# Accelerated RISC-V for Post-Quantum SIKE

Rami Elkhatib, Brian Koziel, Reza Azarderakhsh, *Member, IEEE*, and Mehran Mozaffari Kermani, *Senior Member, IEEE*

***Abstract*— In this work, we present a fast and area-efficient software-hardware implementation of the supersingular isogeny key encapsulation (SIKE) mechanism. Our software-hardware design achieves both the flexibility of software as well as the efficient performance of intense computations of hardware. In particular, our implementation takes advantage of new and highly optimized hardware modules for addition, multiplication, and hardware-software control, targeted at Xilinx FPGAs. In conjunction with a small RISC-V processor, we can support all four SIKE parameter sets. On a Virtex-7 FPGA, this implementation occupies 3,492 slices, 78 DSPs, and 29 BRAMs, to perform encapsulation and decapsulation over SIKEp434, SIKEp503, SIKEp610, and SIKEp751 in 14.5, 19.2, 29.8, and 42.7 ms, respectively. Despite supporting all four parameter sets, this design has the best area-time product of all isogeny accelerators in the literature.**

***Index Terms*— Isogeny-based cryptography, Montgomery multiplication, post-quantum cryptography, RISC-V, SIKE, software-hardware co-design.**

## I. Introduction

COMPUTER engineering studies and evaluates implementations based on software code as well as electronic hardware. General software computers feature Turing Complete coding to accomplish virtually any real-world purpose, just limited by computation size, memory size, and program size. Electronic hardware, such as that using digital transistors, can also accomplish virtually any real-world purpose, but is much slower to prototype and build. Between these types of devices, there is an inherent tradeoff between flexibility and performance. Digital hardware such as application specific integrated circuits (ASIC) can accomplish tasks and complex computations much faster and with less power and energy than a general-purpose software processor, but is limited only to such tasks.

Manuscript received November 12, 2021; revised January 14, 2022 and March 8, 2022; accepted March 8, 2022. Date of publication April 21, 2022; date of current version May 27, 2022. This work was supported in part by the Department of Defense (DoD) under Grant N001741910031. This article was recommended by Associate Editor J. Di. *(Corresponding author: Reza Azarderakhsh.)*

Rami Elkhatib and Brian Koziel are with the Computer and Electrical Engineering and Computer Science Department, Florida Atlantic University, Boca Raton, FL 33431 USA (e-mail: relkhatib2015@fau.edu; bkoziel2017@fau.edu).

Reza Azarderakhsh is with the Computer and Electrical Engineering and Computer Science Department and I-SENSE, Florida Atlantic University, Boca Raton, FL 33431 USA (e-mail: razarderakhsh@fau.edu).

Mehran Mozaffari Kermani is with the Computer Science and Engineering Department, University of South Florida, Tampa, FL 33620 USA (e-mail: mehran2@usf.edu).

Digital Object Identifier 10.1109/TCSI.2022.3162626

In the realm of cryptographic implementations, there are a variety of platforms to build and deploy a cryptosystem. For instance, a small ARM Cortex-M4 device can choose to employ a full cryptosystem like AES with only the ARM Cortex-M4 instruction set, or offload the computation to another chip or device that may be better equipped for it. Rather than be limited to strictly a hardware or software implementation, a hybrid software-hardware co-design implementation can be made such that the software processor acts as the main control unit and the digital hardware accelerates intense computations. By taking advantage of the flexibility of software and the efficiency of hardware, this new co-design implementation can achieve much better performance, power, and energy consumption.

Software-hardware co-design implementations are becoming even more necessary with the advent of post-quantum cryptography (PQC). It is well known that today's deployed public-key cryptosystems such as RSA or ECC will be broken once a large-scale quantum computer emerges. These schemes rely on hard mathematical problems such as factoring or the elliptic curve discrete logarithm that are resistant to attacks by classical computers but vulnerable to quantum computers. Namely, employing Shor's algorithm [1] on a large-scale quantum computer has been shown to break RSA or ECC. AES and other symmetric key cryptosystems are vulnerable to Grover's algorithm [2], but this is not a complete break and larger key or block sizes can allow their use in the post-quantum era.

With quantum fears in mind, NIST has created a post-quantum standardization process for standardizing new PQC algorithms [3] for use by the US government. Of the initial 3 rounds of scrutiny, the original 69 submissions has now dwindled to 15. With no final standard, an ASIC implementation of quantum-safe cryptography seems out of the question as most of the submissions have gone through revisions. So long as intense low level arithmetic is isolated, software-hardware co-design implementations shine here by achieving good performance and allowing flexibility if a scheme has changed, new improvements have been found, and so on.

Here, we propose and implement a software-hardware co-design implementation of NIST Round 3 alternative candidate SIKE. Based on isogenies of supersingular elliptic curves, SIKE features the smallest key sizes of all NIST PQC key establishment schemes. However, SIKE is among the slowest candidates, requiring a multitude of intense finite field computations. In our software-hardware implementa-

tion, we offload these expensive computations to highly optimized hardware. As an example, the finite field multiplication in the fastest ARM Cortex-M4 implementation to date requires 2,135 cycles for the NIST Level 5 parameter set [4], whereas our hardware co-processor requires 138 cycles, which is about 15 times faster. With software-hardware co-design, our implementation shows that even small embedded processors can achieve competitive performance. Furthermore, this flexibility allows us to support new schemes, such as the SIKEX hybrid key exchange based on SIKE and ECDH [5].

*Related Work:* The first software-hardware co-design implementation of SIKE was implemented by Massolino *et al.* [6] as a compact design. Based upon a simple custom-designed processor, this work could swap between each SIKE parameter set and even support newly proposed SIKE parameters with primes up to 1,008 bits. In [7], Banerjee *et al.* implemented a software-hardware co-design of SIKE by using a RISC-V processor to control an elliptic curve accelerator. Since this was not fine-tuned for SIKE computations, the design is two orders of magnitude slower than state-of-the-art hardware-based SIKE implementations. In [8], Banerjee *et al.* also utilized a software-hardware co-design methodology to efficiently support multiple lattice-based cryptosystems. Lastly, in [9], Roy *et al.* also provided an efficient hardware/software co-design with SIKE on both ARM and RISC-V based microcontrollers, achieving a nice speedup over software only processors.

This work features a new software-hardware co-design of SIKE utilizing a RISC-V processor as the main controller. Based on the structure of SIKE parameters, we have optimized the hardware performance of low-level arithmetic for FPGA devices. As a bridge between the RISC-V software core and our hardware accelerator, we utilize a small APB bus to exchange data between the two cores and facilitate the full SIKE operation. This is an extension of our previous work, "Accelerated RISC-V for Post-Quantum SIKE." [10] Upon these preliminary results, we have developed further optimizations to the addition, multiplication, and control units, achieving a 30% improvement in performance, while also occupying 17% fewer slices. Our implementation supports all SIKE implementations up to 752-bit primes and also features the best area-time product of any design in the literature.

*Our Contributions:*

- We propose and implement highly optimized hardware accelerators for Montgomery multiplication and addition over SIKE primes
- We design a new and more efficient dedicated instruction controller to efficiently hand off intense field arithmetic from a RISC-V core to our specialized hardware
- We architect and implement our software-hardware co-design over all four SIKE parameter sets, achieving the best area-time trade-off in the literature

The organization of the paper is as follows. In Section II, we give an overview of isogeny-based cryptography and the NIST PQC candidate SIKE. In Section III, we propose our highly optimized designs for field addition and multiplication in SIKE. In Section IV, we propose our hardware-software co-design to achieve efficient SIKE operation. In Section V, we implement our design on FPGA and present our results. In Section VI, we give our final thoughts and discuss future work.

## II. Preliminaries

Here, we provide some preliminaries related to isogeny-based cryptography and the NIST PQC isogeny-based submission SIKE [11]. Roughly, isogeny-based cryptography is an extension of elliptic curve cryptography (ECC). Rather than utilize point arithmetic on a single elliptic curve as is the case for ECC, isogeny-based cryptography utilizes maps between elliptic curves on top of point arithmetic. We point the reader to [12] for a more in-depth review of elliptic curves and [13] for isogeny fundamentals.

### A. Isogenies on Elliptic Curves

*1) Elliptic Curves:* Isogeny-based cryptography has focused on the use of isogenies as maps between elliptic curves to achieve some cryptographic application. Similar to ECC, an elliptic curve is also defined over a finite field. 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:

$$E/\mathbb{F}_q : y^2 = x^3 + ax + b$$

where $a, b, x, y \in \mathbb{F}_q$. This collection forms an abelian group over addition. Scalar point multiplications $Q = kP$, where $k \in \mathbb{Z}$ and $P, Q \in E$ forms the basis of ECC. As the order of $E$ grows very large, it becomes infeasible to solve for $k$ when given $Q$ and $P$. However, Shor's algorithm [1] provides a polynomial time algorithm to break this hard problem with a sufficiently large quantum computer.

*2) Isogenies:* Isogenies on elliptic curves are an extension of ECC. An elliptic curve isogeny over $\mathbb{F}_q$, $\phi : E \rightarrow E'$ is defined as a non-constant rational map from $E(\mathbb{F}_q)$ to $E'(\mathbb{F}_q)$ that preserves the point at infinity. Rather than performing scalar point multiplication on a single elliptic curve, isogenies of elliptic curves move between elliptic curves in such a manner that can be difficult for quantum computers when constructed correctly. Isogenies move between isomorphism classes, which can be identified via their $j$-invariant. A unique isogeny can be computed over a kernel, $\phi : E \rightarrow E/\langle ker \rangle$, by using Vélu's formulas [14]. The degree of an isogeny is its degree as a rational map. We can efficiently compute large-degree isogenies of the form $\ell^e$ by chaining $e$ isogenies of degree $\ell$.

