Fast Hardware Architectures for Supersingular Isogeny Diffie-Hellman Key Exchange on FPGA

Brian Koziel[1], Reza Azarderakhsh[2], and Mehran Mozaffari-Kermani[3]

[1]Texas Instruments, kozielbrian@gmail.com.
[2]CEECS Dept and I-SENSE FAU, razarderakhsh@fau.edu.
[3]EME Dept, RIT, mmkeme@rit.edu.

Abstract. In this paper, we present a constant-time hardware implementation that achieves new speed records for the supersingular isogeny Diffie-Hellman (SIDH), even when compared to highly optimized Haswell computer architectures. We employ inversion-free projective isogeny formulas presented by Costello et al. at CRYPTO 2016 on an FPGA. Modern FPGA's can take advantage of heavily parallelized arithmetic in $\mathbb{F}_{p^2}$, which lies at the foundation of supersingular isogeny arithmetic. Further, by utilizing many arithmetic units, we parallelize isogeny evaluations to accelerate the computations of large-degree isogenies by approximately 57%. On a constant-time implementation of 124-bit quantum security SIDH on a Virtex-7, we generate ephemeral public keys in 10.6 and 11.6 ms and generate the shared secret key in 9.5 and 10.8 ms for Alice and Bob, respectively. This improves upon the previous best time in the literature for 768-bit implementations by a factor of 1.48. Our 83-bit quantum security implementation improves upon the only other implementation in the literature by a speedup of 1.74 featuring fewer resources and constant-time.
Key Words: Post-quantum cryptography, elliptic curve cryptography, isogeny-based cryptography, field programmable gate array

1 Introduction

Post-quantum cryptography (PQC) has been gaining a large amount of interest in the wake of NIST's announcement to standardize post-quantum cryptosystems for use by the US government [1]. Fears of the emergence of a quantum computer that could break today's current cryptosystems and expose a wealth of private information have been increasing the demand for systems to be quantum-safe. Notably, Shor's algorithm [2] could be used in conjunction with a quantum computer to quickly break elliptic curve cryptography (ECC) and RSA. Fortunately, such computers do not currently exist, but it is unclear how long this will last. As such, there is a need to consider viable alternatives to today's popular cryptosystems before the next major quantum computing breakthrough. Similar to ECC, isogeny-based cryptography also uses points on an elliptic curve to provide security. However, as opposed to security based on the difficulty to factor large point multiplications (which is the case for ECC), isogeny-based cryptography has security based on the difficulty to compute isogenies between supersingular elliptic curves. Currently, this is considered difficult even for quantum computers. An isogeny can be thought of as a unique algebraic map from one elliptic curve to another elliptic curve that satisfies group homomorphism. With the emergence of the supersingular isogeny Diffie-Hellman protocol from Jao and De Feo [3] in 2011, numerous aspects of the protocol have also been studied. Most recently, Costello, Longa, and Naehrig [4] have proposed projective isogeny formulas, which effectively eliminate the numerous inversions in the SIDH protocol and allow for a constant-time implementation. This is naturally immune to most types of simple power analysis and timing analysis. Although the SIDH protocol has been slower than other quantum-resistant schemes, it does feature smaller keys, smaller signatures, and forward secrecy, making it a viable candidate in NIST's PQC standardization workshop. In this paper, we provide the first implementation of the projective isogeny formulas presented in [4] on reconfigurable hardware. This constant-time implementation features 83-bit and 124-bit quantum security. Field programmable gate arrays (FPGA) can take advantage of a large amount of parallelism in basic arithmetic in the extension field $\mathbb{F}_{p^2}$ as well as the computation of large-degree isogenies. Aside from presenting a new speed record for SIDH, the goal of this paper is to show that hardware architectures can take advantage of the large amount of parallelism in SIDH and make it more viable in NIST's PQC workshop. The main contributions of this paper can be summarized as follows: (i) First constant-time SIDH implementation on reconfigurable hardware, 83-bit and 124-bit quantum security levels, utilizing projective isogeny formulas featured in [4], (ii) This SIDH implementation is approximately 50% faster than any other implementation in the literature. (iii) New approach to parallelizing isogeny evaluations to speed-up large-degree isogeny computations by over a factor of 1.5.

2 Preliminaries

Here, we briefly discuss the basis for isogeny-based cryptography. The isogeny-based Diffie-Hellman key exchange was first published by Rostovtsev and Stolbunov in [5]. This was originally defined over ordinary elliptic curves and was thought to feature quantum resistance. However, Childs, Jao, and Stolbunov [6] discovered a quantum algorithm to compute isogenies between ordinary curves in subexponential time. Later, David Jao, Luca De Feo, and Jerome Plut adapted the isogeny-based key exchange to be over supersingular elliptic curves in [3] and [7], which features no

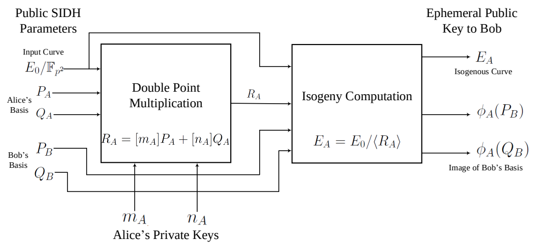

Fig. 1. Alice's first round computations for the SIDH protocol

known quantum attack. As we review elliptic curve and isogeny theory, we point the reader to [8] for a much more in-depth explanation of elliptic curve theory.

