Post Quantum ECC on FPGA Platform

Debapriya Basu Roy[1] and Debdeep Mukhopadhyay[1]

Indian Institute of Technology, Kharagpur, (dbroy24,debdeep.mukhopadhyay)@gmail.com

Abstract. Post-quantum cryptography has gathered significant attention in recent times due to the NIST call for standardization of quantum resistant public key algorithms. In that context, supersingular isogeny based key exchange algorithm (SIKE) has emerged as a potential candidate to replace traditional public key algorithms like RSA and ECC. SIKE provides $\mathbf{O}(\sqrt[4]{p})$ classical security and $\mathbf{O}(\sqrt[6]{p})$ quantum security where p is the characteristic of the underlying field. Additionally, SIKE has the smallest key sizes among all the post-quantum public algorithm, making it very suitable for bandwidth constrained environment. In this paper, we present an efficient implementation of SIKE protocol for FPGA based applications. The proposed architecture provides the same latency as that of the best existing implementation of SIKE protocol while consuming 48% less DSPs and 58% less block RAM resources. Thus, our design is substantially more efficient compared to that of existing implementations of SIKE.

Keywords: SIKE, FPGA, Montgomery, BRAM, DSP

1 Introduction

In recent years, research in quantum computers has gone through a significant advancement and the possibility of having commercial quantum computers in the near future is not distant any more. IBM has recently launched a cloud based quantum computing service *IBM-Q* which allows user to avail quantum computing services. Quantum computers can solve many mathematical problems which can not be solved by the conventional computers. Unfortunately, this includes many mathematical hard problems which form the basis of several existing public key cryptographic algorithms. Therefore, all the existing public key cryptographic algorithms (RSA and ECC) will cease to be secure in a post-quantum world. Hence, there is a need for new public key algorithms which will be secure against both traditional and quantum computing. Understanding this, NIST has announced the standardization process for post-quantum secure public key algorithms in 2017 and subsequently, there have been multiple submissions of post-quantum secure public key schemes in the NIST's portal. The submitted post-quantum algorithms can be broadly classified into five different categories: lattice based algorithms, multivariate equation based algorithms, code based algorithms, hash based algorithms and supersingular isogeny based algorithm (SIKE). In this paper, we will concentrate on the efficient implementation of SIKE protocol on FPGA platform.

Supersingular isogeny based Diffie-Hellman key exchange algorithm (SIKE) is based on the isogeny property of supersingular elliptic curves. Elliptic curve is a well studied topic and key exchange algorithm based on elliptic curve scalar multiplication is already deployed in various security critical services. In that context, transition to supersingular isogeny based cryptosystem will be easier compared to other post-quantum secure public key algorithms. Additionally, among all the available post-quantum public key algorithms, SIKE requires the smallest key. To provide 124 bit security, the public key size of SIKE is 564 bytes only. Additionally with the recently proposed compression techniques[1],

the size of the public key can be reduced further. This makes SIKE extremely suitable for applications which have a very strict bandwidth requirement. However, the timing performance of SIKE is still significantly slow compared to lattice based post-quantum algorithms, and therefore performance of SIKE must be improved to compete with other post-quantum public key algorithms.

The fast implementation of traditional elliptic curve cryptosystems generally takes advantage of the fast modular reduction routine of pseudo-Mersenne or Solinas prime on which the curve is based on. However, such advantage is not present in case of SIKE. The SIKE protocol is implemented on a supersingular elliptic curve, defined over the field F_{p^2}. The prime p is of the form $2^{e_2}.3^{e_3} \pm 1$. Due to this special form, fast modular reduction routine of traditional elliptic curves can not be applied in this case. This becomes a major hurdle in the efficient implementation of SIKE which needs to be solved. Moreover, the size of the underlying field in case of SIKE is substantially larger than traditional elliptic curve cryptography. For example, popular elliptic curves like NIST P-256 and Curve25519 provides 128 bits security with field size of 256. However, to have 124 bit quantum security using SIKE, the field size needs to be 752 which adds to the complexity of the design.

In this paper, we propose an efficient implementation of SIKE for FPGA platforms. The proposed architecture exhibits similar latency to that of the existing best implementation of SIKE while consuming significantly less resources in terms of DSP and BRAM modules. To achieve this, we use a special Montgomery multiplier, operating in the redundant number system. This multiplier can execute multiple modular multiplications in parallel with very low critical path and is tailor made for speed-critical applications like SIKE. We combine this multiplier with efficient and customized field addition and subtraction modules which perform significantly better compared to the standard field adder and subtractors, deployed in existing implementation of SIKE. Finally, these base field operation circuitry are integrated with an efficient scheduling structure to implement a high performance architecture of SIKE. The contribution of the paper can be summarized as below:

- Our first objective will be to develop fast and efficient modules for implementing field operations for execution of SIKE. The most time consuming field operation is multiplication in F_p. To execute this, we have used an updated implementation of Montgomery multiplier in the redundant number system [2]. The proposed Montgomery multiplier has following features:
    1. The execution of Montgomery multiplication involves integer multiplication and multi-operand additions. As our proposed Montgomery multiplier is based on the redundant number system, the carry propagation during these operations can be restricted to a small values. For example, in our proposed implementation, the carry propagation is limited to 16 bits only which makes the critical path of the architecture considerably small.
    2. To improve the timing performance of the design further, we have incorporated carry save adders for the multi-operand addition operations. It must be noted that implementation of carry save adders using only look up tables (LUTs) will be slower compared to ripple carry adders on FPGAs. This is due to the fact that ripple carry addition operations on FPGAs are automatically mapped to the fast carry chains by the CAD tools. Therefore, to truly unlock the advantages of carry save adders, we have implemented the carry save adders using the fast carry chains of modern FPGAs.
    3. This multiplier that we have deployed in our architecture can execute either three or four parallel modular multiplications in F_p depending upon the user's choice. This flexibility in terms of choosing the number of parallel multiplications to be executed distinguish our implementation from the existing implementations of Montgomery multiplier. It will also play a key part in improving the overall performance of SIKE.

- To augment the field multiplier module, we have also used an efficient field adder circuitry [3]. This field adder module is based on a hybrid architecture of ripple carry and carry look ahead adders and utilizes the fast carry chains of modern FPGAs in much improved way compared to standard ripple carry adders.
- BRAM is a critical component of modern FPGAs and the existing implementations of SIKE consumes a large number of BRAMs during its execution. Compared to them, due to our memory optimization policy, our proposed architecture consumes around 58% less BRAMs compared to existing implementations
- We have also determined an optimal scheduling of the field operations so that we can tap out the maximum efficiency from the high speed field operation modules. The resulting architecture of SIKE which we have developed is significantly more efficient compared to existing implementations of SIKE.

The rest of the paper is organized as follows. In section 2, we will briefly discuss the relevant literature regarding SIKE and its implementation. Section 3 will focus on the preliminaries of SIKE protocol. In this section, we will also introduce the basics of redundant number system which will be useful in understanding the architecture of the Montgomery field multiplication. In section 4 and in 5, we will explain the architecture of different field operation modules and in section 6 we will introduce our proposed architecture of SIKE. Section 7 will provide detailed performance analysis of the proposed architecture and finally we will conclude the paper in section 8.

2 Related Works

The first work which focuses on the application of isogeny to implement post-quantum secure Diffie-Hellman key exchange algorithm was proposed in [4]. This work was based on the difficulty of computing isogeny between ordinary elliptic curves. However, in [5], a sub-exponential attack in the quantum domain was proposed against the algorithm of [4]. Subsequently, in [6], the authors proposed a modified Diffie-Hellman key exchange algorithm using the the property of difficulty in computing isogeny between supersingular elliptic curves. Supersingular elliptic curves are generally not used in traditional elliptic curve cryptography as computing elliptic curve discrete curve logarithm problem (ECDLP) is easy on those curves. However, isogeny based Diffie-Hellman key exchange algorithm on supersingular elliptic curves is found to thwart the attack proposed in [5]. The proposed supersingular isogeny based Diffie-Hellman key exchange (SIDH) algorithm provides $\mathsf{O}(\sqrt[4]{p})$ classical security and $\mathsf{O}(\sqrt[6]{p})$ quantum security where p is the characteristic of the underlying field on which the supersingular elliptic curve is built. The work of [6] was later updated in [7], where the authors optimized the implementation of SIDH significantly. The main contribution of this work is to propose the usage of projective coordinates during the computation of SIDH. Projective coordinates reduces the number of field inversions required in the SIDH computation. This improves the performance of the SIDH computation significantly as inversion is a costly field operation. The final supersingular isogeny based key exchange algorithm (SIKE) which was submitted in the NIST post-quantum public key standardization procedure is based on this work.

The implementation of SIDH and SIKE on FPGA has emerged as an interesting design problem in recent years. The first implementation of SIDH was proposed in [8] where the authors have implemented the SIDH architecture on Virtex-7 FPGA. This implementation was based on [6] and uses `Kaliski inverter` to implement the field inversion. The field multiplier was implemented using systolic array based Montgomery multiplication algorithm. Later, after the publication of [7], which significantly reduces the number of inversions required to implement SIDH, the authors proposed an updated implementation of SIDH in [9]. This implementation is significantly faster compared to the [8] and exploits the parallelism involved during the computation of isogeny. This work was further improved in [10] with increased usage of field multipliers.

All the aforementioned implementations of SIDH or SIKE exhibit the following features:

- The architectures use systolic array based Montgomery multiplier to implement field multiplication in F_p. Their proposed multiplier can interleave two parallel modular multiplication which in turn improves the performance of the architecture significantly.
- The multiplication in F_{p^2} is implemented using Karatsuba multiplier. This reduces the number of field multiplication in F_p
- All the above mentioned architecture are built on extensive usage of DSP blocks and block RAM resources of modern FPGAs.

In our proposed implementation, we will use an alternative implementation of Montgomery multiplier to implement field multiplication in F_p. More specifically, we will be using redundant number system to implement Montgomery multiplier with the capability either three or four parallel modular multiplications in F_p. Additionally, our proposed architecture requires significantly lesser number of DSP blocks and block RAMs compared to existing implementation with similar timing performance.

3 Preliminaries

In this section we will present a brief overview of SIDH and SIKE protocol. Additionally, we will also present basics of redundant number system which plays a major role in the proposed architecture of SIDH implementation.