### B. Supersingular Isogeny Key Encapsulation

The supersingular isogeny key encapsulation (SIKE) mechanism [11] is the only isogeny-based cryptosystem submitted to the NIST PQC standardization process. Currently a third round alternative, SIKE prominently features the smallest public keys of all key encapsulation mechanisms (KEMs), has

a straightforward parameter selection, simple generic attacks, and no decryption errors. Notably, advances in supersingular isogeny cryptanalysis revealed that SIKE primes were too conservative, and they were reduced in the second round. The main downside of SIKE is its slow performance, as it requires a large number of finite-field addition and multiplication operations over 434-bit primes at NIST Level 1 or 751-primes at NIST Level 5.

*1) History:* SIKE comes from a rich history of new emerging applications based on isogenies of elliptic curves. The use of isogenies for a cryptosystem was first proposed independently by Couveignes [15] and Rostovtsev and Stolbunov [16] in 2006. Initially, they proposed an isogeny-based key-exchange relying on the hardness of computing isogenies between ordinary curves. Charles *et al.* [17] also proposed a new isogeny-based hash function based on the hardness of computing isogenies between supersingular elliptic curves in 2009. It was not until 2010 that Childs *et al.* [18] broke the initial isogeny-based key-exchange algorithms by proposing a new quantum algorithm to compute isogenies between ordinary elliptic curves in subexponential time. Next, in 2011, Jao and De Feo [19] proposed a new isogeny-based key exchange that was instead protected by the difficulty to compute isogenies between elliptic curves, called the supersingular isogeny Diffie-Hellman (SIDH) key exchange. SIDH is the precursor to SIKE. Notably, SIKE is the 2017 NIST submission that applies the Hofheinz, Hövelmanns, and Kiltz transform [20] to the original supersingular isogeny public-key encryption scheme proposed by Jao and De Feo [19]. With new and focused research on isogenies, optimizations/implementations [4]–[6], [10], [21]–[32] and security analysis [33]–[40] continue to emerge.

*2) SIKE Overview:* SIKE is an IND-CCA KEM based on the hardness of computing isogenies between supersingular elliptic curves. In the third round of the NIST PQC standardization process, there are four parameter sets which are summarized in Table I. Each parameter set targets a NIST security level from 1 to 5. NIST security level 1 is hard to break as an exhaustive key search attack on AES128, level 2 is as hard to break as finding a SHA256 collision, level 3 is as hard to break as an exhaustive key search attack on AES192, and level 5 is as hard to break as an exhaustive key search attack on AES256. Each of these parameter sets feature the smallest public keys of all NIST PQC KEMs and almost the smallest ciphertext. In addition to the regular SIKE operation, there are compressed SIKE variants that reduce the size of public keys and ciphertexts by slightly less than half.

*3) KEM Operation:* In the KEM scenario, Alice and Bob are attempting to agree on a shared secret that can be used for a secure session. Bob initiates the KEM by performing key generation, which generates a secret key and a public key for Bob. Bob shares this public key over a public channel so that Alice can proceed by performing key encapsulation over Bob's public key, which generates a ciphertext and shared secret for Alice. Alice then sends this ciphertext back to Bob over a public channel. Bob finishes the key establishment by performing key decapsulation over Alice's ciphertext with his own secret key. If the operation was performed correctly, Bob will compute the same shared secret that Alice did and the two can continue with encrypted communications such as with AES. If not, Bob still generates a seemingly random secret that will not reveal his secret key.

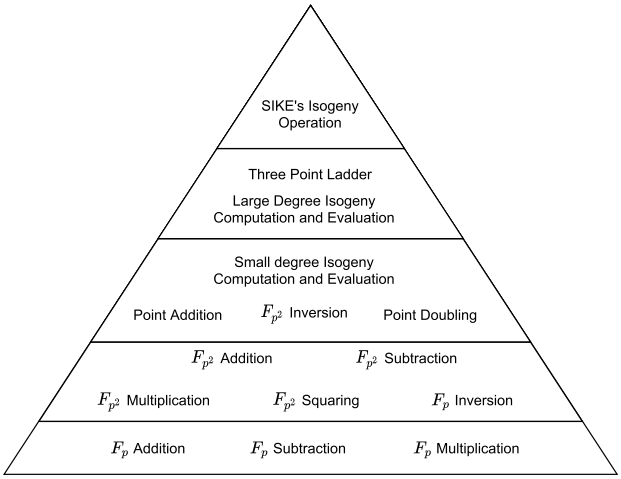


Fig. 1. Computational hierarchy for SIKE's isogeny computations.

*4) SIKE Arithmetic:* SIKE can be made up of a large-degree isogeny as well as the SHAKE256 hashing operation. Of these two, the large-degree isogeny generally requires the majority of the total SIKE computation time. Thus, unlike most lattice and other PQC schemes, a fast SHAKE256 module is not required. Rather, an efficient implementation of SIKE is made by efficiently implementing the algorithms and subroutines needed to chain together a large-degree isogeny. The computations required for this isogeny operation are shown in Figure 1. At a low level, isogenies of supersingular elliptic curves are composed of many $\mathbb{F}_{p^2}$ quadratic extension field arithmetic operations, which are then constructed via $\mathbb{F}_p$ addition, subtraction, and multiplication. Here, $p$ is a SIKE prime which is also included in Table I for each parameter set. In the following sections, we propose efficient modules to perform these finite field operations as well as process many such operations to facilitate SIKE algorithms.

## III. Proposed Hardware Acceleration of Intense Arithmetic Operations

This section presents our optimized hardware for finite field addition and finite field multiplication, which provide the lowest level arithmetic needed for SIKE. These finite field operations are defined over a prime field, so the functionality is similar to modular addition and modular multiplication. We summarize our field arithmetic operations and compare them with [10] in Table II. Notably, our design achieves a much higher frequency to expedite the arithmetic over the prior design.

### A. Proposed Finite Field Adder

Given elements $a, b, c \in \mathbb{F}_p$, finite field addition performs $a + b = c$, where all values are mod $p$. Since inputs $a$ and

TABLE I
SIKE PARAMETER SETS FOR EACH NIST SECURITY LEVEL [11]. ALL SIZES ARE IN BYTES

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

TABLE II
LATENCY OF ARITHMETIC OPERATIONS IN OUR ALU ARCHITECTURE ON A VIRTEX-7 FPGA

| Prime | Max Freq. (MHz) | Latency ($cc$) | | | Latency ($ns$) | | |
|---|---|---|---|---|---|---|---|
| | | $\mathbb{F}_p$ Add | $\mathbb{F}_p$ Multiplication | | $\mathbb{F}_p$ Add | $\mathbb{F}_p$ Multiplication | |
| | | | Mult. | Interleave | | Mult. | Interleave |
| Prior Work [10] | | | | | | | |
| SIKEp434 | 243.6 | 2 | 81 | 52 | 8.2 | 333 | 213 |
| SIKEp503 | | | 93 | 60 | | 382 | 246 |
| SIKEp610 | | | 111 | 72 | | 456 | 296 |
| SIKEp751 | | | 138 | 90 | | 567 | 369 |
| **This Work** | | | | | | | |
| SIKEp434 | 303.0 | 2 | 81 | 52 | 6.6 | 267 | 172 |
| SIKEp503 | | | 93 | 60 | | 307 | 198 |
| SIKEp610 | | | 111 | 72 | | 366 | 238 |
| SIKEp751 | | | 138 | 90 | | 455 | 297 |

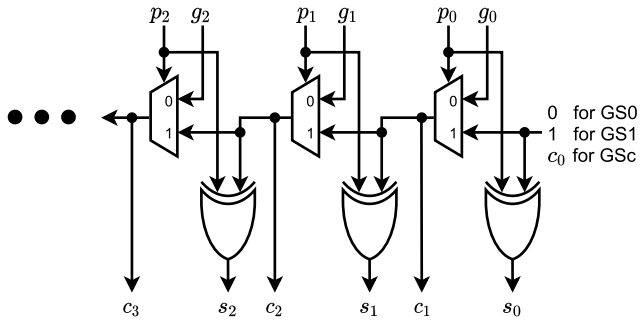


Fig. 2. Xilinx FPGAs use a Manchester carry chain. We call this a GS block.

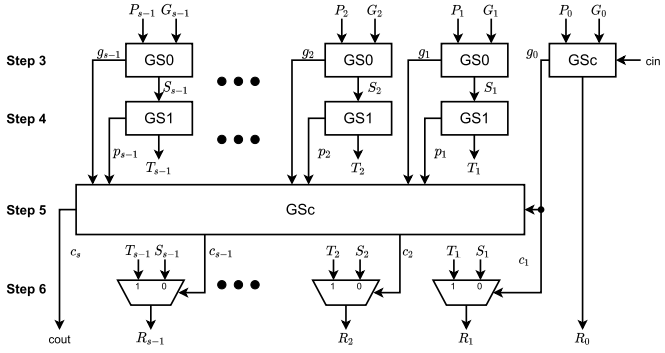


Fig. 3. Proposed fast addition units for SIKE on Xilinx FPGAs.