SIDH Protocol: In the SIDH scheme, Alice and Bob decide on 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 further decide on a base supersingular elliptic curve $E_0(\mathbb{F}_q)$ where $q = p^2$. Over this starting supersingular curve E_0, Alice and Bob pick the bases $\{P_A, Q_A\}$ and $\{P_B, Q_B\}$ which generate the torsion groups $E_0[\ell_A^{e_A}]$ and $E_0[\ell_B^{e_B}]$, respectively, such that $\langle P_A, Q_A \rangle = E_0[\ell_A^{e_A}]$ and $\langle P_B, Q_B \rangle = E_0[\ell_B^{e_B}]$. The SIDH protocol proceeds as follows. Alice and Bob each perform a double-point multiplication with two selected private keys that span $\mathbb{Z}/\ell^a\mathbb{Z}$ and $\mathbb{Z}/\ell^b\mathbb{Z}$, respectively. This generates a secret kernel point on each side that is used to efficiently perform a large-degree isogeny. In the first round, Alice calculates $\phi_A : E \rightarrow E_A/\langle m_A P_A + n_A P_A \rangle$ and Bob calculates $\phi_B : E \rightarrow E_B/\langle m_B P_B + n_B P_B \rangle$, where m and n are the party's secret keys. For the first round, the opposite party's basis points are pushed through the isogeny. At the end of the first round, Alice and Bob each exchange their new supersingular elliptic curve and the basis points of the opposite party on that new curve. With the exchanged information, Alice computes $\phi_{BA} : E_B \rightarrow E_{BA}/\langle m_A \phi_B(P_A) + n_A \phi_B(P_A) \rangle$ and Bob computes $\phi_{AB} : E_A \rightarrow E_{AB}/\langle m_B \phi_A(P_B) + n_B \phi_A(P_B) \rangle$. The two now share isomorphic curves with a common j-invariant that can be used as a shared secret. We illustrate the computations necessary for the first round from the perspective of Alice in Figure 1. A round can essentially be broken down into a double point multiplication and a large-degree isogeny computation.

Optimizations to the SIDH Protocol: The supersingular isogeny Diffie-Hellman protocol was first proposed by David Jao and Luca De Feo in [3] in 2011. Since then it has been interesting to see how further papers have improved the protocol. The two main papers that have improved the protocol are [7] by De Feo, Jao, and Plut and [4] by Costello, Longa, and Naehrig. Here, we highlight the main protocol optimizations that we adapt. As introduced in [7], we utilize points on Montgomery curves [9] and optimize arithmetic around them. We define a Montgomery curve, E, as the set of all points (x, y) that satisfy $E_{(A,B)}: \; By^2 = x^3 + Ax^2 + x$ and a point at infinity. When the value $A_{24} = (A + 2)/4$ is known, these curves feature extremely fast point arithmetic along their Kummer line, $(x, y) \rightarrow (X : Z)$, where $x = X/Z$. Isogenies still work for this representation because P and $-P$ generate the same set subgroup of points. This reduces the total number of computations as the y-coordinate does not need to be updated for point arithmetic or when the point is pushed to a new curve by evaluating an isogeny. Projective isogeny formulas over Montgomery curves were introduced in [4]. These formulas projectivize the curve equation with a numerator and denominator, similar to projective point arithmetic. We define a projective Montgomery curve, $\hat{E}$, as the set of all points (x, y) that satisfy $\hat{E}_{(\hat{A},\hat{B},\hat{C})}: \; \hat{B}y^2 = \hat{C}x^3 + \hat{A}x^2 + \hat{C}x$ and a point at infinity. In this representation, the corresponding affine Montgomery curve would have coefficients $A = \hat{A}/\hat{C}$ and $B = \hat{B}/\hat{C}$. To perform a double point multiplication, we specify that one of Alice and Bob's secret keys is 1, as introduced in [7]. Costello et al. [4] also greatly simplified the starting parameters for SIDH by proposing to use the starting Montgomery curve $E_0/\mathbb{F}_{p^2}: \; y^2 = x^3 + x$. By specifying points in the base field and trace-zero torsion subgroup, the first round of the SIDH protocol can be performed as a Montgomery [9] ladder followed by a point addition, with all operations in $\mathbb{F}_p$. The second round of the protocol involves a double-point multiplication with elements in $\mathbb{F}_{p^2}$. For this, we utilize the 3-point ladder proposed in [7] that computes $P + mQ$ in $\log_2(m)$ steps. Each step requires 2 point additions and 1 point doubling. We closely follow the projective isogeny formulas presented in [4] for isogenies of degree $\ell_{Alice} = 4$ and $\ell_{Bob} = 3$. For the first round, we push the Kummer coordinates of the other party's basis P, Q, and $Q - P$ through the large-degree isogeny rather than the projective version of P and Q to remove a point subtraction before the 3-point ladder. As proposed by [10], large-degree isogenies can be decomposed into a chain of smaller degree isogeny computations and computed iteratively. From a base curve E_0 and point R of order ℓ^e, we compute a chain of ℓ-degree isogenies: $E_{i+1} = E_i/\langle \ell^{e-i-1} R_i \rangle$, $\phi_i : E_i \rightarrow E_{i+1}$, $R_{i+1} = \phi_i(R_i)$. This problem can be visualized as an acylic graph, which is shown in Figure 3 in Section 4.3. In Figure 4 In Section 4.3, we further illustrate a sample strategy to compute each of the

Table 1. SIDH Public Parameters

Curve: $E_0/\mathbb{F}_{p^2}: \ y^2 = x^3 + x$			
Prime	Classical/Quantum Security (bits)	P_A	P_B
$p_{503} = 2^{250}3^{159} - 1$	125/83	$[3^{159}](14, \sqrt{14^3 + 14})$	$[2^{250}](6, \sqrt{6^3 + 6})$
$p_{751} = 2^{372}3^{239} - 1$	186/124	$[3^{239}](11, \sqrt{11^3 + 11})$	$[2^{372}](6, \sqrt{6^3 + 6})$

ℓ-degree isogenies at the peak of the triangle by saving points at certain nodes to a point queue.

