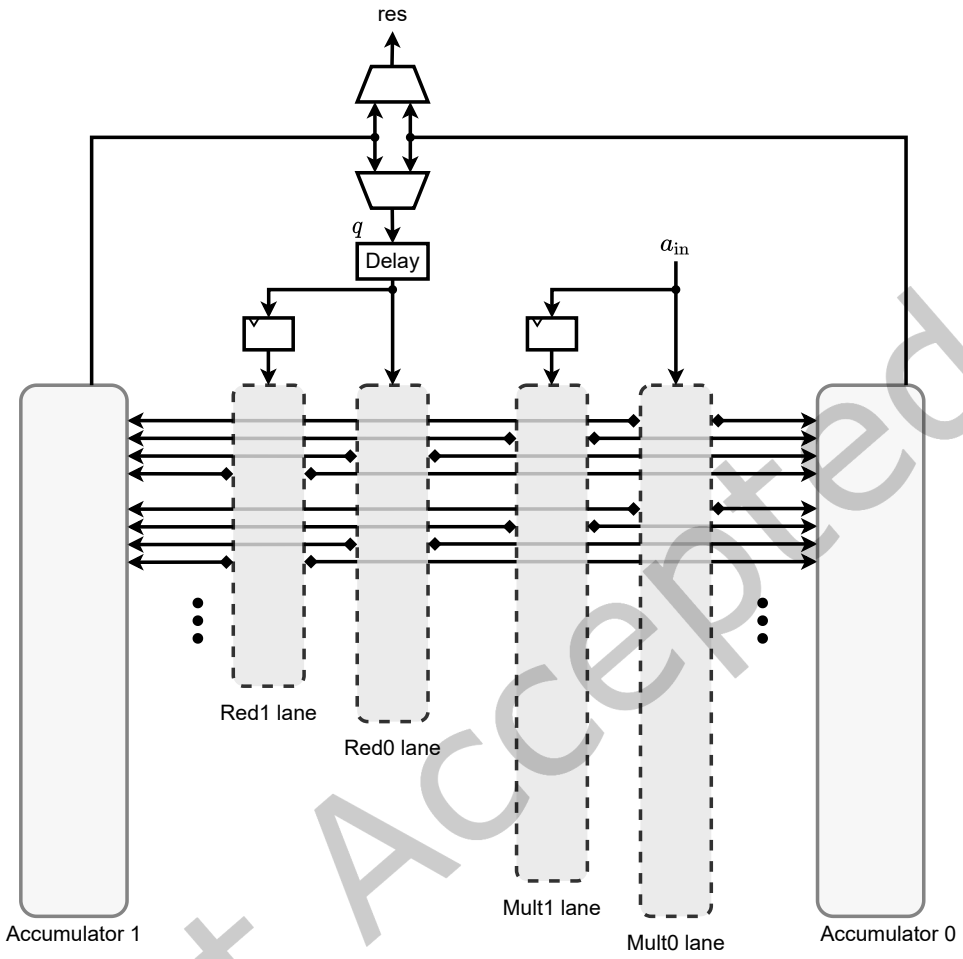


Fig. 3. Proposed multiplication architecture

The accumulators can also be controlled. In each Montgomery multiplication, the first time the accumulator uses S and C, 0 is pushed instead of the value of the register. On cycle 0, inputs S and C of cell 0 are both 0. On the following cycle, inputs S and C of cell 1 are both 0. This is effectively a resetting mechanism for the accumulator. The values of operand a are pushed from cycles 0 to $s-1$ while the values of the quotient q are pushed from cycles s_A to $s_A + s - 1$. The accumulator takes $2s$ cycles. Whenever the product of an $a \times b$ cell or $q \times n$ cell of the Montgomery multiplication algorithm is 0, a 0 is pushed for that respective lane in the accumulator.

The use of two accumulators also enables multiplication interleaving, whereby we can begin a multiplication before the current multiplication has finished. Since each cell of the lanes are only used for s cycles while each cell of the accumulator is used for $2s$ cycles (the total latency of the Montgomery multiplication), an additional accumulator is utilized to interleave multiplications. Each cell of the lanes uses s cycles for the first accumulator and s cycles for the second accumulator to achieve an s cycle interleave with $2s$ cycles of multiplication. The

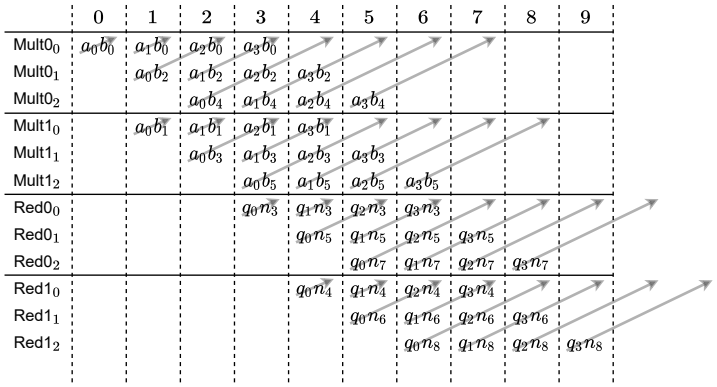


Fig. 4. Multiplication timing waveform for $s_A = 3$

result of the accumulator is the quotient q of the Montgomery multiplication. The quotient q is pushed in the reduction lanes with a delay such that $q[0]$ (the first quotient) aligns with $a[s_A]$ in the first cells of Red0 and Mul0, respectively. In the second s cycles of the accumulator, the accumulator's result is the output of the Montgomery multiplication. Coming back to the Montgomery multiplication, if we accumulate in $2s$ digits the following: $a[s-1:0] \times b[s-1:0] + n[s-1:0] \times q[s-1:0]$, then we will get q in the first s digits and the Montgomery multiplication result in the second s digits.

To further illustrate the functionality of this multiplication, we have included the waveform shown in Figure 4 to show the order of operations in our lanes. This figure shows the first 4 digits of operand a and quotient q for the first 3 cells in each lane assuming $s_A = 3$ as a function of cycles. The subscript indicates the cell number. As is shown, in cycle 0 we compute a_0b_0 in Mult0 cell 0 and in cycle 1 we compute a_1b_0 in Mult0 cell 1. Also in cycle 1, we compute a_0b_1 in Mult 1 cell 0. After $s_A = 3$ cycles are passed, Red0 cell 0 performs q_0n_3 which aligns with a_3b_0 performed by Mult0 cell 0.

To summarize the functionality of the multiplier as is shown in Figure 4, we note that the arrows show the sum S path inside the accumulator. This indicates that all values along the arrow are added together in the accumulator. Cells with the same number are added together. Going horizontally along the same cell shows the carry C path in the accumulator. For example, the carry of a_0b_0 is added to a_1b_0 in cell 0.

3.2.3 Multiplication Wrapper. This multiplier can perform two interleaved multiplications. As a result, we have implemented a higher level wrapper to handle the inputs and outputs. Notably, we have two sets of k-bit registers to handle two sets of input operands a and b. This wrapper pushes operand a into the multiplier w bits at a time using a shift register while operand b is pushed in parallel as soon as it is used to achieve the s-cycle interleaving. The results of our multiplications are retrieved w bits at a time using a separate k-bit shift register from the result of the accumulator.

For the Xilinx 7 series FPGAs, we chose a digit size $w = 48$. Thus, a 48×48 unsigned multiplication is performed among multiple DSPs. The DSP48E on these FPGA boards can perform up to a 24×16 unsigned multiplication. Therefore, to perform a 48×48 unsigned multiplication, we tiled 6 DSP48E units in the orientation shown in Figure 5. One operand's digit is split into 2 chunks of 24 bits while the other operand is split into 3 chunks of 16 bits. Every combination of these partial products are then pushed to one DSP to give a total of 6 partial results. The partial results are added together and correctly aligned to complete the 48×48 unsigned multiplication.

