Post-Quantum Cryptography on FPGA Based on Isogenies on Elliptic Curves

Brian Koziel, Reza Azarderakhsh, Mehran Mozaffari Kermani, and David Jao

***Abstract*— To the best of our knowledge, we present the first hardware implementation of isogeny-based cryptography available in the literature. Particularly, we present the first implementation of the supersingular isogeny Diffie-Hellman (SIDH) key exchange, which features quantum-resistance. We optimize this design for speed by creating a high throughput multiplier unit, taking advantage of parallelization of arithmetic in $\mathbb{F}_{p^2}$, and minimizing pipeline stalls with optimal scheduling. Consequently, our results are also faster than software libraries running affine SIDH even on Intel Haswell processors. For our implementation at 85-bit quantum security and 128-bit classical security, we generate ephemeral public keys in 1.655 million cycles for Alice and 1.490 million cycles for Bob. We generate the shared secret in an additional 1.510 million cycles for Alice and 1.312 million cycles for Bob. On a Virtex-7, these results are approximately 1.5 times faster than known software implementations running the same 512-bit SIDH. Our results and observations show that the isogeny-based schemes can be implemented with high efficiency on reconfigurable hardware.**

***Index Terms*— Elliptic curve cryptography (ECC), field programmable gate array (FPGA), isogeny-based cryptography, post-quantum cryptography.**

I. Introduction

PUBLIC-KEY cryptography is the foundation of internet security as we know it today, allowing for two parties to communicate securely without the need to exchange confidential key material in advance. All public key cryptosystems in widespread use today are based on either the problem of factoring large integers (e.g., RSA) or the problem of computing discrete logarithms in some group such as elliptic curves. Elliptic curve cryptography (ECC) was invented by Victor Miller [1] and Neal Koblitz [2] in 1985 with the aim of providing an alternative to the popular public-key cryptosystems of the time, such as multiplicative group over finite field - RSA. In recent years, ECC has been the primary cryptographic protocol for secure web pages, online banking, encrypted email, and many other types of data. Breaking these would have significant ramifications for electronic privacy and security. ECC adoption has been accelerated through the recommendation of several standardization bodies including IEEE, NIST, ANSI, IETF, and the like. In comparison to RSA, ECC represents the most efficient public key cryptosystems available today for a desired level of security. In a seminal paper from 1994, Shor showed that both of these problems would be easy to solve on a quantum computer, one which uses quantum mechanics to perform calculations faster than any classical computer can achieve. Since then, much work has been done on the topic of constructing post-quantum public-key cryptosystems which would be secure against quantum computers. Quantum computers are different from digital computers based on transistors. Large-scale quantum computers will be able to solve some currently hard problems much quicker than any classical computer using certain algorithms. Although large-scale quantum computers do not yet exist, the goal is to develop quantum-resistant cryptosystems in anticipation of these quantum threats. Recent announcements by the US government of upcoming plans to require post-quantum cryptosystems for all future US government security applications have provided new impetus to develop and deploy post-quantum cryptosystems [3].

Isogeny-based cryptography is a method of designing cryptosystems based on isogenies on elliptic curves which computationally constructs an algebraic map between elliptic curves. In particular, unlike traditional ECC, isogeny computations over supersingular elliptic curves appear resistant to quantum attacks, and hence such systems are suitable for quantum-resistant cryptography. Originally, Rostovtsev and Stolbunov [4] presented a key exchange based on isogenies of ordinary elliptic curves in 2006. However, Childs *et al.* [5] discovered a quantum algorithm to compute isogenies on ordinary curves in subexponential time in 2010, only assuming the Generalized Riemann Hypothesis. To address these quantum concerns, Jao and De Feo [6] created a key exchange based instead on isogenies of supersingular elliptic curves in 2011. The scheme features quantum resistance and the best known quantum attack has complexity $O(p^{1/6})$. Several other papers on the topic have appeared in the literature such as fast isogeny computations and zero-knowledge identification [7], undeniable signatures [8], key compression [9], and projective isogeny formulas [10]. Isogeny-based cryptography is a strong candidate for standardized PQC applications because it resembles its primitive, ECC, features very small key sizes and signature sizes, and provides forward secrecy.

Manuscript received May 10, 2016; revised July 12, 2016; accepted July 26, 2016. Date of publication October 31, 2016; date of current version January 6, 2017. This paper was recommended by Associate Editor Y. Ha.

B. Koziel is with Texas Instruments, Dallas, TX 75243 USA (e-mail: kozielbrian@gmail.com).

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

M. Mozaffari Kermani is with the Electrical and Microelectronic Engineering Department, Rochester Institute of Tecchnology, Rochester, NY, USA (e-mail: mmkeme@rit.edu).

D. Jao is with the Department of Mathematics and Optimziation, University of Waterloo, ON, CANADA (e-mail: djao@math.uwaterloo.ca).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TCSI.2016.2611561

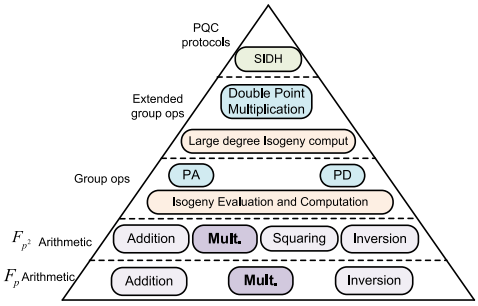

Fig. 1. Breakdown of supersingular isogeny computations.

In this paper, we propose efficient algorithms and architecture for isogeny-based cryptography, particularly the SIDH key exchange. To the best of the authors' knowledge, there is no work available in the literature to consider research in implementations of isogeny-based cryptography on hardware. Thus, this paper provides an initial view at SIDH's impact on reconfigurable hardware.

Our contribution:

- We discuss the design of fast and scalable architectures for isogeny-based cryptography in reconfigurable hardware.
- To the best of our knowledge, we provide the first implementation of the supersingular isogeny Diffie-Hellman key exchange on reconfigurable hardware.
- We heavily parallelize the operations in $\mathbb{F}_{p^2}$ through the use of several Montgomery multipliers and efficient scheduling.
- We provide a high-radix Montgomery multiplier that features interleaved multiplications, suitable for high-throughput arithmetic in $\mathbb{F}_{p^2}$.
- We achieve a performance that is 1.5 times faster than top desktop processors running the protocol in [9].

II. Preliminaries: Supersingular Isogeny Diffie-Hellman

In this section, we provide an overview of the supersingular isogeny Diffie-Hellman (SIDH) key exchange protocol. The breakdown of all computations for the isogeny-based protocols is shown in Fig. 1. As one can see, isogeny-based cryptography is similar to standard elliptic curve cryptography, but also includes the use of isogenies as a way to move from elliptic curve to elliptic curve. We point the reader to [6] and [7] for a full look at the SIDH scheme and [11] for a more complete look at elliptic curve background necessary for isogenies.

A. Isogeny Background

The most popular form of public-key cryptography for today's applications has been transitioning to Elliptic Curve Cryptography (ECC). ECC defines points on an elliptic curve and specific point doubling and point addition formulas to go from point to point. Scalar point multiplication uses a sequence of point doublings and point additions to efficiently evaluate point multiplications $Q = kP = P + P + \cdots + P$. Cryptosystems based on ECC rely on the difficulty of solving the Elliptic Curve Discrete Log (ECDL) problem, such that given Q and P in the previous equation, it is infeasible to determine the scalar multiple k for elliptic curves with points of a large order. However, with the emergence of quantum computers in the near future, such cryptosystems that rely on the ECDL are no longer safe as the scalar multiple can be easily recovered using Shor's algorithm [12] on a quantum computer. Hence, standard ECC is no longer applicable for long-term security and other quantum resilient schemes have been proposed.

Isogeny-based cryptography also utilizes points on an elliptic curve, but is instead based on the difficulty of computing isogenies between elliptic curves. An isogeny can be thought of as a unique algebraic mapping between two elliptic curves that satisfies the group law. An algorithm for computing isogenies on ordinary curves in subexponential time was presented by Childs *et al.* [5], rendering the use of cryptosystems based on isogenies on ordinary curves unsafe in the presence of quantum computers. However, there is no known algorithm for computing isogenies on supersingular curves in subexponential time.

A curve's endomorphism ring is defined as the ring formed by the set of endomorphisms of an elliptic curve together with the null map under point addition and functional composition. Supersingular curves have an endomorphism ring with $\mathbb{Z}$-rank equal to 4. These curves can be defined over $\mathbb{F}_p$ or $\mathbb{F}_{p^2}$. Thus, all supersingular curves can be represented in $\mathbb{F}_{p^2}$. Specifically, supersingular curves have the property that for every prime $\ell \neq p$, there exist $\ell + 1$ isogenies of degree ℓ from a base curve.

We compute an isogeny between curves by utilizing a kernel, k, such that $\phi : E \rightarrow E/\langle\kappa\rangle$. Further we also bring points on the original curve to the isogenous curve by evaluating the isogeny at the points.

The j-invariant is a discriminant based on the elliptic curve coefficients. Two curves are isomorphic iff they have a shared j-invariant.