$b$ are already reduced modulo $p$, finite field addition can be implemented by performing $a + b = c$ and then performing a conditional subtraction $c' = c - p$, if $c \geq p$. Based on the finite field multiplier proposed in Section III-B, we make a slight caveat to this where general arithmetic is mod $2p$, which is common for SIKE implementations. Thus, we are performing $a + b = c$ with the conditional subtraction $c' = c - 2p$, if $c \geq 2p$. Finite field subtraction is performed in a similar way with $a - b = c$ and the conditional addition $c' = c + 2p$, if $c < 0$. We perform a final correction to reduce modulo $p$ at the final step of a protocol. Our new proposed finite field addition unit utilizes a specialized architecture to achieve high performance, throughput, and reasonable area.

*1) Xilinx Carry Chains:* In this implementation, we are targeting Xilinx FPGA devices, so we are utilizing their internal fast carry-chains for fast addition. The carry chain used in Xilinx FPGAs is shown in Figure 2. This is based on a standard Manchester carry chain. When performing $A + B$, where $A$ and $B$ are $n$ bits, this design would require pushing $A \oplus B$ into $p_i$ and $A$ into $g_i$ for $0 \leq i < n$. We limit the use of $i$ for the Manchester carry chain. This carry chain then produces a sequence of carry and sum bits. $s_i$ is the propogated sum and $c_i$ is the propagated carry. For the following discussion, we call this figure a GS block for Generate Sum. Lastly, GS0 means you have a carry-in of "0", GS1 for a carry-in of "1", and GSc for a carry-in of $c_0$.

We now present our new modular adder. This design uses the parallel prefix network adder from [41] as a base and applies three innovations to better suit SIKE arithmetic. Given the fine-tuned granularity of our optimization, we describe the adder through its operation in the following steps. Specifically, our optimizations are described in Step 5, Step 6, and a special application to SIKE addition arithmetic. This architecture is shown in Figure 3.

*Step 1:* In Step 1, we are simply acquiring the inputs needed to perform addition. For the standard Manchester carry chain, $G$ is one of the inputs and $P$ is the XOR of the addends.

*Step 2:* Next, we break $P$ and $G$ into blocks. These blocks do not need to be of the same size. If the least significant blocks of the operands are smaller, then the carry-look ahead is entered earlier. If the most significant blocks of the operand are smaller, then there will be less routing delays post-carry look ahead. In our design, we use a pipeline stage, so a single block size was used. For clarity, there are at most $s$ elements in the adder and we use the variable $j$ to denote an output for $0 \leq j < s$.

*Step 3:* In this step, we apply the GS function on each $G$ and $P$ block to obtain propagated sums and carries. In Figure 3, this refers to the top row of computations. The least significant block utilizes a GSc block with a carry-in. Each of these GS computations produce propagated sums $S_j$ and the last propagated carry $g_j$.

*Step 4:* Step 4 moves to the second row of computations, which is again the GS computation. Here, the $S$ block acts inputs for $g_i$ and $p_i$ in the Manchester chains to produce the propagated sum $T_j$ and the last propagated carry $p_j$. From these two rows of computations, we have values $S_j, T_j, p_j$, and $g_j$. The state-of-the-art design [41] does not generate $T_j$ which we utilize as part of one innovation in Step 6. Of these variables, $g_j$ tells us if the block generates a carry for the next block. $p_j$ tells us whether or not all bits of the sum $S_j$ in the block are 1. This means that if $p_j = 1$, then the carry from the previous block is propagated to the next block. The end goal here is to get $c_j = g_j | p_j \,\&\, c_{j-1}$, where $c_j$ is the propagated carry. One thing to note here is that unless the GS block in Step 3 has a carry-in of 1, $g_j$ and $p_j$ cannot both be 1 as all GS blocks are GS0, there are no $g_j$ and $p_j$ carry chain inputs where both are 1. Lastly, $p_0$ does not exist because the first block is already fully propagated, i.e. $g_0$ is the propagated carry for the first block.

*Step 5:* Step 5 is the first innovation, by providing a new method to get the propagated carry which is shown in the third row of Figure 3. The state-of-the-art utilizes a parallel prefix carry-look ahead unit based on Kogge-Stone and Brent-Kung. These units require a large amount of bitwise-operations, thus consuming many LUTs in an FPGA. Instead, we simply apply another GSc block to each $g_j$ and $p_j$ with $g_0$ as the $c_0$ input. This GS block produces unused propagated sum and propagated carry $c_j$. The GSc unit from the figure gives $c_j = (\sim p_j \,\&\, g_j) \mid (p_j \,\&\, c_{j-1}\}$. As we mentioned in Step 4, both $p_j$ and $g_j$ cannot both be "1", so this equation collapses to $c_j = g_j | p_j \,\&\, c_{j-1}$ which is what we are searching for from Step 4.

*Step 6:* Step 6 then provides a second innovation to find the final sum in the fourth row of Figure 3. The state-of-the-art utilizes an additional GSc block with $S_j$ as both inputs and $c_j$ as the propagated carry. Instead, we observed that $S_j$ from Step 3 is the sum if $c_j = 0$ and $T$ from Step 4 is the sum if $c_j = 1$. Thus, we simply use a 2:1 multiplexer where $c_j$ is the select to select the final result $R$ from $S$ or $T$. The final propagated carry cout in the figure is the last propagated carry from carry-look ahead unit from Step 5.

*2) Applying to SIKE:* We also provide a third innovation and optimization when we apply this adder for SIKE. For finite field addition and subtraction, our SIKE adder produces results $a \pm b \mp 2p$. For a high frequency implementation, we compute $a \pm b$ and $a \pm b \mp 2p$ independently over 2 cycles. To compute $a \pm b$, we use the adder architecture described above with $P = a \oplus b \oplus sub$, $G = a$, and $cin = sub$, where $sub$ is '1' if a subtraction is performed and '0' otherwise.

When computing $a \pm b \mp 2p$, we apply a new trick to perform two additions/subtractions simultaneously. The classical way to perform this computation is to compute $S = a \pm b$ and then $R = S \mp 2p$. However, to perform this in 2 cycles would severely limit the operating clock frequency. Instead, our proposed methodology gets two partial sums and feeds them to the proposed adder. For the sake of simplicity, let us compute $a + b + m$. Let us take two consecutive bits from each input $a$, $b$, and $m$. Assume that we have $a_1, b_1, m_1 = '0'$, while $a_0, b_0, m_0 = '1'$. In this case the result will be $r_0 = '1'$ and $r_1 = '1'$ and no bits are propagated to $r_2$. From here, we achieve two partial sums:

1) Zero out all even bits. Each odd-even pair will have its bits coming from its corresponding odd bits.
2) Zero out all odd bits. Each even-odd pair will have its bits coming from its corresponding even bits.

Once we have these two partial sums, we can add them together using the proposed adder.

Lastly, our design uses two pipeline stages to achieve a high frequency of 300 MHz on Xilinx Virtex-7. In our explained Steps, our first pipeline registers $g_j$, $p_j$, $S_j$ and $T_j$ at the end of Step 4 and then our second pipeline registers the output from Step 6. Since we compute two addition/subtraction operations for modular arithmetic, we get two final carries and two sums. The final carries are used to select the final result between $a \pm b$ and $a \pm b \mp 2p$.

**Algorithm 1** Proposed Montgomery Multiplication

**Input** : $m = 2^{e_A} \cdot 3^{e_B} < 2^{K-2}$, $R = 2^K$, $w$, $s$, $K = w \cdot s$, $s_A = \lfloor 2^{e_A}/w \rfloor$, $a, b < 2m - 1$
**Output**: `MontMult`$(a, b)$

```
1  T ← 0
2  for i ← 0 to s − 1 do
3      for j ← 0 to s − 1 do
4          U[j] ← a[i] · b[j]                          } Mul
5      (C, S) ← T[0] + U[0]
6      q ← S                                           } First Acc
7      for j ← s_A to s − 1 do
8          U[j] ← U[j] + q · m[j]                      } Red
9      for j ← 0 to s − 1 do
10         (C, S) ← T[j] + U[j] + C
11         T[j − 1] ← S                                } Acc
12     (C, S) ← C
13     T[s − 1] ← S                                    } Final Acc
14 return T
```

### B. Proposed Finite Field Multiplier

Given elements $a, b, c \in \mathbb{F}_p$, finite field multiplication performs $a \times b = c$, where all values are mod $p$. Even if inputs

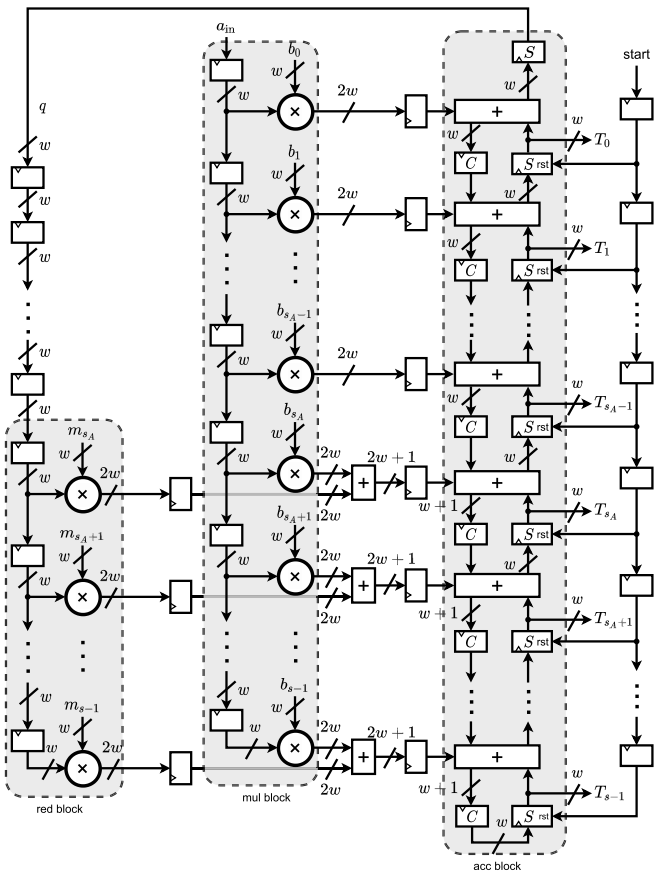


Fig. 4. Architecture of multiplier core.