SIDH Protocol Parameters: To make our implementation comparable to the first hardware implementation of affine SIDH in [11] and the first software implementation of projective SIDH in [4], we chose to test our architecture over the primes $p_{503} = 2^{250}3^{159} - 1$ and $p_{751} = 2^{372}3^{239} - 1$. These primes offer 83 and 124 bits of quantum security, respectively.

Similar to the strategy proposed by Costello et al. [4], we begin with a simple Montgomery curve, technically also a short Weierstrass curve: $E_0/\mathbb{F}_{p^2}: \ y^2 = x^3 + x$. To determine generator points for the torsion subgroups $\ell_A^{e_A}$ and $\ell_B^{e_B}$, we again turn to Costello et al.'s method [4]. For the $\ell_A^{e_A}$-torsion points P_A and Q_A, we find a point $P_A \in E_0(\mathbb{F}_p)[\ell_A^{e_A}]$ as $[f\ell_B^{e_B}](z, \sqrt{z^3 + z})$, where z is the smallest positive integer such that $\sqrt{z^3 + z} \in \mathbb{F}_p$ and P_A has order $\ell_A^{e_A}$. We apply a distortion map over E_0 to P_A to find Q_A such that it is the endomorphism $\tau: \ E_0(\mathbb{F}_{p^2}) \rightarrow E_0(\mathbb{F}_{p^2})$,$(x + 0i, y + 0i) \rightarrow (-x + 0i, 0 + iy)$. Thus, $Q_A = \tau(P_A)$. The $\ell_B^{e_B}$-torsion points are found in a similar matter. We find $P_B \in E_0(\mathbb{F}_p)[\ell_B^{e_B}]$ as $[f\ell_A^{e_A}](z, \sqrt{z^3 + z})$, where z is the smallest positive integer such that $\sqrt{z^3 + z} \in \mathbb{F}_p$ and P_B has order $\ell_B^{e_B}$. Lastly, $Q_B = \tau(P_B)$. For the selected primes, our starting parameters are given in Table 1.

3 Proposed Architectures for Isogeny Computations

In this section, we investigate the design of an SIDH core, focusing on optimizing finite-field addition and multiplication. The goal is to design a scalable architecture that features a secure and efficient implementation of SIDH. The proposed projective SIDH formulas presented in [4] make it reasonable to exclude a dedicated inversion module. Further, the simplification of the SIDH parameters allow for a reduction of the number of registers to store the SIDH parameters as well as the ability to perform Montgomery's powering ladder [9] in a base field rather than the 3-point differential Montgomery ladder over a quadratic field first proposed in [3]. In fact, the Montgomery ladder used to perform the first double point multiplication for both Alice and Bob may demonstrate a slight advantage to

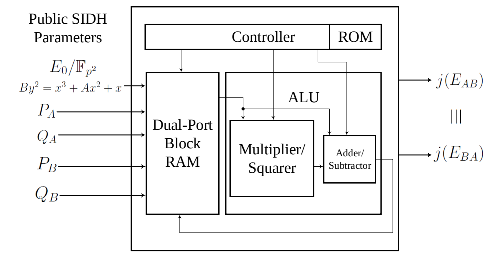

Fig. 2. Proposed High-level Architecture of an SIDH Core

implementing a more efficient squaring unit. However, this squaring unit would not see much action as it is only used in the ladder of the first round of the key exchange and inversion. A dedicated squaring unit was not implemented for this paper, but should be investigated in the future. The high level design of the isogeny core is depicted in Figure 2. This core features a single adder unit, multiplier unit with replicated multipliers, dual-port RAM file for registers, and a program ROM file for the controls. The RAM file contained 256 values in $\mathbb{F}_p$, or 256 m-bit entries. For our implementations, $m = 512$ and $m = 752$ for the choices of p_{503} and p_{751}, noted in Section 2. 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. The major constants that are initially put into the RAM file are the constants 0, 1, 2, 4^{-1}, and 6, the base Montgomery curve coefficients A, B, and A_{24}, and the basis points P_A, Q_A, $Q_A - P_A$, P_B, Q_B, $Q_B - P_B$. There are more intermediate values necessary for higher key sizes as the graph traversal of the large degree isogeny is more expansive, but 256 values is more than enough, even for 768-bit SIDH, which allows more flexibility and optimization with routines. The program ROM contains the controls for the adder, multiplier, and RAM for every cycle for various SIDH routines (listed in Section 4.4). The size of the program ROM unit depends on the number of replicated multipliers as more multipliers will allow for fewer clock cycles. A stall counter was added to the control unit to diminish the impact of stall cycles that fill the program ROM.