3.1 Supersingular Montgomery Elliptic Curve

All the isogeny based cryptographic protocols are built on supersingular Montgomery elliptic curves. A Montgomery curve $E_{A,B}$ over a finite field F_q can be defined as below:

$$E_{a,b} : by^2 = x^3 + ax^2 + x, \quad a, b \quad F_q \tag{1}$$

The curve equation can also be expressed in projective domain as follows:

$$E_{A/C,B/C} : By^2 = Cx^3 + Ax^2 + Cx, \quad A, B, C \quad F_q \quad and \quad a = A/C, b = B/C \tag{2}$$

In recent years, Montgomery curves have become immensely popular as it provides more efficient elliptic curve operations compared to short Weierstrass curves. Montgomery curves support differential addition and doubling computation using only the x coordinates ($X : Z$ coordinates in projective domain) of the input points. This feature of the Montgomery curves will also be utilized during the computation of SIDH (and SIKE). More details on Montgomery curves in the context of elliptic curve cryptography can be found in [11].

A Montgomery curve $E_{A/C,B/C}$ defined over a field F_q with characteristic p is said to be supersingular if $p|(q + 1 - \#E_{A/C,B/C})$ and ordinary otherwise. Supersingular curves are generally not used in ECDH or ECDSA algorithms as ECDLP on supersingular curve $E_{A/C,B/C}$ defined over F_q can be mapped to DLP in F_p using MOV reduction [12]. DLP in F_q can be solved in subexponential time using index calculus method. However, in case of isogeny based cryptography, supersingular curves are preferred over ordinary elliptic curves as supersingular curves support non-commutative endomorphism ring, preventing the attack by [5].

3.1.1 $j - invariant$

$j - invariant$ of an supersingular Montgomery elliptic curve can be defined as follows:

$$j(E_{a,b}) = \frac{256(a^2 - 3)^3}{a^2 - 4} \tag{3}$$

In projective domain, the $j - invariant$ can be computed as below:

$$j(E_{A/C,B/C}) = \frac{256(A^2 - 3C^2)^3}{C^4(A^2 - 4C^2)} \tag{4}$$

In case of isogeny based cryptography, the objective of the Di e-Hellman key exchange protocol is to make sure that the two communicating property ends up with two di erent but isomorphic curves. This can be veri ed by computing the j invariant of the curves as isomorphic curves will have same j invariant value. Therefore this j invariant values of the nal curves can act as the shared secret key. Another important observation is that during the computation of j invariant , we do not require the value of b or B. Thus is our future discussion, we will denote the Montgomery curve $E_{A=C;B=C}$ as $E_{A=C}$.

3.2 Isogeny

An isogeny $: E_1 ! E_2$ is a non-constant rational map between two elliptic curves E_1 and E_2 de ned over a nite eld F_q. It can also be de ned as a group homomorphism which preserves the group structure of the elliptic curve. More speci cally, isogeny maps an elliptic curve of one particular isomorphism class to a new elliptic curve of another isomorphism class. A separable isogeny can be uniquely de ned by the kernel of it. Moreover in this case, the degree of the isogeny will be equal to the order of the kernel. Kernel of the isogeny can be de ned as follows:

$$\ker( ) = f P 2 E_1; \ (P) = Og \tag{5}$$

For any subgroup H in E_1, there exist a unique isogeny (upto isomorphism) such that $\ker( ) = H$ and $\deg( ) = jHj$. In other words, the isogeny map can be uniquely identi ed by its kernel H. For any prime, $l 6= p$, there exist $l + 1$ isogenies of degree l. Additionally, larger degree isogeny computation can be decomposed into computing multiple smaller degree isogeny. An isogeny map of degree l^e can be computed as a chain of degree l isogeny as shown below

$$= \ _{e 1} \ _{e 2} \ ::: \ _0 \tag{6}$$

Each $_i$ indicates an isogeny map of degree l. Degree l isogeny can be uniquely computed from its kernel using Velu's formula [13].

In case of isogeny computation, we can create a graph where each node indicates a particular isomorphism class and each edge indicates a degree l isogeny. For a large such graph, the path between two distant nodes is di cult to determine without the knowledge of the kernel. The security of the isogeny based public key cryptosystem relies on the hardness of computing the isogeny map , given the image and preimage curve E_2 and E_1. It must be noted that kernel H of the isogeny map is kept secret and is not known to the adversary.

3.3 Public Parameters

In this section, we will brie y state the di erent public and secret parameters used in isogeny based cryptography.

Integer e_2 and e_3 which will de ne the prime $p = 2^{e_2}3^{e_3} \ 1$ and the nite eld F_{p^2}. Any element $a 2 F_{p^2}$ can be de ned as $a_0 + i:a_1$ where $a_0; a_1 2 F_p$ and $i = \sqrt{-1}$.
Starting curve : $E_0=F_{p^2} : y^2 = x^3 + x$. This curve can also be de ned as $E_0(F_{p^2}) = E_0[2^{e_2}] \ E_0[3^{e_3}] \ F$. Here $E_0[2^{e_2}]$ and $E_0[3^{e_3}]$ denotes the subgroup of 2^{e_2} torsion points and 3^{e_3} torsion points in $E_0(F_{p^2})$ respectively. F denotes the set which contains the points having order co-prime to 2 and 3.
Public generator points: We choose $P_2 2 E_0=F_{p^2} n E_0(F_p)$, $Q_2 2 E_0(F_p)$ such that $f P_2; Q_2 g$ forms the basis of $E_0(F_{p^2})[2^{e_2}]$. Similarly we choose another pair of points $f P_3; Q_3 g$ such that $P_3 2 E_0=F_{p^2} n E_0(F_p)$, $Q_3 2 E_0(F_p)$. $f P_3; Q_3 g$ forms the basis of $E_0(F_{p^2})[3^{e_3}]$. For e ciency reason, rather than using $f P_2; Q_2 g$, we will use $f x_{P_2}; x_{Q_2}; x_{D_2} g$ as public parameters ($D_2 = P_2 \ Q_2$). This is done to utilize the

Algorithm 1: Computing the degree-4 isogenous curve

Input : $(X_{P_4} : Z_{P_4})$ where P_4 constitutes the kernel $< P_4 >$ of degree 4 isogeny and order of P_4 is 4 on $E_{A=C}$
Result: Curve coefficients $(A^+_{24} : C_{24})$ $(A^0 + 2C^0 : 4C^0)$ of $E_{A^0=C^0} = E_{A=C} = < P_4 >$ and constants $(K_1; K_2; K_3)$ 2 $(F_p^2)^3$

$K_2 = X_{P_4} - Z_{P_4}$
$K_3 = X_{P_4} + Z_{P_4}$
$K_1 = Z^2_{P_4}$
$K_1 = K_1 + K_1$
$C_{24} = K_1^2$
$K_1 = K_1 + K_1$
$A^+_{24} = X^2_{P_4}$
$A^+_{24} = A^+_{24} + A^+_{24}$
$A^+_{24} = (A^+_{24})^2$
return $A^+_{24}; C_{24}; K_1; K_2; K_3$

efficient x coordinate only arithmetic of Montgomery curves. Similarly, we encode points $\{P_3, Q_3\}$ as $\{x_{P_3}, x_{Q_3}, x_{D_3}\}$ as public parameters ($D_3 = P_3 - Q_3$).

3.4 Isogeny Computation

In this subsection, we will focus on the computation of large degree isogenies from smaller degree isogeny values. We will first focus on the computation of degree-4 and degree-3 isogeny as they serve as the basic building block in the execution of SIDH and SIKE.

3.4.1 Degree 4 and degree 3 isogeny

The computation of a degree-4 isogenous curve $E_{A^0=C^0}$ from the curve $E_{A=C}$ given the kernel generator point P_4 is shown in algorithm 1. The computation involves 4 field squarings, 4 field additions and one field subtraction (in F_{p^2}). It must be noted that algorithm 1 returns $A^0 + 2C^0$ and $4C^0$ instead of A^0 and C^0. This is done to reduce the number of field operations during the computation of the isogeny mapping of a given point [14]. Apart from $A^0 + 2C^0$ and $4C^0$, the algorithm 1 also produces three constants K_1, K_2 and K_3 which are utilized in the computation of the degree-4 isogeny mapping of a given point. The steps for computing the mapping of a point on the degree-4 isogenous curve is shown in algorithm 3. It requires 6 field multiplications, 2 field squarings, 3 field additions and 3 field subtractions (in F_{p^2}).

Algorithm 2 lists the steps for computing degree-3 isogenous curve. In this case, the algorithm returns $A^0 + 2C^0$ and $A^0 - 2C^0$ instead of A^0 and C^0. Additionally, algorithm 2 also returns curve constants K_1 and K_2 which are used during the computation of degree 3 isogeny mapping of a given point as shown in algorithm 4. Algorithm 2 requires 3 field squarings, 2 field multiplication, 3 field subtractions and 12 field additions (in F_{p^2}). Algorithm 4, on the other hand, requires 2 field squarings, 4 field multiplications, 2 field subtraction and 2 field additions. The formulations of computing degree 4 and degree 3 isogeny, described in this section, are optimized interpretation of Velu's formula [13] and is in accordance with [14].

3.5 Computing Larger Degree Isogenies

In the previous section we have illustrated how to compute degree-3 and degree-4 isogenous curves. We have also shown how to evaluate the map of point on degree-3 and 4 isogenous curves. In this section, we will show how to compute degree-3^{e_3} or degree-$4^{\frac{e_2}{2}}$ isogenous curves using the aforementioned functions. More specifically, we will discuss the

Algorithm 2: Computing the degree-3 isogenous curve

Input : $(X_{P_3} : Z_{P_3})$ where P_3 constitutes the kernel $< P_3 >$ of degree 3 isogeny and order of P_3 is 3 on $E_{A=C}$
Result: Curve coefficients $(A^+_{24} : A_{24})$ $(A^0 + 2C^0 : A^0 - 2C^0)$ of $E_{A^0=C^0} = E_{A=C} = < P_3 >$ and constants $(K_1; K_2)$ 2 $(F_p^2)^2$

$K_1 = X_{P_3} - Z_{P_3}$
$t_0 = K_1^2$
$K_2 = X_{P_3} + Z_{P_3}$
$t_1 = K_2^2$
$t_2 = t_0 + t_1$
$t_3 = K_1 + K_2$
$t_3 = t_3^2$
$t_3 = t_3 - t_2$
$t_2 = t_1 + t_3$
$t_3 = t_3 + t_0$
$t_4 = t_3 + t_0$
$t_4 = t_4 + t_4$
$t_4 = t_1 + t_4$
$A_{24} = t_2 \cdot t_4$
$t_4 = t_1 + t_2$
$t_4 = t_4 + t_4$
$t_4 = t_0 + t_4$
$t_4 = t_3 \cdot t_4$
$t_0 = t_4 - A_{24}$
$A^+_{24} = A_{24} + t_0$
return $A^+_{24}; A_{24}; K_1; K_2$