B. Computing Large Degree Isogenies

The degree of an isogeny is its degree as an algebraic map. As shown in [13], isogeny computations can be done iteratively. Given an elliptic curve E and a point R of order ℓ^e, we compute $\phi\colon E \rightarrow E/\langle R\rangle$ by decomposing ϕ into a chain of degree ℓ isogenies, $\phi = \phi_{e-1} \circ \cdots \circ \phi_0$, as follows. Set $E_0 = E$ and $R_0 = R$, and define

$$E_{i+1} = E_i/\langle \ell^{e-i-1} R_i \rangle \phi_i : E_i \rightarrow E_{i+1} R_{i+1} = \phi_i(R_i).$$

Essentially, point additions are used to compute the kernel at each iteration and Vélu's formulas [14] are used to compute ϕ_i and E_{i+1}. This method applies specifically to isogenies and curve construction. An optimal strategy to compute these isogenies relies on walking a large directed acyclic graph in the shape of a triangle to the leaves, which is shown in Fig. 2. For this graph, performing a multiplication by ℓ results in walking left and evaluating an isogeny relationship results in walking right. Computing an isogeny at each of the leaves is used to compute the full isogenous mapping. Refer to [7]

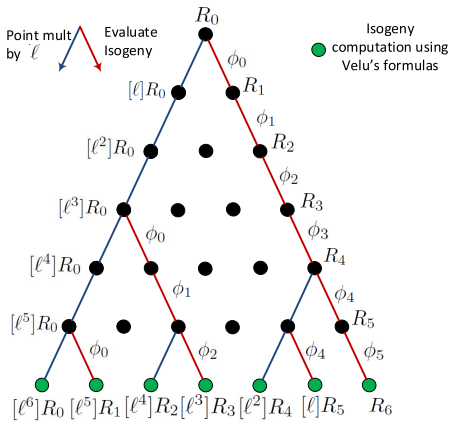

Fig. 2. Large degree isogeny computation structure.

for more information regarding the optimal strategy. The optimal strategy differs based on the relative costs of point multiplications and isogeny evaluations.

C. SIDH Key Exchange Scheme

In [6], Jao and De Feo proposed a key exchange based on isogenies of supersingular elliptic curves. The scheme for SIDH resembles the standard Elliptic Curve Diffie-Hellman (ECDH), but goes a step further by computing isogenies over large degrees. In the scenario, Alice and Bob want to exchange a secret key over an insecure channel. They pick a smooth isogeny prime p of the form $\ell_A^a \ell_B^b \cdot f \pm 1$ where ℓ_A and ℓ_B are small primes, a and b are positive integers, and f is a small cofactor to make the number prime. They define a supersingular elliptic curve, $E_0(\mathbb{F}_q)$ where $q = p^2$. Lastly, they choose four points on the curve that form a bases $\{P_A, Q_A\}$ and $\{P_B, Q_B\}$, which act as generators for $E_0[\ell_A^a]$ and $E_0[\ell_B^b]$, respectively. In a graph of supersingular isogenies where the vertices represent isomorphic curves and the edges represent $\ell-$degree isogenies, the infeasibility to discover a path that connects two particular vertices provides security for this protocol. Essentially, each party takes seemingly random walks in the graphs of isogenies of degree ℓ_A^a and ℓ_B^b to both arrive at a curve with the same j-invariant.

Alice chooses two private keys $m_A, n_A \in \mathbb{Z}/\ell_A^a\mathbb{Z}$ with the stipulation that are not both divisible by ℓ_A^a. On the other side, Bob chooses two private keys $m_B, n_B \in \mathbb{Z}/\ell_B^b\mathbb{Z}$, where both private keys are not divisible by ℓ_B^b. From there, the key exchange protocol can be broken down into two rounds of the following:

1. Compute $R = \langle [m]P + [n]Q \rangle$ for points P, Q.
2. Compute the isogeny $\phi : E \to E/\langle R \rangle$ for a supersingular curve E.
3. Compute the images $\phi(R)$ and $\phi(S)$, where R and S are the basis of the opposite party, only for the first round.

The key exchange protocol is shown in Fig. 3. Here, we will describe the key exchange. Alice performs the double point multiplication with her private keys to obtain a kernel, $R_A = \langle [m_A]P_A + [n_A]Q_A \rangle$ and computes an isogeny $\phi_A : E_0 \to E_A = E_0/\langle [m_A]P_A + [n_A]Q_A \rangle$. As she

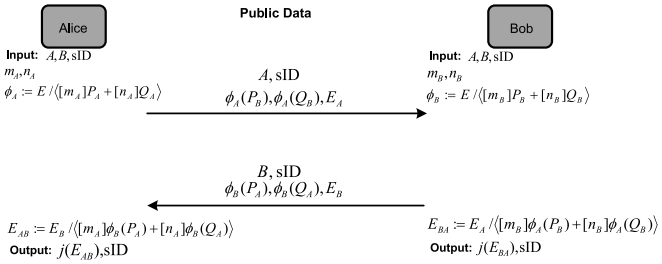

Fig. 3. Supersingular Isogeny Diffie-Hellman Key Exchange [6]. "sID" stands for unique session ID.

computes the isogeny, she also computes the projection $\{\phi_A(P_B), \phi_A(Q_B)\} \subset E_A$ of the basis $\{P_B, Q_B\}$for $E_0[\ell_B^b]$ under her secret isogeny ϕ_A, which can be done efficiently by pushing the points P_B and Q_B through the isogeny at each smaller isogeny. Over a public channel, she sends these points and curve E_A to Bob. Bob performs the same computations and sends the curve E_B and points $\phi_B(P_A)$ and $\phi_B(Q_A)$ to Alice. For the second round, Alice performs the double point multiplication to find a second kernel, $R_{AB} = \langle [m_A]\phi_B(P_A) + [n_A]\phi_B(Q_A) \rangle$, to compute a second isogeny $\phi'_A : E_B \to E_{AB} = E_B/\langle [m_A]\phi_B(P_A) + [n_A]\phi_B(Q_A) \rangle$. Bob also performs a double point multiplication and computes a second isogeny $\phi'_B : E_A \to E_{BA} = E_A/\langle [m_B]\phi_A(P_B) + [n_B]\phi_A(Q_B) \rangle$. Alice and Bob now have isomorphic curves and can use the common j-invariant as a shared secret key.

D. Optimizations to the SIDH Scheme

Here, we discuss several optimizations to the SIDH scheme that increase the performance of the scheme tremendously. First, [7] implements all arithmetic on a Montgomery curve's [15] Kummer line $(x, y) = (X : Z)$, where $x = X/Z$. Particularly, the Kummer arithmetic features extremely fast differential point addition and doubling formulas, which greatly speeds up traversing left on the large degree isogeny graph. This is also utilized for a 3-point differential addition ladder for the double point multiplication, which is shown in Algorithm 1. This operates under the assumption that for the point multiplication $[m]P + [n]Q = R$, that either m or n is 1, or $R = P + [m^{-1}n]Q$. This does not diminish the security of the protocol since m or n will be invertible modulo the order of the group and, thus, $P + [m^{-1}n]Q$ is still a generator. We mention a slight optimization to the above scheme, which involves ensuring that P,Q, and $Q - P$ have a Z-coordinate of 1. This reduces the cost of a differential addition by 1 multiplication, which reduces the cost of a three-point ladder step by 2 multiplications. This requires three additional inversions and three additional multiplication in $\mathbb{F}_{p^2}$ before the ladder, but each ladder step requires 6 squarings and 9 multiplications in $\mathbb{F}_{p^2}$. "dadd" refers to differential addition. Formulas for fast computation of differential addition and doubling for Kummer coordinates come from [15].

Second, the choice of $\ell_A = 2$ and $\ell_B = 3$ allows for fast isogeny computations and fast isogeny evaluations between Montgomery curves, which are given in [7]. We note that an isogeny computation refers to computing a map between

Algorithm 1 Three-point ladder to compute $P + [t]Q$ [6]

Input: Points P and Q on an elliptic curve E, scalar t
1: Set $A = 0, B = Q, C = P$
2: Compute $Q - P$
3: **for** i decreasing **from** $|t|$ **downto** 1 **do**
4: Let t_i be the i-th bit of t
5: **if** $t_i = 0$ **then**
6: $B =$dadd(A, B, Q), $C =$dadd(A, C, P)
7: $A = 2A$
8: **else**
9: $A =$dadd(A, B, Q), $C =$dadd$(B, C, Q - P)$
10: $B = 2B$
11: **end if**
12: **end for**
Ensure: $C = P + [t]Q$

curves based on a kernel and an isogeny evaluation refers to converting points on one curve to its corresponding isogenous curve. We can perform these computations on Kummer coordinates because P and $-P$ generate the same subgroup of points. For the choice of a large degree isogeny computation over $\ell_A = 2$, we note that there is an additional isogeny across Montgomery curves to get a point of order 8 at the beginning for fast 2-isogenies. A 4-isogeny is required to finish the large degree isogeny at the end, when the point of order 8 is no longer valid.

No standardized parameters exist, so we determined a curve and basis points of the proper cardinality based on the prime $p = 2^{253}3^{161}7 - 1$ through a Sage script for use as a proof of concept.

III. Proposed Architectures for Isogeny Computations

In this section, we discuss the major design considerations to implement the SIDH key exchange protocol in hardware. To give an initial look into computing large isogenies in hardware, we chose to stick with 512-bit primes for our keys, which feature 85-bit quantum security and 128-bit classical security.

The high level design of the isogeny core is depicted in Fig. 4. This core features an adder unit, multiplier unit, inversion unit, RAM file for registers, and a ROM file for the controls. The RAM file contained 256 values in $\mathbb{F}_p$, or 256 512-bit entries. The RAM file contains constants for the parameters of the protocol, intermediate values within the protocol, and intermediate values for $\mathbb{F}_{p^2}$ computations. There are more intermediate values necessary for higher key sizes as the graph traversal of the large degree isogeny is higher, but 256 values is slightly more than enough, which allows more flexibility and optimization with routines. The size of the ROM unit depends on the size of the multiplier unit since more multipliers indicates that more multiplications operate in parallel for fewer stalls waiting for the multiplication result. Refer to Section IV-E for more details.