In order to achieve a high operating frequency, we have introduced six pipelines into our 48×48 unsigned multiplication. There is one pipeline for loading the operands into the DSP multiplier. One pipeline for the DSP

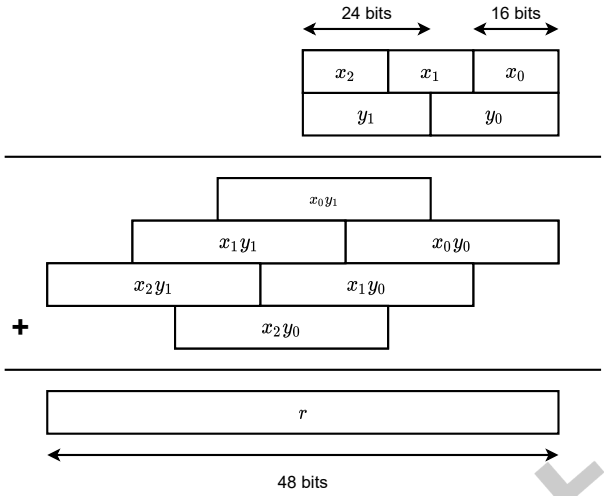


Fig. 5. DSP tiling to achieve 48 × 48 multiplications

Table 4. Area and timing results for the proposed standalone multiplier on Virtex-7 FPGA

SIKE Parameters	Area #FFs	#LUTs	#DSPs	#Slices	Freq (MHz)	Mult Latency Interleave	Total
SIKEp434	5,929	6,098	84	2,257	282.5	10	26
SIKEp503	6,570	6,585	98	2,478	290.7	11	28
SIKEp610	7,964	7,915	120	2,881	299.4	13	32
SIKEp751	9,758	9,711	146	3,559	301.2	16	38

multiplication. One pipeline for the 6 partial product addition. Two pipelines for the accumulator. Finally one last pipeline to compute the result. These six pipelines increase the total multiplication cost to $2s + 6$ cycles while keeping the interleave cost at s cycles.

3.2.4 Multiplication Area and Timing Results. We summarize the synthesized area and timing results of these multipliers on a Xilinx Virtex-7 FPGA in Table 4. These results are post-place and result in a similar fashion as described in Section 6. As we can see, the total area ranges from 2,257 slices and 84 DSPs for a 434-bit SIKE prime up to 3,559 slices and 146 slices for a 751-bit prime. Interestingly, the frequency appears to improve as the prime gets larger. This is attributed to the q delay which ranges from 0 delay cycles for SIKEp434 up to 6 delay cycles for SIKEp751, which is shown in Figure 3. In terms of latency, we reiterate that this multiplier can accept new multiplication operations based on the interleave latency and the multiplication result will be ready after the total latency. For instance, SIKEp434 will be able to perform new multiplications every 10 cycles and the result will be ready after 26 cycles.

4 SIKE-OPTIMIZED KECCAK

SIKE utilizes the SHAKE256 hash function which is built on top of the Keccak sponge function [10], as defined for SHA3. Unlike lattice candidates in the NIST PQC standardization process, hashing with SHAKE256 only requires a small proportion of SIKE's total execution time. For instance, SIKE's total execution time for high-performance

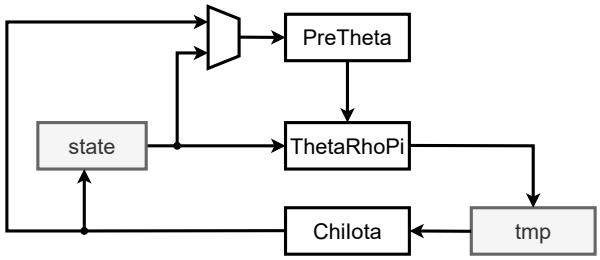


Fig. 6. Proposed Keccak accelerator architecture

hardware is a few million cycles, for which SHAKE256 may require only a few thousand cycles. However, past works in the literature have targeted SHAKE256 performance/area tradeoffs that are not properly aligned. This paper seeks a better trade-off between Keccak performance and area for SIKE.

For high-performance implementations, most implementations have typically opted to use the Keccak's team high-performance Keccak implementation. This implementation performs a Keccak round function in 1 cycle and a Keccak permutation function in 24 cycles. As is reported in [36], the total size of the Keccak block was found to be 3,747 LUTs and 2,703 flip-flops on a Xilinx Virtex-7 FPGA. For around 1,000 total cycles of SHAKE256 in SIKE, a few thousand LUTs and flip-flops are required.

On the other hand, some implementations have opted for a minimal profile Keccak, such as [21, 24]. In these cases, a tiny 32-bit RISC-V processor performs the Keccak operations. This RISC-V processor was the primary controller, so almost no additional area was required to support Keccak. However, these implementations require a significant number of cycles, as high as 60,000 cycles per permutation, to operate. Even at a million cycles, the Keccak hashing is adding noticeable latency overhead.

4.1 Keccak Description

Keccak is a family of hash functions that utilize a sponge construction. Notably, there is a large internal state where data is absorbed into and then the result is eventually squeezed out. All variants of SHA3 have an internal state of size 1,600 bits. SHAKE256 has a rate of 1,088 bits, which means that data is absorbed or squeezed out in chunks of 1,088 bits. 1,088 bits of the hash input are absorbed by XORing with the 1,088 least significant bits of the internal state. A squeeze operation simply retrieves the requested number of output bits (up to the rate) from least significance first. After each absorb and squeeze operation, a Keccak permutation function is performed on the entire state.

In SHA3-based hash functions, a Keccak permutation function consists of 24 rounds of the Keccak round function. Each Keccak round function consists of the Theta, Rho, Pi, Chi, and Iota functions. In general, these are based on simple bit manipulation operations, which are simple to implement in hardware. The Theta function computes the parity of various columns in the Keccak state. The Rho function computes a bitwise rotate of the Keccak state. The Pi function performs a state permutation. The Chi function performs a bitwise combination along the state. Lastly, the Iota function XORs a Keccak round constant into a word of the Keccak state.

4.2 Proposed Keccak Architecture

In this work, we have designed our own Keccak accelerator to achieve a balance between performance and area in a high-performance SIKE implementation. As is described above, the majority of operations in Keccak are basic logical operations. A round function is complete when the Theta, Rho, Pi, Chi, and Iota permutations are applied on the state to obtain a new state. One caveat to implementing the round function is that the Theta function

Table 5. Summary of SHAKE256 execution time in SIKE parameter sets. Total SIKE encapsulation and decapsulation latency (E + D) is for our one multiplier architecture.

SIKE Parameters	# Permutations	Total Cycles [cc×1000]	% of SIKE E+D cycles	# Slices	# BRAMs
Hardware SHAKE256					
SIKEp434	14	21	1.26%	177	1
SIKEp503	16	24	1.15%	177	1
SIKEp610	20	30	0.91%	177	1
SIKEp751	24	36	0.79%	177	1
Software SHAKE256					
SIKEp434	14	840	51.10%	0	~1.7
SIKEp503	16	960	46.47%	0	~1.7
SIKEp610	20	1,200	36.64%	0	~1.7
SIKEp751	24	1,440	31.92%	0	~1.7

requires some computations on a different Keccak plane from the other operations. As is specified by the Keccak team [10], this operation is known as Pretheta and is typically performed separately.

Our Keccak accelerator architecture is presented in Figure 6. The state size for SHAKE256 is 1,600 bits. We utilize the "state" and "tmp" blocks as registers to hold the entire state. These are stored in a 64 x 64 simple dual port block RAM. 25 addresses in the first 32 addresses are utilized by the state register while 25 addresses in the last 32 addresses are utilized by the temporary register.

The Keccak accelerator starts with an initial state stored in the "state" block. First, a 320-bit PreTheta is computed in 25 cycles and stored in the "PreTheta" block. The PreTheta value is a parity value that is reused in the Theta function. Next, the Theta, Rho, and Pi function are applied to the state ("ThetaRhoPi") and stored in the temporary register, which requires 25 cycles. The round function is then completed after the Chi and Iota function ("ChiIota") are applied to the temporary register, requiring 35 cycles. The ChiIota results are stored in the "state" block while the "PreTheta" value is simultaneously computed for the next round, which also requires 35 cycles. Overall, we have a 25 cycle initialization time followed by 24 rounds each requiring 60 cycles. There is additionally a 35 cycle overhead, most of which is coming from the RISC-V controller covered in Section 5, so this totals to about 1,500 cycles per permutation.