3.1 Finite Field Adder

Finite-field addition computes the sum $C = A + B$, where $A, B, C \in \mathbb{F}_p$. If the sum C is greater than p, then there is a reduction by performing the subtraction $C = C - p$ to have $C \in \mathbb{F}_p$. A similar situation occurs for finite-field subtraction, $C = A - B$, where $A, B, C \in \mathbb{F}_p$. An adder can be used as a subtractor if the second operands input bits are flipped. The input operands to our adder/subtractor were selected with two 3:2 multiplexers. Operand 1 could be a value from port A of the RAM, the result from the adder/subtractor, or result from the multiplier. Operand 2 could be a value from port B of the RAM, zero, or the prime. Based on the interface between the RAM unit and the adder/subtractor module, which incurs delays from the register file logic and the 3:2 multipliexer into the adder/subtractor module, we decided to split the addition/subtraction into multiple cycles by cascading multiple, smaller adder/subtractors. We tried to match the critical path delay of the adder with that of the multiplier to ensure that both modules operated efficiently. Our smaller adder/subtractor units were based around 256-bit addition and subtraction. In practice, we utilized 252-bit and 251-bit adder/subtractor units for p_{503} and one 250-bit and two 251-bit adder/subtractor units for p_{751}. Xilinx's default IP was used to create these blocks. Partial sums and operands were pipelined to achieve a high-throughput adder/subtractor. An addition or subtraction was finished in 2 cycles for p_{503} and 3 cycles for p_{751}.

3.2 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. The two known multiplier architectures targeting smooth isogeny primes are in [12] and [11]. Both utilize Montgomery [13] multiplication and reduction to efficiently perform the large modular multiplications. Montgomery multiplication performs a modular multiplication by transforming integers to m-residues, or the Montgomery domain, and performing multiplications with this representation. Montgomery multiplication converts time-consuming trial divisions to shift operations, which is simple to do in hardware. At the end of computations, the result can be converted out of the Montgomery domain with a single Montgomery multiplication. Algorithm 1 demonstrates the Montgomery reduction procedure. In [12], the authors present an efficient method for modular multiplication over smooth isogeny primes of the form $p = 2 \cdot 2^a 3^b - 1$ by using the representation $A = a_1 2^a 3^b + a_2 2^{a/2} 3^{b/2} + a_3$, determining smaller partial products, and then performing an efficient division with some precalculations. The results appear interesting for a software implementation, achieving a 62% speed-up in modular reduction and 43% speed-up in modular multiplication. However, the hardware architecture for the multiplication algorithm appears to suffer. For a 768-bit prime, the Virtex-6 architecture required 11,924 registers and 12,790

Algorithm 1 High-Radix Montgomery Multiplication Algorithm [15]

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

lookup-tables, while operating at only 31 MHz and taking 236 cycles per modular multiplication. The other modular multiplier in [11] featured a systolic Montgomery multiplier based on [14]. Using a 2^{16}radix for a 1024-bit modular multiplication, the basic multiplier proposed in [14], operates at a clock frequency of 101.86 MHz, requires 5,709 slices and 131 DSP48's, and performs a modular multiplication in 199 clock cycles, all on a Virtex2 Pro. Further, this multiplier can perform 2 multiplications simultaneously. This already runs rings over the multiplier proposed in [12]. The target of this implementation is a high-throughput and fast multiplier. The implementation in [11] improved this systolic multiplier to allow higher throughput by featuring interleaving multiplications approximately 2/3 of the multiplication latency as well as one fewer stage in the systolic array. Thus, this allows for a 99 cycle multiplication and 68 cycle interleaving for a 512-bit multiplication.

Ultimately, we chose to go with the same interleaved systolic Montgomery multiplier proposed in [11]. This multiplier utilizes the high-radix Montgomery multiplication procedure, which is shown in Algorithm 1. As was originally proposed in [14], we can use a systolic architecture to perform the iterative computations in Algorithm 1. Consider a systolic array of $m + 2$ processing elements that each compute $S_{i+1} = (S_i + q_i\overline{m_j})/2^k + a_i b_j$, where j is the number of the processing element in the array. We can effectively setup a "pump" that pushes a_i and $q_i = (S_i) \bmod 2^k$ from processing element j to processing element $j + 1$. Thus, to perform the high-radix Montgomery multiplication, we start by pushing a 0 through the systolic arrays so that $q_0 = 0$. Following that, we push a_i through the processing elements, such that it performs $a_i b_j$ and adds that result to $(S_i + q_i\overline{m_j})/2^k$ in each processing element. Essentially, each processing element performs $q_i\overline{m_j}$ and $a_i b_j$ in parallel, and then performs a 4-operand addition with $q_i\overline{m_j}$, a_ib_j, S_i, and a carry. After $m+3$ cycles, the least significant k-bit word of the result is ready. The last word is ready after $3m+7$ cycles. Interestingly, for a given multiplication, only half of the processing elements are used on a specific cycle. Thus, we can use a single multiplier architecture to handle two multiplications simultaneously, at the cost of multiplexers on the input and output that cycle between an even or odd multiplication. The design in [11] features an interleaved version of [14]. As one multiplication is finishing up, the earlier processing elements are no longer in use. Thus, we can interleave multiplications every $2m+3$ cycles by gradually filling in these processing elements whose previous task just finished. As is also noted in [11], $\bar{M} = M$ since $M' = 1$ for SIDH primes of the form $2^{e_a}\ell_b^{e_b}f-1$, which is applicable to both of our test primes. This simplification reduces the total size of the systolic array by one processing element and reduces the latency by 3 cycles. Since a DSP48 block effectively computes up to an 18x18 multiplication, we decided to make our Montgomery multiplier with radix 2^{16}. Using this, we calculated the latency of multiplication and interleaving. For p_{503}, a multiplication required 99 cycles and multiplications could be interleaved 68 cycles into a multiplication. For p_{751}, a multiplication required 144 cycles and multiplications could be interleaved every 98 cycles. We also implemented a larger multiplier unit that featured replicated multiplier units. Multiplications are the main bottleneck in the finite-field operations given by the smooth isogeny primes. As such, we implemented a first-in-first-out circular buffer. Multiplication instructions are issued cyclically starting from multiplier 0 to multiplier $2n-1$ for n dual multipliers. This comes at the cost of a large multiplexer of size $2n : \log_2 2n$ for the output.