Similar to standard ECC, the performance of the key exchange is heavily dependent on its finite-field arithmetic. Here, we provide the designs used in our Field Arithmetic

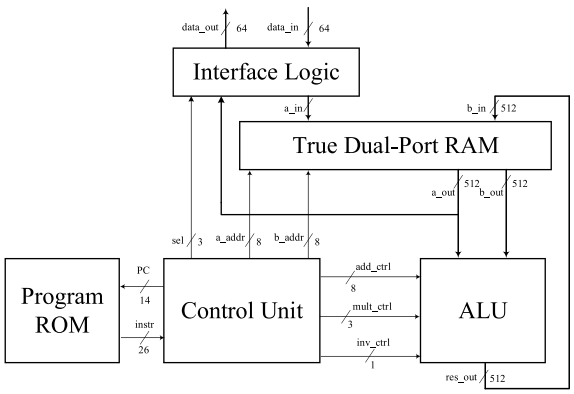

Fig. 4. Proposed High-level Architecture of an SIDH Core.

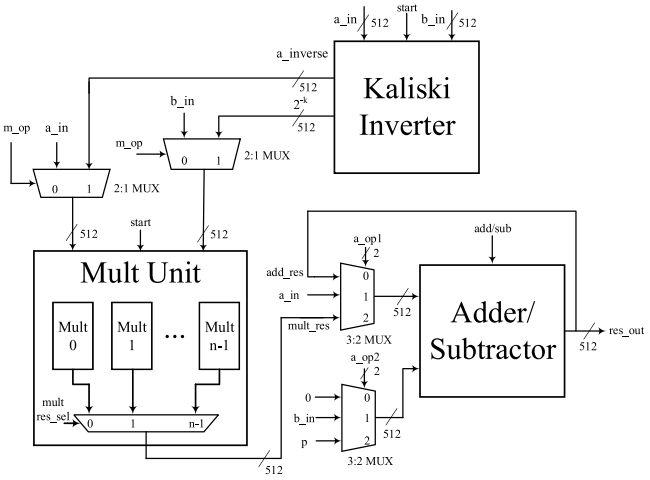

Fig. 5. Proposed Field Arithmetic Unit.

Unit (FAU) for finite field addition, multiplication, and inversion. Generally, we focused on optimizing the speed as much as possible. Thus, we chose state-of-the-art designs found in the literature that provide the best fit for our 512-bit prime. The FAU is shown in Fig. 5.

A. Field Adder

Finite-field addition is used to add two elements in $\mathbb{F}_p$ or for reduction. Further, addition is the essential part of the Kaliski almost inverse algorithm, discussed in Section III-D. High-radix implementations of adders are a powerful technique to design wide-word operands for adders. To the best of our knowledge, a high-radix implementation of the parallel prefix adder is the fastest adder circuit on modern FPGA's, which was designed by Rogawski *et al.* [16]. The high-radix parallel prefix adder (HRPPA) is designed based on using optimally embedded fast carry chain hardware on modern FPGA's. For demonstration purposes, assume the adder performs the addition $X+Y$, where $X, Y \in \mathbb{F}_p$. As shown in Fig. 6a, the HRPPA performs $|X + Y|_{2^n}$ in radix R, where $X = x_{n-1}...x_0$ and $Y = y_{n-1}...y_0$, by computing $P_{i\times R+R-1:i\times R} = \bigwedge_{l=(i\times R+R-1)}^{i\times R} p_l$ as a Group Propagate Signal (GPS) and $G_{i\times R+R-1:i\times R} = g_{i\times R+R-1} \vee p_{i\times R+R-1} G_{i\times R+R-2:i\times R}$ as a Group Generate Signal (GGS) as in Fig. 6b,where $p_j = x_j \oplus y_j$, $g_j = x_j y_j$, $0 < j < n$, $0 \leq i < k$, and $k = \lceil n/R \rceil$. The GPS and GGS are sent to a parallel prefix network (PPN) which can

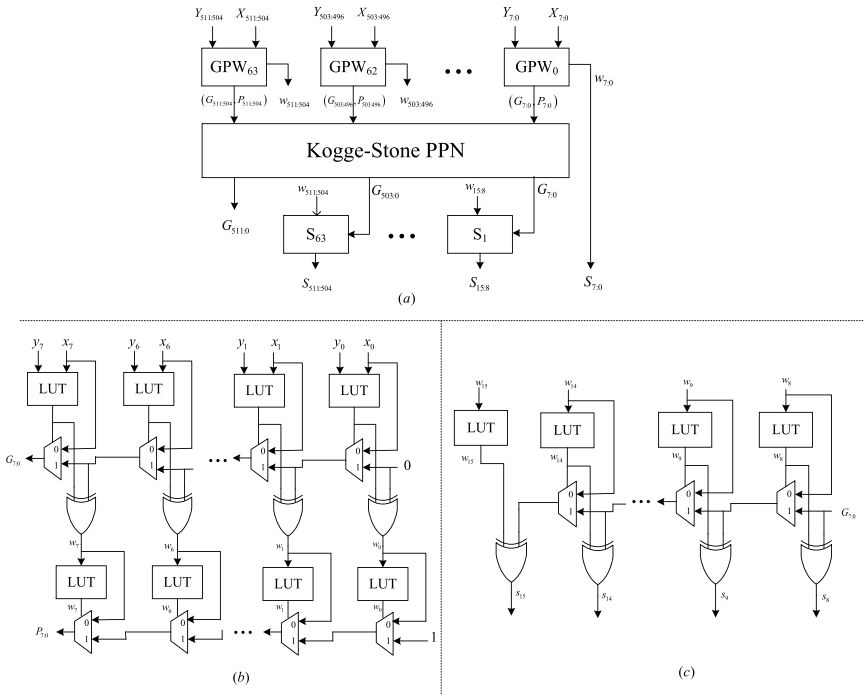

Fig. 6. High Radix Parallel Prefix Adder. (a) Design of the adder. (b) Design of the GGS-GPS-Intermediate Sum with carry chain in FPGAs. (c) Design of a sum unit. [16] We utilize $n = 512$ and $R = 8$,which is the best R to fit in the our design.

be designed using a variety of algorithms [17]. Finally, the outputs of the PPN are used as carry inputs for the final summation as in Fig. 6c. Our design implements the field adder using the HRPPA technique, where $n = 512$ and $R = 8$,which is the best R to fit in our design. Note that subtraction is also designed by flipping the second input and adding 1 as a carry-in. In practice, we found that using this adder method reduced the critical path delay of a single cycle addition over 512-bits by a factor of 15–20% over the standard IP in Xilinx Vivado with insignificant difference in area.

Unfortunately, the datapath from the RAM unit through a multiplexer to the adder/subtractor in the adder unit increased the critical path too much. Since the adder/subtractor was the bottleneck, we decided to perform the 512-bit addition/subtraction over two cycles with a carry-out from the first 256-bit addition/subtraction used as a carry-in for the second 256-bit addition/subtraction. This also required additional registers to register the control signals and output. However, this did not affect the performance of the protocol much, as the addition/subtraction functionality was easily pipelined and two operations could fit in the pipeline at any given time.

On the other hand, our inversion module, which utilizes individual 512-bit additions or subtractions was able to compute individual 512-bit additions or subtractions in a single cycle with a high clock frequency. The use of the fast addition and subtraction here improved our overall timing by around 600 hundred thousand cycles for the entire protocol. We discuss this further in Section III-D.

B. Field Multiplier

Finite-field multiplication computes the product $C = A \times B$, where $A, B, C \in \mathbb{F}_p$. Since the product is double the size of the inputs, a reduction must be performed so that the product is still within the field. However, the smooth isogeny primes of the form $\ell_A^a \ell_B^b \cdot f \pm 1$ do not resemble the form of pseudo-Mersenne primes, $2^m - c$, and cannot take advantage of fast reduction. Montgomery reduction [18] and the fast-Fourier transform [19] are the primary algorithms which are used for designing modular multipliers over long operand integers. However, the fast-Fourier transform method is most often implemented only when the operand width is large enough (e.g., more than 3072 bits [20]). On the other hand, Montgomery multiplication is a suitable choice for general moduli since the algorithm combines multiplications and modular reduction. Montgomery multiplication converts time-consuming trial divisions to shift operations, which is simple to do in hardware. Thus, we propose to use Montgomery [18] modular multiplication. Algorithm 2 demonstrates the Montgomery reduction procedure.

Algorithm 2 Montgomery Reduction [18]

Input: An odd m-bit modulus M
Montgomery radix $R = 2^m$
An operand T where $T = A \cdot B$ in the range $[0, 2M - 1]$
Pre-computed constant $M' = -M^{-1} \bmod R$
Output: Montgomery product $Z = T \cdot R^{-1} \bmod M$
1. $Q = T \cdot M' \bmod R$
2. $Z = (T + Q \cdot M)/R$
3. **if** $Z \geq M$ **then** $Z = Z - M$ **end if**
4. **return** Z

Montgomery multiplication utilizes both input operands in the Montgomery form $\hat{A} = AR \bmod p$, where R is a power of 2 just greater than p. For our 512-bit implementation, $R = 2^{512}$. We convert our starting parameters for

Algorithm 3 High-Radix Montgomery Multiplication Algorithm [22]

Input: $M = p$, $M' = -M^{-1} \bmod p$
$A = \sum_{i=0}^{m+2} (2^k)^i a_i, a_i \in \{0, 1 \ldots 2^k - 1\}, a_{m+2} = 0 B = \sum_{i=0}^{m+1} (2^k)^i b_i, b_i \in \{0, 1 \ldots 2^k - 1\} \overline{M} = (M' \bmod 2^k) M = \sum_{i=0}^{m+1} (2^k)^i m_i A, B < 2\overline{M};\ 4\overline{M} < 2^{km} R = 2^{\lceil \log_2 p \rceil}$
Output: $A \times B \times R^{-1} \bmod M$
1. $S_0 = 0$
2. **for** $i = 0$ **to** $m + 2$ **do**
3. $q_i = (S_i) \bmod 2^k$.
$S_{i+1} = (S_i + q_i \overline{M})/2^k + a_i B$
5. **end for**
6. **return** $S_{m+3} = A \times B \times R^{-1} \bmod M$

the standard domain to the Montgomery domain by using $\hat{A} = \text{MonMult}(A, R^2) = A \times R^2 \times R^{-1} = AR$. When both inputs are in the Montgomery form, the result will also be in the Montgomery domain, $\hat{C} = \text{MonMult}(\hat{A}, \hat{B}) = AR \times BR \times R^{-1} = ABR$.