Based on the SIKE Round 3 parameter sets, we can quantify the total number of cycles occupied from the SHAKE256 hashing. This is summarized in Table 5, where we show how many total permutations are required for each NIST security level. Furthermore, we also calculate the percentage of time we are hashing for each security level based on our one multiplier architecture and results. There are a total of 14 permutations in the smallest parameter set and 24 permutations in the largest parameter set. However, because the large-degree isogeny computations scale slower than the SHAKE computations, we see that the percentage of total SIKE encapsulation and decapsulation latency drops from 1.26% for SIKEp434 to 0.79% for SIKEp751. Opting for a faster SHAKE256 accelerator would have cost significantly more LUTs and flip-flops for only a small improvement in SIKE performance.

To further express the need for a hardware SHAKE256 accelerator, we include the % of SIKE computations when a bare-bones RISC-V software processor performs the hash operations. To fully perform the Keccak operations across the 1,600 bit state, we require about 1.7 BRAMs. For SIKEp434, the hashing takes just above 50% of the total SIKE encapsulation + decapsulation latency. This is lower for larger parameter sets as the arithmetic becomes much more expensive, costing 32% of the total SIKE operation time for the NIST Level 5 SIKEp751. Even though

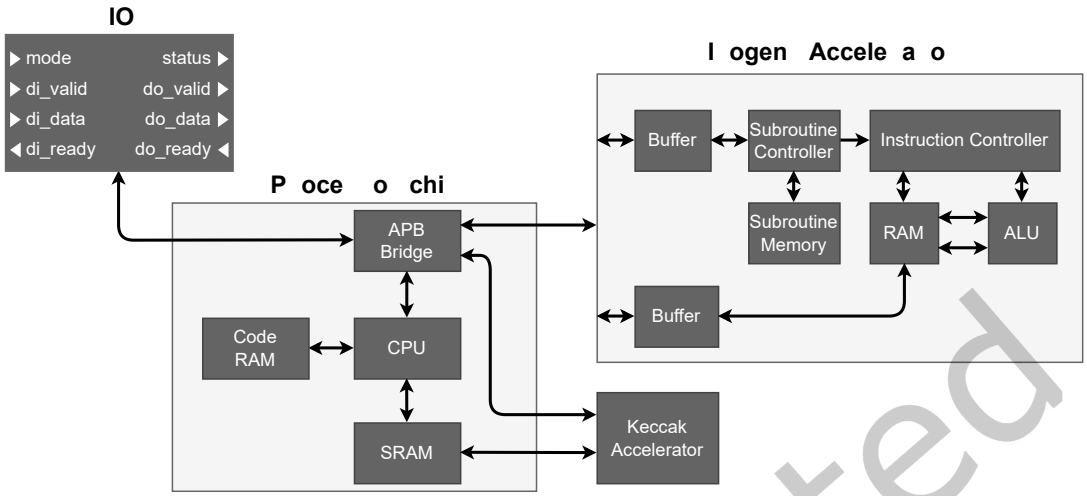


Fig. 7. High-level view of our SIKE design

the hardware accelerator is 40 times faster than the RISC-V implementation, we found that the area cost is similar between the two. The 177 slices needed for SHAKE256 are roughly equivalent to a BRAM, which is now needed by the software processor for the large internal state. Thus, the small SHAKE256 hardware module is extremely area-time efficient for the SIKE application.

5 A RISC-V TOP-LEVEL CONTROLLER

In this section, we discuss the top-level components of our SIKE design which is depicted in Figure 7. The goal of the top-level design is to efficiently control our isogeny accelerator and Keccak unit to facilitate the SIKE operation. Similar to [21, 24], we utilize a RISC-V processor as our top-level controller. Our primary difference is that all isogeny functions and subroutines have been moved into the hardware. These isogeny subroutines and functions cover a core set of basic elliptic curve group operations such as a small degree isogeny, point doubling, or point addition. Higher level algorithms and control are performed by the software. The RISC-V processor performs the following:

- Memory management: Simplifying loading and unloading data between the memories of the different hardware accelerators (Keccak accelerator and isogeny accelerator) as well as the IO.
- Program flow (transitioning between the different isogeny subroutines, looping through isogeny subroutines).
- Loading special cases for some isogeny subroutines:
  - Copying data between addresses in the isogeny RAM
  - Selecting pivot points in the large-degree isogeny
  - Selecting between two points in the three-point ladder
  - Selecting multiplication value in the $\mathbb{F}_p$ inversion sliding window method
- Modular correction to ensure the arithmetic result is between 0 and m, which is required at the end of each isogeny operation.
- Perform the comparison needed for key decapsulation.

The highlight of this RISC-V controller is that it greatly reduces the time needed to implement control logic at the cost of a slight increase in area compared to a pure hardware implementation.

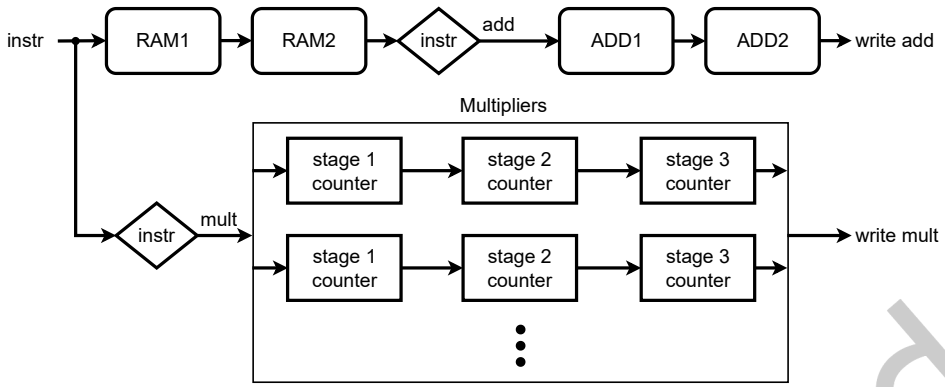


Fig. 8. Field arithmetic unit pipeline

5.1 Isogeny Accelerator

The isogeny accelerator is the primary computational workhorse in our SIKE architecture. There are two input/output buffers for the isogeny accelerator. The first is to receive subroutines from the master APB bus. Essentially, upon receiving a specific subroutine, the subroutine controller will retrieve the instructions from the subroutine memory and execute the instructions one by one. These subroutines represent a block of code that is executed through a series of $\mathbb{F}_p$ addition and $\mathbb{F}_p$ multiplication operations. Although SIKE performs $\mathbb{F}_{p^2}$ operations such as $\mathbb{F}_{p^2}$ addition, multiplication, squaring, or inversion, these can be broken down into $\mathbb{F}_p$ addition, subtraction, and multiplication.

The isogeny accelerator subroutines were created by breaking down each isogeny operation into a combination of $\mathbb{F}_p$ addition, $\mathbb{F}_p$ subtraction, and $\mathbb{F}_p$ multiplication and then using the scheduling algorithm from Farzam et al. [25]. This is a greedy scheduling algorithm that schedules the isogeny formulas using contraint programming to ensure a high throughput with various computing resources. A simple custom assembly language was created that had a strong correspondence to to the isogeny accelerator machine code instructions. The isogeny accelerator supports 3 assembly instructions:

(1) FPADD OUTPUT INPUT1 INPUT2 - Performs OUTPUT = INPUT1 + INPUT2 mod m
(2) FPSUB OUTPUT INPUT1 INPUT2 - Performs OUTPUT = INPUT1 - INPUT2 mod m
(3) FPMUL OUTPUT INPUT1 INPUT2 - Performs OUTPUT = INPUT1 $\times$ INPUT2 mod m

Likewise, the subroutine memory also holds the complex subroutines needed for large-degree isogenies. Our implementation uses the fastest known isogeny and scalar point multiplication formulas, which can be found in the SIKE submission [4]. These same formulas are found in the majority of the state-of-the-art hardware implementations. We summarize our formulas in Table 6 for our latencies with a one multiplier SIKEp434 implementation. Notably, the large-degree isogeny is performed by chaining together many small isogeny computations (GET2_ISO, GET3_ISO, GET4_ISO), evaluations (EVAL2_ISO, EVAL3_ISO, EVAL4_ISO), and scalar point multiplications (xDBL, xTPL, xQUAD), which accounts for about 80% of SIKE operations. The three-point differential ladder as proposed by Faz-Hernández et al. [27] is also used to generate a kernel point via an elliptic curve scalar point multiplication by using differential point arithmetic (xDBLADD) similar to the Montgomery powering ladder [49]. Lastly, the $\mathbb{F}_p$ inversion (INV) is composed of many $\mathbb{F}_p$ addition and multiplication operations in a sliding window fashion.