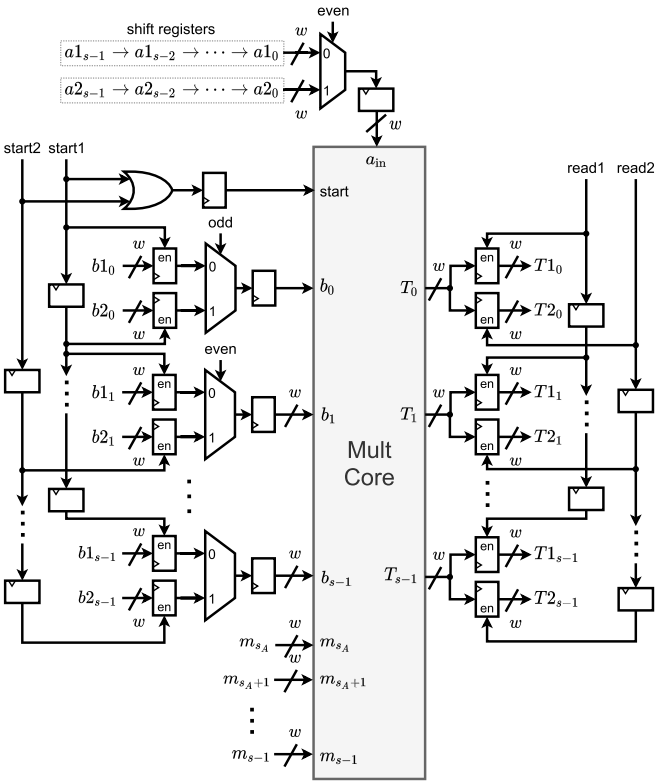


Fig. 5. Dual multiplier IO for the multiplier core.

$a$ and $b$ are already reduced modulo $p$, a simple multiplication between $a$ and $b$ produces a value that can be up to twice the bit-length of $p$. To solve this problem, we use the specialized Montgomery multiplication [42] method that combines multiplication and reduction. By trading expensive trial divisions for simple shifts, Montgomery multiplication is the most popular modular multiplication method for SIKE. In this work, we improve the Montgomery multiplication proposed in [26] to consume less DSPs and operate at higher frequency while keeping the same number of interleave and multiplication cycles.

Our Montgomery multiplier operably performs the functionality shown in Algorithm 1. Similar to [21], [26], we utilize a systolic architecture where processing elements compute partial products and sums. As is shown in Figure 4, we break Algorithm 1 into multiplication "mul", accumulation "acc", and reduction blocks "red". This proposed architecture computes the result in $w$ bits at a time. There are a total of $s$ processing elements. Compared to our prior work [10], we make a small optimization in the reduction by feeding it $p+1$ rather than $p$, which saves an adder.

The multiplication block simply computes the product $U[j] = a[i] \cdot b[j]$. This corresponds to Line 4 in Algorithm 1. Input $a$ is loaded serially every two cycles and input $b$ is loaded in parallel. Each processing element within the systolic architecture computes $w$ bit operations, so the end product is $2w$ bits which is sent to the accumulation block.

The accumulation block sums and propagates the Montgomery multiplication values between each iteration $i$. First, the accumulation block, utilizes the first accumulation result $T[0] + U[0]$ to compute the Montgomery quotient $q$ that can then be pushed to the reduction block. Since Montgomery multiplication produces values with zeroes in the $w$ least significant bit positions, the accumulation block propagates the least significant $w$ bits $S$ backwards and the remaining bits $C$ forward along the systolic array. The first accumulation block, corresponding to Lines 5-6 of Algorithm 1, computes the quotient $q$ for the reduction block and the first $C$ for the accumulation block. The remaining elements perform Lines 10-11 of Algorithm 1 with the final $C$ loaded to $S$ of the same element similar to Lines 12-13. The first word of the final product is obtained from $S$ of the second element after $2s$ cycles and the following words are obtained from the following elements in the following cycles until a total of $3s$ cycles are passed.

Similar to the multiplication block, the reduction block computes the product of a quotient $q$ obtained from the accumulation block and $m = p + 1$ with $m$ loaded in parallel and $q$ loaded serially. Each element in the reduction block is added to its corresponding element in the multiplication block to perform one iteration of Line 8 in Algorithm 1. Since the first $e_A$ bits of $m$ are 0, the first $s_A = \lfloor e_A/w \rfloor$ blocks are skipped. Furthermore, $q$ is delayed such that the first reduction element aligns with the $s_A$'th multiplication element when the addition is performed.

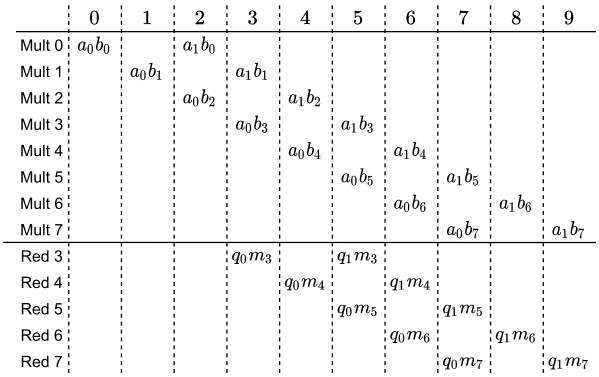


Fig. 6. Waveform showing the output at each multiplication and reduction block for the first 2 digits of operand $a$ and quotient $q$. Columns indicate cycles. Rows indicate blocks according to Figure 4. Parameters: $s = 8$ and $s_A$=3.

TABLE III
DSP COST OF MONTGOMERY MULTIPLIER FOR $w = 17$

| Reference | SIKEp434 | SIKEp503 | SIKEp610 | SIKEp751 |
|---|---|---|---|---|
| Elkhatib *et al.* [26] | 65 | 75 | 90 | 113 |
| **This work** | **40** | **46** | **55** | **69** |

We further include Figure 6 as an additional method to illustrate the functionality of our multiplier core with cycles. In this figure, we set $s = 8$ and $s_A = 3$. The Mult 0-7 and Red 3-7 represent the multiplication circle blocks in Figure 4 for the multiplication and reduction blocks, respectively. For instance, Mult 0 refers to the multiplication block where the operand is $b_0$, Mult 3 refers to the multiplication block where the operand is $b_{s_A}$, and Red 3 refers to the reduction block where the operand is $m_{s_A}$. For simplicity, Figure 6 only shows the first 2 digits of operand $a$ and quotient $q$. First, we highlight that the operand $a_i$ moves from Mult 0 to Mult 1 and so on each cycle, just as the quotient $q_i$ moves from Red 3 to Red 4 and so on each cycle. Second, we reiterate that each $a_i$ and $q_i$ are aligned such that $a_i$ enters the lower multiplication blocks, aka mult $s_A$ and beyond, at the same time $q_i$ enters the reduction blocks. This corresponds to Cycle 3, where we compute $a_0b_3$ with Mult 3 and $q_0m_3$ with Red 3. Third, we note that we wait 2 cycles before pushing the next $a_i$ and $q_i$ values. During this gap, we schedule a second multiplier if needed to achieve the dual multiplier architecture.

Unlike [26], the accumulation block is split from the multiplication block with each element of the multiplication and reduction blocks mapped to 1 DSP. Thus, the number of DSPs is reduced from $3s - s_A - 1$ to $2s - s_A$. Table III shows the number of DSPs reduced between our design and the design in [26] for $w = 17$. In Xilinx FPGAs, the DSP48E can perform up to a $17 \times 17$ bit unsigned multiplication. In addition, the adders not belonging to the accumulation block in Figure 4 are mapped into the adder part of the DSP. Furthermore, the critical path is reduced to one DSP which improves the frequency from 294 MHz to 401 MHz for the Xilinx Virtex-7 device. The accumulation block is mapped into the fabric of the FPGA. To support all security levels, we set $s = 45$ and $s_A = 12$ with $w = 17$. Therefore, the architecture occupies a total of 78 DSPs.

This multiplier is also a dual multiplier in that it can handle two simultaneous multiplications. This is an odd-even multiplier as on a given cycle, the architecture is only operating over an "odd" or "even" multiplication. Thus, by carefully scheduling operations and storing intermediate partial products, we can support two multiplications in parallel. As is shown in Figure 5, we include additional latching and shift mechanisms to swap between the two sets of operands.