Ultimately, we chose to utilize the high-radix Montgomery multiplier proposed in [21]. This multiplier uses a systolic array of processing elements to compute various parts of the Montgomery multiplication and reduction. Furthermore, it can perform two multiplications simultaneously, to utilize all of its processing elements. Lastly, we liked the utilization of DSP48 blocks in the multiplications to utilize more features of FPGA designs.

1) Interleaved Montgomery Multiplication: The McIvor [21] multiplication is based on the high-radix form of Montgomery multiplication, which is shown in Algorithm 3. Essentially, there are $m+2$ processing elements. M' is a constant based on the form of the prime. Similar to the best results in [21], we utilize a 2^{16} radix, so $m = 32$ and there are 34 processing elements. The critical path delay of this design is two 16-bit multiplications in parallel followed by a 32-bit addition. A 16x16 multiplication is efficiently implemented with a single DSP48 block in an FPGA. Two Montgomery multiplications can be performed in parallel because a single multiplication only occupies the even or odd processing elements at a given time. This introduces the idea of an even-odd dual multiplier. Additional registers and control is necessary for the even-odd dual multiplier, but reuse of the processing elements achieves almost full utilization of the multiplication hardware.

The multiplier closely follows Algorithm 3. Initially, the starting value must be cleared, so a reset pump is initiated. This reset pump slowly pumps through the processing elements, one cycle at a time. This reset clears the output value S_i for the processing element. Next, the values of a_i are shifted through the processing elements one by one. The least significant digit of A passes into the second processing element just as the reset pump clears the summation register, allowing the first multiplication of $a_0 b_0$. The value for A is funneled through until it has multiplied with all b_i. Thus, this performs the computation of $a_i B$. In the algorithm, there is also a use of feedback to multiply the Montgomery modulus by a previous sum modded by 2^k and added by S_i. This functionality also goes through a divide by 2^k, which is simply a shift in hardware. Thus, after $m + 3$ cycles, the full result of the least significant digit of the result is ready. Every 2 cycles, an additional digit of the result is ready. Therefore, the final Montgomery result is available after $3(m+2)+1$ cycles with $m + 2$ processing elements.

We note that smooth isogeny primes that have the form $2^a 3^b f - 1$ have all "1"s for their least significant digits. This has also been known as a Montgomery friendly [23] prime. This ensures that M' is 1 and no multiplication from the standard modulus is necessary. This also removes two additional processing elements at the end, since the final $m+1$ and $m + 2$ digits are 0. Thus, our scheme requires only $3m + 3 = 99$ cycles instead of the proposed $3m + 3 = 103$ cycles in the original paper.

However, the processing elements are not always in use. We note that on the last m cycles of the multiplication that the least significant processing elements are increasingly not used. Thus, we propose to slowly shift in the new operand B in order just as the previous multiplication does not need the processing element. The second multiplication can start on cycle $2m + 3 = 68$. Thus, now we can interleave multiplications and achieve 100% utilization of the multiplier. Overall, this means that the multiplier performs a single multiplication in 99 cycles, but can simultaneously perform two multiplications and further multiplications can be interleaved to make the multiplier seem like it is has a latency of 68 cycles. We wanted high throughput for our multiplier and these two characteristics of this particular multiplier make it a strong choice for maximum throughput.

Fig. 7 shows the design of the multiplier. (a) has the multiplier overview, (b) has the Processing Element, (c) shows logic for the inputs A1 and A2, and (d) shows the first processing element. Signals like B2c4 indicate the fourth cycle that B2 has been started. MSD stands for most significant digit and LSD stands for least significant digit, as in the most significant or least significant digits in the $2k + 1$ bit result. On even cycles, the even positions are performing the even multiplications and the odd shift register is shifting. Similarly, on odd cycles, the odd positions are performing the odd multiplications and the even shift register is shifting.

C. Multiplier Unit

We note that the latency of multiplication in $\mathbb{F}_p$ is much greater than that of addition. Further, the Montgomery multiplier is not pipelinable as it already utilizes all of its processing elements. Multiplication in $\mathbb{F}_p$ is used numerous times in squaring, multiplication, and inversion in $\mathbb{F}_{p^2}$, and multiple multiplications can easily be done in parallel. Thus, we propose to use a multiplier unit that features replicated Montgomery multipliers based on our interleaved variant to push the parallelization of isogeny-based algorithms further.

Our multiplier unit is designed as a first-in-first-out (FIFO) circular buffer. Multiplication instructions are issued cyclically starting from multiplier 0 to multiplier $2n - 1$ for n dual multipliers. Likewise, the results are read starting from multiplier

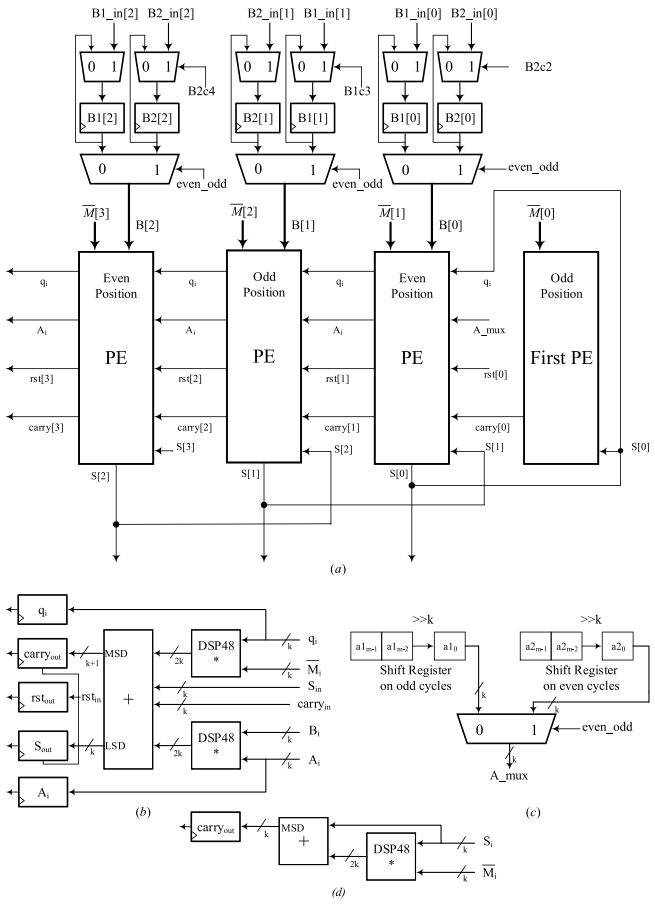

Fig. 7. Proposed Interleaved high-radix [21] Montgomery multiplier. (a) Systolic array of the processing elements. (b) Single processing element. (c) Design of the input for A. (d) Design of the first processing element.

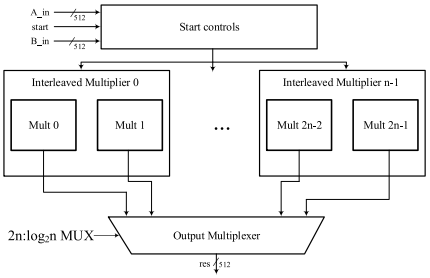

Fig. 8. FIFO Multiplier Unit Datapath.

Algorithm 4 Kaliski Almost Montgomery Inverse Algorithm [24] [25]

Input: $A \in \mathbb{F}_p$, p with $\gcd(A, p) = 1$ and $0 \leq A < pR = 2^{\lceil \log_2 p \rceil}$
Output: $A^{-1} \times R \bmod p$,
where $2\lceil \log_2(p) \rceil \leq k \leq 2\lceil \log_2(p) \rceil$
1. $u \leftarrow p,\ v \leftarrow A,\ r \leftarrow 0,\ s \leftarrow 1,\ k \leftarrow 0$
2. **while** $v \neq 1$ **do**
3. **if** $u \equiv 0 \pmod 2$ **then**
4. $u \leftarrow u/2, s \leftarrow 2 \times s$
5. **else if** $v \equiv 0 \pmod 2$ **then**
6. $v \leftarrow v/2, r \leftarrow 2 \times r$
7. **else if** $u > v$ **then**
8. $u \leftarrow (u - v)/2, r \leftarrow r + s, s \leftarrow 2 \times s$
9. **else**
10. $v \leftarrow (v - u)/2, s \leftarrow r + s, r \leftarrow 2 \times r$
11. **end if**
12. $T = \text{MonMult}(s, 2^{2m-k})$
13. **return** $A^{-1} \times R \bmod p = \text{MonMult}(T, R^2)$

0 to multiplier $2n - 1$. This ensures that the data integrity of the multiplier is valid. The first multiplier in the buffer holds multipliers 0 and 1, the second holds multipliers 2 and 3, and so on. With the even-odd nature of the multiplier, this means that a multiplication can be issued every cycle so long as there are enough multiplications and the multiply unit starts on the correct even-odd cycle. We discuss scheduling these multiplications in Section IV-C.

The output of the multiplier unit is directly connected with the addition unit so that the final reduction can occur. The total cost of a multiplication in $\mathbb{F}_p$ is a memory load, multiplication, reduction, and store, which is $2 + 99 + 2 + 1 = 104$ cycles.

The FIFO multiplier unit requires wires and connections for the input operands and an additional 512-bit multiplexer of size $2n : \log_2 2n$ for the output. This increases the area of the multiplier unit considerably, but the target is speed and the protocol can benefit from issuing many multiplications in parallel. We show a miniature design of the FIFO multiplier in Fig. 8. In this figure, the individual multipliers are started in order and the output multiplier outputs the currently read multiplier.