Algorithm 3: Evaluating degree-4 isogeny on a point

Input : Constants $K_1, K_2, K_3 \in (F_{p^2})^3$ from algorithm 1 and $(X_Q : Z_Q)$ where the point Q is on the curve $E_{A=C}$
Result: $Q' = (X_{Q'} : Z_{Q'})$ where $Q' \in E_{A'=C'}$

$t_0 = X_Q + Z_Q$
$t_1 = X_Q - Z_Q$
$X_Q = t_0 \cdot K_2$
$Z_Q = t_1 \cdot K_3$
$t_0 = t_0 \cdot t_1$
$t_0 = t_0 \cdot K_1$
$t_1 = X_Q + Z_Q$
$Z_Q = X_Q - Z_Q$
$t_1 = t_1^2$
$Z_Q = Z_Q^2$
$X_Q = t_0 + t_1$
$t_0 = Z_Q - t_0$
$X_{Q'} = X_Q \cdot t_1$
$Z_{Q'} = Z_Q \cdot t_0$
return $(X_{Q'} : Z_{Q'})$

methodology of computing l^e degree of isogeny from the computation of l degree isogeny. As shown in [6], from a curve E' and a kernel generator point R_0 of degree l^e, we can compute a chain of degree l isogenies as

$$E^{i+1} = E^i / < l^{e-i-1} R_i >; \quad \phi_i : E^i \to E^{i+1}; \quad R_{i+1} = \phi(R_i); \quad 0 \le i < e \qquad (7)$$

As per this algorithm, the total number of scalar multiplications by l is equal to $\frac{e(e-1)}{2}$. Therefore, the complexity of this algorithm is $O(e^2)$ scalar multiplication by l operations which can be improved significantly as shown next.

3.5.1 Optimum Larger Isogeny Computation

Figure 1(a) shows an acyclic graph structure for computing isogeny of degree l^6. The solid line in the figure indicates scalar multiplication by l and the dashed line indicates degree l isogeny computation. At the start, only the kernel generator point R_0 is known. To compute the isogeny of degree l^6, we need to compute all the leaf nodes' value of the tree structure of the figure 1(a). One of the way to compute the value of the leaf nodes is shown in figure 1(b), where each leaf node is computed by repeated executions of scalar multiplication by l. This computational strategy is actually analogous to equation (7) which has complexity of $O(e^2)$. However, if we follow the computational strategy of figure 1(c), we can compute the isogeny map with much lesser number of scalar multiplication operations. In [6], the authors have shown that such triangular decomposition yields a computational strategy with $O(e \log e)$ number of scalar multiplication by l operations and similar number of degree l isogeny computation. More details on finding such optimum computational strategy can be found in [6, 14]. It must be noted that selection of such optimum computational strategy also depends upon the cost of scalar multiplication and degree-l isogeny computation.

Once the computational strategy gets defined, the computation of the isogeny map becomes straightforward. We label each node which has more than one child with number of leaf nodes to its right side. We then walk the tree in depth-first left-first manner and output the node labels as we encounter them. This process is called linearization. For example, if we apply linearization on the computational strategy shown in figure 1, its output will be $\{3, 1, 1, 1, 1\}$. More formally, the size of the computational strategy after the linearization will be $e-1$ for l^e degree isogeny computation. In appendix C, we have provided the Algorithm 6 [14] which shows the methodology of computing l^e degree isogeny using optimum computational strategy.

Algorithm 4: Evaluating degree-3 isogeny on a point

Input : Constants $K_1, K_2 \in (F_{p^2})^2$ from algorithm 2 and $(X_Q : Z_Q)$ where the point Q is on the curve $E_{A=C}$
Result: $Q' = (X_{Q'} : Z_{Q'})$ where $Q' \in E_{A'=C'}$

$t_0 = X_Q + Z_Q$
$t_1 = X_Q - Z_Q$
$t_2 = K_1 \cdot t_0$
$t_1 = K_2 \cdot K_1$
$t_2 = t_0 + t_1$
$t_0 = t_1 - t_0$
$t_2 = t_2^2$
$t_0 = t_0^2$
$X_{Q'} = X_Q \cdot t_2$
$Z_{Q'} = Z_Q \cdot t_0$
return $(X_{Q'} : Z_{Q'})$

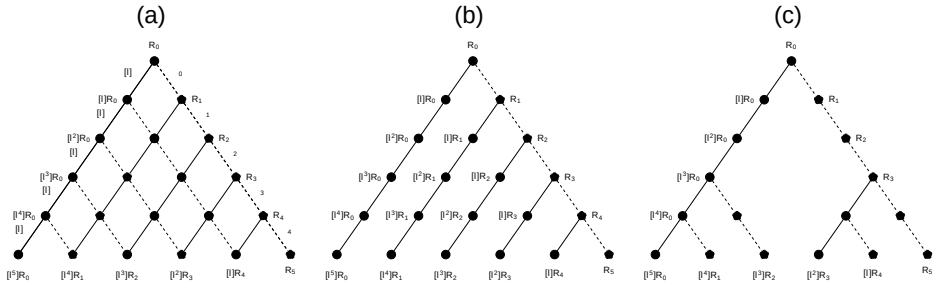


Figure 1: Various Strategy for Computing Isogeny Map of degree 3^6

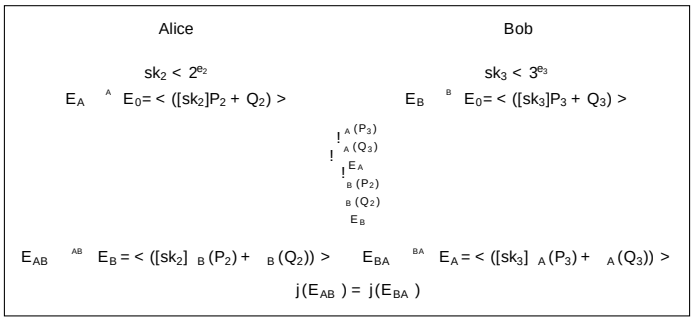


Figure 3: The Key Exchange Protocol using SIDH

3.6 SIKE Protocol

In this section, we will brie y describe the supersingular isogeny based key exchange algorithm. The protocol description is shown in gure 3. Alice selects her secret key $sk_2 < 2^{e_2}$ and computes the point $R_2 = [sk_2]P_2 + Q_2$ and computes the image curve $E_A \quad {}^{A} \quad E_0 = < R_2 >$ where E_0 is the starting curve. Similarly Bob also selects his secret key $sk_3 < 3^{e_3}$ and computes the corresponding image curve $E_B \quad {}^{B} \quad E = < R_3 = [sk_3]P_3 + Q_3 >$. Alice and Bob then publish E_A, $_A P_3$, $_A Q_3$ and E_B, $_B P_2$, $_B Q_2$ respectively. Subsequently, Alice and Bob computes $E_{AB} \quad {}^{AB} \quad E_B = < ([sk_2] \ _B(P_2) + \ _B(Q_2)) >$ and $E_{BA} \quad {}^{BA} \quad E_A = < ([sk_3] \ _A(P_3) + \ _A(Q_3)) >$ respectively. The curves E_{AB} and E_{BA} are isomorphic to each other. Therefore, their j invariant will be same which can be used as shared secret.

3.7 Redundant Number System

The idea of applying redundant number system to enhance the performance of eld arithmetic circuits was rst proposed in [15]. In this section we will brie y introduce the concepts of redundant number system.

A d digit non-redundant number X can be represented as $(X_{d-1}; \ldots; X_1; X_0)$ where each X_i is an r bit number. This is also known as radix-2^r number. The value of X is $\sum_{i=0}^{d-1} X_i \cdot 2^{ir}$ and it can represent a number up to $2^{dr} - 1$. Additionally, such representation is unique. A d digit and n bit redundant number X' can be represented as $(X'_{d-1}; \ldots; X'_1; X'_0)$ where each X'_i is an $(r + n)$ bit number and $n < r$. The value of X' is $\sum_{i=0}^{d-1} X'_i \cdot 2^{ir}$. We can partition each X'_i into two parts: $X'_i[r-1:0]$ and $X'_i[r+n-1:r]$. $X'_i[r-1:0]$

are known as the principal bits and $X_i^0[r+n-1:r]$ are known as the redundant bits. Redundant number can be converted into non-redundant number by simple addition as follows:

$$\begin{array}{rlll} & \cdots & X_1^0[r+n-1:r] & X_0^0[r+n-1:r] \\ + & \cdots & X_2^0[r-1:0] & X_1^0[r-1:0] \quad X_0^0[r-1:0] \\ \hline & \cdots & X_2 & X_1 \quad X_0 \end{array}$$

The representation of a redundant number is not unique. Additionally, every non-redundant number by default can be represented as a redundant number by considering the value of the redundant bits as zero.

Example: Carry Save form : A number X in carry save form is characterized by two values X_{sum} and X_{carry}. It is a special case of a redundant number in radix-2, where X_{sum} constitutes the principal bits and X_{carry} constitutes the redundant bits of the number X.

3.7.1 Over ow condition

A d digit radix-2^r redundant number X^0 represents a maximum value of $\sum_{i=0}^{d-1}(2^{r+2}-1)2^{ir} > 2^{dr}$. However, the permissible value of X^0 is $2^{dr}-1$ as it is a representation of d digit radix-2^r non-redundant number. Whenever the value of X^0 becomes more than $2^{dr}-1$, we consider that as over ow.

As we have already mentioned, the application of redundant number system on Montgomery multiplication was rst proposed by [15]. Later, this implementation was updated in [2] where the authors have modi ed the methodology of [15] to apply it to the asymmetric multipliers of modern Xilinx FPGAs. The details of di erent arithmetic operations in redundant number system along with its application using asymmetric multipliers of modern FPGAs are discussed in appendix 8.

4 Field Operation in F_p

4.1 Field Multiplication in F_p

The underlying eld on which the SIKE protocol is built does not support reduction friendly primes. Therefore, e cient implementation of eld multiplication in this eld is challenging. In the previous works of SIKE implementations, the authors have used Montgomery multiplication algorithm to perform eld multiplication in F_p ($p = 2^{e_2}3^{e_3}-1$, $e_2, e_3 \in Z^+$). More categorically, they have used a speci c implementation of Montgomery multiplier based on systolic array architecture. The di erences between the proposed Montgomery multiplier and the exiting systolic array based architecture [8, 9, 10] are manifold:

- The proposed Montgomery multiplier is based on the redundant number system and is capable of performing three or four parallel modular multiplication simultaneously. Compared to this, systolic array based Montgomery architecture performs only two parallel modular multiplications in an interleaved fashion.
- The existing systolic array based architectures uses DSP blocks of modern FPGAs as 16×16 integer multipliers. However, the modern 7 series Xilinx FPGAs have 24×17 unsigned integer multipliers embedded inside the DSP blocks. This results in under-utilization of the the DSP blocks. Our proposed architecture on the other hand uses DSP blocks as 24×17 unsigned multiplier, unlocking its full capability.
- In this section, we will provide detailed description of the multiplier architecture.