4 Parallelizing SIDH

This section details our attempt to maximize the throughput of our architecture throughout the SIDH protocol. Since we used the same even-odd multiplier as [11], we scheduled our instructions with a greedy algorithm that incurs stalls if a multiplication is not on the right even-odd cycle.

4.1 Scheduling

Our program ROM features many different routines such as a small scalar point multiplication or isogeny evaluation of degree 4. Each instruction is 26 bits long and proceeds as follows: bits 0-7 determine the address for port A of the RAM, bits 8-15 determine the address for port B of the RAM, bit 16 signals a write to port A, bits 17-19 indicate the adder operation, bit 20 indicates a read from both RAM ports, bits 21-22 indicate multiplier operation, bits 23 and 24 indicate if operand A and B, respectively, should point to the address of the final point in the isogeny point queue, and bit 25 indicates if the previous bits are a stall counter. We utilized a greedy algorithm to assemble our own assembly code that consists of addition, subtraction, multiplication, and squaring in $\mathbb{F}_p$ or $\mathbb{F}_{p^2}$ to 26-bit aligned instructions. It is assumed that every routine starts on an even cycle. Since a store is the final instruction in a routine, we also reset the multiplier even_odd at the last cycle of a routine so that the next routine starts on an even cycle from the multiplier's perspective. Every instruction was compiled in order, so if an instruction needed the result from a previous instruction, then pipeline stalls were incurred until that value was ready. The greedy algorithm to schedule each operation would check that the RAM, addition, or multiplier unit were available for the particular instruction. For instance, an addition in $\mathbb{F}_p$ could be scheduled if the memory unit at time t, addition unit at time $t + \text{mem\_latency}$,addition unit at time $t + \text{mem\_latency} + \text{add\_latency}$, and memory unit at time $t + \text{mem\_latency} + 2 * \text{add\_latency}$ were each available, as the entire operation must go through that exact sequence. Based on the specifications of the dual-port RAM unit, memory load operations require 2 cycles and memory write operations require 1 cycle. The add latency is 2 cycles for p_{503} and 3 cycles for p_{751}. The multiplication and multiplication interleave delays are 99 cycles and 68 cycles for p_{503}, respectively, and 144 cycles and 98 cycles for p_{751}, respectively. If a multiplication occurred on the wrong even_odd cycle, we reschedule the operations by pushing the multiplication a single cycle forward, and pushing any previous instructions that are not a load or multiply by 1 or more cycles, according to the algorithm provided by [11].

4.2 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 $x^2 + 1$, applicable to SIDH primes of the form $2^{e_a}\ell_b^{e_b}f - 1$. 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. Let $i = \sqrt{-1}$ be the most significant $\mathbb{F}_p$ in $\mathbb{F}_{p^2}$. Let $A, B \in \mathbb{F}_{p^2}$ and $a_0, b_0, a_1, b_1 \in \mathbb{F}_p$, where $A = a_0 + ia_1$ and $B = b_0 + ib_1$ Then we define the extension field arithmetic $\mathbb{F}_{p^2}$ in terms of $\mathbb{F}_p$ as: $A + B = a_0 + b_0 + i(a_1 + b_1)$, $A - B = a_0 - b_0 + i(a_1 - b_1)$, $A \times B = (a_0 + a_1)(b_0 - b_1) + a_0b_1 - a_1b_0 + i(a_0b_1 + a_1b_0)$, $A^2 = (a_0 + a_1)(a_0 - a_1) + i2a_0a_1$, $A^{-1} = (a_0 - ia_1)(a_0^2 + a_1^2)^{-1}$. Based on these representations, parallel calculations could easily be performed for a single operation in $\mathbb{F}_{p^2}$. For instance, three separate multiplications in $\mathbb{F}_p$ could be carried out simultaneously for the calculation of a multiplication in $\mathbb{F}_{p^2}$. With other non-dependent instructions in the scheduling, many multipliers can be used in parallel. Unfortunately, an inversion in $\mathbb{F}_p$ was difficult to parallelize, and suffered as a result. We utilized a k-ary method with $k = 4$ to perform Fermat's little theorem for inversion. We were able to parallelize the generation of the windows $1, 2, 3, \cdots, 2^k - 1$, but after that, the inversion was done serially. k squarings were done in serial followed by a multiplication. The inversion added many lines to the program ROM, and was difficult to parallelize, showing that there may still be some merit to having a dedicated inversion unit.

4.3 Scheduling Isogeny Computations and Evaluations

Large-degree isogeny calculations were performed by traversing a large directed acyclic graph in the shape of a triangle to the leaves, where a smaller degree isogeny was computed. This is illustrated in Figure 3. From a node in the graph, a point multiplication by ℓ moves to the left and an evaluation of a ℓ-isogeny moves to the right. Based on the cost of an isogeny evaluation and point multiplication, there exists an optimal strategy that traverses the graph to the leave with the minimal computational cost. Notably, an optimal strategy is composed of two optimal sub-strategies. Thus, by recursively optimizing sub-strategies, the overall strategy is determined. We calculated the optimal strategy with the Magma code provided by [4]. In this code, we used the relative ratio of a single point multiplication by ℓ and half of a single ℓ-isogeny evaluation to create an optimal strategy that emphasized point evaluations. In our implementation, we utilize a recursive function to compute the large-degree isogeny with an optimal strategy. We utilized a look-up-table in ROM to hold the optimal strategy and efficiently traverse the acyclic graph. A queue was used to keep track of multiple points on the current curve. As isogenies were computed, these points were pushed through the isogenous mapping to the corresponding point on the new curve. As a method for further parallelization, we noticed that isogeny evaluations have typically been carried out iteratively. Thus, we attempted to parallelize the evaluations by adding additional isogeny evaluation functions for when there were 2 points, 3 points, $\cdots$, up to 9 points in the queue. Specifically, there were no data dependencies between isogeny evaluations of any of the points in the queue. Thus, our assembly code reordered many instructions in a row that had no limiting data dependency, similar to unrolling the loop in a software implementation. We unrolled a max of 6 iterations of the loop at a time to ensure that enough hardware registers were available to hold intermediate values. We found this greatly increase the speed