D. Inversion Unit

Finite-field inversion finds some A^{-1} such that $A \cdot A^{-1} = 1$, where $A, A^{-1} \in \mathbb{F}_p$. There are many schemes to perform this efficiently. Fermat's little theorem exponentiates $A^{-1} = A^{p-2}$. This requires many multiplications and squarings, but is a constant set of operations. The Extended Euclidean Algorithm (EEA) and Kaliski's almost inverse algorithm [24] both have a significantly lower time complexity of $O(\log^2 n)$ compared to $O(\log^3 n)$ for Fermat's little theorem. There are over 800 inversions in the full SIDH protocol for 85-bit quantum security, so we chose to use a non-constant time inversion. Further, we use operands in the Montgomery space. Therefore, Kaliski's almost inverse is a suitable fit for this particular implementation because it uses only simple shifts and additions and the final result is also in the Montgomery domain. Kaliski's almost inverse is shown in Algorithm 4. In total, the inversion unit requires four shift registers, one 512-bit addition, and two 512-bit subtraction modules. Both results from the Kaliski module are fed directly to the multiplier. After two additional multiplications the inversion result is ready.

The Kaliski almost-inverse can run from 512 iterations to 1024 iterations. The experiment in [26] showed that the average number of iterations is around 1.412 times the word size. Thus, we will assume that the average case for the Kaliski almost inverse requires 723 iterations. With our choice of fast adder and subtractor, we could perform a single 512-bit

addition or subtraction in one cycle. Every other cycle, we check the registers for how to proceed and then update the registers the next cycle with the result from the adders and subtractors. Thus, each iteration requires 2 cycles and the expected time of the inversion in $\mathbb{F}_p$ is $1446+2M = 1656$ cycles. If we had used 512-bit addition or subtractions over two cycles, the expected time of the inversion would increase to 2379 cycles, so the fast addition and subtraction improved the computation of inversion by about 31%.

1) Side-Channel Considerations:: The Kaliski almost inverse is not constant time, as the algorithm can run from 512 to 1024 iterations. To make it constant-time, one can ensure that the full 1024 iterations is always experienced. This increases the total time of the inversion by 602 cycles. Over 800 inversions, this increases the total latency by 481 600 cycles. This reduces the impact of timing analysis, but the inverter is most likely stalled for the last iterations, which could still reveal information about the number being inverted. Under this scheme, the cost of the inversion is 2258 cycles, or 1.36 times more.

Another approach to mystify the inversion is to multiply the value to be inverted by a random value before and after the inversion. This aids in defending against simple power analysis attacks. Adding two multiplications increases the cost of the inversion to 1866 cycles, or 2468 cycles for constant-time.

We provide these alternatives to the standard Kaliski almost inverse because inversion is the only non-constant finite field computation. Addition chains have been created as a way to perform large exponentiations with as few multiplications as possible. Using exponentiation with small addition chains is still much more expensive than the Kaliski method. For instance, let us assume that we utilize an addition chain with 600 multiplications for our 512-bit prime. Little parallelism can be taken advantage of in large exponentiations as it is primarily an accumulator being squared and multiplied. Using a serial multiply strategy for inversion requires $600 \times 105 = 63\,000$ cycles. This is more than 27 times greater than the Kaliski inversion with added side-channel resistance measures. Thus, for the large number of inversions in the current SIDH scheme the constant-time inversion is impractical.

IV. Instruction Scheduling

This section details our approach to maximizing the throughput of our architecture. Instruction scheduling was done in a Python script and the results were put into a ROM unit to issue instructions to the control unit. The total number of instructions depends on the size of the multiplier unit. We optimized based on SIDH, but a similar approach can be taken for the other protocols.

A. Extension Field Arithmetic

As was previously stated, SIDH operates in the extension field $\mathbb{F}_{p^2}$. For this extension field, we use the irreducible polynomial $x^2 + 1$ based on our prime choice. With this, we propose reduced arithmetic in $\mathbb{F}_{p^2}$ based on fast arithmetic in $\mathbb{F}_p$. These equations were made in a Karatsuba-like fashion to reduce the total number of multiplications and squarings.

TABLE I
Cost of Operations in $\mathbb{F}_{p^2}$ for 512-bit SIDH

Ops in $\mathbb{F}_{p^2}$	Ops in $\mathbb{F}_p$			Latency
	(A)	(M)	(I)	(cc)
Addition ($\tilde{A}$)	2	0	0	8
Squaring ($\tilde{S}$)	3	2	0	114
Multiplication ($\tilde{M}$)	5	3	0	128
Inversion ($\tilde{I}$)	2	4	1	1886[1]

1. Assuming a Kaliski almost inverse requires 723 iterations.

The lazy reduction technique was also employed for inversion to minimize computational cost. For the equations below, assume $A = (A_0, A_1)$, $B = (B_0, B_1) \in \mathbb{F}_{p^2}$.The results of operations in $\mathbb{F}_{p^2}$ are $C = (C_0, C_1)$

$$\begin{aligned} A + B &= (A_0 + B_0, A_1 + B_1) \\ A - B &= (A_0 - B_0, A_1 - B_1) \\ A \times B &= (A_0B_0 - A_1B_1, (A_0 + A_1), \\ &\qquad (B_0 + B_1) - A_0B_1 - A_1B_0) \\ A^2 &= ((A_0 + A_1)(A_0 - A_1), 2A_0A_1) \\ A^{-1} &= (A_0(A_0^2 + A_1^2)^{-1}, -A_1(A_0^2 + A_1^2)^{-1}) \end{aligned}$$

The total cost of all arithmetic in $\mathbb{F}_{p^2}$ is shown in Table I. The total number of clock cycles assume that there is no competition for resources and enough multipliers available to issue multiplications. The latency also indicates when the results have been written to the RAM unit. For this table, $\tilde{I}$ stands for inversion, $\tilde{M}$ stands for multiplication, $\tilde{S}$ stands for squaring, and $\tilde{A}$ stands for addition, all in $\mathbb{F}_{p^2}$. The absence of a tilde indicates an operation in $\mathbb{F}_p$. We did not implement a dedicated squaring unit, so squaring has the same latency as multiplication. Implementing a dedicated squaring would have only a small effect on the latency of the protocol since squaring is only used twice in inversion in $\mathbb{F}_{p^2}$ and the multiplications can already be interleaved effectively.

B. Scheduling Methodology

For our architecture, a few things must be kept in mind in regards to scheduling. First, a memory load and memory store cannot occur on the same cycle as the address for the dual-port RAM is the same for each. Second, multiplications are issued and reduced in order. Third, instructions should be ordered in such a way to minimize data dependency and emphasize parallel computations (e.g., 3 point Montgomery ladder that performs 2 point additions and 1 point doubling in parallel).

Extension field arithmetic was scheduled by using a greedy algorithm. The instructions were compiled in order. Additions and subtractions were issued if a load, add, reduce, and store sequence could fit. Multiplications were issued if the load and multiply controls were available as well as an available multiplier. For multiplications, squarings, and inversions in $\mathbb{F}_{p^2}$, the multiplication and addition instructions were issued in advance, meaning that multiplications had priority in the load and store controller for the RAM unit. In practice, we found this to be most effective as the multiplication latency in $\mathbb{F}_p$ was 15 times greater than the add/subtract latency in $\mathbb{F}_p$.

TABLE II
EVEN-ODD MULTIPLICATION PIPELINE STALL EXAMPLES

Cycle	-7	-6	-5	-4	-3	-2	-1	0	1	2	3	4	#stalls
Before Stall	ldadd1	ldadd2	nop	nop	nop	ldmult	store	store	nop	nop	nop	nop	
After Stall	nop	ldadd1	ldadd2	nop	nop	nop	ldmult	store	store	nop	nop	nop	1
Before Stall	nop	ldadd1	ldadd2	storeX	storeY	1dmult	ldadd3	store1	store2	nop	nop	nop	
After Stalls	nop	nop	ldadd1	storeX	storeY	ldadd2	ldmult	nop	store1	ldadd3	nop	store2	3
Before Stall	nop	nop	ldmult1	ldmult2	nop	ldmult3	ldadd1	nop	nop	nop	nop	nop	
After Stall	nop	nop	ldmult1	ldmult2	nop	nop	ldmult3	ldadd1	nop	nop	nop	nop	1

Thus, even if the addition or subtraction was pushed back several cycles because of multiply logic in the pipeline, the additions and subtractions could still be finished long before the multiplications are done. Further, the data dependencies based on the multiplication required much more time than those on the additions, so starting multiplications first is imperative to reducing the total number of cycles.

Each instruction is 26 bits long. The first and second bit indicate a special register address for a load or store, as indicated by the controller instead of the instruction line. This was used for scheduling isogeny computations, shown in Section IV-D. The third bit was reserved for starting the inversion unit with the data loaded from RAM. Bits 4 and 5 were reserved for the multiplication unit. The multiplication could use registers outputted from the Kaliski inversion module or RAM file and the even-odd parameter could be reset. Bit 6 indicated if the registers should be read into the adder/subtractor unit. Bits 7–9 were used by the adder/subtractor. The adder/subtractor could perform a memory add, accumulator add, and reduction. The subtraction option was also available and the multiplication result could be reduced here as well. Bit 10 indicates if a write to the RAM unit at address A should be done. Lastly, bits 11-18 indicated the address used by the first port (A) of the RAM unit and bits 19–26 indicated the address used by the second port (B) of the RAM unit.

Pipeline stalls were experienced for data dependencies. If an input to a multiplication or addition was not ready, further instructions could not be issued.