## IV. COMBINING RISC-V WITH A COPROCESSOR

Here, we present our software hardware co-design methodology to accelerate SIKE. Our high-level architecture is shown in Figure 7. Our hardware ALU was described in the previous section and now we present our RISC-V primary controller to facilitate the SIKE cryptosystem by interacting with the hardware coprocessor. In particular, the software coprocessor initializes the hardware's ALU functionality and the software's RAM with the SIKE scheme's parameters. The hardware processor is then used within the software's flow for the intense operations. For the following subsections, we elaborate on the critical components and flow from the left-most component (RISC-V CPU) to the right-most component (inner ALU). Finally, we summarize our instruction controller operation.

### A. RISC-V Processor

Our software-hardware co-design architecture utilizes a RISC-V software processor as the main brains of the device. Our RISC-V core is based on Murax, which is a publicly available system-on-chip for VexRiscv CPU. This core uses the basic rv32i instruction set to support simple basic integer operations excluding multiplication and division.

Inside our RISC-V chip, a configurable RAM block is connected that acts as the program memory. Here, we compile and assemble a machine code based on Microsoft's SIKE library [43]. However, we swap out all $\mathbb{F}_p$ addition, subtraction, and multiplication operations for a call to our hardware co-processor's instruction. Since our addition and dual-multiplier run separately, we applied a scheduler based on [25] to optimally schedule $\mathbb{F}_p$ operations. The other major operation, SHAKE256, was implemented in C, to require around 46,000 cycles per permutation.

Our software was compiled using the RISC-V GNU compiler toolchain with rv32i architecture and optimization level 1. We found that increasing optimization level further had minimal impact on performance but resulted in larger memory. Each SIKE parameter set was compiled separately. To run a SIKE operation, the parameter set machine code was loaded into the RISC-V RAM unit and then executed. SIKEp434 required 28 KB, SIKEp503 required 30 KB, SIKEp610 required 32 KB, and SIKEp751 required 32 KB. Thus, the RISC-V RAM block was a 32 KB RAM, which fits into 8 BRAMs on Xilinx FPGAs.

### B. APB Bridge

In order to share data between the RISC-V chip and the hardware coprocessor, we chose to use the Advanced Periph-

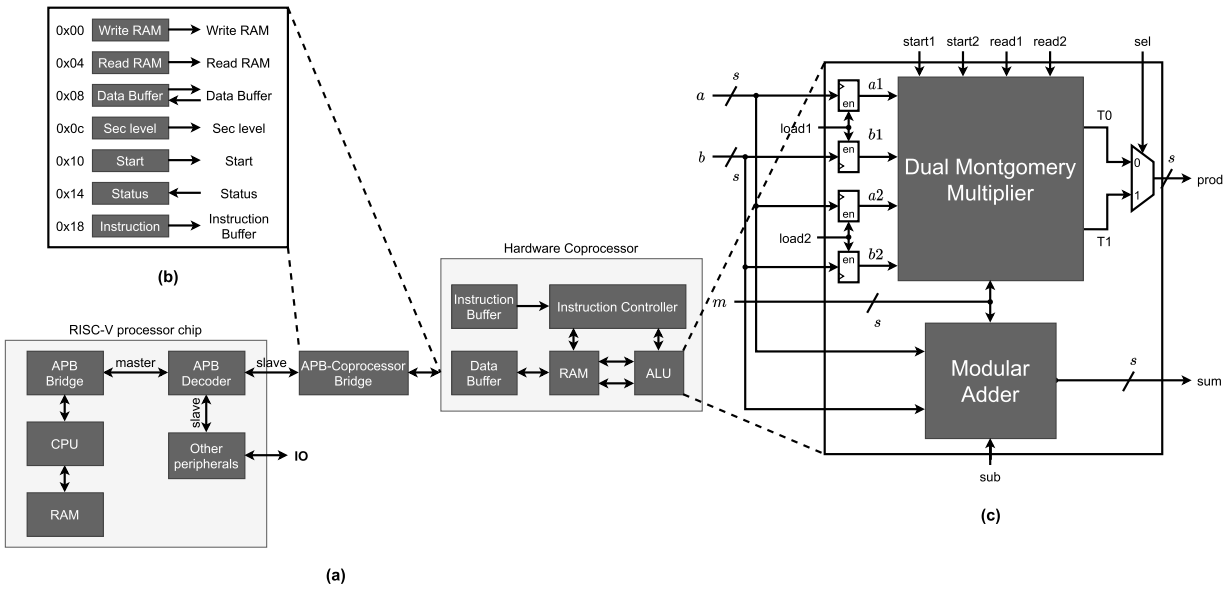


Fig. 7. (a) RISC-V accelerator architecture. (b) Coprocessor-APB bridge. (c) ALU architecture.

eral Bus (APB) bridge. The APB protocol is a royalty-free protocol to connect low-bandwidth peripherals by ARM. This was chosen because it has a very low area cost. In general, our multiplication operations take far longer than the number of cycles for a RISC-V processor to send an instruction. We found that the CPU is on average sending an instruction in less than 10 cycles, where as the smallest interleave delay is 52 cycles. Other protocols such as the Advanced eXtensible Interface (AXI) could have sent instructions much faster, but as we show in our coprocessor architecture, this is not a problem as we keep a buffer of instructions to execute.

As is shown in Figure 7, the RISC-V processor chip features a APB decoder to send the data (our coprocessor instructions) to the coprocessor. In this bus scenario, the RISC-V processor is the master and the coprocessor is the slave. In addition to the coprocessor, the APB bus is also used to connect to the GPIO and UART peripherals for IO operations.

*1) APB Operations:* The APB interface utilizes 7 addresses reserved in the CPU to interact with the coprocessor. All communication between the CPU and coprocessor are performed through the APB interface. The special addresses are:

1) Write RAM: Writes the 752-bit data from data buffer to coprocessor RAM at the specified address.
2) Read RAM: Reads the 752-bit data from the coprocessor RAM at the specified address to the data buffer.
3) Data Buffer: Writes or reads between data buffer and CPU (32-bit data at a time).
4) Sec Level: Loads the security level. This will initialize the $2p$ and $p+1$ in the ALU for the adder and multiplier, respectively, with the SIKE parameter set values. This will also initialize the cycle counts for the instruction controller.
5) Start: Allows the instruction controller to start processing instructions in the instruction buffer. This must be issued before loading any instructions to the instruction buffer.
6) Status: Reports whether the instruction controller is active or not.
7) Instruction: Loads an instruction into the instruction buffer. The four opcodes are $\mathbb{F}_p$ addition, $\mathbb{F}_p$ subtraction, $\mathbb{F}_p$ multiplication, and end.

### C. Coprocessor Architecture

The coprocessor is the primary working unit to accomplish SIKE by efficiently scheduling $\mathbb{F}_p$ addition/subtraction and $\mathbb{F}_p$ multiplication operations. This is facilitated by an instruction controller that reads instructions received from the APB bus, an ALU that performs modular addition and multiplication according to these instructions, and a RAM unit where input and output data is stored. It is important to note that the coprocessor RAM unit is separate from the RISC-V RAM unit.

*1) Buffers:* The APB bridge can be used to send data to the data buffer and instructions to the instruction buffer inside the coprocessor. The data buffer is a 768-bit register that shift 32 bits at a time in and out from the CPU and can write and read 752 bits from the RAM. The instruction buffer is a $32 \times 26$ simple dual port RAM containing 26-bit instructions in a circular buffer. Of these 26 bits, three 8-bit chunks are used for operand A's address, operand B's address, and the destination's address. The final 2 bits indicate an $\mathbb{F}_p$ addition, $\mathbb{F}_p$ subtraction, $\mathbb{F}_p$ multiplication, or end opcode. Since the instruction buffer is a FIFO circular buffer, we have an empty and full signal. The empty signal is connected to the instruction controller and the full signal is connected to the APB interface. We give more details about the instruction controller in the next subsection.

*2) RAM:* The RAM unit is a $256 \times 752$ (depth $\times$ width) true dual-port RAM used to store all isogeny constants and

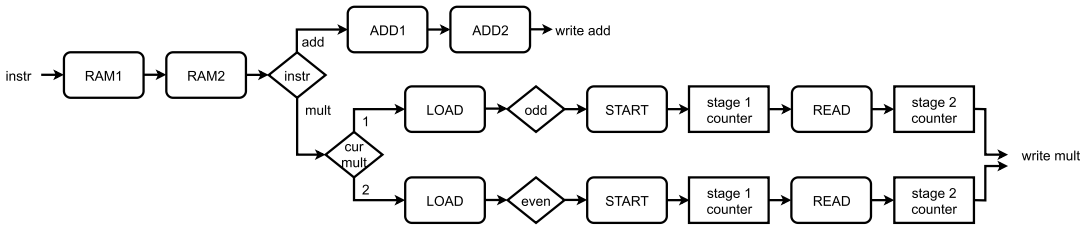


Fig. 8. Instruction controller state machine.

computations. In our selected Virtex-7 FPGA device this requires 21 BRAM units. The RAM unit uses 9 bits to control each port with 8 bits to select the address and 1 bit to enable writing. To minimize the critical path, the data is available to be read from this RAM unit 2 cycles after the address is set.