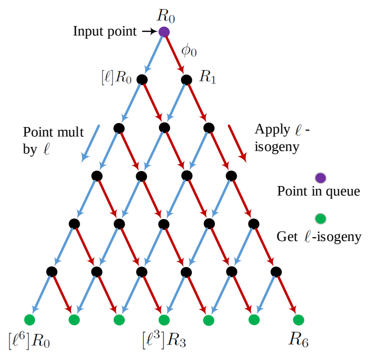

Fig. 3. Acyclic graph structure for performing isogeny computation of ℓ^6.

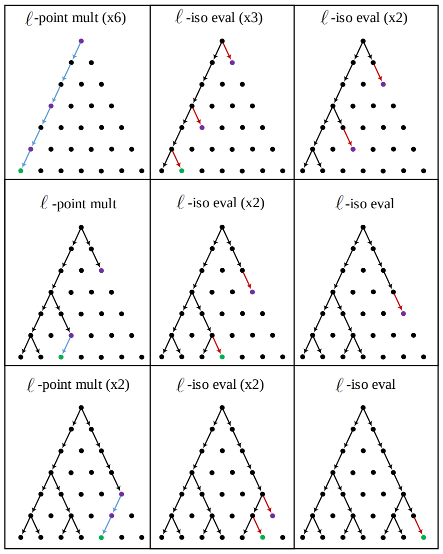

Fig. 4. Performing an isogeny computation of ℓ^6 with a sample strategy and parallel isogeny evaluations.

of our isogeny computations. For instance, this method reduced the total time to compute all 4 large-degree isogenies from 7.15 million cycles to 4.54 million cycles for p_{751} and 4 replicated multipliers. We provide an example of isogeny evaluation parallelization in Figure 3. Consider computing an ℓ^6-degree isogeny. Following an ℓ-degree isogeny computation, each point in the point queue is pushed through the isogenous mapping. We do this in parallel to utilize our hardware results more effectively. The parallelization is much more evident in larger degree isogeny computations. For instance, there is an average of 4.2 points in Alice's queue after each isogeny computation in our p_{751} implementation. Parallelization of isogeny evaluation could also be applicable to multi-core CPU implementations of SIDH. Our particular hardware implementation was able to parallelize the isogeny evaluations because of the number of multipliers that were readily available. In a software implementation, the multiplication and addition arithmetic might be complex and consume most of the arithmetic units. However, because there is no data dependency, the task to push all of the points through the isogeny could be divided among different cores. For instance, consider pushing 8 points through an isogenous mapping in a quad-core CPU. Each core could evaluate an isogeny for 2 points in the queue to better take advantage of resources. Of course, there would be overhead in distributing the task, but a nice speedup could be achieved when there are several points in the queue.

4.4 Total Cost of Routines

Here, we break up the relative costs of routines within our implementation of the SIDH protocol. Table 2 contains the results of various routines, which closely follows the formulas provided in [4]. $\tilde{A}$, $\tilde{S}$, and $\tilde{M}$ refer to addition, squaring, and multiplication, respectively, in $\mathbb{F}_{p^2}$. Routines with a note of ($\mathbb{F}_p$) count operations in $\mathbb{F}_p$.

– *Mont. Ladder Step ($\mathbb{F}_p$):* We perform a single step of the Montgomery ladder [9] in $\mathbb{F}_p$, which requires 1 point addition and 1 point doubling.
– *3-point Ladder Step*: We perform a single step of the 3-point Montgomery ladder [7], which requires 2 point additions and 1 point doubling.
– *Mont Quadruple/Triple*: We perform a scalar point multiplication by 4 in the case of quadrupling and scalar point multiplication by 3 in the case of tripling.
– *Get ℓ Isog*: We compute an isogeny of degree ℓ. Alice operates over isogenies of degree 4 and Bob operates over isogenies of degree 3.
– *Eval ℓ Isog (x times)*: We push points through the isogenous mapping from their old curve to their new curve. This code is unrolled x times from 1 point to 9 points.
– $\mathbb{F}_{p^2}$ *inversion ($\mathbb{F}_p$)*: We compute the inverse of an element using Fermat's little theorem.

5 FPGA Implementations Results and Discussion

The SIDH core was compiled with Xilinx Vivado 2015.4 to a Xilinx Virtex-7 xc7vx690tffg1157-3 board. All results were obtained after place-and-route. The area and timing results of our SIDH core are shown in Table