4.1.1 Montgomery Multiplication

Montgomery multiplication was proposed in [18] and it showed how to perform modular reduction without trial division. The initial version of Montgomery multiplication [18] was not constant time as it involved a conditional subtraction operation. In [16, 17],

Algorithm 5: Constant Time Montgomery Multiplication [16, 17]

INPUT: $M = \sum_{i=0}^{m-1} m_i^0 \, 2^{ri}$, $A = \sum_{i=0}^{m+2} a_i \, 2^{ri}$ with $a_{m+2} = 0$, $B = \sum_{i=0}^{m+1} b_i \, 2^{ri}$, $M^0 = -M^{-1} \bmod R$, $\overline{M} = (M^0 \bmod 2^r) \, M = \sum_{i=0}^{m+1} \overline{m}_i \, 2^{ri}$, $A, B < 2\overline{M}$, $4\overline{M} < 2^{rm}$, $R = 2^{r(m+2)}$

OUTPUT: $A \, B \, R^{-1} \bmod M$ $S_0 = 0$

for $i \leftarrow 0$ to $m+2$ do

  $q_i = S_i \bmod 2^r$

  $S_{i+1} = (S_i + q_i \, \overline{M})/2^r + a_i \, B$

end

return $S_{m+3} = A \, B \, R^{-1} \bmod M$

the authors proposed an alternative high radix Montgomery multiplication algorithm which removed the conditional subtraction operation to make its implementation constant time. This algorithm is shown in algorithm 5 which represent a r-radix Montgomery multiplication. Here the modulus M is divided into $\{m_0^0, m_1^0, \ldots, m_{m-1}^0\}$, r bit words such that $M = \sum_{i=0}^{i=m-1} m_i^0 \, 2^{ri}$. Similarly, the multiplicand operands A and B are represented as $m+3$ and $m+2$, r bits word respectively ($A = \sum_{i=0}^{m+2} a_i \, 2^{ri}$, $B = \sum_{i=0}^{m+1} b_i \, 2^{ri}$). The value of a_{m+2} is always zero. The Montgomery multiplication also requires the knowledge three parameters: $R = 2^{r(m+2)}$, $M^0 = -M^{-1} \bmod R$ and $\overline{M} = (M^0 \bmod 2^r) \, M$. These values can be precomputed and henceforth do not contribute to the complexity of the architecture. The range of the input A and B are $[0; 2\overline{M}]$ and the range of the output S_{m+3} is $[0; 2\overline{M}]$, therefore a final range correction is necessary. Input to the Montgomery multiplication should be transformed into Montgomery domain. However, such conversion is executed only once as the entire SIKE protocol can be executed in the Montgomery domain. The value of an operand A in Montgomery domain will be $A \, R \bmod M$ which can be calculated by performing Montgomery multiplication between A and R^2. The conversion of the final result from Montgomery domain to normal integer domain can be executed by multiplying the final result with 1 using the Montgomery multiplier.

4.1.2 Multiplier Architecture

The usage of redundant number system to implement Montgomery multiplication was first proposed in [15]. In [2], the authors have updated this implementation for better utilization of asymmetric multipliers. Additionally, the authors have also provided a methodology to perform multiple modular multiplication using their proposed architecture. In this work, we will take advantage of this property.

Figure 4a shows the architecture of the Montgomery multiplier implemented using redundant number system in F_p. The input operand A is a redundant number in base r_1 where $A = \sum_{i=0}^{m_a+2} a_i \, 2^{r_1 i}$ with $a_{m_a+2} = 0$ where $m_a = \lceil \frac{\log_2(M)}{r_1} \rceil$. Similarly, we encode B as a redundant number in radix-2^{r_2} ($B = \sum_{i=0}^{m_b+1} b_i \, 2^{r_2 i}$, where $m_b = \lceil \frac{\log_2(M)}{r_2} \rceil$). The parameter $\overline{M}$ is also encoded as a redundant number of base r_2 and the final result (stored in the register S_i in figure 4a) will be a redundant number in base r_2. In [2], the authors have used asymmetric embedded multipliers of dimension 24×17 in the architecture. They have chosen the the value of r_1 as 22 and the value of r_2 is 15 (refer appendix A). This is motivated by the fact that two bits are needed to be preserved for redundant bits to ensure no over flow during the multi-operand additions, involved in the Montgomery multiplication algorithm. However, if we compute the multi-operand addition in carry save form, we do not have the issue of over flow. For example, when we perform addition of three redundant numbers, each with two redundant bits, the output of the addition will still be a redundant number with two redundant bits. But if we add three redundant numbers, each with one redundant bit, the output of the addition will not remain a redundant number with one redundant bits due to the over flow. However, if we add those three redundant numbers in a carry save form, we do not have the issue of over flow as we are storing the carry bits

separately. In our proposed architecture, we have followed this strategy and have used only one bit as the redundant bit. This allows us to use 23 as value of r_1 and 16 as value of r_2. We will illustrate the advantage behind this choice later.

The different sub-operations involved during the implementation of Montgomery multiplications (refer algorithm 5) are: 1. Multiplication (a_i B), 2. Multiplication and Accumulation ($S_i + q_i$ $\overline{M}$), 3. Right Shift ($(S_i + q_i$ $\overline{M})/2^{r_1}$), 4. Addition ($(S_i + q_i$ $\overline{M})/2^{r_1} + a_i$ B) and 5. Mod (S_{i+1} mod 2^{r_1})

The multiplication operation between a_i and B can be performed as shown in appendix A in redundant number system using DSP blocks. Similarly, implementation of the multiply and accumulation operation ($S_i + q_i$ $\overline{M}$) in redundant number system is shown in appendix A. The operations like right shift, addition and mod operations are easy to execute. As we have already mentioned, the implementation of Montgomery multiplier requires multiple multi-operand addition operations in redundant number system. For example, simple multiplication operation (refer appendix B) requires addition between three operands whereas multiply and accumulate operation requires addition operations between 5 input operands. To achieve this, we have implemented $3:2$, $4:2$ and $5:2$ compressor circuits using the fast carry chains of modern Xilinx FPGAs. Deployment of such compressor circuits in the architecture improves the performance of the design significantly. Details of the implementation of compressor circuits using fast carry chains can be found in appendix B. The compressor circuits produces the output in carry save form which needs to be transformed in to the redundant number of base r_1 which is achieved by the base converter.

In this architecture, DSP blocks are required mainly for computing partial products of a_i B and q_i $\overline{M}$, where the dimension of a_i and q_i are $r_1 + 1$, and dimension of B and $\overline{M}$ are $(m_b + 2)(r_2 + 1)$. If a single DSP block can execute $(r_1 + 1)$ $(r_2 + 1)$ integer multiplication in a single cycle, we would require $2m_b + 4$ DSP blocks to generate all the partial products of operation a_i B and q_i $\overline{M}$ in a single cycle. This motivates the choice of choosing r_1 as 23 and r_2 as 16 as that allow us to fully utilize the embedded 24×17 unsigned multiplier of DSP blocks. For 752 bits modular multiplication, we will require $2 \cdot \lceil 752/16 \rceil + 4 = 98$ DSP blocks. If we remove the register layer in the architecture, the clock cycle requirement of the design becomes $m_a + 3$. This is due to the fact that in each clock cycle we are executing a_i B and i varies from 1 to $m_a + 3$, as the dimension of A is $(m_a + 3)(r_1 + 1)$. In this scenario, the critical path of the architecture consists of DSP blocks, $5:2$ compressor circuits, shifter module, $4:2$ compressor and base converter module. However, in [2], the authors made an interesting observation that when the multiplier modules are active, the other modules (compressor, shifter and base converter module) are inactive. Therefore, if we place a register layer between the multipliers and the other modules, we can actually execute two modular multiplications simultaneously (refer figure 4a). Total clock cycle requirement to execute these two parallel modular multiplications will be $2(m_a + 3) + 1 = 2m_a + 7$, one extra clock cycle is because of the initial latency. The efficiency arises from the reduced critical path which is now max(delay of multipliers, delay of other modules combined). Figure 4a shows the architecture of the Montgomery multiplier performing two parallel modular multiplications using two redundant bits. The usage of the compressor circuits for carry save additions is a key distinguishing feature compared to [2], leading to a significant saving in terms of clock cycles which we explain below.

By introducing more register layers at the appropriate positions, the architecture can be extended further to execute three or even four parallel modular multiplications. As shown in figure 4b, if we introduce another register layer between the first level of compressor circuits (consist of $3:2$ and $5:2$ compressor circuits) and the second level of compressor circuits (consist of $4:2$ compressor and base converter module), we can perform three parallel modular multiplications. In this case, the critical path will reduce further as the critical path of figure 4a lies in the compressor modules. If we introduce

another register module between the $4:2$ compressor module and the base converter, we can execute four parallel modular multiplications in F_p. The clock cycle requirement for executing three and four parallel modular multiplication will be $3(m_a + 3) + 2 = 3\ m_a + 11$ and $4(m_a + 3) + 3 = 4\ m_a + 15$ respectively. It must be noted that if we have used 2 redundant bits, the value of r_1 and r_2 would be 22 and 15. In that case, the DSP block requirement would have become $2 \cdot \lceil (752/15) \rceil + 4 = 106$ and clock cycle requirement to perform three and four parallel modular multiplication is $3 \cdot \lceil (752/22) \rceil + 11 = 116$ and $4 \cdot \lceil (752/22) \rceil + 15 = 155$ respectively. On the other hand, if we use only one bit as redundant bit, the DSP block requirement becomes $2 \cdot \lceil (752/16) \rceil + 4 = 98$ and clock cycle requirement to perform three and four parallel modular multiplication is $3 \cdot \lceil (752/23) \rceil + 11 = 110$ and $4 \cdot \lceil (752/23) \rceil + 15 = 147$ respectively. This shows the advantage of using compressor circuits as it allow us to use only one bit as redundant bit.