There are 256 values in $\mathbb{F}_p$ in memory to hold and store values. Squaring, multiplication, and inversion in $\mathbb{F}_{p^2}$ each require temporary variables to store intermediate values. Multiplication requires the most at 4 intermediate registers to achieve full throughput. We further note that there can be at most 6–10 multiplication instructions at one time, so we dedicate the final $10 \times 4 = 40$ values in our RAM unit to store the intermediate values for these operations. Based on the number of intermediate values needed for the rest of this SIDH protocol, reserving 40 values for finite-field operations did not contest with any other parts of the protocol.

Inversions in $\mathbb{F}_p$, which are required for the inversion in $\mathbb{F}_{p^2}$ require all previous additions to be finished. Since the Kaliski inversion is not constant-time, the control unit waits for a response from the inversion unit indicating that the inversion result is ready.

C. Scheduling with Even-Odd Multiplications

The only caveat to using the interleaved multiplier that we proposed earlier was that the multiplications must be issued on even instructions that go to even starting multipliers and odd instructions that go to odd starting multipliers. Our proposed design issues in order: odd multiplier, then even multiplier, then odd multiplier again. This way, multiplication instructions could theoretically be issued every cycle so long as there are enough multipliers. However, the data dependencies in the SIDH protocol only require about 3–5 double multipliers at max. This equates to a maximum of 10 multiplications running at any given time.

The greedy algorithm for scheduling above does not necessarily account for the even/odd nature of the multiplier. Further, assigning multiplications in the future may have different even-odd values if an additional multiplication was inserted earlier. Thus, we enhanced the scheduling by tracking if even multiplier or odd multipliers were next. Our approach checks all instructions in order, changing the value of even-odd if a multiplication is experienced. If a multiplication occurred on a cycle that did not match the even-odd nature, then we incurred stalls. Unfortunately, there are a few specific cases that need to be taken into account when delivering the stall cycle. Notably, if there was a memory store the cycle after the memory load for the multiplication, then the addition/subtraction pipeline that got this result must be stalled as well. We show a few exceptions in Table II. For this table, assume that a multiplication initially occurs at cycle 0 and the load for the multiplication occurs at cycle -2. If cycle 0 does not match up with the current selection of even and odd, then it must be pushed back to have a load at-1 and a multiply at 1. Our observations show that we must push any add/sub instruction starting with a load starting on or after -7 each back at least 1 cycle. The cycle -7 is critical because it is the total delay of a load, add, reduce, and store instruction pipeline. We must push all reduction instructions (including multiplication reduction) starting on or after -3 each back 1 cycle, since the reduction indicates a store on -1 which will be used by the load multiply. We must push any memory loads on -1 back a single cycle. Lastly, all instructions in the memory, add, multiply, and inversion datapath starting on or after 0 must be pushed back a single cycle. With these corrections, we ensure data integrity.

Table II also illustrates an example of a multiple stall incurred by the multiplication starting on the wrong cycle. We ensure that any stores or load mults before cycle -2 must be preserved. This ensures that the previous multiplications (which are already on the correct starting cycle) stay valid and the datapath leading up to the store is not altered. We also cannot interfere with the addition pipeline that led up to the stall. With these additional requirements, multiple stalls could occur if there are load add instructions before

Algorithm 5 Rescheduling Incorrect Starting Multiplications

Input: Multiplication on current_cycle is on the wrong even-odd cycle
Output: Multiplication pushed back by 1 cycle and all data pipeline in tact
1. index = current_cycle − full_add_delay, num_stalls = 1
2. **while**index = current_cycle−read_delay **do**
3. **if**mem_ctrl[index] has a load and add path **then**
4. **while**(index + num_stalls is a store —— load mult) && index+num_stalls < current_cycle−read_delay **do**
5. copy load pipeline at index
6.num_stalls+ = 1
7. **end while**
8. Copy contents from index to index+num_stalls
9. **end if**
10.index+ = num_stalls
11. **end while**
12. **for**all other cycles after current_cycle–1
13. Copy contents num_stalls forward
14. **end for**
15. Push multiplication on wrong cycle forward

a load multiplication instruction and other stores or load multiplications blocking the load add from simply going a single cycle forward. The worst possible case is a 4 cycle stall, but this was not experienced. For our routines, there are a total of 104 single stalls, 28 double stalls, and 26 triple stalls, and 0 quadruple stalls.

Our rescheduling algorithm for these incorrect even/odd starting multiplications is depicted in Algorithm 5. For our implementation, the full_add_delay is 7 cycles and the read_delay is 2 cycles. All copies are made to a new schedule and the new schedule is copied back at the end.

There are a few exceptions to the above algorithm that are accounted for. Notably, if a first load add has a stall of one cycle and the second load add has a stall of two cycles, the addition unit might experience a collision from both load adds utilizing it. We carefully considered each case and ensure that the minimum number of stalls is incurred.

D. Scheduling Isogeny Computations and Evaluations

Here, we describe how our control unit implemented the large degree isogeny computation. For our particular implementation, we targeted isogeny degrees of size 2^{253} and 3^{161}. However, we give a generic implementation for any parameter in Algorithm 6, which was adapted from [7]. This algorithm efficiently iterates to the leaves of the large degree isogeny graph, shown previously in Fig. 2, as long as optimal splits are given. Finding optimal splits was done externally using a Sage script and is based on a combinatorial problem. The optimal splits were placed in the control unit as a large lookup table in ROM. One lookup table was used to hold the optimal splits of 2^{253} and 3^{161}, but the most significant address bit indicated if it was Alice's splits or Bob's splits. There were 418 ROM lines of 8 bits to hold both Alice's and Bob's splits.

Algorithm 6 Computing a large degree isogeny using an optimal strategy

Input: Isogeny degree e in ℓ^e
A lookup table of size e with the optimal strategy for S
Kernel point (X_0, Z_0)
Montgomery curve: $A_0y^2 = x^3 + B_0x^2 + xs_i$ is the optimal split for isogeny graph
Stack structure composed of (X_i, Z_i, s_i)
Output: Isogenous curve $A_ey^2 = x^3 + B_ex^2 + x$
1. **push**(X_0, Z_0, e)onto the empty stack
2. **while**the stack is not empty **do**
3. (X_i, Z_i, s_i) =the top of the stack
4. All points in stack on $A_ky^2 = x^3 + B_kx^2 + x$
5. $h = s_i$,split $= S_{s_i}$
6. **while**$h > 1$**do**
7. **for** j in range(0,h − split)
8. $(X_i, Z_i) = \ell \times (X_i, Z_i)$, scalar point multiplication
9. **end for**
10. **push**$(X_i, Z_i,$ split)
11.$h =$ split
12. **end while**
13. compute ℓ-isogeny mapping
14. e.g., $(iso, A_{k+1}, B_{k+1}) = \ell$comp$(A_k, B_k, X_i, Z_i)$
15. **for** j in range(0,len(stack))
16. push point from curve (A_k, B_k) to curve (A_{k+1}, B_{k+1}) using isogeny mapping, iso
17.$(X_j, Z_j) = \ell$apply(iso, X_j, Z_j)
18. **end while**
19. **return**$A_ey^2 = x^3 + B_ex^2 + x$

As Algorithm 6 shows, the ordering of the point multiplications resembles a recursive function. The initial large isogeny degree is broken down by an optimal split. When that split is reached through a series of point multiplications, the result of that point multiplication is pushed to the top of a stack. The isogeny 2^{253} requires a maximum stack size of 13 points in Kummer coordinates. Thus, we must reserve 52 values in $\mathbb{F}_p$ in the RAM unit to hold each of these. To ensure correct scheduling of point multiplications, we kept track of the size of the stack and would apply point multiplications and isogeny computations to the top of the stack. The control unit would alter the addresses for these instructions to point to the coordinates at the top of the stack. To apply isogeny mappings, the control unit would iterate from the top of the stack to the bottom and push each Kummer point (X_i, Z_i) to the new curve from the isogeny relationship. Only one curve, E, is needed to be stored at any given time. The total increase to the area for implementing this queue system and other control logic was minimal compared to the size of the FAU and datapath.

E. Total Cost of Routines

Table III demonstrates the total cost of the routines for our isogeny core, dependent on the number of multipliers in the multiply unit. It is also assumed that there is multiplication interleaving every 68 cycles. The completion of a protocol indicates that all of its results have been stored to memory.

TABLE III
COST OF ROUTINES FOR OUR PROCESSOR

Routine	Ops in $\mathbb{F}_{p^2}$				#ops in	Latency for n mults (cc)				
	$(\tilde{A})$	$(\tilde{S})$	$(\tilde{M})$	$(\tilde{I})$	protocol	2	4	6	8	10
Subtract Points (init)	15	2	17	3	4	7702	7072	6810	6790	6715
Differential Ladder Step	12	6	9	0	1016	1544	945	802	709	652
Mont Double	4	2	3	0	1668	579	461	410	409	409
Mont Triple	8	4	7	0	996	1255	994	828	828	828
Compute Mont-isogeny	13	4	12	2	2	5587	5289	5241	5241	5241
Evaluate Mont-isogeny (proj)	2	0	6	0	1	706	471	355	348	343
Evaluate Mont-isogeny (Kummer)	1	0	2	0	24	263	262	266	267	266
Compute 2-isogeny	5	1	5	1	502	2614	2562	2504	2504	2504
Evaluate 2-isogeny (proj)	2	6	8	0	251	1352	757	576	505	476
Evaluate 2-isogeny (Kummer)	1	1	2	0	2472	349	252	254	255	254
Compute 3-isogeny	6	1	5	1	322	2566	2436	2421	2421	2421
Evaluate 3-isogeny (proj)	9	2	24	0	161	2814	1534	1130	1021	970
Evaluate 3-isogeny (Kummer)	2	2	4	0	1440	663	467	404	402	403
Compute 4-isogeny	4	0	3	1	2	2202	2156	2157	2158	2158
Evaluate 4-isogeny (proj)	20	10	18	0	1	2652	1470	1121	1040	990
Compute j-invariant	11	2	2	1	2	2438	2254	2253	2254	2254

We counted operations in $\mathbb{F}_p$ as a half operation in $\mathbb{F}_{p^2}$. There were a few smaller routines for the protocol, but these merely include moving registers so that the routines only have to be implemented once over specific registers.