*3) ALU:* The coprocessor's ALU is composed of a single modular adder and a dual modular multiplier. The inputs to these modules is at most $s$ bits, where $s$ is the largest supported SIKE prime. Here, $s = 752$. The modulus $m$ inside the ALU is used for modular addition and modular multiplication. This can be updated by writing to the correct memory-mapped registers with the APB bridge. Since the dual Montgomery multiplier can perform two simultaneous multiplications, two signals "load1" and "load2" are used to initialize the inputs for each multiplier, respectively. Because there are two outputs of the multiplier, a 2:1 multiplexer is used to select the correct result to store to memory. With the true dual-port RAM, we assign one write port to the multiplication circuit "prod" and one write port to the addition circuit "sum."

### D. Instruction Controller

The basic flow of the instruction controller is shown in Figure 8. Since our RAM has a two cycle delay for reads, we begin with "RAM1" and "RAM2" cycles to decode the instruction. From there, we have three pipelines: addition, odd multiplication, and even multiplication. Our adder requires 2 cycles, so its result is simply written after 2 cycles.

*1) Multiplier Pipelines:* The multiplier is a dual multiplier, so it supports two separate multiplication operations, one on an odd cycle and the other on an even cycle. A single bit is used to keep track of starting the odd or even multiplier next and another single bit is used to keep track of which multiplier result is to be stored. If the current multiplication is an odd multiplication, then we wait for an odd cycle and start the odd multiplication pipeline. Likewise, if the current multiplication is an even multiplication, then we start the even multiplication pipeline. Within these multiplication pipelines, the stage 1 counter is the same as the interleave delay, and we wait this many cycles before we start reading the multiplier results. The stage 2 counter indicates when the multiplier is completely done with its operation, which is multiplication latency - interleave latency cycles. After the stage 2 counter is done, the multiplication result is written back to the coprocessor RAM.

*2) FSM Locks:* In this finite state machine, instructions will continue to process until a lock happens. When a lock happens, everything before RAM1 halts. The three possible locks are:

TABLE IV
AREA RESULTS OF RISC-V ACCELERATOR CHIP IN XILINX FPGAS

| Device | Freq [MHz] | # FFs | # LUTs | # Slices | # DSPs | # BRAMs |
|---|---|---|---|---|---|---|
| Virtex-7 | 303.0 | 14,534 | 9,562 | 3,942 | 78 | 29 |
| Artix-7 | 217.4 | 14,666 | 7,604 | 3,608 | 78 | 30 |

TABLE V
DETAILED AREA BREAKDOWN OF RISC-V ACCELERATOR CHIP IN VIRTEX-7 FPGA

| Component | # FFs | # LUTs | # DSPs | # BRAMs |
|---|---|---|---|---|
| RISC-V processor chip | 1,269 | 1,340 | 0 | 8 |
| Hardware Coprocessor | 13,228 | 8,211 | 78 | 21 |
| - Field Adder | 4,594 | 3,875 | 0 | 0 |
| - Field Multiplier | 6,991 | 1,650 | 78 | 0 |
| Total | **14,534** | **9,562** | **78** | **29** |

TABLE VI
TIMING RESULTS OF RISC-V ACCELERATOR CHIP IN XILINX FPGAS. V7 IS VIRTEX-7 AND A7 IS ARTIX-7

| Security Level | #CC ($\times 10^6$) | | | | Total time [ms] | |
|---|---|---|---|---|---|---|
| | K | E | D | E+D | V7 @ 303.0 MHz | A7 @ 217.4 MHz |
| 1 | 1.19 | 2.12 | 2.27 | 4.39 | 14.5 | 20.2 |
| 2 | 1.61 | 2.80 | 3.01 | 5.81 | 19.2 | 26.7 |
| 3 | 2.36 | 4.47 | 4.57 | 9.04 | 29.8 | 41.6 |
| 5 | 3.68 | 6.23 | 6.71 | 12.94 | 42.7 | 59.5 |

1) Multiplier Lock: This happens when there are 2 multiplication instructions between RAM1 and the stage 1 counter. This means that the dual multiplier is fully occupied and we must wait until one of the multiplications passes the stage 1 counter state.
2) Memory Lock: This happens when the input of an instruction at the start of RAM1 is an output of any previous instruction being processed and not written yet. This previous instruction could be in an arithmetic pipeline or still being decoded in RAM2.
3) Write Lock: This happens when either the multiplier or adder are storing their result. The true dual port RAM unit cannot perform a read and write in the same cycle, so all fetch and decode operations are delayed by one cycle.

## V. FPGA IMPLEMENTATION

The SIKE software-hardware co-design architecture was implemented, tested, and synthesized on Xilinx FPGAs. The

TABLE VIII
IMPROVEMENTS OVER STATE-OF-THE-ART SOFTWARE-HARDWARE CO-DESIGN IMPLEMENTATIONS ON VIRTEX-7 FPGA. NOTE THAT AREA IMPROVEMENTS ARE A DIFFERENCE AND LATENCY IMPROVEMENTS ARE A RATIO

| Reference | Area | | | | | SIKEpXXX E+D Latency [ms] | | | |
|---|---|---|---|---|---|---|---|---|---|
| | FFs | LUTs | Slices | DSPs | BRAMs | p434 | p503 | p610 | p751 |
| Massolino *et al.* [6] (Small) | 7,132 | 10,937 | 3,415 | 57 | 21.0 | 50.4 | 59.5 | 107.2 | 179.6 |
| This work | 14,534 | 9,562 | 3,942 | 78 | 29.0 | 14.5 | 19.2 | 29.8 | 42.7 |
| **Improvement** | -7,402 | 1,375 | -527 | -21 | -8.0 | 3.48 | 3.10 | 3.60 | 4.21 |
| Massolino *et al.* [6] (Full) | 13,657 | 21,210 | 7,408 | 162 | 38.0 | 24.3 | 28.7 | 51.8 | 60.8 |
| This work | 14,534 | 9,562 | 3,942 | 78 | 29.0 | 14.5 | 19.2 | 29.8 | 42.7 |
| **Improvement** | -877 | 11,648 | 3,466 | 84 | 9.0 | 1.68 | 1.49 | 1.74 | 1.42 |
| Elkhatib *et al.* [10] | 14,807 | 9,524 | 4,611 | 78 | 34.5 | 19.2 | 25.1 | 38.7 | 55.0 |
| This work | 14,534 | 9,562 | 3,942 | 78 | 29.0 | 14.5 | 19.2 | 29.8 | 42.7 |
| **Improvement** | 273 | -38 | 669 | 0 | 5.5 | 1.32 | 1.31 | 1.30 | 1.29 |

architecture is written in Verilog and SystemVerilog with the RISC-V processor chip generated in Verilog using SpinalHDL. The architecture is implemented on Xilinx Virtex-7 xc7vx690tffg1157-3 to be similar to the rest of the literature and the Xilinx Artix-7 xc7a200tffg 1156-3 to be consistent with the NIST PQC benchmarking platform. All results obtained are post-place and route. The goal of this design is to achieve a fast and area-efficient implementation of SIKE using our proposed software-hardware co-design architecture.

### *A. Area Results*

Table IV shows the area results of the design on our target Virtex-7 and Artix-7 FPGAs. Interestingly the Artix-7 synthesis exchanged about 2,000 LUTs (300 slices) for a BRAM when compared to the Virtex-7 synthesis. The maximum frequency of the Virtex-7 and Artix-7 implementations were 303 and 217 MHz, respectively. Thus, the Virtex-7 results are approximately 40% faster than the Artix-7 results.

### *B. Timing Results*

We summarize the SIKE timing results in Table VI. The three main SIKE operations are key generation (K), key encapsulation (E), and key decapsulation (D). Since a party only needs to generate a single private-public keypair with key generation, the encapsulation and decapsulation combined latency (E+D) is a good summary of the implementation's performance. As these results show, the E+D latency ranges from 4.39 million cycles to 12.94 million cycles for the NIST level 1 and level 5 parameter sets, respectively. On the Virtex-7 FPGA, this corresponds to E+D timings of 14.5 ms to 42.7 ms. The Artix-7 FPGA results are about 40% slower, ranging from 20.2 ms to 59.5 ms. In normal use cases, it is expected that a party's latency will be about half of that as they will be performing one of key encapsulation or decapsulation.

### *C. Comparisons*

It is difficult to make a fair comparison to other SIKE implementations as there are a variety of optimization targets. Nevertheless, many implementations use the Virtex-7 FPGA, so we can roughly compare the area, performance, and area-time product of these implementations, which are summarized in Table VII. In Table VII, the "Time" column indicates the combined latency of key encapsulation and key decapsulation (E+D) for a given SIKE parameter set for the implementation's max frequency. The area-time (AT) product is used as a further metric to indicate the relative efficiency of area to perform cryptographic operations. As an attempt to equalize the cost of FPGA resources between slices, DSPs, and BRAMs on a Virtex-7 FPGA, we use the equivalence 1 DSP=100 Slices and 1 BRAM=200 Slices for area-time calculations. With this metric, our area-time product results outperform all existing hardware implementations in the literature. Our implementation utilizes a single dual multiplier, whereas other designs achieve a higher parallelization with a plurality of multipliers and complex scheduling. Notably, the high-speed implementations from [25], [26] are about 60% faster than ours, but require 3 to 4 times as many simultaneous multiplication units. For SIKEp751, this requires 2.8-3.9 times as many slices and 5.8-6.6 times as many DSPs. This shows the diminishing returns from adding additional multiplication units and elevates the impact of our highly optimized adder, multiplier, and controller units. The high-speed implementation from [45] shows the top-end performance of SIKE, but at the cost of significantly more slices, DSPs, and BRAMs, resulting in a higher area-time product.