One disadvantage of this architecture is that if we perform three modular multiplications in F_p using the architecture which can perform four parallel modular multiplications, the clock cycle requirement will be similar to performing four parallel modular multiplications. This could be problematic as we may not require four parallel modular multiplications in every step of the SIKE execution. To mitigate this issue, we have introduced a multipliexer in the architecture as shown in figure 4b. This multipliexer takes the combinatorial and the registered output of $4:2$ compressor modules and depending upon the control signal $three/\overline{four}$, pass one of them to the base converter module. If we need to perform three parallel modular multiplications, the multiplexer passes the combinatorial output of the $4:2$ compressor module to the base converter circuitry. On the other hand, it passes the registered output of the $4:2$ compressor modules when we need to perform four parallel modular multiplications. This flexibility in choice of performing three or four parallel modular multiplications allow us to have an efficient scheduling to implement SIKE. More details on this is given in the next section.

As we have already mentioned, existing implementations of SIKE uses Montgomery multiplier implemented using systolic arrays. As shown in [8, 10], implementation of SIKE with prime size 504 bits requires 64 DSP blocks and has a latency of 68 cycles for a modular multiplication operation. On the other hand, our proposed architecture will require 68 DSP blocks. But, it can execute three parallel modular multiplication in 80 cycles, which makes the effective latency of single modular multiplication $\lceil (80/3) \rceil = 27$. Similarly, if we consider primes of size 752 bits, the modular multiplier of [8, 10] will require 96 DSP blocks with a latency of 101 cycles for single modular multiplication. For the same prime size, our proposed architecture requires 98 DSP blocks performing three simultaneous modular multiplications in 110 cycles.

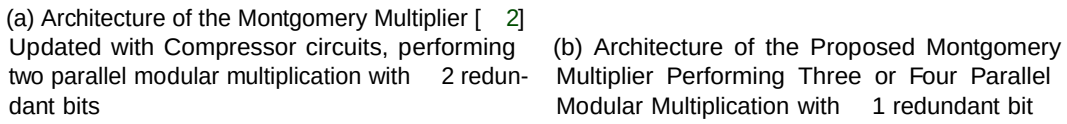
4.2 Field Addition in F_p

In this section, we will first introduce the architecture of the field addition module. For this, we will first introduce the architecture of an efficient ripple carry adder.

4.2.1 Ripple Carry Adder

Modern FPGAs are equipped with fast carry chains which makes the implementation of normal ripple carry adders significantly efficient. The fast carry chains exhibit less routing delay compared to traditional routing through LUTs.

There have been few works which have tried to use the fast carry chain of the FPGAs in an optimum manner to reduce the delay of the ripple carry adder. The idea is to implement a carry look ahead adder using the carry chain of the FPGAs. We can approximate the delay of a ripple carry adder by the length of the carry chain. Now, the key observation in this case is that a single LUT can handle a six input, single output function or five input two output function. Hence, in a single LUT, we can actually pass four inputs (two consecutive bits from each operand) and generate the corresponding p propagate and generate signals which are used for carry implementation of carry lookahead adder [19].



(a) Architecture of the Montgomery Multiplier [2] Updated with Compressor circuits, performing two parallel modular multiplication with 2 redundant bits

(b) Architecture of the Proposed Montgomery Multiplier Performing Three or Four Parallel Modular Multiplication with 1 redundant bit

Figure 4: Montgomery Multiplication in Redundant Number System

Now, we can generate this two signals from a single look up table which implies that using this method, we can look ahead the carry of two bits from a single look up table. Hence, this architecture can generate the carry output of the lower $n=2$ bits using a carry chain of length $n=4$. For n bits addition, we can divide it into two $n=2$ bits addition. For the upper $n=2$ bits, a separate carry look ahead logic can be implemented using the aforementioned method. Hence the total delay for the n bits addition is equivalent to delay of a carry chain of length $3n=4$ ($n=4$ for carry prediction and $n=2$ for upper bit addition). The addition results of lower $n=2$ bits are computed using a normal ripple carry adder. The total LUT count for this adder is $5n=4$.

This architecture was considerably improved in [3] where the authors have altered the approach of [19]. In this case, the n bit addition is partitioned into two blocks of K and M bits. The first block predicts the carry output of the lower K bits of inputs using the carry chain of length $K=2$. Additionally, in this scenario the addition result of lower K bits are also computed by additional $K=2$ LUTs as we can compute two consecutive sum bits by using the LUT as a 5 input, 2 output function. The upper block of M bits are computed in a normal ripple carry fashion. The key observation is that if we have implemented this adder in ripple carry fashion, it would have consumed $K + M = n$ number of LUTs. In this modified approach, the total consumption of the LUT is $K=2$ for generating carry output of K bits addition, $K=2$ for generating the sum of K bits and M for M bit addition using ripple carry adder. Thus the total LUT count is $K=2 + K=2 + M = K + M = n$. The delay of this adder is equivalent to the delay of the carry chain of length $K=2 + M$. Thus this architecture provides speed up without any additional requirement of extra LUTs, making this design optimum from area point of view.

For the implementation of field adder, we have developed a 256 bits adder following the methodology of [3]. In our architecture, the value of M is chosen as 32. Therefore the delay of 256 bit ripple carry adder is equivalent to delay of carry chain of length 144.

4.2.2 Modular Adder Architecture

The architecture of the modular adder is shown in figure 5. The basic building block of the architecture is the 256 bit ripple carry adder, details of which we have discussed in the

Figure 5: Modular Adder Architecture in F_p

previous section. As our proposed modular multiplier produces redundant number in base r_2 as output, the inputs to the modular adder are considered as redundant numbers. The first two 256 bit adder converts the redundant numbers into non-redundant number. The subsequent adder and subtractor modules perform $a + b$ and $a + b - 2\overline{M}$. The range of the output is $[0; 2\overline{M} - 1]$ and depending on the sign value, either addout $(a + b)$ or subout $(a + b - 2\overline{M})$ is stored as result. The modular subtractor can be implemented in a similar manner. the modular adder will require 5 clock cycles to perform a single modular addition for prime length of 504 bits. For prime length of 752 bits, the clock cycle requirement is 6.

5 Field Operation in F_{p^2}

In this section, we will present the implementation details of field operation in F_{p^2}.

5.1 Field Addition Operation in F_{p^2}

Any element a in the target field F_{p^2} can be represented as $a = a_0 + a_1 \cdot i$. Addition of such two elements a and b can be defined as below:

$$a + b = (a_0 + a_1 \cdot i) + (b_0 + b_1 \cdot i) = (a_0 + b_0) + (a_1 + b_1) \cdot i \quad (8)$$

Similarly subtraction operation can be defined as below:

$$a - b = (a_0 + a_1 \cdot i) - (b_0 + b_1 \cdot i) = (a_0 - b_0) + (a_1 - b_1) \cdot i \quad (9)$$

If we have two instances of modular adders in F_p (refer section 4.2), the modular addition in F_{p^2} will require 6 clock cycles for prime of size 752 bits and 5 clock cycles for prime of size 504 bits.

5.2 Field Multiplication in F_{p^2}

Multiplication of two elements a and b in the field F_{p^2} can be defined as below:

$$a \cdot b = (a_0 + a_1 \cdot i) \cdot (b_0 + b_1 \cdot i) = (a_0 \cdot b_0 - a_1 \cdot b_1) + (a_0 \cdot b_1 + a_1 \cdot b_0) \cdot i \quad (10)$$

Table 1: Scheduling of eld multiplication operation in F_{p^2} where the length of p is 752

Steps	Multiplication in F_p			Addition in F_p	Subtraction in F_p	Clock cycles
1	-	-	-	$a_0 + a_1$	b_0 b_1	6
2	a_0 b_1	a_1 b_0	$(a_0 + a_1)$ $(b_0$ $b_1)$	-	-	110
3	-	-	-	a_0 $b_1 + a_1$ b_0	a_0 b_1 a_1 b_0	6
4	-	-	-	-	$(a_0 + a_1)$ $(b_0$ $b_1)$- $(a_0$ b_1 a_1 $b_0)$	6

This requires 4 multiplications, one addition and one subtraction operations in F_p. However, the number of multiplication operations can be reduced if we apply Karatsuba technique which is illustrated below:

$$a \quad b = (a_0 + a_1\ i) \quad (b_0 + b_1\ i) = ((a_0 + a_1)\ (b_0 \quad b_1) \quad a_1\ b_0 + a_0\ b_1) + (a_0\ b_1 + a_1\ b_0)\ i \quad (11)$$

In this scenario, we require three multiplication and ve addition/subtraction operations in F_p. Thus, this method will require fewer number of clock cycles compared to straightforward multiplication methodology as long as the cost of single multiplication operation is more than the combined cost of three addition/subtraction operations. Assuming that the architecture has one modular multiplier which can execute three modular multiplication in F_p simultaneously, one adder and one subtractor in F_p, the scheduling of the modular multiplication operation in F_{p^2} is shown in table 1. The total clock cycle requirement to execute one eld multiplication in F_{p^2} is 128 for prime of size 752. It must be noted that our implementation of Montgomery multiplier in F_p which can perform three modular multiplications simultaneously is tailor made for this architecture.

5.3 Field Squaring in F_{p^2}

Squaring of an element a in F_{p^2} is executed as follows:

$$a \quad a = (a_0 + a_1 \quad i) \quad (a_0 + a_1 \quad i) = (a_0^2 \quad a_1^2) + 2\, a_0 \quad a_1 \quad i \qquad (12)$$

It requires three eld multiplications, one addition and one subtraction operation in F_p.

6 Architectural Details of SIKE Implementation

In this section, we will present the details of the proposed SIKE architecture. The block diagram of the proposed architecture is shown in gure 6. The major components of this architecture are: a) Block-RAM (BRAM) based memory modules, b) Three Montgomery multipliers in F_p, c) Two modular adders in F_p, d) Two modular subtractors in F_p and e) Control unit

As we have previously mentioned, the implemented Montgomery multiplier requires one of the multiplicand operands to be in the redundant form with base r_2 (in our case $r_2 = 16$) and other operand to be in the redundant form with base r_1 (in our case $r_1 = 23$). The output of the multiplication result will be a redundant number with base r_2. The base converter module , shown in the gure 6 converts a redundant number in base r_2 into a redundant number in base r_1 which ensures smooth execution of Montgomery multiplication. The control unit is encoded as a nite state machine which can administer public key generation along with secret key agreement for both Alice and Bob. The details of the eld multipliers, adders and subtractors circuits are already discussed in previous sections. We will start our discussion with the details of the memory module.