The isogeny accelerator's ALU contains an $\mathbb{F}_p$ addition unit and one or many $\mathbb{F}_p$ multiplication units. More multiplication units can take advantage of parallelism in some subroutines, resulting in a speedup at the cost of

Table 6. Summary of isogeny accelerator subroutines. Latency for SIKEp434 implementation with 1 multiplier.

SIKE Subroutine	Latency [cc]	Algorithm	Description
xDBL	206	Large-degree isogeny	Double a point, $Q = 2P$
xTPL	401	Large-degree isogeny	Triple a point, $Q = 3P$
xQUAD	414	Large-degree isogeny	Quadruple a point, $Q = 4P$
GET2_ISO	67	Large-degree isogeny	Compute a 2-isogeny, $\phi_2 : E/\langle P_2 \rangle$
GET3_ISO	188	Large-degree isogeny	Compute a 3-isogeny, $\phi_3 : E/\langle P_3 \rangle$
GET4_ISO	108	Large-degree isogeny	Compute a 4-isogeny, $\phi_4 : E/\langle P_4 \rangle$
EVAL2_ISO	174	Large-degree isogeny	Push a point through a 2-isogeny, $P' = \phi_2(P)$
EVAL3_ISO	235	Large-degree isogeny	Push a point through a 3-isogeny, $P' = \phi_3(P)$
EVAL4_ISO	273	Large-degree isogeny	Push a point through a 4-isogeny, $P' = \phi_4(P)$
xDBLADD	349	Kernel generation	Double and add a point, $R = 2P + Q$
INV	14,964	Large-degree isogeny Coeffient Recovery	Perform an $\mathbb{F}_p$ inversion, A^{-1}

more resources. Figure 8 illustrates our pipelines for field addition and field multiplication. Field addition is a simple linear process by which there are 2 cycles to fetch the instruction operands and 2 cycles to perform the arithmetic. The field multiplication pipeline has a three stage pipeline for each supported multiplier. The stage 1 counter is s cycles which indicates the number of cycles before a new multiplication can be interleaved. The stage 2 counter is an additional s cycles to complete the multiplication. Lastly, the stage 3 counter is 8 cycles to keep track of the 6 cycle pipeline from the multiplier as well as 2 cycles from fetching the instruction operands. Both operations require 1 cycle to store the result during which fetching new instructions is halted.

5.2 RISC-V SIKE Controller

The RISC-V processor acts as the top-level brains of our SIKE accelerator. Inside the RISC-V SIKE accelerator, we have connected the CPU's data bus and instruction bus using VexRiscV's native interface[1]. Specifically, the code RAM block is connected through both the data bus and the instruction bus while the SRAM and APB bridges are only connected through the data bus. The code RAM is designed to hold the text and data sections of the codes, which totals to 8 KB for all security levels. The SRAM holds the Keccak state and temporary registers as well as the BSS, heap (unused), and stack sections of the code. The SRAM is a 4 KB simple dual port RAM at all security level. This SRAM is actually implemented as two simple dual port RAMs of size 512×32. From the perspective of the CPU, it sees it as a single port RAM of size 1024×32. In a two-word aligned address, the first word maps to the first RAM and the second word maps to the second RAM. However, from the perspective of the Keccak accelerator, it sees this as a dual port RAM of size 512×64.

The APB bridge is used to connect the RISC-V processor to the isogeny accelerator, Keccak accelerator, and IO. For the isogeny accelerator, it controls both data and instruction interfaces of the isogeny accelerator. The data interface is implemented as a shift-register buffer to transfer data to and from the isogeny accelerator RAM. The instruction interface is used to send subroutines to the isogeny accelerator. This is buffered to increase throughput. To reduce the amount of instructions that need to be stored in the subroutine memory, we also send special addresses that are used in copying data between addresses in the RAM, selecting pivot points in the large-degree isogeny, selecting between two points in the three-point ladder, and multiplication in the $\mathbb{F}_p$ inversion sliding window method.

[1] https://github.com/SpinalHDL/VexRiscv

Table 7. Summary of BRAM usage across our implementations

SIKE Parameters	Isogeny Accelerator #BRAM	Controller #BRAM
SIKEp434	12.5	5
SIKEp503	14	5
SIKEp610	17	5
SIKEp751	21	5

The APB bridge's connection to the Keccak accelerator is only used to control the instruction interface of the Keccak accelerator as the data interface is handled by the SRAM block.

The APB bridge's connection to the IO is used for top-level SIKE control. Here, each signal indicates parts of an operation. The mode indicates whether to load constants, perform keygen, perform key encapsulation, or perform key decapsulation. The status indicates if the SIKE accelerator is ready to receive any data after the mode is changed. The `di` and `do` signals are used as a bus to exchange data, byte by byte. In the load constant mode, constants are loaded through `di`. In the keygen mode, we load Bob's secret key sk_B through `di` and return the resulting public key through `do`. In key encapsulation mode, we load the Alice's secret message msg_A followed by Bob's public key through `di` and get the resulting ciphertext followed by the shared secret through `do`. In key decapsulation mode, we load Bob's secret key followed by Alice's ciphertext through `di` and get the resulting shared secret through `do`.

5.3 Total BRAM Usage

Across each parameter set, we use five Xilinx 7th generation BRAMs (36k) for our top-level control. One BRAM is used for the CPU register file. Two BRAMs are used for the code RAM. One BRAM is used for SRAM. One last BRAM is used for the isogeny subroutine memory. Then, our remaining BRAMs are used in the isogeny accelerator's RAM to hold intermediate values as a register file in the SIKE computations. This total BRAM sizes are summarized in Table 7.

6 FPGA IMPLEMENTATION RESULTS

6.1 Summary of Results

In this section, we present and discuss the results of our FPGA implementation. In general, our code is written in SpinalHDL, a high-level HDL that can generate a Verilog implementation. The only exceptions are the highly optimized adder, multiplier, and Keccak blocks which are written in SystemVerilog. The architecture is implemented in Xilinx Virtex-7 xc7vx690tffg1157-3 as well as the Xilinx Artix-7 xc7a200tffg 1156-3 as these are used by the majority of SIKE implementations found in the literature. All results obtained are post-place and route.