Table 2. Cost of major routines for p_{751}

Routine	Ops in $\mathbb{F}_{p^2}$			#ops in	Latency for n mults (cc)				
	($\tilde{A}$)	($\tilde{S}$)	($\tilde{M}$)	protocol	2	4	6	8	10
Mont. Ladder Step ($\mathbb{F}_p$)	9	4	5	751	619	495	495	495	495
3-point Ladder Step	14	6	9	751	2181	1329	1120	972	908
Mont Quadruple	11	4	8	1276	1874	1306	1151	1151	1151
Mont Triple	15	5	8	1622	1954	1289	1124	1145	1145
Get 4 Isog	7	5	0	370	586	386	367	363	363
Eval 4 Isog	6	1	9	14	1655	1461	1225	1221	1147
Eval 4 Isog (3 times)	18	3	27	255	4537	2855	2104	1917	1642
Eval 4 Isog (5 times)	30	5	45	98	7427	4212	3036	2489	2215
Eval 4 Isog (7 times)	42	7	63	16	10543	6293	4674	4168	3716
Get 3 Isog	8	3	3	478	833	496	471	434	434
Eval 3 Isog	2	2	6	12	1252	1001	812	810	734
Eval 3 Isog (3 times)	6	6	18	309	3442	2026	1461	1306	1103
Eval 3 Isog (5 times)	10	10	30	112	5638	3123	2229	1776	1535
Eval 3 Isog (7 times)	14	14	42	72	7972	4411	3154	2667	2389
$\mathbb{F}_{p^2}$ Inversion ($\mathbb{F}_p$)	2	757	196	4	142307	142059	142059	141973	141973

Table 3. Implementation results of SIDH architectures on a Xilinx Virtex-7 FPGA

Type	# Mults	Area					Time			SIDH/s
		# FFs	# LUTs	# Slices	# DSPs	# BRAMs	Freq (MHz)	Latency ($cc \times 10^6$)	Total time (ms)	
p_{503}	6	26,659	19,882	8,918	192	40	181.4	3.80	20.9	47.8
	8	32,541	23,404	11,205	256	37.5	186.8	3.63	19.4	51.5
	10	39,446	28,520	12,962	320	34.5	175.9	3.48	19.8	50.5
p_{751}	6	36,728	25,975	11,801	282	47	177.3	8.21	46.3	21.6
	8	46,857	32,726	15,224	376	45.5	182.1	7.74	42.5	23.5
	10	56,979	40,327	18,094	470	44	172.6	7.41	42.9	23.3

3. We focused on 3-5 replicated multipliers in our design to ensure the parallelism in SIDH could be taken advantage of. The implementation was optimized to reduce the net delay to maximize the clock frequency. These are constant-time results. Our SIDH parameters are discussed in Section 2. As these results show, the architectures continue to reduce the total number of clock cycles for SIDH, even at 10 multipliers. This is primarily a result of the parallelism achieved in isogeny evaluation and the 3-point ladder. Furthermore, the architecture appears fairly scalable. Moving from a 503-bit prime to a 751-bit prime did not have much impact on the maximum frequency of the device and added a small proportion of additional resources. For 5 multipliers under the 751-bit prime, approximately 16.71% of the Virtex-7's slices were occupied. Many more

Table 4. Hardware comparison of SIDH architectures on a Virtex-7 with 3 replicated multipliers

Work		Area					Time		
	Prime (bits)	# FFs	# LUTs	# Slices	# DSPs	# BRAMs	Freq (MHz)	Latency ($cc \times 10^6$)	Total time (ms)
Koziel et al. [11]	511	30,031	24,499	10,298	192	27	177	5.967	33.7
This Work	503	26,659	19,882	8,918	192	40	181.4	3.80	20.9

Table 5. Comparison to the software implementations of SIDH over 512-bit keys

Work	Platform	Smooth Isogeny	Time (ms)				
			Alice	Bob	Alice	Bob	Total
		Prime	Rnd 1	Rnd 1	Rnd 2	Rnd 2	Time
Jao et al.[3]	2.4 GHz Opt.	$2^{253}3^{161}7-1$	365	318	363	314	1360
Jao et al.[7]	2.4 GHz Opt.	$2^{258}3^{161}186-1$	28.1	28.0	23.3	22.7	102.1
Azarderakhsh et al.[16]	4.0 GHz i7	$2^{258}3^{161}186-1$	-	-	-	-	54.0
Koziel et al. [11]	Virtex-7	$2^{253}3^{161}7-1$	9.35	8.41	8.53	7.41	33.70
This Work ($M = 2 \times 4$)	Virtex-7	$2^{250}3^{159}-1$	4.83	5.25	4.41	4.93	19.42

resources could be used to attempt more parallelization, but the clock frequency may suffer as a result, which is evident in our implementations of 5 replicated dual-multipliers.

Comparison to Previous Works: The only other hardware implementation is [11], which served as an introductory look into the SIDH protocol on hardware. We provide a rough comparison for 3 replicated multipliers at the 512-bit security level. Our architecture performs an entire SIDH key-exchange approximately 1.61 times faster than that of [11]. This is most likely a result of using the new projective isogeny formulas as well as parallelism in the isogeny evaluations. In terms of area, our architecture requires about 15% less flip-flops, look-up-tables, and slices, but requires about 1.5 times as many 36k block RAM modules.

Overall, this is to be expected as our architecture does not include an inversion unit. In [11], the $\mathbb{F}_{p^2}$ inversion required about 1886 cycles for each isogeny computation. Our isogeny computations did not require this expensive operation and we were able to parallelize the projective isogeny evaluations that are more complex than their affine isogeny couterparts. The difference in prime sizes does not make much of a difference for area because both are based on a radix 2^{16} multiplier. Most importantly, our implementation is constant-time and the previous one is not, which provides security against simple power analysis and timing attacks. Next, we look at the overall speed of this implementation compared to the state-