6.1 Memory Optimization

The proposed SIKE architecture is built for SIKEp751 con guration which provides 124 bits quantum security. The supersingular elliptic curve supporting SIKEp751 is built on

Figure 6: Block Diagram of the Proposed Architecture of SIKE

the eld F_{p^2} where p is a 752 bits prime. Therefore, each element which belongs to this eld is of size 1504 bits. Naturally, we need an e cient implementation of memory architecture to store di erent elements of aforementioned nite elds. Fortunately, FPGA provides block RAM as hard-IP which can be deployed for this purpose. We have implemented the BRAMs in a true dual port RAM configuration with write first mode.

For the proposed implementation, we can implement the block RAMs in two di erent ways. We can either store an element of F_{p^2} at a particular address location or we can store an element of F_p at a particular address location. In the rst scenario, memory read and write operation would be simpler as we can access an element from F_{p^2} in a single cycle. On the other hand, the second strategy will require two clock cycles to do the same. However, we have observed that the consumption of the block RAMs increases more rapidly with the increase in the data width compared to increase in memory depth. We will illustrate this with an example. Suppose, we want to develop a true dual port RAM using block RAMs which can store 512 elements of width 1504. We have found that we would require 45 BRAMs to achieve this. However, if we implement a true dual port RAM which can store 1024 elements with width 752, we would require 22.5 BRAMs only. Motivated by this observation, we have followed the second strategy where we store an element of F_p at a particular address location of the BRAM based memory module.

6.2 Scheduling of Di erent Operations For Execution of SIKE

The execution of the complete SIKE protocol involves three steps:

- Generation of public key by Bob.
- Encapsulation of shared secret by Alice using Bob's public and Alice's secret key.
- Decapsulation of shared secret by Bob using Alice's public key and Bob's secret key.

Each of these steps can be divided into following operation:

- Generation of the kernel generator point ($[sk_2]P_2 + Q_2$ for Alice and $[sk_3]P_3 + Q_3$ for Bob)

Table 2: Scheduling of Field Operations For Computing The Kernel Generator Point

Step No:	Addition in F_{p^2}	Subtraction in F_{p^2}	Modmul-1	Modmul-2	Modmul-3	Modmul-4
1	$t_0 = X_P + Z_P$	$t_1 = X_P \quad Z_P$	-	-	-	-
2	$t_4 = X_Q + Z_Q$	$t_1 = X_P \quad Z_P$	-	-	-	-
3	-	-	$t_2 = t_0^2$	$t_6 = t_1^2$	$t_5 = t_0 \quad t_3$	$t_7 = t_1 \quad t_4$
4	$t_{12} = t_5 + t_7$	$t_8 = t_2 \quad t_6$	-	-	-	-
5	-	$t_{10} = t_5 \quad t_7$	-	-	-	-
6	-	-	$t_{13} = t_{10}^2$	$t_{14} = t_{12}^2$	$X_{2P} = t_2 \quad t_6$	$t_9 = A_{24}^+ \quad t_8$
7	$t_{11} = t_9 + t_6$	-	-	-	-	-
8	-	-	$Z_{2P} = t_{11} \quad t_8$	$Z_{P+Q} = X_{Q \quad P} \quad t_{13}$	$X_{P+Q} = Z_{Q \quad P} \quad t_{14}$	-

Point doubling and tripling operations
Computation of degree4 and degree3 isogenous curve
Evaluation of a point on degree4 and degree3 isogenous curve.

The algorithm for computing of degree4 and degree3 isogenous curves along with evaluation of a point on degree4 and degree3 isogenous curves are shown in algorithm 1, 3, 2, 4. For details of other algorithms, one can refer to [14]. In this section, we will discuss how to schedule di erent eld operations involved in the aforementioned algorithms for e cient execution of SIKE.

Computing the Kernel Generator Point Table 2 shows the desired scheduling of di erent eld operations required for computing the kernel generator point. Alice (or Bob) chooses the secret key sk_2 (or sk_3) and computes the kernel generator point $R_2 = sk_2P_2 + Q_2$ (or $R_3 = sk_3P_3 + Q_3$). As shown in [14], this computation is achieved by repeated execution of combined doubling and di erential addition operation. This operation takes X and Z coordinates of point P, Q, and Q P in projective domain and produces X and Z coordinates of point 2P and P + Q in projective domain. Each iteration of combined doubling and di erential addition requires 11 eld multiplications (4 squarings and 7 multiplications), 4 eld additions and 3 eld subtractions in F_{p^2} (as shown in table 2). Moreover, each eld multiplication in F_{p^2} involves one eld addition and subtraction for operand splitting, three eld multiplication , and one eld addition, two eld subtractions in F_p to produce the nal multiplication result in F_{p^2}[1].

In our proposed architecture, we have implemented three modular multipliers in F_p, each capable of performing either three or four parallel modular multiplications. Therefore, we can actually execute either 9 or 12 eld multiplications in parallel. The clock cycle requirement when we execute 9 parallel eld multiplications in F_p is 110. For 12 parallel modular multiplications, the clock cycle requirement will be 147. As we have shown in table 2, the 11 eld multiplications in F_{p^2}, required to accomplish a single iteration of the combined doubling and di erential addition, can be executed in three multiplicative steps (steps 3, 6 and 8). In steps 3 and 6, we need to perform four parallel modular multiplications in F_{p^2} (12 modular multiplications in F_p), whereas in step 8, we need to execute three parallel modular multiplications in F_{p^2} (9 modular multiplications in F_p). Therefore, in steps 3 and 6, all the three modular multipliers in F_p are con gured to perform 4 parallel modular multiplications each, whereas in step 8, they are con gured to perform three modular multiplications each. It must be noted that if we had modular multipliers capable of performing only three modular multiplications in parallel, we would have required four multiplicative steps, each consuming 110 cycles (Total 440 cycles). On the other hand, having the exible multipliers allow us to complete the computation of eld multiplication in F_{p^2} in three multiplicative steps only. The rst two multiplicative

[1] In the scheduling table, we have not shown the eld addition and subtractions operations required for Karatsuba decomposition

steps consume 147 cycles each and the third multiplicative step requires 110 cycles (total 2 147 + 110 = 404 cycles). This shows the advantage of having the choice of switching between executing three or four parallel modular multiplications.

Point Doubling and Point Tripling : The scheduling of point doubling and point tripling are shown in table 3 and in table 4. Point doubling requires 2 squarings, 4 multiplications, 2 subtractions and 2 addition operations in F_{p^2}, On the other hand, point tripling requires 5 squarings, 7 multiplications, 7 subtractions and 5 addition operations. Point tripling is signi cantly a more expensive operation compared to point doubling and this makes Bob's public key generation more expensive compared to Alice's public key generation. The algorithm for computing point doubling and point tripling are provided in appendix C.

Computation and Evaluation of Degree 3 and Degree 4 Isogeny : The scheduling of the computation of degree 3 and degree 4 isogenous curves are shown in table 7 and in table 5 respectively. Similarly, scheduling of computing the map of a point on the degree 3 and 4 isogenous curves are shown in table 8 and in table 6. Details of these algorithms are already discussed in section 3. The eld inversions in F_p are computed using Fermat's theorem.

One interesting observation is that, apart from computing the kernel generator point, all other operations (point doubling, point tripling and other isogeny related functions) does not use the third multiplier at all. Thus, it may seem that that third multiplier in the architecture is underutilized. Moreover, the point doubling, point tripling and computation of the isogenous curves can not be executed in parallel. However, in the execution of SIKE protocol, we need to compute map of multiple points on the isogenous curves and these can be executed in parallel. Therefore, in our architecture, we have simultaneously computed the map of two points on the isogenous curves. This allow us to use all the three multipliers, each executing four parallel modular multiplications.

Similar strategy has been applied in [8, 9, 10] where the authors have uses multiple multiplier cores to reduce the latency of the design. However, this comes at the cost of drastic increase in the DSP block requirement. In next section we will show that we can achieve similar timing performance of the existing architectures without paying the penalty of excessive DSP block usage.

Table 3: Scheduling of Field Operations For Point Doubling

Step No:	Addition in F_{p^2}	Subtraction in F_{p^2}	Modmul-1	Modmul-2
1	$t_1 = X_P + Z_P$	$t_0 = X_P \quad Z_P$	-	-
2	-	-	$t_2 = t_0^2$	$t_3 = t_1^2$
3	-	$t_6 = t_3 \quad t_2$	-	-
4	-	-	$t_4 = C_{24} \quad t_2$	$t_7 = A_{24}^{+} \quad t_6$
5	$t_8 = t_4 + t_7$	-	-	-
6	-	-	$X_{2P} = t_4 \quad t_3$	$Z_{2P} = t_8 \quad t_6$

Table 4: Scheduling of Field Operations For Point Tripling

Step No:	Addition in F_{p^2}	Subtraction in F_{p^2}	Modmul-1	Modmul-2
1	$t_1 = X_P + Z_P$	$t_0 = X_P \quad Z_P$	-	-
2	$t_4 = t_1 + t_0$	$t_5 = t_1 \quad t_0$	$t_2 = t_0^2$	$t_3 = t_1^2$
3	$t_7 = t_2 + t_3$	-	$t_6 = t_4^2$	$t_9 = t_3 \quad A_{24}^{+}$
3	$t_6 = t_2 + t_3$	$t_7 = t_2 \quad t_3$	-	-
4	-	$t_8 = t_6 \quad t_7$	$t_{10} = t_3 \quad t_9$	$t_{11} = t_2 \quad A_{24}$
5	-	$t_{14} = t_9 \quad t_{11}$	-	-
6	-	-	$t_{12} = t_2 \quad t_{11}$	$t_{15} = t_{14} \quad t_8$
7	-	$t_{13} = t_{12} \quad t_{10}$	-	-
8	$t_{16} = t_{13} + t_{15}$	$t_{18} = t_{13} \quad t_{15}$	-	-
9			$t_{17} = t_{16}^2$	$t_{19} = t_{18}^2$
10			$X_{3P} = t_{17} \quad t_4$	$Z_{3P} = t_{18} \quad t_5$

Table 5: Scheduling of Field Operations For Computing Degree-4 Isogeny

Step No:	Addition in F_{p^2}	Subtraction in F_{p^2}	Modmul-1	Modmul-2
1	$K_3 = X_{P_4} + Z_{P_4}$	$K_2 = X_{P_4} \quad Z_{P_4}$	$t_1 = Z_{P_4}^2$	$t_3 = X_{P_4}^2$
2	$t_2 = t_1 + t_1$	-	-	-
3	$t_4 = t_3 + t_3$	-	-	-
4	$K_1 = t_2 + t_2$	-	$C_{24} = t_2^2$	$A_{24}^+ = t_4^2$