We summarize our comparison to other software-hardware co-design implementations in Table VIII. We note that we did not include the RISC-V implementation from [9] as it used a Xilinx Zedboard platform, making a fair comparison even more difficult. On the smaller end, the software-hardware co-design in [6] uses another software-hardware architecture, but targets a compact use case. Their small implementation is about 3-4 times slower than our implementation across the parameter sets, but saves 527 slices, 21 DSPs, and 8 BRAMs. Their fast implementation uses almost double the resources of our implementation and is about 1.5-1.7 times slower than our implementation. Interestingly, the results from [6] show a much different ratio of FFs to LUTs, preferring less sequential FFs for more combinational LUTs. Lastly, we used a very similar approach to [10], but our optimizations to the addition, multiplication, and controller units edge out 669 slices and 5.5 fewer BRAMs even with 1.3 times performance improvement.

TABLE VII

COMPARISON OF AREA AND TIMING RESULTS IN VIRTEX-7 FPGA. NOTE: AT = (SLICES + 100 × DSPS + 200 × BRAMS) × TIME

| Reference | Slices | DSPs | BRAMs | Time | AT ($\times 10^{-3}$) |
|---|---|---|---|---|---|
| **SIKEp434** | | | | | |
| Koziel *et al.* [21] | 8,121 | 240 | 26.5 | 11.3 | 423 |
| Elkhatib *et al.* [26] | 5,527 | 195 | 32.0 | 8.8 | 277 |
| Elkhatib *et al.* [44] | 5,527 | 195 | 32.0 | 8.0 | 251 |
| Massolino *et al.* [6] (S) | 3,415 | 57 | 21.0 | 50.4 | 822 |
| Massolino *et al.* [6] (F) | 7,408 | 162 | 38.0 | 24.3 | 446 |
| Prior Work [10] | 4,611 | 78 | 34.5 | 19.2 | 371 |
| **This Work** | **3,942** | **78** | **29** | **14.5** | **254** |
| **SIKEp503** | | | | | |
| Koziel *et al.* [23]* | 10,298 | 192 | 27.0 | 33.7 | 1176 |
| Koziel *et al.* [22]* | 8,918 | 192 | 40.0 | 20.9 | 755 |
| Koziel *et al.* [24]* | 7,491 | 192 | 43.5 | 16.5 | 584 |
| Koziel *et al.* [21] | 8,907 | 264 | 33.5 | 14.1 | 592 |
| Elkhatib *et al.* [26] | 6,163 | 225 | 34.0 | 11.8 | 418 |
| Elkhatib *et al.* [44] | 6,163 | 225 | 34.0 | 10.9 | 387 |
| Massolino *et al.* [6] (S) | 3,415 | 57 | 21.0 | 59.5 | 792 |
| Massolino *et al.* [6] (F) | 7,408 | 162 | 38.0 | 28.7 | 896 |
| Prior Work [10] | 4,611 | 78 | 34.5 | 25.1 | 485 |
| **This Work** | **3,942** | **78** | **29** | **19.2** | **337** |
| **SIKEp610** | | | | | |
| Koziel *et al.* [21] | 10,675 | 312 | 39.5 | 21.6 | 1075 |
| Elkhatib *et al.* [26] | 7,461 | 270 | 38.5 | 19.1 | 805 |
| Elkhatib *et al.* [44] | 7,461 | 270 | 38.5 | 17.8 | 750 |
| Massolino *et al.* [6] (S) | 3,415 | 57 | 21.0 | 107.2 | 1427 |
| Massolino *et al.* [6] (F) | 7,408 | 162 | 38.0 | 51.8 | 1617 |
| Prior Work [10] | 4,611 | 78 | 34.5 | 38.7 | 747 |
| **This Work** | **3,942** | **78** | **29** | **29.8** | **523** |
| **SIKEp751** | | | | | |
| SIKE Team [11] | 16,756 | 376 | 56.5 | 33.4 | 2193 |
| Koziel *et al.* [21] | 15,834 | 512 | 43.5 | 27.8 | 2105 |
| Farzam *et al.* [25]** | 15,336 | 512 | 45.0 | 24.1 | 1820 |
| Elkhatib *et al.* [26] | 11,136 | 452 | 41.5 | 25.5 | 1648 |
| Elkhatib *et al.* [44] | 11,136 | 452 | 41.5 | 22.5 | 1454 |
| Massolino *et al.* [6] (S) | 3,415 | 57 | 21.0 | 179.6 | 2391 |
| Massolino *et al.* [6] (F) | 7,408 | 162 | 38.0 | 60.8 | 1897 |
| Tian *et al.* [45] | 27,286 | 834 | 73.5 | 9.3 | 1166 |
| Prior Work [10] | 4,611 | 78 | 34.5 | 55.0 | 1062 |
| **This Work** | **3,942** | **78** | **29** | **42.7** | **749** |

∗ SIDH

∗∗ SIKE Round 1 Parameters

## VI. CONCLUSION

In this paper, we implemented a fast an efficient software-hardware co-design for SIKE targeting all security levels in one design. We proposed and implemented optimized units for addition, multiplication, and control. Our presented Xilinx FPGA results prominently feature the best area-time product of any hardware implementation of SIKE in the literature. This work shows that careful optimization and isolation of low-level arithmetic can be made to greatly improve the efficiency of cryptosystems.

## ACKNOWLEDGMENT

The authors would like to thank the reviewers for their comments.

## REFERENCES

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

[2] L. K. Grover, "A fast quantum mechanical algorithm for database search," in *Proc. 28th Annu. ACM Symp. Theory Comput. (STOC)*. New York, NY, USA: Association for Computing Machinery, 1996, pp. 212–219.

[3] The National Institute of Standards and Technology. (2018). *Post-Quantum Cryptography Standardization*. [Online]. Available: https://csrc.nist.gov/projects/post-quantum-cryptography/post-quantum-cryptography-standardization

[4] M. Anastasova, R. Azarderakhsh, and M. M. Kermani, "Fast strategies for the implementation of SIKE round 3 on ARM Cortex-M4," *IEEE Trans. Circuits Syst. I, Reg. Papers*, vol. 68, no. 10, pp. 4129–4141, Oct. 2021.

[5] R. Azarderakhsh, R. E. Khatib, B. Koziel, and B. Langenberg, "Hardware deployment of hybrid PQC," Cryptol. ePrint Arch., New York, NY, USA, Tech. Rep. 2021/541, 2021. [Online]. Available: https://ia.cr/2021/541

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

[7] U. Banerjee, S. Das, and A. P. Chandrakasan, "Accelerating post-quantum cryptography using an energy-efficient TLS crypto-processor," in *Proc. IEEE Int. Symp. Circuits Syst. (ISCAS)*, Oct. 2020, pp. 1–5.

[8] U. Banerjee, T. S. Ukyab, and A. P. Chandrakasan, "Sapphire: A configurable crypto-processor for post-quantum lattice-based protocols," *IACR Trans. Cryptograph. Hardw. Embedded Syst.*, vol. 2019, pp. 17–61, Aug. 2019.

[9] D. B. Roy, T. Fritzmann, and G. Sigl, "Efficient hardware/software co-design for post-quantum crypto algorithm SIKE on ARM and RISC-V based microcontrollers," in *Proc. 39th Int. Conf. Comput.-Aided Design (ICCAD)*, Nov. 2020, pp. 1–9.

[10] R. Elkhatib, R. Azarderakhsh, and M. Mozaffari-Kermani, "Accelerated RISC-V for SIKE," in *Proc. IEEE 28th Symp. Comput. Arithmetic (ARITH)*, Lyngby, Denmark, Jun. 2021, pp. 131–138.

[11] R. Azarderakhsh *et al.*, "Supersingular isogeny key encapsulation," NIST Post-Quantum Standardization Project, New York, NY, USA, Tech. Rep., 2020.

[12] J. H. Silverman, *The Arithmetic of Elliptic Curves*, vol. 106. New York, NY, USA: Springer, 1992.

[13] L. D. Feo, "Mathematics of isogeny based cryptography," 2017, *arXiv:1711.04062*.

[14] J. Vélu, "Isogénies entre courbes elliptiques," *Comptes Rendus de l'Académie des Sciences Paris A-B*, vol. 273, pp. A238–A241, Jun. 1971.

[15] J.-M. Couveignes, "Hard homogeneous spaces," Cryptol. ePrint Arch., New York, NY, USA, Tech. Rep. 2006/291, 2006.

[16] A. Rostovtsev and A. Stolbunov, "Public-key cryptosystem based on isogenies," Cryptol. ePrint Arch., New York, NY, USA, Tech. Rep. 2006/145, 2006.

[17] D. X. Charles, K. E. Lauter, and E. Z. Goren, "Cryptographic hash functions from expander graphs," *J. Cryptol.*, vol. 22, no. 1, pp. 93–113, Jan. 2009.

[18] A. Childs, D. Jao, and V. Soukharev, "Constructing elliptic curve isogenies in quantum subexponential time," *J. Math. Cryptol.*, vol. 8, no. 1, pp. 1–29, 2014.

[19] D. Jao and L. De Feo, "Towards quantum-resistant cryptosystems from supersingular elliptic curve isogenies," in *Proc. 4th Int. Workshop Post-Quantum Cryptogr. (PQCrypto)* 2011, pp. 19–34.