Table 6. Comparison to software implementations of SIDH over 768-bit keys

Work	Platform	Smooth Isogeny	Time (ms)				
			Alice	Bob	Alice	Bob	Total
		Prime	Rnd 1	Rnd 1	Rnd 2	Rnd 2	Time
Jao et al.[7]	2.4 GHz Opt.	$2^{258}3^{161}186-1$	65.7	54.3	65.6	53.7	239.3
Azarderakhsh et al.[16]	4.0 GHz i7	$2^{386}3^{242}2-1$	-	-	-	-	133.7
Costello et al. [4]	3.4 GHz i7	$2^{372}3^{239}-1$	15.0	17.3	13.8	16.8	62.9
This Work ($M=2\times 4$)	Virtex-7	$2^{372}3^{239}-1$	10.6	11.6	9.5	10.8	42.5

of-the-art, shown in Tables 5 and 6, which demonstrate the fastest SIDH implementations over approximately 512 and 768-bit keys. These feature approximately 85 and 128-bits of quantum security, respectively. We compare against our implementations with 4 replicated dual-multipliers, which featured the fastest times for our results. These benchmarks have shown that the total time of the SIDH protocol has continued to drop since its inception by Jao and De Feo in [3]. Our 512-bit implementation operated approximately 74% faster than the previous best implementation in hardware in [11]. These results are approximately 48% faster than those of [4], despite the powerful nature of Haswell architectures. Smaller SIDH implementations on ARM also exist [17], but these utilize far fewer resources so it is difficult to make a fair comparison.

6 Conclusion

Overall, this paper served as the first constant-time hardware implementation of the supersingular isogeny Diffie-Hellman protocol over projective isogeny formulas. As our results show, our architecture is scalable and is even faster than the previously fastest implementations of the protocol on Haswell PC architectures. Hardware can take advantage of much more parallelism in $\mathbb{F}_{p^2}$ operations and isogeny evaluations over standard software. Our implementation runs at 48% faster than a Haswell architecture running an optimized C version of the same SIDH protocol. By removing the multitude of inversions in the protocol, this new implementation features a faster constant-time performance with less resources than the previous best hardware implementation in the literature. 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.

7 Acknowledgment

This material is based upon work supported by the NSF CNS-1464118 and NIST 60NANB16D246 grants awarded to Reza Azarderakhsh.

References

1. Chen, L., and Jordan, S.: Report on Post-Quantum Cryptography, (2016) NIST IR 8105.
2. Shor, P.W.: Algorithms for Quantum Computation: Discrete Logarithms and Factoring. In: 35th Annual Symposium on Foundations of Computer Science (FOCS 1994). 124–134 (1994)
3. Jao, D. and De Feo, L.: Towards Quantum-Resistant Cryptosystems from Supersingular Elliptic Curve Isogenies. In: Post-Quantum Cryptography–PQCrypto 2011. LNCS 19–34 (2011)
4. Costello, C., Longa, P., and Naehrig, M.: Efficient Algorithms for Supersingular Isogeny Diffie-Hellman. In: Advances in Cryptology - CRYPTO 2016 - 36th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 14-18, 2016, Proceedings, Part I. Volume 9814 of Lecture Notes in Computer Science. 572–601 (2016)
5. Rostovtsev, A., Stolbunov, A.: Public-Key Cryptosystem Based on Isogenies. IACR Cryptology ePrint Archive 2006, 145 (2006)
6. Childs, A., and Jao, D., and Soukharev, V.: Constructing Elliptic Curve Isogenies in Quantum Subexponential Time (2010)
7. De Feo, L., Jao, D., and Plut, J.: Towards Quantum-Resistant Cryptosystems from Supersingular Elliptic Curve Isogenies. Journal of Mathematical Cryptology 8(3), 209–247 (Sep. 2014)
8. Silverman, J.H.: The Arithmetic of Elliptic Curves. Volume 106 of GTM. Springer, New York (1992)
9. Montgomery, P. L.: Speeding the Pollard and Elliptic Curve Methods of Factorization. Mathematics of computation, 243–264 (1987)
10. Couveignes, J.-M.: Hard Homogeneous Spaces. Cryptology ePrint Archive, Report 2006/291 (2006)
11. Koziel, B., Azarderakhsh, R., Kermani, M.M., Jao, D.: Post-Quantum Cryptography on FPGA Based on Isogenies on Elliptic Curves. Cryptology ePrint Archive, Report 2016/672 (2016) http://eprint.iacr.org/2016/672.
12. Karmakar, A., Roy, S., Vercauteren, F., and Verbauwhede, I.: Efficient Finite Field Multiplication for Isogeny Based Post Quantum Cryptography. In: International Workshop on the Arithmetic of Finite Fields, WAIFI 2016. to appear
13. Montgomery, P. L.: Modular Multiplication without Trial Division. Mathematics of Computation 44(170), 519–521 (1985)
14. McIvor, C., McLoone, M., and McCanny, J. V.: High-Radix Systolic Modular Multiplication on Reconfigurable Hardware. In: IEEE International Conference on Field-Programmable Technology. 13–18 (Dec. 2005)
15. Orup, H.: Simplifying Quotient Determination in High-Radix Modular Multiplication. In: Proceedings of the 12th Symposium on Computer Arithmetic. ARITH '95, Washington, DC, USA, IEEE Computer Society 193–9 (1995)
16. Azarderakhsh, R., Jao, D., Kalach, K., Koziel, B., Leonardi, C.: Key Compression for Isogeny-Based Cryptosystems. In: Proceedings of the 3rd ACM International Workshop on ASIA Public-Key Cryptography. AsiaPKC '16, New York, NY, USA, ACM 1–10 (2016)
17. Koziel, B., Jalali, A., Azarderakhsh, R., Jao, D., Mozaffari-Kermani, M.: NEON-SIDH: Efficient Implementation of Supersingular Isogeny Diffie-Hellman Key Exchange Protocol on ARM. In: 15th International Conference on Cryptology and Network Security, CANS 2016