Table 6: Scheduling of Field Operations For Evaluating Degree-4 Isogeny

Step No:	Addition in F_{p^2}	Subtraction in F_{p^2}	Modmul-1	Modmul-2
1	$t_0 = X_Q + Z_Q$	$t_1 = X_Q \quad Z_Q$	-	-
2	-	-	$t_2 = t_0 \quad K_2$	$t_4 = t_0 \quad t_1$
3	-	-	$t_3 = t_1 \quad K_3$	$t_5 = t_4 \quad K_1$
3	$t_6 = t_2 + t_3$	$t_7 = t_2 \quad t_3$	-	-
4	-	-	$t_8 = t_6^2$	$t_9 = t_7^2$
5	$t_{10} = t_5 + t_8$	$t_{11} = t_9 \quad t_5$	-	-
6	-	-	$X_Q = t_10 \quad t_8$	$Z_Q = t_9 \quad t_11$

Table 7: Scheduling of Field Operations For Computing Degree-3 Isogeny

Step No:	Addition in F_{p^2}	Subtraction in F_{p^2}	Modmul-1	Modmul-2
1	$K_2 = X_{P_3} + Z_{P_3}$	$K_1 = X_{P_3} \quad Z_{P_3}$	-	-
2	$t_3 = K_1 + K_2$	-	$t_0 = K_1^2$	$t_1 = K_2^2$
3	$t_2 = t_0 + t_1$	-	$t_3 = t_3^2$	-
4	-	$t_3 \quad t_2$	-	-
5	$t_2 = t_1 + t_3$	-	-	-
6	$t_3 = t_3 + t_0$	-	-	-
7	$t_4 = t_3 + t_0$	-	-	-
8	$t_4 = t_4 + t_4$	-	-	-
9	$t_5 = t_1 + t_4$	-	-	-
10	$t_4 = t_1 + t_2$	-	-	-
11	$t_4 = t_4 + t_4$	-	-	-
12	$t_4 = t_0 + t_4$	-	-	-
13	-	-	$A_{24} = t_2 \quad t_5$	$t_4 = t_3 \quad t_4$
14	-	$t_0 = t_4 \quad A_{24}$	-	-
15	$A_{24}^+ = A_{24} + t_0$	-	-	-

Table 8: Scheduling of Field Operations For Evaluating Degree-3 Isogeny

Step No:	Addition in F_{p^2}	Subtraction in F_{p^2}	Modmul-1	Modmul-2
1	$t_0 = X_Q + Z_Q$	$t_1 = X_Q \quad Z_Q$	-	-
2	-	-	$t_0 = K_1 \quad t_0$	$t_1 = K_2 \quad t_1$
3	$t_2 = t_0 + t_1$	$t_0 = t_1 \quad t_0$	-	-
4	-	-	$t_2 = t_2^2$	$t_0 = t_0^2$
5	-	-	$X_Q = X_Q \quad t_2$	$Z_Q = Z_Q \quad t_0$

7 Implementation Result

Table 9: Clock Cycle Requirement of Di erent Stages of SIKE Protocol

Work	Alice Keygen	Bob Keygen	Alice Key Agreement	Bob Key Agreement	Total
Present Work	1757597	2007206	1545357	1812507	7122667

We have implemented the downloaded the bit le of the proposed SIKE architecture on Xilinx VC 707 evaluation board and veri ed the correctness of the implementation by comparing the hardware result with the reference software code obtained from NIST portal. Such on chip validation ensures that our proposed architecture is working correctly on the actual hardware (refer appendix D). For comparing the performance of our design with the exiting implementations, we have also implemented our proposed architecture on Xilinx Virtex-7 xc7vx690t g1157-3 FPGA. Table 10 shows the area and the latency requirement of the proposed architecture. The area and the timing report has been obtained after executing place and route implementation

Table 10: Comparison between the Proposed SIKE Design and Existing SIKE Designs

Work	#Mult. Cores	#FFs	#LUTs	#Slices	#DSPs	#BRAMs	Freq. (MHz)	Latency (cc 10^6)	Total time (ms)
[10]	3	38489	27713	11277	288	60:5	204:9	7:46	36:4
	4	48688	34742	14447	384	58:5	203:7	6:86	33:7
	5	58846	42390	16983	480	56	197:7	6:56	33:2
	6	69054	50084	19892	576	54:5	201:5	6:37	31:6
[14]	4	51914	44822	16756	376	56:5	198	6:603	33:35
This Work	3	62124	49099	18711	294	22 :5	225 :7	7:123	31 :55

on Xilinx Vivado 2017.4 tool. The synthesis was done using Vivado default option, whereas the implementation has been optimized by the Performance Explore option. The clock cycle requirements of di erent phases of SIKE protocol are shown in table 9. Among the existing implementations of SIKE, the work in [10] shows the best timing performance till now. The authors in [10] achieves this timing performance by computing multiple isogeny map in parallel, for which they needed to replicate their proposed dual systolic Montgomery modular multiplier multiple times. Their rst implementation uses three multiplier cores and requires around 36:4 ms for one complete execution of SIKE (involving key generation and key agreement of both Alice and Bob). They improve their timing performance by around 20% when they use six multiplier cores with penalty of almost twice resource usage compared to the design with three multiplier cores.

In our proposed architecture, we compute two isogeny map in parallel and uses only three multiplier cores. However, when we compare our design with the design of [10], we observe that the timing performance of the design is nearly same with that of [10]. More speci cally, latency of our proposed architecture is less than the latency of the architectures of [10] using 3, 4 and 5 multiplier cores. Only the design with six multiplier cores exhibit the latency similar to our proposed architecture. However, we can achieve this timing performance with much less area overhead compared to the design of [10] using six multiplier cores. More speci cally, our design consumes 10% less ip- ops (FFs), around 2% less LUTs, 6% less slices, 49% less DSPs and 58:7% less BRAMs than the design of [10] while exhibiting same latency. This shows that our architecture is more e cient that the design of [10] as we can achieve the low latency requirement of SIKE protocol by paying signi cantly less resources. It must be noted that the clock cycle requirement of the proposed design is more compare to the six multiplier core design of [10]. However, as we employ e cient compressor circuits for multi-operand addition during Montgomery multiplication along with fast adders with e cient usage of carry chains for ripple carry adders, the critical path of the proposed architecture is low. This o sets the disadvantage of increased clock cycle requirement of the proposed architecture.

8 Conclusion

The existing implementations of SIKE architectures o er low latency in exchange of drastic increase in DSP and BRAM blocks requirement. The proposed architecture of SIKE addresses this issue as it provides low latency (same as that of the existing best implementation of SIKE from latency point of view) while consuming around 49% less DSP blocks and 58% less BRAM modules. The modular multiplier, which is the most critical component of the proposed architecture, is built on redundant number system. Additionally, the application of compressor circuits to support computation in carry save form increase the e ciency of the design signi cantly. Moreover, the architecture provides user the choice between performing either three or four parallel modular multiplications in F_p which is one of the unique feature of the design. The proposed architecture is more e cient than the existing implementation of SIKE as it can achieve the low latency requirement without requiring excessive number of DSP blocks and BRAMs.

References

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

[2] Debdeep Mukhopadhyay and Debapriya Basu Roy. Revisiting fpga implementation of montgomery multiplier in redundant number system for efficient ecc application in gf (p). In *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*, pages 323–3233. IEEE, 2018.

[3] Petter Källström and Oscar Gustafsson. Fast and area efficient adder for wide data in recent xilinx fpgas. In *26th International Conference on Field-Programmable Logic and Applications, Lausanne, Switzerland August 29-September 2, 2016*, pages 338–341, 2016.

[4] Anton Stolbunov. Constructing public-key cryptographic schemes based on class group action on a set of isogenous elliptic curves. *Advances in Mathematics of Communications*, 4(2):215–235, 2010.

[5] Andrew Childs, David Jao, and Vladimir Soukharev. Constructing elliptic curve isogenies in quantum subexponential time. *Journal of Mathematical Cryptology*, 8(1):1–29, 2014.

[6] David Jao and Luca De Feo. Towards quantum-resistant cryptosystems from supersingular elliptic curve isogenies. In *International Workshop on Post-Quantum Cryptography*, pages 19–34. Springer, 2011.

[7] Craig Costello, Patrick Longa, and Michael Naehrig. Efficient algorithms for supersingular isogeny diffie-hellman. In *Annual Cryptology Conference*, pages 572–601. Springer, 2016.

[8] Brian Koziel, Reza Azarderakhsh, Mehran Mozaffari Kermani, and David Jao. Post-quantum cryptography on fpga based on isogenies on elliptic curves. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 64(1):86–99, 2017.

[9] Brian Koziel, Reza Azarderakhsh, and Mehran Mozaffari-Kermani. Fast hardware architectures for supersingular isogeny diffie-hellman key exchange on fpga. In *International Conference in Cryptology in India*, pages 191–206. Springer, 2016.

[10] Brian Koziel, Reza Azarderakhsh, and Mehran Mozaffari-Kermani. A high-performance and scalable hardware architecture for isogeny-based cryptography. *IEEE Transactions on Computers*, 2018.

[11] Craig Costello and Benjamin Smith. Montgomery curves and their arithmetic. *Journal of Cryptographic Engineering*, 8(3):227–240, 2018.

[12] Alfred J Menezes, Tatsuaki Okamoto, and Scott A Vanstone. Reducing elliptic curve logarithms to logarithms in a finite field. *iEEE Transactions on information Theory*, 39(5):1639–1646, 1993.

[13] Josep Maria Miret, Ramiro Moreno Chiral, and Anna Rio. Generalization of vélu's formulae for isogenies between elliptic curves. *Publicacions matemàtiques, 2007, vol. Extra, p. 147–163*, 2007.

[14] David Jao et al. Sidh-spec.pdf. https://sike.org/files/SIDH-spec.pdf. (Accessed on 01/04/2019).

[15] Koji Shigemoto, Kensuke Kawakami, and Koji Nakano. Accelerating montgomery modulo multiplication for redundant radix-64k number system on the fpga using dual-port block rams. In *EUC'08. IEEE/IFIP International Conference on*, volume 1, pages 44–51. IEEE, 2008.

[16] Holger Orup. Simplifying quotient determination in high-radix modular multiplication. In *Computer Arithmetic, 1995., Proceedings of the 12th Symposium on*, pages 193–199. IEEE, 1995.

[17] T Blum and C Paar. High radix montgomery modular exponentiation on reconfigurable hardware for public-key cryptography,". *IEEE Transactions on Computers. to appear.*

[18] Peter L Montgomery. Modular multiplication without trial division. *Mathematics of computation*, 44(170).