[20] D. Hofheinz, K. Hövelmanns, and E. Kiltz, "A modular analysis of the Fujisaki-Okamoto transformation," in *Theory of Cryptography* (Lecture Notes in Computer Science). Heidelberg, Germany, 2017, pp. 341–371.

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

[22] B. Koziel, R. Azarderakhsh, and M. Mozaffari-Kermani, "Fast hardware architectures for supersingular isogeny Diffie–Hellman key exchange on FPGA," in *Prog. 17th Int. Conf. Cryptol. India*, 2016, pp. 191–206.

[23] B. Koziel, R. Azarderakhsh, M. M. Kermani, and D. Jao, "Post-quantum cryptography on FPGA based on isogenies on elliptic curves," *IEEE Trans. Circuits Syst.*, vol. 64, no. 1, pp. 86–99, Jan. 2017.

[24] B. Koziel, R. Azarderakhsh, and M. M. Kermani, "A high-performance and scalable hardware architecture for isogeny-based cryptography," *IEEE Trans. Comput.*, vol. 67, no. 11, pp. 1594–1609, Nov. 2018.

[25] M.-H. Farzam, S. Bayat-Sarmadi, and H. Mosanaei-Boorani, "Implementation of supersingular isogeny-based Diffie–Hellman and key encapsulation using an efficient scheduling," *IEEE Trans. Circuits Syst. I, Reg. Papers*, vol. 67, no. 12, pp. 4895–4903, Dec. 2020.
[26] R. Elkhatib, R. Azarderakhsh, and M. Mozaffari-Kermani, "Highly optimized Montgomery multiplier for SIKE primes on FPGA," in *Proc. IEEE 27th Symp. Comput. Arithmetic (ARITH)*, Jun. 2020, pp. 64–71.
[27] H. Seo, M. Anastasova, A. Jalali, and R. Azarderakhsh, "Supersingular isogeny key encapsulation (SIKE) round 2 on ARM cortex-M4," *IEEE Trans. Comput.*, vol. 70, no. 10, pp. 1705–1718, Oct. 2021.
[28] M. Anastasova, R. Azarderakhsh, and M. M. Kermani, "Fast strategies for the implementation of SIKE round 3 on ARM Cortex-M4," Cryptol. ePrint Arch., New York, NY, USA, Tech. Rep. 2021/115, 2021. [Online]. Available: https://eprint.iacr.org/2021/115
[29] B. Koziel, A. Jalali, R. Azarderakhsh, D. Jao, and M. Mozaffari-Kermani, "NEON-SIDH: Efficient implementation of supersingular isogeny Diffie–Hellman key exchange protocol on ARM," in *Proc. 15th Int. Conf. Cryptol. Netw. Secur. (CANS)*, 2016, pp. 88–103.
[30] B. Koziel, R. Azarderakhsh, D. Jao, and M. M. Kermani, "On fast calculation of addition chains for isogeny-based cryptography," in *Proc. 12th Int. Conf. Inf. Secur. Cryptol. (Inscrypt)*, Beijing, China, Nov. 2016, pp. 323–342.
[31] A. Faz-Hernández, J. López, E. Ochoa-Jiménez, and F. Rodríguez-Henríquez, "A faster software implementation of the supersingular isogeny Diffie–Hellman key exchange protocol," *IEEE Trans. Comput.*, vol. 67, no. 11, pp. 1622–1636, Nov. 2018.
[32] C. Costello and H. Hisil, "A simple and compact algorithm for SIDH with arbitrary degree isogenies," in *Proc. 23rd Int. Conf. Theory Appl. Cryptol. Inf. Secur.*, 2017, pp. 303–329.
[33] S. D. Galbraith, C. Petit, B. Shani, and Y. B. Ti, "On the security of supersingular isogeny cryptosystems," in *Advances in Cryptology—ASIACRYPT* (Lecture Notes in Computer Science). Heidelberg, Germany, 2016, pp. 63–91.
[34] C. Costello, P. Longa, M. Naehrig, J. Renes, and F. Virdia, "Improved classical cryptanalysis of the computational supersingular isogeny problem," Cryptol. ePrint Arch., New York, NY, USA, Tech. Rep. 2019/298, 2019. [Online]. Available: https://eprint.iacr.org/2019/298
[35] B. Koziel, R. Azarderakhsh, and D. Jao, "An exposure model for supersingular isogeny Diffie–Hellman key exchange," in *Proc. Cryptographers Track RSA Conf. (CT-RSA)*, 2018, pp. 452–469.
[36] B. Koziel, R. Azarderakhsh, and D. Jao, "Side-channel attacks on quantum-resistant supersingular isogeny Diffie–Hellman," in *Proc. 24th Int. Conf. Sel. Areas Cryptogr. (SAC)*, 2018, pp. 64–81.
[37] A. Gélin and B. Wesolowski, "Loop-abort faults on supersingular isogeny cryptosystems," in *Proc. 8th Int. Workshop Post-Quantum Cryptogr. (PQCrypto)*, 2017, pp. 93–106.
[38] Y. B. Ti, "Fault attack on supersingular isogeny cryptosystems," in *Proc. 8th Int. Workshop Post-Quantum Cryptography (PQCrypto)*, Utrecht, The Netherlands. Cham, Switzerland: Springer, Jun. 2017, pp. 107–122.
[39] S. Jaques and J. M. Schanck, "Quantum cryptanalysis in the RAM model: Claw-finding attacks on SIKE," Cryptol. ePrint Arch., New York, NY, USA, Tech. Rep. 2019/103, 2019. [Online]. Available: https://eprint.iacr.org/2019/103
[40] G. Adj, D. Cervantes-Vázquez, J. Chi-Domínguez, A. Menezes, and F. Rodríguez-Henríquez, "On the cost of computing isogenies between supersingular elliptic curves," Cryptol. ePrint Arch., New York, NY, USA, Tech. Rep. 2018/313, 2018.
[41] M. Rogawski, E. Homsirikamol, and K. Gaj, "A novel modular adder for one thousand bits and more using fast carry chains of modern FPGAs," in *Proc. 24th Int. Conf. Field Program. Log. Appl. (FPL)*, Sep. 2014, pp. 1–8.
[42] P. L. Montgomery, "Modular multiplication without trial division," *Math. Comput.*, vol. 44, no. 170, pp. 519–521, Apr. 1985.
[43] C. Costello, P. Longa, and M. Naehrig, "Efficient algorithms for supersingular isogeny Diffie–Hellman," in *Proc. 36th Annu. Int. Cryptol. Conf. Adv. Cryptol. (CRYPTO)*, 2016, pp. 572–601.
[44] R. Elkhatib, R. Azarderakhsh, and M. Mozaffari-Kermani, "High-performance FPGA accelerator for SIKE," *IEEE Trans. Comput.*, early access, May 10, 2021, doi: 10.1109/TC.2021.3078691.
[45] J. Tian, B. Wu, and Z. Wang, "High-speed FPGA implementation of SIKE based on an ultra-low-latency modular multiplier," *IEEE Trans. Circuits Syst. I, Reg. Papers*, vol. 68, no. 9, pp. 3719–3731, Sep. 2021.

**Rami Elkhatib** received the bachelor's degree from the American University of Beirut in 2013. He is currently pursuing the Ph.D. degree in computer engineering with Florida Atlantic University. His research areas include finite field arithmetic, cryptographic engineering, post-quantum cryptography, quantum cryptanalysis, FPGA implementations, and optimization.

**Brian Koziel** received the dual B.Sc. and M.Sc. degree in computer engineering from the Rochester Institute of Technology in 2016. He is currently pursuing the Ph.D. degree with Florida Atlantic University. His current research interests include constructions, implementations, and deployment of post-quantum cryptography. He has been awarded an NSF Graduate Research Fellowship. At RIT, he was a recipient of the prestigious Outstanding Undergraduate Scholar Award.

**Reza Azarderakhsh** (Member, IEEE) received the Ph.D. degree in electrical and computer engineering from Western University in 2011. He was a recipient of the NSERC Post-Doctoral Research Fellowship working at the Center for Applied Cryptographic Research and the Department of Combinatorics and Optimization, University of Waterloo. He is currently an Associate Professor at the Department of Electrical and Computer Engineering, Florida Atlantic University. His current research interests include finite field and its application, elliptic curve cryptography, pairing-based cryptography, and post-quantum cryptography. He is serving as an Associate Editor for IEEE TRANSACTIONS ON CIRCUITS AND SYSTEMS.

**Mehran Mozaffari Kermani** (Senior Member, IEEE) received the B.Sc. degree from the University of Tehran, Iran, and the M.E.Sc. and Ph.D. degrees from the University of Western Ontario, London, Canada, in 2007 and 2011, respectively. In 2012, he joined the Electrical Engineering Department, Princeton University, NJ, USA, as an NSERC Post-Doctoral Research Fellow. From 2013 to 2017, he was an Assistant Professor with the Rochester Institute of Technology and starting 2017, he has joined the Computer Science and Engineering Department, University of South Florida, where he is currently an Associate Professor. He is serving as an Associate Editor for the IEEE TRANSACTIONS ON VERY LARGE SCALE INTEGRATION SYSTEMS, the *ACM Transactions on Embedded Computing Systems*, and the IEEE TRANSACTIONS ON CIRCUITS AND SYSTEMS—I: REGULAR PAPERS. He has been a Guest Editor of the IEEE TRANSACTIONS ON DEPENDABLE AND SECURE COMPUTING.