Tables 8 and 9 summarize the timing and area results of our architecture, respectively. Specifically, we synthesized our designs on the Virtex-7 and Artix-7 FPGA platforms. Some other SIKE implementations include Xilinx UltraScale+ results, but our design was specially crafted to run on the Virtex-7/Artix-7 FPGA's carry chain and DSP units. Each implementation targets and supports only one parameter set. For instance, the NIST security level 1 implementations only support the SIKEp434 parameter set and the NIST security level 5 implementations only support the SIKEp751 parameter set. The multiplier and adder could be made generic to support all smaller parameter sets, but this would require additional control logic as well as arithmetic logic, resulting in extra area overhead as well as a slowdown. Within each security level, we also feature two implementations, targeting one or two replicated multipliers. More multipliers enable additional parallelism to achieve a speedup. An additional

Table 8. Timing results of SIKE accelerator in selected FPGA devices

Security Level	# Multipliers	Freq. [MHz]	#CC ($\times 10^6$) Keygen	Encap	Decap	E+D	Total time E+D [ms]
Virtex-7 FPGA							
1	1	275.5	0.502	0.796	0.869	1.664	6.04
	2	274.0	0.367	0.611	0.654	1.264	4.62
2	1	283.3	0.636	0.997	1.092	2.089	7.37
	2	273.2	0.459	0.763	0.815	1.578	5.78
3	1	284.1	0.921	1.627	1.677	3.304	11.63
	2	279.3	0.618	1.167	1.171	2.338	8.37
5	1	284.1	1.375	2.175	2.371	4.546	16.00
	2	279.3	0.906	1.483	1.595	3.078	11.02
Artix-7 FPGA							
1	1	189.8	0.502	0.796	0.869	1.664	8.77
	2	186.6	0.367	0.611	0.654	1.264	6.78
2	1	186.9	0.636	0.997	1.092	2.089	11.18
	2	185.5	0.459	0.763	0.815	1.578	8.51
3	1	180.2	0.921	1.627	1.677	3.304	18.34
	2	185.5	0.618	1.167	1.171	2.338	12.60
5	1	186.6	1.375	2.175	2.371	4.546	24.37
	2	178.6	0.906	1.483	1.595	3.078	17.24

Table 9. Area results of SIKE accelerator on selected FPGA devices

Security Level	# Multipliers	Freq. [MHz]	Area #FFs	#LUTs	#DSPs	#BRAMs	#Slices	Total time E+D [ms]
Virtex-7 FPGA								
1	1	275.5	10,700	10,915	84	17.5	3,857	6.04
	2	274.0	17,047	17,371	168		5,978	4.62
2	1	283.3	12,430	11,584	98	19.0	4,152	7.37
	2	273.2	18,505	18,616	196		6,541	5.78
3	1	284.1	15,219	13,790	120	22.0	4,962	11.63
	2	279.3	21,638	22,382	240		8,000	8.37
5	1	284.1	17,765	16,325	146	26.0	5,730	16.00
	2	279.3	27,015	26,837	292		9,556	11.02
Artix-7 FPGA								
1	1	189.8	10,755	10,412	84	17.5	3,721	8.77
	2	186.6	16,743	16,703	168		5,964	6.78
2	1	186.9	11,812	11,030	98	19.0	4,011	11.18
	2	185.5	18,374	18,040	196		6,499	8.51
3	1	180.2	14,489	12,972	120	22.0	4,754	18.34
	2	185.5	22,241	21,599	240		7,669	12.60
5	1	186.6	16,829	15,472	146	26.0	5,568	24.37
	2	178.6	26,383	25,776	292		9,340	17.24

Table 10. Area breakdown of SIKEp434 with 1 multiplier on Virtex-7 FPGA

Design Unit	#FFs	#LUTs	#DSPs	#BRAMs	#Slices
Keccak	597	500	0	0.0	177
RISC-V CPU	614	982	0	1.0	347
CPU Code RAM	-	-	-	2.0	-
CPU/Keccak Shared RAM	-	-	-	1.0	-
- Isogeny Accelerator	9,324	9,240	84	13.5	3,273
- Isogeny RAM	72	1,089	0	12.5	384
- Isogeny Subroutine Controller	137	47	0	1.0	41
- Isogeny Instruction Controller	139	455	0	0.0	269
- Isogeny Multiplier	5,821	5,973	84	0.0	2,373
- Isogeny Adder	2,663	1,291	0	0.0	902
Total	**10,700**	**10,915**	**84**	**17.5**	**3,857**

replicated multiplier speeds up the execution time by about 25% for the smallest parameter set SIKEp434 and by about 33% for the largest parameter set SIKEp751. However, the additional replicated multiplier also increases the total area by about 33% more slices and 100% more DSPs.

Table 10 provides an area breakdown for our SIKEp434 implementation with one multiplier on the Virtex-7 FPGA device. As we can see, the isogeny accelerator accounts for approximately 85% of the total slices. This is to be expected as a high-speed 434-bit modular addition or multiplication operation requires a significant usage of resources. The multiplier is the only block that requires DSPs and uses a total of 84 DSPs to efficiently perform the multiplications needed for Montgomery multiplication and reduction. Note that in the isogeny accelerator that some subcomponents share slices. As we have mentioned in the previous section, this design attempts to minimize large BRAM blocks and keeps to only 17.5 BRAMs.

6.2 Comparison to State-of-the-Art

Table 11 shows a detailed area and timing comparison among state-of-the-art SIKE implementations. Unfortunately, many of these implementation papers target various optimization metrics, making a fair comparison difficult. Nevertheless, this work's implementation shines as the fastest known implementation for SIKEp434, SIKEp503, and SIKEp610. This FPGA with 2 multipliers is about 2 milliseconds faster than the next best work for SIKEp434 and just under 4 milliseconds faster than the next best work for SIKEp610. For SIKEp751, only the work of Tian *et al. [64]* outperforms this work by 1.7 ms, however, at the cost of about three times as many resources.

This work's implementations, both the 1 and 2 multiplier variants, achieve the highest area-time product compared to the state-of-the-art. In order to equalize the impact of the various FPGA resources to area, we have used the equivalence 1 DSP = 100 Slices and 1 BRAM = 200 Slices. With this conversion, we can add up the equivalent number of slices for each of these implementations and multiply them by the SIKE execution time in milliseconds to get an area-time product. This area-time product is listed in the final column of Table 11. As we can see, the 1 multiplier implementation has a slightly better area-time product, but both implementations outperform all other implementations by at least 30%.

Across SIKE implementations, our implementations generally achieve the second highest frequency of 275 MHz, only losing out to Elkhatib *et al.* [24] by about 25 MHz. We attribute this high frequency as a result of using an incredibly optimized addition and multiplication unit. Most other implementations chose to use an addition unit with a larger critical path, slowing down their entire implementation.