[19] Paolo Zicari and Stefania Perri. A fast carry chain adder for virtex-5 fpgas. In *MELECON 2010-2010 15th IEEE Mediterranean Electrotechnical Conference*, pages 304–308. IEEE, 2010.

Appendix A

Addition Operation in Redundant Number System

Addition of two d digit redundant number X and Y can be computed as follows

$$Z[0] = X_0[r-1:0] + Y_0[r-1:0]$$
$$Z[i] = X_i[r-1:0] + Y_i[r-1:0] + X_{i-1}[r+1:r] + Y_{i-1}[r+1:r] \qquad (1 \leq i < d)$$

In [15], the authors have shown how to perform multiplication operation using 18×18 multipliers in redundant number system. This was later updated in [2] where the authors modified it. We discuss this below.

Redundant Base Multiplication using Asymmetric Multipliers

Let $X = (X_{d-1}, \ldots X_1, X_0)$ be a d digit redundant number where the length of each X_i is $(r_2 + 2)$ bits. Similarly Y is a single digit redundant number having length (r_1+2), where $2r_2 < r_1+r_2+4 < 3r_2$. For DSP blocks containing 24×17 asymmetric multipliers, the maximum value of r_1 is 22 and r_2 is 15 with two bits as redundant bits. Such value of r_1 and r_2 will satisfy the relation $2r_2 < r_1 + r_2 + 4 < 3r_2$. We can compute the partial products $P_i = X_i \cdot Y$ using the asymmetric multiplier of dimension $(r_1 + 2) \times (r_2 + 2)$. The length of each P_i would be $(r_1 + r_2 + 4)$. We can compute the multiplication result $X \cdot Y = K = \{K_{d+1}, K_d, \ldots K_0\}$ as follows:

$$K_0 = P_0[r_2 - 1 : 0]$$
$$K_1 = P_0[2r_2 - 1 : r_2] + P_1[r_2 - 1 : 0]$$
$$K_i = P_{i-2}[r_1 + r_2 + 3 : 2r_2] + P_{i-1}[2r_2 - 1 : r_2] + P_i[r_2 - 1 : 0] \qquad (2 \leq i \leq d-1)$$
$$K_d = P_{d-2}[r_1 + r_2 + 3 : 2r_2] + P_{d-1}[2r_2 - 1 : r_2]$$
$$K_{d+1} = P_{d-1}[r_1 + r_2 + 3 : 2r_2]$$

The value of any $K_i = P_{i-2}[r_1 + r_2 + 3 : 2r_2] + P_{i-1}[2r_2 - 1 : r_2] + P_i[r_2 - 1 : 0] < 3 \cdot 2^{r_2} < 2^{r_2+2}$. This ensures that K is a valid redundant number with radix 2^{r_2} and there is no overflow. For computation of each $X_i \cdot Y$, we will require one DSP block. Therefore, for the entire computation we will require d number of DSP blocks. It must be noted that if we would have considered only one bit as a redundant bit, overflow would have occurred if we implement the multi-operand additions using ripple carry adders. This is the reason behind using carry save adders while handling redundant number with only one bit as a redundant bit.

Multiplication and Accumulation

Let $X = (X_{d-1}, \ldots X_1, X_0)$ be a d digit and $C = (C_d, \ldots C_1, C_0)$ be a $d + 1$ digit redundant number where the length of each X_i and C_i is (r_2+2) bits. Additionally, Y

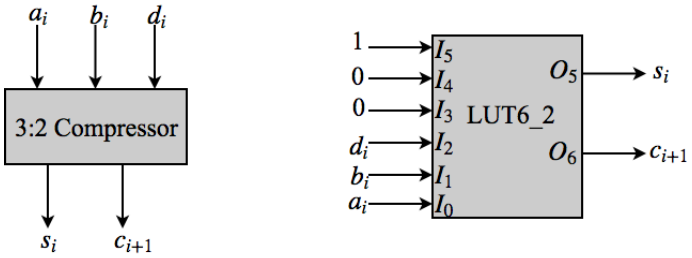


Figure 7: 3:2 Compressor Circuit

is a single digit redundant number having length (r_1+2). The result of $X \cdot Y + C = T = \{T_{d+1}, T_d, \ldots T_0\}$ can be computed as follow:

$$
\begin{aligned}
T_0 &= P_0[r_2-1:0] + C_0[r_2-1:0] \\
T_1 &= P_0[2r_2-1:r] + P_1[r_2-1:0] + C_1[r_2-1:0] + C_0[r_2+1:r_2] \qquad (2 \le i \le d-1) \\
T_i &= P_{i-2}[r_1+r_2+3:2r_2] + P_{i-1}[2r_2-1:r_2] + P_i[r_2-1:0] + C_i[r_2-1:0] + C_{i-1}[r_2+1:r_2] \\
T_d &= P_{d-2}[r_1+r_2+3:2r_2] + P_{d-1}[2r_2-1:r_2] + C_d[r_2-1:0] + C_{d-1}[r_2+1:r_2] \\
T_{d+1} &= P_{d-1}[r_1+r_2+3:2r_2] + C_d[r_2+1:r_2]
\end{aligned}
$$

The value of $T_i = P_{i-2}[r_1+r_2+3:2r_2]+P_{i-1}[2r_2-1:r_2]+P_i[r_2-1:0]+C_i[r_2-1:0]+C_{i-1}[r_2+1:r_2] < 3\cdot 2^{r_2}+2^{r_1+r_2+4-2r_2}+2^2 < 2^{r_2+2}$. Thus T is a valid redundant number with radix 2^{r_2} and there is no overflow.

Appendix B

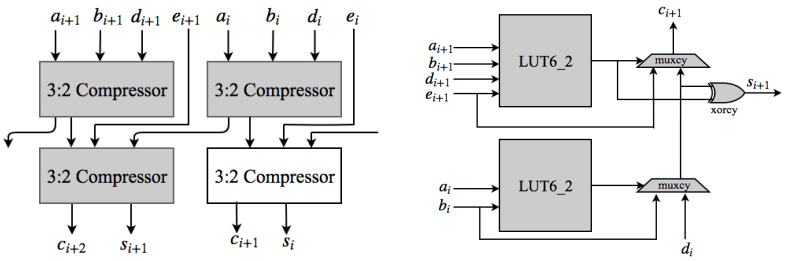
3:2 Compressor

3:2 compressor takes three single bit input a_i, b_i and d_i and produces the corresponding sum and carry output (s_i and c_{i+1}). Implementation of 3:2 compressor is shown in Figure 7. The modern FPGAs contain 6 input look up tables which can be also configured to implement a dual function of five inputs. Therefore a single LUT is enough to implement a 3 : 2 compressor.

4:2 Compressor and 5:2 compressor

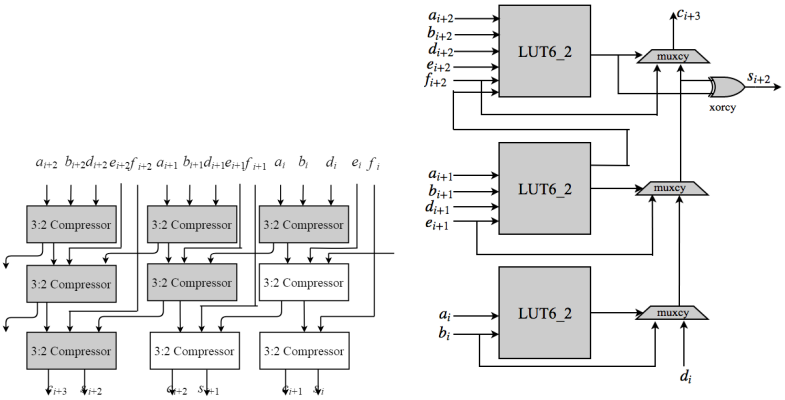
A 4 : 2 compressor takes four single bit input $a_{i+1}, b_{i+1}, d_{i+1}, e_{i+1}$ and produces the corresponding sum and carry output (s_{i+1} and c_{i+2} respectively). The basic building block of a 4 : 2 compressor circuit is 3 : 2 compressor module as shown in Figure 8(a). The LUT implementation of the shaded region is shown in Figure 8(b).
We have extended the design of 4 : 2 compressor to implement the 5 : 2 compressor as shown in Figure 9(a). The LUT implementation of the shaded region is shown in Figure 9(b).
We will now focus on the application of the aforementioned compressor circuits on the architecture of the proposed Montgomery multiplier. As we have already mentioned, multiple multi-operand additions are involved in the computation of the Montgomery product value. These multi-operand additions can be classified as follows:

- Three input addition: This is used during the computation of $a_i.B$. Partial products of $a_i.B$ can be added using 3 : 2 compressor modules.
- Five input addition:. This is used during the computation of $S_i + q_i.\overline{M}$. It can be implemented using 5 : 2 compressor.



(a) 4:2 Compressor (b) LUT implementation

Figure 8: 4:2 Compressor Module and LUT Implementation



(a) 5:2 Compressor (b) LUT implementation

Figure 9: 5:2 Compressor Module and LUT Implementation

– **Four Input addition:** This is used to add the result of the three input and five input addition blocks $((S_i + q_i.\overline{M})/2^r + a_i.B)$. As the addition results are in redundant representation, this can be implemented using $4:2$ compressor.

Appendix C

Algorithm 6: Computing l^e Isogeny Using Optimum Computational Strategy

Input:Starting curve E, kernel generator point R corresponding to l^e isogeny map, computational strategy $\{s_1, s_2, \ldots, s_{e-1}\}$ and a list of points $(P_1, P_2, \ldots)$ [14] **Result:** Image curve $E = E/ < R >$ corresponding to isogeny map (φ) of degree l^e, the ;ist of image points $(\varphi(P_1), \varphi(P_2), \ldots)$ on E
if $e == 1$ *(empty strategy)* **then**
  $\varphi_l : E = E/ < R >$ (Compute degree l isogeny)
  Return E , $(\varphi_l(P_1), \varphi_l(P_2), \ldots)$
end
$n = s_1$
$Left = s_2, \ldots, s_{e-n}$ and $Right = s_{e-n+1}, \ldots, s_{e-1}$
$T = [l^n]R$
Compute $E, (U, P_1, P_2, \ldots) =$ Recurse on $E, T, (R, P_1, P_2, \ldots)$ with strategy $Left$
Compute $E, (P_1, P_2, \ldots) =$ Recurse on $E, U, (P_1, P_2, \ldots)$ with strategy $Right$