Description of routines:

- *Subtract Points (init):* We perform a projective subtraction of Montgomery coordinates and perform inversions on the projective coordinates so that the Z-coordinate is 1 to speed up the differential ladder by 2 multiplications.
- *Differential Ladder Step:* We compute two differential point additions and one differential point doubling for each step of a 3-point differential Montgomery ladder.
- *Compute ℓ-isogeny:* We compute an isogeny of degree ℓ. The Mont isogeny produces a point of order 2 for 2-isogenies. The 4-isogeny is only used at the end of the large degree isogeny computation for 2^a.
- *Evaluate ℓ-isogeny:* We convert points from their starting curve to the isogenous curve. We convert the public points using the projective formula and the points in the kernel with the Kummer formula. Projective conversion only needs to be done on the first round for both parties. Projective conversion always uses two points, so we loop unrolled the routine for two points, which reduces stalls by data dependencies and allows us to reuse shared computations.
- *Compute j-invariant:* We compute the quantity that determines a curve's isomorphism structure, which is used as the shared secret.

As this table shows, generally increasing the number of multipliers allows more parallelism to be exploited, thus improving the performance. However, this impacts certain algorithms much more than others. For instance, evaluation of isogenies over projective coordinate has many multiplications in parallel, which means that more multipliers allow these to be done quicker. The large degree isogeny computation required approximately 65% of the round time for Alice and 58% of the round time for Bob.

With more multiplications, the memory and addition units become the bottleneck. There are a few cases where more multiplications did not improve the time, such as computing a 4-isogeny. For this case, 2 dual multipliers provides a better time most likely because the order of additions and storings is more efficient. The code generation software aggressively schedules multiplications in the future, which might not benefit for cases like this. However, the difference is only 1 to 3 cycles over the course of the entire routine for these.

TABLE IV
SIDH ROUND COMPUTATIONS

Routine	Latency for n interleaved mults ($cc \times 10^6$)				
	2	4	6	8	10
Alice1	2.328	1.792	1.655	1.615	1.592
Bob1	2.373	1.720	1.490	1.447	1.425
Alice2	1.986	1.602	1.510	1.487	1.472
Bob2	1.925	1.477	1.312	1.287	1.272
ROM	24,157	17,656	15,666	15,210	14,888

One can also loop unroll several isogeny evaluations. As an example, instead of performing a single Kummer evaluation over 2-isogenies the control unit could perform multiple 2-isogeny evaluations at once, depending on the queue size. For instance, three 2-isogeny evaluations over Kummer coordinates could be easily parallelized since there are relatively few data dependencies. Instead of requiring 254 cycles for a 2-isogeny evaluation over 3 dual multipliers, it requires 373 cycles for a triple 2-isogeny evaluation. However, this makes the control logic more complex with how the controller determines the 2-isogeny evaluations as well as the inputs to the 2-isogeny evaluation equations. This shows that the isogeny evaluations can benefit from loop unrolling since the isogeny evaluations over Kummer coordinates has fairly trivial computations.

We also give an approximate latency of the entire protocol for both party's computations in Table IV as well as the total number of lines in the ROM file for the number of interleaved multipliers.

V. FPGA IMPLEMENTATIONS

Our SIDH core was compiled with Xilinx Vivado 2015.4 to a Xilinx Virtex-7 xc7vx690tffg1157-3 board. Since the software results available to compare against are primarily

TABLE V

IMPLEMENTATION RESULTS OF SINGLE-CORE ARCHITECTURES FOR KEY EXCHANGE PROTOCOL BASED ON ISOGENIES ON ELLIPTIC CURVES FOR XILINX VIRTEX-7 FPGA FOR 512-BIT PRIME (85-BIT QUANTUM AND 128-BIT CLASSIC SECURITY LEVEL)

Type	#Mults	Area					Time			SIDH/s
		#FFs	#LUTs	#Slices	#DSPs	#BRAMs	Freq (MHz)	Latency ($cc \times 10^6$)	Total time (ms)	
Virtex-7	2	15,232	13,311	5,298	64	33	191.2	8.610	45.0	22.2
	4	22,170	18,129	7,464	128	28.5	159.6	6.592	41.3	24.2
	6	30,031	24,499	10,298	192	27	177.1	5.967	33.7	29.7

TABLE VI

IMPLEMENTATION RESULTS OF MULTI-CORE ARCHITECTURES FOR KEY EXCHANGE PROTOCOL BASED ON ISOGENIES ON ELLIPTIC CURVES FOR XILINX VIRTEX-7 FPGA DEVICE FOR 512-BIT PRIME (85-BIT QUANTUM AND 128-BIT CLASSIC SECURITY LEVEL)

Type	#Cores	Area					Timing			Op/s
		#FFs	#LUTs	#Slices	#DSPs	#BRAMs	Freq (MHz)	Latency ($cc \times 10^6$)	Total time (ms)	
Virtex-7	10	152,134	129,784	55,838(51%)	640	270	167.4	8.610	51.4	195
	15	228,148	194,528	81,602(75%)	960	405	155.0	8.610	55.5	270
	18	273,756	233,301	94,835 (88%)	1152	486	153.1	8.610	56.2	320

using high performance desktop processors from the Opteron or Haswell architectures, we justify the use of this powerful board to make a comparison between software and hardware implementations of the SIDH protocol. All results were obtained after place-and-route.

A. Implementation Results and Discussion

Our first architecture is a single SIDH core which features a variable number of multipliers. We place all area and timing results for the single-core architecture in Table V. The implementation was optimized to reduce the net delay to maximize the clock frequency. The latency results are based on the assumption that Kaliski inversion requires 723 iterations.

Thus, as we can see, the performance of the protocol does improve by up to a factor of 1.35 as the number of multipliers increases. There are diminishing returns from using more multipliers. Three interleaved multipliers appears to be the best choice for performance. Four interleaved multipliers provides a similar complexity as three multiplications, but with many more resources. Increasing the number of multipliers decreases the number of block RAM's because there are fewer instructions in the instruction ROM. Approximately 30 full supersingular isogeny Diffie-Hellman key exchanges can be pushed through per second with 3 dual multipliers.

The critical path delay was typically the cost of the single cycle adders and subtractors used in the Kaliski inversion module. The logic delay was relatively low for these modules, but the net delay grew larger for more multipliers. Notably, for 2 dual multipliers, the Xilinx software had trouble implementing the design without high net delays. As such, the frequency of that implementation is actually lower than that of using 3 dual multipliers.

Compared to standard ECC modules, the above resource usage appears fairly large and the protocol takes many cycles. However, this protocol is based on a very large prime field. All operations are in $\mathbb{F}_{p^2}$. Further, we also particularly geared our implementation for a fast performance. The quantum-resistance of the scheme make it much more valuable as a post-quantum cryptographic element. Lastly, our time results include the total time of both parties computations. Normally, the metric for ECDH is the rate at which scalar point multiplications occur, which is a fourth of the entire ECDH key exchange. SIDH first round computations are slightly higher than the second round, so we opted to use the total protocol time as our metric.

We also implemented the above design as a multi-core implementation. A single interleaved dual multiplier featured the best performance per area, so we replicated that core and used a FIFO buffer to issue and read results from the SIDH cores. This method is virtually identical to our use of replicated multipliers in a multiplier unit. To start the protocol with standardized parameters, Alice and Bob's private keys are inserted, which only required 8 cycles. Reading the j-invariants required 16 more cycles, so the overhead to start and stop the protocol was insignificant.

With multiple cores, we were able to improve the throughput of SIDH protocols by a factor of 11 over the single-core results. As our results show, the frequency of the device appeared to slowly drop as more cores were added. The bottleneck of resources were slices, of which 88% of slices were occupied for our device with 18 cores.

B. Comparison to Previous Works

Our protocols and formulas resemble that of [7]. We did improve the double point multiplication to have two fewer multiplications per step with an inversion at the beginning. Other than that, the main difference lies in the fact that we implemented on reconfigurable hardware and the other work is implemented in C for a computer. [9] contains a similar implementation of the protocol, but for different devices, notably on an Haswell PC. We compare our implementation to these previous software works in Table VII.

As we compare to other works, it should be noted that [6] served as an introduction to isogeny-based cryptography and does not feature many of the optimizations that the other works include. Otherwise, we note that our implementation does feature a smaller prime, but the difference in performance between a 511-bit prime and a 521-bit is expected to be small. The security of isogeny-based cryptography is based on the minimum of ℓ^a and ℓ^b. The larger prime utilizes the same

TABLE VII
COMPARISON TO SOFTWARE IMPLEMENTATIONS OF AFFINE SIDH OVER 512-BIT KEYS

Work	Platform	Smooth Isogeny Prime	Time (ms)				
			Alice Round 1	Bob Round 1	Alice Round 2	Bob Round 2	Total Time
Jao et al.[6]	2.4 GHz Opteron PC	$2^{253}3^{161}7-1$	365	318	363	314	1360
Jao et al.[7]	2.4 GHz Opteron PC	$2^{258}3^{161}186-1$	28.1	28.0	23.3	22.7	102.1
Azarderakhsh et al.[9]	4.0 GHz i7-4790k PC	$2^{258}3^{161}186-1$	-	-	-	-	54.0
This Work ($M=2\times 3$)	Virtex-7 FPGA	$2^{253}3^{161}7-1$	9.35	8.41	8.53	7.41	33.70

size of ℓ^b, so the difference in security is relatively small. The complexity of the computations are also similar, so our implementations are comparable.