Table 11. Comparison of area and timing results in Virtex-7 FPGA

Reference	Area					Time			
	#FFs	#LUTs	#Slices	#DSPs	#BRAMs	Freq [MHz]	E+D [$cc \times 10^6$]	[ms]	AT ($\times 10^{-3}$)
SIKEp434 (NIST Level 1)									
Koziel et al. [36]	23,819	21,059	8,121	240	26.5	168.4	1.91	11.3	422.9
Elkhatib et al. [22]	18,271	12,818	5,527	195	32.0	249.6	1.99	8	251.4
Massolino et al. [47] (S)	7,132	10,937	3,415	57	21.0	152.2	7.67	50.4	671.1
Massolino et al. [47] (F)	13,657	21,210	7,408	162	38.0	142.2	3.46	24.3	758.4
Elkhatib et al. [21]	-	-	4,611	78	34.5	243.6	4.68	19.2	370.8
Farzam et al. [26]	31,869	25,317	9,855	264	45.5	198.1	1.41	7.1	323.4
Elkhatib et al. [24]	14,666	7,604	3,942	78	29	303	4.39	14.5	254.4
Ni et al. [51]	30,327	29,468	9,578	108	23	251	1.65	6.6	164.1
This work (1 mult)	**10,700**	**10,915**	**3,857**	**84**	**17.5**	**275.5**	**1.66**	**6.0**	**95.2**
This work (2 mults)	**17,047**	**17,371**	**5,978**	**168**	**17.5**	**274.0**	**1.26**	**4.6**	**121.4**
SIKEp503 (NIST Level 2)									
Koziel et al. [36]	27,609	23,746	8,907	264	33.5	165.9	2.35	14.1	592.3
Elkhatib et al. [22]	19,935	13,963	6,163	225	34.0	243.7	2.65	10.9	386.5
Massolino et al. [47] (S)	7,132	10,937	3,415	57	21.0	152.2	9.06	59.5	792.2
Massolino et al. [47] (F)	13,657	21,210	7,408	162	38.0	142.2	4.08	28.7	895.7
Elkhatib et al. [21]	-	-	4,611	78	34.5	243.6	6.11	25.1	484.7
Farzam et al. [26]	36,731	27,148	10,707	312	47	197.9	1.72	8.68	445.3
Elkhatib et al. [24]	14,666	7,604	3,942	78	29	303	5.81	19.2	336.8
Ni et al. [51]	36,200	34,255	12,478	192	25	227	1.92	8.45	309.9
This work (1 mult)	**12,430**	**11,584**	**4,152**	**98**	**19**	**273.2**	**2.09**	**7.37**	**130.8**
This work (2 mults)	**18,505**	**18,616**	**6,541**	**196**	**19**	**284.1**	**1.578**	**5.78**	**173.1**
SIKEp610 (NIST Level 3)									
Koziel et al. [36]	33,297	28,217	10,675	312	39.5	165.8	3.59	21.6	1075.1
Elkhatib et al. [22]	26,757	16,226	7,461	270	38.5	239	4.26	17.8	750.5
Massolino et al. [47] (S)	7,132	10,937	3,415	57	21.0	152.2	16.3	107.2	1427.4
Massolino et al. [47] (F)	13,657	21,210	7,408	162	38.0	142.2	7.37	51.8	1616.6
Elkhatib et al. [21]	-	-	4,611	78	34.5	243.6	9.43	38.7	747.3
Farzam et al. [26]	44,753	30,562	12,848	384	50	183.4	2.40	13.1	801.1
Elkhatib et al. [24]	14,666	7,604	3,942	78	29	303	9.04	29.8	522.8
Ni et al. [51]	37,331	40,769	13,566	243	27.5	206	2.53	12.3	532.5
This work (1 mult)	**15,219**	**13,790**	**4,962**	**120**	**22**	**284.1**	**3.30**	**11.6**	**248.4**
This work (2 mults)	**21,638**	**22,382**	**8,000**	**240**	**22**	**279.3**	**2.34**	**8.37**	**304.7**
SIKEp751 (NIST Level 5)									
Koziel et al. [36]	50,079	39,953	15,834	512	43.5	163.1	4.55	27.8	2105.4
Farzam et al. [25]**	-	-	15,336	512	45.0	160.9	3.88	24.1	1820.4
Elkhatib et al. [22]	39,339	20,207	11,136	452	41.5	232.7	5.24	22.5	1454.3
Massolino et al. [47] (S)	7,132	10,937	3,415	57	21.0	152.2	27.34	179.6	2391.4
Massolino et al. [47] (F)	13,657	21,210	7,408	162	38.0	142.2	8.65	60.8	1897.4
Tian et al. [64]*	68,695	90,940	27,286	834	73.5	155.8	1.44	9.3	1166.1
Elkhatib et al. [21]	-	-	4,611	78	34.5	243.6	13.40	55.0	1062.1
Farzam et al. [26]	54,121	37,502	15,246	456	54	182.3	3.31	18.2	1301.1
Elkhatib et al. [24]	14,666	7,604	3,942	78	29	303	12.94	42.7	749.0
Ni et al. [51]	50,941	44,604	16,834	432	33	178	2.98	16.7	1115.5
This work (1 mult)	**17,765**	**16,325**	**5,730**	**146**	**26**	**284.1**	**4.55**	**16.0**	**408.5**
This work (2 mults)	**27,015**	**26,837**	**9,556**	**292**	**26**	**279.3**	**3.08**	**11.0**	**484.4**

* SIDH

** SIKE Round 1 Parameters

When considering memory efficiency, our implementation notably uses the fewest number of BRAM units for SIKEp434 and SIKEp503. Only the small implementation of Massolino *et al.* [47] requires fewer BRAM blocks for SIKEp610 and SIKEp751. This shows that our implementation minimizes the impact of memory storage in multiple places, including program ROM, isogeny accelerator RAM, and isogeny accelerator instructions.

7 CONCLUSION

In this paper, we designed and implemented a fast and efficient FPGA implementation of SIKE. Although SIKE is slower than most of its competitors, SIKE's speed continues to improve and SIKE shines with its extremely small public key sizes. Our hardware implementation achieves new speed records across most of SIKE's parameter sets while still maintaining an efficient area profile. With new area-time product records, efficient deployment of SIKE becomes more feasible. Our future work will involve exploring efficient implementation of compressed SIKE, for which no hardware implementation yet exists.

At the time of completing this work, SIKE was shown to be secure as detailed in Sections 1 and 2. However, recent attacks [11, 46, 53] have shown that SIKE is not secure and should not be used as a cryptosystem in the current state. This work might be beneficial for implementation of signatures based on isogenies such as SQISign by De Feo et al. [19].

8 ACKNOWLEDGMENT

The authors would like to thank the reviewers for their comments. This work is supported in parts by NSF grant 2101085.

REFERENCES

[1] Gora Adj, Daniel Cervantes-Vázquez, Jesús-Javier Chi-Domínguez, Alfred Menezes, and Francisco Rodríguez-Henríquez. 2018. On the Cost of Computing Isogenies Between Supersingular Elliptic Curves. Cryptology ePrint Archive, Report 2018/313. https://eprint.iacr.org/2018/313

[2] Gorjan Alagic, Jacob Alperin-Sheriff, Daniel Apon, David Cooper, Quynh Dang, John Kelsey, Yi-Kai Liu, Carl Miller, Dustin Moody, Rene Peralta, Ray Perlner, Angela Robinson, and Daniel Smith-Tone. 2020. Status Report on the Second Round of the NIST Post-Quantum Cryptography Standardization Process. (2020). NIST IR 8309.

[3] Mila Anastasova, Reza Azarderakhsh, and Mehran Mozaffari Kermani. 2021. Fast Strategies for the Implementation of SIKE Round 3 on ARM Cortex-M4. *IEEE Transactions on Circuits and Systems I: Regular Papers* (2021), 1–13. https://doi.org/10.1109/TCSI.2021.3096916

[4] Reza Azarderakhsh, Matthew Campagna, Craig Costello, Luca De Feo, Basil Hess, Aaron Hutchinson, Amir Jalali, David Jao, Koray Karabina, Brian Koziel, Brian LaMacchia, Patrick Longa, Michael Naehrig, Geovandro Pereira, Joost Renes, Vladimir Soukharev, and David Urbanik. 2020. Supersingular Isogeny Key Encapsulation. Submission to the NIST Post-Quantum Standardization Project. https://sike.org/

[5] Reza Azarderakhsh, Matthew Campagna, Craig Costello, Luca De Feo, Basil Hess, Amir Jalali, David Jao, Brian Koziel, Brian LaMacchia, Patrick Longa, Michael Naehrig, Joost Renes, Vladimir Soukharev, and David Urbanik. 2017. Supersingular Isogeny Key Encapsulation. Submission to the NIST Post-Quantum Standardization Project. https://sike.org/

[6] Reza Azarderakhsh, David Jao, Kassem Kalach, Brian Koziel, and Christopher Leonardi. 2016. Key Compression for Isogeny-Based Cryptosystems. In *Proceedings of the 3rd ACM International Workshop on ASIA Public-Key Cryptography* (Xi'an, China). 1–10.

[7] Reza Azarderakhsh, David Jao, Brian Koziel, Jason T. LeGrow, Vladimir Soukharev, and Oleg Taraskin. 2020. How Not to Create an Isogeny-Based PAKE. In *Applied Cryptography and Network Security - 18th International Conference, ACNS 2020, Rome, Italy, October 19-22, 2020, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 12146)*, Mauro Conti, Jianying Zhou, Emiliano Casalicchio, and Angelo Spognardi (Eds.). Springer, 169–186. https://doi.org/10.1007/978-3-030-57808-4_9

[8] Reza Azarderakhsh, Rami El Khatib, Brian Koziel, and Brandon Langenberg. 2021. Hardware Deployment of Hybrid PQC. Cryptology ePrint Archive, Report 2021/541. https://ia.cr/2021/541.