When we compare our results to [7], our timings per round are approximately 3 times faster. Compared to the performance on the 4.0 GHz Haswell architecture in [9], our timings are still about 1.5 times faster for the entire protocol. This shows that our implementation, which took advantage of parallel operations in hardware, is now the fastest known SIDH protocol for the 512-bit level. There are no other known hardware implementations in the literature, but these results show that implementation of the SIDH protocol in hardware is feasible and can perform very well.

Costello *et al.* [10] recently proposed several new optimizations to the SIDH protocol that our implementation could greatly benefit from. Among these, the use of "projective" isogeny formulas allow for the computation of a large degree isogeny with only a single inversion. In our implementation, inversions were the bottleneck. Not only will this allow us to parallelize more parts of the isogeny computation, but the adaptation of these new formulas allow for a constant-time implementation and remove the need for an inversion unit. Costello also mentions various other improvements to the protocol such as finite-field optimizations and evaluating the isogeny at the points P, Q, and $P-Q$ in Kummer coordinates for the protocol. However, a direct comparison to Costello's software implementation is difficult as we targeted different prime sizes (511-bit vs. 751-bit). Costello's work did improve the protocol by a factor of 2.5 times and we believe that these optimizations could provide a similar improvement to our implementation.

VI. CONCLUSION

Overall, this paper served as the first hardware implementation of the supersingular isogeny Diffie-Hellman protocol. This is one such protocol for isogeny-based cryptography, but our approach to elaborating the large degree isogenies has merits for other protocols as well, such as a zero-knowledge identification scheme [7] or undeniable signatures [8]. We presented efficient finite-field arithmetic, scheduling methods, and design of isogeny-based cores. Hardware can take advantage of much more parallelism in $\mathbb{F}_{p^2}$ operations than standard software. Our implementation runs at 1.5 times faster than a Haswell architecture running an optimized C version of the same SIDH protocol [9]. Minimizing the numerous inversions with Costello's [10] formulas would greatly benefit a future implementation. Isogeny-based cryptography represents one possible solution to the impending quantum computing revolution because it features forward-secrecy, small keys, and resembles current protocols based on classical ECC.

ACKNOWLEDGEMENT

The authors would like to thanks the reviewers for their constructive comments. This material is based upon work supported by the National Institute of Standards and Technology (NIST) under award 60NANB16D246.

REFERENCES

[1] V. S. Miller, "Use of elliptic curves in cryptography," in *Proc. Adv. Cryptol.*, Dec. 1986, pp. 417–426.
[2] N. Koblitz, "Elliptic curve cryptosystems," *Math. Comput.*, vol. 48, no. 177, pp. 203–209, 1987.
[3] L. Chen and S. Jordan, *Report on Post-Quantum Cryptography*, 2016.
[4] A. Rostovtsev and A. Stolbunov, *Public-Key Cryptosystem Based on Isogenies*, 2006.
[5] A. Childs, D. Jao, and V. Soukharev, *Constructing Elliptic Curve Isogenies in Quantum Subexponential Time*, 2010.
[6] D. Jao and L. D. Feo, "Towards quantum-resistant cryptosystems from supersingular elliptic curve isogenies," in *Proc. Post-Quantum Cryptography–PQCrypto*, 2011, pp. 19–34.
[7] L. De Feo, D. Jao, and J. Plut, "Towards quantum-resistant cryptosystems from supersingular elliptic curve isogenies," *J. Math. Cryptol.*, vol. 8, no. 3, pp. 209–247, Sep. 2014.
[8] D. Jao and V. Soukharev, "Isogeny-based quantum-resistant undeniable signatures," in *Proc. 6th Int. Workshop Post-Quantum Cryptogr. (PQCrypto)*, Waterloo, ON, Canada, Oct. 2014, pp. 160–179.
[9] R. Azarderakhsh, D. Jao, K. Kalach, B. Koziel, and C. Leonardi, "Key compression for isogeny-based cryptosystems," in *Proc. Int. Workshop ASIA Public-Key Cryptogr.*, 2016, p. 1–10.
[10] C. Costello, P. Longa, and M. Naehrig, "Efficient algorithms for supersingular isogeny diffie-hellman," in *Advances in Cryptology*. New York, NY, USA: Springer, 2016.
[11] J. H. Silverman, *The Arithmetic of Elliptic Curves*. New York, NY, USA: Springer-Verlag, 1992, vol. 106.
[12] P. W. Shor, "Algorithms for quantum computation: Discrete logarithms and factoring," in *Proc. 35th Annu. Symp. Found. Comput. Sci.*, 1994, pp. 124–134.
[13] J.-M. Couveignes, "Hard homogeneous spaces," Cryptology ePrint Archive, Tech. Rep. 2006/291, 2006.
[14] J. Gärtner "Courbes elliptiques," *Comp. Rendus De l'Académie Des Sciences Paris Series A-B*, vol. 273, pp. A238–A241, 1971.
[15] P. L. Montgomery, "Speeding the Pollard and elliptic curve methods of factorization," *Math. Comput.*, vol. 48, no. 177, pp. 243–264, 1987.
[16] M. R. 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. Logic Appl.*, Sep. 2014, pp. 1–8.
[17] D. Harris, "A taxonomy of parallel prefix networks," in *Proc. IEEE Conf. Rec. 37th Asilomar Conf. Signals, Syst., Comput.*, vol. 2. Nov. 2003, pp. 2213–2217.
[18] P. L. Montgomery, "Modular multiplication without trial division," *Math. Comput.*, vol. 44, no. 170, pp. 519–521, Apr. 1985.
[19] M. Fürer, "Faster integer multiplication," in *Proc. 39th Annu. ACM Symp. Theory Comput.*, 2007, pp. 57–66.
[20] D. D. Chen, G. X. Yao, R. C. C. Cheung, D. Pao, and C. K. Koc, "Parameter space for the architecture of fft-based montgomery modular multiplication," *IEEE Trans. Comput.*, vol. 65, no. 1, pp. 147–160, Jan. 2016.

[21] M. M. C. McIvor and J. V. McCanny, "High-radix systolic modular multiplication on reconfigurable hardware," in *Proc. IEEE Int. Conf. Field-Program. Technol.*, Dec. 2005, pp. 13–18.
[22] T. Blum and C. Paar, "High-radix montgomery modular exponentiation on reconfigurable hardware," *IEEE Trans. Comput.*, vol. 50, no. 7, pp. 759–764, Jul. 2001.
[23] S. Gueron and V. Krasnov, "Fast prime field elliptic-curve cryptography with 256-bit primes," *J. Cryptograph. Eng.*, vol. 5, no. 2, pp. 141–151, 2014.
[24] B. S. Kaliski, "The Montgomery inverse and its applications," *IEEE Trans. Comput.*, vol. 44, no. 8, pp. 1064–1065, Aug. 1995.
[25] E. Savas and C. K. Koc, "The Montgomery modular inverse-revisited," *IEEE Trans. Comput.*, vol. 49, no. 7, pp. 763–766, Jul. 2000.
[26] J. W. Bos, "Constant time modular inversion," *J. Cryptograph. Eng.*, vol. 4, no. 4, pp. 275–281, 2014.

Brian Koziel (M'16) received a dual-degree B.Sc. and M.Sc. degree in computer engineering from Rochester Institute of Technology (RIT), Rochester, NY, USA, in May, 2016. He is currently with Embedded Processing group at Texas Instruments. His current research interests include efficient software and hardware implementations of elliptic curve cryptography and post-quantum cryptography. At RIT, he was a recipient of the prestigious Outstanding Undergraduate Scholar award.

Reza Azarderakhsh received the Ph.D. degree in electrical and computer engineering from Western University, London, ON, Canada. After that he was a Postdoctoral Research Fellow at Center for Applied Cryptographic Research and the Department of Combinatorics and Optimization, University of Waterloo, Canada, while he was the recepient of NSERC PDF award. After that he was with the Department of Computer Engineering at Rochester Institute of Technology, Rochester, NY, USA. Currently, he is an Assistant Professor at the Department of Computer, Electrical Engineering and Computer Science and I-SENSE Fellow at Florida Atlantic University, Boca Raton, FL, USA. His current research interests include finite field arithmetic and its application, elliptic curve cryptography, pairing based cryptography, and post-quantum cryptography. He is serving as an Associate Editor of IEEE TRANSACTIONS ON CIRCUITS AND SYTEMS—PART I (TCAS-I).

Mehran Mozaffari Kermani received the B.Sc. degree in electrical and computer engineering from the University of Tehran, Tehran, Iran, in 2005, and the M.E.Sc. and Ph.D. degrees from the Department of Electrical and Computer Engineering, University of Western Ontario, London, Canada, in 2007 and 2011, respectively. He was a recipient of the prestigious Natural Sciences and Engineering Research Council of Canada (NSERC) Post-Doctoral Research Fellowship. Currently, he is with the Department of Electrical and Microelectronic Engineering, Rochester Institute of Technology, Rochester, NY, USA. His current research interests include emerging security measures for embedded systems, fault diagnosis in cryptographic hardware, and low-power secure and efficient FPGA and ASIC designs. He served as the guest editor of the IEEE TRANSACTIONS ON EMERGING TOPICS IN COMPUTING special issue on Emerging Security Trends for Deeply-Embedded Computing Systems (2014–2015).

David Jao received the Ph.D degree in mathematics from Harvard University, Cambridge, MA, USA, in 2003. From 2003 to 2006, he worked in the Cryptography and Anti-Piracy Group at Microsoft Research, contributing cryptographic software modules for several Microsoft products. He is currently an associate professor in the Mathematics Faculty at the University of Waterloo, Canada, and the director of the Centre for Applied Cryptographic Research. His research interests include elliptic curve cryptography, protocol design, and implementation, and post-quantum cryptography.