[9] U. Banerjee, S. Das, and A. P. Chandrakasan. 2020. Accelerating Post-Quantum Cryptography using an Energy-Efficient TLS Crypto-Processor. In *2020 IEEE International Symposium on Circuits and Systems (ISCAS)*. 1–5. https://doi.org/10.1109/ISCAS45731.2020.9180550

[10] Guido Bertoni, Joan Daemen, Michaël Peeters, Gilles Van Assche, and Ronny Van Keer. 2012. Keccak Implementation Overview. https://keccak.team/files/Keccak-implementation-3.2.pdf

[11] Wouter Castryck and Thomas Decru. 2022. An efficient key recovery attack on SIDH (preliminary version). Cryptology ePrint Archive, Paper 2022/975. https://eprint.iacr.org/2022/975 https://eprint.iacr.org/2022/975.

[12] Denis X. Charles, Kristin E. Lauter, and Eyal Z. Goren. 2009. Cryptographic Hash Functions from Expander Graphs. *Journal of Cryptology* 22, 1 (01 Jan 2009), 93–113. https://doi.org/10.1007/s00145-007-9002-x

[13] Andrew M. Childs, David Jao, and Vladimir Soukharev. 2014. Constructing Elliptic Curve Isogenies in Quantum Subexponential Time. *Journal of Mathematical Cryptology* 8, 1 (2014), 1–29.

[14] Craig Costello, David Jao, Patrick Longa, Michael Naehrig, Joost Renes, and David Urbanik. 2017. Efficient compression of SIDH public keys. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 679–706.

[15] Craig Costello, Patrick Longa, and Michael Naehrig. 2016. Efficient Algorithms for Supersingular Isogeny Diffie-Hellman. In *Advances in Cryptology - CRYPTO 2016 - 36th Annual International Cryptology Conference*. 572–601.

[16] Craig Costello, Patrick Longa, Michael Naehrig, Joost Renes, and Fernando Virdia. [n. d.]. Improved Classical Cryptanalysis of the Computational Supersingular Isogeny Problem. Cryptology ePrint Archive, Report 2019/298. https://eprint.iacr.org/2019/298.

[17] Jean-Marc Couveignes. 2006. Hard Homogeneous Spaces. Cryptology ePrint Archive, Report 2006/291.

[18] Luca De Feo, David Jao, and Jérôme Plût. 2014. Towards Quantum-Resistant Cryptosystems from Supersingular Elliptic Curve Isogenies. Cryptology ePrint Archive, Report 2011/506. *Journal of Mathematical Cryptology* 8, 3 (Sep. 2014), 209–247.

[19] Luca De Feo, David Kohel, Antonin Leroux, Christophe Petit, and Benjamin Wesolowski. 2020. SQISign: compact post-quantum signatures from quaternions and isogenies. In *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 64–93.
[20] Rami El Khatib, Reza Azarderakhsh, and Mehran Mozaffari-Kermani. 2019. Optimized Algorithms and Architectures for Montgomery Multiplication for Post-quantum Cryptography. In *International Conference on Cryptology and Network Security*. Springer, 83–98.
[21] Rami Elkhatib, Reza Azarderakhsh, and Mehran Mozaffari Kermani. 2021. Accelerated RISC-V for SIKE. In *28th IEEE Symposium on Computer Arithmetic, ARITH 2021, Lyngby, Denmark, June 14-16, 2021*. IEEE, 131–138. https://doi.org/10.1109/ARITH51176.2021.00035
[22] R. Elkhatib, R. Azarderakhsh, and M. Mozaffari-Kermani. 2020. Highly Optimized Montgomery Multiplier for SIKE Primes on FPGA. In *2020 IEEE 27th Symposium on Computer Arithmetic (ARITH)*. 64–71. https://doi.org/10.1109/ARITH48897.2020.00018
[23] Rami Elkhatib, Reza Azarderakhsh, and Mehran Mozaffari-Kermani. 2021. High-Performance FPGA Accelerator for SIKE. *IEEE Trans. Comput.* (2021), 1–1. https://doi.org/10.1109/TC.2021.3078691
[24] Rami Elkhatib, Brian Koziel, Reza Azarderakhsh, and Mehran Mozaffari Kermani. 2022. Accelerated RISC-V for SIKE (extended version). *IEEE Transactions on Circuits and Systems I: Regular Papers* (2022). https://doi.org/10.1109/TCSI.2022.3162626
[25] Mohammad-Hossein Farzam, Siavash Bayat-Sarmadi, and Hatameh Mosanaei-Boorani. 2020. Implementation of Supersingular Isogeny-Based Diffie-Hellman and Key EncapsulationUsing an Efficient Scheduling. *IEEE Transactions on Circuits and Systems I: Regular Papers* (2020).
[26] Mohammad-Hossein Farzam, Siavash Bayat-Sarmadi, Hatameh Mosanaei-Boorani, and Armin Alivand. 2021. Hardware Architecture for Supersingular Isogeny Diffie-Hellman and Key Encapsulation Using a Fast Montgomery Multiplier. *IEEE Transactions on Circuits and Systems I: Regular Papers* 68, 5 (2021), 2042–2050. https://doi.org/10.1109/TCSI.2021.3062871
[27] Armando Faz-Hernández, Julio López, Eduardo Ochoa-Jiménez, and Francisco Rodríguez-Henríquez. 2017. A faster software implementation of the supersingular isogeny Diffie-Hellman key exchange protocol. *IEEE Trans. Comput.* 67, 11 (2017), 1622–1636.
[28] Luca De Feo. 2017. Mathematics of Isogeny Based Cryptography. *CoRR* abs/1711.04062 (2017). http://arxiv.org/abs/1711.04062
[29] Steven D. Galbraith, Christophe Petit, Barak Shani, and Yan Bo Ti. 2016. On the Security of Supersingular Isogeny Cryptosystems. In *Advances in Cryptology - ASIACRYPT 2016*. 63–91. https://doi.org/10.1007/978-3-662-53887-6_3
[30] Steven D. Galbraith, Christophe Petit, and Javier Silva. 2017. Identification Protocols and Signature Schemes Based on Supersingular Isogeny Problems. In *Advances in Cryptology – ASIACRYPT 2017*. Cham, 3–33.
[31] Alexandre Gélin and Benjamin Wesolowski. 2017. Loop-Abort Faults on Supersingular Isogeny Cryptosystems. In *Post-Quantum Cryptography : 8th International Workshop, PQCrypto 2017*. 93–106. https://doi.org/10.1007/978-3-319-59879-6_6
[32] Lov K. Grover. 1996. A Fast Quantum Mechanical Algorithm for Database Search. In *Proceedings of the Twenty-Eighth Annual ACM Symposium on Theory of Computing* (Philadelphia, Pennsylvania, USA) *(STOC 1996)*. Association for Computing Machinery, New York, NY, USA, 212–219. https://doi.org/10.1145/237814.237866
[33] David Jao and Luca De Feo. 2011. Towards Quantum-Resistant Cryptosystems from Supersingular Elliptic Curve Isogenies. In *Post-Quantum Cryptography: 4th International Workshop, PQCrypto 2011*. 19–34. https://doi.org/10.1007/978-3-642-25405-5_2
[34] Samuel Jaques and John M. Schanck. [n. d.]. Quantum cryptanalysis in the RAM model: Claw-finding attacks on SIKE. Cryptology ePrint Archive, Report 2019/103. https://eprint.iacr.org/2019/103.
[35] A. Karmakar, S. Roy, F. Vercauteren, and I. Verbauwhede. [n. d.]. Efficient Finite Field Multiplication for Isogeny Based Post Quantum Cryptography. In *International Workshop on the Arithmetic of Finite Fields, WAIFI 2016*. to appear.
[36] B. Koziel, A. Ackie, R. El Khatib, R. Azarderakhsh, and M. M. Kermani. 2020. SIKE'd Up: Fast Hardware Architectures for Supersingular Isogeny Key Encapsulation. *IEEE Transactions on Circuits and Systems I: Regular Papers* (2020), 1–13.
[37] Brian Koziel, Reza Azarderakhsh, and David Jao. 2018. An Exposure Model for Supersingular Isogeny Diffie-Hellman Key Exchange. In *Topics in Cryptology - CT-RSA 2018 - The Cryptographers' Track at the RSA Conference 2018*. 452–469. https://doi.org/10.1007/978-3-319-76953-0_24
[38] Brian Koziel, Reza Azarderakhsh, and David Jao. 2018. Side-Channel Attacks on Quantum-Resistant Supersingular Isogeny Diffie-Hellman. In *Selected Areas in Cryptography – SAC 2017, 24th International Conference*. 64–81.
[39] Brian Koziel, Reza Azarderakhsh, and Mehran Mozaffari Kermani. 2018. A high-performance and scalable hardware architecture for isogeny-based cryptography. *IEEE Trans. Comput.* 67, 11 (2018), 1594–1609.
[40] Brian Koziel, Reza Azarderakhsh, and Mehran Mozaffari-Kermani. 2016. 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*. 191–206.
[41] Brian Koziel, Reza Azarderakhsh, Mehran Mozaffari-Kermani, and David Jao. 2017. Post-Quantum Cryptography on FPGA Based on Isogenies on Elliptic Curves. *IEEE Transactions on Circuits and Systems I: Regular Papers* 64, 1 (Jan 2017), 86–99. https://doi.org/10.1109/TCSI.2016.2611561
[42] Brian Koziel, Amir Jalali, Reza Azarderakhsh, David Jao, and Mehran Mozaffari-Kermani. 2016. NEON-SIDH: Efficient Implementation of Supersingular Isogeny Diffie-Hellman Key Exchange Protocol on ARM. In *Cryptology and Network Security: 15th International Conference, CANS 2016*. 88–103. https://doi.org/10.1007/978-3-319-48965-0_6

[43] Chunyang Liu, Jian Ni, Weiqiang Liu, Zhe Liu, and Máire O'Neill. 2018. Design and Optimization of Modular Multiplication for SIDH. In *2018 IEEE International Symposium on Circuits and Systems (ISCAS)*. 1–5. https://doi.org/10.1109/ISCAS.2018.8351082
[44] Weiqiang Liu, Jian Ni, Zhe Liu, Chunyang Liu, and Máire O'Neill. 2019. Optimized Modular Multiplication for Supersingular Isogeny Diffie-Hellman. *IEEE Trans. Comput.* 68, 8 (2019), 1249–1255. https://doi.org/10.1109/TC.2019.2899847
[45] W. Liu, Z. Ni, J. Ni, C. Rafferty, and M. O'Neill. 2020. High Performance Modular Multiplication for SIDH. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39, 10 (2020), 3118–3122. https://doi.org/10.1109/TCAD.2019.2960330
[46] Luciano Maino and Chloe Martindale. 2022. An attack on SIDH with arbitrary starting curve. Cryptology ePrint Archive, Paper 2022/1026. https://eprint.iacr.org/2022/1026 https://eprint.iacr.org/2022/1026.
[47] Pedro Maat C Massolino, Patrick Longa, Joost Renes, and Lejla Batina. 2020. A Compact and Scalable Hardware/Software Co-design of SIKE. *IACR Transactions on Cryptographic Hardware and Embedded Systems* (2020), 245–271.
[48] Peter L. Montgomery. 1985. Modular Multiplication without Trial Division. *Math. Comp.* 44, 170 (1985), 519–521.
[49] Peter L. Montgomery. 1987. Speeding the Pollard and Elliptic Curve Methods of Factorization. *Math. Comp.* (1987), 243–264.
[50] Michael Naehrig and Joost Renes. 2019. Dual Isogenies and Their Application to Public-Key Compression for Isogeny-Based Cryptography. In *Advances in Cryptology – ASIACRYPT 2019*, Steven D. Galbraith and Shiho Moriai (Eds.). Springer International Publishing, Cham, 243–272.
[51] Ziying Ni, Dur-e-Shahwar Kundi, Máire O'Neill, and Weiqiang Liu. 2022. A High-Performance SIKE Hardware Accelerator. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* (2022), 1–13. https://doi.org/10.1109/TVLSI.2022.3152011
[52] Geovandro C. C. F. Pereira and Paulo S. L. M. Barreto. 2021. Isogeny-Based Key Compression Without Pairings. In *Public-Key Cryptography – PKC 2021*, Juan A. Garay (Ed.). Springer International Publishing, Cham, 131–154.
[53] Damien Robert. 2022. Breaking SIDH in polynomial time. Cryptology ePrint Archive, Paper 2022/1038. https://eprint.iacr.org/2022/1038 https://eprint.iacr.org/2022/1038.
[54] Alexander Rostovtsev and Anton Stolbunov. 2006. Public-Key Cryptosystem Based on Isogenies. Cryptology ePrint Archive, Report 2006/145.
[55] Debapriya Basu Roy, Tim Fritzmann, and Georg Sigl. 2020. Efficient Hardware/Software Co-Design for Post-Quantum Crypto Algorithm SIKE on ARM and RISC-V based Microcontrollers. In *2020 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*. 1–9.
[56] Debapriya Basu Roy and Debdeep Mukhopadhyay. 2019. Post Quantum ECC on FPGA Platform. Cryptology ePrint Archive, Report 2019/568. https://eprint.iacr.org/2019/568.
[57] Hwajeong Seo, Mila Anastasova, Amir Jalali, and Reza Azarderakhsh. 2020. Supersingular Isogeny Key Encapsulation (SIKE)Round 2 on ARM Cortex-M4. *IEEE Trans. Comput.* (2020), 1–1. https://doi.org/10.1109/TC.2020.3023045
[58] Hwajeong Seo, Pakize Sanal, Amir Jalali, and Reza Azarderakhsh. 2020. Optimized Implementation of SIKE Round 2 on 64-bit ARM Cortex-A Processors. *IEEE Trans. Circuits Syst. I Regul. Pap.* 67-I, 8 (2020), 2659–2671. https://doi.org/10.1109/TCSI.2020.2979410
[59] Peter W. Shor. 1994. Algorithms for Quantum Computation: Discrete Logarithms and Factoring. In *35th Annual Symposium on Foundations of Computer Science (FOCS 1994)*. 124–134.
[60] Oleg Taraskin, Vladimir Soukharev, David Jao, and Jason T. LeGrow. 2021. Towards Isogeny-Based Password-Authenticated Key Establishment. *Journal of Mathematical Cryptology* 15, 1 (2021), 18–30. https://doi.org/doi:10.1515/jmc-2020-0071
[61] The National Institute of Standards and Technology (NIST). 2017–2018. Post-Quantum Cryptography Standardization. https://csrc.nist.gov/projects/post-quantum-cryptography/post-quantum-cryptography-standardization.
[62] Yan Bo Ti. 2017. Fault Attack on Supersingular Isogeny Cryptosystems. In *Post-Quantum Cryptography : 8th International Workshop, PQCrypto 2017, Utrecht, The Netherlands, June 26-28, 2017, Proceedings*. Springer International Publishing, Cham, 107–122. https://doi.org/10.1007/978-3-319-59879-6_7
[63] Jing Tian, Jun Lin, and Zhongfeng Wang. 2019. Ultra-Fast Modular Multiplication Implementation for Isogeny-Based Post-Quantum Cryptography. In *2019 IEEE International Workshop on Signal Processing Systems (SiPS)*. 97–102. https://doi.org/10.1109/SiPS47522.2019.9020384
[64] Jing Tian, Bo Wu, and Zhongfeng Wang. 2021. High-Speed FPGA Implementation of SIKE Based on an Ultra-Low-Latency Modular Multiplier. *IEEE Transactions on Circuits and Systems I: Regular Papers* 68, 9 (2021), 3719–3731. https://doi.org/10.1109/TCSI.2021.3094889
[65] Jacques Vélu. 1971. Isogénies Entre Courbes Elliptiques. *Comptes Rendus de l'Académie des Sciences Paris Séries A-B* 273 (1971), A238–A241.
[66] Youngho Yoo, Reza Azarderakhsh, Amir Jalali, David Jao, and Vladimir Soukharev. 2017. A Post-quantum Digital Signature Scheme Based on Supersingular Isogenies. In *Financial Cryptography and Data Security: 21st International Conference, FC 2017*. Springer International Publishing, Cham, 163–181. https://doi.org/10.1007/978-3-319-70972-7_9